Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática
Maestría en Diseño Electrónico



Implementation of Vector Engine for RISCV Architecture

TESIS que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **MIGUEL ÁNGEL BUCIO MACÍAS**

Director **DR. JOSÉ LUIS PIZANO ESCALANTE**
Codirector **DR. OMAR HUMBERTO LONGORIA GÁNDARA**

Instituto Tecnológico y de Estudios Superiores de Occidente

Campus Guadalajara

School of Engineering and Sciences

The committee members, hereby, certify that have read the thesis presented by Miguel Ángel Bucio Macías (Student) and that it is fully adequate in scope and quality as a partial requirement for the degree of Master of Science in Electronics Design.

José Luis Pizano Escalante
Instituto Tecnológico y de Estudios Superiores de Occidente
Principal Advisor

Omar Humberto Longoria Gándara
Instituto Tecnológico y de Estudios Superiores de Occidente
Committee Member

Leonel Alejandro Haro Ruiz
Instituto Tecnológico y de Estudios Superiores de Occidente
Committee Member

Dr. Omar Humberto Longoria Gandara
Associate Dean of Graduate Studies
School of Engineering and Sciences

Guadalajara, Jalisco, November, 2025

Declaration of Authorship

I, Miguel Ángel Bucio Macías (Student), declare that this thesis titled, ”Implementation of Vector Enginge for RISCV Architecture” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Miguel Ángel Bucio Macías (Student)
Guadalajara, Jalisco, November, 2025

©2025 by Miguel Ángel Bucio Macías (Student)
All Rights Reserved

Acknowledgements

I would like to express my gratitude to all the people who have supported me throughout this process.

Starting with my parents Flora Beatriz and Miguel Angel that gave me the blessing of education, allowing me to study one of the best careers, Electronics Engineering, and supporting me through the career and always being by my side.

To my brothers Luis David and Pablo Raul for helping me to destress during the long nights and helping me to use software that I didn't know.

To my girlfriend Mariana for always hear me talking about my project for hours and hours, trying to solve a problem, even if she didn't fully understand what I was saying always heard me and supported me unconditionally to complete the project.

To my friends and colleagues Miguel Mendoza, Miguel Triana and Braulio Martínez for reviewing this document and giving me feedback about what could be improved.

To my manager and friend Leonel Haro for trusting in me, that I would complete my master's degree and helped me to get Intel's tuition support.

To my advisors Cuauhtémoc Aguilera, José Luis Pizano and Omar Longoria for being with me throughout the project, giving me advice and directions about how to proceed with the project and helping me to get all the necessary tools to complete it.

And to Intel, AheadComputing and ITESO for giving me the support, tools and knowledge that I used to complete the project.

Summary

In this project a *Vector Engine* is implemented using the *RISC-V Vector Extension* specification [1] as a base. The *Vector Engine* adheres to the general guidelines and architectural specifications defined in the *RISC-V specification* but does not implement the full set of operations and features described in the specification. Instead, a carefully selected subset of functionalities has been developed and integrated. The *Vector Engine* implements a subset of the addition/subtraction, bitwise, logical, shifting, sign/zero extend, multiply, compare, masking and permutation operations. These operations were chosen to demonstrate the core capabilities of vector processing while maintaining manageable design complexity within the scope of this project.

The logic required to execute the selected vector operations is fully implemented in hardware. However, the logic responsible for instruction fetching has been intentionally excluded, as the focus of the project is on the execution and processing units rather than the instruction handling pipeline. Furthermore, the set of control and status registers (CSRs) necessary for configuring and managing the *Vector Engine* is also implemented. While the full range of features associated with these registers is not utilized in this implementation, their inclusion allows for potential future extensions and serves to maintain alignment with the RISC-V architectural model.

The development of the *Vector Engine* encompasses multiple key phases. First, the microarchitecture of the unit is defined, specifying the internal structure and data paths required for operation. Following this, a Register-Transfer Level (RTL) description is created to model the behavior and structure of the engine in a synthesizable format. To ensure the correctness of the implementation, functional validation is carried out using the Universal Verification Methodology (*UVM*), a standardized approach for verifying complex digital designs. Finally, the design is subjected to physical synthesis targeting five different semiconductor technologies, enabling the evaluation of area, performance, and power characteristics across a range of implementation scenarios.

List of Figures

2.1 Two's complement . . . 5
2.2 Half Adder . . . 5
2.3 Full Adder . . . 6
2.4 Ripple Carry Adder . . . 7
2.5 Carry Save Adder . . . 8
2.6 Carry Look-Ahead Adder . . . 9
2.7 Carry Increment Adder . . . 10
2.8 Carry Skip Adder . . . 11
2.9 Carry Select Adder . . . 12
2.10 Carry Select Adder Big . . . 12
2.11 Kogge-Stone Adder . . . 14
2.12 Kogge-Stone Adder Carry Generation . . . 15
2.13 Right Barrel Shifter . . . 17
2.14 Array Multiplier . . . 19
2.15 Critical path array multiplier . . . 20
2.16 Urdhava Tiryakbhyam Sutra . . . 21
2.17 Vedic multiplication . . . 24
2.18 Vedic Multiplier . . . 25
2.19 Vedic 2-Bits . . . 26
2.20 Critical path Vedic Multiplier . . . 27
2.21 UVM Phases . . . 31
2.22 Vector Normal Operation . . . 36
2.23 Vector Reduction Operation . . . 37
2.24 Output multiplexing with normal multiplexer . . . 38
2.25 Output multiplexing with data gating . . . 39
2.26 Bit Swizzle . . . 41
2.27 Byte Swizzle . . . 42
2.28 Vector Instruction Format . . . 44
2.29 Pipeline stages flow diagram . . . 45
2.30 Reduction Source Multiplexor . . . 48
2.31 Reduction Possible sources . . . 49
2.32 Reduction Source . . . 51
2.33 Reduction Source Mux selection . . . 52
2.34 Prestart, Active, Inactive and Tail Elements . . . 58
2.35 Vector Engine . . . 60
2.36 Register Vector Register File . . . 61

2.37 Vector Register File . . . 62
2.38 Read Port Vector Register File . . . 63
2.39 Decode unit with reuse . . . 64
2.40 Decode unit with comparators . . . 64
2.41 CSR Control Unit . . . 68
2.42 Bypass unit . . . 70
2.43 Bypass module . . . 71
2.44 Replicator module . . . 71
2.45 Decode Element . . . 72
2.46 Greater/Less than OSIZE . . . 73
2.47 Decode element start/len . . . 74
2.48 Decode element operation mask . . . 75
2.49 Decode element Masking . . . 76
2.50 Decode element final mask . . . 76
2.51 Decode element merge . . . 77
2.52 Execution Unit . . . 78
2.53 Bitwise Normal Logical . . . 79
2.54 Bitwise Reduction Logical . . . 80
2.55 Logic ALU . . . 81
2.56 Bitwise AND . . . 82
2.57 Bitwise OR . . . 83
2.58 Bitwise XOR . . . 84
2.59 Shifter . . . 85
2.60 Shift count mask . . . 86
2.61 Shift count . . . 86
2.62 Shifter Layer . . . 88
2.63 Byte Shifter . . . 89
2.64 Arithmetic ALU . . . 90
2.65 Vector adder . . . 91
2.66 Adder carry_in vector . . . 92
2.67 Simple adder . . . 93
2.68 Adder with subtract . . . 94
2.69 Adder with reduction . . . 95
2.70 Adder flags . . . 96
2.71 Adder compare . . . 97
2.72 Adder Max/Min . . . 97
2.73 Vedic multiplier internal results . . . 99
2.74 Vector multiplier(Array Multiplier) . . . 100
2.75 Vedic multiplier signed . . . 101
2.76 Vector Two's complement block . . . 102
2.77 Vector Multiplier . . . 103
2.78 sign/zero extend . . . 104
2.79 Permutation ALU . . . 105
2.80 Vector to Integer . . . 106
2.81 Integer to Vector . . . 106

2.82 Result swizzle (Block) . . . 107
2.83 Result swizzle (Byte) . . . 107

3.1 Base Components UML . . . 108
3.2 Base Configuration object UML . . . 109
3.3 Base BFM UML . . . 110
3.4 Base BFM sequence diagram . . . 110
3.5 Base Subscriber Two Ports UML . . . 113
3.6 Base Coverage UML . . . 114
3.7 Base Driver UML . . . 114
3.8 Driver Configurations . . . 115
3.9 Base Monitor UML . . . 116
3.10 Base Model UML . . . 116
3.11 Base Scoreboard UML . . . 117
3.12 Base Sequence Item-Sequence-Sequencer UML . . . 118
3.13 Base Tracker UML . . . 119
3.14 Base Agent UML . . . 121
3.15 Build Agent flow diagram . . . 122
3.16 Base Environment UML . . . 123
3.17 RF Model UML . . . 124
3.18 RF BFM UML . . . 125
3.19 Rotator UML . . . 126
3.20 RISCV RF components UML . . . 127
3.21 RISCV RF BFM UML . . . 128
3.22 RISCV RF Coverage UML . . . 129
3.23 RISCV RF Driver/Monitor UML . . . 131
3.24 RISCV RF sequence item-sequence UML . . . 132
3.25 RISCV RF Scoreboard UML . . . 133
3.26 RISCV RF Tracker UML . . . 134
3.27 RISCV RF Tracker Example . . . 135
3.28 RISCV-V Base components UML . . . 135
3.29 RISCV-V RF components UML . . . 135
3.30 RISCV-V Register UML . . . 136
3.31 RISCV-V RF BFM UML . . . 137
3.32 RISCV-V RF Coverage UML . . . 138
3.33 RISCV-V RF Driver/Monitor UML . . . 140
3.34 RISCV-V RF Model UML . . . 141
3.35 RISCV-V RF Sequence Item-Sequence UML . . . 142
3.36 RISCV-V RF Scoreboard UML . . . 143
3.37 RISCV-V RF Tracker UML . . . 144
3.38 RISCV-V RF Tracker Example . . . 145
3.39 RISCV-V CSR components UML . . . 145
3.40 RISCV-V CSR Model UML . . . 146
3.41 RISCV-V CSR Sequence item UML . . . 147
3.42 RISCV-V CSR Coverage UML . . . 148

3.43 RISCV-V CSR Tracker Example . . . 151
3.44 RISCV-V External CSR components UML . . . 151
3.45 RISCV-V External CSR BFM UML . . . 152
3.46 RISCV-V External CSR Driver/Monitor UML . . . 153
3.47 RISCV-V External CSR Sequence item-Sequence UML . . . 154
3.48 RISCV-V IF components UML . . . 155
3.49 RISCV-V Instruction UML . . . 156
3.50 RISCV-V IF BFM UML . . . 157
3.51 RISCV-V IF Coverage UML . . . 159
3.52 RISCV-V IF Driver/Monitor UML . . . 160
3.53 RISCV-V IF Sequence item-Sequence UML . . . 161
3.54 RISCV-V IF Model UML . . . 162
3.55 RISCV-V IF Tracker UML . . . 163
3.56 RISCV-V Instruction tracker example . . . 164
3.57 RISCV-V ALU components UML . . . 164
3.58 RISCV-V ALU Coverage UML . . . 167
3.59 RISCV-V ALU Driver/Monitor UML . . . 169
3.60 RISCV-V ALU Sequence item-Sequence UML . . . 172
3.61 RISCV-V ALU Model UML . . . 172
3.62 RISCV-V ALU Scoreboard UML . . . 174
3.63 RISCV-V ALU Tracker UML . . . 176
3.64 RISCV-V ALU Tracker Example . . . 176
3.65 RISCV-V components UML . . . 177
3.66 RISCV-V Coverage UML . . . 177
3.67 RISCV-V Driver/Monitor UML . . . 179
3.68 RISCV-V Model UML . . . 181
3.69 RISCV-V Model Flow diagram . . . 182
3.70 RISCV-V Scoreboard UML . . . 184
3.71 RISCV-V TB . . . 186
3.72 RISCV-V VADD Test . . . 188
3.73 Test Status . . . 191
3.74 Regression Status . . . 191
3.75 Final Coverage . . . 192
3.76 RISCV RF write coverage . . . 192
3.77 RISCV RF read port A coverage . . . 193
3.78 RISCV RF read port B coverage . . . 194
3.79 RISCV-V RF write coverage . . . 195
3.80 RISCV-V RF read port A coverage . . . 195
3.81 RISCV-V RF read port B coverage . . . 196
3.82 RISCV-V RF Mask coverage . . . 196
3.83 RISCV-V CSR write coverage . . . 197
3.84 RISCV-V CSR read coverage . . . 197
3.85 RISCV-V IF coverage . . . 198
3.86 RISCV-V coverage . . . 199
3.87 RISCV-V Logic ALU coverage . . . 200

3.88 RISCV-V Arithmetic ALU coverage . . . 201
3.89 RISCV-V Permutation ALU coverage . . . 202
3.90 RISCV-V Coverage Complete . . . 204

4.1 OpenROAD flow [15]. . . . 206
4.2 Mul array 8-Bits RTL Schematic . . . 207
4.3 Mul Vedic 8-Bits RTL Schematic . . . 208
4.4 Vector array multiplier RTL Schematic . . . 211
4.5 Vector Vedic multiplier RTL Schematic . . . 212
4.6 FPGA vector multiplier DSP use . . . 212
4.7 RISCV-V Decode RTL Schematic . . . 214
4.8 RISCV-V Control Unit RTL Schematic . . . 215
4.9 RISCV-V CSR Control RTL Schematic . . . 217
4.10 RISCV-V CSR RTL Schematic . . . 219
4.11 RISCV-V Vector Register File Control RTL Schematic . . . 221
4.12 RISCV-V Vector Register File RTL Schematic . . . 223
4.13 RISCV-V Execute Unit RTL Schematic . . . 225
4.14 RISCV-V Decode Element RTL Schematic . . . 227
4.15 RISCV-V Bypass Unit RTL Schematic . . . 229
4.16 RISCV-V Reduction Source RTL Schematic . . . 230
4.17 RISCV-V Swizzle result RTL Schematic . . . 231
4.18 RISCV-V Logic ALU RTL Schematic . . . 232
4.19 RISCV-V BW AND RTL Schematic . . . 234
4.20 RISCV-V BW OR RTL Schematic . . . 235
4.21 RISCV-V BW XOR RTL Schematic . . . 236
4.22 RISCV-V Shifter RTL Schematic . . . 238
4.23 RISCV-V Arithmetic ALU RTL Schematic . . . 239
4.24 RISCV-V Adder RTL Schematic . . . 240
4.25 RISCV-V Multiplier RTL Schematic . . . 242
4.26 RISCV-V Extender RTL Schematic . . . 243
4.27 RISCV-V Permutation ALU RTL Schematic . . . 244
4.28 RISCV Vector Engine RTL Schematic . . . 246
4.29 RISCV Vector Engine All Units . . . 249
4.30 RISCV Vector Engine Decode/Execute Units . . . 250
4.31 RISCV Vector Engine VRF . . . 251
4.32 RISCV Vector Engine VRF Ctrl . . . 252
4.33 RISCV Vector Engine CSR . . . 253
4.34 RISCV Vector Engine Decoder . . . 254
4.35 RISCV Vector Engine Bypass . . . 255
4.36 RISCV Vector Engine Shifter . . . 256
4.37 RISCV Vector Engine Bitwise blocks . . . 257
4.38 RISCV Vector Engine Adder . . . 258
4.39 RISCV Vector Engine Extender . . . 259
4.40 RISCV Vector Engine Permutation ALU . . . 260
4.41 RISCV Vector Engine Multiplier . . . 261

4.42 RISCV Vector Engine Routing with power 262
4.43 RISCV Vector Engine Routing 263
4.44 RISCV Vector Engine Routing density heat map 264
4.45 RISCV Vector Engine Placement density heat map 265
4.46 RISCV Vector Engine Power heat map 266
4.47 RISCV Vector Engine All Instances 267
4.48 RISCV Vector Engine Combinational Instances 268
4.49 RISCV Vector Engine Sequential Instances 269
4.50 RISCV Vector Engine Clock Tree 270
4.51 RISCV Vector Engine Clock Tree 1 Level 271
4.52 RISCV Vector Engine Clock Tree 2 Level 271
4.53 RISCV Vector Engine Clock Tree 3 Level 272

List of Tables

2.1 Array VS Vedic Multiplier comparison . . . 28
2.2 UVM Phases . . . 30
2.3 UVM Components . . . 33
2.4 UVM Verbosity . . . 35
2.5 Funct3 Encodings . . . 43
2.6 Pipeline Actions . . . 46
2.7 Reduction sources . . . 50
2.8 Implemented CSR . . . 53
2.9 Vsew supported Values . . . 55
2.10 RISCV-V Control signals . . . 65
2.11 RISCV-V Stage Control signals . . . 66
2.12 VRF Stage signals . . . 67
2.13 CSR Stage signals . . . 69
2.14 Vector multiplier Array vs Vedic . . . 100

3.1 Base BFM configuration Object methods and attributes . . . 111
3.2 Base BFM methods and attributes . . . 112
3.3 RISCV RF Functional Coverage . . . 130
3.4 RISCV RF Driver/Monitor signals . . . 130
3.5 RISCV RF Scoreboard checks . . . 132
3.6 RISCV-V RF Functional Coverage . . . 139
3.7 RISCV-V-RF Driver/Monitor signals . . . 139
3.8 RISCV-V RF Scoreboard checks . . . 142
3.9 RISCV-V write CSR Functional Coverage . . . 149
3.10 RISCV-V read CSR Functional Coverage . . . 150
3.11 RISCV-V External CSR Driver/Monitor signals . . . 154
3.12 RISCV-V IF BFM configuration knobs . . . 158
3.13 RISCV-V Instruction Functional Coverage . . . 159
3.14 RISCV-V IF Driver/Monitor signals . . . 161
3.16 Arithmetic ALU Functional Coverage . . . 166
3.15 Logic ALU Functional Coverage . . . 168
3.17 Permutation ALU Functional Coverage . . . 169
3.18 RISCV-V Logic ALU Driver/Monitor signals . . . 170
3.19 RISCV-V Arithmetic ALU Driver/Monitor signals . . . 171
3.20 RISCV-V Permutation ALU Driver/Monitor signals . . . 171
3.21 RISCV-V Logic ALU Scoreboard checks . . . 173

3.22 RISCV-V Arithmetic ALU Scoreboard checks . . . 175
3.23 RISCV-V Permutation ALU Scoreboard checks . . . 175
3.24 RISCV-V Functional Coverage . . . 178
3.25 RISCV-V Monitor signals . . . 180
3.26 RISCV-V Scoreboard checks . . . 185
3.27 RISCV-V Test configuration knobs . . . 187
3.28 Logic Tests . . . 189
3.29 Arithmetic Tests . . . 190
3.30 Permutation Tests . . . 191

4.1 Unsigned Multiplier synthesis statistics . . . 209
4.2 Vector Multiplier synthesis statistics . . . 210
4.3 RISCV-V Decode synthesis statistics . . . 215
4.4 RISCV-V Control unit synthesis statistics . . . 216
4.5 RISCV-V CSR control synthesis statistics . . . 218
4.6 RISCV-V CSR synthesis statistics . . . 220
4.7 RISCV-V Vector Register File Control synthesis statistics . . . 222
4.8 RISCV-V Vector Register File synthesis statistics . . . 224
4.9 RISCV-V Execute Unit synthesis statistics . . . 225
4.10 RISCV-V Decode Element synthesis statistics . . . 228
4.11 RISCV-V Bypass Unit synthesis statistics . . . 230
4.12 RISCV-V Reduction Source synthesis statistics . . . 231
4.13 RISCV-V Swizzle Result synthesis statistics . . . 232
4.14 RISCV-V Logic ALU synthesis statistics . . . 233
4.15 RISCV-V BW AND synthesis statistics . . . 237
4.16 RISCV-V BW OR synthesis statistics . . . 237
4.17 RISCV-V BW XOR synthesis statistics . . . 238
4.18 RISCV-V Shifter synthesis statistics . . . 239
4.19 RISCV-V Arithmetic ALU synthesis statistics . . . 240
4.20 RISCV-V Adder synthesis statistics . . . 241
4.21 RISCV-V Multiplier synthesis statistics . . . 243
4.22 RISCV-V Extender synthesis statistics . . . 244
4.23 RISCV-V Permutation ALU synthesis statistics . . . 245
4.24 RISCV Vector Engine synthesis statistics . . . 247
4.25 RISCV Vector Engine synthesis statistics 2.4 GHZ . . . 247
4.26 RISCV Vector Area comparison . . . 248

Contents

Abstract iv

List of Figures x

List of Tables xii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement and Context . . . 2
- 1.3 Research question . . . 2
- 1.4 Contributions of the Thesis . . . 3

2 Design 4
- 2.1 Theoretical Framework . . . 4
  - 2.1.1 SIMD . . . 4
  - 2.1.2 Two's complement . . . 4
  - 2.1.3 Half Adder . . . 5
  - 2.1.4 Full Adder . . . 6
  - 2.1.5 Ripple Carry Adder . . . 6
  - 2.1.6 Carry Save Adder . . . 7
  - 2.1.7 Carry Look-Ahead Adder . . . 8
  - 2.1.8 Carry Increment Adder . . . 9
  - 2.1.9 Carry Skip Adder . . . 11
  - 2.1.10 Carry Select Adder . . . 11
  - 2.1.11 Kogge-Stone Adder . . . 13
  - 2.1.12 Barrel Shifter . . . 16
  - 2.1.13 Array Multiplier . . . 18
  - 2.1.14 Vedic Multiplier . . . 21
  - 2.1.15 BFM . . . 28
  - 2.1.16 Virtual Interface . . . 28
  - 2.1.17 UVM . . . 29
- 2.2 Reduction Operations . . . 35
- 2.3 Output Multiplexing . . . 38
- 2.4 Swizzle . . . 40
  - 2.4.1 Bit Swizzle . . . 40

2.4.2 Byte Swizzle . . . 41
2.5 Vector Instruction . . . 42
2.6 Pipeline Stages . . . 44
2.6.1 Instruction Fetch . . . 47
2.6.2 Instruction Decode . . . 47
2.6.3 Execute . . . 47
2.6.4 Memory Access . . . 47
2.6.5 Write Back . . . 48
2.7 Reduction Source . . . 48
2.8 Control Status Registers . . . 52
2.8.1 VSSTAUS . . . 54
2.8.2 VTYPE . . . 54
2.8.3 VL . . . 55
2.8.4 VLENB . . . 55
2.8.5 VSTART . . . 55
2.8.6 VXRM . . . 56
2.8.7 VXSAT . . . 56
2.8.8 VCSR . . . 56
2.9 Active and Inactive Elements . . . 56
2.10 Masking . . . 59
2.11 RISCV Vector Engine . . . 59
2.12 Vector Register File . . . 60
2.13 Control Units . . . 63
2.13.1 RISCV-V Control Unit . . . 65
2.13.2 VRF Control Unit . . . 67
2.13.3 CSR Control Unit . . . 67
2.14 Bypass Unit . . . 69
2.15 Decode Element . . . 72
2.16 Execution Unit . . . 77
2.17 Logic ALU . . . 78
2.17.1 Bitwise AND . . . 81
2.17.2 Bitwise OR . . . 82
2.17.3 Bitwise XOR . . . 83
2.17.4 Shifter . . . 84
2.18 Arithmetic ALU . . . 89
2.18.1 Vector Adder . . . 91
2.18.2 Vector Multiplier . . . 98
2.18.3 Extender . . . 104
2.19 Permutation ALU . . . 104
2.19.1 Vector to Integer . . . 105
2.19.2 Integer to Vector . . . 106
2.20 Result Swizzle . . . 106

3 Validation **108**
3.1 Base Components . . . 108
3.1.1 Base Configuration Object . . . 109
3.1.2 Base BFM . . . 109
3.1.3 Base Subscriber Two Ports . . . 113
3.1.4 Base Coverage . . . 113
3.1.5 Base Driver . . . 114
3.1.6 Base Monitor . . . 115
3.1.7 Base Model . . . 116
3.1.8 Base Scoreboard . . . 117
3.1.9 Base Sequence item-Sequence-Sequencer . . . 118
3.1.10 Base Tracker . . . 118
3.1.11 Base Agent . . . 121
3.1.12 Base Environment . . . 123
3.1.13 RF Model . . . 123
3.1.14 RF BFM . . . 124
3.1.15 Rotator . . . 126
3.2 RISCV-RF Agent . . . 127
3.2.1 RISCV RF BFM . . . 127
3.2.2 RISCV RF Coverage . . . 129
3.2.3 RISCV RF Driver/Monitor . . . 130
3.2.4 RISCV RF Sequence Item-Sequence . . . 132
3.2.5 RISCV RF Scoreboard . . . 132
3.2.6 RISCV RF Tracker . . . 134
3.3 RISCV-V Base . . . 135
3.4 RISCV-V-RF Agent . . . 135
3.4.1 RISCV-V Register . . . 136
3.4.2 RISCV-V-RF BFM . . . 136
3.4.3 RISCV-V-RF Coverage . . . 138
3.4.4 RISCV-V-RF Driver/Monitor . . . 139
3.4.5 RISCV-V RF Model . . . 141
3.4.6 RISCV-V RF Sequence Item-Sequence . . . 141
3.4.7 RISCV-V RF Scoreboard . . . 142
3.4.8 RISCV-V RF Tracker . . . 144
3.5 RISCV-V-CSR Agent . . . 145
3.5.1 RISCV-V CSR Model . . . 145
3.5.2 RISCV-V CSR Sequence Item . . . 147
3.5.3 RISCV-V CSR Coverage . . . 147
3.5.4 RISCV-V CSR Tracker . . . 151
3.6 RISCV-V- External CSR Agent . . . 151
3.6.1 RISCV-V External CSR BFM . . . 151
3.6.2 RISCV-V External CSR Driver/Monitor . . . 153
3.6.3 RISCV-V External CSR Sequence Item-Sequence . . . 154
3.7 RISCV-V-IF Agent . . . 155
3.7.1 RISCV-V Instruction . . . 155

3.7.2 RISCV-V IF BFM . . . 157
3.7.3 RISCV-V IF Coverage . . . 158
3.7.4 RISCV-V IF Driver/Monitor . . . 160
3.7.5 RISCV-V IF Sequence Item-Sequence . . . 161
3.7.6 RISCV-V IF Model . . . 161
3.7.7 RISCV-V IF Tracker . . . 163
3.8 RISCV-V-ALU Agent . . . 164
3.8.1 RISCV-V ALU Coverage . . . 164
3.8.2 RISCV-V ALU Driver/Monitor . . . 169
3.8.3 RISCV-V ALU Sequence Item-Sequence . . . 172
3.8.4 RISCV-V ALU Model . . . 172
3.8.5 RISCV-V ALU Scoreboard . . . 173
3.8.6 RISCV-V ALU Tracker . . . 176
3.9 RISCV-V Agent . . . 177
3.9.1 RISCV-V Coverage . . . 177
3.9.2 RISCV-V Monitor . . . 178
3.9.3 RISCV-V Model . . . 180
3.9.4 RISCV-V Scoreboard . . . 183
3.10 RISCV-V TB . . . 186
3.11 RISCV-V Tests . . . 186
3.12 Validation Results . . . 191
3.12.1 Coverage Results . . . 192

4 Synthesis **205**
4.1 Multipliers . . . 207
4.1.1 Unsigned Multiplier . . . 207
4.1.2 Vector Multiplier . . . 210
4.2 RISCV-V Decode . . . 213
4.2.1 Control Unit . . . 215
4.2.2 CSR Control . . . 216
4.2.3 CSR . . . 218
4.2.4 Vector Register File Control . . . 220
4.2.5 Vector Register File . . . 222
4.3 RISCV-V Execution Unit . . . 224
4.3.1 Decode Element . . . 226
4.3.2 Bypass Unit . . . 228
4.3.3 Reduction Source . . . 230
4.3.4 Swizzle result . . . 231
4.3.5 Logic ALU . . . 232
4.3.6 Arithmetic ALU . . . 239
4.3.7 Permutation ALU . . . 244
4.4 RISCV Vector Engine . . . 245

5 Conclusion **273**

A Appendix **277**
A.1 Glossary . . . 277
A.1.1 DEST_SIZE . . . 277
A.1.2 SRC_SIZE . . . 277
A.1.3 OSIZE . . . 277
A.2 Operations implemented . . . 278
A.2.1 Arithmetic Instructions . . . 278
A.2.2 Logic Instructions . . . 282
A.2.3 Compare Instructions . . . 285
A.2.4 Vector reduction Instructions . . . 292
A.2.5 Vector mask Instructions . . . 295
A.2.6 Vector permutation Instructions . . . 298
A.3 Auxiliar functions . . . 298
A.3.1 get_num_elements() . . . 298
A.3.2 zero_extend() . . . 298
A.3.3 sign_extend() . . . 299
A.3.4 min() . . . 299
A.3.5 max() . . . 299

Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence (*AI*), is an emergent and rapidly evolving technology that has gained significant relevance across a wide range of industries in recent years. Its applications span from image and speech recognition to natural language processing, autonomous systems, and advanced decision-making models. As a result, most leading technology companies are actively investing in and developing applications that leverage *AI* to enhance their products and services. In parallel, manufacturers of computing hardware, particularly those in the Personal Computer (PC) industry, are increasingly focused on delivering systems capable of executing *AI* workloads with the highest possible performance. This growing interest and adoption of *AI* across sectors has led to a rising demand for specialized hardware capable of efficiently executing complex *AI* algorithms.

To meet this demand, it becomes essential to design and implement new chips and processing units that are optimized for *AI* tasks. These chips must be capable of handling large volumes of data and executing a high number of operations in parallel. *AI* workloads are characterized by their intensive data processing requirements, where vast datasets are collected, stored, and continuously analyzed. To effectively process such data, a large number of arithmetic, logical, and comparison operations must be carried out rapidly and simultaneously.

Given these computational needs, *Graphic Processor Units* (*GPUs*) have emerged as the preferred hardware platform for implementing *AI* algorithms. Their inherent architectural design, which supports the execution of multiple operations in parallel, aligns well with the highly parallel nature of *AI* computations. The vectorized and data parallel processing capabilities of *GPUs* enable them to achieve high throughput and performance when running neural networks and other machine learning models. As a result, *GPUs* and hardware accelerators designed with similar principles play a central role in the current landscape of *AI* hardware. This context motivates the exploration and development of alternative architectures, such as the RISC-V Vector Extension, to build efficient and flexible *AI* capable hardware solutions.

1.2 Problem Statement and Context

GPUs are highly effective for implementing *AI* workloads due to their parallel processing capabilities. However, their inclusion is not always practical or feasible, especially in environments with strict constraints on cost, power consumption, or physical space. These limitations become particularly significant in the context of portable or embedded devices such as laptops, tablets, smartphones, and Internet of Things (IoT) nodes where integrating a dedicated *GPU* may not be a viable option. In such systems, the addition of a *GPU* can lead to increased energy demands, elevated system costs, and larger form factors, all of which may be detrimental to the overall design goals of portability, efficiency, and affordability.

As the demand for on device *AI* processing grows driven by applications such as voice assistants, real-time translation, object detection, and predictive modeling, it becomes increasingly important to explore alternative hardware solutions capable of providing sufficient computational capabilities without incurring the drawbacks associated with discrete *GPUs*. One promising approach to this challenge is the use of Central Processing Units (*CPUs*) that include architectural features specifically designed to support data-parallel computation.

Among these features, vector operations stand out as a key enabler for accelerating *AI* workloads on general purpose processors. Vector operations, also known as *SIMD* (Single Instruction, Multiple Data) operations, allow a single instruction to be applied simultaneously to multiple data elements. This enables significant improvements in computational throughput and efficiency when processing large datasets or performing repetitive mathematical operations, which are common characteristics of *AI* algorithms. By leveraging these capabilities, *CPUs* can partially close the performance gap with *GPUs* for certain *AI* tasks, while maintaining advantages in terms of integration, cost, and energy efficiency. As such, the exploration of vector extensions such as the RISC-V Vector Extension represents a relevant and timely approach to addressing the limitations of traditional *GPU* based acceleration in constrained computing environments.

1.3 Research question

The primary objective of this project is to investigate and determine the minimum hardware area required to implement a core subset of vector instructions that are essential for executing fundamental computational tasks. This subset includes operations such as basic arithmetic (e.g., addition and subtraction), logical operations (e.g., AND, OR, XOR), comparison operations (e.g., equal, less than), and permutation operations (e.g., vector to integer operations). These instructions form the foundation for a wide range of data-parallel applications. By identifying the minimum viable set of vector operations and quantifying the area they occupy in silicon, the project aims to assess the practicality of integrating vector processing capabilities into constrained hardware environments, such as embedded systems or portable devices.

A secondary but equally important research question addresses the maximum operating frequency that can be achieved by the vector processing unit when synthesized using *ASIC* technology. Determining the highest attainable clock frequency provides insight into the performance potential of the proposed design, especially when compared to more traditional processing units. This analysis also allows for a better understanding of the trade-offs

between area, speed, and complexity in custom hardware implementations. Together, these two questions aim to evaluate the feasibility of implementing lightweight vector engines tailored for *AI* and high-performance computing tasks, without incurring the overhead typically associated with full scale *GPU* integration.

1.4 Contributions of the Thesis

As part of the contributions of this research, the work led to the publication of a peer-reviewed article entitled “Design of an ASIC Vector Engine for a RISC-V Architecture” in the journal MDPI [2]. This publication condenses several sections of the present thesis, with a particular emphasis on the design of the vector unit and, more specifically, on the architecture and synthesis of a Vedic multiplier for vector operations. The article was published prior to the submission of this thesis, serving as an early dissemination and external validation of the research outcomes.

Chapter 2

Design

2.1 Theoretical Framework

2.1.1 SIMD

The single-instruction multiple-data or *SIMD* operations are instructions that perform multiple operations using a single instruction, for example, one *SIMD* instruction can execute multiple additions to add the elements of two vectors instead of executing multiple scalar *ADD* instructions. These instructions are useful to operate over vectors, arrays and matrices.

2.1.2 Two's complement

The Two's Complement is the preferred way to represent a binary signed number since it only has one representation for the number zero, sign extension works for positive and negative numbers, the most significant bit represents the sign and it allows to perform a subtraction using an adder [3].

The two's complement operation inverts the sign of the operand by flipping all the operand bits and adding a **1'b1**. For example, the two's complement of the number **4'b0101** is **4'b1011** which represents five and negative five in decimal notation respectively. A circuit capable of performing the two's complement operation is shown in Figure 2.1.

One of the main advantages of the Two's complement representations is the ability to perform subtractions using an adder. Instead of investing hardware in a subtractor, the subtraction can be performed by adding the minuend plus the two's complement of the subtrahend. For example, to operate **4'b0111 - 4'b0001** using the two's complement representations the operation can be translated to **4'b0111 + two's_complement(4'b0001)** that means **4'b0111 + 4'b1111 = 4'b0110**.

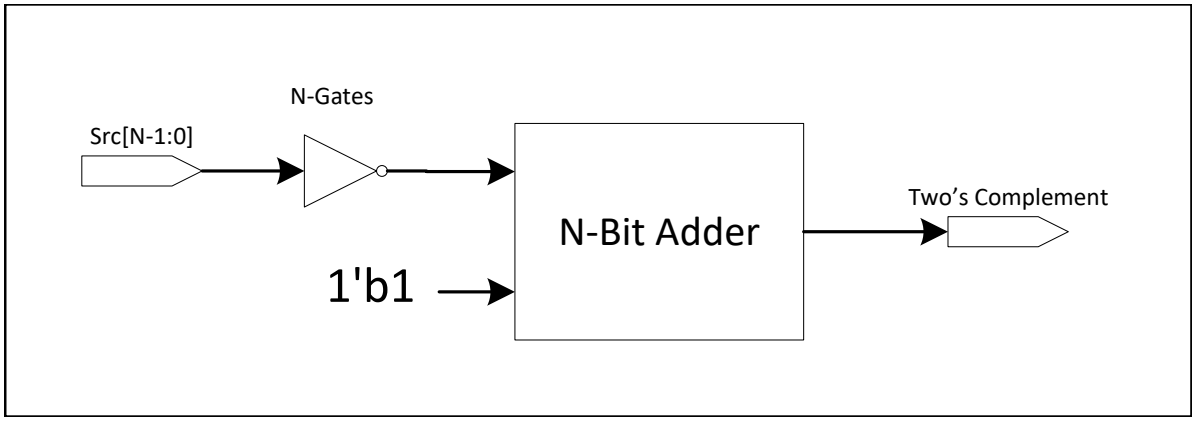


Figure 2.1: Two's complement

2.1.3 Half Adder

A *Half Adder* is a digital circuit that performs the addition of two single binary digits and provides two bits as output, the *sum* of the two numbers and the *carry_out* of the addition [4]. The circuit is described in Figure 2.2 and the outputs and latency are described with:

$$SUM = A \oplus B \tag{2.1}$$
$$CARRYOUT = A.B \tag{2.2}$$
$$latency(HA) = 1Gate \tag{2.3}$$

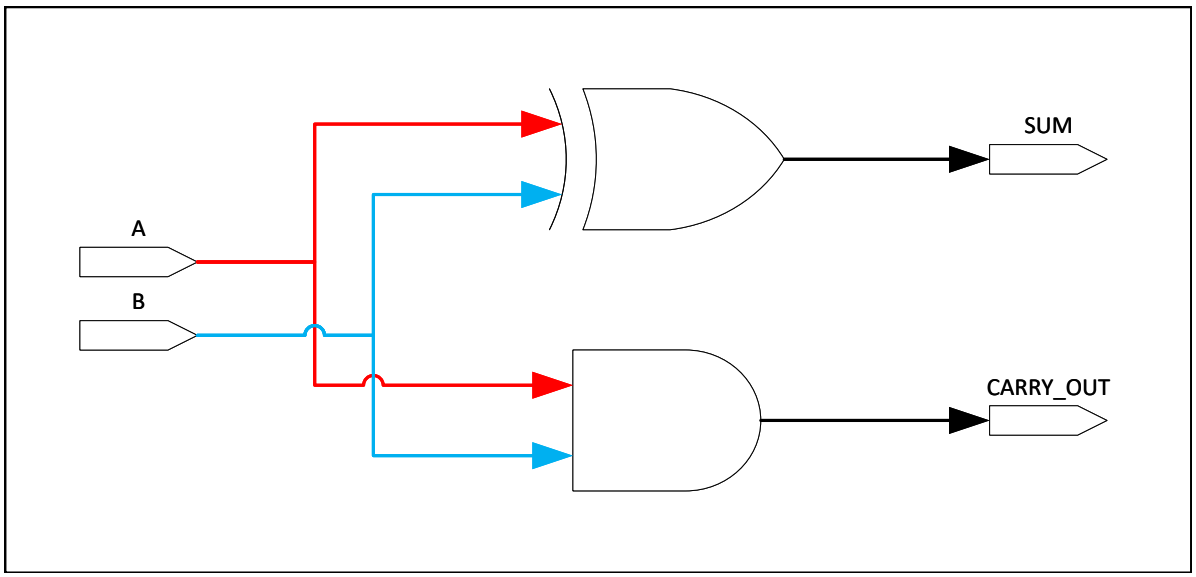


Figure 2.2: Half Adder

2.1.4 Full Adder

A *Full Adder* is a digital circuit that performs the addition of two single binary digits and one bit *carry_in*. The *carry_in* is a bit carried in from the previous less-significant stage [5]. If the adder is the less-significant stage of the addition the *Full Adder* can be replaced with a *Half Adder* to remove the *carry_in* bit or it can be used to perform a subtraction in *two's complements format* by adding one to the addition. The circuit is described in Figure 2.3 and the outputs and latency are described with:

$$SUM = A \oplus B \oplus C_IN \tag{2.4}$$

$$C_OUT = (A.B) + (A.C_IN) + (B.C_IN) \tag{2.5}$$

$$latency(FA) = 3Gates \tag{2.6}$$

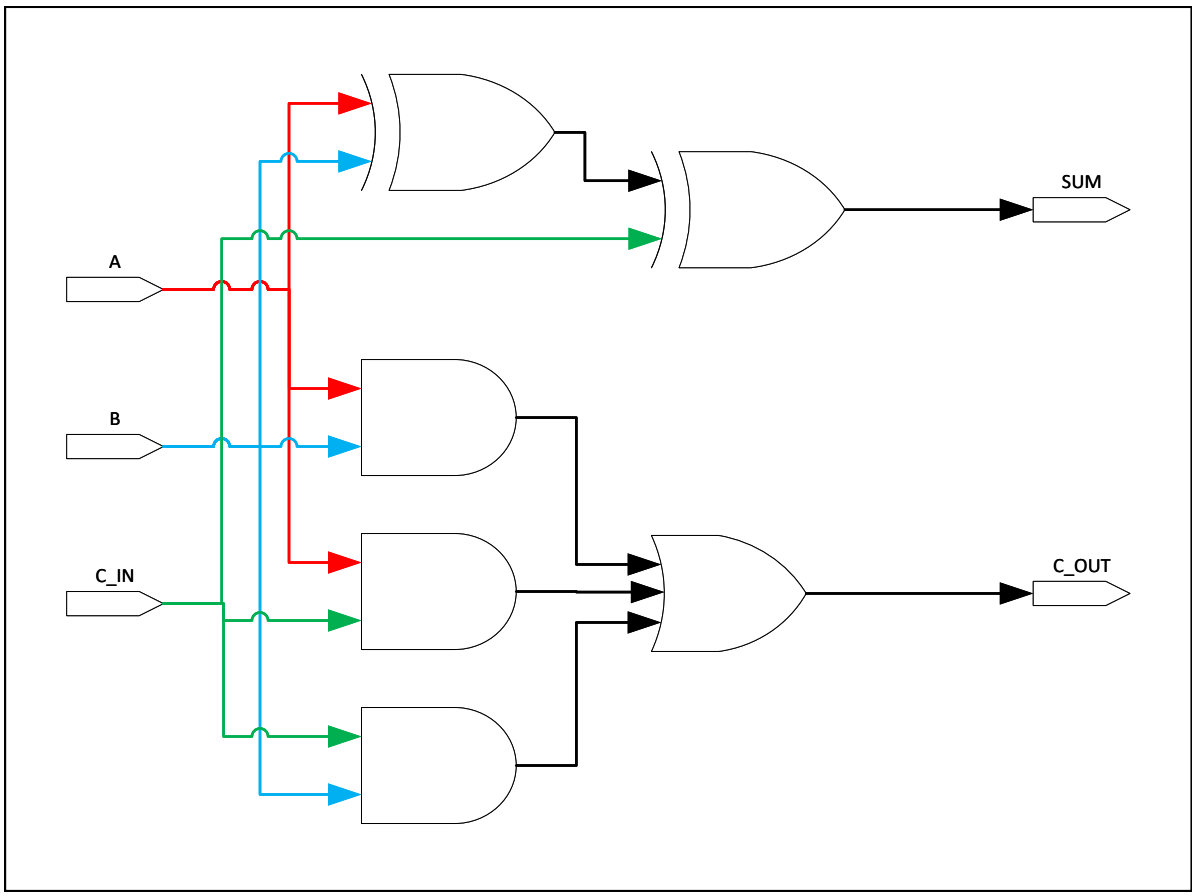


Figure 2.3: Full Adder

2.1.5 Ripple Carry Adder

The *Ripple Carry Adder* is built using N Full Adders, where N is the number of bits to be added. As shown in Figure 2.4 the Full Adders are cascaded in series by connecting the

carry_out of one stage to the *carry_in* of the next stage. The *carry_in* of the first stage can be connected to the *carry_in* of the adder or a *half adder* can be used instead if the adder does not have *carry_in* port, while the *carry_out* of the last stage can be connected to the *carry_out* of the adder. This adder is low power and is smaller in terms of area compared with other adders, but the combinational delay is its principal disadvantage, the carry needs to be propagated from the first stage of the adder to the last stage, causing the delay to be linearly proportional to N [6].

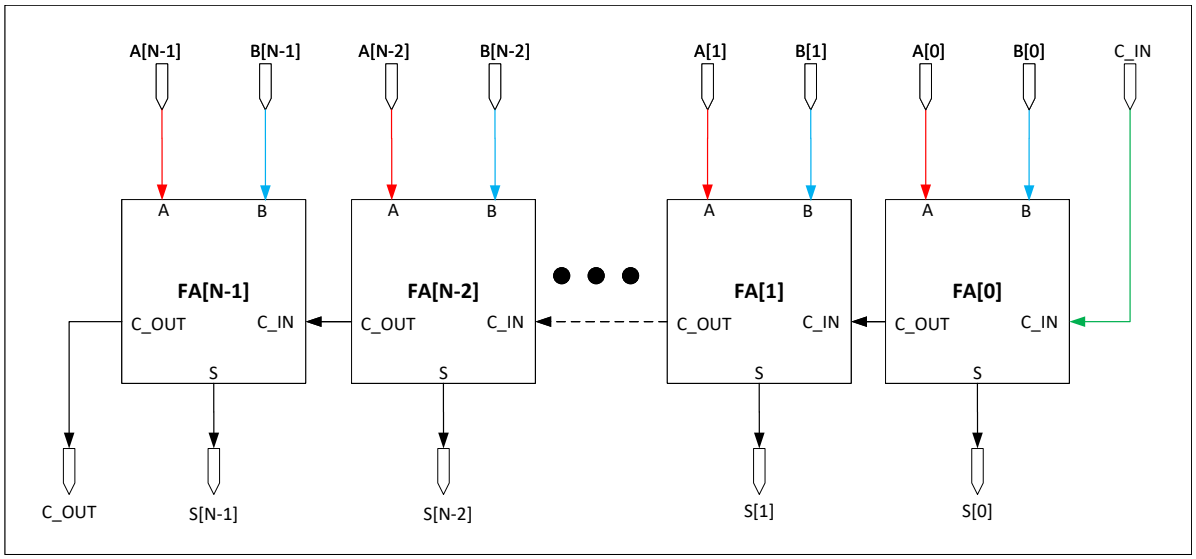


Figure 2.4: Ripple Carry Adder

2.1.6 Carry Save Adder

The *Carry Save Adder* is an adder used to add three N-bit operands by reducing the operation to an addition of two numbers, with a propagation delay of three gates, the delay of a *Full Adder*, regardless of the number of bits and produces two partial products as result, *S* and *C'*, where *S* is the sum of the three digits without carry propagation and *C'* is the carry generated by the previous individual sums, these partial results can be added to get the final result [6].

The partial products are computed with:

$$S_i = A_i \oplus B_i \oplus C_i \tag{2.7}$$

$$C'_{i+1} = (A_i.B_i) + (A_i.C_i) + (B_i.C_i) \tag{2.8}$$

As shown in Figure 2.5, first N *Full Adders* are used in parallel to add *A_i*, *B_i* and *C_i*, the *C_out* of one adder is not connected to the *C_in* of the next adder, so the output only depends on the inputs to achieve the propagation delay of three gates and producing the two partial results. Finally, the partial results are added using a N-bit *Full Adder* to get the final result.

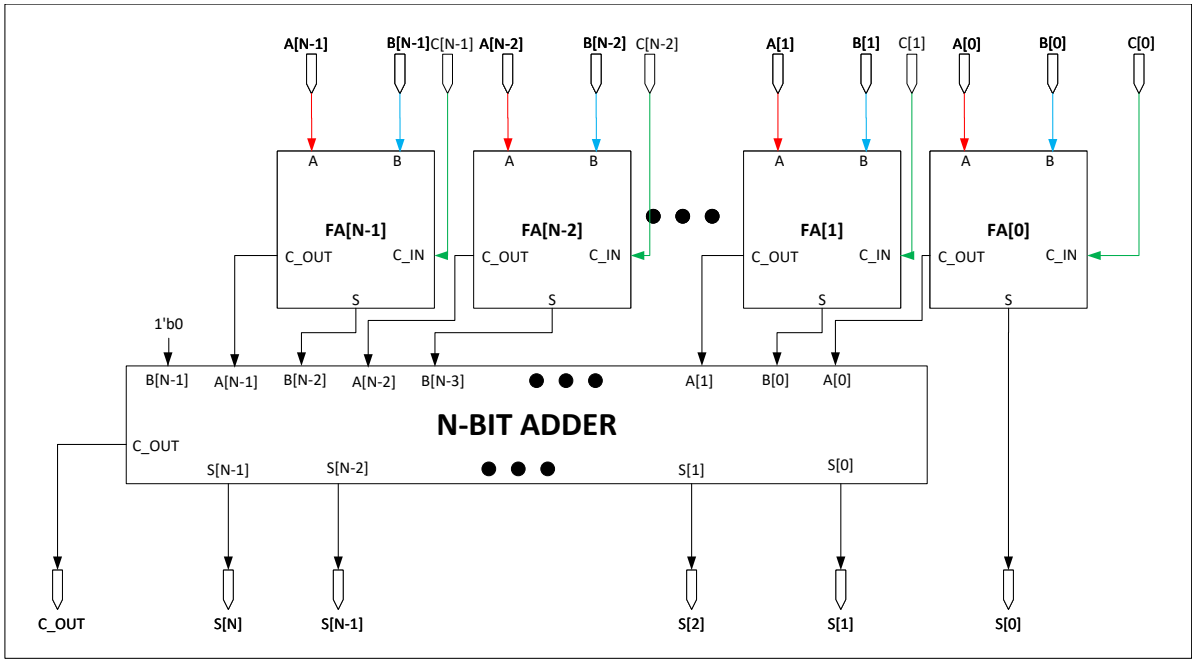


Figure 2.5: Carry Save Adder

2.1.7 Carry Look-Ahead Adder

The *Carry Look-Ahead Adder* is used to reduce the propagation latency from the carry bit by calculating the carry bit before the sum instead of waiting for the propagation of the carry like in the *Ripple Carry Adder* [6].

The *Carry Look-Ahead Adder* has three stages, propagate, generate and sum. Each stage is calculated using the following:

$$P_i = A_i \oplus B_i \tag{2.9}$$
$$G_i = A_i.B_i \tag{2.10}$$
$$C_0 = C_{in} \tag{2.11}$$
$$C_{i+1} = G_i + P_i.C_i \tag{2.12}$$
$$S_i = P_i \oplus C_i \tag{2.13}$$

Using the *carry look ahead* can be observed that the carry is generated using only the input operands of the adders, it does not need to wait for the *carry_out* from the previous adder reducing the latency of the carry bit. These equations could be used with an N-bit number, but as the position of the bit increases it requires more gates to generate the carry and gets more complex. For this reason, to add numbers with a great number of bits multiple small carry look ahead generators are used, for example, to add two 8-bit numbers two 4-bit carry look ahead generators can be used by connecting the *carry_out* of the first generator to the *carry_in* of the next one. In this way, the circuit does not get too complex and the latency is still reduced.

In Figure 2.6 is observed a N-bit *Carry Look-Ahead Adder*, the adder is built using N

Full Adders, but instead of connecting the carries of the adders like in the *Ripple Carry Adder* the carries are generated using the carry look ahead generator and are connected to the adders.

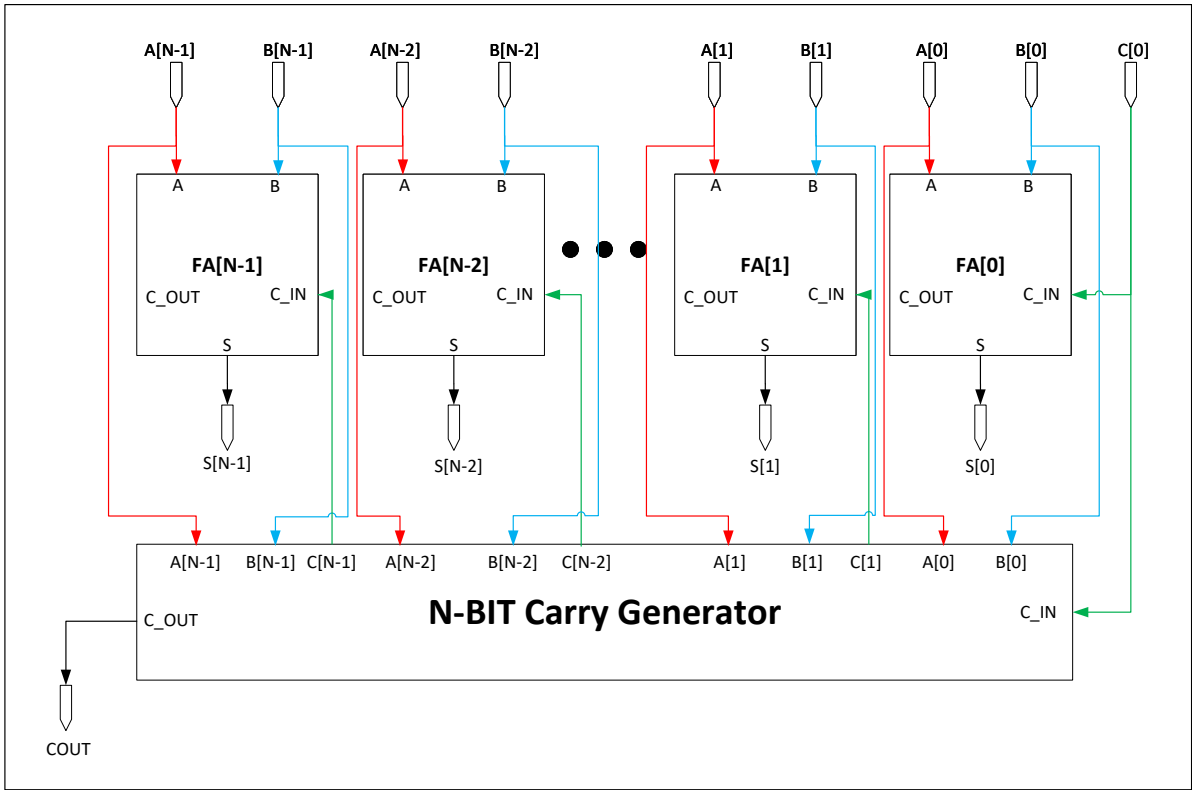


Figure 2.6: Carry Look-Ahead Adder

2.1.8 Carry Increment Adder

As shown in 2.7 the *Carry Increment Adder* is built using two N/2 Adders and one N/2 increment block made of *Half Adders* [6].

The first adder is fed with the lower part of the operands and outputs the lower part of the result and one carry bit that is sent to the increment block. The second adder is fed with the high part of the operands and its outputs are sent to the increment block. The increment block increments by one the result from the second adder if the first adder generates a carry out bit to get the final result.

This adder is slightly faster than the *Ripple Carry Adder* since the upper half of the result is precomputed and it only needs to be incremented if the lower half generates a carry out but it is not too used since it also has a carry chain delay.

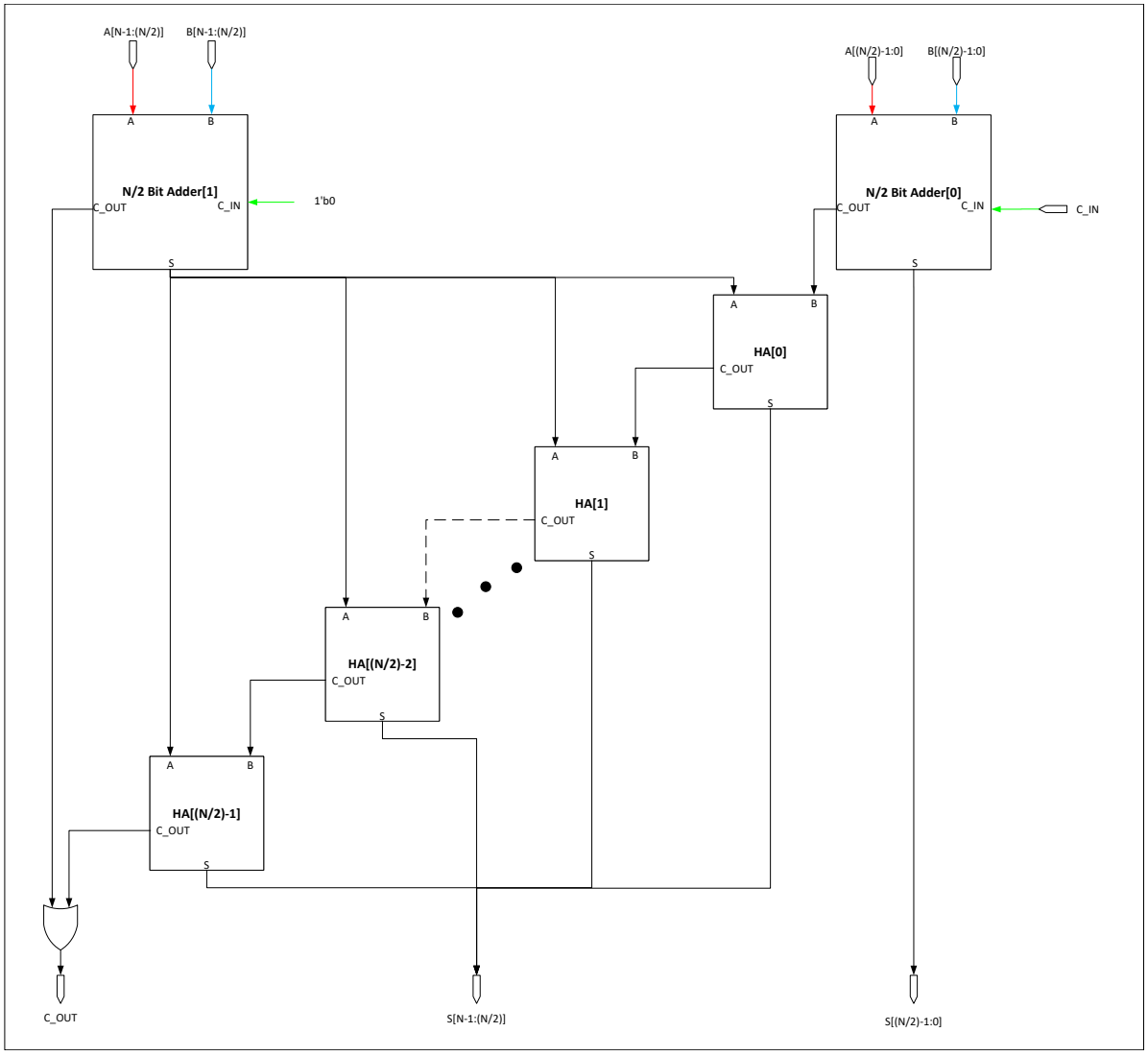


Figure 2.7: Carry Increment Adder

2.1.9 Carry Skip Adder

The Carry Skip Adder is used to improve the delay of the carry chain by adding a carry skip chain to a *Ripple Carry Adder*. This chain skips the carry from one block to the next block faster than the traditional carry chain of the *Ripple Carry Adder* [6].

The Figure 2.8 shows one block of a Carry Skip Adder. Each Full Adder generates its propagate bit, if all the propagate bits are one and the carry in bit is one the carry can be skipped or propagated to the next block, but if not all the propagate bits are one or the carry in is not one it needs to wait for the final computation of carry out from the *Ripple Carry Adder* to get the correct carry out bit.

The improvement in the delay of this adder is just probabilistic, it could improve the delay of a big sum if many blocks skip the adder, but in some cases, the delay of this adder could be worse of a *Ripple Carry Adder* if any block skips the carry.

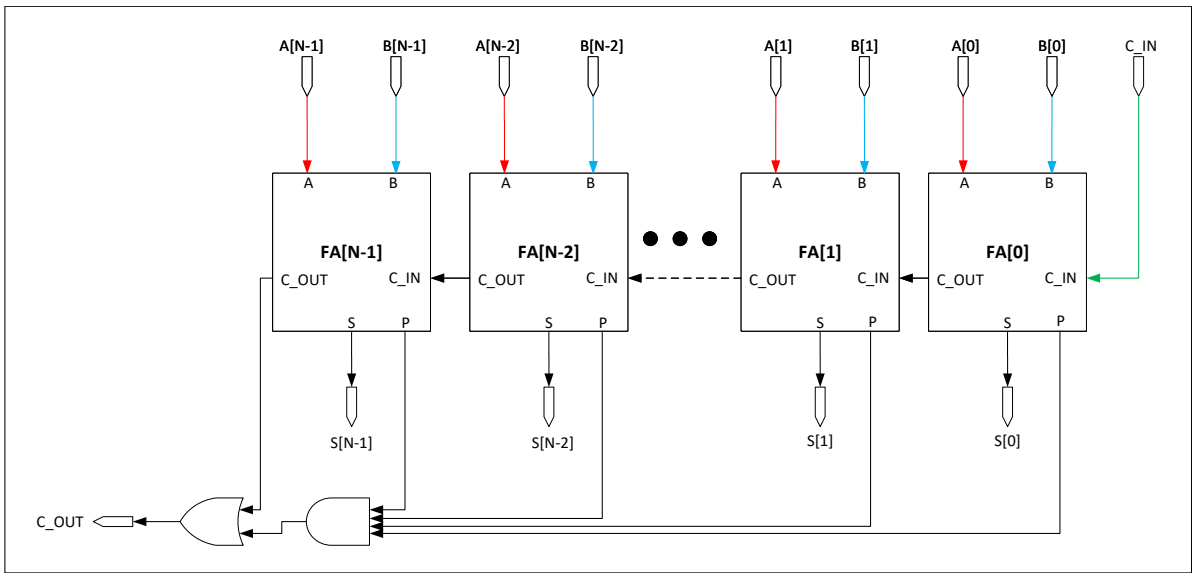


Figure 2.8: Carry Skip Adder

2.1.10 Carry Select Adder

As shown in 2.9 the *Carry Select Adder* is built using two *Ripple Carry Adders* in parallel, both adders add the same operands, but one adder assumes a carry in of one while the other adder assumes a carry in of zero. After the addition one multiplexer selects the correct result depending on the carry in [6].

This adder increments the speed of the adder compared to a *Ripple Carry Adder* but this speed can be increased if the adder is combined with a different adder like shown in Figure 2.10. The lower part of the operands is added with a conventional adder, i.e *Ripple Carry Adder*, at the same time the high part of the operands are added in parallel using a *Carry Select Adder*, once the addition of the lower part is completed the carry out of the lower part selects the correct result from the *Carry Select Adder*. This structure cut down the carry chain by half reducing the latency of the adder.

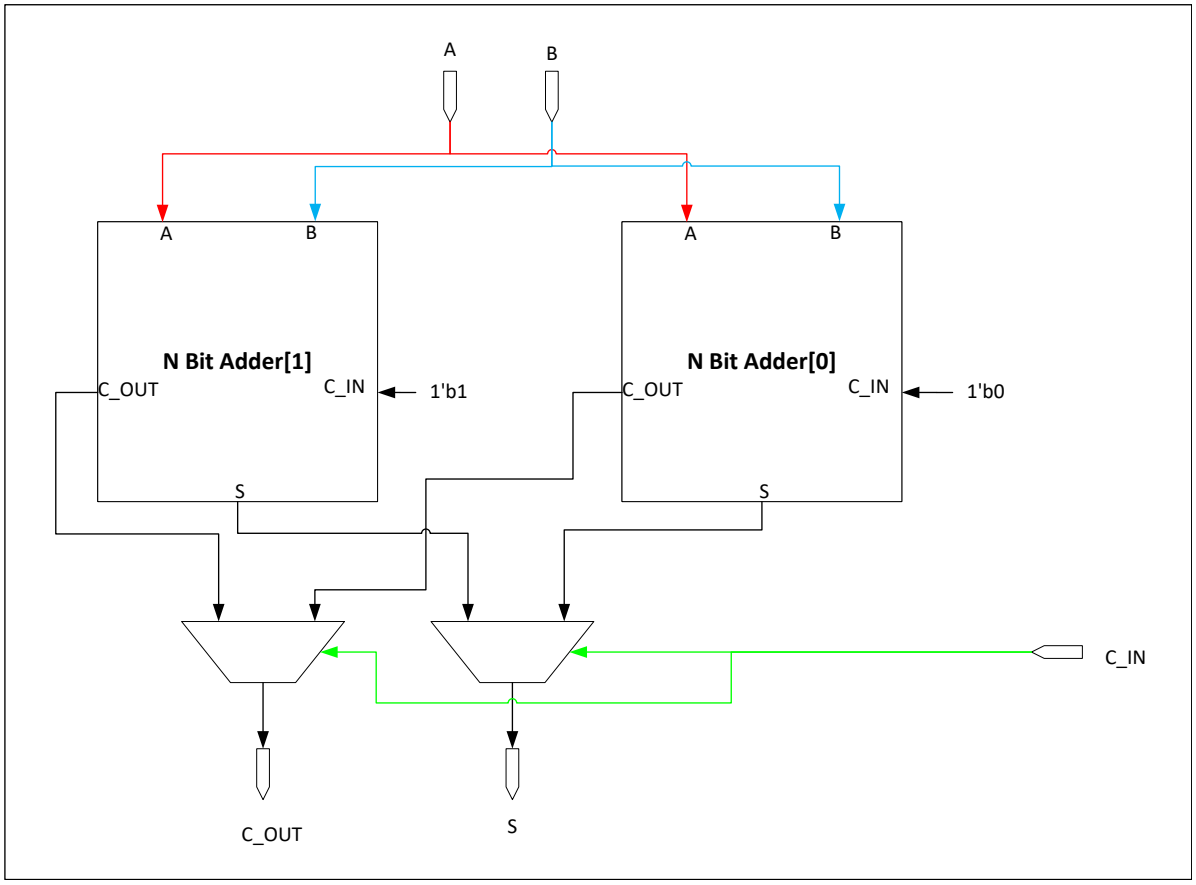


Figure 2.9: Carry Select Adder

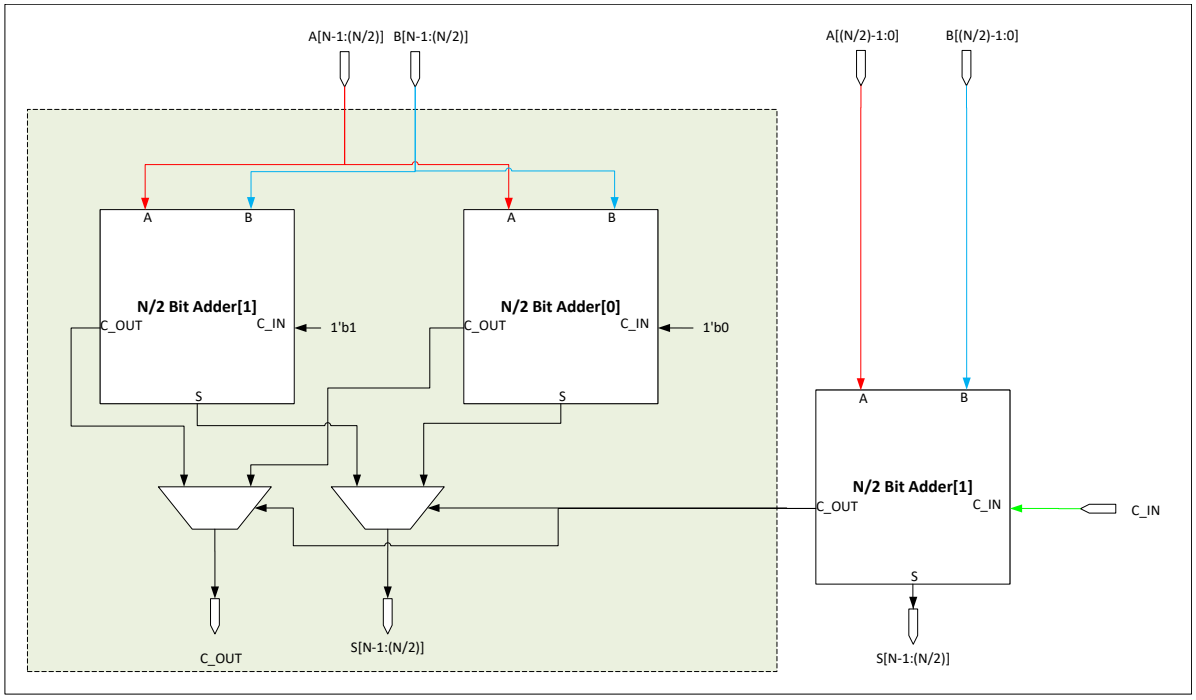


Figure 2.10: Carry Select Adder Big

2.1.11 Kogge-Stone Adder

The Kogge-Stone Adder is similar to the Carry Look-Ahead Adder. As shown in Figure 2.11 N full adders are implemented to perform the addition of the operands, but the carries are not connected like in a Ripple Carry Adder, they come from a Carry Generator like in the Carry Look-Ahead Adder to calculate the carries earlier and increment the speed of the adder.

The Kogge-Stone Adder is built in three stages, *Pre-Processing Block*, *Carry Generation Block* and *Post Processing Block* [7].

As shown in Figure 2.12 the adder is composed of $log2(N) + 1$ layers of Carry blocks, the first layer, shown in pink, is the *Preprocessing Block*, the following layers, shown in yellow, are the *Carry Generation Blocks* and the *Post Processing Blocks*.

In the *Preprocessing Block* the generate and propagate bits are computed using only the input operands as shown in:

$$P_{0,i} = A_i \oplus B_i \tag{2.14}$$
$$G_{0,i} = A_i.B_i \tag{2.15}$$

In the *Carry Generation Blocks* the carries of the blocks are computed using the generate and propagate bits from previous stages as shown in:

$$BIAS = 2^{j-1} \tag{2.16}$$
$$P_{j,i} = \begin{cases} P_{(j-1),i}, \text{ if } i < BIAS \\ P_{(j-1),i}.P_{(j-1),(i-BIAS)}, \text{otherwise} \end{cases} \tag{2.17}$$
$$G_{j,i} = \begin{cases} G_{(j-1),i}, \text{ if } i < BIAS \\ (P_{(j-1),i}.G_{(j-1),(i-BIAS)}) + G_{(j-1),i}, \text{otherwise} \end{cases} \tag{2.18}$$

Finally, in the *Post Processing Block* the final sum is computed using the operands and the carries generated in the *Carry Generation Blocks*

$$S_i = P_{(log2(N)),i} \oplus G_{(log2(N)),(i-1)} \tag{2.19}$$

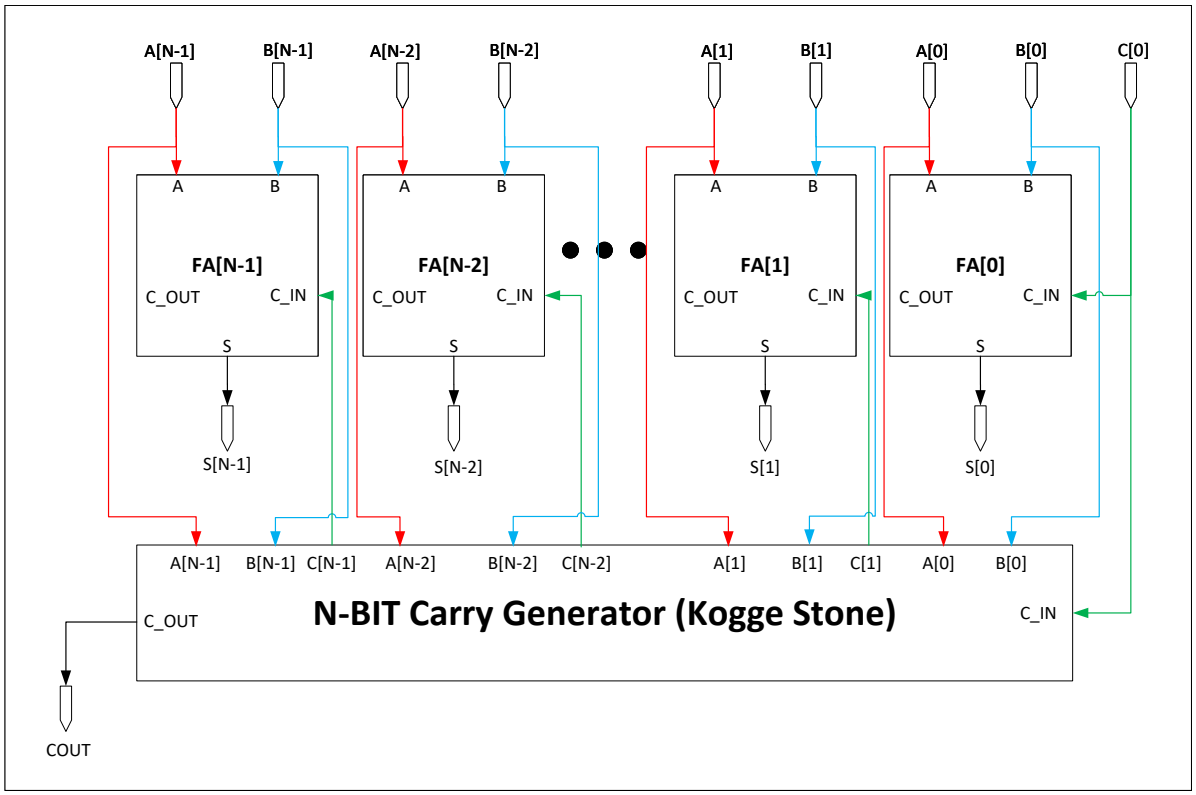


Figure 2.11: Kogge-Stone Adder

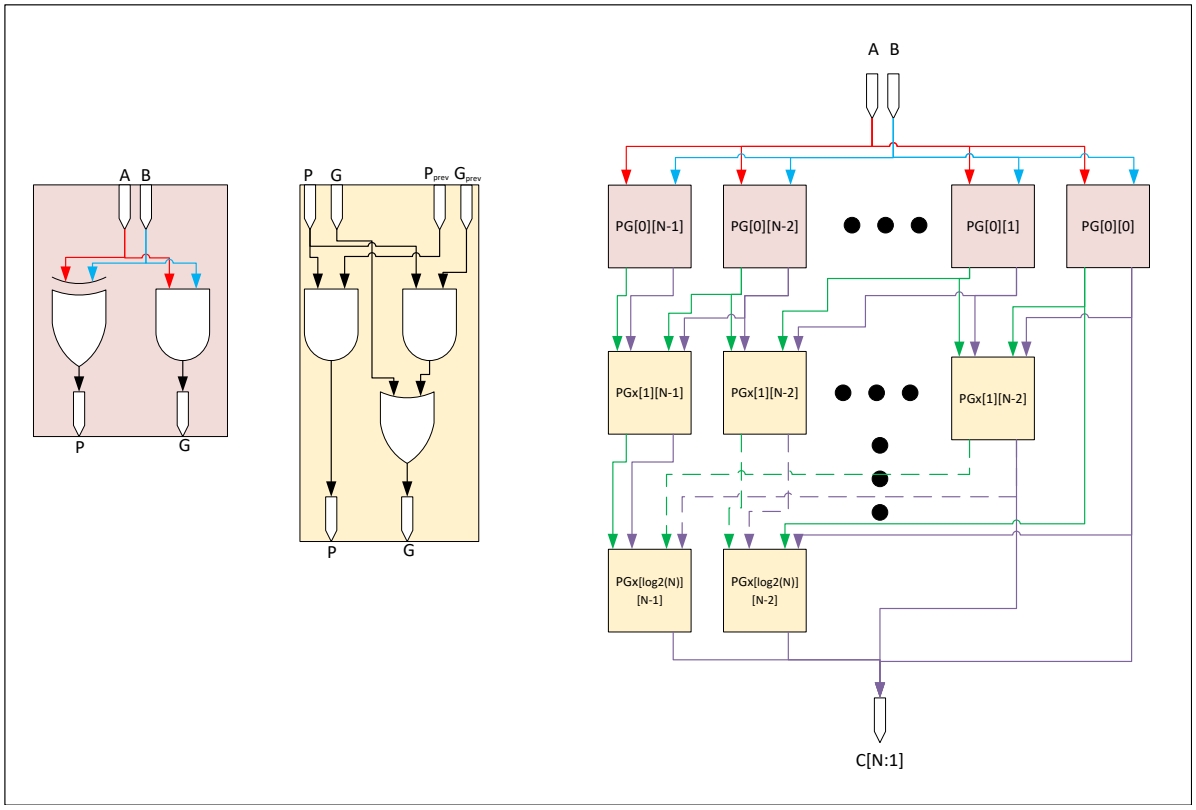


Figure 2.12: Kogge-Stone Adder Carry Generation

2.1.12 Barrel Shifter

A Barrel Shifter is a circuit capable of shifting left or right an N-bit word by a specified number of bits on one clock cycle [8].

There are many ways to build a barrel shifter but the most common is to use layers of multiplexors as shown in Figure 2.13. The word to be shifted is connected to the first layer of multiplexors, the connection of the inputs depends on the direction of the shifter, whether it is a left shifter, a right shifter or a bidirectional shifter. The less significant bit of the *shift_amount* is connected to the first layer, the first layer shifts the word by one position. The output of the first layer is connected to the second layer of multiplexors, the connection depends again on the direction of the shifter, and the next significant bit of the *shift_amount* is connected to the control input of this layer, in this case this layer shifts the word by two positions, the number of bits shifted by each layer can be calculated with:

$$shift_layer = 2^{LAYER} \tag{2.20}$$

The number of layers of multiplexers depends on the number of bits that the word can be shifted in a single cycle, usually the number of layers is $log2(N)$ to be able to shift the whole word in a single cycle. The last layer outputs the word shifted by the amount indicated in *shift_amount*.

The inputs of the multiplexors that don't have a direct connection with the previous layer or with an input in the case of the first layer can be connected directly to a zero in the case of logical shifts or to the most significant bit of the word in case of an arithmetical shift to perform a sign extend.

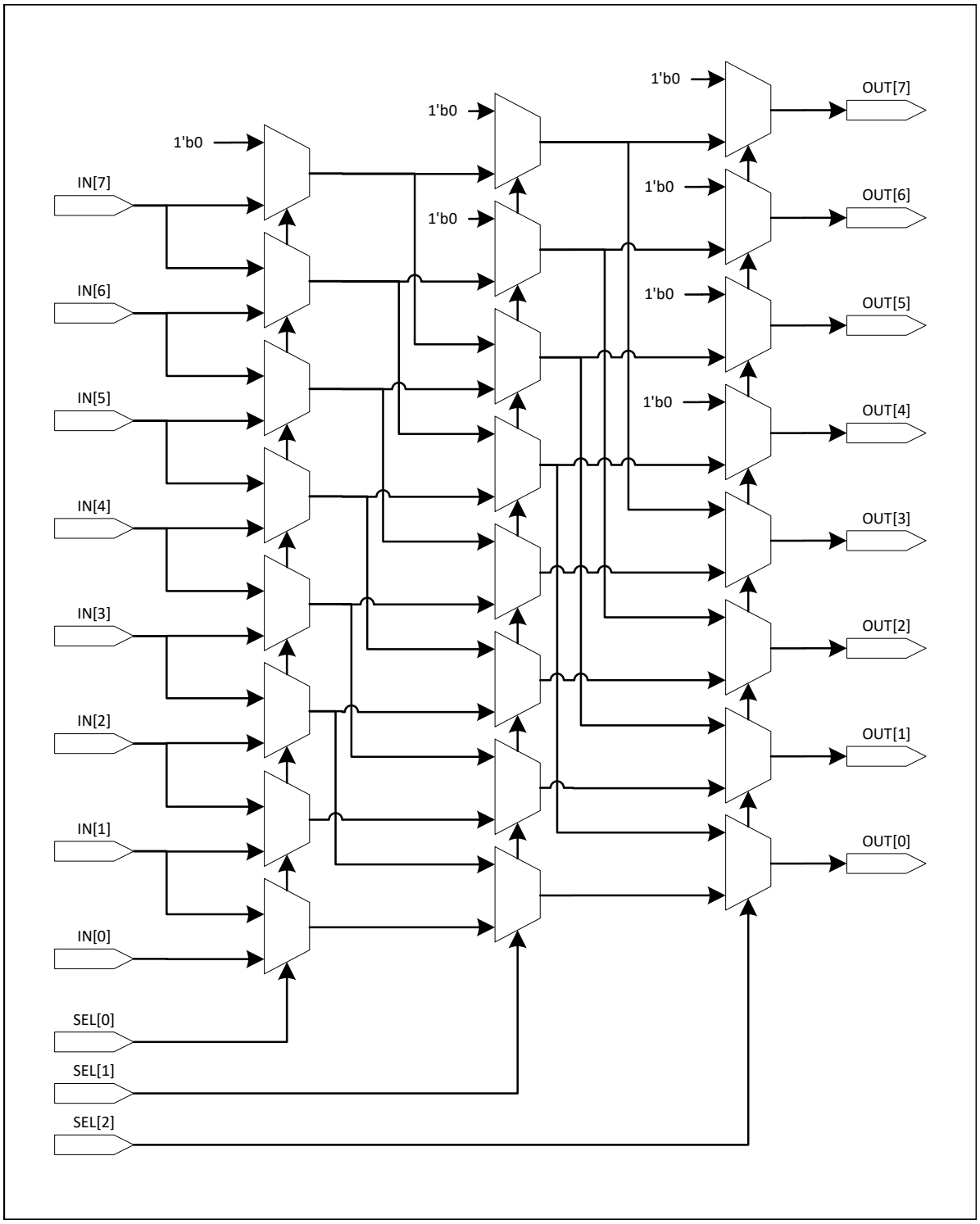


Figure 2.13: Right Barrel Shifter

2.1.13 Array Multiplier

The Array multiplier is the simplest multiplier used to multiply two N-bit numbers. This multiplier consists of N layers of partial products and N-1 layers of adders to add the partial products. Each layer of the partial products is generated using N AND gates, all the bits of the Multiplicand are anded with the bit[L] of the Multiplier, where L represents the level of the layer, on each level the partial product is shifted L places to the left. The adders are used to add all the layers of partial products generated to get the final result [9]. A diagram of an array multiplier is shown in Figure 2.14 and the number of gates needed in 2.21.

$$array_multiplier_gates = J * N + (J-1)Adder \tag{2.21}$$

$$array_multiplier_gates(RCA) = J * N + J(HA) + (J-1)(N-1)FA - 1FA \tag{2.22}$$

$$\tag{2.23}$$

The Array multiplier is simple but its main disadvantage is the combinational delay. In Figure 2.15 is observed the critical path of the array multiplier, all the layers of partial products can be computed in parallel and have a delay of only one AND gate, but most of the delay comes from the adders, the total delay of the adders is the delay of 2N-1 *Full Adders* and one *Half Adder* since it needs to wait for the carry propagation of all the stages as shown in:

$$Array_multiplier_delay = dly(AND) + (2N-1)dly(FA) + dly(HA) \tag{2.24}$$

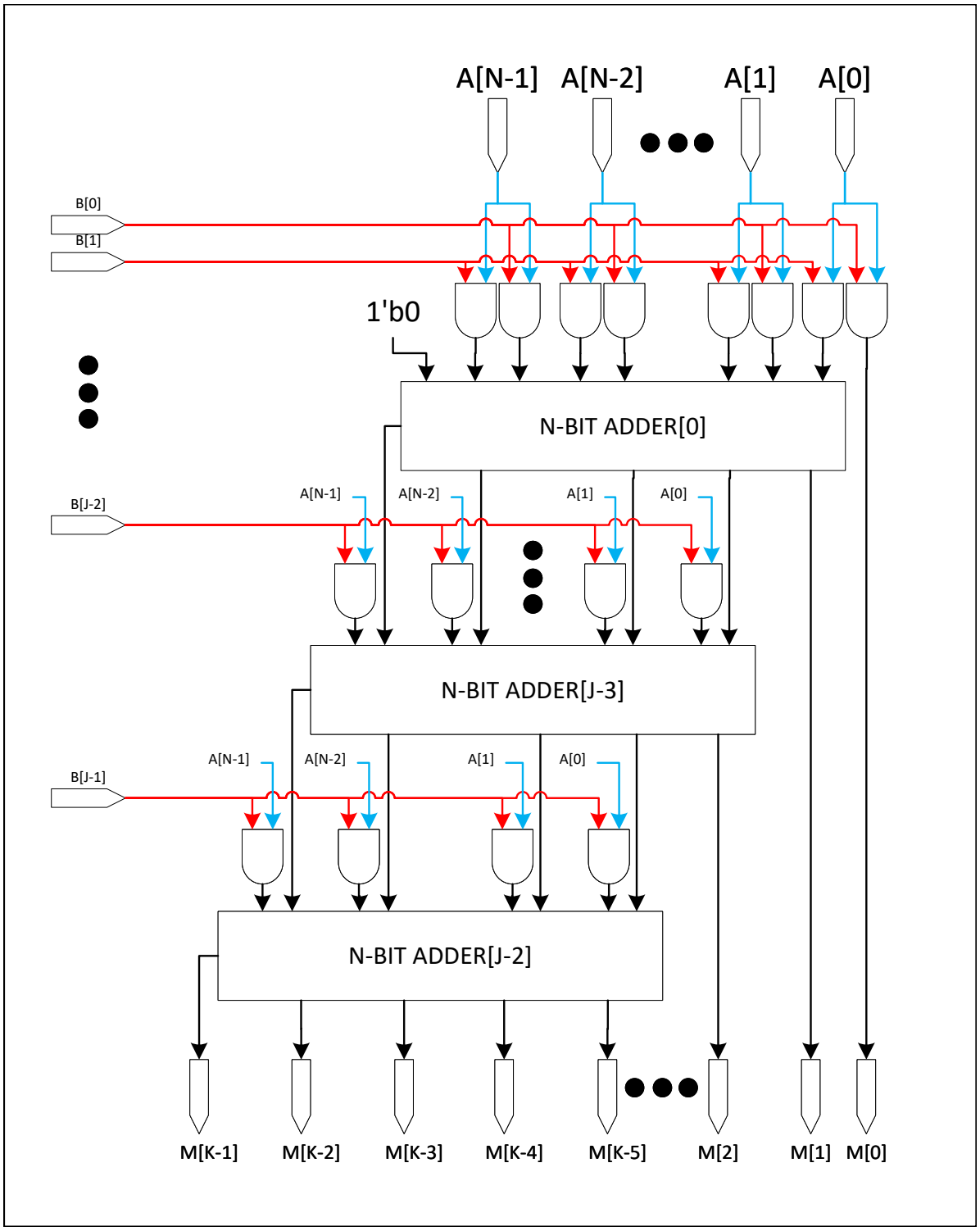


Figure 2.14: Array Multiplier

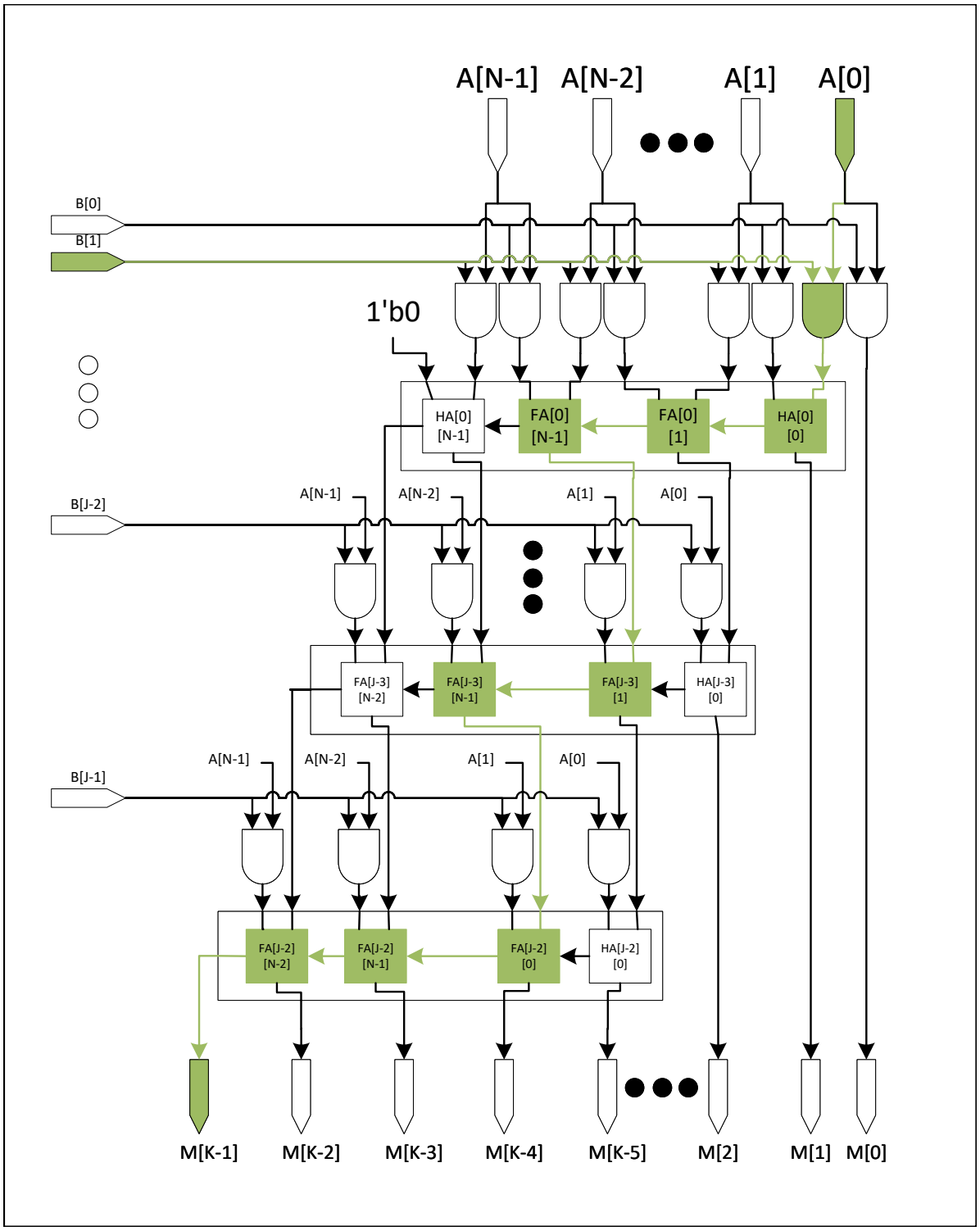


Figure 2.15: Critical path array multiplier

2.1.14 Vedic Multiplier

The *Vedic Multiplier* is based on an ancient Indian mathematic system rediscovered and founded by Sri Bharati Krishna Tirthaji between 1911 and 1918. The Vedic mathematics is based on sixteen Sutras, one of the Sutras is the Urdhava Tiryakbhyam Sutra, which means vertical and crosswise calculation and it is the sutra where the *Vedic Multiplier* is based [10]

The Urdhava Tiryakbhyam Sutra is used to multiply two decimal numbers, it multiplies the digits of the number vertically and crosswise to generate partial products and then adds the partial products to get the final result as shown in Figure 2.16.

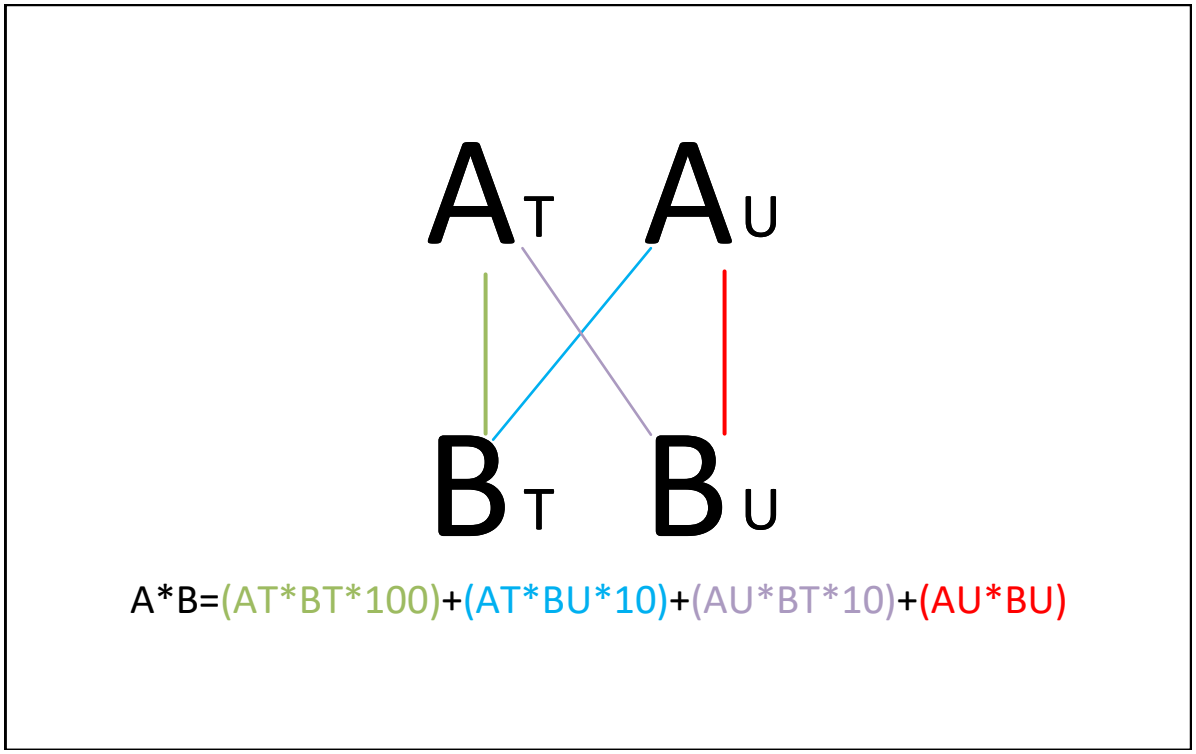


Figure 2.16: Urdhava Tiryakbhyam Sutra

This method can be easily understood with an example, let's consider:

$$82 * 76 \tag{2.25}$$

Since it is a base 10 system the numbers can be divided into units and tens.

$$82 * 76 = (80 + 2)(70 * 6) \tag{2.26}$$

Then the distributive property of the multiplication is applied.

$$(80 + 2)(70 * 6) = (80 * 70) + (80 * 6) + (70 * 2) + (2 * 6) \tag{2.27}$$

The numbers can be factorized to simplify the operation.

$$\begin{aligned}(80 * 70) &= ((10 * 8) * (10 * 7)) \\ (80 * 6) &= ((10 * 8) * (6)) \\ (70 * 2) &= ((10 * 7) * (2)) \\ (2 * 6) &= (2 * 6)\end{aligned} \tag{2.28}$$

The factors can be reordered by applying the associative property of the multiplication.

$$\begin{aligned}((10 * 8) * (10 * 7)) &= ((10 * 10) * (8 * 7)) = (100 * (8 * 7)) \\ ((10 * 8) * (6)) &= ((10) * (8 * 6)) \\ ((10 * 7) * (2)) &= ((10) * (7 * 2)) \\ (2 * 6) &= (2 * 6)\end{aligned} \tag{2.29}$$

Finally, applying all the properties mentioned above the equation can be expanded from (2.25) to (2.30) getting the Vedic Formula.

$$82 * 76 = (100 * (8 * 7)) + (10 * (8 * 6)) + (10 * (7 * 2)) + (2 * 6) \tag{2.30}$$

The result of the multiplication is obtained by multiplying the factors and adding the partial products.

$$\begin{aligned}82 * 76 &= (100 * (56)) + (10 * (48)) + (10 * (14)) + (12) \\ 82 * 76 &= 5600 + 480 + 140 + 12 \\ 82 * 76 &= 6232\end{aligned} \tag{2.31}$$

This example can be expanded to more digits by getting more cross-products with their respective offset and adding all the partial products.

This method can also be overlapped to a base two system, in other words to the binary system, and by supporting multiplications with more than one bit the operands can be split by half in two parts, low and high parts, reducing the number of multipliers and adders needed, the number of partial products generated and intermediate stages, and getting a generic way to implement the Vedic multiplication in a computer.

Let's take the same numbers in binary.

$$0101_0010 * 0100_1100 \tag{2.32}$$

Now instead of dividing the number by units and tens let's divide the number by half.

$$\begin{aligned}0101_0010 &= (0101_0000 + 0000_0010) \\ 0100_1100 &= (0100_0000 + 0000_1100)\end{aligned} \tag{2.33}$$

Apply the distributive property.

$$\begin{aligned} 0101_0010 * 0100_1100 &= (0101_0000 + 0000_0010) * (0100_0000 + 0000_1100) \\ 0101_0010 * 0100_1100 &= (0101_0000 * 0100_0000) + (0101_0000 * 0000_1100) \\ &\quad + (0100_0000 * 0000_0010) + (0000_0010 * 0000_1100) \end{aligned} \tag{2.34}$$

Factorize elements.

$$\begin{aligned} 0101_0000 &= (0101\ *\ 'd16) \\ 0100_0000 &= (0100\ *\ 'd16) \\ 0000_0010 &= (0010) \\ 0000_1100 &= (1100) \end{aligned} \tag{2.35}$$

Apply associative property.

$$\begin{aligned} (0101\ *\ 'd16) * (0100\ *\ 'd16) &= (('d16\ *\ 'd16) * (0101 * 0100)) = (('d256) * (0101 * 0100)) \\ (0101\ *\ 'd16) * (1100) &= (('d16) * (0101 * 1100)) \\ (0100\ *\ 'd16) * (0010) &= (('d16) * (0100 * 0010)) \\ (1100) * (0010) &= (1100 * 0010) \end{aligned} \tag{2.36}$$

The multiplications by a power of two can be substituted with a left shift.

$$\begin{aligned} (('d256) * (0101 * 0100)) &= ((0101 * 0100) << 'd8) \\ (('d16) * (0101 * 1100)) &= ((0101 * 1100) << 'd4) \\ (('d16) * (0100 * 0010)) &= ((0100 * 0010) << 'd4) \\ (1100 * 0010) &= (1100 * 0010) \end{aligned} \tag{2.37}$$

The final equation is obtained.

$$\begin{aligned} 0101_0010 * 0100_1100 &= ((0101 * 0100) << 'd8) + ((0101 * 1100) << 'd4) \\ &\quad + ((0100 * 0010) << 'd4) + (1100 * 0010) \end{aligned} \tag{2.38}$$

The final result is obtained by applying the operations.

$$\begin{aligned} ((0101 * 0100) << 8) &= (0001_0100 << 'd8) = 0001_0100_0000_0000 \\ ((0101 * 1100) << 4) &= (0011_1100 << 'd4) = 0011_1100_0000 \\ ((0100 * 0010) << 4) &= (1000 << 'd4) = 1000_0000 \\ (1100 * 0010) &= 0001_1000 \\ 0101_0010 * 0100_1100 &= 0001_0100_0000_0000 + 0011_1100_0000 + 1000_0000 + 0001_1000 \\ 0101_0010 * 0100_1100 &= 0001_1000_0101_1000 = 'd6232 \end{aligned} \tag{2.39}$$

The general formula can be observed in:

$$binary_vedic = ((high_srca * high_srcb) << N) + ((high_srca * low_srcb) << N/2) + ((low_srca * high_srcb) << N/2) + (low_srca * low_srcb) \quad (2.40)$$

The *Vedic Multiplier* produces a total of four 2N-bits partial products that need to be added, it would require three 2N-bits adders, but all the partial products have static '0 due to the shifting of the partial products as shown in Figure 2.17. The lower N/4 bits, highlighted in red, can be bypassed directly without an adder since only one partial product has a valid result while the other three have static zeroes. The partial products generated by the cross factors, the products with an N/2 shift left, highlighted in blue can be added using an N-bit adder that is called *adder_1*. The result from *adder_1* can be added with the higher N/2 bits of the *low*low* partial product using an N-bit adder while the higher bits are static zeroes, this adder is highlighted in green and is called *adder_2*. The higher N/2 bits from the result of *adder_2* are added with the *high*high* partial product, this adder is highlighted in purple and is called *adder_3*, the carry outs of the *adder_1* and *adder_2* are added with a *Half Adder* to produce two bits that can affect *adder_3* and are connected in the two most significative bits adjacent to the result of *adder_2* while the remaining most significant bits are static zeroes.

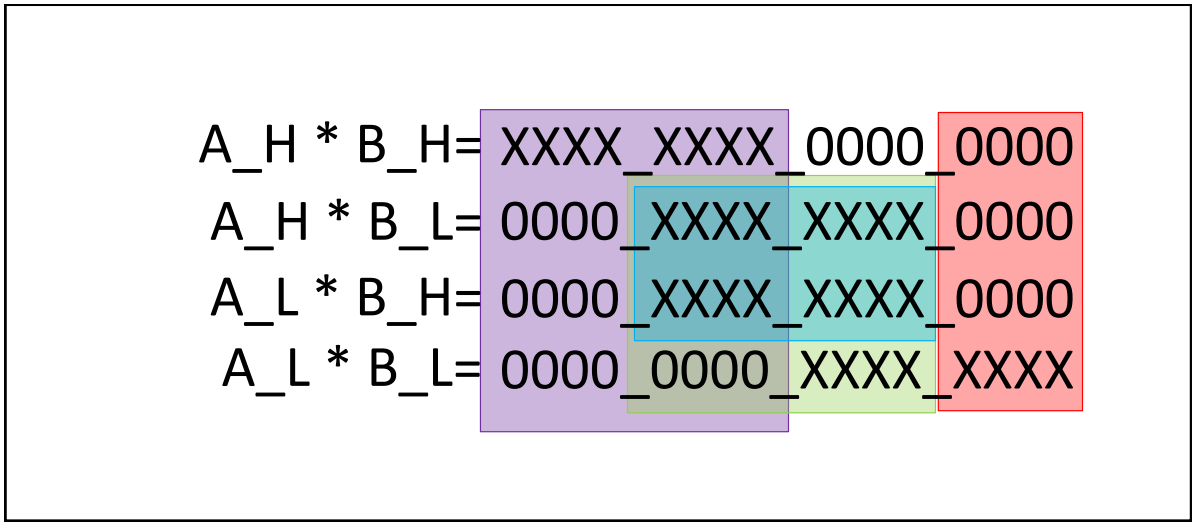


Figure 2.17: Vedic multiplication

The hardware needed to implement a N-bit *Vedic Multiplier* is shown in Figure 2.18. The diagram shows a generic adder, this adder can be replaced with any type of adder. The auxiliary multipliers are also represented as generic N/2 multipliers, but these multipliers can also be implemented using N/2 *Vedic Multiplier*, that recursively can be implemented with N/4 *Vedic Multiplier*. The smallest *Vedic Multiplier* is the multiplier with a width of 2-bits and has a different structure shown in Figure 2.19, with a combinational delay shown in:

$$2bit_vedic_delay = latency(AND) + 2latency(HA) \quad (2.41)$$

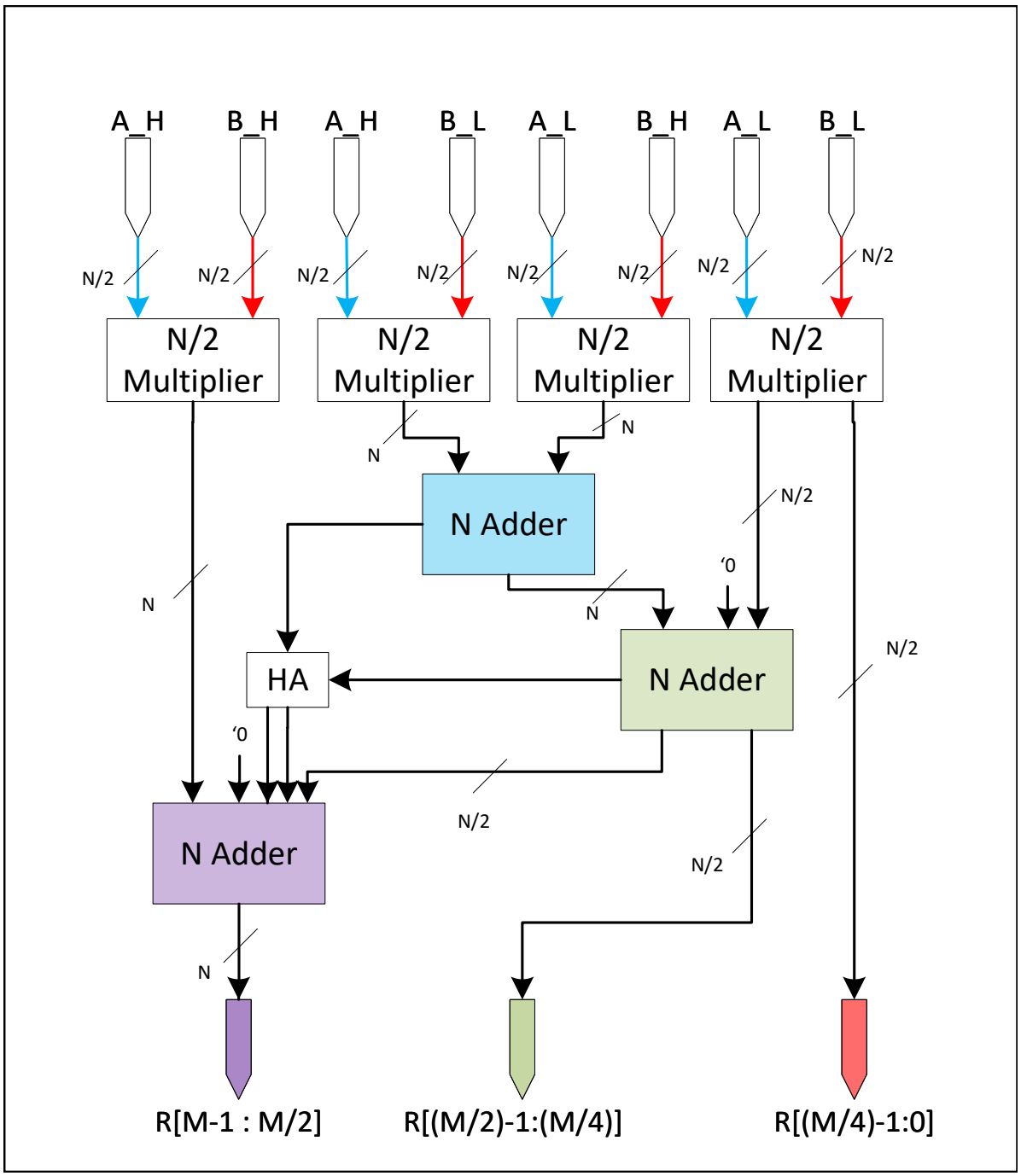


Figure 2.18: Vedic Multiplier

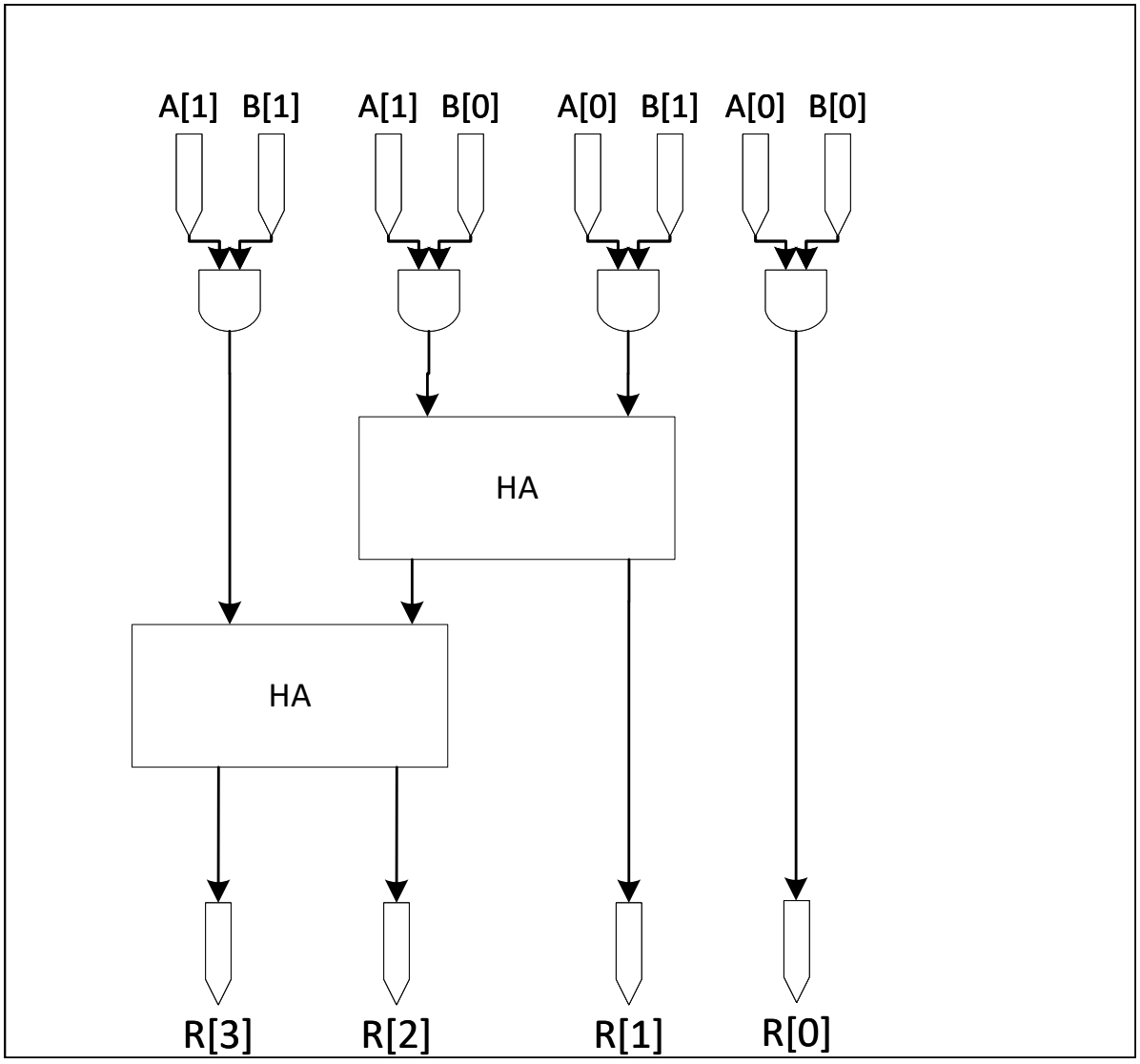


Figure 2.19: Vedic 2-Bits

The *Vedic Multiplier* requires more hardware and gates compared with the *Array Multiplier*, the number of gates required is shown in (2.42), but its main advantage is its speed, the structure of the *Vedic Multiplier* allows the computation of the sum of the partial products in parallel. While the delay of an *Array Multiplier* is shown in (2.24) the delay of the *Vedic Multiplier* is shown in Figure 2.20 and (2.43).

