Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL

Alejandro Daniel Lemus Najera
Asesorado por el Ing. Iván René Morales Argueta

Guatemala, julio de 2021

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ALEJANDRO DANIEL LEMUS NAJERA
ASESORADO POR EL ING. IVÁN RENÉ MORALES ARGUETA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO ELECTRÓNICO

GUATEMALA, JULIO DE 2021

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Christian Moisés de la Cruz Leal
VOCAL V	Br. Kevin Armando Cruz Lorente
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
EXAMINADOR	Ing. Guillermo Antonio Puente Romero
EXAMINADOR	Ing. Carlos Alberto Navarro Fuentes
EXAMINADOR	Ing. Francisco Javier González López
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha 6 de febrero de 2020.

Alejandro Daniel Lemus Najera

Guatemala, 23 de septiembre de 2020

Ingeniero
Julio Solares Peñate
Coordinador de Área de Electrónica
Facultad de Ingeniería
Universidad de San Carlos de Guatemala

Señor Coordinador:

Por este medio tengo el gusto de informarle que he concluido con el asesoramiento y revisión del trabajo de graduación con título: **Diseño e implementación de un procesador RISC de 32 bits de código abierto en un dispositivo de lógica programable, utilizando VHDL**, desarrollado por el estudiante Alejandro Daniel Lemus Najera con carné 201404193. Después de revisar su contenido final doy mi entera aprobación al mismo.

Atentamente,

Iván René Morales Argueta
Ingeniero Electrónico
Colegiado 12489

MSc. Ing. Iván René Morales Argueta
Colegiado activo No. 12489

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Guatemala, 7 de octubre de 2020

Señor Director
Armando Alonso Rivera Carrillo
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería, USAC

Estimado Señor Director:

Por este medio me permito dar aprobación al Trabajo de Graduación titulado **DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL,** desarrollado por el estudiante **Alejandro Daniel Lemus Najera**, ya que considero que cumple con los requisitos establecidos.

Sin otro particular, aprovecho la oportunidad para saludarlo.

Atentamente,

ID Y ENSEÑAD A TODOS

Ing. Julio César Solares Peñate
Coordinador de Electrónica

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA MECÁNICA ELÉCTRICA
FACULTAD DE INGENIERÍA

REF. EIME 60. 2021.

El Director de la Escuela de Ingeniería Mecánica Eléctrica, después de conocer el dictamen del Asesor, con el Visto Bueno del Coordinador de Área, al trabajo de Graduación del estudiante; ALEJANDRO DANIEL LEMUS NÁJERA titulado; DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL, procede a la autorización del mismo.

Ing. Armando Alonso Rivera Carrillo

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA
FACULTAD DE INGENIERIA

GUATEMALA, 8 DE ABRIL 2,021.

Decanato
Facultad de Ingeniería
24189101 - 24189102

DTG. 282-2021

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería Mecánica Eléctrica, al Trabajo de Graduación titulado: **DISEÑO E IMPLEMENTACIÓN DE UN PROCESADOR RISC DE 32 BITS DE CÓDIGO ABIERTO EN UN DISPOSITIVO DE LÓGICA PROGRAMABLE, UTILIZANDO VHDL,** presentado por el estudiante universitario**: Alejandro Daniel Lemus Najera,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANA
FACULTAD DE INGENIERÍA

Inga. Anabela Cordova Estrada
Decana

Guatemala, julio de 2021

AACE/cc

ACTO QUE DEDICO A:

Mi abuela Evangelina Nájera. Por su admirable papel de madre. Gracias por todo el amor, la atención y los cuidados recibidos durante tantos años. He aquí el resultado de sus acciones, no se puede esperar menos de una mujer tan responsable y con un corazón tan grande.

Mi madre Miriam Yolanda Lemus. Por brindarme su apoyo y cariño de manera constante, antes y después del día en que la vida nos separó. Gracias por su paciencia, y por su presencia, a pesar de la distancia.

Mis tíos Arminda Lemus Nájera y Randal Rubén Alvarenga Vargas. Por tratarme como a uno de sus hijos y velar siempre por mi bienestar académico y personal. Son un verdadero refugio para mí.

Mis primos Josseline, Melissa Alvarenga, Suly, Tania Lemus y Andrés Alvarenga. Por ser esos hermanos que tanto necesité en diferentes momentos de mi vida.

Mis tías María Lemus, Marina Chinchilla y Olga Armírez. Por darme su cariño y apoyo, y alentarme siempre a cumplir mis metas.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por permitirme estudiar una carrera universitaria y cumplir una de mis principales metas profesionales.
Facultad de Ingeniería	Por ser el escenario en el cual pude incrementar mis conocimientos y mejorar mi percepción de la realidad.
Departamento de Física	Por darme la oportunidad de vivir muchas experiencias inolvidables al lado de gente verdaderamente maravillosa.
Mis compañeros de clases	Por su compañía y apoyo, especialmente durante los momentos más estresantes y difíciles de la carrera.
Mis amigos de la colonia	Por su comprensión y apoyo, especialmente durante todas esas ocasiones en las que no podía salir a jugar.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES VII
LISTA DE SÍMBOLOS XVII
GLOSARIO XIX
RESUMEN XXVII
OBJETIVOS XXIX
INTRODUCCIÓN XXXI

1. CIRCUITOS DIGITALES 1
1.1. Sistemas de notación posicional 1
1.1.1. Sistema decimal 3
1.1.2. Sistema binario 3
1.1.3. Sistema hexadecimal 4
1.1.4. Conversión entre sistemas 6
1.1.4.1. Conversiones a decimal 6
1.1.4.2. Conversiones a hexadecimal 7
1.1.4.3. Conversiones a binario 10
1.2. Codificación de números enteros 11
1.2.1. Enteros sin signo 11
1.2.2. Enteros con signo 11
1.3. Álgebra booleana y compuertas lógicas 14
1.3.1. Funciones booleanas 17
1.3.2. Teoremas y propiedades básicas 19
1.3.3. La disciplina estática 21
1.3.4. Compuertas lógicas digitales 25
1.3.5. Análisis transitorio de compuertas 30

1.3.5.1. Retardo de propagación 30
1.3.5.2. Retardo de contaminación 32
1.4. Lógica combinacional 34
1.4.1. Especificaciones funcionales 34
1.4.2. Representación de suma de productos 36
1.4.3. Multiplexores 39
1.4.4. Decodificadores 45
1.4.5. Memoria de sólo lectura, ROM 46
1.5. Lógica secuencial síncrona 51
1.5.1. Circuitos secuenciales síncronos 51
1.5.2. Latch D 53
1.5.3. La disciplina dinámica 56
1.5.4. Flip-flop D 59
1.6. Máquinas de estados finitos, FSM 63
1.6.1. Diagramas de transición de estados, STD 64
1.6.2. Máquinas de Moore 66
1.6.3. Máquinas de Mealy 67
1.6.4. Implementación de una máquina de estados 68
1.7. Parámetros de desempeño 71
1.7.1. Latencia 71
1.7.2. Rendimiento 75
1.7.3. Segmentación o pipelining 76
1.8. Dispositivos lógicos programables 80
1.8.1. PROM 82
1.8.2. PLA 84
1.8.3. PAL 85
1.8.4. FPGA 85
1.8.4.1. Bloques lógicos configurables, CLB 87
1.8.4.2. Estructura de interconexión 89

1.8.4.3. Bloques optimizados 91
1.9. Lenguaje de descripción de hardware VHDL 92
1.9.1. Organización y estructura 92
1.9.1.1. Entidad 93
1.9.1.2. Arquitectura 96
1.9.2. Ventajas y desventajas 98
1.10. Tarjeta de desarrollo Nexys 2 99
1.10.1. Especificaciones 100
1.10.2. FPGA Xilinx Spartan 3E-1200 102

2. CONJUNTO DE INSTRUCCIONES DEL PROCESADOR 105
2.1. Rutas de datos o datapaths 106
2.1.1. De propósito único 107
2.1.2. Programables 110
2.2. Modelo Von Neumann 112
2.2.1. Memoria 114
2.2.1.1. Direccionamiento por bytes 116
2.2.2. Unidad central de procesamiento, CPU 117
2.2.2.1. Datapath 119
2.2.2.2. Unidad de control 120
2.3. Instrucciones 120
2.3.1. Estructura 121
2.3.2. Ejecución en una máquina Von Neumann 122
2.4. Arquitectura del conjunto de instrucciones, ISA 123
2.4.1. Arquitectura RISC 125
2.4.2. Arquitectura CISC 126
2.5. Lenguaje de transferencia de registros, RTL 127
2.6. Especificación del ISA del procesador 128
2.6.1. Modelo de la máquina 129

2.6.1.1. Estado del procesador 130
2.6.1.2. Memoria principal 131
2.6.2. Codificación de las instrucciones 131
2.6.3. Resumen de instrucciones 132
2.6.4. Especificación de instrucciones 133
2.6.5. Extensión para excepciones 152
2.6.5.1. Modo supervisor 152
2.6.5.2. Manejo de excepciones 153
2.6.6. Registros reservados 154

3. DISEÑO DEL HARDWARE 155
3.1. Método de diseño 155
3.2. Diseño del datapath 159
3.2.1. Instrucciones tipo ALU 164
3.2.2. Instrucciones de acceso a memoria 173
3.2.2.1. Load, LD 174
3.2.2.2. Store, ST 177
3.2.3. Instrucciones de salto 179
3.2.3.1. Jump, JMP 179
3.2.3.2. Saltos condicionales, BEQ/BNE 182
3.2.4. Instrucción de carga relativa, LDR 185
3.2.5. Excepciones 186
3.3. Bloque de registros de múltiples puertos, REGFILE 190
3.4. Unidad aritmética lógica, ALU 193
3.4.1. Unidad aritmética 197
3.4.1.1. Sumador de 32 bits 202
3.4.1.2. Sumador completo 203
3.4.2. Unidad de comparación 204
3.4.3. Unidad booleana 208

3.4.4. Unidad de desplazamiento 209
3.5. Unidad de control, CU 213
3.5.1. Lógica de control, CTL..... 213
3.5.1.1. Resumen de señales de control 215
3.5.1.2. Tabla de contenidos de la ROM 217
3.5.1.3. Hardware adicional 220
3.5.2. Contador de programa, PC..... 223

4. IMPLEMENTACIÓN Y SIMULACIÓN DEL DISEÑO 233
4.1. Repositorio remoto 233
4.2. Bloque de registros de múltiples puertos, REGFILE 234
4.2.1. RAM de múltiples puertos..... 234
4.2.2. Circuito general..... 236
4.2.2.1. Diagramas esquemáticos RTL..... 236
4.2.2.2. Simulación 239
4.3. Unidad aritmética lógica, ALU 240
4.3.1. Sumador completo..... 240
4.3.1.1. Diagramas esquemáticos RTL..... 240
4.3.1.2. Simulación 244
4.3.2. Sumador de 32 bits..... 245
4.3.3. Compuerta NOR de 32 entradas 246
4.3.4. Unidad aritmética..... 248
4.3.4.1. Diagramas esquemáticos RTL..... 249
4.3.4.2. Simulación 251
4.3.5. Unidad de comparación..... 252
4.3.5.1. Diagramas esquemáticos RTL..... 253
4.3.5.2. Simulación 254
4.3.6. Unidad booleana..... 255
4.3.6.1. Diagramas esquemáticos RTL..... 255

4.3.6.2. Simulación 258
4.3.7. Unidad de desplazamiento 258
4.3.7.1. Diagramas esquemáticos RTL 259
4.3.7.2. Simulación 260
4.3.8. Diseño general 261
4.3.8.1. Diagramas esquemáticos RTL 261
4.3.8.2. Simulación 263
4.4. Lógica de control, CTL 264
4.4.1. ROM de 64x18 265
4.4.2. Diseño general 266
4.4.2.1. Diagramas esquemáticos RTL 266
4.4.2.2. Simulación 269
4.5. Contador de programa, PC 271
4.5.1. Diagramas esquemáticos RTL 271
4.5.2. Simulación 273
4.6. Módulo TOP 274

CONCLUSIONES 279
RECOMENDACIONES 281
BIBLIOGRAFÍA 283

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Mapeo de niveles lógicos a voltaje 22
2. Dispositivo combinacional 23
3. Dispositivo combinacional compuesto 24
4. Compuertas lógicas básicas 25
5. Compuerta $\boldsymbol{OR}$ de 4 entradas 26
6. Compuerta $\boldsymbol{AND}$ de 6 entradas 27
7. Compuertas lógicas $\boldsymbol{NAND}$, $\boldsymbol{NOR}$ y $\boldsymbol{XOR}$ 28
8. Compuerta $\boldsymbol{NAND}$ de 8 entradas 29
9. Retardo de propagación 31
10. $\boldsymbol{tPD}$ en un sistema combinacional 32
11. Retardo de contaminación 33
12. $\boldsymbol{tCD}$ en un sistema combinacional 34
13. Síntesis de suma de productos 38
14. Diagrama lógico de un multiplexor 2 a 1 40
15. Circuito interno de un multiplexor 2 a 1 41
16. Circuito interno de un multiplexor 4 a 2 42
17. Diagrama lógico de un multiplexor 4 a 2 43
18. Multiplexor compuesto 44
19. Diagrama lógico de un decodificador 45
20. Circuito interno de un decodificador 46
21. Representación alterna de compuertas lógicas 48
22. Circuito interno de una ROM 49
23. Información binaria programada en una ROM 50

24. Estructura básica de un circuito secuencial 52
25. Señal de reloj 53
26. Circuito interno de un latch D 54
27. Símbolo lógico de un latch D 55
28. Disciplina dinámica para un latch 58
29. Diagrama lógico de un flip-flop D 60
30. Circuito interno de un flip-flop D 61
31. Diagrama de tiempos de un flip-flop D 62
32. Ejemplo de un STD 65
33. STD de una máquina de Moore 66
34. STD de una máquina de Mealy 67
35. Implementación de una FSM en hardware 68
36. STD de una máquina con 5 estados 69
37. Compuerta $\boldsymbol{AND}$ de 4 entradas en cascada 72
38. Compuerta $\boldsymbol{AND}$ de 4 entradas en paralelo 72
39. Circuito secuencial con 1 flip-flop 73
40. Circuito secuencial con 3 flip-flops 74
41. Etapas en un circuito combinacional 76
42. Diagrama de tiempos de un circuito combinacional 77
43. Pipeline de 2 etapas 78
44. Arreglo $\boldsymbol{AND}$ no programado 81
45. Arreglo $\boldsymbol{OR}$ no programado 82
46. Esquema básico de una PROM 83
47. Esquema básico de un PLA 84
48. Esquema básico de un PAL 85
49. Estructura básica de una FPGA 87
50. LUT de 3 entradas 88
51. Estructura básica de un CLB 89
52. Estructura de interconexión de una FPGA 90

53. Entidad de un sumador completo 93
54. Declaración de una entidad en VHDL 95
55. Arquitectura de un comparador de 2 bits 97
56. Tarjeta de desarrollo Nexys 2 100
57. Diagrama de bloques de la Nexys 2 101
58. Diagrama de un datapath controlado por una FSM 106
59. FSM de alto nivel para el factorial de $\boldsymbol{M}$ 107
60. Datapath de propósito específico 108
61. Diagrama de una FSM de control 109
62. Diagrama de un datapath programable 111
63. Modelo Von Neumann 113
64. Diagrama de CPU y su conexión con una memoria 118
65. Unidad aritmética lógica o ALU 119
66. Módulo semisumador 156
67. Abstracción del módulo semisumador 157
68. Módulo sumador completo 158
69. Abstracción del módulo sumador completo 159
70. Bloque de registros de múltiples puertos, REGFILE 161
71. Unidad aritmética lógica, ALU 162
72. Lógica de control, CTL 163
73. Contador de programa, PC 164
74. I/Os para la etapa de búsqueda 166
75. Búsqueda de la instrucción 166
76. Decodificación del OPCODE 167
77. Obtención de los operandos $\boldsymbol{Ra}$, $\boldsymbol{Rb}$ y $\boldsymbol{Rc}$ 168
78. Ejecución de la operación 169
79. Escritura en el registro de destino 170
80. Extensión para instrucciones con constante 171
81. Puertos adicionales para accesos a memoria 174

82. Hardware para instrucción LD 175
83. Hardware para instrucción ST 178
84. Hardware para la instrucción JMP 180
85. Hardware para instrucciones BEQ/BNE 183
86. Hardware para la instrucción LDR 184
87. Hardware para excepciones 187
88. Memoria scratchpad de 3 puertos 190
89. Configuración de puertos de la memoria scratchpad 191
90. Conexión de multiplexores de control 192
91. Circuito interno del bloque de registros 193
92. Diseño general del circuito interno de la ALU 196
93. Conexión del sumador de 32 bits 198
94. Compuerta $\boldsymbol{NOR}$ de 32 bits para la unidad aritmética 199
95. Circuito interno de la unidad aritmética 200
96. Circuito interno del sumador de 32 bits 203
97. Circuito sumador completo usando suma de productos 206
98. Circuito interno de la unidad de comparación 207
99. Circuito interno de la unidad booleana 209
100. Circuito interno de la unidad de desplazamiento 212
101. ROM de 64×18 bits para el módulo CTL 214
102. Hardware adicional para la señal $\boldsymbol{pcsel}[\mathbf{2}:\mathbf{0}]$ 221
103. Circuito interno de la lógica de control 222
104. Registro de 32 bits del módulo PC 224
105. Hardware para el cálculo de $\boldsymbol{PC}+\mathbf{4}$ 225
106. Hardware para el cálculo de $\boldsymbol{PC}+\mathbf{4}+\mathbf{4}\cdot\boldsymbol{SEXT}(\boldsymbol{constante})$ 226
107. Multiplexor de 5 entradas controlado por $\boldsymbol{pcsel}[\mathbf{2}:\mathbf{0}]$ 227
108. Entradas restantes del multiplexor PCSEL 229
109. Circuito interno del módulo PC 230
110. Diagrama RTL de alto nivel RAM_REGS 234

111. Diagrama RTL del módulo RAM_REGS 235
112. Diagrama RTL de alto nivel REGFILE..... 236
113. Diagrama RTL parcial del módulo REGFILE..... 237
114. Diagrama RTL del módulo REGFILE 238
115. Simulación del módulo REGFILE..... 239
116. Diagrama RTL de alto nivel FA 241
117. Diagrama RTL parcial del módulo FA 242
118. Diagrama RTL del módulo FA..... 243
119. Simulación del módulo FA..... 244
120. Diagrama RTL de alto nivel ADDR_32..... 245
121. Diagrama RTL del módulo ADDR_32 246
122. Diagrama RTL de alto nivel NOR_32..... 247
123. Diagrama RTL del módulo NOR_32..... 248
124. Diagrama RTL de alto nivel ARITH 249
125. Diagrama superior del módulo ARITH..... 250
126. Diagrama inferior del módulo ARITH..... 251
127. Simulación del módulo ARITH 252
128. Diagrama RTL de alto nivel CMP..... 253
129. Diagrama RTL del módulo CMP 254
130. Simulación del módulo CMP 255
131. Diagrama RTL de alto nivel BOOL..... 256
132. Diagrama RTL parcial del módulo BOOL..... 257
133. Simulación del módulo BOOL 258
134. Diagrama de alto nivel SHIFT 259
135. Diagrama RTL parcial del módulo SHIFT..... 260
136. Simulación del módulo SHIFT..... 261
137. Diagrama RTL de alto nivel ALU..... 262
138. Diagrama RTL del módulo ALU 263
139. Simulación del módulo ALU 264

140. Diagrama RTL de alto nivel ROM 265
141. Diagrama RTL del módulo ROM 266
142. Diagrama RTL de alto nivel CTL 267
143. Diagrama RTL superior del módulo CTL 268
144. Diagrama RTL inferior del módulo CTL 269
145. Simulación del módulo CTL 270
146. Diagrama RTL de alto nivel PC 271
147. Diagrama RTL superior del módulo PC 272
148. Diagrama RTL inferior del módulo PC 273
149. Simulación del módulo PC 274
150. Diagrama RTL de alto nivel del módulo TOP 275
151. Diagrama RTL superior del módulo TOP 276
152. Diagrama RTL inferior del módulo TOP 277

TABLAS

I. Números en binario y hexadecimal 9
II. Tabla de verdad para la función $\boldsymbol{AND}$ 16
III. Tabla de verdad para la función $\boldsymbol{OR}$ 17
IV. Tabla de verdad para la función $\boldsymbol{NOT}$ 17
V. Tabla de verdad para la función $\boldsymbol{F1}$ 18
VI. Funciones $\boldsymbol{NAND}$, $\boldsymbol{NOR}$ Y $\boldsymbol{XOR}$ 27
VII. Especificación funcional de $\boldsymbol{Z}$ 35
VIII. Tabla de verdad de un multiplexor de 2 entradas 40
IX. Tabla de verdad con 3 entradas y 3 salidas 50
X. Tabla de verdad de un Latch D 54
XI. Tabla de verdad de un latch D permisivo 57
XII. Tabla de verdad de un STD 70
XIII. Diagrama de un pipeline de 2 etapas 79
XIV. Atributos de la FPGA Spartan 3E-1200 103
XV. Tabla de estados de la FSM de control 109
XVI. Información en una memoria de 32 bits 114
XVII. Información de una memoria en hexadecimal 115
XVIII. Memoria direccionable por bytes 117
XIX. Ejemplo de una instrucción de 32 bits 121
XX. Mnemónicos de algunos opcodes 122
XXI. Algunos operadores utilizados en expresiones RTL 128
XXII. Estado del procesador 130
XXIII. Formato de una instrucción sin constante 132
XXIV. Formato de una instrucción con constante 132
XXV. Lista de opcodes y mnemónicos 133
XXVI. Formato de instrucción ADD 134
XXVII. Formato de instrucción ADDC 134

XXVIII. Formato de instrucción AND 135
XXIX. Formato de instrucción ANDC 135
XXX. Formato de instrucción BEQ 136
XXXI. Formato de instrucción BNE 137
XXXII. Formato de instrucción CMPEQ 138
XXXIII. Formato de instrucción CMPEQC 139
XXXIV. Formato de instrucción CMPLE 139
XXXV. Formato de instrucción CMPLEC 140
XXXVI. Formato de instrucción CMPLT 140
XXXVII. Formato de instrucción CMPLTC 141
XXXVIII. Formato de instrucción JMP 142
XXXIX. Formato de instrucción LD 142
XL. Formato de instrucción LDR 143
XLI. Formato de instrucción OR 144
XLII. Formato de instrucción ORC 144
XLIII. Formato de instrucción SHL 145
XLIV. Formato de instrucción SHLC 145
XLV. Formato de instrucción SHR 146
XLVI. Formato de instrucción SHRC 147
XLVII. Formato de instrucción SRA 147
XLVIII. Formato de instrucción SRAC 148
XLIX. Formato de instrucción ST 148
L. Formato de instrucción SUB 149
LI. Formato de instrucción SUBC 149
LII. Formato de instrucción XOR 150
LIII. Formato de instrucción XORC 150
LIV. Formato de instrucción XNOR 151
LV. Formato de instrucción XNORC 151
LVI. Registros reservados 154

LVII. Señales de control para instrucción tipo ALU 172
LVIII. Señales de control para instrucción tipo ALUC 173
LIX. Señales de control para instrucción LD 176
LX. Señales de control para la instrucción ST 177
LXI. Señales de control para la instrucción JMP 181
LXII. Señales de control para instrucciones BEQ/BNE 182
LXIII. Señales de control para instrucción LDR 185
LXIV. Señales de control para manejo de excepciones 189
LXV. Codificación de operaciones mediante la señal $\boldsymbol{FN}[\mathbf{5}:\mathbf{0}]$ 194
LXVI. Tabla de verdad para operaciones booleanas 195
LXVII. Condiciones para que exista sobreflujo 202
LXVIII. Especificación funcional del sumador completo 204
LXIX. Codificación de operaciones de comparación 205
LXX. Codificación de desplazamientos 210
LXXI. Codificación de las señales de control del módulo CTL 215
LXXII. Resumen de valores de señales de control 216
LXXIII. Valores de la señal $\boldsymbol{alufn}[\mathbf{5}:\mathbf{0}]$ 217
LXXIV. Contenidos de la ROM de control 218
LXXV. Especificación funcional para el bit $\boldsymbol{next\_pc}[\mathbf{31}]$ 228

LISTA DE SÍMBOLOS

Símbolo	Significado
ST	Almacenamiento en memoria
LDR	Almacenamiento relativo en memoria
T	Ancho de dirección
W	Ancho de palabra
LD	Carga desde memoria
CMPEQ	Comparación igualdad
CMPEQC	Comparación igualdad con constante
CMPLE	Comparación menor o igual que
CMPLEC	Comparación menor o igual que con constante
CMPLT	Comparación menor que
SHR	Desplazamiento a la derecha
SHRC	Desplazamiento a la derecha con constante
SHL	Desplazamiento a la izquierda
SHLC	Desplazamiento a la izquierda con constante
SRA	Desplazamiento aritmético
SRAC	Desplazamiento aritmético con constante
I/O	Entrada y/o salida
GB	Gigabyte
L	Latencia
Mb/s	Megabit por segundo
MB	Megabyte
MHz	Megahertz
ms	Milisegundo

ns	Nanosegundo
ADD	Operación de suma
ADDC	Operación de suma con constante
SUB	Operación de sustracción
SUBC	Operación de sustracción con constante
AND	Operación lógica AND
ANDC	Operación lógica AND con constante
OR	Operación lógica OR
ORC	Operación lógica OR con constante
XNOR	Operación lógica $XNOR$
XNORC	Operación lógica $XNOR$ con constante
XOR	Operación lógica XOR
XORC	Operación lógica XOR con constante
~	Operador complemento a uno
≫	Operador de desplazamiento lógico a la derecha
≪	Operador de desplazamiento lógico a la izquierda
&	Operador lógico AND
|	Operador lógico OR
^	Operador lógico XOR
$\boldsymbol{T_{clk}}$	Período de reloj
$\boldsymbol{R}$	Rendimiento
BEQ	Salto condicional cero
BNE	Salto condicional diferente de cero
JMP	Salto incondicional
s	Segundo
V	Voltio

GLOSARIO

Álgebra Booleana Sistema matemático aplicado en el diseño de sistemas digitales, para la resolución de preposiciones lógicas por medio de variables que toman el valor de 0 o 1 para representar los valores falso y verdadero respectivamente.

Algoritmo Secuencia finita de pasos o instrucciones para resolver un problema de manera sistemática.

ALU Siglas en inglés para *arithmetic logic unit*.

ARM Siglas en inglés para *acorn RISC machine*, familia de arquitecturas RISC para procesadores de computadoras.

ASIC Siglas en inglés para *application-specific integrated circuit*, circuito integrado personalizado y optimizado para una aplicación específica.

Bit Unidad básica de información en la teoría de la computación y comunicaciones digitales. Representa un estado lógico con uno de dos posibles valores, 0 y 1.

Bus	Sistema de comunicación que permite la transferencia de datos entre componentes dentro de una computadora, o entre computadoras.
Byte	Unidad de información, regularmente conformada por la agrupación de ocho bits.
CLA	Siglas en inglés para *carry-lookahead adder*, circuito sumador de alta velocidad. Utiliza un algoritmo complejo que reduce el tiempo de respuesta requerido para determinar los bits de acarreo.
CSA	Siglas en inglés para *carry-select adder*, circuito sumador simple de alta velocidad. Utiliza una mayor cantidad de hardware para reducir el tiempo de respuesta requerido para determinar los bits de acarreo.
Clock skew	Fenómeno presente en circuitos digitales síncronos en el que la misma señal de reloj llega a diferentes componentes en diferentes instantes de tiempo.
DCM	Siglas en inglés para *digital clock manager*, componente electrónico contenido dentro de algunas FPGAs. Se encarga de manipular las señales de reloj y evitar el clock skew.
DRAM	Siglas en inglés para *dynamic random-access memory*, un tipo de RAM que almacena cada bit de

	información en una celda de memoria conformada por un transistor y un capacitor.
EDVAC	Una de las primeras computadoras electrónicas binarias. Su diseño se convirtió en el estándar de arquitectura para la mayoría de computadoras modernas.
ENIAC	La primera computadora electrónica digital de propósito general, capaz de resolver una gran cantidad de problemas numéricos.
Flanco de bajada	Cambio de estado lógico alto a estado lógico bajo en una señal digital.
Flanco de subida	Cambio de estado lógico bajo a estado lógico alto en una señal digital.
FSM	Siglas en inglés para *finite state machine*.
HA	Siglas en inglés para *half adder.*
Hardware	Conjunto de componentes eléctricos, electrónicos y electromecánicos que conforman la parte tangible o física de una computadora.
HDL	Siglas en inglés para *hardware description language*, lenguaje de especificación para la descripción de la estructura y comportamiento de circuitos electrónicos.

ISA	Siglas en inglés para *instruction set architecture*, especificación funcional detallada de las operaciones y mecanismos de almacenamiento de un procesador que actúa como una interfaz entre diseñadores de hardware y programadores.
ISE	Siglas en inglés para *integrated synthesis enviroment*, una herramienta de software producida por Xilinx para la síntesis y análisis de diseños HDL.
ISim Simulator	Herramienta de software contenida dentro de ISE Design Suite que permite simular el comportamiento de diseños destinados a FPGAs de Xilinx.
Jade Circuit Simulator	Herramienta de software producida por MIT para la simulación de diseños de circuitos analógicos y digitales.
LED	Siglas en inglés para *light-emitting diode*, dispositivo semiconductor que emite luz visible cuando fluye corriente eléctrica a través de sus terminales.
LSB	Siglas en inglés para *least significant bit*, el bit que acompaña a la potencia más baja de la base en una cadena binaria.
MIPS	Siglas en inglés para *microprocessor without interlocked pipelined stages*, una arquitectura de conjunto de instrucciones RISC.

MIT	Siglas en inglés para *Massachusetts Institute of Technology*, instituto privado de investigación ubicado en Cambridge, Massachussets.
Modo usuario	Modo de ejecución del procesador en el que usualmente corren las aplicaciones del usuario.
Modo supervisor	Modo de ejecución en el que usualmente corre el sistema operativo. Se permite la ejecución de todas las instrucciones, incluyendo las privilegiadas.
MSB	Siglas en inglés para *most significant bit*, el bit que acompaña a la potencia más alta de la base en una cadena binaria.
Oscilador	Circuito capaz de convertir la energía de corriente continua en corriente alterna de una determinada frecuencia.
RAM	Siglas en inglés para *random-access memory*, un tipo de memoria volátil cuya información puede ser leída o modificada en cualquier orden.
ROM	Siglas en inglés para *read-only memory,* un tipo de memoria no volátil. La información almacenada no puede ser modificada después de su producción.
SDRAM	Siglas en inglés para *synchronous dynamic random-access memory*, un tipo de DRAM en donde la

	operación de su interfaz externa se coordina mediante una señal externa de reloj.
Software	Secuencia ordenada de instrucciones almacenadas en memoria que define las operaciones que el procesador debe ejecutar.
Spartan 3E	Familia de FPGAs de Xilinx, su diseño se encuentra optimizado para satisfacer las necesidades de aplicaciones que requieren una alta densidad de bloques lógicos.
SRAM	Siglas en inglés para *static random access memory*, un tipo de RAM basada únicamente en dispositivos semiconductores.
STD	Siglas en inglés para *state transition diagram*.
Testbench	Modelo utilizado para evaluar y verificar el correcto funcionamiento de un modelo de hardware. Genera estímulos y los aplica a la entidad bajo evaluación y recolecta los valores de salida para compararlos con los valores esperados.
Virtex-5	Familia de productos desarrollados por Xilinx, destinados para aplicaciones de lógica intensiva.
WERF	Siglas en inglés para *write enable register file*.

Xilinx	Compañía tecnológica estadounidense, uno de los principales fabricantes y distribuidores de dispositivos lógicos programables en el mundo.

RESUMEN

En el presente trabajo de graduación se desarrolla el diseño de un procesador de 32 bits, capaz de implementar la funcionalidad de un conjunto de instrucciones simple y representativo de una arquitectura RISC.

En el primer capítulo se desarrollan los fundamentos teóricos relacionados con el diseño de circuitos combinacionales, circuitos secuenciales síncronos y dispositivos lógicos programables, necesarios para la comprensión del diseño de cada uno de los bloques que conforman el diseño del procesador.

En el segundo capítulo se utilizan los conceptos del diseño de circuitos digitales para la descripción de un sistema digital complejo: la computadora de propósito general. El capítulo concluye con la especificación detallada del conjunto de instrucciones que el procesador debe ser capaz de ejecutar, el ISA Beta.

En el tercer capítulo se presenta el diseño del hardware necesario para implementar la funcionalidad del ISA Beta, utilizando un enfoque de diseño basado en diferentes niveles de abstracción, facilitando así, el proceso de comprensión para el lector.

En el cuarto capítulo se presentan los resultados de la implementación y simulación del diseño utilizando el lenguaje de especificación VHDL con el entorno de desarrollo ISE Design Suite 14.6, para la FPGA Spartan 3E-1200 de Xilinx.

OBJETIVOS

General

Implementar el diseño de un procesador RISC de 32 bits y de código abierto en un dispositivo de lógica programable utilizando el lenguaje de especificación VHDL.

Específicos

1. Especificar el conjunto de instrucciones reducido que el procesador será capaz de ejecutar.

2. Realizar el diseño del procesador en función de su conjunto de instrucciones.

3. Describir la funcionalidad del procesador utilizando el lenguaje de especificación VHDL en un entorno de desarrollo.

4. Sintetizar el diseño del procesador en una tarjeta de desarrollo.

5. Verificar el funcionamiento del hardware aprovechando las herramientas de simulación que ofrece el entorno de desarrollo.

INTRODUCCIÓN

En la actualidad existe una gran cantidad de personas que utilizan varios dispositivos con amplia capacidad computacional, los cuales permiten realizar tareas que, hace algunas pocas décadas, resultaban difíciles de imaginar. Muchas veces, inmersas en lo cotidiano del uso de estos dispositivos, resulta inevitable para algunas personas intentar indagar acerca de qué misteriosos componentes se encuentran embebidos dentro de estos dispositivos, así como la manera en que se encuentran interconectados entre sí.

De manera superficial, es posible describir a estos dispositivos en términos de la interconexión de un procesador, CPU, encargado de ejecutar una secuencia de instrucciones específica; una unidad de memoria, encargada de almacenar las instrucciones y los datos; y algunos dispositivos de entrada y salida, que proveen una interfaz de comunicación con el mundo exterior. Dentro de este conjunto de componentes, la mayor cantidad de preguntas giran en torno a qué tipo de instrucciones ejecuta el procesador y cómo realiza esta función.

El presente trabajo brinda las bases teóricas necesarias para guiar al lector a través del diseño y la síntesis de un procesador RISC de 32 bits en un dispositivo de lógica programable, capaz de implementar la funcionalidad del conjunto de instrucciones Beta, o ISA Beta, creado y utilizado por el MIT con propósitos didácticos, dada su simplicidad y representatividad de las arquitecturas modernas. Una de las principales características de una arquitectura RISC es el uso de un conjunto de instrucciones de tamaño fijo y en un número reducido de formatos, lo cual permite que el hardware necesario para decodificar y ejecutar las instrucciones sea relativamente sencillo.

El proceso de diseño utiliza un enfoque basado en la creación de varios niveles de abstracción, a partir de componentes simples de menor nivel. El diseño se descompone en varias etapas: especificación del conjunto de instrucciones y requerimientos de hardware, diseño del hardware, implementación utilizando VHDL y simulación dentro de un entorno de desarrollo.

Bajo la idea de hardware libre, es conveniente y necesario implementar un procesador RISC, con una arquitectura sencilla que permita exponer de una manera más fácil e intuitiva el funcionamiento de sus bloques lógicos fundamentales al lector, facilitando así, el proceso de innovación. Además, el uso del lenguaje VHDL dota al diseño de universalidad y versatilidad.

Las especificaciones del procesador, los diagramas esquemáticos y código VHDL son de acceso público y gratuito; por lo tanto, el lector puede estudiar, modificar, y utilizar el contenido libremente con cualquier fin y redistribuirlo con cambios y/o mejoras o sin ellas.

1. CIRCUITOS DIGITALES

Para realizar el diseño de un sistema digital complejo como lo supone un procesador RISC de 32 bits, es necesario comprender los fundamentos teóricos para el diseño de circuitos digitales de menor complejidad, los cuales conforman los bloques principales de diseño para circuitos digitales más complejos. El objetivo principal del capítulo es brindar al lector las principales herramientas de análisis y diseño para los capítulos posteriores, así como también mostrar las características principales de la tarjeta de desarrollo en la cual se implementará el diseño del microprocesador.

1.1. Sistemas de notación posicional

En general, un número N expresado en un sistema de notación posicional de base r, consta de una cadena de dígitos expuestos de la siguiente forma:

$$N = (a_n a_{n-2} \cdots a_1 a_0 \, . \, a_{-1} \cdots a_{-m})_r$$

Un sistema de notación posicional es un sistema de numeración en el que cada dígito a multiplica a una potencia determinada de un número r, o la base del sistema, de la siguiente manera:

$$N = a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + \cdots + a_1 \cdot r^1 + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + \cdots + a_{-m} \cdot r^{-m}$$

Se observan dos características importantes:

- El dígito a_k multiplica a la potencia de r equivalente a k, en otras palabras, la potencia de la base depende de la posición del dígito en la cadena, de aquí el nombre del sistema.
- En un sistema de notación posicional de base r, existe una cantidad finita de r símbolos; el valor de los dígitos enteros a_k varía entre 0 y $r-1$.

Es posible representar el número N de una manera más compacta, si se divide en sus partes entera y fraccionaria:

$$N = \sum_{k=0}^{n-1} a_k \cdot r^k + \sum_{j=1}^{m} a_{-j} \cdot r^{-j}$$

Por razones prácticas y didácticas, el presente trabajo se centra únicamente en la representación de números enteros utilizando este esquema. Por lo tanto, la representación que se utilizará para un número entero Z de n dígitos en un sistema de base r es la siguiente:

$$Z = \sum_{k=0}^{n-1} a_k \cdot r^k$$

En sistemas digitales se utilizan sistemas de notación posicional para la representación de la información, entre dichos sistemas se encuentran el sistema decimal, binario y hexadecimal; estos sistemas se explican en las siguientes subsecciones.

1.1.1. Sistema decimal

El sistema de numeración decimal es un sistema de notación posicional que utiliza como base $r = 10$. En otras palabras, cualquier entero positivo Z de n dígitos se puede representar como

$$Z = (a_{n-1}a_{n-2} \cdots a_1 a_0)_{10} = \sum_{k=0}^{n-1} a_k \cdot 10^k$$

En este sistema, el listado de dígitos a_k que multiplican a las potencias de 10 (r) comprende los dígitos del 0 al 9 ($1 - r$). Algunos ejemplos de números representados en este sistema son:

$$(8502)_{10}$$
$$(45988)_{10}$$
$$(22)_{10}$$

Generalmente se omite el subíndice que denota la base del sistema por motivos prácticos. En este caso se asume que el número está representado en el sistema decimal.

1.1.2. Sistema binario

En este sistema se utiliza como base $r = 2$. Cualquier entero positivo Z de n dígitos se puede representar como

$$Z = (a_{n-1}a_{n-2} \cdots a_1 a_0)_2 = \sum_{k=0}^{n-1} a_k \cdot 2^k$$

En este sistema, el listado de dígitos a_k que multiplican a las potencias de la base 2 son únicamente los dígitos 1 y 0, los cuales se conocen como dígitos binarios o simplemente bits. Algunos ejemplos de números representados en este sistema son los siguientes:

$$(100010)_2$$
$$(101111)_2$$
$$(1011)_2$$

En ciencias de la computación y diseño digital existe otro formato para la representación de números binarios, en el cual se antepone un $0b$ al número para indicar que se está trabajando con un sistema binario. Se muestran los números mostrados anteriormente bajo este nuevo formato:

$$0b100010$$
$$0b101111$$
$$0b1011$$

Debido a que el presente documento se centra en el diseño de un sistema digital complejo, de ahora en adelante, se utilizará la última representación como principal formato.

1.1.3. Sistema hexadecimal

En el sistema hexadecimal se utiliza como base $r = 16$. Cualquier entero positivo Z de n dígitos, de la misma manera que en las subsecciones anteriores, se puede representar como:

$$Z = (a_{n-1}a_{n-2} \cdots a_1 a_0)_{16} = \sum_{k=0}^{n-1} a_k \cdot 16^k$$

Para este sistema, el número de dígitos a_k a utilizar es de 15; por esta razón, el conjunto de dígitos comprende 15 dígitos alfanuméricos: los 10 dígitos correspondientes al sistema decimal y 5 dígitos adicionales. Los símbolos A, B, C, D, E representan los valores 11, 12, 13, 14 y 15 respectivamente. Algunos ejemplos de números enteros utilizando este sistema de numeración son los siguientes:

$$(A8F1E)_{16}$$
$$(EFFA342)_{16}$$
$$(568BAD)_{16}$$

El formato de representación que se emplea comúnmente en ciencias de la computación consiste en un $0x$ que se antepone al número para indicar que está representado en notación hexadecimal, utilizando este formato los números anteriores.

$$0xA8F1E$$
$$0\text{x}EFFA342$$
$$0\text{x}568BAD$$

Por las mismas razones expuestas en la subsección anterior, de ahora en adelante, se utilizará la última representación como principal formato.

1.1.4. Conversión entre sistemas

En un sistema digital, un sistema de numeración dado puede ser más conveniente que otro para representar o procesar la información. La elección de un sistema en particular depende la acción que se desee realizar; por ejemplo, los sistemas decimal y hexadecimal son regularmente utilizados para la representación de la información mientras que el sistema binario se utiliza para representar y/o procesar la información en circuitos digitales.

Al existir varios sistemas de numeración para representar la información, es conveniente definir métodos sistemáticos para convertir una representación de un número Z en un sistema de base r a otro de base R y viceversa. A continuación, se detallan los diferentes métodos de conversión entre los sistemas expuestos con anterioridad: decimal, hexadecimal y binario.

1.1.4.1. Conversiones a decimal

El método para convertir números de base r al sistema decimal es bastante sencillo y aplica tanto para números hexadecimales como binarios. El procedimiento consiste en expandir el número como una serie de potencias de la base r, tal y como lo indica la fórmula para un sistema de notación posicional. Considérese el ejemplo de convertir el número $Z = 0b10110$ a decimal, para esto se debe representar a Z como una composición lineal de potencias de la base 2, es decir:

$$Z = \sum_{k=0}^{5} a_k \cdot 2^k = a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + a_4 \cdot 2^4$$

en donde,

$$a_0 = 0, \qquad a_1 = 1, \qquad a_2 = 1, \qquad a_3 = 0, \qquad a_4 = 1$$

Al simplificar la expansión se obtiene el siguiente resultado

$$Z = (0,1) + (1,2) + (1,4) + (0,8) + (1,16) = 2 + 4 + 16$$
$$Z = 22$$

Para convertir números hexadecimales a decimal se sigue el mismo procedimiento, supóngase que se desea convertir el número $Z = 0xAF85$ a decimal, representando el número como una composición de potencias de $r = 16$ se obtiene

$$Z = \sum_{k=0}^{3} a_k \cdot 16^k = a_0 \cdot 16^0 + a_1 \cdot 16^1 + a_2 \cdot 16^2 + a_3 \cdot 16^3$$

Los valores numéricos de los dígitos A y F son 10 y 15 respectivamente, sustituyendo los valores de los coeficientes a_k y simplificando se obtiene el siguiente resultado

$$Z = (5 \cdot 16) + (8 \cdot 16) + (10 \cdot 256) + (15 \cdot 4096)$$
$$Z = 64208$$

1.1.4.2. Conversiones a hexadecimal

Para convertir un número decimal al sistema hexadecimal se divide el número y todos sus cocientes sucesivos entre la base $r = 16$, los residuos representan los dígitos hexadecimales. Para ilustrar este procedimiento, se pretende convertir el número $Z = 542$ a hexadecimal, para esto se realiza el siguiente conjunto de cocientes sucesivos:

$$542 \div 16 = 33 + \mathbf{14}/16$$
$$33 \div 16 = 2 + \mathbf{1}/16$$
$$2 \div 16 = 0 + \mathbf{2}/16$$

Los residuos de los cocientes sucesivos se encuentran resaltados en negrita, estos valores son equivalentes los dígitos a_0, a_1 y a_2, en ese orden. Sustituyendo $A = 14$, el número Z representado en el sistema hexadecimal es equivalente a

$$Z = 0x21E$$

Ahora, considérese el caso de cualquier número entero representado en el sistema binario, el método de conversión es más sencillo e intuitivo que el utilizado para convertir números decimales a hexadecimales y es uno de los más utilizados en sistemas digitales. El método asigna una correspondencia única entre cada dígito hexadecimal y una cadena de 4 bits, esto quiere decir que para cada posible cadena de 4 bits existe un dígito hexadecimal único que le corresponde.

Para demostrar que cada cadena de bits puede ser representada únicamente por un digito hexadecimal, obsérvese que el número de cadenas diferentes que se pueden obtener con 4 bits es de $2^4 = 16$ o bien, igual al número de dígitos en el sistema hexadecimal. Se utiliza la correspondencia presentada en la tabla I, donde los elementos se encuentran en orden ascendente.

Tabla I. **Números en binario y hexadecimal**

Binario $(r = 2)$	**Hexadecimal** $(r = 16)$
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Fuente: elaboración propia.

Para realizar la conversión de $Z = 0b1010010011010011$, se procede a dividir el número en cadenas de 4 bits y se utiliza la tabla I para sustituir las cadenas con los dígitos hexadecimales correspondientes:

$$Z = 0b1010 \quad 0100 \quad 1101 \quad 0011$$

$$Z = 0xA4D3$$

Este método es muy útil para analizar sistemas computacionales ya que permite representar largas cadenas de bits con una cantidad menor de dígitos, debido a que a cada dígito hexadecimal le corresponde una cadena única de 4 bits.

1.1.4.3. Conversiones a binario

Para convertir números decimales al sistema binario se realiza el mismo procedimiento que se realizó para convertir números decimales al sistema hexadecimal. se divide el número y todos sus cocientes sucesivos entre la base $r = 2$, los residuos representan los dígitos binarios. Por ejemplo, para representar el número 80:

$$80 \div 2 = 40 + \mathbf{0}$$
$$40 \div 2 = 20 + \mathbf{0}$$
$$20 \div 2 = 10 + \mathbf{0}$$
$$10 \div 2 = 5 + \mathbf{0}$$
$$5 \div 2 = 2 + \mathbf{1}/2$$
$$2 \div 2 = 1 + \mathbf{0}$$
$$1 \div 2 = 0 + \mathbf{1}/2$$

Al utilizar los valores de los residuos como los dígitos del número en binario, se obtiene $Z = 0b1010000$.

El proceso para convertir números hexadecimales es igual de sencillo que su homólogo, la conversión de binario a hexadecimal. Dado un número en hexadecimal, cada dígito se expande en cadenas de 4 bits utilizando la tabla I como referencia. Considerar el ejemplo de convertir $Z = 0xF45C$,

$$Z = 0xF45C$$
$$Z = 0b111 \quad 0100 \quad 0101 \quad 1100$$

1.2. Codificación de números enteros

Como ya se mencionó en la subsección 1.1.4, para representar y procesar la información en sistemas digitales se utiliza el sistema binario. Es importante reconocer que no sólo se pueden representar números enteros sin signo, sino también enteros con signo utilizando el sistema binario.

1.2.1. Enteros sin signo

Para codificar enteros sin signo se sigue la representación de la subsección 1.1.2, en donde a cada bit se le asigna un peso o una potencia específica de la base 2. El valor de un número Z de n bits, codificado de esta manera, está dado por:

$$Z = \sum_{k=0}^{n-1} a_k \cdot 2^k$$

Con este esquema de codificación es posible representar el rango de números decimales enteros que se encuentran en el intervalo de 0 a $2^n - 1$.

1.2.2. Enteros con signo

En circuitos digitales que realizan operaciones aritméticas, es posible realizar tanto sumas y restas utilizando un solo circuito sumador si se escoge un esquema de codificación apropiado. Este esquema de codificación se conoce regularmente como representación en complemento a 2 y permite simplificar el diseño del hardware para la unidad aritmética del procesador.

En este esquema de codificación, al bit más significativo, la potencia más alta de la base, se le asigna un peso negativo así, un número entero con signo V de n bits se representa mediante la expresión

$$V = -2^{n-1} + \sum_{k=0}^{n-2} a_k \cdot 2^k$$

Algunos ejemplos de números signados de 8 bits en los que se utiliza este esquema se muestran a continuación:

$$0b10001011 = -2^7 + 2^3 + 2^1 + 2^0 = -128 + 8 + 2 + 1 = -117$$
$$0b11100101 = -2^7 + 2^6 + 2^5 + 2^1 + 2^0 = -128 + 64 + 32 + 2 + 1 = -29$$
$$0b00111001 = 2^5 + 2^4 + 2^3 + 2^0 = 32 + 16 + 8 + 1 = 57$$

Es fácil ver que para los números negativos el bit más significativo siempre es 1 mientras que en el caso de los números positivos es 0. El número más negativo que es posible representar se obtiene al eliminar todos los bits con peso positivo de la sumatoria y el número más positivo se obtiene al eliminar el bit más significativo y asignar el valor de 1 a todos los coeficientes a_k dentro de la sumatoria, entonces los números mínimo $V_{mín}$ y máximo $V_{máx}$ son

$$V_{mín} = -2^{n-1} = 0b10 \cdots 0000$$
$$V_{máx} = 2^{n-1} - 1 = 0b01 \cdots 1111$$

Al sumar $V_{mín}$ y $V_{máx}$ se obtiene la representación para el número -1

$$V_{máx} + V_{mín} = 2^{n-1} - 1 - 2^{n-1} = -1$$
$$-1 = 0b11 \cdots 1111$$

El número cero posee una representación única, una característica que hace a este esquema más eficiente ante aquellos en los que existen múltiples representaciones para dicho número. Si se suman los números -1 y 1 se obtiene la representación para el número 0, en la que todos los bits son iguales a cero:

$$-1 + 1 = 0b11 \cdots 1111 + 0b00 \cdots 0001$$
$$0 = 0b00 \cdots 0000$$

Como ya se mencionó previamente, el uso de números en complemento a 2 también permite utilizar el mismo hardware para sumar y restar números enteros. La operación $A - B$ en complemento a 2 puede reescribirse como $A + (-B)$, únicamente es necesario encontrar el negativo del número B y realizar la operación de adición. Para encontrar la representación de $-B$ se escribe

$$B + (-B) = 0 = 1 + (-1)$$
$$-B = 1 + (-1) - B$$

reordenando términos se obtiene:

$$-B = (-1 - B) + 1$$

Para operar $(-1 - B)$, -1 y B se representan en complemento a 2 y se realiza la resta. Se sabe que $-1 = 0b11 \cdots 1111$ y $B = 0ba_{n-1}a_{n-2} \cdots a_3a_2a_1a_0$, considerando la sustracción de dos dígitos arbitrarios de ambos números se llega al siguiente resultado

$$1 - a_k = \sim a_k$$

En la ecuación anterior, $\sim a_k$ se conoce como el complemento a uno del dígito a_k; si $a_k = 0$, entonces $\sim a_k = 1$ y viceversa. El resultado anterior implica que

$$(-1 - B) = \sim B$$

y, por lo tanto

$$-B = \sim B + 1$$

Este resultado indica que para obtener la representación de $-B$, es necesario obtener el complemento a 1 de B, es decir, obtener el complemento a uno de cada uno de los dígitos a_k, y sumar 1 al resultado. Regresando a la operación $A + (-B)$, es posible reescribirla de la siguiente manera:

$$A + (-B) = A + (\sim B + 1)$$

Como se verá más adelante, este es el tipo de enfoque bajo el cual se diseña el circuito sumador de la unidad aritmética del procesador.