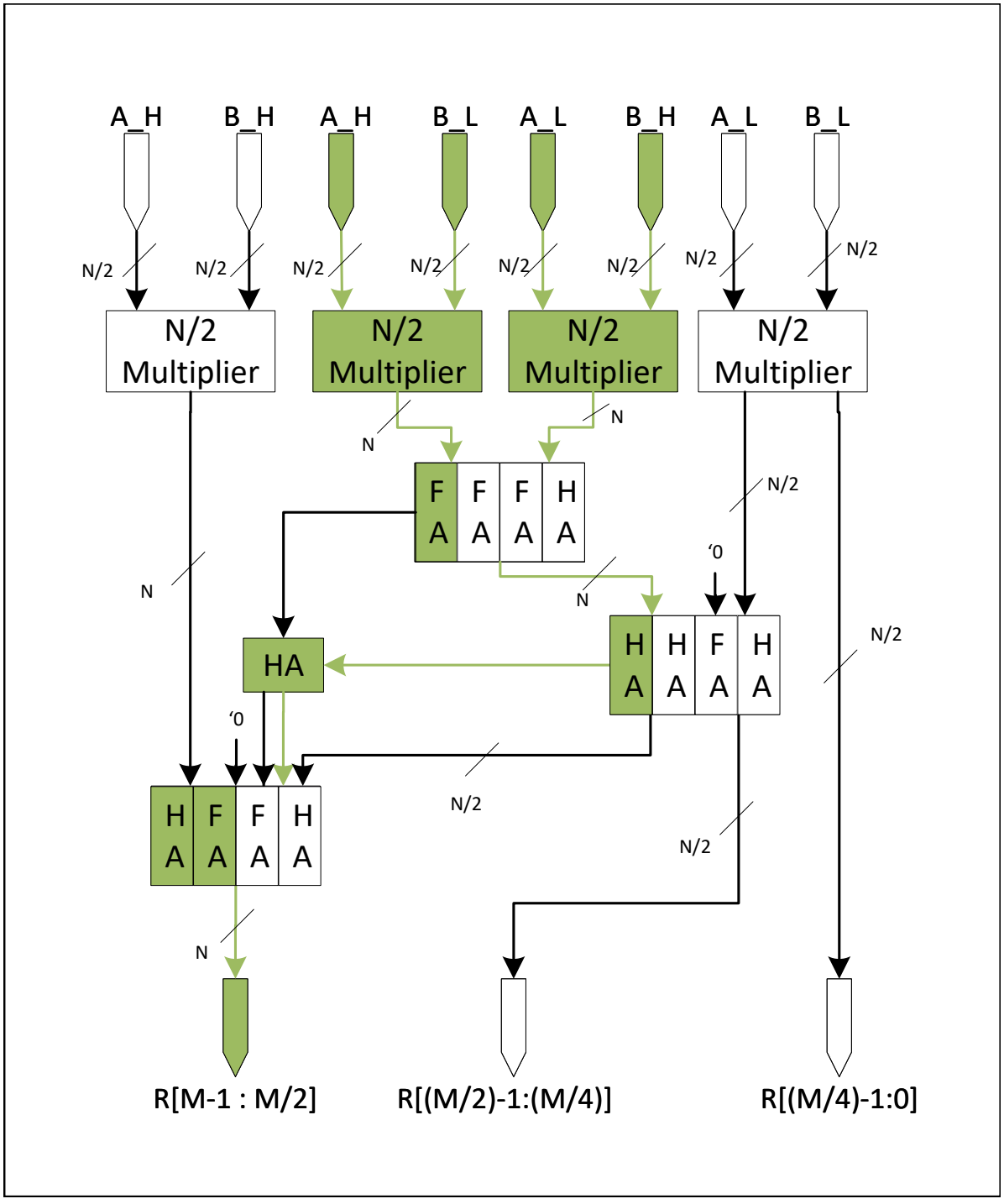


Figure 2.20: Critical path Vedic Multiplier

$$Vedic_gates[N] = \begin{cases} 4 + 2(HA), \text{ if } N = 2 \\ 4vedic_gates[N/2] + 2HA + NHA + 2NFA - 1FA \text{ if } N > 2 \end{cases} \tag{2.42}$$

$$Vedic_latency[N] = \begin{cases} latency(AND) + 2latency(HA), \text{ if } N = 2 \\ vedic_latency[N/2] + 3latency(HA) + 0.5Nlatency(FA) \text{ if } N > 2 \end{cases} \tag{2.43}$$

As shown in Table 2.1 the *Vedic Multiplier* consumes about 30% more gates than the *Array Multiplier* and it is almost 2 times faster than the *Array Multiplier*. It is important to note that these results are only theoretical and it's hard to get an exact equation, especially for the delay of the *Vedic Multiplier*, empirical results are shown in section *Multiplier Synthesis*.

Table 2.1: Array VS Vedic Multiplier comparison

	Array Multiplier		**Vedic Multiplier**		**Array vs Vedic**			
N	**Gates**	**Delay**	**Gates**	**Delay**	**Δ Gates**	**%Gates**	**Δ Delay**	**%Delay**
2	8	3	8	3	0	100%	0	100%
4	64	23	79	12	15	123%	-11	191%
8	320	47	411	27	91	128%	-20	174%
16	1408	95	1835	54	427	130%	-41	175%
32	5888	191	7723	105	1835	131%	-86	181%
64	24064	383	31659	204	7595	131%	-179	187%
128	97280	767	128171	399	30891	131%	-368	192%
256	391168	1535	515755	786	124587	131%	-749	195%
512	1568768	3071	2069163	1557	500395	131%	-1514	197%

2.1.15 BFM

BFM stands for *Bus Functional Model*, a *BFM* is a simplified simulation model that reflects the input/output level behavior of a device without modeling its internal computational abilities. They are commonly used in test benches to simulate design models and verify their functionality [11].

2.1.16 Virtual Interface

The *Virtual Interface* is a variable that represents an interface instance, it provides a mechanism for separating abstract models and test programs from the actual signals that make up the design [12].

The real interface is instantiated in the testbench and connected to the DUT, the virtual interface is declared in the *monitor* and *driver* and is referenced to the real interface to access the interface signals connected to the DUT.

2.1.17 UVM

UVM stands for *Universal Verification Methodology*, is the first verification methodology to be standardized. It is built on the Open Verification Methodology (OVM), combined with the Advanced Verification Methodology (AVM), Universal Reuse Methodology (URM), and implements concepts from the e Reuse Methodology (eRM) and Verification Methodology Manual (VMM) [13].

UVM PHASES

The *UVM PHASES* are the set of functions and tasks that all *UVM components* execute together, they are used to synchronize the *components* [13]. The phases are executed in the order shown in Figure 2.21 and the main phases are described in Table 2.2

Table 2.2: UVM Phases

Phase	Description
UVM_BUILD_PHASE	The UVM_BUILD_PHASE creates and configures the test-bench structure. Typically during this stage, the sub-components are instantiated, the register model is instantiated and the configuration values are get and set to sub-components. At the entry point of this phase, the top-level components have been instantiated and the simulation time is still zero. The phase exits when all the UVM components have been instantiated [13].
UVM_CONNECT_PHASE	The UVM_CONNECT_PHASE establishes the cross-component connections. Typically during this stage the TLM ports and exports are connected, the register model is connected to adapter components and the explicit phase domains are set up. At the entry point of this phase all the components have been instantiated and the simulation time is still zero. The phase exits when all the cross-components are connected and all the independent phase domains are set [13].The CONNECT PHASE establishes the cross-component connections. Typically during this stage the TLM ports and exports are connected, the register model is connected to adapter components and the explicit phase domains are set up. At the entry point of this phase all the components have been instantiated and the simulation time is still zero. The phase exits when all the cross-components are connected and all the independent phase domains are set [13].
UVM_RUN_PHASE	The UVM_RUN_PHASE stimulates the DUT, this phase is the only phase that is a task and consumes simulation time. At the entry point of this phase, the simulation time is still zero and there should not have been any active clock edges before entry to this phase. The phase exits when the DUT no longer needs to be simulated and can terminate if all run phase objections have been dropped or if the timeout expires before all objections are dropped [13].
UVM_CHECK_PHASE	The UVM_CHECK_PHASE checks for any unexpected condition in the verification environment. Typically during this phase, it is checked that no unaccounted-for data remains. At the entry point of this phase, all the data has been collected. The phase exits when the test is known to have passed or failed [13].

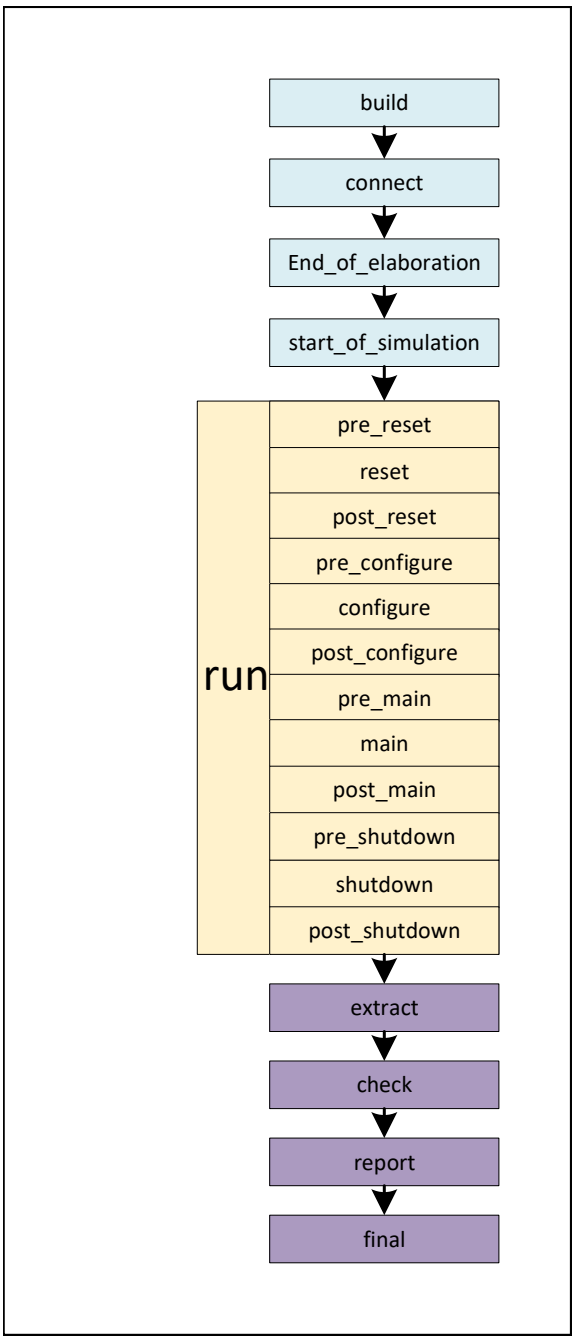


Figure 2.21: UVM Phases

UVM TESTBENCH

The *UVM TESTBENCH* typically instantiates the DUT module and the UVM Test class and configures the connections between them. The UVM Test is dynamically instantiated at run-time, this allows the *UVM TESTBENCH* to be compiled once and run with different tests [14].

UVM OBJECT

The *UVM OBJECT* class is the base class of all *UVM* components and transactions. Its primary function is to define a set of methods for such common operations as create, copy, compare, print and record [13].

UVM COMPONENT

The *UVM COMPONENT* class is the base class for *UVM* components [13].

Components are quasi-static objects that exist throughout the simulation. This allows us to establish a structural hierarchy like modules. Some examples of *UVM components* are *UVM test*, *UVM environment*, *UVM agent*, *UVM driver*, *UVM monitor* and *UVM scoreboard* [13]. The principal *UVM components* are described in Table 2.3

Table 2.3: UVM Components

Component	Description
UVM_TEST	The UVM_TEST is the top-level UVM component in the UVM Testbench. Typically the UVM test instantiates the top-level environment, configures the environment and applies stimulus to the DUT by invoking sequences [14].
UVM_ENVIRONMENT	The UVM_ENVIRONMENT is the UVM component that instantiates and groups together other interrelated verification components. Typically the environment instantiates the agent, scoreboard and other environments [14].
UVM_DRIVER	The UVM_DRIVER receives UVM sequence items from the UVM sequencer and applies it on the DUT interface, in other words converts transaction-level stimulus into pin-level stimulus [14].
UVM_MONITOR	The UVM_MONITOR samples the DUT interface and captures the information in transactions that are sent to the rest of components for further analysis, in other words, converts pin-level activity to transaction-level activity [14].
UVM_SEQUENCE_ITEM	The UVM_SEQUENCE_ITEM is a class object that includes whatever information is needed to model a unit of communication between two components. It includes variables, constraints, and other fields and methods necessary for generating and operating the transaction [13].
UVM_SEQUENCE	The UVM_SEQUENCE is an object that contains the behavior for generating stimulus, they are not part of the component hierarchy and can be transient or persistent. A sequence can be used to drive a single transaction, to drive stimulus for the duration of the simulation or anywhere in-between [14].
UVM_SEQUENCER	The UVM_SEQUENCER works as an arbiter to control the transaction flow from multiple stimulus sequences, in other words, controls the flow of sequence item transactions generated by one or more sequences [14].
UVM_SUBSCRIBER	The UVM_SUBSCRIBER is an UVM component that provides an analysis export for receiving transactions from a connected analysis export. The connection between the analysis exports subscribes the component to any transaction emitted by the connected analysis port [13].

Continued on next page

Table 2.3: UVM Components (Continued)

UVM_SCOREBOARD	The UVM_SCOREBOARD checks the behavior of the DUT. It usually receives transactions carrying inputs and outputs of the design through analysis ports, runs the input transactions through a reference model, also called predictor, to produce expected transactions and compares the expected transaction with the RTL transaction [14].
UVM_AGENT	The UVM_AGENT instantiates and groups together other verification components related to a specific DUT interface. Typically it instantiates the driver, monitor and sequencer. The agent is capable of operating in active mode, where it is capable of generating stimulus, and passive mode, where it only monitors the interface without driving stimulus [14].
UVM_ANALYSIS_PORT	The UVM_ANALYSIS_PORT is a specialized TLM port used to broadcast a value to all subscribers implementing a UVM analysis import [14].
UVM_ANALYSIS_EXPORT	The UVM_ANALYSIS_EXPORT is a specialized TLM port that exports a lower-level UVM analysis import to its parent, it allows the export to supply the implementation of write() to get the transactions broadcasted from another UVM analysis ports [14].
UVM_ANALYSIS_IMPORT	The UVM_ANALYSIS_IMPORT is a specialized TLM port that receives all transactions broadcasted by an UVM analysis port. It serves as the termination port of an analysis port/export/import connection [13].

UVM VERBOSITY

The *UVM VERBOSITY* defines standard verbosity levels for reports. The report is printed only if the *VERBOSITY* of the report is equal or above than the reporter *VERBOSITY* [13].

UVM_NONE has the lower *VERBOSITY*, meaning that all reports with *VERBOSITY* *UVM_NONE* are printed and can not be disabled using the *VERBOSITY* [13].

The *VERBOSITY* levels are defined from lower to higher in the following way: *UVM_NONE* $<$ *UVM_LOW* $<$ *UVM_MEDIUM* $<$ *UVM_HIGH* $<$ *UVM_FULL*.

The Table 2.4 shows the different UVM macros used to print messages during the simulation.

Table 2.4: UVM Verbosity

Verbosity	Description
UVM_INFO	The `uvm_info(ID, MSG, VERBOSITY) macro issues an info message if VERBOSITY is lower than the configured verbosity of the message [13].
UVM_WARNING	The `uvm_warning(ID, MSG) macro issues a warning message with VERBOSITY UVM_NONE. The message cannot be turned off using the reporter's verbosity but can be turned off by setting the action for the message [13].
UVM_ERROR	The `uvm_error(ID, MSG) macro issues an error message with VERBOSITY UVM_NONE. The message cannot be turned off using the reporter's verbosity but can be turned off by setting the action for the message [13].
UVM_FATAL	The `uvm_fatal(ID, MSG) macro issues a fatal message with VERBOSITY UVM_NONE. The message cannot be turned off using the reporter's verbosity but can be turned off by setting the action for the message [13].

2.2 Reduction Operations

Most of the vector operations operate the elements of the operands, the element N of *srcA* operates with element N of *srcB*, the width and number of elements vary depending on the *OSIZE* of the operation.

In a normal operation one element does not affect the operation of the elements next to it, as shown in Figure 2.22 and in the following snippet.

```
for (int idx = 0; idx < NUM_ELEMENTS; idx++) begin
    result[idx] = srca[idx] (op) srcb[idx];
end
```

In a reduction operation, all the active elements of *srcA* are used, but just the first element of *srcB* is used and the result is stored in the first element of the *result*, in this case, one element affects the following elements, the operation of the next element depends on the result of the operation of the previous element as shown in Figure 2.23 and the following snippet.

```
result[0] = srca[0] (op) srcb[0];
for (int idx = 1; idx < NUM_ELEMENTS; idx++) begin
    result[0] = srca[idx] (op) result[0];
end
```

In the hardware implementation all the blocks always produce a result, even if that block is not needed, the control unit is responsible for enabling the Vector Register File to only write the result of the blocks that have a valid result. The result is also produced in the last block instead of the first block, after the ALU computation, the result can be inverted if needed to accommodate the result in the correct position.

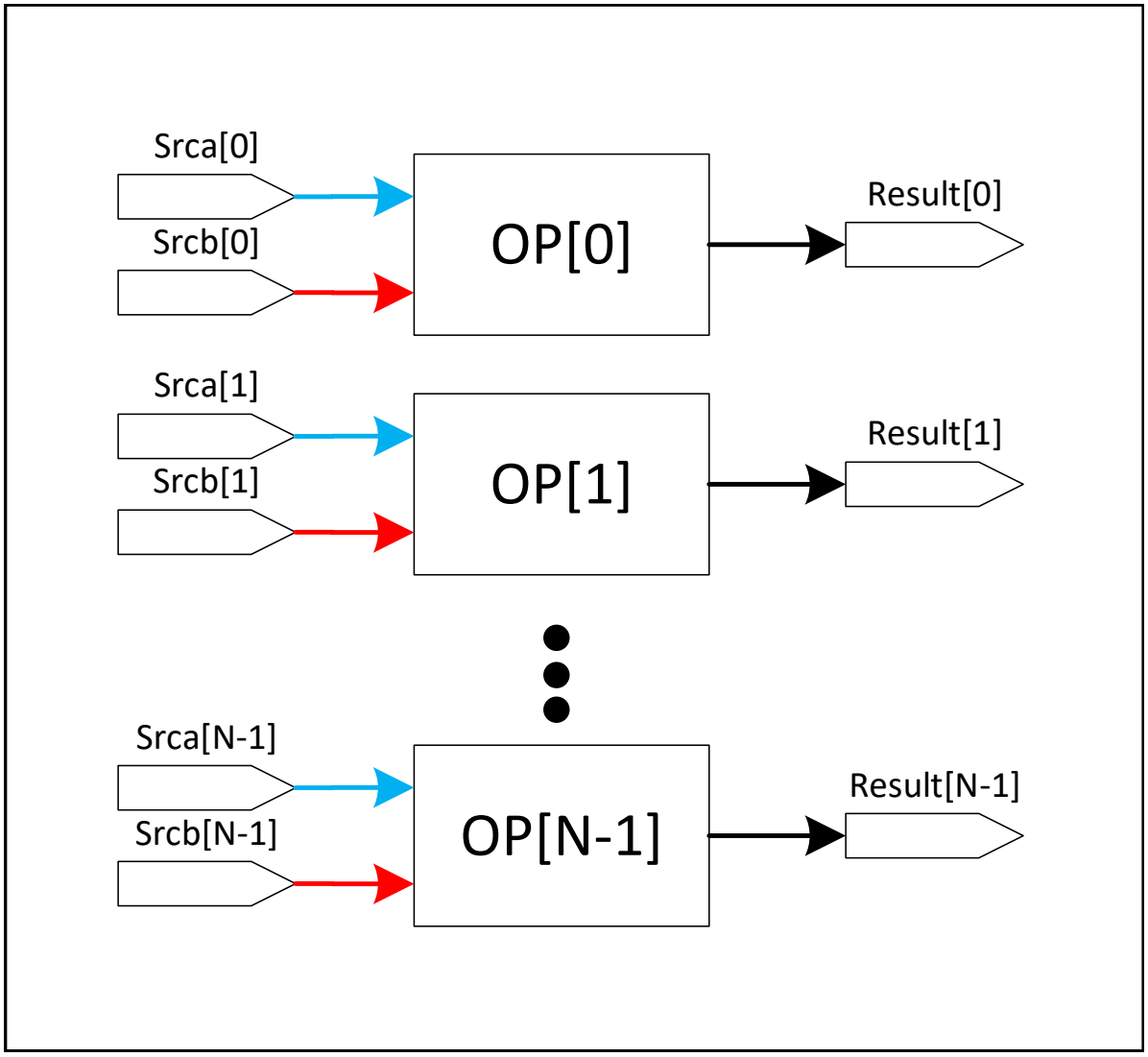


Figure 2.22: Vector Normal Operation

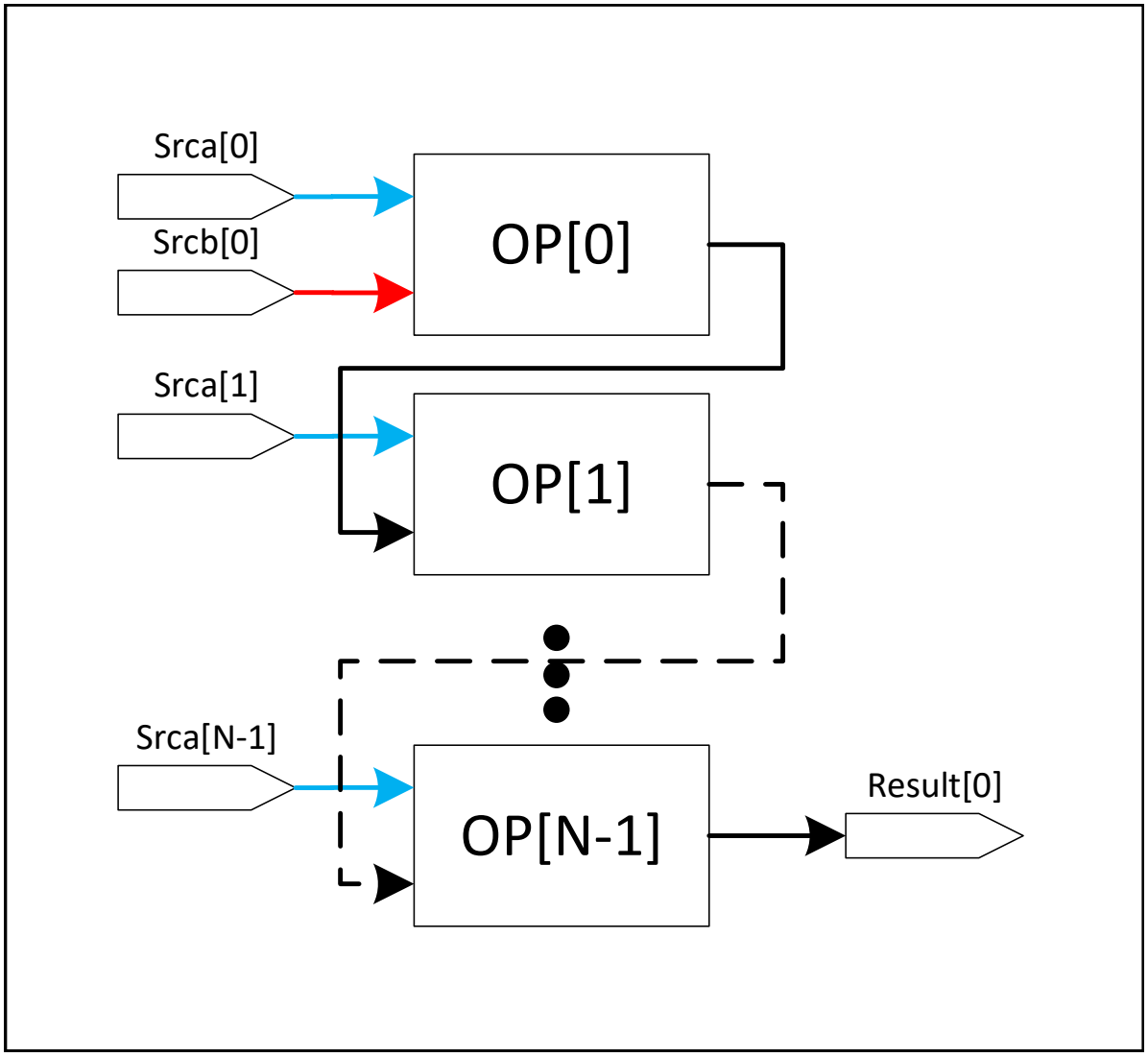


Figure 2.23: Vector Reduction Operation

2.3 Output Multiplexing

To get the result of an operation coming out from an ALU the source data is gated at the beginning of the data path and the result of all the ALUs is ored together to get the result, instead of the regular multiplexer used in most of the academic implementations.

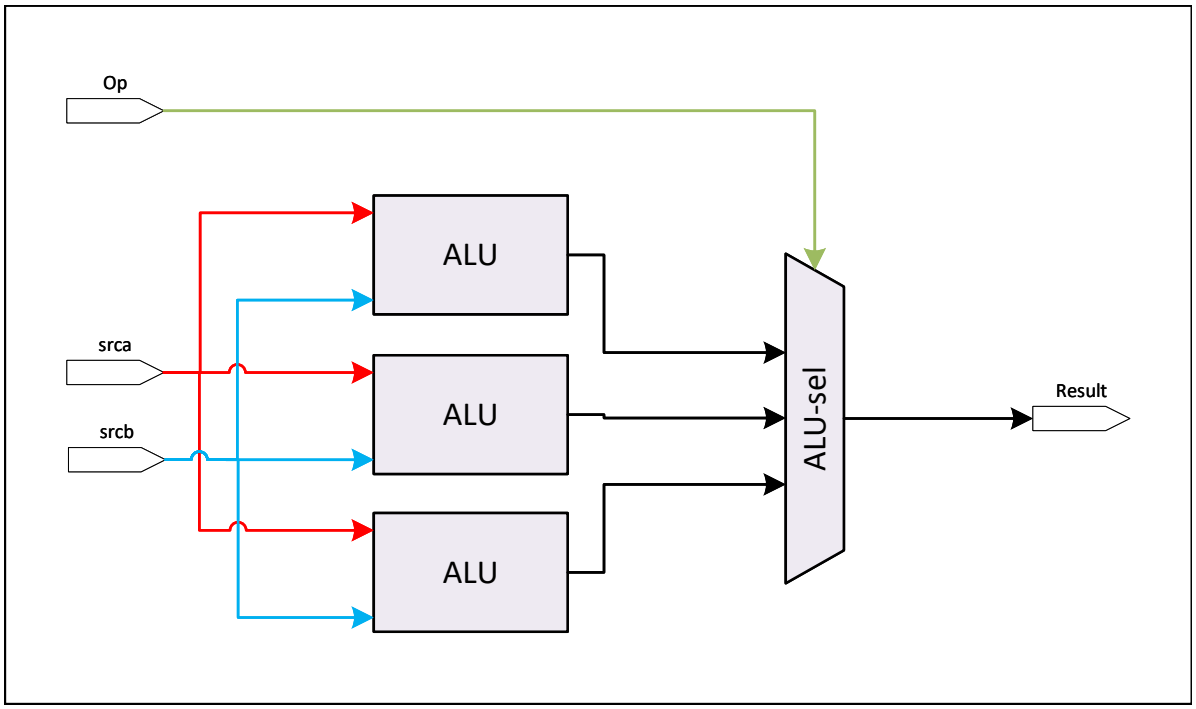


Figure 2.24: Output multiplexing with normal multiplexer

In Figure 2.24 is observed the multiplexing that is implemented in academic works, in this implementation all ALU operations are performed at the same time, and the output of the ALUs is connected to the multiplexer that is controlled by the ALU's opcode to select the result of the operation, this implementation is easy to implement and the HDL code is easy to read since it can be implemented using an if or a case statement, this implementation also assures that only one result is selected at the time preventing some bugs.

In Figure 2.25 is represented the implementation that is used in this project. In this implementation instead of using a multiplexer at the end of the data path, the sources are data gated with the ALU's opcode and all the results are ored at the end of the data path. One advantage of this implementation is the power saving, since the inputs are gated at the beginning of the data path the ALUs that are not active do not toggle and save power, this also allows the implementation of merge instructions by enabling more than one ALU and using the OR gate at the end to merge the results. The main disadvantage of this implementation is that it could introduce some bugs since more than one ALU could be active when it is not required, and the HDL code could be more complex since the source gating and the final OR gate could be implemented in different modules instead of a single procedural block like the academic implementation.

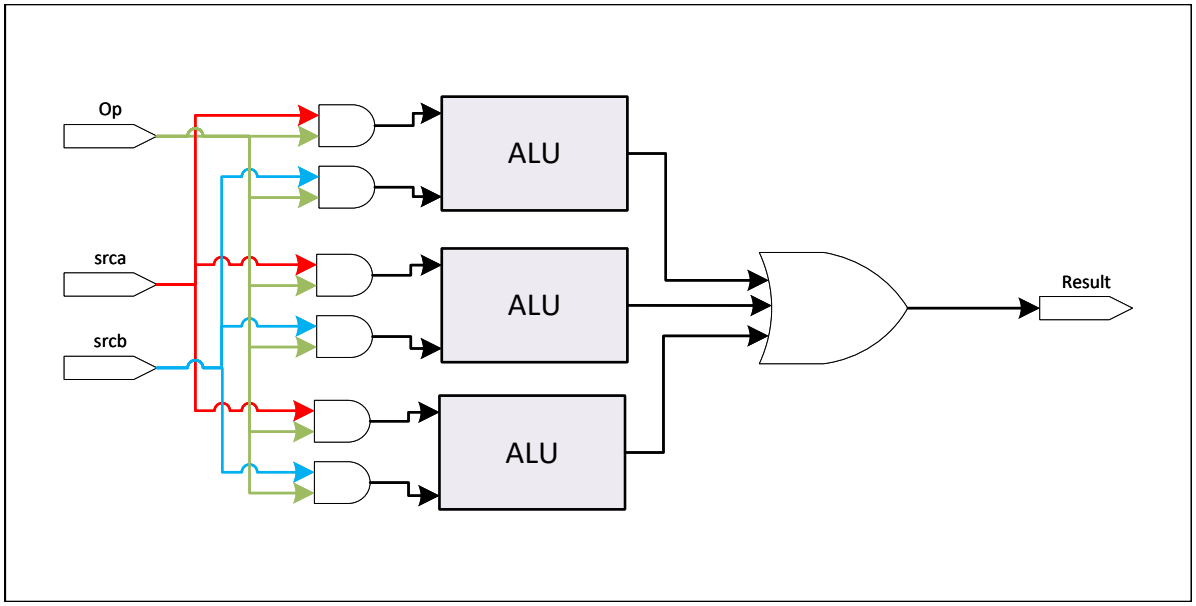


Figure 2.25: Output multiplexing with data gating

```
/////////Academic implementation////////////
always_comb begin
    case(Op)
        2'b00 : result = ALU_0;
        2'b01 : result = ALU_1;
        2'b10 : result = ALU_2;
        default : result = '0;
    endcase
end
```

```
///////Data gate implementation/////////////////
//Source gating
always_comb begin
    srca_0 = srca & {{$bits(srca)}(op == 2'b00)};
    srcb_0 = srcb & {{$bits(srcb)}(op == 2'b00)};
    srca_1 = srca & {{$bits(srca)}(op == 2'b01)};
    srcb_1 = srcb & {{$bits(srcb)}(op == 2'b01)};
    srca_2 = srca & {{$bits(srca)}(op == 2'b10)};
    srcb_2 = srcb & {{$bits(srcb)}(op == 2'b10)};
end

//Final Or
always_comb begin
    result = 0;
    result |= ALU_0;
    result |= ALU_1;
    result |= ALU_2;
end
```

In the above code snippets, an example of the two implementations is shown, in this example the opcode is coded using two bits to represent three different operations, but this opcode can also be encoded in a one-hot codification, using a one-hot opcode could increase

the number of flops to stage the opcodes but eliminates the comparators and improves the timing.

In the example above *srca* and *srcb* are gated, but depending on the operation to be implemented only one source could be gated, for example, if the ALU implements a bitwise AND operation only one source needs to be gated to ensure that the result is zero if the ALU is not active.

In the case of the Data gating implementation an assertion can be coded to make sure that only one output is valid at a time, this property can be easily proven by a Formal Verification tool removing the disadvantage mentioned above about the bugs that could be introduced if more than one output is active at time, the assertion can be observed in the following snippet.

```
1    check_alu_output: assert property ( @(posedge clk) disable iff (rst)
2        $onehot0({|ALU_0, |ALU_1, |ALU_2})
3    ) else $fatal("More than 1 ALU output is valid at the same time");
```

2.4 Swizzle

Some operations require a reorder of the bits or bytes of the sources, to make this adjustment of the data the *Swizzle block* is used. There are two variants of the *Swizzle block* the *Bit Swizzle* and the *Byte Swizzle*.

2.4.1 Bit Swizzle

The Bit Swizzle gets one byte of data and inverts the order of the bits, the least significant bit is the most significant bit and the most significant bit is the least significant bit as shown in Figure 2.26 and the following snippet.

```
1    logic[7:0] data;
2    logic[7:0] data_swizzle;
3    for (int idx=0; idx < 8; idx++) begin
4        data_swizzle[7-idx] = data[idx];
5    end
```

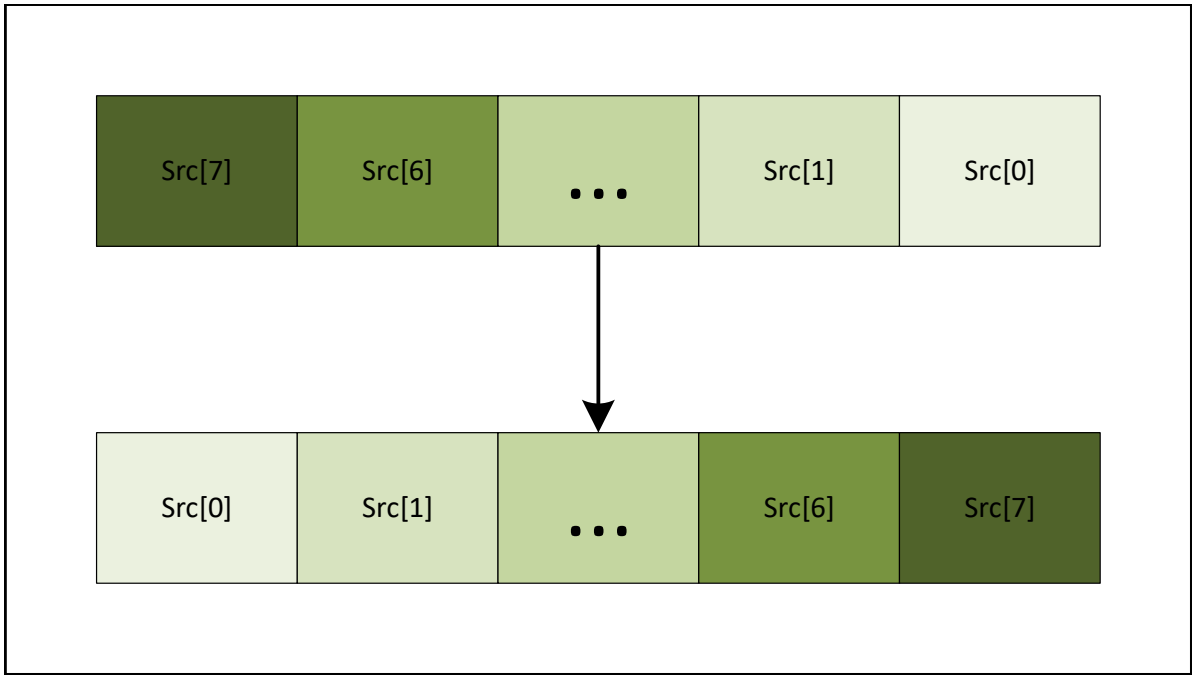


Figure 2.26: Bit Swizzle

2.4.2 Byte Swizzle

The Byte Swizzle gets an array of 16 bytes, 128 bits in total, and invertss the order of the bytes, the least significant byte is the most significant byte and the most significant byte is the least significant byte, the bits inside the bytes remain unaltered as shown in Figure 2.27 and the following snippet.

```
1     logic[15:0][7:0] data;
2     logic[15:0][7:0] data_swizzle;
3     for (int idx=0; idx < 16; idx++) begin
4         data_swizzle[15-idx][7:0] = data[idx][7:0];
5     end
```

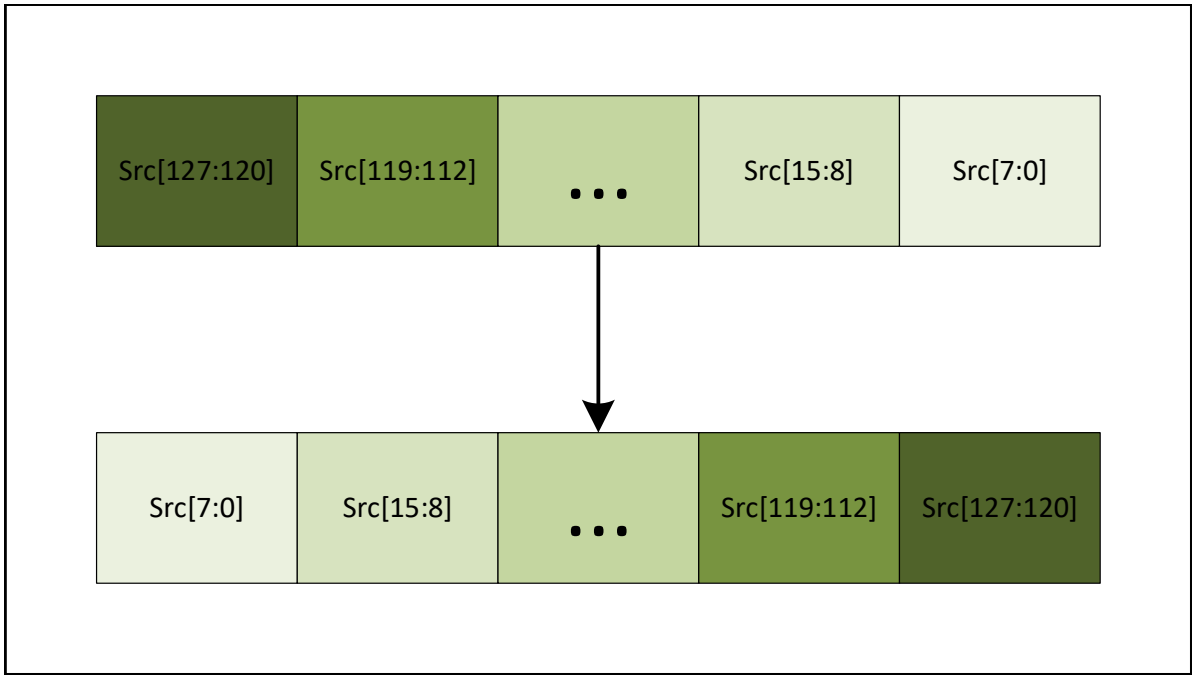


Figure 2.27: Byte Swizzle

2.5 Vector Instruction

Most of the vector instructions are encoded under the major opcode *OP-V = 7'b1010111*, some instructions like vector loads and stores are encoded under the opcodes *LOAD-FP* and *STORE-FP* but since stores and loads are not implemented in this project only the opcode *OP-V* is implemented leaving the opcodes *LOAD-FP* and *STORE-FP* out of scope.

As shown in Figure 2.28 the Vector Instruction is encoded using a 32-bit instruction divided into seven fields.

Opcode

The *Major Opcode* or *OP[6:0]* field is a 7-bit field located in the bits six to zero of the vector instruction. Indicates the type of operation to be executed. As indicated in Vector Instruction all the vector instructions implemented in this project are encoded under the major opcode *OP-V = 7'b1010111*, this opcode enables the vector control unit, if the opcode is different from the vector opcode the control unit is not enabled, all the vector control signals are set to zero and the *Vector Engine* does not execute the operation.

Vector Destination

The *Vector Destination* or *VD[11:7]* field is a 5-bit field located in the bits eleven to seven of the vector instruction. This field indicates the destination address of the vector operation, it can refer to the address of the Integer register file or the address of the *VRF* depending on the operation, most of the vector operations write to the *VRF*, but some operations like the operation *VMV* can write the result directly in the Integer register file

FUNCT3

The *FUNCT3[14:12]* field is a 3-bit field located in the bits fourteen to twelve of the vector instruction. This field is used to differentiate between operations that share the same *FUNCT6* encoding and between the variants of the operations to be executed, for example, if the source one is a Vector, Integer, Immediate or Floating source.

This field is divided into three groups, integer, mask and float. Each encoding of *FUNCT6* can be mapped to any of the three groups, or to the three groups, meaning that one encoding of *FUNCT6* can encode up to three different operations. Each group has different encodings that encode the different variants of the operation.

The Table 2.5 shows the common use for the different encodings of *FUNCT3* used by the operations implemented. The description shown in Table 2.5 is not necessarily true for all operations, some operations could have a different meaning for the same encoding, for more information about a specific instruction consult the RISCV-V spec.

Table 2.5: Funct3 Encodings

FUNCT3[2:0]	Common Use	Group
OPIVV(3'b000)	Vector-Vector Integer Operations	Integer
OPIVX(3'b100)	Vector-Integer Integer Operations	
OPIVI(3'b011)	Vector-Immediate Integer Operations	
OPMVV(3'b010)	Vector-Vector Reduction Operations	Mask
OPMVX(3'b110)	Vector-Integer Reduction and Mask Operations	
OPFVV(3'b001)	Vector-Vector Float Operations	Float
OPFVF(3'b101)	Vector-Float Float Operations	

Vector Sources

The *Vector source 1* and *Vector source 2* or *VS2[24:20]* and *VS1[19:15]* fields are 5-bits located in the bits nineteen to fifteen and twenty-four to twenty respectively of the vector instruction. This field indicates the register address of the operands to be used by the Vector operation.

VS2 always refers to an address in the *VRF*, while *VS1* can refer to an address in the *VRF*, an address in the Integer Register File or a immediate value depending on the operation and the variant encoded in *FUNCT3*

Vector Mask

The *Vector Mask* or *VM[25]* field is a 1-bit field located in the bit twenty-five of the vector instruction. In most operations, this field indicates if the operation is masked or unmasked, with a value of 1'b1 indicating that masking is disabled and 1'b0 that masking is enabled, not all operations support masking and this field could be ignored, other operations could require to this field to be always 1'b0 or 1'b1 or can give the field a different meaning.

FUNCT6

The *FUNCT6[31:26]* field is a 6-bit field located in the bits thirty-one to twenty-six of the vector instruction. This field is used to encode the different Vector operations along with *FUNCT3*. The combination of *FUNCT6* and *FUNCT3* encode the specific operation to be executed.

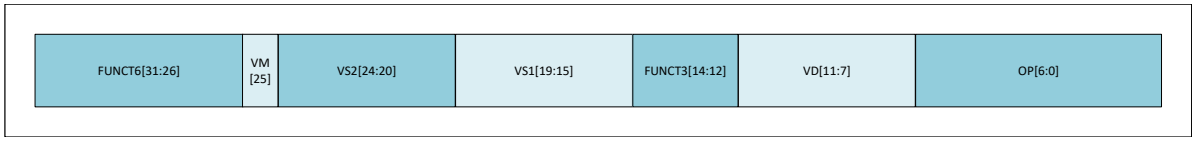


Figure 2.28: Vector Instruction Format

2.6 Pipeline Stages

The *Vector Engine* is built using a traditional pipeline of five stages, *Instruction Fetch*, *Instruction Decode*, *Execute*, Memory Access and *Write Back*. Each stage is executed in one clock cycle, the actions performed in each stage are shown in Table 2.6 and a flow diagram can be observed in Figure 2.29.

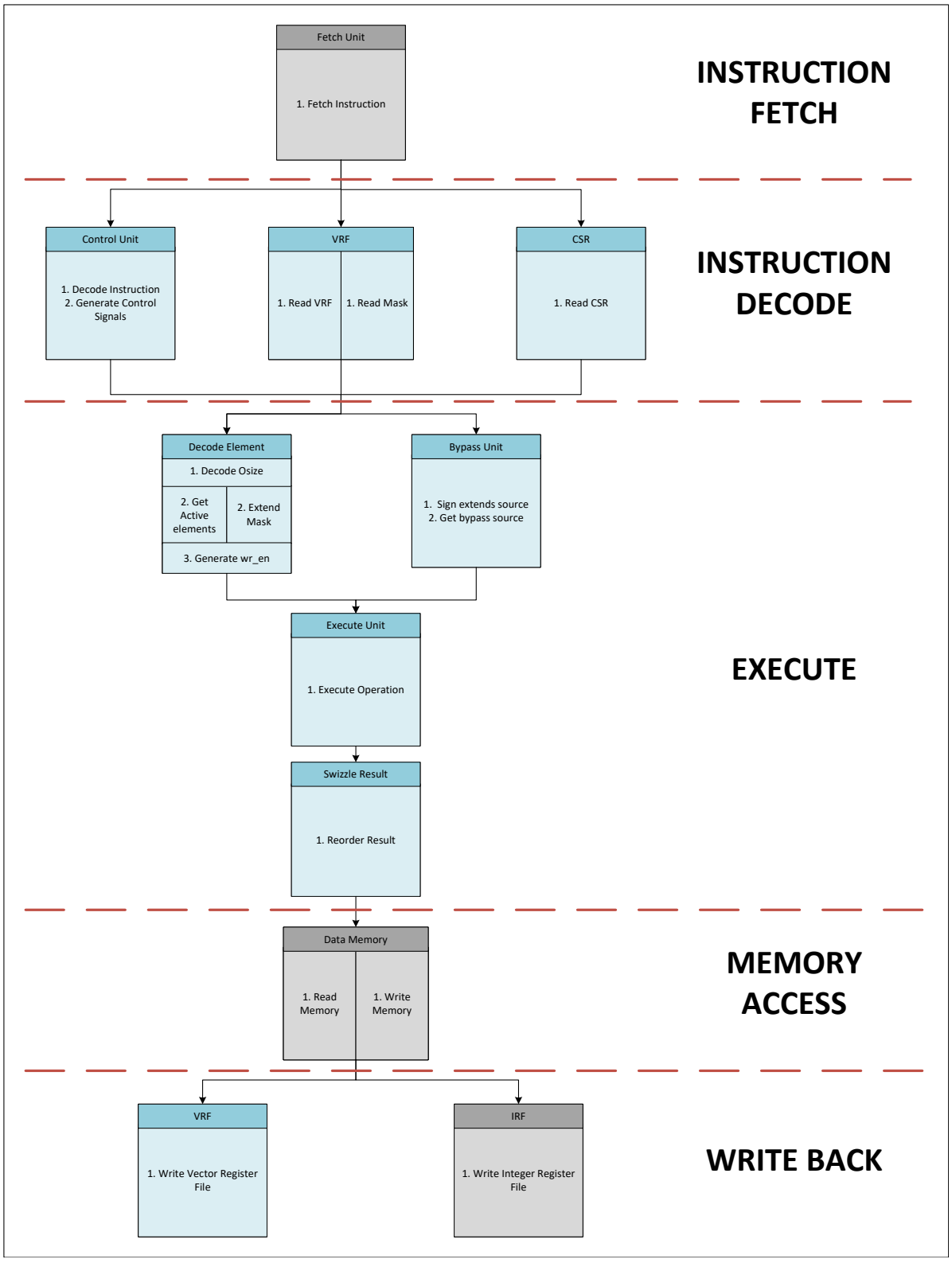


Figure 2.29: Pipeline stages flow diagram

Table 2.6: Pipeline Actions

STAGE	ACTIONS
Instruction Fetch	* Fetches Instruction from Instruction Memory
Instruction Decode	* Decodes Instruction * Generates Vector control signals * Reads Integer/Vector Register File * Reads Mask * Reads CSR
Execute	* Decodes OSIZE * Get active elements * Extends Mask * Generates wr_en * Gets sources from bypass unit * Signs extend source * Executes Instruction * Arrange result
Memory Access	* Writes data in data memory (Stores) * Reads data from data memory (Loads)
Write Back	* Writes result in Integer/Vector Register File

2.6.1 Instruction Fetch

The *Instruction Fetch* stage is not implemented inside the *Vector Engine*, it is implemented externally, the *Vector Engine* gets the fetched instruction at Instruction Decode.

2.6.2 Instruction Decode

The main purpose of the Instruction Decode stage is to decode the RISCV instruction and generate the appropriate vector control signals to perform the vector instruction. The fetched instruction is sent to the *Vector Engine* to be decoded, the *Vector Control unit* decodes the vector instruction and generates the vector control signals, only the vector instructions are decoded inside the *Vector Engine*, other types of instructions need to be decoded externally.

At this stage the *VRF* is also read, both sources are read from the *VRF* using the VS1 and VS2 fields from the vector instruction, the mask is also read from the *VRF* and staged to be used at Execute.

At the same time the *CSRs* are read and staged to be used at Execute. *CSRs* are always read so they can be read at the same time the instruction is being decoded.

2.6.3 Execute

The third stage of the pipeline is the *Execute stage* and its main purpose is to execute the vector instruction. First the *CSRs* are decoded in the *Decode Element* unit to get the *OSIZE* of the operation and get the active elements of the operation. At the same time in the same unit, the mask is extended depending on the *OSIZE* to mask the active elements and generate the *wr_en* that are used to only modify the active and masked elements of the operation.

At the same time the correct source is selected in the *Bypass Unit*, the source can come from the *VRF*, the *Integer Register File*, the result at *Memory Access* or the result at *Write Back*. In this unit, the scalar operands are also sign extended depending on the *OSIZE* of the operation.

Once the *CSRs* are decoded and the correct sources are obtained, the operation is executed in the *Execution Unit*, the operations can be executed on different *ALUs* depending on the type of the operation. When the operation is complete the result is sent to the *Result Swizzle* unit to reorder the result if it is necessary, this is done for Reduction Operations.

2.6.4 Memory Access

During the *Memory Access* stage the accesses to the data memory are performed, the memory is written for stores and it is read for loads. Since the data memory is not located inside the *Vector Engine* and Loads/Stores are not implemented in this project the result of the operation is just staged to the next stage. This stage is not removed to maintain the simplicity of the conventional 5-pipe stage CPU and be able to integrate the *Vector Engine* with a 5-pipe stage RISCV CPU.

2.6.5 Write Back

The final stage is the *Write Back*. The only action performed during this stage is the write of the result in the *VRF* or in the *Integer Register File* depending on the operation.

2.7 Reduction Source

To reduce the hardware needed in the implementation of the *Vector Engine* the hardware used for normal operations is reused to implement the *Reduction Operations*. To achieve this, most of the block operations are 8-bit blocks, for example, 8-Bit adders, 8-Bit shifters, 8-Bit bitwise operators, etc. These blocks can be used to implement normal or *Reduction Operations* by selecting the correct input depending on the type of operation as shown in Figure 2.30.

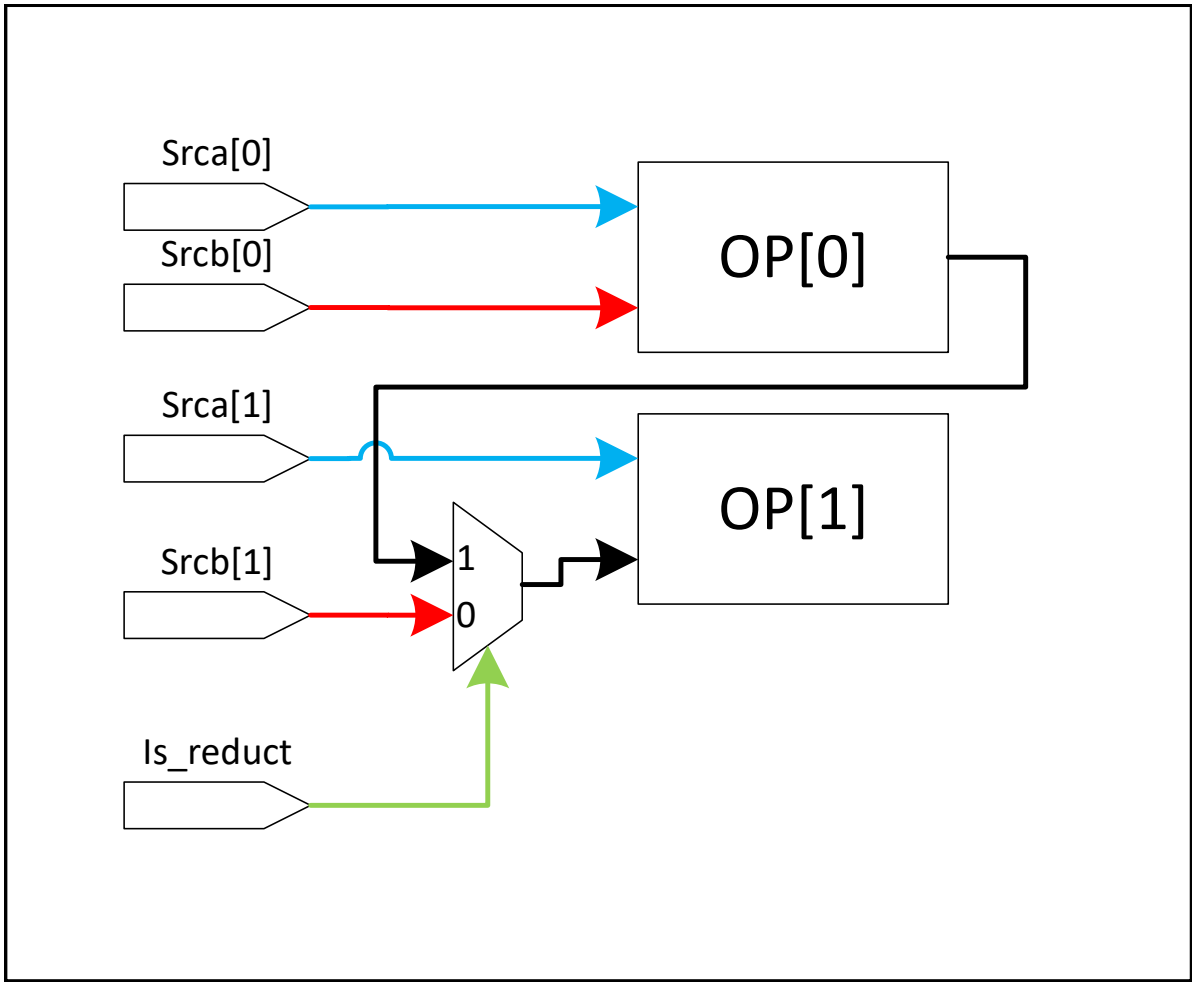


Figure 2.30: Reduction Source Multiplexor

A multiplexor can be added to the input of the operation block to select between the *srcB* to implement a normal operation or the result from the previous block to implement a *Reduction Operation*. Figure 2.32 only shows the connection for the second operation block, but depending on the index of the block could have multiple entries, for example, the block[1]

only requires one extra entry, because it can only receive data from the previous block if the *OSIZE* is *OSIZE_8*, but the block[2] can get data from block[1] if it is *OSIZE_8* or from block[0] if it is *OSIZE_16*. The number of extra entries per block can be calculated with (2.44) and the input associated with each entry with (2.45). The different extra entries per block are shown in Figure 2.31 and in Table 2.7.

$$num_extra_entries[block_idx] = floor(log2(block_idx + 1)) \tag{2.44}$$

$$extra_entry[block_idx][entry_idx] = result[block_idx - (2^{entry_idx})] \tag{2.45}$$

```
always_comb begin
    if (is_reduct) begin
        case(OSIZE)
            OSIZE_8:  entry_active[block_idx] = (num_extra_entries >= 1) ? extra_entry[block_idx][0] : srcb[block_idx];
            OSIZE_16: entry_active[block_idx] = (num_extra_entries >= 2) ? extra_entry[block_idx][1] : srcb[block_idx];
            OSIZE_32: entry_active[block_idx] = (num_extra_entries >= 3) ? extra_entry[block_idx][2] : srcb[block_idx];
            OSIZE_64: entry_active[block_idx] = (num_extra_entries >= 4) ? extra_entry[block_idx][3] : srcb[block_idx];
        endcase
    end else begin
        entry_active[block_idx] = srcb[block_idx];
    end
end
```

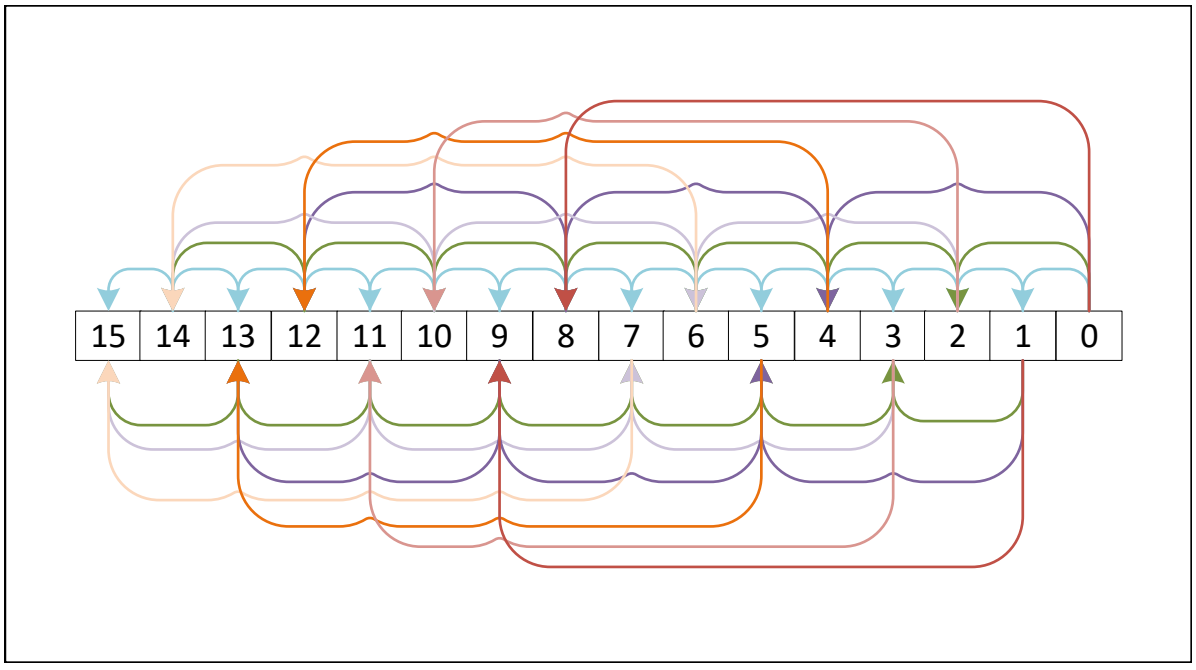


Figure 2.31: Reduction Possible sources

Table 2.7: Reduction sources

	Extra[0] OSIZE_8	**Extra[1] OSIZE_16**	**Extra[2] OSIZE_32**	**Extra[3] OSIZE_64**
BLOCK[0]	SRCB[0]	SRCB[0]	SRCB[0]	SRCB[0]
BLOCK[1]	RESULT[0]	SRCB[1]	SRCB[1]	SRCB[1]
BLOCK[2]	RESULT[1]	RESULT[0]	SRCB[2]	SRCB[2]
BLOCK[3]	RESULT[2]	RESULT[1]	SRCB[3]	SRCB[3]
BLOCK[4]	RESULT[3]	RESULT[2]	RESULT[0]	SRCB[4]
BLOCK[5]	RESULT[4]	RESULT[3]	RESULT[1]	SRCB[5]
BLOCK[6]	RESULT[5]	RESULT[4]	RESULT[2]	SRCB[6]
BLOCK[7]	RESULT[6]	REUSLT[5]	RESULT[3]	SRCB[7]
BLOCK[8]	RESULT[7]	RESULT[6]	RESULT[4]	RESULT[0]
BLOCK[9]	RESULT[8]	RESULT[7]	RESULT[5]	RESULT[1]
BLOCK[10]	RESULT[9]	RESULT[8]	RESULT[6]	RESULT[2]
BLOCK[11]	RESULT[10]	RESULT[9]	RESULT[7]	RESULT[3]
BLOCK[12]	RESULT[11]	RESULT[10]	RESULT[8]	RESULT[4]
BLOCK[13]	RESULT[12]	RESULT[11]	RESULT[9]	RESULT[5]
BLOCK[14]	RESULT[13]	RESULT[12]	RESULT[10]	RESULT[6]
BLOCK[15]	RESULT[14]	RESULT[13]	RESULT[11]	RESULT[7]

Implementing the different multiplexors using (2.44) and (2.45) can be a little confusing, for that reason the module *Reduction Source* is implemented.

As shown in Figure 2.32 this module is implemented using four constant shifters. In the upper part of the diagram the *input_src* is gated with *is_reduct* and *osize* control signals. If the operation is not a *Reduction Operation* or if it is *OSIZE_128* the *input_src* is passed to the *output_src*, also depending on the *OSIZE* some lower bits of the *input_src* are passed to the *output_src*.

In the lower part of the diagram, the *Result* of the operation blocks is gated with *is_reduct* and is shifted eight, sixteen, thirty-two and sixty-four positions, with a multiplexor one shifter is selected depending on the *OSIZE)*. Finally, the result shifted and the *input_src* are ored to get the *output_src* that feeds the operation blocks. In Figure 2.33 are shown the vectors that are formed and that can be selected as *output_src*.

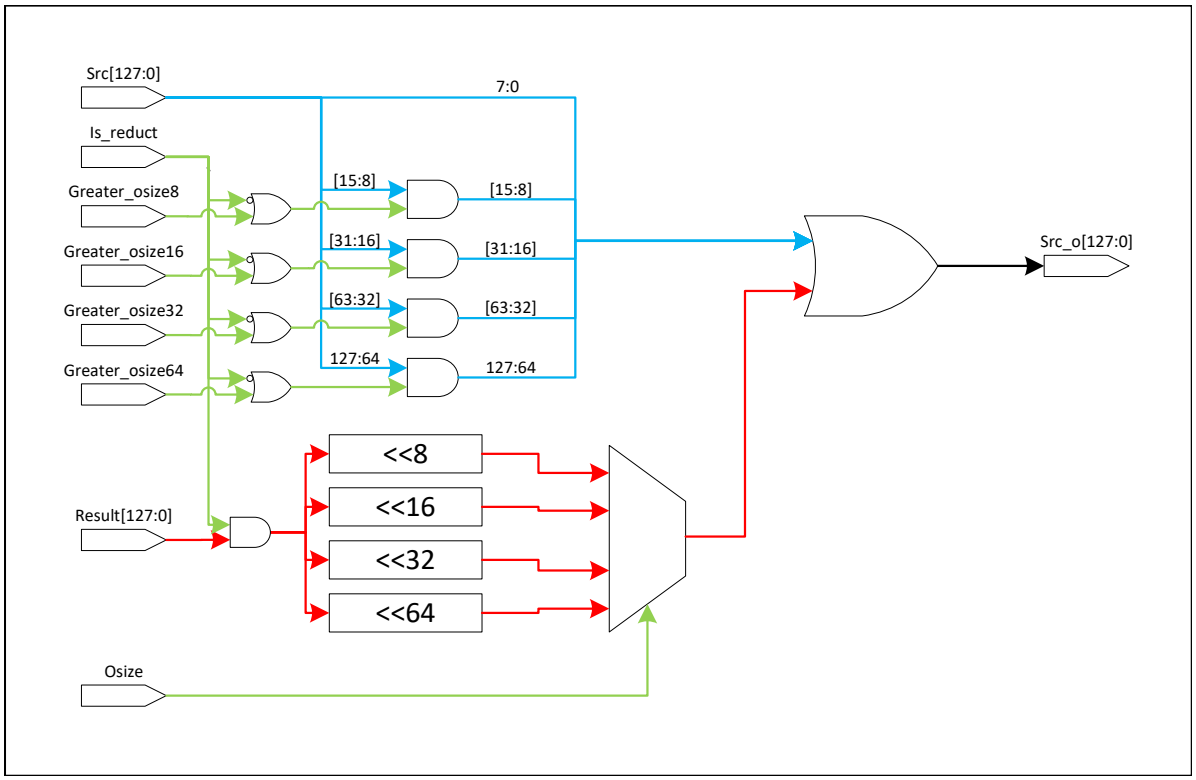


Figure 2.32: Reduction Source

The *Reduction Source* module implementation is logically equivalent to the implementation with the multiplexors and the equations, but it is easier to understand and implement in HDL. Also if the synthesizer is well configured it optimizes the design to remove the bits that are shifted out and that are not used and generates a similar hardware as the multiplexor implementation.

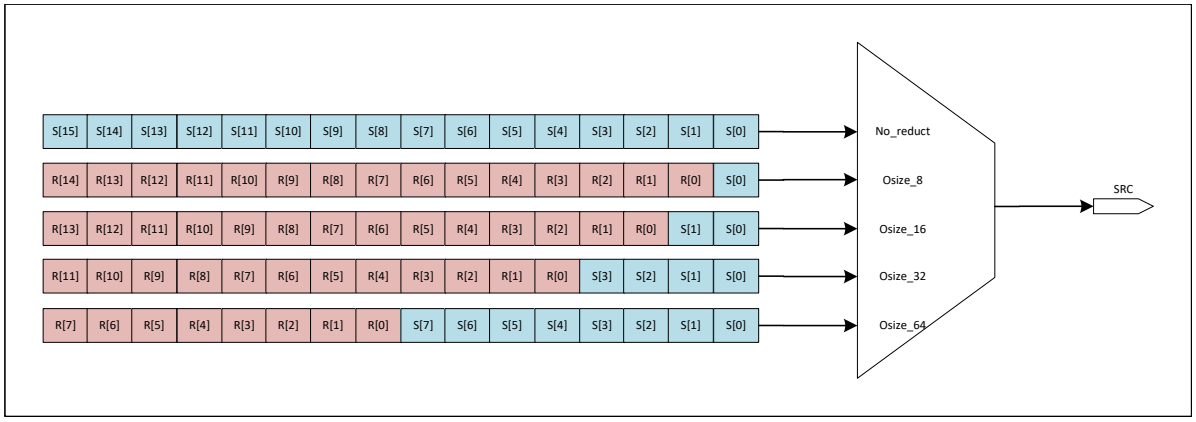


Figure 2.33: Reduction Source Mux selection

2.8 Control Status Registers

The *Control Status Registers* or *CSR* hold the configuration of the *Vector Engine* like the width of the elements and the number of elements to be used by the operation and shows the status of the *Vector Engine* like the vector context status used by the hypervisor. The *CSRs* implemented are shown in Table 2.8. The Table 2.8 shows the valid values for the implementation described in this document, those values do not represent all the valid values described in the RISCV-V documentation.

The *CSRs* can be written externally by the *RISCV processor* or internally by the *Vector Engine*. For that reason, two interfaces are implemented to modify the *CSRs*. The *Vector Engine* can modify the *CSRs* using a vector instruction as *vset* or *vl* or can be modified by some internal processes like interrupts or faults. In this project neither the CSR instructions, interrupts nor faults are implemented, for that reason the only way to modify the *CSRs* is by using the external interface.

Not all the valid Values of the *CSRs* are implemented in this project, in the following sections there are some considerations about the *CSRs*.

Table 2.8: Implemented CSR

Register	Field	Description	Valid Values
VSSTATUS	Vs[10:9]	Vector Status Field (Only valid with hypervisor)	N/A
	reserved[8:0]	Reserved	N/A
VTYPE	Vill[8]	Used to encode that a previous vset{i}vl{i} instruction attempted to write an illegal value	N/A
	vma[7]	Vector mask agnostic	N/A
	vta[6]	Vector tail agnostic	N/A
	vsew[5:3]	Sets the element width	3'b000 = OSIZE_8 3'b001 = OSIZE_16 3'b010 = OSIZE_32 3'b011 = OSIZE_64 3'b100 = OSIZE_128
	vlmul[2:0]	Represents the number of vector registers that are combined to form a vector register group	3'b000 = LMUL_1
VL	len[4:0]	Specifies the number of elements to be updated with the result from a vector instruction.	[0-16]
VLENB	lenb[3:0]	Holds VLEN/8 or the vector register length in bytes	16
VSTART	index[4:0]	Specifies the index of the first element to be executed by a vector instruction	[0-16]
VXRM	rounding_mode[1:0]	Indicates the rounding mode for fixed point operations	N/A
VXSAT	saturate[0]	Indicates if a fixed point operation saturated the result	N/A
VCSR	saturate[2]	Accesses VXSAT.saturate	N/A
	rounding_mode[1:0]	Accesses VXRM.rounding_mode	N/A

2.8.1 VSSTAUS

The VSSTATUS CSR is only implemented if the *hypervisor* extension is also implemented, since *hypervisor* extension is not implemented this *CSR* is not used by the *Vector Engine*.

2.8.2 VTYPE

The *VTYPE* provides the information to interpret the contents of the *VRF*. *VTYPE* can only be modified by the instructions *vset* and *vl* but for testing purposes this *CSR* can also be modified using the external interface.

Vill

Vector Type Illegal or Vill is used to indicate that a previous *vset* or *vl* tried to write an unsupported value. This field is not modified and is not supported by the *Vector Engine*.

Vma

Vector mask Agnostic or *vma* Indicates the behavior of the inactive elements of a vector instruction, if it is set to zero the inactive elements are **undisturbed** holding its previous value, if it is set to one the inactive elements can hold its value or be overwritten with all ones. The *Vector Engine* does not read this *CSR* and always execute the instruction using an **undisturbed** policy,

Vta

Vector tail Agnostic or *Vta* Indicates the behavior of the tail elements of a vector instruction, if it is set to zero the tail elements are **undisturbed** holding its previous value, if it is set to one the tail elements can hold its value or be overwritten with all ones. The *Vector Engine* does not read this *CSR* and always execute the instruction using an **undisturbed** policy,

Vsew

Vector selected element width or *Vsew* indicates the width of the vector elements, depending on *Vector selected element width* each *Vector Register* holds a different number of elements. The Table 2.9 shows the supported values for *Vsew* and the number of elements in each register considering that *VLEN* is 128-bits.

Although *OSIZE_128* is not formally supported by *RISCV-V Specification* this projects supports that *OSIZE* by encoding *Vsew* with a value of **3'b100**.

Table 2.9: Vsew supported Values

VSEW[2:0]	Element Width	Number of Elements
3'b000	8	16
3'b001	16	8
3'b010	32	4
3'b011	64	2
3'b100	128	1

Vlmul

Vector Register Grouping or *Vlmul* indicates how many vector registers are grouped to form a single Vector operand. This implementation only supports a *Vlmul* with a value of **3'b00** meaning that only one vector register is used per operand and does not support grouping of vector registers.

2.8.3 VL

Vector Length or *VL* indicates how many elements are updated with the execution of the vector instruction. This implementation supports any value of *VL*, if *VL* is zero any element is updated, all the elements hold its previous value, if *VL* is greater than the number of elements in the vector operand all the elements are updated.

According to the *RISCV-V Specification* this *CSR* can only be updated by the instructions *vset* or *vl* or by the *fault-only-first* load instructions, but for testing purposes in this implementation this *CSR* can also be updated using the external *CSR* interface.

It is important note that the indices of *VL* starts in zero and not in *VSTART*, for example if *VL* is one and *VSTART* is one it does not update any element because *VSTART* indicates that the first element is the element with the index one while *VL* is indicating that it operates only on the first element, the element zero.

2.8.4 VLENB

Vector Byte Length or *VLENB* is a read only register that holds *VLEN/8*, in other words, the vector register length in bytes, this is a constant and can't be modified neither by the *Vector Engine* nor the external *CSR* interface.

2.8.5 VSTART

Vector Start Index or *VSTART* indicates the index of the first element to be executed by a vector instruction. Elements with an index lower than *VSTART* remain *undisturbed*. This implementation supports all the values of *VSTART*, if *VSTART* is greater than *VL* or greater than the number of elements in the vector operand all elements remain *undisturbed*.

Not all operations support a different value than 0 in *VSTART*, for example, *Reduction operations* only supports a value of zero.

2.8.6 VXRM

Vector Fixed-Point Rounding Mode or *VXRM* defines the rounding mode used for fixed point instructions. Since Fixed point instructions are not implemented this *CSR* is not used by the *Vector Engine*.

2.8.7 VXSAT

Vector Fixed-Point Saturate Flag or *VXSAT* is a flag that indicates if a fixed point instruction had to saturate a result. Since Fixed point instructions are not implemented this *CSR* is not used by the *Vector Engine*.

2.8.8 VCSR

Vector Control and Status Register or *VCSR* is a different way to access *VXRM* and *VXSAT*, it does not implement a new register and can not be written by the *Vector Engine* nor the external *CSR* interface, it just provides access to read *VCSR* and *VXRM*.

2.9 Active and Inactive Elements

During the execution of a vector instruction, the elements are divided into two categories, *Active* and *Inactive* elements. The *Active* elements are the elements that update the *VRF* with the result of the vector operation. The *Inactive* elements can hold its previous value or be filled with all ones depending on *Vma* and *Vta*, this implementation does not support those *CSRs* and always applies an *undisturbed policy* meaning that *Inactive elements* do not update the register file.

The elements can also be divided into different subcategories:

Prestart

The Prestart elements are elements whose index is less than *VSTART*, these are *Inactive elements* and always apply an *undisturbed* policy and are not affected by *Vma* and *Vta*. If *VSTART* is equal or greater than *VL* or the number of elements in the vector operand all elements are Prestart elements and remain *undisturbed*. The Prestart elements can be identified using:

$$is_prestart[idx] = idx < VSTART \tag{2.46}$$

Body

The Body elements are elements whose index is greater or equal than *VSTART* and is less than *VL* and the number of elements in the vector operand. These elements can be *Active* or *Inactive* elements depending on the masking, if the operation is unmasked all the Body elements are *Active elements*, if the operation is a masked operation only the Body elements which *mask* is *1'b1* are *Active elements* and the other ones are *Inactive*, the *Body Inactive* elements are affected by *Vma*. The body, mask active, body active and inactive elements can be identified using:

$$is_body[idx] = VSTART <= idx < VL \tag{2.47}$$
$$is_mask_active[idx] =!use_mask \vee mask[idx] \tag{2.48}$$
$$is_body_active[idx] = is_body[idx] \wedge is_mask_active[idx] \tag{2.49}$$
$$is_body_inactive[idx] = is_body[idx] \wedge !is_mask_active[idx] \tag{2.50}$$

Tail

The Tail elements are elements whose index is greater or equal than *VL* and are *Inactive elements*, these elements are affected by *Vta*. If *VL* is zero or less or equal than *VSTART* any element is updated, *undisturbed policy* is applied. The Tail elements are identified using:

$$is_tail[idx] = idx >= VL \tag{2.51}$$

In Figure 2.34 are shown some examples of prestart. body and tail elements with different configurations.

The following code is an example of how a vector instruction is executed updating only body active elements.

```
1    //Read Sources from register File
2    srca = vrf[srca_addr];
3    srcb = vrf[srcb_addr];
4    //Read destination from the register file to apply undisturbed policy
     to inactive elements
5    dest = vrf[dest_addr];
6
7    //Execute operation updating only active elements
8    for (int idx = vstart; idx < vlen; idx++) begin
9        if (mask_op()) begin
10           if (mask[idx]) begin
11               dest[idx] = srca[idx] (op) srcb[idx];
12           end
13       end else begin
14           dest[idx] = srca[idx] (op) srcb[idx];
15       end
16   end
17
18   //Update VRF
19   vrf[dest_addr] = dest;
```

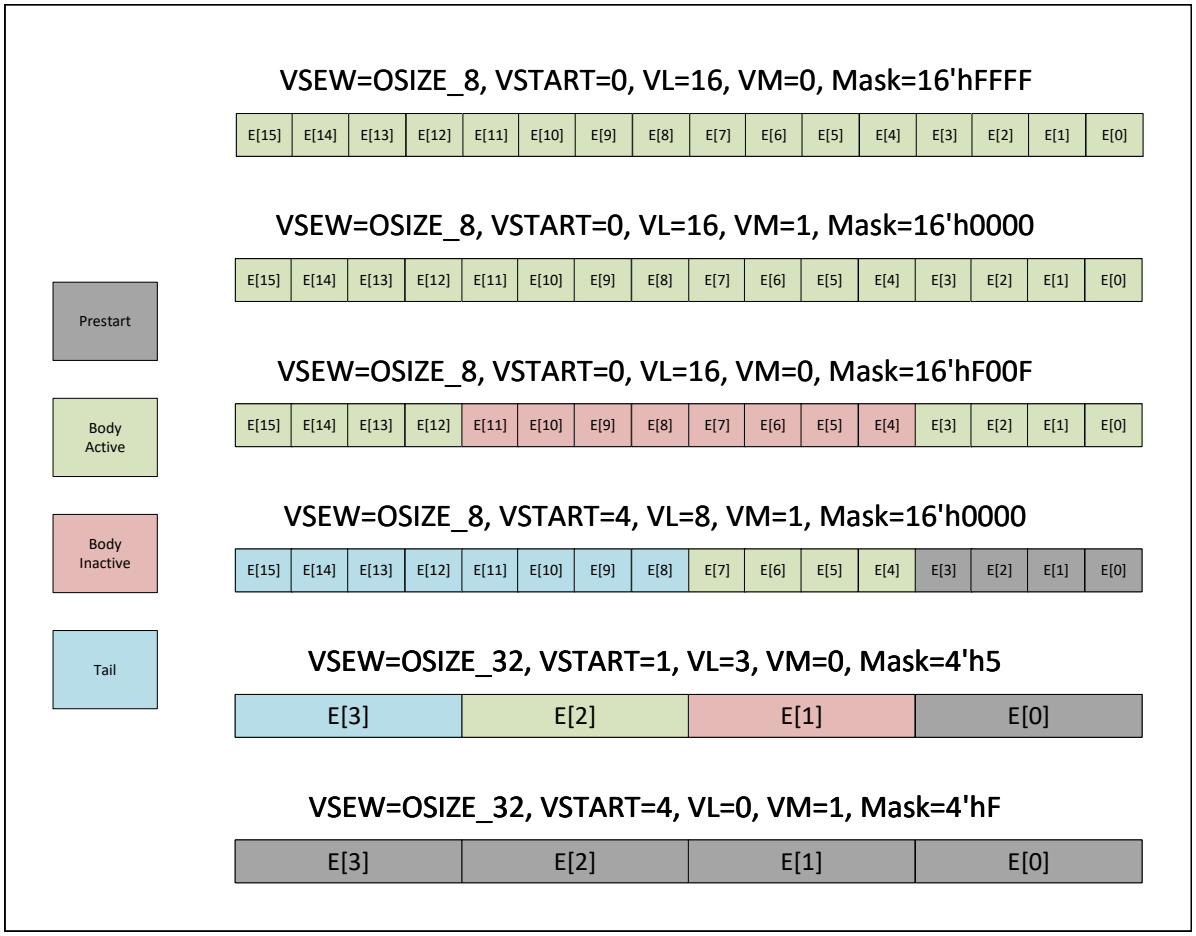


Figure 2.34: Prestart, Active, Inactive and Tail Elements

2.10 Masking

Most of the vector operations support *Masking*. As mentioned in the *Active/Inactive Elements* section the masking allows the execution of operations only on certain elements while the inactive elements remain undisturbed or are filled with all ones depending on *Vma*.

The vector masks are stored in the *VRF* along with the other data. There is no restriction about the location of the masks inside the *VRF*, any register can store a mask and a mask instruction can have as a destination any register, but in an operation that supports masking the mask is read from register[0], if the mask is located in another register it needs to be moved to register[0] prior the execution of the instruction that uses the mask.

The masking is enabled by writing 1'b0 to the field *VM* of the Vector instruction, not all instructions support masking and this field could have a different meaning in some instructions. The number of bits used as the mask depend on the *OSIZE* of the operation, one bit is used per element in the operand and the lower bits of the mask register are used, meaning that if the operation is *OSIZE_8* sixteen elements are present in the operation and the bits fifteen to zero of the mask register are used as a mask, if the operation is OSIZE_32 only the bits three to zero of the mask register are be used. The mask bit *i* corresponds to element *i* of the operation.

2.11 RISCV Vector Engine

The *RISCV Vector Engine* is an extension to the RV32I architecture that adds support for Vector instructions as described in the *RISC-V "V" Vector Extension Spec* [1]. It is important to mention that not all features added by the RISCV-V extension are supported by this project and not all vector instructions are implemented, the implemented instructions with its description can be located in appendix *Operations implemented*.

The top module of the *Vector Engine* is shown in Figure 2.35, the light blue rectangle represents the *Vector Engine* while the blocks outside the rectangle and colored in gray are blocks that are not implemented by the *Vector Engine* and should be implemented externally.

The light red dotted lines represent the pipe stages, meaning that all signals and arrows that go through those lines are staged by a pipe register, except for the lines that go from write back to the *VRF*, those signals are already staged and are connected directly.

The diagram can be read from left to right. First the instruction is fetched externally and it is sent to the *Vector Control unit* and to the *VRF Control Unit* to decode the instruction and read the operands from the *VRF*. At the same time the *CSRs* are read. The *CSR Control Unit* is also connected to an external CSR block, this is done to be able to modify the *CSRs* externally in the testbench or by the RISCV-CPU since the instructions and features that modify the *CSRs* internally are not implemented. The *VRF* also has a connection to an external port, this is done to be able to debug the internal registers in post silicon and to avoid the synthesizer to optimize the design and remove almost all the RTL.

In the next stage, the Execute stage, the sources and some control signals are sent to the *Bypass Unit* to select the correct source to be sent to the *Execution Unit* depending on the dependencies and the type of the operation. At the same time the control signals and *CSRs* are sent to the *Decode Element* block to generate the appropriate write enables depending on the

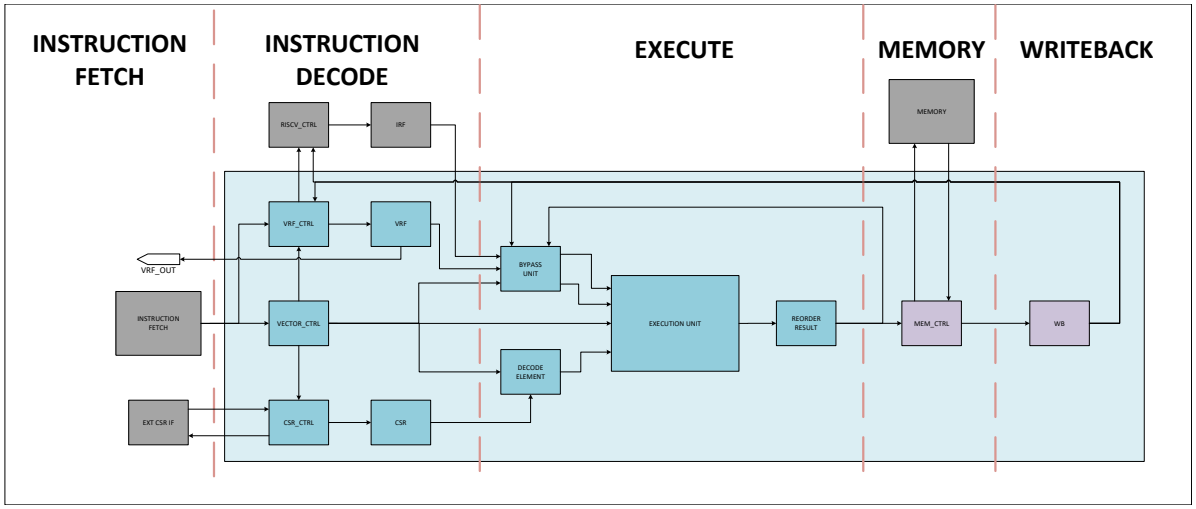


Figure 2.35: Vector Engine

operation and the *Active/Inactive* elements of the sources. Once the correct source is selected from the *Bypass Unit* the sources are sent to the *Execution Unit* along with some control signals to execute the corresponding operation. Finally, the result of the *Execution Unit* is connected to the *Result Swizzle* block to reorder the elements of the result if is necessary.

The *Memory stage* is not implemented inside the *Vector Engine* and Load/Store operations are also not implemented, for that reason the logic in the *Memory stage* is just a bypass of the data by staging the data to the next pipe stage.

Finally, the logic in the *Write Back stage* is also just a bypass to send the staged data to the *VRF* or to the *IRF* to write the result of the operation.

2.12 Vector Register File

The *Vector Register File* or *VRF* is an entity separated from the *Integer Register File*. It is composed of thirty-two 128-bit registers that hold the data used by the *Vector Engine* in the operations.

The structure of a vector register can be observed in Figure 2.36, it is formed by sixteen 8-bit registers, each register with its own *wr_en*. The *wr_en* of each register is formed with an AND gate, the gate ANDs the *addr_en* that comes from the address decoder and one bit from *wr_en* that is a 16-bit bus that indicates which byte of the register file should be modified as shown in (2.52). With the implementation of independent *wr_en* per *byte* the write of the register file can be controlled to avoid writing the masked or inactive elements in the operation.

$$reg_en[idx] = addr_en.wr_en[idx] \tag{2.52}$$

The complete *VRF* is observed in Figure 2.37. As mentioned above it is composed of thirty-two 128-bit registers, one *32 to 1 Decoder* and four *rd_ports*.

To write a register the *wr_addr* is decoded using the *32 to 1 Decoder* to generate the *addr_en* that goes to the registers, all the registers are connected to the *wr_data* and *wr_en*

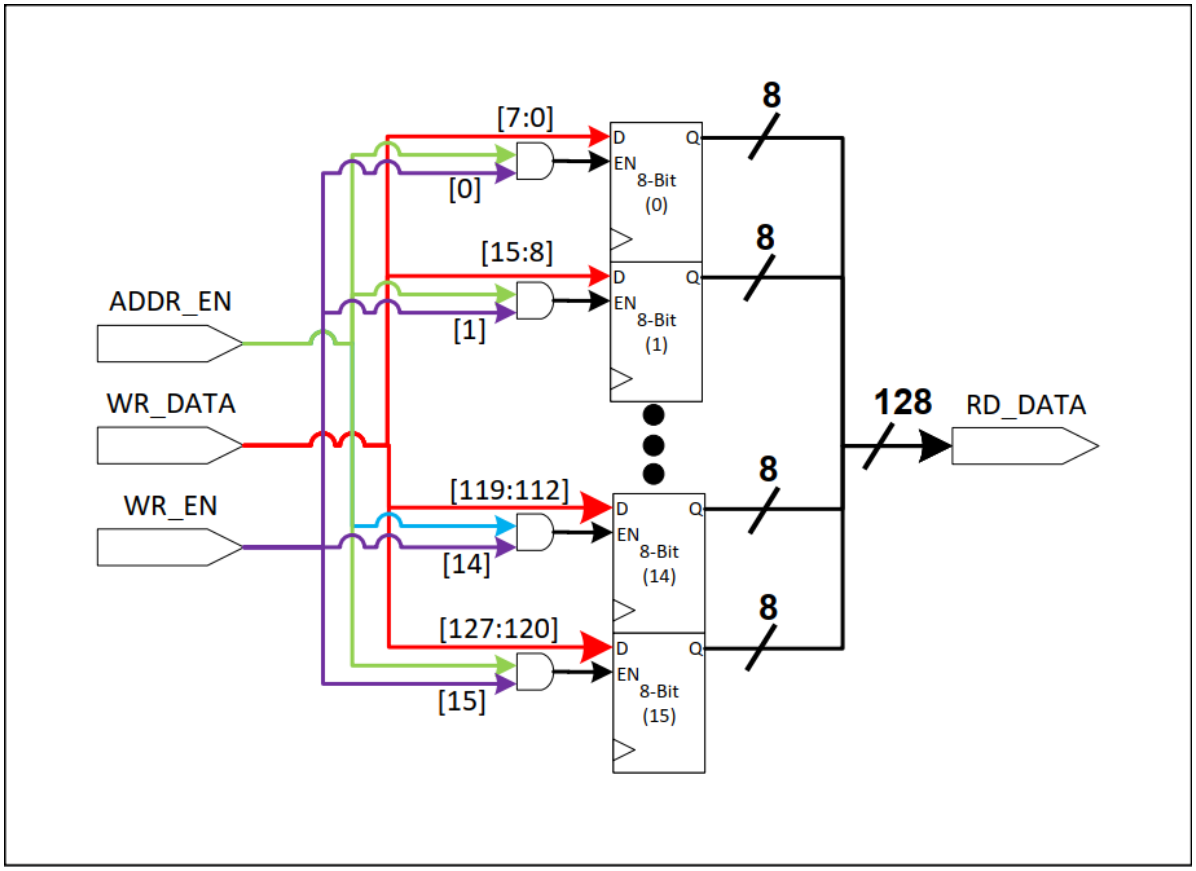


Figure 2.36: Register Vector Register File

bus, depending on the *wr_addr* and *wr_en* the corresponding register is written with the data presented in *wr_data* at the posedge of the *clock*.

The read of the registers is an asynchronous process made by the *rd_ports*. The *VRF* has four *rd_ports*. The *mask_port* reads the mask to be used by the operation, unlike the other ports this port does not receive a *rd_addr* since the mask is always located in *register[0]*, so it doesn't need an address and it only reads the lower 16-bits of the register that is where the mask is located. The *rd_a_port* and *rd_b_port* ports are used to read the operands to be used by the operation, these ports can read any register from the *VRF* and read the 128-bits of the register, each port gets its respective *rd_addr*. Finally, the *merge_port* is used to get the merge value used by some mask and compare operations, the address of this port is the *wr_addr* of the operation to be executed, it can read any register from the *VRF* but only reads the lower 16-bits of the register as the *mask_port*.

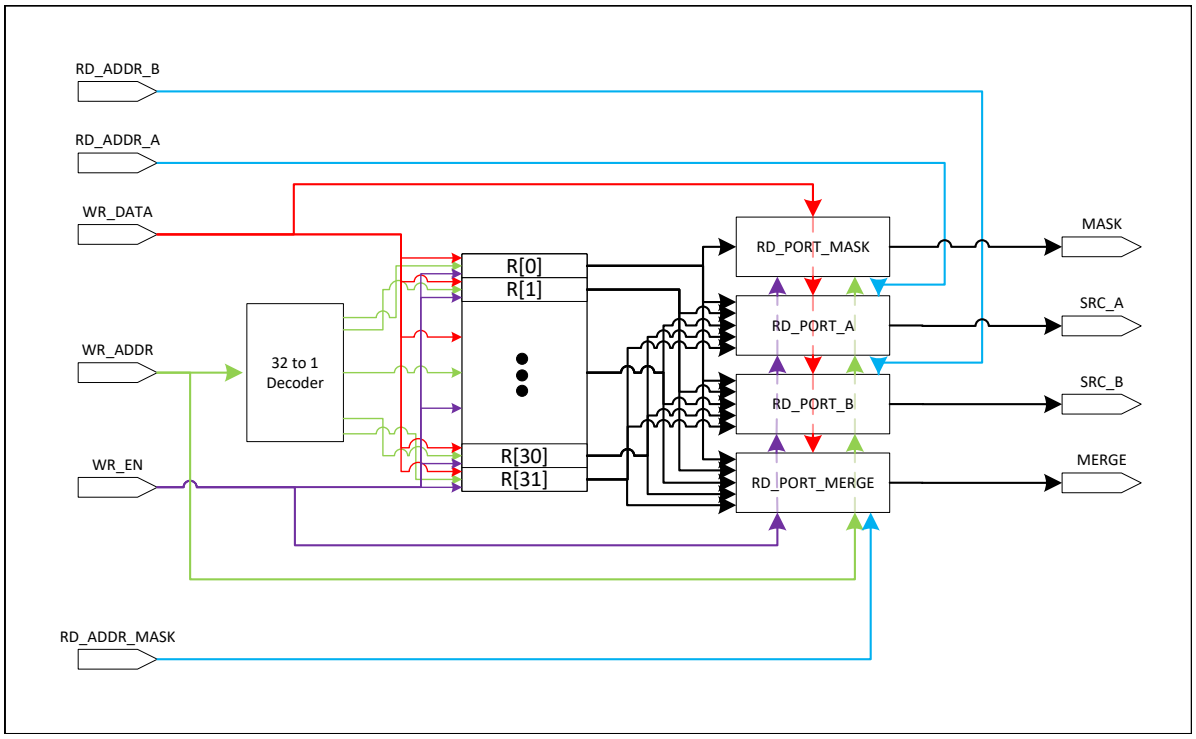


Figure 2.37: Vector Register File

The structure of the *rd_ports* is shown in Figure 2.38. Except for the *mask_port* that only reads *register[0]* all the ports receive the data from the thirty-two *rregisters* using a *rd_addr*. First, a multiplexer is used to select the data between the *registers* and then a small bypass block is implemented to get the correct data if the registers that are being read are the same registers that are being written in the same clock cycle. The bypass block is formed by a 5-bit comparator that compares the *rd_addr* with the *wr_addr*, if they are the same then the corresponding *wr_en* is checked with an AND gate, if it is one and the addresses are the same the *wr_data* is bypassed to the *rd_data*, if the addresses are not the same or *wr_en* is zero the *rd_data* is read from the *VRF*. The bypass block is independent per byte, meaning that some bytes could be read from the *VRF* while others are bypassed from *wr_data* depending on *wr_en*. The *rd_data* can be selected from the *registers* using:

$$reg_sel = regs[rd_addr] \tag{2.53}$$

$$rd_data[byte_idx] = ((wr_addr == rd_addr) \wedge wr_en[byte_idx])? \\ wr_data[byte_idx] : reg_sel[byte_idx] \tag{2.54}$$

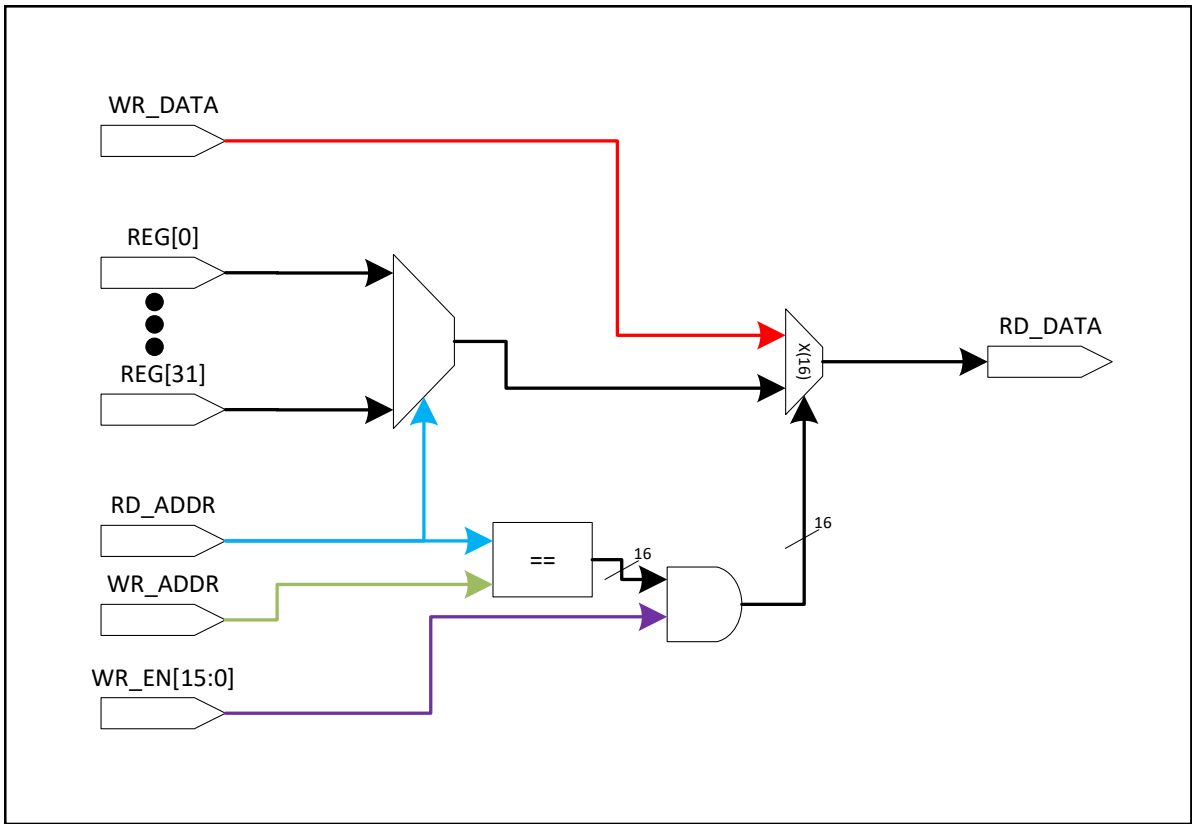


Figure 2.38: Read Port Vector Register File

2.13 Control Units

The main purpose of the *control units* is to decode the control signals, like the *opcode*, to get 1-bit control signals that are used by the other units to execute the corresponding operation, The control signals are decoded using multiple comparators, each comparator compares the control signal with a specific value to get the 1-bit decoded signal, this method does not reuse other comparators to get more complex control signals, it generates more hardware but the decoding is faster saving time in the decode stage that is a timing critical stage in most CPU's. This method also reduces the development time since it is easier to code the comparator with a fixed value than analyzing the control signals with the different combinations to check what logic can be reused, an example of this is shown in Figures 2.39 and 2.40.

The second purpose of the *control units* is to stage the signals that are used in the following stages, each control unit instantiates some registers to stage the needed signals, these

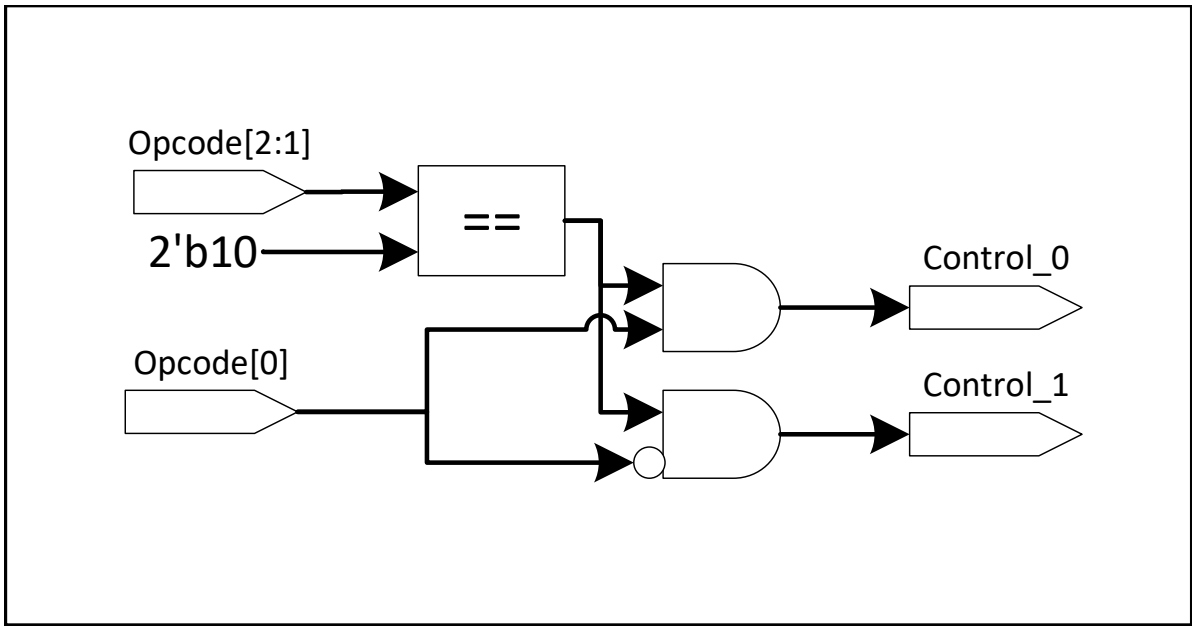


Figure 2.39: Decode unit with reuse

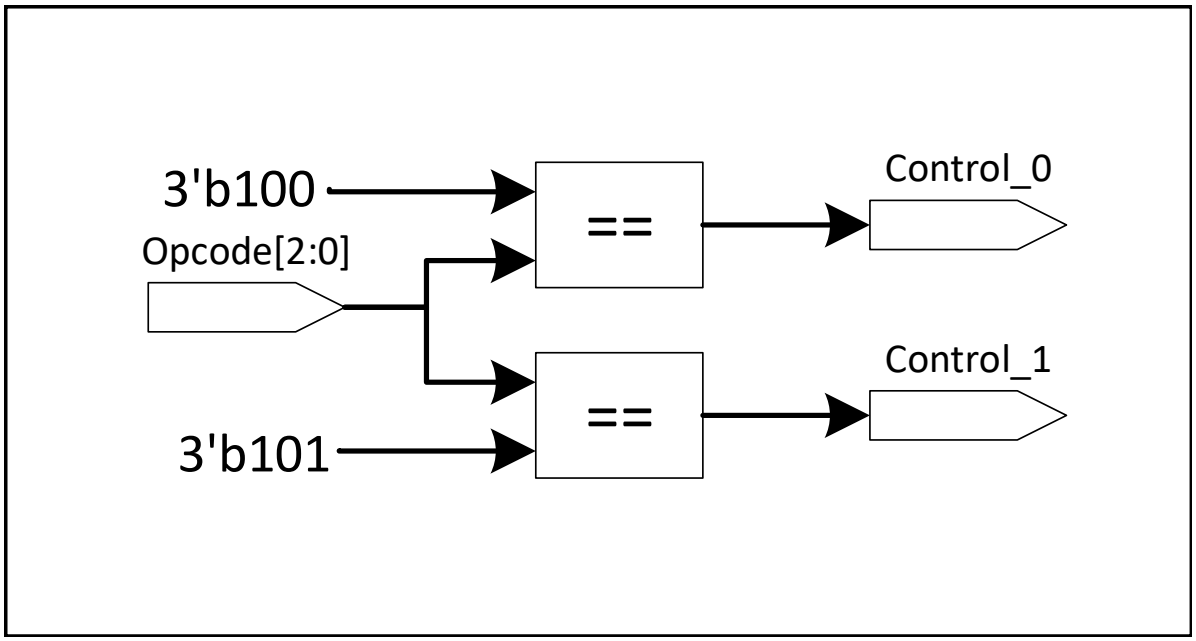


Figure 2.40: Decode unit with comparators

registers can be enabled/disabled to stall the processor or can be flushed for miss prediction recoveries.