1.3. Álgebra booleana y compuertas lógicas

La representación binaria posee una correspondencia natural con el álgebra booleana de dos valores, falso y verdadero, y por lo tanto muchas propiedades de varios circuitos digitales biestables, como los circuitos con únicamente dos niveles de voltaje, pueden representarse utilizando este tipo de álgebra. Considérese la siguiente ecuación booleana:

$$Z = (X)\ \ AND\ \ (Y)$$

Esta ecuación implica el siguiente enunciado lógico: Si X e Y son verdaderos, entonces Z es verdadero. La palabra AND es un operador lógico y se abrevia con el símbolo "·". Reescribiendo el enunciado anterior:

$$Z = X \cdot Y = XY$$

De la misma manera, considerar la siguiente ecuación booleana:

$$Z = (X)\ \ OR\ \ (Y)$$

La ecuación anterior implica lo siguiente: Si X o Y son verdaderos, entonces Z es verdadero. Igualmente, la palabra OR es un operador binario y se abrevia utilizando el símbolo "+". Reescribiendo el enunciado anterior:

$$Z = X + Y$$

Para definir el último operador básico del álgebra booleana, se analiza la siguiente ecuación:

$$Z = NOT\ \ (X)$$

Esta ecuación implica el siguiente enunciado: Si X es verdadero, entonces Z es falso y si X es falso, entonces Z es verdadero. En otras palabras, Z es la negación o complemento de X; el operador lógico NOT se abrevia utilizando el símbolo "~" o bien utilizando una línea en la parte superior de la variable que se desea negar. Reescribiendo la ecuación utilizando las dos diferentes abreviaciones:

$$Z = \sim X$$

$$Z = \bar{X}$$

Debido a la correspondencia existente entre los valores de la representación binaria, 0 y 1, y los valores *falso* y *verdadero*, se utilizarán los dígitos binarios como los valores de las variables y constantes en las expresiones booleanas posteriores.

Las ecuaciones booleanas descritas anteriormente también se consideran funciones booleanas debido a que existe una regla de correspondencia que asigna un único valor a la variable Z para cada posible combinación de valores de entrada X e Y. Esta idea se ilustra mejor utilizando el concepto de una tabla de verdad, una tabla de verdad enumera todas las posibles combinaciones de entrada y los valores de salida correspondientes. En las tablas II, III y IV se muestran las tablas de verdad para las funciones AND, OR y NOT respectivamente, se observa que para cada combinación de entradas existe un valor único de salida.

Tabla II. **Tabla de verdad para la función AND**

X	Y	$Z = XY$
0	0	0
0	1	0
1	0	0
1	1	1

Fuente: elaboración propia.

Tabla III. **Tabla de verdad para la función OR**

X	Y	$Z = X + Y$
0	0	0
0	1	0
1	0	0
1	1	1

Fuente: elaboración propia.

Tabla IV. **Tabla de verdad para la función NOT**

X	$Z = \bar{X}$
0	0
0	0
1	0
1	1

Fuente: elaboración propia.

1.3.1. Funciones booleanas

Una vez definidas las funciones booleanas básicas, se procede a dar una definición más consistente de una función booleana. Una función booleana consiste en una expresión que involucra variables binarias, operadores lógicos y las constantes 1 y 0; además, para cada combinación de entradas, existe un único valor de salida para la función, pudiendo ser el valor 1 o 0. A continuación se muestra un ejemplo de una función booleana Z de 4 entradas; A, B, C y D:

$$Z = A + \bar{B} + CD$$

El valor de la función es igual a 1 cuando cualquiera de las expresiones $A, \bar{B}$ o CD es igual a 1. En otras palabras, la función es igual a 1 cuando $A = 1$, $B = 0$, debido al operador complemento, o $CD = 1$.

Como se explicó al inicio de la sección, también es posible analizar el comportamiento de una función booleana utilizando una tabla de verdad, en la que se enlistan todas las posibles combinaciones para los valores de entrada y sus salidas correspondientes. Considerar el caso de la función booleana

$$F_1 = A + BC$$

En la tabla V se muestra la tabla de verdad para la función booleana F_1. En general, una tabla de verdad para una función de n entradas tendrá un total de 2^n filas debido a que ese es el número total de combinaciones posible. Se invita al lector a comprobar este hecho ya sea encontrando de manera independiente todas las posibles combinaciones de n entradas o utilizando el principio fundamental del conteo.

Tabla V. **Tabla de verdad para la función F_1**

A	B	C	F_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Fuente: elaboración propia.

Si se desea representar otra función booleana F_2 de 3 entradas, las primeras tres columnas de su tabla de verdad serán idénticas a las de la tabla de F_1. Lo que realmente permite distinguir las tablas de verdad de estas dos funciones es la última columna, en la cual se indica la regla de correspondencia entre entradas y salidas. En base a esta observación, se dice que dos funciones F_1 y F_2 son funciones distintas si difieren por lo menos en un bit de su última columna, de lo contrario se dice que las funciones son equivalentes.

La cantidad de todas las posibles funciones booleanas de 3 entradas posibles es equivalente a todas las combinaciones posibles de los bits en las 8 filas de la columna de salida de su tabla de verdad, es decir, $2^8 = 256$ posibles funciones booleanas de 3 entradas. En general, con n entradas se obtienen 2^n filas en la tabla de verdad y, por lo tanto, existe un total de 2^{2^n} posibles funciones booleanas de n entradas. Este tipo de enfoque es el que se utiliza en capítulos posteriores para diseñar la unidad booleana del procesador.

Si bien dos funciones booleanas pueden ser diferentes a primera vista, es necesario comparar sus tablas de verdad para verificar si son funciones realmente diferentes. Por este motivo, se define al conjunto de todas las posibles funciones booleanas de n entradas con base en la estructura de su tabla de verdad. En la siguiente subsección se presenta un conjunto de herramientas que permite manipular algebraicamente funciones booleanas para determinar si dos funciones booleanas son equivalentes.

1.3.2. Teoremas y propiedades básicas

En lógica digital, la circuitería está diseñada con base en funciones booleanas y, por lo tanto, mientras más simple sea una expresión particular, menor será el costo del circuito. Minimizar una expresión booleana consiste en

reducir una expresión Z a otra equivalente W, se dice que dos expresiones booleanas son equivalentes si ambas poseen la misma tabla de verdad. Por ejemplo, considerar el caso de la expresión

$$Z = A + B + 1$$

Se sabe que si al menos una de las entradas de la función OR es igual a 1, entonces el resultado de la función es igual a 1 y, por lo tanto, es posible reducir la expresión Z a la expresión equivalente

$$W = 1$$

En este caso, al minimizar la expresión se demuestra que el valor de salida de la función Z es independiente de la combinación de las entradas A y B para este caso en particular. A continuación, se muestra un conjunto de teoremas y propiedades útiles que se pueden utilizar para minimizar funciones booleanas.

$$A \cdot \bar{A} = 0$$
$$A \cdot A = A$$
$$A \cdot 0 = 0$$
$$A \cdot 1 = A$$
$$A + \bar{A} = 1$$
$$A + A = A$$
$$A + 0 = A$$
$$A + 1 = 1$$
$$A + AB = A$$
$$A + \bar{A}B = A + B$$
$$A(B + C) = AB + AC$$
$$AB = BA$$

$$A + B = B + A$$
$$(AB)C = A(BC)$$
$$(A + B) + C = A + (B + C)$$

El lector puede comprobar los teoremas anteriores utilizando tablas de verdad o manipulando algebraicamente las expresiones. Por último, se muestran los teoremas de De Morgan, los cuales son bastante útiles en la minimización de expresiones booleanas.

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$
$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

1.3.3. La disciplina estática

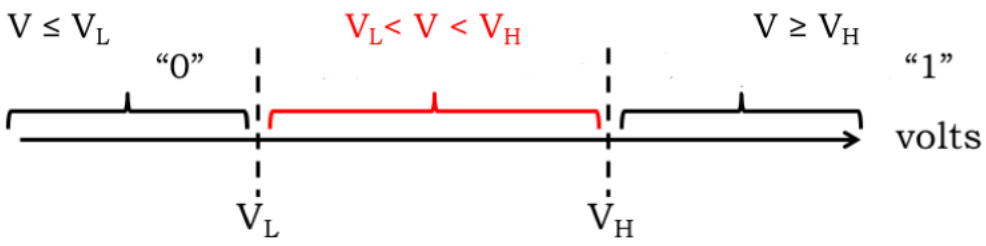
Una vez desarrolladas algunas de las propiedades básicas del álgebra booleana y los esquemas de codificación para números enteros, es momento de aplicar estas herramientas matemáticas en el diseño de circuitos digitales capaces de realizar una tarea específica. Como punto de partida del proceso, es necesario tener una especificación no ambigua de las características que deben cumplir los elementos de procesamiento digital o bloques fundamentales de diseño.

En un circuito eléctrico, es posible utilizar magnitudes eléctricas como la corriente o el voltaje para representar los valores lógicos o digitales 1 y 0 aunque regularmente se utilizan intervalos de voltaje para representar estos valores o niveles lógicos. Si, por ejemplo, se supone una variable binaria d, es posible realizar un mapeo de la variable digital hacia una variable continua de voltaje V que puede variar entre 0 y V_{DD}. En la figura 1 se muestra una posible forma de realizar dicho mapeo siguiendo una convención de señalización en la que se

utiliza un límite inferior V_L y uno superior V_H, se observa que para $V \leq V_L$ el valor de la variable d es igual a 0 y para $V \geq V_H$ es igual a 1.

Es necesario designar una región de voltaje prohibida que, por motivos prácticos, permita distinguir fácilmente entre los intervalos de voltaje válidos asociados con los niveles lógicos. En esta región, será prohibido intentar indagar acerca del comportamiento de los dispositivos ya que estos pueden interpretar el voltaje ya sea como 1 o 0.

Figura 1. **Mapeo de niveles lógicos a voltaje**



Fuente: TERMAN, Chris. *The Digital Abstraction*. www.computationstructures.org/lectures/digital/digital.html. Consulta: 26 de febrero de 2020.

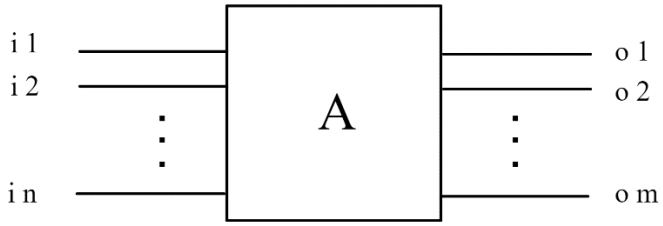
Una vez escogida la convención de señalización es posible definir los elementos de procesamiento digital básicos, los dispositivos combinacionales. Un dispositivo combinacional es aquel que cumple con los siguientes criterios:

- Posee una o más entradas digitales, por lo que, si el dispositivo sigue la convención de señalización descrita anteriormente, interpretará los voltajes de entrada inferiores o iguales a V_L como el valor digital 0, y los voltajes superiores o iguales a V_H como el valor digital 1.
- Posee una o más salidas digitales, por lo que, si el dispositivo sigue la convención de señalización descrita anteriormente, producirá un valor

digital de 0 al generar voltajes inferiores o iguales a V_L y, un valor digital de 1 al generar voltajes superiores o iguales a V_H.

- Existe una especificación funcional que detalla los valores de cada salida para cada posible combinación de valores de entrada válidos.
- Posee una especificación de tiempo, la cual consiste, por lo menos, en un límite superior t_{PD} en el cual el dispositivo produce un conjunto de salidas válidas dada una combinación arbitraria de entradas válidas, generalmente se conoce como retardo de propagación.

Figura 2. **Dispositivo combinacional**



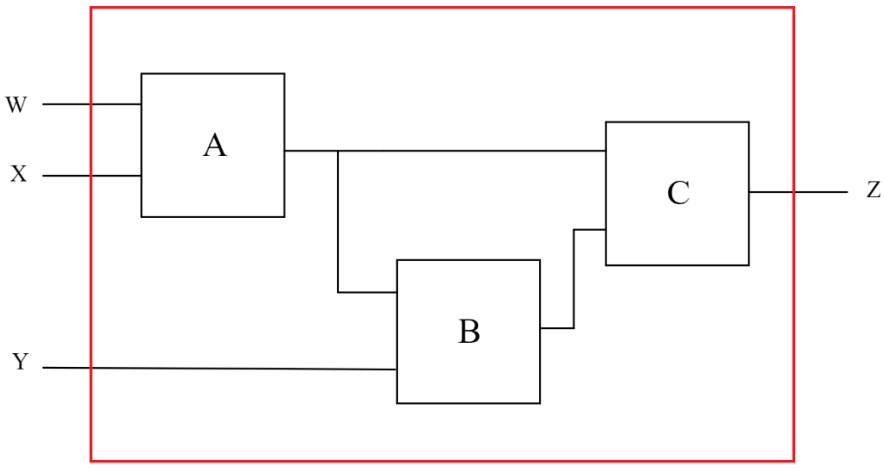
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este conjunto de requerimientos básicos se conoce como la disciplina estática, los cuales deben cumplirse para todos los dispositivos digitales. Utilizando la disciplina estática, es posible representar un dispositivo digital de n entradas y m salidas de manera abstracta como se muestra en la figura 2; de hecho, no es necesario comprender los detalles del circuito eléctrico interno para comprender la funcionalidad del dispositivo.

Los dispositivos combinacionales son un caso particular de los dispositivos digitales y, pueden entenderse como representaciones abstractas de un circuito eléctrico cuyas salidas son únicamente función de sus entradas y que ejecutan

funciones booleanas específicas. El diseñador puede conocer únicamente las características de los dispositivos referentes a la disciplina estática y ser capaz de utilizarlos en la construcción de un sistema más complejo, agilizando el proceso de diseño.

Figura 3. **Dispositivo combinacional compuesto**



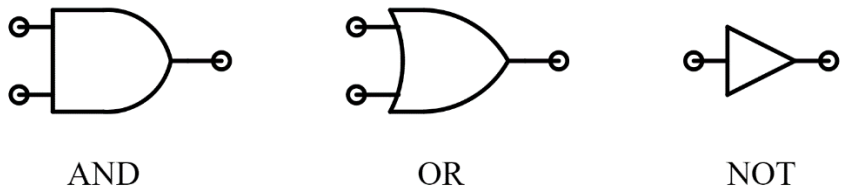
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Es posible interconectar diferentes dispositivos combinacionales para construir un dispositivo combinacional más complejo, supóngase el caso de la figura 3, en donde A, B y C son dispositivos combinacionales, se observa que el dispositivo nuevo, dentro del recuadro rojo, hereda las mismas propiedades de sus componentes: posee entradas y salidas digitales, una descripción funcional y un retardo de propagación t_{PD} a partir los retardos de propagación de los componentes individuales; por lo tanto, el dispositivo también satisface la disciplina estática. En otras palabras, es posible construir dispositivos combinacionales complejos a partir de otros componentes más sencillos que cumplan con los requerimientos de la disciplina estática.

1.3.4. Compuertas lógicas digitales

Las compuertas lógicas digitales son dispositivos combinacionales que implementan funciones booleanas básicas. En la figura 4 se muestran las compuertas lógicas que implementan las funciones AND, OR y NOT; en este caso, las compuertas son de 2 entradas exceptuando el caso de la compuerta NOT que posee únicamente 1 entrada.

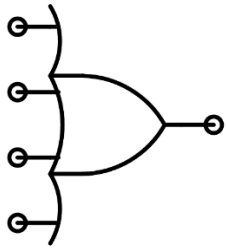
Figura 4. **Compuertas lógicas básicas**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Las compuertas lógicas pueden tener una cantidad arbitraria de entradas dependiendo de la cantidad de variables de entrada de la función booleana, en la figura 5 se muestra una compuerta OR de 4 entradas. Debido a las limitaciones de los dispositivos eléctricos con los que se construyen estas compuertas, existe un límite en la cantidad de entradas que es posible asignar a cada compuerta, en este trabajo se asume que ese límite es igual a 4 y, por lo tanto, no será posible utilizar compuertas lógicas individuales cuyo número de entradas supere dicho límite.

Figura 5. **Compuerta OR de 4 entradas**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Sin embargo, es posible construir compuertas lógicas de más de 4 entradas utilizando compuertas con un número menor de entradas. Si se aplica la propiedad asociativa a una función AND de 6 entradas, es posible reescribir la función de la siguiente manera:

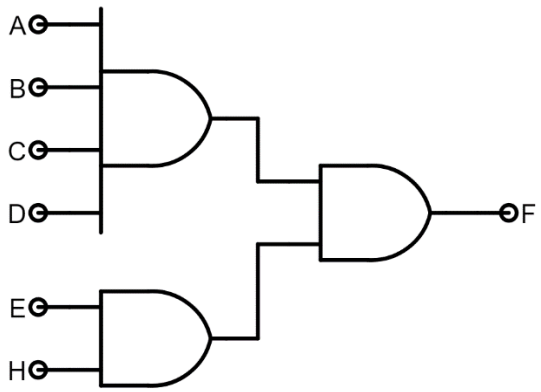
$$F = ABCDEH$$
$$F = (ABCD) \cdot (EH)$$

Como se ve, es posible sintetizar la función F utilizando 3 compuertas AND, una de 4 entradas y dos de 2 entradas, en la figura 6 se muestra el circuito que representa la función AND de 6 entradas. Utilizando el mismo razonamiento, es posible construir una compuerta OR de más de 4 entradas. Es necesario hacer énfasis que esto es posible debido a la propiedad asociativa de las funciones AND y OR.

Es momento de introducir tres nuevas funciones que serán de mucha utilidad en el diseño de circuitos en secciones posteriores: $NAND$, NOR y XOR. Las primeras dos funciones se conocen como funciones son el resultado de la negación de las funciones AND y OR respectivamente, sus expresiones

booleanas se escriben como $\overline{A \cdot B}$ y $\overline{A + B}$, en ese orden. En la tabla VI se muestra la tabla de verdad de dichas funciones, si se comparan los valores de las columnas 3 y 4 con las tablas II y III respectivamente, se observa que los valores de una tabla son el complemento o negación de la otra.

Figura 6. **Compuerta AND de 6 entradas**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Tabla VI. **Funciones $NAND$, NOR Y XOR**

A	B	$F_1 = \overline{A \cdot B}$	$F_2 = \overline{A + B}$	$F_3 = A \oplus B$
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

Fuente: elaboración propia.

Las compuertas lógicas $NAND$ y NOR de 2 entradas se muestran en la figura 7, sus diagramas de circuito son similares a los de las funciones AND y OR,

exceptuando por una pequeña burbuja en las terminales de salida. Las funciones $NAND$ y NOR no son asociativas y, por lo tanto, no es posible diseñar compuertas de más de 4 entradas utilizando el mismo razonamiento que se utilizó para las compuertas AND y OR de más de 4 entradas. Sin embargo, es posible utilizar los teoremas de De Morgan para hacer este diseño, considérese la siguiente función de 8 entradas:

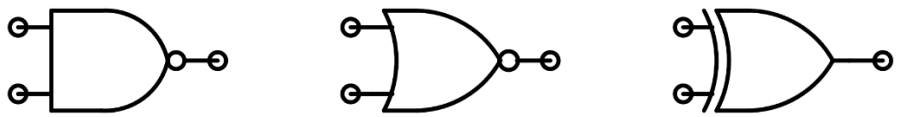
$$F = \overline{\overline{\overline{AB} + \overline{CD}} \cdot \overline{\overline{EH} + \overline{XY}}}$$

Si se aplican los teoremas de De Morgan y la propiedad $\bar{\bar{A}} = A$, es posible demostrar lo siguiente:

$$F = \overline{\overline{\overline{ABCD}} \cdot \overline{\overline{EHXY}}}$$
$$F = \overline{(ABCD) \cdot (EHXY)} = \overline{ABCDEHXY}$$

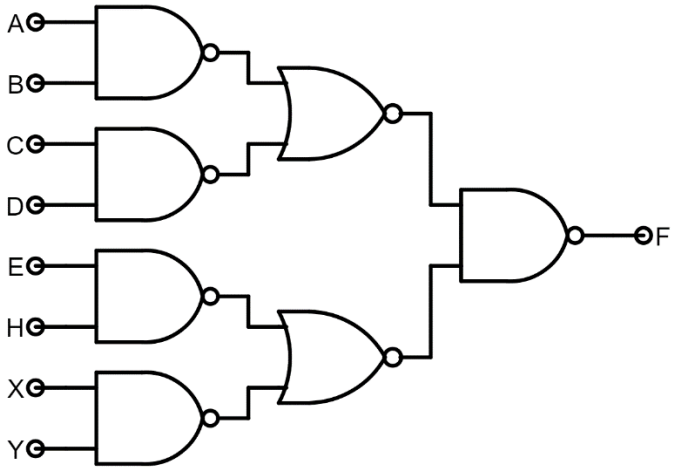
La última expresión confirma que es posible representar una función $NAND$ de una cantidad arbitraria de entradas mayor a 4 utilizando alternando compuertas $NAND$ y NOR tal y como se muestra en la figura 8.

Figura 7. **Compuertas lógicas $NAND$, NOR y XOR**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Figura 8. **Compuerta $NAND$ de 8 entradas**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

La tercera función se conoce como XOR y su compuerta lógica se muestra en la figura 7. Dicha función representa una función de desigualdad; en la quinta columna de la tabla VI se observa que el valor de la expresión $A \oplus B$ es igual a 1 si los valores de A y B son diferentes, una característica que resulta bastante útil en el diseño de circuitos comparadores.

La idea de la función XOR como una función de desigualdad se puede escribir algebraicamente de la siguiente manera:

$$F = A \oplus B = A\bar{B} + \bar{A}B$$

Como se verá en capítulos posteriores, las compuertas descritas en esta subsección constituyen los bloques lógicos de diseño básicos, a partir de los cuales es posible construir componentes lógicos y circuitos digitales más complejos.

1.3.5. Análisis transitorio de compuertas

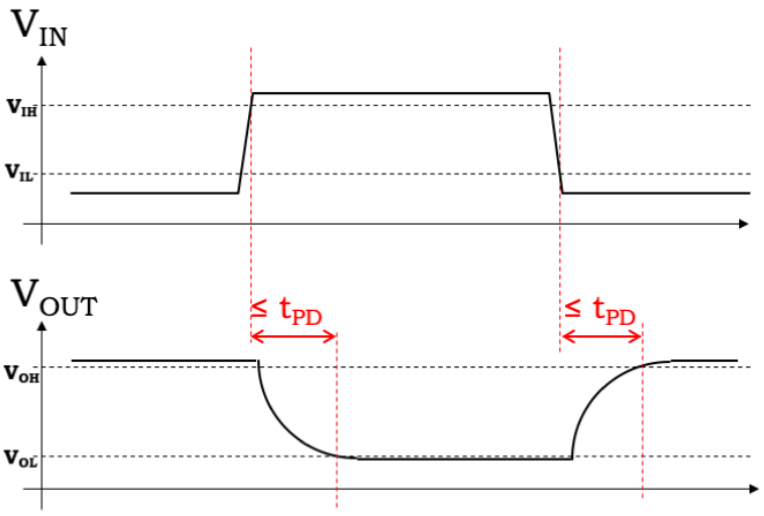
Como ya se explicó en los criterios de la disciplina estática, es necesario especificar un intervalo de tiempo máximo de respuesta para compuertas lógicas o circuitos digitales más complejos. Es erróneo asumir que los cambios en las entradas de un dispositivo digital se reflejarán instantáneamente en las salidas del dispositivo, aunque el tiempo de respuesta de los dispositivos combinacionales básicos es relativamente pequeño, del orden de $10^{-9}s$, los tiempos de respuesta en circuitos que posean millones de componentes pueden llegar a ser perceptibles para el ser humano.

1.3.5.1. Retardo de propagación

Se define como retardo o delay de propagación al límite superior en el retardo de una configuración de entradas digitales válidas para producir salidas digitales válidas. En la figura 9 se ilustra este concepto para el caso de un inversor o compuerta NOT; cuando el valor del voltaje de entrada V_{IN} del dispositivo es mayor o igual al límite superior de entrada V_{IH}, el dispositivo producirá un valor de voltaje de salida V_{OUT} menor o igual al límite inferior de salida V_{OL} en un tiempo no mayor al retardo de propagación t_{PD}.

Cuando se utilizan dispositivos combinacionales es necesario identificar apropiadamente este parámetro e identificar que únicamente es un límite superior o valor máximo, es decir, que los valores de retardos pueden ser menores a este valor. En el diseño de circuitos digitales, se suele asumir que el tiempo de respuesta de las compuertas es equivalente a t_{pd}, debido a que este valor supone el máximo posible y el peor de los casos de retardo.

Figura 9. **Retardo de propagación**

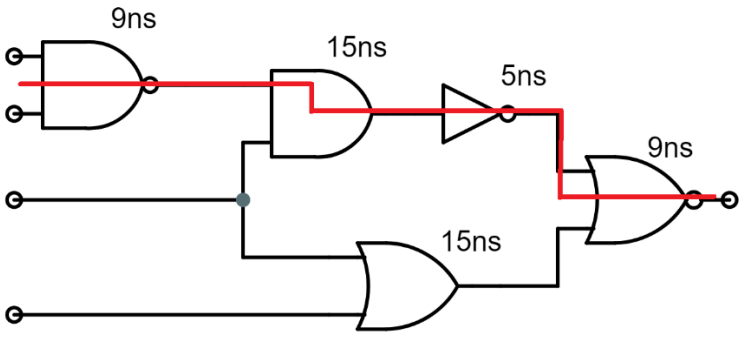


Fuente: TERMAN, Chris. *CMOS Technology*. www.computationstructures.org/ lectures/cmos/cmos.html. Consulta: 1 de marzo de 2020.

En un sistema digital o dispositivo combinacional compuesto por elementos más simples en donde cada uno de los dispositivos posee su propio retardo de propagación t_{PD_x}, el retardo de propagación t_{PD} del sistema se determina acumulando los retardos de propagación de todas las rutas posibles desde las entradas hasta las salidas y seleccionando el valor máximo de todas las rutas.

Para ilustrar esta idea supóngase el ejemplo del circuito de la figura 10 en donde se muestran los retardos de propagación individuales de los dispositivos, para determinar el retardo de propagación del circuito se observa que la ruta marcada por la línea roja posee el retardo de propagación t_{PD} acumulado máximo, cuyo valor es de $38\,ns$.

Figura 10. **t_{PD} en un sistema combinacional**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

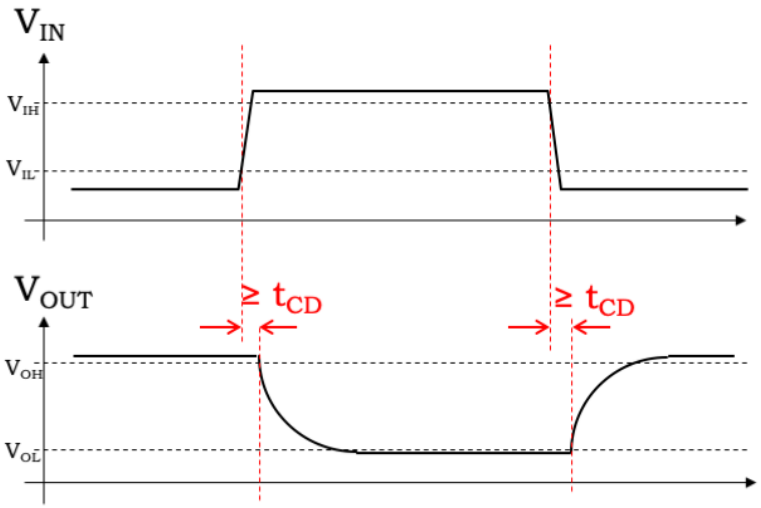
1.3.5.2. Retardo de contaminación

A pesar de que la disciplina estática no especifique su requerimiento, resulta bastante útil definir el retardo de contaminación para el diseño de cierto tipo especial de circuitos digitales que requieren el almacenamiento de información. El retardo de contaminación t_{CD} se define como el límite inferior en el retardo que existe de una entrada digital inválida a una salida digital inválida; en otras palabras, el parámetro mide el tiempo que una compuerta retiene su antiguo valor de salida después de que el valor de entrada comenzó a cambiar y se volvió inválido.

En la figura 11 se ilustra el concepto para el caso de un inversor digital; cuando el valor del voltaje de entrada V_{IN} del dispositivo aumenta y entra a la región inválida de entrada, $V_{IL} \leq V_{IN} \leq V_{IH}$, el dispositivo producirá un valor de voltaje de salida V_{OUT} inválido, $V_{OL} \leq V_{OUT} \leq V_{OH}$, en un tiempo mayor o igual al retardo de contaminación t_{CD}. Generalmente no es necesario especificar o considerar el parámetro t_{CD}, únicamente es relevante en circuitos que poseen la

propiedad de memoria como registros o máquinas de estado. Existen muchos casos en los que el retardo de contaminación no se especifica y, comúnmente se asume que es igual a cero, lo cual significa que se asume una transición de entradas inválidas a salidas inválidas instantánea.

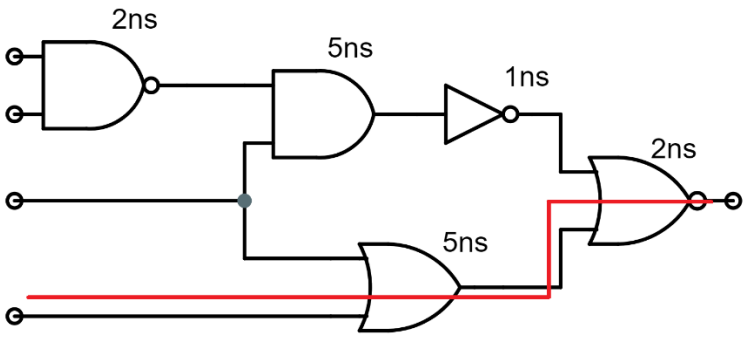
Figura 11. **Retardo de contaminación**



Fuente: TERMAN, Chris. *CMOS Technology*. www.computationstructures.org/ lectures/cmos/cmos.html. Consulta: 1 de marzo de 2020.

De la misma manera en que se encontró el retardo de propagación para un sistema o dispositivo combinacional compuesto, para determinar el retardo de contaminación t_{CD} de un sistema se escoge el camino de entradas a salidas que posea la suma acumulada mínima de retardos de contaminación de los dispositivos individuales t_{CD_x}. En la figura 12 se muestra nuevamente el circuito de la figura 10 con los retardos de contaminación de los dispositivos, en rojo se indica el camino o ruta que minimiza el retardo de contaminación global t_{CD}, cuyo valor es de 7 ns.

Figura 12. **t_{CD} en un sistema combinacional**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

1.4. Lógica combinacional

A partir de la abstracción de la compuerta lógica digital como bloque fundamental de diseño de circuitos digitales, se procede a realizar una descripción de los métodos básicos de diseño para el caso de los circuitos combinacionales. Se hace uso de la propiedad constructiva de los dispositivos combinacionales expuesta en la sección anterior: un circuito es combinacional si no posee ningún tipo de realimentación y todos sus componentes son dispositivos combinacionales.

1.4.1. Especificaciones funcionales

Para realizar el diseño de un circuito combinacional, es necesario tener una descripción no ambigua del comportamiento de las entradas y salidas del circuito. Existen dos maneras de representar sin ambigüedades la funcionalidad de un circuito: tablas de verdad y expresiones booleanas. Como se explicó en la sección 1.3, una tabla de verdad especifica el valor de salida de una función Z para cada posible combinación de las n entradas. En la tabla VII se muestra la

tabla de verdad para una función booleana Z de $n = 3$ entradas, en esta función, la salida es equivalente a 1 si la cantidad de entradas equivalentes a 1 es impar.

Tabla VII. **Especificación funcional de Z**

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Fuente: elaboración propia.

Es bastante sencillo representar la funcionalidad de cualquier circuito de n entradas, sólo basta con enumerar todas las posibles combinaciones de entradas y asignar el valor de salida para cada una de ellas. Sin embargo, el número total de combinaciones posibles y, por ende, el número de filas en la tabla crece exponencialmente; en la práctica es necesario buscar otra alternativa cuando el número de entradas sobrepasa a $n = 4$ o $n = 5$. Para realizar la tabla de verdad de una función de 7 entradas se necesitaría una cantidad de $2^7 = 128$ filas, por lo que el diseñador tendría que enumerar las 128 posibles combinaciones de las entradas, algo realmente tedioso y propenso al error para la labor de un diseñador.

Las expresiones booleanas son especificaciones funcionales alternativas que permiten calcular los valores de salida a partir de los valores de entrada utilizando álgebra booleana, la información de la tabla VII se puede representar de manera compacta a través de la siguiente expresión:

$$Z = \bar{A}(C \oplus B) + A(\overline{B + C} + BC)$$

Es fácil comprobar que tanto la tabla como la expresión contienen la misma información, únicamente es necesario introducir todas las posibles combinaciones de entradas en la expresión booleana y comparar los resultados con los de la tabla de verdad. Por este motivo, se dice que la tabla de verdad y la expresión booleana son representaciones intercambiables, esto quiere decir que es posible derivar una expresión booleana a partir de una tabla de verdad y viceversa.

Las expresiones booleanas son una alternativa de representación bastante conveniente cuando el número de entradas se vuelve relativamente grande. Además, es posible derivar el circuito combinacional de manera directa y bastante intuitiva a partir de este tipo de especificación funcional.

1.4.2. Representación de suma de productos

El enfoque de diseño descrito en esta subsección transforma una tabla de verdad en una expresión booleana utilizando la representación estándar de suma de productos. La estructura básica de dicha expresión consiste en la suma booleana, operación OR, de varios términos que consisten en el producto booleano, operación AND, de algún conjunto de valores de entrada, que algunas entradas pueden estar negadas. Cada término t_i corresponde a una línea de la tabla de verdad en la que la salida es igual a 1, estos términos se conocen como los implicantes de la función.

En otras palabras, la representación de suma de productos de una función booleana Z es la operación OR de todos sus términos implicantes t_i. Así, para

una función con m implicantes, la expresión de suma de productos posee la siguiente estructura:

$$Z = t_0 + t_1 + t_2 + \cdots + t_{m-2} + t_{m-1}$$

Si se observa de nuevo la función de la tabla VIII, su salida es equivalente a 1 para cada una de las combinaciones de entradas $ABC = 001, 010, 100, 111$. La función posee 4 términos implicantes y, es necesario construir cada uno de los términos implicantes t_i de tal forma que su valor sea igual a 1 si se proporciona la combinación correcta de entradas; para que esto suceda, se debe incluir cada variable de entrada en el término si su valor es igual a 1 e invertirla, o negarla, si su valor es igual a 0. De esta manera, los términos implicantes para este caso en particular son:

$$t_0 = \bar{A}\bar{B}C$$
$$t_1 = \bar{A}B\bar{C}$$
$$t_2 = A\bar{B}\bar{C}$$
$$t_3 = ABC$$

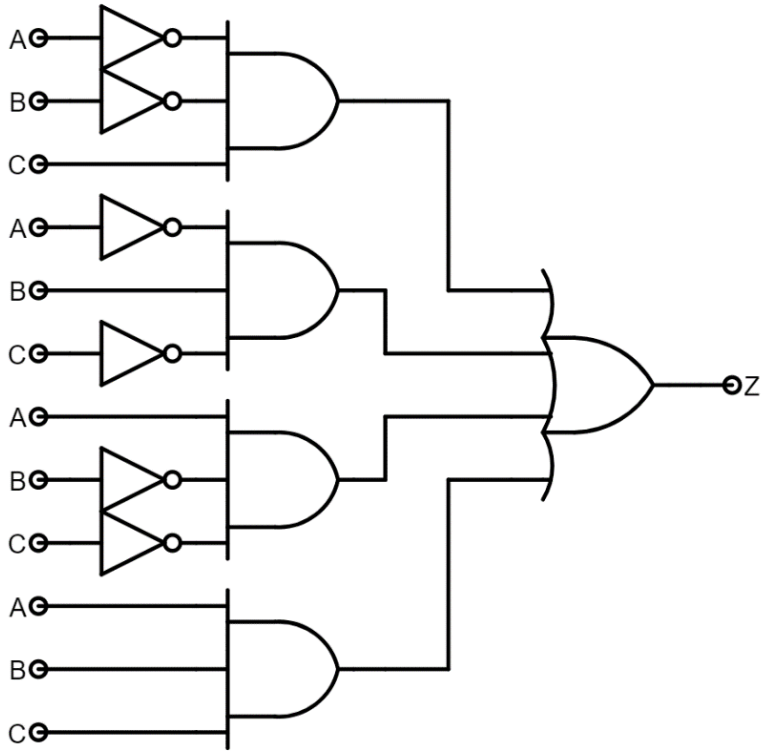
Es importante observar que los implicantes tienen un valor de 1 únicamente si se proporciona la combinación de entradas de la tabla de verdad con la cual se encuentran asociados, de lo contrario producirán un valor de 0; como resultado de esto, sólo un implicante puede ser equivalente a 1 dada una combinación de entradas arbitraria. Reescribiendo la función Z de la tabla VII en su forma estándar de suma de productos se obtiene:

$$Z = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C}$$

Si se observa cuidadosamente la estructura de la representación de suma de productos, únicamente contiene las operaciones lógicas AND, OR y NOT. Por

lo tanto, debido a la correspondencia directa que existe entre los circuitos combinacionales y las expresiones booleanas, es posible utilizar únicamente compuertas AND, OR e inversores como bloques de diseño de circuitos combinacionales. La estructura de los circuitos sigue exactamente la misma estructura de la expresión booleana, en la figura 13 se puede apreciar el circuito digital correspondiente a la expresión anterior, se utilizaron compuertas AND de 3 entradas y 1 compuerta OR de 4 entradas.

Figura 13. **Síntesis de suma de productos**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este procedimiento se puede generalizar para funciones con una cantidad arbitraria k de entradas. El circuito deberá contener una etapa de inversores para

generar las entradas invertidas, un conjunto de N compuertas AND de k entradas en donde N representa el número de implicantes y, por último, una compuerta OR de N entradas.

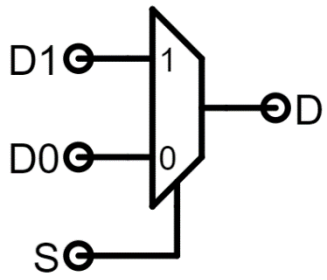
En resumen, para diseñar un circuito combinacional utilizando este enfoque, es necesario realizar la especificación funcional del circuito con una tabla de verdad para posteriormente transformarla a la representación de suma de productos y, por último, utilizar las compuertas lógicas AND, OR y NOT para sintetizar el circuito en función de la expresión booleana. En algunos casos, es posible minimizar las expresiones booleanas y así, reducir el número de compuertas lógicas necesarias para implementar el circuito.

1.4.3. Multiplexores

Un multiplexor es un dispositivo combinacional que selecciona información binaria de una de muchas entradas y la envía hacia una sola línea de salida. La selección de una entrada particular depende de un conjunto de entradas, llamadas entradas de selección. Si existen n líneas de selección, entonces existe una cantidad de 2^n líneas de entrada que el dispositivo es capaz de distinguir. En la figura 14 se muestra su diagrama lógico para el caso particular de un multiplexor con una línea de selección y 2 entradas y en la tabla VIII se muestra su especificación funcional, el dispositivo produce una salida $D = D_0$ si $S = 0$, por el contrario, produce una salida $D = D_1$ si $S = 1$.

Los multiplexores son de gran utilidad en el diseño de circuitos digitales en los que se requiere que varias entradas compartan un mismo canal o línea de salida, el valor de una entrada arbitraria se ve reflejado en la salida si la combinación de entradas de selección es la correcta. Este tipo de dispositivos se utilizarán con frecuencia en el diseño de circuitos en capítulos posteriores.

Figura 14. **Diagrama lógico de un multiplexor 2 a 1**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Tabla VIII. **Tabla de verdad de un multiplexor de 2 entradas**

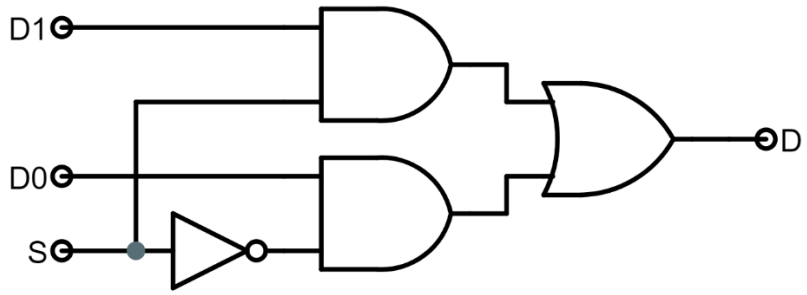
S	D_0	D_1	D
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Fuente: elaboración propia.

En la figura 15 se muestra una forma de implementar el multiplexor de dos entradas utilizando compuertas lógicas digitales, se observa que cuando $S = 0$ la compuerta AND asociada a la entrada D_1 queda inhabilitada, mientras que la compuerta asociada a la otra entrada se encuentra habilitada y su valor de salida depende únicamente del valor de D_0, cuando $S = 1$ se habilita la compuerta asociada a la entrada D_1 mientras que la otra queda inhabilitada; por último, se

utiliza una compuerta OR para seleccionar cualquiera de las salidas de las compuertas AND. Una característica particular acerca de esta implementación es que sólo una compuerta AND se encuentra habilitada a la vez.

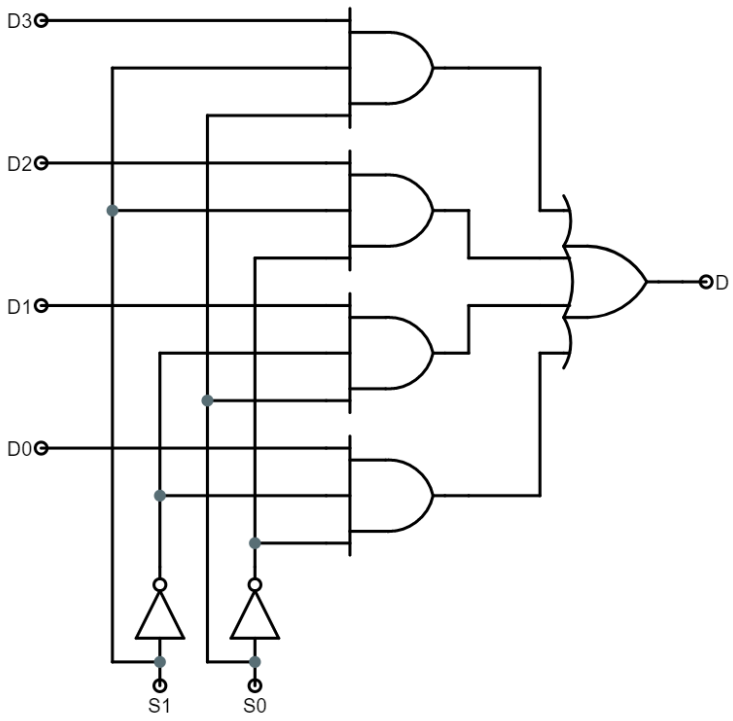
Figura 15. **Circuito interno de un multiplexor 2 a 1**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este tipo de implementación se puede generalizar para un multiplexor de 2^n líneas de entrada y n líneas de selección; considérese el caso de la figura 16 en donde se muestra la estructura interna de un multiplexor de 4 entradas y 2 líneas de selección, o multiplexor 4 a 2, su funcionamiento es similar al circuito de la figura 15 en donde sólo una compuerta AND se encuentra habilitada para cada posible combinación de las señales de entrada S_0 y S_1 las cuales, se tratan como un solo número o arreglo binario $S[1:0] = 0bS_1S_0$; el circuito habilita las compuertas AND asociadas a las entradas D_3, D_2, D_1 y D_0 para los valores de $S[1:0] = 0b11$, $0b10$, $0b01$ y $0b00$ respectivamente.

Figura 16. **Circuito interno de un multiplexor 4 a 2**



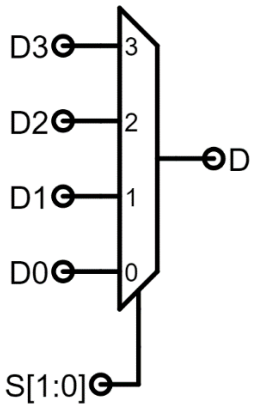
Fuente: elaboración propia, empleando Jade Circuit Simulator.

El diagrama lógico de un multiplexor 4 a 2 se muestra en la figura 17, el diagrama posee una terminal de selección única que consiste en un arreglo de líneas de selección codificadas dentro del número binario $S[1:0] = 0bS_1S_0$, la cual se utiliza únicamente por motivos prácticos, dibujar todas las líneas de entrada en el diagrama sería innecesario, especialmente cuando el número de líneas de selección aumenta notablemente.

Si se considera que las compuertas AND poseen el mismo retardo de propagación, entonces el retardo de propagación t_{PD} del multiplexor está determinado por el retardo acumulado desde las entradas de selección hacia la

salida D. Como se verá más adelante, es importante considerar el retardo de propagación en multiplexores debido a que estos dispositivos combinacionales serán de gran utilidad en el diseño de dispositivos con capacidad de almacenar información.

Figura 17. **Diagrama lógico de un multiplexor 4 a 2**



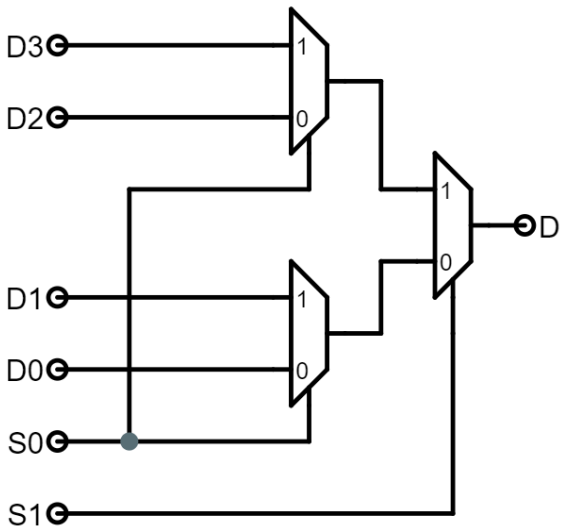
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Por último, cuando el número de entradas del multiplexor que se desea implementar es significativamente grande, es posible conectar multiplexores más pequeños para formar un multiplexor mucho más grande como se ilustra en la figura 18, este circuito realiza la misma función que el multiplexor 4 a 2 de las figuras 16 y 17, la entrada S_0 selecciona dos combinaciones de entradas: D_3, D_1 y D_2, D_0, las dos señales de los multiplexores de la primera etapa se propagan hacia el último multiplexor en el que la entrada S_1 selecciona un valor de la combinación seleccionada en la primera etapa y lo refleja en la salida D.

Esta alternativa de diseño supone una ventaja, ya que no es necesario preocuparse por la conexión interna de las compuertas lógicas AND, OR y NOT;

por lo tanto, el proceso de diseño de multiplexores más grandes se agiliza y se vuelve más cómoda. La desventaja del uso de este método radica en el aumento del retardo de propagación del multiplexor t_{PD}, para el caso particular de la figura 18 se puede observar que ahora el retardo es mayor debido a que el dispositivo cuenta con dos etapas de multiplexores conectadas una después de la otra; cuando se realiza la conexión interna de compuertas lógicas, el retardo de propagación es independiente del número de entradas del multiplexor.

Figura 18. **Multiplexor compuesto**



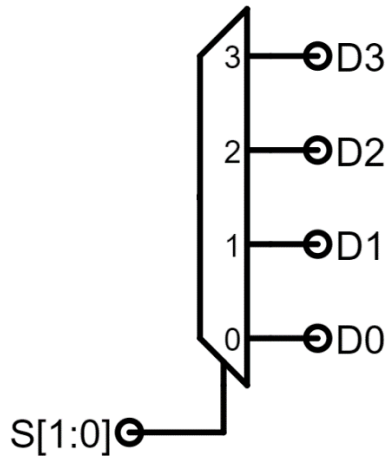
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Los multiplexores son una parte importante de toda librería de bloques lógicos fundamentales de diseño en circuitos digitales y, su relevancia, se expondrá en capítulos posteriores, tanto para el diseño de la ALU como para otros bloques del procesador.

1.4.4. Decodificadores

En esta subsección se introduce un nuevo dispositivo que se utilizará como bloque de diseño de circuitos digitales más complejos: el decodificador. Un decodificador es un dispositivo combinacional que convierte información binaria de n líneas de entrada a un máximo de 2^n líneas de salida diferentes. En la figura 19 se muestra el diagrama lógico para el caso particular de un decodificador de 4 salidas, el dispositivo toma como entrada $n = 2$ líneas de selección cuyos valores seleccionan una de $2^n = 4$ salidas, se observa que para una combinación de entradas válidas se seleccionará una línea de salida única.

Figura 19. **Diagrama lógico de un decodificador**

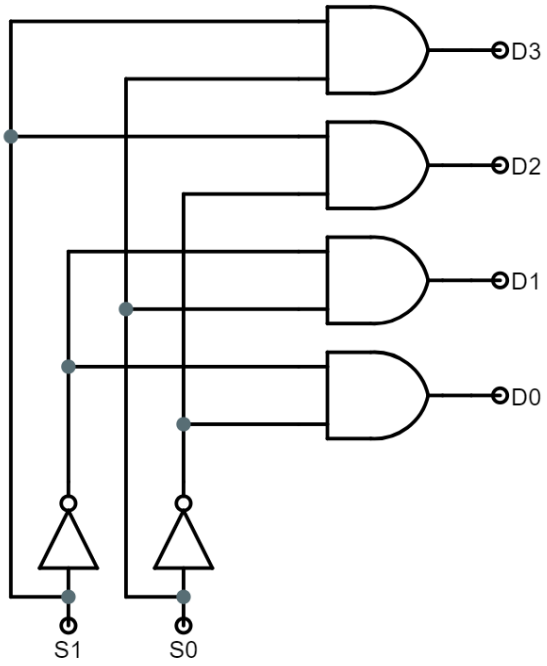


Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la figura 20 se muestra una implementación de un decodificador de 4 salidas utilizando compuertas lógicas, una compuerta AND particular se habilitará y su salida será igual a 1 si se aplica la combinación de entradas con la cual se encuentra asociada. Específicamente, las salidas D_3, D_2, D_1 y D_0 serán

equivalentes a 1 para las combinaciones de entrada $S[1:0] = 0b11,\ 0b10,\ 0b01,$ $0b00$ respectivamente.

Figura 20. **Circuito interno de un decodificador**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Los decodificadores son componentes fundamentales en la construcción de memorias de sólo lectura, ROM; como se verá en la siguiente subsección, las ROM son componentes bastante convenientes que simplifican las tareas de diseño de circuitos combinacionales de varias entradas y salidas.

1.4.5. Memoria de sólo lectura, ROM

Normalmente, la manera más eficiente de implementar una función booleana compleja en términos de tamaño y velocidad de respuesta es a través

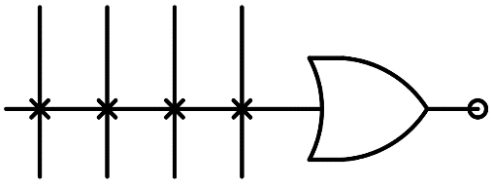
de un circuito combinacional personalizado y optimizado para esa función específica, este enfoque utiliza como principales herramientas los métodos de diseño descritos en las subsecciones 1.4.1 y 1.4.2. Sin embargo, en algunas ocasiones este procedimiento se vuelve bastante costoso en términos de producción y diseño.

Un enfoque de diseño alternativo consiste en técnicas sistemáticas de implementación de circuitos las cuales, permiten implementar de una manera más sencilla una función arbitraria de k entradas y n salidas a partir de un circuito combinacional de propósito general. Uno de los dispositivos capaces de ser utilizados para esta tarea es la memoria de sólo lectura, comúnmente abreviada como ROM.