There are three main control units instantiated in the *Vector Engine*, *RISCV-V Control Unit*, *VRF Control Unit* and *CSR Control Unit*.

2.13.1 RISCV-V Control Unit

This *control unit* decodes the *opcode* to generate the control signals used by the *Execution Unit* to execute the operations and stages these control signals from the *Instruction Decode Stage* to the *Execute Stage*. The *control unit* only decodes the Vector operations, the normal operations are decoded by the RISCV decode unit, if the operation is not a vector operation all the control signals generated by this unit are set to zero. The control signals decoded are shown in Table 2.10 and the control signals staged in Table 2.11.

Table 2.10: RISCV-V Control signals

Control Signal	Decode Signal	Purpose
is_vector_op	opcode	Indicates if it is a vector operation used to enable all the other control signals
is_vector_vector	funct3	Indicates if both operands are vector sources
is_scalar_op	funct3	Indicates if srca is a scalar operand
is_scalar_imm	funct3	Indicates if srca is an immediate scalar operand
is_scalar_int	funct3	Indicates if srca is an integer scalar operand
is_scalar_fp	funct3	Indicates if srca is a floating point scalar operand
is_i2v	funct6 and funct3	Indicates if the operation is an integer to vector operation
is_v2i	funct6 and funct3	Indicates if the operation is a vector to integer operation
is_and	funct6 and funct3	Indicates if the bitwise AND block is enabled
is_or	funct6 and funct3	Indicates if the bitwise OR block is enabled
is_xor	funct6 and funct3	Indicates if the bitwise XOR block is enabled
is_negate_srca	funct6 and funct3	Indicates if the srca is negated before the execution of the operation
is_negate_srcb	funct6 and funct3	Indicates if the srcb is negated before the execution of the operation
is_negate_result	funct6 and funct3	Indicates if the result is negated after the execution of the operation
use_mask	funct6 and funct3	Indicates if it is a masked operation
is_shift	funct6 and funct3	Indicates if the shifter is enabled
is_left	funct6 and funct3	Indicates if it is a left shift operation
is_arith	funct6 and funct3	Indicates if it is an arithmetic shift operation
is_reduct	funct6 and funct3	Indicates if it is a reduction operation
is_add	funct6 and funct3	Indicates if the adder block is enabled
is_sub	funct6 and funct3	Indicates if the adder is used as subtractor
is_mul	funct6 and funct3	Indicates if the multiplier block is enabled
is_zero_ext	funct6 and funct3	Indicates if it is a zero extend operation
is_sign_ext	funct6 and funct3	Indicates if it is a sign extend operation
is_set_equal	funct6 and funct3	Indicates if the set_equal block is enabled
is_compare	funct6 and funct3	Indicates if it is a compare operation
is_set_nequal	funct6 and funct3	Indicates if the set_nequal block is enabled
is_set_less	funct6 and funct3	Indicates if the set_less block is enabled
is_set_greater	funct6 and funct3	Indicates if the set_greater block is enabled
is_max	funct6 and funct3	Indicates if it is a MAX operation
is_min	funct6 and funct3	Indicates if it is a MIN operation
is_high	funct6 and funct3	Indicates if only the high part of the operands are used
is_signed	funct6 and funct3	Indicates if it is a signed operation
use_carry	funct6 and funct3	Indicates if the operation makes use of the carry bus (MASK bus)
rf_wr_en	funct6 and funct3	Indicates if the operations writes the register file

Table 2.11: RISCV-V Stage Control signals

Control Signal	Start Stage	End Stage	Num Stages	Num Bits
is_vector_vector	INSTRUCTION DECODE	EXECUTE	1	1
is_scalar_op	INSTRUCTION DECODE	EXECUTE	1	1
is_scalar_imm	INSTRUCTION DECODE	EXECUTE	1	1
is_scalar_int	INSTRUCTION DECODE	EXECUTE	1	1
is_scalar_fp	INSTRUCTION DECODE	EXECUTE	1	1
is_i2v	INSTRUCTION DECODE	EXECUTE	1	1
is_v2i	INSTRUCTION DECODE	EXECUTE	1	1
is_and	INSTRUCTION DECODE	EXECUTE	1	1
is_or	INSTRUCTION DECODE	EXECUTE	1	1
is_xor	INSTRUCTION DECODE	EXECUTE	1	1
is_negate_srca	INSTRUCTION DECODE	EXECUTE	1	1
is_negate_srcb	INSTRUCTION DECODE	EXECUTE	1	1
is_negate_result	INSTRUCTION DECODE	EXECUTE	1	1
use_mask	INSTRUCTION DECODE	EXECUTE	1	1
is_shift	INSTRUCTION DECODE	EXECUTE	1	1
is_left	INSTRUCTION DECODE	EXECUTE	1	1
is_arith	INSTRUCTION DECODE	EXECUTE	1	1
is_reduct	INSTRUCTION DECODE	EXECUTE	1	1
is_add	INSTRUCTION DECODE	EXECUTE	1	1
is_sub	INSTRUCTION DECODE	EXECUTE	1	1
is_mul	INSTRUCTION DECODE	EXECUTE	1	1
is_zero_ext	INSTRUCTION DECODE	EXECUTE	1	1
is_sign_ext	INSTRUCTION DECODE	EXECUTE	1	1
is_set_equal	INSTRUCTION DECODE	EXECUTE	1	1
is_compare	INSTRUCTION DECODE	EXECUTE	1	1
is_set_nequal	INSTRUCTION DECODE	EXECUTE	1	1
is_set_less	INSTRUCTION DECODE	EXECUTE	1	1
is_set_greater	INSTRUCTION DECODE	EXECUTE	1	1
is_max	INSTRUCTION DECODE	EXECUTE	1	1
is_min	INSTRUCTION DECODE	EXECUTE	1	1
is_high	INSTRUCTION DECODE	EXECUTE	1	1
is_signed	INSTRUCTION DECODE	EXECUTE	1	1
use_carry	INSTRUCTION DECODE	EXECUTE	1	1
imm	INSTRUCTION DECODE	EXECUTE	1	5
vs1	INSTRUCTION DECODE	EXECUTE	1	5
vs2	INSTRUCTION DECODE	EXECUTE	1	5

2.13.2 VRF Control Unit

The purpose of the VRF Control unit is to stage the signals that are produced or used by the *VRF* or by the *Integer Register File*. It stages the *vector register file write address* decoded in *Instruction Decode* to *Write Back*, the *sources*, *mask* and *merge read data* in *Instruction Decode* to *Execute* and the *result* and *vector register file write enable* generated in *Execute* to *Write Back*. It also stages the *integer register file write enable* and *write data* to *Write Back*. The signals staged by this control unit are shown in Table 2.12.

Table 2.12: VRF Stage signals

Control Signal	Start Stage	End Stage	Num Stages	Num Bits
rf_wr_addr	INSTRUCTION DECODE	WRITE BACK	3	5
rf_wr_en	EXECUTE	WRITE BACK	2	16
rf_wr_data	EXECUTE	WRITE BACK	2	128
rf_rd_data_srca	INSTRUCTION DECODE	EXECUTE	1	128
rf_rd_data_srcb	INSTRUCTION DECODE	EXECUTE	1	128
int_rf_rd_data	INSTRUCTION DECODE	EXECUTE	1	32
int_rf_wr_data	EXECUTE	WRITE BACK	2	32
int_rf_wr_en	INSTRUCTION DECODE	WRITE BACK	3	1
mask	INSTRUCTION DECODE	EXECUTE	1	16
mask_merge	INSTRUCTION DECODE	EXECUTE	1	16

2.13.3 CSR Control Unit

The CSR Control Unit generates the write enables needed by the *CSRs* and stages the data read from the CSRs in *Instruction Decode* to be used in *Execute* by the *Execution Unit*. The signals staged by this control unit are shown in Table 2.13.

Since the CSR can be written internally by the *Vector Engine* or externally by the main RISCV CPU this control unit arbitrates between both providers by enabling the *CSR write enables*. The external and internal write enables are received during *Instruction Decode* while the data to be written to the *CSRs* is received during *Execute*. The *CSR* can be written only by one interface, external or internal, per clock cycle, therefore the *write enables* are ored together and the *CSR data* is also ored to get the data to be written to the *CSRs* during *Execute* as shown in Figure 2.41.

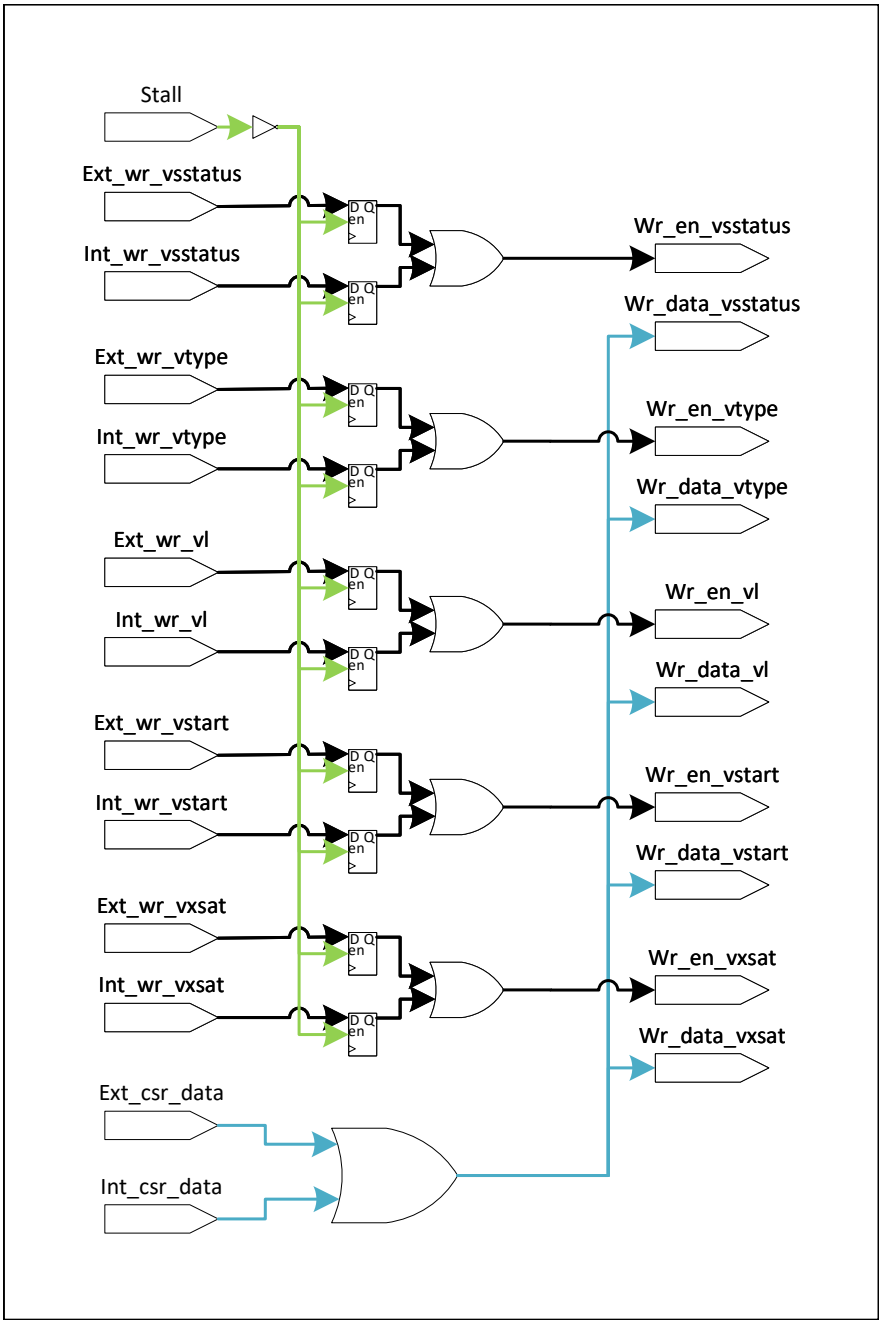


Figure 2.41: CSR Control Unit

Table 2.13: CSR Stage signals

Control Signal	Start Stage	End Stage	Num Stages	Num Bits
ext_wr_vsstatus	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vsstatus	INSTRUCTION DECODE	EXECUTE	1	1
ext_wr_vtype	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vtype	INSTRUCTION DECODE	EXECUTE	1	1
ext_wr_vl	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vl	INSTRUCTION DECODE	EXECUTE	1	1
ext_wr_vstart	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vstart	INSTRUCTION DECODE	EXECUTE	1	1
ext_wr_vxrm	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vxrm	INSTRUCTION DECODE	EXECUTE	1	1
ext_wr_vxsat	INSTRUCTION DECODE	EXECUTE	1	1
vec_wr_vxsat	INSTRUCTION DECODE	EXECUTE	1	1
vsstatus	INSTRUCTION DECODE	EXECUTE	1	11
vtype	INSTRUCTION DECODE	EXECUTE	1	9
vl	INSTRUCTION DECODE	EXECUTE	1	4
vstart	INSTRUCTION DECODE	EXECUTE	1	4
vxrm	INSTRUCTION DECODE	EXECUTE	1	2
vxsat	INSTRUCTION DECODE	EXECUTE	1	1

2.14 Bypass Unit

The *Bypass Unit* has two main purposes. The first one is bypassing the data from the *Memory* and *Write Back* stages to the *Execution Unit* to remove read after write dependencies. The second purpose is to replicate and sign extend the immediate and scalar operand to be used as a vector operand. The structure of the *Bypass Unit* is shown in Figure 2.42.

The *Bypass Unit* can bypass *srca*, *srcb*, *mask* and *mask_merge*, each structure can be bypassed from the *Memory* and *Write Back* stages and each byte of the structures can be bypassed independently, meaning that some bytes could be bypassed while other bytes are gotten directly from the *VRF* depending on the *wr_en* of the bytes.

Each structure has its independent bypass module that can be observed in Figure 2.43. First, the read address of the structure is compared with the write address of the structure in the *Memory* and *Write Back* stages to check for dependencies, then the dependency signal generated in the previous stage is anded with the *wr_en* of each byte to get a selector signal per byte that is used to select the data between the *VRF*, the *Memory* stage or the *Write Back* stage.

At the same time that the bypass modules are getting the selector signals, the scalar/immediate data is replicated and sign extended. To maintain the simplicity of the design the *Execution Unit* only operates on vector operands, it is agnostic if the operand is a vector, a scalar integer or a 5-bit immediate value, for that reason the bypass unit replicates this data to all the elements of the operand, for example, if the source is immediate and the operation is *OSIZE_8* the immediate value is sign extended to 8-bits and replicated sixteen times, one per

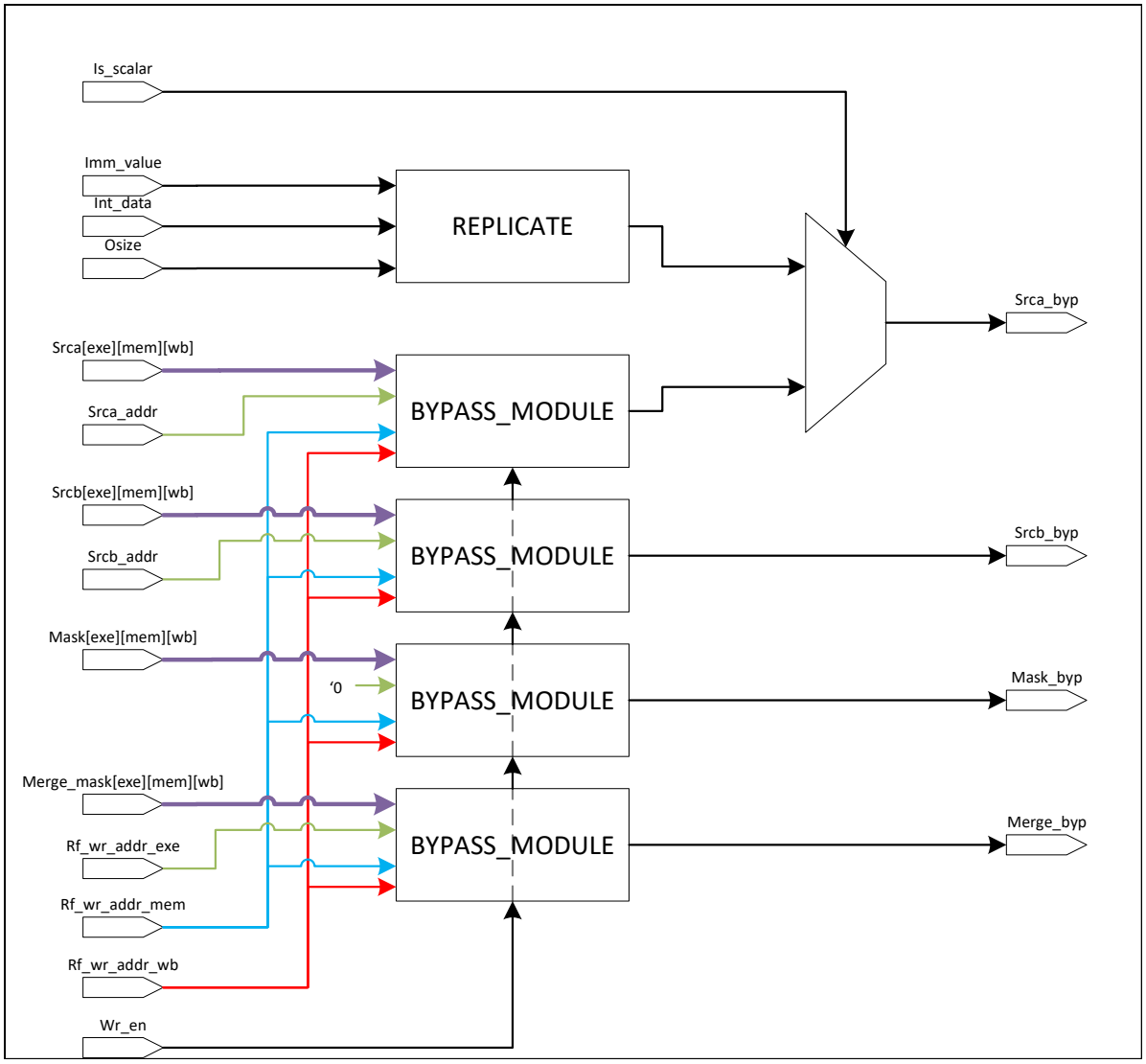


Figure 2.42: Bypass unit

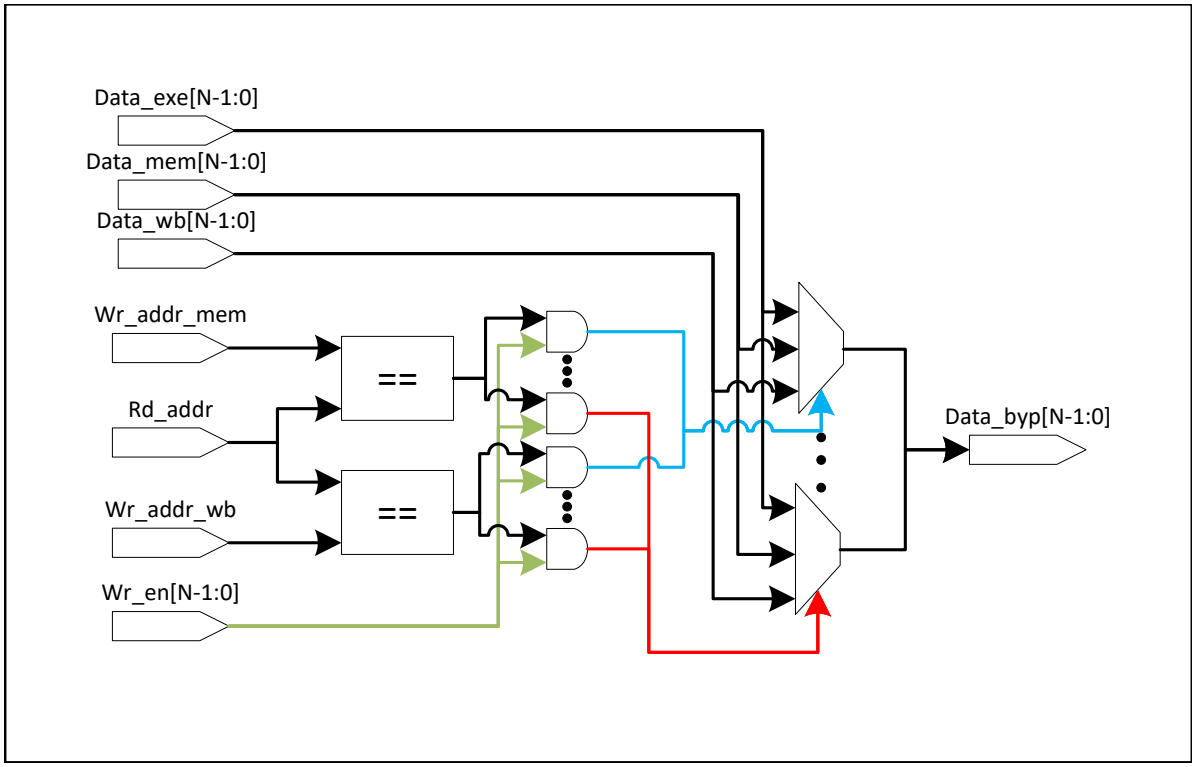


Figure 2.43: Bypass module

element in the operand, if the source is a 32-bit integer source instead of an immediate value it is truncated to 8-bits and replicated sixteen times.

This action is achieved using the replicator module shown in Figure 2.44. First, the scalar data is selected between the 32-bit integer source or the immediate source depending on the type of operation. Then the scalar data is sign extended to 128-bits, the maximum supported *OSIZE*, the extended data is replicated to the different *OSIZEs* in five different buses, one per *OSIZE* and getting the appropriate number of elements depending on the *OSIZE*. Finally one bus is selected depending on the operation *OSIZE*. Only one replicator module is needed since only the *srca* can be replaced with a scalar value and a multiplexor is used to select between the replicated data or the bypass data for *srca*.

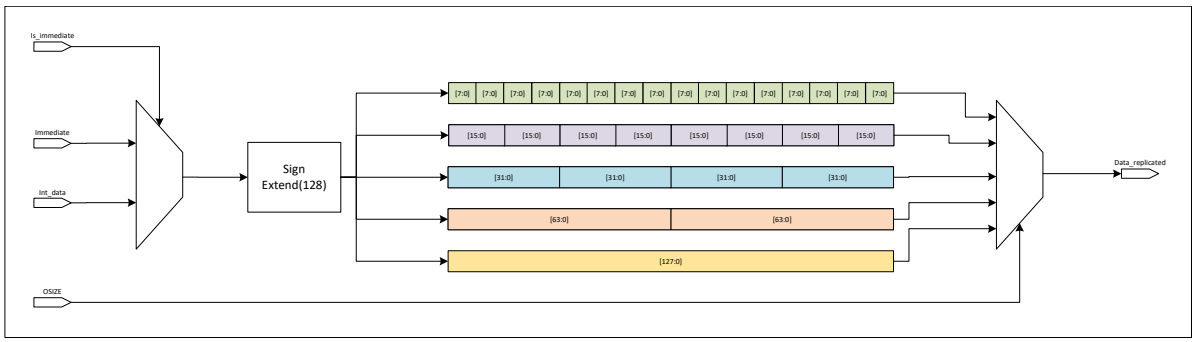


Figure 2.44: Replicator module

2.15 Decode Element

The *Decode Element* block reads the *CSRs* to decode the operation *OSIZE*, decode the *Active/Inactive* elements, generate the *wr_en* signals and generate the appropriate *mask* to be used. The complete structure of the *Decode Element* block is shown in Figure 2.45.

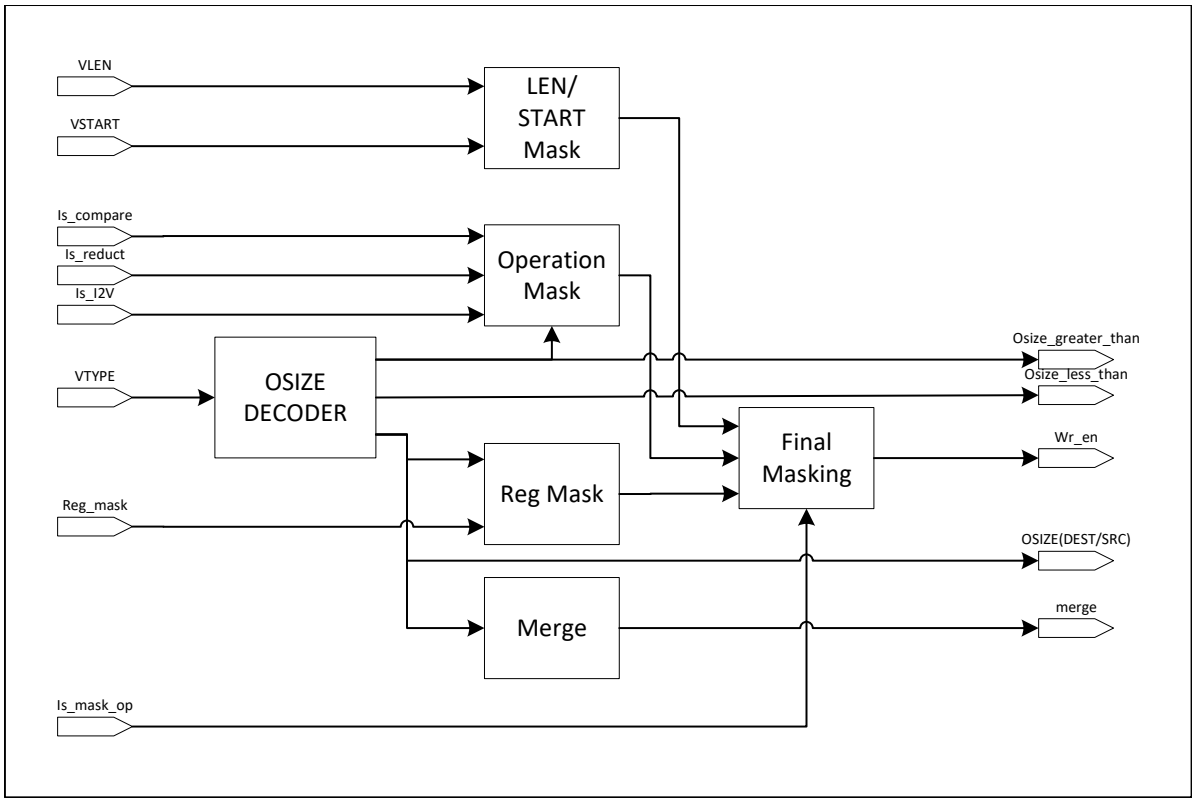


Figure 2.45: Decode Element

The operation *OSIZE* can be obtained directly from *VTYPE.vsew* and a one-hot vector is generated using this information with the less significant bit signaling OSIZE_8. For Most of the operations the *SRC_OSIZE* and *DST_OSIZE* are the same, but for zero/sign extend operations the *DST_OSIZE* is gotten from *VTYPE.vsew* and *SRC_OSIZE* by decoding *srca_addr* as shown in the following snippet.

```
    dst_osize = VTYPE.sew
    if (is_extend) begin
        case(srca_addr[1:0])
            2'b01 : src_osize = dst_osize >> 3 (i.e OSIZE_64 -> OSIzE_8);
            2'b10 : src_osize = dst_osize >> 2 (i.e OSIZE_64 -> OSIZE_16);
            2'b11 : src_osize = dst_osize >> 1 (i.e OSIZE_64 -> OSIzE_32);
        endcase
    end else begin
        src_osize = dst_osize;
    end
```

Once the *OSIZE* is decoded two 5-bit signals are generated using a thermometer encoding as shown in Figure 2.46. The first signal is *is_greater_osize* that indicates if the *OSIZE* is

greater than a specific *OSIZE*, with the less significant bit indicating that is greater than *OSIZE_0* and the most significant bit greater than *OSIZE_64*. The second thermometer signal is *is_less_osize* that indicates if the *OSIZE* is less than a specific *OSIZE*, the most significant bit indicates that is less than *OSIZE_256* and the less significant bit less than *OSIZE_16*. These signals are used by the *Execution Unit* to configure the *ALUs* and execute the operation. The *is_greater_osize* and *is_less_osize* vectors are generated using:

$$is_greater_osize[idx] = \begin{cases} 1, \text{ if } idx = 0 \\ (OSIZE > (8 * (2^{idx-1}))), \text{otherwise} \end{cases} \tag{2.55}$$

$$is_less_osize[idx] = \begin{cases} 1, \text{ if } idx >= 4 \\ (OSIZE < (8 * (2^{idx+1}))), \text{otherwise} \end{cases} \tag{2.56}$$

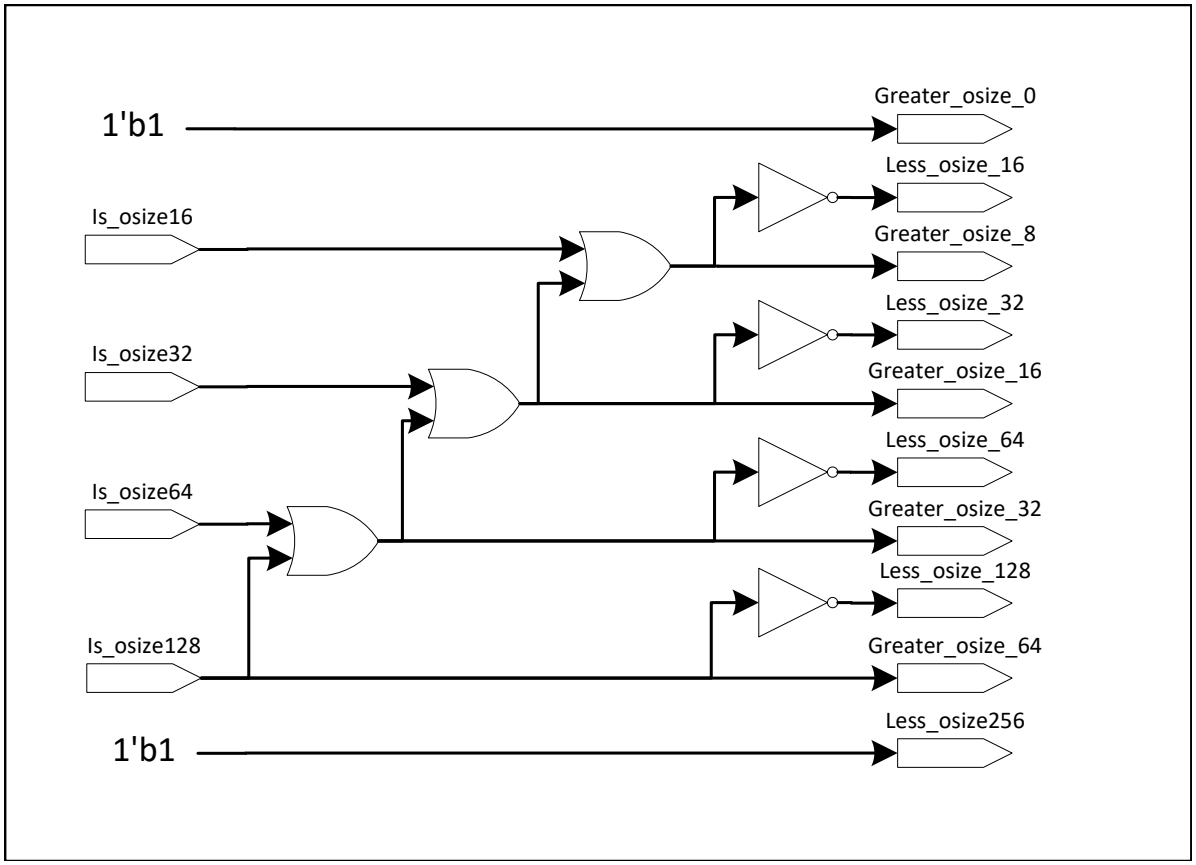


Figure 2.46: Greater/Less than OSIZE

At the same time *VL* and *VSTART* are read to decode the *Active/Inactive* elements of the operation. First a thermometer signal *len_greater_than* is encoded with the least significant bit meaning that *VL* is greater than zero and the most significant bit that *VL* is greater than fifteen. In the same way the thermometer signal *start_less_than* is encoded with the less significant bit indicating that *VSTART* is less than one and the most significant bit meaning that *VSTART* is less than sixteen. Finally *len_greater_than* and *start_less_than* are anded together to get a

mask with the *Active/Inactive* elements of the operation according to *VL* and *VSTART*. This logic is shown in Figure 2.47 and in:

$$len_greater_than[idx] = (LEN > idx) \tag{2.57}$$
$$start_less_than[idx] = (START <= idx) \tag{2.58}$$

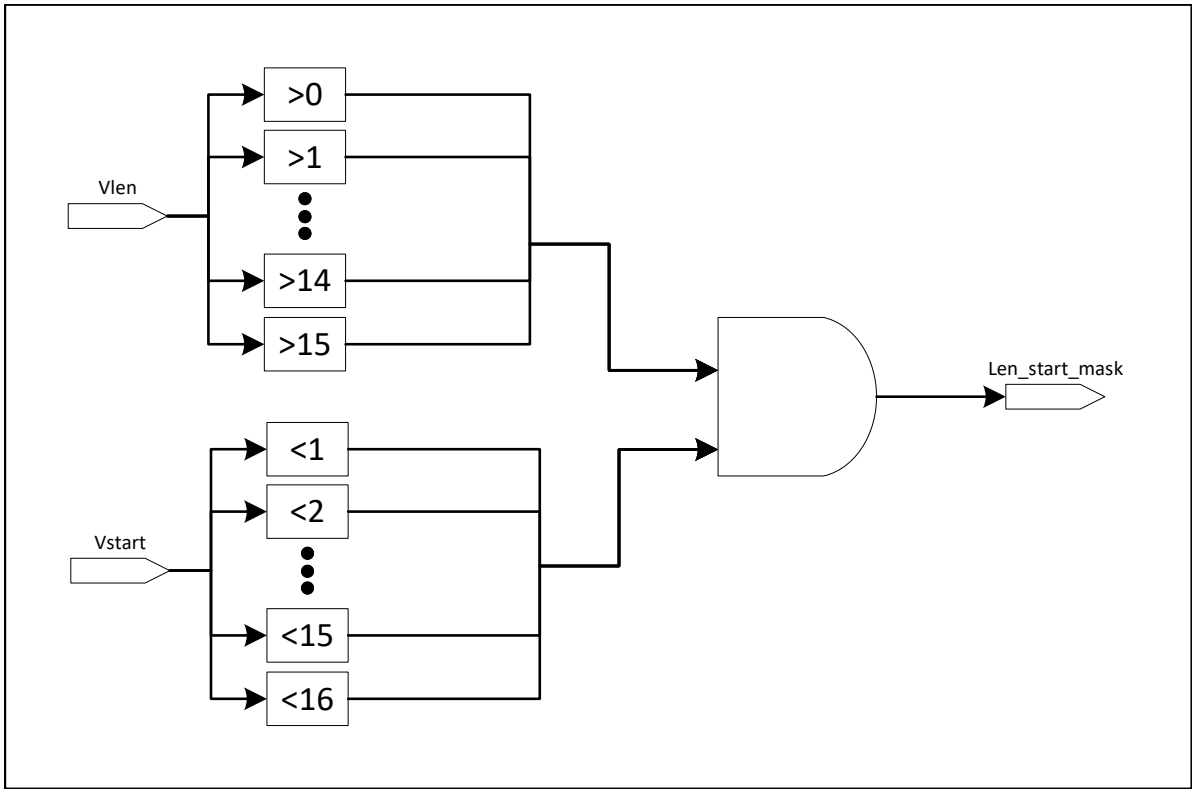


Figure 2.47: Decode element start/len

Figure 2.48 shows how the *operation mask* is generated in parallel to turn off the elements that are not used depending on the operation. The *I2V_compare_reduct_mask* is generated by turning off the bits of the elements that are not used depending on *OSIZE*, if the operation is not an I2V, a compare or a *reduction* operation all the bits are turned on, if it is an I2V, a compare or a *reduction* operation just the lower bits are set to one depending on the *OSIZE*, additionally if the operation is a *compare* operation the higher fourteen bits are set to zero, and the bit[1] is set to zero if it is a *compare* operation and is not *OSIZE_8*. The following snippet shows how the *I2V_compare_reduct_mask* mask is generated.

```
1    I2V_compare_reduct_mask = (is_i2v || is_compare || is_reduct) ? 'd0 :
     'd1;
2    case (osize)
3        OSIZE_8 :  I2V_compare_reduct_mask[0]    = 1'b1;
4        OSIZE_16:  I2V_compare_reduct_mask[1:0]  = 2'b11;
5        OSIZE_32:  I2V_compare_reduct_mask[3:0]  = 4'b1111;
6        OSIZE_64:  I2V_compare_reduct_mask[7:0]  = 8'b1111_1111;
```

```
7          OSIZE_128: I2V_compare_reduct_mask[15:0] = 16'b1111_1111_1111_1111
   ;
8      endcase
9      I2V_compare_reduct_mask[15:2] &= ~{14{is_compare}};
10     I2V_vompare_reduct_mask[1] &= ~(is_compare & ~OSIZE_8);
```

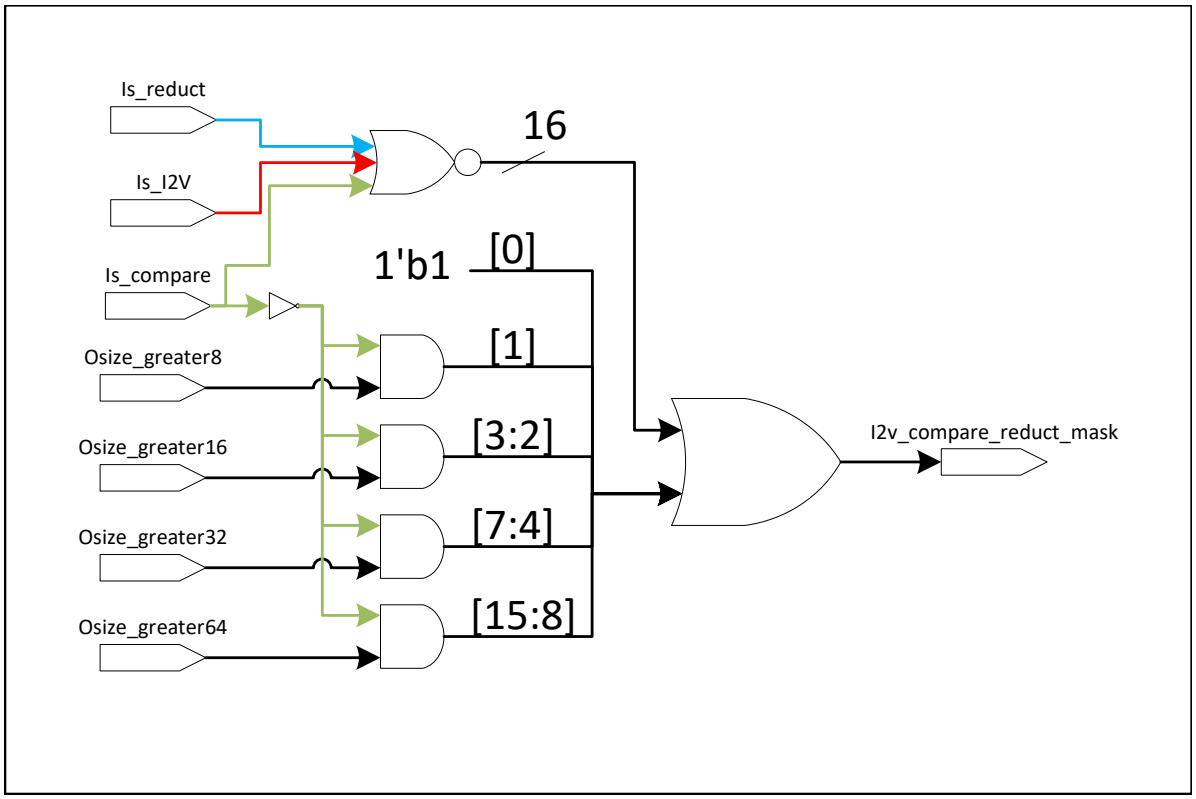


Figure 2.48: Decode element operation mask

The *Vector Masking* is also generated in this module. The Vector mask is read from the *VRF*, but depending on the *OSIZE* just some bits are used as *Mask*, for example if the operation is *OSIZE_8* the sixteen bits of the mask are used, one per element, but if the operation is *OSIZE_128* just the bit zero is used to mask all the bytes of the operation. For that reason five different masks are generated by replicating the bits of the mask register and one of those masks is selected depending on the *OSIZE* as shown in Figure 2.49.

Once all the masks are generated they are anded together and the result is ored with an extra *is_mask_op* mask to turn on the two less significant bits if the operation is a mask operation. With all masks merged the final mask that is used as *wr_en* is obtained to just modify the selected elements, bytes, of the *VRF* as shown in Figure 2.50.

In parallel to the mask generation, the *merge* vector is generated. This vector is used to indicate if the bytes of the sources correspond to the same element of the operation or a new element, if the byte[i] has the merge bit asserted this means that this byte is merged with the following byte and both bytes are of the same element, if the byte[i] does not has the merge bit asserted this means that the byte is not merged to the next byte and the byte is the last byte of the element. There are only five different combinations of the *merge* one per *OSIZE*, the

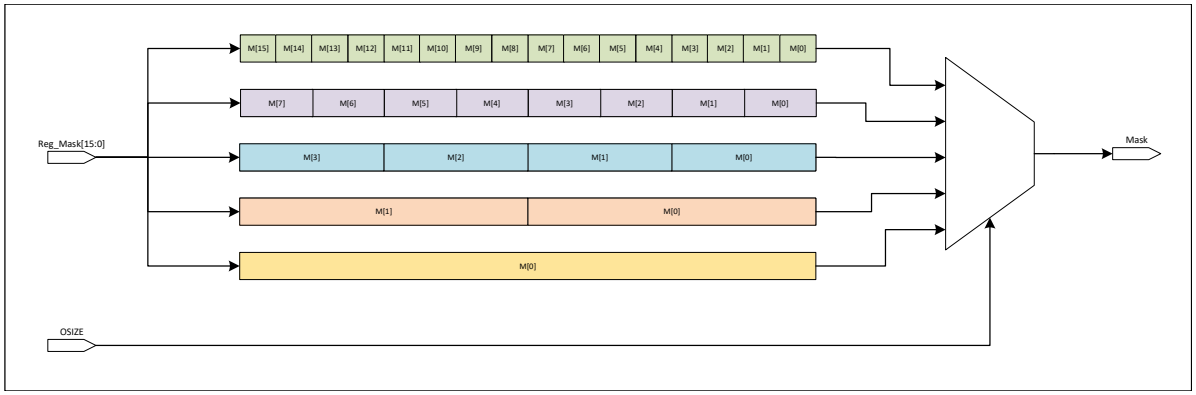


Figure 2.49: Decode element Masking

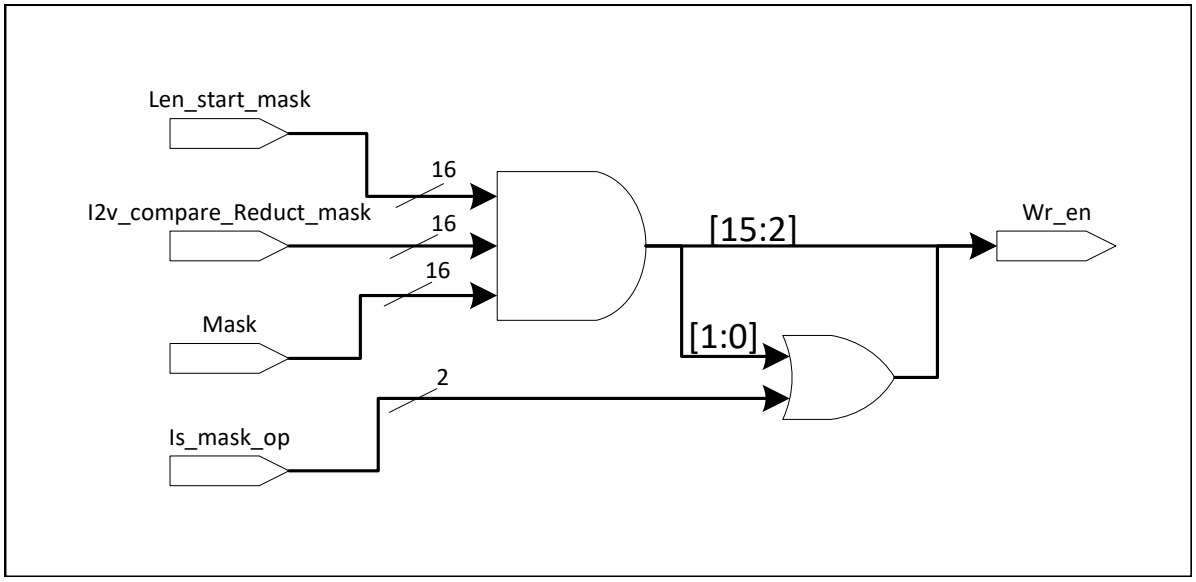


Figure 2.50: Decode element final mask

five merge vectors are generated and one merge vector is selected depending on the *OSIZE* as shown in Figure 2.51.

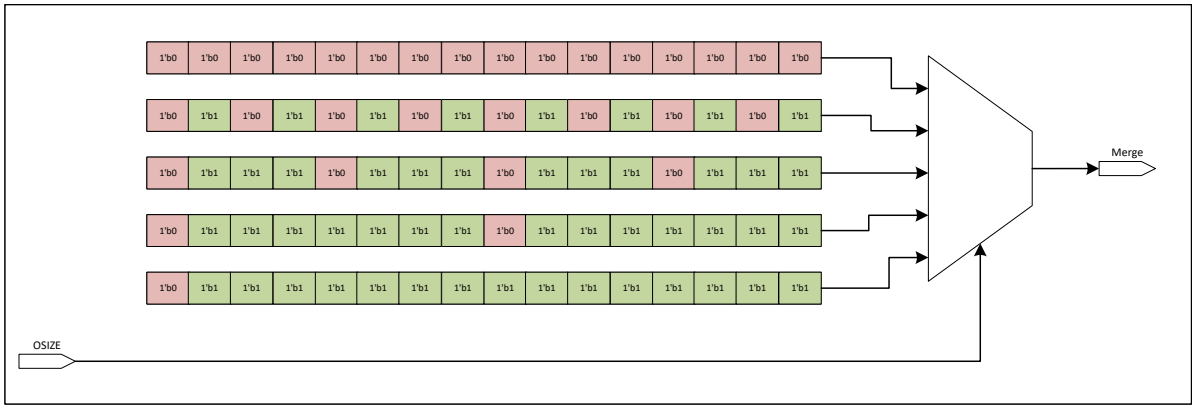


Figure 2.51: Decode element merge

2.16 Execution Unit

The main purpose of the *Execution Unit* is to execute the vector operations, this is done in three different ALUs, *Logic ALU*, *Arithmetic ALU* and Permutation ALU, the operation is sent to the corresponding ALU depending on the type of operation, only one instruction is executed per clock cycle, meaning that only one ALU produces a valid result, the other ALUs send zeroes on the bus result. At the end, the results of the three ALUs and the merge value are ored together to get the final result of the operation. The structure of the *Execution Unit* can be observed in Figure 2.52.

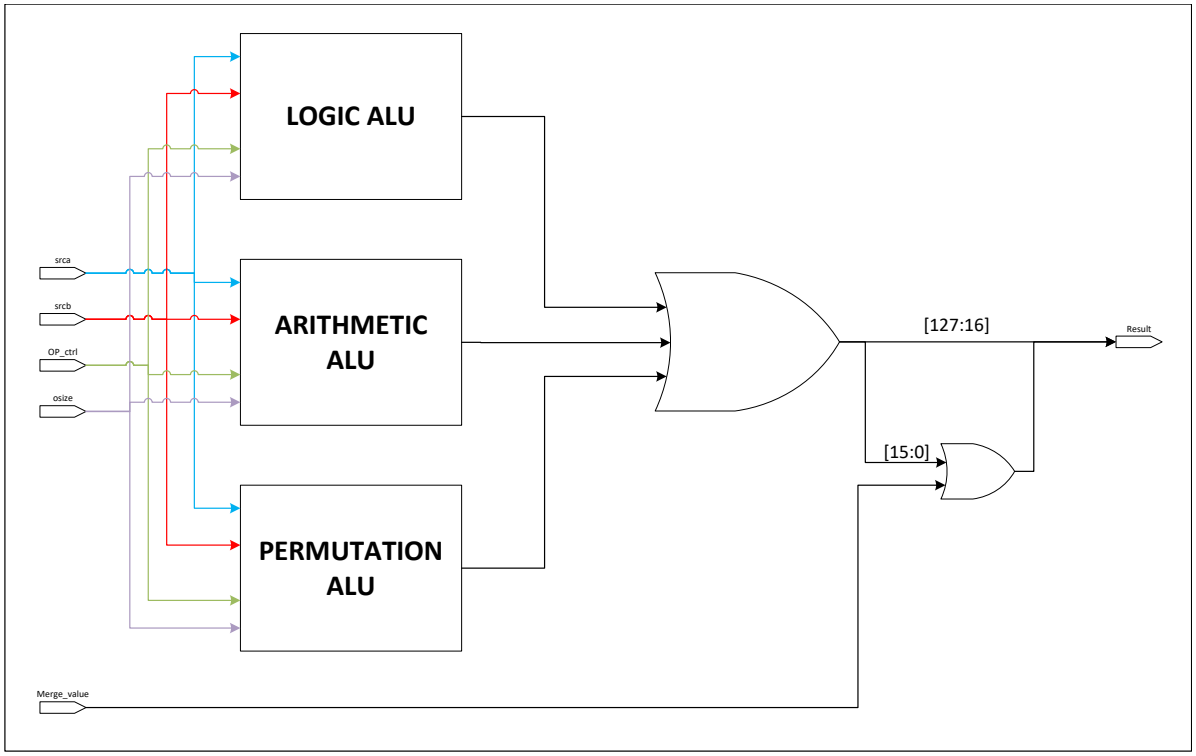


Figure 2.52: Execution Unit

2.17 Logic ALU

The *Logic ALU* is located inside the *Execution Unit* and implements the logic operation as Bitwise, Logic and Mask AND, OR and XOR operations and shift operations. The inputs of the ALU are the two sources, the *OSIZE* of the operation, the modifiers of the operation and the control signals to indicate the operation to be executed. The sources are gated inside the execution blocks depending on the operation to be executed and the results of the execution blocks are ored together to form the result of the ALU as shown in Figure 2.55.

The *Logic ALU* can execute Bitwise operations, where each bit of *srca* is computed against each bit of *srcb* and are not overlapped, is a bit to bit operation as shown in Figure 2.53. The *Logic ALU* can also execute *Reduction Operations*, as explained in its section the *Reduction Operations* take the result from the previous element to be computed against the following element and only produce one element as result, logically the operation would be implemented as Figure 2.54 by connecting in series the different blocks of the operation, but physically is implemented using the *Reduction Source* block to reuse the same execution blocks for bitwise and *Reduction Operations*.

The Mask operations are implemented inside the *Logic ALU* by using the Bitwise Execution blocks, the mask operations implement some operations as VMNAND or VMANDN that negates the *result* or the *srca* respectively, for that reason the *srca* can be negated before the execution blocks and the result after the execution blocks.

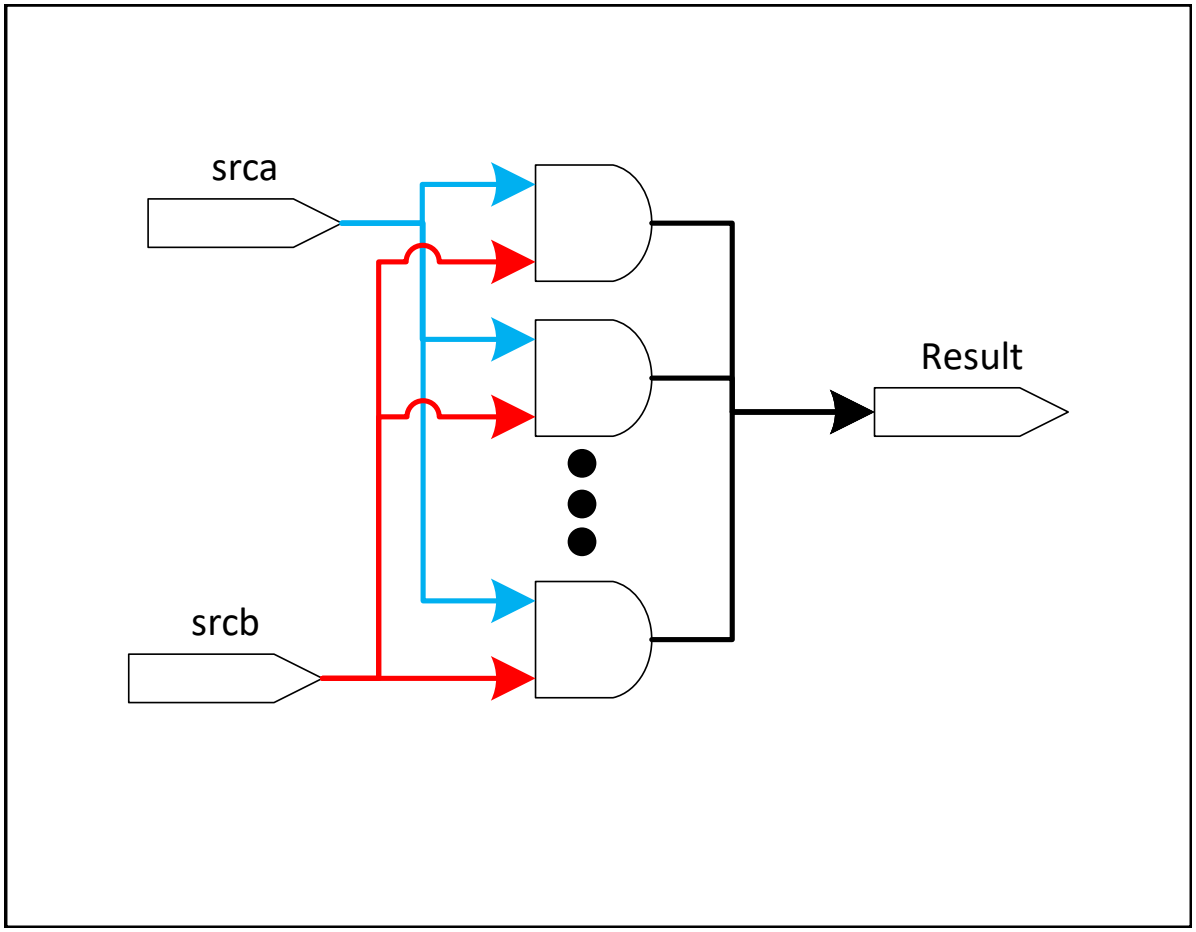


Figure 2.53: Bitwise Normal Logical

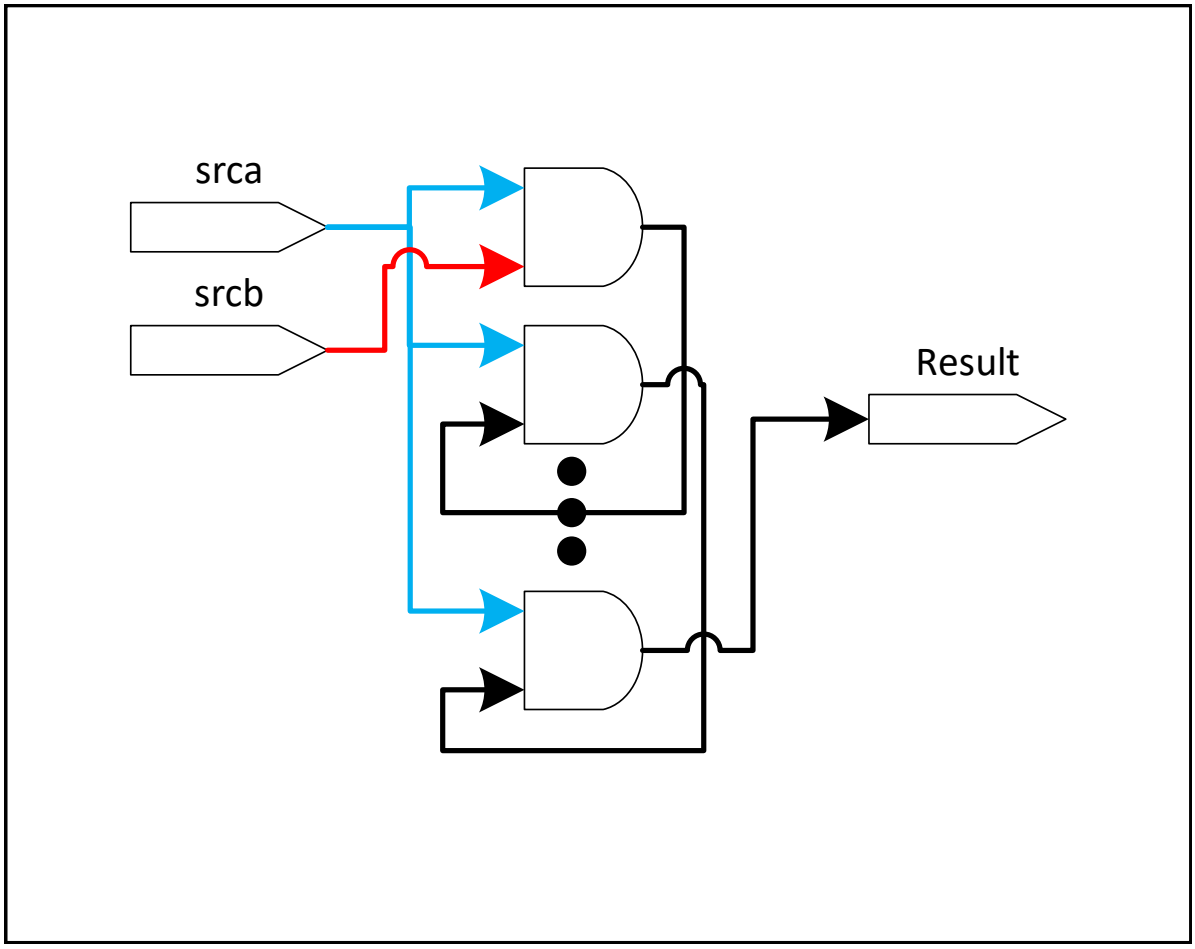


Figure 2.54: Bitwise Reduction Logical

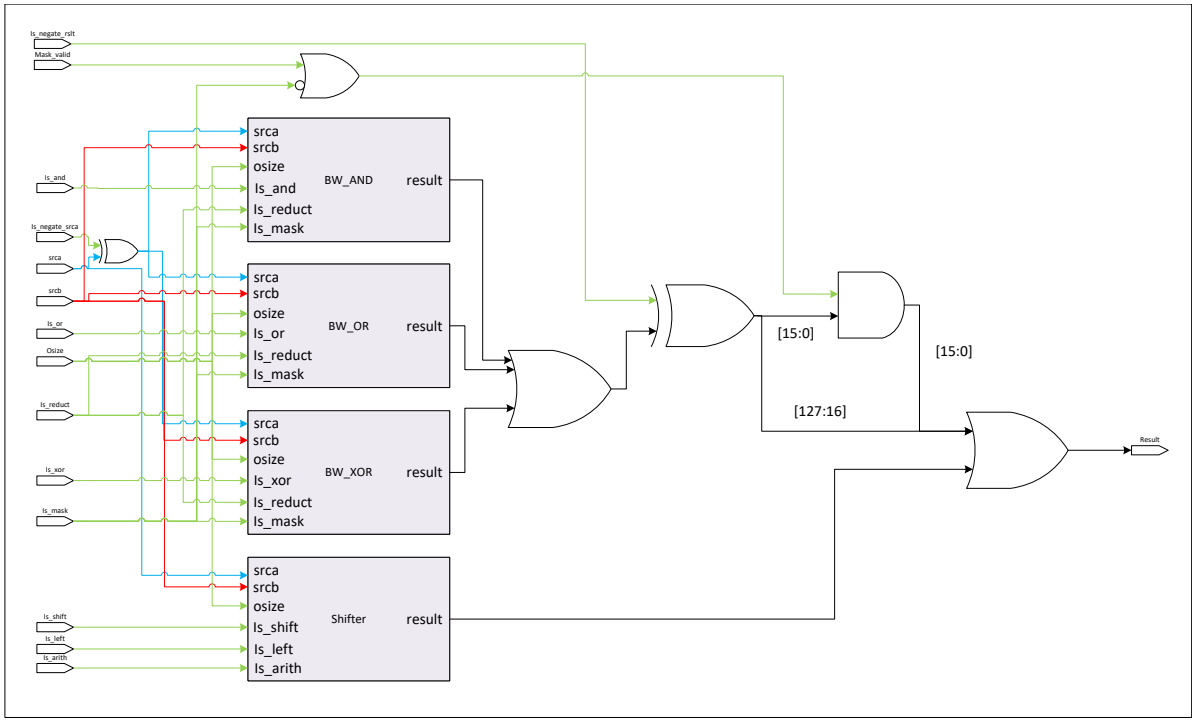


Figure 2.55: Logic ALU

2.17.1 Bitwise AND

The Execution block *Bitwise AND* implements the Bitwise, Reduction and Mask AND operations. First *srca* is qualified with *is_and* to only produce a result if the operation is executed by this block. *srcb* is ored with *is_mask* and *srcb_valid*, if it is a mask operation, or *srcb_valid[i]* is asserted means that the byte *srcb[i]* is used by the operation and the original *srcb[i]* is used as operand, if it is a mask operation or *srcb_valid[i]* is not asserted means that the byte is an Inactive byte and the *srcb[i]* operand is replaced with ones to propagate the result to the following element, used by *Reduction Operations* that needs to propagate the result to the last element even if the intermediate elements are not active.

The qualified *srca* is sent as input to the *Reduction Source* block along with the *OSIZE*, *is_reduct* and the *result* to multiplex the operand and get the correct operand depending if it is a bitwise or a *Reduction Operation*. The operand that is output from *Reduction Source* is sent to an array of 128 AND gates to be anded with the qualified *srcb* to get the final result of the operation as shown in Figure 2.56.

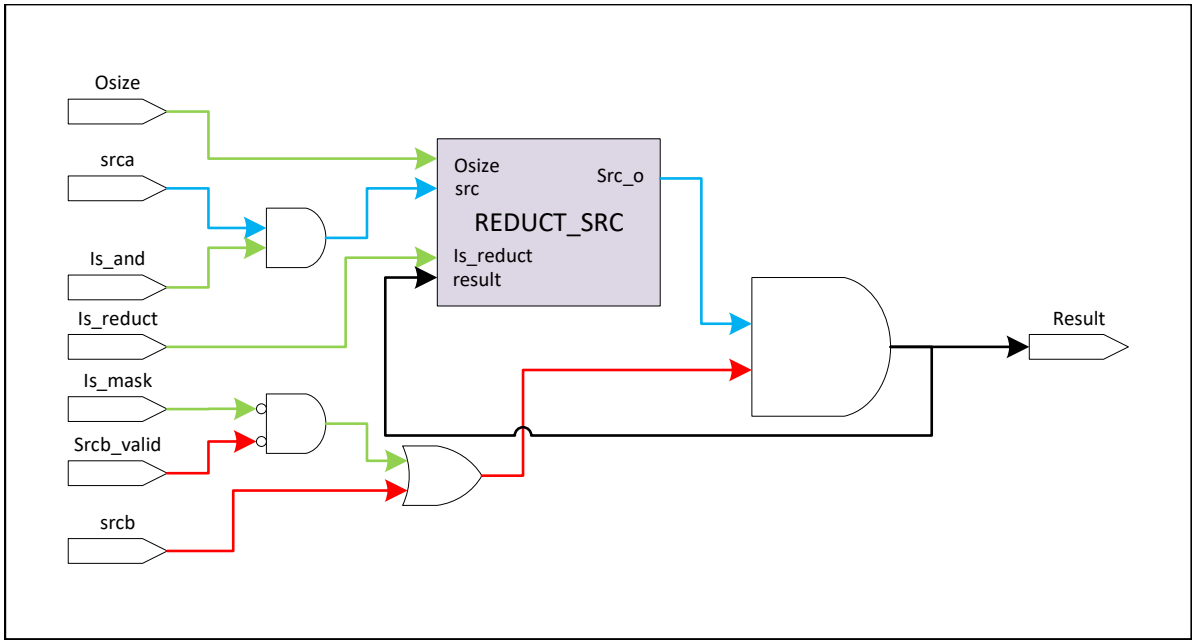


Figure 2.56: Bitwise AND

2.17.2 Bitwise OR

The execution block *Bitwise OR* implements the Bitwise, Reduction and MASK OR operations. The structure of this block is similar to *Bitwise AND*, but in this case both operands are qualified with *is_or* since if any operand is one it produces a one in the result instead of zero, and in this block *srcb* is anded with *srcb_valid* and *is_mask* to replace *srcb[i]* with zeroes and do not affect the result of the other elements if the element is not active. Finally, the output from *Reduction Source* is ored with the qualified *srcb* to produce the final result as shown in Figure 2.57.

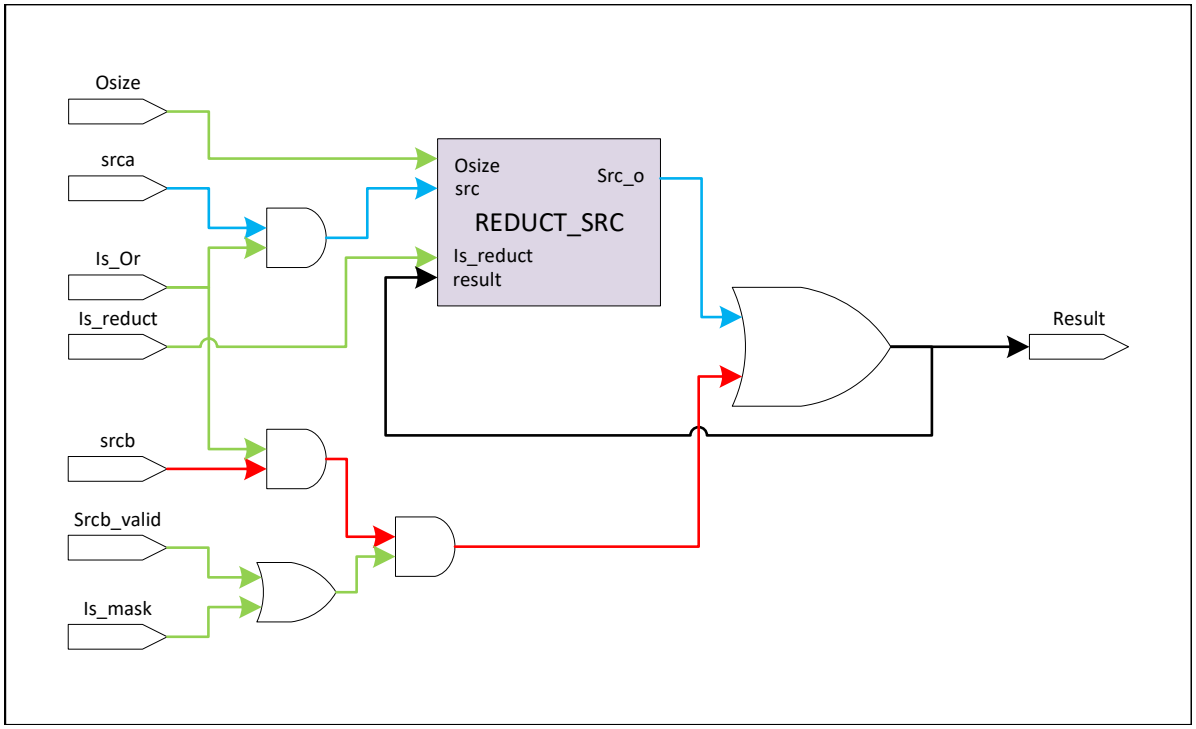


Figure 2.57: Bitwise OR

2.17.3 Bitwise XOR

The execution block *Bitwise XOR* implements the Bitwise, Reduction and Mask XOR operations. This block is similar to Bitwise OR, both sources are gated with *is_xor* to produce a zero in the result if the operation is not executed by this block, *srcb* is also anded with *srcb_valid* and *is_mask* to replace *srcb* with zeroes if the element is not active to do not affect the other elements. Finally, the qualified operands are xored together using 128 XOR gates to produce the final result.

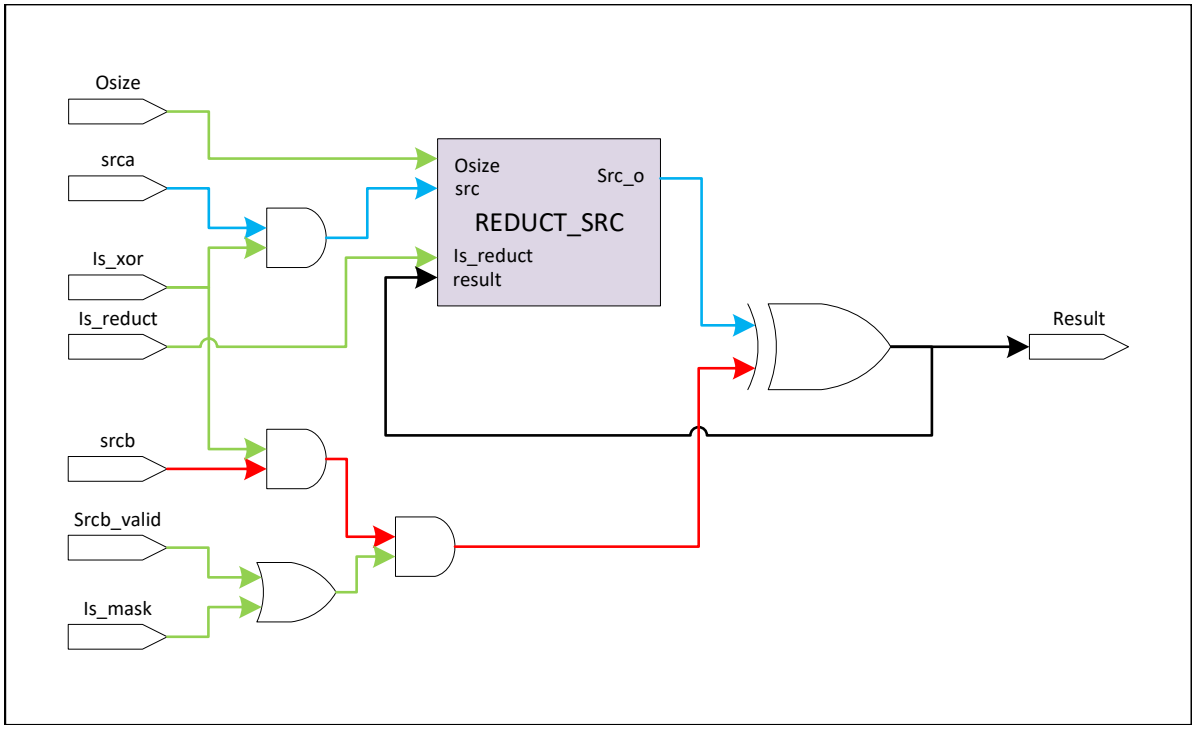


Figure 2.58: Bitwise XOR

2.17.4 Shifter

The *Shifter* implements the Logic and Arithmetic Left and Right Shifts. Left and Right shifts are implemented using only one right barrel shifter. To perform the left shifts using the right shifter first the operand is swizzled, the elements of the operand are inverted, and after the shifting, the result is also swizzled as shown in the following snippet.

```
1     //Right Shift using the right shifter
2     if (is_right) begin
3         result = srcB >> srcA;
4     //Left Shift using the right shifter
5     end else begin
6         //Swizzle operands
7         for  (int idx = 0; idx < srcB.length(); idx++) begin
8             srcB_tmp[idx] = srcB[srcB.length()-idx];
9         end
10        //Right Shift
11        result_tmp = srcB_tmp >> srcA;
12        //Swizzle result
13        for  (int idx = 0; idx < result.length(); idx++) begin
14            result[idx] = result_tmp[result.length()-idx];
15        end
16    end
```

The complete structure of the shifter can be observed in Figure 2.59. The *srcA* is connected to the Shift Count block to get the correct shift count depending on the *OSIZE*. Each *Shifter layer* and *Byte shifter* have its own shift count. *Srcb* is connected to a multiplexor to

select between the original *srcb* or the swizzle *srcb* depending on the direction of the shift. The selected operands are sent to four *Shifter Layers* that shift the upper bytes of the operand, these layers shift the whole byte as a block, do not modify the internal bits of the bytes, the result of the *Shifter Layers* is connected to sixteen *Byte shifters*, one per byte that shifts the internal bits of the bytes previously shifted. Finally, the *result* is connected to another multiplexor to select between the original *result* or the swizzled *result* depending on the direction of the shift.

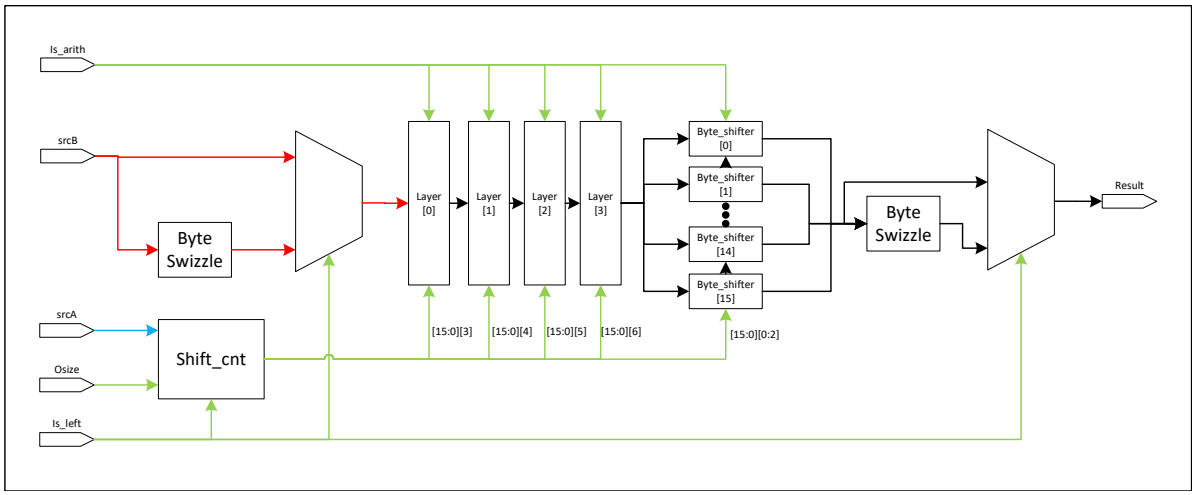


Figure 2.59: Shifter

Shift Count

The Shift count block is used to get the shift count used by each *Shifter Layer* and *Byte shifter*. Since the maximum amount of bits to be shifted is 128 just the lower seven bits of each byte of *srca* are used as shift count, and depending on the *OSIZE* fewer bits could be used, for example, if it is *OSIZE_8* just the lower three bits are used. For that reason, a mask is generated using the *OSIZE* to get just the needed bits from *srca* as shown in Figure 2.60.

Depending on the *OSIZE* the shift count of each element can come from different bytes of *srca*, if it is *OSIZE_8* each byte of *srca* is a shift count for each byte of *srcb*, but if the operation is *OSIZE_128* just the byte[0] of *srca* is used as shift count for all bytes of *srcb*. For that reason, all the bytes of *srca* are qualified with the *shift_cnt_mask* to get only the valid bits of each byte, these qualified bytes then are replicated depending on the *OSIZE* getting five different combinations, one of these combinations is selected depending on the *OSIZE* and finally the bytes of the selected combination could be swizzled if it is a shift left to match the position of the swizzled bytes of *srca*. The diagram of the complete shift count is shown in Figure 2.61.

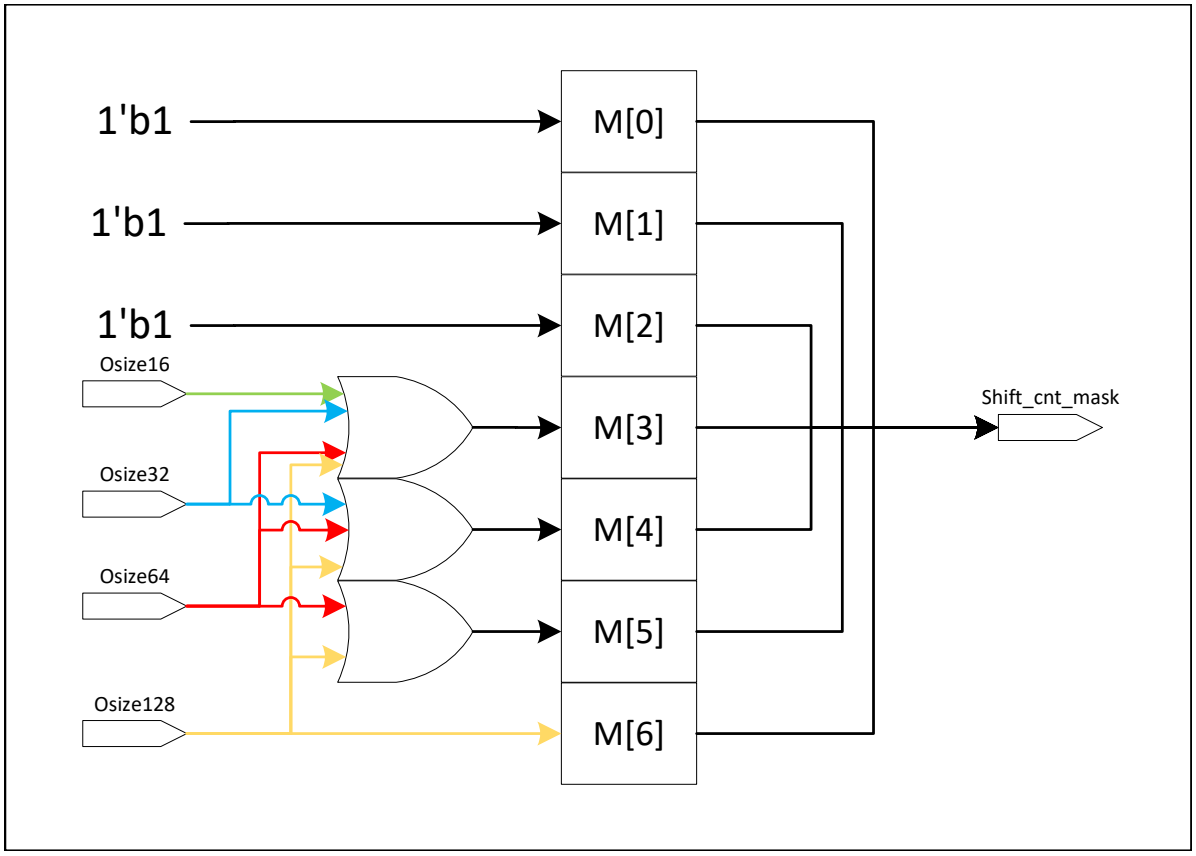


Figure 2.60: Shift count mask

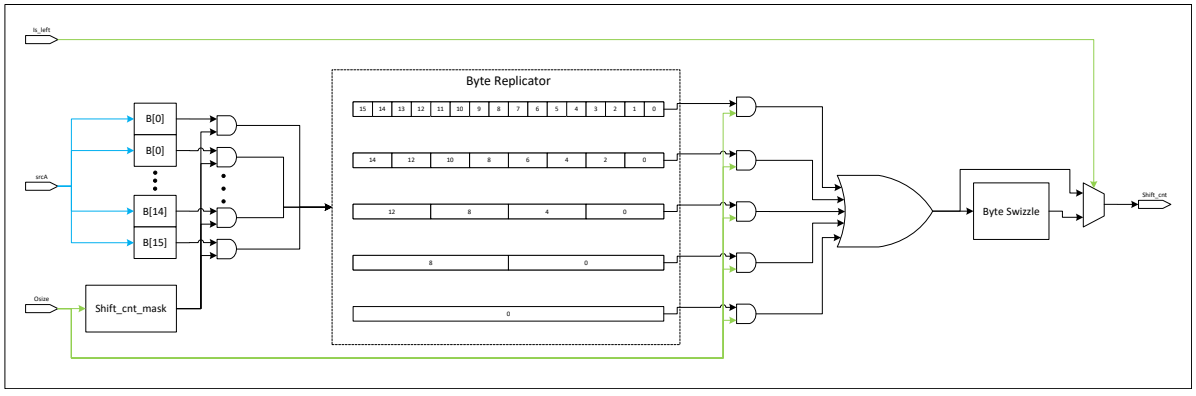


Figure 2.61: Shift count

Shifter Layer

The purpose of the *Shifter layers* is to shift the bytes of the elements. The bytes are shifted as a whole block, meaning that the internal bits of the bytes are not modified during this stage. Each layer can be seen like a normal layer of a Barrel Shifter, each layer is composed of a mux with a 1-bit selector that shifts the block one position to the right, the difference with the layer of a traditional Barrel Shifter is that instead of having 1-bit controlling all the multiplexors each multiplexor is controlled by an independent bit since depending on the *OSIZE* the shift count can be selected from a different byte of srca as declared in *Shift count*.

These layers also implement an extra multiplexor previous to the main multiplexor to implement arithmetic shifts, this extra multiplexor selects between the data of the previous layer or the sign-bit of the element, this multiplexor is controlled with the *merge* signal and the sign-bit is gated with *is_arith* to produce a zero for logical shifts.

The layers are instantiated from less significant to most significant, meaning that the first layer uses the bit[3] of the shift count and the bytes are shifted one position. In contrast, the last layer uses the bit[6] of the shift count and shifts the bytes eight positions in total. The structure of the layer is shown in Figure 2.62.

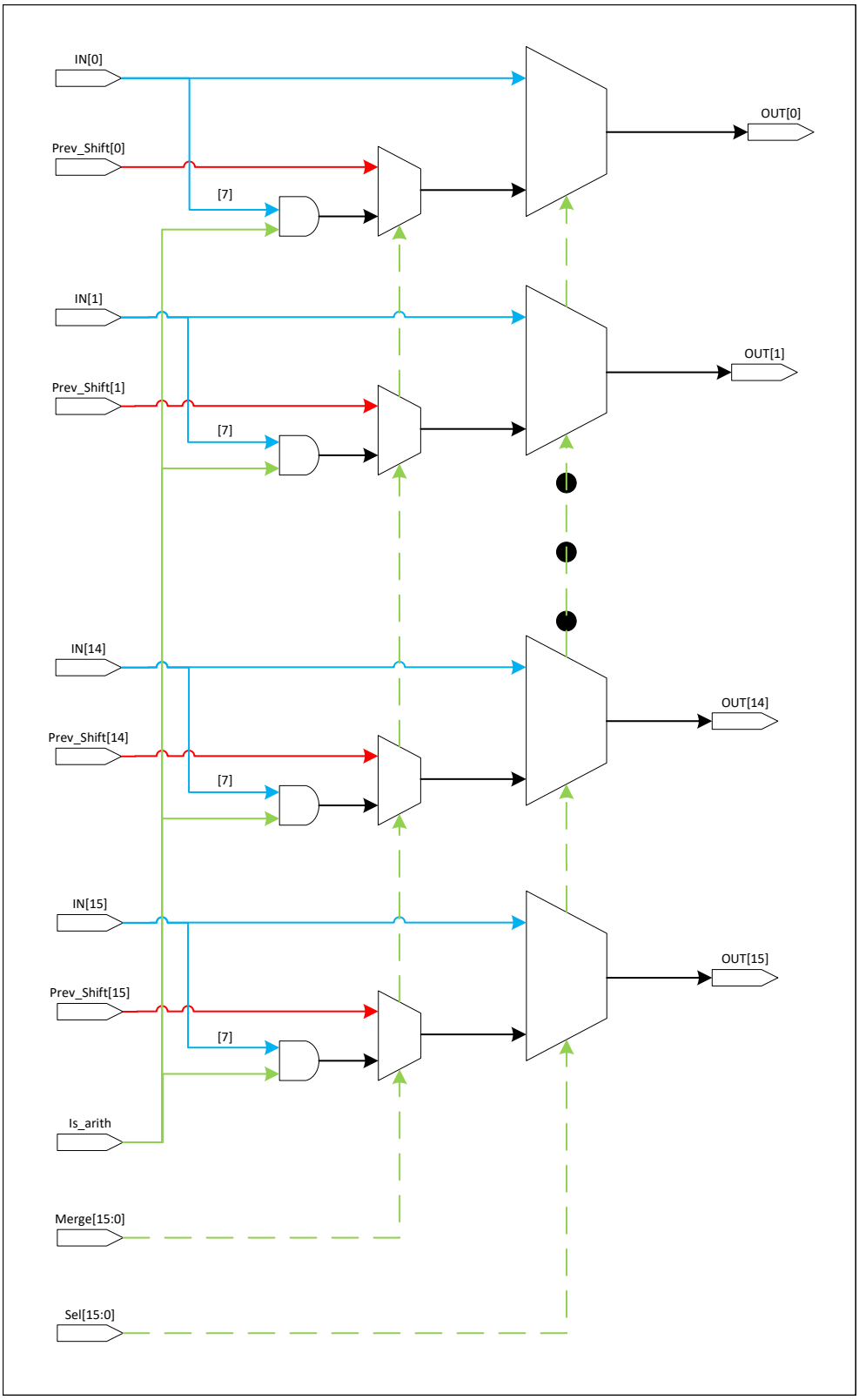


Figure 2.62: Shifter Layer

Byte Shifter

The Byte Shifter is a traditional 8-bit Barrel Shifter used to shift the internal bits of the bytes of the operand using the three less significant bits of the shift count. Sixteen Byte Shifters are instantiated, one per byte in the operand, each with its independent shift count. The Byte Shifters are implemented using the swizzle method to perform left and right shifts, in the case of the *shift_in*, the data from the previous shift stage, the multiplexor that selects between the original and the swizzled data can also select the sign-bit of the src to implement the arithmetic shifts. The diagram of the Byte Shifter is observed in Figure 2.63.

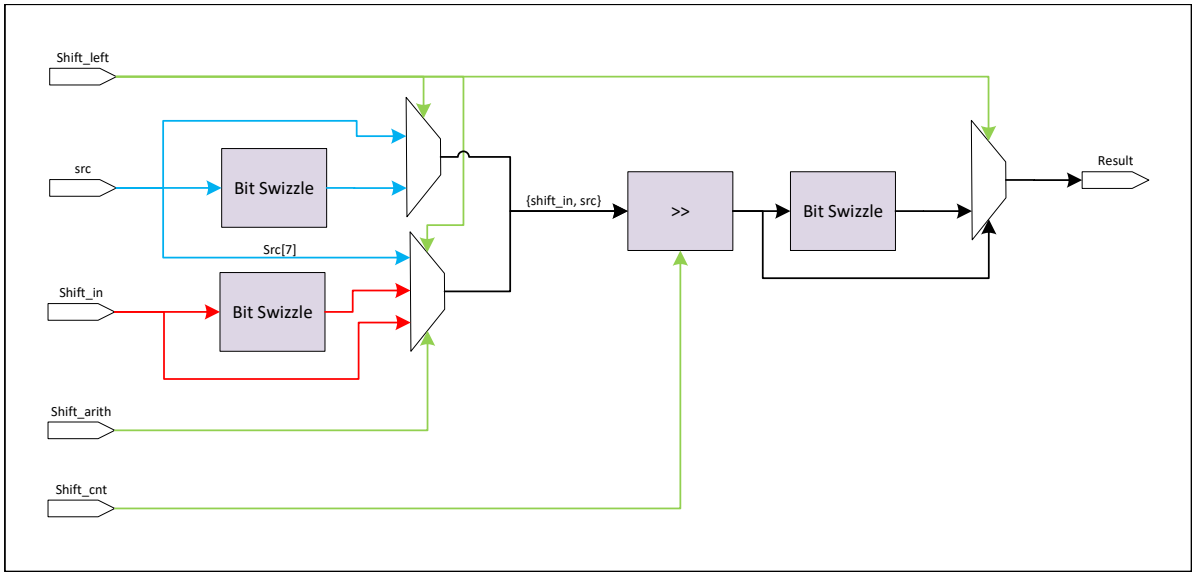


Figure 2.63: Byte Shifter

2.18 Arithmetic ALU

The *Arithmetic ALU* performs arithmetic operations such as addition, subtraction, multiplication and compare. It is composed of three execution blocks, the *Adder*, *Multiplier* and the *Extender*. As the other ALUs receive as inputs the two operands, the *OSIZE* and operation control signals and its outputs are the result and arithmetic flags, the result and flags are set to zero if the operation is not executed by this ALU. As shown in Figure 2.64 the operands are qualified internally in the execution blocks and the results of the blocks are ored together to get the final result.

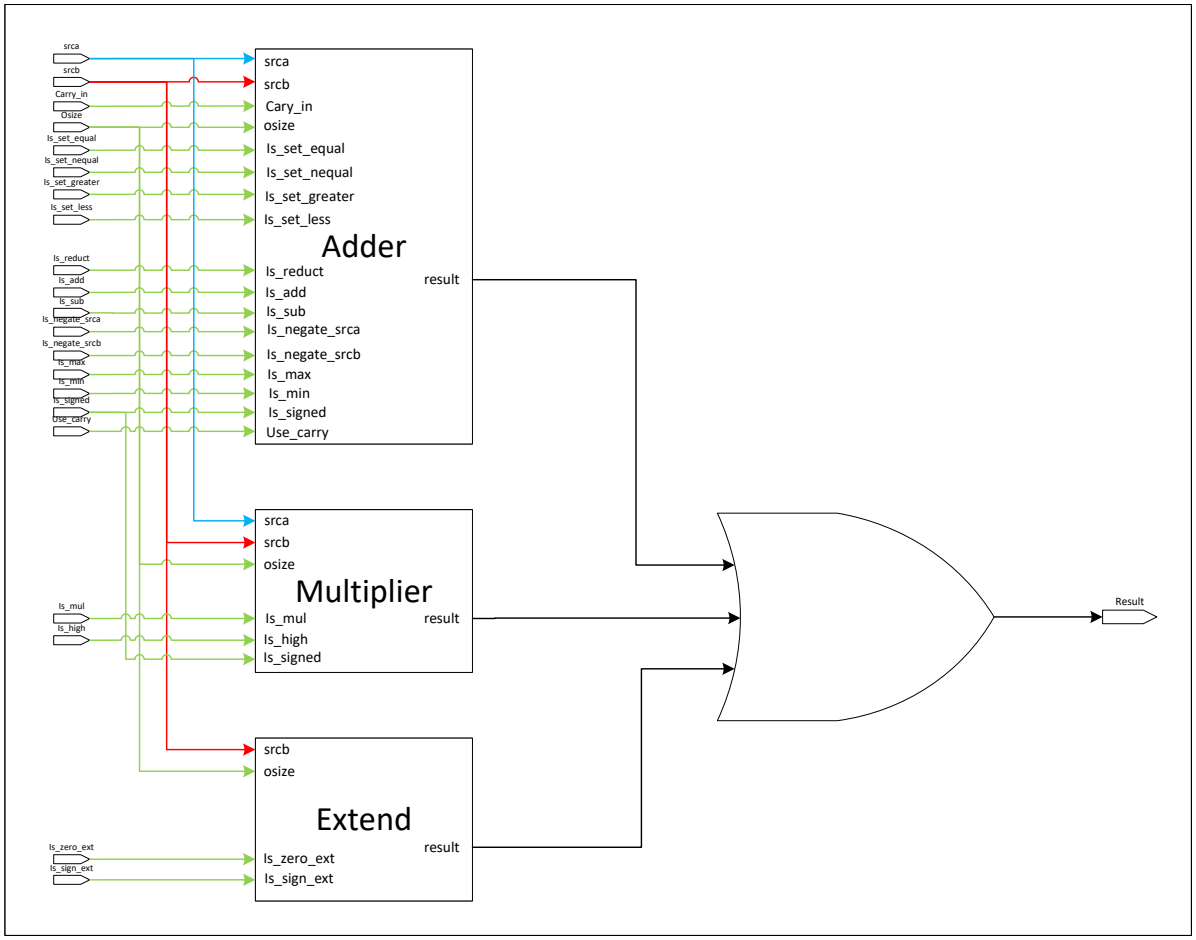


Figure 2.64: Arithmetic ALU

2.18.1 Vector Adder

The execution block *Vector Adder* implements the addition/subtraction, min/max, and compare operations, all the operations implemented in this block make use of sixteen 8-bit adders configured in different ways to perform the operations, each 8-bit adder can be implemented using any of the adders mentioned in the *Theoretical framework* section, each adder has its advantages and disadvantages and could perform differently on this engine, this work is mainly focused on the functional part of the engine, for that reason the *Ripple Carry Adder* was used in this project.

The structure of the Vector adder is shown in Figure 2.65. First, both sources are gated with *valid_adder* to produce a valid result only when the operation is executed in this execution block, then both sources are xored with *is_negate_srca* and *is_negate_srcb*, this is done to perform the subtraction using the same adder used by the additions by inverting the operand to be subtracted and setting to one the first *carry_in* of the element.

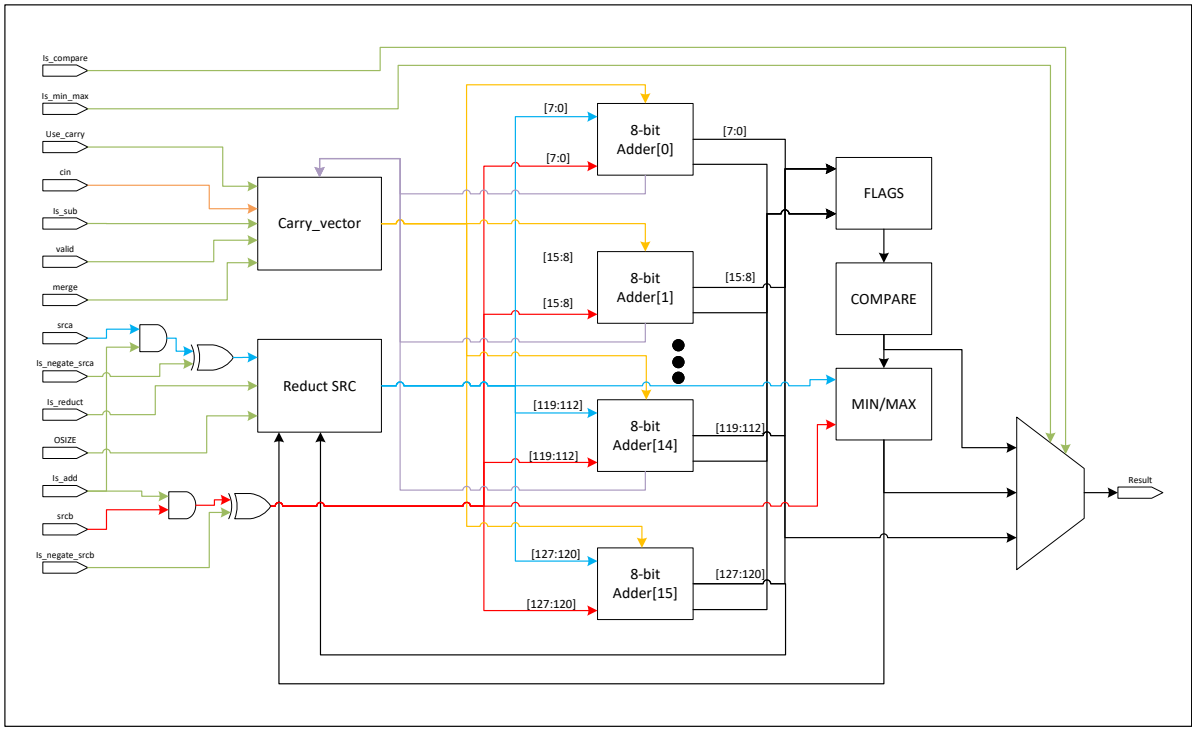


Figure 2.65: Vector adder

They *carry_ins* of the 8-bit adders are connected to a 16-bit *carry_in* vector, the vector is formed as shown in Figure 2.66. First one *merge_carry_in* vector is formed using the *carry_outs* of the 8-bit adders and qualifying the carry out with the *merge* signal of the operand, with this vector the adders can be connected in series to perform additions/subtractions of different width by connecting the *carry_out* of adder[N] to the *carry_in* of the adder[N+1]. The *merge_carry_in* vector is ored with the vector *sub_carry_in* that is formed with the signals *is_sub*, *src_valid* and *src_merge*, this vector indicates if a one needs to be added to the *carry_in* of the 8-bit adder to perform the subtraction as mentioned above. Finally the result of oring *merge_carry_in* and *sub_carry_in* is xored with the vector *use_carry_in*, this

vector indicates that the *Mask* vector is used as *carry_in*, with the xor, the bit obtained from *merge_carry_in* and *sub_carry_in* is flipped to perform the correct operation, for example, if the operations is a *VSBC* it flips the *carry_in* bit from one to zero to subtract the bit from the borrow(carry) vector. The vectors mentioned above are formed as shown in:

$$merge_carry_in[idx] = cout_adder[idx-1] \wedge merge[idx-1] \tag{2.59}$$

$$sub_carry_in[idx] = is_sub \wedge valid[idx] \wedge\ merge[idx-1] \tag{2.60}$$

$$use_carry_in[idx] = use_carry \wedge mask[idx] \wedge\ merge[idx] - 1 \tag{2.61}$$

$$cin_adder[idx] = (merge_carry_in[idx] \vee sub_carry_in[idx]) \oplus use_carry_in[idx] \tag{2.62}$$

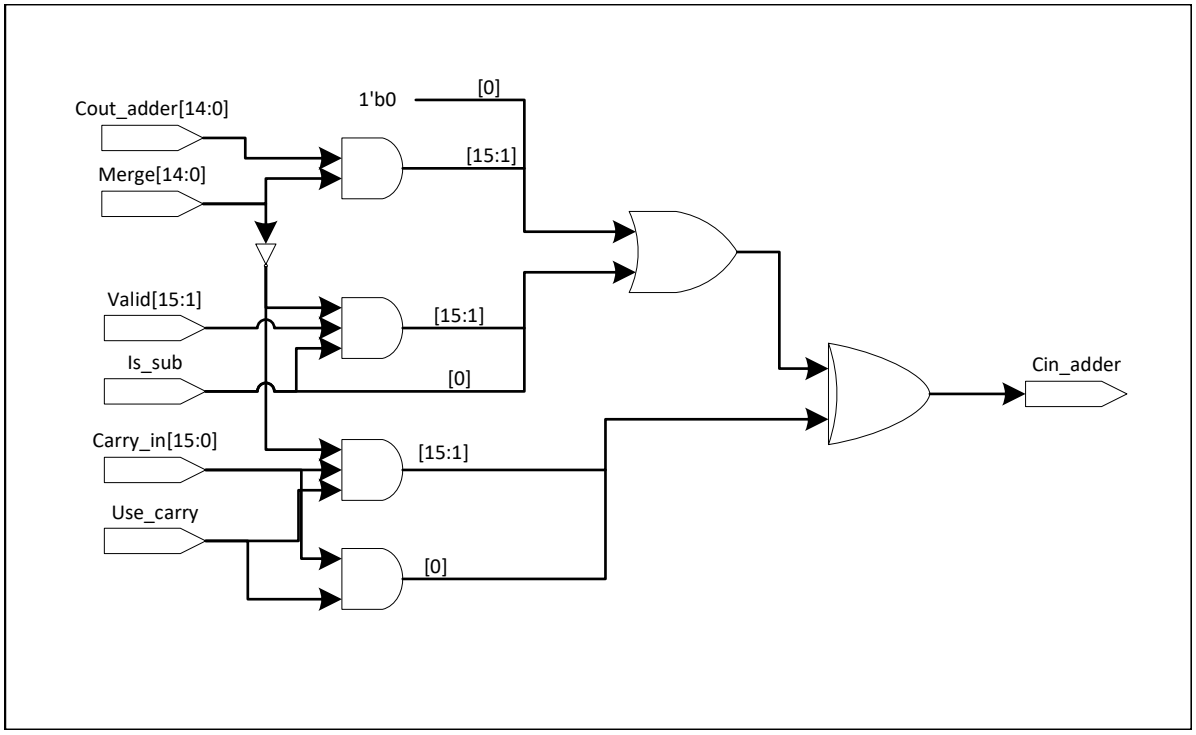


Figure 2.66: Adder carry_in vector

To perform the additions of different widths using only the sixteen 8-bit adders the *cin_adder* vector is used to interconnect the adders. The Figure 2.67 represents the interconnection of the adders to implement just the ADD operations, the *carry_out* of the adders are qualified with the *merge* vector, if *merge[N]* is one it means that the 8-bit adder[N] needs to be connected to the 8-bit adder[N+1] so the *carry_out[N]* is connected to *carry_in[N+1]*, if *merge[N]* is zero the *carry_in[N+1]* is set to zero to use the 8-bit adders as independent adders.

As mentioned above the subtraction is implemented by inverting the source to be subtracted and adding one to the *carry_in* of the 8-bit adder to perform the two's complement of the operand to be subtracted, the Figure 2.68 shows the hardware that is added to the adder

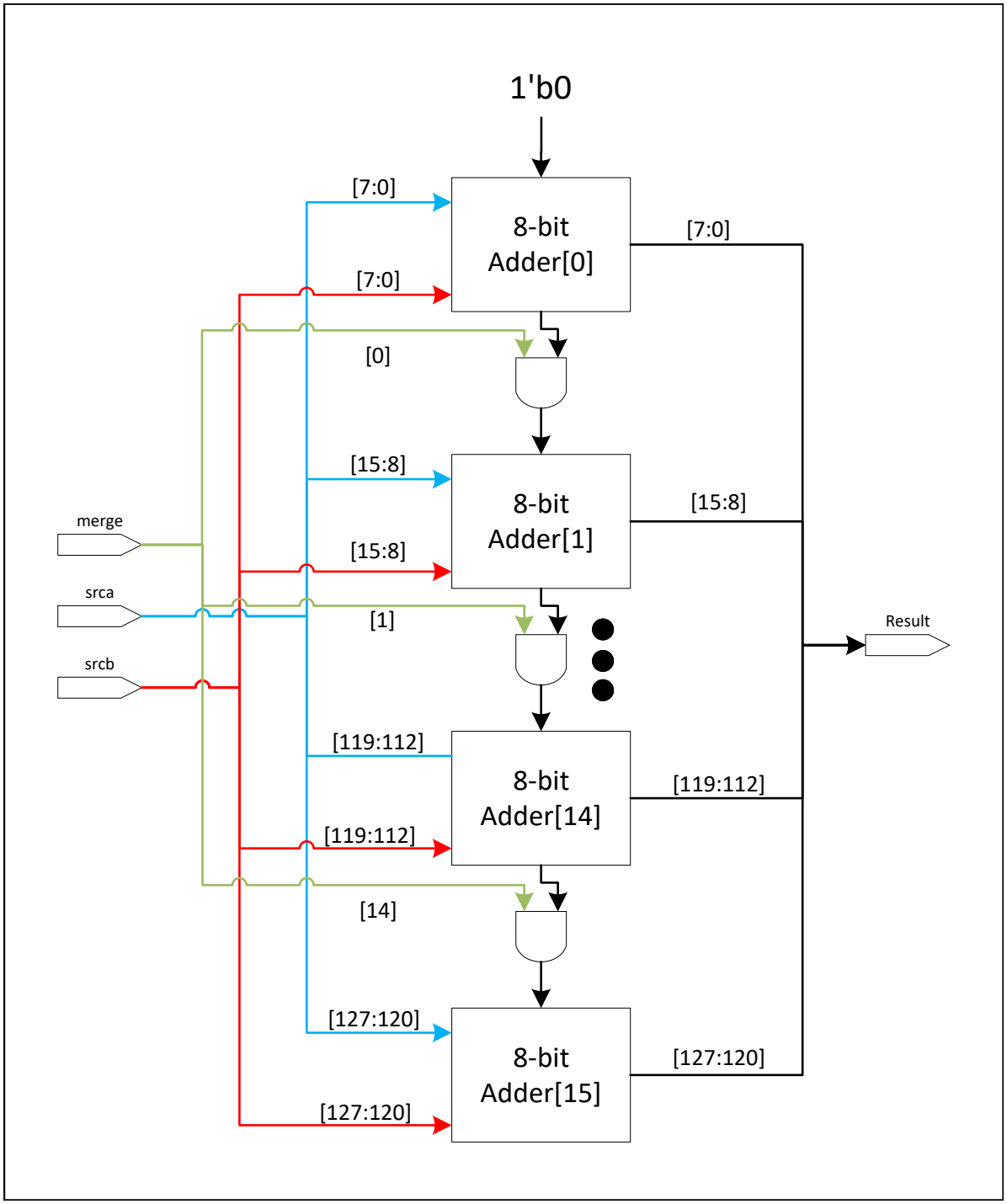


Figure 2.67: Simple adder

structure to implement the subtract operation. It is important to note that each C[N] block shown in 2.68 refers to a bit calculated by the *vector_carry_in* shown in Figure 2.66.

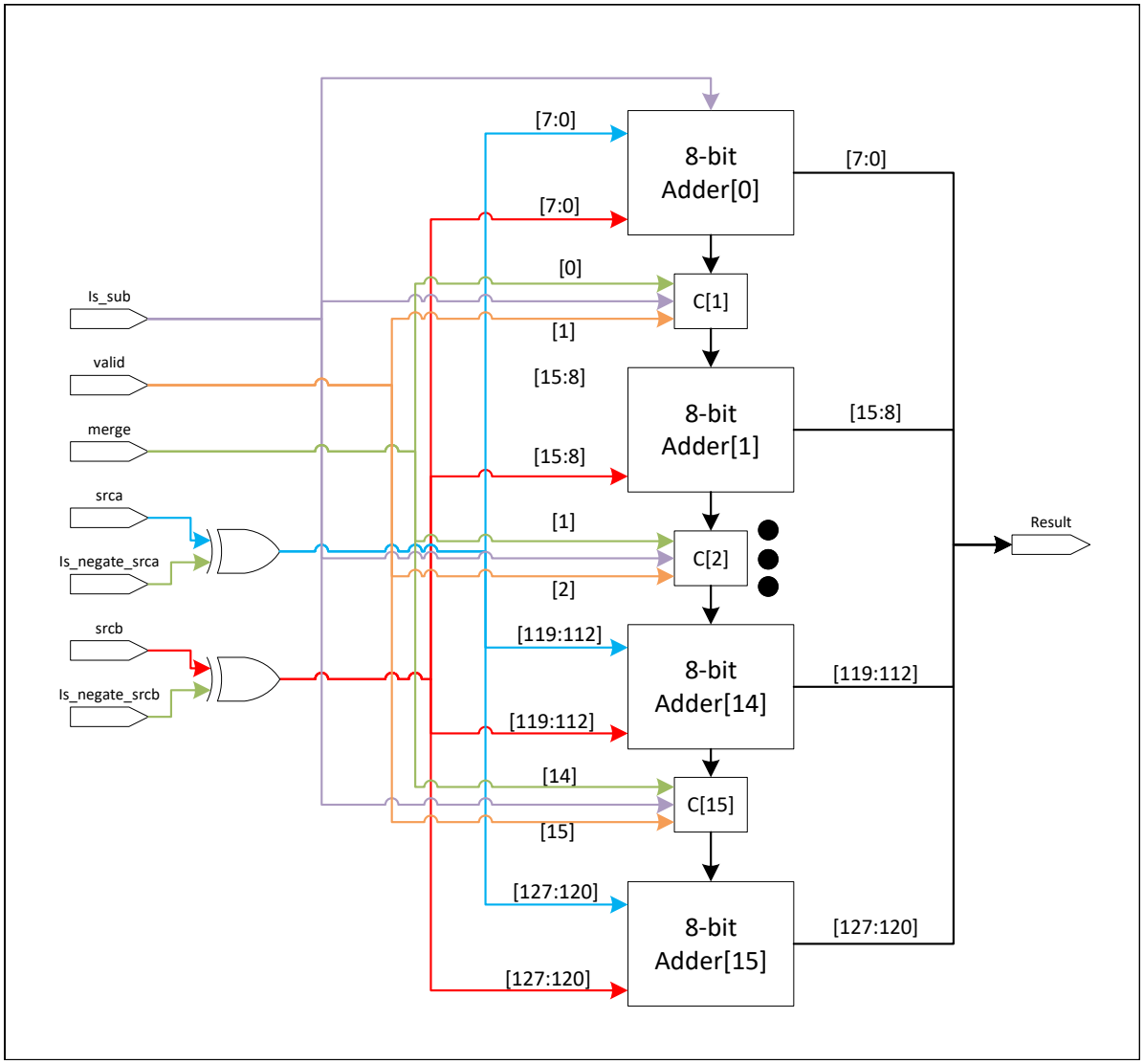


Figure 2.68: Adder with subtract

The *Reduction operations* are implemented using the *Reduction source* block, the Figure 2.69 shows the hardware needed to implement the *Reduction operations*, the *srca* is connected to the *Reduction source* along with the *results* of the 8-bits adder to get the correct source depending on the operation.

The flags are calculated using the outputs of the 8-bit adders and the *OSIZE* of the operation as shown in Figure 2.70. The *carry_flag* can be obtained directly from *carry_outs* of the 8-bit adders. The *zero_flag* is set to one if all the 8-bit adders of the element return as result zero and the flag is set in the most significant byte of the element, to get this flag five *zero_flags* vectors are calculated and the correct one is selected using the *OSIZE*. To calculate *overflow_flag* the *unsigned_overflow* and *signed_overflow* are calculated, these flags are calculated using the *carry_outs* of the 8-bit adders and the control signals. The flags are

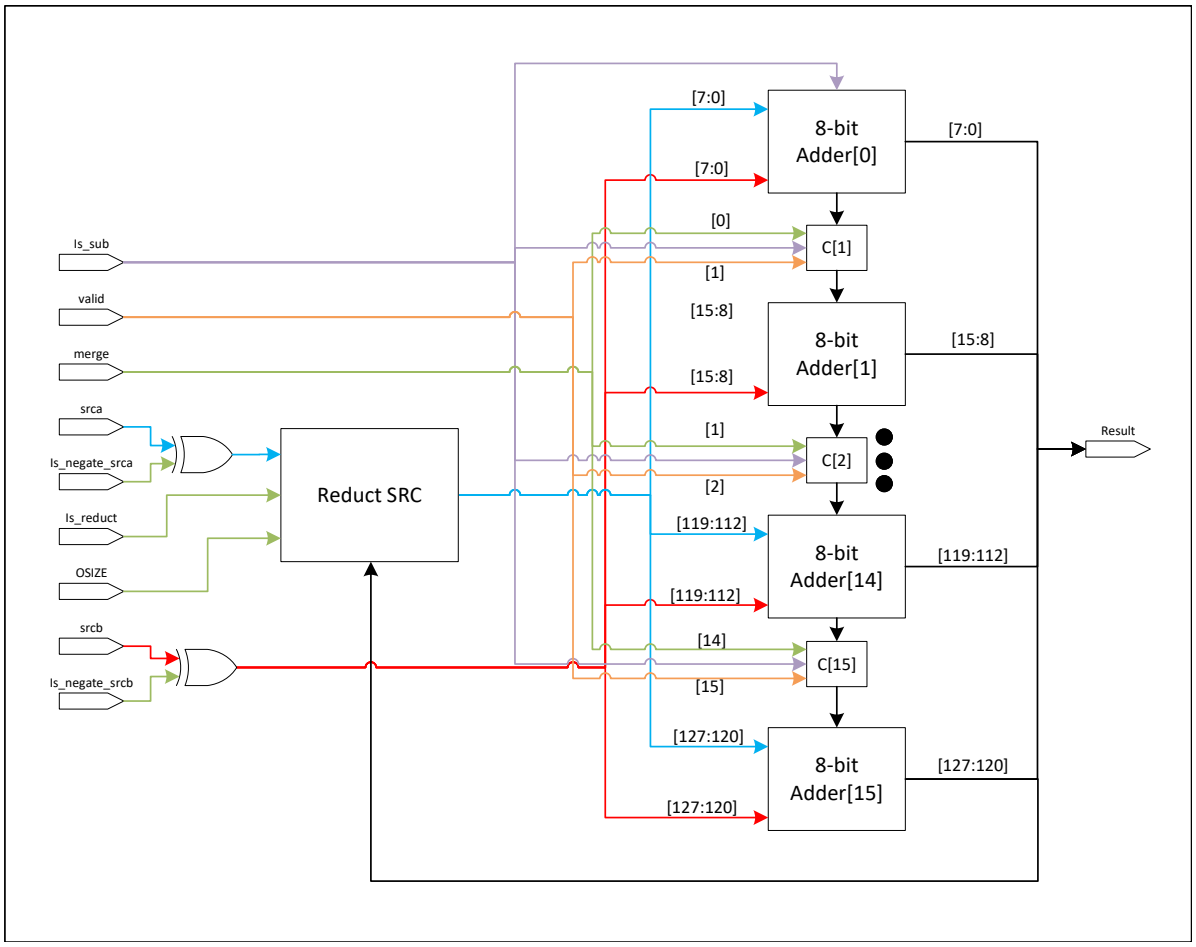


Figure 2.69: Adder with reduction

calculated as shown in:

$$carry_flag[idx] = cout_adder[idx] \tag{2.63}$$

$$signed_overflow[idx] = (cout_adder[idx] \oplus prev_cout_adder[idx]) \wedge is_signed \tag{2.64}$$

$$unsigned_overflow[idx] = (cout_adder[idx] \oplus is_sub) \wedge \ is_signed \tag{2.65}$$

$$overflow[idx] = unsigned_overflow[idx] \vee signed_overflow[idx] \tag{2.66}$$

$$zero[idx] = (result_element[idx] == 0) \tag{2.67}$$

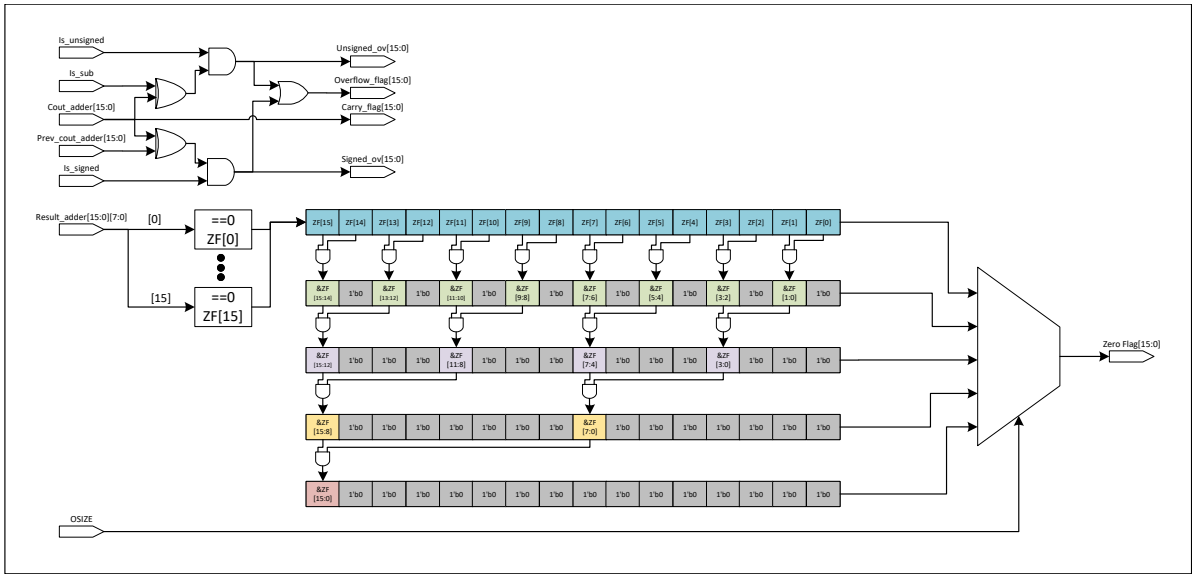


Figure 2.70: Adder flags

The compare instructions are executed by performing a subtraction between the two operands and reading the flags described above, the vectors *result_set_equal*, *result_set_nequal*, *result_set_less* and *result_set_greater* are computed and then they can be ored to form the different combinations of the compare instructions. Each of the compare vectors is formed by reading the flags per *OSIZE* and then selecting the correct *OSIZE* vector using *OSIZE* as selector as shown in Figure 2.71 and in:

$$set_equal[idx] = zero[idx] \tag{2.68}$$

$$set_nequal[idx] = \ zero[idx] \tag{2.69}$$

$$set_less_than_signed[idx] = (result_sign[idx] \oplus signed_overflow[idx]) \wedge is_signed \tag{2.70}$$

$$set_less_than_unsigned[idx] = unsigned_overflow[idx] \tag{2.71}$$

$$set_less_than[idx] = set_less_than_signed[idx] \vee set_less_than_unsigned[idx] \tag{2.72}$$

$$set_greater_than[idx] = \ (set_less_than[idx] \vee set_equal[idx]) \tag{2.73}$$

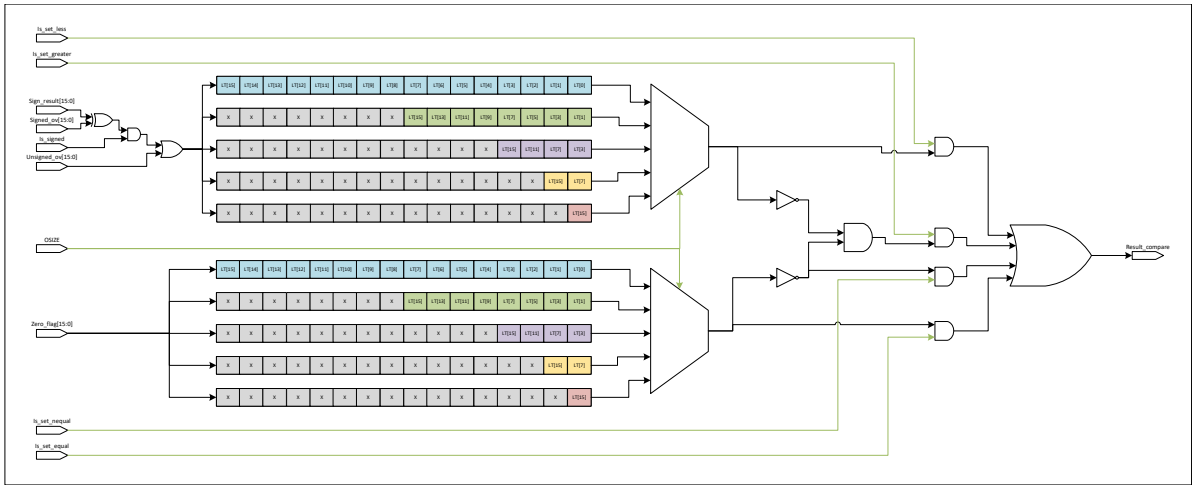


Figure 2.71: Adder compare

To implement the max/min operations as in the compare operations the operands are subtracted and the flags are read to get the minimum and maximum value, as shown in Figure 2.72 a multiplexor is used to select between *srca* and *srcb* using the flags as selector since the max/min result is selected from the sources and not from the adders result, an extra *Reduction Source* stage needs to be added after the one that is already instantiated to implement the max/min *Reduction operations*. The computation of the operation is shown in:

$$max_min_result[idx] = (is_max \oplus set_less_than[idx] \vee \ valid[idx])?srca[idx] : srcb[idx] \tag{2.74}$$

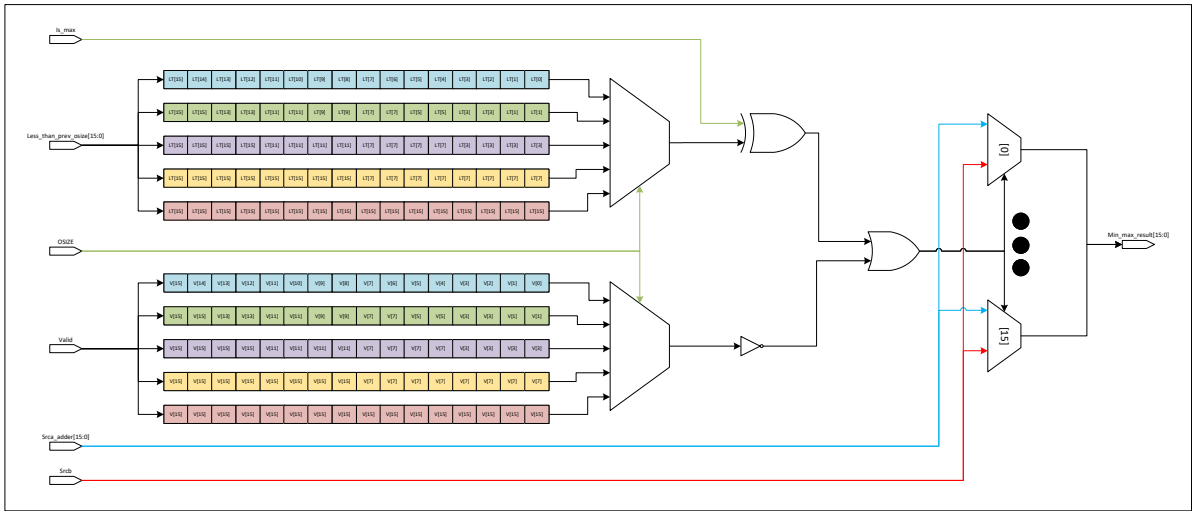


Figure 2.72: Adder Max/Min

2.18.2 Vector Multiplier

The *Vector Multiplier* implements the instructions *VMUL*, *VMULH* and *VMULHU*. These instructions multiply two N-bit signed or unsigned numbers and return the low or high part of the result. All the *OSIZEs* are supported, this means that multiple combinations of multiplications can be performed, for example, the instruction *VMUL_OSIZE8* executes sixteen 8-bit signed multiplications and returns the lower part of the results of those multiplications, while the instruction *VMULHU_OSIZE32* executes four 32-bit unsigned multiplications and returns the high part of the results.

The multipliers can not be connected in series to perform a bigger multiplication unlike the adders where two N-bit adders can be connected in series by connecting the carries to get a 2N-bit adder. For that reason the *Vedic Multiplier* is used to implement the multiply instructions, as explained in the *Vedic Multiplier* section a N-bit multiplier is built using four N/2-bit multipliers, using this structure five different multiplications are obtained, the **(N*N)** original multiplication, the **(N/2_low * N/2_low)**, **(N/2_high * N/2_low)**, **(N/2_low * N/2_high)** and **(N/2_high * N/2_high)**.

In the case of the *Vector Multiplier* three of the five multiplications performed by the *Vedic Multiplier* are relevant, the original **N*N**, **(N/2_low * N/2_low)** and **(N/2_high * N/2_high)** multiplications. For example, in a 16-bit multiplier where the width of each operand is two bytes, using the *Vedic Multiplier* the result of **(srca.word * srcb.word)** multiplication is obtained from the whole multiplier, but the result of **(srca.byte[0] * srcb.byte[0])** is obtained from the **(N/2_low * N/2_low)** multiplier and the result of **(srca.byte[1] * srcb.byte[1])** from the **(N/2_high * N/2_high)** multiplier, the three results can be obtained from the same multiplier without adding extra hardware and just adding a multiplexer at the end to select the correct result depending on the size of the operation.

The *Vedic Multiplier* can be built using other *Vedic Multiplier* as internal multipliers, using this recursive structure the *Vector Multiplier* is built by using one 128-bit *Vedic Multiplier*, that is built using four 64-bit *Vedic Multiplier*, each one using four 32-bit *Vedic Multiplier* that are built using four 16-bit *Vedic Multiplier*, in a similar way these are built using four 8-bit *Vedic Multiplier* which are built using four 4-bit *Vedic Multiplier*, the 4-bit multipliers does not instantiate other *Vedic Multipliers*, they are built using four 2-bit *Array Multipliers*.

Using this recursive structure all the valid *OSIZEs* results can be obtained by getting the results from the intermediate multipliers, the Figure 2.73 shows the multipliers from where the intermediate results are gotten. If a traditional *Array Multiplier* is used to implement the *Vector Multiplier* an individual multiplier would need to be used for each *OSIZE* since it does not produce useful intermediate results, it would require one 128-bit multiplier, two 64-bit multipliers, four 32-bit multipliers, eight 16-bit multipliers and sixteen 8-bit multipliers, the Figure 2.74 shows the structure that would be needed if an *Array Multiplier* is used and the Table 2.14 compares the number of elements used between the implementation with *Array Multipliers* and *Vedic Multiplier*.

By looking at the number of elements used by each implementation, it is observed that the *Vedic Multiplier* uses more resources, the total area consumed by each multiplier is discussed in the section *Multiplier Synthesis*, but as mentioned in the *Vedic Multiplier* section this multiplier reduces the propagation delay to get the result of the operation, being faster than the *Array Multiplier*, the speed comparison about both implementations is also discussed

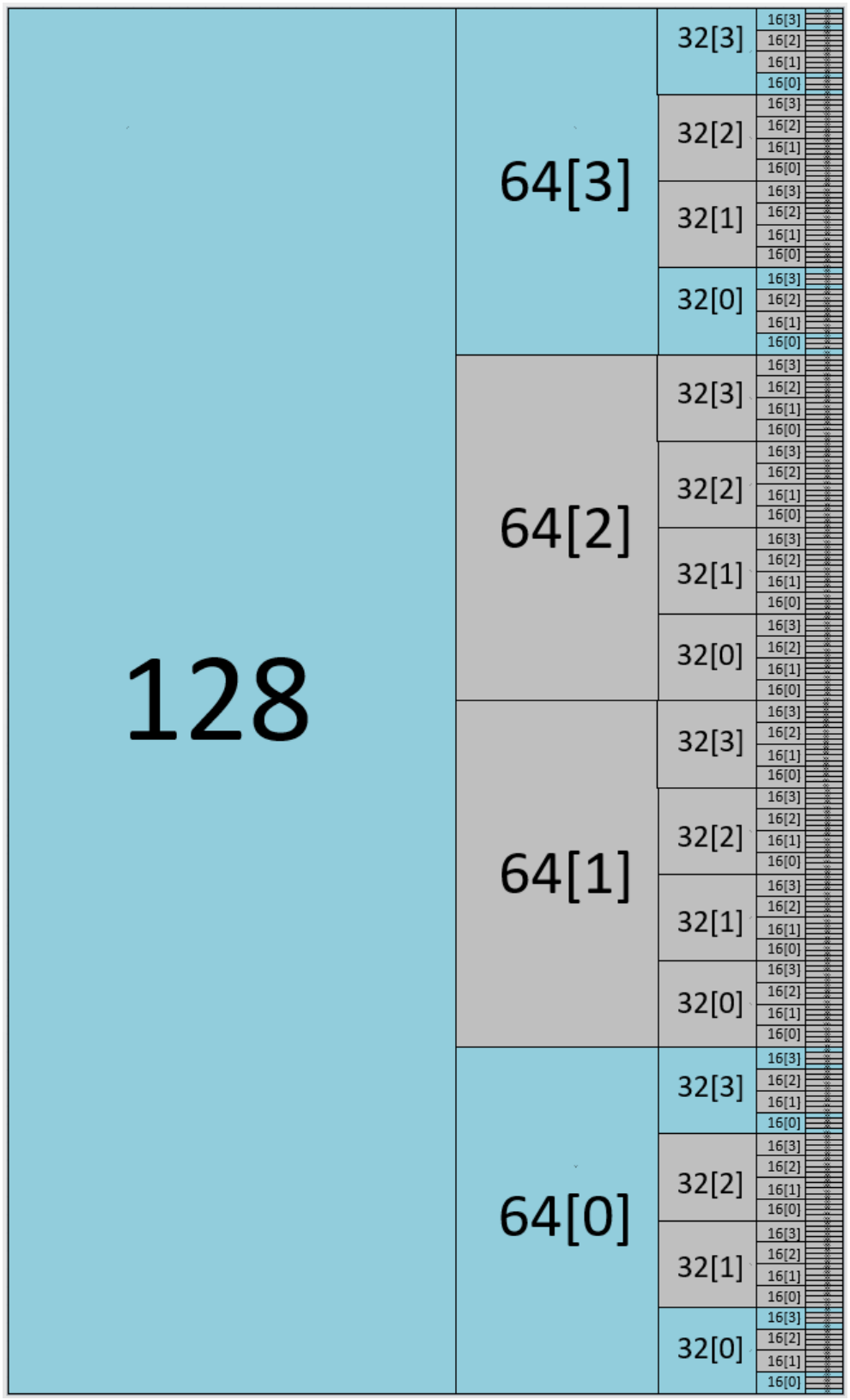


Figure 2.73: Vedic multiplier internal results

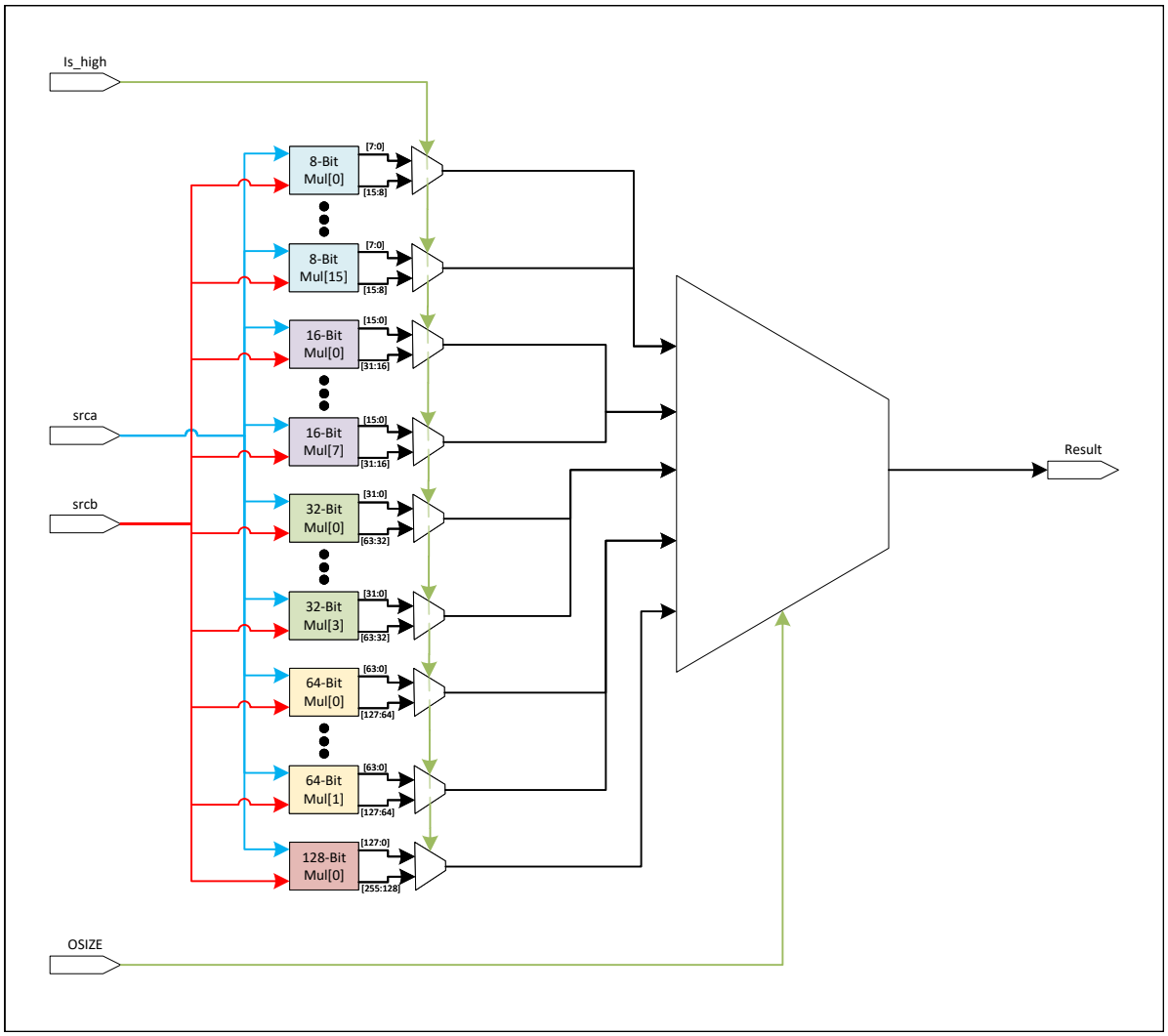


Figure 2.74: Vector multiplier(Array Multiplier)

Table 2.14: Vector multiplier Array vs Vedic

MAX_OSIZE	Array Gates	Vedic Gates	Δ Gates	%Gates
8	320	411	91	128%
16	2048	1835	-213	89%
32	9984	7723	-2261	77%
64	44032	31659	-12373	71%
128	185344	128171	-57173	69%

in *Multiplier Synthesis* section, due to the speed of the *Vedic Multiplier* and its simplicity to get the partial results it was selected to be used as base of the *Vector Multiplier*.

The *Vedic Multiplier* only performs unsigned multiplications, to implement the signed multiplication both sources are sent to a *Two's complement* block and a multiplexer selects between the original source or the source from the *Two's complement* block using the sign of the source as selector to select always a positive number that is sent to the *Vedic Multiplier*. The result of the multiplier is also sent to a *Two's complement* block and the sign of both sources is xored, if the result is one it means that the result is negative, this bit is used to select between the positive or the negative result of the multiplier. The diagram of the signed/unsigned *Vedic Multiplier* is shown in Figure 2.75.

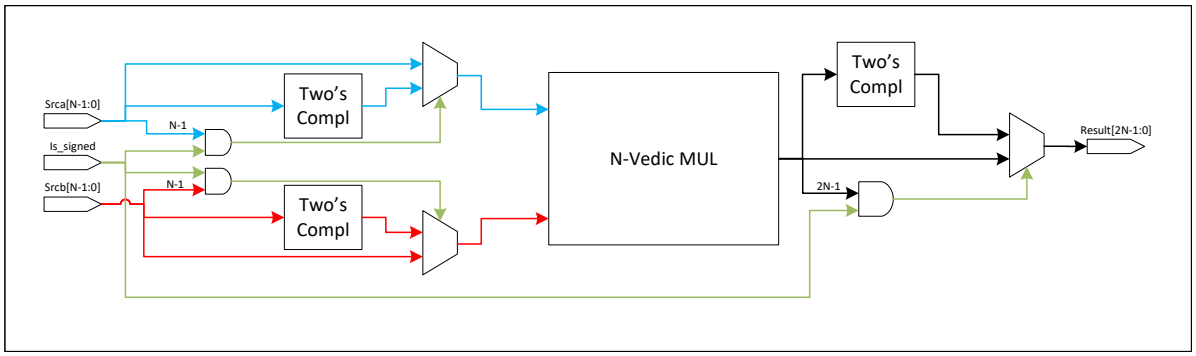


Figure 2.75: Vedic multiplier signed

Since the multiplier supports five different *OSIZEs* the data can not be sent to a 128-bit *Two's complement* block, it needs to be split into 8-bit blocks, for that reason a special *Two's complement* block shown in Figure 2.76 is used to convert the sources and the result. First, five different 16-bit vectors are formed with the signs of the elements one per valid *OSIZE* and one vector is selected, this vector is anded with *is_signed* to get if the elements need to be negated, the resultant vector is called *complement*. The *complement* vector is xored with the src to invert the src if it is negated, the *complement* vector is also anded with merge to deassert specific bits depending on the *OSIZE*, for example for *OSIZE_8* all bits are used, but if it is *OSIZE_32* only bits [0][4][8][12] are used and the remaining bits are set to zero. The *complement* vector anded with *OSIZE* is ored with the *c_out* from the incrementer[N-1] and the merge signal to indicate if the incrementer[N] is enabled as shown in (2.75). The results of the incrementers are the result of the modified *Two's complement* block. The *increment* signal is calculated using (2.75).

$$increment[idx] = \begin{cases} complement[0], \text{ if } idx == 0 \\ (complement[idx] \wedge !merge[idx-1]) \vee \\ (c_out[idx-1] \wedge merge[idx-1]), \text{otherwise} \end{cases} \tag{2.75}$$

The complete structure of the *Vector Multiplier* is shown in Figure 2.77. Just the *srca* is gated with *is_mul* since all multiplication by zero is zero, so the result is zero if it is not a mul operation. Both sources are connected to the modified *Two's complement* block, this block uses sixteen 8-bit incrementers to perform the *Two's complement* operation on the 128-bit

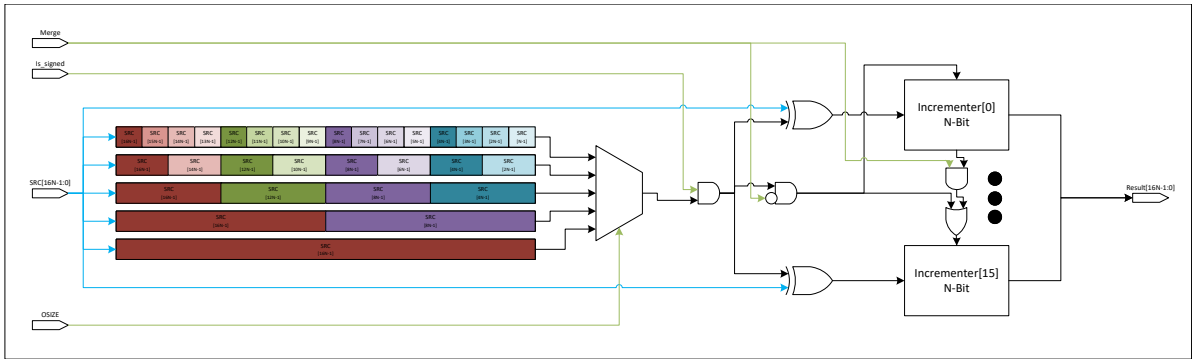


Figure 2.76: Vector Two's complement block

sources. The sources after the *Two's complement* block are connected to the 128-Bit *Vedic Multiplier* that produces five 256-bit results, one result is selected using *OSIZE* as a selector. The results are connected to another modified *Two's complement* block, but this block uses sixteen 16-bit incrementers to perform the *Two's complement* operation on the 256-bit result, then the result is arranged in five different vectors and one vector is selected using the *OSIZE*. Finally, the selected vector is connected to a final multiplexor that selects between the low or the high part of the result depending on *is_high*.

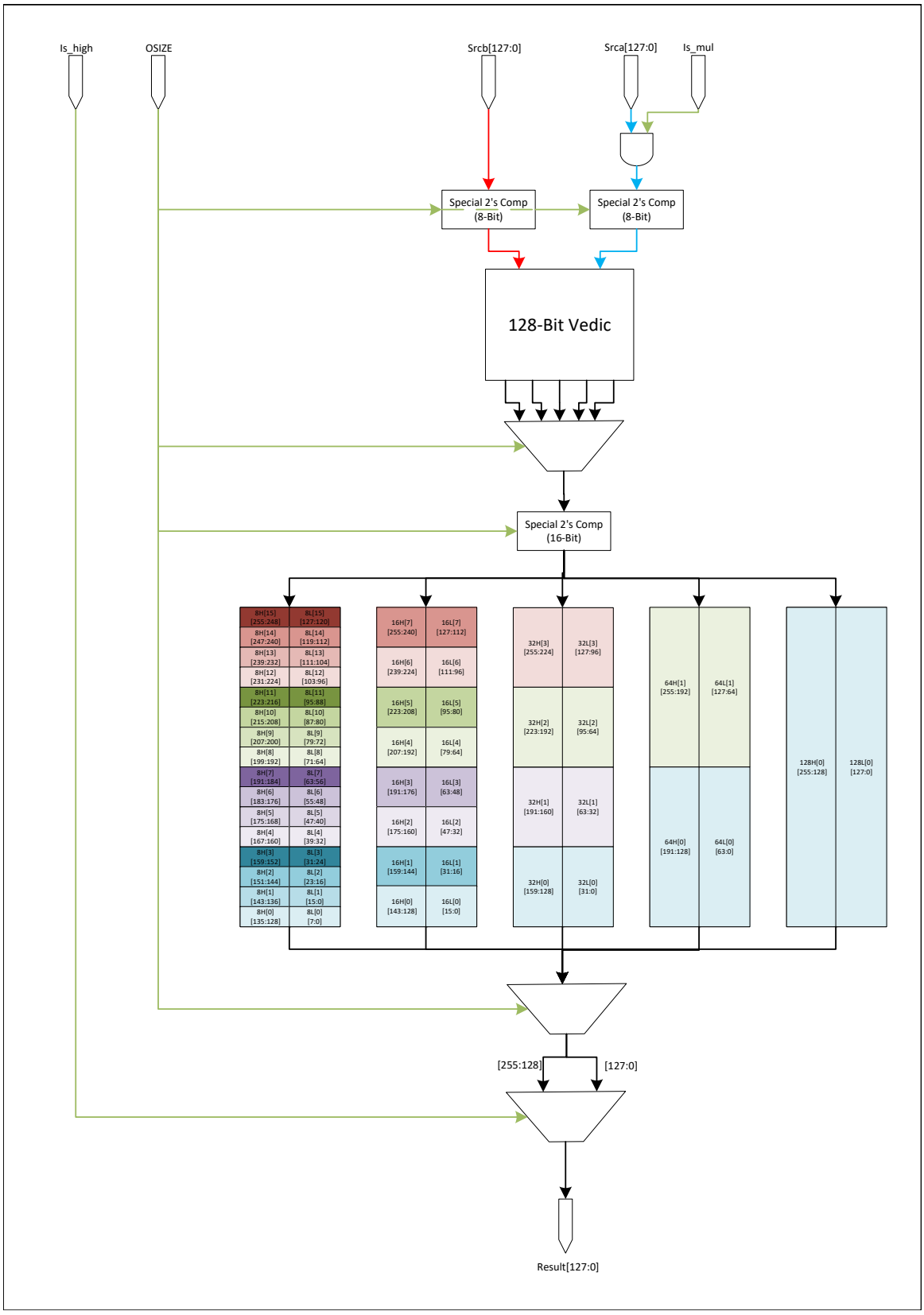


Figure 2.77: Vector Multiplier

2.18.3 Extender

As shown in Figure 2.78 the Extender is implemented with multiple extenders, one extender per each valid combination between *src_size* and *dst_size*.

First, the src is gated with *is_sign_ext* or *is_zero_ext* to qualify the operation and only produce a result if it is an extend operation. Then the source gated is gated again with *src_size* and *dst_size* to enable the correct extender. All extenders are connected to *is_sign_ext* to extend the source with zeroes or the source's sign depending on the control signal. Finally, the results of all the extenders are ored together to get the final result.

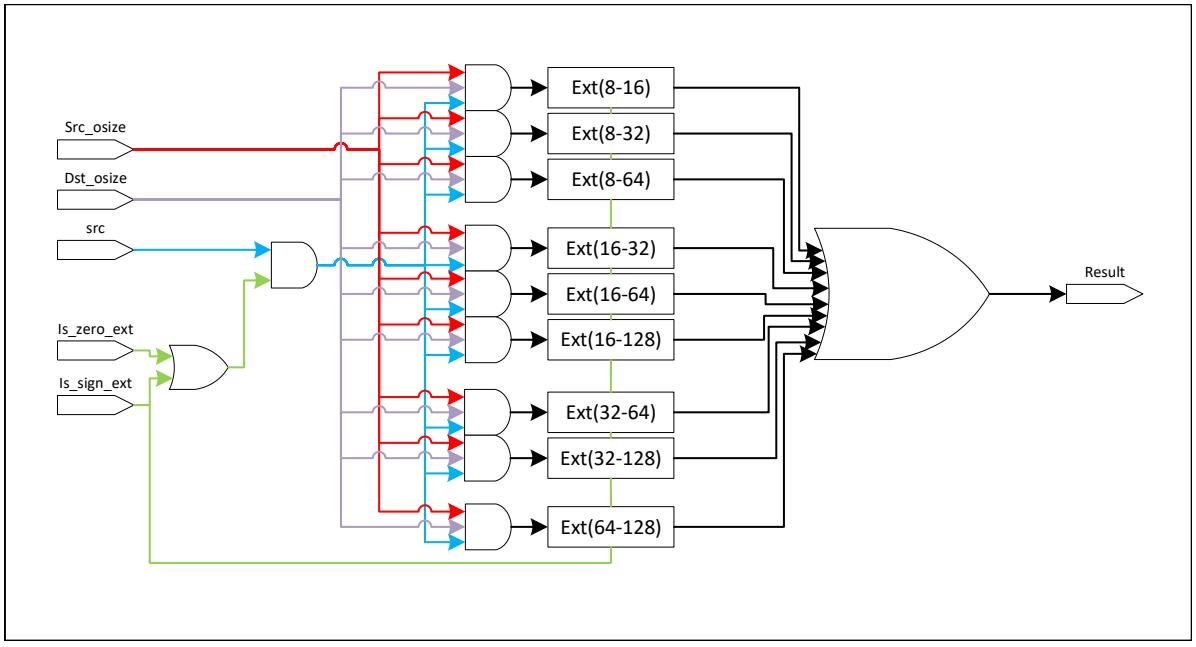


Figure 2.78: sign/zero extend

2.19 Permutation ALU

The Permutation ALU implements operations to transfer data between the integer Register File and the Vector Register File. The ALU is composed of two main blocks, Vector to Integer that transforms vector data to integer data and Integer to Vector that transforms integer data to vector data. The structure of the ALU is shown in Figure 2.79.

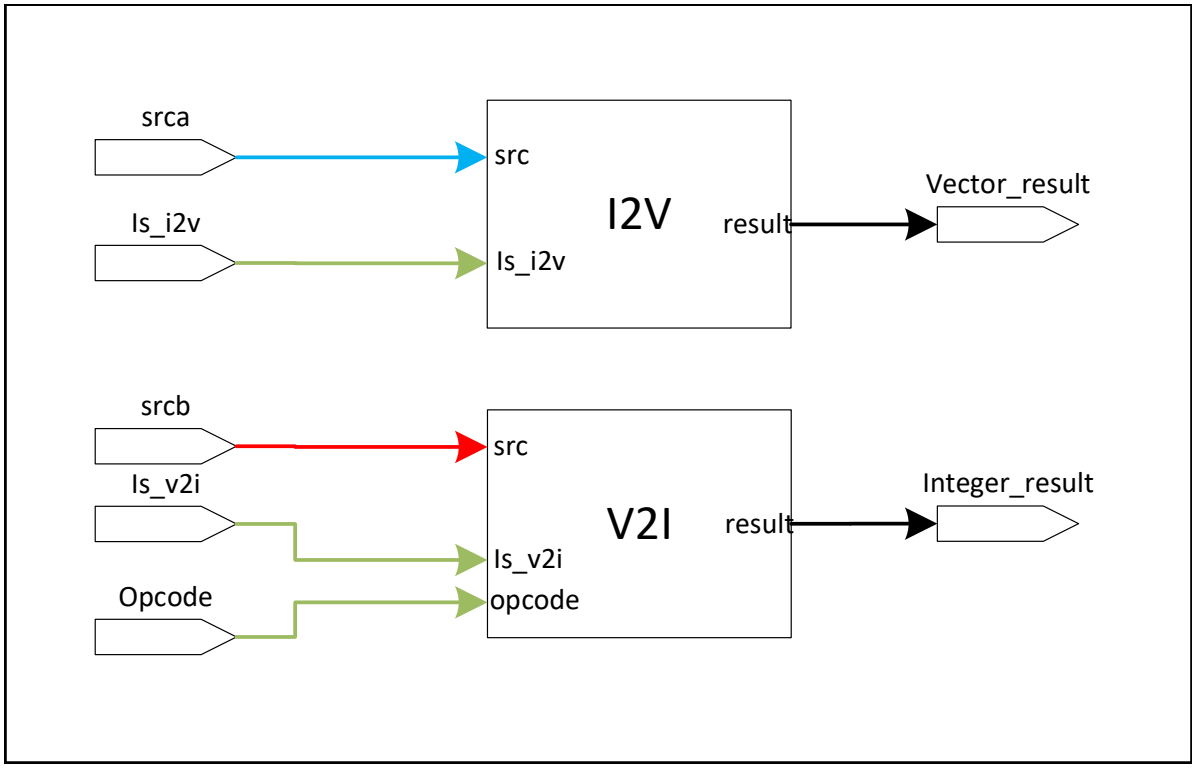


Figure 2.79: Permutation ALU

2.19.1 Vector to Integer

The *Vector to Integer* block transforms Vector data from the *Vector Register File* to Integer data to be stored in the Integer Register File.

It is built using two sign extenders, one to extend 8-bits to 32-bits and the second one to extend 16-bits to 32-bits. First, the source is gated with *is_v2i* to make sure that the result is only produced if it is a Vector to Integer operation, then the gated source goes to the sign extenders and finally, the result is selected with a multiplexor depending on the *OSIZE*. If the operation is *OSIZE_8* the result comes from the first extender, 8-bits to 32-bits. If the operation is *OSIZE_16* the result comes from the second extender, 16-bits to 32-bits. If the operation is greater than *OSIZE_16* the result comes from the lower 32 bits of the source without an extender.

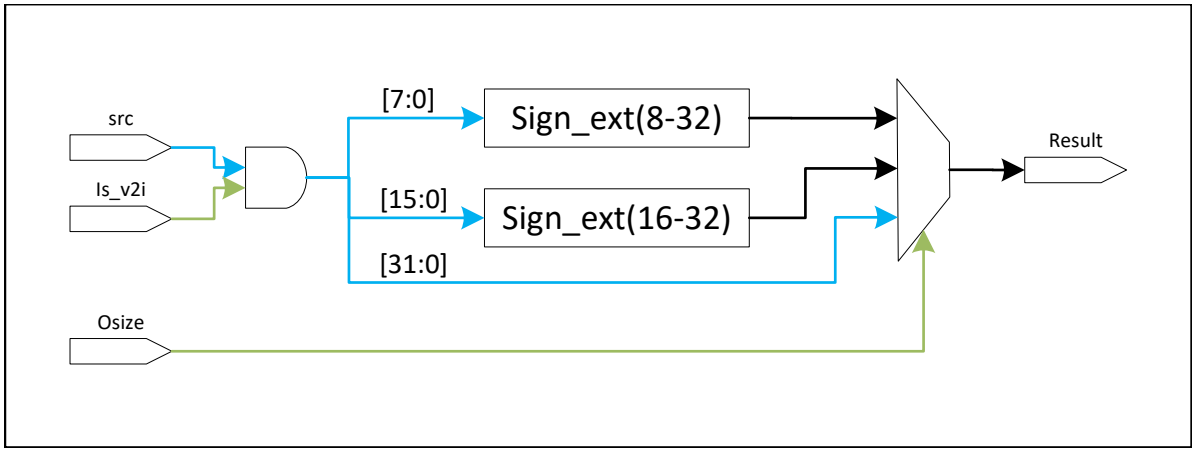


Figure 2.80: Vector to Integer

2.19.2 Integer to Vector

The main purpose of the *Integer to Vector* block is to transform Integer data from the Integer Register File to vector data to be stored in the *Vector Register File*.

This block would require to sign extend the integer data to the required size depending on the operation *OSIZE*. But since this extension is already done in the Bypass Unit this block only needs to qualify the source with *is_i2v* to produce a valid result if it is an Integer to Vector operation.

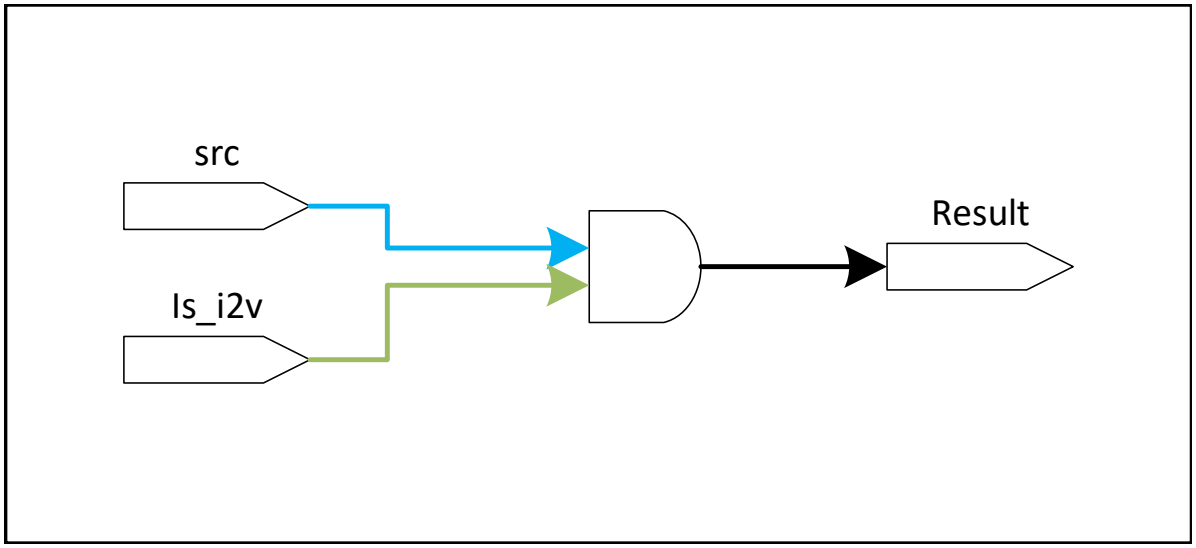


Figure 2.81: Integer to Vector

2.20 Result Swizzle

The *Result Swizzle* block is used to accommodate the result in the correct order to be written to the *VRF*. This block is needed to complete the execution of the *Reduction Operations*.

Architecturally the *Reduction Operations* write the result in the element[0] of the *VRF*, but physically the *Execution Unit* produces the result on the last element of the result bus, for that reason the result needs to be reordered.

Depending on the *OSIZE* the result can be reordered in five different ways, one per valid *OSIZE*, the result is ordered in the five ways and the correct format is selected using *OSIZE* as shown in Figures 2.82 and 2.83. Both Figures represent the same hardware the only difference is the visual representation, in Figure 2.82 the blocks are represented by *OSIZE* to visualize how the different *OSIZEs* are arranged, while in the Figure 2.83 the blocks are represented by Byte to visualize the different position of the bytes. If the operation is *OSIZE_128* or is not a *Reduction Operation* the option at the bottom is selected that corresponds to the original arrangement of the source.

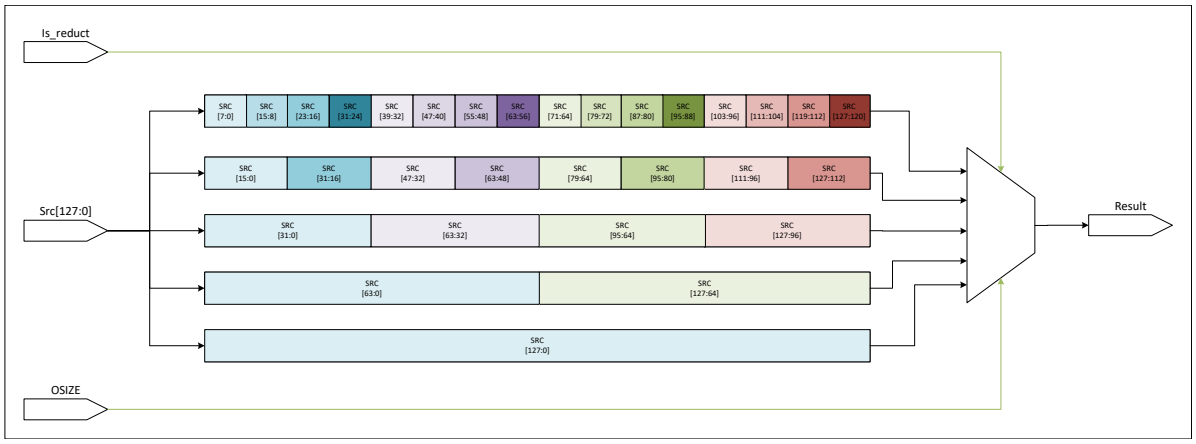


Figure 2.82: Result swizzle (Block)

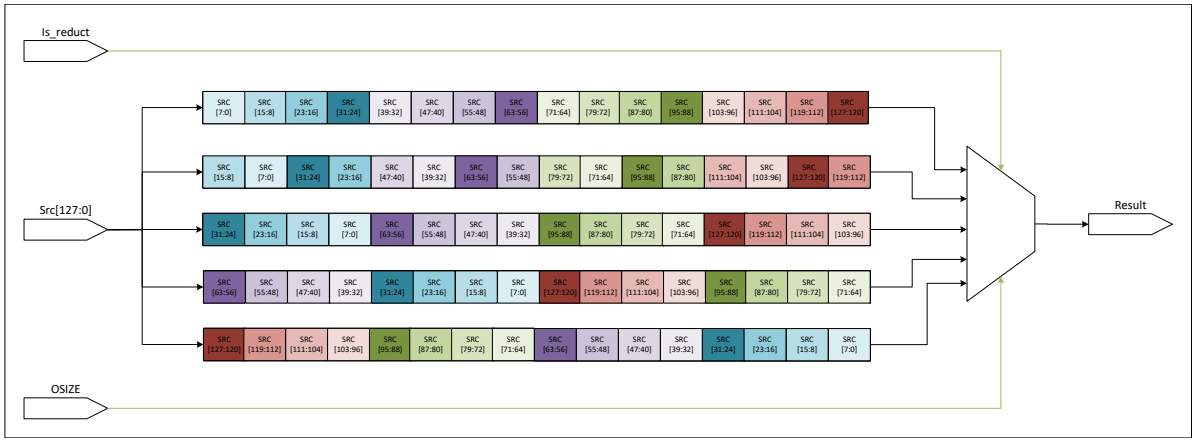


Figure 2.83: Result swizzle (Byte)

Chapter 3

Validation

Functional validation is a critical stage in the digital design flow, as it ensures that the *RTL* description of the hardware behaves as intended and meets the functional requirements defined during the design specification phase. This chapter presents the validation process applied to the vector execution unit developed in this project.

Since the vector engine implements a subset of vector instructions including arithmetic, logical, comparison, and permutation operations, the focus of the validation effort is to verify the correct functionality of each supported instruction under a variety of input conditions and operational scenarios. This includes testing different vector lengths, masking configurations, and edge cases to ensure robust and reliable behavior.

To perform this validation, *UVM* methodology is used to build a modular and reusable testbench environment. In this chapter, the structure of the verification environment is described, along with the key components such as drivers, monitors, scoreboards, and sequences. Several representative test cases are also discussed, illustrating how specific functionalities of the vector unit were exercised and verified.

Finally, a detailed coverage analysis is conducted using the data collected during simulation. This analysis includes only functional coverage metrics to ensure that all relevant parts of the RTL code were exercised during testing. The goal of this analysis is to provide a quantitative measure of the completeness of the verification effort and to confirm that all functional areas of the circuit have been thoroughly tested.

3.1 Base Components

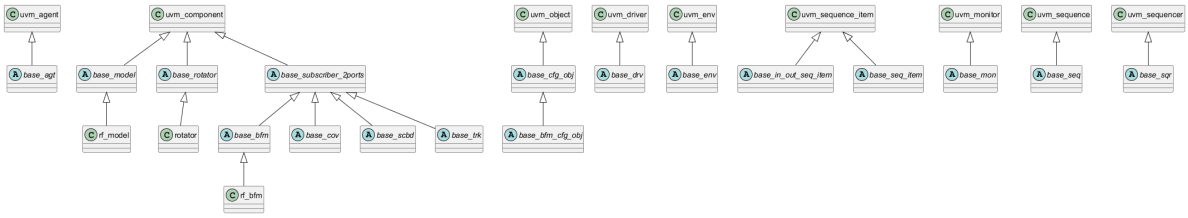


Figure 3.1: Base Components UML

All the *UVM* base components are shown in Figure 3.1, these components are the base of all the components used to verify the *RISCV Vector engine*. The base components are generic components that can be used in any project, they are parametrizable classes that can be specialized to validate different circuits. Each one of the elements is described in the following sections.

3.1.1 Base Configuration Object

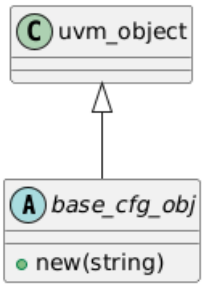


Figure 3.2: Base Configuration object UML

The *Base Configuration Object* or *base_cfg_obj* is a virtual class that extends from *uvm_object*. The purpose of this object is to hold the different configuration knobs used by the different verification components. This class does not implement any method nor attribute, the configuration knobs should be added by child classes.

3.1.2 Base BFM

The *Base BFM* is the base class used to implement a *BFM* and extends from *Base Subscriber Two Ports*. This is a parametrizable class that can be specialized, as parameter it receives the type of *Base Sequence Item*, *Base Sequence*, *Base Sequencer*, *BFM configuration object* and *Base Model* to be used.

The *Configuration object* used by this class is a *BFM Configuration Object* that extends from *Base Configuration object*. This object adds three configuration Knobs used to control the behavior of the *BFM* and declares as pure virtual function the method *check_cfg()*, this method is called by the *BFM* during the *build phase* and needs to be overridden by the child class to check that the configuration of the *BFM* is valid, if it is not an UVM fatal is raised. The description of the attributes and methods is shown in Table 3.1.

The main purpose of the *Base BFM* is to simulate a component and send *transactions* to the RTL or another component. It makes use of the *sequence* and *sequencer* to communicate with the *driver* and instantiates a *Model* that is used to model the behavior of the component, process the data and produce the *transactions*.

The *Base BFM* implements some methods to control the flow of the *BFM* as described in Table 3.2. The *BFM* is connected to the *monitor* via an analysis port, this port wakes up the *BFM* to process the data and generates new *transactions* to be sent to the driver as shown in Figure 3.4.

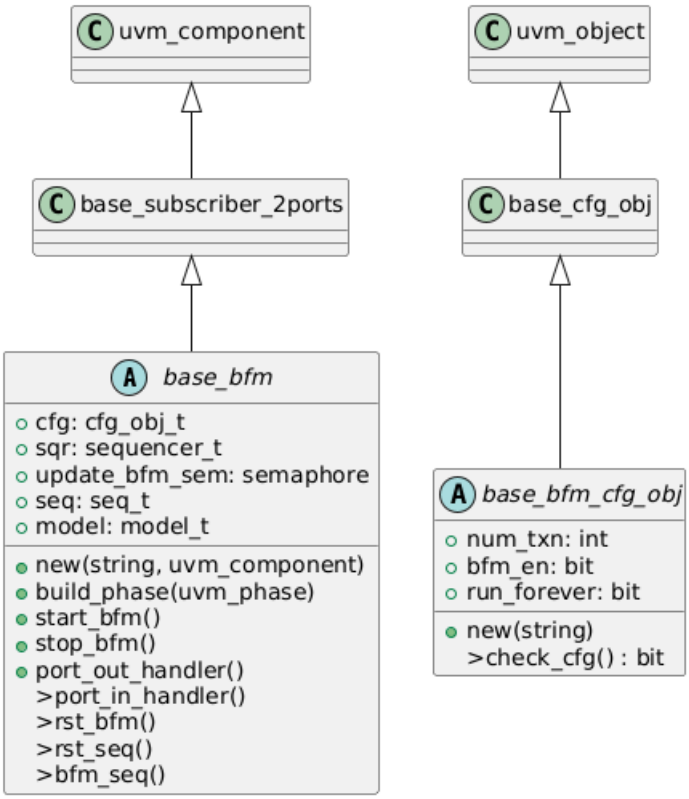


Figure 3.3: Base BFM UML

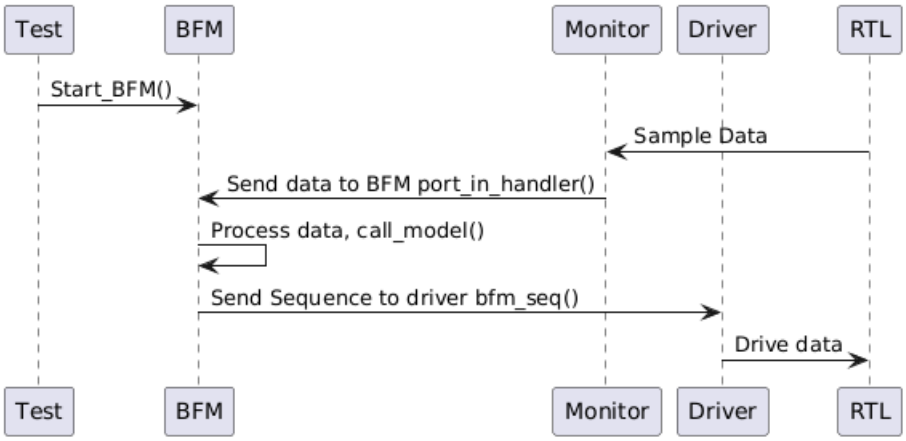


Figure 3.4: Base BFM sequence diagram

Table 3.1: Base BFM configuration Object methods and attributes

Attribute/ Method	Description
int num_txn	Indicates the number of transactions to be sent before finishing the test. This knob is only used if run_forever is set to 1'b0 If this knob is used the BFM raises an objection before sending the first transaction and drops the objection once all the transactions have been sent. Don't wait for the completion of the last transaction
bit bfm_en	Bit used to enable/start the BFM. The initial value of this bit is 1'b0, the BFM waits until this bit is 1'b1 to start sending the transactions. The bit can be set to 1'b0 at any time to pause the BFM, the BFM waits until the bit is 1'b1 again to continue sending transactions
bit run_forever	This bit is used to indicate if the BFM runs forever instead of sending a specific number of transactions, if this bit is 1'b1 num_txn is ignored. If this mode of operation is used the BFM does not raise nor drop objections, the test is responsible for the control of the objections
new()	Constructor of the class
bit check_cfg()	This function is called by the BFM during the build phase and is used to make sure that the configuration of the BFM is valid. If it is valid returns 1'b1, if it is not returns 1'b0. It is a virtual function that needs to be overridden by the child class

Table 3.2: Base BFM methods and attributes

Attribute/ Method	**Description**
cfg_t cfg	Configuration object that holds the configuration knobs of the BFM
sequencer_t sqr	Sequencer used to communicate with the Driver
semaphore update_bfm_sem	Semaphore used to sync the BFM processes
seq_t seq	Sequence to be sent to the Driver
model_t model	Model used to process the data and generate the transactions
new()	Constructor of the class
build_phase	Gets the configuration object from the database, and checks if the configuration is valid. Builds the sequencer, sequence, model and semaphore to be used
start_bfm()	Enables/Starts the BFM
stop_bfm()	Disables/Pauses the BFM
port_out_handler()	Handler of the analysis_port connected to the monitor port that reads the outputs of the RTL, this method may or may not be used by the BFM
port_in_handler()	Handler of the analysis_port connected to the monitor port that reads the inputs of the RTL. In this method the data that is sent to the RTL is also sent to the BFM to call the model if necessary and generate a new transaction in response
rst_bfm()	Resets the internal data of the BFM
rst_seq()	Generates a reset sequence to be sent to the RTL
bfm_seq()	Generates a sequence to be sent to the RTL with the data provided by the model

3.1.3 Base Subscriber Two Ports

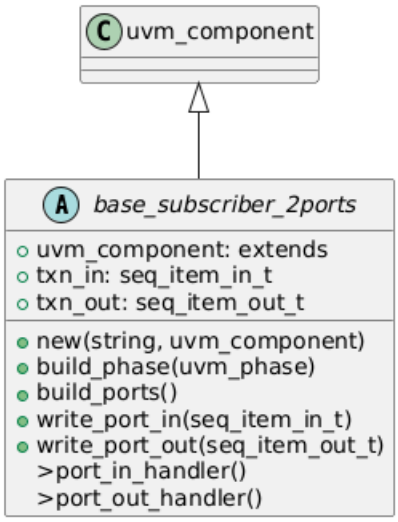


Figure 3.5: Base Subscriber Two Ports UML

The *Base Subscriber Two Ports* is a *UVM Subscriber* with one one extra port, the purpose of adding an extra port to a *UVM Subscriber* is to have a component capable to receive the input and output RTL transactions on different ports. This class implements two pure virtual methods *port_in_handler()* and *port_out_handler()*. These methods should be overridden by the child class and are called by the write methods, these methods process the data received by the ports.

3.1.4 Base Coverage

The *Base Coverage* class is used to implement coverage collectors. This class extends from Base Subscriber Two Ports to collect the input and output transactions on different ports.

The child class should implement the cover groups and override the *cov_in()* and *cov_out* methods to call the sample methods of the cover groups, these methods are called every time the *Monitor* captures and sends a *transaction*.

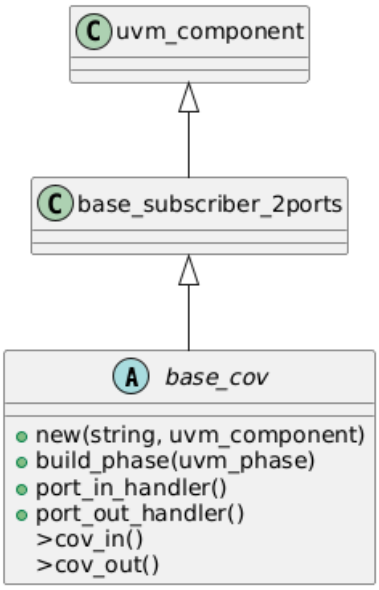


Figure 3.6: Base Coverage UML

3.1.5 Base Driver

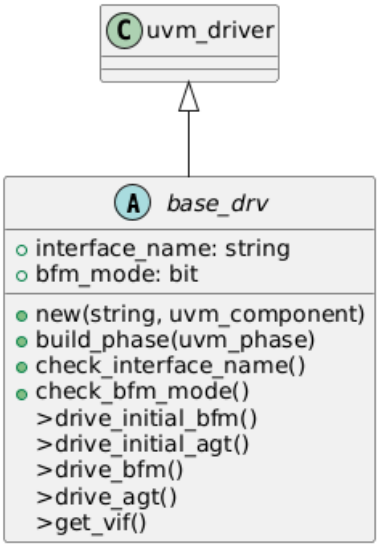


Figure 3.7: Base Driver UML

The *Driver* is used to convert the transactions to pin-level activity to drive stimulus to the RTL. The *Driver* has two modes of operation, *Agent mode* and *BFM mode*.

In *Agent mode* the driver is used to drive stimulus to the unit, which means it drives the inputs of the unit, for example, if the driver is an SPI-Master Driver the driver drives the input to the SPI-Master, in other words, drives the MISO signal.

In *BFM mode* the driver is used to drive the stimulus as if the driver was the unit, it means it drives the outputs of the unit. Using the same example if the driver is an SPI-Master

driver in *BFM mode* the driver drives SCLK, MOSI and SC.

Figure 3.8 shows the driver configurations of an SPI-Master Agent.

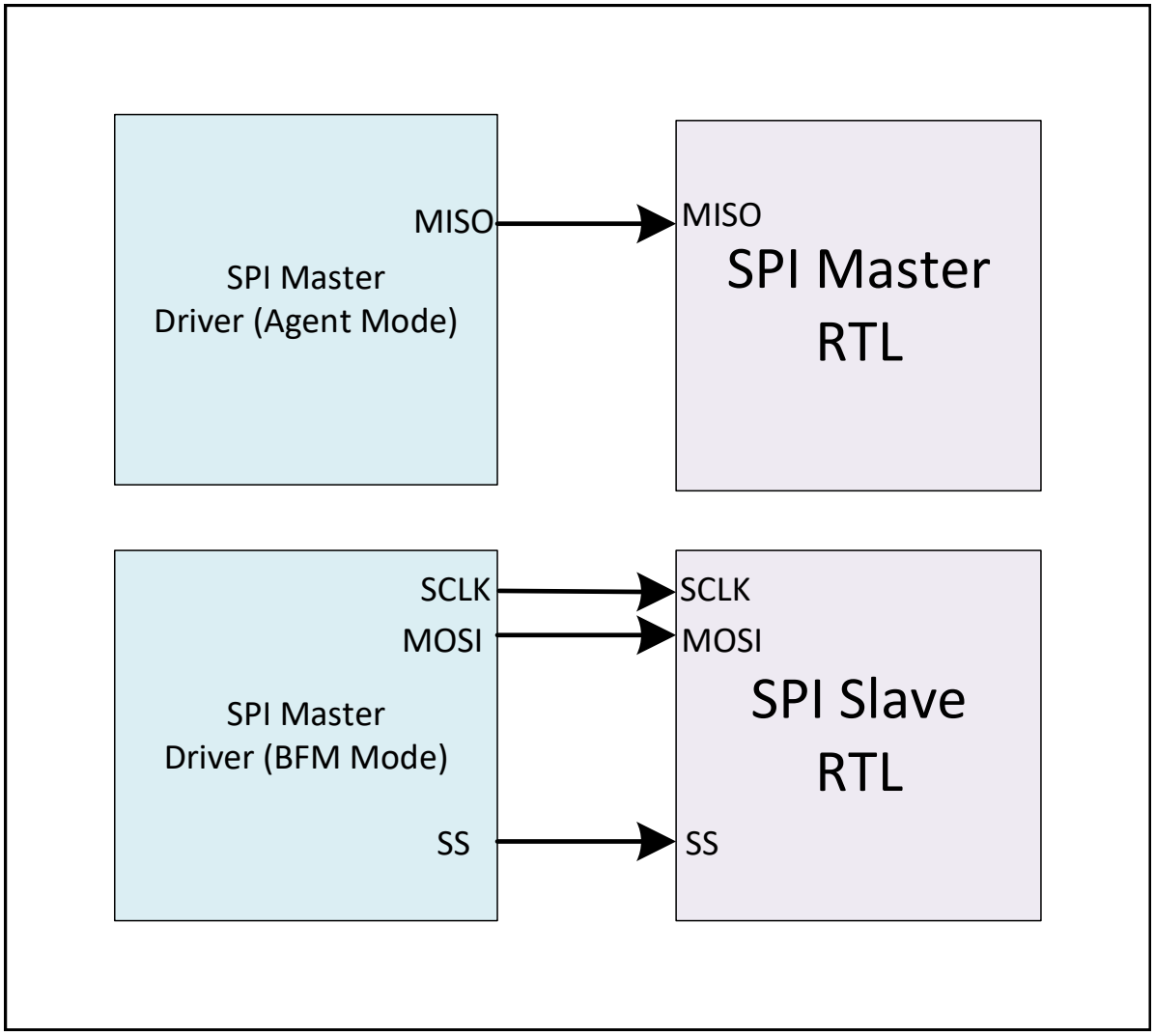


Figure 3.8: Driver Configurations

The *Driver* performs some checks during the *build phase* to make sure the driver can be instantiated correctly, if the checks fail an *UVM fatal* is raised and the test is finished.

It also defines the virtual methods *drive_initial_bfm/agt()* and *drive_bfm/agt()*, the methods should be overridden by the child class and are used to drive the RTL signals using the *Virtual interface*. The *drive_initial()* methods drive the signals at reset, usually by setting the control signals to '0 and the data signals to 'x, the *drive()* methods get the transaction from the *sequencer* and use its data to drive the signals.

3.1.6 Base Monitor

The *Base Monitor* is used to sample the RTL signals and convert them to *transactions* and send them to the different verification components via the *analysis ports*.

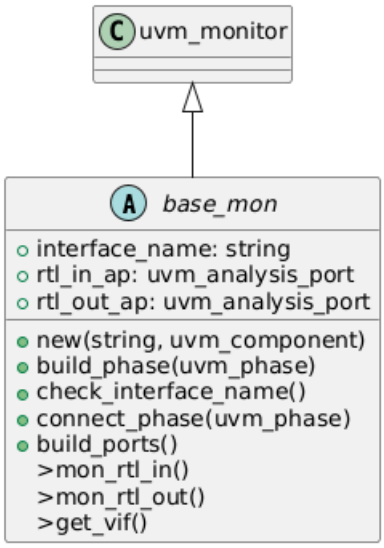


Figure 3.9: Base Monitor UML

The *Base Monitor* defines the virtual methods *mon_rtl_in()* and *mon_rtl_out()* that should be overridden by the child class. These methods are used to sample the input and output RTL signals respectively and send the transactions to the validation components.

3.1.7 Base Model

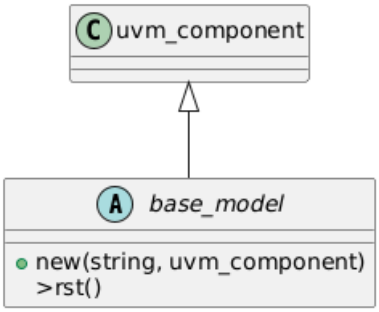


Figure 3.10: Base Model UML

The *Base Model* is used to model the behavior of the unit. This *model* can be used by the *BFM* to generate stimulus or by the *scoreboard* as a reference model to compare the RTL outputs.

This class defines the virtual method *rst()* that should be overridden by the child class, this method is called during reset and is used to reset all the internal states of the *model*.

3.1.8 Base Scoreboard

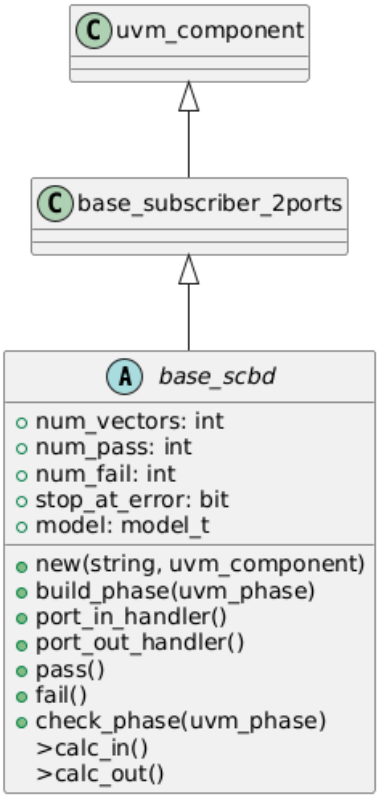


Figure 3.11: Base Scoreboard UML

The *Base Scoreboard* is used to perform the checks of the agent. It extends from *Base Subscriber Two Ports* to implement one port for RTL input *transactions* and one port for RTL output *transactions*.

The *Base Scoreboard* is connected to the monitor using the *analysis ports* and instantiates one *model* that is used as a golden model to get the expected values that the RTL should produce.

The attribute *num_vectors* counts the number of comparisons that have been done by the *Scoreboard*, while the attributes *num_pass* and *num_fail* counts the number of comparisons that have passed and failed respectively. The attribute *stop_at_error* is used to control the flow of the simulation, if it is one the simulation is stopped if a comparison fails, this mode is useful while debugging a specific test, if it is zero the simulation continues until the end of the test, this mode is useful to run regression and get the total number of errors.

The methods *pass()* and *fail()* can be called after the comparison of a vector, they increment the count of *num_vectors*, *num_pass* and *num_fail* depending if the comparison passed or failed. At the *check phase* the count of the vectors is printed and if more than one error failed an *UVM error* is flagged.

The methods *calc_in()* and *calc_out()* are pure virtual methods that should be overridden by the child class, these methods are called by *port_in_handler()* and *port_out_handler()* respectively when a transaction is received from the *monitor*. Typically *calc_in()* can be used

to call the *model* and get the expected value when an RTL input *transaction* is received and *port_out_handler()* to perform the comparison when the RTL output *transaction* is received.

3.1.9 Base Sequence item-Sequence-Sequencer

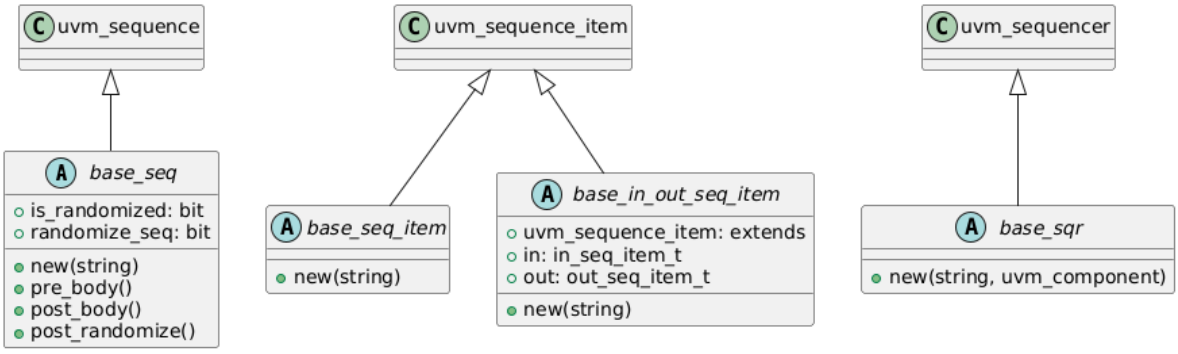


Figure 3.12: Base Sequence Item-Sequence-Sequencer UML

Two Different *sequence items* extends from *uvm_sequence_item*, *base_seq_item* and *base_in_out_seq_item*. *Base_seq_item* is used as a normal *uvm sequence item* grouping the transaction in single software object, this class is used to hold input and output *transactions* independently. *Base_in_out_seq_item* is used to group both, input and output *transactions* in a single software object by declaring two attributes one for the input *transaction* and one for the output *transaction* this is useful for components like the *tracker* that makes use of input and output *transactions* at the same time.

The *base_seq* extends from *uvm_sequence*. This class receives as parameter the type of *sequence item* to be used and implements the attributes *is_randomized* and *randomize_seq* and overrides the methods *pre_body()*, *post_body()* and *post_randomize()*, these are used to allow the randomization of the *sequence* or the *uvm_sequence_item* independently, the *uvm_sequence_item* can be randomized by the test and used in the *sequence* or randomized directly in the body of the *sequence*.

The *base_sqr* just extends from *uvm_sequencer* and it is specialized with the type of *sequence_item* to be used. It is used to send *uvm_sequences* from the test or *BFM* to the *driver*.

3.1.10 Base Tracker

The *Base Tracker* is used to keep track of the transactions received by the *monitor* and print them in a table format that is more readable for humans.

The *Tracker* is connected to the *monitor* using the *analysis ports* and implements many methods to format the data to be printed. Most of the methods do not need to be overridden by the child class, the child class only needs to specify the file to store the table using the attribute *file* and override the method *print_header()* to select the fields to be printed and the method *print_data()* to select the data to be printed in each field. The methods *trk_in()* and

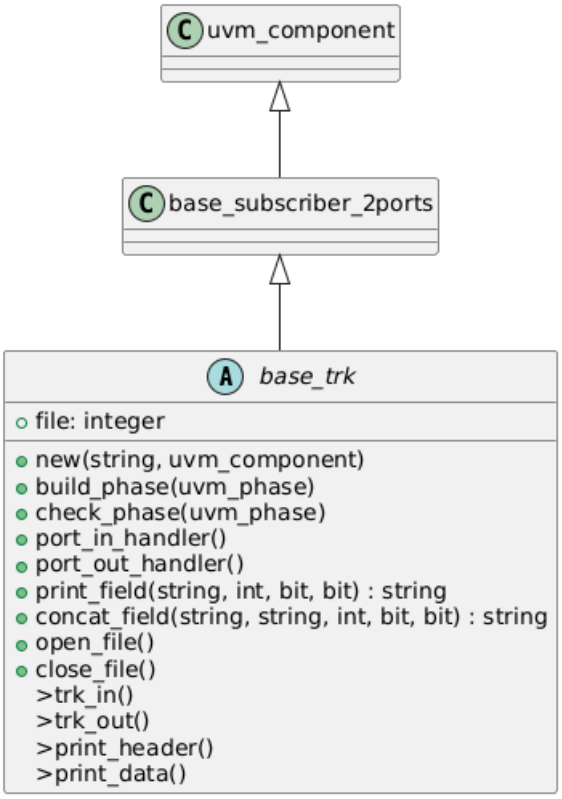


Figure 3.13: Base Tracker UML

trk_out() are called every time a *transaction* is received and calls *print_data()* to print a new line in the table.

3.1.11 Base Agent

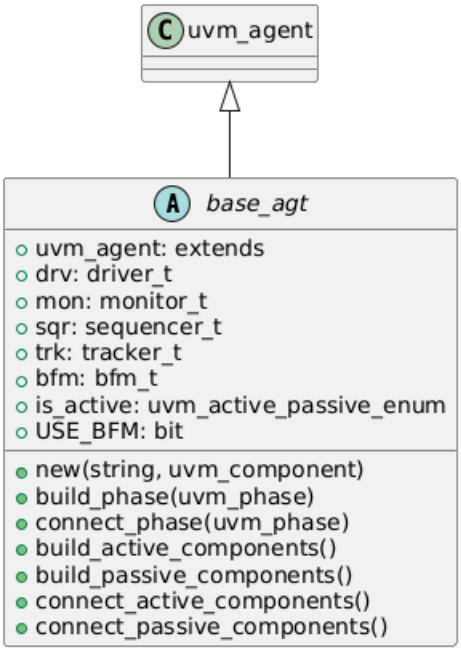


Figure 3.14: Base Agent UML

The *Base Agent* extends from *uvm_agent* and instantiates and connects the *driver*, *monitor*, *sequencer*, *BFM* and *tracker*.

Not all the components are instantiated and connected, if the agent is configured as an *Passive Agent* using the attribute *is_active* just the *monitor* and *tracker* are instantiated since the agent does not drive stimulus, just spies the interface and get the *transactions* using the *monitor*. If the agent is configured as *Active Agent* the *driver* is also instantiated, if *USE_BFM* is one the *BFM* is instantiated and the *sequencer* is not, the *driver* is connected to the *BFM's sequencer*, if *USE_BFM* is zero the *BFM* is not instantiated and the *sequencer* is instantiated and connected to the *driver*, in this mode the test generates the stimulus instead of the *BFM*. Figure 3.15 shows a flow diagram with the instantiation and connection of the components.

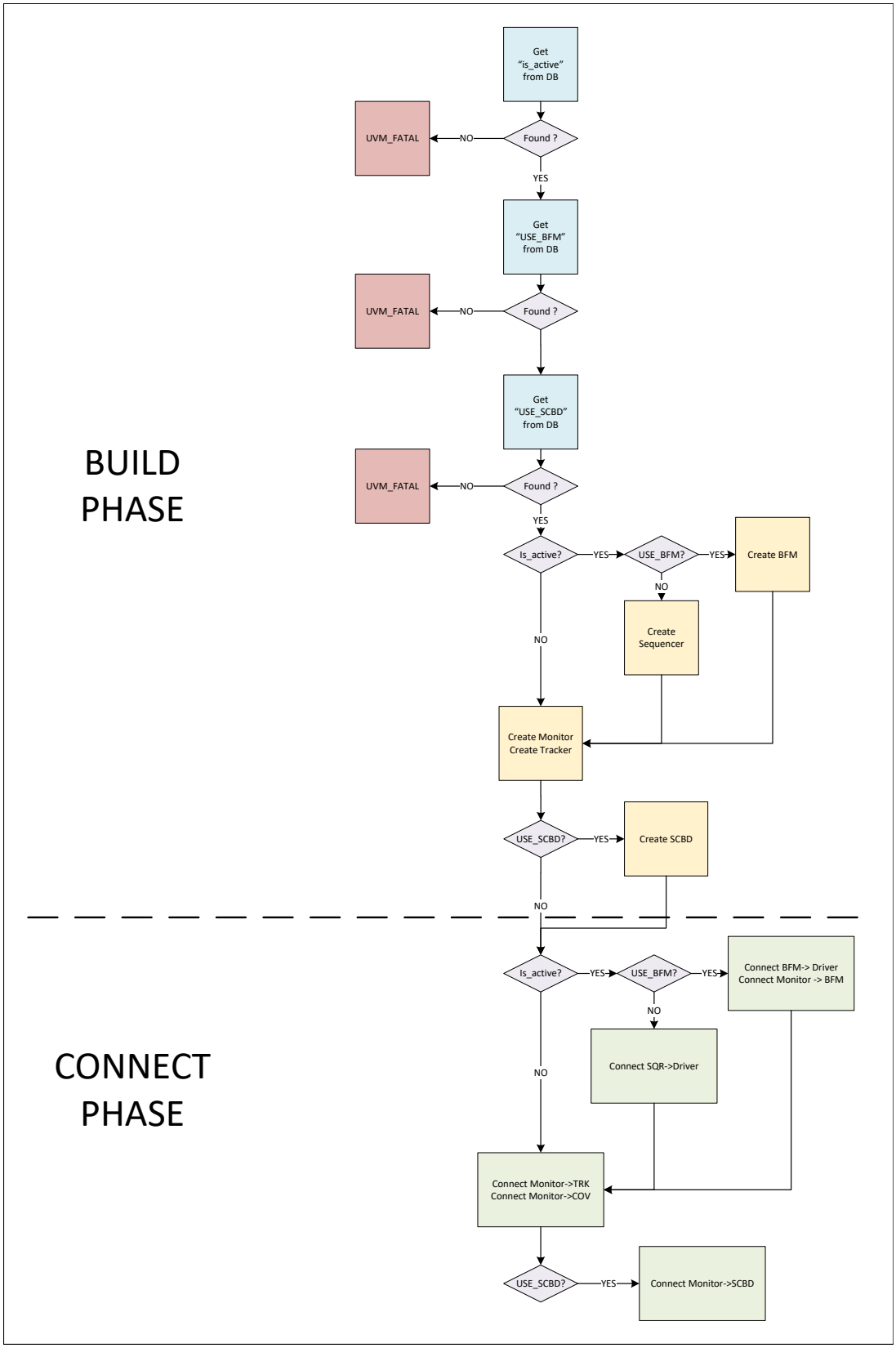


Figure 3.15: Build Agent flow diagram

3.1.12 Base Environment

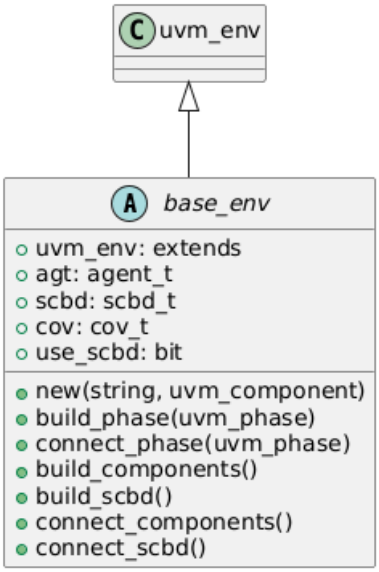


Figure 3.16: Base Environment UML

In the *Base environment* the *Agent*, *scoreboard* and *coverage* are instantiated and connected. The *scoreboard* is instantiated only if the attribute *use_scbd* is one, if it is zro it is not instantiated and the checkers are disabled. The *coverage* and *scoreboard* are connected to the *Agent's monitor* during the *connect phase*.

3.1.13 RF Model

The *RF Model* models a register file, it receives as a parameter the number of registers that contain the register file and the type of data that stores each register, for example, logic[31:0].

The model implements the functions *rst()* and *rst_rand()* these functions can be called during reset to give the registers an initial value, *rst()* sets *rst_val* to all registers, this value can be set using *set_rst_val()* and can be 'x, *rst_rand()* sets a random value to each register.

The method *write_data()* is used to write a register, the first argument is the address and the second argument is the data to be written. The method *read_data()* is used to read the register file receiving as argument the address and returning the data of the register. Both methods validate the address, if the address is 'x or is out of range of the valid addresses it flags an *UVM fatal*.

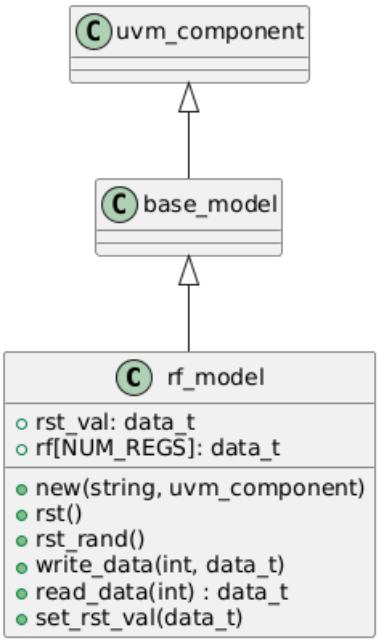


Figure 3.17: RF Model UML

3.1.14 RF BFM

The *RF BFM* is used to simulate a register file and generate stimulus by using an *RF Model*. The difference between the *RF BFM* and the *RF Model* can be seen as the control path and the data path, the *RF Model* models a register file by writing and reading registers while the *RF BFM* controls the *RF Model* by deciding when to read, when to write and what register should be read and write.

The *RF BFM* implements logic to perform all reads before updating the *RF Model* with the write. The child class only needs to override the methods *update_wr_rf()* that is called every time a write *transaction* is received and *update_rd_rf()* that is called every time a read *transaction* is received.

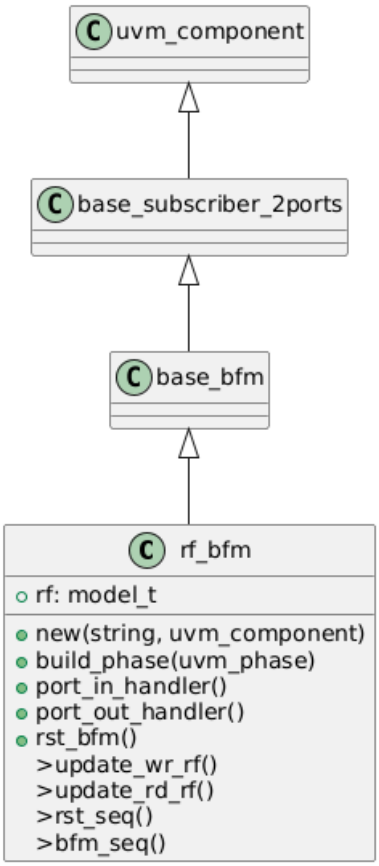


Figure 3.18: RF BFM UML

3.1.15 Rotator

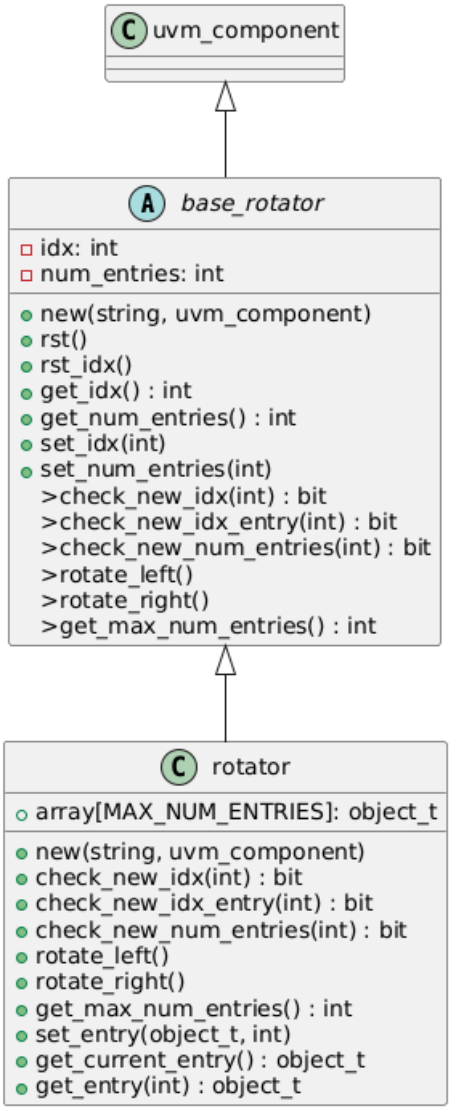


Figure 3.19: Rotator UML

The *Rotator* implements a ring buffer. It receives as a parameter the maximum number of entries in the buffer and the type of object to be allocated in the buffer. It can allocate from one to *MAX_NUM_ENTRIES* by updating the attribute *num_entries* using the method *set_num_entries()*, this can be updated dynamically during the test but it should be always greater or equal than one and less or equal than the parameter *MAX_NUM_ENTRIES*.

The methods *check_new_idx()*, *check_new_idx_entry()* and *check_num_entries()* are methods used internally by the *Rotator* to validate the data and are not intended to be called by an external class.

The elements can be added to the buffer with the method *set_entry()* by indicating the object to store and the index to store the object, the data is validated and if the object can not be added to the buffer it raises an *UVM fatal*. To remove the object from the buffer the

indexes should be updated to override that object, set the object as null or decrease the size of the *Rotator*.

The methods *rotate_left()* and *rotate_right()* rotate the head of the buffer to the left or the right, but do not return any data, only update the index that is pointing to the head, to get the data the method *get_current_entry()* can be used to get the data pointed by the head without updating the head or the method *get_entry()* to get the data from an arbitrary position without updating the head.

3.2 RISCV-RF Agent

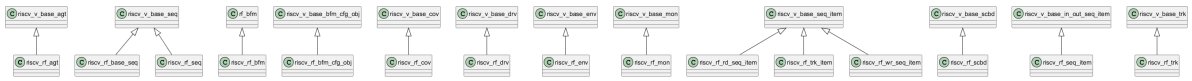


Figure 3.20: RISCV RF components UML

The *RISCV RF Agent* is the agent used to validate and simulate an RISCV32i register file. This agent contains an *scoreboard* capable of performing checks to the register file, but since the *RISCV-V Engine* does not implement the integer register file the *scoreboard* is disabled and the agent is used in *BFM* mode to simulate the integer register file and generate stimulus to the *RISCV-V Engine*.

Only the classes that add extra logic or modify the behavior of the base class are described in the following sections, if the extended class remains the same as the base class the class is not described.

3.2.1 RISCV RF BFM

The *RISCV RF BFM* adds the configuration knob *init_rand_rf* to the *BFM configuration object*, this knob indicates how to initialize the registers at reset, if it is 1'b1 all the registers get a random value except for register zero that is always initialized with '0.

The *RISCV RF BFM* adds logic to check for the *write_enable* of the data before writing the register file and to read both ports of the register file, in other words, performs two reads to the *RF Model*, the data that is read from the ports is encapsulated in the *sequence* and sent to the *driver* to be sent to the RTL.

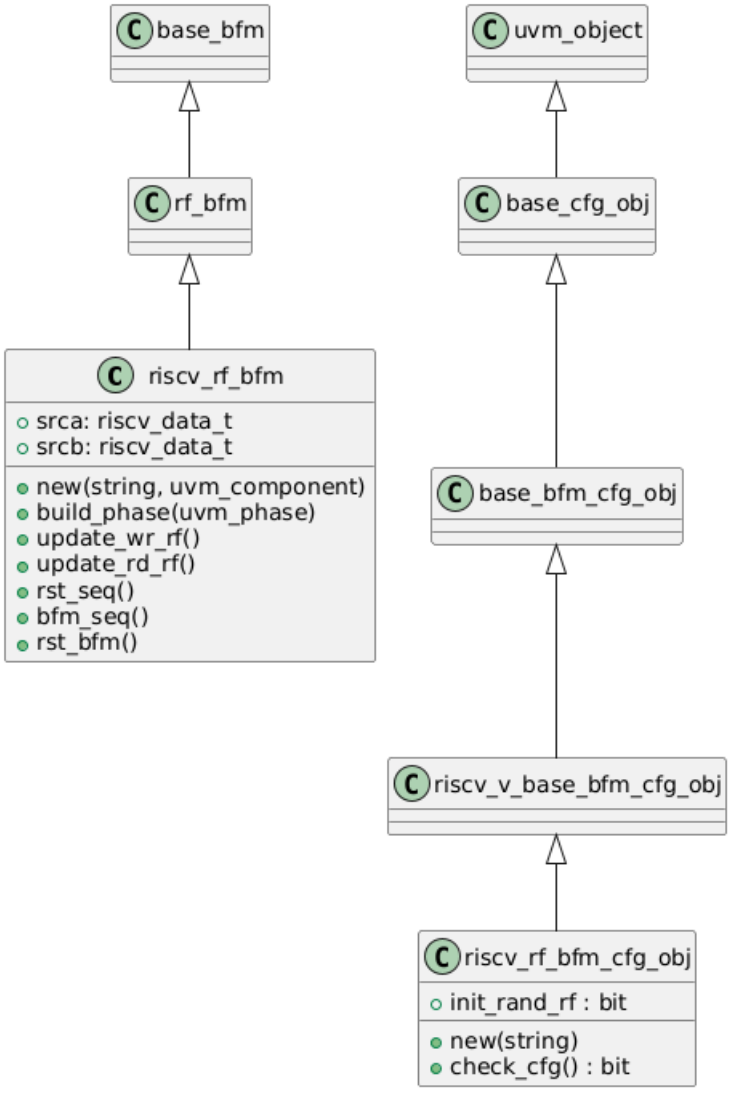


Figure 3.21: RISCV RF BFM UML

3.2.2 RISCV RF Coverage

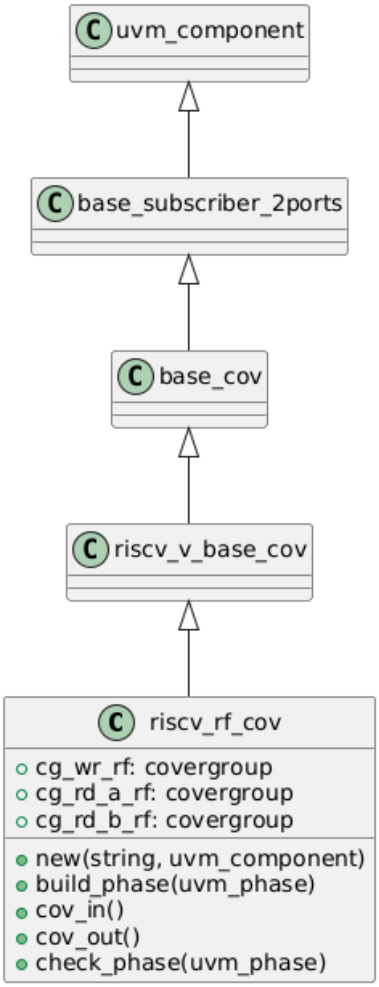


Figure 3.22: RISCV RF Coverage UML

The *RISCV RF Coverage* is the coverage collector for the integer register file and implements a cover point to make sure that all registers are exercised during simulation. It implements three cover groups, one to cover the writes to the register file and two to cover the reads to the register file, one per port. One cover bin is created per each valid read/write address to cover that all registers are written/read, in the case of the write/read data to avoid having too many cover bins the range is split into four cover bins that capture different ranges of the data. The cover points with its description are shown in Table 3.3

Table 3.3: RISCV RF Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
WR	wr_addr	addr[32]	Creates 32 cover bins to check that all addresses are selected
	wr_en	wr_enable	Creates 1 cover bin to check that wr_en is asserted
	wr_addr_en	addr[32]	Creates 32 cover bins to check that all addresses are written
	wr_data	data[4]	Creates 4 cover bins to cover all the range of the write data
	WR_ADDR_EN X WR_DATA	cross[128]	Cross cover to check that all addresses are written with different combinations of data.
RD_A	rd_addr	addr[32]	Creates 32 covers to check that all addresses are read by port A
	rd_data	data[4]	Creates 4 cover bins to cover all the range of the data read by port A
	RD_ADDR X RD_DATA	cross[128]	Cross cover to check that different combinations of data are read from port A
RD_B	rd_addr	addr[32]	Creates 32 covers to check that all addresses are read by port B
	rd_data	data[4]	Creates 4 cover bins to cover all the range of the data read by port B
	RD_ADDR X RD_DATA	cross[128]	Cross cover to check that different combinations of data are read from port B

3.2.3 RISCV RF Driver/Monitor

The *RISCV RF Driver* transforms the *RISCV RF transaction* to pin level to be read by the RTL, this action is performed at the rising edge of the clock and only takes one clock cycle. The *RISCV RF Monitor* works similarly, transforming the RTL signals from pin level to *uvm sequence items* in one clock cycle and sampling the signals at the rising edge of the clock.

The Table 3.4 shows the signals that are driven by the *Driver* and sampled by the *Monitor*.

Table 3.4: RISCV RF Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**wr_addr**	logic[4:0]	Register File write address
	rd_addr_A	logic[4:0]	Register File read port A read address
	rd_addr_B	logic[4:0]	Register File read port B read address
	data_in	logic[31:0]	Register File write data
	wr_en	logic	Register File write enable
BFM/ RTL OUT	**data_out_A**	logic[31:0]	Register File read port A read data
	data_out_B	logic[31:0]	Register File read port B read data

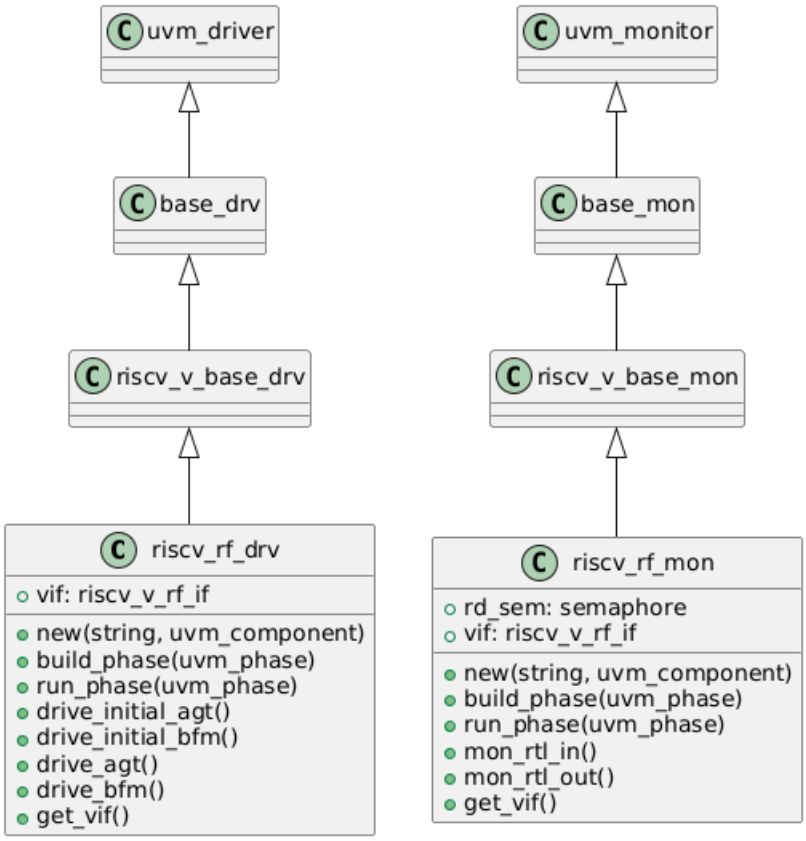


Figure 3.23: RISCV RF Driver/Monitor UML

3.2.4 RISCV RF Sequence Item-Sequence

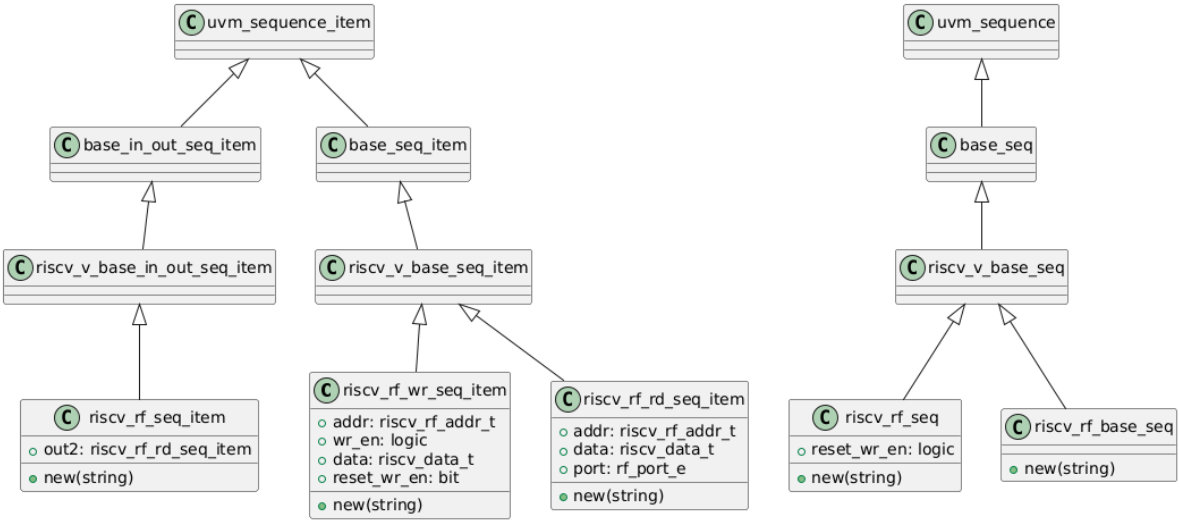


Figure 3.24: RISCV RF sequence item-sequence UML

The *RISCV-RF Agent* makes use of three different *sequence items*, one for read transactions, one for write transactions and one that includes write and read transactions.

The *riscv_rf_rd_seq_item* is the transaction used for the Register File read transactions, it includes the *read_addr*, *rd_data* and *rd_port*. The *riscv_rf_wr_seq_item* is used in the Register File write transactions and includes the *wr_addr*, *wr_en* and *wr_data*. Finally the *riscv_rf_seq_item* is used to group the read and write transactions in a single transaction, it instantiates one *riscv_rf_wr_seq_item* and two *riscv_rf_rd_seq_item*, one per read port.

The *riscv_rf_seq* is a generic sequence used for both read and write transactions. It does not implement additional logic, just randomizes all the fields if they have not been set by the test previously, the *driver* reads the corresponding signals to be driven accordingly to its configuration.

3.2.5 RISCV RF Scoreboard

The *RISCV Integer Register File Scoreboard* is not used actively in this project since the Integer Register File is not implemented by the *RISCV Vector Engine* and therefore is not validated, nevertheless the *scoreboard* is implemented with a basic functionality.

The *scoreboard* make use of the *RF Model* as reference model, every time a write transaction is captured the *model* is updated with the new value, every time a read transaction is captured the *model* is read and the data compared with the data coming from the RTL, the Table 3.5 shows the checkers performed by the *scoreboard*.