Una ROM es un dispositivo que almacena información binaria de manera permanente, la información se programa dentro de la estructura interna del dispositivo como bits de datos en localidades que tienen una correspondencia biunívoca con las combinaciones de entradas. Una ROM posee k entradas y n salidas, las entradas proporcionan la dirección de memoria y las salidas suministran los bits de datos almacenados en esa dirección.

Las ROM pueden llegar a tener miles de compuertas interconectadas por trayectorias internas complejas. Por este motivo se introduce un nuevo diagrama lógico para la representación de compuertas lógicas de múltiples entradas, esta notación es bastante útil para la representación esquemática de arreglos lógicos programables y se utilizará ampliamente en la sección 1.8. En la figura 21 se muestra la nueva representación para una compuerta OR de 4 entradas, se traza una sola línea hacia la compuerta y las líneas de entrada se dibujan perpendiculares a esa línea única, se indica explícitamente su conexión a través del símbolo “×” en las intersecciones.

Figura 21. **Representación alterna de compuertas lógicas**

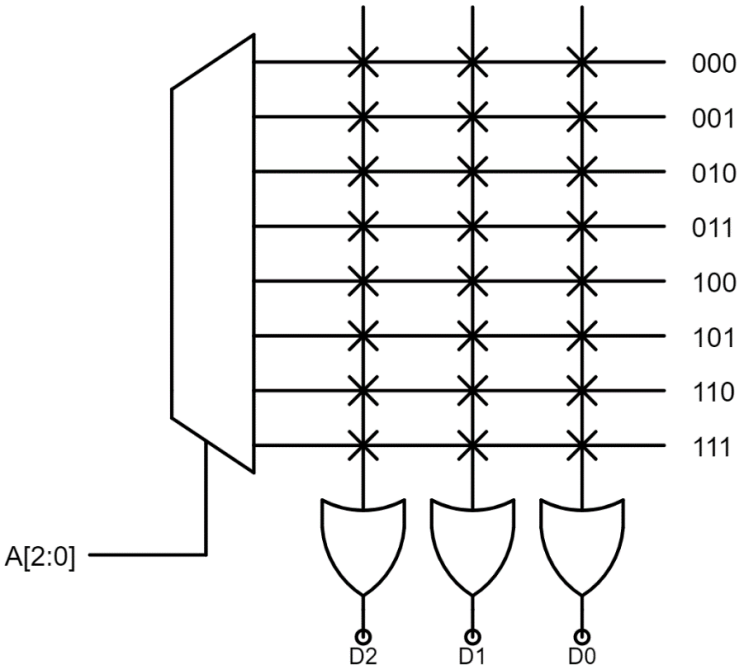


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Utilizando esta nueva representación, es posible representar la estructura interna de una ROM de manera concisa, considérese el ejemplo de una memoria de 3 entradas y 3 salidas, el circuito interno de la memoria se muestra en la figura 22, se necesita un decodificador de 3×8 el cual, posee como entradas las 3 líneas de dirección codificadas dentro del arreglo binario $A[2:0]$ que permiten seleccionar las 8 posibles líneas de salida; estas líneas de salida se pueden conectar a las entradas de las compuertas OR para producir los bits de datos correspondientes.

Para la configuración particular de la figura 22, se observa que todas las líneas de salida del decodificador se encuentran conectadas a las compuertas OR, por lo tanto, se dice que la memoria aún no ha sido programada; para programar la memoria se procede a alterar la interconexión de líneas de salida con las compuertas OR siguiendo la especificación funcional de una tabla de verdad. Una conexión programable o punto de cruce entre dos líneas equivale lógicamente a un interruptor que se puede alterar de modo que esté cerrado o abierto, una de las tecnologías más sencillas que permiten hacer esto es a través de la quema de fusibles localizados en los puntos de cruce.

Figura 22. **Circuito interno de una ROM**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

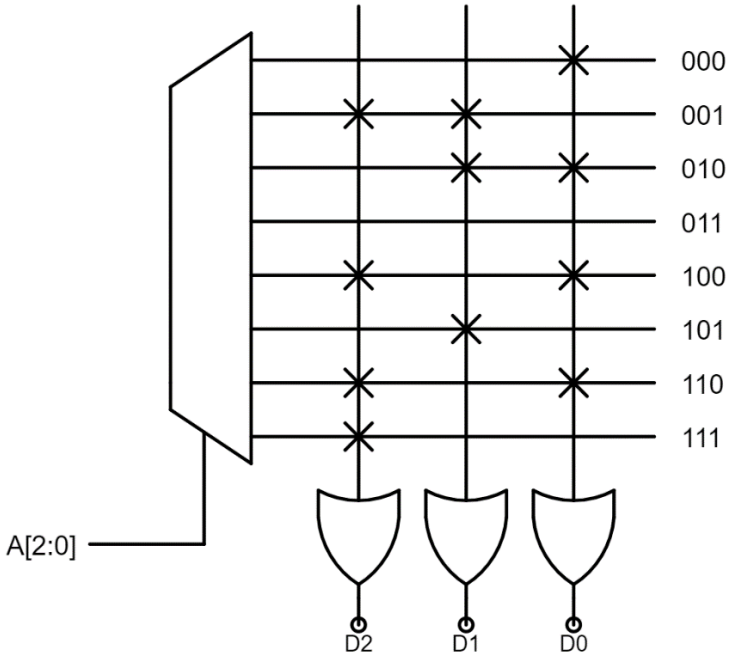
Supóngase que se desea construir un circuito que implemente la especificación funcional de la tabla IX, las columnas de salida en cada fila indican el contenido que se debe programar en cada dirección de la memoria. Cada 0 de la tabla de verdad especifica la ausencia de una conexión, y cada 1, indica una conexión; en la figura 23 se muestra la ROM con los valores programados, para el caso de la información binaria de la fila 5, el valor a almacenar es $0b101$ por lo que en la práctica se procedería a quemar el fusible de la fila 5 y la columna 2 para programar esta información.

Tabla IX. **Tabla de verdad con 3 entradas y 3 salidas**

A_2	A_1	A_0	D_2	D_1	D_0
0	0	0	0	0	1
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	1	0
1	1	0	1	0	1
1	1	1	1	0	0

Fuente: elaboración propia.

Figura 23. **Información binaria programada en una ROM**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

1.5. Lógica secuencial síncrona

En la sección 1.4 se expuso una característica importante acerca de los circuitos combinacionales: los valores de salida dependen únicamente de los valores de entrada. Esta sección se centra en la descripción de un nuevo tipo de circuito digital, el circuito secuencial síncrono, este tipo de circuito es capaz de recordar información binaria y producir salidas que dependen de los valores de entrada y la información almacenada. En otras palabras, las salidas de un circuito secuencial no dependen únicamente de los valores de entrada, sino también de la información previa codificada dentro de una cadena de bits que el circuito ha memorizado y que recibe el nombre de estado.

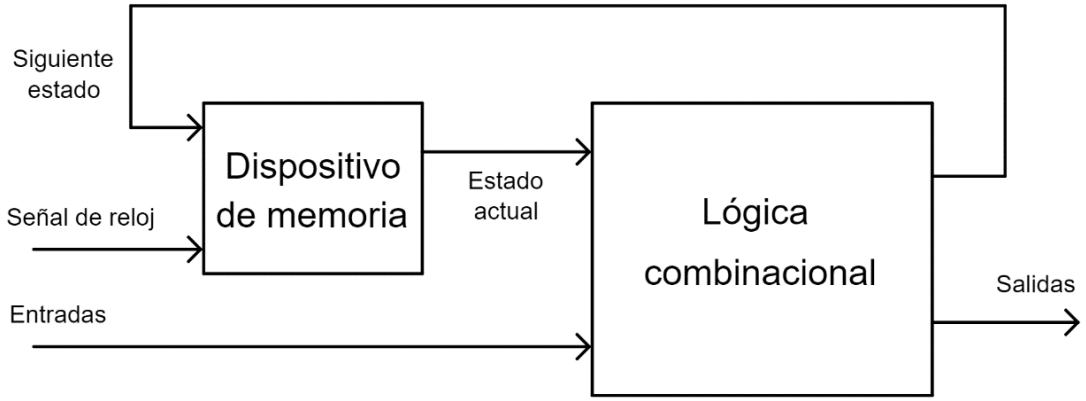
En este tipo de circuitos normalmente se producen diferentes salidas para los mismos valores de entrada, esto se debe a que el valor de salida depende también del estado en que se encuentra el circuito. La noción de estado en un circuito secuencial los hace bastante útiles para procesar información binaria de manera secuencial, en los capítulos 3 y 4 se explotará su propiedad de memoria para diseñar los registros internos que necesitan almacenar datos dentro del procesador.

1.5.1. Circuitos secuenciales síncronos

Un circuito secuencial síncrono es un sistema en el que las salidas son funciones de las entradas y del estado almacenado dentro de un elemento de memoria, el valor de dicho estado se define en instantes de tiempo discretos a través de una señal de control llamada reloj del sistema. La estructura básica de un circuito secuencial síncrono se muestra en la figura 24, se observa que el circuito consta de un dispositivo de memoria y de un circuito combinacional; la lógica combinacional produce las salidas y el siguiente estado a partir del estado

actual y las entradas del circuito, el dispositivo de memoria produce el valor del estado actual a partir de la información binaria del bloque combinacional y la señal de reloj.

Figura 24. **Estructura básica de un circuito secuencial**



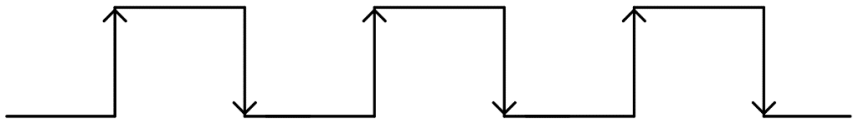
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Se debe aclarar que el dispositivo de memoria almacena el estado del circuito como una cadena de bits de tamaño arbitrario k que depende de la cantidad de estados diferentes que el circuito es capaz de distinguir, por lo tanto, la cantidad de bits utilizados para las señales del estado actual y siguiente deben tener la misma cantidad k de bits. Específicamente, si el dispositivo de memoria puede almacenar k bits, el límite de estados que puede tener el circuito secuencial es de 2^k.

La señal de reloj se muestra en la figura 25, consiste en un tren periódico de pulsos de reloj, esta señal se distribuye a todos los elementos de memoria dentro de un sistema digital. A las transiciones de 0 a 1 marcadas con una flecha hacia arriba se les llama flancos de subida y a las transiciones de 1 a 0 marcadas

con una flecha hacia abajo se les llama flancos de bajada. Dependiendo del tipo de dispositivo de memoria, este puede definir su estado en función de transiciones del reloj, flancos de subida o flancos de bajada; o en función de algún nivel lógico en particular, uno o cero.

Figura 25. **Señal de reloj**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

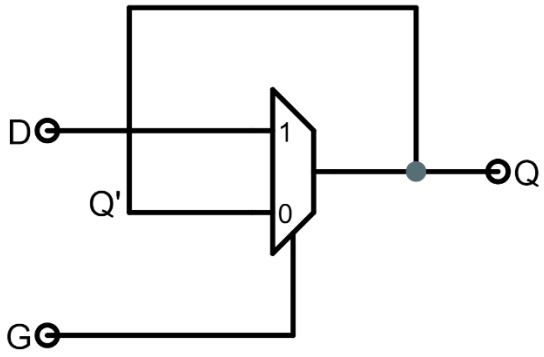
El resto del análisis de esta sección se centra en el diseño de algunos dispositivos de memoria capaces de cumplir la tarea de almacenar información a partir de dispositivos combinacionales y el uso de realimentación entre salidas y entradas. La siguiente sección se centra en la descripción de técnicas sistemáticas de diseño de circuitos secuenciales capaces de ejecutar una tarea en particular.

1.5.2. Latch D

Es posible construir un dispositivo de memoria biestable, capaz de almacenar sólo 1 bit, a partir de un multiplexor 2 a 1 en el que su salida se conecta directamente a una de sus entradas, esto se ilustra en la figura 26; la salida Q representa el estado del dispositivo y se conecta a la entrada 0 del multiplexor representada con la variable Q', la entrada D representa la entrada de datos del dispositivo y la entrada G representa la señal de control del dispositivo que permite al dispositivo seleccionar entre la entrada Q' o D del multiplexor. Es

importante señalar que este dispositivo ya no es combinacional debido a la presencia del lazo de realimentación entre la salida y una de las entradas.

Figura 26. **Circuito interno de un latch D**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Tabla X. **Tabla de verdad de un Latch D**

G	D	Q'	Q
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

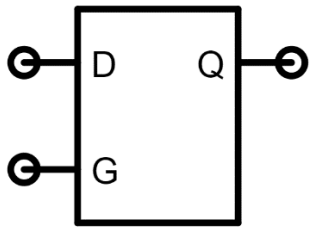
Fuente: elaboración propia.

En la tabla X se muestra la especificación funcional compacta del dispositivo, en este caso, se utiliza el símbolo X para indicar que el valor de esa variable es irrelevante para definir el valor de salida en una fila particular. En la tabla de verdad se observa que si $G = 0$ la salida Q es equivalente al estado anterior Q' sin importar el valor de D y, por lo tanto, se dice que el dispositivo se

encuentra en modo de almacenamiento, cuando $G = 1$ se obtiene una salida Q que sigue al valor de entrada D sin importar el valor del estado anterior almacenado Q', en este caso el dispositivo almacena un estado nuevo.

En resumen, cuando $G = 1$ se dice que el latch está abierto y la información de la entrada D se refleja en la salida Q y cuando $G = 0$ se dice que está cerrado o en modo de memoria. En la figura 27 se muestra el símbolo lógico de un latch D, únicamente se indican las entradas G y D, y la salida Q.

Figura 27. **Símbolo lógico de un latch D**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Es importante recordar, que el latch D posee un retardo de propagación asociado con el del multiplexor que lo compone, si se asume un retardo t_{PD} para el latch, entonces cuando $G = 1$ el dispositivo producirá un valor $Q = D$ en un tiempo no mayor a t_{PD} después de que la entrada D ha alcanzado un valor válido. Según los criterios de la disciplina estática aplicados al multiplexor interno, no es confiable asumir ningún valor de salida Q dentro del intervalo de tiempo $t_{CD} \leq t \leq t_{PD}$, el multiplexor puede producir cualquier valor de salida ya sea válido o inválido durante este intervalo.

Debido al retardo t_{PD} surge un inconveniente asociado a la transición de la entrada G cuando $G = 1$ y $Q = D$, si G realiza una transición de 1 a 0, según la disciplina estática, el dispositivo reflejará este cambio en la salida Q en un tiempo no mayor a t_{PD}, como consecuencia de esto, es posible que el valor $Q = D$ se vuelva inválido durante este intervalo de tiempo y se pierda la información que se desea almacenar. Para hacer este dispositivo de memoria más confiable es necesario establecer un conjunto de criterios dinámicos que deben satisfacer las señales de entrada.

1.5.3. La disciplina dinámica

Para que un latch D se comporte de manera confiable en la transición de 1 a 0 en la entrada G, es necesario utilizar un multiplexor que cumpla con los requerimientos de la tabla XI, en este caso el dispositivo especifica que si $D = Q'$ el valor de G es irrelevante para determinar la salida Q. Al examinar las filas de la tabla de verdad, se observa que este dispositivo se comporta de manera confiable sin contaminar la salida bajo las siguientes condiciones:

- $G = 1$, $D = V_x$ se mantienen válidos por al menos t_{PD}: $Q = V_x$ sin importar el valor de Q'. En este caso, el valor de D se ha escogido como la salida del multiplexor y el valor de Q' es irrelevante.
- $D = Q' = V_x$ se mantienen válidos por al menos t_{PD}: $Q = V_x$ sin importar el valor de G. En este caso, el multiplexor debe elegir entre dos valores idénticos de entrada, por lo que el valor de G es irrelevante.
- $G = 0$, $Q' = V_x$ se mantienen válidos por al menos t_{PD}: $Q = V_x$ sin importar el valor de D. En este caso, el valor de Q' se ha escogido como la salida del multiplexor y el valor de D es irrelevante.

Tabla XI. **Tabla de verdad de un latch D permisivo**

G	D	Q'	Q
1	0	X	0
1	1	X	1
X	0	0	0
X	1	1	1
0	X	0	0
0	X	1	1

Fuente: elaboración propia.

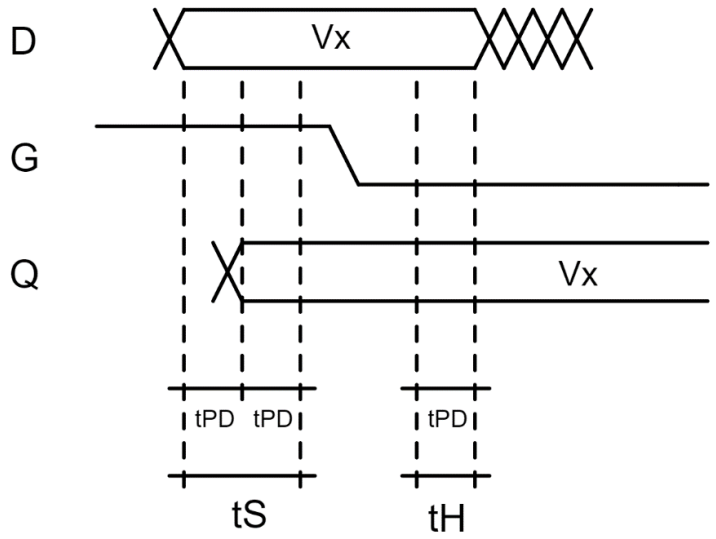
Además de la especificación de la tabla XI, es necesario que las entradas cumplan los requerimientos de tiempo para asegurarse de que el dispositivo de pueda almacenar valores de manera confiable como lo indica la tabla de verdad, a este conjunto de requerimientos se les conoce como la disciplina dinámica. La disciplina dinámica requiere que las entradas del latch cumplan con las siguientes especificaciones de sincronización:

- $D = V_x$ se mantiene válido por un intervalo de tiempo mayor o igual a un tiempo de fijado $t_S = 2 \cdot t_{PD}$ antes de que ocurra una transición decreciente de G.
- $D = V_x$ se mantiene válido por un intervalo de tiempo mayor o igual a un tiempo de espera $t_H = t_{PD}$ después de la transición decreciente de G.

En la figura 28 se ilustra el diagrama de tiempo que deben seguir las señales del latch para almacenar el valor $D = V_x$ correctamente. Se identifican 3 intervalos temporales que garantizan el cumplimiento de la tabla XI:

- Los valores $D = V_x$ y $G = 1$ deben mantenerse válidos por al menos t_{PD} antes de la transición de G, esto garantiza que $Q = Q' = V_x$ después de t_{PD}; esta condición garantiza el comportamiento de las primeras dos filas de la tabla de verdad.
- Los valores $D = V_x$ y $G = 1$ deben mantenerse válidos por al menos otro t_{PD} antes de la transición de G, esto garantiza que Q' se propagará a través del dispositivo antes de que ocurra un cambio en la entrada G; esta condición garantiza el comportamiento de las filas 3 y 4 de la tabla de verdad.
- El valor $D = V_x$ debe mantenerse válido por al menos t_{PD} luego de la transición de G, esto garantiza que las entradas $G = 0$ y $Q' = Q$ han sido válidas por al menos t_{PD} y por lo tanto son suficientes para mantener la salida $Q = V_x$ independientemente de D; esta condición garantiza el comportamiento de las últimas dos filas de la tabla de verdad.

Figura 28. **Disciplina dinámica para un latch**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Es importante observar que, cuando se realiza la transición de la entrada G, el dispositivo es capaz de mantener $Q = Q' = V_x$ si $D = V_x$. En resumen, para que un latch almacene un valor de manera confiable, es necesario mantener el valor $D = V_x$ válido por al menos $t_S = 2 \cdot t_{PD}$ antes de la transición decreciente de G, y por al menos $t_H = t_{PD}$ después de la transición de G.

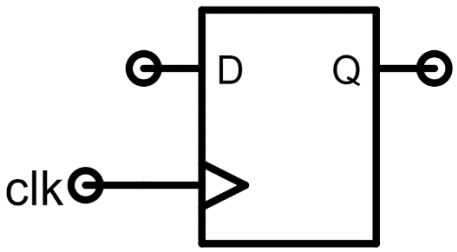
1.5.4. Flip-flop D

Un latch D es capaz de almacenar información en función del valor de la señal de control G, es decir, de un nivel lógico en particular, esta característica los hace poco convenientes para el diseño de circuitos en los que la información en la entrada D cambia rápidamente. En otras palabras, si se desea almacenar un valor $D = V_x$ dentro del latch, se debe mantener el valor $G = 1$ durante el tiempo suficiente para que cumpla con los requerimientos de la disciplina dinámica, pero debe ser lo suficientemente corto para que el latch se cierre antes de que un nuevo valor de entrada no deseado V_y sea capaz de producir cambios en el latch.

Este tipo de enfoque de diseño no es muy confiable debido a que es casi imposible conocer los instantes de tiempo en que aparecerán determinados valores de entrada. Debido a estas complicaciones, resulta bastante útil utilizar un flip-flop D para almacenar la información en un sistema digital, este dispositivo tiene la capacidad de almacenar información en un instante de tiempo específico, es decir, únicamente cuando ocurren transiciones de la señal de control.

El diagrama lógico de un flip-flop D se muestra en la figura 29, posee un aspecto similar al de un latch D a excepción del triángulo utilizado para denotar la entrada de control, debido a que se utiliza en el diseño de circuitos digitales síncronos, esta entrada se conecta normalmente a la señal de reloj.

Figura 29. **Diagrama lógico de un flip-flop D**

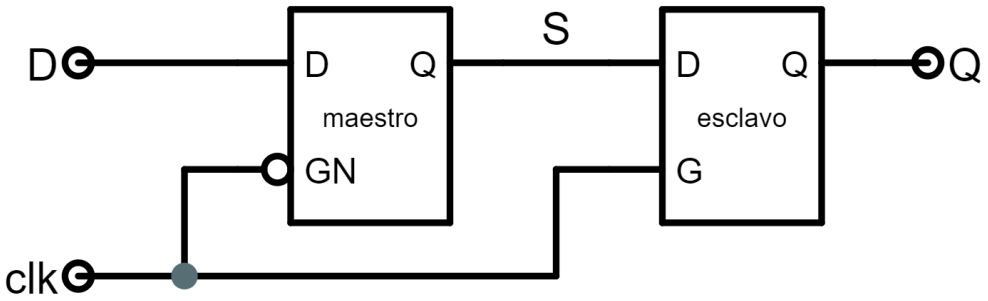


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para construir un flip-flop D se utilizan dos latches conectados en serie tal y como se muestra en la figura 30, la entrada D se conecta al latch maestro y la salida Q se obtiene del latch esclavo, la señal intermedia S se denota como la salida del latch maestro conectada a la entrada del latch esclavo y será de utilidad para verificar el cumplimiento de la disciplina dinámica en el latch esclavo. El círculo en la entrada de control del latch maestro indica que dicha entrada se encuentra negada, por lo tanto, el latch se activará cuando $GN = 0$ y se cerrará para $G = 1$; para construir un latch de este tipo únicamente es necesario intercambiar las entradas del multiplexor interno.

Utilizando este arreglo, únicamente uno de los latches se encuentra abierto a la vez, cuando $clk = 1$ el latch esclavo está abierto y el maestro está cerrado, y cuando $clk = 0$ el latch maestro está abierto y el esclavo está cerrado, por lo tanto, nunca existe una ruta directa entre la entrada D y la salida Q.

Figura 30. **Circuito interno de un flip-flop D**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

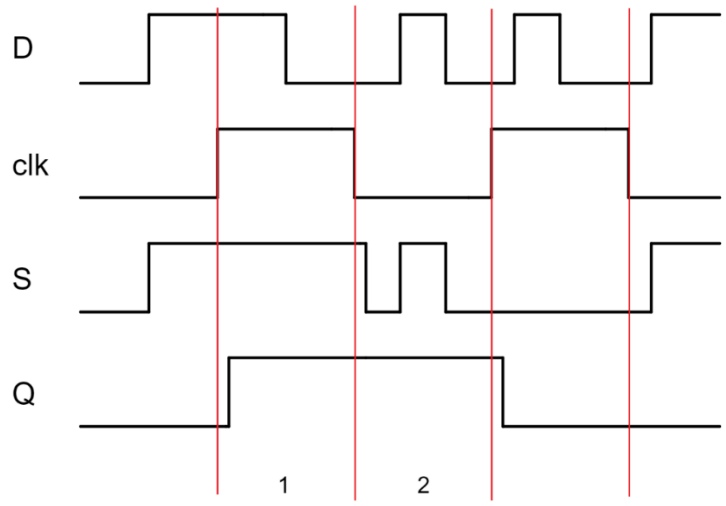
Para analizar el comportamiento del flip-flop D, considérese el diagrama de tiempos de la figura 31; para el latch maestro, cuando existe un flanco de subida del reloj clk, este se cierra y almacena el valor $S = D$ a partir de ese instante hasta que ocurre el siguiente flanco de bajada del reloj; en el intervalo en el que $clk = 0$ el latch maestro refleja en la señal S cualquier valor que exista en la entrada D.

Para el latch esclavo, se observa que después de que ocurre un flanco de subida en el reloj, este se encuentra abierto y, después de un retardo t_{PD_E} refleja en su salida el valor $Q = S$, debido a que el valor de S se mantiene estable durante este intervalo, el valor de la salida del latch esclavo es la misma a pesar de que se encuentre abierto; cuando $clk = 0$, el latch esclavo almacena el valor de S en ese instante y se abre hasta que ocurre el siguiente flanco de subida del reloj.

Es importante resaltar que en el intervalo 1, el latch maestro se encuentra cerrado y el esclavo se encuentra abierto; por el otro lado, en el intervalo 2, el latch maestro se encuentra abierto y el esclavo se encuentra cerrado, de esta manera nunca existe una ruta directa entre la entrada D y la salida Q del flip-flop.

Es sumamente necesario que el latch maestro posea un retardo de contaminación t_{CD_M} no nulo que cumpla con el siguiente requerimiento para el latch esclavo; específicamente, $t_{CD_M} \geq t_{H_E}$ en donde t_{H_E} es el tiempo de espera del latch esclavo, esta restricción evita que el valor de S cambie antes de que pueda ser almacenado por el latch esclavo.

Figura 31. **Diagrama de tiempos de un flip-flop D**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Además del latch esclavo, el latch maestro también debe cumplir con los requerimientos de la disciplina dinámica; por lo tanto, la entrada D debe mantenerse válida y estable un tiempo no menor a t_{S_M} antes del flanco de subida del reloj y t_{H_M} después del flanco de bajada del reloj, t_{S_M} y t_{H_M} son los tiempos de fijado y espera del latch maestro respectivamente. Utilizando estos requerimientos, es posible describir el comportamiento dinámico de un flip-flop D utilizando 4 parámetros. Si se hace $t_S = t_{S_M}$ y $t_H = t_{H_M}$ para denotar los tiempos de fijado y espera del flip-flop D, en ese orden, sus parámetros dinámicos se resumen de la siguiente manera:

- t_{PD}: retardo de propagación máximo después de un flanco de subida de reloj.
- t_{CD}: retardo de contaminación mínimo después de un flanco de subida de reloj.
- t_S: tiempo de fijado, la señal D debe permanecer estable antes de un flanco de subida de reloj. Garantiza que la entrada D se ha propagado a través del latch maestro antes de que este se cierre.
- t_H: tiempo de espera, la señal D debe permanecer estable después de un flanco de subida de reloj. Garantiza que el maestro se ha cerrado y la información almacenada es estable antes de permitir que D cambie.

Ahora que se tiene una descripción completa del comportamiento general e interno del flip-flop D, es posible utilizarlo como un bloque de diseño en circuitos digitales síncronos de mayor complejidad. En la siguiente sección se muestra su uso en el diseño de máquinas de estados finitos.

1.6. Máquinas de estados finitos, FSM

Las máquinas de estados finitos o FSM son un modelo bastante conveniente para implementar sistemas secuenciales que poseen una cantidad finita y pequeña de estados discretos. La estructura de una FSM es idéntica a la de un circuito secuencial síncrono de propósito general, como el que se muestra en la figura 24. Consta de un dispositivo de memoria encargado de almacenar el estado de la máquina y un circuito combinacional cuya función es producir las salidas y el siguiente estado en función de las entradas y el estado actual de la máquina.

En general, una FSM es un dispositivo que posee:

- Un número k finito de estados discretos, $\{S_0, \ldots, S_{k-1}\}$, uno de los cuales se designa como el estado inicial S_i;
- Un número m finito de entradas digitales $\{I_0, \ldots, I_{m-1}\}$;
- Un número n finito de salidas digitales $\{O_0, \ldots, O_{n-1}\}$;
- Un conjunto de reglas de transición que especifican el siguiente estado $s'(s, I_0, \ldots, I_{m-1})$ para cada estado s y cada combinación de entradas $I_0, \ldots, I_{m-1}$;
- Un conjunto de reglas de salida que especifican los valores de salida $O_0(s, I_0, \ldots, I_{m-1}), \ldots, O_{n-1}(s, I_0, \ldots, I_{m-1})$ para cada estado s y cada combinación de entradas $I_0, \ldots, I_{m-1}$.

La máquina comienza en el estado inicial S_i. En cada flanco de subida del reloj, la máquina realiza la transición hacia el siguiente estado $s'(s, I_0, \ldots, I_{m-1})$ en función del estado actual s y la combinación de entradas actuales $I_0, \ldots, I_{m-1}$. Los valores de salida dependen del estado actual s y opcionalmente, de la combinación de entradas actuales $I_0, \ldots, I_{m-1}$.

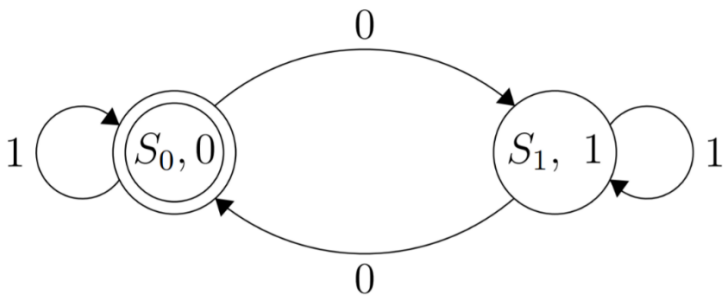
1.6.1. Diagramas de transición de estados, STD

Para especificar la funcionalidad de una FSM se utiliza un diagrama de transición de estados o STD, el cual consta de una cantidad k de nodos que poseen una correspondencia biunívoca con los estados de la máquina, cada nodo posee una flecha de transición única hacia el siguiente nodo o estado para cada posible combinación de entradas.

En la figura 32 se ilustra el caso particular de un STD con una entrada, una salida y 2 estados, en esta máquina, las salidas dependen únicamente del estado actual s de la FSM; en cada nodo se especifica el nombre del estado y el valor de salida, para este caso particular se observa que los estados son S_0 y S_1 cuyas

salidas son 0 y 1 respectivamente. En cada flecha se indica el valor de la entrada que provoca la transición; es necesario dibujar las flechas para todas las transiciones y asegurarse de que se hayan considerado todas las combinaciones de entradas.

Figura 32. **Ejemplo de un STD**



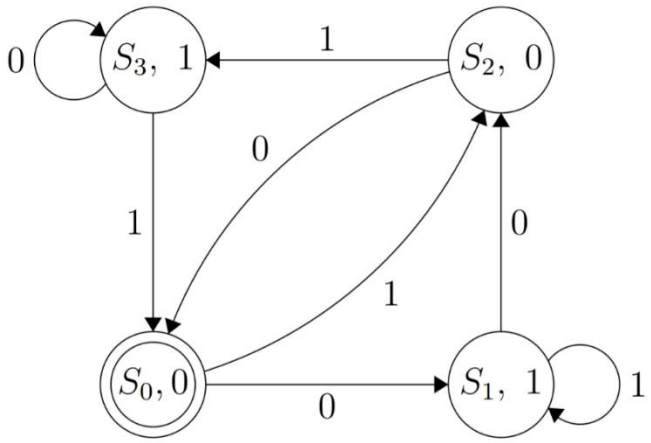
Fuente: elaboración propia, empleando LaTeX.

En un STD se utiliza una circunferencia interna para denotar el estado inicial S_i, para este caso $S_i = S_0$. El comportamiento de una FSM puede deducirse fácilmente a partir de su STD, en el ejemplo anterior, la máquina inicia en el estado S_0 y se mantiene en ese estado siempre y cuando la entrada sea igual a 1, cuando la entrada es igual a 0, la máquina hace la transición hacia el estado S_1; el estado S_1 responde de la misma manera ante las mismas entradas, con la diferencia de que la transición se hace de vuelta al estado S_0. En resumen, el comportamiento de la máquina consiste en un conmutador que cambia de estado cuando la entrada es igual a 0, de lo contrario el valor de salida se almacena indefinidamente.

Es importante resaltar que el STD expuesto en esta subsección corresponde a una máquina cuyos valores de salida únicamente dependen de

los valores de entrada, esto no siempre es así y, como se verá en las siguientes subsecciones, existen otros tipos de FSM en las que sus salidas son funciones tanto del estado actual como de la actual combinación de entradas, lo que provoca que la notación para los diagramas de estado cambie ligeramente.

Figura 33. **STD de una máquina de Moore**



Fuente: elaboración propia, empleando LaTeX.

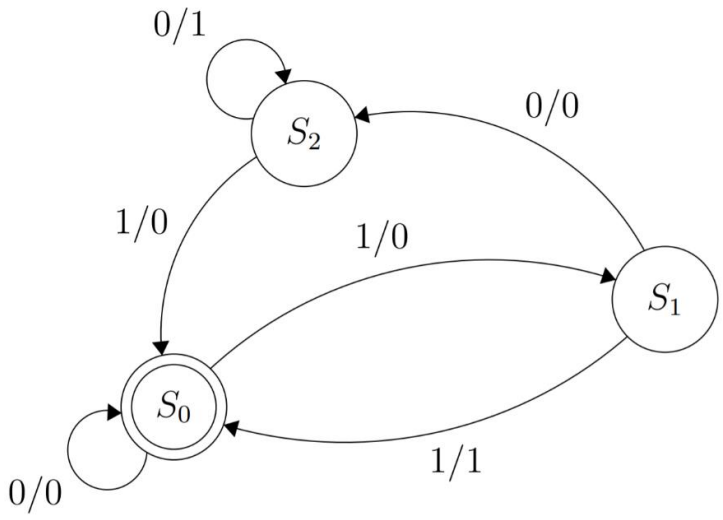
1.6.2. Máquinas de Moore

El STD de la figura 32 corresponde a una máquina de Moore, este tipo de FSM posee la principal característica de que los valores de las salidas digitales son únicamente una función del estado actual s. Para el STD, los valores de salida se colocan en el centro de los nodos, a un lado del estado, mientras que las entradas que provocan las transiciones se colocan al lado de las flechas. En la figura 33 se observa otro ejemplo de un STD de una máquina de Moore con 4 estados, 1 entrada y 1 salida; en este caso también se observa que las salidas dependen únicamente del estado actual.

1.6.3. Máquinas de Mealy

Una máquina de Mealy es un tipo particular de FSM cuya característica principal es que la salida es una función del estado actual s y de la combinación de entradas $I_0, \ldots, I_{m-1}$. Puesto que el valor de salida también depende del valor de la entrada, el valor de la salida ya no se coloca dentro del nodo, al lado de cada entrada se coloca una diagonal seguida del valor de salida que corresponde con dicha entrada, tal y como se muestra en el STD de la figura 34.

Figura 34. **STD de una máquina de Mealy**



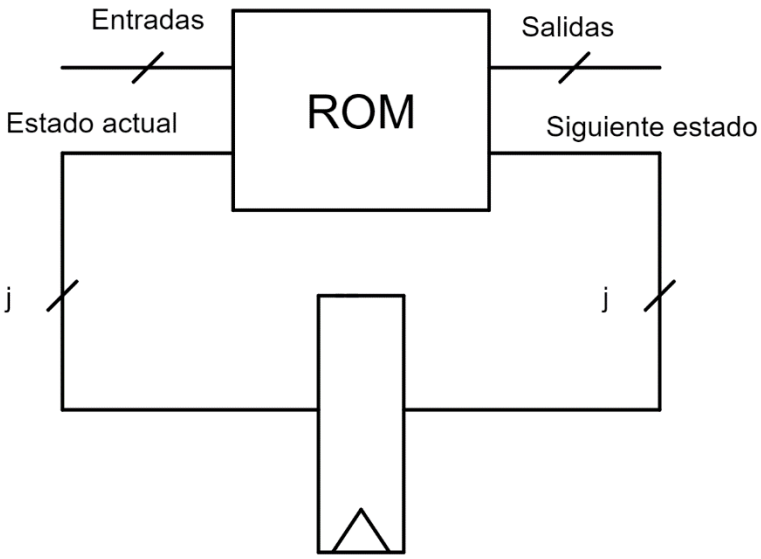
Fuente: elaboración propia, empleando LaTeX.

En este trabajo, cuando se hace referencia a una máquina de estados, se asume que se está utilizando una máquina de Moore, salvo cuando sea necesario, se indicará claramente cuando se trate de una máquina de Mealy.

1.6.4. Implementación de una máquina de estados

Un STD se puede convertir fácilmente en un circuito secuencial síncrono que implemente la funcionalidad de la FSM que representa. El procedimiento se basa en la codificación de los k estados discretos utilizando números binarios almacenados en un registro de j bits como se muestra en la figura 35, se utiliza una ROM como bloque combinacional para producir las salidas y el siguiente estado dependiendo de la funcionalidad de la FSM.

Figura 35. **Implementación de una FSM en hardware**

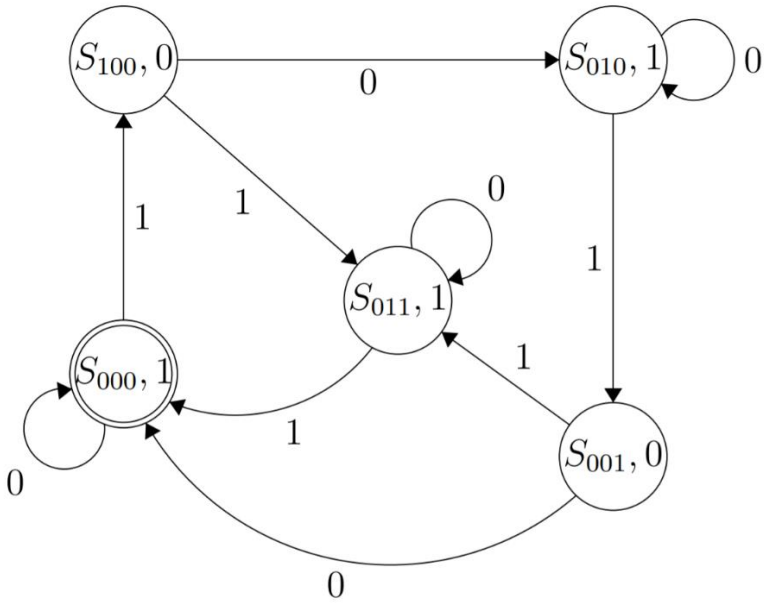


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Si se desea representar a todos los estados discretos k de la máquina sin ambigüedades utilizando cadenas únicas de j bits, se debe cumplir la condición $2^j \geq k$, es decir, el número de estados discretos debe ser menor o igual al número de combinaciones posibles de una cadena de j bits.

Considérese el caso del diagrama con 5 estados, 1 entrada y 1 salida de la figura 36, este diagrama utiliza cadenas binarias de 3 bits para codificar los estados, obsérvese que $2^3 = 8 \geq 5$, por lo tanto, es posible asignar a cada estado una combinación única utilizando 3 bits. El esquema de codificación para este caso particular, utiliza la correspondencia entre estados con numeración decimal y su equivalente en binario, de esta manera $S_0 = 000$, $S_1 = 001, \ldots, S_4 = 100$.

Figura 36. **STD de una máquina con 5 estados**



Fuente: elaboración propia, empleando LaTeX.

Para programar el contenido de la ROM es necesario trasladar el STD a una representación en forma de tabla de verdad, las variables I_0 y $s[2:0]$ representan la única entrada digital y el estado actual respectivamente, por otra parte, las variables O_0 y $s'[2:0]$ representan la única salida digital y el estado siguiente respectivamente. En la tabla XII se observa la información del STD convertida en una tabla de verdad en la que sólo se consideran las combinaciones de entradas

asociadas a los estados definidos en el diagrama, utilizando el enfoque de diseño de la subsección 1.4.5 se procede a programar los contenidos de la ROM, colocando los valores de salida asociados con las direcciones que no aparecen en la tabla como términos X.

Tabla XII. **Tabla de verdad de un STD**

I_0	$s[2]$	$s[1]$	$s[0]$	$s'[2]$	$s'[1]$	$s'[0]$	O_0
0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1
0	0	0	1	0	0	0	0
1	0	0	1	0	1	1	0
0	0	1	0	0	1	0	0
1	0	1	0	0	0	1	0
0	0	1	1	0	1	1	1
1	0	1	1	0	0	0	1
0	1	0	0	0	1	0	0
1	1	0	0	0	1	1	0

Fuente: elaboración propia.

Para implementar el diseño de la FSM se necesita una ROM con 4 entradas de dirección y 4 líneas de salida, es decir, una memoria con una capacidad de $16 \times 4 = 64$ bits, más adelante se verá que esto es equivalente a una memoria con una capacidad de 8 bytes. Este enfoque no es el único ni el óptimo, pero es uno de los que más agiliza el proceso de diseño, si el lector lo desea, puede utilizar los métodos de las subsecciones 1.4.1 y 1.4.2 para realizar el diseño de la lógica combinacional optimizada.

1.7. Parámetros de desempeño

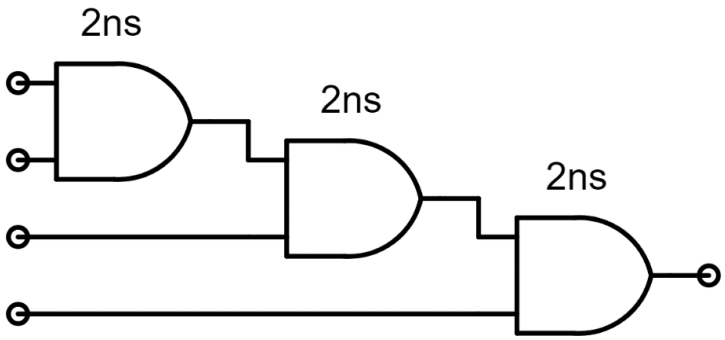
En el diseño de sistemas digitales, existen varios enfoques de diseño con los que se pueden crear diferentes circuitos que implementen la misma funcionalidad, debido a esto se utilizan dos parámetros dinámicos capaces de medir la velocidad de un circuito en particular: latencia y rendimiento. Utilizando estos parámetros, es posible realizar una comparación entre dos circuitos y decidir cuál resulta más conveniente para una aplicación determinada.

1.7.1. Latencia

La latencia L de un circuito digital se define como el límite superior del intervalo de tiempo que el sistema tarda en producir salidas válidas dada una combinación de entradas válidas. En otras palabras, la latencia mide el tiempo de respuesta de un circuito para una entrada particular.

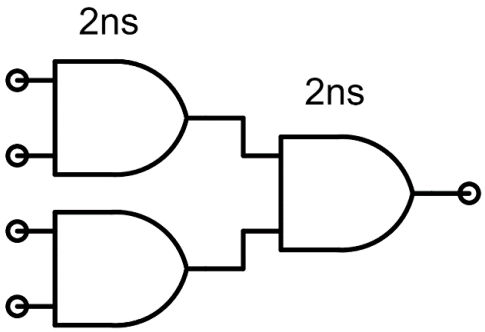
En circuitos combinacionales, la latencia es equivalente al retardo de propagación t_{PD}; en la figura 37 se muestra una compuerta AND de 4 entradas utilizando una conexión en cascada mientras que en la figura 38 se presenta una implementación alternativa de la misma compuerta utilizando una conexión en forma de árbol, en ambos diagramas se asume que las compuertas AND de 2 entradas poseen un retardo de propagación $t_{PD_x} = 2\ ns$. Al calcular los retardos de propagación de cada uno de los circuitos se observa que el circuito en cascada posee una latencia mayor, $L = t_{PD} = 6\ ns$, a la del circuito con conexión tipo árbol, $L = t_{PD} = 4\ ns$, y, por lo tanto, la conexión en forma de árbol posee un tiempo de respuesta menor.

Figura 37. **Compuerta AND de 4 entradas en cascada**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Figura 38. **Compuerta AND de 4 entradas en paralelo**

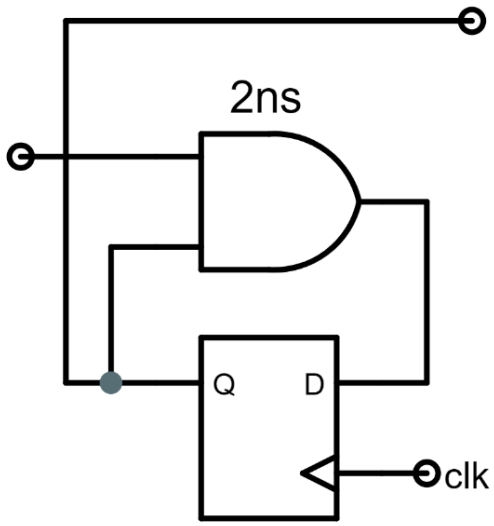


Fuente: elaboración propia, empleando Jade Circuit Simulator.

En circuitos secuenciales, la latencia es equivalente a algún múltiplo entero n del período de la señal de reloj T_{clk}, específicamente $L = n \cdot T_{clk}$. En todos los circuitos secuenciales, el período de reloj deberá ser mayor o igual a la latencia máxima entre todos los dispositivos o segmentos combinacionales, es decir, $T_{clk} \geq t_{PD_{máx}}$.

Considérese el caso de la figura 39 en donde se ilustra un circuito secuencial síncrono de 1 entrada y 1 salida con un flip-flop D o registro de 1 bit, el retardo de propagación de la compuerta AND es de $t_{PD} = 2\ ns$ y para simplificar el análisis, se asume que el registro cuenta con un retardo de propagación despreciable. Como a salida del circuito se toma directamente de la salida Q del registro y esta se actualiza en cada flanco de subida de reloj, es decir, cada T_{clk} segundos, la latencia del sistema es $L = T_{clk}$ siempre y cuando se cumpla que $T_{clk} \geq 2\ ns$.

Figura 39. **Circuito secuencial con 1 flip-flop**

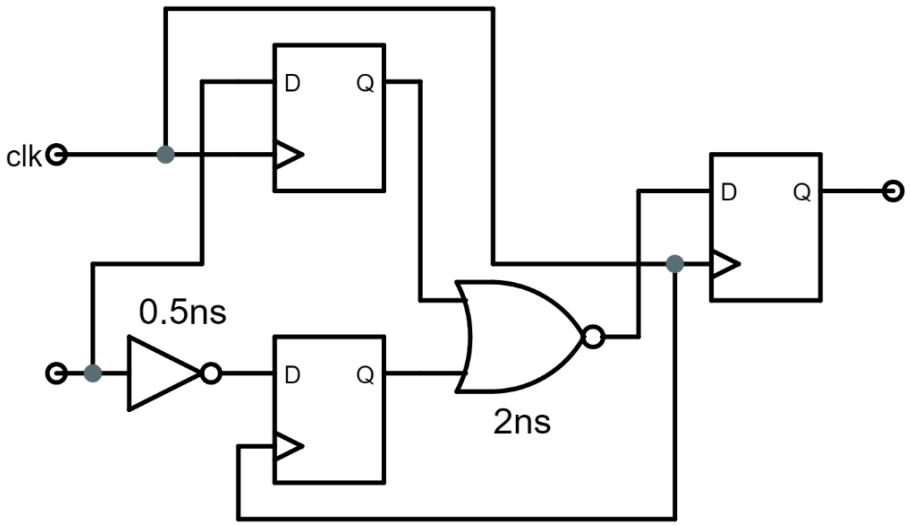


Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la figura 40 se muestra otro circuito secuencial con 1 entrada, 1 salida y 3 registros de 1 bit, los tiempos de propagación se indican para cada dispositivo combinacional, se asume de nuevo que los registros poseen un retardo despreciable. En este circuito, la entrada y su negación se encuentran conectadas a las entradas de dos registros en paralelo, las salidas de estos

registros se conectan a la entrada de la compuerta NOR, por último, la salida de la compuerta se conecta a un último registro cuya salida Q conforma la salida del circuito.

Figura 40. **Circuito secuencial con 3 flip-flops**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Debido a que las salidas de los registros cambian de valor únicamente cuando hay un flanco de subida del reloj, el valor de salida para una entrada particular tarda 2 ciclos de reloj en producirse, nótese que los dos registros de la izquierda transfieren la información en un mismo ciclo. La latencia del circuito es $L = 2 \cdot T_{clk}$, en donde $T_{clk} \geq 2\ ns$ debido a que la compuerta NOR es el dispositivo combinacional que posee el retardo de propagación máximo.

1.7.2. Rendimiento

El rendimiento R de un circuito digital se define como la tasa de producción de salidas a partir de nuevas entradas por unidad de tiempo. En circuitos combinacionales, el rendimiento R_C es equivalente al recíproco de la latencia, es decir,

$$R_C = \frac{1}{L} = \frac{1}{t_{PD}}$$

En el caso del circuito de la figura 38, el rendimiento es equivalente a $\frac{1\ salida}{4\ ns}$, en otras palabras, el circuito es capaz de producir 250 mil salidas en 1 milisegundo; por otra parte, el circuito de la figura 37 es capaz de producir alrededor de 167 mil salidas cada milisegundo.

Para circuitos secuenciales, el rendimiento R_S es equivalente al recíproco del período de reloj T_{clk}, esto se deriva del hecho de que el circuito produce una salida nueva en cada flanco de subida del reloj, por lo tanto,

$$R_S = \frac{1}{T_{clk}}$$

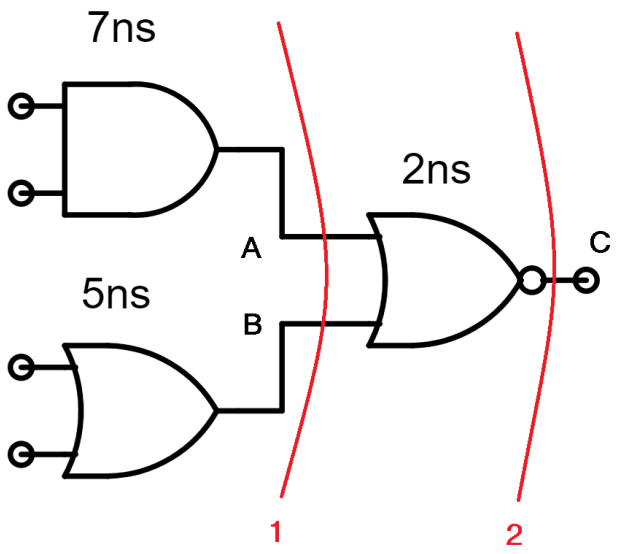
Retomando el caso del circuito de la figura 40, se observa que si se escoge el período de reloj mínimo $T_{clk} = 2\ ns$ el rendimiento es equivalente a $\frac{1\ salida}{2\ ns}$, es decir, 250 mil salidas por milisegundo; si se aumenta el valor de T_{clk} por encima del valor mínimo es posible apreciar una disminución en el rendimiento, si $T_{clk} = 4\ ns$ el rendimiento se vería reducido a la mitad, 125 mil salidas por milisegundo. En general, el rendimiento y la latencia son 2 cantidades intercambiables, es decir, la variación entre las variables es inversamente proporcional. A pesar de

esto, existen técnicas de diseño de circuitos que permiten, en algunos casos, obtener rendimientos mayores manteniendo la latencia se mantiene constante.