Table 3.5: RISCV RF Scoreboard checks

Checker	Signal	Description
Read Data	int_rf.rd_data[31:0]	Checks that the Integer register file read data is correct

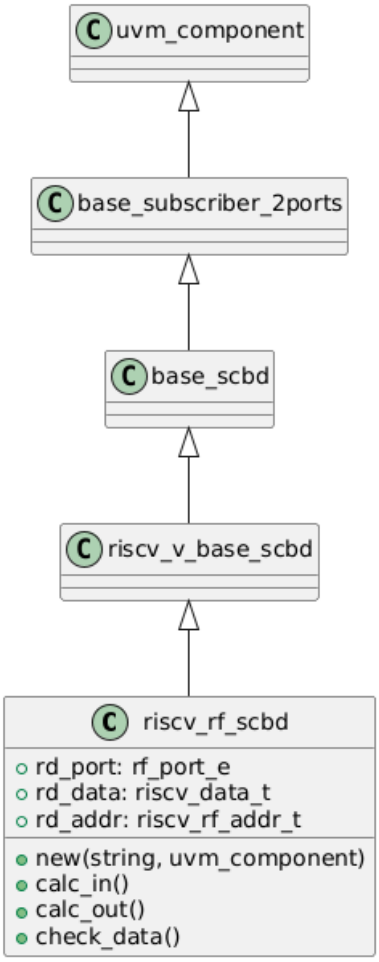


Figure 3.25: RISCV RF Scoreboard UML

3.2.6 RISCV RF Tracker

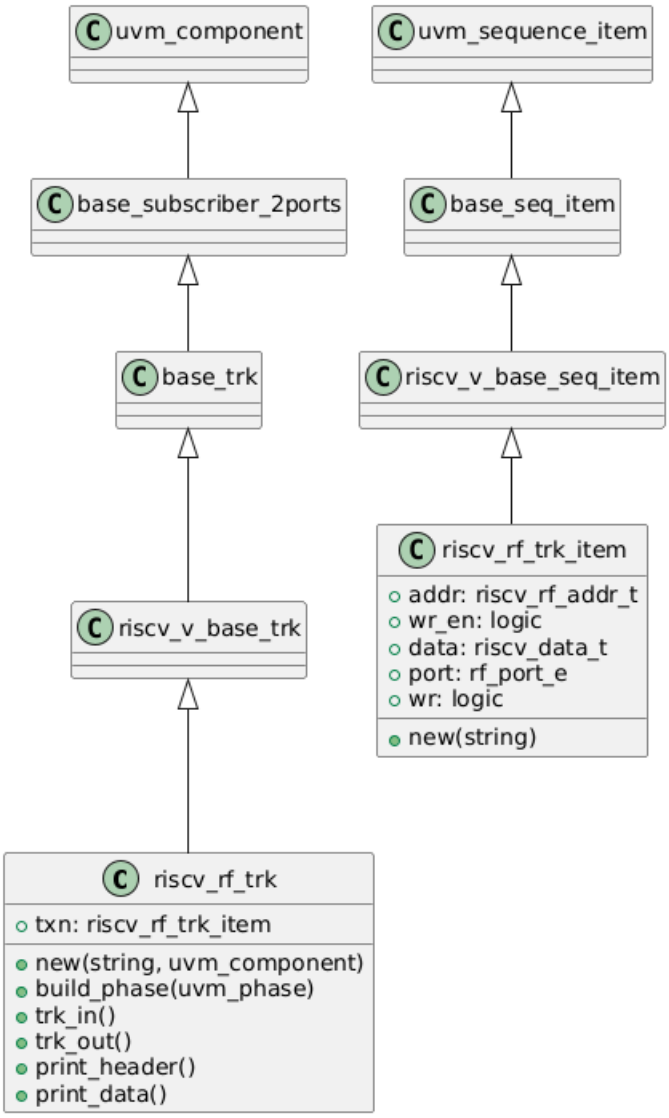


Figure 3.26: RISCV RF Tracker UML

The *RISCV RF Tracker* tracks all the Integer register file transactions, it prints one line in the tracker table per read or write transaction. One example of the tracker generated is shown in Figure 3.27.

Time	Addr	RD/WR	Port	wr_en	data
2150000	0x2	WR	RF_WR_PORT	0x1	0xfffffffd
2200000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2200000	0x15	RD	RF_RD_PORT_B	0xx	0x1dd04470
2200000	0x12	WR	RF_WR_PORT	0x1	0xfffffff7
2250000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2250000	0x5	RD	RF_RD_PORT_B	0xx	0x39574f3
2250000	0x16	WR	RF_WR_PORT	0x1	0xfffffffc
2300000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2300000	0x8	RD	RF_RD_PORT_B	0xx	0x10b12c38
2300000	0x5	WR	RF_WR_PORT	0x1	0xfffffff9
2350000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2350000	0xc	RD	RF_RD_PORT_B	0xx	0xf3af11be
2350000	0x11	WR	RF_WR_PORT	0x1	0xfffffff2
2400000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2400000	0x5	RD	RF_RD_PORT_B	0xx	0x9125cf67
2400000	0x1f	WR	RF_WR_PORT	0x1	0xd
2450000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2450000	0x1f	RD	RF_RD_PORT_B	0xx	0xfffffff9
2450000	0x1a	WR	RF_WR_PORT	0x1	0xfffffff2
2500000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170
2500000	0x19	RD	RF_RD_PORT_B	0xx	0xd
2500000	0x16	WR	RF_WR_PORT	0x1	0xfffffffd
2550000	0x0	RD	RF_RD_PORT_A	0xx	0xe28c6170

Figure 3.27: RISCV RF Tracker Example

3.3 RISCV-V Base



Figure 3.28: RISCV-V Base components UML

The *RISCV-V Base* components extend from the *Base Components*. These components are an extra layer of abstraction to implement logic that is common to all riscv-v components but is not common to other IPs.

In this project no extra logic is implemented in these components, for that reason they are not explained in further sections, but all the riscv-v components extend from the *RISCV-V Base* components instead of *Base Components*.

3.4 RISCV-V-RF Agent

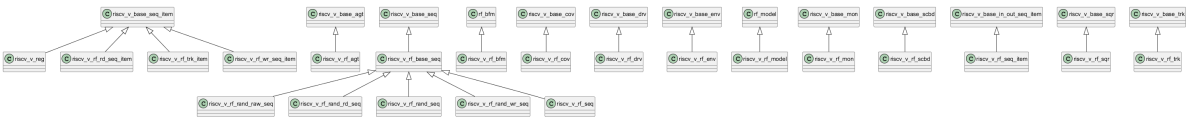


Figure 3.29: RISCV-V RF components UML

The purpose of the *RISCV-V-RF Agent* is to validate the *VRF*. This agent can be instantiated as an active agent to generate stimuli to the *VRF* or as a passive agent to read the stimuli that are received by the *VRF* and verify the component.

3.4.1 RISCV-V Register

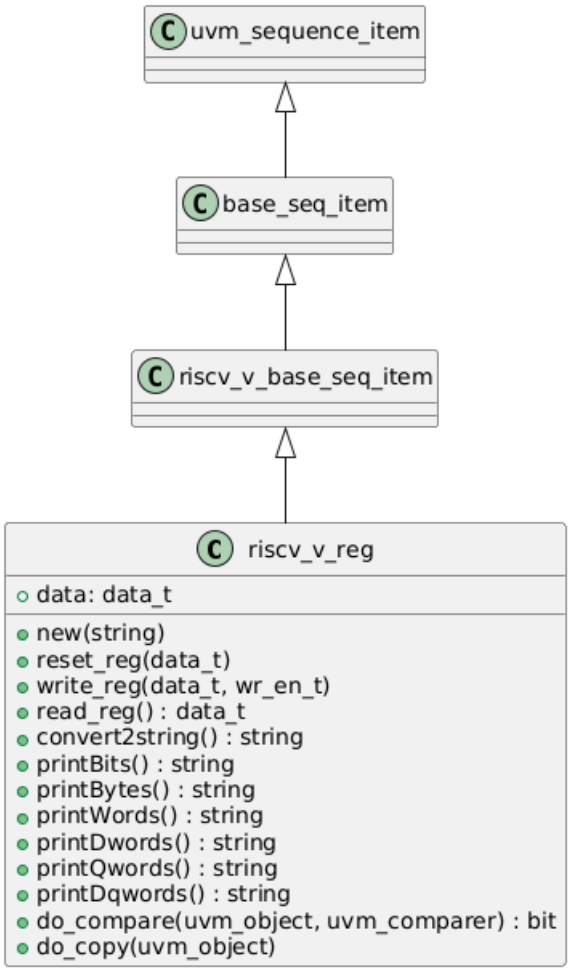


Figure 3.30: RISCV-V Register UML

The *RISCV-V Register* class is used to simulate the behavior of a Vector register. It implements the methods *write_reg()* and *read_reg()* that are used to write and read the content of the register, the write method also receives the write enable to only modify the bytes that are enabled. It also implements some print methods to print the data of the register in different formats, useful for debugging. Finally implements the methods *do_compare()* and *do_copy()* that are used by the *scoreboard* to compare the RTL register with the expected register.

3.4.2 RISCV-V-RF BFM

The *RISCV-V-RF BFM* is not used directly in this project since the RTL *VRF* is always used and there is no need to simulate the *VRF* as a verification component, but this class can be used to simulate the *VRF* if a different top level is used, for example, to validate the ALU without including the rest of the RTL this class can be used to send stimulus to the ALU.

As the *RISCV RF BFM* this *BFM* adds logic to check the *write_en* before writing the register file and to read both ports in the same transaction to return the requested data.

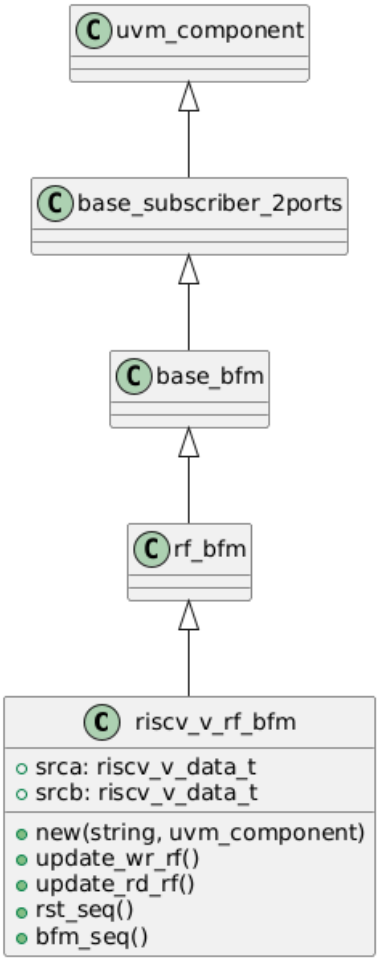


Figure 3.31: RISCV-V RF BFM UML

3.4.3 RISCV-V-RF Coverage

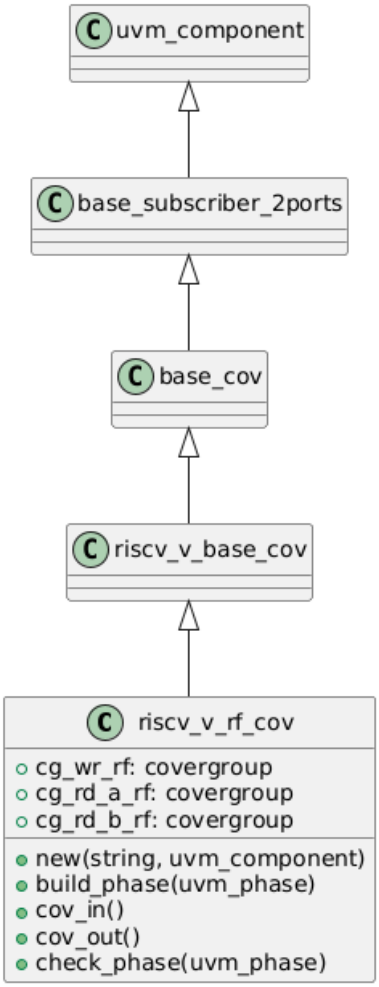


Figure 3.32: RISCV-V RF Coverage UML

The *RISCV-V-RF Coverage* collector is similar to the *RISCV RF coverage* collector. It instantiates four cover groups, one cover group for the writes, two cover groups for the reads and adds one extra cover group for the mask. The main difference in the write cover group is the *write_enable*, instead of only creating one bin to cover that the *write_enable* is asserted this collector creates sixteen cover bins, one bin per bit in the *write_enable* to cover that all the bytes are written. Another difference is in the number of bins created to sample the read/write data, since the vector registers have a width of 128 bits this collector requires sixteen cover bins to cover all the range of the data. The mask cover group creates thirty-two cover bins, sixteen bins to cover that all mask bits are set to one and sixteen bins to check that are set to zero. The cover points with its description are shown in Table 3.6

Table 3.6: RISCV-V RF Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
WR	wr_addr	addr[32]	Creates 32 cover bins to check that all addresses are selected
	wr_en	wr_enable[16]	Creates 16 cover bins to check that all bits of write enable are asserted
	wr_data	data[16]	Creates 16 cover bins to cover all the range of the write data, one bin per byte
	WR_ADDR X WR_EN	cross[512]	Cross cover to check that all bytes of all address are written.
RD_A	rd_addr	addr[32]	Creates 32 covers to check that all addresses are read by port A
	rd_data	data[16]	Creates 16 cover bins to cover all the range of the data read by port A, one bin per byte
	RD_ADDR X RD_DATA	cross[512]	Cross cover to check that different combinations of data are read from port A
RD_B	rd_addr	addr[32]	Creates 32 covers bins to check that all addresses are read by port B
	rd_data	data[16]	Creates 16 cover bins to cover all the range of the data read by port B, one bin per byte
	RD_ADDR X RD_DATA	cross[512]	Cross cover to check that different combinations of data are read from port B
MASK	rd_mask	mask_set[16] mask_unset[16]	Creates 16 cover bins to cover that all mask bits are set to 1 And 16 cover bins to cover that all mask bits are set to 0

3.4.4 RISCV-V-RF Driver/Monitor

The *monitor and driver* of the *VRF* behaves similar to its integer counterpart the *riscv rf driver/monitor*. The *driver* transforms the *sequence item* to pin level and the *monitor* samples the data in one clock cycle at the rising edge of the clock. The Table 3.7 shows the signals driven/sampled by the driver/monitor.

Table 3.7: RISCV-V-RF Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**wr_addr**	logic[4:0]	Register File write address
	rd_addr_A	logic[4:0]	Register File read port A read address
	rd_addr_B	logic[4:0]	Register File read port B read address
	data_in	logic[127:0]	Register File write data
	wr_en	logic[15:0]	Register File write enable one bit per byte in the register.
BFM/ RTL OUT	**data_out_A**	logic[127:0]	Register File read port A read data
	data_out_B	logic[127:0]	Register File read port B read data

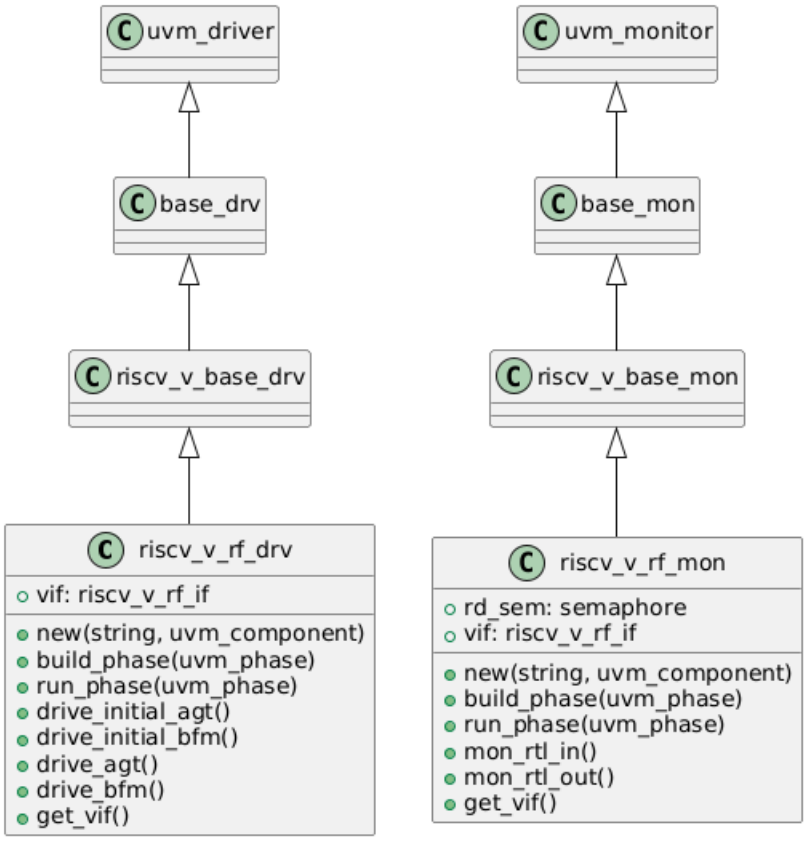


Figure 3.33: RISCV-V RF Driver/Monitor UML

3.4.5 RISCV-V RF Model

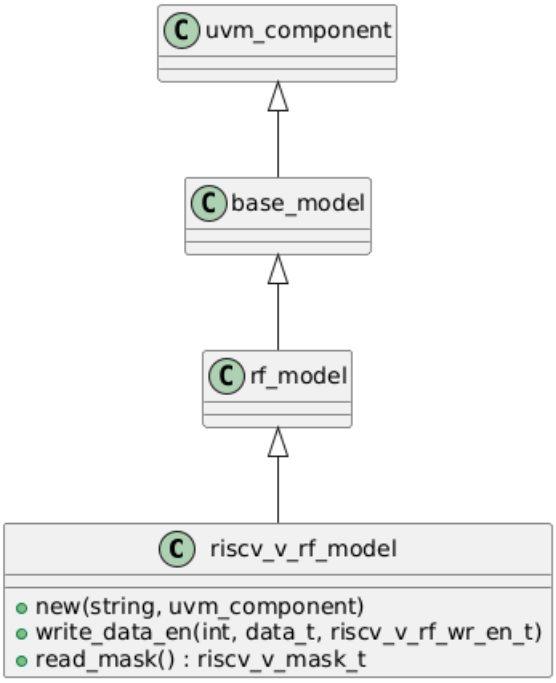


Figure 3.34: RISCV-V RF Model UML

The *RISCV-V RF Model* simulates the behavior of the *VRF* and can be used to replace the RTL in a simulation environment. It implements logic to validate the write/read address, if it is not valid, is out of range or is 'x, it raises an *UVM fatal*. It also implements logic to use sixteen bits for the *write_en* instead of one to write each byte of the register independently.

3.4.6 RISCV-V RF Sequence Item-Sequence

As the *RISCV-V RF sequence items* the vector version makes use of three *sequence items*. The *riscv_v_rf_rd_seq_item* used for read transactions, the *riscv_v_rf_wr_seq_item* used for write transactions and *riscv_v_rf_seq_item* that includes one *riscv_v_rf_wr_seq_item* and two *riscv_v_rf_rd_seq_item* to include the complete interaction of the *VRF* in a single *sequence item*.

Five Different sequences are implemented to drive stimulus to the *VRF*.

The *riscv_v_rf_seq* is a sequence without a predefined behavior, its fields should be set or randomized by the test and sends one *riscv_v_rf_seq_item*, this sequence can be used to make directed tests to send transactions with specific values.

The *riscv_v_rf_rand_seq* is a sequence to send completely random transactions, all the fields are randomized in the body of the sequence and the test doesn't need to do it, the test only needs to start the *sequence* and send it to the *uvm sequencer*. This sequence is useful to generate random tests.

The *riscv_v_rf_rand_rd_seq* sends random read transactions, the *write_en* is set to zero to remove the writes and the *rd_addra* and *rd_addrb* are randomized by the sequence. This

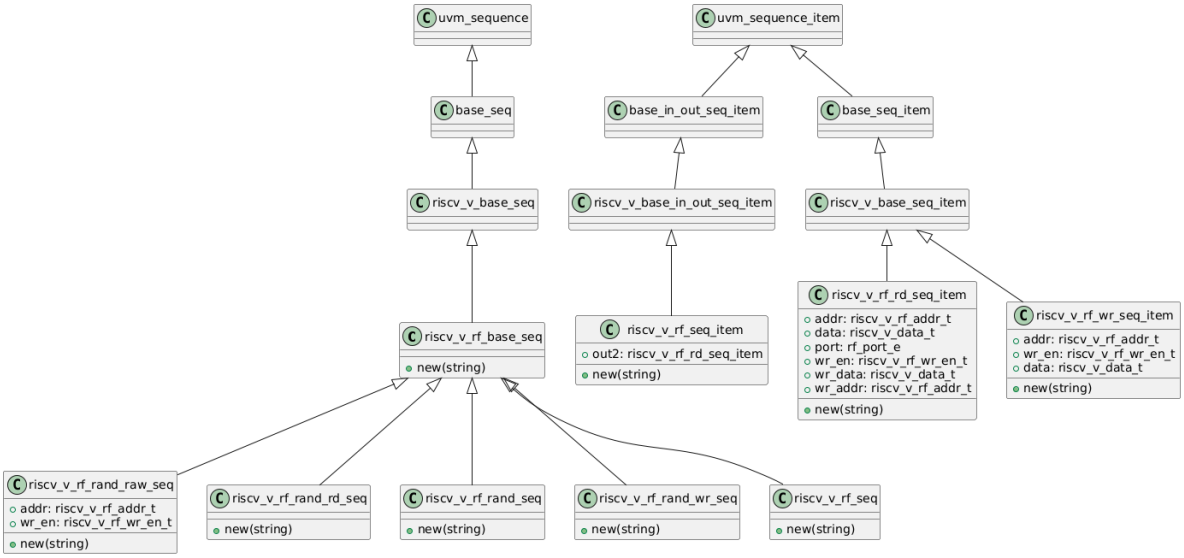


Figure 3.35: RISCV-V RF Sequence Item-Sequence UML

sequence can be used to generate random read tests to validate that the read data and the internal bypass of the *VRF* is correct.

The *riscv_rf_rand_wr_seq* sends random write transactions, the *write_en*, *wr_addr* and *wr_data* are randomized to send random write transactions. The sequence is used to send multiple write transactions to fill the *VRF* with random data.

Finally, the *riscv_v_rf_rand_raw_seq* sends a write transaction followed by a read transaction to the same address. In the first transaction the *write_en*, *wr_addr* and *wr_data* are randomized to write random data to a random register, one cycle later in the second transaction both read addresses, *rd_addra* and *rd_addrb* are set to the *wr_addr* used in the previous transaction. This sequence is used to verify that the data is stored correctly in the *VRF* by reading the register one cycle after it was written.

3.4.7 RISCV-V RF Scoreboard

The *RISCV-V RF Scoreboard* is used to validate the behavior of the *VRF*. It instantiates an array of *RISCV-V Registers* to use them as a reference model, every time a write transaction is sampled the *Scoreboard* updates the data of the *RISCV-V Registers*, every time a read transaction is sampled the *Scoreboard* reads the corresponding *RISCV-V Registers* and compares its data with the data coming from the RTL. The checkers performed by the *RISCV-V RF Scoreboard* are shown in Table 3.8.

Table 3.8: RISCV-V RF Scoreboard checks

Checker	Signal	Description
Read Data	vec_rf.rd_data[127:0]	Checks that the Vector register file read data is correct

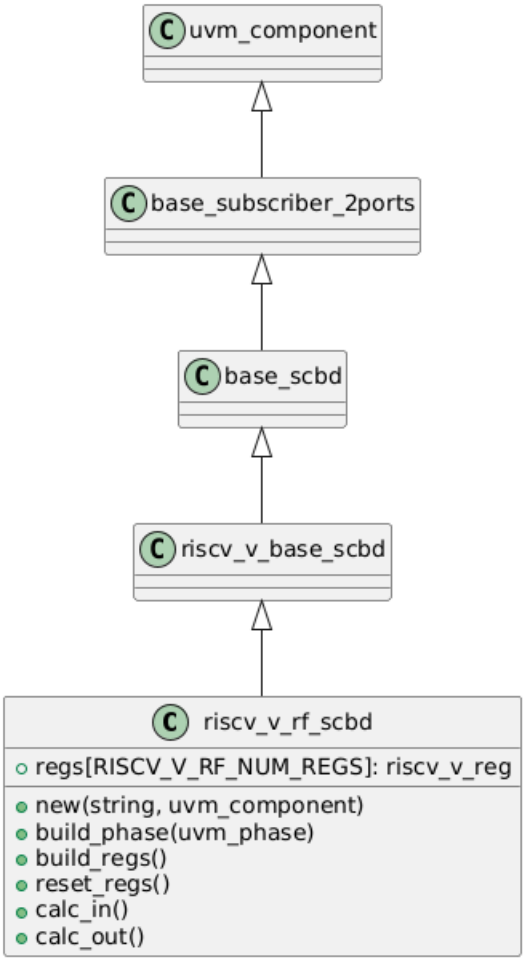


Figure 3.36: RISCV-V RF Scoreboard UML

3.4.8 RISCV-V RF Tracker

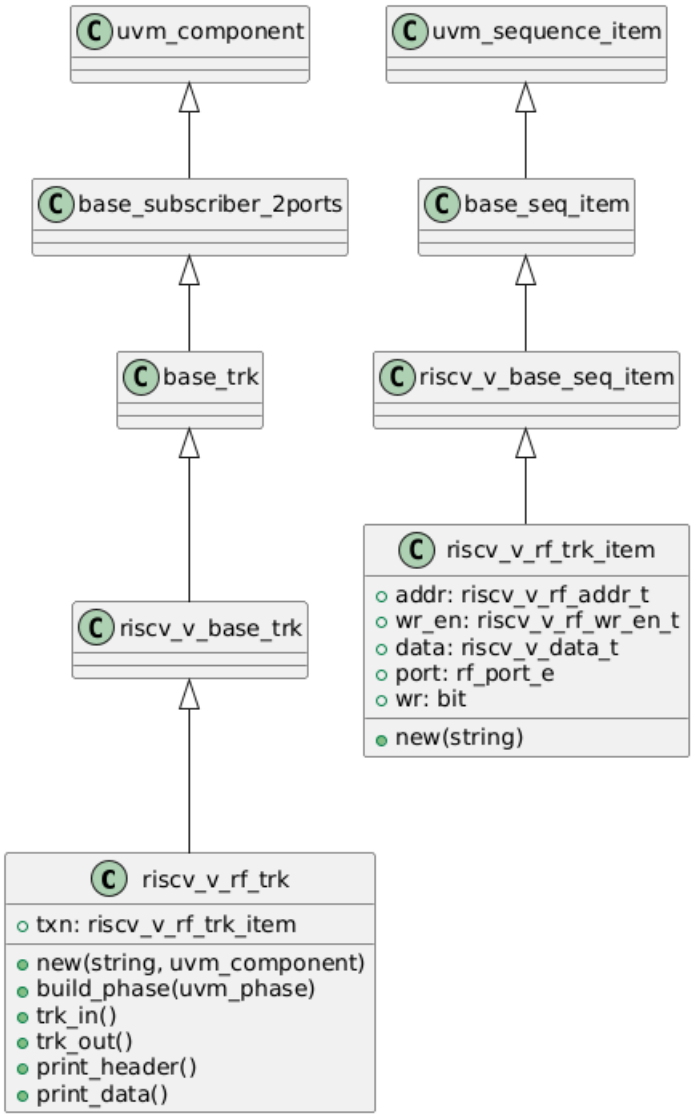


Figure 3.37: RISCV-V RF Tracker UML

The *RISCV-V RF Tracker* is similar to the *RISCV RF Tracker*, tracks all the transactions coming to the *VRF* and prints them in a table printing one transaction per line. One example of the *Tracker* is shown in Figure 3.38.

Time	Addr	RD/WR	Port	wr_en	data
33175000	0x12	RD	RF_RD_PORT_B	0xxxxx	0x978d8a22d576ca3c380a049a95c306ce
33175000	0xf	WR	RF_WR_PORT	0xfffff	0x1a191a2407b8de7d1a1a1a2407b9de7c
33225000	0x1c	RD	RF_RD_PORT_A	0xxxxx	0x91918426d17ac64132b9b0cc640d4adb
33225000	0x9	RD	RF_RD_PORT_B	0xxxxx	0x3c4dadd45f0730d8b0cfafe07ff38d5f
33225000	0x5	WR	RF_WR_PORT	0xfffff	0x1a0d1a1b07acde741a0eafef7ff58d6c
33275000	0x18	RD	RF_RD_PORT_A	0xxxxx	0x4a460a09d51b2aae0a0a0a098c5ad0fd
33275000	0x1d	RD	RF_RD_PORT_B	0xxxxx	0x403c00f4cc112198000000f48350c7e7
33275000	0x15	WR	RF_WR_PORT	0xfffff	0x1a0d1a0a07acde621a0eafde7ff58d5a
33325000	0x1d	RD	RF_RD_PORT_A	0xxxxx	0x403c00f4cc112198000000f48350c7e7
33325000	0x3	RD	RF_RD_PORT_B	0xxxxx	0x1a0d1a0407acde5d1a0eafd87ff58d55
33325000	0x10	WR	RF_WR_PORT	0xfffff	0x978d8a18d576ca31380a049095c306c3
33375000	0x8	RD	RF_RD_PORT_A	0xxxxx	0x1a0d1a0c07acde671a0eafe07ff58d5f
33375000	0x2	RD	RF_RD_PORT_B	0xxxxx	0x403800fbcc0d21a0fffd00fb834dc7ef
33375000	0x6	WR	RF_WR_PORT	0xfffff	0x3c4dadd15f0730d4b0cfafdd7ff38d5b
33425000	0x5	RD	RF_RD_PORT_A	0xxxxx	0x1a0d1a1b07acde741a0eafef7ff58d6c
33425000	0x1e	RD	RF_RD_PORT_B	0xxxxx	0x15156237652a8bc8ca58ef106982cb59
33425000	0x14	WR	RF_WR_PORT	0xfffff	0x403c00edcc112190000000ed8350c7df
33475000	0x1	RD	RF_RD_PORT_A	0xxxxx	0x1a0d1a0c07acde661a0eafe07ff58d5e
33475000	0xb	RD	RF_RD_PORT_B	0xxxxx	0x978d8a22d576ca3d380a049a95c306cf
33475000	0x1f	WR	RF_WR_PORT	0xfffff	0x1a0d1a0207acde5a1a0eafd67ff58d52
33525000	0x1	RD	RF_RD_PORT_A	0xxxxx	0x1a0d1a0c07acde661a0eafe07ff58d5e
33525000	0x1b	RD	RF_RD_PORT_B	0xxxxx	0x938f8624d178c63f340c009c91c502e0
33525000	0x4	WR	RF_WR_PORT	0xfffff	0x40380104cc0d21a8fffd0104834dc7f7

Figure 3.38: RISCV-V RF Tracker Example

3.5 RISCV-V-CSR Agent

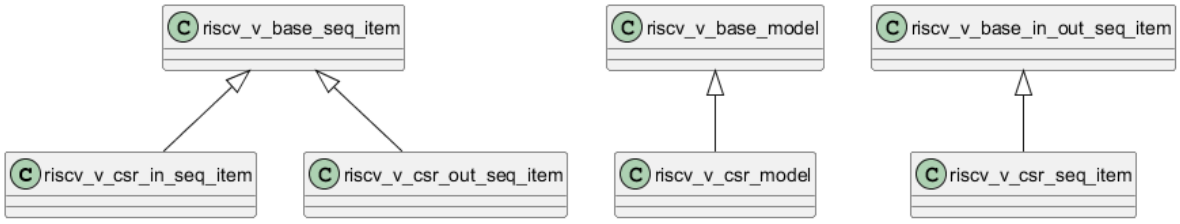


Figure 3.39: RISCV-V CSR components UML

The *RISCV-V-CSR Agent* is not intended to validate the *CSR* block as an independent block, the purpose of this validation agent is to be instantiated inside the *RISCV-V Agent* to model the *CSR* block, get the reference model of the *CSRs* and validate the *RISCV Vector Engine* as a whole. For that reason only the *Model* and *sequence item* are implemented.

3.5.1 RISCV-V CSR Model

The *RISCV-V CSR Model* models the behavior of the *CSR* block. It instantiates each one of the *CSRs* to hold the value and implements read and write methods for each one of the *CSRs*. As the RTL it implements two ways of writing the *CSRs*, one using the external csr interface that only supports the writing of one *CSR* per clock cycle and using the internal csr interface that supports the writing of all the *CSRs* at the same time. The *CSR* only can be written by one interface at a time, if both interfaces try to write the same *CSR* at the same time it raises an *UVM error*.

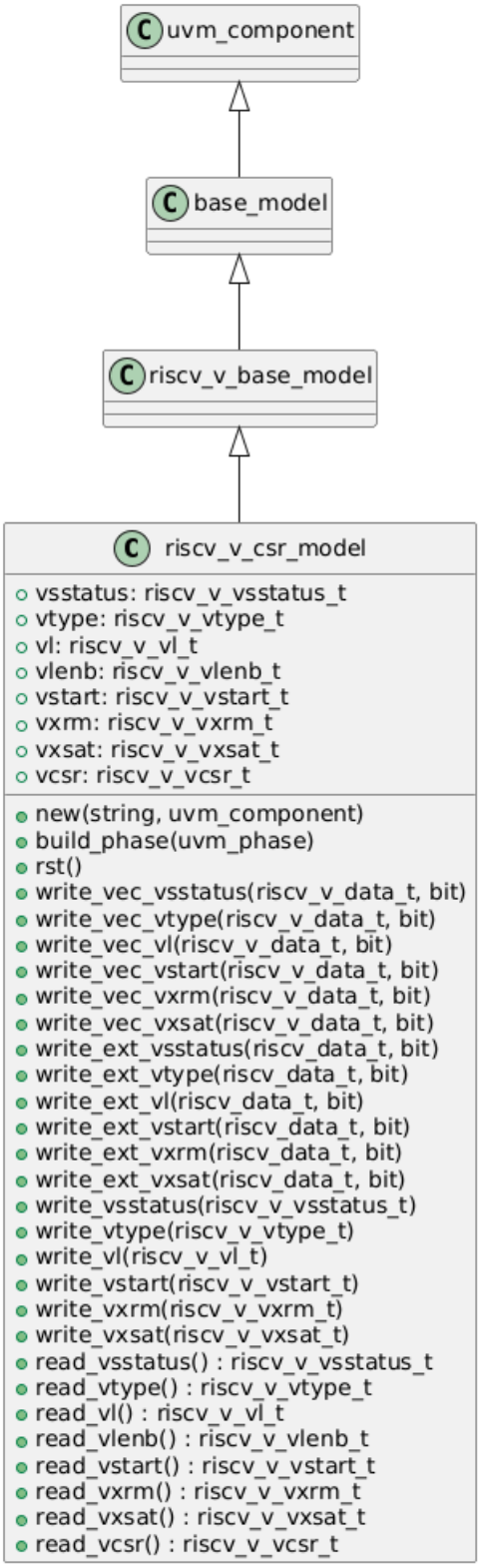


Figure 3.40: RISCV-V CSR Model UML

3.5.2 RISCV-V CSR Sequence Item

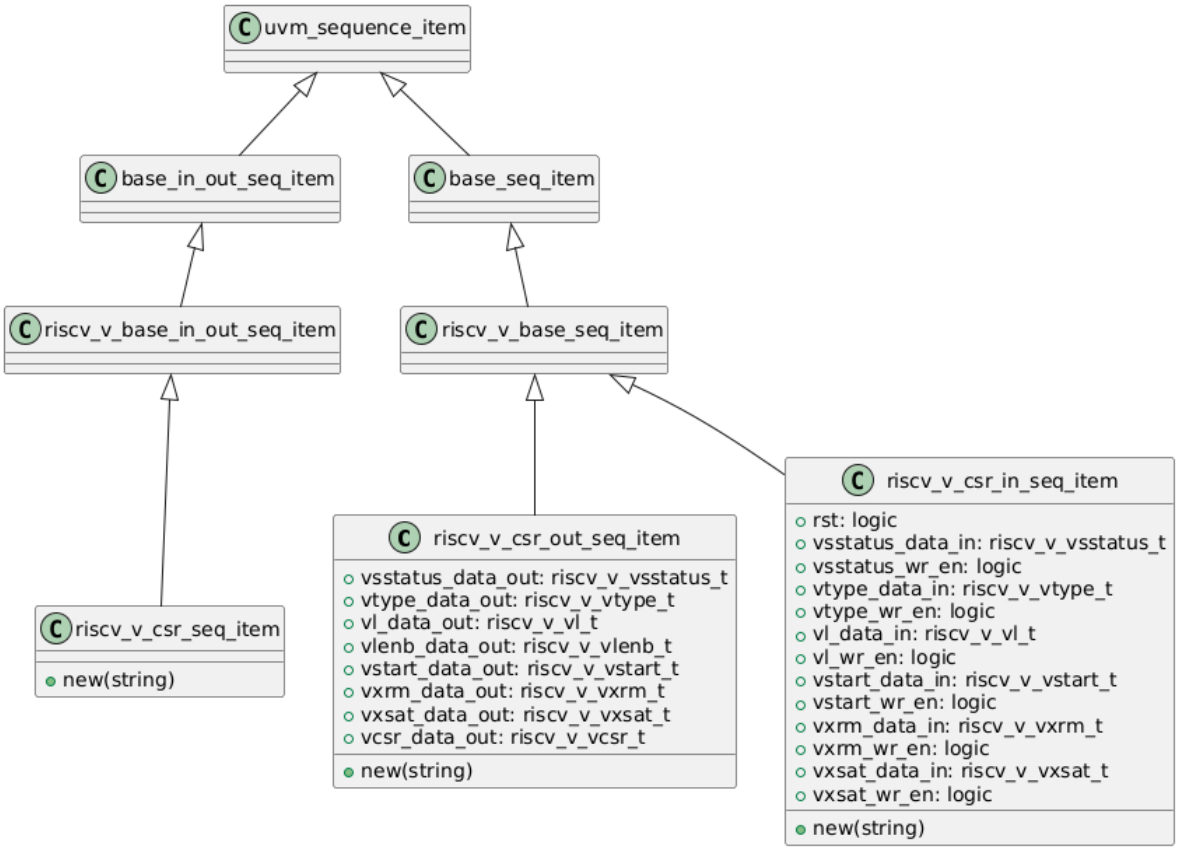


Figure 3.41: RISCV-V CSR Sequence item UML

The *sequence item* is used to send and get transactions from the *model*, the *riscv_v_csr_out_seq_item* is used to read the *CSRs* from the *model* while the *riscv_v_csr_in_seq_item* is used to write the *CSRs* of the *model*.

3.5.3 RISCV-V CSR Coverage

The *RISCV-V CSR Coverage* collector instantiates two cover groups, one to cover the write transactions and one to cover the read transactions. The write cover group covers that all the *write_enables* of the *CSRs* are asserted. Both cover groups, the write and read cover groups, create cover points and cover bins for each valid configuration that can hold the *CSR*. The cover point with its description are shown in Table 3.9.

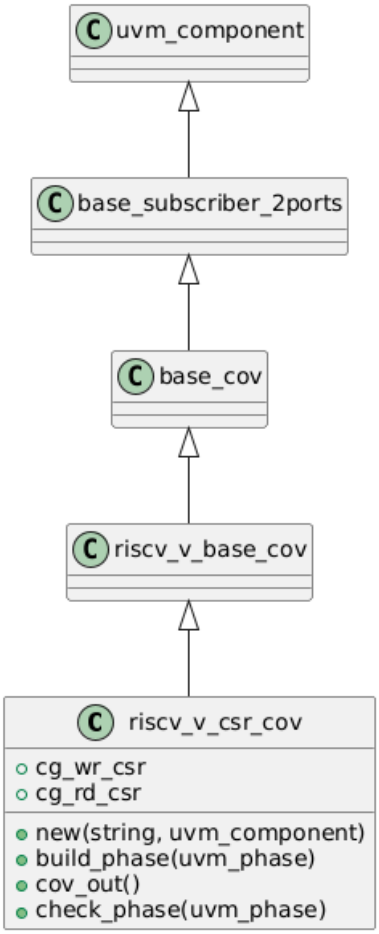


Figure 3.42: RISCV-V CSR Coverage UML

Table 3.9: RISCV-V write CSR Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
WR_CSR	wr_en_vsstatus	wr_en	Bin to cover that vstatus CSR is written
	wr_en_vtype	wr_en	Bin to cover that vtype CSR is written
	wr_en_vl	wr_en	Bin to cover that vl CSR is written
	wr_en_vstart	wr_en	Bin to cover that vstart CSR is written
	wr_en_vxrm	wr_en	Bin to cover that vxrm CSR is written
	wr_en_vxsat	wr_en	Bin to cover that vxsat CSR is written
	wr_vstatus	NO_VS VS	Bins to cover the valid configuration of vstatus CSR
	wr_vtype	VILL NO_VILL VMA NO_VMA VTA NO_VTA VSEW128 VSEW64 VSEW32 VSEW16 VSEW8 LMUL8 LMUL4 LMUL2 LMUL1	Bins to cover the valid configurations of vtype CSR
	wr_vl	len[17] len_greater_16	Creates 17 bins to cover the range of vl from 0 to 16 and 1 bin to cover all the values that are grater than 16
	wr_vstart	start[17] start_greater_16	Creates 17 bins to cover the range of vstart from 0 to 16 and 1 bin to cover all the values that are greater than 166
	wr_vxrm	RNU RNE RDN ROD	Bins to cover the valid configuration of vxrm CSR
	wr_vxsat	NO_SATURATE SATURATE	Bins to cover the valid configuration of vxsat CSR

Table 3.10: RISCV-V read CSR Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
RD_CSR	wr_vstatus	NO_VS VS	Bins to cover the valid configuration of vstatus CSR
	wr_vtype	VILL NO_VILL VMA NO_VMA VTA NO_VTA VSEW128 VSEW64 VSEW32 VSEW16 VSEW8 LMUL8 LMUL4 LMUL2 LMUL1	Bins to cover the valid configurations of vtype CSR
	wr_vl	len[17] len_greater_16	Creates 17 bins to cover the range of vl from 0 to 16 and 1 bin to cover all the values that are grater than 16
	wr_vstart	start[17] start_greater_16	Creates 17 bins to cover the range of vstart from 0 to 16 and 1 bin to cover all the values that are greater than 166
	wr_vxrm	RNU RNE RDN ROD	Bins to cover the valid configuration of vxrm CSR
	wr_vxsat	NO_SATURATE SATURATE	Bins to cover the valid configuration of vxsat CSR

3.5.4 RISCV-V CSR Tracker

The *RISCV-V CSR Tracker* keeps track of all the write CSR transactions and prints the transactions in the table, one example of the tracker is shown in Figure 3.43.

Time	CSR Written	CSR Data
1975000	vstart	0x0
2025000	vtype	0xc0
2075000	vl	0x42
2125000	vstart	0x0
2175000	vtype	0xc0
2225000	vl	0x10
2275000	vstart	0x0
2325000	vtype	0xc0
2375000	vl	0x10
2425000	vstart	0xe
2475000	vtype	0xc0
2525000	vl	0x10
2575000	vstart	0x2
2625000	vtype	0xc0
2675000	vl	0x10

Figure 3.43: RISCV-V CSR Tracker Example

3.6 RISCV-V- External CSR Agent



Figure 3.44: RISCV-V External CSR components UML

The purpose of the *RISCV-V External CSR Agent* is to model an external block capable of sending external *CSR* transactions to the *RISCV Vector Engine* to configure the engine from the exterior, not to validate an RTL block, for that reason, the *scoreboard* is not implemented.

3.6.1 RISCV-V External CSR BFM

The *RISCV-V External CSR BFM* is the main component of the *Agent*, it's the brain of the *Agent* and controls the external *CSR* writes to configure the *RISCV Vector Engine*.

The *BFM* instantiates a *rotator* to control the *CSRs* to be written, by default the *rotator* starts empty, meaning that no *CSRs* are written, the test should configure the rotator by adding the *CSR* and the data to write to the *CSR* to the *rotator*, the *rotator* iterates across all the *CSRs* that were added and sends one write transaction per cycle to the selected *CSR*.

For example, if the following *CSRs* are added to the *rotator*: *vstatus='h5*, *vstart='h2* and *vlen='h10*. The *BFM* sends the following transactions:

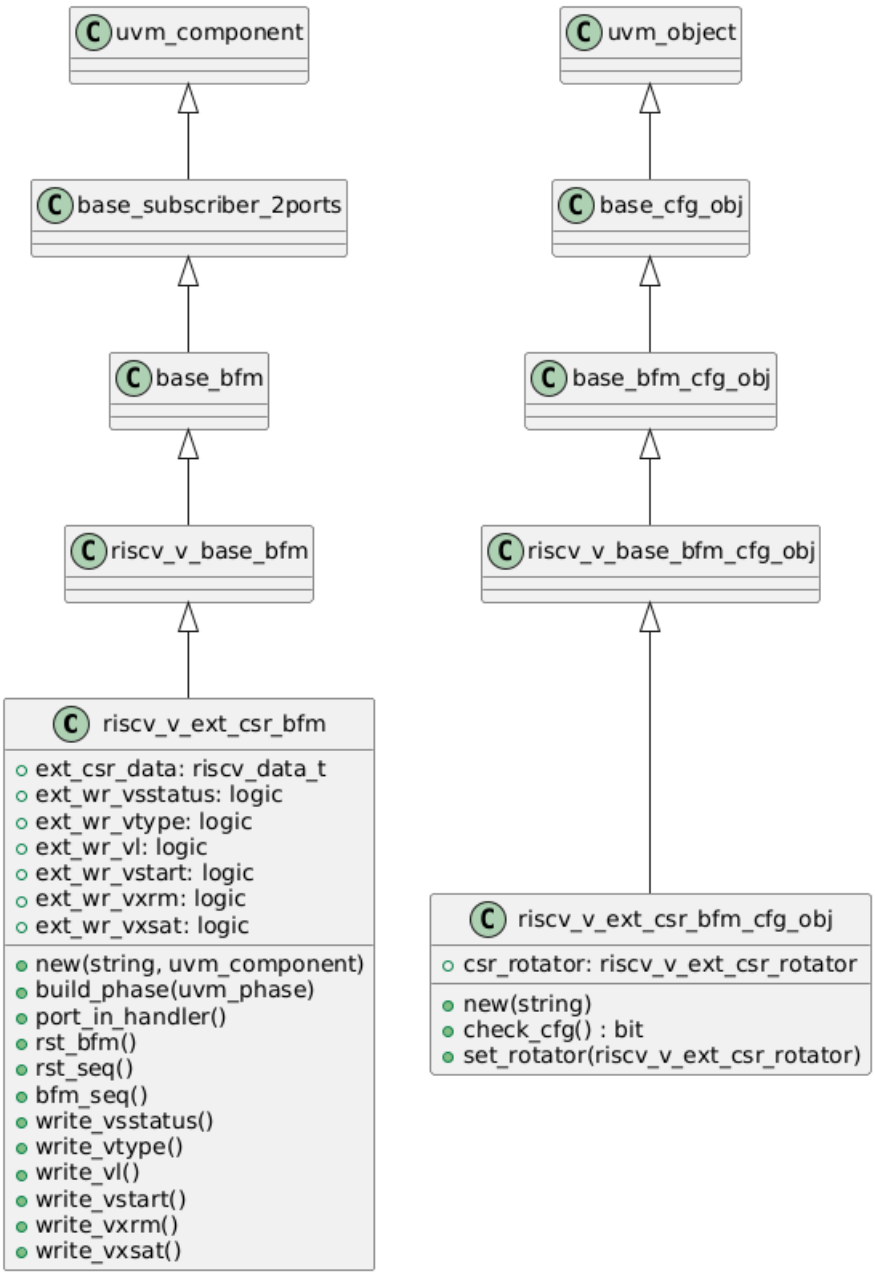


Figure 3.45: RISCV-V External CSR BFM UML

$$
\begin{aligned}
1. - ext_csr_wr_vstatus = 1'b1, ext_csr_wr_data =' h5 \\
2. - ext_csr_wr_vstart = 1'b1, ext_csr_wr_data =' h2 \\
3. - ext_csr_wr_vlen = 1'b1, ext_csr_wr_data =' h10 \\
4. - ext_csr_wr_vstatus = 1'b1, ext_csr_wr_data =' h5
\end{aligned}
\tag{3.1}
$$

The *BFM* continues iterating over the configured *CSRs* sending write transactions to the *RISCV Vector Engine*, it starts sending the transactions once the *BFM* is enabled and continues until it is disabled or the test finishes, the *rotator* can be updated at any time to change the data to be written or the *CSRs* to be updated.

3.6.2 RISCV-V External CSR Driver/Monitor

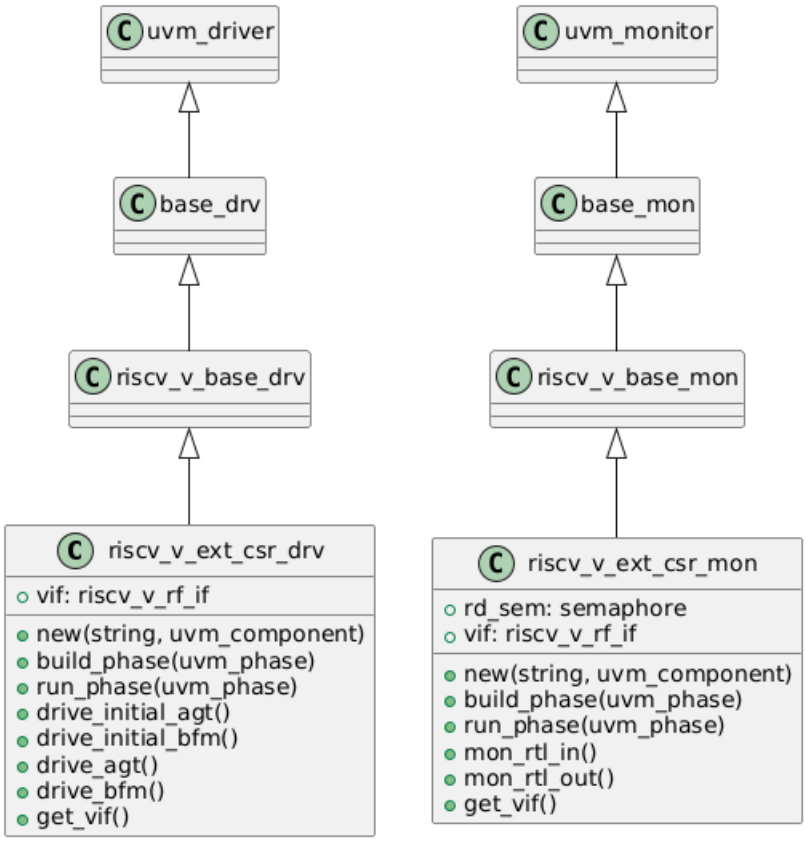


Figure 3.46: RISCV-V External CSR Driver/Monitor UML

The *RISCV-V External CSR Driver* transforms the external *CSR* transactions from *external CSR sequence item* to pin level. Since the write control signals are received during the *Instruction Decode* stage and the write data signal is received during the *Execute* stage the transaction can not be sent in one clock cycle, it needs to be split in two cycles, during the

first cycle the write enable signals are sent and the write data is buffered, during the second cycle the write data is sent. The *driver* is pipelined meaning that it supports back-to-back transactions.

The *monitor* behaves similarly as the *driver* splitting the transaction into two cycles, during the first cycle the control signals are sample and buffered, in the second cycle the data is sampled, the *external CSR sequence item* is completed and sent to the other components.

The signals driven and sampled by the driver and monitor are shown in 3.11.

Table 3.11: RISCV-V External CSR Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**reset**	logic	Active high reset signal
BFM/ RTL OUT	**ext_csr_data**	logic[31:0]	Data to be written to the CSR
	ext_wr_vsstatus	logic	If 1'b1 vsstatus is updated with ext_csr_data
	ext_wr_vtype	logic	If 1'b1 vtype is updated with ext_csr_data
	ext_wr_vl	logic	If 1'b1 vl is updated with ext_csr_data
	ext_wr_vstart	logic	If 1'b1 vstart is updated with ext_csr_data
	ext_wr_vxrm	logic	If 1'b1 vxrm is updated with ext_csr_data
	ext_wr_vxsat	logic	If 1'b1 vxsat is updated with ext_csr_data

3.6.3 RISCV-V External CSR Sequence Item-Sequence

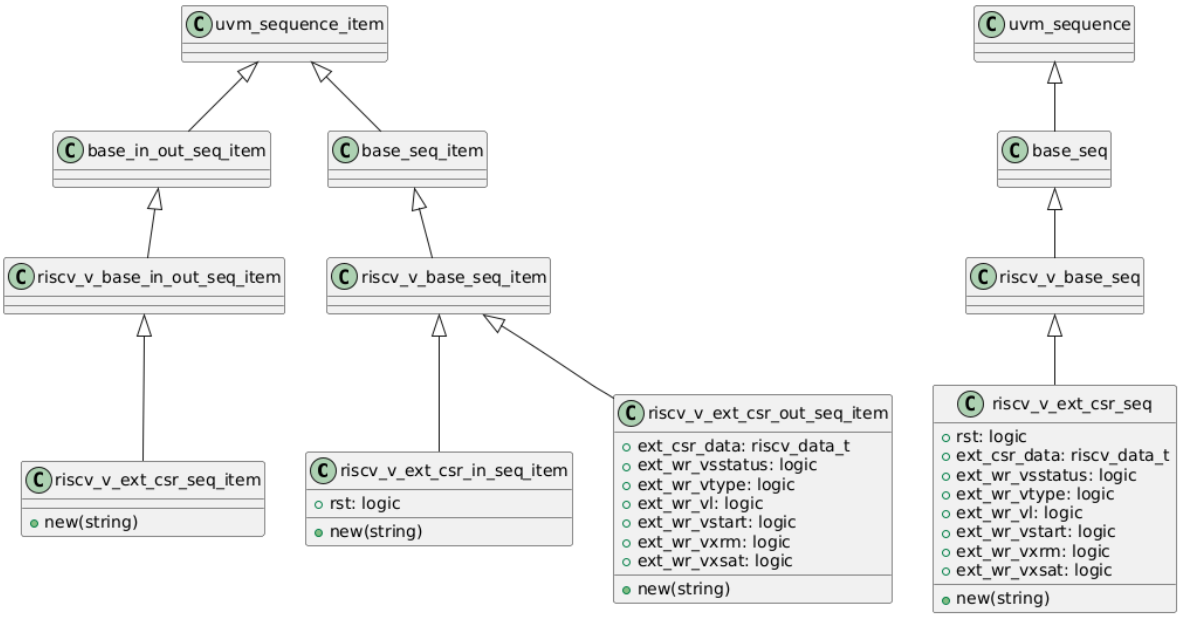


Figure 3.47: RISCV-V External CSR Sequence item-Sequence UML

The *external CSR sequence item* is used to communicate the *BFM* with the *Driver* to send the transactions to the *RISCV Vector Engine*. In this case, just the *riscv_v_ext_csr_out_seq_item*

is used since the *CSR* block only receives inputs from the exterior and does not send data to the exterior. This *sequence item* contains all the write enables and the write data that is used by the *Driver* to send the transaction.

3.7 RISCV-V-IF Agent



Figure 3.48: RISCV-V IF components UML

The *RISCV-V-IF Agent* is used to generate valid *Vector Instructions* and not to validate an RTL unit, for that reason the *scoreboard* is not implemented and just the *BFM* and *Model* are implemented to generate the *Instructions*.

3.7.1 RISCV-V Instruction

The *RISCV-V Instruction* component is used to get valid *Vector Instructions*, it implements one method per *Instruction*, the method is called to get the *Instruction*, it receives as parameter one bit to indicate that the *FUNCT3* field is set manually, if the bit is one the *Instruction* is generated using the *FUNCT3* set in the second argument of the method, the method returns a *Vector Instructions* with *Opcode*, *FUNCT6* and *FUNCT3* set accordingly to the method called and *VD*, *VS1*, *VS2* and *VM* randomized. All the instructions supported by this project and described in appendix *Operations implemented* are supported by this component.



Figure 3.49: RISCV-V Instruction UML

3.7.2 RISCV-V IF BFM

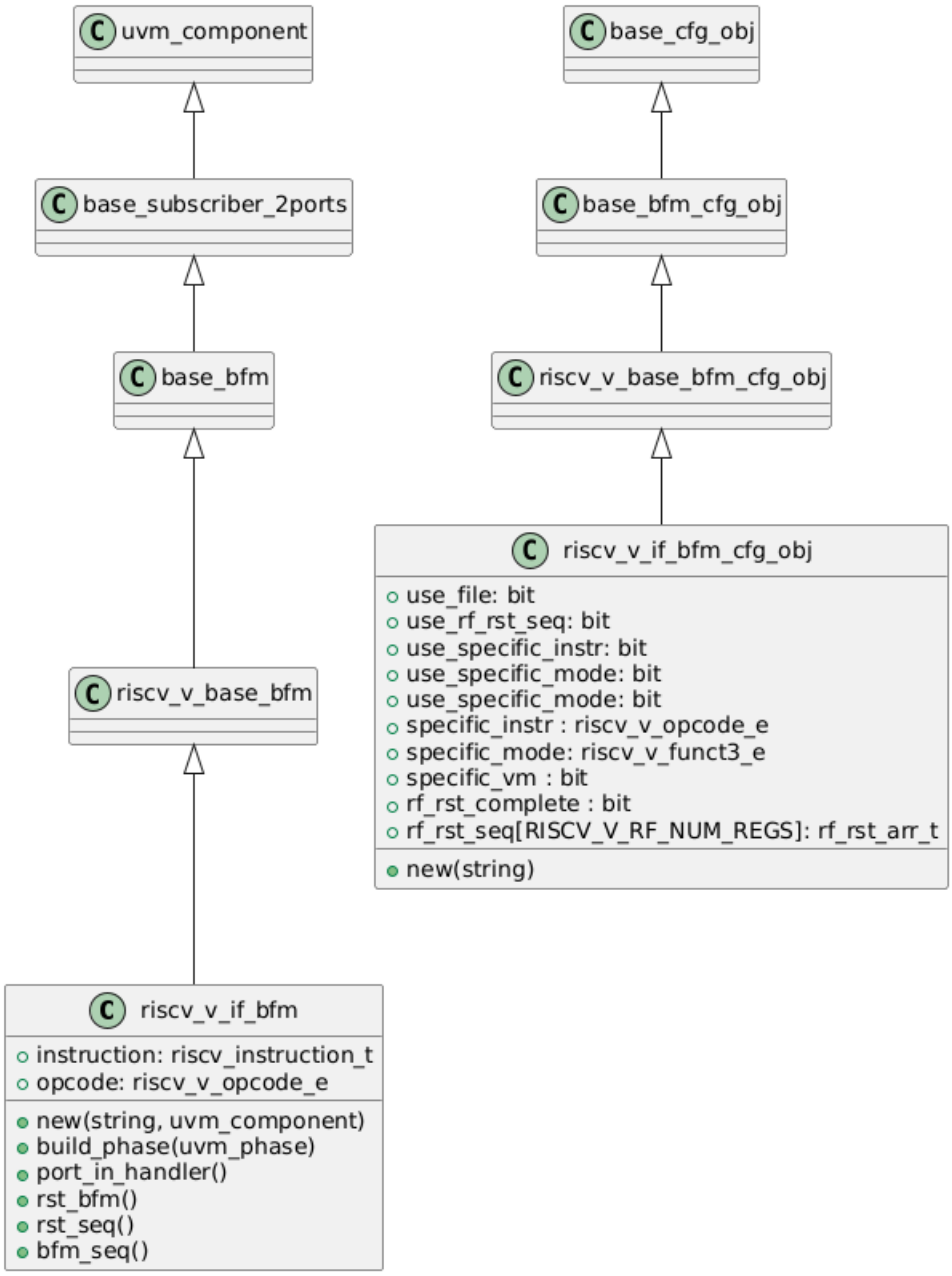


Figure 3.50: RISCV-V IF BFM UML

The *RISCV-V IF BFM* is the component that controls the *RISCV-V IF Model* to generate *Vector Instructions*. It can be configured by the test using the *riscv_v_if_bfm_cfg_obj*, it can be configured to generate random *Instructions* or to generate specific *Instructions*, the configuration object also includes an array of instructions that is used as a reset sequence, after a reset this array is read and the instructions in the array are executed, this is done to initialize all the registers of the *VRF* with random data, once all the instructions in the array are executed the *BFM* starts generating *Instructions* using its configuration, it generates one *instruction* per

cycle that is sent to the *RISCV Vector Engine*. The configuration knobs of the *RISCV-V IF BFM* are shown in Table 3.12.

Table 3.12: RISCV-V IF BFM configuration knobs

Knob	Type	Description
use_file	bit	Reads a file with the instructions to be executed instead of generating the instructions using the model
use_rf_rst_seq	bit	if 1'b1 the rf_rst_seq array is read to initialize the registers of the VRF after a reset
use_specific_instr	bit	If 1'b1 the instruction is generated using specific_instr instead of generating a random instruction
use_specific_mode	bit	If 1'b1 the funct3 field of the instruction is set to specific_mode instead of using a random funct3 when generating the instruction
use_specific_vm	bit	If 1'b1 the VM field of the instruction is set to specific_vm instead of using a random VM when generating the instruction
specific_instr	riscv_v_opcode_e	Instruction to be generated if use_specific_instr == 1'b1
specific_mode	riscv_v_funct3_e	funct3 to be used if use_specific_mode == 1'b1
specific_vm	logic	VM to be used if use_specific_vm == 1'b1
rf_rst_seq	rf_rst_arr_t[RISCV_V_RF_NUM_REGS]	Array holding the instructions to be executed to initialize the VRF after a reset
rf_rst_complete	bit	Status bit to indicate that the reset sequence has been completed and the BFM is generating the instructions.

3.7.3 RISCV-V IF Coverage

The RISCV-V IF Coverage collector samples the instruction that is sent to the *RISCV Vector Engine* to cover that all the implemented instructions described in appendix *Operations implemented* are executed. It creates a cover point to cover that all registers are used as destination and src1/src2 registers. It also creates a cover point to cover that all the valid encodings for *funct3* and funct6 are executed. Finally it creates one cross cover point between the *funct3*, funct6 and *mask* to cover that all the supported instructions are executed with all its variants, the variants that are invalid or are not supported are excluded from the coverage report. The cover points created with its description are shown in Table 3.13.

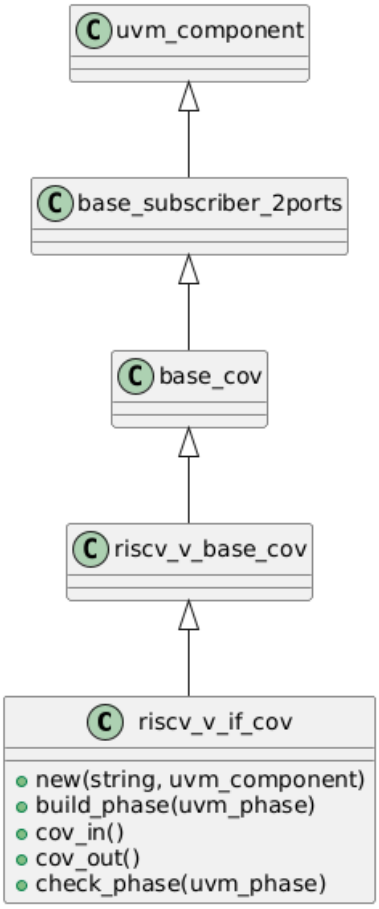


Figure 3.51: RISCV-V IF Coverage UML

Table 3.13: RISCV-V Instruction Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
INSTRUCTION	vec_opcode	vector_op	Bin to cover that the opcode of the RISC-V instruction corresponds to a vector instruction
	vec_dst	dst_addr[32]	Creates 32 bins to cover that all vector registers are selected as destination of the vector operation
	vec_funct3	OPIVV OPFVV OPMVV OPIVI OPIVX OPFVF OPMVX OPCFG	Bins to cover that all encodings of funct3 are selected
	vec_vs1	vs1_addr[32]	Creates 32 bins to cover that all vector registers are selected as src1 of the vector operation
	vec_vs2	vs2_addr[32]	Creates 32 bins to cover that all vector registers are selected as src2 of the vector operation
	vec_mask	masked unmasked	Bins to cover that masked and unmasked vector operations are executed
	vec_funct6	OPS[53]	Creates 1 bin per funct6 that is supported by the RISC-V vector engine
	MASK X FUNCT3 X FUNCT6	cross[161]	Cross cover to cover that all the valid formats of the supported vector instructions are executed

3.7.4 RISCV-V IF Driver/Monitor

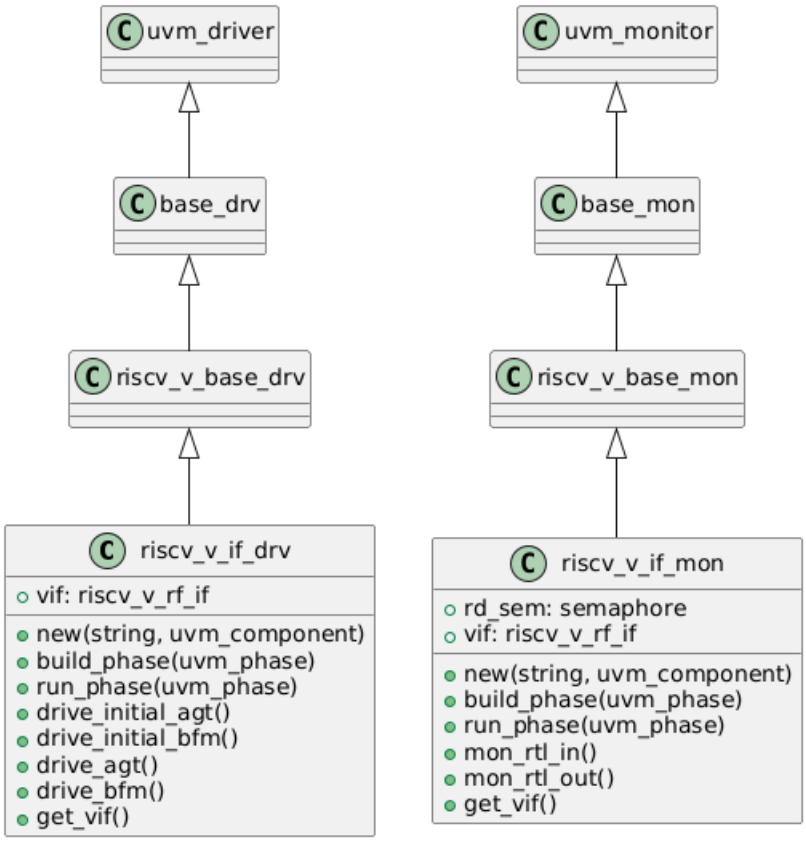


Figure 3.52: RISCV-V IF Driver/Monitor UML

The *RISCV-V IF Driver* converts the *Instructions* generated by the *RISCV-V IF BFM* to pin level *Instructions* to be sent to the RTL and the *RISCV-V IF Monitor* samples these *Instructions* to send them to the *RISCV-V IF Tracker* and *RISCV-V IF Coverage*. Both components, *Driver* and *Monitor* add the *opcode* to the signals driven/sampled, but this signal is an instrumentation signal only used for debug and is not synthesized. The signals driven/sampled are shown in Table 3.14.

Table 3.14: RISCV-V IF Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**rst**	logic	Active high reset signal
BFM/ RTL OUT	**instruction**	logic[31:0]	RISCV Instruction
	opcode	logic[5:0]	RISCV-V Internal opcode This is an instrumentation signal not used by real RTL.

3.7.5 RISCV-V IF Sequence Item-Sequence

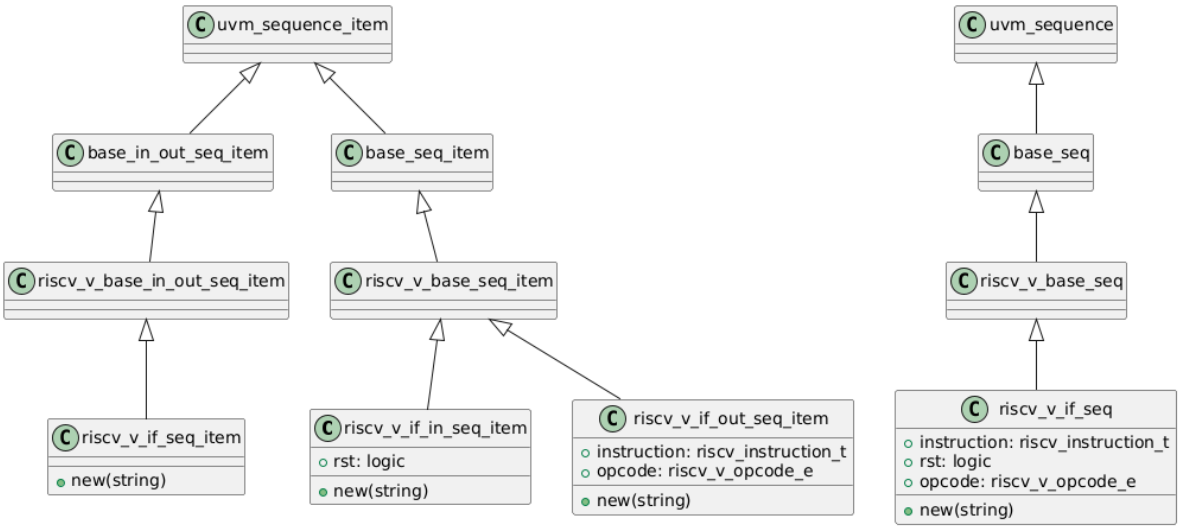


Figure 3.53: RISCV-V IF Sequence item-Sequence UML

The *RISCV-V IF Sequence item* is used to communicate the *RISCV-V IF BFM* with the *driver* and the *monitor* with the *tracker* and *coverage*. It just implements the *instruction* and *opcode*.

3.7.6 RISCV-V IF Model

The *RISCV-V IF Model* makes use of the *RISCV-V Instruction* component to generate *instructions*. The *Model* implements the method *get_instruction()* that receives as a parameter the configuration necessary to generate the *instruction*, the first bit indicates if a file should be read to generate the instruction from that file, the next three bits indicates if a specific *FUNCT6*, *FUNCT3* and *VM* should be set manually, if these bits are 1'b1 the following three arguments are the fields to be set, if the bits are 1'b0 the fields are randomized. The method also validates that the instruction generated is valid, for example, that the mask operations

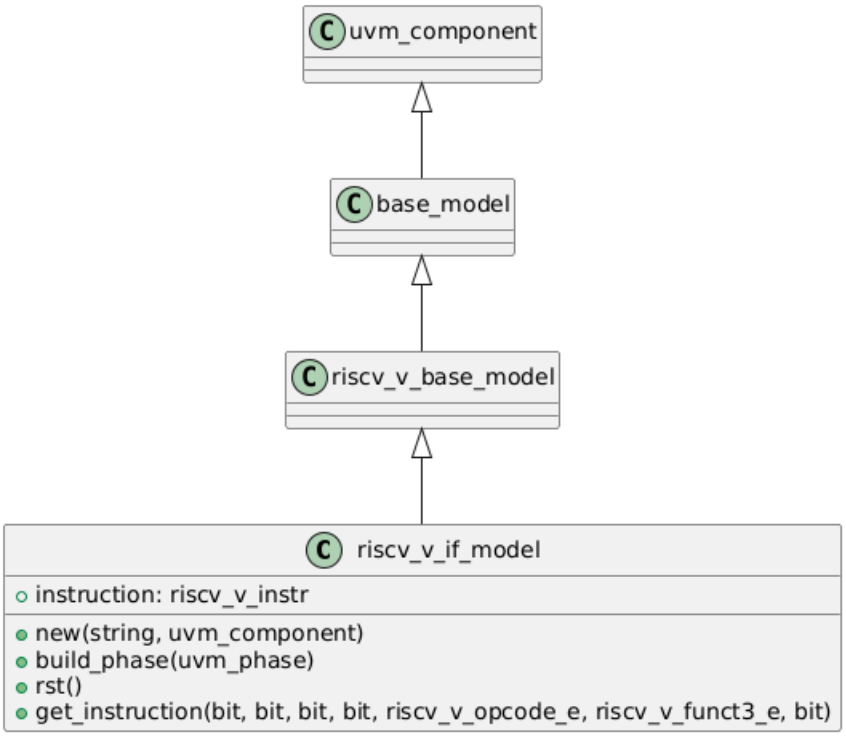


Figure 3.54: RISCV-V IF Model UML

can not have as the destination register the register zero, or some operations that are always masked or unmasked.

3.7.7 RISCV-V IF Tracker

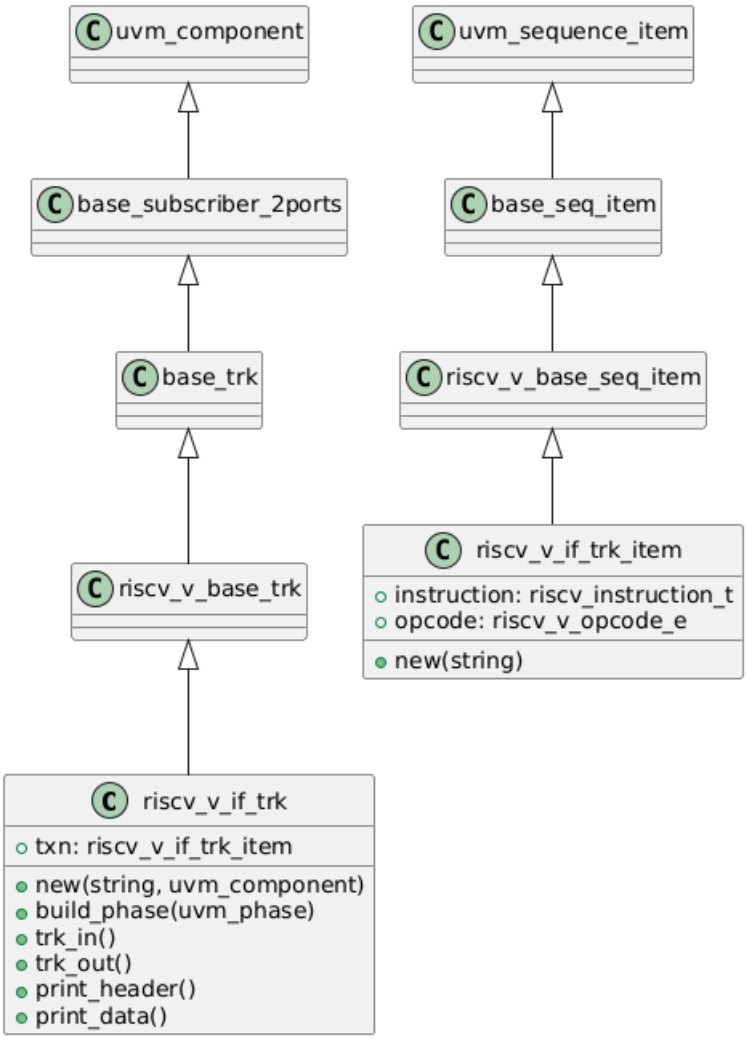


Figure 3.55: RISCV-V IF Tracker UML

The *RISCV-V IF Tracker* keeps track of all the *instructions* sent to the *RISCV Vector Engine* and prints the instructions in the table showing all the fields. One example of the generated table is shown in Figure 3.56.

Time	Instruction	funct6	vm	vs2	vs1	funct3	vd	opcode
25000	0x0	0x0	MASKED	0x0	0x0	OPIVV	0x0	NOP
325000	0x26003057	0x9	UNMASKED	0x0	0x0	OPIVI	0x0	BW_AND
1225000	0x2a053957	0xa	UNMASKED	0x0	0xa	OPIVI	0x12	BW_OR
1925000	0x2a053057	0xa	UNMASKED	0x0	0xa	OPIVI	0x0	BW_OR
1975000	0x9fa021d7	0x27	UNMASKED	0x1a	0x0	OPMVV	0x3	MULHS
2025000	0x9ecf2157	0x27	UNMASKED	0xc	0x1e	OPMVV	0x2	MULHS
2075000	0x9f4ea957	0x27	UNMASKED	0x14	0x1d	OPMVV	0x12	MULHS
2125000	0x9fb0ab57	0x27	UNMASKED	0x1b	0x1	OPMVV	0x16	MULHS
2175000	0x9efda2d7	0x27	UNMASKED	0xf	0x1b	OPMVV	0x5	MULHS
2225000	0x9f59a8d7	0x27	UNMASKED	0x15	0x13	OPMVV	0x11	MULHS
2275000	0x9e51afd7	0x27	UNMASKED	0x5	0x3	OPMVV	0x1f	MULHS
2325000	0x9e812d57	0x27	UNMASKED	0x8	0x2	OPMVV	0x1a	MULHS
2375000	0x9ec02b57	0x27	UNMASKED	0xc	0x0	OPMVV	0x16	MULHS
2425000	0x9e552a57	0x27	UNMASKED	0x5	0xa	OPMVV	0x14	MULHS
2475000	0x9ff4a257	0x27	UNMASKED	0x1f	0x9	OPMVV	0x4	MULHS
2525000	0x9f982957	0x27	UNMASKED	0x19	0x10	OPMVV	0x12	MULHS
2575000	0x9f952b57	0x27	UNMASKED	0x19	0xa	OPMVV	0x16	MULHS
2625000	0x9f41a557	0x27	UNMASKED	0x14	0x3	OPMVV	0xa	MULHS
2675000	0x9e1a2857	0x27	UNMASKED	0x1	0x14	OPMVV	0x10	MULHS
2725000	0x9fe3abd7	0x27	UNMASKED	0x1e	0x7	OPMVV	0x17	MULHS
2775000	0x9ef42a57	0x27	UNMASKED	0xf	0x8	OPMVV	0x14	MULHS
2825000	0x9f3fa657	0x27	UNMASKED	0x13	0x1f	OPMVV	0xc	MULHS
2875000	0x9e5f28d7	0x27	UNMASKED	0x5	0x1e	OPMVV	0x11	MULHS

Figure 3.56: RISCV-V Instruction tracker example

3.8 RISCV-V-ALU Agent



Figure 3.57: RISCV-V ALU components UML

The *RISCV-V-ALU Agent* is used to validate the different ALUs of the *Execution Unit*. To validate the ALUs the same *AGENT* is used and only one instance of the *AGENT* is needed to validate the three ALUs, the *AGENT* is capable of classifying the *Vector Instruction* and act accordingly.

3.8.1 RISCV-V ALU Coverage

The *RISCV-V ALU Coverage* collector creates three cover groups, one for each ALU. Each group creates two cover points with eight cover bins each one to cover that *srca* and *srcb* are executed with different ranges of data. It also creates cover points to cover all the valid *OSIZEs* and *LENs*.

The cover groups creates cover points to cover that each control signal of each ALU is asserted, and creates independent cover points to cover that the combination of control signals needed to implement a specific operation is asserted. Finally, it also creates cross cover points between the control signals, the *OSIZE* and *LEN* to cover that all the supported operations are executed with different combinations of *OSIZE* and *LEN*.

The cover points with its description of the logic, arithmetic and permutation ALUs are shown in Tables 3.15, 3.16 and 3.17 respectively.

Cover Group	Cover Point	Cover Bin	Description
	sra	srca[8]	Creates 8 cover bins to cover all the range of the possible values of srca
	srcb	srcb[8]	Creates 8 cover bins to cover all the range of the possible values of srcb

Table 3.16 continued from previous page

Cover Group	Cover Point	Cover Bin	Description
	osize	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	Creates 1 cover bin per valid OSIZE
	src_osize	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	Creates 1 cover bin per valid SRC_OSIZE Only valid for zero/sign extend operations
	dst_osize	OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	Creates 1 cover bin per valid DST_OSIZE Only valid for zero/sign extend operations
	len	len[17] len_greater_16	Creates 17 cover bins to cover all the possible values of len. One bine per value [0-16] and one extra bin to cover if len is greater than 16
	is_reduct	is_reduct	Covers that the control signal is_reduct is asserted
	is_add	is_add	Covers that the control signal is_add is asserted
	is_sub	is_sub	Covers that the control signal is_sub is asserted
	is_mul	is_mul	Covers that the control signal is_mul is asserted
	is_zero_ext	is_zero_ext	Covers that the control signal is_zero_ext is asserted
	is_sign_ext	is_sign_ext	Covers that the control signal is_sign_ext is asserted
	is_set_equal	is_set_equal	Covers that the control signal is_set_equal is asserted
	is_set_nequal	is_set_nequal	Covers that the control signal is_set_nequal is asserted
	is_set_less	is_set_less	Covers that the control signal is_set_less is asserted
	is_set_greater	is_set_greater	Covers that the control signal is_set_greater is asserted
	is_max	is_max	Covers that the control signal is_max is asserted
	is_min	is_min	Covers that the control signal is_min is asserted
	is_high	is_high	Covers that the control signal is_high is asserted
	is_signed	is_signed	Covers that the control signal is_signed is asserted
	use_carry	use_carry	Covers that the control signal use_carryy is asserted
	carry_in	carry_in[16]	Creates 16 cover bins to cover that each bit of carry_in is asserted
	op_VADC	VADC	Covers that the control signals needed for VADC operation are asserted
	op_VADD	VADD	Covers that the control signals needed for VADD operation are asserted
	op_VREDADD	VREDADD	Covers that the control signals needed for VREDADD operation are asserted
	op_VSUBB	VSUBB	Covers that the control signals needed for VSUBB operation are asserted
	op_VSUB	VSUB	Covers that the control signals needed for VSUB operation are asserted
	op_VMINS	VMINS	Covers that the control signals needed for VMINS operation are asserted
	op_VREDMINS	VREDMINS	Covers that the control signals needed for VREDMINS operation are asserted
	op_VMINU	VMINU	Covers that the control signals needed for VMINU operation are asserted
	op_VREDMINU	VREDMINU	Covers that the control signals needed for VREDMINU operation are asserted
	op_VMAXS	VMAXS	Covers that the control signals needed for VMAXS operation are asserted
	op_VREDMAXS	VREDMAXS	Covers that the control signals needed for VREDMAXS operation are asserted
	op_VMAXU	VMAXU	Covers that the control signals needed for VMAXU operation are asserted
	op_VREDMAXU	VREDMAXU	Covers that the control signals needed for VREDMAXU operation are asserted
	op_VSEQ	VSEQ	Covers that the control signals needed for VSEQ operation are asserted
	op_VSNEQ	VSNEQ	Covers that the control signals needed for VSNEQ operation are asserted
	op_VSLE	VSLE	Covers that the control signals needed for VSLE operation are asserted
	op_VSLEU	VSLEU	Covers that the control signals needed for VSLEU operation are asserted
	op_VSLT	VSLT	Covers that the control signals needed for VSLT operation are asserted
	op_VSLTU	VSLTU	Covers that the control signals needed for VSLTU operation are asserted
	op_VSGT	VSGT	Covers that the control signals needed for VSGT operation are asserted
	op_VSGTU	VSGTU	Covers that the control signals needed for VSGTU operation are asserted
	op_VMULLS	VMULLS	Covers that the control signals needed for VMULLS operation are asserted
	op_VMULHS	VMULHS	Covers that the control signals needed for VMULHS operation are asserted
	op_VMULHU	VMULHU	Covers that the control signals needed for VMULHU operation are asserted
	op_VZEXT	VZEXT	Covers that the control signals needed for VZEXT operation are asserted
	op_VSEXT	VSEXT	Covers that the control signals needed for VSEXT operation are asserted
	VADC X OSIZE X LEN	cross[90]	Cross cover to check that VADC operation is executed with all the OSIZE and LEN combinations
	VADD X OSIZE X LEN	cross[90]	Cross cover to check that VADD operation is executed with all the OSIZE and LEN combinations
	VREDADD X OSIZE X LEN	cross[72]	Cross cover to check that VREDADD operation is executed with all the OSIZE and LEN combinations
	VSUBB X OSIZE X LEN	cross[90]	Cross cover to check that VSUBB operation is executed with all the OSIZE and LEN combinations
	VSUB X OSIZE X LEN	cross[90]	Cross cover to check that VSUB operation is executed with all the OSIZE and LEN combinations
	VMINS X OSIZE X LEN	cross[90]	Cross cover to check that VMINS operation is executed with all the OSIZE and LEN combinations
	VREDMINS X OSIZE X LEN	cross[72]	Cross cover to check that VREDMINS operation is executed with all the OSIZE and LEN combinations
	VMINU X OSIZE X LEN	cross[90]	Cross cover to check that VMINU operation is executed with all the OSIZE and LEN combinations
	VREDMINU X OSIZE X LEN	cross[72]	Cross cover to check that VREDMINU operation is executed with all the OSIZE and LEN combinations
	VMAXS X OSIZE X LEN	cross[90]	Cross cover to check that VMAXS operation is executed with all the OSIZE and LEN combinations
	VREDMAXS X OSIZE X LEN	cross[72]	Cross cover to check that VREDMAXS operation is executed with all the OSIZE and LEN combinations
	VMAXU X OSIZE X LEN	cross[90]	Cross cover to check that VMAXU operation is executed with all the OSIZE and LEN combinations
	VREDMAXU X OSIZE X LEN	cross[72]	Cross cover to check that VREDMAXU operation is executed with all the OSIZE and LEN combinations
	VSEQ X OSIZE X LEN	cross[90]	Cross cover to check that VSEQ operation is executed with all the OSIZE and LEN combinations
	VSNEQ X OSIZE X LEN	cross[90]	Cross cover to check that VSNEQ operation is executed with all the OSIZE and LEN combinations
	VSLE X OSIZE X LEN	cross[90]	Cross cover to check that VSLE operation is executed with all the OSIZE and LEN combinations
	VSLEU X OSIZE X LEN	cross[90]	Cross cover to check that VSLEU operation is executed with all the OSIZE and LEN combinations

Arithmetic **Table 3.16 continued from previous page**

Cover Group	Cover Point	Cover Bin	Description
	VSGT X OSIZE X LEN	cross[90]	Cross cover to check that VSGT operation is executed with all the OSIZE and LEN combinations
	VSGTU X OSIZE X LEN	cross[90]	Cross cover to check that VSGTU operation is executed with all the OSIZE and LEN combinations
	VMULLS X OSIZE X LEN	cross[90]	Cross cover to check that VMULLS operation is executed with all the OSIZE and LEN combinations
	VMULHS X OSIZE X LEN	cross[90]	Cross cover to check that VMULHS operation is executed with all the OSIZE and LEN combinations
	VMULHU X OSIZE X LEN	cross[90]	Cross cover to check that VMULHU operation is executed with all the OSIZE and LEN combinations
	VZEXT X SRC_OSIZE X DST_OSIZE X LEN	cross[162]	Cross cover to check that VZEXT operation is executed with all the SRC_OSIZE DST_OSIZE and LEN combinations
	VSEXT X SRC_OSIZE X DST_OSIZE X LEN	cross[162]	Cross cover to check that VSEXT operation is execited with all the SRC_OSIZE DST_OSIZE and LEN combinations

Table 3.16: Arithmetic ALU Functional Coverage

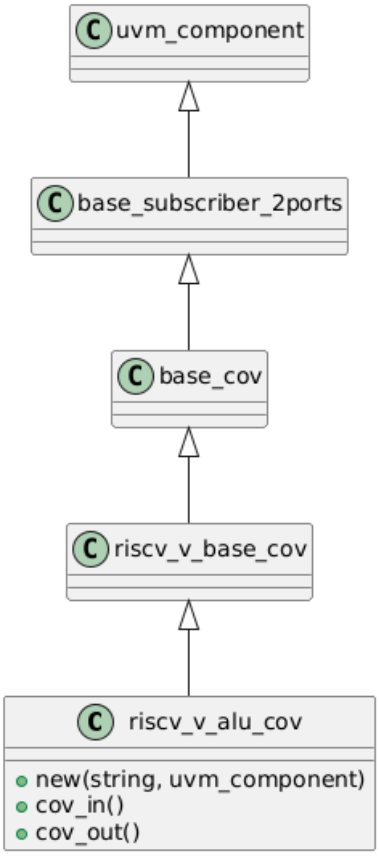


Figure 3.58: RISCV-V ALU Coverage UML

Table 3.15: Logic ALU Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
Logic	srca	srca[8]	Creates 8 cover bins to cover all the range of the possible values of srca
	srcb	srcb[8]	Creates 8 cover bins to cover all the range of possible values of srcb
	Osize	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	Creates 1 cover bin per valid OSIZE
	Len	len[17] len_greater_16	Creates 17 cover bins to cover all the possible values of len. One bine per value [0-16] and one extra bin to cover if len is greater than 16
	cp_mask_valid	mask_valid[16]	Creates 16 cover bins to cover that each bit of mask_valid is asserted
	is_reduct	is_reduct	Covers that the control signal is_reduct is asserted
	is_and	is_and	Covers that the control signal is_and is asserted
	is_or	is_or	Covers that the control signal is_or is asserted
	is_xor	is_xor	Covers that the control signal is_xor is asserted
	is_mask	is_mask	Covers that the control signal is_mask is asserted
	is_shift	is_shift	Covers that the control signal is_shift is asserted
	is_left	is_left	Covers that the control signal is_left is asserted
	is_arith	is_arith	Covers that the control signal is_arith is asserted
	is_negate_result	is_negate_result	Covers that the control signal is_negate_result is asserted
	is_negate_srca	is_negate_srca	Covers that the control signal is_negate_srca is asserted
	op_VAND	VAND	Covers that the control signals needed for VAND operation are asserted
	op_VREDAND	VREDAND	Covers that the control signals needed for VREDAND operation are asserted
	op_VOR	VOR	Covers that the control signals needed for VOR operation are asserted
	op_VREDOR	VREDOR	Covers that the control signals needed for VREDOR operation are asserted
	op_VXOR	VXOR	Covers that the control signals needed for VXOR operation are asserted
	op_VREDXOR	VREDXOR	Covers that the control signals needed for VREDXOR operation are asserted
	op_VSLL	VSLL	Covers that the control signals needed for VSLL operation are asserted
	op_VSRL	VSRL	Covers that the control signals needed for VSRL operation are asserted
	op_VSRA	VSRA	Covers that the control signals needed for VSRA operation are asserted
	op_VMAND	VMAND	Covers that the control signals needed for VMAND operation are asserted
	op_VMNAND	VMNAND	Covers that the control signals needed for VMNAND operation are asserted
	op_VMANDN	VMANDN	Covers that the control signals needed for VMANN operation are asserted
	op_VMOR	VMOR	Covers that the control signals needed for VMOR operation are asserted
	op_VMNOR	VMNOR	Covers that the control signals needed for VMNOR operation are asserted
	op_VMORN	VMORN	Covers that the control signals needed for VMORN operation are asserted
	op_VMXOR	VMXOR	Covers that the control signals needed for VMXOR operation are asserted
	op_VMXNOR	VMXNOR	Covers that the control signals needed for VMXNOR operation are asserted
	VAND X OSIZE X LEN	Cross[90]	Cross cover to check that VAND operation is executed with all the OSIZE and LEN combinations
	VREDAND X OSIZE X LEN	Cross[90]	Cross cover to check that VREDAND operation is executed with all the OSIZE and LEN combinations
	VOR X OSIZE X LEN	Cross[90]	Cross cover to check that VOR operation is executed with all the OSIZE and LEN combinations
	VREDOR X OSIZE X LEN	Cross[90]	Cross cover to check that VREDOR operation is executed with all the OSIZE and LEN combinations
	VXOR X OSIZE X LEN	Cross[90]	Cross cover to check that VXOR operation is executed with all the OSIZE and LEN combinations
	VREDXOR X OSIZE X LEN	Cross[90]	Cross cover to check that VREDXOR operation is executed with all the OSIZE and LEN combinations
	VSLL X OSIZE X LEN	Cross[90]	Cross cover to check that VSLL operation is executed with all the OSIZE and LEN combinations
	VSRL X OSIZE X LEN	Cross[90]	Cross cover to check that VSRL operation is executed with all the OSIZE and LEN combinations
	VSRA X OSIZE X LEN	Cross[90]	Cross cover to check that VSRA operation is executed with all the OSIZE and LEN combinations
	VMAND X MASK_VALID	Cross[16]	Cross cover to check that VMAND operation is executed with all the MASK_VALID bits
	VMNAND X MASK_VALID	Cross[16]	Cross cover to check that VMNAND operation is executed with all the MASK_VALID bits
	VMANDN X MASK_VALID	Cross[16]	Cross cover to check that VMANDN operation is executed with all the MASK_VALID bits
	VMOR X MASK_VALID	Cross[16]	Cross cover to check that VMOR operation is executed with all the MASK_VALID bits
	VMNOR X MASK_VALID	Cross[16]	Cross cover to check that VMNOR operation is executed with all the MASK_VALID bits
	VMORN X MASK_VALID	Cross[16]	Cross cover to check that VMORN operation is executed with all the MASK_VALID bits
	VMXOR X MASK_VALID	Cross[16]	Cross cover to check that VMXOR operation is executed with all the MASK_VALID bits
	VMXNOR X MASK_VALID	Cross[16]	Cross cover to check that VMXNOR operation is executed with all the MASK_VALID bits

Table 3.17: Permutation ALU Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
Permutation	sra	srca[8]	Creates 8 cover bins to cover all the range of the possible values of srca
	srcb	srcb[8]	Creates 8 cover bins to cover all the range of the possible values of srcb
	osize	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	Creates 1 cover bin per valid OSIZE
	is_i2v	is_i2v	Covers that the control signal is_i2v is asserted
	is_v2i	is_v2i	Covers that the control signal is_v2i is asserted
	op_I2V	I2V	Covers that the control signals needed for I2V operation are asserted
	op_V2I	V2I	Covers that the control signals needed for V2I operation are asserted
	I2V X OSIZE	cross[5]	Cross cover to check that I2V operation is executed with all the OSIZE combinations
	V2I X OSIZE	cross[5]	Cross cover to check that V2I operation is executed with all the OSIZE combinations

3.8.2 RISCV-V ALU Driver/Monitor



Figure 3.59: RISCV-V ALU Driver/Monitor UML

The *RISCV-V ALU Driver* transforms the *RISCV-V ALU sequence item* to pin level, but

unlike the other *drivers* this translation is not a one to one translation and it is more complex, inside the *sequence item* the operation is stored encoded using an enumerated type, the control signals needed to execute that operation are not included in the *sequence item* for that reason the *RISCV-V ALU Driver* needs to decode the operation and generate the control signals that are sent to the ALUs.

The *RISCV-V ALU Monitor* performs the inverse flow of the *driver*, it reads the control signals that are received by the ALUs and depending on the control signals encodes the operation that is executed to add it to the *sequence item* and send it the other components.

The signals driven/sampled by the *driver*/*monitor* are shown in Figures 3.18, 3.19 and 3.20.

Table 3.18: RISCV-V Logic ALU Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**is_reduct**	logic	Indicates if is a reduction operation
	is_and	logic	Indicates if is a logic/mask and operation
	is_mask	logic	Indicates if is a mask operation
	is_or	logic	Indicates if is a logic/mask or operation
	is_xor	logic	Indicates if is a logic/mask xor operation
	is_shift	logic	Indicates if is a shift operation
	is_left	logic	Indicates if is a shift left operation
	is_arith	logic	Indicates if is an arithmetic operation
	srca	logic[127:0]	ALU operand A
	srcb	logic[127:0]	ALU operand B
	dst_osize_vector	logic[4:0]	Osize of operation
	is_greater_osize_vector	logic[4:0]	Greater osize vector
BFM/ RTL OUT	**result**	logic[127:0]	ALU result

Table 3.19: RISCV-V Arithmetic ALU Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**is_reduct**	logic	Indicates if is a reduction operation
	is_add	logic	Indicates if adder is used
	is_sub	logic	Indicates if adder is used as subtractor
	is_mul	logic	Indicates if is a multiplier operation
	is_zero_ext	logic	Indicates if is a zero extend operation
	is_sign_ext	logic	Indicates if is a sign extend operation
	is_set_equal	logic	Indicates if is a set on equal operation
	is_set_nequal	logic	Indicates if is a set on not equal operation
	is_set_less	logic	Indicates if is a set on less operation
	is_set_greater	logic	Indicates if is a set on greater operation
	is_max	logic	Indicates if is a max operation
	is_min	logic	Indicates if is a min operation
	is_signed	logic	Indicates if operands are signed
	is_high	logic	Indicates if returns high part of result
	use_carry	logic	Indicates if operation uses mask bus as carry_in bus
	srca	logic[127:0]	ALU operand A
	srcb	logic[127:0]	ALU operand B
	carry_in	logic[15:0]	carry_in/mask Bus
	dst_osize_vector	logic[4:0]	Osize of operation destination
	src_osize_vector	logic[4:0]	Osize off operation source
	is_greater_osize_vector	logic[4:0]	Is greater osize vector
	is_lesss_osize_vector	logic[4:0]	Is less osize vector
BFM/ RTL OUT	**result**	logic[127:0]	ALU result
	cf	logic[15:0]	Carry flag bus
	zf	logic[15:0]	Zero flag bus
	of	logic[15:0]	Overflow flag bus

Table 3.20: RISCV-V Permutation ALU Driver/Monitor signals

Driver Mode / Monitor Port	Name	Type	Description
AGENT/ RTL IN	**is_i2v**	logic	Indicates if is an integer to vector operation
	is_v2i	logic	Indicates if is a vector to integer operation
	srca	logic[127:0]	ALU operand A
	srcb	logic[127:0]	ALU operand B
BFM/ RTL OUT	**vector_data_out**	logic[127:0]	ALU integer to vector result
	integer_data_out	logic[31:0]	ALU vector to integer result

3.8.3 RISCV-V ALU Sequence Item-Sequence

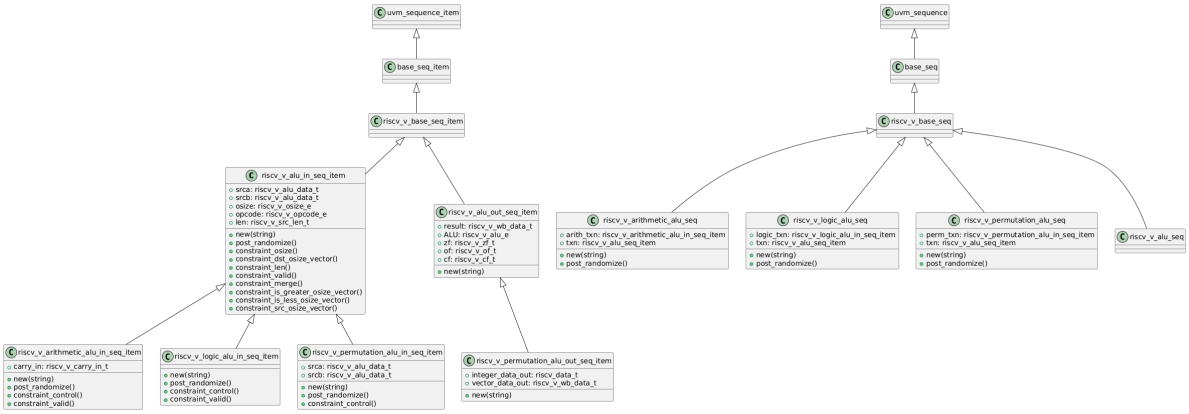


Figure 3.60: RISCV-V ALU Sequence item-Sequence UML

The *RISCV-V ALU Sequence item* is used to generate transactions from the test to be sent to the *driver* to be decoded and sent to the RTL. As mentioned in the *driver* section the *sequence item* does not hold all the control signals needed by the RTL to execute the operation, they just hold some elements like the sources, the *OSIZE*, the opcode and the length of the operation.

3.8.4 RISCV-V ALU Model

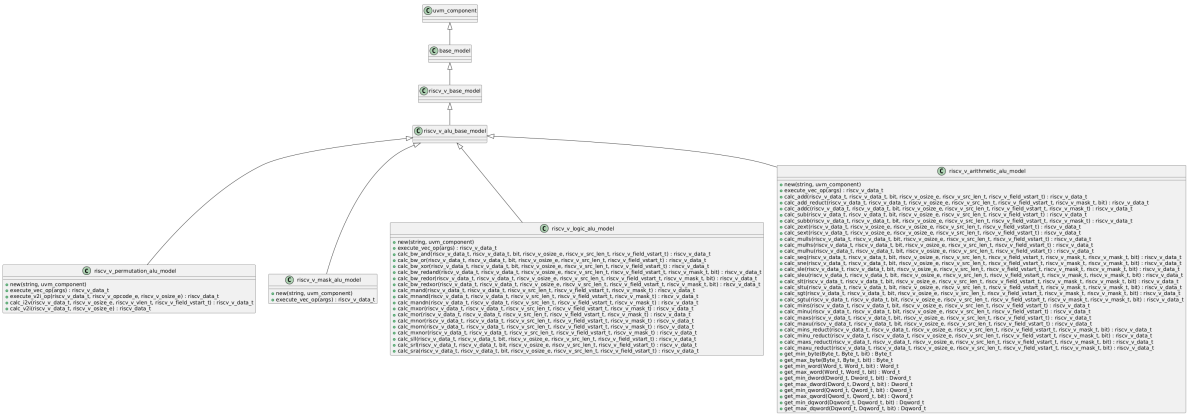


Figure 3.61: RISCV-V ALU Model UML

The *RISCV-V ALU Model* is used to get the expected result of the operation executed by the ALU. Each ALU has its own *model* with the instructions that are executed by that ALU. The *models* implement many methods to get the expected result, one method per instruction implemented, they receive as parameter the sources of the operation, the *OSIZE*, the length and the start index of the operation and return the result of the operation.

These *models* can be used by the *scoreboard* to get the expected result of the operation or by the *BFM* to replace the ALU RTL with a verification component to validate another component independently.

3.8.5 RISCV-V ALU Scoreboard

The *RISCV-V ALU Scoreboard* is the component responsible for checking the correctness of the different ALUs. The *Scoreboard* receives the operation to be executed from the *monitor* and calls the corresponding method to calculate the expected result of the operation, one method is called per operation but the same method supports multiple *OSIZEs*.

This *Scoreboard* only receives the sources from the *monitor* and operates using these sources, it is agnostic to the type of source, if it is a source coming from an immediate value, a vector register or an integer register. It also calculates the result as the RTL ALU is supposed to compute the result, meaning that the result of one operation could not correspond to the final result defined by the RISCV-V specification, for example, some bits could be in a different order since they are masked or reordered in a different stage and not by the ALU.

The *Scoreboard* also validates some operations that are supported by the ALU but are not supported by the RISCV-V specification, for example, the subtract with reduction operation, these operations are highlighted in purple. The instructions validated by this *Scoreboard* are shown in Tables 3.21, 3.22 and 3.23.

Table 3.21: RISCV-V Logic ALU Scoreboard checks

Checker	Signal	Description
BW_AND	logic_result[127:0]	Checks the result of VAND instruction
BW_AND_REDUCT	logic_result[127:0]	Checkss the result of VREDAND instruction
BW_OR	logic_result[127:0]	Checks the result of VOR instruction
BW_OR_REDUCT	logic_result[127:0]	Checks the result of VREDOR instruction
BW_XOR	logic_result[127:0]	Checks the result of VXOR instruction
BW_XOR__REDUCT	logic_result[127:0]	Checks the result of VREDXOR instruction
SLL	logic_result[127:0]	Checks the result of VSLL instruction
SRL	logic_result[127:0]	Checks the result of VSRL instruction
SRA	logic_result[127:0]	Checks the result of VSRA instruction
MAND	logic_result[15:0]	Checks the result of VMAND instruction
MNAND	logic_result[15:0]	Checks the result of VMNAND instruction
MANDN	logic_result[15:0]	Checks the result of VMANDN instruction
MOR	logic_result[15:0]	Checks the result of VMOR instruction
MNOR	logic_result[15:0]	Checks the result of VMNOR instruction
MORN	logic_result[15:0]	Checks the result of VMORN instruction
MXOR	logic_result[15:0]	Checks the result of VMXOR instruction
MXNOR	logic_result[15:0]	Checks the result of VMXNOR instruction

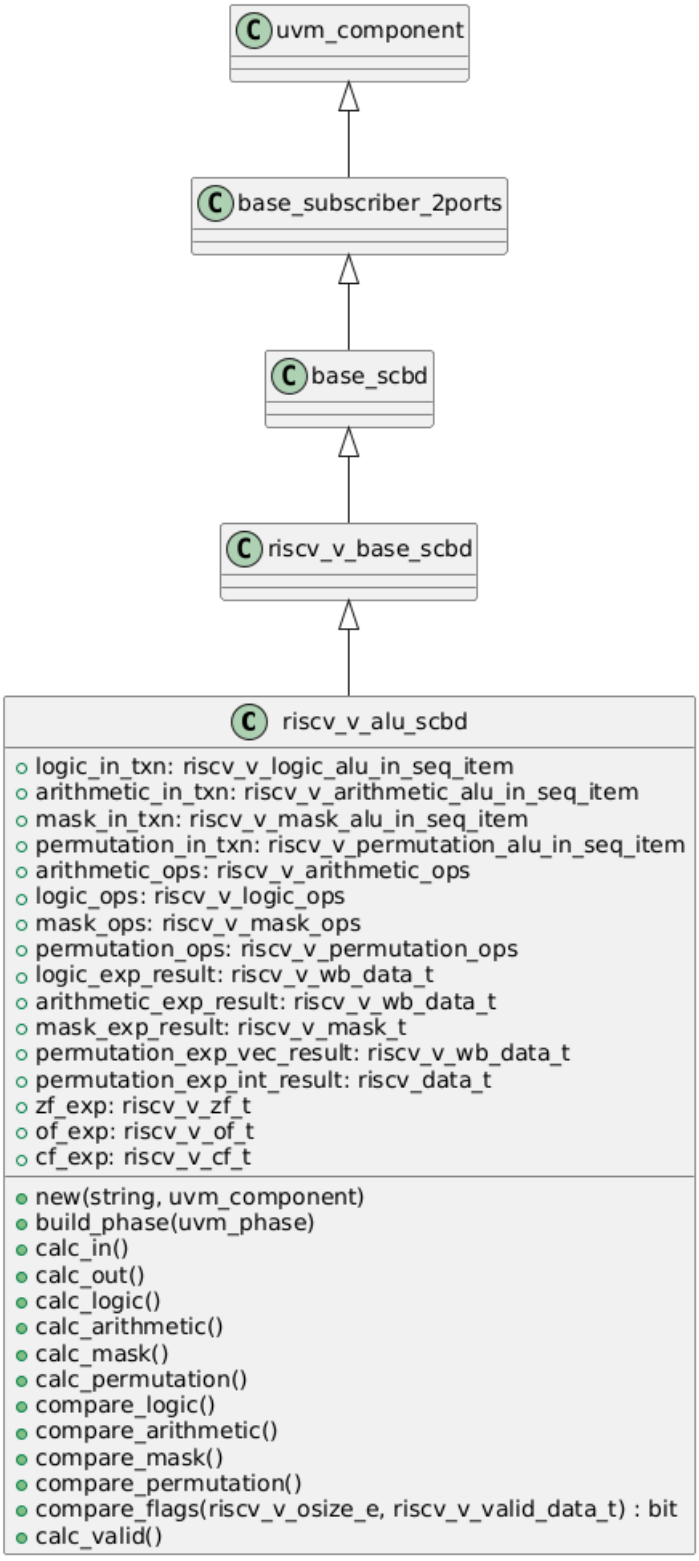


Figure 3.62: RISCV-V ALU Scoreboard UML

Table 3.22: RISCV-V Arithmetic ALU Scoreboard checks

Checker	Signal	Description
ADDC	arithmetic_result[127:0]	Checks the result of VADC instruction
ADD	arithmetic_result[127:0]	Checkss the result of VADD instruction
ADD_REDUCT	arithmetic_result[127:0]	Checks the result of VREDSUM instruction
SUBB	arithmetic_result[127:0]	Checks the result of VSBC instruction
SUB	arithmetic_result[127:0]	Checks the result of VSUB instruction
SUB_REDUCT	arithmetic_result[127:0]	Checks the result of VREDSUM instruction (Instruction not implemented in RISCV-V spec)
SIGN_EXT	arithmetic_result[127:0]	Checks the result of VSEXT instruction
ZERO_EXT	arithmetic_result[127:0]	Checks the result of VZEXT instruction
MINS	arithmetic_result[127:0]	Checks the result of VMIN instruction
MINS_REDUCT	arithmetic_result[127:0]	Checks the result of VREDMIN instruction
MINU	arithmetic_result[127:0]	Checks the result of VMINU instruction
MINU_REDUCT	arithmetic_result[127:0]	Checks the result of VREDMINU instruction
MAXS	arithmetic_result[127:0]	Checks the result of VMAX instruction
MAXS_REDUCT	arithmetic_result[127:0]	Checks the result of VREDMAX instruction
MAXU	arithmetic_result[127:0]	Checks the result of VMAXU instruction
MAXU_REDUCT	arithmetic_result[127:0]	Checks the result of VREDMAXU instruction
MULLU	arithmetic_result[127:0]	Checks the result of VMULU instruction (Instruction not implemented in RISCV-V spec)
MULLS	arithmetic_result[127:0]	Checks the result of VMUL instruction
MULHU	arithmetic_result[127:0]	Checks the result of VMULHU instruction
MULHS	arithmetic_result[127:0]	Checks the result of VMULH instruction
SEQ	arithmetic_result[15:0]	Checks the result of VMSEQ instruction
SNE	arithmetic_result[15:0]	Checks the result of VMSNE instruction
SLE	arithmetic_result[15:0]	Checks the result of VMSLE instruction
SLEU	arithmetic_result[15:0]	Checks the result of VMSLEU instruction
SLT	arithmetic_result[15:0]	Checks the result of VMSLT instruction
SLTU	arithmetic_result[15:0]	Checks the result of VMSLTU instruction
SGT	arithmetic_result[15:0]	Checks the result of VMSGT instruction
SGTU	arithmetic_result[15:0]	Checks the result of VMSGTU instruction
ZF	arithmetic_alu.zf[15:0]	Checks the zero flag of arithmetic ALU
CF	arithmetic_alu.cf[15:0]	Checks the carry flag of arithmetic ALU
OV	arithmetic_alu.ov[15:0]	Checks the overflow flag of arithmetic ALU

Table 3.23: RISCV-V Permutation ALU Scoreboard checks

Checker	Signal	Description
I2V	permutation_alu.vector_result[127:0]	Checks the result of VMV.I instruction
V2I	permutation_alu.integer_result[31:0]	Checks the result of VMV.V instruction

3.8.6 RISCV-V ALU Tracker

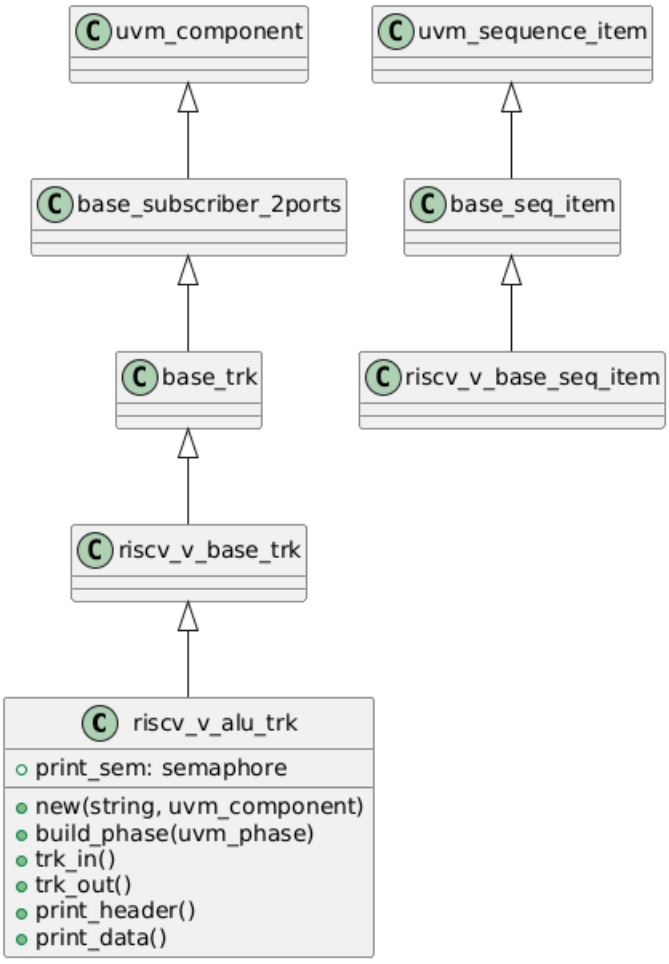


Figure 3.63: RISCV-V ALU Tracker UML

The *RISCV-V ALU Tracker* keeps track of all the operations executed by the ALUs, it prints the operation, the sources, the length and the ALU that executed the operation in the table. Figure 3.64 shows an example of the table generated.

Time	ALU	opcode	osize	dst_osize_vector	src_osize_vector	len	srca data	srca valid	srca merge	srcb data	srcb valid	srcb merge	result
1775000	LOGIC_ALU	BW_OR	OSIZE_8	1	x	16	0xfdfdfdfdfdfdfdfdfdfdfdfdfdfdfdfd	0xffff	0x0	0x0	0xffff	0x0	0xfdfdfdfdfdfdfdfdfdfdfdfdfdfdfdfd
1825000	LOGIC_ALU	BW_OR	OSIZE_8	1	x	16	0x0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d	0xffff	0x0	0x0	0xffff	0x0	0x0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d
1875000	LOGIC_ALU	BW_OR	OSIZE_8	1	x	16	0x0505050505050505050505050505	0xffff	0x0	0x0	0xffff	0x0	0x0505050505050505050505050505
1925000	LOGIC_ALU	BW_OR	OSIZE_8	1	x	16	0x0303030303030303030303030303	0xffff	0x0	0x0	0xffff	0x0	0x0303030303030303030303030303
1975000	LOGIC_ALU	BW_OR	OSIZE_8	1	x	16	0xa0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0	0xffff	0x0	0x0	0xffff	0x0	0xa0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0
2000000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0xa0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0	0xffff	0x0	0x5050505050505050505050505050	0xffff	0x0	0xf0f0f0f100f100f0f0f0f0f100f100ff
2050000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x5050505050505050505050505050	0xffff	0x0	0xfdfdfdfdfdfdfdfdfdfdfdfdfdfdfdfd	0xffff	0x0	0x2020202030203020302020203020302
2100000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	2	0xd0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d	0xffff	0x0	0xf7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f	0xffff	0x0	0xd0d0d0d0505040d0d0d0d0d0505040504
2150000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	2	0x4040404040404040404040404040404	0xffff	0x0	0xfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfc	0xffff	0x0	0x1000100000000000100100
2200000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	2	0xfcfcfcfcfcfcfcfcfcfcfcfcfcfcfcfc	0xffff	0x0	0xf9f9f9f9f9f9f9f9f9f9f9f9f9f9f9f9	0xffff	0x0	0xf5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5
2250000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x5050505050505050505050505050	0xffff	0x0	0xf2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2	0xffff	0x0	0xf7f7f7f7f7f7f7f7f7f7f7f7f7f7f7f7
2300000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0xf0f0f0f100f100f0f0f0f100f100ff	0xffff	0x0	0xf5f5f5f5f6f5f6f5f5f5f5f5f6f5f6f5	0xffff	0x0	0xd0d0d0d0705070d0d0d0d0d0705070d4
2350000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x2020202030203020202020203020302	0xffff	0x0	0xf2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2	0xffff	0x0	0xf4f4f4f4f6f4f6f4f4f4f4f4f6f4f6f4
2400000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0xa0a0a0a0a0a0a0a0a0a0a0a0a0a0a0a0	0xffff	0x0	0xfdfdfdfdfdfdfdfdfdfdfdfdfdfdfdfd	0xffff	0x0	0x7070707080708070707070807080707
2450000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d	0xc000	0x0	0xf5f5f5f5f6f5f6f5f5f5f5f5f6f5f6f5	0xc000	0x0	0x2020d0d0d0d0d0d0d0d0d0d0d0d0d0d
2500000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x1010101010101010101010101010101	0xc000	0x0	0x4040404070407040404040407040704	0xc000	0x0	0x5050102010201010101020102010201
2550000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0x6060606060606060606060606060606	0xc000	0x0	0x8080808080808080808080808080808	0xc000	0x0	0xe0e0e0e0607060706060606060706070
2600000	ARITHMETIC_ALU	ADDC	OSIZE_8	1	1	16	0xd0d0d0d0d0d0d0d0d0d0d0d0d0d0d0d	0xfffc	0x0	0x8080808080808080808080808080808	0xfffc	0x0	0x1515151516151615151515151615 0e0d

Figure 3.64: RISCV-V ALU Tracker Example



Figure 3.65: RISCV-V components UML

3.9 RISCV-V Agent

The *RISCV-V Agent* is used to validate the *RISCV Vector Engine* as a whole. The complete *RISCV Vector Engine* is used as DUT and multiple validation *models* to model the different parts of the *RISCV Vector Engine* to generate the expected result to be compared with the RTL. It receives as input the *instruction* to be executed, executes the *Vector instruction* using the *models* and compares the data to be written to the Integer or Vector Register File.

3.9.1 RISCV-V Coverage

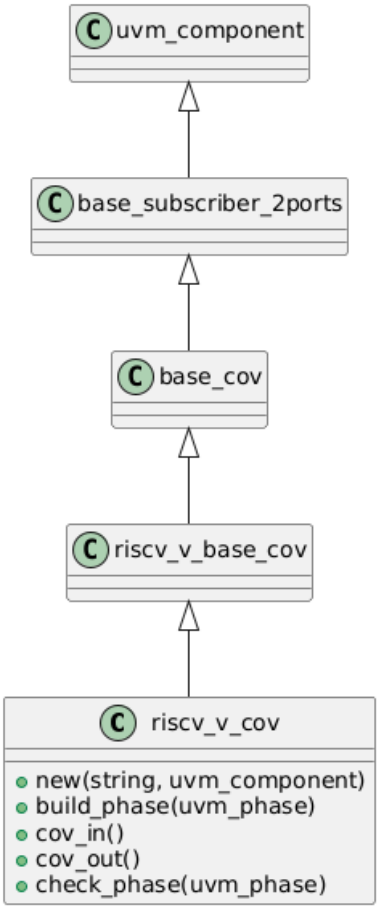


Figure 3.66: RISCV-V Coverage UML

The *RISCV-V Coverage* collector is similar to the *RISCV-V IF Coverage* collector. This collector also samples the instruction to cover that all the fields are exercised, but this collector also adds cover points for the *OSIZE*, the *Len* and the *Start*. It instantiates one cross cover

point between the *funct6*, *funct3*, *mask* and *OSIZE* to cover that all the supported instructions, with all the variants and all the *OSIZEs* are executed. It also creates two extra cover points, one crossing *funct6* with *Len* and the second one crossing *funct6* with *Start*, these cross cover points are used to cover that each major vector opcode is executed with all the combinations of *Len* and *Start*. The cover points with its description are shown in Table 3.24

Table 3.24: RISCV-V Functional Coverage

Cover Group	Cover Point	Cover Bin	Description
INSTRUCTION	vec_opcode	vector_op	Bin to cover that the opcode of the RISC-V instruction corresponds to a vector instruction
	vec_funct3	OPIVV OPFVV OPMVV OPIVI OPIVX OPFVF OPMVX OPCFG	Bins to cover that all encodings of funct3 are selected
	vec_mask	masked unmasked	Bins to cover that masked and unmasked vector operations are executed
	vec_funct6	OPS[53]	Creates 1 bin per funct6 that is supported by the RISC-V vector engine
	osize	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	Creates 5 bins to cover all the valid OSIZEs
	len	len[17] len_greater_16 len_less_start	Creates 17 bins to cover the values of len from 0 to 16, 1 bin to cover all the values greater than 16 and 1 bin to cover if len is less than start
	start	start[17] start_greater_16	Creates 17 bins to cover the values of start from 0 to 16 and 1 bin to cover all the values greater than 16
	FUNCT6 X FUNCT3 X MASK X OSIZE	cross[777]	Cross cover to cover that all the valid formats of the supported vector instructions are executed, including the supported OSIZEs
	FUNCT6 X LEN	cross[1007]	Cross cover to check if the major vec opcode encoded in funct6 is executed with the different combinations of len
	FUNCT6 X START	cross[954]	Cross cover to check if the major vec opcode encoded in funct6 is executed with the different combinations of start

3.9.2 RISCV-V Monitor

The *RISCV-V Agent* does not implement a driver since the signals needed to drive stimulus to the RTL can be driven by other agents like the *External CSR Agent* and the *IF Agent*, but it implements a *monitor* to read the interface signals.

The *Monitor* sample the signals at different times, first, the *instruction* and *external_csr)* signals are sampled and one cycle later the *ext_data_in* is sampled, the signals are encapsulated in the *sequence item* but it is not sent to the other components, it is buffered two extra cycles so the *sequence item* is sent at the time the instruction received at *INSTRUCTION DECODE* is at *WRITE BACK*. By buffering and delaying the *sequence item* the *Scoreboard* receives the instruction to be executed and the result of the instruction in the same cycle, in that way the



Figure 3.67: RISCV-V Driver/Monitor UML

Scoreboard can execute the instruction in one cycle and compare the expected result with the RTL result, removing the need of synchronize the input and output transactions and removing the logic needed to solve hazards since all the instructions are always executed in order and one cycle inside the *Scoreboard*.

The signals sampled by the monitor are shown in Figure 3.25.

Table 3.25: RISCV-V Monitor signals

Monitor Port	Name	Type	Description
RTL IN	**rst**	logic	Active high reset signal
	clear_pipe	logic	Active high signal to flush pipe
	riscv_stall	logic	Active high signal to stall pipe
	instruction	logic[31:0]	RISCV-V Instruction
	int_rf_rd_data	logic[31:0]	Data coming Integer Register File
	ext_data_in	logic[31:0]	Data to be written to CSR
	ext_wr_vsstatus	logic	If 1'b1 vsstatus is updated with ext_data_in
	ext_wr_vtype	logic	If 1'b1 vtype is updated with ext_data_in
	ext_wr_vl	logic	If 1'b1 vl is updated with ext_data_in
	ext_wr_vstart	logic	If 1'b1 vstart is updated with ext_data_in
	ext_wr_vxrm	logic	If 1'b1 vxrm is updated with ext_data_in
	ext_wr_vxsat	logic	If 1'b1 vxsat is updated with ext_data_in
RTL OUT	**riscv_v_stall**	logic	Stall signal generated by Vector Unit
	int_rf_wr_data	logic[31:0]	Data to be written to Integer Register File
	int_rf_wr_en	logic	Integer register File write enable
	vec_rf_wr_addr	logic[4:0]	Synthesis Spare Signal: Vector Register File write address
	vec_rf_wr_en	logic[15:0]	Synthesis Spare Signal: Vector Register File write enable
	vec_rf_wr_data	logic[127:0]	Synthesis Spare Signal: Vector Register File write data

3.9.3 RISCV-V Model

The *RISCV-V Model* is used to model the *RISCV Vector Engine*, it models the whole process of the *vector instruction* since it is dispatched and received by the *RISCV Vector Engine* until the result is written in the Integer Register File or the *VRF*.

The *RISCV-V Model* instantiates other *models* to execute the *instruction*. It instantiates one *Vector RF Model* to model the *VRF*, one *CSR Model* to model the *CSR* block, one *Decode Model* to decode the *instruction* and one *Execute Model* to execute the instruction. The flow diagram of the *RISCV-V Model* is shown in Figure 3.69.

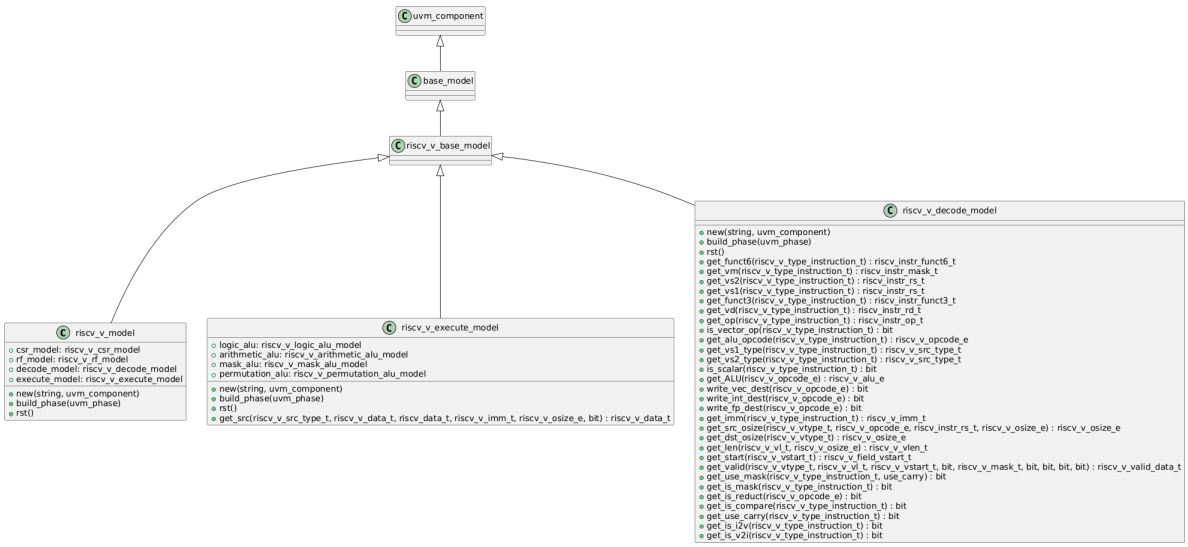


Figure 3.68: RISCV-V Model UML

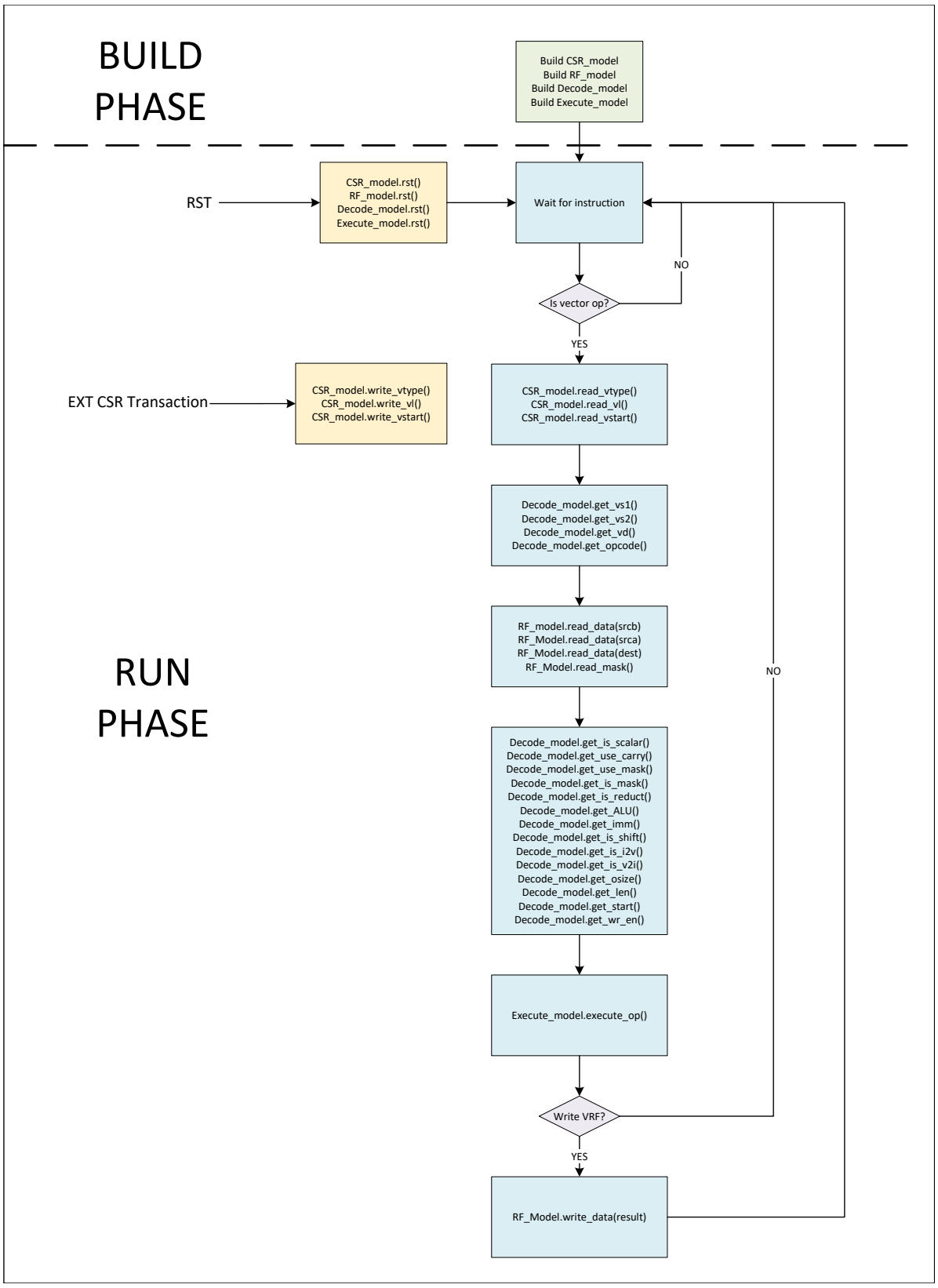


Figure 3.69: RISCV-V Model Flow diagram

RISCV-V Decode Model

The *RISCV-V Decode Model* is the model used to decode the *vector instructions*, it implements many methods that receive as argument the *instruction*, decodes the instruction and returns the field or the property requested from the instruction, for example it can return the source/destination address or it can return one bit indicating if it is a *reduction operation*.

RISCV-V Execute Model

The *RISCV-V Execute Model* models the three different ALUs and is used to execute a *vector instruction* and get the result of the *instruction*. The *execute model* instantiates the *ALU models* to execute the operation, depending on the type of operation it is sent to the corresponding *ALU model*.

3.9.4 RISCV-V Scoreboard

The *RISCV-V Scoreboard* is used to check the correctness of the *RISCV Vector Engine*. This *Scoreboard* use as reference model the *RISCV-V Model* that models the whole flow of the *RISCV Vector Engine*. At the same time the *scoreboard* receives the instruction to be executed and the result of that instruction from the *monitor*. It sends the instruction to the *model* to execute the instruction, gets the expected result and compares the expected result with the RTL result.

The *scoreboard* performs checks on the integer and vector RF *wr_en* to verify that the correct Register File is written, it also performs checks on all the valid operations described in *Operations implemented* as the *ALU Scoreboard*, but unlike the *ALU Scoreboard* this *scoreboard* checks that the final result corresponds to the architectural result described in the RISCV-V specification, meaning that all the bits should be in the correct order, and this *scoreboard* only checks the *Vector Instructions* specified in the RISCV-V Specification, removing the instruction supported by the ALU but not specified in the RISCV-V specification. The Table 3.26 shows all the checks performed by the *scoreboard*.

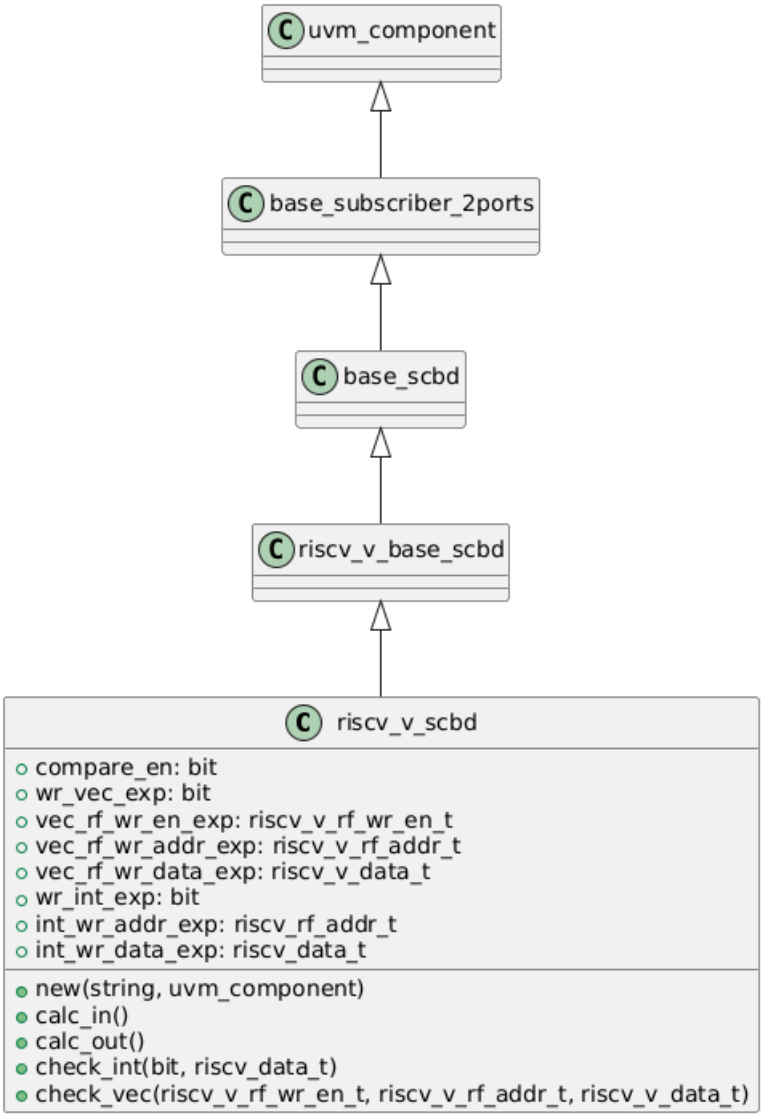


Figure 3.70: RISCV-V Scoreboard UML

Table 3.26: RISCV-V Scoreboard checks

Checker	Signal	Description
Integer RF Write Enable	int_rf_wr_en	Checks the write enable of the integer register file
Vector RF Write Enable	vec_rf_wr_en[15:0]	Checks the write enable of the vector register file
Vector RF Write Address	vec_rf_wr_addr[4:0]	Checks the write address of the vector register file
BW_AND	vec_rf_wr_data[127:0]	Checks the result of VAND instruction
BW_AND_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDAND instruction
BW_OR	vec_rf_wr_data[127:0]	Checks the result of VOR instruction
BW_OR_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDOR instruction
BW_XOR	vec_rf_wr_data[127:0]	Checks the result of VXOR instruction
BW_XOR__REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDXOR instruction
SLL	vec_rf_wr_data[127:0]	Checks the result of VSLL instruction
SRL	vec_rf_wr_data[127:0]	Checks the result of VSRL instruction
SRA	vec_rf_wr_data[127:0]	Checks the result of VSRA instruction
MAND	vec_rf_wr_data[15:0]	Checks the result of VMAND instruction
MNAND	vec_rf_wr_data[15:0]	Checks the result of VMNAND instruction
MANDN	vec_rf_wr_data[15:0]	Checks the result of VMANDN instruction
MOR	vec_rf_wr_data[15:0]	Checks the result of VMOR instruction
MNOR	vec_rf_wr_data[15:0]	Checks the result of VMNOR instruction
MORN	vec_rf_wr_data[15:0]	Checks the result of VMORN instruction
MXOR	vec_rf_wr_data[15:0]	Checks the result of VMXOR instruction
MXNOR	vec_rf_wr_data[15:0]	Checks the result of VMXNOR instruction
ADDC	vec_rf_wr_data[127:0]	Checks the result of VADC instruction
ADD	vec_rf_wr_data[127:0]	Checks the result of VADD instruction
ADD_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDSUM instruction
SUBB	vec_rf_wr_data[127:0]	Checks the result of VSBC instruction
SUB	vec_rf_wr_data[127:0]	Checks the result of VSUB instruction
SIGN_EXT	vec_rf_wr_data[127:0]	Checks the result of VSEXT instruction
ZERO_EXT	vec_rf_wr_data[127:0]	Checks the result of VZEXT instruction
MINS	vec_rf_wr_data[127:0]	Checks the result of VMIN instruction
MINS_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDMIN instruction
MINU	vec_rf_wr_data[127:0]	Checks the result of VMINU instruction
MINU_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDMINU instruction
MAXS	vec_rf_wr_data[127:0]	Checks the result of VMAX instruction
MAXS_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDMAX instruction
MAXU	vec_rf_wr_data[127:0]	Checks the result of VMAXU instruction
MAXU_REDUCT	vec_rf_wr_data[127:0]	Checks the result of VREDMAXU instruction
MULLS	vec_rf_wr_data[127:0]	Checks the result of VMUL instruction
MULHU	vec_rf_wr_data[127:0]	Checks the result of VMULHU instruction
MULHS	vec_rf_wr_data[127:0]	Checks the result of VMULH instruction
SEQ	vec_rf_wr_data[15:0]	Checks the result of VMSEQ instruction
SNE	vec_rf_wr_data[15:0]	Checks the result of VMSNE instruction
SLE	vec_rf_wr_data[15:0]	Checks the result of VMSLE instruction
SLEU	vec_rf_wr_data[15:0]	Checks the result of VMSLEU instruction
SLT	vec_rf_wr_data[15:0]	Checks the result of VMSLT instruction
SLTU	vec_rf_wr_data[15:0]	Checks the result of VMSLTU instruction
SGT	vec_rf_wr_data[15:0]	Checks the result of VMSGT instruction
SGTU	vec_rf_wr_data[15:0]	Checks the result of VMSGTU instruction
I2V	vec_rf_wr_data[127:0]	Checks the result of VMV.I instruction
V2I	int_rf_wr_data[31:0]	Checks the result of VMV.V instruction

3.10 RISCV-V TB

To verify the correctness of the *RISCV Vector Engine* it is instantiated in the *RISCV-V TB* as shown in Figure 3.71. The *TB* also instantiates the *RISCV-V Test* to be executed that instantiates the different *RISCV-V Agents* to validate the design.

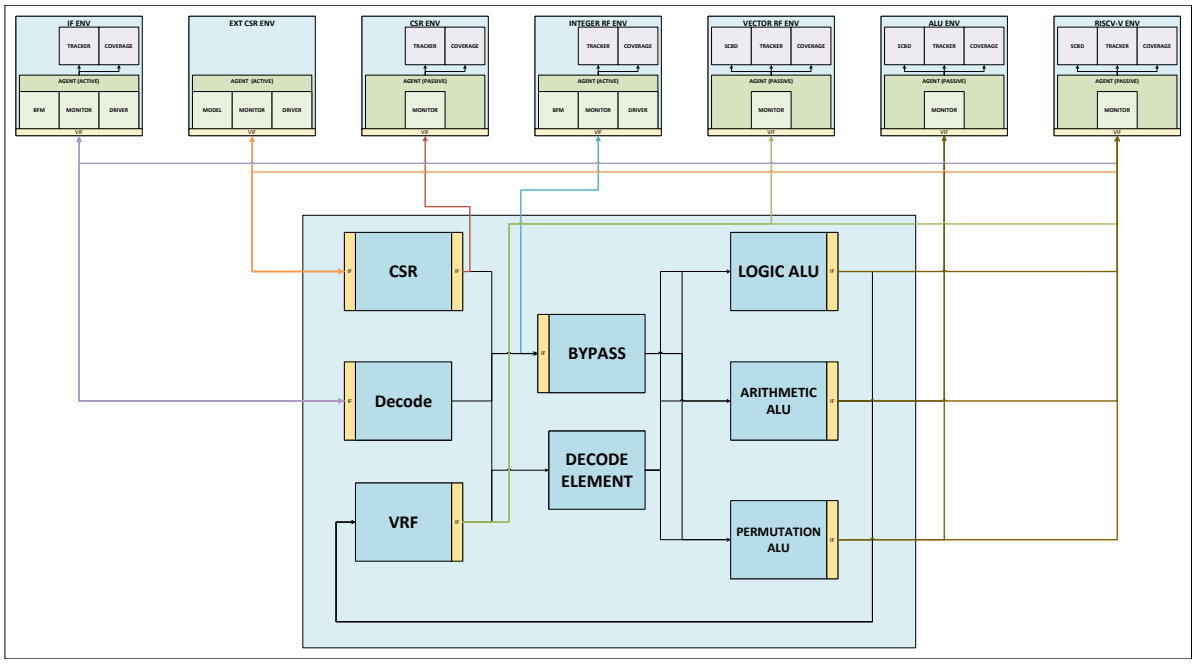


Figure 3.71: RISCV-V TB

The *TB* does not drive any stimulus nor perform any checking, it just instantiates and connects to the DUT the *Virtual Interfaces*, and instantiates the *Test*. All the stimuli are generated by the *BFMs* and driven through the *Drivers* using the *Virtual Interfaces*. The Checks are performed by the *Scoreboards* using the *Models* as reference.

3.11 RISCV-V Tests

Each *Supported vector operation* has its test to verify its correctness. The test generates stimuli to execute all the supported variants of the instructions including all the supported *OSIZEs*. All the tests use the same structure to generate the stimuli, they are configured with the desired stimuli to be generated and the tests configure and control the *CSR BFM* and the *IF BFM*. The configuration knobs are shown in Table 3.27.

First, the test initializes the *VRF* with random data, once all the registers have been initialized the test starts sending different blocks of instructions, the length of the block is configured with the configuration knob *NUM_TXN_BLOCK*, this knob indicates how many instructions are sent in each block, each block represents one variant of the instruction. The test iterates over all the valid variants of the instruction and sends one block of instructions with each variant. The test iterates over the variants using three nested for loops, the external loop iterates over the valid *funct3* variants, the middle loop iterates over the valid *OSIZEs*

Table 3.27: RISCV-V Test configuration knobs

Knob	Type	Description
RST_CYCLES	INT	Indicates the number of reset cycles before starting the test
INIT_CYCLES	INT	Indicates the number of cycles used to initialize the VRF
NUM_TXN_BLOCK	INT	Indicates the number of transactions to be sent in each block
SUPPORTS_MASK	bit	Indicates if the instruction supports masking encoding
SUPPORTS_UNMASK	bit	Indicates if the instruction supports unmasking encoding
NUM_OSIZES	int	Indicates the number of OSIZEs to be used
valid_osizes	riscv_v_osize_e[]	Array holding the OSIZEs to be used
NUM_FORMATS	int	Indicates the number of formats to be used
valid_formats	riscv_v_funct3_e[]	Array holding the formats to be used
TOTAL_BLOCKS	int	Indicates the number of blocks to be executed
USE_RAND_START	bit	Indicates if VSTART is randomized
USE_RAND_LEN	bit	Indicates if VLEN is randomize

while the internal loop iterates over the masked and unmasked version of the instruction. The following snippet shows the structure of the loops used to generate the instructions.

```
//Execute instructions
foreach (valid_funct3[i]) begin  //Iterate over valid funct3 formats
    foreach (valid_osize[j]) begin //Iterate over valid osizes
        foreach (valid_mask[k]) begin //Iterate over valid masking formats
            repeat(NUM_TXN_BLOCK) begin
              instruction(valid_funct3[i], valid_osize[j], valid_mask[k]);
            end
        end
    end
end
```

The *Len* and *Start* of the instruction are randomized using the following distributions:

1. $(\quad LEN == 16 \quad) -> 80\%$
2. $(0 <= LEN <= 16 \quad) -> 15\%$
3. $(\quad LEN > 16 \quad) -> 5\%$
4. $(\quad START == 0 \quad) -> 80\%$
5. $(0 < START < 15 \quad) -> 15\%$
6. $(\quad START > 15 \quad) -> 5\%$

Those distributions are used so most of the time all the elements of the vector register are *Active Elements*, the 80% of the time *LEN* is sixteen and *START* is zero, meaning that all the elements are active. The 15% of the time *LEN* and *START* are randomized with values between zero and sixteen, so only some elements of the register are active. The 5% of the time they are randomized with values that are out of range of the valid values, meaning that

all the elements are inactive.

This is done to exercise the ALUs by enabling most of the elements most of the time, the logic to enable/disable the elements is common for all the operations, so that logic can be exercised with the combination of all the operations executed and there is no need to increment the distribution of the invalid ranges to verify that part isolated.

In Figure 3.72 the RISCV-V VADD test is shown as an example of a test. First the *VRF* is initialized, this step is shown in the pink square, after the initialization the test modifies the configuration of the *BFMs* to start sending the appropriated instructions, the gray square identifies one *TXN BLOCK*, the *TXN BLOCK* is a block of instructions with the same *funct6*, *funct3*, *VM* and *OSIZE*, meaning that is the same variant of the instruction, the *VS1*, *VS2*, *Len* and *Start* are randomized. The red square identifies one iteration of the masking format, the blue square one iteration of the supported *OSIZEs* and the *orange square* the iteration over the supported *funct3* values.

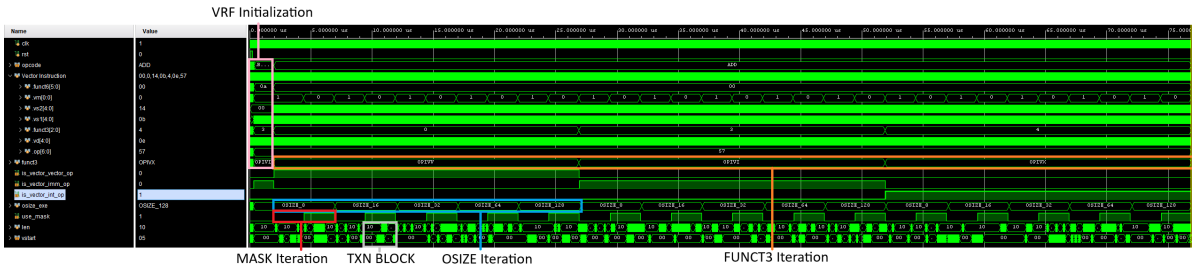


Figure 3.72: RISCV-V VADD Test

The Tables 3.28, 3.29 and 3.30 show all the tests implemented with the supported variants of each test.

Table 3.28: Logic Tests

Type of Test	Test	Valid FUNCT3	Valid OSIZEs	Valid Masking	Number of variants (Number of TXN)
LOGIC	VMAND	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMANDN	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMNAND	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMORN	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMNOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMXOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VMXNOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	VAND	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VOR	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VXOR	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VREDAND	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VREDOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VREDXOR	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VSLL	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VSRL	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VSRA	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)

Table 3.29: Arithmetic Tests

Type of Test	Test	Valid FUNCT3	Valid OSIZEs	Valid Masking	Number of variants (Number of TXN)
ARITHMETIC	VADD	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VADDC	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED	15(750)
	VREDSUM	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VSUB	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VSUBB	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED	10(500)
	VMAX	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	10(500)
	VMAXU	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	10(500)
	VMIN	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	10(500)
	VMINU	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	10(500)
	VREDMAX	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VREDMAXU	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VREDMIN	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VREDMINU	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64	MASKED UNMASKED	8(400)
	VMSEQ	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VMSNEQ	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VMSGT	OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMSGTU	OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMSLE	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VMSLEU	OPIVV OPIVI OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	30(1500)
	VMSLT	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMSLTU	OPIVV OPIVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMUL	OPMVV OPMVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMULH	OPMVV OPMVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VMULHU	OPMVV OPMVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	20(1000)
	VSEXT	OPMVV	OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	8(400)
	VZEXT	OPMVV	OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	MASKED UNMASKED	8(400)

Table 3.30: Permutation Tests

Type of Test	Test	Valid FUNCT3	Valid OSIZEs	Valid Masking	Number of variants (Number of TXN)
PERMUTATION	V2I	OPMVV	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)
	I2V	OPMVX	OSIZE_8 OSIZE_16 OSIZE_32 OSIZE_64 OSIZE_128	UNMASKED	5(250)

3.12 Validation Results

All the tests were run using *Vivado 2024.1* using a TCL script to run the tests in shell mode, saving the logs of the tests indicating if the test passes or fails and the coverage database of the tests. The test only is set as *PASS* if there are no *error* nor *fatal* messages in the log file, if one *error* or *fatal* is encountered the test status is set to *FAIL*, the Figure 3.73 shows the last lines of the log of a test, were the statistics of the *Scoreboards* and *Coverage collectors* are printed and the final status of the test. Once all the tests were executed the coverage databases are merged into a single database and a Python script is executed, this script reads all the log files and indicates the final status of each test, indicating also if the regression passed or failed as shown in Figure 3.74.

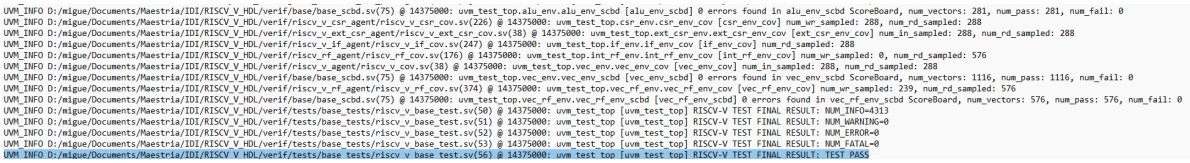


Figure 3.73: Test Status

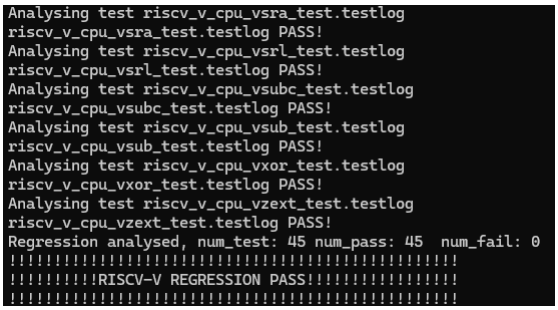
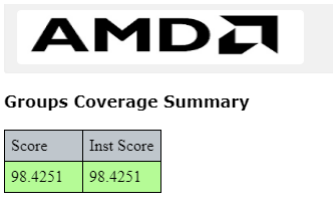


Figure 3.74: Regression Status

3.12.1 Coverage Results

As mentioned in *Validation Results* section the coverage databases of all the tests are merged into a single database to get the final coverage statistics. Figure 3.75 shows the final score of all the cover groups instantiated, not all cover groups/cover bins described in the previous sections affect this coverage score, some of them were disabled since the scope of this project does not include some features measured by those cover groups/bins and including them would affect the coverage statistics. More detailed information about each cover group is described in the following sections.



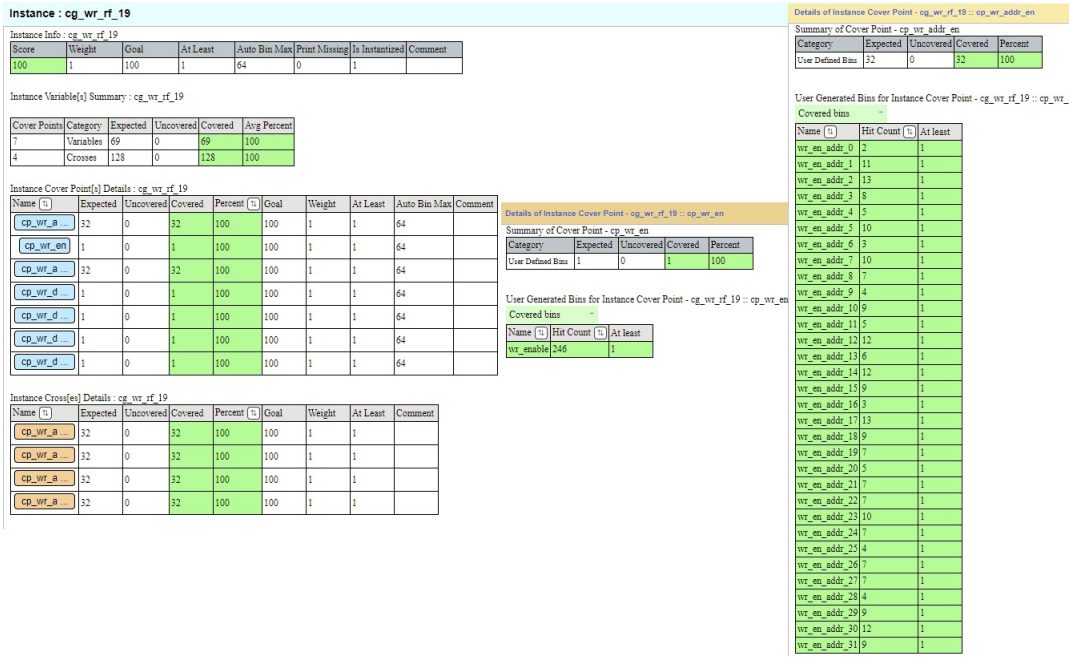
Groups Coverage Summary

Score	Inst Score
98.4251	98.4251

Figure 3.75: Final Coverage

RISCV RF

Figure 3.76 shows the coverage report of the write cover group, it has a score of 100% meaning that all cover bins were hit, meaning that all registers were written with at least four different ranges of data. The *Integer Register File* is only written by the *VMV.x.s* instruction for that reason just 246 write transactions are sent to the *Integer Register File* but that number of transactions is enough to write to all registers at least two times.



Instance : cg_wr_rf_19

Instance Info : cg_wr_rf_19

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_wr_rf_19

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
7	Variables	69	0	69	100
4	Crosses	128	0	128	100

Instance Cover Point[s] Details : cg_wr_rf_19

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_wr_a ...	32	0	32	100	100	1	1	64	
cp_wr_en	1	0	1	100	100	1	1	64	
cp_wr_a ...	32	0	32	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_wr_rf_19

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_wr_a ...	32	0	32	100	100	1	1	
cp_wr_a ...	32	0	32	100	100	1	1	
cp_wr_a ...	32	0	32	100	100	1	1	
cp_wr_a ...	32	0	32	100	100	1	1	

Details of Instance Cover Point - cg_wr_rf_19 :: cp_wr_en

Summary of Cover Point - cp_wr_en

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	1	0	1	100

User Generated Bins for Instance Cover Point - cg_wr_rf_19 :: cp_wr_en

Covered bins

Name	Hit Count	At least
wr_enable	246	1

Details of Instance Cover Point - cg_wr_rf_19 :: cp_wr_addr_en

Summary of Cover Point - cp_wr_addr_en

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	32	0	32	100

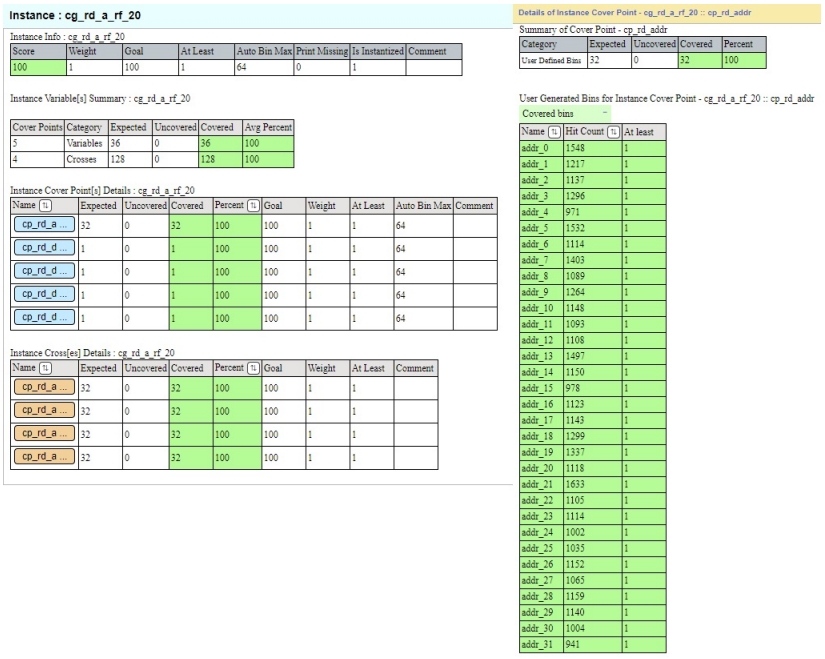
User Generated Bins for Instance Cover Point - cg_wr_rf_19 :: cp_wr_

Covered bins

Name	Hit Count	At least
wr_en_addr_0	2	1
wr_en_addr_1	11	1
wr_en_addr_2	13	1
wr_en_addr_3	8	1
wr_en_addr_4	5	1
wr_en_addr_5	10	1
wr_en_addr_6	3	1
wr_en_addr_7	10	1
wr_en_addr_8	7	1
wr_en_addr_9	4	1
wr_en_addr_10	9	1
wr_en_addr_11	5	1
wr_en_addr_12	12	1
wr_en_addr_13	6	1
wr_en_addr_14	12	1
wr_en_addr_15	9	1
wr_en_addr_16	3	1
wr_en_addr_17	13	1
wr_en_addr_18	9	1
wr_en_addr_19	7	1
wr_en_addr_20	5	1
wr_en_addr_21	7	1
wr_en_addr_22	7	1
wr_en_addr_23	10	1
wr_en_addr_24	7	1
wr_en_addr_25	4	1
wr_en_addr_26	7	1
wr_en_addr_27	7	1
wr_en_addr_28	4	1
wr_en_addr_29	9	1
wr_en_addr_30	12	1
wr_en_addr_31	9	1

Figure 3.76: RISCV RF write coverage

The coverage reports of the rd_A and rd_B cover groups are shown in Figure 3.77 and 3.78 respectively. As the write cover group, they have a score of 100% meaning that all registers were read with at least four different ranges of data, but in this case, the *Integer Register File* is always read, even if it is not used by the vector operation, for that reason the number of hits in each cover bin increases substantially compared with the write cover group.



Instance : cg_rd_a_rf_20

Instance Info : cg_rd_a_rf_20

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_rd_a_rf_20

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
5	Variables	36	0	36	100
4	Crosses	128	0	128	100

Instance Cover Point[s] Details : cg_rd_a_rf_20

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_rd_a ...	32	0	32	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_rd_a_rf_20

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	

Details of Instance Cover Point - cg_rd_a_rf_20 :: cp_rd_addr

Summary of Cover Point - cp_rd_addr

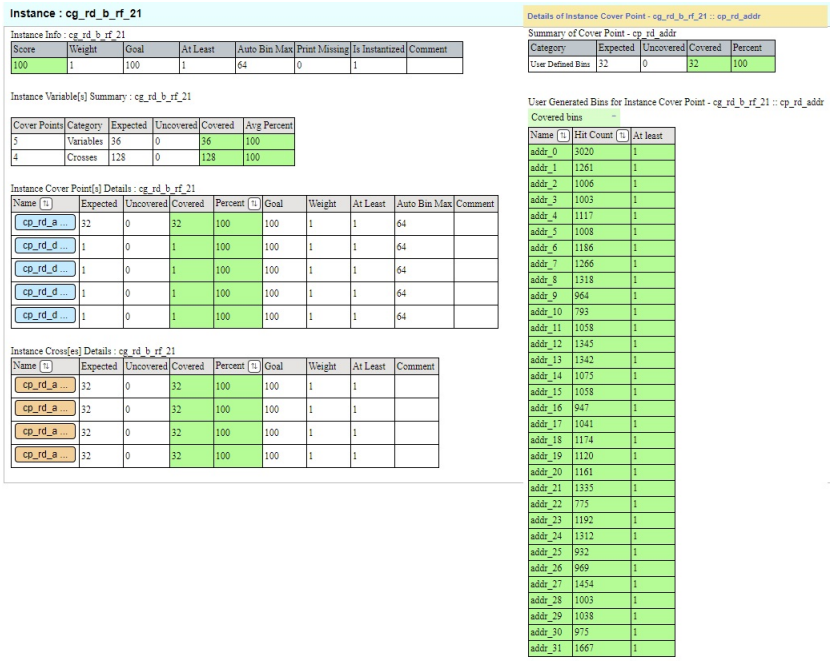
Category	Expected	Uncovered	Covered	Percent
User Defined Bins	32	0	32	100

User Generated Bins for Instance Cover Point - cg_rd_a_rf_20 :: cp_rd_addr

Covered bins

Name	Hit Count	At least
addr_0	1548	1
addr_1	1217	1
addr_2	1137	1
addr_3	1296	1
addr_4	971	1
addr_5	1532	1
addr_6	1114	1
addr_7	1403	1
addr_8	1089	1
addr_9	1264	1
addr_10	1148	1
addr_11	1093	1
addr_12	1108	1
addr_13	1497	1
addr_14	1150	1
addr_15	978	1
addr_16	1123	1
addr_17	1143	1
addr_18	1299	1
addr_19	1337	1
addr_20	1118	1
addr_21	1633	1
addr_22	1105	1
addr_23	1114	1
addr_24	1002	1
addr_25	1035	1
addr_26	1152	1
addr_27	1065	1
addr_28	1159	1
addr_29	1140	1
addr_30	1004	1
addr_31	941	1

Figure 3.77: RISCV RF read port A coverage



Instance : cg_rd_b_rf_21

Instance Info : cg_rd_b_rf_21

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_rd_b_rf_21

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
5	Variables	36	0	36	100
4	Crosses	128	0	128	100

Instance Cover Point[s] Details : cg_rd_b_rf_21

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_rd_a ...	32	0	32	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_rd_b_rf_21

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	
cp_rd_a ...	32	0	32	100	100	1	1	

Details of Instance Cover Point - cg_rd_b_rf_21 :: cp_rd_addr

Summary of Cover Point - cp_rd_addr

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	32	0	32	100

User Generated Bins for Instance Cover Point - cg_rd_b_rf_21 :: cp_rd_addr

Covered bins

Name	Hit Count	At least
addr_0	3020	1
addr_1	1261	1
addr_2	1006	1
addr_3	1003	1
addr_4	1117	1
addr_5	1008	1
addr_6	1186	1
addr_7	1266	1
addr_8	1318	1
addr_9	964	1
addr_10	793	1
addr_11	1058	1
addr_12	1345	1
addr_13	1342	1
addr_14	1075	1
addr_15	1058	1
addr_16	947	1
addr_17	1041	1
addr_18	1174	1
addr_19	1120	1
addr_20	1161	1
addr_21	1335	1
addr_22	775	1
addr_23	1192	1
addr_24	1312	1
addr_25	932	1
addr_26	969	1
addr_27	1454	1
addr_28	1003	1
addr_29	1038	1
addr_30	975	1
addr_31	1667	1

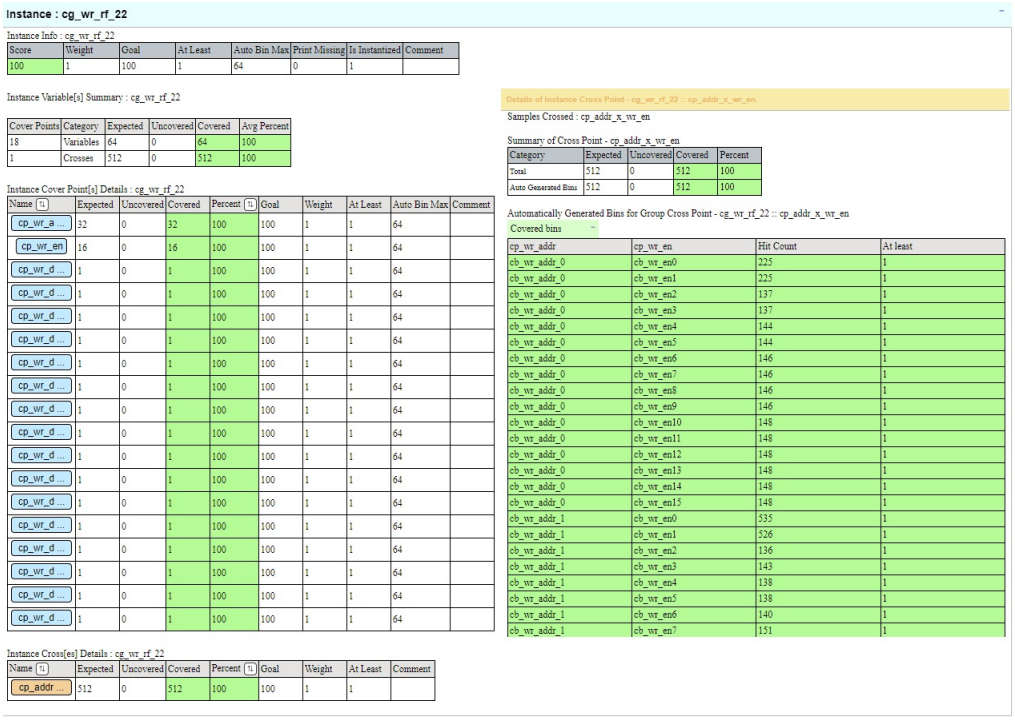
Figure 3.78: RISCV RF read port B coverage

RISCV-V RF

The coverage report of the write cover group is shown in Figure 3.79. It has a score of 100% meaning that all data bytes of all addresses were written at least one time, this can be confirmed with the cross cover *WR_ADDR X WR_EN*, that crosses the *wr_addr* with *wr_en*. This cover group has more hits than its integer counterpart since almost all instructions write to the *VRF*.

The Figures 3.80 and 3.81 show the coverage reports of the rd_A and rd_B cover groups. Both cover groups are similar and have a score of 100% meaning that all registers were read returning at least sixteen different ranges of data. These ports are always read by all instructions increasing the number of hits.

In Figure 3.82 the mask cover group coverage report is shown. It also has a score of 100% meaning that all mask bits were used to mask/unmask the elements of the vector operation. The mask registers is always read even if it is an unmasked operation increasing the hit rate of the mask cover group.



Instance : cg_wr_rf_22

Instance Info : cg_wr_rf_22

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_wr_rf_22

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
18	Variables	64	0	64	100
1	Crosses	512	0	512	100

Instance Cover Point[s] Details : cg_wr_rf_22

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_wr_a ...	32	0	32	100	100	1	1	64	
cp_wr_en	16	0	16	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	
cp_wr_d ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_wr_rf_22

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_addr ...	512	0	512	100	100	1	1	

Details of Instance Cross Point - cg_wr_rf_22 :: cp_addr_x_wr_en

Samples Crossed : cp_addr_x_wr_en

Summary of Cross Point - cp_addr_x_wr_en

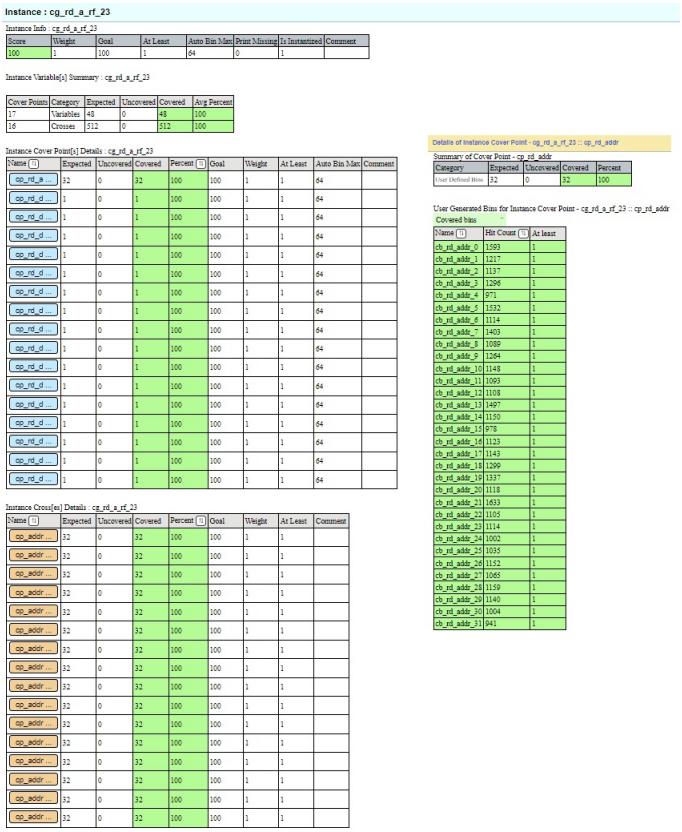
Category	Expected	Uncovered	Covered	Percent
Total	512	0	512	100
Auto Generated Bins	512	0	512	100

Automatically Generated Bins for Group Cross Point - cg_wr_rf_22 :: cp_addr_x_wr_en

Covered bins

cp_wr_addr	cp_wr_en	Hit Count	At least
cb_wr_addr_0	cb_wr_en0	225	1
cb_wr_addr_0	cb_wr_en1	225	1
cb_wr_addr_0	cb_wr_en2	137	1
cb_wr_addr_0	cb_wr_en3	137	1
cb_wr_addr_0	cb_wr_en4	144	1
cb_wr_addr_0	cb_wr_en5	144	1
cb_wr_addr_0	cb_wr_en6	146	1
cb_wr_addr_0	cb_wr_en7	146	1
cb_wr_addr_0	cb_wr_en8	146	1
cb_wr_addr_0	cb_wr_en9	146	1
cb_wr_addr_0	cb_wr_en10	148	1
cb_wr_addr_0	cb_wr_en11	148	1
cb_wr_addr_0	cb_wr_en12	148	1
cb_wr_addr_0	cb_wr_en13	148	1
cb_wr_addr_0	cb_wr_en14	148	1
cb_wr_addr_0	cb_wr_en15	148	1
cb_wr_addr_1	cb_wr_en0	535	1
cb_wr_addr_1	cb_wr_en1	526	1
cb_wr_addr_1	cb_wr_en2	136	1
cb_wr_addr_1	cb_wr_en3	143	1
cb_wr_addr_1	cb_wr_en4	138	1
cb_wr_addr_1	cb_wr_en5	138	1
cb_wr_addr_1	cb_wr_en6	140	1
cb_wr_addr_1	cb_wr_en7	151	1

Figure 3.79: RISCV-V RF write coverage



Instance : cg_rd_a_rf_23

Instance Info : cg_rd_a_rf_23

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_rd_a_rf_23

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
17	Variables	48	0	48	100
16	Crosses	512	0	512	100

Instance Cover Point[s] Details : cg_rd_a_rf_23

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_rd_a ...	32	0	32	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	
cp_rd_d ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_rd_a_rf_23

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	
cp_addr ...	32	0	32	100	100	1	1	

Details of Instance Cover Point - cg_rd_a_rf_23 :: cp_rd_addr

Summary of Cover Point - cp_rd_addr

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	32	0	32	100

User Generated Bins for Instance Cover Point - cg_rd_a_rf_23 :: cp_rd_addr

Covered bins

Name	Hit Count	At least
cb_rd_addr_0	1593	1
cb_rd_addr_1	1217	1
cb_rd_addr_2	1137	1
cb_rd_addr_3	1296	1
cb_rd_addr_4	971	1
cb_rd_addr_5	1532	1
cb_rd_addr_6	1114	1
cb_rd_addr_7	1403	1
cb_rd_addr_8	1089	1
cb_rd_addr_9	1264	1
cb_rd_addr_10	1148	1
cb_rd_addr_11	1093	1
cb_rd_addr_12	1108	1
cb_rd_addr_13	1497	1
cb_rd_addr_14	1150	1
cb_rd_addr_15	978	1
cb_rd_addr_16	1123	1
cb_rd_addr_17	1143	1
cb_rd_addr_18	1299	1
cb_rd_addr_19	1337	1
cb_rd_addr_20	1118	1
cb_rd_addr_21	1633	1
cb_rd_addr_22	1105	1
cb_rd_addr_23	1114	1
cb_rd_addr_24	1002	1
cb_rd_addr_25	1035	1
cb_rd_addr_26	1152	1
cb_rd_addr_27	1065	1
cb_rd_addr_28	1159	1
cb_rd_addr_29	1140	1
cb_rd_addr_30	1004	1
cb_rd_addr_31	941	1

Figure 3.80: RISCV-V RF read port A coverage

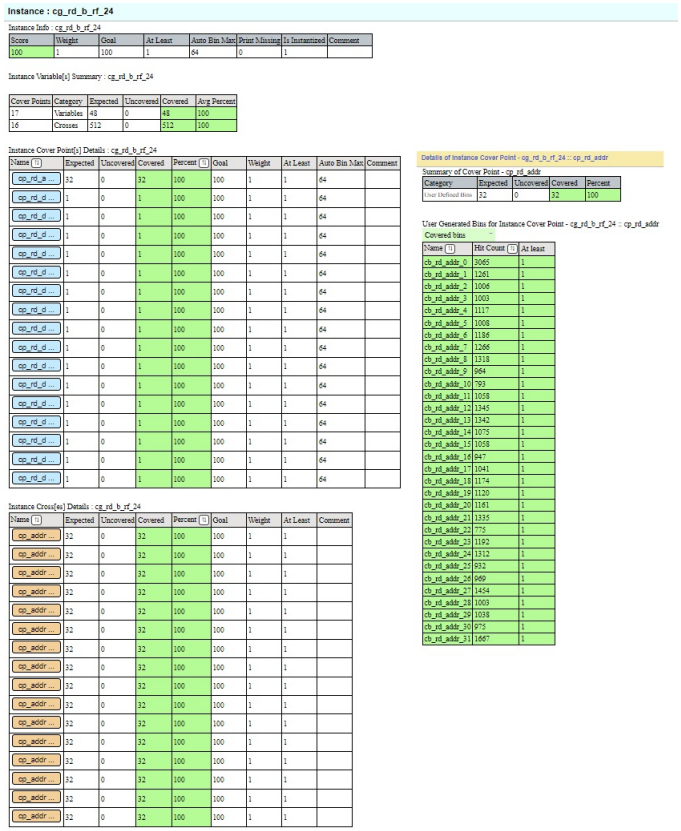


Figure 3.81: RISCV-V RF read port B coverage

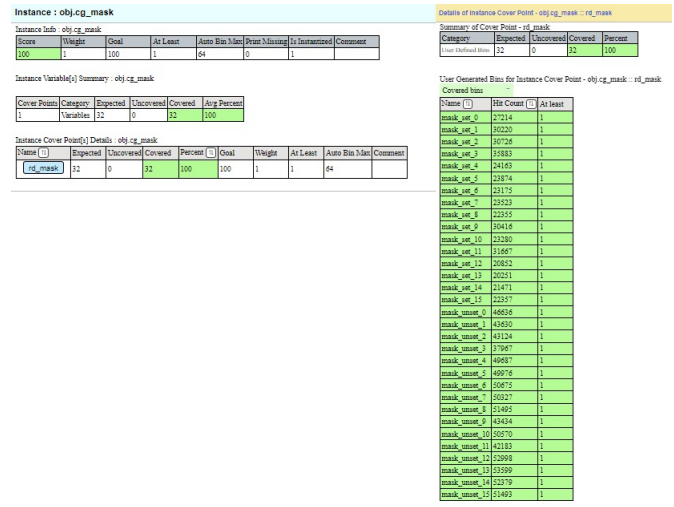


Figure 3.82: RISCV-V RF Mask coverage

CSR

The coverage reports of the Write and Read *CSR* cover groups are shown in Figure 3.83 and 3.84 respectively. Both groups have a score of 100% meaning that all the supported *CSR* configurations are used to execute the vector operations. The configurations that are not supported are excluded from the coverage report, if the configuration that is not supported is sampled in a cover bin within a cover point that has other valid cover bins the cover bin is coded as an *ignore_bin* to exclude the bin from the coverage report, this strategy is used in the read cover group, if the whole cover point needs to be excluded the weight of the cover group is set to zero to exclude it.

In Figure 3.83 the overall score of the write cover group is 100%, but the coverage reports show some cover points that were not hit, these cover points were the points that were excluded from the report by setting the weight to zero, but Vivado has a bug that is showing the metric in the report with a weight of one, even if they are excluded correctly from the overall score.

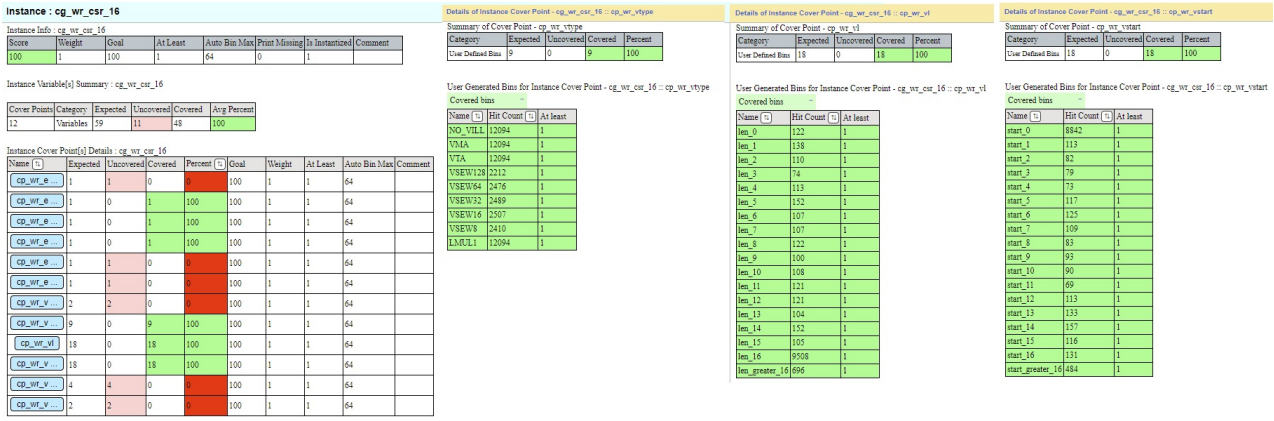


Figure 3.83: RISCV-V CSR write coverage

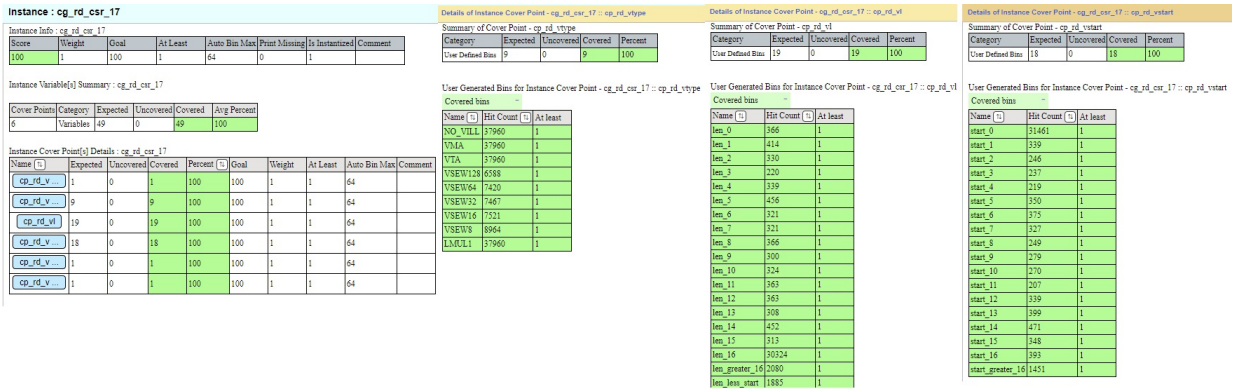
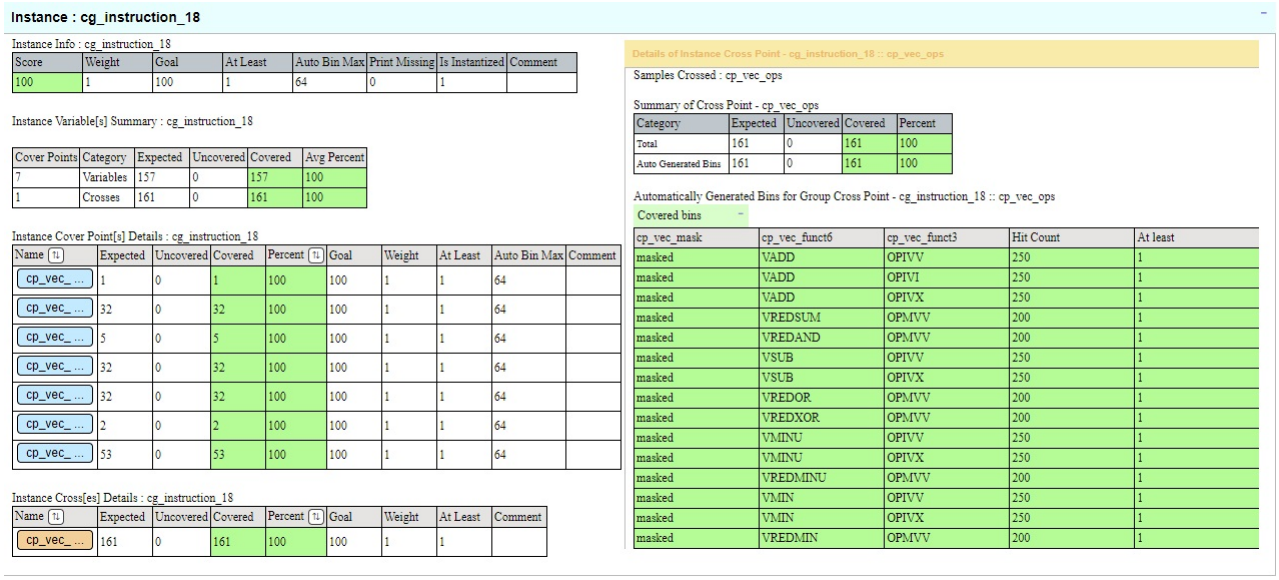


Figure 3.84: RISCV-V CSR read coverage

RISCV-V IF

Figure 3.85 shows the coverage report of the Instruction Fetch coverage collector. This group has a score of 100% meaning that all the fields of the *Vector Instruction* were exercised, the thirty-two registers of the *VRF* were selected as destination, source one and source two registers, masked and unmasked operations were executed and all the *Operations Implemented* were executed.



Instance : cg_instruction_18

Instance Info : cg_instruction_18

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_instruction_18

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
7	Variables	157	0	157	100
1	Crosses	161	0	161	100

Instance Cover Point[s] Details : cg_instruction_18

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_vec_ ...	1	0	1	100	100	1	1	64	
cp_vec_ ...	32	0	32	100	100	1	1	64	
cp_vec_ ...	5	0	5	100	100	1	1	64	
cp_vec_ ...	32	0	32	100	100	1	1	64	
cp_vec_ ...	32	0	32	100	100	1	1	64	
cp_vec_ ...	2	0	2	100	100	1	1	64	
cp_vec_ ...	53	0	53	100	100	1	1	64	

Instance Cross[es] Details : cg_instruction_18

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_vec_ ...	161	0	161	100	100	1	1	

Details of Instance Cross Point - cg_instruction_18 :: cp_vec_ops

Samples Crossed : cp_vec_ops

Summary of Cross Point - cp_vec_ops

Category	Expected	Uncovered	Covered	Percent
Total	161	0	161	100
Auto Generated Bins	161	0	161	100

Automatically Generated Bins for Group Cross Point - cg_instruction_18 :: cp_vec_ops

Covered bins

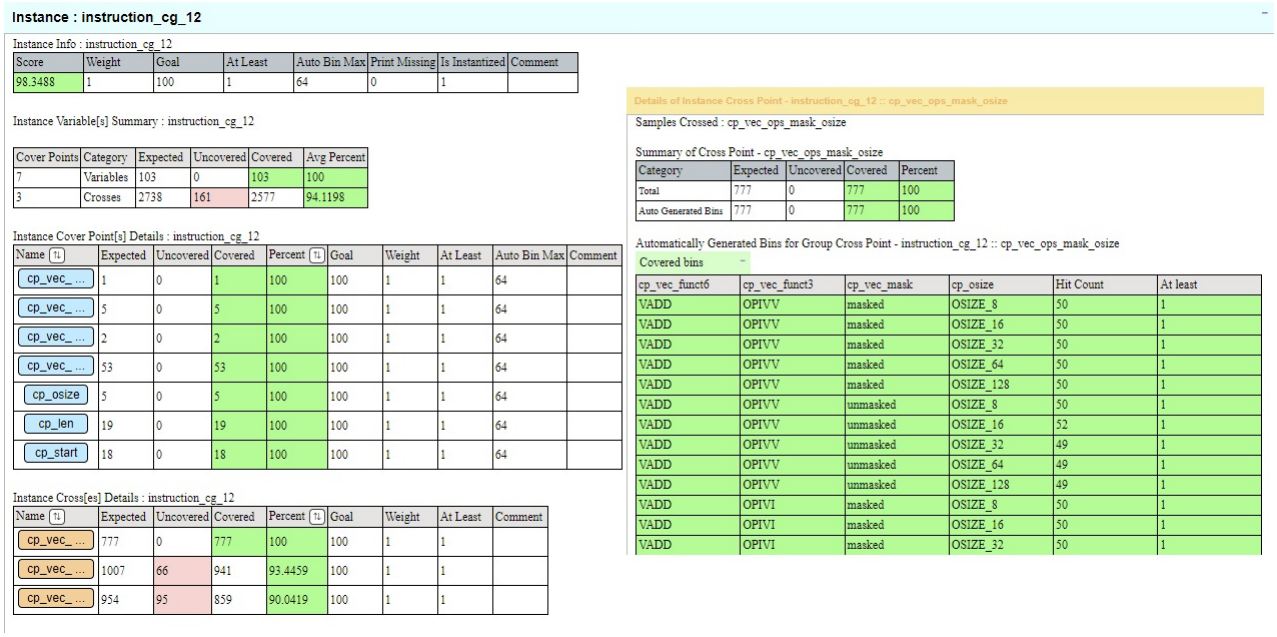
cp_vec_mask	cp_vec_funct6	cp_vec_funct3	Hit Count	At least
masked	VADD	OPIVV	250	1
masked	VADD	OPIVI	250	1
masked	VADD	OPIVX	250	1
masked	VREDSUM	OPMVV	200	1
masked	VREDAND	OPMVV	200	1
masked	VSUB	OPIVV	250	1
masked	VSUB	OPIVX	250	1
masked	VREDOR	OPMVV	200	1
masked	VREDXOR	OPMVV	200	1
masked	VMINU	OPIVV	250	1
masked	VMINU	OPIVX	250	1
masked	VREDMINU	OPMVV	200	1
masked	VMIN	OPIVV	250	1
masked	VMIN	OPIVX	250	1
masked	VREDMIN	OPMVV	200	1

Figure 3.85: RISCV-V IF coverage

RISCV-V

The coverage report of the RISCV-V coverage collector is shown in Figure 3.86. This cover group has a score of 98% meaning that all *Operations Implemented*, *OSIZEs*, *LEN*, *START* and *MASK* were executed but not all the combinations were hit during the tests.

The cross points of *VEC_OP X LEN* and *VEC_OP X START* have some cover bins that were not hit during the tests, this means that not all the *supported operations* were executed with all the valid *LEN* and *START*. The main reason for these holes is the randomization distribution of *LEN* and *START*, as explained in the *Test* section these fields are randomized to be between 0-16 the 80% of the time, for that reason some operations were not executed with *START* or *LEN* values outside of that range.



Instance : instruction_cg_12

Instance Info : instruction_cg_12

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
98.3488	1	100	1	64	0	1	

Instance Variable[s] Summary : instruction_cg_12

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
7	Variables	103	0	103	100
3	Crosses	2738	161	2577	94.1198

Instance Cover Point[s] Details : instruction_cg_12

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_vec_ ...	1	0	1	100	100	1	1	64	
cp_vec_ ...	5	0	5	100	100	1	1	64	
cp_vec_ ...	2	0	2	100	100	1	1	64	
cp_vec_ ...	53	0	53	100	100	1	1	64	
cp_osize	5	0	5	100	100	1	1	64	
cp_len	19	0	19	100	100	1	1	64	
cp_start	18	0	18	100	100	1	1	64	

Instance Cross[es] Details : instruction_cg_12

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_vec_ ...	777	0	777	100	100	1	1	
cp_vec_ ...	1007	66	941	93.4459	100	1	1	
cp_vec_ ...	954	95	859	90.0419	100	1	1	

Details of Instance Cross Point - instruction_cg_12 :: cp_vec_ops_mask_osize

Samples Crossed : cp_vec_ops_mask_osize

Summary of Cross Point - cp_vec_ops_mask_osize

Category	Expected	Uncovered	Covered	Percent
Total	777	0	777	100
Auto Generated Bins	777	0	777	100

Automatically Generated Bins for Group Cross Point - instruction_cg_12 :: cp_vec_ops_mask_osize

Covered bins

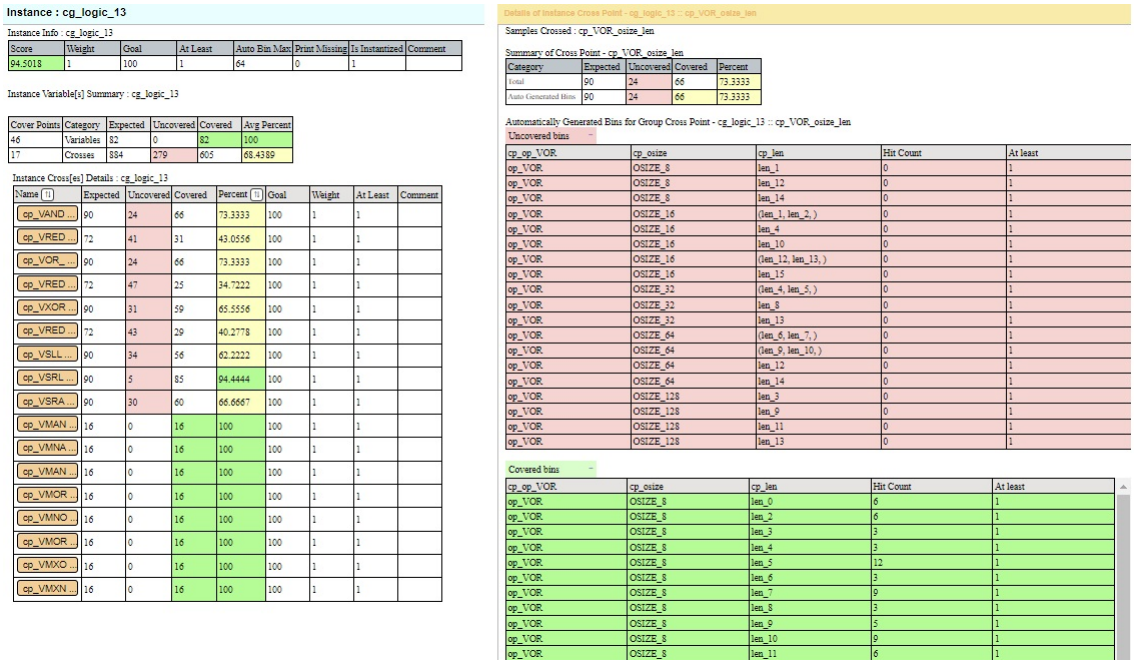
cp_vec_funct6	cp_vec_funct3	cp_vec_mask	cp_osize	Hit Count	At least
VADD	OPIVV	masked	OSIZE_8	50	1
VADD	OPIVV	masked	OSIZE_16	50	1
VADD	OPIVV	masked	OSIZE_32	50	1
VADD	OPIVV	masked	OSIZE_64	50	1
VADD	OPIVV	masked	OSIZE_128	50	1
VADD	OPIVV	unmasked	OSIZE_8	50	1
VADD	OPIVV	unmasked	OSIZE_16	52	1
VADD	OPIVV	unmasked	OSIZE_32	49	1
VADD	OPIVV	unmasked	OSIZE_64	49	1
VADD	OPIVV	unmasked	OSIZE_128	49	1
VADD	OPIVI	masked	OSIZE_8	50	1
VADD	OPIVI	masked	OSIZE_16	50	1
VADD	OPIVI	masked	OSIZE_32	50	1

Figure 3.86: RISCV-V coverage

RISCV-V Logic ALU

The coverage result of the Logic ALU cover group is shown in Figure 3.87 with a score of 94%. In this cover group, all the implemented logical operations were covered along with all its control signals, all the operations were executed with all the valid *OSIZEs* but each combination of *Operation-Osize* was not executed with all the valid *LENs* causing the score to not be 100%.

The vector mask operations ignore the *LEN* and are crossed with the mask value instead that indicates which bit of the operation is valid, in this case, all the mask operations are executed with all the mask bits set with 1'b1 getting a score of 100%.



Instance : cg_logic_13

Instance Info : cg_logic_13

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
94.5018	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_logic_13

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
46	Variables	82	0	82	100
17	Crosses	884	279	605	68.4389

Instance Cross[es] Details : cg_logic_13

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_VAND...	90	24	66	73.3333	100	1	1	
cp_VRED...	72	41	31	43.0556	100	1	1	
cp_VOR_...	90	24	66	73.3333	100	1	1	
cp_VRED...	72	47	25	34.7222	100	1	1	
cp_VXOR...	90	31	59	65.5556	100	1	1	
cp_VRED...	72	43	29	40.2778	100	1	1	
cp_VSLL...	90	34	56	62.2222	100	1	1	
cp_VSRL...	90	5	85	94.4444	100	1	1	
cp_VSRA...	90	30	60	66.6667	100	1	1	
cp_VMAN...	16	0	16	100	100	1	1	
cp_VMNA...	16	0	16	100	100	1	1	
cp_VMAN...	16	0	16	100	100	1	1	
cp_VMOR...	16	0	16	100	100	1	1	
cp_VMNO...	16	0	16	100	100	1	1	
cp_VMOR...	16	0	16	100	100	1	1	
cp_VMXO...	16	0	16	100	100	1	1	
cp_VMXN...	16	0	16	100	100	1	1	

Details of Instance Cross Point - cg_logic_13 :: cp_VOR_osize_len

Samples Crossed : cp_VOR_osize_len

Summary of Cross Point - cp_VOR_osize_len

Category	Expected	Uncovered	Covered	Percent
Total	90	24	66	73.3333
Auto Generated Bins	90	24	66	73.3333

Automatically Generated Bins for Group Cross Point - cg_logic_13 :: cp_VOR_osize_len

Uncovered bins

cp_op_VOR	cp_osize	cp_len	Hit Count	At least
op_VOR	OSIZE_8	len_1	0	1
op_VOR	OSIZE_8	len_12	0	1
op_VOR	OSIZE_8	len_14	0	1
op_VOR	OSIZE_16	(len_1, len_2,)	0	1
op_VOR	OSIZE_16	len_4	0	1
op_VOR	OSIZE_16	len_10	0	1
op_VOR	OSIZE_16	(len_12, len_13,)	0	1
op_VOR	OSIZE_16	len_15	0	1
op_VOR	OSIZE_32	(len_4, len_5,)	0	1
op_VOR	OSIZE_32	len_8	0	1
op_VOR	OSIZE_32	len_13	0	1
op_VOR	OSIZE_64	(len_6, len_7,)	0	1
op_VOR	OSIZE_64	(len_9, len_10,)	0	1
op_VOR	OSIZE_64	len_12	0	1
op_VOR	OSIZE_64	len_14	0	1
op_VOR	OSIZE_128	len_3	0	1
op_VOR	OSIZE_128	len_9	0	1
op_VOR	OSIZE_128	len_11	0	1
op_VOR	OSIZE_128	len_13	0	1

Covered bins

cp_op_VOR	cp_osize	cp_len	Hit Count	At least
op_VOR	OSIZE_8	len_0	6	1
op_VOR	OSIZE_8	len_2	6	1
op_VOR	OSIZE_8	len_3	3	1
op_VOR	OSIZE_8	len_4	3	1
op_VOR	OSIZE_8	len_5	12	1
op_VOR	OSIZE_8	len_6	3	1
op_VOR	OSIZE_8	len_7	9	1
op_VOR	OSIZE_8	len_8	3	1
op_VOR	OSIZE_8	len_9	5	1
op_VOR	OSIZE_8	len_10	9	1
op_VOR	OSIZE_8	len_11	6	1

Figure 3.87: RISCV-V Logic ALU coverage

RISCV-V Arithmetic ALU

The coverage report of the Arithmetic ALU cover group is shown in Figure 3.88 and is similar to the *Logic ALU coverage report*. All the supported arithmetic operations were executed along with all the control signals and all the *OSIZEs* but not all *OPERATION-OSIZE* combinations were executed with all the valid *LENs* causing the score to be only 86%.

Instance : cg_arithmetic_14

Instance Info : cg_arithmetic_14

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
86.6755	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_arithmetic_14

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
62	Variables	104	1	103	99.8992
24	Crosses	2214	1075	1139	51.4453

Instance Cross[es] Details : cg_arithmetic_14

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
cp_VADC ...	90	43	47	52.2222	100	1	1	
cp_VADD ...	90	16	74	82.2222	100	1	1	
cp_VRED ...	72	46	26	36.1111	100	1	1	
cp_VSUB ...	90	56	34	37.7778	100	1	1	
cp_VSUB ...	90	41	49	54.4444	100	1	1	
cp_VMIN ...	90	43	47	52.2222	100	1	1	
cp_VMIN ...	90	39	51	56.6667	100	1	1	
cp_VRED ...	72	48	24	33.3333	100	1	1	
cp_VRED ...	72	48	24	33.3333	100	1	1	
cp_VMAX ...	90	35	55	61.1111	100	1	1	
cp_VMAX ...	90	39	51	56.6667	100	1	1	
cp_VRED ...	72	49	23	31.9444	100	1	1	
cp_VRED ...	72	42	30	41.6667	100	1	1	
cp_VSEQ ...	90	27	63	70	100	1	1	
cp_VSNE ...	90	28	62	68.8889	100	1	1	
cp_VSLE ...	90	25	65	72.2222	100	1	1	
cp_VSLE ...	90	19	71	78.8889	100	1	1	
cp_VSGT ...	90	40	50	55.5556	100	1	1	
cp_VSGT ...	90	41	49	54.4444	100	1	1	
cp_VMUL ...	90	28	62	68.8889	100	1	1	
cp_VMUL ...	90	39	51	56.6667	100	1	1	
cp_VMUL ...	90	41	49	54.4444	100	1	1	
cp_VZEX ...	162	123	39	24.0741	100	1	1	
cp_VSEX ...	162	119	43	26.5432	100	1	1	

Details of Instance Cross Point - cg_arithmetic_14 :: cp_VZEXT_osize_len

Samples Crossed : cp_VZEXT_osize_len

Summary of Cross Point - cp_VZEXT_osize_len

Category	Expected	Uncovered	Covered	Percent
Total	162	123	39	24.0741
Auto Generated Bins	162	123	39	24.0741

Automatically Generated Bins for Group Cross Point - cg_arithmetic_14 :: cp_VZEXT_osize_len

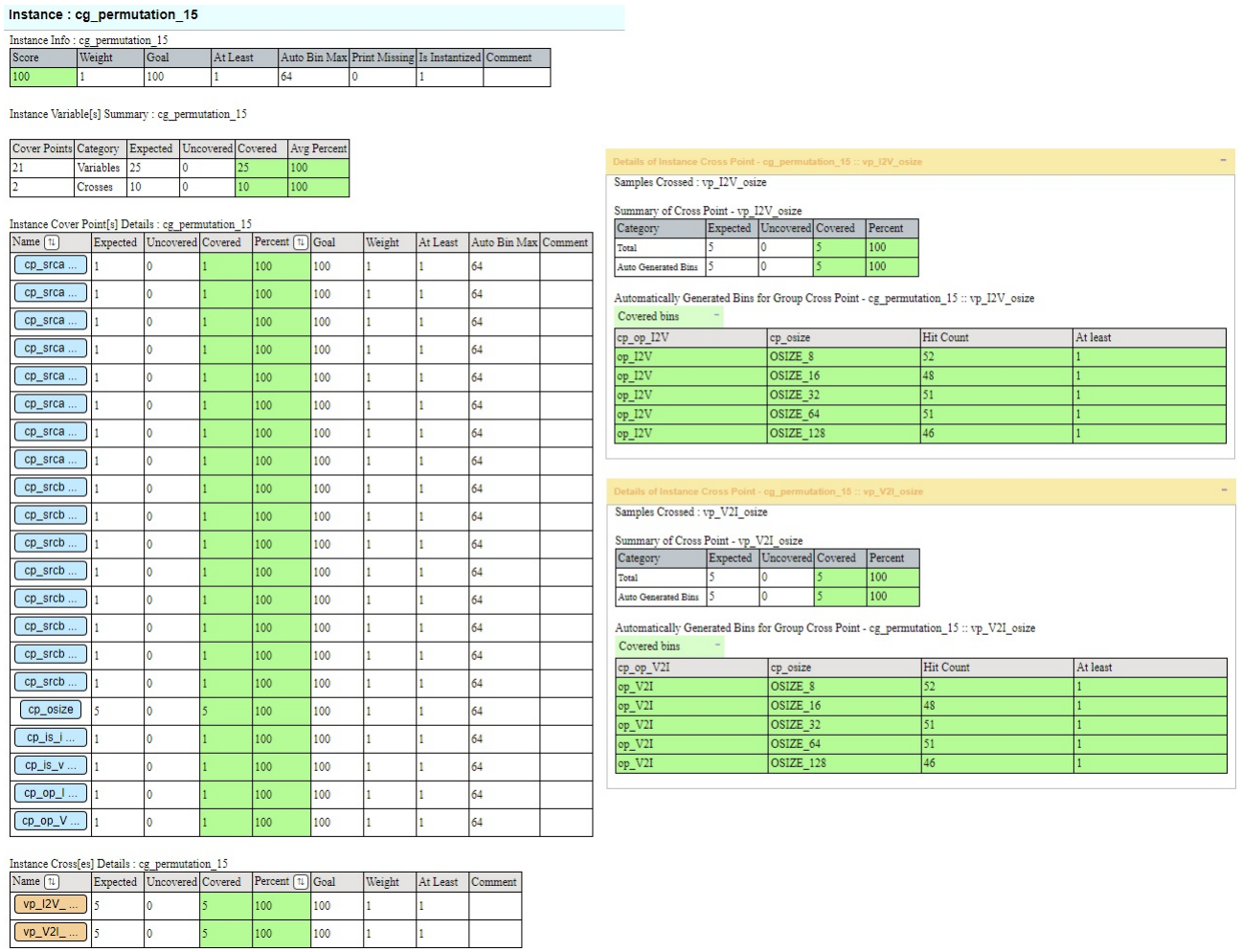
Uncovered bins

cp_op_VZEXT	cp_src_osize	cp_dst_osize	cp_len	Hit Count	At least
op_VZEXT	OSIZE_8	OSIZE_16	len_1	0	1
op_VZEXT	OSIZE_8	OSIZE_16	(len_3, len_4,)	0	1
op_VZEXT	OSIZE_8	OSIZE_16	len_6	0	1
op_VZEXT	OSIZE_8	OSIZE_16	len_8	0	1
op_VZEXT	OSIZE_8	OSIZE_16	len_10	0	1
op_VZEXT	OSIZE_8	OSIZE_16	(len_13, len_14, len_15,)	0	1
op_VZEXT	OSIZE_8	OSIZE_16	len_greater_16	0	1
op_VZEXT	OSIZE_8	OSIZE_32	(len_0, len_1, len_2, len_3, len_4, len_5, len_6,)	0	1
op_VZEXT	OSIZE_8	OSIZE_32	(len_8, len_9,)	0	1
op_VZEXT	OSIZE_8	OSIZE_32	len_11	0	1
op_VZEXT	OSIZE_8	OSIZE_32	(len_14, len_15,)	0	1
op_VZEXT	OSIZE_8	OSIZE_64	(len_0, len_1, len_2, len_3, len_4,)	0	1

Figure 3.88: RISCV-V Arithmetic ALU coverage

RISCV-V Permutation ALU

In Figure 3.89 the coverage report of the permutation ALU cover group is shown. It is similar to the *logic* and *arithmetic* ALU cover groups, but in this case this cover groups has a score of 100% since the permutation operations ignore the *LEN* field, for that reason the operations are only crossed with *OSIZE* getting a score of 100%



Instance : cg_permutation_15

Instance Info : cg_permutation_15

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
100	1	100	1	64	0	1	

Instance Variable[s] Summary : cg_permutation_15

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
21	Variables	25	0	25	100
2	Crosses	10	0	10	100

Instance Cover Point[s] Details : cg_permutation_15

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srca ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_srcb ...	1	0	1	100	100	1	1	64	
cp_osize	5	0	5	100	100	1	1	64	
cp_is_i ...	1	0	1	100	100	1	1	64	
cp_is_v ...	1	0	1	100	100	1	1	64	
cp_op_I ...	1	0	1	100	100	1	1	64	
cp_op_V ...	1	0	1	100	100	1	1	64	

Instance Cross[es] Details : cg_permutation_15

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Comment
vp_I2V_...	5	0	5	100	100	1	1	
vp_V2I_...	5	0	5	100	100	1	1	

Details of Instance Cross Point - cg_permutation_15 :: vp_I2V_osize

Samples Crossed : vp_I2V_osize

Summary of Cross Point - vp_I2V_osize

Category	Expected	Uncovered	Covered	Percent
Total	5	0	5	100
Auto Generated Bins	5	0	5	100

Automatically Generated Bins for Group Cross Point - cg_permutation_15 :: vp_I2V_osize

Covered bins

cp_op_I2V	cp_osize	Hit Count	At least
op_I2V	OSIZE_8	52	1
op_I2V	OSIZE_16	48	1
op_I2V	OSIZE_32	51	1
op_I2V	OSIZE_64	51	1
op_I2V	OSIZE_128	46	1

Details of Instance Cross Point - cg_permutation_15 :: vp_V2I_osize

Samples Crossed : vp_V2I_osize

Summary of Cross Point - vp_V2I_osize

Category	Expected	Uncovered	Covered	Percent
Total	5	0	5	100
Auto Generated Bins	5	0	5	100

Automatically Generated Bins for Group Cross Point - cg_permutation_15 :: vp_V2I_osize

Covered bins

cp_op_V2I	cp_osize	Hit Count	At least
op_V2I	OSIZE_8	52	1
op_V2I	OSIZE_16	48	1
op_V2I	OSIZE_32	51	1
op_V2I	OSIZE_64	51	1
op_V2I	OSIZE_128	46	1

Figure 3.89: RISCV-V Permutation ALU coverage

Coverage completion

As described in previous sections, not all the cover groups have a score of 100% mainly for the *LEN*, *START* crosses that were not hit with all the *supported operations*. For that reason to increase the coverage score, two more regressions were run modifying the randomization distribution of *LEN* and *START* to increase the probability of getting the values that were not hit during the first regression and incrementing the size of the operation block from 50 to 100 to get more operations and also increase the probability of hitting the missing cover bins.

After running the two extra regressions and merging the coverage databases with the original database the overall coverage score increased to 100%, meaning that all the cover

bins that were missed in the original regression were hit, this means that all the *supported operations* were executed along with all the supported *OSIZEs*, *LEN* and *START* combinations.

It is important to note that the cross covers *(OPERATION X OSIZE X LEN)* and *(OPERATION X OSIZE X START)* are two separate cover points, meaning that not all the *LEN-START* combinations are covered, if both cover points are merged into a single cover point it would increase the number of cover bins exponentially and Vivado would not be able to support that amount of cover bins.

The final result of the coverage report is shown in Figure 3.90.

Groups Coverage Summary

Score	Inst Score
100	100

Group - riscv_rf_agt_pkg::riscv_rf_cov::cg_wr_rf

Group Info : cg_wr_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_rf_agt_pkg::riscv_rf_cov::cg_rd_a_rf

Group Info : cg_rd_a_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_rf_agt_pkg::riscv_rf_cov::cg_rd_b_rf

Group Info : cg_rd_b_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_rf_agt_pkg::riscv_v_rf_cov::cg_wr_rf

Group Info : cg_wr_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_rf_agt_pkg::riscv_v_rf_cov::cg_rd_a_rf

Group Info : cg_rd_a_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_rf_agt_pkg::riscv_v_rf_cov::cg_rd_b_rf

Group Info : cg_rd_b_rf

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_rf_agt_pkg::riscv_v_rf_cov::cg_mask

Group Info : cg_mask

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	0	64	0	

Group - riscv_v_alu_agt_pkg::riscv_v_alu_cov::cg_logic

Group Info : cg_logic

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_alu_agt_pkg::riscv_v_alu_cov::cg_arithmetic

Group Info : cg_arithmetic

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_alu_agt_pkg::riscv_v_alu_cov::cg_permutation

Group Info : cg_permutation

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_if_agt_pkg::riscv_v_if_cov::cg_instruction

Group Info : cg_instruction

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_csr_agt_pkg::riscv_v_csr_cov::cg_wr_csr

Group Info : cg_wr_csr

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_csr_agt_pkg::riscv_v_csr_cov::cg_rd_csr

Group Info : cg_rd_csr

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Group - riscv_v_tb.int_rf_vif.int_rf_vif.int_rf_vif.vec_rf_vif.vec_rf_vif.vec_rf_vif.vec_ari ...

Group Info : instruction_cg

Group Hier Name :

riscv_v_tb.int_rf_vif.int_rf_vif.int_rf_vif.vec_rf_vif.vec_rf_vif.vec_rf_vif.vec_arithmetic_alu_vif.vec_arithmetic_alu_vif.vec_arithmetic_alu_vif.vec_logic_alu_vif.vec_lo

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
100	1	100	1	100	0	0	1	64	0	

Figure 3.90: RISCV-V Coverage Complete

Chapter 4

Synthesis

To perform the synthesis, five different platforms were used: one FPGA platform and four ASIC platforms.

The FPGA platform used is the ARTIX7 FPGA from Xilinx built on the 28nm HPL process, specifically the chip xc7a200tfbq676-2. This chip is the chip with more resources in the ARTIX7 family, provides 215,360 logic devices, 740 DSPs and 500 I/O pins. This FPGA can also be programmed using the free version of Vivado, which is the tool used to perform the synthesis of the circuit to get the metrics.

The ASIC synthesis is performed using open source software, the software used is *OpenROAD*. *openROAD* is a fully autonomous RTL to GDSII flow for rapid design exploration and physical design implementation. *openROAD* makes use of other open-source softwares like the logic synthesizer Yosys to enable the full flow. The flow starts at the RTL with the logical synthesis, mapping the logical synthesis to the technology cells, defining the floor plan of the circuit, placing the cells, defining the clock tree synthesis, routing the design and finishing with the GDSII layout as shown in Figure 4.1 gotten from the official *openROAD-flow repository* [15].

This repository provides synthesis scripts that were used to perform the synthesis of the circuit, the scripts can be modified to perform fine tuning but they were used as they are presented in the original repository without modifying the scripts, just some configuration variables like the clock period, the layers to route the design and the core area were modified to perform the synthesis according to the circuit necessities.

The repository also includes four open source PDKs that were used to synthesize the circuit.

The first PDK is the *GF180MCU* PDK. This is an open source PDK that was developed by Global Foundries in collaboration with Google. The PDK provides the resources needed to enable the creation of designs manufactured at the Global Foundries facilities using its 180nm process [16]. The PDK is still at alfa release and it is not intended to be used in production, just for research and testing purposes. The PDK can be found in this repository *GF180*.

The second PDK is the *SKY130* PDK. This open source PDK is a collaboration between SkyWater Technology Foundry and Google. As the previous PDK is still at alfa release and is intended to enable the creation of designs that are manufacturable at SkyWater's facilities using its 130nm process [17]. The node used by this PDK is a 180nm-130nm hybrid technology. The PDK can be found in this repository *SKY130*.

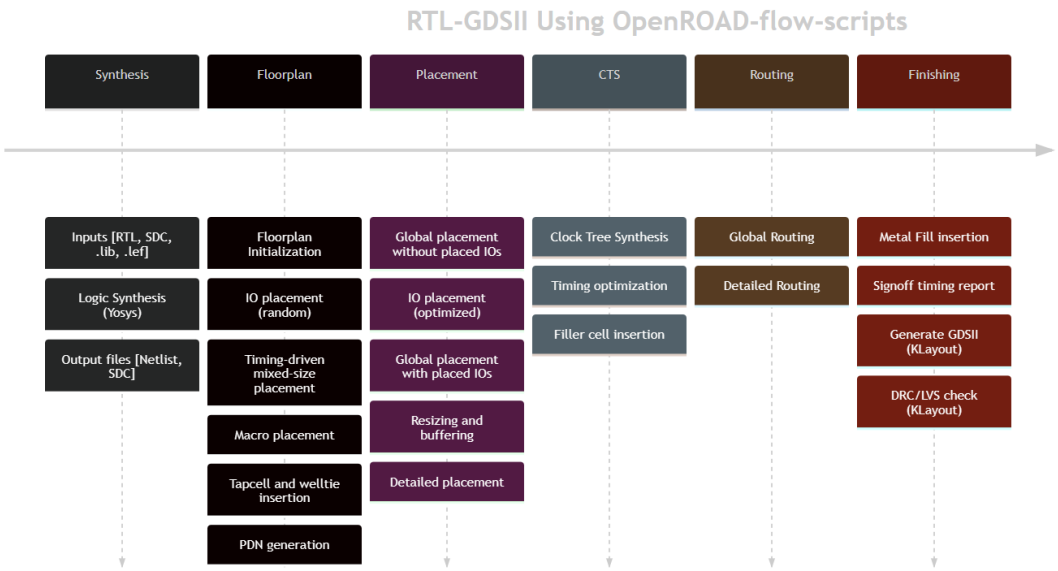


Figure 4.1: OpenROAD flow [15].

The third PDK is the *FREEPDK45* PDK. This PDK unlike the previous PDK is a generic 45nm PDK that is not mapped to a technology or a fabrication node, which means that the circuit can not be fabricated using this PDK [18]. It was developed by the NC State University and can be obtained using SVN from this repository *FREEPDK45*.

The last PDK is the *ASAP7* PDK. This is an open-source academic PDK developed by the Arizona State University and ARM using a FinFet 7nm Technology. As the *FREEPDK45* this PDK is only intended for research and academic purposes and can not be manufactured at any fabric [19]. The PDK can be found at this repository *ASAP7*.

The *RISCV Vector Engine* was synthesized using the four ASIC PDKs and the FPGA platform but the main analysis was done using only the *ASAP7* PDK.