1.7.3. Segmentación o pipelining

La segmentación o pipelining es una técnica de diseño que permite incrementar el valor del rendimiento R de un circuito digital utilizando una cantidad k de registros de longitud variable conectados a las salidas de algunos o todos los dispositivos combinacionales que lo componen.

Figura 41. **Etapas en un circuito combinacional**



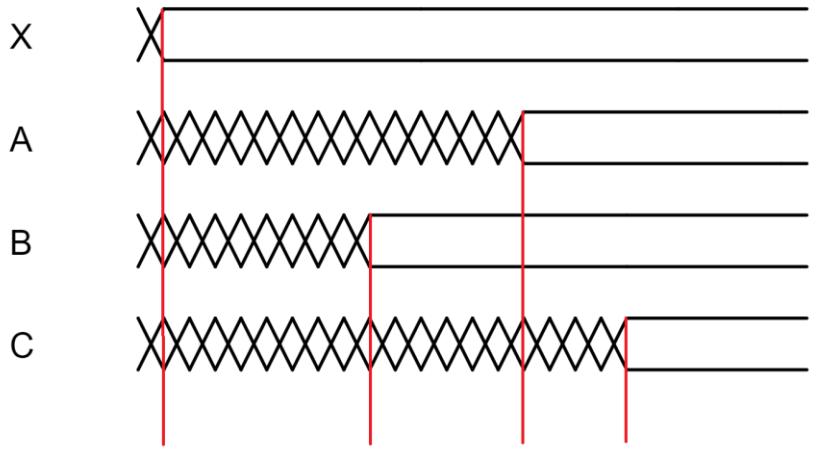
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Supóngase el caso de un circuito combinacional como el que se ilustra en la figura 41, el circuito posee 3 compuertas con su respectivo retardo de propagación. Cuando una combinación de entradas $X[3:0]$ válida se ha mantenido estable por al menos $t_{PD_1} = 7\,ns$, se dice que la información se ha

propagado desde las entradas hacia la primera etapa del circuito denotada por la primera línea roja, la cual contiene a las señales A y B, cuando la información de la primera etapa se ha mantenido estable por al menos $t_{PD_2} = 2\ ns$, la información se propaga hacia la segunda etapa y, por ende, la salida C del circuito, esta se indica a través de la segunda línea roja.

En la figura 42 se ilustra un bosquejo del diagrama de tiempos para las señales A, B, C y para una combinación de entradas arbitraria X; obsérvese que durante $t_{PD_1} = 7\ ns$ las compuertas de la primera etapa están ocupadas determinando los valores de salida A y B, debido a esto las entradas de la última compuerta son inválidas y esta no puede determinar su valor de salida, pasado este intervalo, la salida C será válida después de $t_{PD_2} = 2\ ns$, durante esta etapa, las compuertas de la primera etapa sólo se encargan de mantener las señales A y B estables.

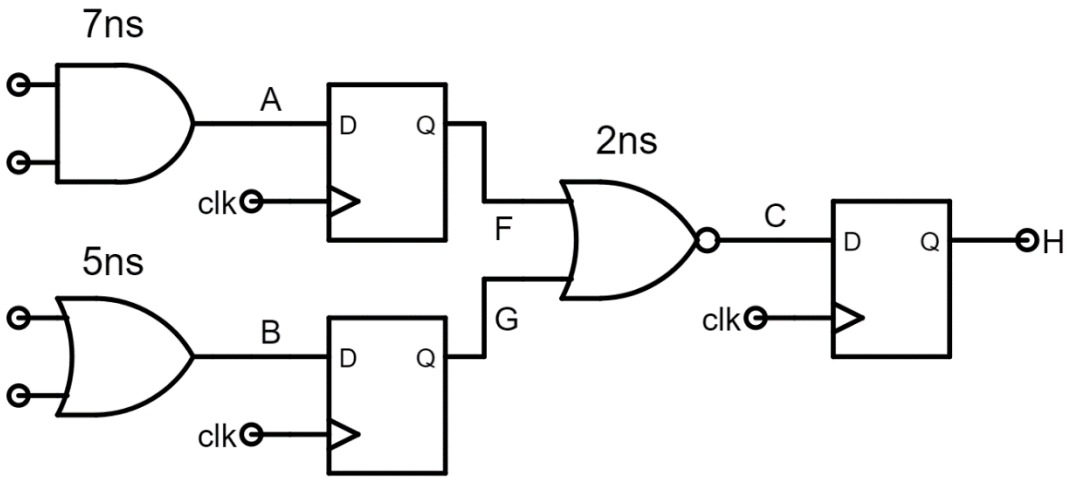
Figura 42. **Diagrama de tiempos de un circuito combinacional**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Si se deseara introducir una entrada X_{i+1} durante la segunda etapa del cálculo de la salida producida por X_i se necesitaría introducir registros encargados de almacenar los valores de salida de cada etapa permitiendo así, desacoplar las etapas del circuito combinacional formando un circuito con un pipeline de 2 etapas. En la figura 43 se ilustra esta idea, para la primera etapa se utiliza un registro de 2 bits mientras que en la segunda etapa se utiliza un registro de 1 bit, de nuevo se asume que los retardos de los registros son despreciables y que todos los registros comparten una señal de reloj común.

Figura 43. **Pipeline de 2 etapas**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Si se carga una entrada X_i cerca del inicio del ciclo de reloj i, los valores A_i y B_i se cargarán al registro de 2 bits al inicio del ciclo $i+1$, en donde permanecerán hasta que el registro se actualice al inicio del ciclo $i+2$ cuando el valor de C_i sea cargado al registro de la etapa final.

El período de reloj se escoge de manera que se cumpla con $T_{clk} \geq 7\ ns$, debido a que la compuerta AND limita el período mínimo posible, de ser menor el período, dicha compuerta no tendría tiempo suficiente para producir salidas válidas. Es importante resaltar que la latencia del circuito es de $14\ ns$ la cual, es mayor a la latencia del circuito sin registros ($9\ ns$); a pesar de esto, el rendimiento del circuito es de $\frac{1\ salida}{7ns}$ el cual es equivalente a la frecuencia del reloj ($143\ MHz$), un rendimiento significativamente mayor al del circuito sin registros ($\frac{1\ salida}{9\ ns}$). A pesar de que la latencia ha aumentado, este efecto se ve atenuado por un crecimiento en el rendimiento del circuito, demostrando así, que estas dos cantidades son intercambiables.

Tabla XIII. **Diagrama de un pipeline de 2 etapas**

		Ciclo de reloj				
		$\boldsymbol{i}$	$\boldsymbol{i+1}$	$\boldsymbol{i+2}$	$\boldsymbol{i+3}$	. . .
Etapa	**Entrada**	X_i	X_{i+1}	X_{i+2}	. . .	. . .
	Registro 1		F_1 G_1	F_{i+1} G_{i+1}	F_{i+2} G_{i+2}	. . .
	Registro 2			H_1	H_{i+1}	H_{i+2}

Fuente: elaboración propia.

En la tabla XIII se muestra un diagrama un diagrama del pipeline del circuito de la figura 43 que permite visualizar el flujo de la información a través de las diferentes etapas del circuito. El circuito tarda 2 ciclos de reloj para procesar una entrada particular, sin embargo, tiene la capacidad de procesar una nueva entrada en cada ciclo de reloj permitiendo así, que se trabaje en varias entradas independientes de manera simultánea. Cuando todas las etapas del circuito se encuentran funcionando simultáneamente, se dice que el pipeline se encuentra

lleno, para este caso esto sucede 2 ciclos de reloj después de que se introduce la primera entrada.

Existe mucho más que decir acerca de la técnica de segmentación para incrementar el rendimiento de un circuito digital, sin embargo, los detalles de la metodología sistemática de diseño quedan fuera del alcance del presente trabajo. Si el lector lo desea, puede encontrar más información acerca del tema en el capítulo 8 del texto *Computation Structures,* de Stephen Ward y Robert Halstead, para aplicar esta técnica en futuras modificaciones del diseño del procesador.

1.8. Dispositivos lógicos programables

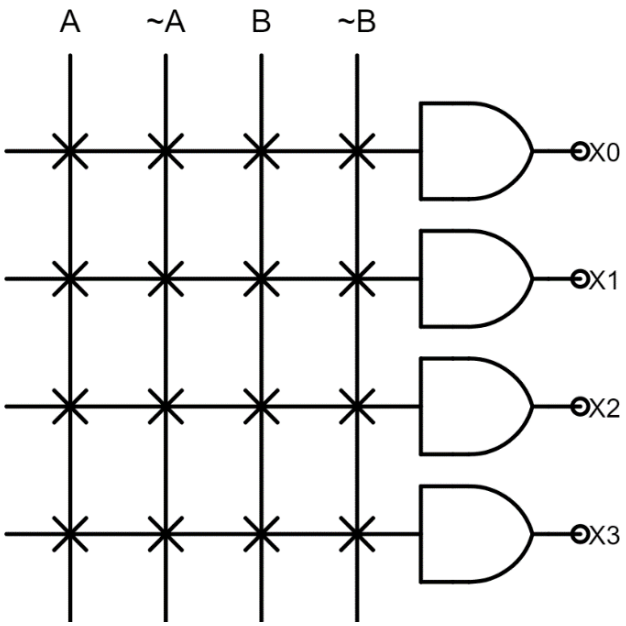
En la subsección 1.4.5 se expuso un método para la implementación sistemática de circuitos combinacionales programando individualmente las celdas de una ROM en base a una especificación funcional predeterminada. La ROM forma parte de un amplio grupo de dispositivos fabricados y revisados que se pueden personalizar desde el exterior utilizando diferentes técnicas de programación, a este grupo de dispositivos se les conoce como dispositivos lógicos programables o PLD.

Existen 2 tipos diferentes de PLD según la cantidad de recursos de hardware que este posea: de bajo y alto nivel de integración. Los PLD de bajo nivel de integración constan de arreglos de compuertas que pueden ser fijos o programables mientras que los de alto nivel de integración se encuentran estructurados a través de bloques lógicos configurables y celdas lógicas de alta densidad.

La estructura básica de un PLD de bajo nivel de integración consiste en arreglos de compuertas AND y OR conectados a las entradas y salidas del dispositivo, dichos arreglos pueden ser programables o fijos.

En la figura 44 se muestra la estructura de un arreglo AND de 4 entradas y 4 salidas sin programar utilizando la notación alternativa de compuertas presentada en la subsección 1.4.5. El arreglo se programa quemando los fusibles de las intersecciones de manera permanente definiendo así, los valores de salida de las compuertas.

Figura 44. **Arreglo AND no programado**



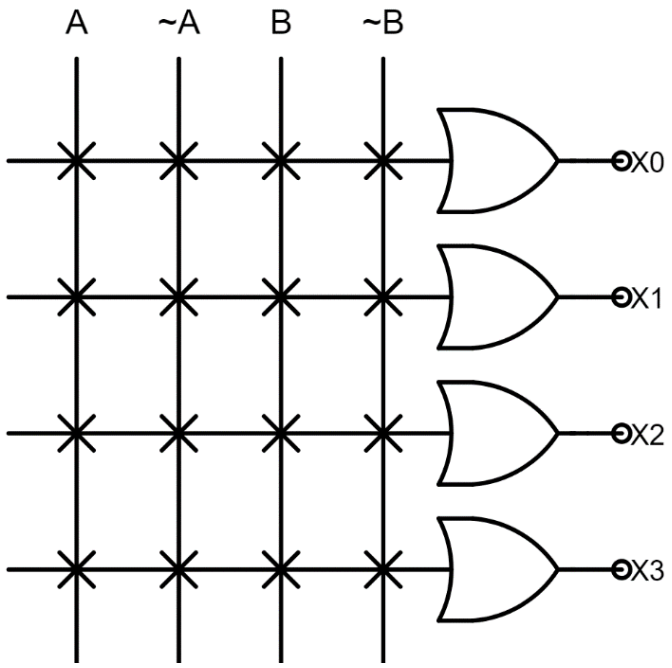
Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la figura 45 se muestra la estructura de un arreglo OR de 4 entradas y 4 salidas sin programar, este arreglo también se programa quemando los fusibles de las intersecciones de manera permanente.

En general, la estructura de un arreglo de n entradas y m salidas tendrá una n cantidad de columnas conectadas a las entradas de m compuertas, ya sea AND u OR, según sea el caso. Es necesario aclarar que estos dispositivos sólo pueden

programarse una vez, por ende, los PLD construidos a partir de estos bloques sólo son programables una única vez.

Figura 45. **Arreglo OR no programado**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

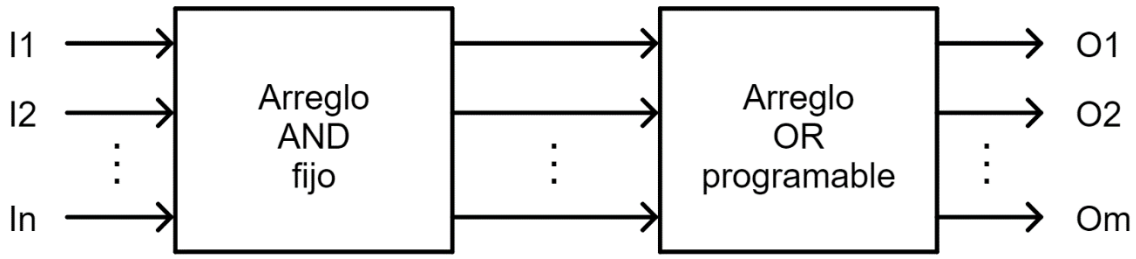
A partir de estos dos arreglos, es posible crear una variedad de dispositivos programables capaces de implementar funciones booleanas como sumas de productos, la estructura de cada uno de estos dispositivos se describe en las subsecciones siguientes.

1.8.1. PROM

Una ROM programable o PROM está formada por un arreglo de compuertas AND fijas dispuesto en forma de decodificador y un arreglo programable OR. El

esquema básico de este dispositivo se muestra en la figura 46, en donde los arreglos se representan como bloques con entradas y salidas, con las salidas del arreglo AND conectadas a las entradas del arreglo OR. Esta definición concuerda con la descripción de la ROM en la subsección 1.4.5, por lo tanto, en el presente trabajo se utilizará tanto PROM como ROM para referirse al mismo dispositivo programable.

Figura 46. **Esquema básico de una PROM**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

En general, una PROM que contiene n entradas o líneas de dirección y m salidas es capaz de implementar 2^{2n} funciones booleanas diferentes en cada salida; en otras palabras, este dispositivo es equivalente a m circuitos lógicos separados, cada uno siendo capaz de calcular únicamente una de 2^{2n} posibles funciones booleanas de n entradas.

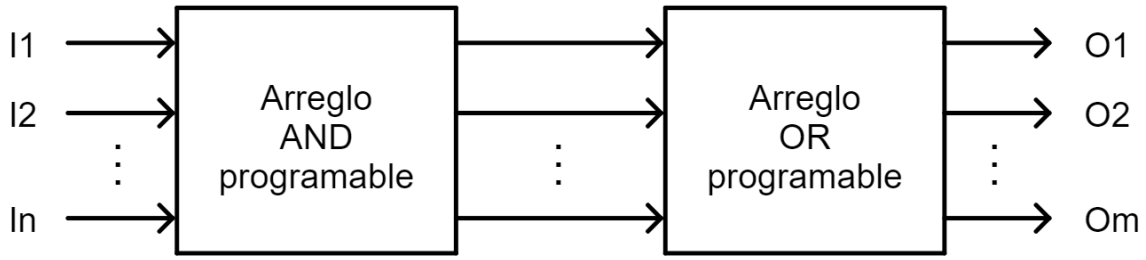
Sin embargo, estos dispositivos poseen una latencia relativamente alta en comparación a los circuitos lógicos de propósito específico, los cuales se encuentran optimizados para una aplicación en particular. Otra desventaja es que regularmente se ocupa poco espacio de la memoria en una sola aplicación, por lo que se hace un uso ineficiente de los recursos de hardware. La mayoría de sus usos se encuentran en el almacenamiento direccionable de datos, pero cuando

sea conveniente, se utilizará para simplificar el diseño de algún bloque combinacional complejo, como la unidad de control del procesador.

1.8.2. PLA

Un arreglo lógico programable o PLA es un dispositivo que está formado por un arreglo AND y un arreglo OR, ambos programables. En la figura 47 se muestra el esquema básico de este PLD para una cantidad de n entradas y m salidas.

Figura 47. **Esquema básico de un PLA**



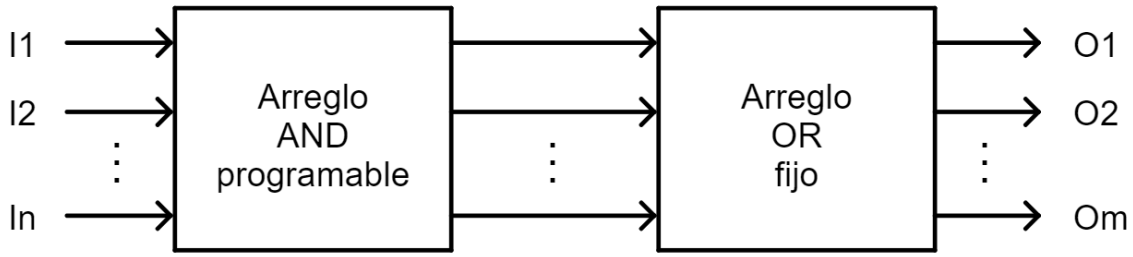
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este tipo de PLD es mucho más flexible debido a que ambos arreglos de compuertas AND y OR son programables, permitiendo así, implementar fácilmente circuitos digitales cuyo diseño esté basado en la representación estándar de suma de productos de alguna función booleana. Sin embargo, una de las principales desventajas de este dispositivo es que su latencia y su tamaño sean significativamente mayores a los de otros PLD.

1.8.3. PAL

Un PAL o lógica de arreglos programable es un PLD que posee una arquitectura más sencilla, consiste en la conexión de un arreglo AND programable y un arreglo OR fijo. En la figura 48 se muestra el esquema básico de este dispositivo para una cantidad arbitraria de n entradas y m salidas.

Figura 48. **Esquema básico de un PAL**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Gracias al arreglo OR fijo, estos dispositivos poseen las características de ser más rápidos, más pequeños y más baratos que los dispositivos PLA, esto ocurre, a cambio de una disminución en la flexibilidad de configuración.

1.8.4. FPGA

Los dispositivos mencionados anteriormente, PROM, PLA y PAL, se encuentran dentro de la categoría de dispositivos lógicos programables de bajo nivel de integración, estos dispositivos no tienen la capacidad de sintetizar circuitos secuenciales sin la necesidad de introducir registros adicionales, además de que la densidad de compuertas es relativamente baja si se considera

que se desea implementar un sistema digital complejo como lo es un procesador de 32 bits.

Los PLD de alto nivel de integración permiten integrar una cantidad mucho mayor de dispositivos en un chip. Estos dispositivos se caracterizan principalmente por la reducción del área que ocupa el conjunto de dispositivos y una notable reducción en el costo, además de ofrecer una mejora en el diseño de sistemas digitales complejos debido a que pueden operar a velocidades y frecuencias de reloj mucho mayores.

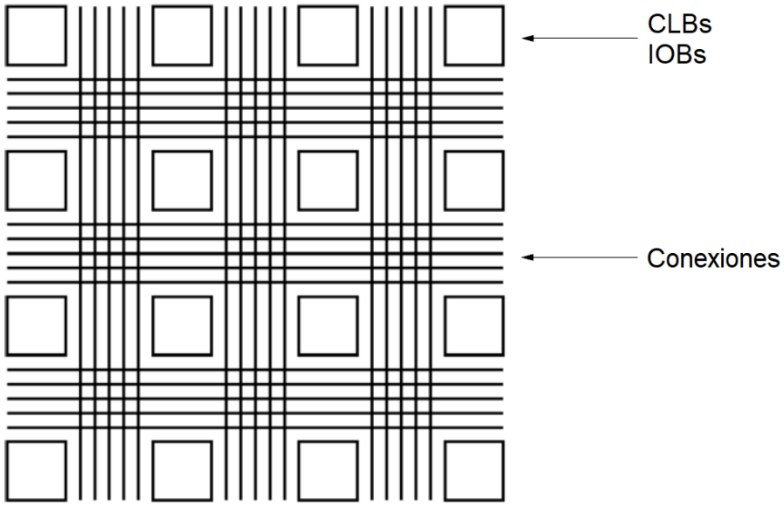
Estos dispositivos permiten al diseñador reducir los costos de diseño y producción debido a que son reconfigurables y reprogramables en campo. Por lo tanto, se puede editar y reprogramar el diseño las veces que sea necesario.

Un arreglo de compuertas programables en campo o FPGA se basa en arreglos de compuertas, los cuales contienen tres elementos configurables: bloques lógicos configurables, o CLB, bloques de entrada y salida, o IOB, y canales de comunicación. En la figura 49 se muestra el diagrama de la estructura básica de una FPGA, los CLB e IOB se encuentran dispuestos en un arreglo bidimensional y se interconectan de alguna manera utilizando una estructura de interconexión compleja, es importante resaltar que solamente es un bosquejo y que el cableado que se muestra no es representativo de ninguna arquitectura en particular.

En términos más generales, únicamente existen dos tipos de recursos en una FPGA: lógica e interconexiones. La lógica es donde se implementan subcircuitos combinacionales y secuenciales mientras que las interconexiones se encargan de conectar los diferentes resultados desde un bloque lógico hacia otro. En las siguientes subsecciones se hará una descripción breve de la estructura

básica de los CLB y del mecanismo de interconexión de bloques, esto permitirá describir la estructura de una FPGA con mayor detalle.

Figura 49. **Estructura básica de una FPGA**



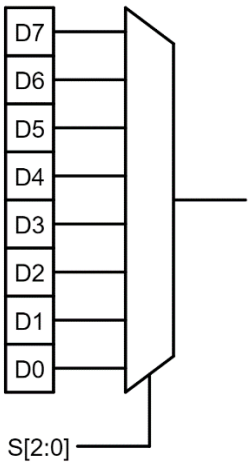
Fuente: HAUCK, Scott; DEHON, Andre. *Reconfigurable computing: The theory and practice of FPGA-based computation*. p. 7.

1.8.4.1. Bloques lógicos configurables, CLB

En la sección 1.4 se explicó que todas las funciones booleanas pueden representarse a través de una tabla de verdad, de hecho, muchas de las técnicas de diseño de circuitos digitales expuestas con anterioridad se basan en una especificación funcional en forma de tabla de verdad. Un elemento de hardware que puede implementar fácilmente cualquier función arbitraria de n entradas es la tabla de búsqueda o lookup table, comúnmente abreviado como LUT, este dispositivo se considera como uno de los bloques fundamentales de hardware.

Una LUT de n entradas se puede construir a partir de un multiplexor con n líneas de selección cuyas entradas de datos se encuentran conectadas a las celdas de memoria de un dispositivo de memoria de 2^n bits. En la figura 50 se muestra una LUT de 3 entradas, este dispositivo puede implementar cualquiera de las 2^8 funciones booleanas de 3 entradas, únicamente es necesario programar los valores adecuados en las celdas de la memoria.

Figura 50. **LUT de 3 entradas**



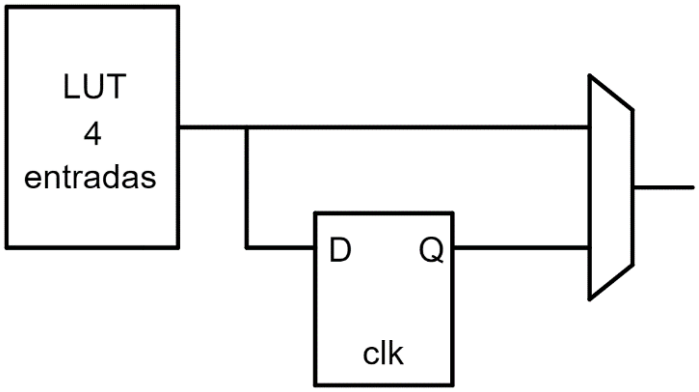
Fuente: elaboración propia, empleando Jade Circuit Simulator.

El tamaño óptimo, en términos de área y latencia, para una LUT es de 4 entradas, aunque existen dispositivos que poseen arquitecturas con LUT de 6 entradas, tal es el caso de la familia de dispositivos Virtex-5 lanzada por Xilinx, uno de los principales fabricantes y distribuidores de FPGA en el mundo.

A pesar de su capacidad para implementar cualquier función booleana de n entradas, las LUT son dispositivos que no pueden almacenar el estado del circuito y, por ende, implementar lógica secuencial, debido a que son dispositivos

combinacionales. Por este motivo se añade un dispositivo de almacenamiento que tiene la capacidad de operar ya sea como latch o flip-flop, ahora el dispositivo es capaz de guardar el estado del circuito. En la figura 51 se muestra la combinación de estos dos dispositivos utilizando un multiplexor, la salida de la LUT y el dispositivo de almacenamiento se conectan a las entradas de datos del multiplexor, esta es, en esencia, la estructura básica de un CLB.

Figura 51. **Estructura básica de un CLB**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

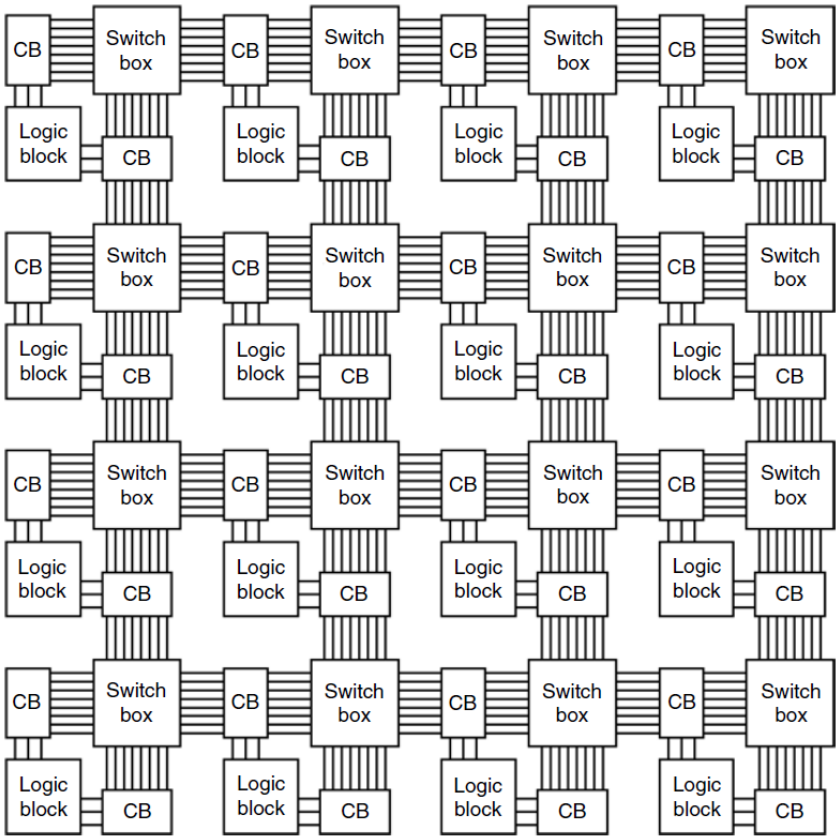
Este bloque lógico configurable posee la característica de ser reprogramable. Los puntos o nodos programables se encuentran en los contenidos de la LUT, la señal de selección del multiplexor y el estado inicial del elemento de memoria. Muchas FPGA utilizan celdas volátiles SRAM conectadas a los puntos de configuración.

1.8.4.2. Estructura de interconexión

La estructura de interconexión se encarga de proveer las rutas de conexiones apropiadas entre diferentes bloques lógicos. En la figura 52 se

muestra una descripción más detallada que la presentada en la figura 49, los bloques lógicos se encuentran ordenados de la misma manera, en forma de matriz mientras que se añaden 2 nuevos elementos configurables, los bloques de conexión o CB, y los bloques de interruptores o switch box.

Figura 52. **Estructura de interconexión de una FPGA**



Fuente: HAUCK, Scott; DEHON, Andre. *Reconfigurable computing: The theory and practice of FPGA-based computation*. p. 9.

Las salidas y entradas de los bloques lógicos se pueden conectar a las líneas verticales y horizontales a través de los CB, lo cual proporciona bastante flexibilidad en el enrutamiento de diferentes bloques. Los bloques de interruptores

permiten configurar individualmente los puntos en donde se intersecan las líneas verticales y horizontales, en general, son matrices de interruptores programables que permiten unir diferentes segmentos de líneas. Con esta estructura, es posible llegar a cualquier bloque lógico vecino con un par de CB y un bloque de interruptores.

Existen estructuras optimizadas en las que se añaden conexiones directas entre bloques lógicos vecinos que permiten reducir el número de recursos de interconexión requeridos por una aplicación en particular; también se emplean líneas cuyas longitudes son capaces de abarcar de 2 a 4 segmentos con el objetivo de reducir la latencia entre bloques lógicos lejanos. En general, la estructura es la misma, salvo por la adición de algunos matices que permiten optimizar algunos parámetros de desempeño.

Puesto que el campo de estudio de las FPGA continúa expandiéndose, este tipo de estructura no es la única que existe, de hecho, existen estructuras que utilizan un enfoque de interconexión jerárquico, si el lector lo desea, puede indagar sobre los diferentes tipos de estructuras de interconexión que existen, aunque para los propósitos del presente trabajo, esta pequeña noción de la estructura de una FPGA será suficiente.

1.8.4.3. Bloques optimizados

Con la estructura de una FPGA como la de la figura 52 se puede implementar cualquier tipo de circuito combinacional y secuencial que esté dentro de las capacidades de hardware que el dispositivo provee, sin embargo, el uso de múltiples bloques lógicos y recursos de interconexión no es muy óptimo para algunas aplicaciones en particular. Las FPGA contienen bloques optimizados para los circuitos que se implementan con mayor frecuencia en el diseño digital,

algunos ejemplos de estos bloques son: cadenas de acarreo rápido, útiles para sintetizar circuitos sumadores, multiplicadores binarios y bancos de celdas dedicadas de almacenamiento SRAM.

1.9. Lenguaje de descripción de hardware VHDL

VHDL es un lenguaje descriptivo orientado al diseño de circuitos digitales que permite describir, analizar y evaluar el comportamiento de circuitos y sistemas digitales de manera versátil y relativamente rápida en comparación con los métodos tradicionales de diseño expuestos en las secciones 1.4, 1.5 y 1.6.

Los resultados son susceptibles a ser implementados en PLDs de bajo y alto nivel de integración a pesar de que las arquitecturas de estos sean muy diferentes, esto es lo que hace que este lenguaje sea una herramienta versátil y poderosa. Para el caso del presente trabajo, en el capítulo 4, se realiza el análisis y la simulación de la solución derivada del diseño del procesador para implementarla en una FPGA con suficientes recursos de hardware, cuyos detalles técnicos se indican en la sección 1.10.

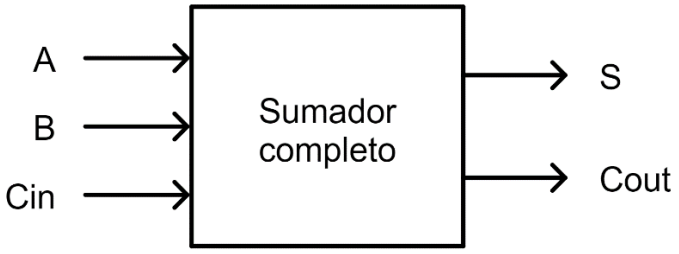
1.9.1. Organización y estructura

El lenguaje VHDL se estructura en diferentes módulos o unidades funcionales, los cuales poseen un conjunto de sentencias que definen y estructuran adecuadamente el comportamiento de un sistema digital. Existen 5 tipos de unidades funcionales: declaración de entidad, o entity, arquitectura o architecture, componente, o component, paquete, o package y bibliotecas o library; las 2 primeras unidades constituyen la columna vertebral de cualquier módulo, mientras que las restantes sirven para generalizar y optimizar la aplicación en futuros desarrollos.

1.9.1.1. Entidad

La entidad es el bloque fundamental de diseño en VHDL, consiste en la descripción de las entradas y salidas del circuito permitiendo representar el circuito a nivel de sistema como se muestra en la figura 53, en donde se muestra la entidad de un circuito sumador completo en donde sólo se indican las entradas, el acarreo de entrada C_{in} y los bits operandos A y B, y las salidas, el resultado S y el acarreo de salida C_{out}.

Figura 53. **Entidad de un sumador completo**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Las entradas y salidas de una entidad se denominan puertos, los cuales deben tener un nombre, un modo y tipo de dato. El nombre permite identificar al puerto fácilmente en otras secciones del código, el modo indica la dirección del flujo de la información y el tipo de dato indica la clase de información que se transmite por el puerto: bit, vector de bits, booleana, entre otros.

Los modos pueden tener los siguientes valores:

- In: se refiere a los puertos de entrada de datos, el flujo de datos es unidireccional.

- Out: se refiere a los puertos de salida de datos, el flujo de datos es unidireccional.
- Inout: permite declarar un puerto de forma bidireccional, es decir como entrada y salida.
- Buffer: permite hacer retroalimentaciones internas y se comporta como un puerto de salida, sin embargo, no se recomienda su uso en diseños nuevos debido a varias restricciones impuestas en el estándar VHDL-97.

Los tipos de datos se asignan considerando las características de un diseño en particular, algunos de los más utilizados son:

- Bit: posee valores de '0' y '1' lógico.
- Boolean: define valores de falso y verdadero en una expresión.
- Bit_vector: representa una cadena de bits.
- Integer: representa algún número entero.
- Std_logic: tipo predefinido en el estándar IEEE 1164. Representa una lógica multivaluada de 9 valores, además del 0 y 1 lógico, posee alta impedancia 'Z', desconocido 'X' o sin inicializar 'U', entre otros.
- Std_logic_vector: representa un vector o cadena de elementos de tipo std_logic.

La forma de declarar una entidad utilizando el ejemplo correspondiente al circuito de la figura 53, es la que se muestra en el segmento de código de la figura 54, los números a un lado de las sentencias sólo sirven para enumerar las líneas de código y no forman parte de este.

Figura 54. **Declaración de una entidad en VHDL**

```
1  -- Entidad de un circuito sumador completo
2  entity sumador_completo is
3     port( A, B, Cin : in STD_LOGIC;
4           S, Cout : out STD_LOGIC
5     );
6  end sumador_completo;
```

Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En la línea 1, se muestran dos guiones '--' para denotar que esa línea es un comentario que será ignorado en la implementación del circuito. La línea 2 inicia la declaración de la entidad con la palabra reservada 'entity' seguido del identificador del circuito (sumador_completo) y la palabra reservada 'is'. Los puertos de entrada se declaran en la línea 3 con el modo 'in' y el tipo std_logic seguido de un ';' para denotar la separación entre diferentes conjuntos de puertos que comparten un tipo en común. En la línea 4 se declaran los puertos de salida utilizando el modo 'out' y el tipo std_logic, como este es el último conjunto de puertos a declarar se cierra la declaración 'port' con los caracteres ')' y ';'. Por último, en la línea 5 se cierra la declaración de la entidad.

Nótese que el lenguaje no es sensible a mayúsculas y minúsculas, por lo que los identificadores 'sumador_completo' y 'SUMADOR_COMPLETO' representan a la misma entidad. De la misma manera, 'STD_LOGIC' y 'std_logic' representan el mismo tipo de dato.

1.9.1.2. Arquitectura

La arquitectura es la estructura que especifica el funcionamiento interno de una entidad previamente declarada y su desarrollo se basa en la especificación funcional del circuito o sistema a implementar.

VHDL cuenta con diferentes enfoques o formatos en los que se puede especificar la arquitectura de una entidad. Es decir, en VHDL es posible describir los circuitos en diferentes niveles de abstracción, desde el uso de ecuaciones booleanas hasta la facilidad de representar las soluciones como sistemas o cajas negras que forman parte de sistemas digitales más complejos. En general, los enfoques de programación utilizados se pueden clasificar como:

- Funcional: expone la forma en que trabaja el sistema, las descripciones únicamente consideran las relaciones existentes entre las entradas y salidas del circuito, sin importar la organización interna de este.
- Flujo de datos: indica la forma en que los datos se transfieren de una señal a otra sin utilizar declaraciones secuenciales como la sentencia ‘if-then-else’.
- Estructural: basa su comportamiento en componentes previamente diseñados, como las compuertas y los sumadores. El usuario puede guardar estructuras y utilizarlas posteriormente.

En la figura 55 se muestra la implementación de un comparador de 2 bits, en la línea 1 se utiliza la librería ieee mientras que en la línea 2 se indica el uso de todos los componentes del paquete ieee.std_logic_1164. La declaración de la entidad se encuentra de la línea 5 hasta la 9 mientras que el algoritmo o la arquitectura abarca desde la línea 11 hasta la 21. En la línea 11 se declara el inicio de la arquitectura con la palabra ‘architecture’ seguida de un identificador

arbitrario, funcional, y la entidad con que se relaciona, comparador. En la línea 12 se usa ‘begin’ para denotar el inicio de la sección en donde se declaran los procesos que rigen el comportamiento del circuito. En la línea 13 se inicializa el proceso con la palabra ‘process’ y la lista de señales a la que este responde.

Figura 55. **Arquitectura de un comparador de 2 bits**

```
1  -- Ejemplo de una descripcion funcional
2  library ieee;
3  use ieee.std_logic_1164.all;
4
5  entity comparador is
6     port( A, B : in STD_LOGIC;
7           C : out STD_LOGIC
8     );
9  end comparador;
10
11 architecture funcional of comparador is:
12 begin
13 comparar process(A, B):
14 begin
15    if a = b then
16       c <= '1';
17    else
18       c <= '0';
19    end if;
20 end process comparar;
21 end funcional;
```

Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En las líneas 15 a la 19 el proceso se describe mediante declaraciones secuenciales ‘if-then-else’ y en las líneas 16 y 18 se muestra el operador de asignación ‘<=’. Una vez terminado el proceso, este se cierra en la línea 20, por último, se cierra la arquitectura en la línea 21.

Debido a que VHDL es un lenguaje bastante robusto, no es posible presentar toda la información correspondiente a su sintaxis y semántica. El presente trabajo asume que el lector posee algo de experiencia en el uso de VHDL, sin embargo, no es un requisito fundamental para comprender el diseño del procesador presentado en el capítulo 3. En el capítulo 4, si el lector desea comprender la implementación del diseño y posteriormente realizar modificaciones, se le recomienda familiarizarse con el lenguaje según sea el caso.

1.9.2. Ventajas y desventajas

En la actualidad, uno de los lenguajes más utilizados es VHDL, capaz de soportar el proceso de diseño de sistemas digitales complejos, con propiedades para reducir el tiempo de diseño y los recursos requeridos. Este lenguaje posee varias ventajas respecto a otros lenguajes de descripción de hardware, algunas de ellas son:

- Notación formal. Permite su uso en cualquier diseño electrónico.
- Disponibilidad pública. VHDL es un estándar no sometido a ninguna patente, por lo que cualquier institución o persona puede utilizarlo libremente y sin restricciones.
- Independencia tecnológica de diseño. VHDL soporta diversas tecnologías de diseño de PLD y ASIC.
- Independencia de la tecnología y proceso de fabricación. Esto significa que es independiente de la tecnología y proceso de fabricación.
- Capacidad descriptiva en varios niveles de abstracción. El proceso de diseño se puede llevar a cabo en distintos niveles de detalles.
- Reutilización de código. Permite utilizar los códigos en varios diseños sin importar que hayan sido diseñados para otras tecnologías.

Sin embargo, VHDL también presenta ciertas desventajas que son importantes a considerar por parte de los diseñadores:

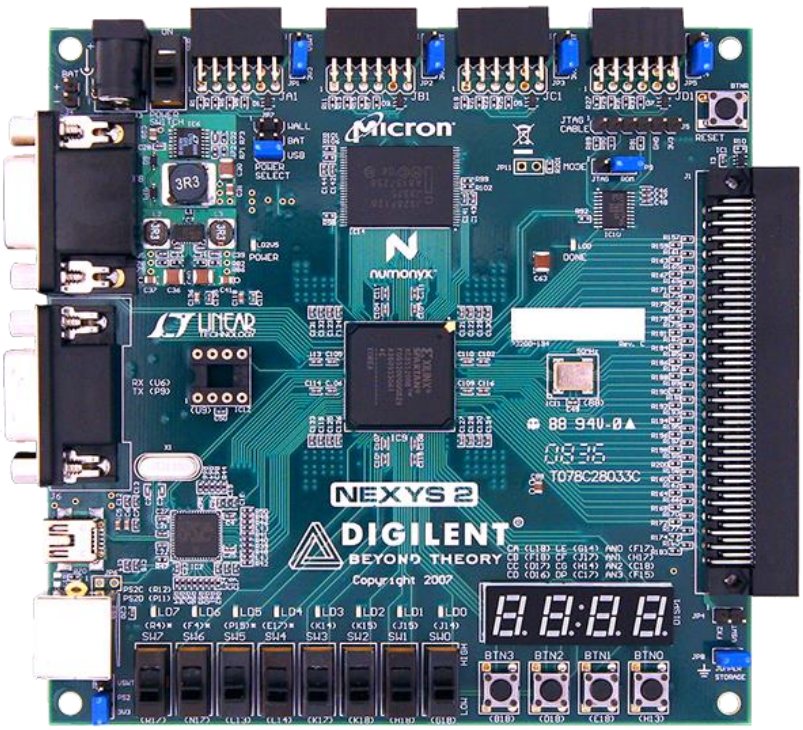
- Hay casos en los que el uso de una herramienta proporcionada por alguna compañía en particular posee características adicionales al lenguaje, con lo que se pierde un poco el carácter de universalidad en el diseño.
- VHDL es un lenguaje diseñado por un comité, por lo que presenta una alta complejidad y se considera poco amigable para personas que no se encuentran familiarizadas con el diseño digital.

1.10. Tarjeta de desarrollo Nexys 2

Como se expuso en la sección 1.9, la solución propuesta para una especificación funcional en particular se puede implementar en una gran cantidad de dispositivos programables ya sea de bajo o alto nivel de integración, para propósitos de este trabajo, la solución del diseño del procesador se implementa en la FPGA contenida en la tarjeta de desarrollo Nexys 2.

La tarjeta Nexys 2 (figura 56) es una plataforma de desarrollo basada en la FPGA Spartan 3E de Xilinx. Posee un puerto USB2 de alta velocidad, 16 MB de RAM y ROM, y varios dispositivos de I/O que permiten desarrollar de manera cómoda una gran variedad de aplicaciones digitales. El puerto USB2 funciona como fuente de alimentación e interfaz de programación, por lo que únicamente se necesita una computadora para sintetizar circuitos en la FPGA.

Figura 56. **Tarjeta de desarrollo Nexys 2**



Fuente: DIGILENT. *Nexys 2 Reference Manual*. www.reference.digilentinc. com/reference/programmable-logic/nexys-2/reference-manual. Consulta: 4 de junio de 2010.

1.10.1. Especificaciones

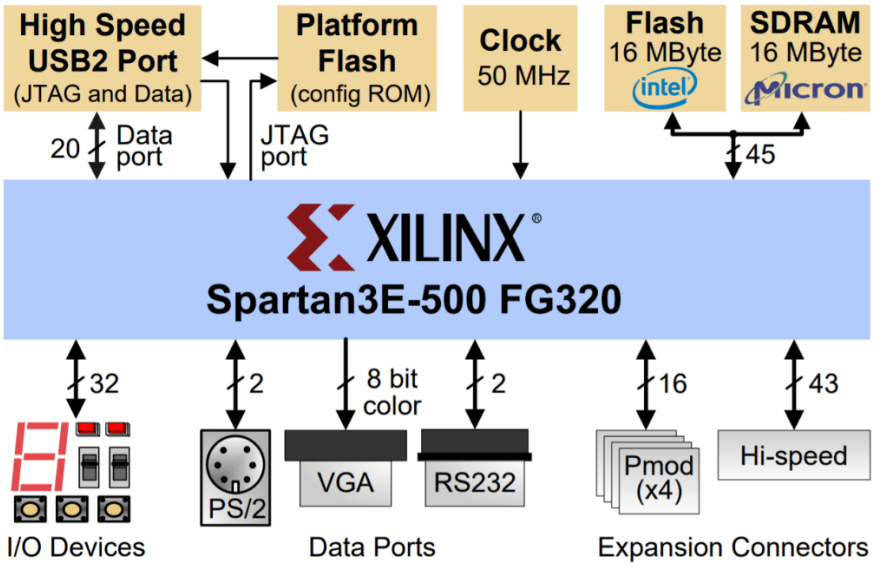
Entre las principales características de hardware que provee la tarjeta se encuentran:

- FPGA Xilinx Spartan 3E-1200 o 3E-500.
- Configuración de la FPGA basada en USB2.
- Alimentación por USB.
- 16MB de SDRAM Micron.
- Plataforma Flash para configuraciones no volátiles de la FPGA.

- Oscilador de 50MHz más un socket para un segundo oscilador.
- 60 dispositivos I/O enrutados con conectores de expansión.
- 8 LED, display de 7 segmentos con 4 dígitos y 8 switches deslizables.

En la figura 57 se muestra un diagrama de bloques de la tarjeta con en donde se muestran varias características de hardware conectadas a la FPGA.

Figura 57. **Diagrama de bloques de la Nexys 2**



Fuente: DIGILENT. *Digilent Nexys2 Board Reference Manual*. p. 1.

1.10.2. FPGA Xilinx Spartan 3E-1200

Esta FPGA pertenece a la familia de dispositivos Spartan-3E, la cual ofrece densidades desde 100 000 a 1,6 millones de compuertas. Algunas de las principales características de esta familia son las siguientes:

- Dispositivos de muy bajo coste y alto rendimiento para aplicaciones de alto volumen orientadas al consumidor.
- Voltajes de señalización de 3,3 V, 2,5 V, 1,8 V, 1,5 V y 1,2 V.
- Velocidad de transmisión de más de 622 Mb/s por cada I/O.
- Recursos lógicos flexibles en abundancia.
    - Densidades de hasta 33 192 celdas lógicas.
    - Multiplexores eficientes.
    - Lógica de acarreo rápido por anticipo.
    - Multiplicadores de 18 x 18 optimizados con pipeline opcional.
- Hasta 8 DCM.
    - Eliminación del clock skew.
    - Síntesis de frecuencias, multiplicación o división.
    - Rango de frecuencias amplio, desde 5 MHz hasta 300 MHz.

En la tabla XIV se muestran los atributos específicos de la FPGA Spartan 3E-1200, en ella se indican la cantidad de CLB, bits de RAM distribuidos, multiplicadores dedicados, DCM y la cantidad máxima de I/O para el usuario. Estas características sólo consisten una pequeña porción de la estructura interna del dispositivo, para mayores detalles se recomienda al lector consultar la hoja de datos del fabricante Xilinx.

Tabla XIV. **Atributos de la FPGA Spartan 3E-1200**

Compuertas	1 200 K
CLB	2 168
Bits de RAM	136 K
Bloques de bits de RAM	504 K
Multiplicadores dedicados	28
DCM	8
Máximo de I/Os de usuario	304

Fuente: XILINX. *Spartan-3E FPGA Family Data Sheet*. p. 2.

2. CONJUNTO DE INSTRUCCIONES DEL PROCESADOR

Una vez presentados los fundamentos teóricos para el análisis y diseño de circuitos digitales relativamente sencillos, es momento de que el lector dirija su atención hacia el uso de estos circuitos como bloques de diseño en la construcción de un sistema digital más complejo: la computadora de propósito general, la cual, permite ejecutar instrucciones para realizar cálculos siguiendo una secuencia o un orden de complejidad considerable.

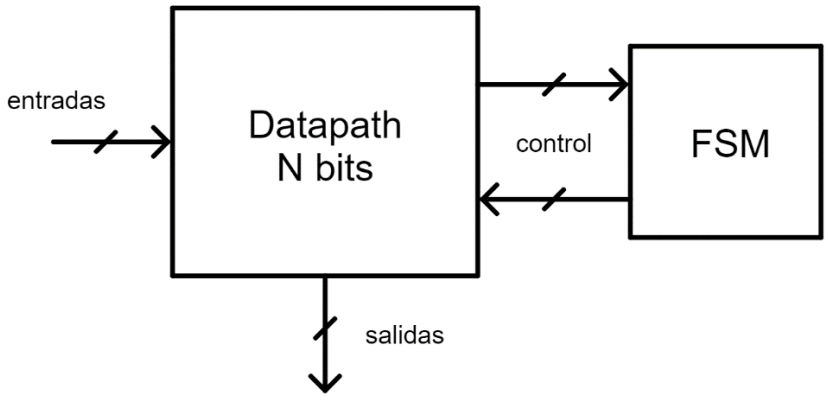
Las instrucciones que permiten realizar las operaciones se deben especificar de manera clara y concisa, es por este motivo que se utiliza la abstracción de la arquitectura del conjunto de instrucciones del procesador o ISA, para especificar el conjunto de funcionalidades que una máquina es capaz de ejecutar. Esta abstracción sirve como una interfaz entre los diseñadores de hardware y software debido a que únicamente indica el conjunto de funcionalidades que provee una arquitectura en particular, ignorando los detalles de cómo se implementan en hardware.

Gracias a esta observación, el alcance del presente capítulo se limita a especificar la estructura básica de una computadora de propósito general utilizando como base, el modelo presentado por John von Neumann durante su trabajo en el proyecto EDVAC en 1945, para posteriormente definir la arquitectura del conjunto de instrucciones del procesador, mientras que, en el capítulo 3, se presenta el diseño de la implementación física del sistema capaz de ejecutar dicho conjunto de instrucciones.

2.1. Rutas de datos o datapaths

Una ruta de datos o datapath es un conjunto de bloques lógicos, líneas de control y líneas de datos cuya interconexión permite realizar operaciones en cadenas binarias de tamaño fijo. Los bloques lógicos se dividen en circuitos combinacionales que llevan a cabo las diferentes operaciones y registros que permiten almacenar los resultados. A las líneas de datos también se les conoce como buses y regularmente son de tamaño fijo mientras que las líneas de control son de tamaño variable, estas últimas son las que permiten llevar a cabo un conjunto de operaciones en algún orden arbitrario.

Figura 58. **Diagrama de un datapath controlado por una FSM**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la figura 58 se muestra el diagrama de bloques general de un datapath de N bits, las entradas y salidas de datos son cadenas binarias de N bits. La secuencia de las operaciones es controlada por un circuito secuencial, en este caso, una máquina de estados la cual, puede depender de algunos indicadores del estado interno del datapath, de aquí que el flujo de datos de control pueda ser o no bidireccional. Es importante resaltar que los datapaths por sí solos no

son capaces de efectuar operación alguna, necesitan de un circuito adicional que controle la secuencia de las operaciones a ejecutar.

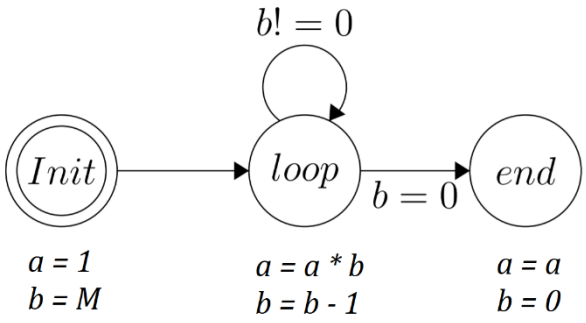
Los datapaths pueden ser ya sea dedicados para una aplicación en particular o circuitos sumamente versátiles capaces de ejecutar un conjunto amplio de funciones, como lo es el caso de los datapaths de propósito general.

2.1.1. De propósito único

Los datapaths de propósito específico poseen hardware dedicado para ejecutar una secuencia de operaciones específica de manera optimizada. Por ejemplo, supóngase el caso en el que se desea construir un datapath capaz de calcular el número factorial de un número entero arbitrario M de 32 bits de manera iterativa, es decir, un circuito que calcule

$$M! = 1 \times 2 \times \ldots (M-1) \times M$$

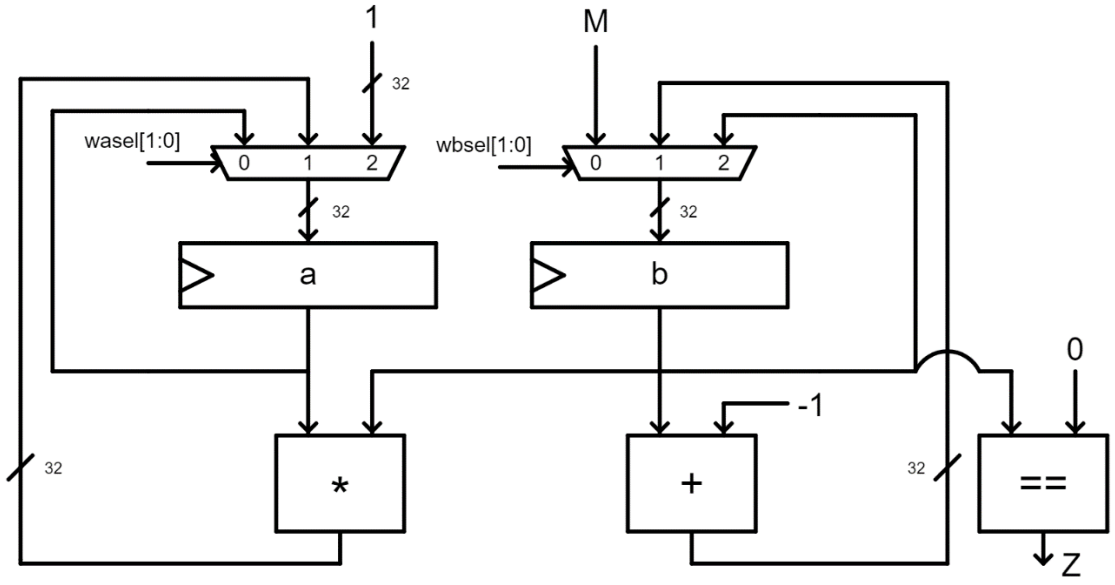
Figura 59. **FSM de alto nivel para el número factorial de M**



Fuente: elaboración propia, empleando LaTeX.

Para comenzar el diseño, es necesario definir una FSM de alto nivel que describa la secuencia de operaciones necesaria tal y como se indica en la figura 59, el estado “Init” inicializa los valores de las variables a y b, en el estado “loop” es donde se llevan a cabo las iteraciones siempre y cuando b sea diferente de cero, en cada iteración, el valor de a se actualiza con el producto $a * b$ y el valor de b disminuye en una unidad, cuando $b = 0$, se realiza la transición hacia el estado “end” que denota el fin del cálculo mostrando el resultado en el registro a.

Figura 60. **Datapath de propósito específico**

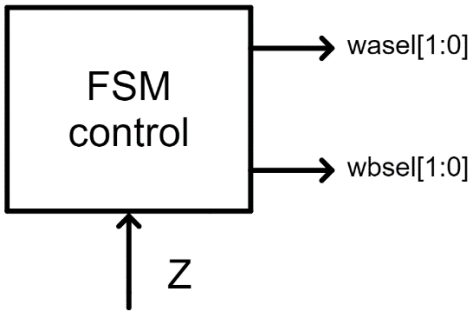


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este circuito no puede operar por sí solo, necesita de una máquina de estados de control cuyas salidas se conecten a las líneas de selección de los multiplexores $wasel[1:0]$ y $wbsel[1:0]$ y así determinar la secuencia de las operaciones, la única entrada de esta máquina proviene del bloque comparador,

que produce una salida $Z = 1$ si $b = 0$. El diagrama de bloques de la máquina de estados se muestra en la figura 61.

Figura 61. **Diagrama de una FSM de control**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la tabla XV se muestra la tabla de estados para la FSM de control de la figura 61, en ella se codifican los estados “init”, “loop” y “end” con los números 00, 01 y 10 respectivamente. Esta funcionalidad se puede implementar utilizando un ROM de 8×4 y un registro de 2 bits.

Tabla XV. **Tabla de estados de la FSM de control**

$\boldsymbol{S[1:0]}$	$\boldsymbol{Z}$	$\boldsymbol{W_{asel}[1:0]}$	$\boldsymbol{W_{bsel}[1:0]}$	$\boldsymbol{S'[1:0]}$
00	0	10	00	01
00	1	10	00	01
01	0	01	01	01
01	1	01	01	10
10	0	00	10	10
10	1	00	10	10

Fuente: elaboración propia.

Nótese que, por motivos prácticos, para implementar la máquina de estados de alto nivel de la figura 59 se descompuso el sistema en un datapath y una máquina de estados de control más sencilla, si se considerara la interconexión del datapath con la máquina de estados de control como una sola FSM se necesitaría considerar todas las posibles combinaciones de los registros de 32 bits y entradas del datapath, lo que supondría que la tarea de realizar la tabla de estados fuera algo impráctico.

Este datapath se encuentra diseñado específicamente para calcular el número factorial de un número entero de 32 bits de manera recursiva, la ventaja principal de un diseño dedicado es que este suele ser más rápido ante otros circuitos de propósito general. Sin embargo, no es un circuito muy versátil, esto quiere decir que el conjunto de operaciones que puede realizar es bastante limitado debido a que el hardware se encuentra optimizado para una aplicación en particular.

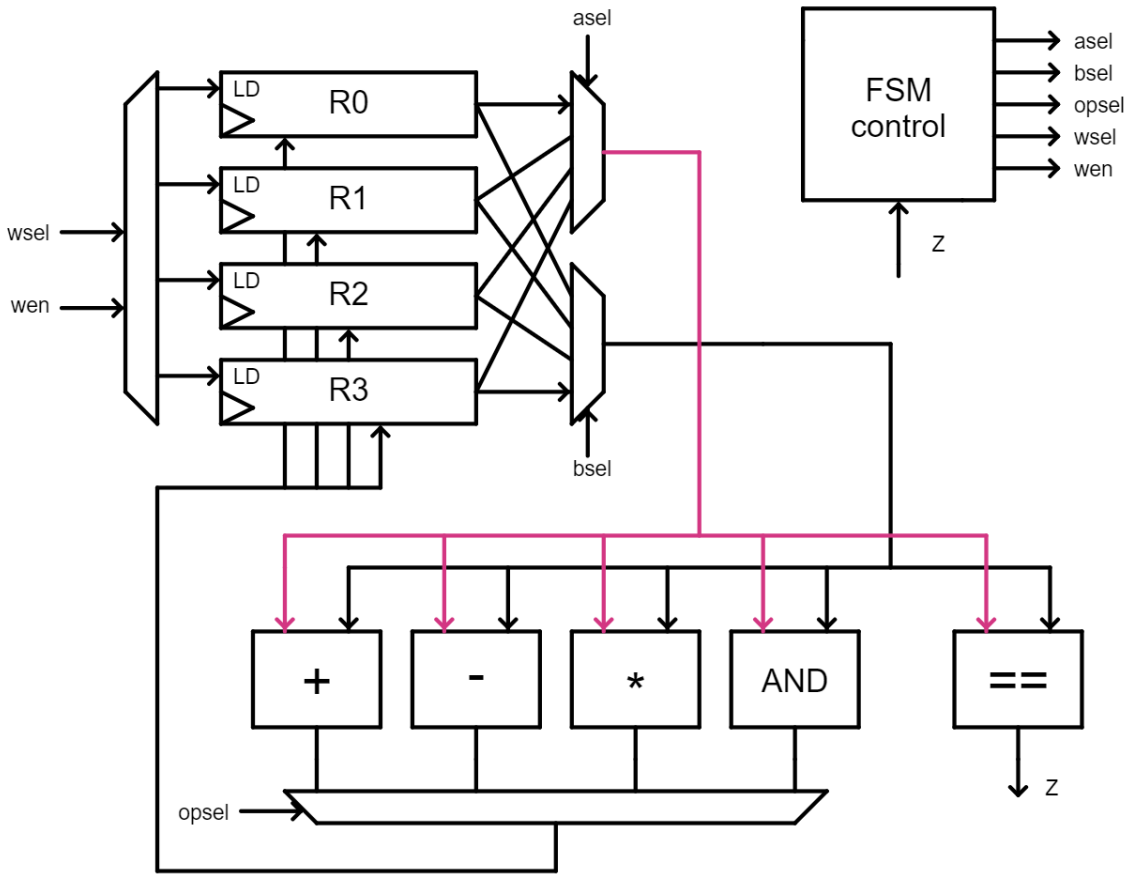
2.1.2. Programables

Los datapaths programables o de propósito general permiten implementar múltiples funcionalidades a través de registros y bloques lógicos adicionales, y, es por este motivo que se consideran circuitos sumamente versátiles capaces de proveer la misma funcionalidad que un conjunto de datapaths de propósito específico.

En la figura 62 se muestra un ejemplo de un datapath programable, contiene 4 registros, un conjunto de 4 bloques de operaciones y un bloque comparador. En cada ciclo de reloj, se seleccionan 2 operandos utilizando las señales de control $asel$ y $bsel$ de los multiplexores conectados a los 4 registros, la señal $opsel$ permite seleccionar el resultado de uno de los 4 bloques lógicos de operaciones,

este resultado se conecta a las entradas de los 4 registros; la señal $wsel$ permite seleccionar qué registro habilitará su entrada de carga LD mientras que la señal wen permite habilitar la salida del decodificador.

Figura 62. **Diagrama de un datapath programable**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Siguiendo con el análisis, también se indica, de forma general, la FSM de control con sus respectivas salidas y entradas. Este datapath permite implementar una gran cantidad de algoritmos que no requieran un almacenamiento temporal de más de 4 registros. Para que el sistema ejecute las

diferentes funcionalidades, es necesario diseñar el hardware de la FSM de control respectivo, al hacer esto, en esencia, se dice que se está programando el datapath. Este principio fue utilizado en las máquinas de propósito múltiple en la década de 1940, un claro ejemplo es la computadora ENIAC, la cual se programaba a través de la modificación de la interconexión de varios interruptores y cables, un proceso bastante complejo que podía tardar varios días.

Este datapath es capaz de realizar una gran cantidad de funciones, incluyendo el número factorial descrito en la subsección 2.1.1, sin embargo, no es capaz de obtener el resultado más rápido que el datapath de propósito específico debido a que sólo puede efectuar una operación en cada ciclo de reloj, mientras que la versión implementada en hardware dedicado puede ejecutar dos operaciones, producto y suma, en un solo ciclo de reloj.

Este enfoque es una de las primeras estrategias que se utilizaron para implementar sistemas digitales complejos de propósito múltiple. A pesar de que la manera de programar los datapaths mediante el diseño del hardware de la FSM de control resulte tedioso y poco práctico, esta idea establece algunas bases para la comprensión de uno de los modelos para una computadora de propósito general más utilizados hoy en día: el modelo Von Neumann.

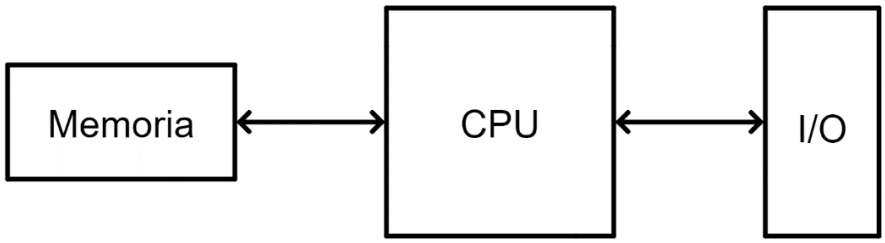
2.2. Modelo Von Neumann

Como se mencionó al final de la subsección 2.1.2, codificar programas o algoritmos a través de la modificación de la circuitería de una FSM de control de un datapath de propósito general es algo sumamente impráctico y complejo que dificulta bastante el uso de la máquina. Es por este motivo, que se necesita un diseño alternativo para la construcción de una computadora de propósito general,

que permita codificar los algoritmos de una manera sistemática y mucho más sencilla.

El modelo o arquitectura Von Neumann que se muestra en la figura 63, es uno de los más utilizados en la actualidad para la construcción de computadoras de propósito general, consta de 3 componentes fundamentales: la unidad central de procesamiento o CPU, encargada de ejecutar una variedad de operaciones; un dispositivo de memoria, que almacena datos e instrucciones para procesarlos; y dispositivos I/O que permitan a la computadora intercambiar información con el exterior.

Figura 63. **Modelo Von Neumann**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Este modelo se diferencia fundamentalmente del datapath programable descrito en la subsección 2.1.2 en la manera en que el algoritmo se codifica en la máquina, en el modelo von Neumann, el algoritmo a ejecutar se encuentra codificado de manera sistemática y no ambigua dentro de un dispositivo dedicado de almacenamiento, una alternativa que, como se verá más adelante, es mucho más práctica y conveniente que codificar el algoritmo dentro de la circuitería de la máquina de estados de control.

2.2.1. Memoria

En este modelo, los datos y las instrucciones de los algoritmos se encuentran codificados como cadenas binarias de W bits en un bloque principal de memoria que posee una cantidad arbitraria finita de celdas, cada una con la capacidad de almacenar W bits. En este bloque de memoria, se utiliza una cadena única de T bits para asignar una dirección a cada celda de memoria, de tal forma que se puede acceder a la información de cualquier celda en cualquier momento y sin ningún tipo de ambigüedad.

Tabla XVI. **Información en una memoria de 32 bits**

Dirección	Contenido	Tipo
0000 0000	00001101 11101001 10000000 10100001	Instrucción
0000 0001	11101001 10000111 01001111 10101011	Instrucción
0000 0010	10000111 01111010 10111111 01111111	Instrucción
0000 0011	10000111 01111111 01101101 10101010	Instrucción
. . .	. . .	. . .
1111 1100	01001011 10101011 10101011 11111111	Dato
1111 1101	10101011 11111001 10000101 10101110	Dato
1111 1110	00000111 00010111 11110011 10111011	Dato
1111 1111	10111111 10110111 10011011 11011101	Dato

Fuente: elaboración propia.

Regularmente los parámetros W y T son equivalentes a potencias de 2, además, en muchos sistemas computacionales modernos, aunque no necesariamente, representan la misma cantidad, esto quiere decir que una dirección de memoria puede ocupar todo el espacio de una celda de memoria. El parámetro W es una característica de la arquitectura de la computadora mientras

que T es un parámetro que puede variar según los requerimientos de almacenamiento.

En la tabla XVI se muestra el caso particular de información almacenada en una memoria de 32 bits, $W = 32$, en la que se utilizan únicamente $T = 8$ bits para direccionar las celdas de memoria, esto significa que existen $2^8 = 256$ celdas de 32 bits, es decir, $256 \times 32 = 8192$ bits, por lo que la memoria tiene una capacidad de almacenamiento equivalente a 1 024 bytes.

En general, una memoria puede verse como una tabla de valores binarios en los que se desconoce a primera vista el tipo de información que reside en cada celda en particular, es por este motivo que regularmente la memoria se divide en dos bloques: información y datos, de esta manera es posible identificar el tipo de información a partir de la dirección de memoria en la que se encuentra almacenada.

Tabla XVII. **Información de una memoria en hexadecimal**

Dirección	Contenido
$0x00$	$0x0DE980A1$
$0x01$	$0xE9874FAB$
$0x02$	$0x877ABF7F$
$0x03$	$0x877F6DAA$
. . .	. . .
$0xFC$	$0x4BABABFF$
$0xFD$	$0xABF985AE$
$0xFE$	$0x0717F3BB$
$0xFF$	$0xBFB79BDD$

Fuente: elaboración propia.

Para facilitar la lectura de los valores por parte de los programadores o diseñadores, las direcciones y valores de las celdas se suelen representar utilizando el sistema hexadecimal, en la tabla XVII se muestra una representación más compacta de los contenidos de la memoria de la tabla XVI.

Por último, es importante señalar que la información almacenada en la memoria no se considera una parte tangible o física del sistema, únicamente consiste en procedimientos o reglas que permiten procesar determinado grupo de datos mientras que, los circuitos lógicos digitales, son los que se encargan de llevar a cabo este conjunto de instrucciones; a partir de esta sencilla, pero importante observación es posible establecer una clara distinción entre software y hardware.

2.2.1.1. Direccionamiento por bytes

Al parámetro W se le conoce como palabra o ancho de palabra de la arquitectura y, establece la cantidad exacta de bits con la que se debe codificar la información. Existen arquitecturas de hardware que segmentan los datos en cadenas de bytes y las almacenan en direcciones de memoria consecutivas, a estas se les conoce como arquitecturas direccionables por bytes, ya que permiten acceder individualmente a un byte específico de una palabra en particular gracias a que cada uno de estos posee una dirección única de memoria.

En la tabla XVIII se muestra un ejemplo de cómo la información se distribuye en una arquitectura direccionable por bytes con un ancho de palabra de 32 bits o 4 bytes, en este caso, se observa que se necesitan 4 localidades de 8 bits para almacenar cada palabra, por ejemplo, la palabra de 32 bits almacenada en la localidad $0x00$ abarca las direcciones $0x00$, $0x01$, $0x02$ y $0x03$.

Ahora bien, existen dos convenciones para almacenar los bytes en este tipo de arquitecturas dependiendo de la dirección del byte menos significativo, estas son:

- Big endian, el byte menos significativo ocupa la localidad de memoria más alta.
- Little endian, el byte menos significativo ocupa la localidad de memoria más baja.

Tabla XVIII. **Memoria direccionable por bytes**

Dirección	**Contenido**			
$0x00$	$0xA2$	$0x35$	$0xFD$	$0x4C$
$0x04$	$0x34$	$0x23$	$0xF4$	$0x3D$
$0x08$	$0xA6$	$0xAD$	$0xAF$	$0xFD$
. . .	. . .			
$0xF4$	$0x46$	$0x67$	$0xFD$	$0xFA$
$0xF8$	$0x56$	$0x32$	$0x44$	$0x4F$
$0xFC$	$0x12$	$0x1D$	$0xD2$	$0xBD$

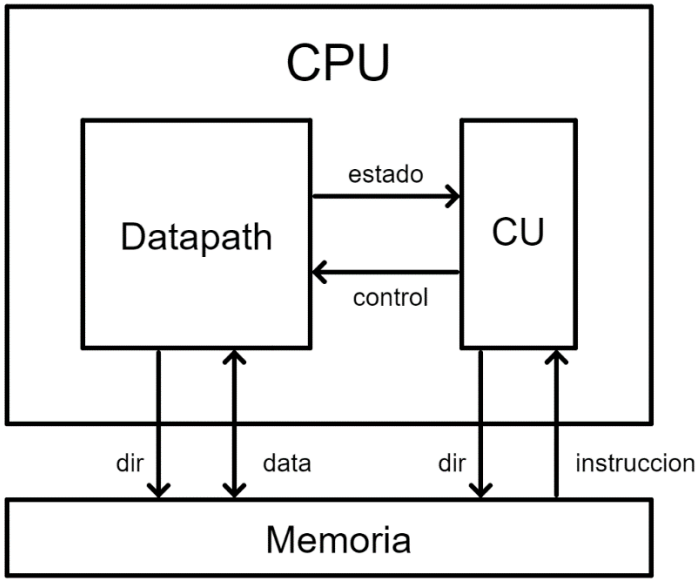
Fuente: elaboración propia.

2.2.2. Unidad central de procesamiento, CPU

La unidad central de procesamiento, CPU o simplemente, el procesador, es el componente encargado de decodificar y ejecutar las instrucciones almacenadas en memoria. El diseño de un procesador se divide regularmente en dos subsistemas: el datapath, encargado de ejecutar las operaciones; y la unidad de control, o CU, que se encarga de decodificar las instrucciones y dirigir la secuencia de las operaciones del datapath. En esta subsección se discute

únicamente la funcionalidad general de estos bloques, los detalles del diseño del hardware se encuentran en el capítulo 3.

Figura 64. **Diagrama de CPU y su conexión con una memoria**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

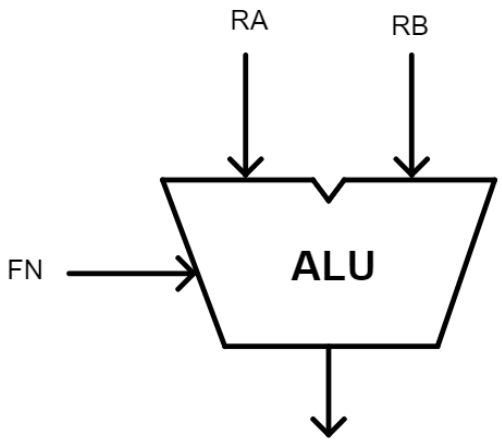
En la figura 64 se muestra el diagrama interno de un CPU y su conexión con el dispositivo de memoria, la unidad de control envía las señales de control necesarias para que el datapath lleve a cabo determinado conjunto de operaciones, del otro lado, el datapath envía los valores de sus indicadores de la última operación que realizó, si el resultado fue cero, número negativo o si hubo sobreflujo, a este grupo de señales se les conoce como el estado actual. El datapath puede intercambiar información en ambos sentidos con la memoria mientras que la CU únicamente puede leer las instrucciones almacenadas que deberá decodificar y ejecutar; para leer o escribir en la memoria, ambos sistemas

deben especificar a través de la señal dir la dirección de la celda a la que desean acceder.

2.2.2.1. Datapath

El datapath de un procesador posee una estructura que se asemeja a la del datapath programable de la figura 62, aunque regularmente, cuenta con una mayor cantidad de registros, para el caso del presente trabajo, se trabajará con 32 registros de propósito general. El repertorio de operaciones lógicas y aritméticas se encuentran dentro de un bloque funcional llamado unidad aritmética lógica o ALU cuyo diagrama se muestra en la figura 65, en él se indican los dos operandos RA y RB, y la señal FN encargada de seleccionar la operación que se ejecutará.

Figura 65. **Unidad aritmética lógica o ALU**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

2.2.2.2. Unidad de control

La unidad de control consta de dos componentes principales: contador de programa o PC y lógica de control. El primero permite mantener el estado de la unidad a través de un registro de almacenamiento interno, encargado de almacenar una dirección de memoria arbitraria. El contenido de este registro es la dirección de memoria donde se encuentra ubicada la siguiente instrucción a ejecutar, regularmente, este registro aumenta automáticamente para ejecutar instrucciones que se encuentran en localidades de memoria consecutivas.

El segundo componente de la unidad de control es la lógica de control, la cual se encarga de generar las señales de control necesarias para que el datapath ejecute las operaciones codificadas dentro de las instrucciones. Regularmente, este componente consiste en un circuito combinacional que puede ser diseñado con los métodos de la sección 1.4.

2.3. Instrucciones

Las instrucciones constituyen el lenguaje nativo de la máquina, estas se codifican de manera sistemática y sin ambigüedades como cadenas binarias de W bits, las cuales, indican el tipo de operación que se debe ejecutar: aritmética, lógica, acceso a memoria, entre otros, los operandos involucrados y el registro en el cual se almacenará el resultado según sea el caso.

Los detalles del formato de codificación concreto de las instrucciones, el ancho de palabra W y los tipos de instrucciones varían según la arquitectura, es por este motivo, que sólo se expondrá generalmente, la manera en la que se codifican y ejecutan las instrucciones en una máquina con arquitectura Von Neumann.

2.3.1. Estructura

En una instrucción de W bits, se reserva una cantidad de q bits para codificar el tipo de operación, a esta cadena de bits, se le conoce como el opcode o código de operación; teóricamente, una arquitectura es capaz de reconocer hasta 2^q opcodes diferentes, aunque en muchos casos, esta cantidad sólo representa un límite superior.

Además del opcode, en una instrucción se deben especificar los registros involucrados en la operación. La cantidad j de bits reservados para codificar cada uno de los N registros de un datapath debe ser el número entero mínimo que cumpla con $2^j \geq N$, un esquema de codificación eficiente es aquel en el que N es equivalente a una potencia de 2, así, a cada una de las combinaciones de j bits le corresponde un registro único. Entonces, según sea el caso, en la instrucción se reserva una cantidad de j a $3 \times j$ bits para especificar los registros operandos y el registro de destino según sea el caso.

Tabla XIX. **Ejemplo de una instrucción de 32 bits**

31 26	25 21	20 16	15 11	10 0
Opcode	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

En la tabla XIX se muestra un ejemplo de un formato de codificación de una instrucción de 32 bits del procesador a implementar, en este caso, se reservan 6 bits para el opcode y 5 bits para cada registro, los operandos son Ra y Rb, y el registro de destino es Rc. Los últimos 11 bits no se utilizan, sino que forman parte de un campo reservado para futuras actualizaciones del formato. Cabe resaltar que, al tratarse de un ejemplo ilustrativo, los detalles completos de los formatos

de codificación de las instrucciones del procesador se reservan para la sección 2.6.

Debido a lo tedioso que resulta para el programador tratar de distinguir entre las diferentes combinaciones de cadenas binarias, se utilizan mnemónicos para facilitar esta tarea; un ejemplo de cómo se pueden asignar estos mnemónicos se ilustran en la tabla XX para el caso de un grupo de opcodes de 6 bits.

Tabla XX. **Mnemónicos de algunos opcodes**

Mnemónico	Opcode
ADD	$0x20$
AND	$0x28$
OR	$0x29$
SUB	$0x21$

Fuente: elaboración propia.

La principal ventaja de un esquema de codificación sencillo, pero conciso, es que permite identificar fácilmente cada uno de los elementos de una instrucción, lo que permite ejecutar una gran cantidad de algoritmos con un único hardware de control cuya estructura es relativamente sencilla.

2.3.2. Ejecución en una máquina Von Neumann

Una vez explicada la estructura básica de algunas instrucciones, es importante identificar los pasos involucrados en la ejecución de estas. En una máquina von Neumann, para ejecutar una instrucción, el procesador realiza el conjunto de operaciones lógicas necesarias de manera secuencial. En términos

generales, las operaciones lógicas necesarias para ejecutar una instrucción y el orden en el que se llevan a cabo se detallan a continuación:

- Búsqueda de la instrucción, la instrucción se carga a la lógica de control desde la localidad de memoria especificada en el contador de programa.
- Decodificación de la instrucción, la lógica de control genera las señales de control necesarias a partir de la instrucción cargada previamente.
- Lectura de los operandos, se cargan los valores de los registros especificados por la lógica de control hacia las entradas de la ALU.
- Ejecución, la ALU realiza la operación especificada por la lógica de control.
- Escritura en el destino, el resultado de la ALU se almacena en el registro de destino especificado en la instrucción.
- Cálculo del siguiente PC, se calcula el valor de la dirección donde se encuentra la siguiente instrucción a ejecutar y se almacena en el contador de programa.

Por último, es necesario hacer ver al lector que, esta secuencia de operaciones se ejecuta cíclicamente, por lo que la secuencia se repite cada vez que se necesita ejecutar una nueva instrucción.

2.4. Arquitectura del conjunto de instrucciones, ISA

Antes de definir detalladamente los formatos de las instrucciones bajo los cuales los programadores deberán codificar sus algoritmos para hacer uso de una computadora de propósito general, es necesario indicar qué información provee el conjunto de instrucciones de una máquina e identificar al mismo como una abstracción o interfaz que permite el desacoplo entre el software y el hardware.

La arquitectura del conjunto de instrucciones o ISA es un modelo abstracto de una máquina que detalla el conjunto de instrucciones que el procesador es capaz de ejecutar, los tipos de datos que soporta, la cantidad de registros disponibles, la arquitectura de la memoria y cómo acceder a ella. Un ISA contiene únicamente la información necesaria para los programadores, pues, especifica cómo se deben codificar los programas para las computadoras que compartan determinada arquitectura.

Es importante hacer ver que el ISA se entiende como una abstracción que permite a los programadores utilizar la funcionalidad de una arquitectura sin la necesidad de conocer su implementación física. Gracias a esta abstracción, tanto el hardware como el software pueden evolucionar de manera independiente; por ejemplo, a medida que la tecnología de hardware mejora con el paso del tiempo, es posible construir implementaciones mucho más rápidas y eficientes de un mismo conjunto de instrucciones sin la necesidad de tener que cambiar el software desarrollado por los programadores.

En pocas palabras, la arquitectura del conjunto de instrucciones es una abstracción que permite a los desarrolladores de hardware y software trabajar de manera independiente y garantiza que los diseños de ambas partes sean compatibles entre sí.

Es común clasificar los diferentes ISAs en función de la complejidad de su arquitectura, con base en este criterio, existen dos clasificaciones: computadora con conjunto de instrucciones reducido (RISC) y complejo (CISC).

2.4.1. Arquitectura RISC

Una computadora con conjunto de instrucciones reducido o RISC es aquella que posee un número reducido de instrucciones y formatos en comparación con otras arquitecturas. Los conjuntos de instrucciones simples reducen los modos en los que se puede acceder a los datos de la memoria al limitar las operaciones de memoria únicamente a dos instrucciones: carga, o load, y almacenamiento, o store.

En esta arquitectura, las instrucciones poseen una longitud constante y abarcan una palabra entera del procesador. Los formatos de las mismas se restringen a unos pocos, siempre y cuando sean del ancho de una palabra. Esta característica permite que la implementación física del procesador sea relativamente más sencilla, por este motivo, el diseño del procesador del presente trabajo posee una arquitectura RISC, con el objetivo de simplificar el diseño del hardware. Las instrucciones únicamente utilizan como operandos a los registros del procesador y, como se dijo anteriormente, load y store son las únicas instrucciones capaces de acceder a memoria.

Un problema que aparece regularmente es que, en implementaciones sencillas de esta arquitectura, los accesos a memoria suelen ser muy lentos; una de las formas en las que se mitiga esta situación es a través del diseño y la implementación de un sistema jerárquico de memoria para alcanzar un tiempo de respuesta promedio menor.

Además, debido a que las instrucciones son de tamaño fijo y abarcan una palabra del procesador, el tamaño del código es relativamente grande, esto supone una dificultad, por ejemplo, en el diseño de aplicaciones para sistemas embebidos, en donde los recursos de memoria son bastante limitados.

En resumen, gracias a la poca flexibilidad en el formato de las instrucciones, un procesador con arquitectura RISC permite que se realicen implementaciones en hardware de alto rendimiento y, además, permite que los compiladores sean más sencillos en el caso del software. Algunos ejemplos de conjuntos de instrucciones que comparten este tipo de arquitectura son ARM y MIPS.

2.4.2. Arquitectura CISC

Una computadora con conjunto de instrucciones complejo o CISC ofrece una mayor cantidad de instrucciones de tamaño variable y formatos que permiten agrupar, en una sola instrucción, diferentes operaciones que requerirían el uso de varias instrucciones de una arquitectura RISC; esto permite que el tamaño del código para un algoritmo en particular disminuya considerablemente.

Una característica importante de las arquitecturas CISC es la microprogramación, esto significa que las instrucciones son decodificadas internamente y ejecutadas con una serie de microinstrucciones o, microprograma, almacenado en una ROM de acceso rápido localizada dentro del procesador. Para llevar a cabo esta función, es necesario diseñar una unidad de control con hardware capaz de decodificar todos los formatos y variantes de las instrucciones hacia su secuencia equivalente de microinstrucciones, una tarea que, resulta más complicada si se compara con la simplicidad del diseño del hardware de una arquitectura RISC.

Con tecnologías de semiconductores comparables e igual frecuencia de reloj, un procesador CISC posee un rendimiento o capacidad de procesamiento de 2 a 4 veces menor que la de un RISC. Esto sucede debido a que las instrucciones se deben decodificar internamente y ejecutarse con una serie de instrucciones adicionales almacenadas en una ROM interna.

En la actualidad, una de las arquitecturas más representativas de CISC es la x86, prácticamente cualquier computadora personal ha utilizado esta arquitectura desde los años 80.

2.5. Lenguaje de transferencia de registros, RTL

Antes de presentar la arquitectura del conjunto de instrucciones del procesador del presente trabajo, es importante que el lector se familiarice con la notación del lenguaje de transferencia de registros o RTL, con la cual, se describen las secuencias de operaciones necesarias para ejecutar una instrucción particular, de una manera independiente al ISA; esto permite describir la semántica de cada una de las instrucciones y facilita el entendimiento de la funcionalidad que provee el hardware para el desarrollador de software.

El lenguaje de transferencia de registros es un tipo de representación que permite especificar una serie de operaciones computacionales asociadas con la ejecución de una instrucción. Considérese el caso de la secuencia de operaciones siguiente:

$$Reg[R0] \leftarrow Mem[0x100]$$
$$tmp \leftarrow Reg[R0] + 3$$
$$Mem[0x104] \leftarrow tmp \ \& \ 0x7$$

En esta secuencia, el contenido de la localidad de memoria con la dirección $0x104$ se transfiere al registro $R0$, se suma el número decimal 3 al contenido de $R0$ y se transfiere a la variable temporal $temp$, por último, el resultado de la operación AND entre tmp y la constante $0x7$ se transfiere a la localidad de memoria con dirección $0x104$.

Tabla XXI. **Algunos operadores utilizados en expresiones RTL**

Operador	Descripción
$a \pm b$	Suma o resta de los operandos a y b
$a \mid b$	OR entre los operandos a y b
$a \,\&\, b$	AND entre los operandos a y b
$\sim a$	Complemento a uno del operando a
$Mem[addr]$	Contenido de localidad con la dirección $addr$
$Reg[Rx]$	Contenido del registro Rx

Fuente: elaboración propia.

En este lenguaje, cada operación involucra la transferencia de un valor especificado por la expresión a la derecha del operador "←" hacia el destino especificado por la expresión que se encuentra a la izquierda de éste. Las expresiones a la derecha del operador "←" suelen involucrar constantes, localidades de memoria, o registros. Las expresiones que especifican el destino de la operación generalmente contienen nombres de variables, localidades de memoria, o registros. En la tabla XXI se muestra la descripción de algunos operadores que se utilizan con regularidad para la descripción de instrucciones en un ISA, obsérvese que las operaciones lógicas, como AND, OR y NOT, hacen referencia a una operación lógica bit por bit.

2.6. Especificación del ISA del procesador

La tarea de diseñar un ISA no es fácil, ésta resulta bastante compleja al tener que considerar múltiples factores: la cantidad de instrucciones que debe contener, el ancho de palabra, la manera en que se almacenan datos en memoria, el esquema de codificación de las instrucciones, entre otros. Además,

el diseño debe satisfacer varios requerimientos de costos, rendimiento y, compatibilidad con ISA pasados y futuras tendencias.

Por motivos prácticos, diseñar un conjunto de instrucciones, considerando los factores mencionados anteriormente, queda completamente fuera del alcance de este trabajo; como alternativa y, sin perder el propósito de la presente investigación, se hace uso de un conjunto de instrucciones ya existente, el cual se describe con detenimiento en la presente sección.

El ISA a utilizar fue introducido por el Instituto de Tecnología de Massachussets, o MIT, y recibe el nombre de Beta o β ISA, el cual es un conjunto de instrucciones simple y representativo de una arquitectura RISC que se utiliza para introducir algunos de los conceptos básicos de arquitectura de computadoras.

2.6.1. Modelo de la máquina

El conjunto de instrucciones β es una arquitectura de 32 bits de propósito general, esto significa que tanto los registros como las direcciones de memoria tienen un ancho de 32 bits. Una vez almacenados valores de direcciones en los registros, estos pueden apuntar hacia cualquier localidad dentro de una memoria direccionable por bytes.

Para el caso de los datos, el conjunto de instrucciones sólo permite la manipulación de números enteros utilizando la codificación de complemento a 2 presentada en la subsección 1.2.2.

2.6.1.1. Estado del procesador

Los programadores pueden observar el estado del procesador a través de los contenidos del contador de programa y un conjunto de 32 registros de propósito general, cada uno almacenando valores de 32 bits. En la tabla XXII se muestra una representación gráfica del estado del procesador, los registros de propósito general se denotan utilizando las variables $R0$, $R1$, …, $R31$, obsérvese que el registro $R31$ es especial, siempre posee un valor de cero cuando se lee y cualquier acción de escritura sobre el registro queda sin efecto. Nótese que, al ser 32 registros de propósito general se requieren 5 bits para codificar cada uno de éstos.

Tabla XXII. **Estado del procesador**

$\boldsymbol{PC}$	Siempre es múltiplo de 4
$\boldsymbol{R0}$	
$\boldsymbol{R1}$	
. . .	. . .
$\boldsymbol{R30}$	
$\boldsymbol{R31}$	Siempre es cero
	◄—— Ancho de 32 bits ——►

Fuente: TERMAN, Chris. *Instruction Set Architectures*. www.computationstructures.org/lectures/isa/isa.html. Consulta: 17 de abril de 2020.

El registro PC tiene una característica particular, los valores de las direcciones de memoria que se almacenan en él son siempre múltiplos de 4. Este conjunto de instrucciones sólo permite accesos de palabra, es decir que, a pesar de que la memoria sea direccionable por bytes, el procesador únicamente es capaz de leer o escribir datos de 32 bits.

2.6.1.2. Memoria principal

La memoria principal consiste en un arreglo de localidades o palabras de 32 bits, las cuales se dividen en 4 bytes, a cada uno de éstos se le asigna una dirección única, por lo que la memoria de la máquina es direccionable por bytes, esto implica que palabras consecutivas dentro de la memoria poseen direcciones de memoria cuya diferencia es 4, además de que todas las direcciones de palabras almacenadas son múltiplos de 4. La memoria principal de la máquina posee la misma estructura de la memoria direccionable por bytes de la tabla XVIII.

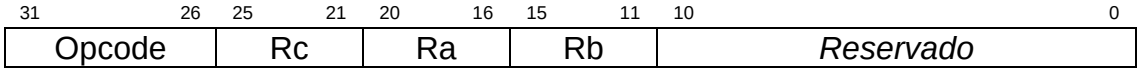
Este conjunto de instrucciones permite utilizar hasta 32 bits para codificar las direcciones de memoria para que sólo sea necesario un registro para almacenar una dirección específica. A partir de esta condición se observa que el máximo tamaño de memoria que se puede utilizar es de 2^{32} bytes, es decir, 4 gigabytes (4 GB). Sin embargo, este valor sólo indica el límite del tamaño de memoria que se puede utilizar con esta arquitectura, algunas implementaciones físicas pueden tener una memoria de menor capacidad.

2.6.2. Codificación de las instrucciones

El tamaño de todas las instrucciones es de 32 bits. Todas las instrucciones especifican una operación que requiere a lo sumo 2 operandos, y un registro de destino para almacenar el resultado. Uno de los dos operandos puede ser la extensión con signo de una constante de 16 bits representada en complemento a 2, esto quiere decir que, el valor de los 16 bits más significativos del segundo operando es equivalente al valor del bit más significativo de la constante de 16 bits. Por ejemplo, el lector puede comprobar fácilmente que los números $0xE4A2$ y $0xFFFFE4A2$, una constante de 16 bits y su extensión con signo de 32 bits respectivamente, son equivalentes al mismo número entero.

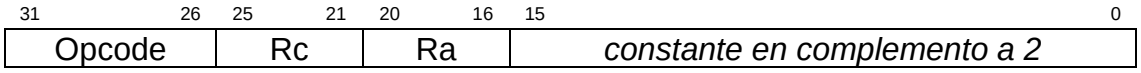
Sólo existen dos tipos de formatos de instrucción: con y sin constante. Las instrucciones sin constante incluyen todas las operaciones lógicas y aritméticas entre dos registros cuyo resultado se almacena en un tercer registro. Las instrucciones con constante incluyen otras instrucciones, como los accesos a memoria. En las tablas XXIII y XXIV se muestra explícitamente ambos formatos de codificación, nótese que se utilizan 6 bits para codificar los opcodes y 5 bits para codificar el número de registro.

Tabla XXIII. **Formato de una instrucción sin constante**



Fuente: TERMAN, Chris. *Instruction Set Architectures*.
www.computationstructures.org/lectures/isa/isa.html. Consulta: 17 de abril de 2020.

Tabla XXIV. **Formato de una instrucción con constante**



Fuente: TERMAN, Chris. *Instruction Set Architectures*.
www.computationstructures.org/lectures/isa/isa.html. Consulta: 17 de abril de 2020.

2.6.3. Resumen de instrucciones

En la tabla XXV se muestra el resumen de las instrucciones que provee la arquitectura Beta, en ella se indican los mnemónicos junto a sus respectivos opcodes de 6 bits representados en hexadecimal. Los detalles de cada una de las instrucciones se presentan en la subsección 2.6.4.

Tabla XXV. **Lista de opcodes y mnemónicos**

Mnemónico	Opcode	Mnemónico	Opcode	Mnemónico	Opcode
ADD	$0x20$	CMPLT	$0x25$	SHRC	$0x3D$
ADDC	$0x30$	CMPLTC	$0x35$	SRA	$0x2E$
AND	$0x28$	JMP	$0x1B$	SRAC	$0x3E$
ANDC	$0x38$	LD	$0x18$	SUB	$0x21$
BEQ	$0x1C$	LDR	$0x1F$	SUBC	$0x31$
BNE	$0x1D$	OR	$0x29$	ST	$0x19$
CMPEQ	$0x24$	ORC	$0x39$	XOR	$0x2A$
CMPEQC	$0x34$	SHL	$0x2C$	XORC	$0x3A$
CMPLE	$0x26$	SHLC	$0x3C$	XNOR	$0x2B$
CMPLEC	$0x36$	SHR	$0x2D$	XNORC	$0x3B$

Fuente: TERMAN, Chris. *Instruction Set Architectures*. www.computationstructures.org/lectures/isa/isa.html. Consulta: 18 de abril de 2020.

2.6.4. Especificación de instrucciones

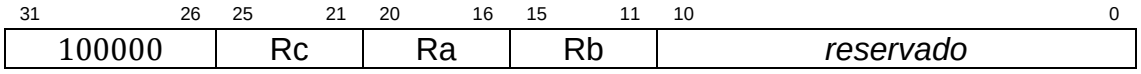
En esta sección se encuentran los detalles de las instrucciones, ordenadas alfabéticamente en función de su mnemónico. En la descripción de cada instrucción, se asume que no existen retardos de propagación, por lo que esta descripción es independiente de la implementación física del ISA. Todas las instrucciones se ejecutan completamente antes de comenzar la ejecución de una nueva instrucción. Por último, debido a la gran cantidad de tablas de aspecto similar, cuando se describe la funcionalidad de las instrucciones, se hace referencia, de manera implícita, a la tabla con el mnemónico de la instrucción.

- ADD:

El contenido del registro Ra se suma con el contenido del registro Rb y el resultado de 32 bits se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] + Reg[Rb]$$

Tabla XXVI. **Formato de instrucción ADD**



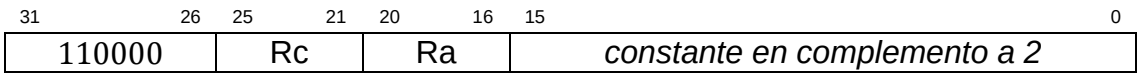
Fuente: elaboración propia.

- ADDC:

El contenido del registro Ra se suma con la extensión con signo de la $constante$, $SEXT(constante)$, y el resultado de 32 bits se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] + SEXT(constante)$$

Tabla XXVII. **Formato de instrucción ADDC**



Fuente: elaboración propia.

- AND:

Realiza la función booleana AND bit a bit entre los contenidos del registro Ra y el registro Rb. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \,\&\, Reg[Rb]$$

Tabla XXVIII. **Formato de instrucción AND**

31 — 26	25 — 21	20 — 16	15 — 11	10 — 0
101000	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- ANDC:

Realiza la función booleana AND bit a bit entre el contenido del registro Ra y la extensión con signo de la $constante$. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \,\&\, SEXT(constante)$$

Tabla XXIX. **Formato de instrucción ANDC**

31 — 26	25 — 21	20 — 16	15 — 0
110000	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- BEQ:

El valor del registro PC actualizado, $PC + 4$, correspondiente a la instrucción que sigue después de la instrucción BEQ, se almacena en el registro Rc. Si el contenido del registro Ra es cero, el registro PC se carga con la dirección de destino EA; de lo contrario, la ejecución continúa de manera secuencial.

La $constante$ de la instrucción, es un parámetro que indica el desplazamiento con signo de palabras a partir del valor del registro PC actualizado, este se debe multiplicar por 4 para convertirlo a un desplazamiento en términos de bytes, tomar su extensión con signo de 32 bits y sumarla con el valor del registro PC actualizado para calcular la dirección de destino EA. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$EA \leftarrow PC + 4 * SEXT(constante)$$
$$temp \leftarrow Reg[Ra]$$
$$Reg[Rc] \leftarrow PC$$
$$if \quad temp = 0, \qquad then \quad PC \leftarrow EA$$

Tabla XXX. **Formato de instrucción BEQ**

31 26	25 21	20 16	15 0
011100	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- BNE:

El valor del registro PC actualizado, $PC + 4$, correspondiente a la instrucción que sigue después de la instrucción BNE, se almacena en el registro Rc. Si el contenido del registro Ra es diferente de cero, el registro PC se carga con la dirección de memoria EA; de lo contrario, la ejecución continúa de manera secuencial.

La $constante$ de la instrucción, es un parámetro que indica el desplazamiento con signo de palabras a partir del valor del registro PC actualizado, este se debe multiplicar por 4 para convertirlo a un desplazamiento en términos de bytes, tomar su extensión con signo de 32 bits y sumarla con el valor del registro PC actualizado para calcular la dirección de destino EA. Su secuencia de operaciones es la siguiente:

Tabla XXXI. **Formato de instrucción BNE**

31 26	25 21	20 16	15 0
011100	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

$$PC \leftarrow PC + 4$$
$$EA \leftarrow PC + 4 * SEXT(constante)$$
$$temp \leftarrow Reg[Ra]$$
$$Reg[Rc] \leftarrow PC$$
$$if \quad temp \neq 0, \qquad then \quad PC \leftarrow EA$$

- CMPEQ:

Si el contenido del registro Ra es igual al contenido del registro Rb, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$if \quad Reg[Ra] = Reg[Rb], \qquad then \ Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXII. **Formato de instrucción CMPEQ**

31 26	25 21	20 16	15 11	10 0
100100	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- CMPEQC:

Si el contenido del registro Ra es igual a la extensión con signo de la $constante$, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$if \quad Reg[Ra] = SEXT(constante), \qquad then \quad Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXIII. **Formato de instrucción CMPEQC**

31 – 26	25 – 21	20 – 16	15 – 0
110100	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- CMPLE:

Si el contenido del registro Ra es menor o igual al contenido del registro Rb, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$if \quad Reg[Ra] \leq Rb, \qquad then \quad Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXIV. **Formato de instrucción CMPLE**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
100110	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- CMPLEC:

Si el contenido del registro Ra es menor o igual a la extensión con signo de la $constante$, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$

$$if \quad Reg[Ra] \leq SEXT(constante), \qquad then \quad Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXV. **Formato de instrucción CMPLEC**

31 – 26	25 – 21	20 – 16	15 – 0
110110	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- CMPLT:

Si el contenido del registro Ra es menor que el contenido del registro Rb, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$if \quad Reg[Ra] < Reg[Rb], \qquad then \quad Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXVI. **Formato de instrucción CMPLT**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
100101	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- CMPLTC:

Si el contenido del registro Ra es menor que la extensión con signo de la $constante$, se escribe el valor de 1 en el registro Rc; de lo contrario, se escribe el valor 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$if \quad Reg[Ra] < SEXT(constante), \qquad then \quad Reg[Rc] \leftarrow 1$$
$$else \quad Reg[Rc] \leftarrow 0$$

Tabla XXXVII. **Formato de instrucción CMPLTC**

31 26	25 21	20 16	15 0
110101	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- JMP:

El valor del registro PC actualizado, $PC + 4$, correspondiente a la instrucción que sigue después de la instrucción JMP, se almacena en el registro Rc, luego se almacena el contenido del registro Ra en el registro PC. Los dos bits menos significativos de Ra se encuentran enmascarados para asegurarse de que la dirección de destino EA sea un múltiplo de 4. Ra y Rc pueden hacer referencia al mismo registro puesto que el cálculo de EA utilizando el valor antiguo se hace antes de que cambien los contenidos del registro. El valor de la $constante$ debe ser igual a 0. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$EA \leftarrow Reg[Ra] \,\&\, 0xFFFFFFFC$$

$$Reg[Rc] \leftarrow PC$$
$$PC \leftarrow EA$$

Tabla XXXVIII. **Formato de instrucción JMP**

31 – 26	25 – 21	20 – 16	15 – 0
011011	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- LD:

El contenido de la localidad de memoria con la dirección EA se carga al registro Rc. La dirección de memoria EA se calcula sumando el contenido del registro Ra con la extensión con signo de la $constante$ de desplazamiento. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$EA \leftarrow Reg[Ra] + 4 * SEXT(constante)$$
$$Reg[Rc] \leftarrow Mem[EA]$$

Tabla XXXIX. **Formato de instrucción LD**

31 – 26	25 – 21	20 – 16	15 – 0
011000	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- LDR:

El contenido de la localidad de memoria con la dirección EA se carga al registro Rc. La dirección de memoria EA se calcula multiplicando la extensión con signo de la $constante$ por 4, para convertir el offset de palabras a bytes, y sumando el resultado con el contenido del registro PC actualizado. El registro Ra es ignorado y su valor en la instrucción debe ser 11111, o $R31$. El bit de supervisor, el cual se discute en la subsección 2.6.5, es ignorado al calcular EA. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$EA \leftarrow PC + 4 * SEXT(constante)$$
$$Reg[Rc] \leftarrow Mem[EA]$$

Tabla XL. **Formato de instrucción LDR**

31 26	25 21	20 16	15 0
011111	Rc	11111	*constante en complemento a 2*

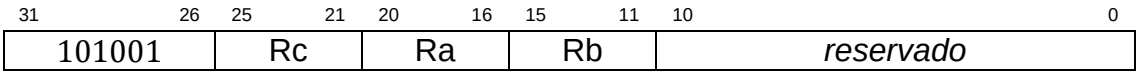
Fuente: elaboración propia.

- OR:

Realiza la función booleana OR bit a bit entre los contenidos del registro Ra y el registro Rb. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \mid Reg[Rb]$$

Tabla XLI. **Formato de instrucción OR**



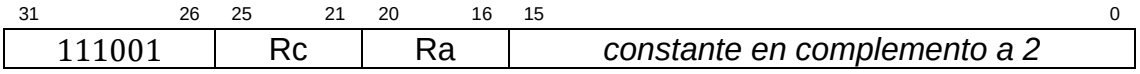
Fuente: elaboración propia.

- ORC:

Realiza la función booleana OR bit a bit entre los contenidos del registro Ra y la extensión con signo de la $constante$. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \mid SEXT(constante)$$

Tabla XLII. **Formato de instrucción ORC**



Fuente: elaboración propia.