The synthesis was performed using each module as the top module to get the statistics of that module. Two clock periods were used, one fast clock was used for small modules that can achieve a higher frequency and one slow clock that is the same frequency used by the whole *RISCV Vector Engine* and was used to synthesize the *Execution Unit*, *Arithmetic ALU*, *Vector Adder* and *Vector Multiplier* that are the units that have the longest latencies and need a slower clock.

The *CORE AREA* used for all the units was the same *CORE AREA* that was used to synthesize the *RISCV Vector Engine* so most of the area in the units is empty, this lead to an issue with the synthesizer, in some modules the synthesizer groups all the logic in a small area increasing the density of that area as it is done in the *RISCV Vector Engine*, but in some modules the synthesizer decides to spread the logic along all the *CORE AREA*, leaving big holes between the cells and increasing the propagation delay due to the distance between the cells.

The logic depth of the units only includes the logic cells used to implement the RTL excluding the buffers and inverters added by the synthesizer, but the slack includes all the

paths, including the buffers, inverters and the propagation delay of the wires.

The power statistics of the modules should be taken cautiously since they are returned by the tool but it was not configured properly with a UPF file, a simulation activity file or with a probability and toggling file. For that reason, the power metrics could not be accurate, especially in the full combinational units where the clock is not present and the tool has some issues calculating the power, in some cases using the maximum toggling frequency of the cells/LUTs to calculate the power and increasing the power consumption. The power statistics are only used for comparison and to get an idea of what could be the power consumption of the circuit but they do not represent real values.

4.1 Multipliers

To measure the performance of the *Array Multiplier* and the *Vedic Multiplier* both multipliers were synthesized using the ARTIX7 and ASAP7 platforms and using different widths. The *Vedic Multiplier* was synthesized as it is used in the *RISCV Vector Engine*, instantiating one N-bits unsigned *Vedic Multiplier* that instantiates recursively more *Vedic Multiplier* with different widths and using *Ripple Carry Adders* to perform the additions. The *Array Multiplier* was synthesized using three different configurations, using only the operator ”*” to let the synthesizer to select the multiplier to instantiate, instantiating one *Array Multiplier* but using the operator ”+” to perform the additions of the partial products and letting the synthesizer to select the adder to instantiate and the third configuration is instantiating one *Array Multiplier* and using *Ripple Carry Adders* to perform the additions. Multiple synthesis runs were performed to get the maximum frequency of each multiplier and the core area was selected to fit the design without having too much free space to increase the density of the cells and reduce the propagation delay of the wires to increment the maximum frequency.

4.1.1 Unsigned Multiplier

The Figure 4.2 shows the RTL schematic of an 8-bit *Array Multiplier*. It instantiates eight layers of eight AND gates to generate the partial products and seven 8-bit Adders to add the partial products, it also instantiates registers at the beginning and at the end of the data path to measure the the maximum frequency without any input/output delay.

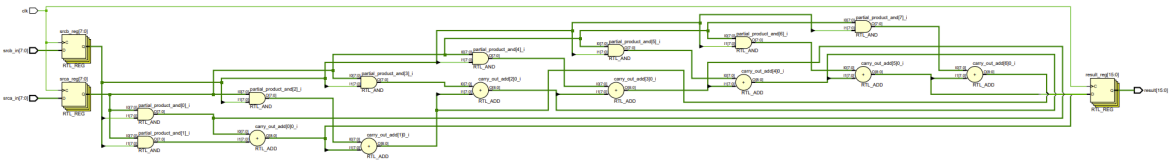


Figure 4.2: Mul array 8-Bits RTL Schematic

The Figure 4.3 shows the RTL schematic of an 8-bit *Vedic Multiplier*. It instantiates one 8-bit *Vedic Multiplier* that instantiates four 4-bit *Vedic Multipliers* and three 8-bit *Ripple Carry Adders*. As the *Array Multiplier* it also instantiates registers at the beginning and at the end of the data path to measure the maximum frequency without any input/output delay.

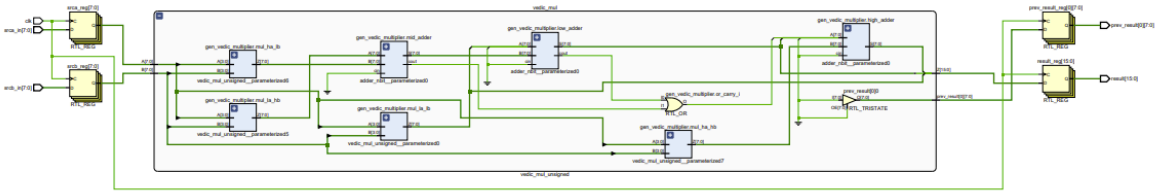


Figure 4.3: Mul Vedic 8-Bits RTL Schematic

Starting the analysis with the ARTIX7 platform the best performance is gotten when the ”*” operator is used. This operator gets the least use of LUTs and the highest frequency, the reason for this is the architecture of the FPGA, the ARTIX7 includes some DSPs that can be used as multipliers, so for multipliers with a WIDTH equal to or greater than sixteen the FPGA uses the embedded multipliers instead of the LUTs with the embedded multipliers being smaller and faster than the LUTs, the 16-bit Multiplier instantiates one DSP while the 128-bit Multiplier instantiates sixty-four DSPs. If the FPGA does not include embedded multipliers the performance will be different.

If an *Array Multiplier* module is instantiated using the operator ”+” to perform the partial products additions the FPGA does not instantiate a DSP as a multiplier, but uses the embedded adders and special LUTs to implement the adders, in this case, the *Vedic Multiplier* is faster than the *Array Multiplier* being up to three times faster than the *Array Multiplier*, but also consuming twice the LUTs in most of the cases, it consumes more LUTs in all the widths but the speed increment is only achieved if the WIDTH is equal or greater than sixteen and being every time faster than the *Array Multiplier* if the WIDTH increments.

The worst performance in terms of LUTs and speed is gotten when an *Array Multiplier* module is instantiated and *Ripple Carry Adders* are also instantiated to perform the additions, this is the slowest multiplier, it is just a little bit slower than the *Array Multiplier* implemented with the ”+” operator but consumes more than two times the LUTs than the *Array Multiplier* with the ”+” operator and even more LUTs than the *Vedic Multiplier*. Theoretically the *Vedic Multiplier* consumes more area and instantiates more gates than the *Array Multiplier*, but this architecture of FPGA has some problems synthesizing big adders and consumes many LUTs, the *Vedic Multiplier* instantiates more adders but the adders are smaller in width, it only instantiates three 128-bit adders while the *Array Multiplier* instantiates 127 128-bit adders.

The implementation of the *Vedic Multiplier* in an FPGA depends on the architecture of the FPGA, if the FPGA has embedded multipliers and adders the best performance is gotten by using the operators ”*” and ”+” to perform the operations to let the synthesizer to use the embedded resources, but if the FPGA doesn’t have those resources nor special LUTs to implement arithmetic operations the *Vedic Multiplier* could have a better performance in terms of area and speed. It also depends on the WIDTH of the multiplier, if the WIDTH is small the speed gain is small but the number of LUTs increases, but as the WIDTH increments the gain in speed also increments considerably.

Now analyzing the ASIC implementation. According to the Yosys documentation if the ”+” operator is used the synthesizer implements *Carry Look-Ahead* cells, and if the operator

Table 4.1: Unsigned Multiplier synthesis statistics

ARCHITECTURE	Array															Vedic				
PLATFORM	ARTIX7						ASAP7									ARTIX7		ASAP7		
WIDTH	LUTs			Max Freq			Instances			Area			Max Freq			LUTs	Max Freq	Instances	Area	Max Freq
	*	+	RP	*	+	RP	*	+	RP	*	+	RP	*	+	RP					
2	2	2	2	100Mhz	100Mhz	100Mhz	6	13	5	0.70	1.05	0.85	9.1Ghz	9Ghz	8.9Ghz	4(200%)(200%)(200%)	100Mhz(100%)(100%)(100%)	5(83%)(38%)(100%)	0.52(74%)(50%)(74%)	9.1Ghz(100%)(101%)(102%)
4	16	19	16	100Mhz	100Mhz	100Mhz	75	71	27	8.94	7.89	4	4.1GHz	4.1Ghz	4.6Ghz	33(206%)(173%)(206%)	100Mhz(100%)(100%)(100%)	33(44%)(46%)(122%)	4.7(53%)(60%)(117%)	5Ghz(122%)(122%)(108%)
8	70	57	127	100Mhz	90.9Mhz	90.9Mhz	364	460	119	45.77	57.62	17	2.2Ghz	1.3Ghz	2.5Ghz	118(168%)(207%)(92%)	100Mhz(100%)(110%)(110%)	157(43%)(34%)(131%)	23.6(52%)(41%)(138%)	3Ghz(136%)(230%)(120%)
16	(0- 1)	241	596	66Mhz	50Mhz	41.6Mhz	1276	2693	495	170	303.7	71	1.3Ghz	526Mhz	1.25Ghz	517(51700%)(214%)(86%)	50Mhz(75%)(100%)(120%)	677(53%)(25%)(136%)	104(61%)(34%)(146%)	1.9Ghz(146%)(361%)(152%)
32	(47-4)	993	3003	55Mhz	24.4Mhz	23Mhz	4929	12613	2015	679	1567	292	1Ghz	235Mhz	625Mhz	2555(5436%)(257%)(85%)	47.7Mhz(86%)(195%)(207%)	2805(57%)(22%)(139%)	437(64%)(28%)(149%)	1.3G(130%)(553%)(208%)
64	(161-16)	4033	13697	55Mhz	11.8Mhz	11.2Mhz	18956	51011	8127	2678	6352	1181	791Mhz	117Mhz	294Mhz	8675(5388%)(215%)(63%)	31.2Mhz(56%)(264%)(278%)	11413(60%)(22%)(140%)	1782(66%)(28%)(150%)	1Ghz(126%)(854%)(340%)
128	(899-64)	16257	40619	55Mhz	5.9Mhz	5.8Mhz	73013	203987	32639	10408	25556	4752	606Mhz	59Mhz	156Mhz	3632O(4040%)(223%)(89%)	17.9Mhz(32%)(303%)(308%)	46037(63%)(22%)(141%)	7208(69%)(28%)(151%)	730Mhz(120%)(1237%)(467%)

"*" is used the synthesizer implements an *Array Multiplier* and makes use of *Kogge-Stone adders* to add the partial products.

The faster multiplier is the *Vedic Multiplier* but it is not the multiplier that uses the most area. The multiplier that uses the less area as expected is the *Array Multiplier* with *Ripple Carry Adders* but contrary to expectations it is not the slowest multiplier. The Multiplier that uses the operator "+" and instantiates *Carry Look-Ahead* adders in paper should be faster than the multiplier with *Ripple Carry Adders* since the *Carry Look-Ahead* adders are faster, but it seems that this version of the synthesizer has some issues synthesizing big *Carry Look-Ahead* adders consuming a lot of gates/area and degrading the speed performance. Analyzing the synthesis results the main issue seems to be the placement of the cells, the synthesizer instead of instantiating many small *Carry Look-Ahead* modules instantiates one big module, causing the increment of the size of the circuit, spreading the cells along all the core area and incrementing the routing delay due to the distance of the cells.

The second faster multiplier is the multiplier with *Kogge-Stone adders* but it is also the second multiplier that more area consumes, this multiplier is faster than the multiplier with *Ripple Carry Adders* since the *Kogge-Stone adders* are faster but also are bigger. This multiplier consumes more area than the *Vedic Multiplier* and it is still slower than the *Vedic Multiplier*.

In conclusion for ASIC implementations the use of the "*" and "+" operators depends completely on the synthesizer that is used, the configuration of the synthesizer and the cells available in the PDK. For some applications, these operators could be used and reduce the time of coding and verifying the RTL. However, for more specific implementations, it may be more appropriate to directly instantiate or implement the desired multiplier to ensure that the synthesized hardware matches the design requirements.

In terms of performance, in paper and real implementation the *Vedic Multiplier* is faster than the *Array Multiplier* even if the *Vedic Multiplier* is implemented using *Ripple Carry Adders* and the *Array Multiplier* using *Kogge-Stone adders*.

The *Vedic Multiplier* could improve its speed performance by using a different type of adder like the *Kogge-Stone adder* but this also increments the area, for that reason the implementation with *Ripple Carry Adders* was selected for this implementation to reduce the area of the multiplier.

The critical path of the 128-bit *Vedic Multiplier* has 39 levels of logic. The *Array Multiplier* with *Kogge-Stone adders* critical path has 47 levels of logic. The *Array Multiplier* implemented with *Ripple Carry Adders* has 257 levels of logic. And the *Array Multiplier* implemented with *CLU Adders* has 543 levels of logic..

The synthesizer statistics are shown in Table 4.1.

Table 4.2: Vector Multiplier synthesis statistics

ARCHITECTURE	Array					Vedic				
PLATFORM	ARTIX7		ASAP7			ARTIX7		ASAP7		
OSIZEs	LUTs	Max Freq	Gates	Area	Max Freq	LUTs	Max Freq	Gates	Area	Max Freq
8,16,32,64,128	(2933-120)-31504(100%)79754	55.5M-5.9Mhz(100%)-5.8Mhz	106623-273076(100%)-49021	15200-32071(100%)-8421	530Mhz-54Mhz(100%)-156Mhz	36393(115%)	17.5Mhz(296%)	47468	7374	700Mhz

4.1.2 Vector Multiplier

The Figure 4.4 shows a vector multiplier implemented using *Array Multipliers*. This vector multiplier instantiates multiple *Array Multipliers*, instantiates sixteen 8-bit multipliers, eight 16-bit multipliers, four 32-bit multipliers, two 64-bit multipliers and one 128-bit multiplier. After the multipliers two 128-bit multiplexors are instantiated to select the correct result from the different multipliers depending on the *OSIZE*, one multiplexor is used to multiplex the low part of the result and the second one to multiplex the high part.

The Figure 4.5 shows a vector multiplier implemented using a *Vedic Multiplier*. Unlike the vector multiplier implemented using *Array Multipliers* this multiplier only needs one 128-bit *Vedic Multiplier* and the results of the different *OSIZEs* can be gotten directly from the internal multipliers used by the *Vedic Multiplier*. As the vector multiplier implemented with *Array Multipliers* this multiplier also instantiates two 128-bit multiplexors to select the correct data depending on the *OSIZE*.

Analyzing the synthesis results when the ARTIX7 platform is used using the "*" returns the best results, this implementation is the one that uses fewer LUTs, makes use of the integrated DSPs using 120 DSPs as shown in Figure 4.6, and it is the multiplier that operates with the highest frequency. If the FPGA's embedded resources are not used the *Vedic Multiplier* presents a better performance in terms of LUTs and speed since it only needs to instantiate one 128-bit multiplier instead of multiple multipliers like the *Array Multiplier* implementation.

Analyzing the ASIC implementation the best performance in terms of area and speed is obtained with the *Vedic Multiplier*, this multiplier has the highest maximum frequency and consumes less area even that the *Array Multiplier*. The maximum frequency and area are similar to the statistics of the 128-bit *Vedic Multiplier* since both multipliers only instantiate 1 128-bit *Vedic Multiplier*, they are slightly different due to placement of the cells and the 128-bit multiplexors used to get the final result.

The statistics of the vector multiplier with both implementations can be observed in Table 4.2.

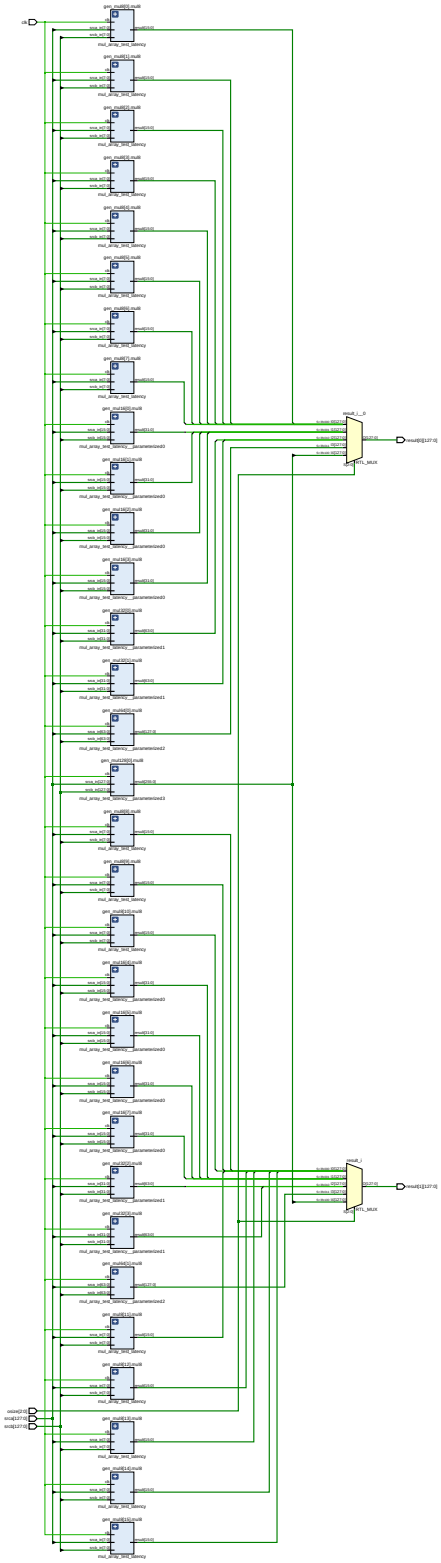


Figure 4.4: Vector array multiplier RTL Schematic

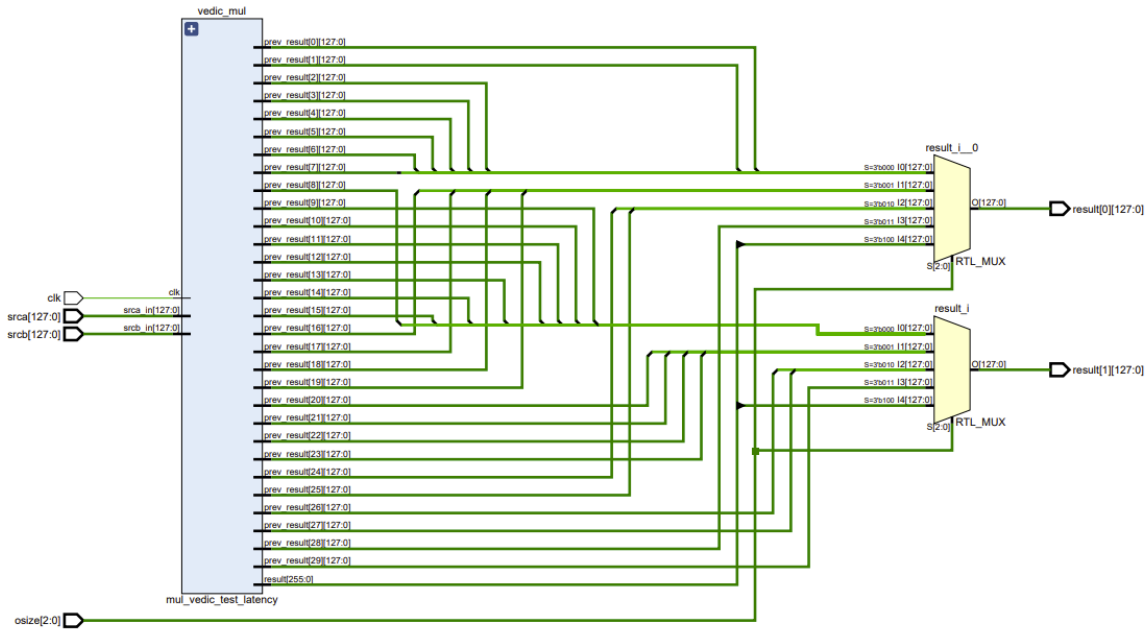


Figure 4.5: Vector Vedic multiplier RTL Schematic

Name	Used
mul_vec_array_test_latency	120
gen_mul128[0].mul128 (mul_array_test_latency__parameterized3)	64
gen_mul64[0].mul64 (mul_array_test_latency__parameterized2)	16
gen_mul64[1].mul64 (mul_array_test_latency__parameterized2_10)	16
gen_mul32[0].mul32 (mul_array_test_latency__parameterized1)	4
gen_mul32[1].mul32 (mul_array_test_latency__parameterized1_7)	4
gen_mul32[2].mul32 (mul_array_test_latency__parameterized1_8)	4
gen_mul32[3].mul32 (mul_array_test_latency__parameterized1_9)	4
gen_mul16[0].mul16 (mul_array_test_latency__parameterized0)	1
gen_mul16[1].mul16 (mul_array_test_latency__parameterized0_0)	1
gen_mul16[2].mul16 (mul_array_test_latency__parameterized0_1)	1
gen_mul16[3].mul16 (mul_array_test_latency__parameterized0_2)	1
gen_mul16[4].mul16 (mul_array_test_latency__parameterized0_3)	1
gen_mul16[5].mul16 (mul_array_test_latency__parameterized0_4)	1
gen_mul16[6].mul16 (mul_array_test_latency__parameterized0_5)	1
gen_mul16[7].mul16 (mul_array_test_latency__parameterized0_6)	1

Figure 4.6: FPGA vector multiplier DSP use

4.2 RISCV-V Decode

The Figure 4.7 shows the RTL schematic of the *RISCV-V Decode* module that instantiates the *Control Unit*, *VRF Control Unit*, *VRF*, *CSR Control Unit* and the *CSR*.

In the RTL schematic is observed how most of the outputs of the *Control Unit* are connected to the output interface of the *RISCV-V Decode* module, these signals are the ALU control signals used to execute the operations.

At the top of the diagram the *CSR Control Unit* and *CSR* modules are instantiated, the *CSR Control Unit* sending control signals to the *CSR* and the *CSR* sending the values of the *CSRs* to the *Execution Unit*

At the bottom part of the diagram the *VRF Control Unit* and *VRF* are instantiated, similar to the *CSR* blocks the *VRF Control Unit* sends control units to the *VRF* and the VRF outputs the data of the registers to the *Execution Unit*.

The synthesis statistics of the *RISCV-V Decode* module are shown in Table 4.3. The ASIC implementation instantiates 4909 flops while the FPGA implementation instantiates 26 extra flops. Most of the area is used by combinational cells, approximately 70% of the area while the 30% remaining is used by the flops.

The critical path of the *RISCV-V Decode* module has a depth of seven levels of logic, it starts at the *VS1/VS2* instruction field, goes through the *VRF* reading the registers and ends at the stage register at *VRF control* unit. Most of the area is used by the *VRF* and the synthesizer is splitting the registers into two parts, one part is located at the top left corner and the second part at the bottom right corner.

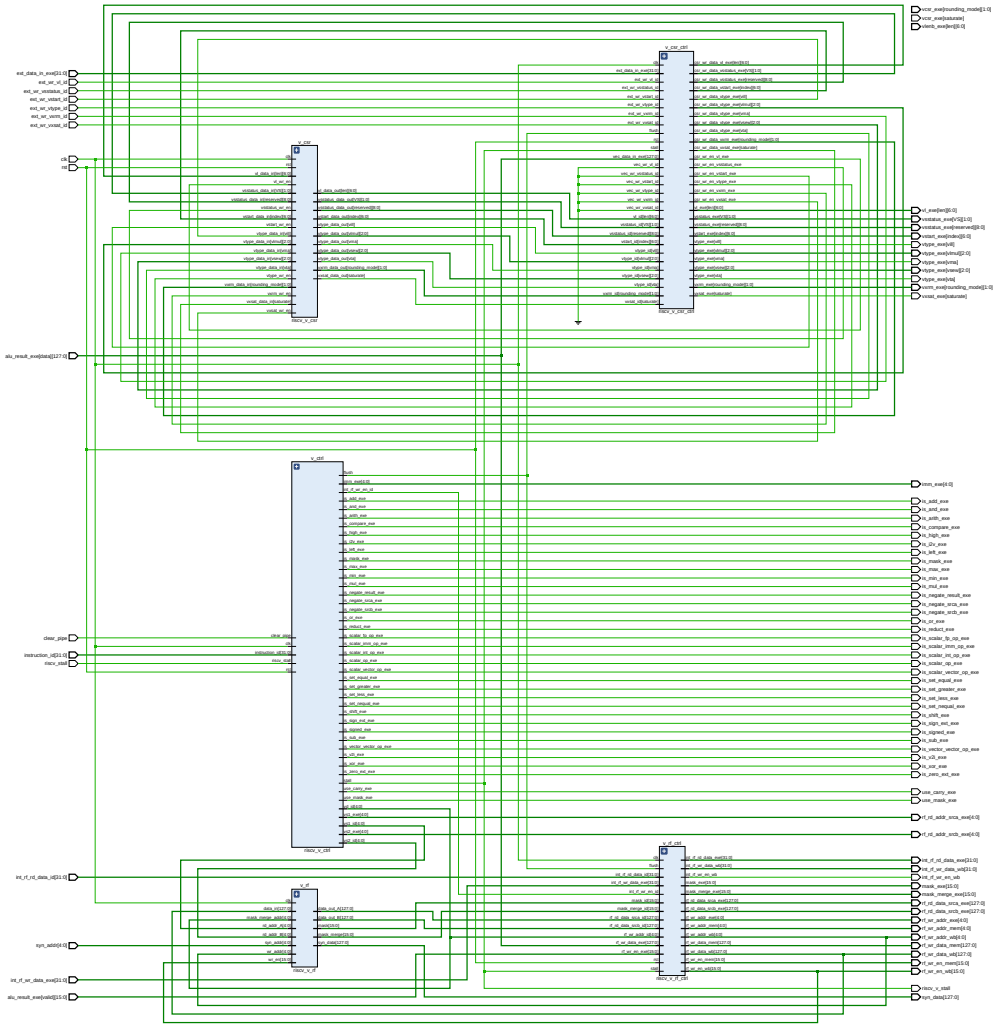


Figure 4.7: RISCV-V Decode RTL Schematic

Table 4.3: RISCV-V Decode synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	9	20	60	15	5
Clock Frequency	111.1MHz	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	0.317	4.75	1.5	6.98	4.06
Number of comb cells / LUTs	5031	13503	20711	25585	35680
Number of seq cells / Slice Registers	4935	4909	4909	4909	4909
Comb Area um^2	N/A	810124(66%)	233753(74%)	40809(64%)	3461(69%)
Seq Area um^2	N/A	418417(34%)	53542(26%)	22634(36%)	1521(31%)
Comb+Seq Area um^2	N/A	1228541	287295	63443	4982
Power (W)	0.644	1.73	0.0119	0.0229	0.00553

4.2.1 Control Unit

The Figure 4.8 shows the RTL schematic of the *Control Unit*. The instruction is received and decoded during *Instruction Decode* using many comparators to get the control signals, at the right of the diagram the stage flops are instantiated, these flops stage the control signals from the *Instruction Decode* stage to the *Execute* stage to be used by the *Execution Unit*.

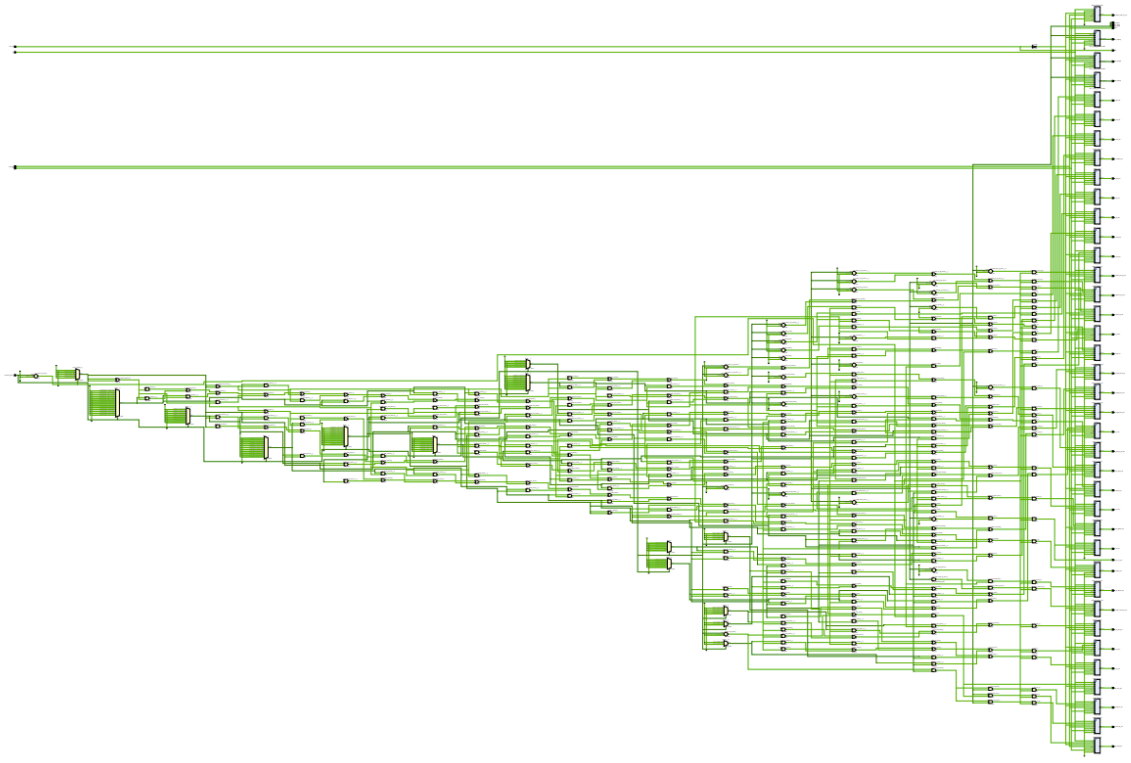


Figure 4.8: RISCV-V Control Unit RTL Schematic

Table 4.4: RISCV-V Control unit synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	14	20	60	15	5
Clock Frequency	71.4MHz	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	0.019	9.72	35.04	8.77	2.840
Number of comb cells / LUTs	69	190	237	300	177
Number of seq cells / Slice Registers	49	44	44	44	44
Comb Area um^2	N/A	7039(62%)	1537(58%)	296(56%)	19(53%)
Seq Area um^2	N/A	4402(38%)	1101(42%)	234(44%)	17(47%)
Comb+Seq Area um^2	N/A	11441	2638	530	39
Power (W)	0.137	0.00663	0.0000363	0.0000346	0.0000331

The synthesis statistics of the *Control Unit* are shown in Table 4.4. Approximately 55% of the area is used by the combinational cells used to decode the instruction and 45% by the flops used to stage the control signals, forty-four flops are instantiated, thirty-four to stage the control signals used by the *Execution Unit*, five to stage the *VS2* field and five to stage the *VS1/IMM* field.

The FPGA implementation adds five extra flops since it is staging the *VS1/IMM* field in separate flops while the ASIC implementation is optimizing that flop and using only five flops since both fields correspond to the same bits in the Vector instruction.

Most of the paths of the decoded control signals, including the critical path, have a depth of seven logic levels, they start at the input instruction and end at the stage register going through the comparators to decode the instruction.

4.2.2 CSR Control

The RTL schematic of the *CSR Control Unit* is shown in Figure 4.9. This block receives the CSR wr_en signals during *Instruction Decode* and stages the signals using the flops showed in the upper half of the diagram, after the flops adds some logic to select the data to be written to the *CSRs* and sends the data to the *CSR* block.

In the lower half of the diagram the *CSRs* are received from the *CSR* block and are staged from the *Instruction Decode* stage to the *Execute* stage to be used by the *Execution Unit*.

The Table 4.5 shows the synthesis statistics of the *CSR Control Unit*. In this unit, the sequential and combinational areas are more balanced, with the sequential area taking more area in some platforms. It instantiates forty-nine flops, twelve to stage the write enables, six for the *CSR* internal interface, six for the *CSR* external interface and thirty-seven to stage the data of the *CSRs*, eleven for *VSTATUS*, nine for *VTYPE*, seven for *VLEN*, seven for *VSTART*, two for *VXRM* and one for *VXSAT*.

The critical path of the *CSR Control Unit* has a logic depth of one, it only instantiates one AO22X1 cell, this cell contains two AND gates connected to one OR gate. The path

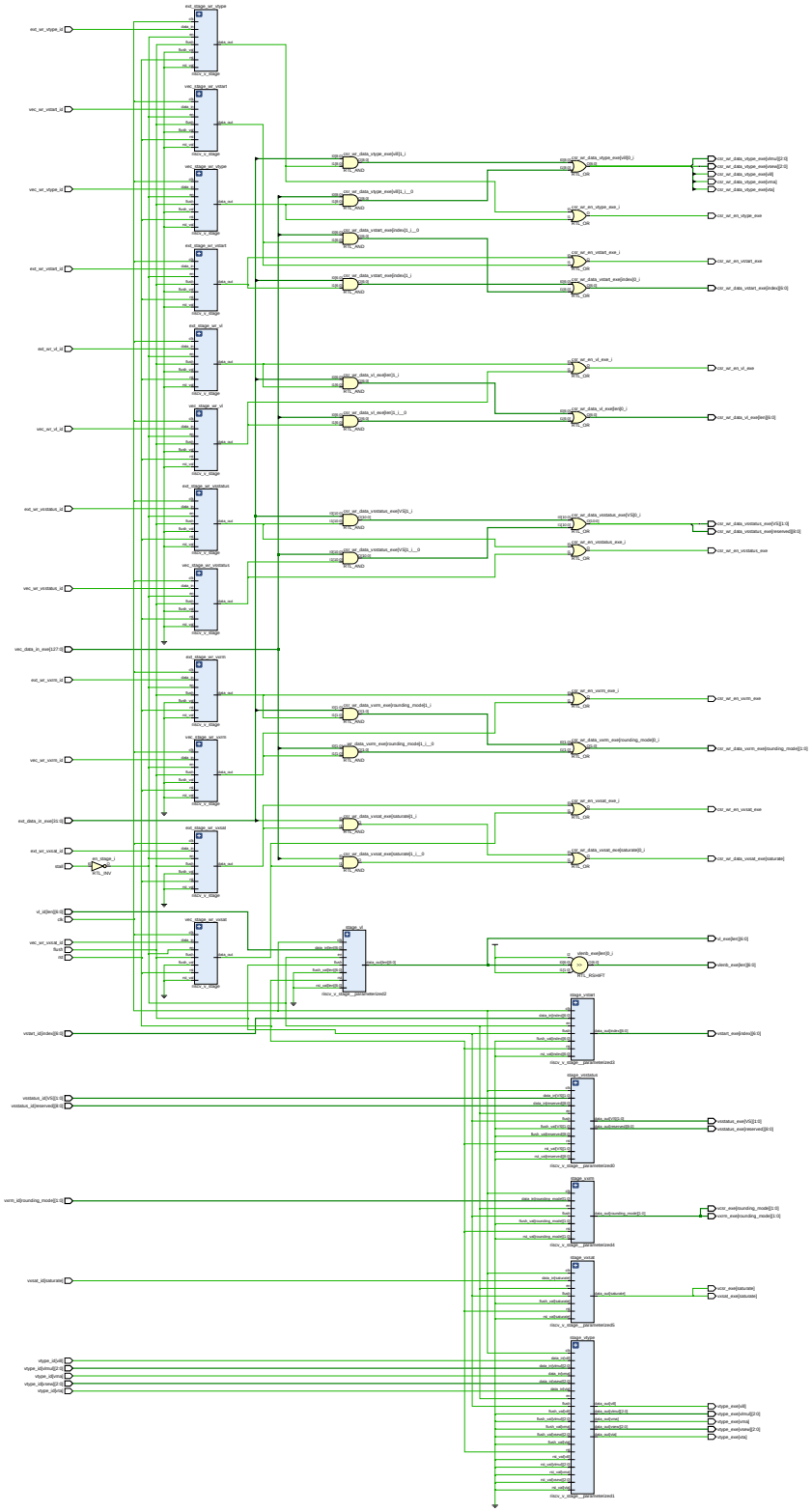


Figure 4.9: RISCV-V CSR Control RTL Schematic

Table 4.5: RISCV-V CSR control synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	17	20	60	15	5
Clock Frequency	58.8MHz	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	0.722	10.26	35.74	6.69	2.947
Number of comb cells / LUTs	71	164	131	175	142
Number of seq cells / Slice Registers	56	49	49	49	49
Comb Area um^2	N/A	5800(55.3%)	1225(50%)	219(46%)	14(42%)
Seq Area um^2	N/A	4702(44.7%)	1226(50%)	260(54%)	19(58%)
Comb+Seq Area um^2	N/A	10502	2451	479	33
Power (W)	0.143	0.0093	0.0000493	0.0000598	0.0000418

starts at the input of the module with the external data to be written to the CSR and ends at the output of the module with the data to be written after qualifying the data and selecting the data between the external and internal data.

4.2.3 CSR

The RTL diagram of the *CSR* block is shown in Figure 4.10. In the diagram, the *CSR* registers are instantiated to the left, in the right part of the diagram some multiplexors are instantiated to implement a small bypass unit to read the correct value if the *CSR* is being written at the same clock cycle.

The *CSRs* synthesis statistics are shown in Table 4.6. In this block the sequential area is bigger than the combinational area, the sequential cells are 70% of the area and the combinational cells the 30% remaining. Thirty-seven flops are instantiated to implement the *CSRs*, eleven for *VSTATUS*, nine for *VTYPE*, seven for *VLEN*, seven for *VSTART*, two for *VXRM* and one for *VXSAT*

The critical path of the *CSR* block has two levels of logic instantiating one NOR2 cell and one AO21 cell. Starts at the write enable of the CSR and ends at the output interface with the data of the CSR going through the small bypass unit, the write enable is used as a selector of the multiplexor used as a bypass unit.

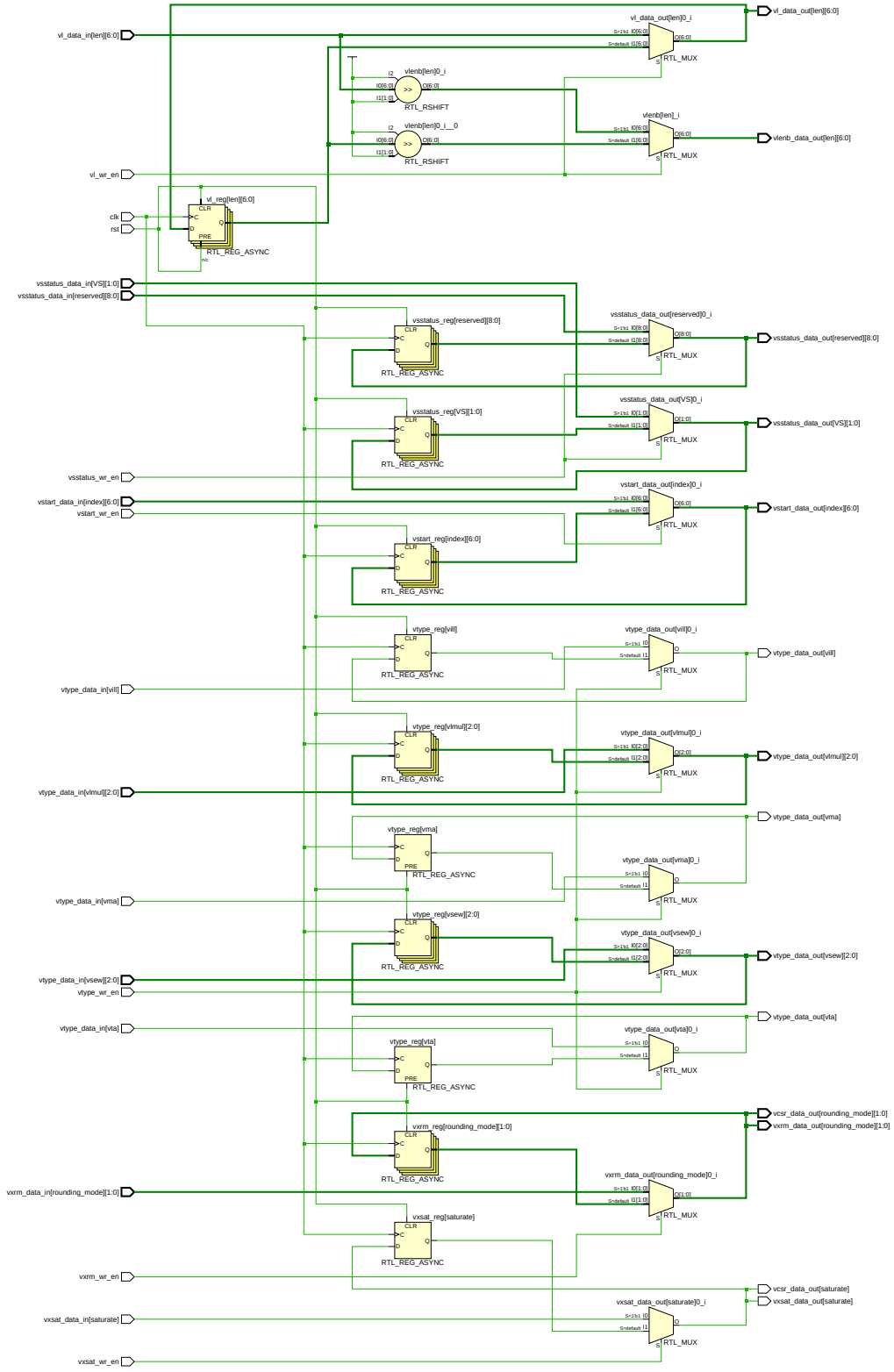


Figure 4.10: RISCV-V CSR RTL Schematic

Table 4.6: RISCV-V CSR synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	15	20	60	15	5
Clock Frequency	66.6MHz	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	0.895	9.55	35.51	6.74	2.942
Number of comb cells / LUTs	21	37	53	53	74
Number of seq cells / Slice Registers	37	37	37	37	37
Comb Area um^2	N/A	3575(69%)	582(38%)	86(30%)	6(30%)
Seq Area um^2	N/A	1600(31%)	929(62%)	196(70%)	14(70%)
Comb+Seq Area um^2	N/A	5175	1511	282	20
Power (W)	0.139	0.0068	0.0000340	0.0000228	0.0000335

4.2.4 Vector Register File Control

The Figure 4.11 represents the RTL diagram of the *VRF Control Unit*. This unit mainly stages signals related to the *VRF* and does not add extra logic.

The Table 4.7 shows the synthesis statistics of the *VRF Control Unit*. The 65% of the area is used to instantiate sequential cells, it instantiates 690 flops to stage all the signals, 15 to stage the *wr_addr* from *Instruction Decode* to *Write Back*, 32 to stage *wr_en* from *Execute* to *Write Back*, 256 to stage the result from *Execute* to *Write Back*, 256 to stage the read data A/B from *Instruction Decode* to *Execute*, 32 to stage the integer data from *Instruction Decode* to *Execute*, 64 to stage the integer result from *Execute* to *Write Back*, 3 to stage the integer *wr_en* from *Instruction Decode* to *Write Back* and 32 to stage the *mask* and *mask_merge* from *Instruction Decode* to *Execute*.

In the RTL diagram could appear that no combinational logic is needed although it is instantiated, the combinational logic is used inside the stage blocks to manage the write enables and the resets of the registers.

The *VRF Control Unit* only instantiates stage registers and does not add extra logic, for that reason all the paths have a logic depth of three, the path is the multiplexors used in the stage registers to implement the enable and reset of the registers. In this case, the synthesizer spread the cells along the core area and placed the inputs on top of the core and the outputs to the right of the core incrementing the propagation delay.

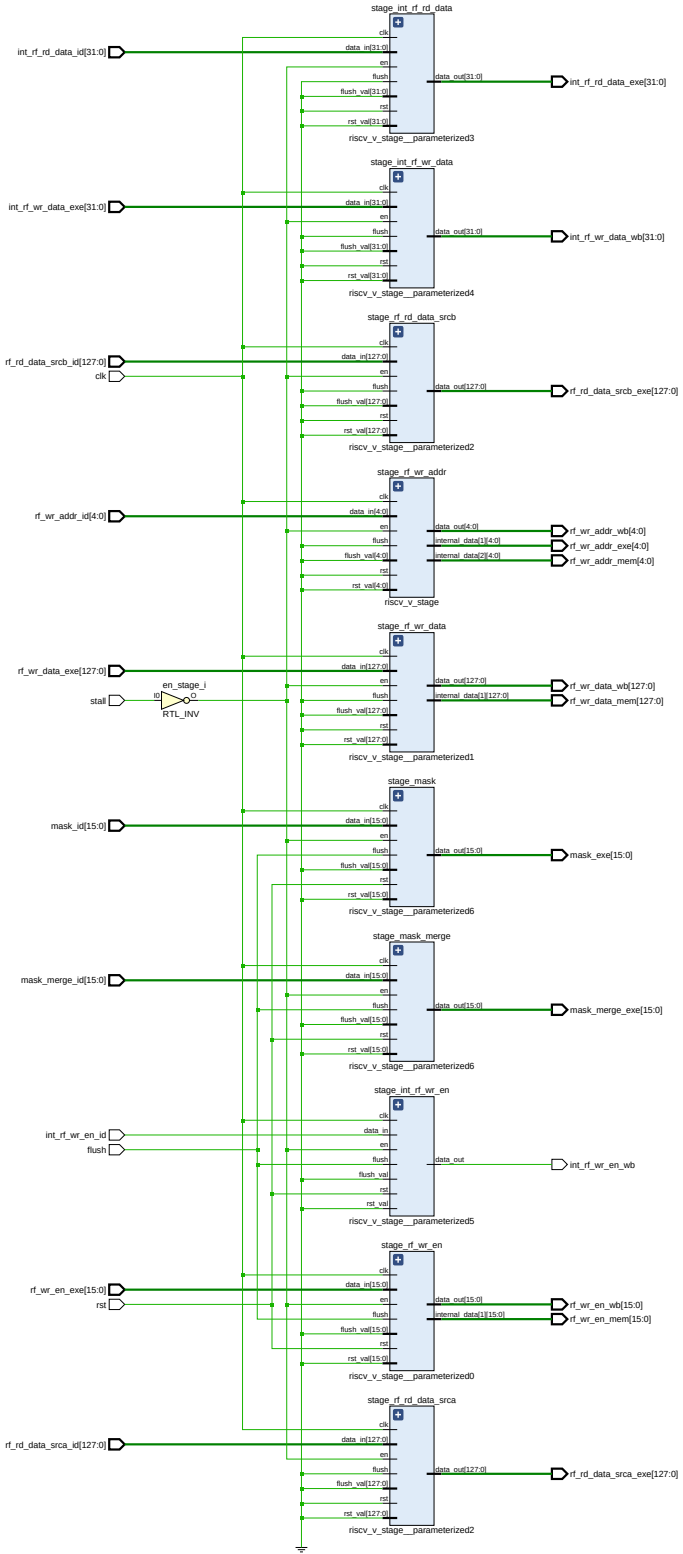


Figure 4.11: RISCV-V Vector Register File Control RTL Schematic

Table 4.7: RISCV-V Vector Register File Control synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	7	20	60	15	5
Clock Frequency	142.8MHz	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	0.251	11.87	44.38	1.2	3.48
Number of comb cells / LUTs	36	726	840	840	1381
Number of seq cells / Slice Registers	690	690	690	690	690
Comb Area um^2	N/A	30146(33.4%)	9103(39%)	1508(32%)	124(36%)
Seq Area um^2	N/A	59942(66.6%)	14148(61%)	3173(68%)	220(64%)
Comb+Seq Area um^2	N/A	90088	23251	4681	344
Power (W)	0.376	0.1	0.000699	0.000912	0.000649

4.2.5 Vector Register File

The RTL diagram of the *VRF* is shown in Figure 4.12. The multiplexors instantiated to the left of the diagram and the ROMs instantiated at the middle are combinational logic used to write the register. The registers are instantiated to the right of the diagram but not all registers are shown in the diagram, the output of the registers is connected to another multiplexor to implement a small bypass unit to read the correct value in case the register that is read is written at the same clock cycle.

The synthesis statistics of the *VRF* are shown in Table 4.8. Most of the area is used by the combinational logic used to write and read the registers, the combinational cells use the 70% of the area while the registers only use the 30% of the area. 4096 flops are instantiated, 32 registers of 128 bits are instantiated getting the 4096 flops.

The critical path of the *VRF* is the path used to read the registers and makes use of the internal bypass unit with a logic depth of six levels, starting at the *rd_addr* and ending at the *rd_data_out*. The synthesizer also spread the cells along the core area and divided the vector registers into two parts, one at the top left corner and the second one at the bottom right corner, this division increments the propagation delay of the circuit.

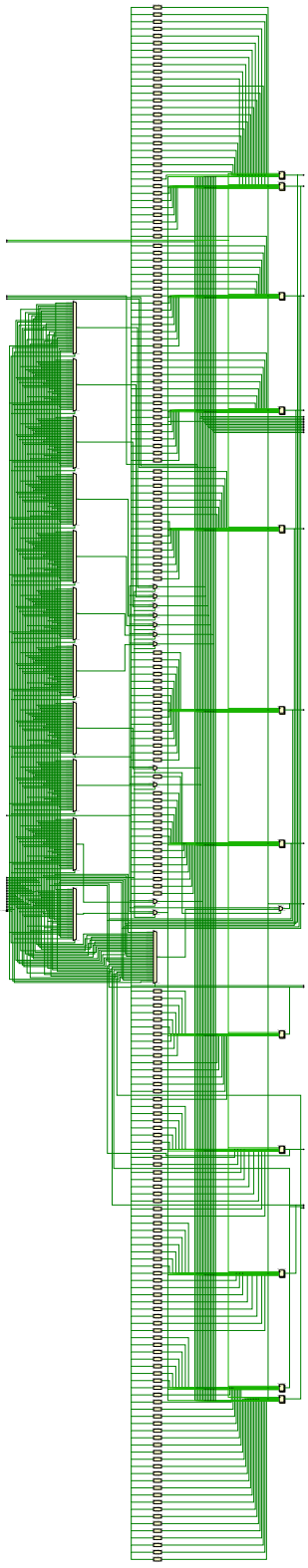


Figure 4.12: RISCV-V Vector Register File RTL Schematic

Table 4.8: RISCV-V Vector Register File synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	10	20	60	30	5
Clock Frequency	100MHz	50MHz	16.6MHz	33.3MHz	200Mhz
WNS(ns)	0.869	3.44	1.2	15.42	1.33
Number of comb cells / LUTs	4895	13554	20034	23475	32031
Number of seq cells / Slice Registers	4096	4096	4096	4096	4096
Comb Area um^2	N/A	776879(69%)	177923(68%)	39524(68%)	3251(72%)
Seq Area um^2	N/A	346858(31%)	81998(32%)	18526(32%)	1255(28%)
Comb+Seq Area um^2	N/A	1123737	259921	58050	4506
Power (W)	0.37	1.32	0.00714	0.00851	0.00621

4.3 RISCV-V Execution Unit

The Figure 4.13 shows the RTL schematic of the *Execution unit*. At the left part of the diagram is instantiated the *Bypass Unit*, at the middle the *Decode Element* block and at the right part the ALUs. Most of the control signals are connected directly from the input interface to the ALUs, the data coming from the Integer/Vector Register Files are connected first to the *Bypass Unit* and then to the *Decode Element* to be connected finally to the ALUs.

The *Execution unit* is the unit that instantiates all the ALUs and has the biggest area and the greatest delay. It's critical path has a logic depth of 198 levels starting at the *VTYPE CSR* that indicates the *OSIZE* of the operation, going through the *Decode Element* unit to decode the *OSIZE* and then through the *Vector Adder*, inside the adder goes through many levels of logic that implement the *Full adders* of the *Carry Ripple Adder* and the logic used to calculate the flags, compare and min/max results, after the adder goes through the *Swizzle Result* block and ends at the ALU result output. The synthesis statistics are shown in Table 4.9.

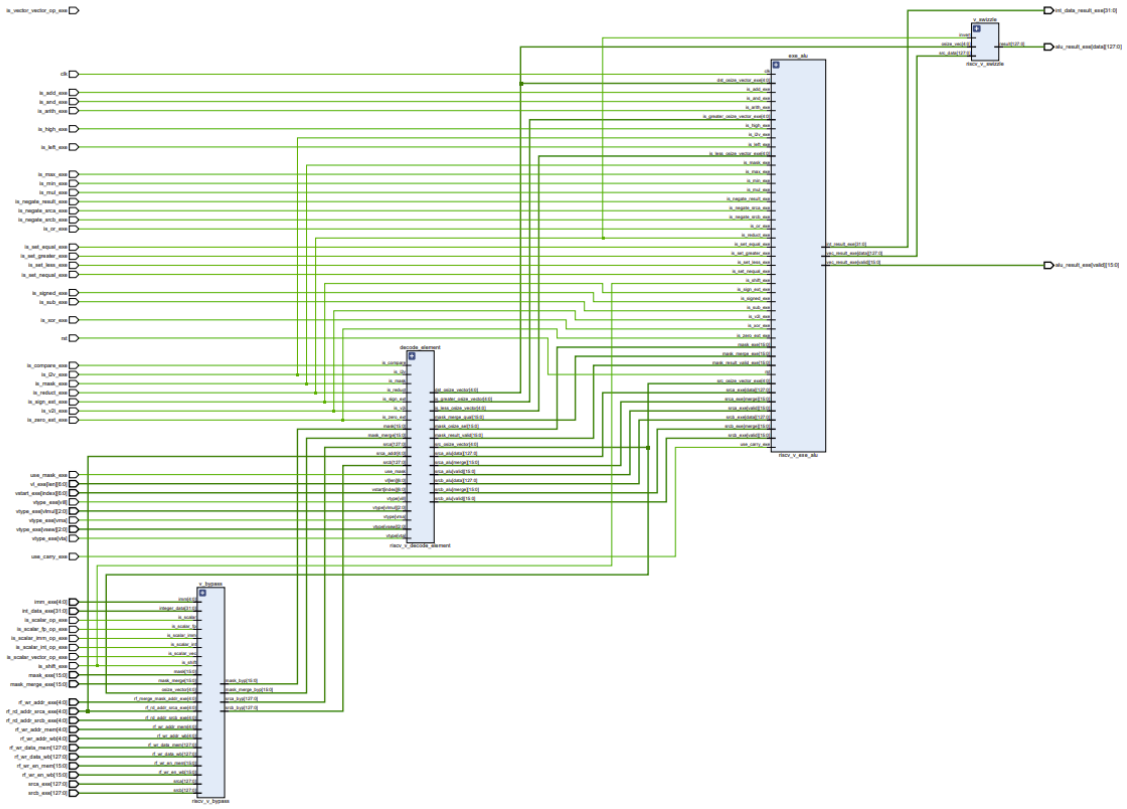


Figure 4.13: RISCV-V Execute Unit RTL Schematic

Table 4.9: RISCV-V Execute Unit synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	100	100	30	5
Clock Frequency	N/A	10MHz	10MHz	33.3MHz	200Mhz
WNS(ns)	N/A	2.3	12.47	3.87	1.7
Number of comb cells / LUTs	44242	171177	160361	184110	65276
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	6434086(100%)	993775(100%)	210676(100%)	9487(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	6434086	993775	210676	9487
Power (W)	0.866	38.8	1.06	0.913	0.861

4.3.1 Decode Element

The *Decode Element* RTL schematic is shown in Figure 4.14. This block only implements combinational logic, it implements some gates and comparators to get the different masks used to get the *Active and Inactive Elements* to get the *wr_en* and the final mask to be used.

The critical path of the *Decode Element* starts with the *VTYPE CSR* input and ends at the valid bit at the output interface going through the different masks used to calculate the valid bit with a logic depth of six levels. The synthesizer decided to set the inputs at the top of the core and the outputs at the right part of the core, spreading the cells and incrementing the propagation delay. The Table 4.10 shows the synthesis statistics of the *Decode Element* block.

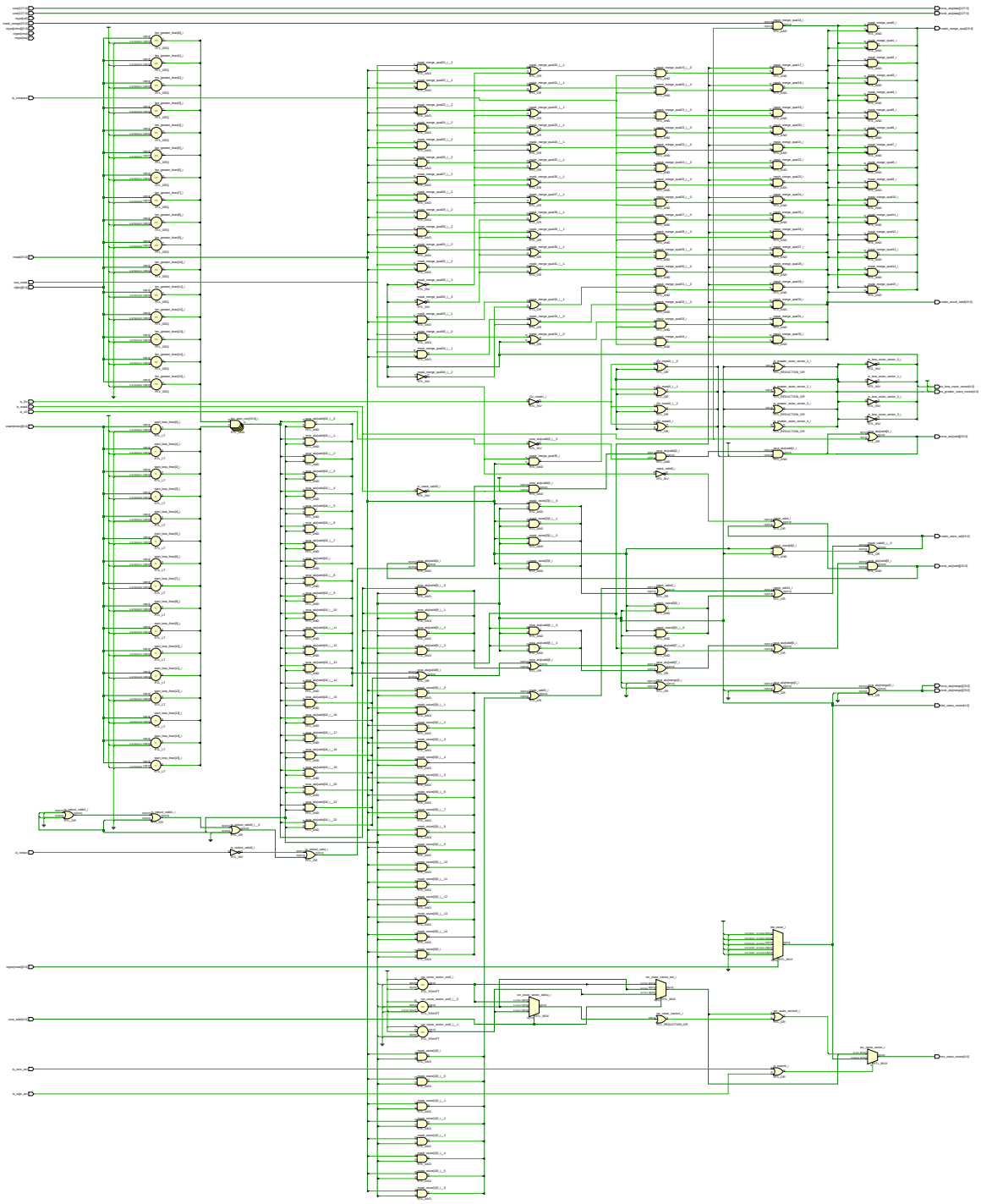


Figure 4.14: RISCV-V Decode Element RTL Schematic

Table 4.10: RISCV-V Decode Element synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	9	24.48	7.81	2.3
Number of comb cells / LUTs	142	299	621	287	281
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	14253(100%)	4090(100%)	606(100%)	32(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	14253	4090	606	32
Power (W)	94.488	0.0068	0.0000799	0.00105	0.000032

4.3.2 Bypass Unit

The RTL schematic of the *Bypass Unit* Unit is shown in Figure 4.15. This block instantiates mainly multiplexors to select the data to be sent to the ALUs depending on the dependencies detected by this unit.

The critical path of the *Bypass Unit* has seven levels of logic, starting at the data coming from the *Write Back* stage and ending at the bypass output data, going through the bypass multiplexers and the logic used to sign extend and replicate the data. As in other modules, the synthesizer placed the inputs at the top of the core and the outputs at the right part of the core increasing the propagation delay. The Table 4.11 shows the synthesis statistics of the bypass unit.

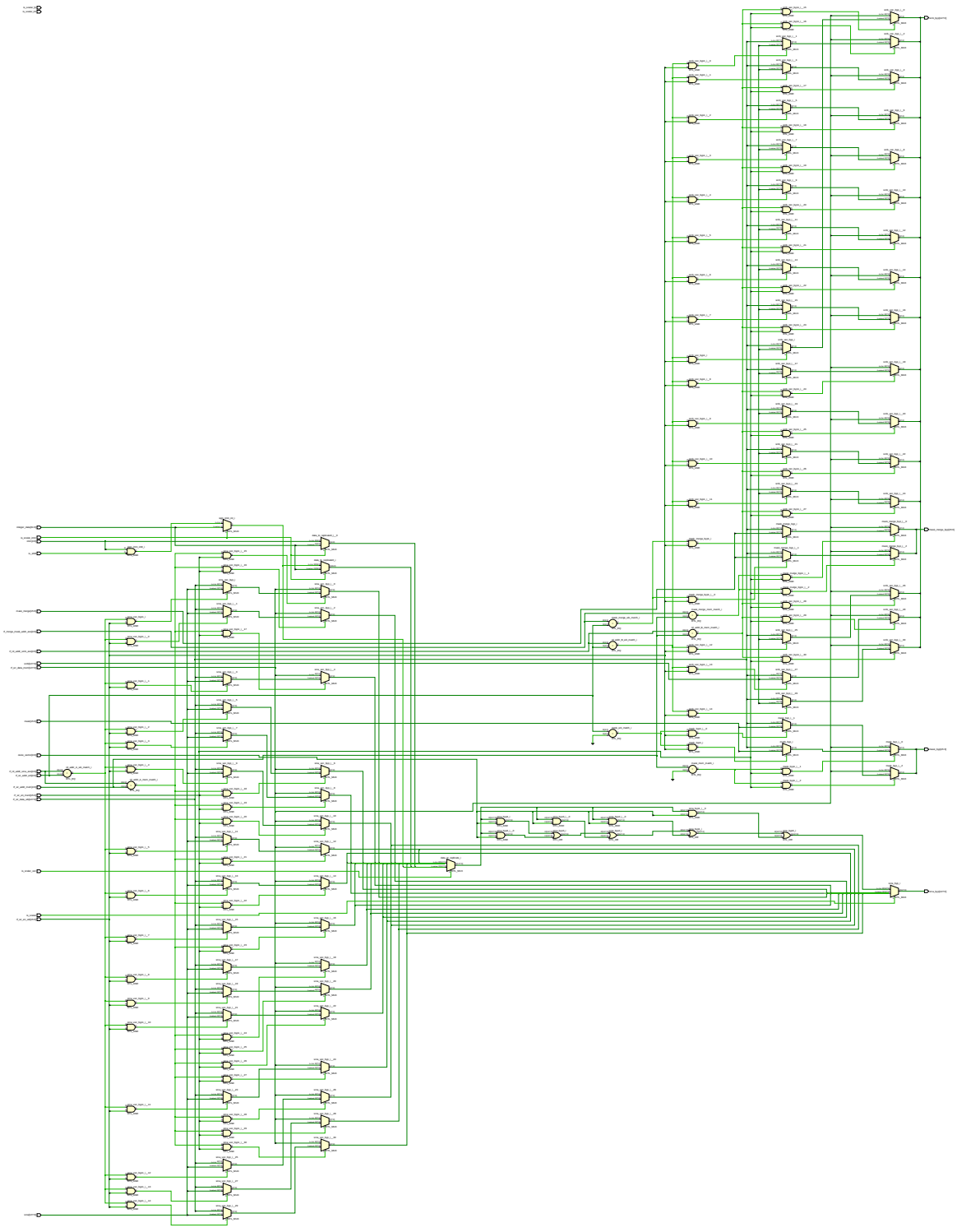


Figure 4.15: RISCV-V Bypass Unit RTL Schematic

Table 4.11: RISCV-V Bypass Unit synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	6.78	23.61	6.52	2.23
Number of comb cells / LUTs	652	1445	1680	1420	1601
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	67020(100%)	10458(100%)	3123(100%)	185(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	67020	10458	3123	185
Power (W)	115	0.0254	0.00223	0.00284	0.000191

4.3.3 Reduction Source

The Figure 4.16 represents the RTL schematic of the *Reduction Source* block. It instantiates some shifters to format the incoming data to the different *OSIZEs*, and some *and*, and *or* gates that behave as multiplexors to select the correct shifter depending on the *OSIZE* and if it is a *Reduction Operation*.

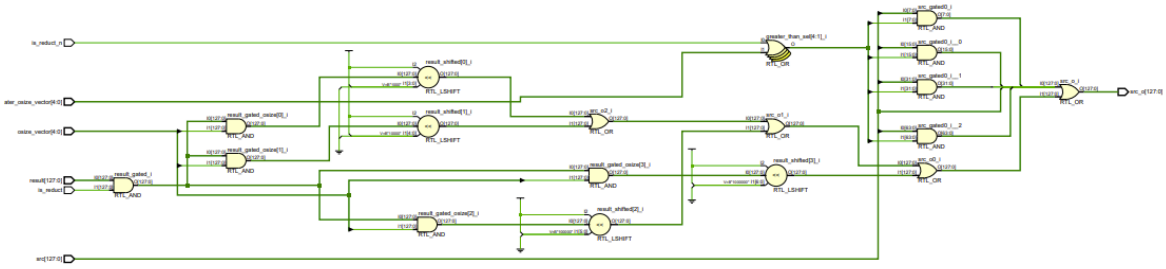


Figure 4.16: RISCV-V Reduction Source RTL Schematic

The critical path of the *Reduction Source* block has three levels of logic, starting at the input data to be shifted and ending at the output data that is sent to the execution block. The critical path goes through three layers of AO21 cells, or three layers of multiplexors. The synthesizer is spreading the cells across the core area as the previous blocks increasing the propagation delay. The Table 4.12 shows the synthesis statistics of the *Reduction Source* block.

Table 4.12: RISCV-V Reduction Source synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	9.26	30.31	7.49	2.56
Number of comb cells / LUTs	282	446	492	414	286
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	18131(100%)	3117(100%)	928(100%)	43(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	18131	3117	928	43
Power (W)	57.7	0.00733	0.0000972	0.00127	0.0000949

4.3.4 Swizzle result

The Figure 4.17 shows the RTL schematic of the *Result Swizzle* block. This block implements five layers of AND and OR gates to invert the order of some bytes of the data depending on the *OSIZE* and one multiplexor at the end to select between the data with the bytes inverted or the original data.

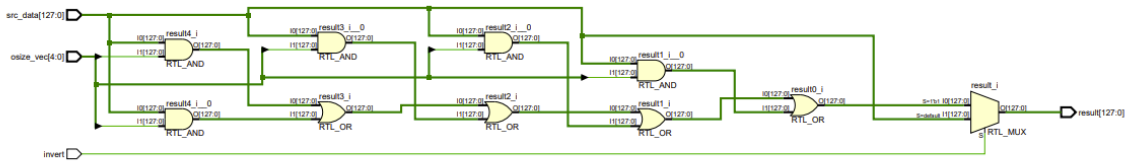


Figure 4.17: RISCV-V Swizzle result RTL Schematic

The logic depth of the *Result Swizzle* is only of three levels, starting at the *OSIZE* control signal and ending at the data formatted with the correct order of the elements, it goes through one inverter, one NAND2x2 cell and one AO21 cell. As in the previous blocks, the synthesizer spreads the cells incrementing the propagation delay. The Table 4.13 shows the synthesis statistics of the *Result Swizzle* block.

Table 4.13: RISCV-V Swizzle Result synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	8.18	32.16	7.29	2.43
Number of comb cells / LUTs	529	513	597	513	385
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	22020(100%)	4168(100%)	775(100%)	53(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	22020	4168	775	53
Power (W)	58.15	0.0825	0.0000245	0.000538	0.0000333

4.3.5 Logic ALU

The RTL schematic of the *Logic ALU* is shown in Figure 4.18. It instantiates the three Bitwise blocks to implement the AND/OR/XOR Bitwise/Mask operations and the shifter to implement the shift operations. The OR and AND gates instantiated at the right bottom corner of the diagram are used to calculate the *result_valid* depending on the masking of the operation.

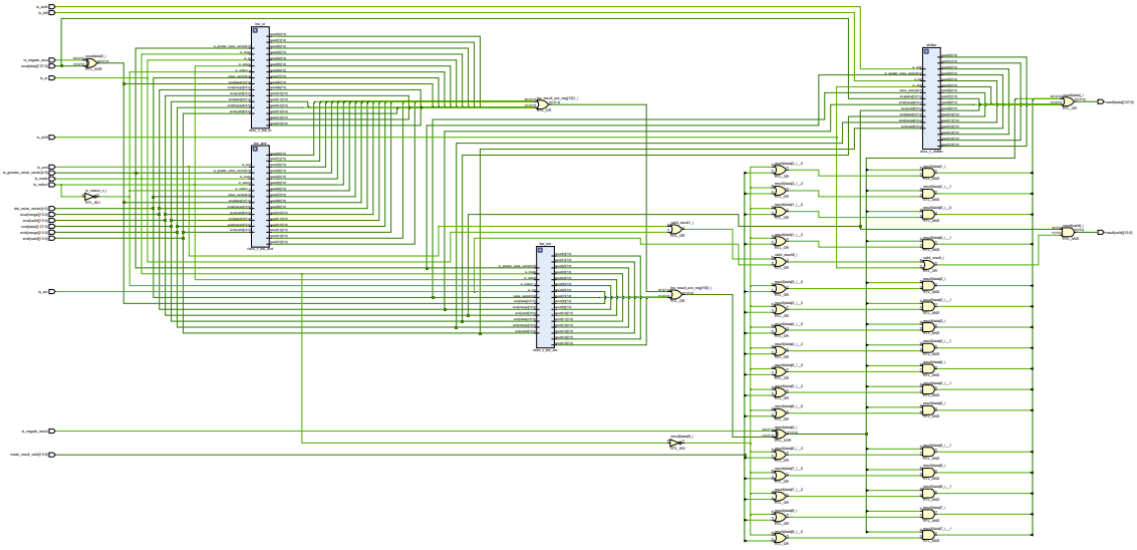


Figure 4.18: RISCV-V Logic ALU RTL Schematic

The critical path of the *Logic ALU* has twenty-two levels of logic starting at the control signal that indicates that it is a *Reduction Operation* and ending at the ALU result. The critical path goes through the *Reduction Source* and the *BW XOR* block. The synthesis statistics of the *Logic ALU* are shown in Table 4.14.

Table 4.14: RISCV-V Logic ALU synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	1.70	15.42	3.1	1.76
Number of comb cells / LUTs	2785	7002	7586	8575	7481
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	305392(100%)	45758(100%)	10376(100%)	847(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	305392	45758	10376	847
Power (W)	123.41	0.129	0.000864	0.000831	0.000639

BW Blocks

The RTL diagrams of the *Bitwise AND*, *Bitwise OR* and *Bitwise XOR* blocks are shown in Figure 4.19, 4.20 and 4.21 respectively. They instantiate some AND gates at the left of the diagram to gate the sources and only enable the blocks if the operation is valid. At the center of the diagram some OR/AND gates and the *Reduction Source* block are instantiated to implement the *Reduction Operations*. Finally, at the right of the diagram, the byte-bitwise blocks are instantiated with the respective AND/OR/XOR gates to implement the operations.

The synthesis statistics of the *Bitwise AND*, *Bitwise OR* and *Bitwise XOR* blocks are shown in Tables 4.15, 4.16 and 4.17 respectively. The Block that requires more area is the *Bitwise XOR* since the XOR gates are bigger than the OR/AND gates while the block that requires less area is the *Bitwise OR* block.

The critical paths of the *Bitwise AND*, *Bitwise OR* and *Bitwise XOR* blocks have thirteen, eleven, and twenty-one levels of logic. The three blocks have similar critical paths starting at the source, going through the *Reduction Source* block and through the respective block of bitwise gates to end at the result.

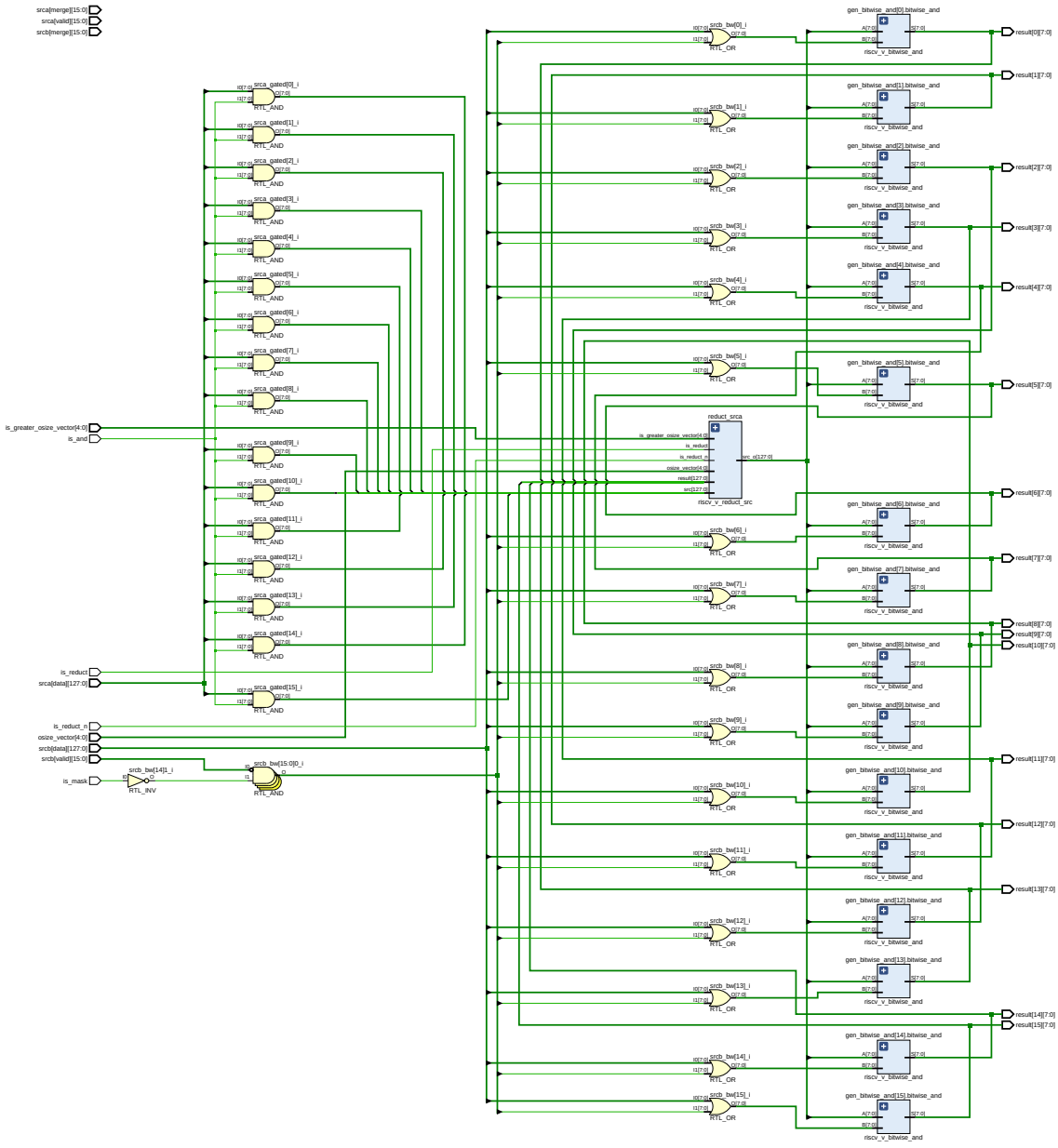


Figure 4.19: RISCV-V BW AND RTL Schematic

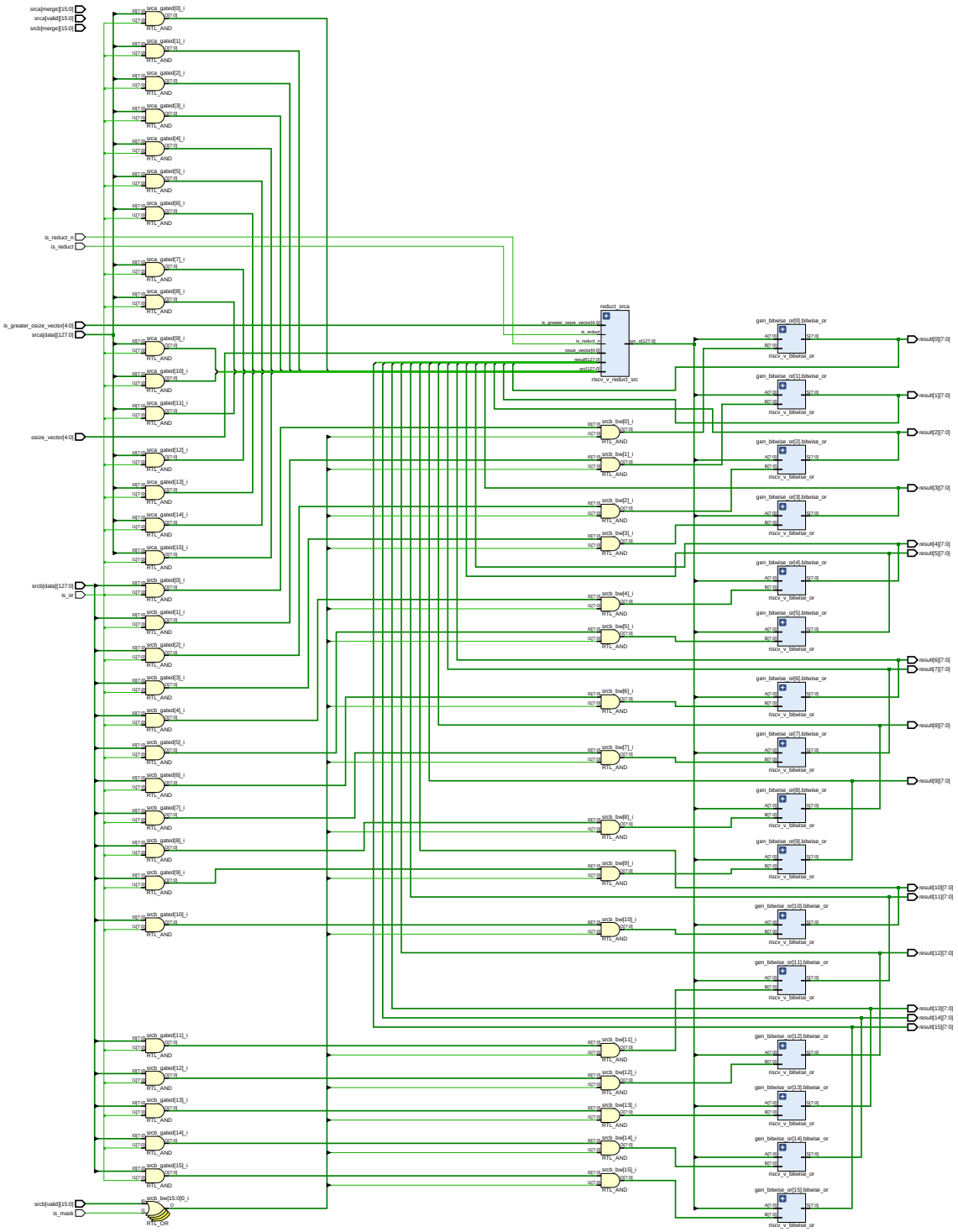


Figure 4.20: RISCV-V BW OR RTL Schematic

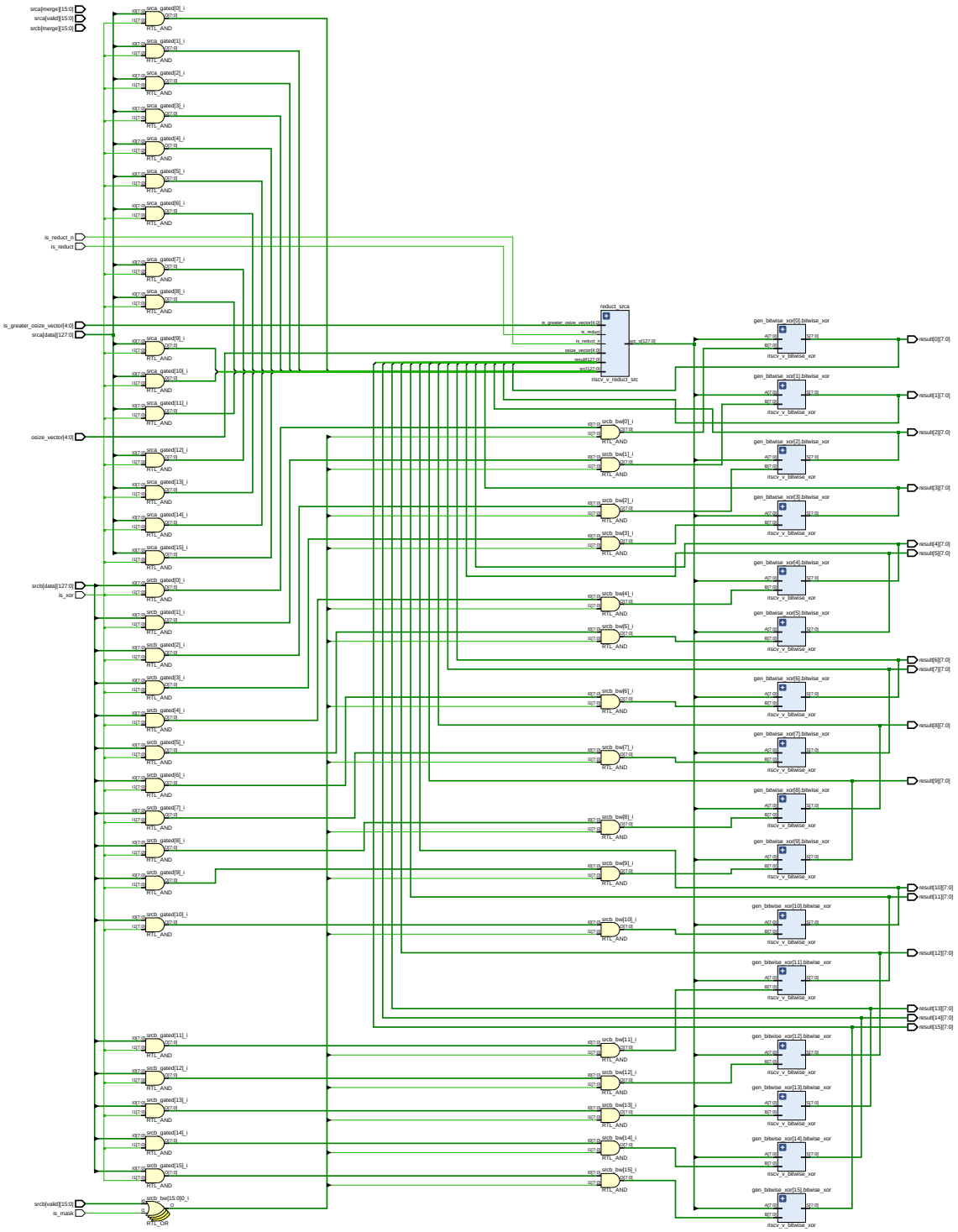


Figure 4.21: RISCV-V BW XOR RTL Schematic

Table 4.15: RISCV-V BW AND synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	5.83	23.42	5.3	2.039
Number of comb cells / LUTs	495	1334	1216	1436	1077
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	59930(100%)	7342(100%)	1644(100%)	121(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	59930	7342	1644	121
Power (W)	45	0.0181	0.000154	0.0000945	0.0000825

Table 4.16: RISCV-V BW OR synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	6.53	22.32	5	2.053
Number of comb cells / LUTs	473	1024	938	1174	819
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	52104(100%)	5873(%)	1360(100%)	97(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	52104	5873	1360	97
Power (W)	63.62	0.0223	0.000139	0.000097	0.000117

Table 4.17: RISCV-V BW XOR synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	3.14	17.56	5.63	1.695
Number of comb cells / LUTs	515	2029	1867	2143	1768
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	87310(100%)	11906(100%)	3696(100%)	225(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	87310	11906	3696	225
Power (W)	86.78	0.0332	0.000204	0.00137	0.000172

Shifter

The RTL diagram of the *Shifter* is shown in Figure 4.22. The RTL shifter is coded in a single module, for that reason all the layers of multiplexors are shown flat in the same diagram along with the logic used to get the different *shift_counts*. At the right side of the diagram the *Byte Shifters* are instantiated to perform the last part of the shift operations.

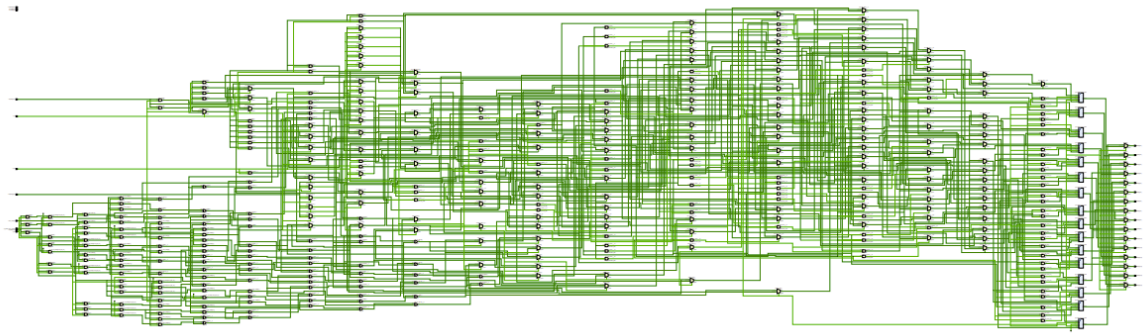


Figure 4.22: RISCV-V Shifter RTL Schematic

The critical path of the *Shifter* has thirteen levels of logic, with most of the cells being AO21 cells that implement the different layers of multiplexors used to shift the data. The critical path starts at the source and ends at the result going through all the layers of the shifter. The synthesis statistics of the *Shifter* are shown in Figure 4.18.

Table 4.18: RISCV-V Shifter synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200Mhz
WNS(ns)	N/A	2.69	23.79	3.4	1.721
Number of comb cells / LUTs	1407	3968	4310	4825	4456
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	188214(100%)	28725(100%)	6193(100%)	511(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	188214	28725	6193	511
Power (W)	81.86	0.0837	0.000639	0.000442	0.000438

4.3.6 Arithmetic ALU

The Figure 4.23 shows the RTL diagram of the *Arithmetic ALU*. As the *Logic ALU* it implements some AND/OR gates to get the *valid_result* and instantiates the *Vector Adder*, *Vector Multiplier* and *Extender* to implement the arithmetic operations.

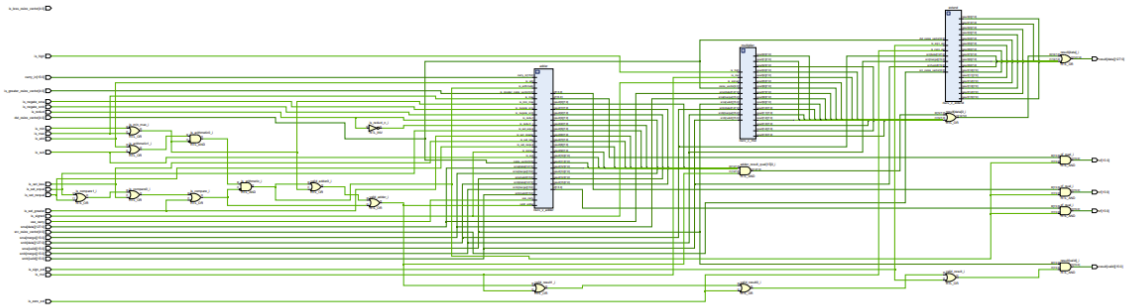


Figure 4.23: RISCV-V Arithmetic ALU RTL Schematic

The critical path of the *Arithmetic ALU* has a depth of 171 levels of logic and corresponds to the *Vector Adder*. The path starts at the source, goes through the Adder's *Reduction Source* block, then to the chain of *Carry Ripple Adders* and through the flag's logic and the logic to calculate the result of the compare and min/max operations. The critical path goes mainly through all the adder, meaning that the area that is not touched by the critical path corresponds to the *Vector Multiplier* that is the block that requires the most area. The synthesis statistics of the *Arithmetic ALU* are shown in Table 4.19.

Table 4.19: RISCV-V Arithmetic ALU synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	100	100	30	5
Clock Frequency	N/A	10MHz	10MHz	33.3MHz	200Mhz
WNS(ns)	N/A	1.2	14.33	6.94	1.2
Number of comb cells / LUTs	40272	144842	148727	172440	55487
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	6073178(100%)	933177(100%)	194186(100%)	8340(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	6073178	933177	194186	8340
Power (W)	2511	207	3.99	2.82	3.52

Adder

The RTL schematic of the *Vector Adder* is shown in Figure 4.24. Almost all the RTL code is coded inside the *Vector Adder* module and just the *Reduction Source* and the byte adders modules are instantiated, they are instantiated at the left of the diagram after the AND gates used to gate the sources. The logic instantiated to the right of the adders is the logic used to calculate the flags, the result of the compare operations and the result of the min/max operations.

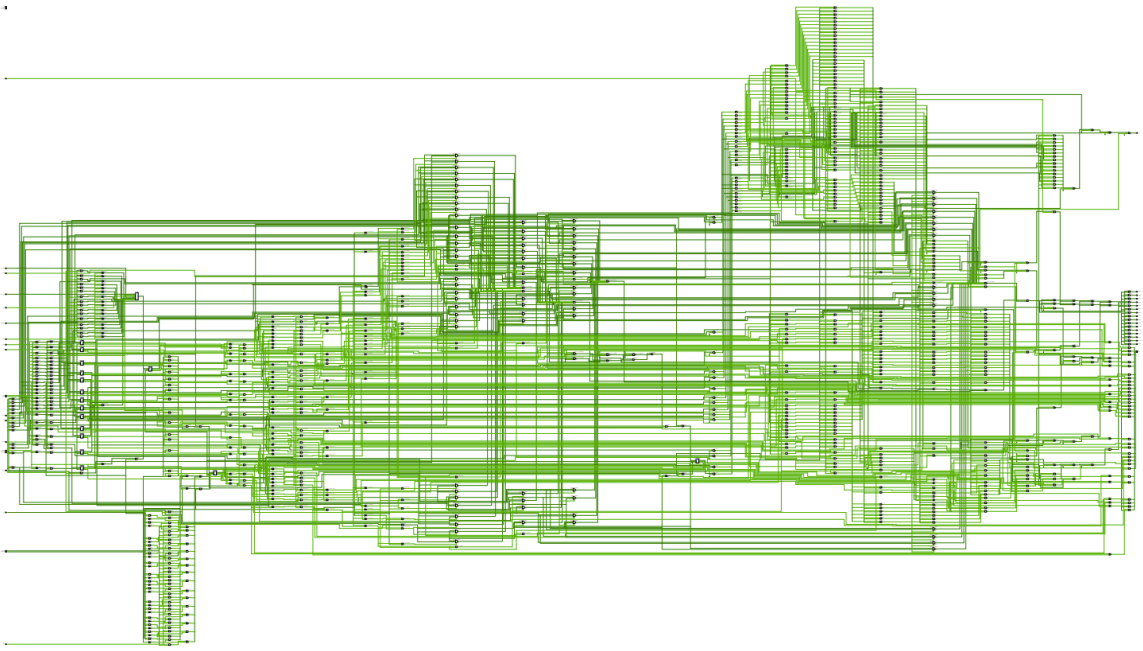


Figure 4.24: RISCV-V Adder RTL Schematic

The *Vector Adder* is the unit with the worst critical path although it is not the unit with

Table 4.20: RISCV-V Adder synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	100	100	30	5
Clock Frequency	N/A	10MHz	10MHz	33.3MHz	200Mhz
WNS(ns)	N/A	0.6	18.81	7.87	0.7
Number of comb cells / LUTs	1753	5936	5964	6974	4057
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	268198(100%)	39107(100%)	8242(100%)	492(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	268198	39107	8242	492
Power (W)	129.41	500	0.000738	0.0113	0.000827

the most area, the unit with the most area is the *Vector Multiplier* but the logic depth of the *Vector Adder* is greater than the *Vector Multiplier*. It has 165 levels of logic starting at the source and going through the two *Reduction Source* units for the implementation of the *Reduction Operations* and the min/max operations, through the inverter for the implementation of the subtracts, through the full chain of *Carry Ripple Adder* including the extra logic added between the byte adders to connect/disconnect the chain of carries, after the adders it still goes through the flags, compare and min/max logic.

Both the *Vector Adder* and the *Vector Multiplier* instantiates one 128-bit *Ripple Carry Adder*, the *Vector Multiplier* even instantiates three 128-bit adders and multiple internal adders with reduced width, but the adders instantiated by the *Vector Multiplier* are not implemented using only *Full Adders*, many bits of the inputs are tied to zero, optimizing the adder and replacing some *Full Adders* with *Half Adders* that are faster. The *Vector Multiplier* also does not implement *Reduction Operations* and does not need *Reduction Source* units, all the carry chain of the adders is connected directly without extra logic in between, and it does not add extra logic to calculate flags or extra operations reducing the levels of logic of the *Vector Multiplier*.

The Table 4.20 shows the synthesis statistics of the *Vector Adder*.

Multiplier

The Figure 4.25 shows the RTL schematic of the *Vector Multiplier*. Going from left to right first it instantiates logic to get the signs of the different elements of the sources and AND gates to gate the sources, then it instantiates two modified *Two's complement* blocks to implement the signed operations, then instantiates one 128-bit unsigned *Vedic Multipler* and collects the five different *OSIZEs* results to send them to a multiplexor to get one result that is connected to another modified *Two's complement* block to get the result with the correct sign, this result is sent to final stage of multiplexors to select between the low and high part of the result.

The critical path of the *Vector Multiplier* has a logic depth of sixty-one levels, having half than the *Vector Adder* but also requiring almost sixteen times more area than the *Vector*

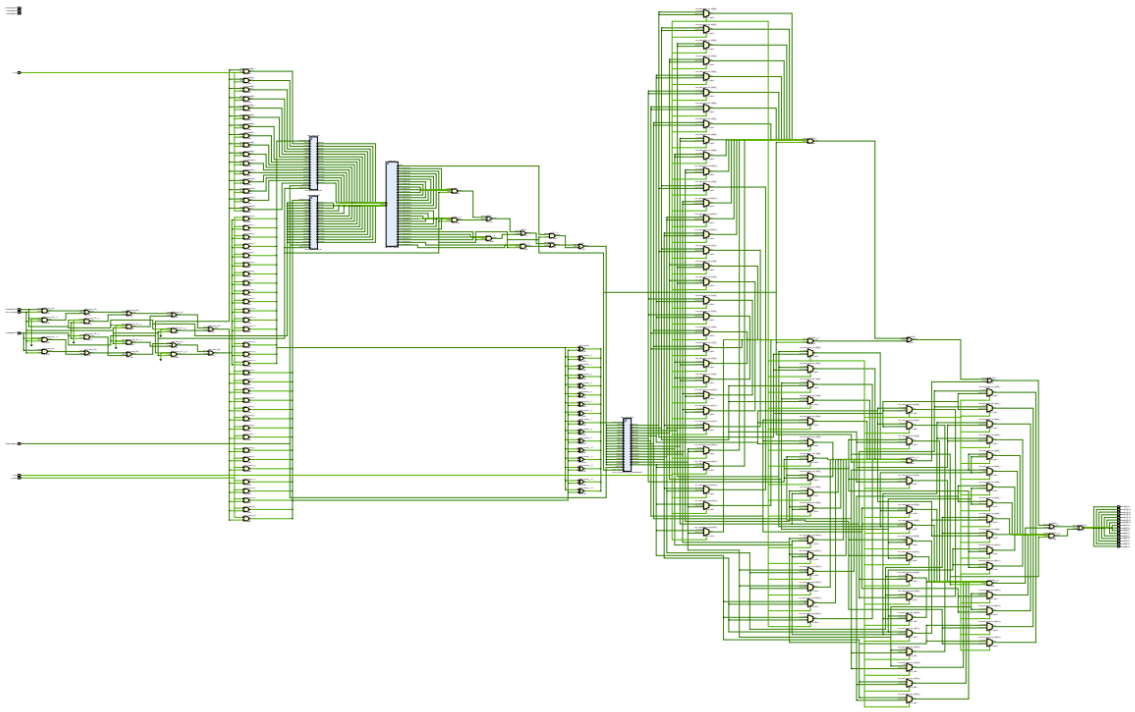


Figure 4.25: RISCV-V Multiplier RTL Schematic

Adder. The critical path starts at the source, going through the *Two's complement* block used to implement signed operations, the 128-bit *Vedic Multiplier*, the second *Two's complement* block and through the multiplexors used to select the *OSIZE* result and the low or high part of the result. The synthesis statistics of the *Vector Multiplier* are shown in Figure 4.21.

Table 4.21: RISCV-V Multiplier synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	100	100	30	5
Clock Frequency	N/A	10MHz	10MHz	33.3MHz	200Mhz
WNS(ns)	N/A	3.27	14.71	5.08	4.3
Number of comb cells / LUTs	38123	155137	153430	186815	50723
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	6760898(100%)	970783(%)	215754(100%)	7773(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	6760898	970783	215754	7773
Power (W)	2537.83	203	2.53	2.97	3.98

Extender

The Figure 4.26 shows the RTL schematic of the *Extender*. This block implements some layers of AND and OR gates to extend the source from the *SRC_SIZE* to the *DST_SIZE*.

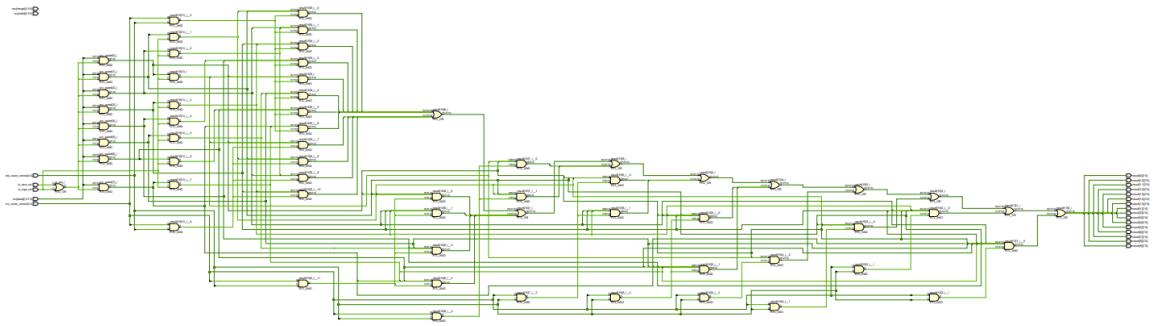


Figure 4.26: RISCV-V Extender RTL Schematic

The critical path of the *Extender* has only three levels of logic going through two AO22 cells and one AO23 cell used as multiplexors to extend the data. The Table 4.22 shows the synthesis statistics of the *Extender*.

Table 4.22: RISCV-V Extender synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	8.81	32.40	7.45	2.569
Number of comb cells / LUTs	155	353	396	465	3355
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	14349(100%)	2593(100%)	516(100%)	39(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	14349	2593	516	39
Power (W)	58	0.00706	0.0000223	0.000198	0.0000332

4.3.7 Permutation ALU

The RTL schematic of the *Permutation ALU* is shown in Figure 4.27. The upper part of the diagram represents the data path needed to implement the *V2I* operations by implementing logic to select the correct portion of the data. The bottom part of the diagram shows the data path of the I2V operations, it just needs AND gates to gate the source and only produce a result if the operation is valid since the operation is implemented using the *Decode Element* and *Bypass* logic.

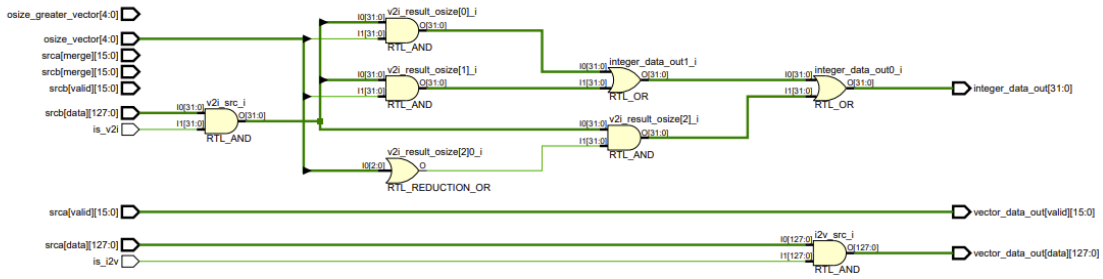


Figure 4.27: RISCV-V Permutation ALU RTL Schematic

The critical path of the *Permutation ALU* has two levels of logic that correspond to the *V2I* data path since the *I2V* data path only requires one AND gate to gate the source. The synthesis statistic of the *Permutation ALU* are shown in Figure 4.23.

Table 4.23: RISCV-V Permutation ALU synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	N/A	20	60	15	5
Clock Frequency	N/A	50MHz	16.6MHz	66.6MHz	200MHz
WNS(ns)	N/A	10.11	34.73	7.71	2.719
Number of comb cells / LUTs	97	179	230	196	172
Number of seq cells / Slice Registers	0	0	0	0	0
Comb Area um^2	N/A	5768(100%)	1380(100%)	411(100%)	18(100%)
Seq Area um^2	N/A	0(0%)	0(0%)	0(0%)	0(0%)
Comb+Seq Area um^2	N/A	5768	1380	411	18
Power (W)	45.76	0.00412	0.0000385	0.000933	0.0000312

4.4 RISCV Vector Engine

The Figure 4.28 shows the RTL schematic of the *RISCV Vector Engine*. It instantiates the two main blocks, the *Decode Unit* and the *Execute Unit*, it also instantiates one *memory ctrl* module that is empty since the memory instructions are not implemented, it is just a placeholder for the *memory ctrl* logic.

The Table 4.24 shows the final synthesis statistics of the *RISCV Vector Engine*. In all the ASIC implementations it instantiates 4863 flops with most of the area being used by the combinational cells, it can go from 90% to 95% depending on the platform. The highest frequency achieved by the ASAP7 platform is 110Mhz with the critical path starting at the *CSR* stage register, going through the *Bypass unit*, the *Arithmetic ALU*, the two *Reduction Source* blocks of the *Vector Adder*, the full chain of *Full Adders*, the flags, compare and min/max logic and the *Swizzle Result* block, all this path has a logic depth of 174 levels of logic. The synthesis statistics are shown in Table 4.24.

To get an idea of how much power the circuit could consume if a frequency of a modern CPU is used a new synthesis run was performed on the ARTIX7 and ASAP7 platforms using a frequency of 2.4Ghz. Using this frequency all the paths were violated increasing the total negative slack and the power consumption increased to 12.2W for the FPGA implementation and 0,171W for the ASIC implementation. The synthesis statistics of the synthesis run using 2.4Ghz are shown in Table 4.25

The Table 4.26 shows a comparison of the area used by each unit, the area reported in the Table corresponds to the area used by the combinational and sequential cells used to implement the RTL logic, ignoring the buffers and fillers. The area is taken from the synthesis runs that use the different units as the top, for that reason the sum of all the areas could not correspond exactly to the area of the *RISCV Vector Engine*.