- SHL:

El contenido del registro Ra se desplaza una cantidad de 0 a 31 bits a la izquierda, dicha cantidad se especifica con los últimos 5 bits menos significativos del contenido del registro Rb. El resultado se almacena en el registro Rc. Las posiciones vacías causadas por el desplazamiento se reemplazan con ceros. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] << Reg[Rb]_{4:0}$$

Tabla XLIII. **Formato de instrucción SHL**

31 26	25 21	20 16	15 11	10 0
101100	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- SHLC:

El contenido del registro Ra se desplaza una cantidad de 0 a 31 bits a la izquierda, dicha cantidad se especifica con los últimos 5 bits menos significativos de la $constante$. El resultado se almacena en el registro Rc. Las posiciones vacías causadas por el desplazamiento se reemplazan con ceros. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] << constante_{4:0}$$

Tabla XLIV. **Formato de instrucción SHLC**

31 26	25 21	20 16	15 0
111100	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- SHR:

El contenido del registro Ra se desplaza una cantidad de 0 a 31 bits a la derecha, dicha cantidad se especifica con los últimos 5 bits menos significativos del contenido del registro Rb. El resultado se almacena en el registro Rc. Las posiciones vacías causadas por el desplazamiento se reemplazan con ceros. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] >> Reg[Rb]_{4:0}$$

Tabla XLV. **Formato de instrucción SHR**

31 26	25 21	20 16	15 11	10 0
101101	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- SHRC:

El contenido del registro Ra se desplaza una cantidad de 0 a 31 bits a la derecha, dicha cantidad se especifica con los últimos 5 bits menos significativos de la $constante$. El resultado se almacena en el registro Rc. Las posiciones vacías causadas por el desplazamiento se reemplazan con ceros. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] >> constante_{4:0}$$

Tabla XLVI. **Formato de instrucción SHRC**

31 26	25 21	20 16	15 0
111101	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- SRA:

El contenido del registro Ra se desplaza aritméticamente una cantidad de 0 a 31 bits, dicha cantidad se especifica con los últimos 5 bits menos significativos del contenido del registro Rb. El resultado se almacena en el registro Rc. El bit de signo, $Reg[Ra]_{31}$, ocupa las posiciones vacantes. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] >> Reg[Rb]_{4:0}$$

Tabla XLVII. **Formato de instrucción SRA**

31 26	25 21	20 16	15 11	10 0
101110	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- SRAC:

El contenido del registro Ra se desplaza aritméticamente una cantidad de 0 a 31 bits, dicha cantidad se especifica con los últimos 5 bits menos significativos de la $constante$. El resultado se almacena en el registro Rc. El bit de signo,

$Reg[Ra]_{31}$, ocupa las posiciones vacantes. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] >> constante_{4:0}$$

Tabla XLVIII. **Formato de instrucción SRAC**

31 — 26	25 — 21	20 — 16	15 — 0
111110	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- ST:

El contenido del registro Rc se escribe en la localidad de memoria con la dirección EA. Dicha dirección es el resultado de la suma del contenido del registro Ra y la extensión con signo de la $constante$. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$EA \leftarrow Reg[Ra] + SEXT(constante)$$

Tabla XLIX. **Formato de instrucción ST**

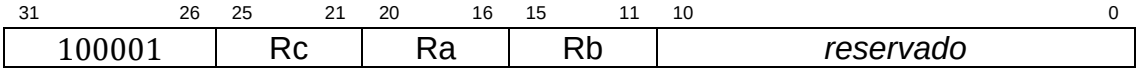
31 — 26	25 — 21	20 — 16	15 — 0
011001	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- SUB:

El contenido del registro Rb se resta del contenido del registro Ra y la diferencia de 32 bits se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

Tabla L. **Formato de instrucción SUB**



Fuente: elaboración propia.

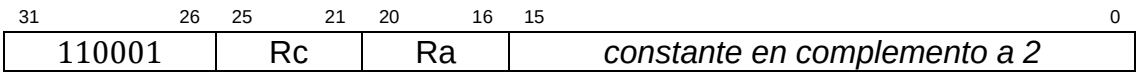
$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] - Reg[Rb]$$

- SUBC:

La extensión con signo de la $constante$ se resta del contenido del registro Ra y la diferencia de 32 bits se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] - SEXT(constante)$$

Tabla LI. **Formato de instrucción SUBC**



Fuente: elaboración propia.

- XOR:

Realiza la función booleana XOR bit a bit entre los contenidos del registro Ra y el registro Rb. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

Tabla LII. **Formato de instrucción XOR**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
100001	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \text{ ^ } Reg[Rb]$$

- XORC:

Realiza la función booleana XOR bit a bit entre el contenido del registro Ra y la extensión con signo de la $constante$. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow Reg[Ra] \text{ ^ } SEXT(constante)$$

Tabla LIII. **Formato de instrucción XORC**

31 – 26	25 – 21	20 – 16	15 – 0
111010	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

- XNOR:

Realiza la función booleana $XNOR$ bit a bit entre los contenidos del registro Ra y el registro Rb. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow \sim(Reg[Ra] \text{ ^ } Reg[Rb])$$

Tabla LIV. **Formato de instrucción XNOR**

31 – 26	25 – 21	20 – 16	15 – 11	10 – 0
101011	Rc	Ra	Rb	*reservado*

Fuente: elaboración propia.

- XNORC:

Realiza la función booleana $XNOR$ bit a bit entre el contenido del registro Ra y la extensión con signo de la $constante$. El resultado se almacena en el registro Rc. Su secuencia de operaciones es la siguiente:

$$PC \leftarrow PC + 4$$
$$Reg[Rc] \leftarrow \sim\big(Reg[Ra] \text{ ^ } SEXT(constante)\big)$$

Tabla LV. **Formato de instrucción XNORC**

31 – 26	25 – 21	20 – 16	15 – 0
111011	Rc	Ra	*constante en complemento a 2*

Fuente: elaboración propia.

2.6.5. Extensión para excepciones

La arquitectura β descrita anteriormente permite el manejo de excepciones e instrucciones privilegiadas. En esta arquitectura se consideran tres tipos de excepciones: trampas, interrupciones y errores.

Las trampas y los errores son el resultado directo de una instrucción y se distinguen en función de los propósitos del programador. Las trampas son intencionales y normalmente corresponden con una solicitud de servicio al sistema operativo. Los errores no son intencionales y a menudo se les asocia con un opcode que no se encuentra dentro del listado especificado en la subsección 2.6.4, en estos casos el opcode se considera como un opcode ilegal.

Las interrupciones son excepciones asíncronas y pueden presentarse en cualquier momento durante la ejecución de alguna instrucción particular, regularmente son causadas por eventos externos en los dispositivos I/O.

2.6.5.1. Modo supervisor

El bit más significativo del PC se designa como el bit de supervisor. La etapa de búsqueda de instrucción y la instrucción LDR ignoran este bit, tratándolo como si tuviera un valor de cero. La instrucción JMP puede limpiar, o asignar el valor 0 a este bit, pero no activarlo, o asignar el valor 1, y las demás instrucciones no tienen efecto sobre él. Sólo las excepciones pueden activar el bit de supervisor.

Cuando el bit de supervisor es igual a 0, se dice que el procesador está en modo usuario, bajo esta condición, las interrupciones se encuentran habilitadas. Cuando este bit es igual a 1, se dice que el procesador está en modo supervisor

o modo kernel, en este caso, las interrupciones se encuentran inhabilitadas y se pueden ejecutar instrucciones privilegiadas.

2.6.5.2. Manejo de excepciones

Para propósitos del manejo de excepciones, se designa el registro 30 como el puntero de excepciones, o registro XP. Cuando ocurre una excepción, el PC actualizado se almacena en el registro XP. Para trampas y errores, este será el PC de la instrucción que sigue a la que causó el error; para interrupciones, este será el PC de la instrucción que sigue a la que estaba a punto de ejecutarse cuando ocurrió la interrupción. El valor de $XP - 4$ indica la dirección de memoria de la instrucción con la que se debe reanudar la ejecución.

Cuando ocurre una excepción y el procesador se encuentra en el modo usuario, el PC actualizado se almacena en el XP, se activa el bit de supervisor, se carga el PC con alguna dirección de memoria que depende de la implementación física, y el procesador comienza la ejecución de instrucciones desde esa dirección. A esta dirección se le conoce como vector de excepción, y su valor depende del tipo de excepción.

Una excepción fundamental y necesaria en toda implementación física es la de reset o reinicio, la cual ocurre antes de que cualquier instrucción sea ejecutada por el procesador. El valor del vector de excepción es siempre 0. De esta forma, cuando se inicializa el sistema, el bit de supervisor está activado, el XP está indefinido y la ejecución de instrucciones comienza con los contenidos de la localidad 0 de memoria.

2.6.6. Registros reservados

Como se señaló anteriormente, el hardware reserva los registros $R30$ y $R31$. Desde el punto de vista del programador, resulta bastante útil reservar una cantidad arbitraria de registros para propósitos específicos, como el puntero de pila y la dirección de retorno para subrutinas; la convención de software de esta arquitectura reserva los registros $R27$, $R28$ y $R29$. En la tabla LVI se muestran los registros reservados en la arquitectura y el propósito para el cual se encuentran designados.

Tabla LVI. **Registros reservados**

Registro	Símbolo	Propósito
$R31$	$R31$	Siempre es cero
$R30$	XP	Puntero de excepciones
$R29$	SP	Puntero de pila
$R28$	LP	Puntero de enlace
$R27$	BP	Puntero base de pila

Fuente: TERMAN, Chris. *Instruction Set Architectures*. www.computationstructures.org/lectures/isa/isa.html. Consulta: 21 de abril de 2020.

3. DISEÑO DEL HARDWARE

En el presente capítulo se especifica el diseño de la implementación del ISA Beta o β ISA, especificado en el capítulo 2, utilizando las técnicas de diseño de circuitos digitales presentadas en el capítulo 1. La estrategia a utilizar está orientada hacia el diseño modular, el cual consiste en la creación de varios bloques lógicos pequeños o módulos, capaces de implementar funcionalidades simples, cuya interconexión permite construir sistemas digitales de mayor complejidad.

La clave del enfoque modular radica en el poder de la abstracción, esto implica que, una vez terminado el diseño del hardware de cada uno de los módulos, es posible representar y describir a cada uno de éstos en términos de sus entradas, salidas y la funcionalidad que implementan. Esto supone una ventaja para los diseñadores al momento de utilizar dichos módulos como componentes en diseños de mayor complejidad, ignorando los detalles del hardware dentro de cada uno de éstos y utilizándolos únicamente en términos de su descripción funcional.

3.1. Método de diseño

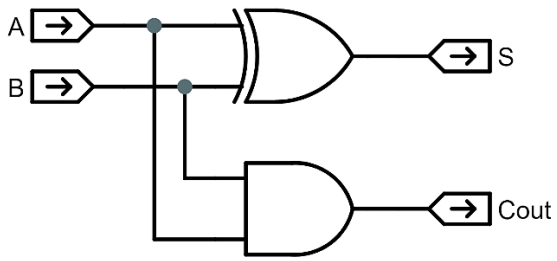
Sin duda, antes de aventurarse en el diseño de un sistema digital complejo, es indispensable definir una estrategia clara y concisa que permita al diseñador alcanzar su objetivo: la implementación física de una funcionalidad, de una manera ordenada y sin complicaciones innecesarias. Esto, además, permite verificar el funcionamiento correcto del diseño, dividiéndolo en varias etapas y

siguiendo un orden específico de evaluación, lo cual disminuye significativamente la predisposición al error por parte del diseñador.

Para este caso particular, se utilizará un enfoque modular, el cual facilita la tarea de diseño al dividir jerárquicamente un sistema en varios módulos funcionales. Esto permite diseñar, analizar y evaluar el comportamiento de cada uno de los módulos por separado para posteriormente ser utilizados en la interconexión del sistema principal.

Una de las principales ventajas de este enfoque es que permite crear, compartir y utilizar módulos utilizando el concepto de abstracción, el cual permite aislarlos de su contexto y concentrarse únicamente en la funcionalidad que proveen, ignorando los detalles de los circuitos digitales que los implementan. Esto quiere decir que, para poder utilizar un módulo del cual se desconocen los detalles de la implementación, sólo es necesario tener una descripción clara y concisa de las entradas, salidas y la funcionalidad del mismo.

Figura 66. **Módulo semisumador**



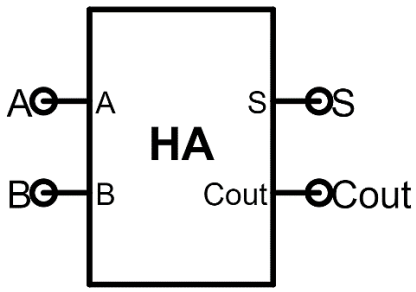
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para ilustrar esta idea, considérese el caso del circuito semisumador de la figura 66, el cual se encarga de sumar dos bits de entrada, en este caso A y B, y

mostrar el resultado en la salida S y el valor del acarreo de salida en $Cout$. Obsérvese la declaración explícita de la dirección del flujo de la información, las flechas salen de los nombres de las entradas y entran a los nombres de las salidas.

La representación abstracta de este circuito o módulo se muestra en la figura 67, en ella únicamente se especifican las entradas y salidas del módulo, además, se coloca una pequeña abreviación, en este caso HA, la cual sirve de identificador. Esta representación debe acompañarse de una descripción clara de la relación funcional existente entre las entradas y salidas del módulo.

Figura 67. **Abstracción del módulo semisumador, HA**

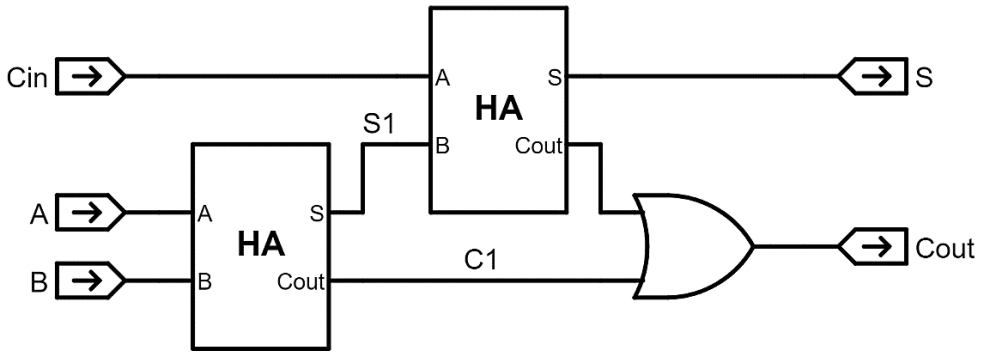


Fuente: elaboración propia, empleando Jade Circuit Simulator.

El módulo medio sumador de la figura 67 se puede utilizar en el diseño de otros sistemas digitales, tal es el caso del circuito sumador completo, o FA, de la figura 68, que utiliza dos módulos HA y una compuerta OR para realizar la suma de las entradas A, B y el acarreo de entrada Cin para mostrarlas el resultado en la salida S y el valor del acarreo de salida en $Cout$. Nótese que, en este caso, se han utilizado las señales internas $S1$ y $C1$ para denotar la suma y el acarreo parcial del primer módulo HA, sin embargo, estas señales sólo sirven para

realizar conexiones internas dentro del módulo FA y no se incluyen en la representación abstracta de éste.

Figura 68. **Módulo sumador completo**

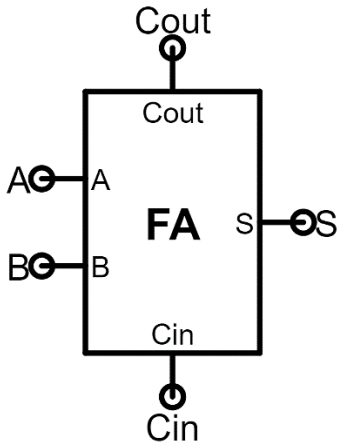


Fuente: elaboración propia, empleando Jade Circuit Simulator.

La figura 69 muestra la representación abstracta del módulo sumador completo o FA, en ella se indican explícitamente las entradas y salidas. Este módulo a su vez, se puede utilizar en el diseño de otros circuitos digitales de mayor complejidad, por ejemplo, un sumador de dos números de 32 bits.

Siguiendo este simple y ordenado procedimiento de diseño, es posible crear grandes sistemas digitales evitando complicaciones innecesarias al organizar el sistema jerárquicamente en distintos niveles de abstracción. Esto supone una gran ventaja, debido a que permite segmentar el diseño en módulos de menor complejidad que a su vez, se dividen en módulos cada vez más pequeños y menos complejos, agilizando el proceso de desarrollo y evaluación del rendimiento del sistema general.

Figura 69. **Abstracción del módulo sumador completo**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Debido a que el ISA Beta indica los requerimientos funcionales que debe cumplir el hardware del procesador, se utilizará una metodología de diseño Top-Bottom, la cual comienza con la descripción del diseño en términos de los módulos de más alto nivel y, siguiendo un orden jerárquico, termina con el diseño de todos los módulos de más bajo nivel dentro del sistema.

3.2. Diseño del datapath

Como se mencionó anteriormente en el capítulo 2, el conjunto de instrucciones Beta o ISA Beta, cuenta con treinta y dos registros de 32 bits encargados de almacenar información para ser procesada dentro del datapath. Si se observa cuidadosamente el esquema de codificación del conjunto de instrucciones, se pueden identificar tres clases de instrucciones en función del código de operación u opcode. Dichas clasificaciones se describen a continuación:

- ALU: Operaciones lógicas o aritméticas entre los contenidos de los registros operandos Ra y Rb, cuya salida se almacena en el contenido del registro Rc. Los dos bits más significativos de su código de operación son equivalentes a $0b10$.
- ALU con constante: Operaciones lógicas o aritméticas entre el contenido del registro operando Ra y la extensión con signo de una constante de 16 bits cuya salida se almacena en el contenido del registro Rc. Los dos bits más significativos de su código de operación son equivalentes a $0b11$.
- Acceso a memoria y saltos: Operaciones que acceden a memoria o cambian la secuencia de ejecución de las instrucciones. Los dos bits más significativos de su código de operación son equivalentes a $0b01$.

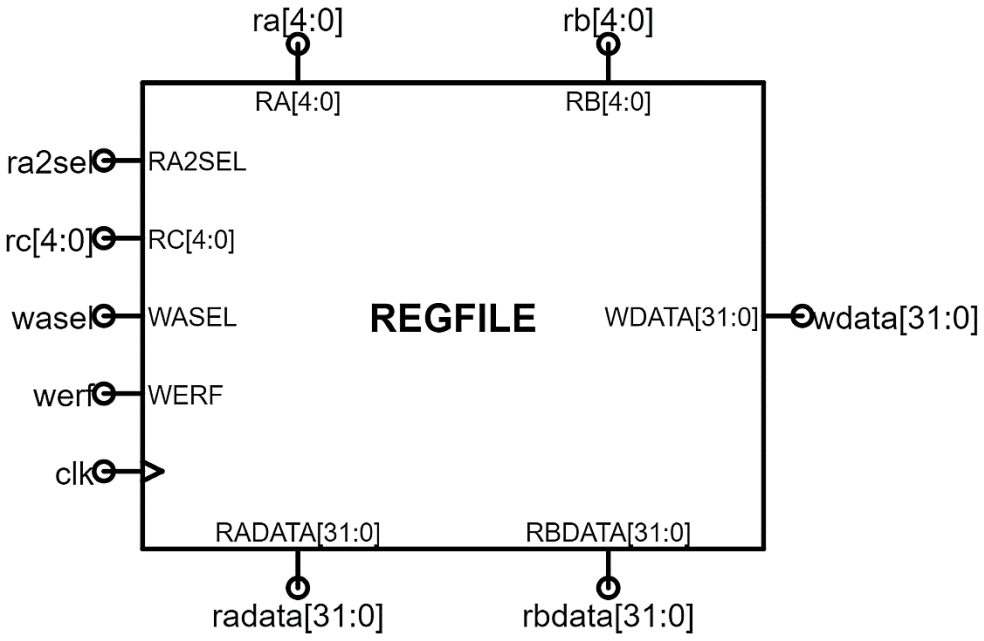
Estas clasificaciones de instrucciones utilizan solamente 2 formatos de codificación. Más adelante el lector podrá apreciar la importancia de una arquitectura RISC, en la que existen formatos reducidos de instrucciones, al observar una notable simplificación del hardware requerido para implementar toda la funcionalidad del Beta ISA.

Al segmentar el conjunto de instrucciones en distintas clasificaciones o categorías, es posible diseñar el datapath siguiendo una estrategia incremental. A partir de esta idea, primero se diseña la lógica necesaria para ejecutar las instrucciones tipo ALU, con y sin constante, posteriormente, se añade la lógica para ejecutar las instrucciones de acceso a memoria y los saltos, y, por último, se agrega el hardware necesario para manejar los casos en los que ocurran excepciones, como los opcodes ilegales y las interrupciones.

Antes de comenzar con la construcción del datapath, es necesario realizar una descripción de la representación abstracta de los componentes a utilizar. El primer componente fundamental es el bloque de registros de múltiples puertos,

el cual se encarga de almacenar información de manera temporal dentro del procesador, su representación abstracta se muestra en la figura 70, la cual contiene 6 entradas, incluida la señal de reloj, y 2 salidas.

Figura 70. **Bloque de registros de múltiples puertos, REGFILE**



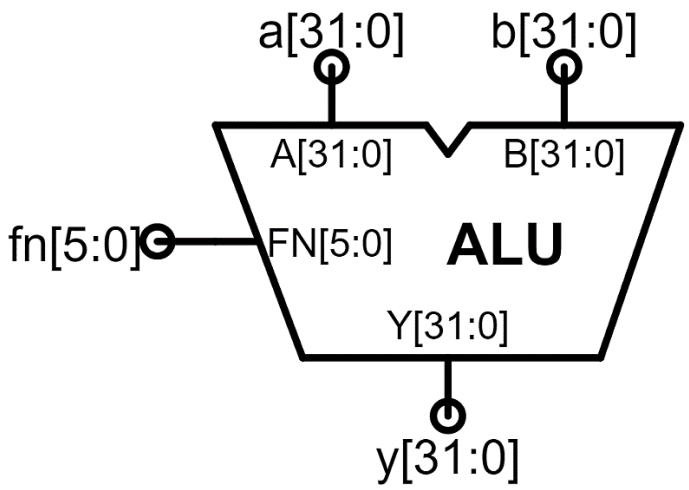
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Las entradas, $RA[4:0]$, $RB[4:0]$ y $RC[4:0]$, especifican las direcciones de los registros Ra, Rb y Rc respectivamente, después de un retardo de propagación, los contenidos de los registros Ra y Rb se reflejan en los puertos de salida $RADATA[31:0]$ y $RBDATA[31:0]$; mientras tanto, la entrada $WERF$, habilita la escritura de la información del puerto de entrada $WDATA[31:0]$ en el contenido del registro de destino especificado por $RC[4:0]$; aquí es necesario resaltar que, debido a que el bloque de registros depende de la señal de reloj clk, el nuevo

valor en el registro Rc se almacena al final de del presente ciclo de reloj, cuando ocurre el flanco de subida del siguiente ciclo.

Nótese que no se describieron las entradas $RA2SEL$ y $WASEL$, esto se hará en las siguientes subsecciones, conforme se vaya construyendo el datapath, puesto que no son señales fundamentales para entender el funcionamiento general del bloque.

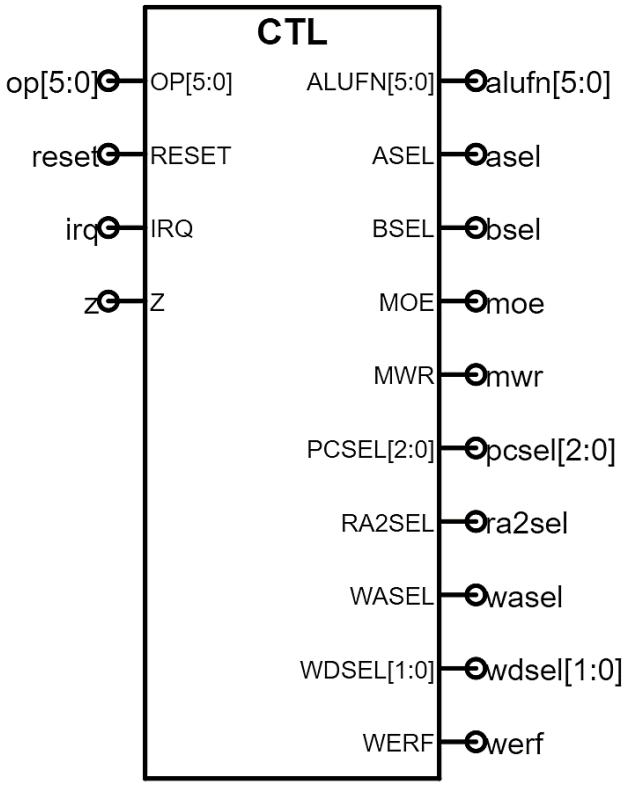
Figura 71. **Unidad aritmética lógica, ALU**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

La unidad aritmética lógica o ALU cuya abstracción se muestra en la figura 71, es la encargada de realizar las operaciones sobre los contenidos de los puertos de salida del bloque de registros, ésta toma las entradas $A[31:0]$ y $B[31:0]$, como sus operandos para realizar una de sus diferentes operaciones en función de la entrada $FN[5:0]$ y reflejar su resultado, después de que haya transcurrido su retardo de propagación, en la salida $Y[31:0]$.

Figura 72. **Lógica de control, CTL**



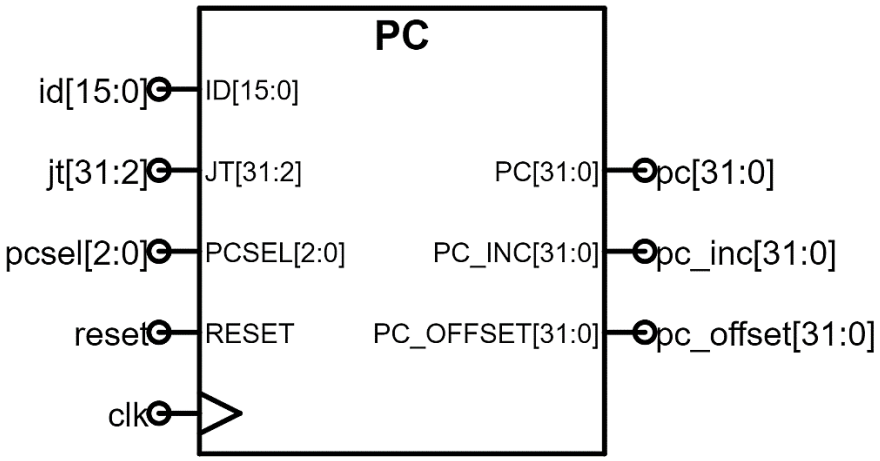
Fuente: elaboración propia, empleando Jade Circuit Simulator.

La abstracción de la lógica de control, encargada de producir las señales necesarias para la ejecución de las instrucciones, se muestra en la figura 72, las terminales de la izquierda denotan las entradas, mientras que a la derecha se ubican las salidas. Debido a la gran cantidad de entradas y salidas, la funcionalidad y relevancia de cada una de ellas se explicará a medida que se vaya construyendo el diseño el datapath.

Por último, el módulo contador de programa, PC, es el encargado de llevar el control de la secuencia de ejecución de los programas, su representación

abstracta se muestra en la figura 73, posee 3 salidas y 5 entradas que serán discutidas en las siguientes subsecciones.

Figura 73. **Contador de programa, PC**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

3.2.1. Instrucciones tipo ALU

En la sección 2.3 se introdujo la secuencia de operaciones necesaria para ejecutar una instrucción en una máquina Von Neumann, la cual se repite aquí por conveniencia:

- Búsqueda de la instrucción para el presente ciclo de reloj.
- Decodificación de la instrucción, la cual se encuentra dentro de las tres categorías descritas anteriormente.
- Lectura de los operandos del bloque de registros.
- Ejecución de la operación.
- Escritura del resultado de la operación en el registro de destino.

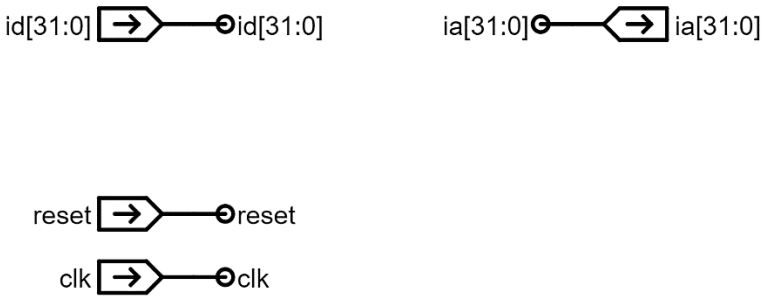
- Cálculo del siguiente valor del contador de programa.

Para todas las instrucciones del ISA Beta, sin importar la categorización hecha en la sección 3.2, se deben ejecutar todos los pasos descritos anteriormente. Considerando este hecho, primero se diseña el hardware necesario para ejecutar las instrucciones tipo ALU, con y sin constante, y posteriormente se añade la circuitería necesaria para el resto de las instrucciones, siguiendo el enfoque incremental mencionado previamente.

Antes de comenzar, es indispensable aclarar que el presente diseño permite ejecutar únicamente una instrucción por ciclo de reloj, un criterio simple que, para los propósitos del presente trabajo, permite al lector familiarizarse de una manera más fácil e intuitiva con el hardware contenido dentro del procesador.

Como punto de partida, es necesario diseñar el hardware para hacer la búsqueda de la instrucción a ejecutar en el actual ciclo de reloj. En la figura 74 se muestra el conjunto de entradas y salidas que permiten obtener la siguiente instrucción a ejecutar, se muestra la entrada de la señal de reloj clk, la entrada binaria $reset$, encargada de restaurar el valor del PC con la dirección predeterminada de reinicio o encendido. La salida $ia[31:0]$ indica la dirección de memoria en donde se encuentra ubicada la instrucción a ejecutar y, después del tiempo de retardo por acceso a memoria, la entrada $id[31:0]$ contiene la instrucción obtenida desde la memoria.

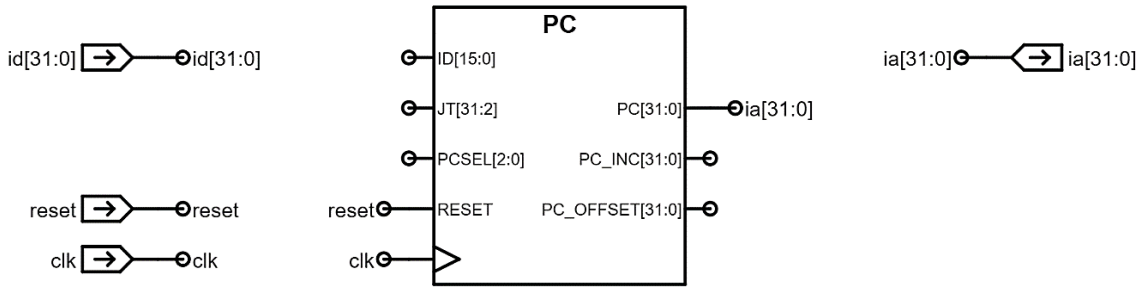
Figura 74. **I/Os para la etapa de búsqueda**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

El módulo encargado de llevar el control del contenido del contador de programa es el módulo PC, en la figura 75 se muestra la conexión para la etapa de búsqueda. Las señales clk y $reset$ se conectan a los puertos de entrada correspondientes del módulo y, el valor del puerto de salida $PC[31:0]$ representa la dirección de la instrucción a ejecutar, por lo que éste se conecta con la señal $ia[31:0]$.

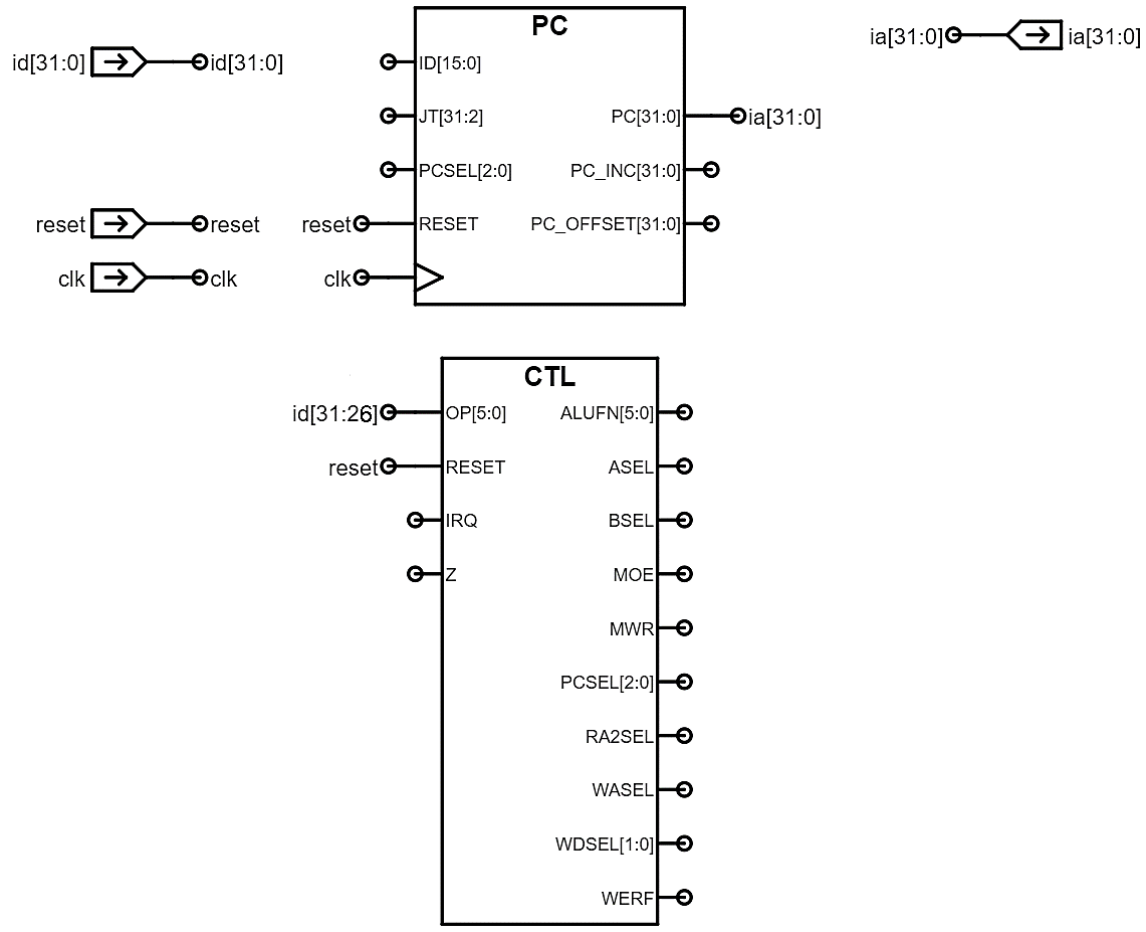
Figura 75. **Búsqueda de la instrucción**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Nótese que, las señales con el mismo nombre dentro de un módulo se encuentran conectadas entre sí de manera implícita. Para los diagramas de circuitos en este diseño, se evita el uso de conexiones explícitas debido a la complejidad del sistema, salvo en algunas excepciones.

Figura 76. **Decodificación del OPCODE**

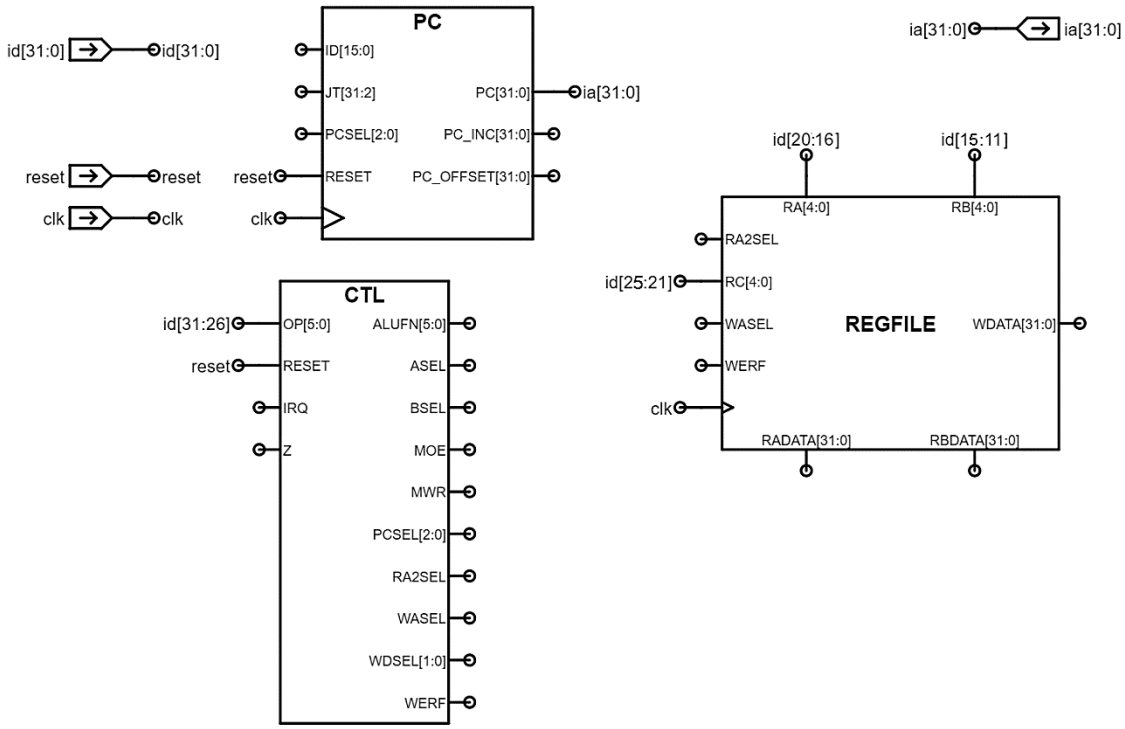


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Una vez recibida la instrucción desde la memoria principal, se procede a decodificarla, como primer paso se extrae el opcode, es decir, $id[31:26]$, y se

conecta con la entrada de la lógica de control $OP[5:0]$, el cual produce las señales de control del datapath. El módulo CTL y su interconexión se muestran en la figura 76, nótese que, además de los bits del opcode también se conecta la entrada $reset$, debido a que el datapath genera un conjunto de señales de control específicas para la condición de encendido o reinicio.

Figura 77. **Obtención de los operandos Ra, Rb y Rc**

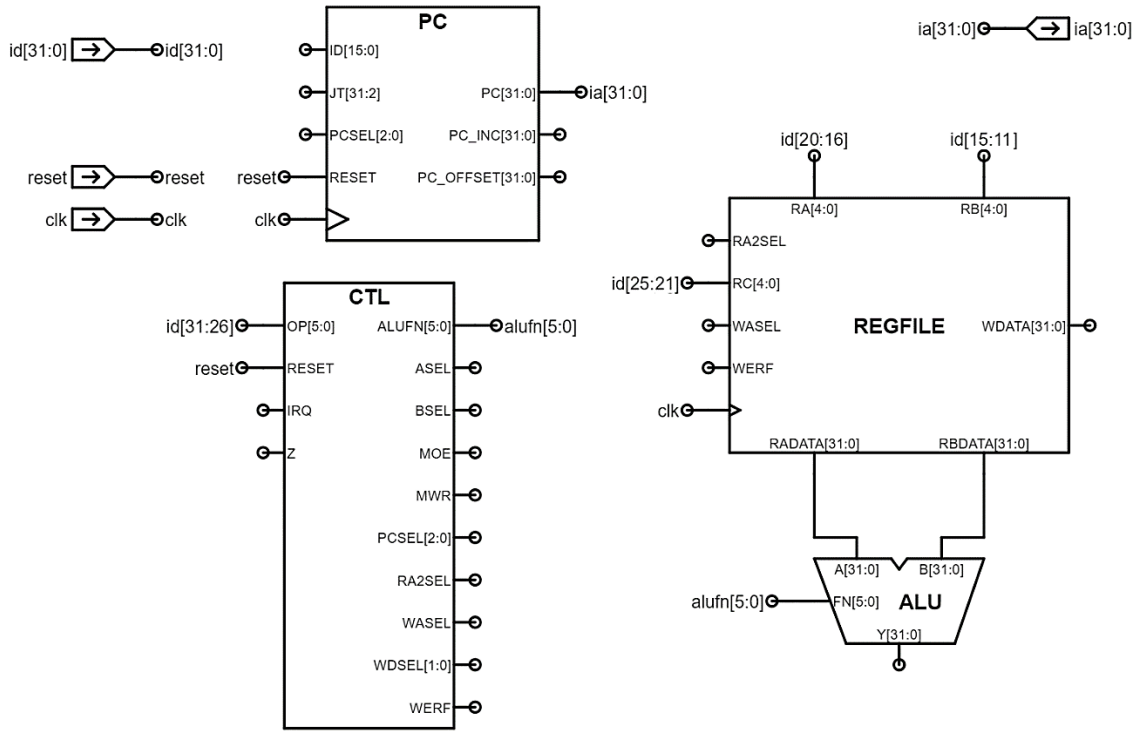


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Al mismo tiempo que ocurre la decodificación del opcode, los campos que contienen los operandos y el registro de destino se utilizan sin ningún tipo de modificación previa, y se conectan a las entradas respectivas del bloque de múltiples puertos, tal y como se muestra en la figura 77. Específicamente, las señales $id[25:21]$, $id[20:16]$ y $id[15:11]$ se conectan a los puertos de entrada

$RC[5:0]$, $RA[5:0]$ y $RB[5:0]$ respectivamente. En caso de no identificar por qué se utilizan determinados fragmentos de la señal $id[31:0]$, se recomienda al lector revisar la estructura de los formatos de las instrucciones del ISA expuestos en la sección 2.6.

Figura 78. **Ejecución de la operación**

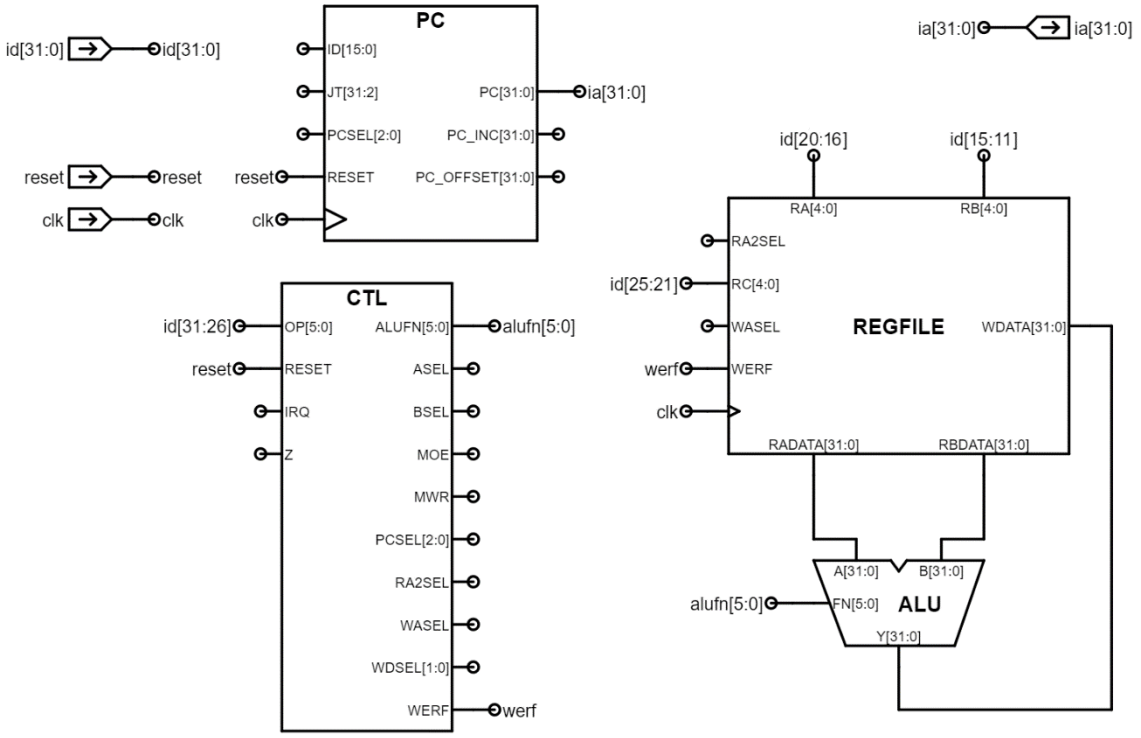


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Posteriormente, las señales de salida de los puertos $RADATA[31:0]$ y $RBDATA[31:0]$ se conectan a los puertos de entrada de la ALU, $A[31:0]$ y $B[31:0]$, respectivamente, la señal de control $alufn[5:0]$ indica la operación específica que se debe ejecutar sobre los dos operandos y, por lo tanto, se conecta al puerto de entrada $FN[5:0]$. Dicha conexión de señales se muestra

explícitamente en la figura 78, después del retardo de propagación de la ALU, el resultado de la operación se muestra en el puerto de salida $Y[31:0]$.

Figura 79. **Escritura en el registro de destino**

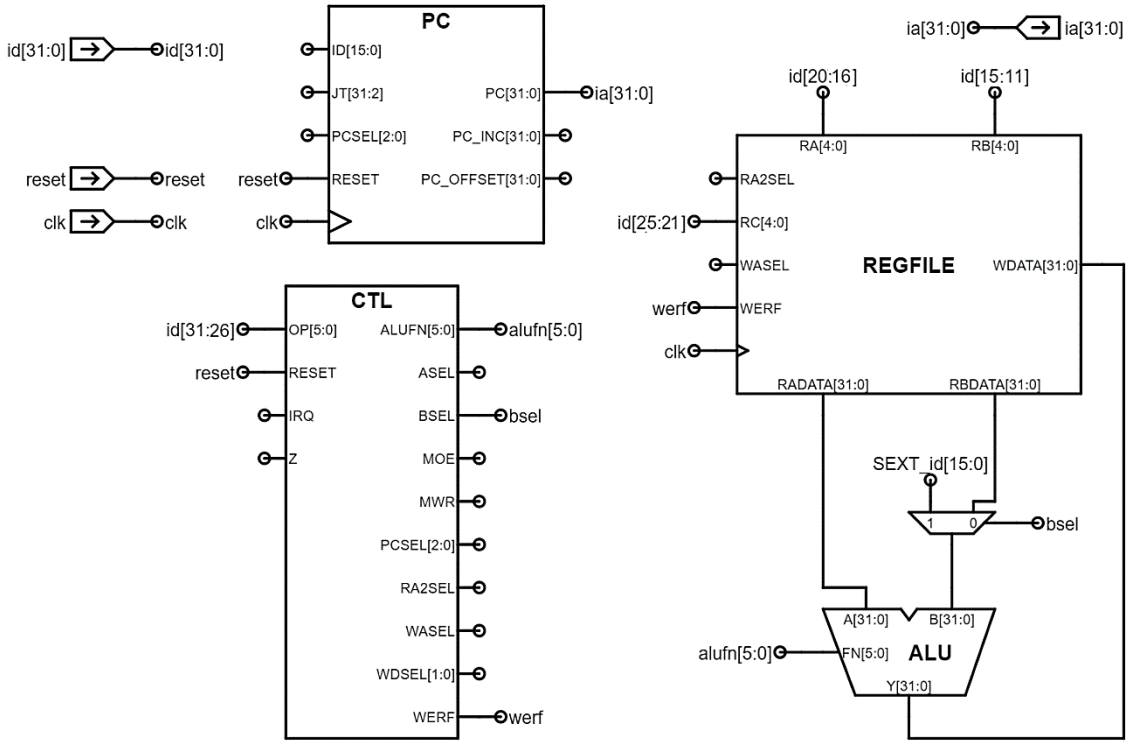


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Por último, el resultado de la operación que se encuentra en el puerto $Y[31:0]$ se conecta con el puerto $WDATA[31:0]$, para escribir la información en el registro de destino, a través de la señal de control $werf$, se habilita la escritura en el registro especificado por la señal $id[26:22]$ (RC). Este proceso se encuentra ilustrado en la figura 79, nótese que, el nuevo valor del registro de destino, se cargará en el flanco de subida del siguiente ciclo de reloj.

Todos los pasos descritos anteriormente, deben llevarse a cabo en un intervalo no mayor al período de la señal de reloj, por lo tanto, es indispensable conocer los retardos de propagación de cada uno de los bloques para calcular el retardo de propagación máximo y así, determinar la máxima frecuencia de reloj que se puede utilizar.

Figura 80. **Extensión para instrucciones con constante**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para las operaciones tipo ALU con constante, únicamente se necesita agregar un multiplexor que permita seleccionar entre dos señales de entrada para el puerto $B[31:0]$. En la figura 80 se muestra dicho hardware adicional, la señal de control $bsel$, conectada a la señal de control del multiplexor, permite

seleccionar entre el puerto $RBDATA[31:0]$ y la extensión con signo de la constante $SEXT\_id[15:0]$.

No se necesita de hardware adicional para realizar la extensión con signo de la constante, esta operación se puede realizar utilizando únicamente 16 señales binarias adicionales cuyo valor es el mismo y, equivalente al bit más significativo de la constante de 16 bits, es decir, el bit $id[15]$.

El cálculo de la siguiente instrucción a ejecutar se realiza dentro del módulo PC, el cual contiene un sumador de 32 bits que se encarga de incrementar el valor actual del registro PC en 4 unidades y mostrar el resultado en el puerto de salida $PC\_INC[31:0]$, y en el puerto $PC[31:0]$ al inicio del siguiente ciclo de reloj. Es necesario resaltar que esta operación se realiza justo después de que se obtiene la instrucción de memoria, esto significa que ocurre de manera simultánea con la fase de decodificación.

En las tablas LVII y LVIII se muestran los valores de las señales de control para las instrucciones tipo ALU y ALUC respectivamente, nótese que el valor de la señal $alufn[5:0]$ es una función del opcode y, por lo tanto, varía dependiendo de la instrucción a ejecutar.

Tabla LVII. **Señales de control para instrucción tipo ALU**

Señal	Valor
$bsel$	0
$alufn[5:0]$	$f(opcode)$
$werf$	1

Fuente: elaboración propia.

Tabla LVIII. **Señales de control para instrucción tipo ALUC**

Señal	Valor
$bsel$	1
$alufn[5:0]$	$f(opcode)$
$werf$	1

Fuente: elaboración propia.

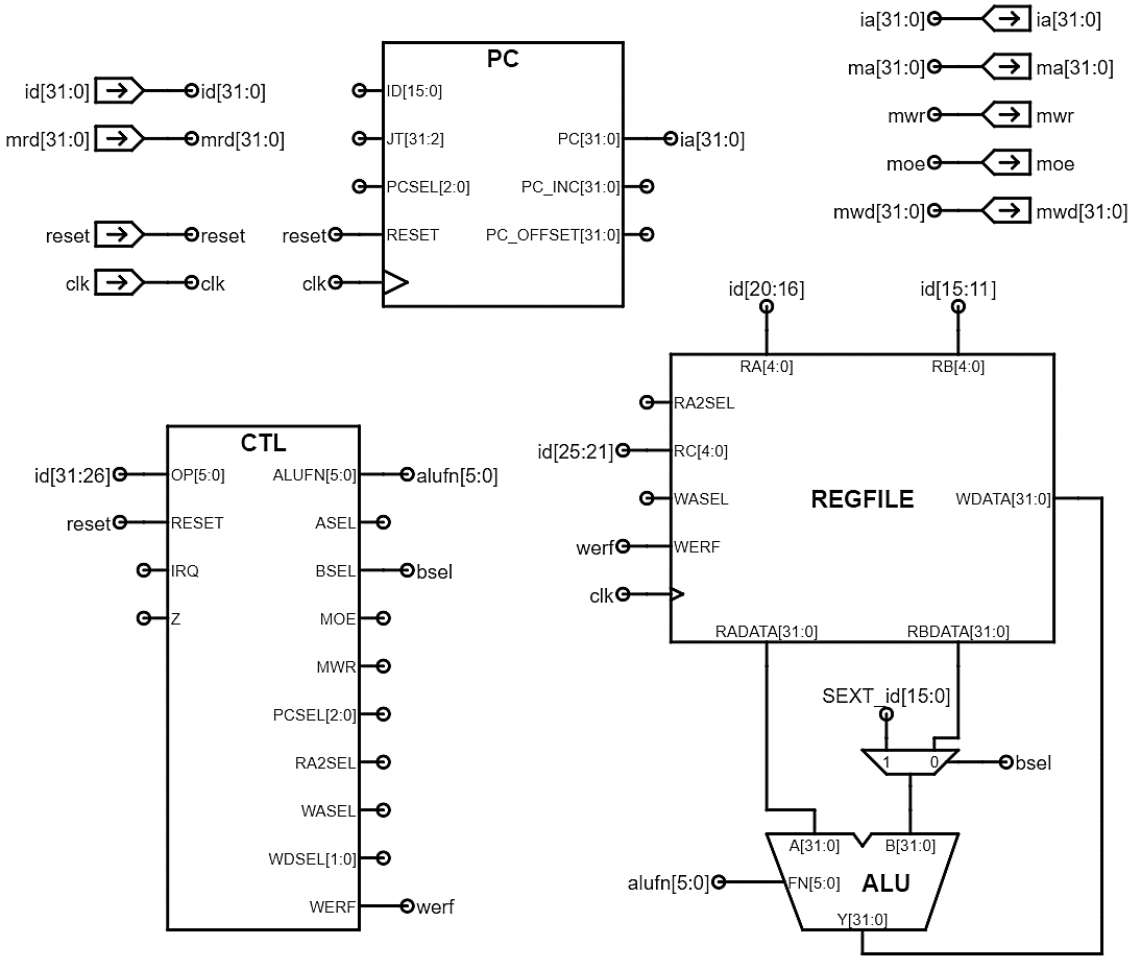
Sorprendentemente, el hardware presentado en esta sección, es capaz de ejecutar muchas de las instrucciones del ISA Beta, únicamente se necesita agregar la lógica para ejecutar las instrucciones de acceso a memoria y de saltos.

3.2.2. Instrucciones de acceso a memoria

Para acceder a los datos almacenados en la memoria principal a través de las instrucciones LD y ST, se necesitan agregar cuatro puertos de salida y uno de entrada al diseño del datapath, estos puertos se muestran en la figura 81. El significado y la relevancia de cada una de estas señales se explican conforme se avanza en el diseño del hardware.

A estas alturas, es necesario aclarar que esta memoria es la misma que almacena las instrucciones. Para los propósitos del presente diseño, la memoria principal se divide en tres puertos: dos puertos de lectura para leer las instrucciones y cargar datos, y un puerto de escritura para almacenar datos. El primer puerto de lectura se utiliza únicamente para la etapa de búsqueda de la instrucción, mientras que los dos restantes se utilizan en esta sección.

Figura 81. **Puertos adicionales para accesos a memoria**

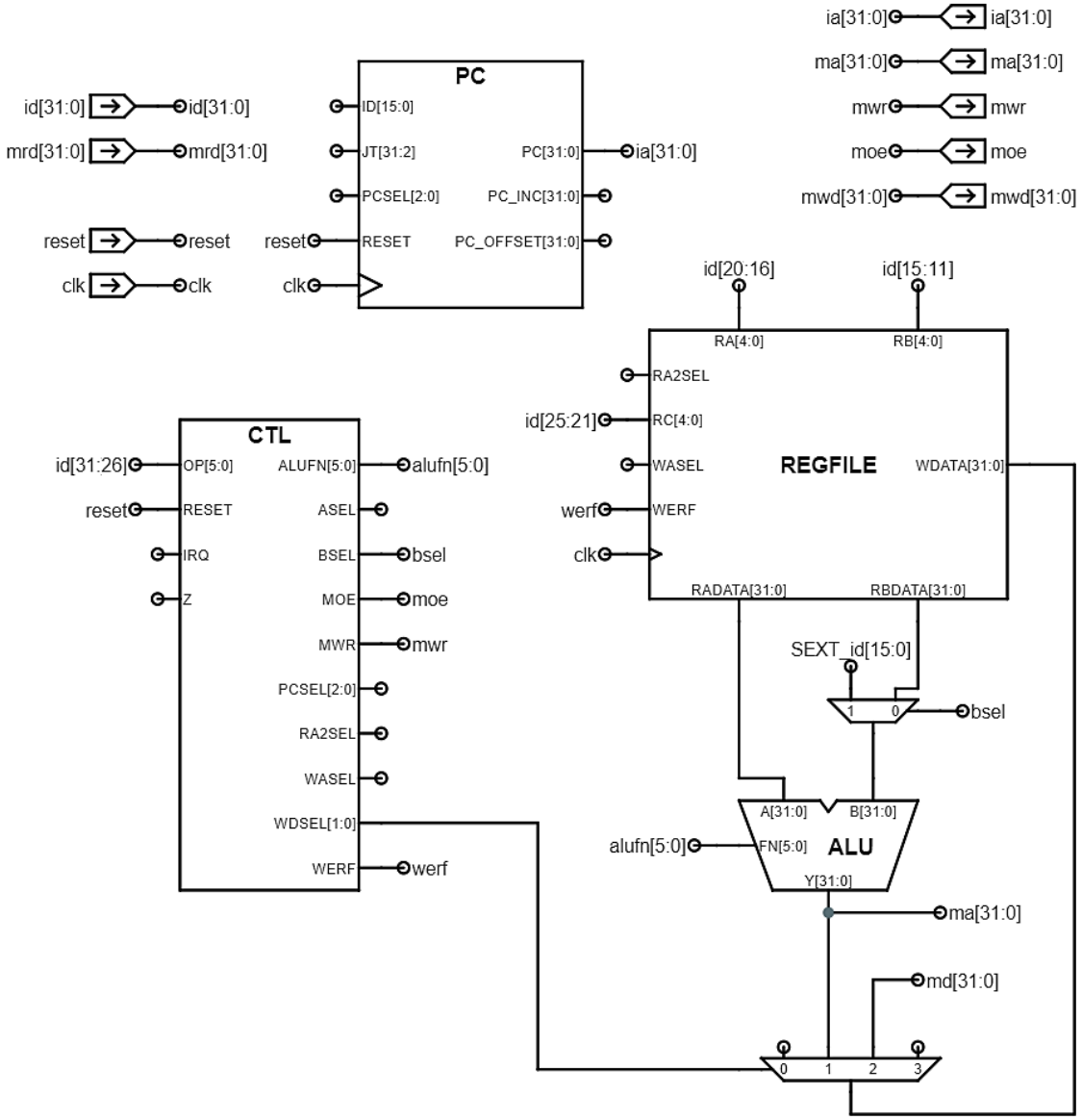


Fuente: elaboración propia, empleando Jade Circuit Simulator.

3.2.2.1. Load, LD

El cálculo de la dirección para ambas instrucciones de memoria es exactamente el mismo realizado por la instrucción ADDC: el contenido del registro Ra se suma con la extensión con signo de la constante de 16 bits. Por lo tanto, se utiliza el hardware ya existente para realizar dicha operación.

Figura 82. **Hardware para instrucción LD**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para el caso de la instrucción LD, el puerto de salida $Y[31:0]$ de la ALU se conecta a la señal $ma[31:0]$, que indica la dirección de localidad de memoria a la que se desea acceder. Después del retardo de propagación por acceso a

memoria, el contenido de dicha localidad se encuentra disponible en la señal $mrd[31:0]$, la cual se debe conectar al puerto de escritura del bloque de registros.

En la figura 82 se muestra el datapath con el hardware adicional para ejecutar la instrucción de carga LD, el puerto $WDSEL[1:0]$ se conecta a la entrada de control del multiplexor de 4 líneas, el cual permite seleccionar entre la señal de salida del puerto $Y[31:0]$ de la ALU y la señal $mrd[31:0]$ proveniente de memoria, como posibles contenidos a escribir en el registro de destino Rc. La señal de control moe permite habilitar la lectura de datos desde la memoria principal, por otra parte, la señal mwr habilita la escritura de datos en memoria.

En la tabla LIX se muestran los valores de las señales de control para para la ejecución de la instrucción LD, la señal $alufn[5:0]$ se asocia con el valor $A + B$, para indicar que la ALU debe realizar la suma entre los dos operandos, además, nótese que, el valor de mwr debe ser equivalente a cero para evitar escribir accidentalmente en la memoria.

Tabla LIX. **Señales de control para instrucción LD**

Señal	Valor
$bsel$	1
$alufn[5:0]$	$A + B$
$werf$	1
$wdsel[1:0]$	$0b10$
mwr	0
moe	1

Fuente: elaboración propia.

3.2.2.2. Store, ST

La ejecución de la instrucción ST es similar a la de la instrucción LD, con una pequeña diferencia. El valor a escribir en la memoria principal proviene del registro Rc, pero hasta ahora, no existe una conexión directa entre este registro y la señal $mwd[31:0]$, la cual se encuentra conectada al puerto de escritura de memoria.

Tabla LX. **Señales de control para la instrucción ST**

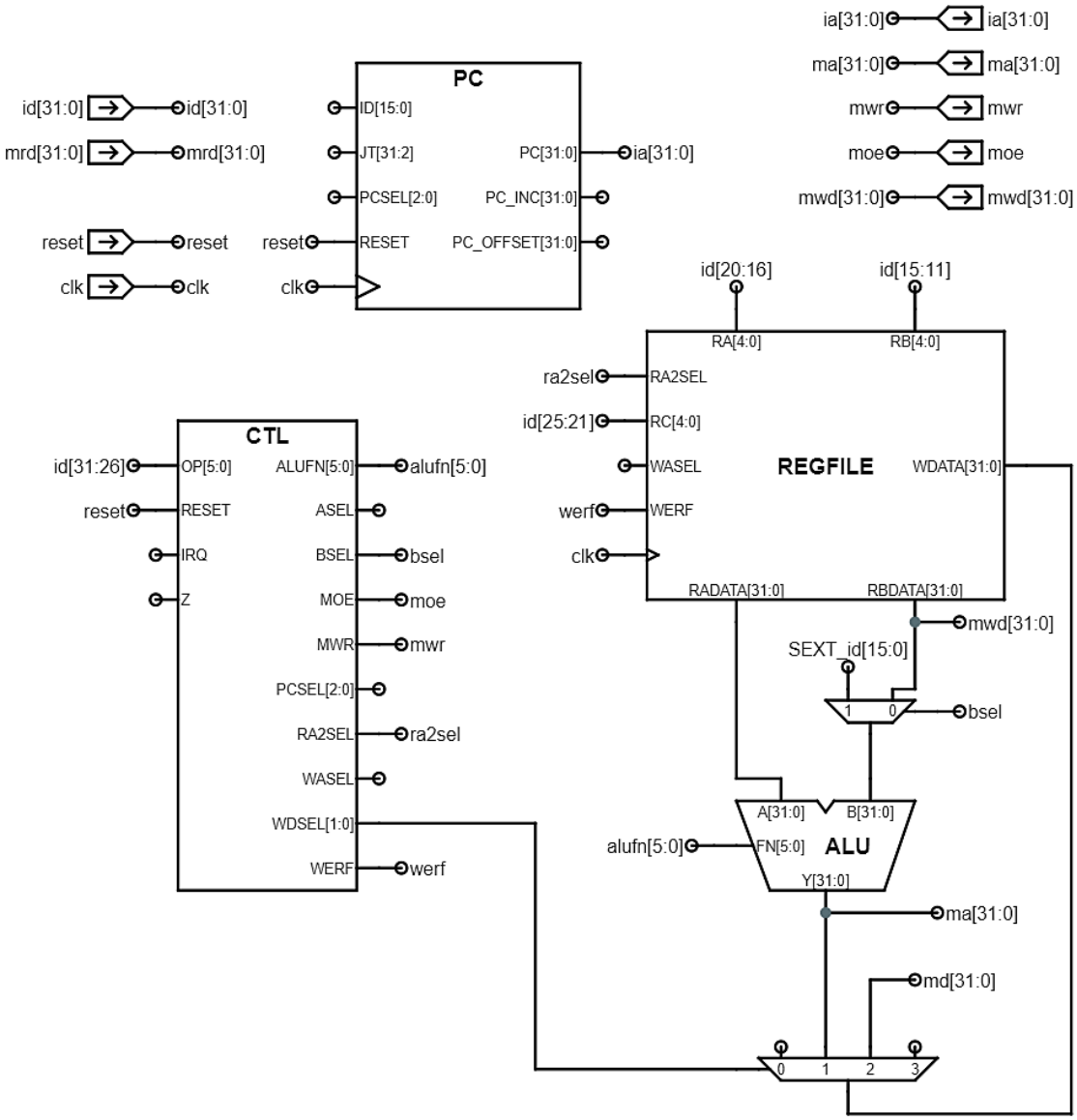
Señal	Valor
$bsel$	1
$alufn[5:0]$	$A + B$
$werf$	0
$wdsel[1:0]$	X
mwr	1
moe	0
$rasel$	1

Fuente: elaboración propia.

Debido a que la ALU utiliza la señal $SEXT\_id[15:0]$ como segundo operando, el puerto $RBDATA[31:0]$ queda disponible para otro uso. La señal de control $ra2sel$ permite seleccionar entre la dirección especificada por el puerto $RC[4:0]$ y el puerto $RB[4:0]$ para mostrar el contenido del registro del puerto $RBDATA[31:0]$. Cuando la señal $ra2sel$ es igual a 0, se selecciona el registro Rb y, si es igual a 1, se selecciona el registro Rc.

El puerto de lectura $RBDATA[31:0]$ se conecta con la señal $mwd[31:0]$, creando una ruta entre el contenido del registro Rc y la localidad de memoria especificada por $ma[31:0]$.

Figura 83. **Hardware para instrucción ST**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Debido a que la instrucción ST es la única que no realiza el proceso de escritura en el registro de destino, el valor de la señal $werf$ debe ser cero. En la figura 83 se muestran las señales adicionales en el diseño general del datapath.

En la tabla LX se muestran los valores de las señales de control necesarios para ejecutar la instrucción ST, el valor de la señal $wdsel[1:0]$ es irrelevante o X debido a que no se escribe ningún valor en el bloque de registros.

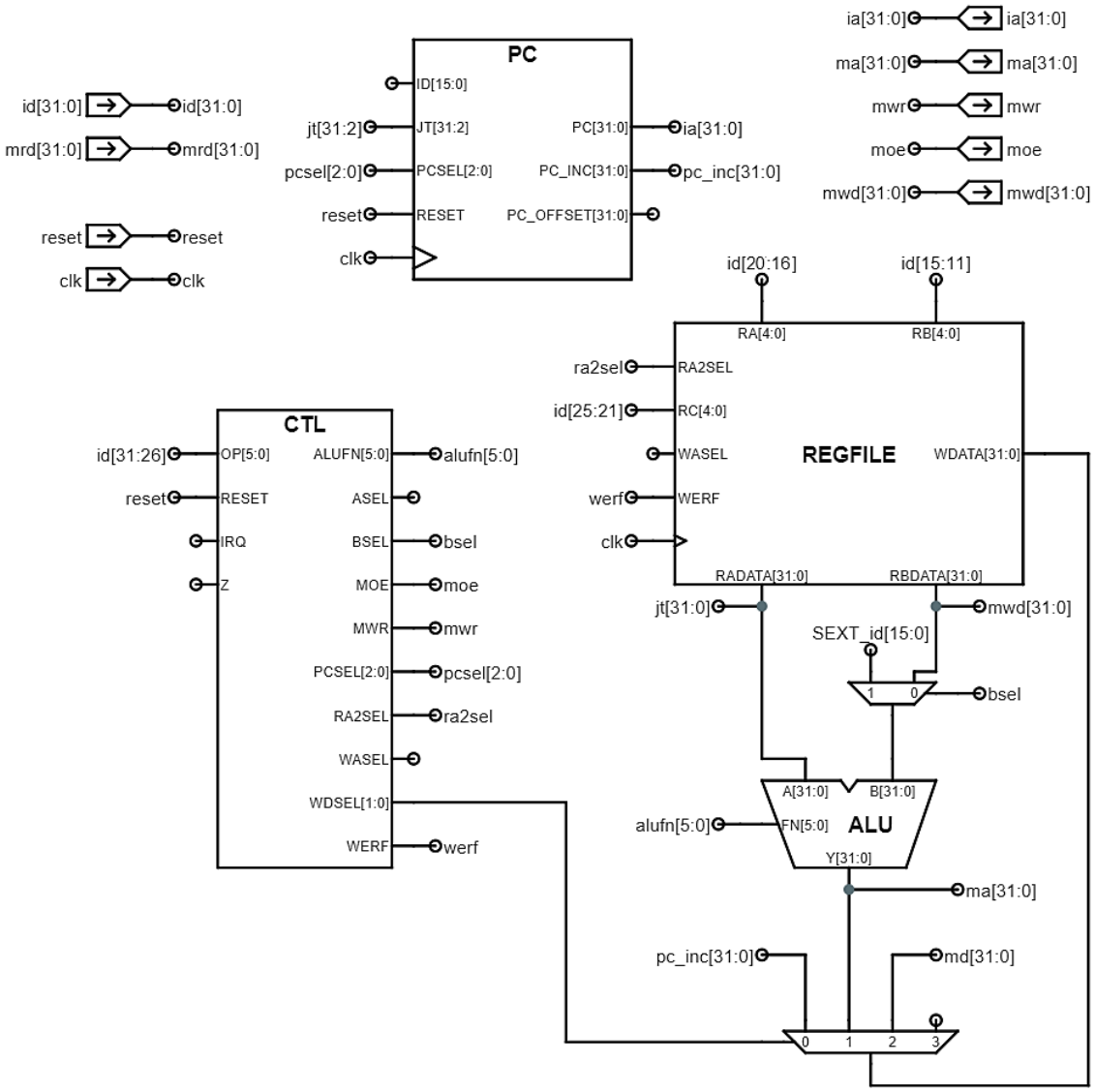
3.2.3. Instrucciones de salto

Hasta ahora, la dirección de la siguiente instrucción se ha calculado a partir de la dirección de la instrucción actual, en otras palabras, el hardware únicamente tiene la capacidad para ejecutar instrucciones de manera secuencial. A continuación, se presenta el hardware adicional para ejecutar las instrucciones de salto, estas permiten controlar el orden de la ejecución de las instrucciones a través de la modificación arbitraria del siguiente valor del registro PC.

3.2.3.1. Jump, JMP

La instrucción JMP utiliza el contenido del registro Ra como el siguiente valor del registro PC, sin embargo, se deben suprimir los dos bits menos significativos en contenido del registro Ra para asegurarse de que la dirección con la cual se cargará el registro PC sea un múltiplo de 4, dando como resultado, la señal de 30 bits $jt[31:2]$, una variación de los contenidos del registro Ra. El módulo PC se encarga de agregar los 2 ceros, al final de la cadena, para convertirla en una dirección de 32 bits.

Figura 84. **Hardware para la instrucción JMP**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Puesto que ahora, existen dos posibles valores para el siguiente valor del registro PC, se necesita una señal de control que permita seleccionar entre estos en función del tipo de instrucción que se encuentra en ejecución. Este es el

propósito de la señal $pcsel[2:0]$, el motivo por el cual es una señal de 3 bits en lugar de 1 bit se debe a que todavía no se ha incluido el hardware para los saltos condicionales y las excepciones, los cuales agregan más posibilidades para el siguiente valor del PC.

Por último, debido a que la instrucción JMP también almacena el valor $PC + 4$ en el registro de destino Rc, se conecta la señal $pc\_inc[31:0]$ a la entrada 0 del multiplexor controlado por la señal $wdsel[1:0]$. El hardware adicional para ejecutar la instrucción JMP se encuentra en la figura 84.

Tabla LXI. **Señales de control para la instrucción JMP**

Señal	Valor
$bsel$	X
$alufn[5:0]$	X
$werf$	1
$wdsel[1:0]$	$0b00$
mwr	0
moe	X
$ra2sel$	X
$pcsel[2:0]$	$0b010$

Fuente: elaboración propia.

En la tabla LXI se muestran los valores de las señales de control necesarios para la ejecución de la instrucción JMP, nótese que el valor de mwr debe ser igual a cero, para evitar escribir información en la memoria de manera accidental. De nuevo, existen varias señales cuyo valor es irrelevante o X, algo que definitivamente ayuda a simplificar los diseños de la lógica de control.

3.2.3.2. Saltos condicionales, BEQ/BNE

Los saltos condicionales suman el offset, contenido en la constante de la instrucción, con el valor $PC + 4$ para calcular la dirección de la siguiente instrucción a ejecutar. Por supuesto, el offset se obtiene realizando la extensión con signo de la constante y multiplicando por un factor de 4 para expresarlo en bytes. El hardware para realizar este cálculo se incluye dentro de la abstracción del módulo PC, el cual recibe como entrada, la constante de 16 bits $id[15:0]$, y muestra en la salida $PC\_OFFSET[31:0]$, el cálculo de la dirección de destino.

Tabla LXII. **Señales de control para instrucciones BEQ/BNE**

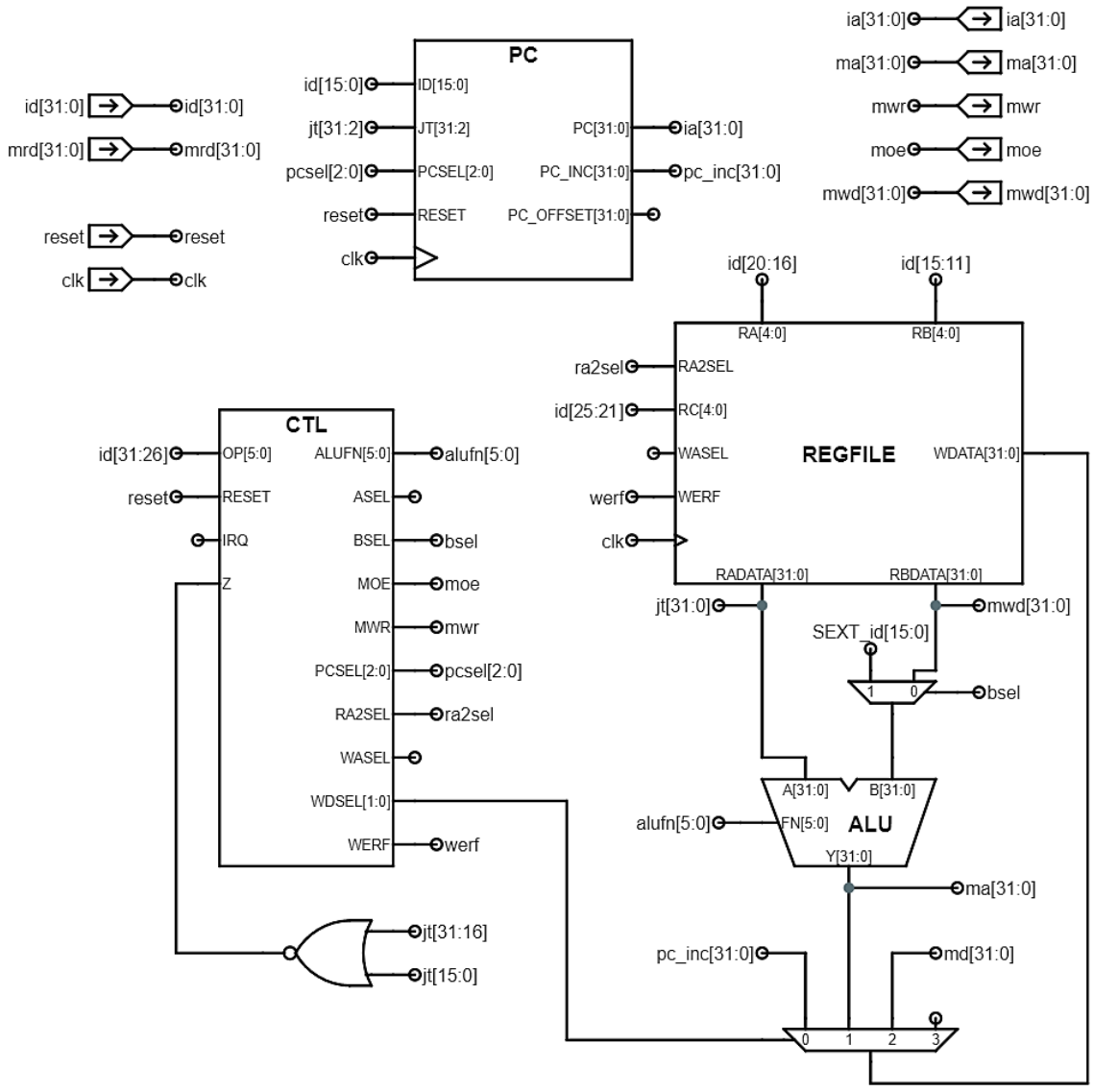
Señal	Valor
$bsel$	X
$alufn[5:0]$	X
$werf$	1
$wdsel[1:0]$	$0b00$
mwr	0
moe	X
$ra2sel$	X
$pcsel[2:0]$	$f(Z)$

Fuente: elaboración propia.

Además, se necesita agregar hardware para determinar si se debe o no realizar el salto, es decir, hay que verificar si el contenido del registro Ra es equivalente a cero. Para cumplir con dicho requerimiento, únicamente se necesita construir una compuerta NOR de 32 entradas a partir de otras compuertas NOR y $NAND$ de menor cantidad de entradas, utilizando los teoremas de De Morgan. En la figura 85 se muestra el hardware para ejecutar los saltos

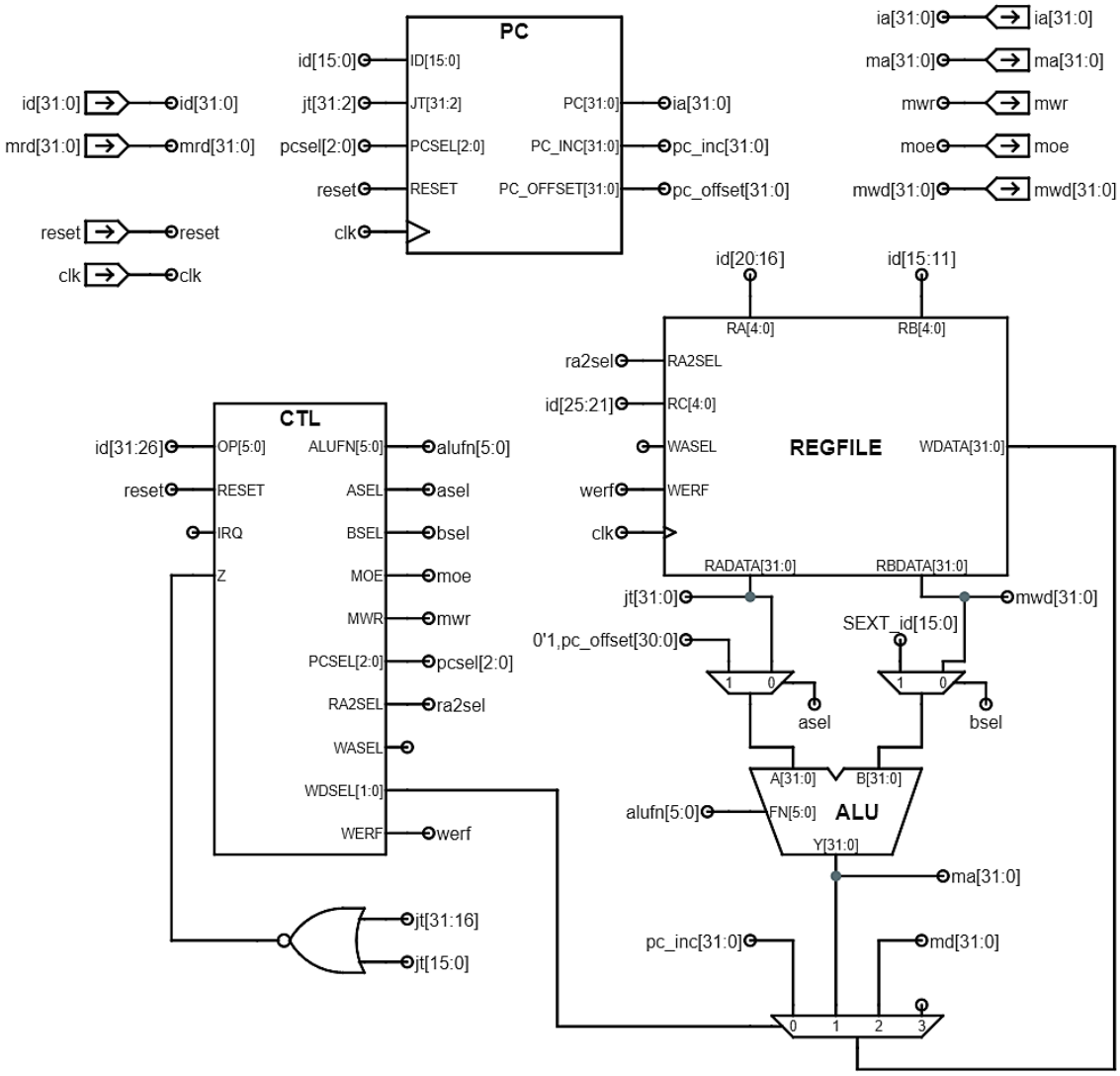
condicionales, nótese que, por motivos de optimización de espacio en la figura, la compuerta NOR de 32 entradas se representa con una compuerta NOR de 2 entradas de 16 bits, $jt[31:16]$ y $jt[15:0]$ respectivamente.

Figura 85. **Hardware para instrucciones BEQ/BNE**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Figura 86. **Hardware para la instrucción LDR**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

A pesar de que, en el capítulo 4, la compuerta NOR de 32 bits se implementa de esta manera, en VHDL se puede implementar directamente como un conjunto de 31 operaciones NOR. En este caso, el sintetizador se encargaría de optimizar el hardware empleado, sin afectar el retardo de propagación del datapath.

En la tabla LXII se muestran los valores de las señales de control que debe generar el módulo CTL. En esta ocasión, la señal $pcsel[2:0]$ es una función de la entrada Z. Para el caso de la instrucción BEQ, si $Z = 1$, entonces $pcsel[2:0] = 0b010$, de lo contrario $pcsel[2:0] = 0b000$; la misma situación aplica para el caso de la instrucción BNE, con la única diferencia de que el salto se toma si $Z = 0$.

3.2.4. Instrucción de carga relativa, LDR

La instrucción de carga relativa o LDR se comporta de manera similar a la instrucción LD, con la única diferencia de que la dirección de memoria se toma del offset calculado para las instrucciones de salto condicional, es decir, el valor $(PC + 4) + 4 * SEXT(constante)$.

Tabla LXIII. **Señales de control para instrucción LDR**

Señal	Valor
$bsel$	X
$alufn[5:0]$	A
$werf$	1
$wdsel[1:0]$	$0b10$
mwr	0
moe	1
$ra2sel$	X
$pcsel[2:0]$	$0b000$

Fuente: elaboración propia.

Es necesario agregar un multiplexor que, a través de la señal de control $asel$, permita seleccionar entre el offset contenido dentro de la señal

$pc\_offset[31:0]$ y los contenidos del registro Ra. Este multiplexor se muestra en la figura 86, obsérvese que el bit de supervisor, $pc\_offset[31]$, se ignora y se sustituye por el valor 0, formando así la señal compuesta $0'1, pc\_offset[30:0]$ que se conecta a una de las líneas de datos.

En la tabla LXIII se muestran los valores de las señales de control para la ejecución de la instrucción LDR, nótese que el valor del segundo operando es irrelevante, la ALU realiza la operación lógica A, es decir, copia el contenido del operando $A[31:0]$ y lo muestra en su salida $Y[31:0]$. Es por este motivo que las señales de control asociadas con el segundo operando poseen el valor X.

3.2.5. Excepciones

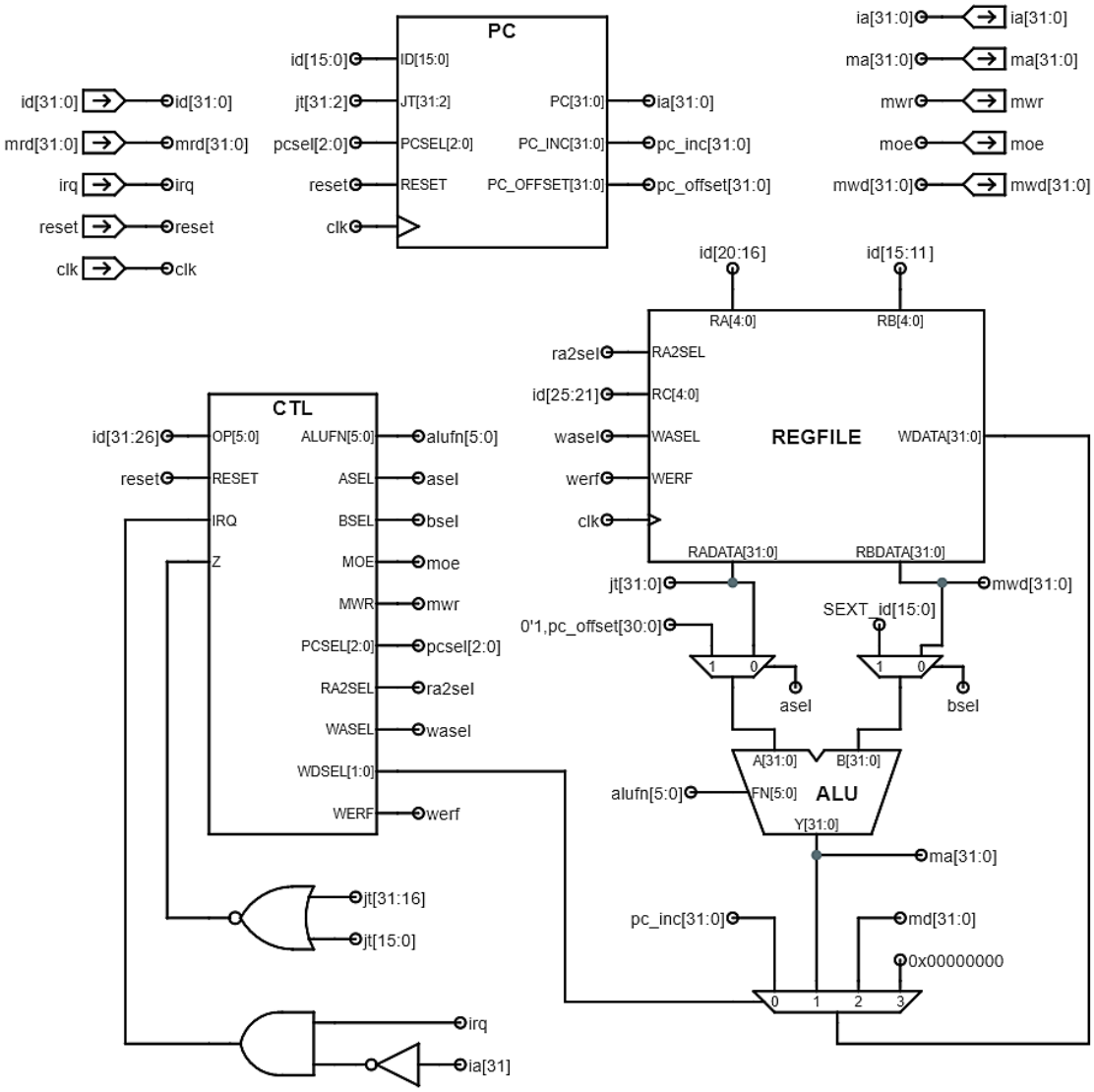
En muchas arquitecturas modernas, cuando ocurren excepciones, ya sean síncronas o asíncronas, se suspende la ejecución del programa, se redirecciona hacia una localidad de memoria que contiene código para manejar dichos eventos y luego, existe la opción de regresar a su ejecución normal.

Para incluir esta característica, se debe añadir hardware capaz de guardar el valor del contador de programa incrementado $PC + 4$ del programa interrumpido de tal forma que el código encargado del manejo de la excepción pueda resumir o no la ejecución según sea el caso. Esta característica permite transferir el control de la ejecución al software, permitiendo manejar cualquier circunstancia que quede fuera del alcance del hardware del presente diseño.

La implementación para ambos tipos de excepciones es la misma. Cuando se detecte una excepción, el hardware toma un salto hacia la dirección $0x00000004$, para excepciones síncronas, o la dirección $0x00000008$, para interrupciones. Las instrucciones en esas localidades son las que producen los

saltos a los segmentos de código o subrutinas encargadas del manejo de estos eventos.

Figura 87. **Hardware para excepciones**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

El valor $PC + 4$ se almacena en el registro $R30$, al que se le conoce como registro XP o puntero de excepciones, este registro pertenece al conjunto de registros reservados expuestos en la subsección 2.6.6. En otras palabras, el registro XP contiene la dirección de memoria a la que el código debe saltar para reanudar la ejecución normal del programa interrumpido utilizando la instrucción JMP.

Supóngase el caso de un opcode ilegal, es posible emular la instrucción faltante en software como si la instrucción se hubiera implementado en hardware, lo cual permite añadir instrucciones al ISA sin la necesidad de modificar el hardware, obviamente esta alternativa viene acompañada de un incremento en el número de ciclos de reloj necesarios para ejecutar dichas instrucciones.

En la figura 87 se muestran las señales adicionales para la correcta operación de la funcionalidad descrita anteriormente, la señal de control $wasel$ permite escoger entre el registro de destino Rc contenido dentro de la instrucción, o el registro XP. Los otros dos valores restantes de $pcsel[2:0]$ son los valores $0b011$ y $0b100$, los cuales corresponden con las líneas de datos que contienen las direcciones $0x00000004$ y $0x00000008$ respectivamente. El hardware solamente debe responder a solicitudes de interrupción cuando este se encuentra en modo usuario (cuando $ia[31] = 0$), por lo que en la parte inferior de la figura se muestra una compuerta AND, con sus entradas conectadas a las señales irq y $\overline{ia[31]}$.

En la tabla LXIV se muestran los valores de las señales de control durante el manejo de las excepciones e interrupciones. El valor de $pcsel[2:0]$ puede ser $0b011$ en caso de que se identifique una operación ilegal, o $0b100$ si se trata de una interrupción. La señal $wdsel$ es igual a 1, indicando que el valor $PC + 4$ se almacena en el registro XP, las demás señales de control son irrelevantes debido

a que ya no se completa la instrucción que se había obtenido de la memoria al principio del ciclo de reloj.

Nótese que, en el caso de una interrupción, la instrucción interrumpida no se ha terminado de ejecutar, por lo que, si el código que maneja la interrupción decide reanudar la ejecución del programa, debe realizar un salto JMP a la dirección denotada por el valor $XP - 4$.

Tabla LXIV. **Señales de control para manejo de excepciones**

Señal	Valor
$bsel$	X
$alufn[5:0]$	X
$werf$	1
$wdsel[1:0]$	$0b00$
mwr	0
moe	X
$ra2sel$	X
$pcsel[2:0]$	$0b011, 0b100$
$wasel$	1

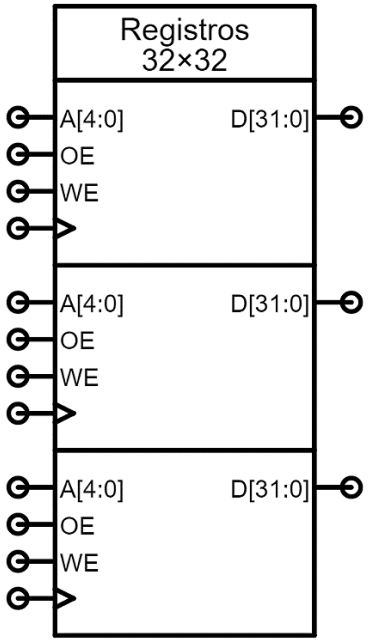
Fuente: elaboración propia.

Finalmente, el diagrama de la figura 87 es capaz de implementar toda la funcionalidad contenida dentro del ISA Beta. Una vez descrita la estructura general del datapath, el siguiente paso consiste en diseñar la circuitería interna de cada uno de los módulos utilizados en esta sección. En las siguientes secciones, se diseña el circuito de los bloques REGFILE, ALU, CTL y PC, para completar el diseño del procesador.

3.3. Bloque de registros de múltiples puertos, REGFILE

El bloque de registros de múltiples puertos de la figura 70 se compone principalmente de 32 registros de 32 bits contenidos dentro de una unidad pequeña de memoria de 3 puertos, 2 puertos de lectura y 1 de escritura, que recibe el nombre de memoria scratchpad o una memoria de borrado. Este dispositivo permite seleccionar un registro arbitrario de un puerto específico por medio de una dirección de memoria de 5 bits.

Figura 88. **Memoria scratchpad de 3 puertos**



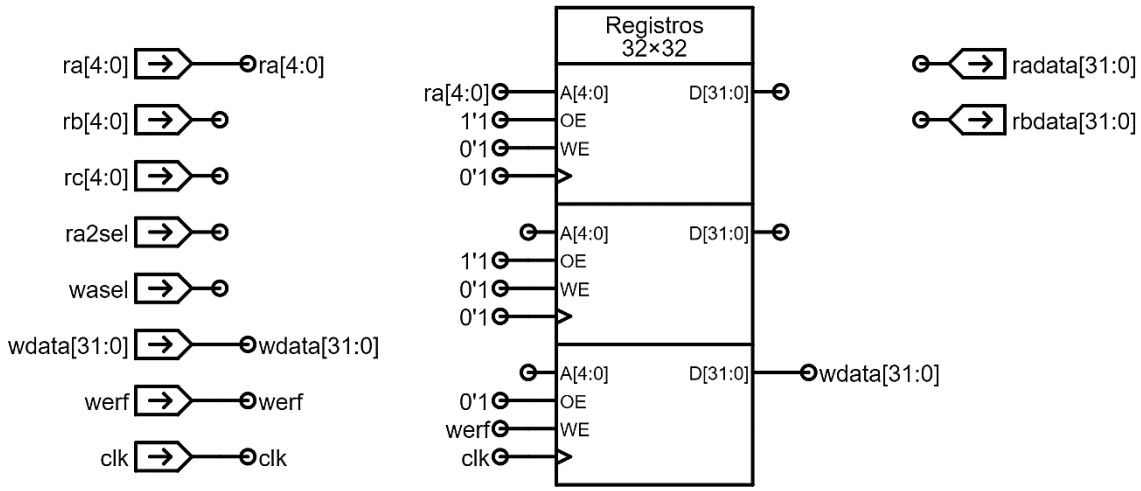
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Esta memoria se distingue de la memoria principal, la cual almacena tanto instrucciones como datos. Una memoria scratchpad es meramente una alternativa para conectar un número de registros por medio de un camino de

transferencia común y su información debe venir normalmente de la memoria principal por medio de las instrucciones del programa.

La memoria scratchpad de 3 puertos de tamaño 32×32 bits, con 32 registros de 32 bits, se muestra en la figura 88. Cada uno de los puertos posee una entrada de dirección $A[4:0]$ que permite seleccionar uno de los 32 registros internos, una entrada OE para habilitar la lectura del puerto, una entrada WE para habilitar la escritura, la entrada clk para la señal de reloj y la terminal de datos $D[31:0]$ que puede funcionar ya sea como entrada o salida, dependiendo de la configuración de las señales de control del puerto.

Figura 89. **Configuración de puertos de la memoria scratchpad**



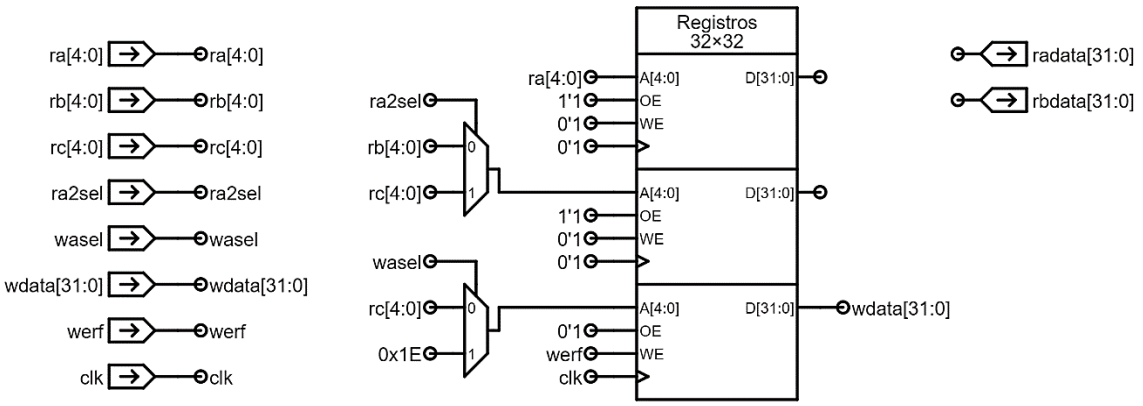
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Dos de los puertos se utilizan para leer contenidos de los registros en la memoria. Los puertos de lectura se configuran fijando las entradas OE a 1, conectándolas con una señal de valor $1'1$, y fijando las entradas WE y clk a 0, conectándolas a la señal $0'1$. El tercer puerto se utiliza para escribir valores en la

memoria, este se configura fijando la entrada OE a 0 y conectando las entradas WE y clk con sus señales respectivas.

Las conexiones de los puertos se muestran en la figura 89, la dirección $ra[4:0]$ se conecta al primer puerto de lectura, la señal $werf$ se conecta con la entrada WE del puerto de escritura y la señal $wdata[31:0]$ se conecta con la entrada $D[31:0]$.

Figura 90. **Conexión de multiplexores de control**



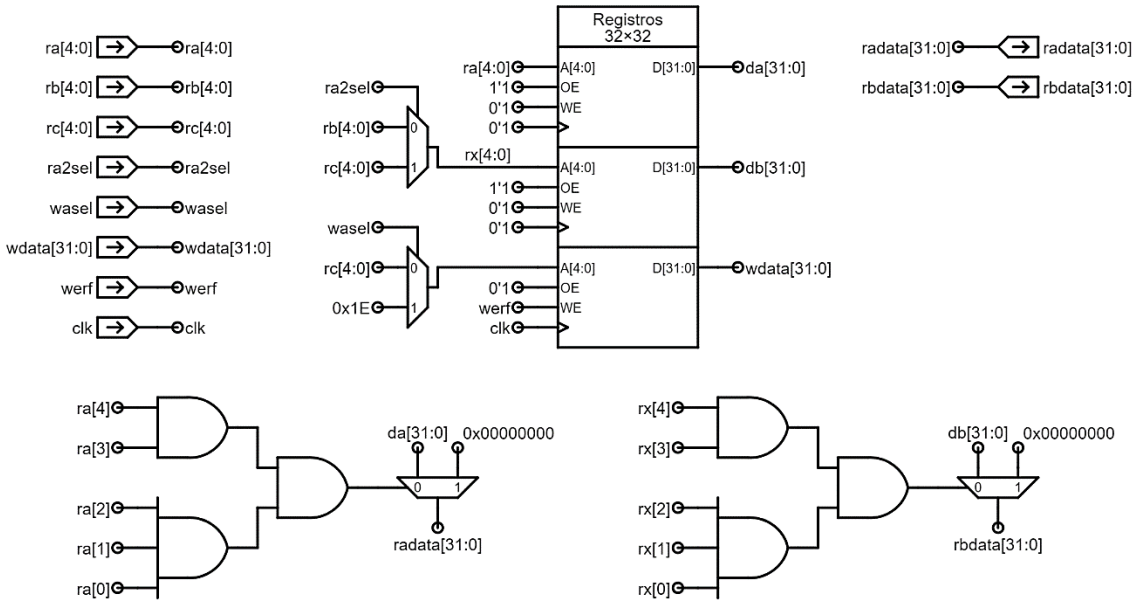
Fuente: elaboración propia, empleando Jade Circuit Simulator.

Las señales $rb[4:0]$ y $rc[4:0]$ se conectan a las líneas de datos de los multiplexores controlados por las señales $ra2sel$ y $wasel$ tal como se muestra en la figura 90. La constante $0x1E$ de 5 bits es la dirección del registro $R30$ o XP. Finalmente, las salidas de ambos multiplexores se conectan a las entradas $A[4:0]$ del segundo puerto de lectura y el puerto de escritura.

En este diseño se permite la escritura en el registro $R31$, por lo que se debe agregar hardware adicional para garantizar que el contenido que se lea de esta

localidad sea igual a cero. En el diagrama de la figura 91, los bits de las señales $ra[4:0]$ y $rx[4:0]$ se conectan a dos compuertas AND de 5 entradas para determinar si sus direcciones son equivalentes a $0b11111$, la salida de estas compuertas se conecta a las entradas de control de dos multiplexores que eligen entre la constante $0x00000000$ y los contenidos de las señales $da[31:0]$ y $db[31:0]$. Por último, las salidas de los multiplexores se conectan a las salidas $radata[31:0]$ y $rbdata[31:0]$.

Figura 91. **Circuito interno del bloque de registros**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

3.4. Unidad aritmética lógica, ALU

La abstracción de la ALU que se muestra en la figura 71 consta de dos entradas de 32 bits ($A[31:0]$ y $B[31:0]$) y una salida de 32 bits ($Y[31:0]$). Para comenzar, se presenta el diseño general de la ALU y se realiza una descripción

a nivel de abstracción de cada uno de los módulos que lo componen. Por último, se diseña cada uno de estos módulos tratándolos como circuitos separados para completar el diseño.

Tabla LXV. **Codificación de operaciones mediante la señal $\boldsymbol{FN[5:0]}$**

$\boldsymbol{FN[5:0]}$	**Operación**	$\boldsymbol{Y[31:0]}$
$0b00X011$	CMPEQ	$Y[31:0] = (A[31:0] == B[31:0])$
$0b00X101$	CMPLT	$Y[31:0] = (A[31:0] < B[31:0])$
$0b00X111$	CMPLE	$Y[31:0] = (A[31:0] \leq B[31:0])$
$0b01XXX0$	ADD	$Y[31:0] = A[31:0] + B[31:0]$
$0b01XXX1$	SUB	$Y[31:0] = A[31:0] - B[31:0]$
$0b10abcd$	Booleana, bit a bit	$Y[i] = F_{abcd}(A[i], B[i])$
$0b11XX00$	SHL	$Y[31:0] = A[31:0] << B[31:0]$
$0b11XX01$	SHR	$Y[31:0] = A[31:0] >> B[31:0]$
$0b11XX11$	SRA	$Y[31:0] = A[31:0] >> B[31:0]$ extensión con signo

Fuente: elaboración propia.

La ALU es un circuito combinacional que toma las dos entradas $A[31:0]$ y $B[31:0]$ y produce una salida $Y[31:0]$ como resultado de una operación aritmética o lógica sobre los operandos A y B. La función particular a ejecutar depende de la entrada de control de 6 bits, $FN[5:0]$, cuyo valor contiene codificada a la función de acuerdo con el esquema de la tabla LXV. Nótese que los valores de algunos bits son irrelevantes para codificar las operaciones.

Las operaciones booleanas bit a bit se especifican a través de $FN[5:4] = 0b10$; los bits restantes de la señal de control, $FN[3:0] = 0babcd$, se toman como las entradas de una tabla de verdad que describe cómo cada bit $Y[i]$ se calcula a partir de los bits $A[i]$ y $B[i]$ correspondientes, tal como se muestra en la tabla LXVI. Nótese que, dependiendo del valor de $FN[3:0]$, se realizará una de las $2^4 = 16$ posibles operaciones booleanas entre los bits $B[i]$ y $A[i]$.