Analyzing the Table is observed that two blocks consume most of the area, the *VRF* highlighted in green consuming between the 15% and the 31% of the area depending on the platform used and the *Multiplier* highlighted in red consuming between the 54% and the 86%

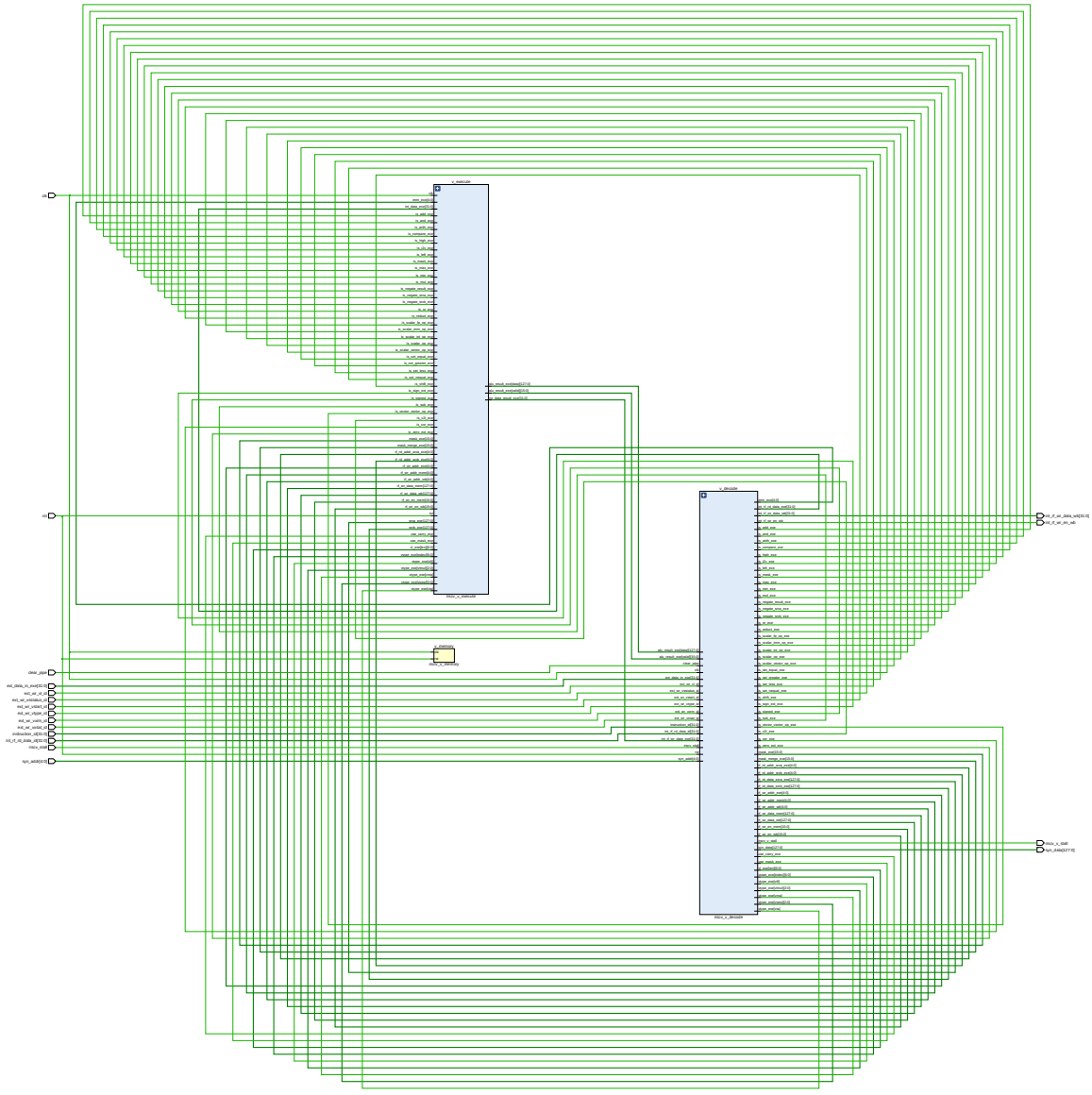


Figure 4.28: RISCV Vector Engine RTL Schematic

Table 4.24: RISCV Vector Engine synthesis statistics

Platform	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
Clock Period(ns)	72	100	100	30	5
Clock Frequency	13.8MHz	10MHz	10MHz	33.3MHz	200Mhz
WNS(ns)	0/0.642	0/36.82	0/3.71	0/4.66	1.1
Number of comb cells / LUTs	51174	166570	174974	212314	100077
Number of seq cells / Slice Registers	4930	4863	4863	4863	4863
Comb Area um^2	N/A	7375196(95%)	1161993(92%)	252390(92%)	12875(90%)
Seq Area um^2	N/A	416002(5%)	98075(8%)	22105(8%)	1503(10%)
Comb+Seq Area um^2	N/A	7791198	1260068	274495	14378
Power (W)	0.201	0.218	0.00295	0.0173	0.0144

Table 4.25: RISCV Vector Engine synthesis statistics 2.4 GHZ

Platform	ARTIX-7	ASAP7
Clock Period(ns)	0.416	0.416
Clock Frequency	2.4GHz	2.4Ghz
TNS/WNS (ns)	-103781/-70.9942	-2639561/-5151
Number of comb cells / LUTs	51174	173052
Number of seq cells / Slice Registers	4930	4863
Comb Area um^2**)**	N/A	20483(93%)
Seq Area um^2**)**	N/A	1504(7%)
Comb+Seq Area um^2**)**	N/A	21987
Power (W)	12.256	0.171

Table 4.26: RISCV Vector Area comparison

Unit	ARTIX-7	GF180	SKY130	NANGATE45	ASAP7
RISCV_V	56104(100%)	7791198(100%)	1260068(100%)	274495(100%)	14378(100%)
Decode	9966(17.7%)	1228541(15.77%)	287295(22.8%)	63443(23.11%)	4982(34.65%)
Control Unit	118(0.21%)	11441(0.15%)	2638(0.21%)	530(0.19%)	39(0.27%)
CSR Control	127(0.23%)	10502(0.13%)	2451(0.19%)	479(0.17%)	33(0.23%)
CSR	58(0.1%)	5175(0.06%)	1511(0.12%)	282(0.1%)	20(0.14%)
VRF Control	696(1.24%)	90088(1.16%)	23251(1.85%)	4681(1.7%)	344(2.39%)
VRF	8991(16%)	1123737(14.42%)	259921(20.62%)	58050(21.15%)	4506(31.34%)
Execute Unit	44242(78.86%)	6434086(82.58%)	993775(78.87%)	210676(76.75%)	9487(65.98%)
Decode Element	142(0.25%)	14253(0.18%)	4090(0.32%)	606(0.22%)	32(0.22%)
Bypass Unit	652(1.16%)	67020(0.86%)	10458(0.83%)	3123(1.14%)	185(1.29%)
Reduction Source	282(0.50%)	18131(0.23%)	3117(0.25%)	928(0.34%)	43(0.3%)
Swizzle Result	529(0.94%)	22020(0.28%)	4168(0.33%)	775(0.28%)	53(0.37%)
Logic ALU	2785(4.96%)	305392(3.92%)	45758(3.63%)	10376(3.78%)	847(5.89%)
BW AND	495(0.88%)	59930(0.77%)	7342(0.58%)	1644(0.6%)	121(0.84%)
BW_OR	473(0.84%)	52104(0.67%)	5873(0.47%)	1360(0.5%)	97(0.67%)
BW_XOR	515(0.92%)	87310(1.12%)	11906(0.94%)	3696(1.35%)	225(1.56%)
Shifter	1407(2.5%)	188214(2.41%)	28725(2.28%)	6193(2.26%)	511(3.55%)
Arithmetic ALU	40272(71.78%)	6073178(77.95%)	933177(74%)	194186(70.74%)	8340(58%)
Adder	1753(3.12%)	268198(3.44%)	39107(3.1%)	8242(3%)	492(3.42%)
Multiplier	38123(67.95%)	6760898(86.77%)	970783(77%)	215754(78.6%)	7773(54%)
Extender	155(0.28%)	14349(0.18%)	2593(0.20%)	516(0.19%)	39(0.27%)
Permutation ALU	97(0.17%)	5768(0.07%)	1380(0.11%)	411(0.15%)	18(0.125%)

of the area. The other two blocks that consume a significant part of the area are the *Shifter* and the *Adder* consuming the 3% of the area each one. The rest of the blocks most of them consume less than 1% of the Area.

The *GF180*, *SKY130* and *NANGATE45* are very similar relating to the proportions of the *VRF* and the *Multiplier*, while in the *ASAP7* implementation the proportion changes incrementing the proportion of the sequential part. The reason for this is the cells used to implement the *multiplier*. In the *ASAP7* implementation the *Full Adders* are implemented using *FA* cells, while in the other platforms, they are implemented using *AND* and *XOR* cells, the *FA* cells implement one *Full Adder* in one cell reducing the cells and the area needed for the *multiplier* and *Adder*.

Figure 4.29 shows the placement of most of the *RISCV Vector Engine* unit. Starting with the *VRF* highlighted in pink and located at the upper half of the circuit, the *VRF* makes use of the 30% of the area approximately, with 30% of that area used by the registers and 70% used by the combinational logic used to read/write the registers. The *VRF Control Unit* is highlighted in purple, this unit contains the registers used to stage the control signals and are located near the *Execution Unit*. The decoder logic is highlighted in orange and is located at the center of the circuit. The *CSRs* are highlighted in yellow and are split in two small blocks at the center of the circuit.

Continuing with the *Execution Unit* at the center of the circuit the *Bypass unit* highlighted in brown is located. Also at the center of the circuit the *BW Blocks* highlighted in lime, the *Shifter* highlighted in red and the *Adder* highlighted in cyan are placed, each one using the 3% of the area approximately. The *Permutation ALU* highlighted in *white* is a small circuit located under the *Shifter*. Finally the *Multiplier* highlighted in blue is located at the bottom part of the circuit, this is the biggest unit and uses the 54% of the area approximately.

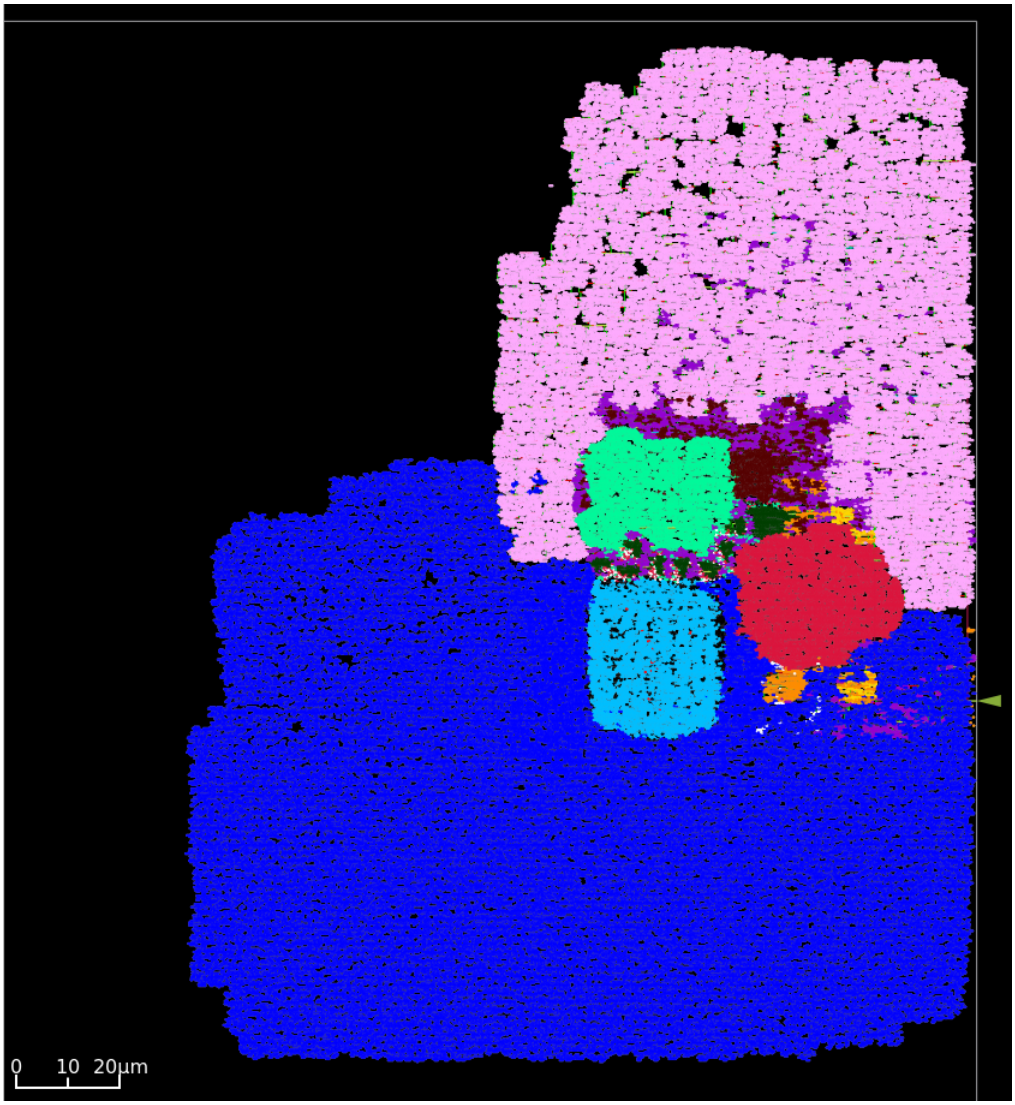


Figure 4.29: RISCV Vector Engine All Units

To identify the units in a better way the Figure 4.30 highlights only the *Decode Unit* in orange and the *Execution Unit* in green. The Figures 4.31, 4.32, 4.33, 4.34 shows the *VRF*, *VRF Control*, *CSR* and *Decoder* units respectively. The Figures 4.35, 4.36, 4.37, 4.39, 4.38, 4.41 and 4.40 corresponds to the *Bypass*, *Shifter*, *BW Blocks*, *Extender*, *Adder*, *Multiplier* and *Permutation ALU* units respectively.

The *Multiplier* shown in Figure 4.41 is also expanded to show the internal 64-bit *Vedic Multipliers*. The *Two's Complement* blocks used in the inputs are highlighted in yellow while

the *Two's Complement* block used at the output is highlighted in green, the other four blocks represented in pink, brown, blue and purple represents the 64-bits multipliers.

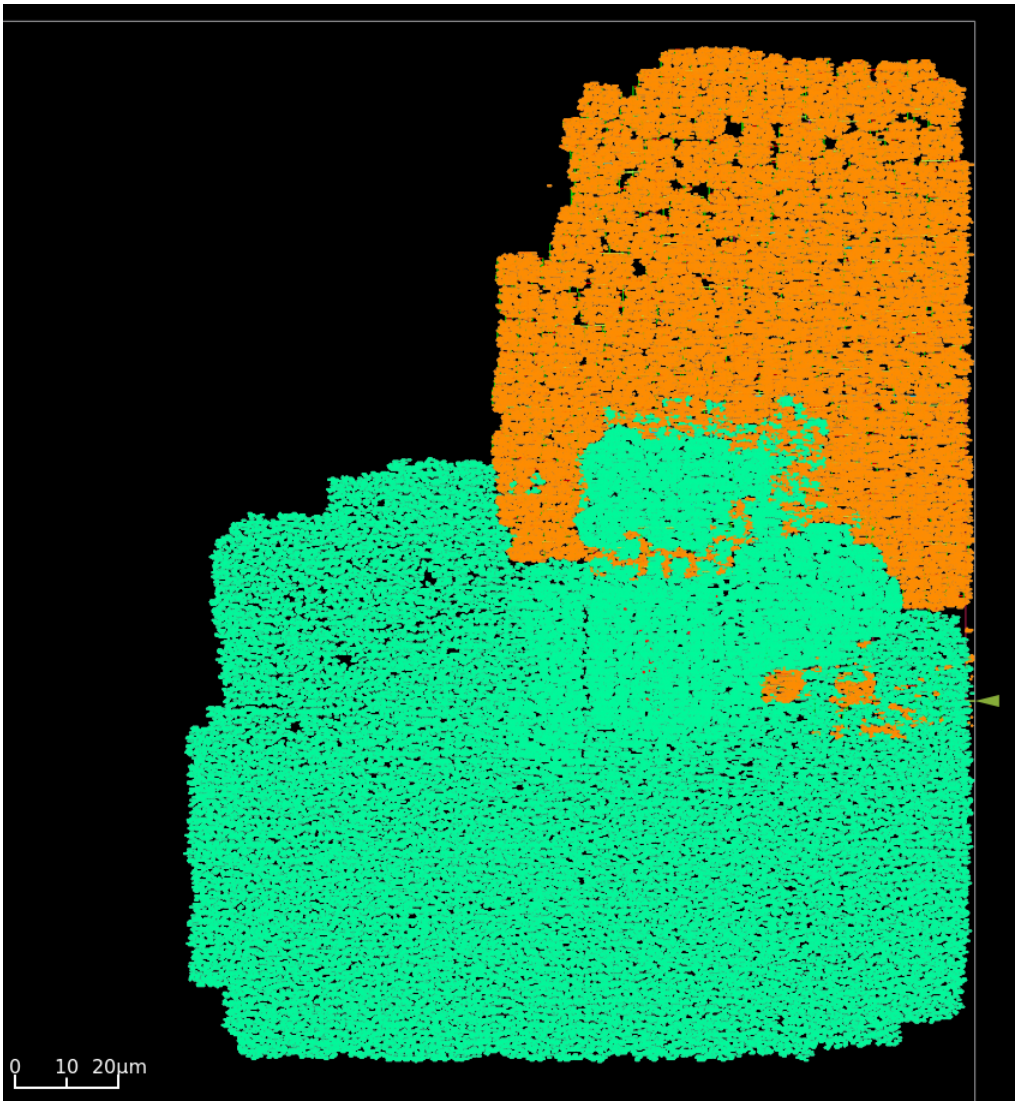


Figure 4.30: RISCV Vector Engine Decode/Execute Units

The Figures 4.42 and 4.43 show the routing of the circuit, with the first Figure showing the routing including the power grid and the second one removing the power grid. Analyzing the Figure is observed that the highest routing congestion occurs at the center of the circuit, specifically in the part where the *Adder* is placed. This can be confirmed by analyzing the heat maps of the routing, 4.44, placement 4.45 and power 4.46. In the routing heat map is observed that the highest density is located over the *Adder*, but with the placement heat map can be observed that the highest density is located under the *Adder*, this area corresponding to the *Multiplier*. Regarding to power consumption, the highest density is located at the top of the circuit, indicating that the registers of the *VRF* are the components that consume the most power of the circuit.

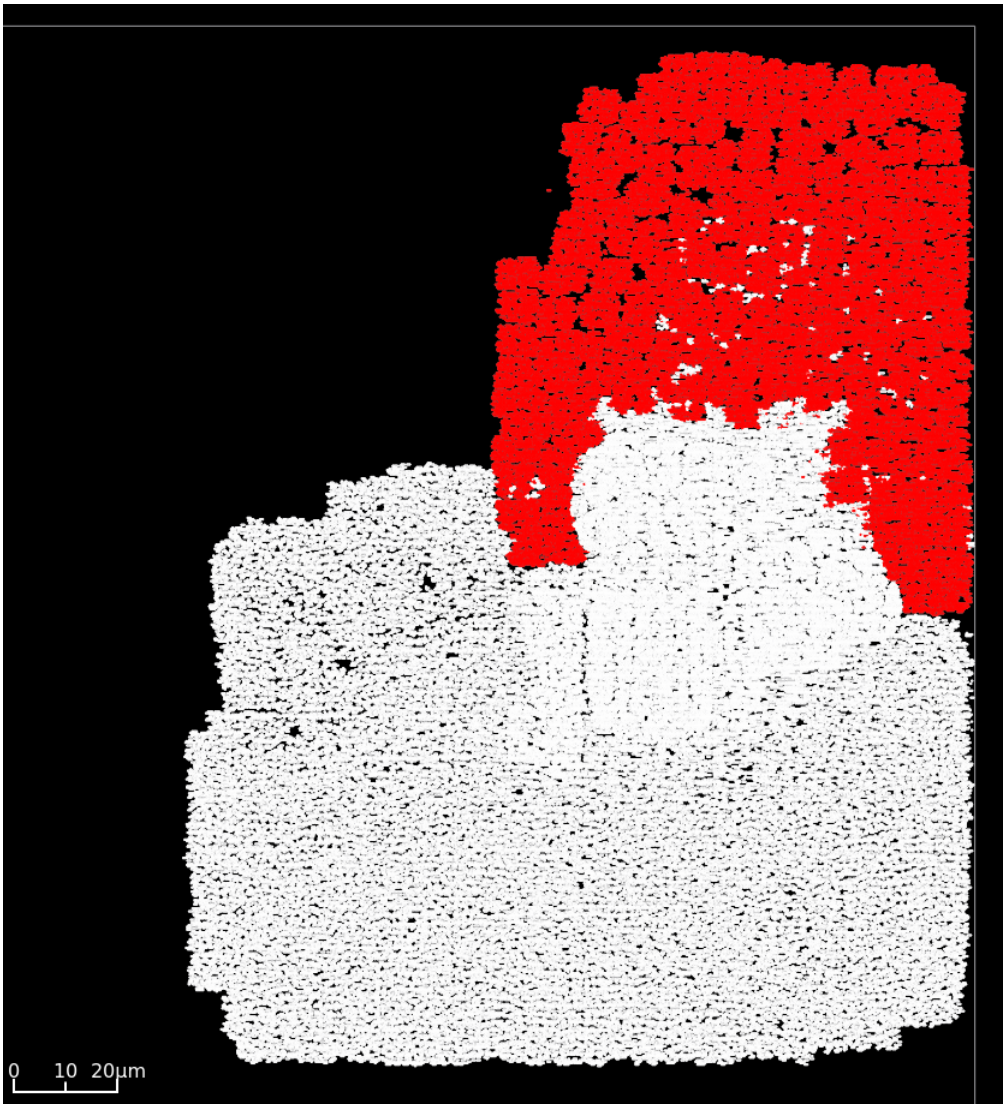


Figure 4.31: RISCV Vector Engine VRF

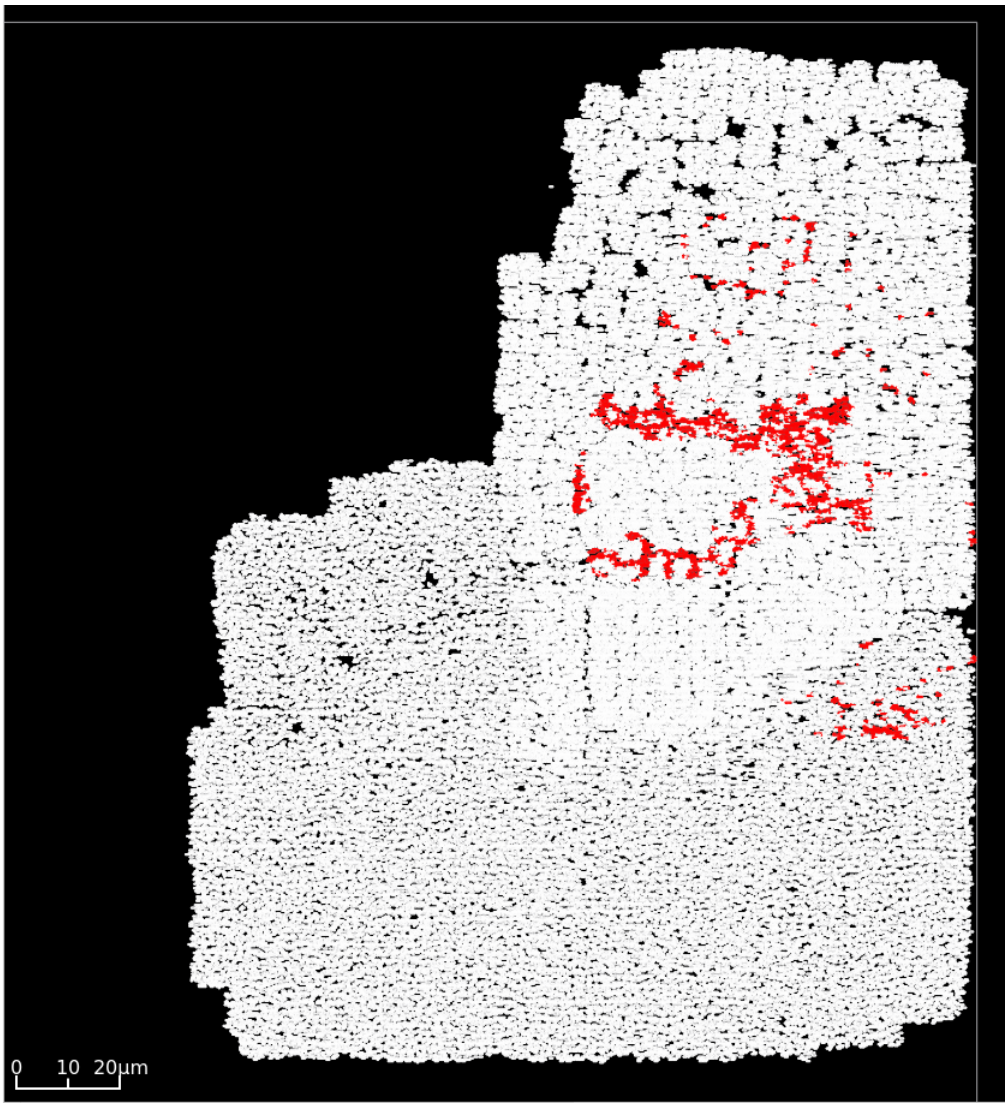


Figure 4.32: RISCV Vector Engine VRF Ctrl

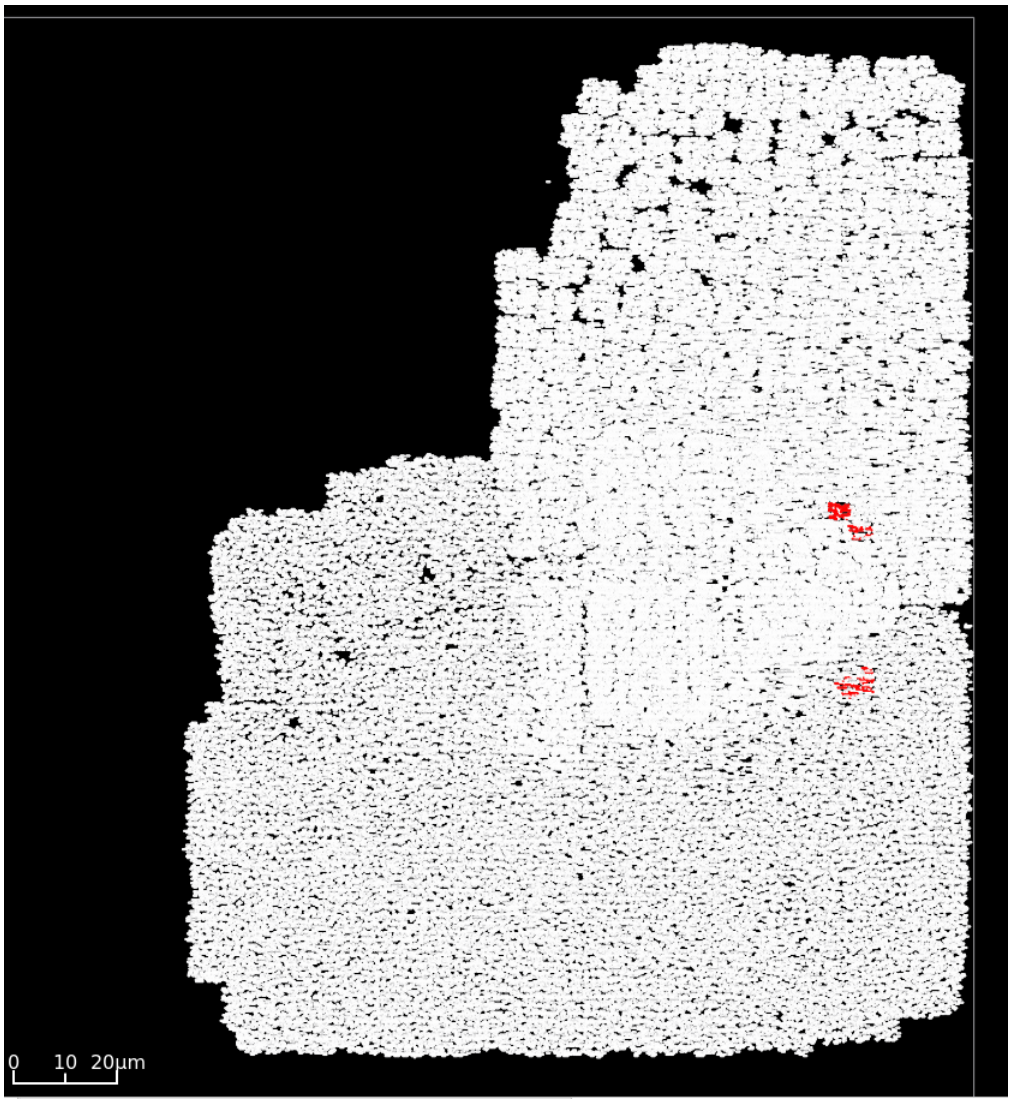


Figure 4.33: RISCV Vector Engine CSR



Figure 4.34: RISCV Vector Engine Decoder



Figure 4.35: RISCV Vector Engine Bypass

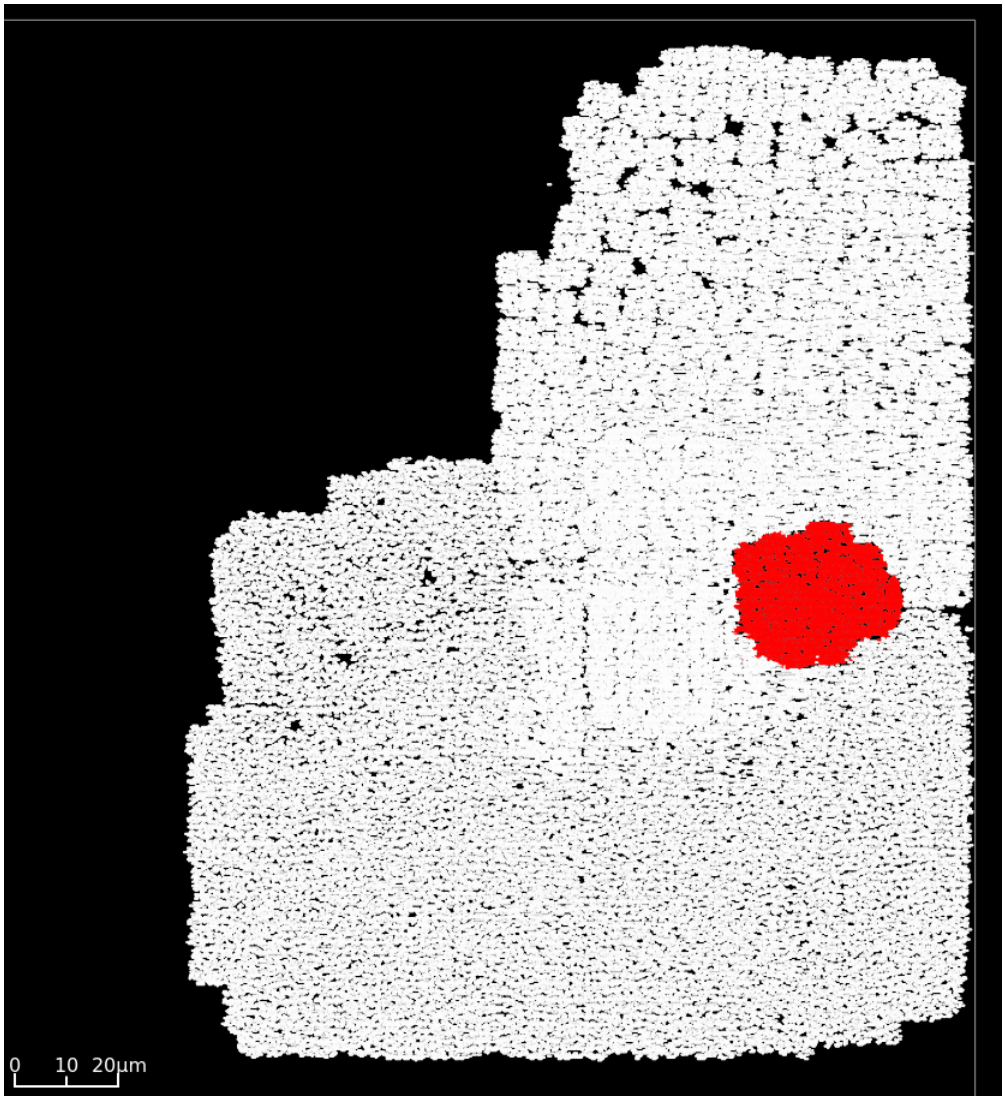


Figure 4.36: RISCV Vector Engine Shifter

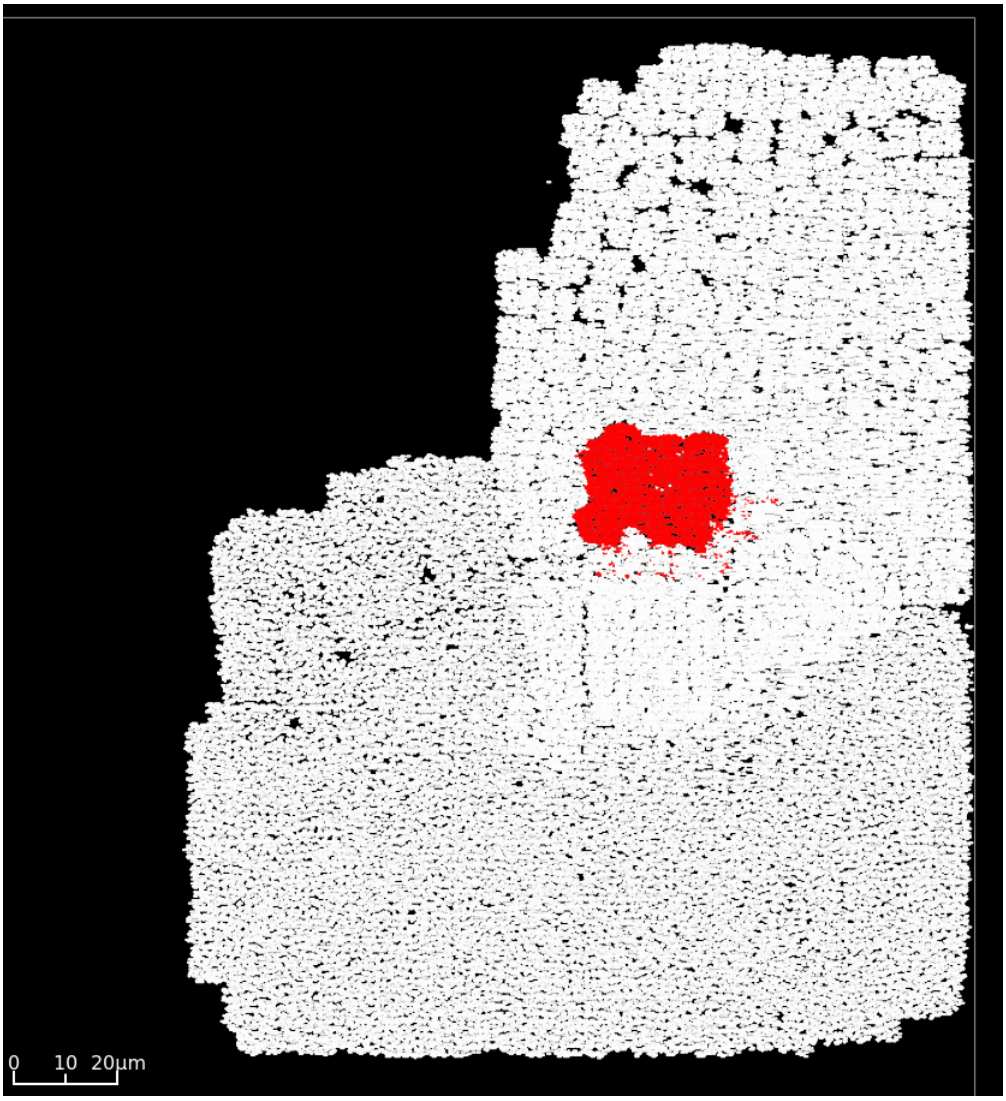


Figure 4.37: RISCV Vector Engine Bitwise blocks

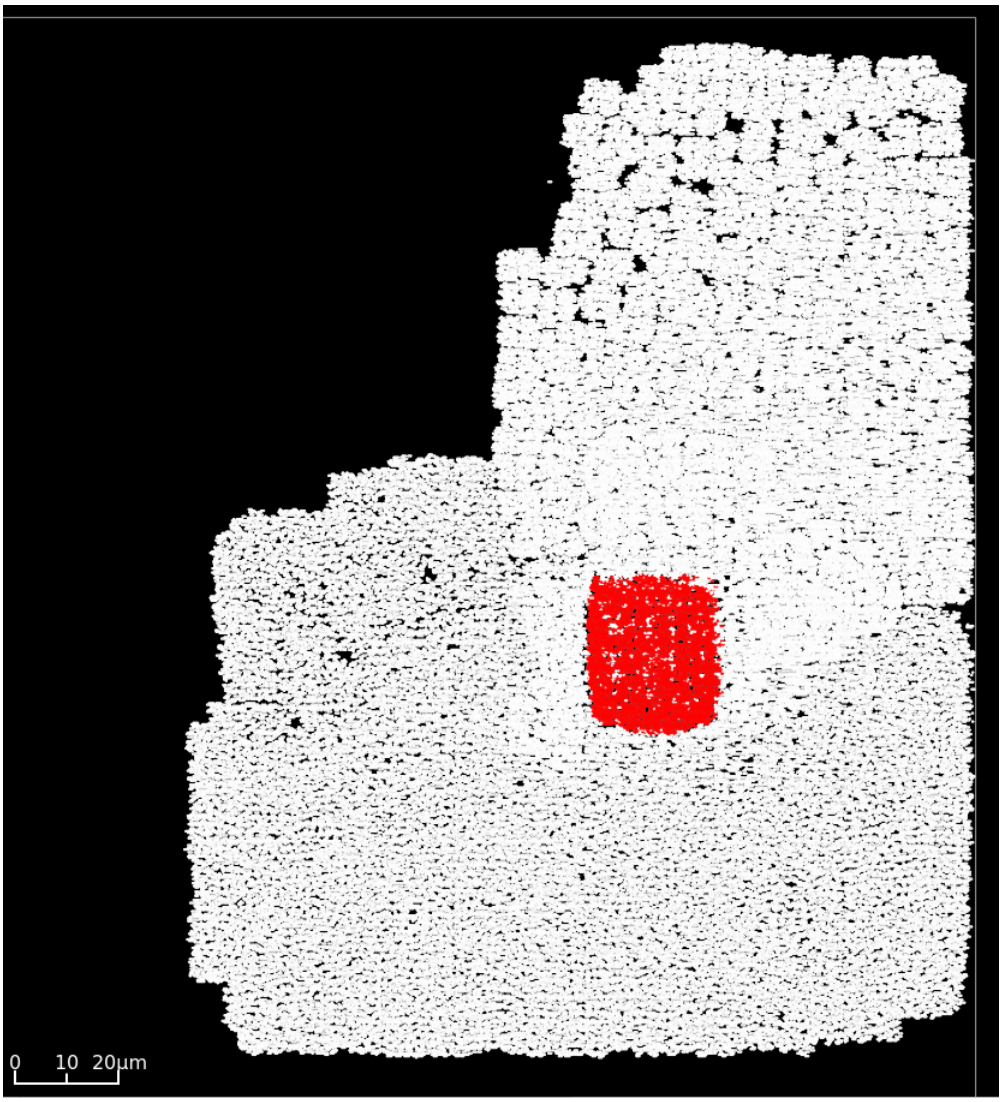


Figure 4.38: RISCV Vector Engine Adder

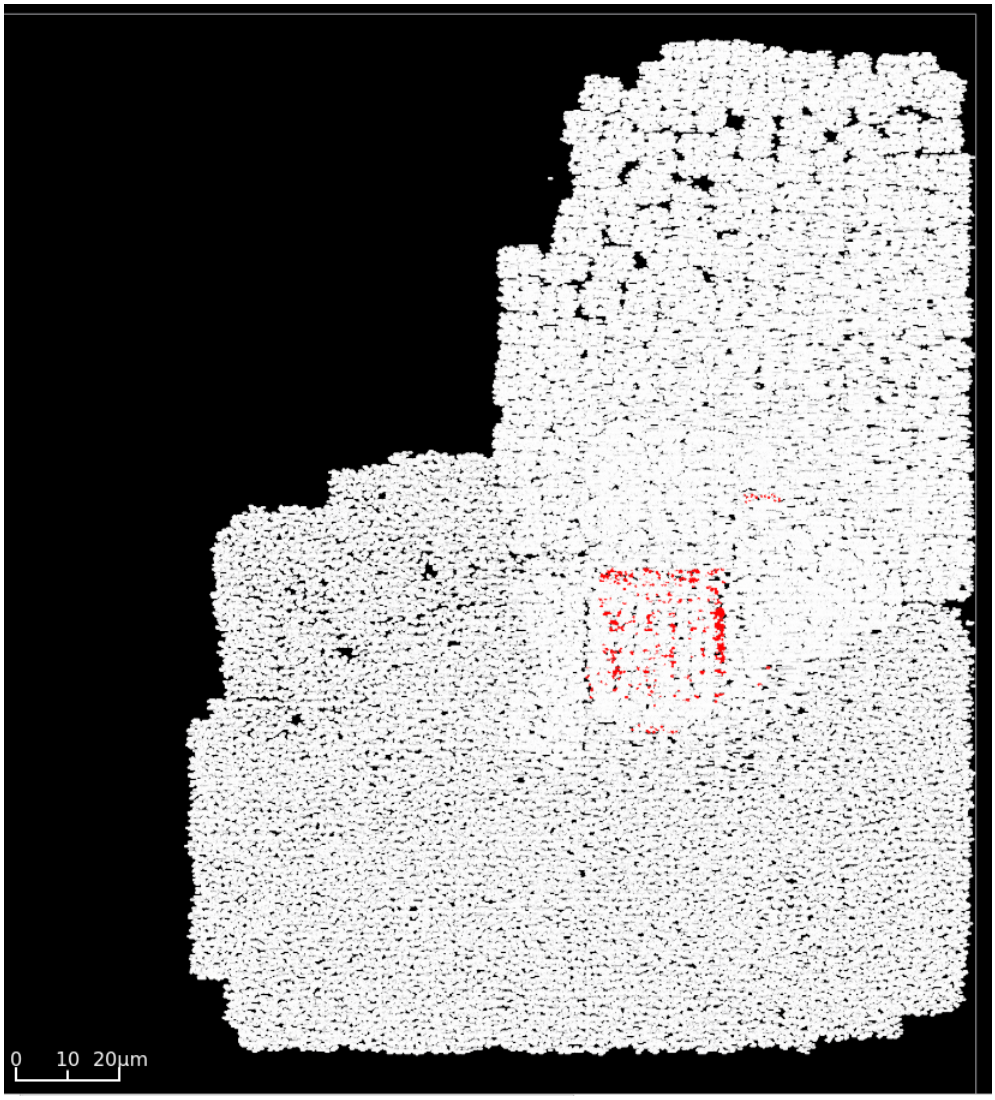


Figure 4.39: RISCV Vector Engine Extender

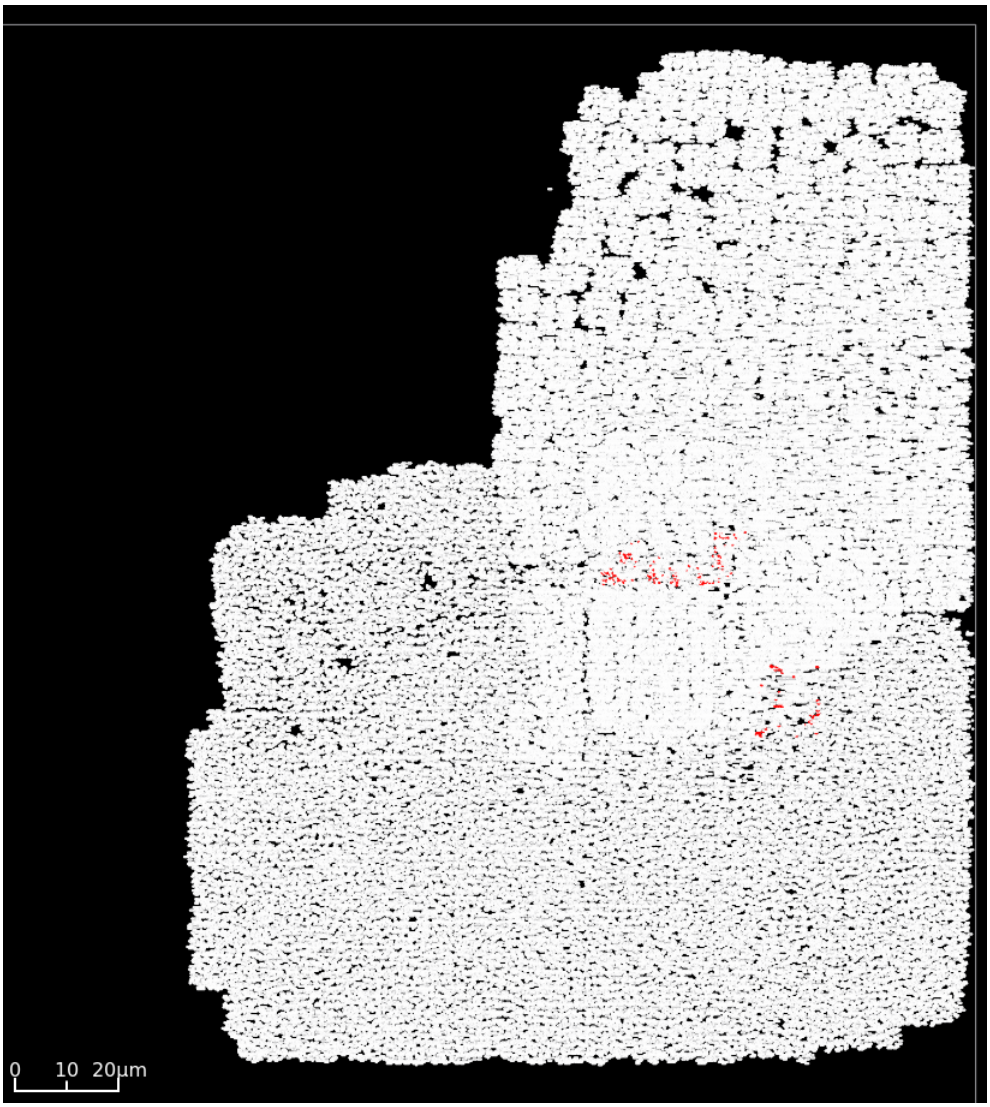


Figure 4.40: RISCV Vector Engine Permutation ALU

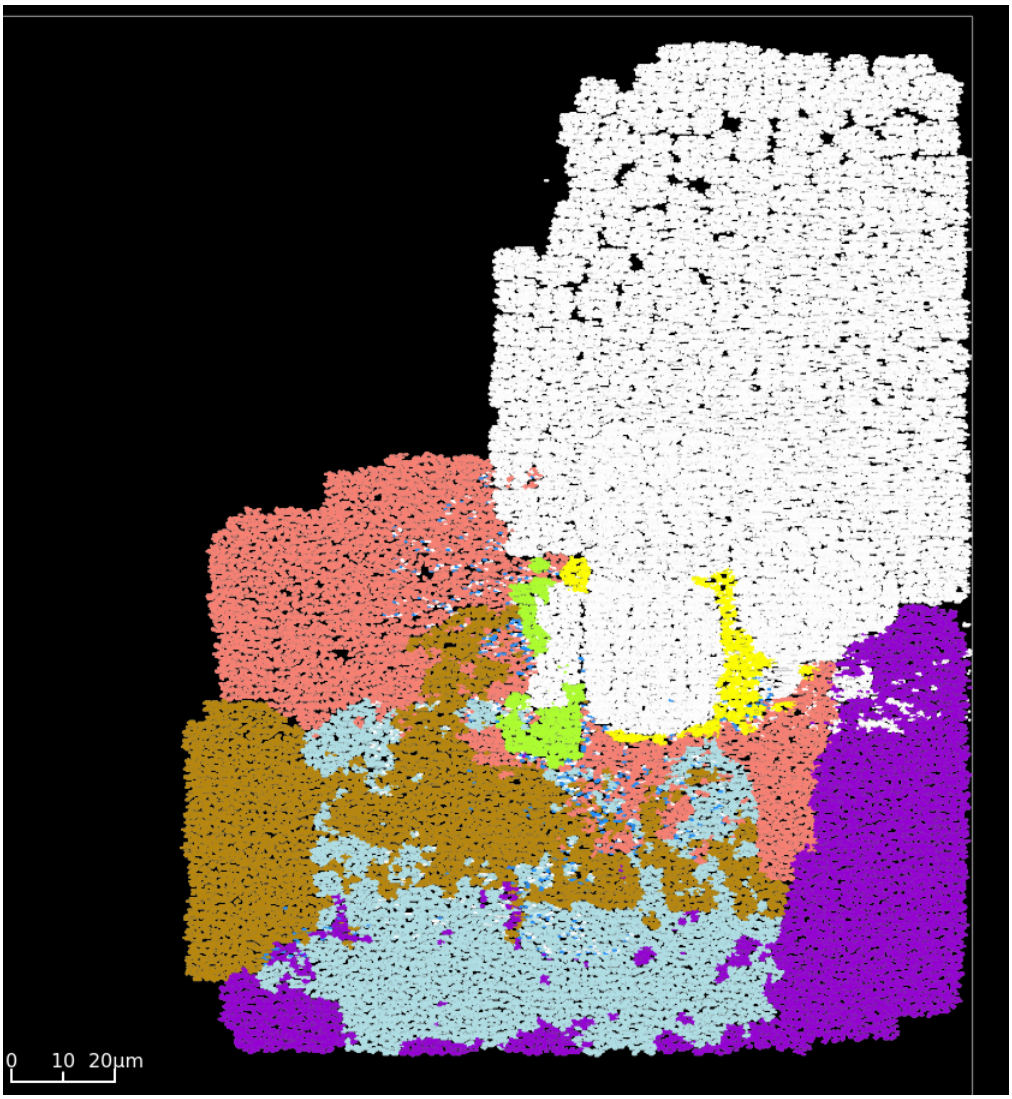


Figure 4.41: RISCV Vector Engine Multiplier

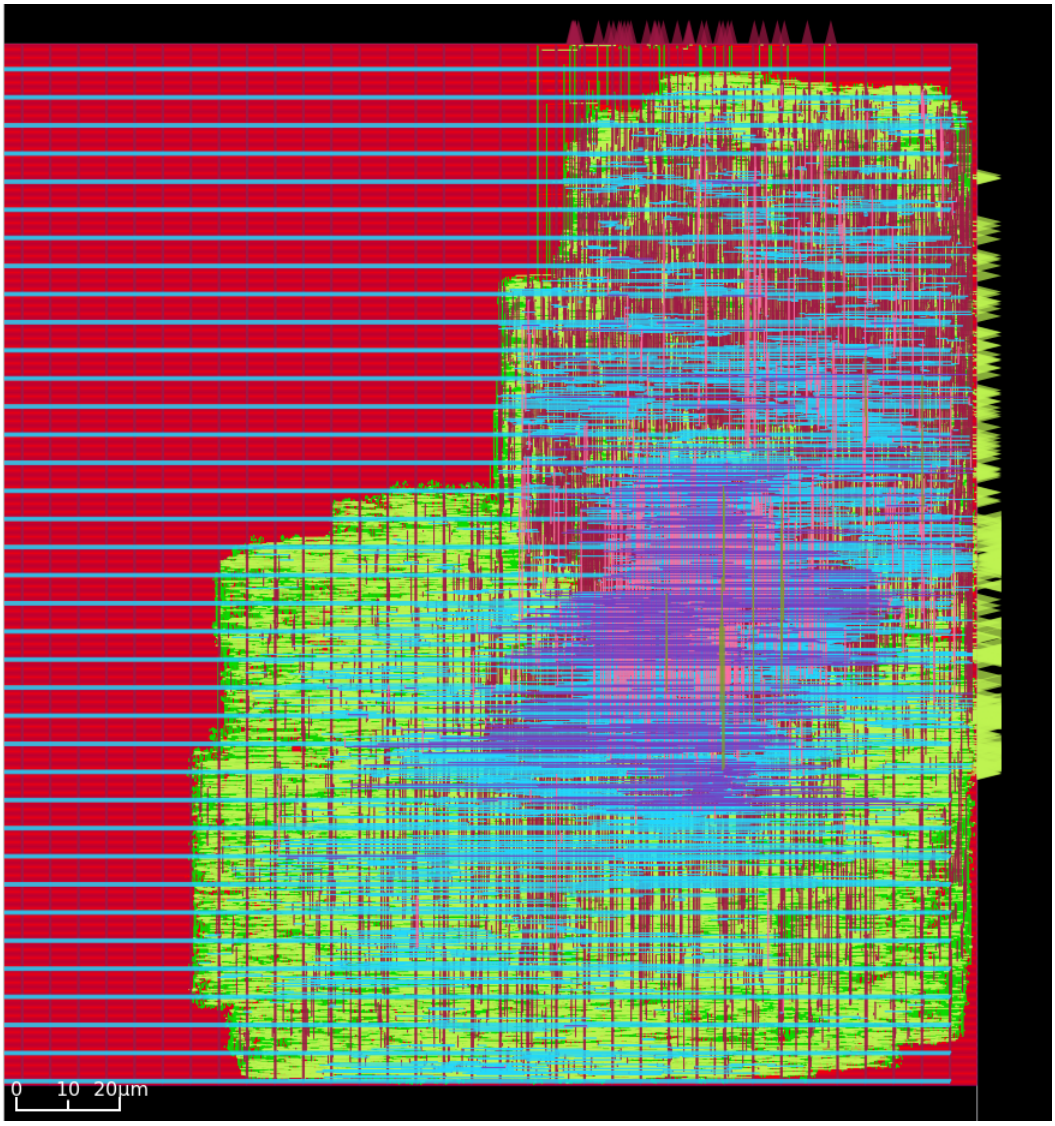


Figure 4.42: RISCV Vector Engine Routing with power

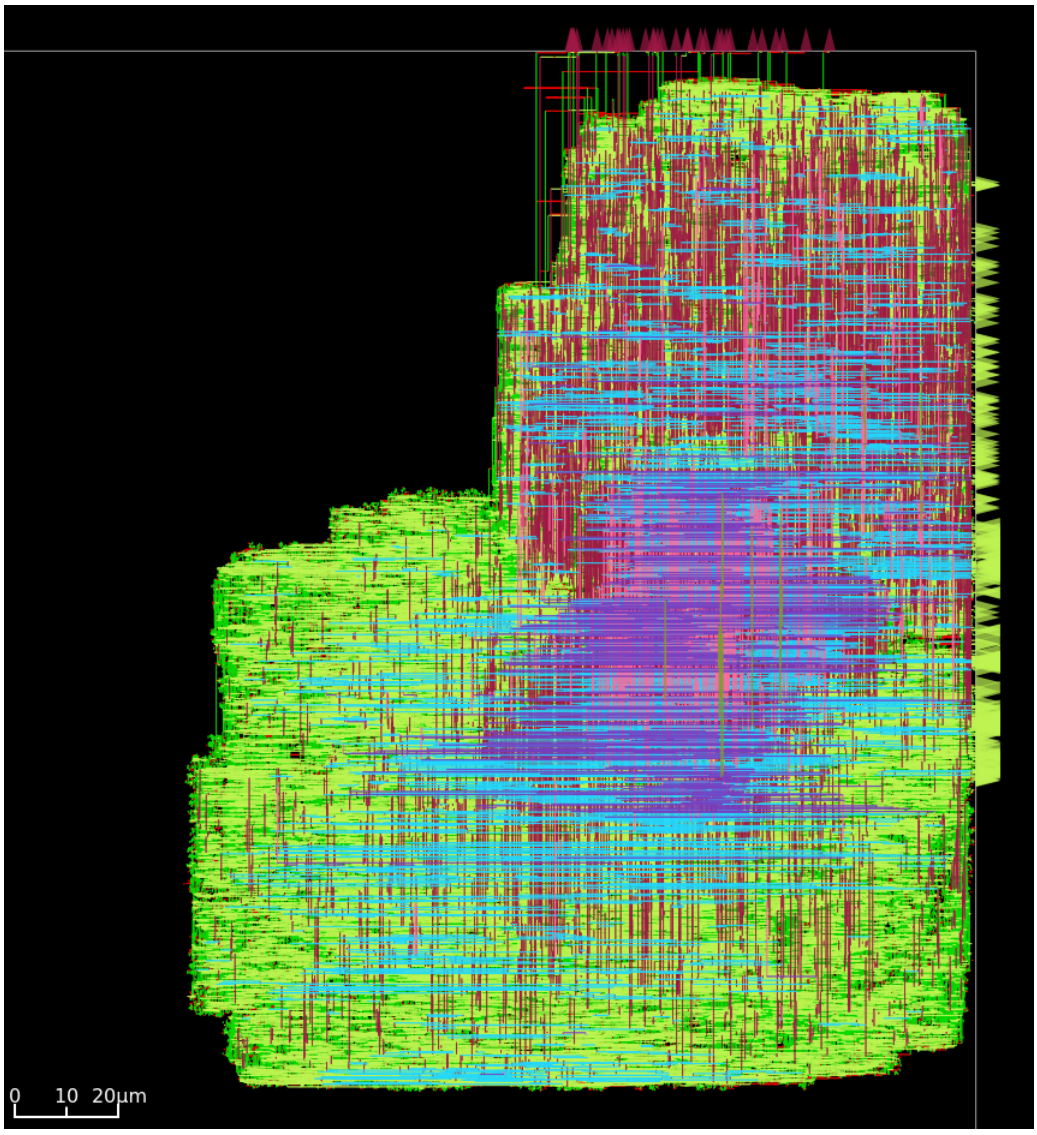


Figure 4.43: RISCV Vector Engine Routing

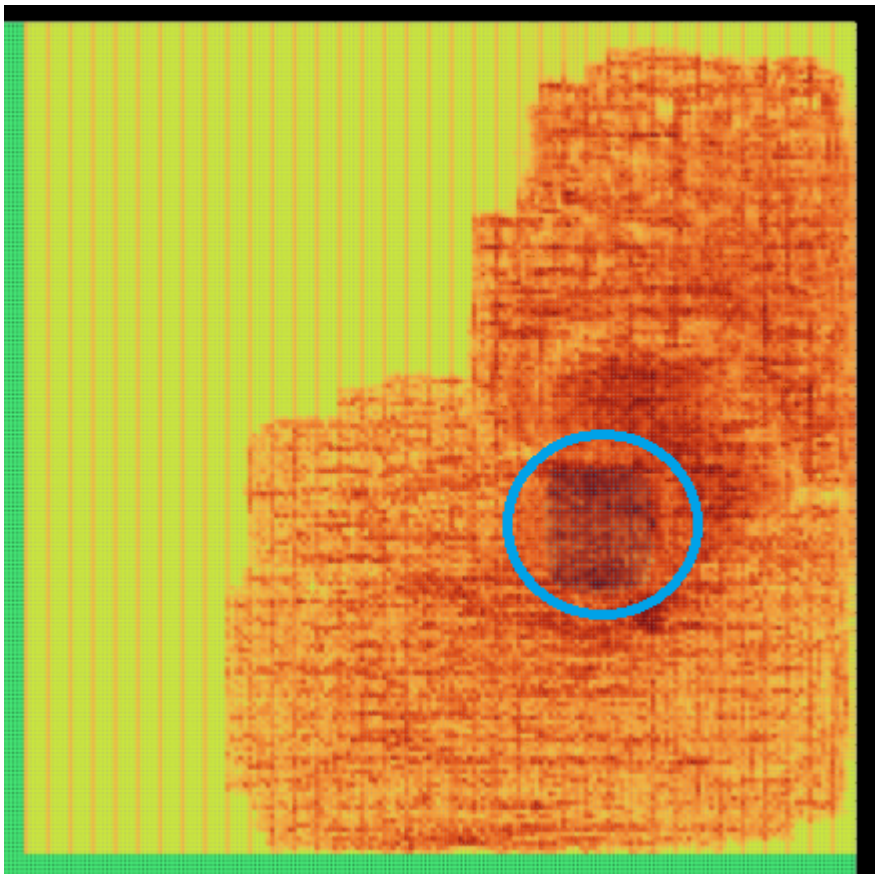


Figure 4.44: RISCV Vector Engine Routing density heat map

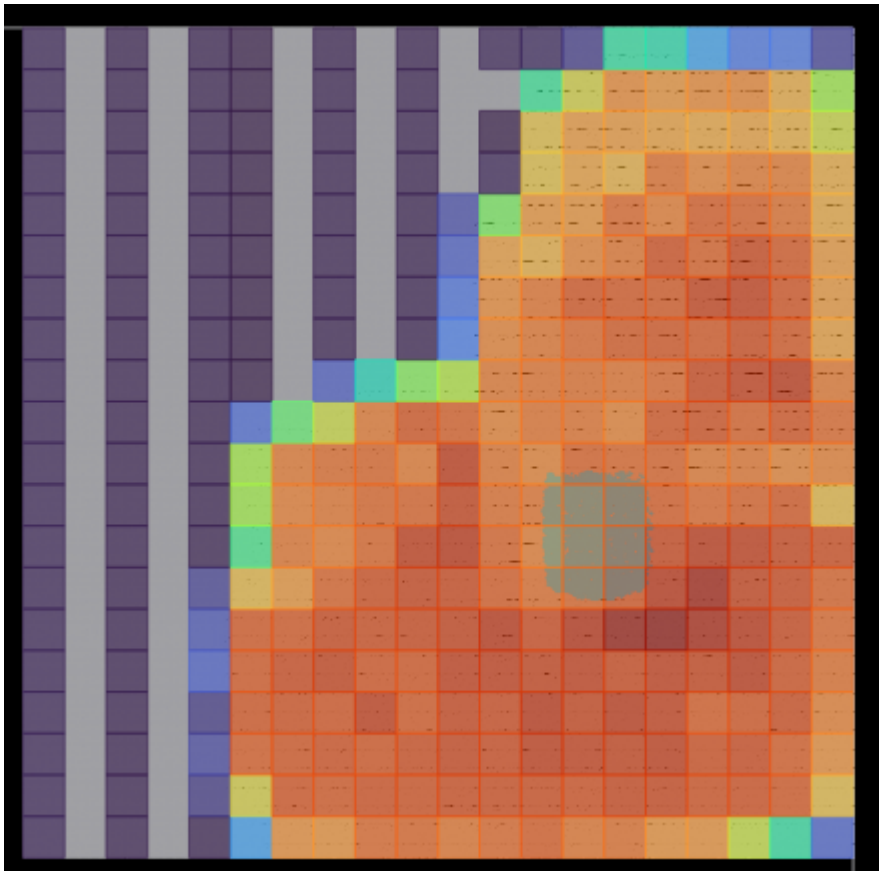


Figure 4.45: RISCV Vector Engine Placement density heat map

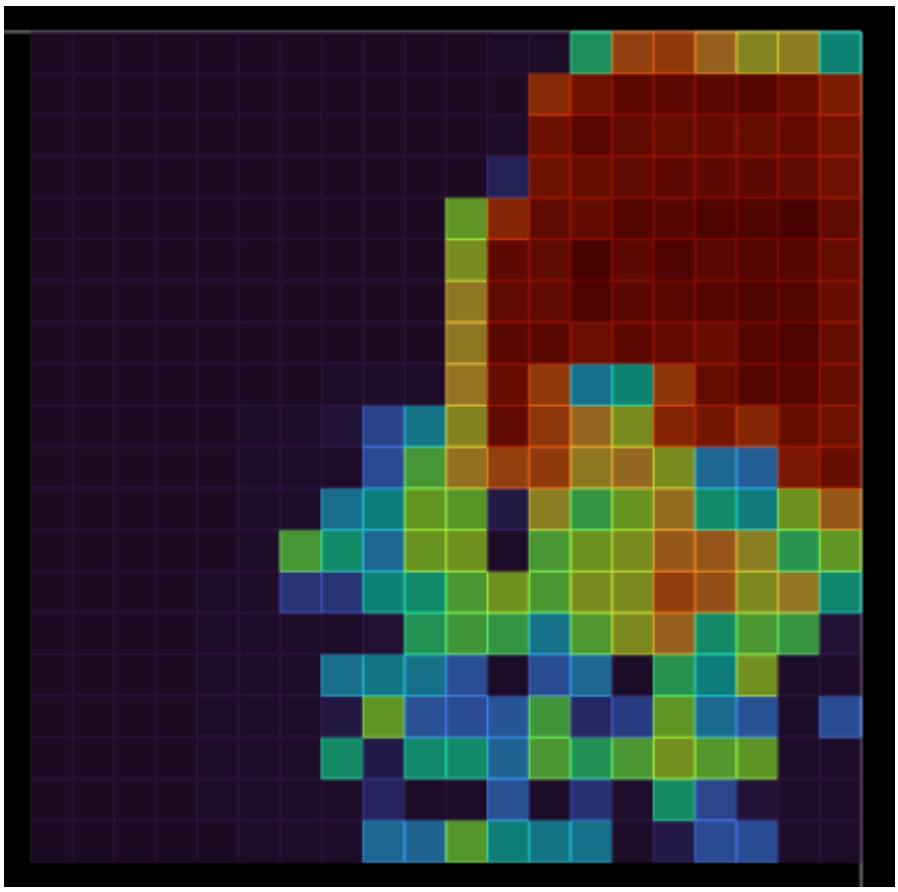


Figure 4.46: RISCV Vector Engine Power heat map

The Figures 4.47, 4.48, 4.49 represent the proportion of the sequential and combinational cells, with the Figure 4.47 showing the full circuit, the Figure 4.48 showing the combinational cells spread along all the circuit and the Figure 4.49 showing the sequential cells located at the top of the circuit.

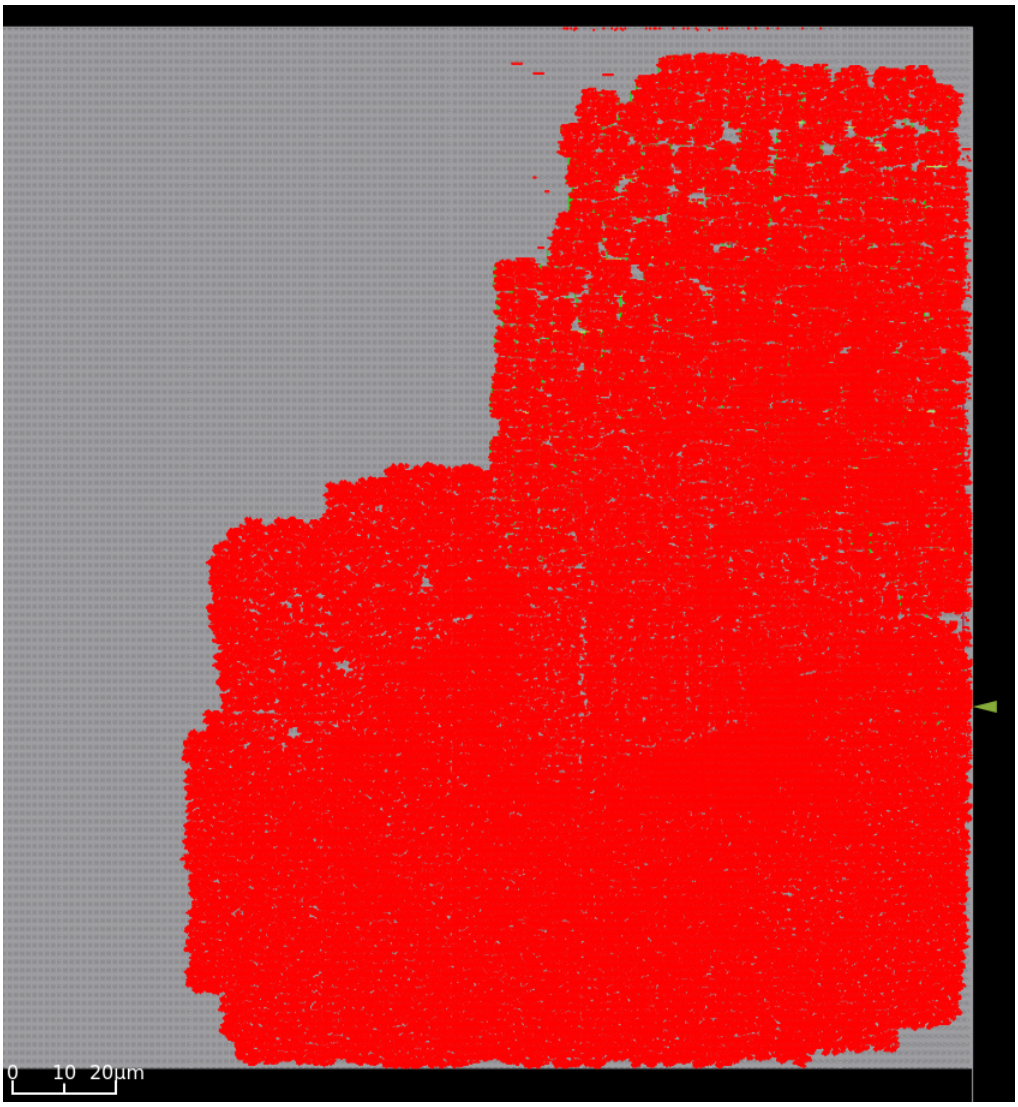


Figure 4.47: RISCV Vector Engine All Instances

Figure 4.50 shows the levels of the clock tree used by the circuit. The clock tree uses three levels of buffers to synthesize the clock tree having a latency of 180ps approximately at the endpoint of the clock tree. The Figures 4.51, 4.52 and 4.53 shows the placement of the three different levels of the clock tree in the circuit.

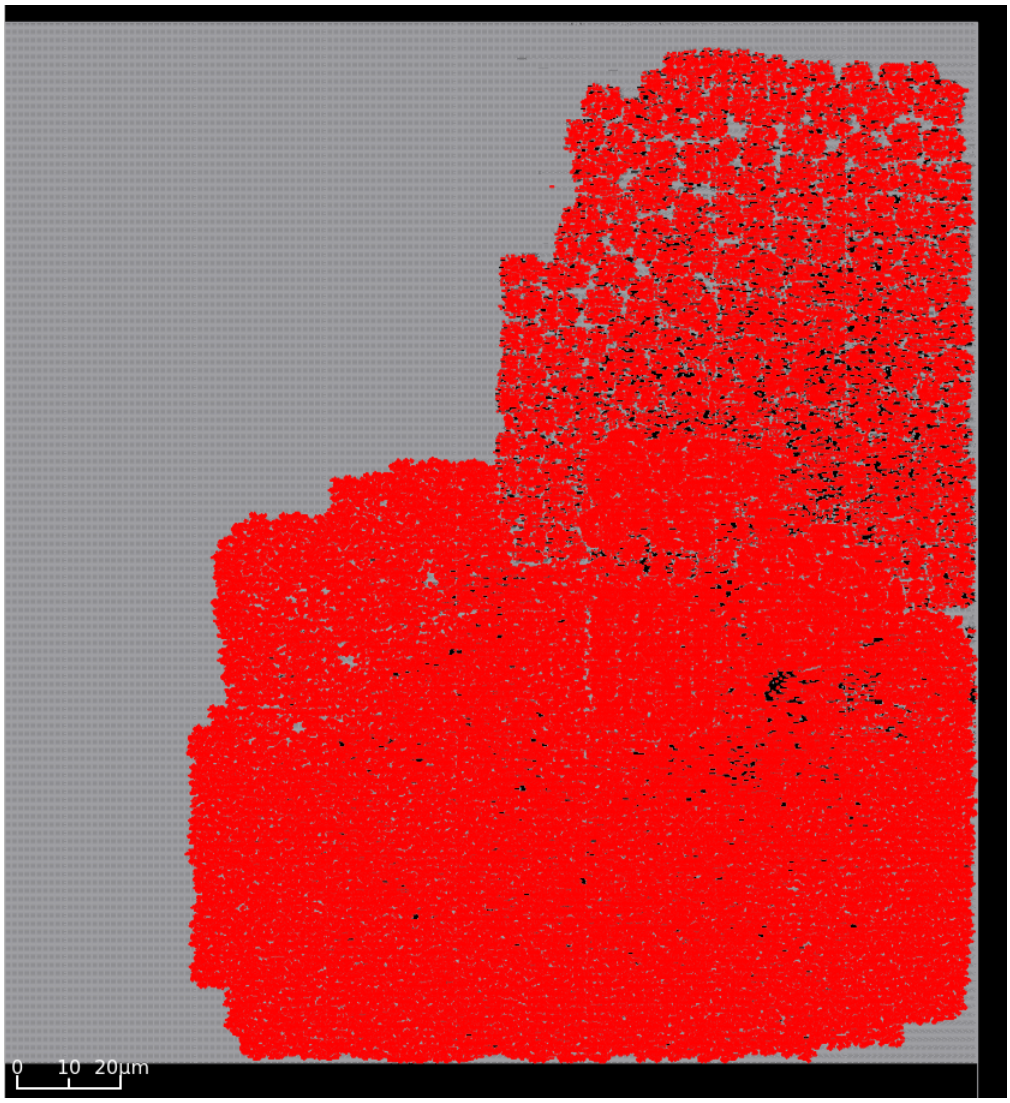


Figure 4.48: RISCV Vector Engine Combinational Instances



Figure 4.49: RISCV Vector Engine Sequential Instances

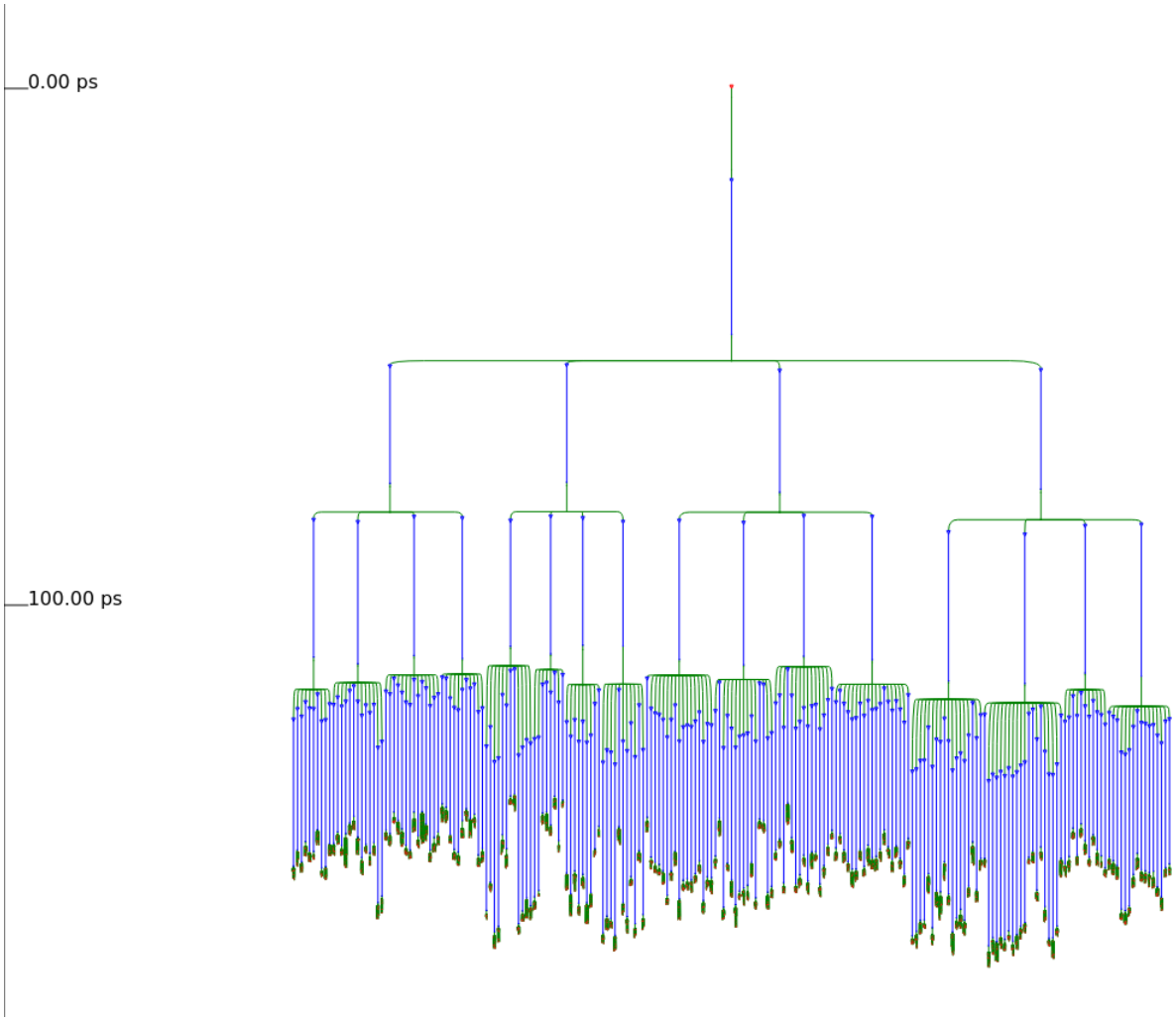


Figure 4.50: RISCV Vector Engine Clock Tree

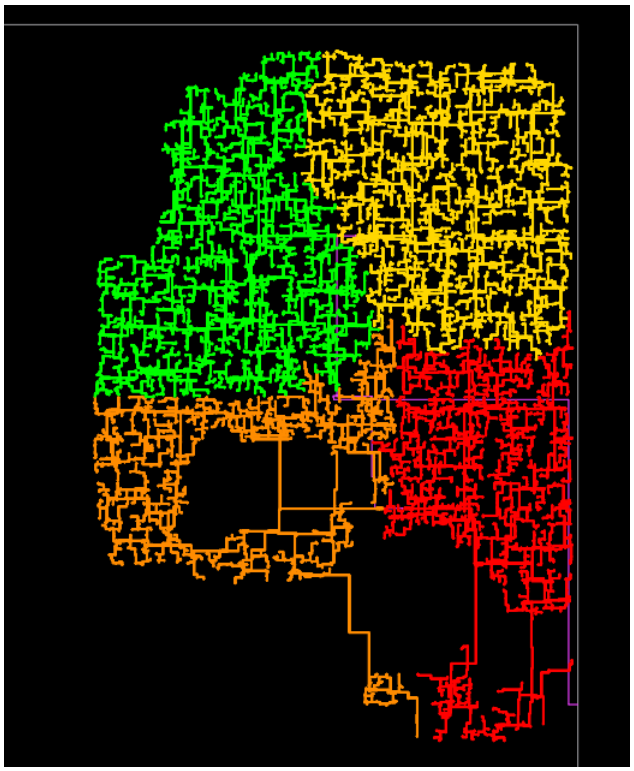


Figure 4.51: RISCV Vector Engine Clock Tree 1 Level

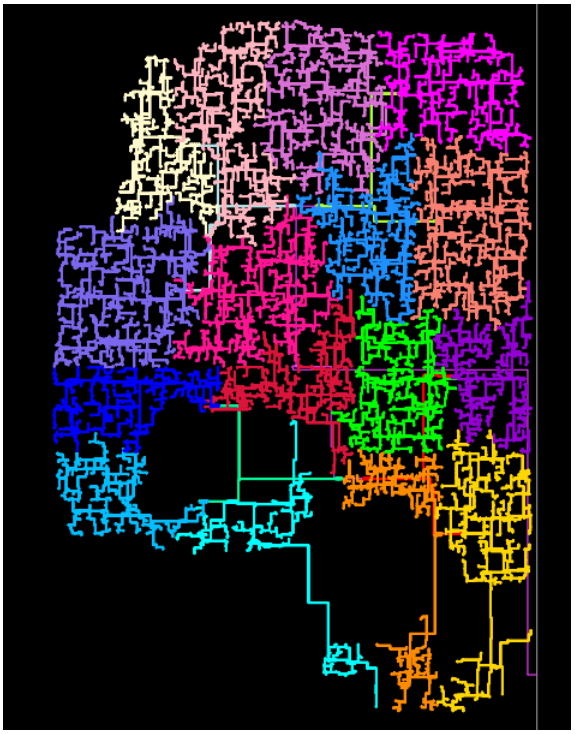


Figure 4.52: RISCV Vector Engine Clock Tree 2 Level

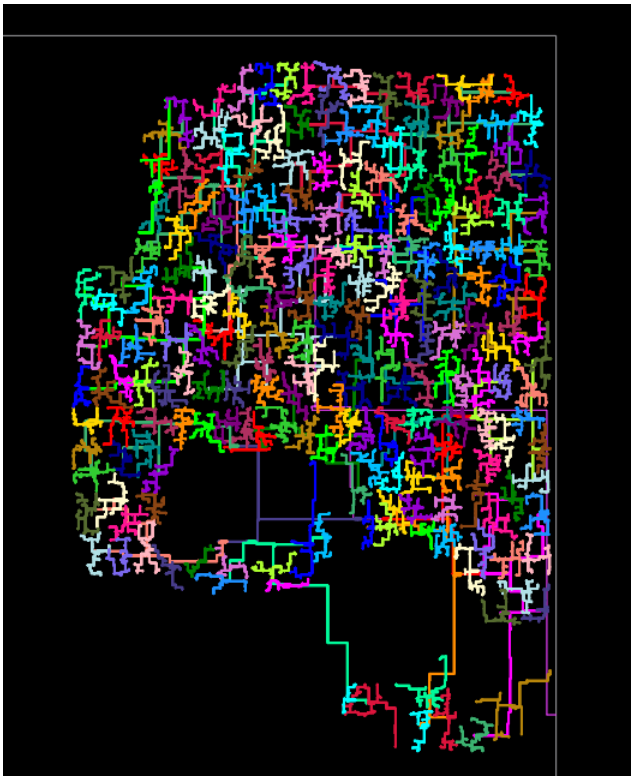


Figure 4.53: RISCV Vector Engine Clock Tree 3 Level

Chapter 5

Conclusion

The implementation of the *RISC-V Vector* extension allows the execution of vector operations improving the performance of some applications, like IA, but the implementation of these operations requires more area and could decrease the maximum frequency of operation of the circuit. Depending on the micro architecture used to implement it, the extension, area and speed of the circuit are affected in different ways, the circuit can be designed to use the least amount of area or to get the highest frequency possible. Although these two approaches are not necessarily mutually exclusive in most of the cases reducing the area of the circuit also reduces the speed, and increasing the speed also increases the area.

To reduce the area of the circuit slow circuits should be used, for example preferring the use of the *Ripple Carry Adder* over other type of adders since it is a slow adder, but it is also the adder that requires the least area.

Modifying the algorithm used to execute the operations is another option to reduce the area of the circuit. For example, implementing the operations using a multi-cycle algorithm instead of a single cycle or a pipelined algorithm. A multi-cycle algorithm reduces the area and could also increase the speed by reducing the critical path, but it also reduces the IPC. A multi-cycle data path doesn't allow back-to-back operations since the hardware is being used by the operation and won't be released until it is completed, reducing the performance of the same operation if it needs to be executed multiple times in a row.

A good way to increment the speed of the circuit is by increasing the number of pipeline stages of the circuit. Increasing the pipelining increases the speed by cutting the critical path of the circuit, but it also increases the area due to the stage registers that should be added, this is especially important in a vector data path since the vector data path uses bigger data buses than a integer data path, increasing significantly the number of registers that should be added.

The power consumption of the circuit is another important specification to be considered. Due to the high area used by the vector hardware, the power consumption also increases considerably, for that reason is important to add power features to reduce the power consumption, for example, the clock gating. A good example of this is the *Vector Multiplier* that consumes 50% of the area, if the data path of the vector is not gated and only a multiplexor at the end of the multiplier is used to select the correct result, the whole data path of the multiplier will be always toggling, even if a multiplication is not executed, causing that the 50% of the power of the circuit is consumed in an area that is not actively used.

The *Vector Register File* is another area where power features could be added. According to the power heat map, the section where the *Vector Register File* registers are placed is the section with the highest power consumption, the clock could be disabled on the flops that are not used by the operation to reduce the power consumption in that area.

The fan out of the cells is another parameter that needs to be analyzed. A high fan out increments the capacitance of the bus, reducing the speed of the circuit and incrementing the size of the gate needed. Due to the size of the data buses and the multiple formats of the operations that connect the operands in different positions the fan out of the cells increases considerably. This fan-out could be reduced by duplicating logic or flops. For example, a set of flops could be used for each ALU, in this way each set of flops can be placed closer to the ALU reducing the propagation delay, and the fan out of each flop is also reduced by reducing the logic that needs to be connected to each flop, but this also increases the area of the circuit.

The inclusion of a vector unit does not need to impact necessarily the performance of the integer unit. The clusterization technique could be used to separate both units, in this way each unit can have its own pipeline with a different number of stages, can perform operations in parallel and even could work on different clock frequencies, but this complicates the control, hazard and scheduling logic.

The synthesis stage is important to get the best results. The synthesizer tries to get the best results according to the configurations and constraints used, but this could consume a lot of time between each run and not always gives the best results. To improve the synthesis results some manual work can be done, for example placing stage registers cells manually closer to where they are used, in this way the run time is reduced since the synthesizer does not need to find an appropriate place for those cells and the speed of the circuit increases reducing the propagation delay.

In most cases improving the speed deteriorates the area and vice versa, but the same technique doesn't need to be applied to the full circuit, depending on the requirements and characteristics of each block one technique or the other can be applied, or a combination of both, for example in this project the adder was the slowest block, but not the biggest, a speed technique could be applied to improve the speed of the adder. The multiplier is the biggest block but is not the slowest, an area technique can be applied to reduce the area, in this way the speed of the multiplier and adder can be balanced increasing the overall performance.

References

[1] *RISC-V V Vector Extension*. Rev. 1.0. RISC-V. Nov. 2021.

[2] Miguel Bucio-Macías, Luis Pizano-Escalante, and Omar Longoria-Gandara. "Design of an ASIC Vector Engine for a RISC-V Architecture". In: *Chips* 4.3 (2025). ISSN: 2674-0729. DOI: 10.3390/chips4030033. URL: https://www.mdpi.com/2674-0729/4/3/33.

[3] Ronald. Tocci, Neal. Widmer, and Gregory Moss. *Sistemas Digitales Principios y Aplicaciones*. Pearson, 2007, p. 300.

[4] Akshitha and Niju Rajan. "Power Reduction of Half Adder and Half Subtractor Using Different Partial Adiabatic Logic Styles". In: *2019 International Conference on Intelligent Sustainable Systems (ICISS)*. 2019, pp. 87–92. DOI: 10.1109/ISS1.2019.8908104.

[5] Omnia Al. Badry and M. A. Abdelghany. "Low power 1-Bit full adder using Full-Swing gate diffusion input technique". In: *2018 International Conference on Innovative Trends in Computer Engineering (ITCE)*. 2018, pp. 205–208. DOI: 10.1109/ITCE.2018.8316625.

[6] M. Mohanapriya R.UMA Vidya Vijayan and Sharon Paul. "Area, Delay and Power Comparison of Adder Topologies". In: *International Journal of VLSI design & Communication Systems* 13.1 (2012), pp. 153–168.

[7] C. N. Shilpa, Kunjan D. Shinde, and H. V. Nithin. "Design, Implementation and Comparative Analysis of Kogge Stone Adder Using CMOS and GDI Design: A VLSI Based Approach". In: *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2016, pp. 570–574. DOI: 10.1109/CICN.2016.117.

[8] R. Rajalakshmi and P. Aruna Priya. "Design and analysis of a 4-bit low power universal Barrel-shifter in 16nm FinFET technology". In: *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*. 2014, pp. 527–532. DOI: 10.1109/ICACCCT.2014.7019141.

[9] Rahul Rathod et al. "Implementation of 32-Bit Complex Floating Point Multiplier Using Vedic Multiplier, Array Multiplier and Combined integer and floating point Multiplier (CIFM)". In: *2020 IEEE International Conference for Innovation in Technology (INOCON)*. 2020, pp. 1–5. DOI: 10.1109/INOCON50539.2020.9298363.

[10] Dravik KishorBhai Kahar and Harsh Mehta. “High speed vedic multiplier used vedic mathematics”. In: *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*. 2017, pp. 356–359. DOI: 10.1109/ICCONS.2017.8250742.

[11] Rhodes B and Dan N. *Coding techniques for bus functional models in verilog, vhdl and c++ [Online]*. Available: https://www.syncad.com/paper_coding_techniques_icu_sep_2003.htm.

[12] *IEEE Standar for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*. IEEE. Dec. 2009.

[13] *Universal Verification Methodology (UVM) 1.2 Class Reference*. Accellera. June 2014.

[14] *Universal Verification Methodology (UVM) 1.2 User’s Guide*. Accellera. Oct. 2015.

[15] Tutu Ajayi et al. “Toward an open-source digital flow: First learnings from the openroad project”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–4.

[16] Google and Global Foundries. *gf180mcu-pdk [Online]*. Available: https://github.com/google/skywater-pdk.

[17] Google and SkyWater Technology. *skywater-pdk [Online]*. Available: https://github.com/google/skywater-pdk.

[18] NC State University. *FreePDK45 [Online]*. Available: https://eda.ncsu.edu/freepdk/freepdk45/.

[19] Lawrence T. Clark et al. “ASAP7: A 7-nm finFET predictive process design kit”. In: *Microelectronics Journal* 53 (2016), pp. 105–115. ISSN: 1879-2391. DOI: https://doi.org/10.1016/j.mejo.2016.04.006. URL: https://www.sciencedirect.com/science/article/pii/S002626921630026X.

Appendix A

Appendix

A.1 Glossary

A.1.1 DEST_SIZE

DEST_SIZE refers to the size of the operation destination, valid *DEST_SIZE* are:
DEST_SIZE_8: Destination size is 8-bits
DEST_SIZE_16: Destination size is 16-bits
DEST_SIZE_32: Destination size is 32-bits
DEST_SIZE_64: Destination size is 64-bits
DEST_SIZE_128: Destination size is 128-bits

A.1.2 SRC_SIZE

SRC_SIZE refers to the size of the operation sources, valid *SRC_SIZE* are:
SRC_SIZE_8: Source size is 8-bits
SRC_SIZE_16: Source size is 16-bits
SRC_SIZE_32: Source size is 32-bits
SRC_SIZE_64: Source size is 64-bits
SRC_SIZE_128: Source size is 128-bits

A.1.3 OSIZE

OSIZE refers to the size of the operation, for most of instructions *OSIZE* is equal to *SRC_SIZE* and equal to *DEST_SIZE*. Valid *OSIZE* are:
OSIZE_8: Operation size is 8-bits
OSIZE_16: Operation size is 16-bits
OSIZE_32: Operation size is 32-bits
OSIZE_64: Operation size is 64-bits
OSIZE_128: Operation size is 128-bits

A.2 Operations implemented

A.2.1 Arithmetic Instructions

VADD

Description: Performs addition operation between *srca* and *srcb*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_8, OSIZE_16, OSIZE_32, OSIZE_64, OSIZE_128*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[(idx*osize) +: osize] +
    srcb[(idx*osize) +: osize];
        end
    end else begin //Scalar and immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[0 +: osize] + srcb[(idx*
    osize) +: osize];
        end
    end

```

VADC

Description: Performs addition operation between *srca* and *srcb* and adds carry coming from *carry_in*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_8, OSIZE_16, OSIZE_32, OSIZE_64, OSIZE_128*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *Carry_in*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[(idx*osize) +: osize] +
    srcb[(idx*osize) +: osize] + carry_in[idx];
        end
    end else begin //Scalar and immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[(idx*osize) +: osize] +
    srcb[0 +: osize] + carry_in[idx];
```

```
        end
    end

```

VSUB

Description: Performs subtraction operation between *srca* and *srcb*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_8, OSIZE_16, OSIZE_32, OSIZE_64, OSIZE_128*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[(idx*osize) +: osize] -
    srcb[(idx*osize) +: osize];
        end
    end else begin //Scalar and immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[0 +: osize] - srcb[(idx*
    osize) +: osize];
        end
    end

```

VSBC

Description: Performs subtraction operation between *srca* and *srcb* and subtracts borrow coming from *borrow_in*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_8, OSIZE_16, OSIZE_32, OSIZE_64, OSIZE_128*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *Borrow_in*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[(idx*osize) +: osize] -
    srcb[(idx*osize) +: osize] - borrow_in[idx];
        end
    end else begin //Scalar and immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(idx*osize) +: osize] = srca[0 +: osize] - srcb[(idx*
    osize) +: osize] - borrow_in[idx];
```

```
        end
    end

```

VZEXT

Description: Zero extends *srcb* to *destination* size. Note: This instruction does not use *OSIZE* directly, it uses a combination of *DESTINATION_SIZE* and *SOURCE_SIZE*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *DEST_128-SRC_64, DEST_128-SRC_32, DEST_128-SRC_16, DEST_64-SRC_32, DEST_64-SRC_16, DEST_64-SRC_8, DEST_32-SRC_16, DEST_32-SRC_8, DEST_16-SRC_8*
Srca valid format: SRC_SIZE
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(dest_size);

    for (int idx = 0; idx < num_elements; idx++) begin
        result[(idx*dest_size) +: dest_size] = zero_ext(srcb[(idx*src_size
    ) +: src_size], dest_size);
    end

```

VSEXT

Description: Sign extends *srcb* to *destination* size. Note: This instruction does not use *OSIZE* directly, it uses a combination of *DESTINATION_SIZE* and *SOURCE_SIZE*
ALU: *Arithmetic ALU*
Valid *OSIZE*: *DEST_128-SRC_64, DEST_128-SRC_32, DEST_128-SRC_16, DEST_64-SRC_32, DEST_64-SRC_16, DEST_64-SRC_8, DEST_32-SRC_16, DEST_32-SRC_8, DEST_16-SRC_8*
Srca valid format: SRC_SIZE
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(dest_size);

    for (int idx = 0; idx < num_elements; idx++) begin
        result[(idx*dest_size) +: dest_size] = sign_ext(srcb[(idx*src_size
    ) +: src_size], dest_size);
    end

```

VMUL

Description: Multiplies signed *srca* and signed *srcb* and returns low half of product
ALU: *Arithmetic ALU*

Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int result_size = 2*osize;
    int num_elements = get_num_elements(osize);
    signed logic [result_size-1] tmp;

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            tmp[(result_size*idx) +: result_size] = signed'(srca[(osize*
    idx) +: osize]) * signed'(srcb[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            tmp[(result_size*idx) +: result_size] = signed'(srca[0 +:
    osize]) * signed'(srcb[(idx*osize) +: osize]);
        end
    end

    for (int idx = 0; idx < num_elements; idx++) begin
        result[(osize*idx) +: osize] = tmp[(result_size*idx) +: osize];
    end

```

VMULH

Description: Multiplies signed *srca* and signed *srcb* and returns high half of product
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int result_size = 2*osize;
    int num_elements = get_num_elements(osize);
    signed logic [result_size-1] tmp;

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            tmp[(result_size*idx) +: result_size] = signed'(srca[(osize*
    idx) +: osize]) * signed'(srcb[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            tmp[(result_size*idx) +: result_size] = signed'(srca[0 +:
    osize]) * signed'(srcb[(idx*osize) +: osize]);
        end
    end
```

```
14
15      for (int idx = 0; idx < num_elements; idx++) begin
16          result[(osize*idx) +: osize] = tmp[((result_size*idx)+osize) +:
    osize];
17      end
18
```

VMULHU

Description: Multiplies unsigned *srca* and unsigned *srcb* and returns high half of product
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
1       int result_size = 2*osize;
2       int num_elements = get_num_elements(osize);
3       unsigned logic [result_size-1] tmp;
4
5       if (srca.format == Vector) begin
6           for (int idx = 0; idx < num_elements; idx++) begin
7               tmp[(result_size*idx) +: result_size] = unsigned'(srca[(osize*
    idx) +: osize]) * unsigned'(srcb[(osize*idx) +: osize]);
8           end
9       end else begin //Scalar
10          for (int idx = 0; idx < num_elements; idx++) begin
11              tmp[(result_size*idx) +: result_size] = unsigned'(srca[0 +:
    osize]) * unsigned'(srcb[(idx*osize) +: osize]);
12          end
13      end
14
15      for (int idx = 0; idx < num_elements; idx++) begin
16          result[(osize*idx) +: osize] = tmp[((result_size*idx)+osize) +:
    osize];
17      end
18
```

A.2.2 Logic Instructions

VAND

Description: Performs bitwise AND between *srca* and *srcb*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[(osize*idx) +: osize] &
    srcb[(osize*idx) +: osize];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[0 +: osize] & srcb[(idx*
    osize) +: osize];
        end
    end

```

VOR

Description: Performs bitwise OR between *srca* and *srcb*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[(osize*idx) +: osize] |
    srcb[(osize*idx) +: osize];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[0 +: osize] | srcb[(idx*
    osize) +: osize];
        end
    end

```

VXOR

Description: Performs bitwise XOR between *srca* and *srcb*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[(osize*idx) +: osize] ^
srcb[(osize*idx) +: osize];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srca[0 +: osize] ^ srcb[(idx*
osize) +: osize];
        end
    end

```

VSLL

Description: Performs logical shift left, the data to be shifted comes from *srcb* and shift count comes from *srca*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srcb[(osize*idx) +: osize] <<
srca[(osize*idx) +: log2(osize)];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srcb[(osize*idx) +: osize] <<
srca[0 +: log2(osize)];
        end
    end

```

VSRL

Description: Performs logical shift right, the data to be shifted comes from *srcb* and shift count comes from *srca*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*

Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srcb[(osize*idx) +: osize] >>
    srca[(osize*idx) +: log2(osize)];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = srcb[(osize*idx) +: osize] >>
    srca[0 +: log2(osize)];
        end
    end

```

VSRA

Description: Performs arithmetic shift right, the data to be shifted comes from *srcb* and shift count comes from *srca*
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = signed'(srcb[(osize*idx) +:
    osize]) >>> srca[(osize*idx) +: log2(osize)];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = signed'(srcb[(osize*idx) +:
    osize]) >>> srca[0 +: log2(osize)];
        end
    end

```

A.2.3 Compare Instructions

VMSEQ

Description: Compares *srca* and *srcb* and writes 1 to the destination mask register element if they are equal, writes 0 otherwise
ALU: *Arithmetic ALU*

Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = srcb[(osize*idx) +: osize] == srca[(osize*
    idx) +: osize];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = srcb[(osize*idx) +: osize] == srca[0 +:
    osize];
        end
    end

```

VMSNE

Description: Compares *srca* and *srcb* and writes 1 to the destination mask register element if they are not equal, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = srcb[(osize*idx) +: osize] != srca[(osize*
    idx) +: osize];
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = srcb[(osize*idx) +: osize] != srca[0 +:
    osize];
        end
    end

```

VMSLTU

Description: Compares unsigned *srca* and unsigned *srcb* and writes 1 to the destination mask register element if srcb is less than srca, writes 0 otherwise

ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) < unsigned'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) < unsigned'(srca[0 +: osize]);
        end
    end

```

VMSLT

Description: Compares signed *srca* and signed *srcb* and writes 1 to the destination mask register element if srcb is less than srca, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) < signed'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) < signed'(srca[0 +: osize]);
        end
    end

```

VMSLEU

Description: Compares unsigned *srca* and unsigned *srcb* and writes 1 to the destination mask register element if srcb is less or equal than srca, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) <= unsigned'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) <= unsigned'(srca[0 +: osize]);
        end
    end

```

VMSLE

Description: Compares signed *srca* and signed *srcb* and writes 1 to the destination mask register element if srcb is less or equal than srca, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) <= signed'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar or immediate
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) <= signed'(srca[0 +: osize]);
        end
    end

```

VMSGTU

Description: Compares unsigned *srca* and unsigned *srcb* and writes 1 to the destination mask register element if srcb is greater than srca, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) > unsigned'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = unsigned'(srcb[(osize*idx) +: osize]) > unsigned'(srca[0 +: osize]);
        end
    end

```

VMSGT

Description: Compares signed *srca* and signed *srcb* and writes 1 to the destination mask register element if srcb is greater than srca, writes 0 otherwise
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Scalar, Immediate*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) > signed'(srca[(osize*idx) +: osize]);
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result_mask[idx] = signed'(srcb[(osize*idx) +: osize]) > signed'(srca[0 +: osize]);
        end
    end

```

VMINU

Description: Compares unsigned *srca* and unsigned *srcb* and writes the minimum value to destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = min(unsigned'(srcb[(osize*idx) +: osize]), unsigned'(srca[(osize*idx) +: osize]));
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = min(unsigned'(srcb[(osize*idx) +: osize]), unsigned'(srca[0 +: osize]));
        end
    end

```

VMIN

Description: Compares signed *srca* and signed *srcb* and writes the minimum value to destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = min(signed'(srcb[(osize*idx) +: osize]), signed'(srca[(osize*idx) +: osize]));
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = min(signed'(srcb[(osize*idx) +: osize]), signed'(srca[0 +: osize]));
        end
    end

```

VMAXU

Description: Compares unsigned *srca* and unsigned *srcb* and writes the maximum value to destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = max(unsigned'(srcb[(osize*idx) +: osize]), unsigned'(srca[(osize*idx) +: osize]));
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = max(unsigned'(srcb[(osize*idx) +: osize]), unsigned'(srca[0 +: osize]));
        end
    end

```

VMAX

Description: Compares signed *srca* and unsigned *srcb* and writes the maximum value to destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    if (srca.format == Vector) begin
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = max(signed'(srcb[(osize*idx) +: osize]), signed'(srca[(osize*idx) +: osize]));
        end
    end else begin //Scalar
        for (int idx = 0; idx < num_elements; idx++) begin
            result[(osize*idx) +: osize] = max(signed'(srcb[(osize*idx) +: osize]), signed'(srca[0 +: osize]));
        end
    end

```

A.2.4 Vector reduction Instructions

VREDSUM

Description: Performs Addition operation between all elements of srcb and the first element of srca and stores the result in the first element of destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector, Scalar*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = srcb[(idx*osize) +: osize] + srca[0 +: osize];
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = srcb[(idx*osize) +: osize] + result[0 +:
osize];
    end

```

VREDMAXU

Description: Performs max operation between all unsigned elements of srcb and the first unsigned element of srca and stores the result in the first element of destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = max(unsigned'(srcb[(idx*osize) +: osize]),
unsigned'(srca[0 +: osize]));
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = max(unsigned'(srcb[(idx*osize) +: osize]),
unsigned'(result[0 +: osize]));
    end

```

VREDMAX

Description: Performs max operation between all signed elements of srcb and the first signed element of srca and stores the result in the first element of destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*

Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = max(signed'(srcb[(idx*osize) +: osize]), signed'(
srca[0 +: osize]));
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = max(signed'(srcb[(idx*osize) +: osize]),
signed'(result[0 +: osize]));
    end

```

VREDMINU

Description: Performs min operation between all unsigned elements of srcb and the first unsigned element of srca and stores the result in the first element of destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = min(unsigned'(srcb[(idx*osize) +: osize]),
unsigned'(srca[0 +: osize]));
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = min(unsigned'(srcb[(idx*osize) +: osize]),
unsigned'(result[0 +: osize]));
    end

```

VREDMIN

Description: Performs min operation between all signed elements of srcb and the first signed element of srca and stores the result in the first element of destination
ALU: *Arithmetic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = min(signed'(srcb[(idx*osize) +: osize]), signed'(
srca[0 +: osize]));
```

```
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = min(signed'(srcb[(idx*osize) +: osize]),
    signed'(result[0 +: osize]));
    end

```

VREDAND

Description: Performs bitwise AND operation between all elements of srcb and the first element of srca and stores the result in the first element of destination
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = srcb[(idx*osize) +: osize] & srca[0 +: osize];
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = srcb[(idx*osize) +: osize] & result[0 +:
    osize];
    end

```

VREDOR

Description: Performs bitwise OR operation between all elements of srcb and the first element of srca and stores the result in the first element of destination
ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = srcb[(idx*osize) +: osize] | srca[0 +: osize];
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = srcb[(idx*osize) +: osize] | result[0 +:
    osize];
    end

```

VREDXOR

Description: Performs bitwise XOR operation between all elements of srcb and the first element of srca and stores the result in the first element of destination

ALU: *Logic ALU*
Valid *OSIZE*: *OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *Vector*
Srcb valid format: *Vector*
Supports Mask: *True*

```
    int num_elements = get_num_elements(osize);

    result[0 +: osize] = srcb[(idx*osize) +: osize] ^ srca[0 +: osize];
    for (int idx = 1; idx < num_elements; idx++) begin
        result[0 +: osize] = srcb[(idx*osize) +: osize] ^ result[0 +:
    osize];
    end

```

A.2.5 Vector mask Instructions

VMAND

Description: Performs logical AND operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = mask_a[idx] && mask_b[idx];
    end

```

VMNAND

Description: Performs logical NAND operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = !(mask_a[idx] && mask_b[idx]);
    end

```

VMANDN

Description: Performs logical AND operation between all negated elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = !mask_a[idx] && mask_b[idx];
    end

```

VMXOR

Description: Performs logical XOR operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = mask_a[idx] ^^ mask_b[idx];
    end

```

VMOR

Description: Performs logical OR operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = mask_a[idx] || mask_b[idx];
    end

```

VMNOR

Description: Performs logical NOR operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = !(mask_a[idx] || mask_b[idx]);
    end

```

VMORN

Description: Performs logical OR operation between all negated elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = !mask_a[idx] || mask_b[idx];
    end

```

VMXNOR

Description: Performs logical XNOR operation between all elements of *mask_a* and *mask_b* and stores the result in *dest_mask*
ALU: *Logic ALU*
Valid *OSIZE*: N/A
Srca valid format: *Mask*
Srcb valid format: *Mask*
Supports Mask: *False*

```
    for (int idx = 1; idx < NUM_MASK_ELEMENTS; idx++) begin
        dest_mask[idx] = !(mask_a[idx] ^^ mask_b[idx]);
    end

```

A.2.6 Vector permutation Instructions

VMV

Description: Performs transfer between a scalar operand and element 0 of vector destination register or element 0 of a vector register to a scalar destination **ALU:** *Logic ALU*
Valid *OSIZE*: *OSIZE_128, OSIZE_64, OSIZE_32, OSIZE_16, OSIZE_8*
Srca valid format: *scalar*
Srcb valid format: *Vector*
Supports Mask: *False*

```
    if (dest.format == scalar) begin //Vector to scalar
        if (osize >= SCALAR_SIZE begin
            scalar_dest = srcb[0 +: SCALAR_SIZE];
        end else begin
            scalar_dest = sign_extend(srcb[0 +: OSIZE], SCALAR_SIZE);
        end
    else begin //Scalar to Vector
        if (osize > SCALAR_SIZE) begin
            vect_dest = sign_extend(srca, osize);
        end else begin
            vect_dest = srca[0 +: SCALAR_SIZE];
        end
    end

```

A.3 Auxiliar functions

A.3.1 get_num_elements()

Return the number of elements depending on *OSIZE*

```
        int function get_num_elements(int osize);
            return (DATA_WIDTH/OSIZE);
        endfunction: get_num_elements

```

A.3.2 zero_extend()

Zero extends source to destination size

```
        logic[dst_size-1:0] function zero_extend(logic[src_size-1:0] src,
    int dst_size)
            logic[dst_size-1:0] result;
            result = {{(dst_size-src_size){1'b0}},src};
            return result;
        endfunction: zero_extend

```

A.3.3 sign_extend()

Sign extends source to destination size

```
        logic[dst_size-1:0] function sign_extend(logic[src_size-1:0] src,
    int dst_size)
            logic[dst_size-1:0] result;
            result = {{(dst_size-src_size){src[src_size-1]}},src};
            return result;
        endfunction: sign_extend

```

A.3.4 min()

Compares both sources and returns the minimum value

```
        data_t function min(data_t srca, data_t srcb)
            if (srca < srcb) begin
                return srca;
            end else begin
                return srcb;
            end
        endfunction: min

```

A.3.5 max()

Compares both sources and returns the maximum value

```
        data_t function max(data_t srca, data_t srcb)
            if (srca > srcb) begin
                return srca;
            end else begin
                return srcb;
            end
        endfunction: max

```