Tabla LXVI. **Tabla de verdad para operaciones booleanas**

$\boldsymbol{B[i]}$	$\boldsymbol{A[i]}$	$\boldsymbol{Y[i]}$
0	0	d
0	1	c
1	0	b
1	1	a

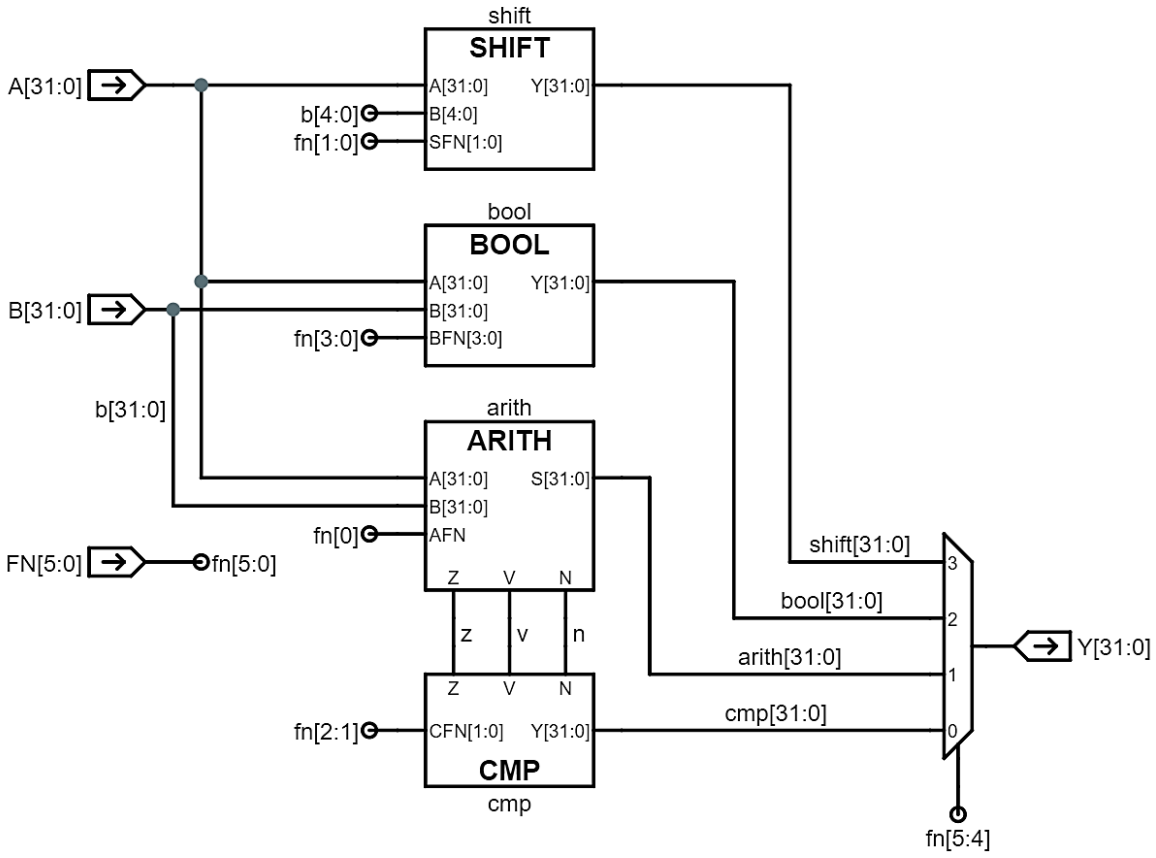
Fuente: elaboración propia.

Las tres operaciones de comparación producen una salida booleana. Esto implica que $Y[31:1] = 0$, y el bit menos significativo $Y[0]$ es equivalente a 0 o 1, lo que refleja el resultado de la comparación entre los operandos $A[31:0]$ y $B[31:0]$.

El diseño de la ALU se divide en 4 módulos principales dedicados a realizar las operaciones aritméticas, booleanas y de desplazamiento como se muestra en la figura 92. Este enfoque permite diseñar y evaluar el correcto comportamiento de cada uno de los subsistemas o módulos por aparte, facilitando así el proceso de diseño.

La unidad o el módulo de desplazamiento SHIFT recibe como entrada el operando $A[31:0]$ y realiza uno de los tres desplazamientos especificados en la tabla LXV, los cuales dependen de la señal $fn[1:0]$ conectada a la entrada $SFN[1:0]$, el número de desplazamientos a realizar se encuentra codificado en la señal $b[4:0]$ (últimos 5 bits del operando $B[31:0]$).

Figura 92. **Diseño general del circuito interno de la ALU**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

La unidad booleana o el módulo BOOL toma como entradas los dos operandos $A[31:0]$ y $B[31:0]$, y realiza una de las 16 operaciones booleanas bit a bit especificada por la señal $fn[3:0]$.

La unidad aritmética o el módulo ARITH toma como entradas los dos operandos $A[31:0]$ y $B[31:0]$, y dependiendo del valor de $fn[0]$ se realiza la suma o la resta entre los dos operandos. Las salidas Z, V y N indican si el resultado de la unidad aritmética es igual a cero, si hubo desbordamiento o si el resultado es negativo, en ese orden. Estas señales se conectan como entradas a la unidad de comparación o módulo CMP y, en función de la señal $fn[2:1]$ se realiza una de las tres operaciones de comparación de la tabla LXVI.

Finalmente, todas las salidas de los módulos principales (señales $shift[31:0]$, $bool[31:0]$, $arith[31:0]$ y $cmp[31:0]$) se conectan a las líneas de datos del multiplexor controlado por la señal $fn[5:4]$ para producir la señal de salida $Y[31:0]$.

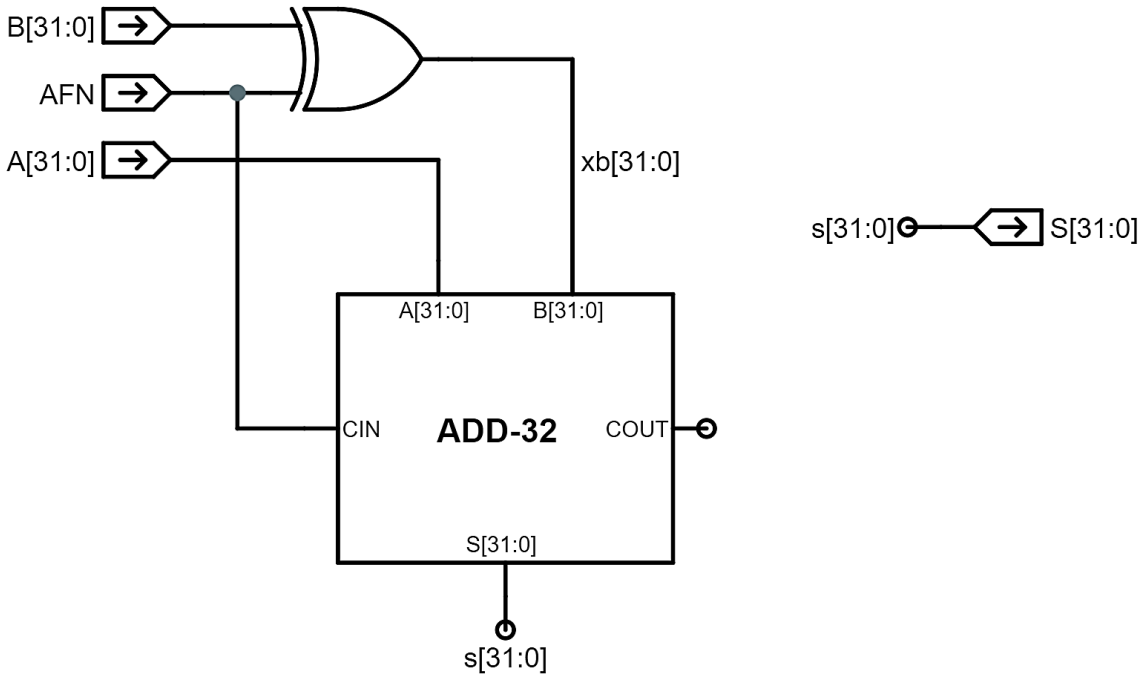
Se aclara al lector que, la señal $FN[5:0]$ utiliza un esquema de codificación que permite mantener la circuitería de la ALU relativamente simple. Nótese que este esquema no es el mismo que se utiliza para codificar el campo del opcode en las instrucciones del ISA. La unidad de control, CTL, se encarga de traducir el opcode de las instrucciones en la señal $FN[5:0]$ correspondiente.

3.4.1. Unidad aritmética

La unidad aritmética consta principalmente de un sumador cuyas entradas son dos números de 32 bits codificados en complemento a 2 y una salida de 32 bits, también representada en complemento a 2. En la figura 93 se puede observar que dicho sumador posee dos entradas de 32 bits, $A[31:0]$ y $B[31:0]$, una entrada para el bit de acarreo, CIN, una salida de 32 bits, $S[31:0]$, que contiene el resultado de la suma y una salida para mostrar el resultado del bit de acarreo de salida, $COUT$.

La señal AFN selecciona el tipo de operación que se realiza; si $AFN = 0$, se realiza la operación ADD, $S[31:0] = A[31:0] + B[31:0]$, y si $AFN = 1$ se realiza la operación SUB, $S[31:0] = A[31:0] - B[31:0]$. Para realizar la operación SUB, el circuito calcula el complemento a dos del operando $B[31:0]$, invirtiendo todos sus bits y sumando $0x00000001$ al resultado.

Figura 93. **Conexión del sumador de 32 bits**

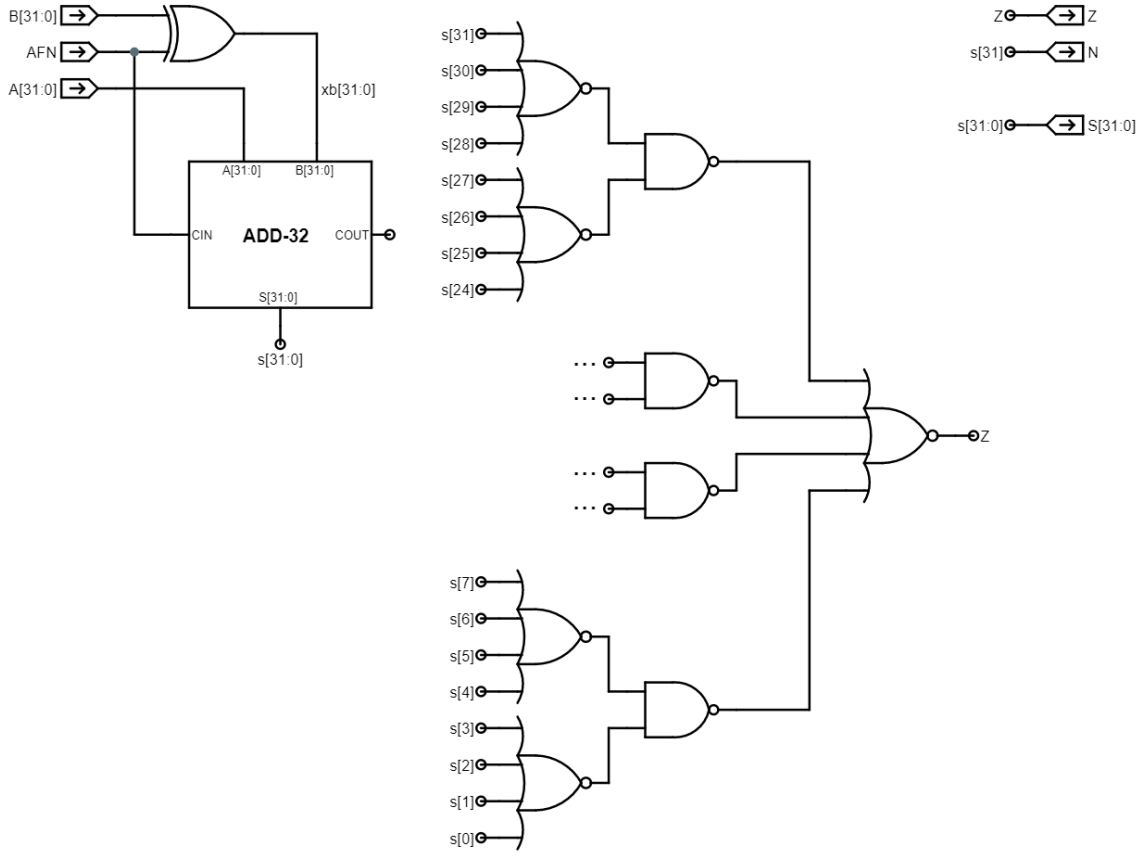


Fuente: elaboración propia, empleando Jade Circuit Simulator.

La inversión de todos los bits se lleva a cabo con una compuerta XOR cuyas entradas son la señal AFN y el bit $B[i]$, en este caso, la compuerta del circuito representa todas las 32 compuertas de dos entradas necesarias para realizar esta operación, dando como resultado la señal $xb[31:0]$ que se conecta como entrada al sumador. Sin embargo, la señal $xb[31:0]$ no representa el

complemento a dos de $B[31:0]$, es necesario sumar 1 a la señal $xb[31:0]$, lo cual se consigue forzando el acarreo de entrada del sumador al valor de 1.

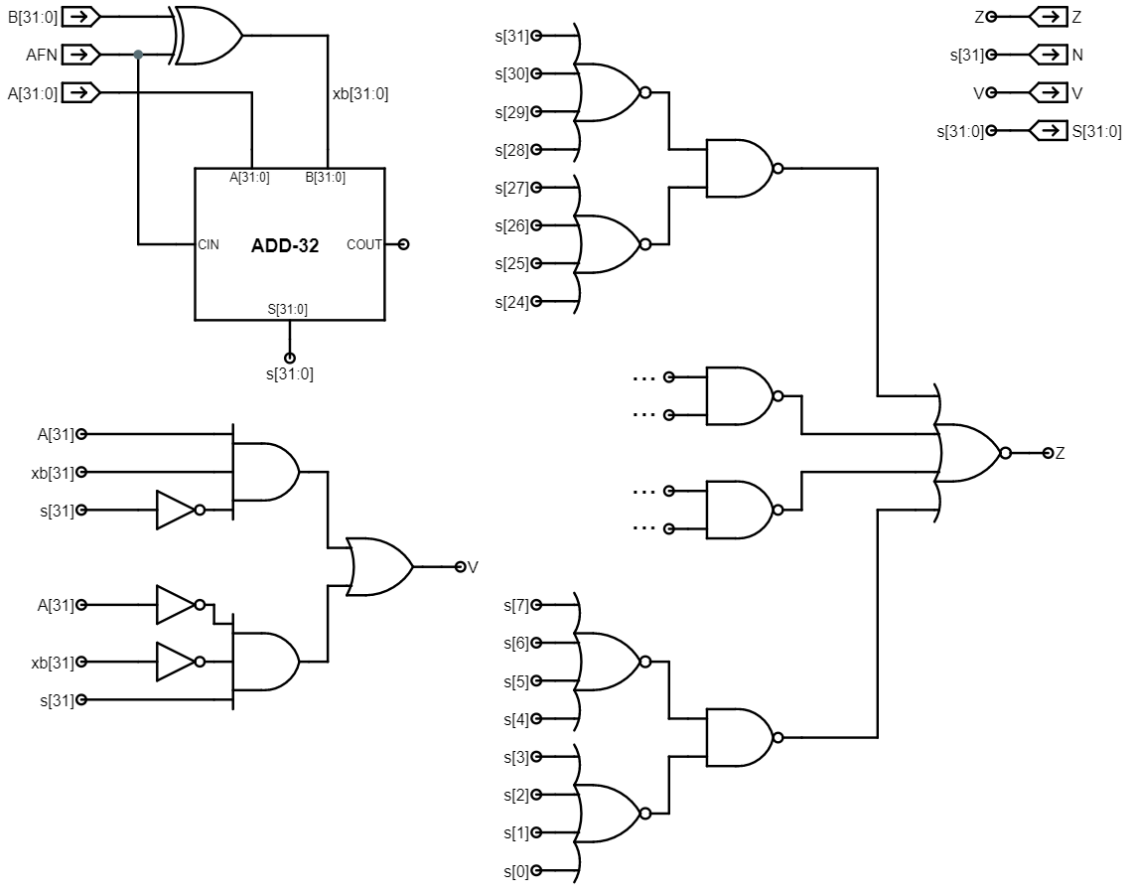
Figura 94. **Compuerta NOR de 32 bits para la unidad aritmética**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Además del sumador de 32 bits, la unidad aritmética debe contener hardware para determinar si el resultado de la operación, contenido dentro de la señal $s[31:0]$, es cero, es negativo o si hubo sobreflujo, situación en la que los 32 bits no son suficientes para representar el resultado, a través de las señales N, Z y V respectivamente.

Figura 95. **Circuito interno de la unidad aritmética**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para determinar si la señal $s[31:0]$ es equivalente a cero, basta con realizar una operación NOR entre sus 32 bits, el resultado de la operación es igual a 1 si y sólo si el valor de todos los bits en la entrada es igual a cero. Esta compuerta se muestra en la figura 94, nótese que, debido a que la operación NOR no es asociativa, se utiliza un árbol en el que se alternan etapas de compuertas $NAND$ y NOR, con base en los teoremas de De Morgan, su resultado se conecta a la señal Z.

Para determinar si el resultado de la operación es negativo, únicamente se conecta el bit más significativo de la señal $s[31:0]$ es decir, $s[31]$, a la salida N. Esto se hace debido a que, en el bit más significativo de un número arbitrario, representado en complemento a 2, se encuentra codificado el signo del mismo por lo que, no se necesita de hardware adicional para obtener esta funcionalidad.

Por último y seguramente, el caso menos intuitivo es el del diseño del hardware para determinar si existe sobreflujo, esta condición se da cuando la cantidad de bits es insuficiente para representar el resultado. Específicamente, existe sobreflujo cuando los dos operandos poseen el mismo signo y el signo del resultado difiere del signo de los primeros. Con base en el razonamiento anterior, no se puede dar un caso de sobreflujo entre dos operandos de diferente signo porque el resultado siempre queda dentro del rango de números que el esquema es capaz de codificar.

Sean $A[31]$, $xb[31]$ y $s[31]$ los bits que indican el signo de los dos operandos y el resultado de la operación respectivamente. La función booleana V permite determinar si existe sobreflujo con base en los valores de estos tres bits, dicha función se expresa de la siguiente manera:

$$V = A[31] \cdot xb[31] \cdot \overline{s[31]} + \overline{A[31]} \cdot \overline{xb[31]} \cdot s[31]$$

De la expresión anterior, se reconoce que la especificación funcional de V se encuentra en representación estándar de suma de productos, en ella, cada producto especifica una de las dos posibles condiciones en las que se presenta el sobreflujo, las cuales se indican en la tabla LXVII. Nótese que dicha tabla es sólo una pequeña parte de la tabla de verdad completa, pero, debido a que todas las demás combinaciones producen una salida $V = 0$, estas se omiten por conveniencia.

Tabla LXVII. **Condiciones para que exista sobreflujo**

$\boldsymbol{A[31]}$	$\boldsymbol{xb[31]}$	$\boldsymbol{s[31]}$	$\boldsymbol{V}$
1	1	0	1
0	0	1	1

Fuente: elaboración propia.

Por último, a partir de la especificación funcional de V, se crea el circuito combinacional correspondiente y, su salida se conecta a la señal V, tal como se muestra en la figura 95.

3.4.1.1. Sumador de 32 bits

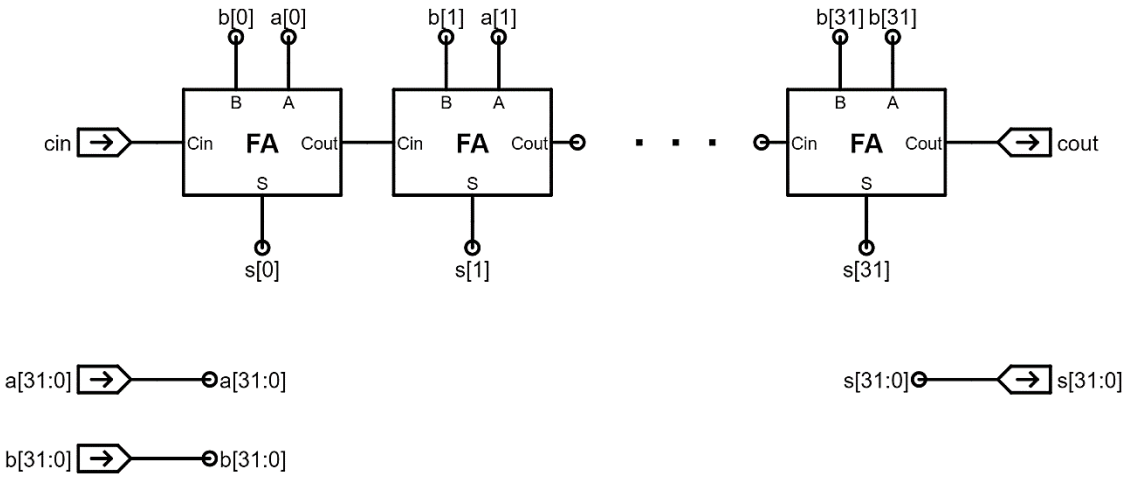
El circuito interno del sumador de 32 bits se compone de 32 sumadores completos como el de la figura 69 conectados en cascada para formar la estructura de un sumador de rizo de la figura 96. Los bits $a[i]$ y $b[i]$ se suman con el acarreo de entrada cin_i, o $cout_{i-1}$, para producir las salidas $cout_i$ y $s[i]$.

Es importante destacar que, a pesar de la simplicidad del diseño y el poco hardware requerido en el mismo, la latencia de un sumador de rizo de N bits es equivalente a $L = Nt_{FA}$ en donde, t_{FA} es el retardo de propagación de un módulo sumador, es decir, la latencia aumenta de manera lineal conforme aumenta N.

Existen otros tipos de sumadores, como los sumadores por acarreo de selección y por anticipo de acarreo, que permiten alcanzar valores de L que varían de manera logarítmica conforme incrementa el valor de N. Sin embargo, la complejidad de los mismos es significativamente mayor y la cantidad de hardware requerido aumenta, esto último puede ser o no, una limitante

dependiendo de la cantidad de recursos del PLD a utilizar. Así que, por motivos prácticos y didácticos, se utiliza un sumador de rizo en el presente diseño a pesar de que no sea la mejor alternativa en términos de latencia.

Figura 96. **Circuito interno del sumador de 32 bits**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

3.4.1.2. Sumador completo

La especificación funcional del sumador completo, FA, utilizado en el diseño del sumador de 32 bits de la figura 96 se muestra en la tabla LXVIII, en ella se muestran todas las posibles combinaciones de las entradas: A, B y Cin; y los valores de salida, S y $Cout$, correspondientes. A partir de esta tabla, se extraen los términos implicantes para construir las expresiones booleanas en representación de suma de productos para las variables S y Cin, tal como se muestra a continuación:

$$S = (\bar{A} \cdot \bar{B} \cdot Cin) + (\bar{A} \cdot B \cdot \overline{Cin}) + (A \cdot \bar{B} \cdot \overline{Cin}) + (A \cdot B \cdot Cin)$$

$$Cout = (\bar{A} \cdot B \cdot Cin) + (A \cdot \bar{B} \cdot Cin) + (A \cdot B \cdot \overline{Cin}) + (A \cdot B \cdot Cin)$$

Tabla LXVIII. **Especificación funcional del sumador completo**

$\boldsymbol{A}$	$\boldsymbol{B}$	$\boldsymbol{Cin}$	$\boldsymbol{S}$	$\boldsymbol{Cout}$
0	0	0	0	0
0	0	1	**1**	0
0	1	0	**1**	0
0	1	1	0	**1**
1	0	0	**1**	0
1	0	1	0	**1**
1	1	0	0	**1**
1	1	1	**1**	**1**

Fuente: elaboración propia.

Los circuitos combinacionales para ambas expresiones se muestran en la figura 97, conformando así el diseño del circuito interno del sumador completo. Nótese que ya no existen más abstracciones en el diseño y, por lo tanto, con este circuito combinacional, se completa el diseño del circuito de la unidad aritmética.

3.4.2. Unidad de comparación

La ALU puede realizar tres operaciones diferentes de comparación entre los operandos $A[31:0]$ y $B[31:0]$. En este diseño, se utiliza el sumador de 32 bits de la unidad aritmética para realizar la operación $A[31:0] - B[31:0]$ y luego verificar las señales Z, V y N para determinar si $A[31:0] = B[31:0]$, $A[31:0] < B[31:0]$ o $A[31:0] \leq B[31:0]$. El resultado de la unidad de comparación es un número de 32 bits que puede ser ya sea $0x00000000$, falso, o $0x00000001$, verdadero.

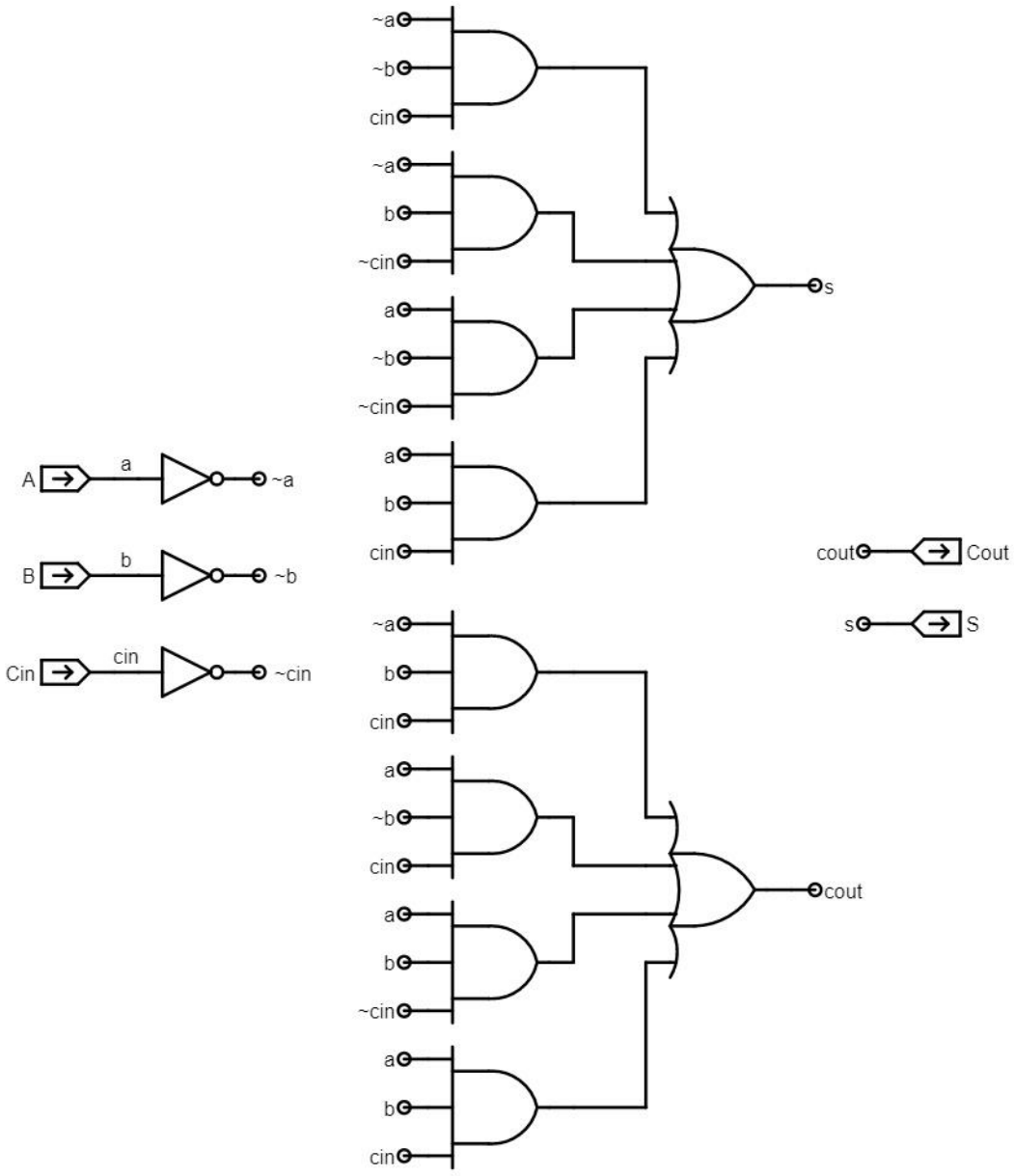
Tabla LXIX. **Codificación de operaciones de comparación**

Comparación	**Ecuación para LSB**	$\boldsymbol{CFN[1:0]}$
$A[31:0] = B[31:0]$	$LSB = Z$	$0b01$
$A[31:0] < B[31:0]$	$LSB = N \oplus V$	$0b10$
$A[31:0] \leq B[31:0]$	$LSB = Z + (N \oplus V)$	$0b11$

Fuente: elaboración propia.

La unidad de comparación de 32 bits debe ser capaz de generar una de dos constantes, 0 o 1, en función de la señal de control $CFN[1:0]$ y de las salidas Z, N y V de la unidad aritmética. Es obvio que los 31 bits más significativos del resultado siempre son iguales a cero mientras que el bit menos significativo, o LSB, codifica el resultado de la comparación. En la tabla LXIX se muestra el esquema de codificación de los tres tipos diferentes de comparaciones y, además, se indica la especificación funcional para la variable LSB.

Figura 97. **Circuito sumador completo usando suma de productos**

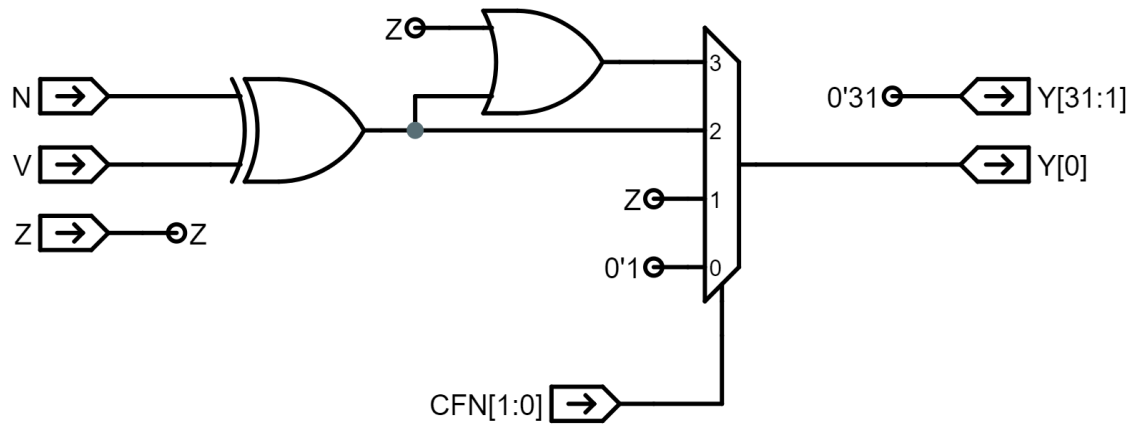


Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para identificar si $A[31:0] = B[31:0]$, se realiza la resta de ambas cantidades y si el resultado es igual a cero, entonces $LSB = Z = 1$ y la comparación es equivalente a 1.

Si $A[31:0] < B[31:0]$, entonces la operación $A[31:0] - B[31:0]$ devuelve un resultado negativo ($N = 1,\ V = 0$) o indica la existencia de sobreflujo ($N = 0,\ V = 1$), algo que, se puede determinar fácilmente mediante la función $LSB = N \oplus V$. Por último, para conocer si $A[31:0] \leq B[31:0]$, únicamente se debe determinar si se cumple una de las expresiones anteriores o ambas, es decir, la función que permite realizar esta comparación es $LSB = Z + N \oplus V$.

Figura 98. **Circuito interno de la unidad de comparación**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Las salidas de las tres funciones que permiten realizar las comparaciones se conectan a las líneas de datos de un multiplexor controlado por la señal $CFN[1:0]$ y, la salida de este se conecta con la salida $Y[0]$. Esto se ilustra en la figura 98, obsérvese que la salida $Y[31:1]$ se conecta a la constante $0'31$ (31

cables con valor 0), además, puesto que la combinación $CFN[1:0] = 0b00$ no se encuentra especificada en la tabla LXIX, se le asigna un valor de 0.

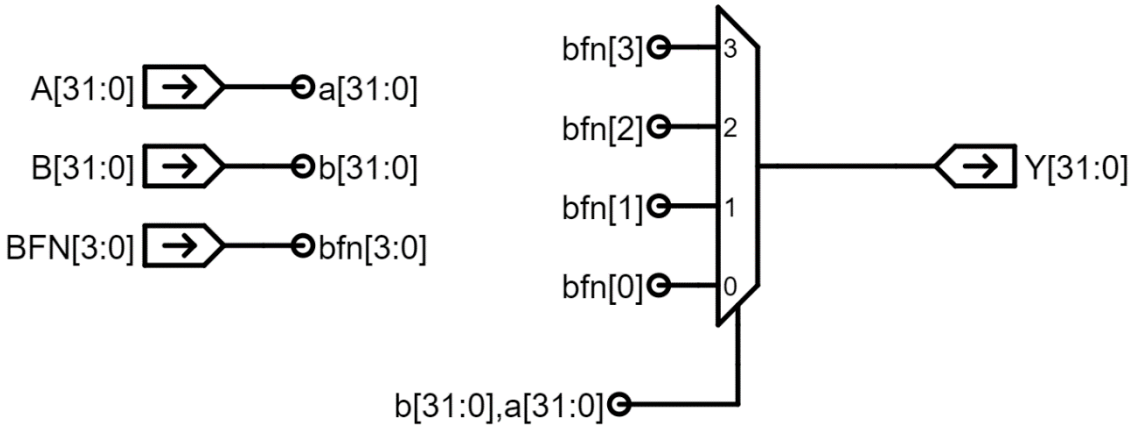
Por último, es importante mencionar que, debido a que las operaciones de comparación dependen del resultado de la unidad aritmética, el retardo de propagación de la unidad de comparación t_{CMP} no representa el intervalo de tiempo necesario para realizar la operación de comparación. Debido a esta relación de dependencia, el tiempo necesario que se debe esperar para obtener una salida válida de la unidad CMP, dada una configuración de entradas válidas, es igual $t_{total} = t_{CMP} + t_{ARITH}$, siendo t_{ARITH}, el retardo de propagación de la unidad aritmética.

3.4.3. Unidad booleana

El diseño del circuito interno de la unidad booleana consiste en la implementación de 32 multiplexores 4:2, cada uno de estos es controlado por la señal compuesta $0ba[i]b[i]$ y sus líneas de datos se conectan con cada uno de los bits de la señal $bfn[3:0] = (abcd)_2$. El funcionamiento de los multiplexores es idéntico al de las tablas de búsqueda de los bloques lógicos configurables en una FPGA (subsección 1.8.4.1), en donde la función a ejecutar sobre los bits $a[i]$ y $b[i]$ se encuentra codificada dentro de la señal $bfn[3:0]$.

En la figura 99 se muestra una representación compacta de la unidad booleana en la que se utiliza sólo un multiplexor al que se conecta la señal $b[31:0], a[31:0]$ como entrada de control para indicar, de manera implícita, la presencia de los 32 multiplexores para cada par de bits $a[i]$ y $b[i]$ en el diseño. La salida $Y[31:0]$ debe entenderse como la unión de todos los resultados $Y[i]$ de los 32 multiplexores.

Figura 99. **Circuito interno de la unidad booleana**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

3.4.4. Unidad de desplazamiento

La unidad de desplazamiento necesita hardware para implementar desplazamientos lógicos a la izquierda, SHL, desplazamientos lógicos a la derecha, SHR, y desplazamientos aritméticos a la derecha, SRA. El operando $A[31:0]$ representa la cantidad a ser desplazada y los 5 bits menos significativos del operando $B[31:0]$, es decir $B[4:0]$, indica la cantidad de posiciones que se debe desplazar $A[31:0]$. Se debe tener en cuenta que $0 \leq B[4:0] \leq 31$, debido a que sólo se pueden representar 32 valores utilizando 5 bits.

Dado que existen 3 alternativas para desplazar el operando $A[31:0]$, la operación que se desea realizar se encuentra codificada dentro de la entrada $SFN[1:0]$ siguiendo el esquema de la tabla LXX. Con este esquema de codificación, el bit $SFN[0]$ es igual a 0 para desplazamientos hacia la izquierda y es igual a 1 para desplazamientos a la derecha y, el bit $SFN[1]$ controla la extensión de signo en el desplazamiento a la derecha.

Para las operaciones SHL y SHR, las posiciones vacantes, como resultado del desplazamiento, se llenan con ceros. Para la operación SRA, las posiciones vacantes se llenan con el bit $A[31]$, el bit de signo del operando $A[31:0]$, así, el resultado es equivalente a dividir el número entero con signo dentro de una potencia de 2.

Tabla LXX. **Codificación de desplazamientos**

Operación	$\boldsymbol{SFN[1:0]}$
SHL (desplazar a la izquierda)	$0b00$
SHR (desplazar a la derecha)	$0b01$
SRA (desplazar a la derecha con extensión de signo)	$0b11$

Fuente: elaboración propia.

Realizar el diseño del hardware de la unidad de desplazamiento resulta una tarea sencilla si se considera que la cantidad de posiciones a desplazar, $B[4:0]$, se puede representar como una suma ponderada de potencias de 2. Es decir,

$$B[4:0] = \sum_{i=0}^{4} B[i] \cdot 2^i$$

Expandiendo la sumatoria anterior, se obtiene

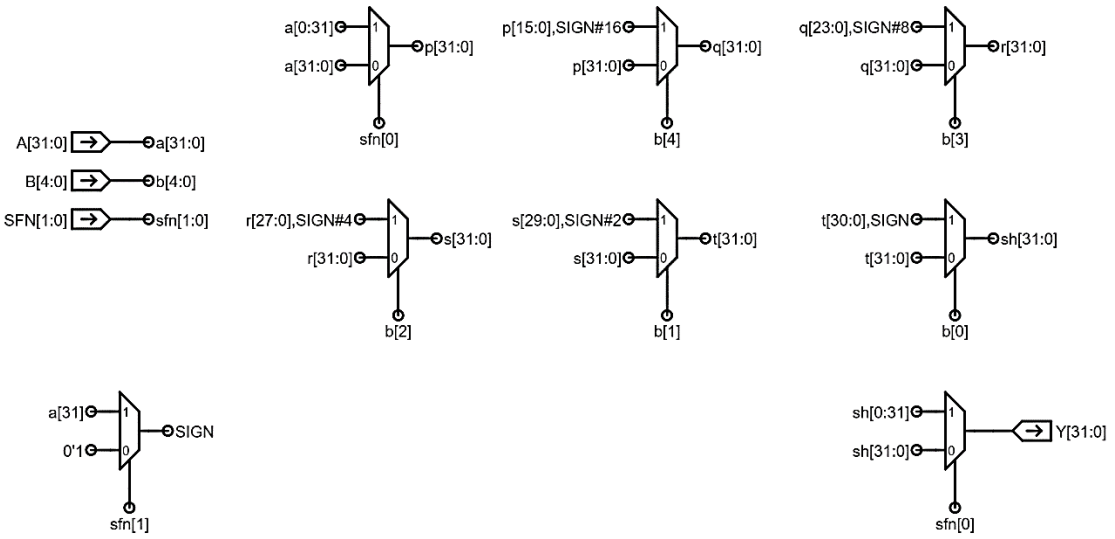
$$B[4:0] = B[4] \cdot \mathbf{2^4} + B[3] \cdot \mathbf{2^3} + B[2] \cdot \mathbf{2^2} + B[1] \cdot \mathbf{2^1} + B[0] \cdot \mathbf{2^0}$$
$$B[4:0] = B[4] \cdot \mathbf{16} + B[3] \cdot \mathbf{8} + B[2] \cdot \mathbf{4} + B[1] \cdot \mathbf{2} + B[0] \cdot \mathbf{1}$$

La expresión anterior implica que, si $B[4] = 1$, entonces ocurren 16 desplazamientos, y la dirección depende de $SFN[0]$; si $B[3] = 1$ ocurren 8 desplazamientos; si $B[2] = 1$ ocurren 4 desplazamientos y así sucesivamente. De esta manera, se representa la cantidad $B[4:0]$ como una composición de desplazamientos de potencias de 2. Por ejemplo, un desplazamiento de 15 bits, se puede implementar como un desplazamiento de 8 bits, seguido de un desplazamiento de 4 bits, seguido de un desplazamiento de 2 bits y un desplazamiento de 1 bit.

Por otra parte, la cantidad de hardware necesaria para implementar la unidad de desplazamiento se reduce considerablemente si se utiliza un único circuito, encargado de realizar desplazamientos a la izquierda. Para realizar desplazamientos a la derecha con este circuito, se revierten los bits de la entrada $A[31:0]$ para producir la señal $A[0:31]$, esto quiere decir que el bit menos significativo $A[0]$ pasa a ocupar la posición más significativa, el bit $A[1]$ la segunda posición más significativa y así sucesivamente hasta llegar al bit $A[31]$, que ocupa la posición menos significativa dentro de la cadena. Una vez realizado el desplazamiento a la izquierda sobre $A[0:31]$, se revierten nuevamente las posiciones de los bits antes de mostrar el resultado.

El circuito interno de la unidad de desplazamiento que se muestra en la figura 100 consta principalmente de 5 multiplexores 2:1 conectados en cascada, controlados por cada uno de los bits de la señal $b[4:0]$. Los multiplexores se encargan de realizar desplazamientos consecutivos hacia la izquierda. Dos multiplexores controlados por la señal $sfn[0]$, indican el tipo de operación de desplazamiento a realizar sobre el operando $a[31:0]$ y existe un multiplexor controlado por la señal $sfn[1]$ que permite seleccionar el valor con el que se llenan las vacantes en los desplazamientos (con el valor 0 o con el bit $a[31]$) y lo muestra en su señal de salida $SIGN$.

Figura 100. **Circuito interno de la unidad de desplazamiento**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Para el multiplexor ubicado en la parte superior izquierda, el valor de $sfn[0]$ elige entre el operando $a[31:0]$ y su versión revertida $a[0:31]$, y muestra el resultado en la señal $p[31:0]$. El bit $b[4]$ permite elegir entre la señal original $p[31:0]$ y la señal compuesta $p[15:0], SIGN\#16$ (la expresión $SIGN\#16$ implica que hay 16 bits con el valor de la señal $SIGN$), que representa el desplazamiento de 16 bits de $p[31:0]$ hacia la izquierda, y muestra el resultado en la señal $q[31:0]$. Posteriormente, $b[3]$ elige entre la señal $q[31:0]$ y su versión desplazada 8 unidades a la izquierda, $q[23:0], SIGN\#8$, y muestra el resultado en $r[31:0]$.

Este patrón de operación se repite en los multiplexores controlados por los bits de la señal $b[4:0]$, hasta que el resultado final de los desplazamientos se muestra en la señal $sh[31:0]$. El multiplexor de la parte inferior derecha, el cual es controlado por la señal $sfn[0]$, elige entre la señal original $sh[31:0]$ para

desplazamientos a la izquierda, o la señal revertida $sh[0:31]$ para desplazamientos a la derecha.

3.5. Unidad de control, CU

La unidad de control o CU es un circuito secuencial cuya función principal consiste en buscar las instrucciones en la memoria principal, decodificarlas y ejecutarlas. Para este caso particular y, para simplificar la tarea de diseño, la unidad de control se divide en dos subsistemas principales, la lógica de control, CTL, y el contador de programa, PC. La interconexión de estos dos subsistemas o módulos permite llevar el control de la ejecución de las instrucciones almacenadas en memoria.

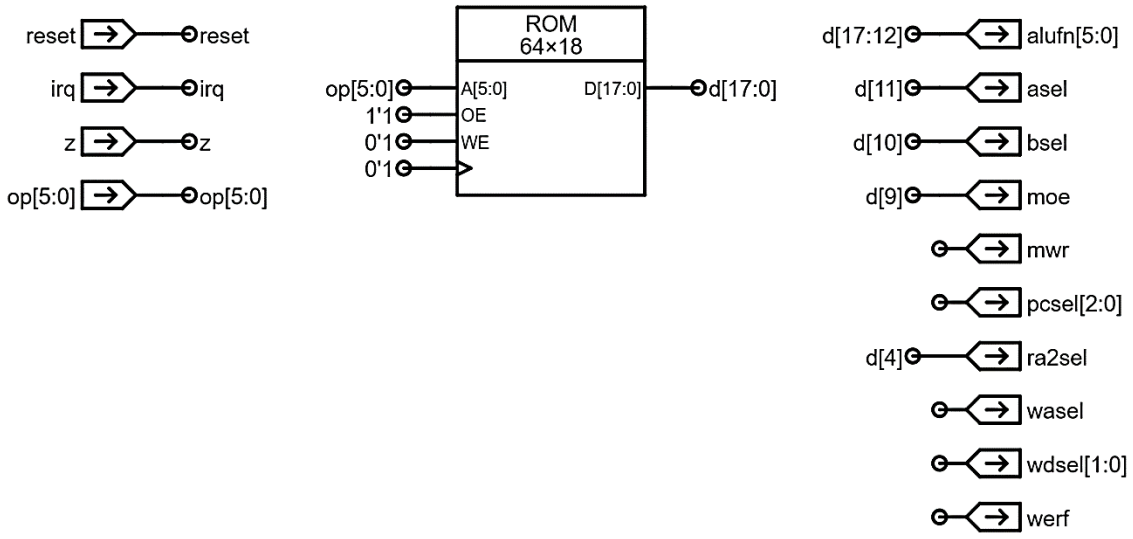
3.5.1. Lógica de control, CTL

La lógica de control o módulo CTL de la figura 72 es un circuito combinacional encargado de producir las señales de control necesarias para ejecutar las instrucciones. Para el diseño de la mayor parte del hardware de este módulo se utiliza una ROM de 64 localidades, cada una con un tamaño o ancho de 18 bits, direccionada por la señal $op[5:0]$, la cual es equivalente al campo del opcode, contenido en la instrucción a ejecutar.

En la figura 101 se muestra un bloque de memoria de un solo puerto, configurado para representar la ROM junto con las terminales de entrada y salida del módulo CTL, los valores de las señales de control de salida se encuentran codificados dentro de la señal $d[17:0]$ de acuerdo al formato especificado en la tabla LXXI. Las señales de control mwr, $pcsel[2:0]$, $wasel$, $wdsel[1:0]$ y $werf$ dependen de otras señales adicionales a la señal $d[17:0]$, por lo que se necesita

añadir hardware para modificar las señales $d[8]$, $d[7:5]$, $d[4]$, $d[3]$, $d[2:1]$ y $d[0]$ respectivamente.

Figura 101. **ROM de 64×18 bits para el módulo CTL**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Antes de añadir hardware que modifique algunos de los bits de la señal $d[17:0]$ en el circuito de la figura 101, se debe conocer el contenido detallado de la ROM de 64×18 bits. En otras palabras, se debe contar con una descripción clara y concisa de los valores de 18 bits de cada una de las 64 localidades de memoria, que corresponden con cada uno de los $2^6 = 64$ códigos de operación posibles del ISA.

Tabla LXXI. **Codificación de las señales de control del módulo CTL**

$\boldsymbol{alufn}[\mathbf{5}:\mathbf{0}]$	$d[17:12]$
$\boldsymbol{asel}$	$d[11]$
$\boldsymbol{bsel}$	$d[10]$
$\boldsymbol{moe}$	$d[9]$
$\boldsymbol{mwr}$	$f(d[8], irq, reset)$
$\boldsymbol{pcsel}[\mathbf{2}:\mathbf{0}]$	$f(d[7:5], irq, z)$
$\boldsymbol{ra2sel}$	$d[4]$
$\boldsymbol{wasel}$	$f(d[3], irq)$
$\boldsymbol{wdsel}[\mathbf{1}:\mathbf{0}]$	$f(d[2:1], irq)$
$\boldsymbol{werf}$	$f(d[0], irq)$

Fuente: elaboración propia.

3.5.1.1. Resumen de señales de control

Los valores requeridos para las señales de control, presentados en las tablas LVII a LXIV, se resumen en la tabla LXXII.

Los valores detallados de la señal $alufn[5:0]$ para cada una de las operaciones de la ALU se muestran en la tabla LXXII. Dichos valores corresponden con el esquema de codificación presentado en la tabla LXV, nótese que el valor X se sustituye, por conveniencia de representación, con el valor cero.

Tabla LXXII. **Resumen de valores de señales de control**

	$aluf n[5:0]$	$asel$	$bsel$	moe	mwr	$pcsel[2:0]$	$ra2sel$	$wasel$	$wdsel[1:0]$	$werf$
RESET	X	X	X	X	0	X	X	X	X	X
IRQ	X	X	X	X	0	4	X	X	0	1
OP	$f(op)$	0	0	X	0	0	0	0	1	1
OPC	$f(op)$	0	1	X	0	0	X	0	1	1
LD	"+"	0	1	1	0	0	X	0	2	1
LDR	"A"	1	X	1	0	0	X	0	2	1
ST	"+"	0	1	0	1	0	1	X	X	0
JMP	X	X	X	X	0	2	X	0	0	1
BEQ	X	X	X	X	0	$f(z)$	X	0	0	1
BNE	X	X	X	X	0	$f(z)$	X	0	0	1
ILLOP	X	X	X	X	0	3	X	1	0	1

Fuente: elaboración propia.

Tabla LXXIII. **Valores de la señal $\boldsymbol{alufn}[\mathbf{5}:\mathbf{0}]$**

$\boldsymbol{alufn}[\mathbf{5}:\mathbf{0}]$	**Operación**
$0b000011$	CMPEQ
$0b000101$	CMPLT
$0b000111$	CMPLE
$0b010000$	ADD
$0b010001$	SUB
$0b101000$	AND
$0b101110$	OR
$0b100110$	XOR
$0b101001$	XNOR
$0b101010$	“A”
$0b110000$	SHL
$0b110001$	SHR
$0b110011$	SRA

Fuente: elaboración propia.

3.5.1.2. Tabla de contenidos de la ROM

Los detalles de los contenidos de la ROM se muestran en la tabla LXXIV, cada una de las 64 localidades direccionadas a través del código de operación de la instrucción, contenido dentro de la señal $op[5:0]$, almacena una cadena de 18 bits que siguiendo el esquema de codificación de la tabla LXXI, en donde el bit $alufn[5]$ es el más significativo y $werf$ representa el bit menos significativo de la salida $d[17:0]$.

Para los opcodes que no se especifican en el ISA Beta, estos se toman como operaciones ilegales, o ILLOP, y se asignan los valores correspondientes a dichas localidades. Nótese que el valor X se ha sustituido, de nuevo por conveniencia, por el valor 0 y, además, se omite el prefijo $0b$ en la representación de las cadenas binarias.

Tabla LXXIV. **Contenidos de la ROM de control**

$op[5:0]$	$alufn[5:0]$	$asel$	$bsel$	moe	mwr	$pcsel[2:0]$	$rasel2$	$wasel$	$wdsel[1:0]$	$werf$	Instrucción
000000	000000	0	0	0	0	011	0	1	00	1	ILLOP
000001	000000	0	0	0	0	011	0	1	00	1	ILLOP
000010	000000	0	0	0	0	011	0	1	00	1	ILLOP
000011	000000	0	0	0	0	011	0	1	00	1	ILLOP
000100	000000	0	0	0	0	011	0	1	00	1	ILLOP
000101	000000	0	0	0	0	011	0	1	00	1	ILLOP
000110	000000	0	0	0	0	011	0	1	00	1	ILLOP
000111	000000	0	0	0	0	011	0	1	00	1	ILLOP
001000	000000	0	0	0	0	011	0	1	00	1	ILLOP
001001	000000	0	0	0	0	011	0	1	00	1	ILLOP
001010	000000	0	0	0	0	011	0	1	00	1	ILLOP
001011	000000	0	0	0	0	011	0	1	00	1	ILLOP
001100	000000	0	0	0	0	011	0	1	00	1	ILLOP
001101	000000	0	0	0	0	011	0	1	00	1	ILLOP
001110	000000	0	0	0	0	011	0	1	00	1	ILLOP
001111	000000	0	0	0	0	011	0	1	00	1	ILLOP
010000	000000	0	0	0	0	011	0	1	00	1	ILLOP
010001	000000	0	0	0	0	011	0	1	00	1	ILLOP
010010	000000	0	0	0	0	011	0	1	00	1	ILLOP

Continuación de la tabla LXXIV

010011	000000	0	0	0	0	011	0	1	00	1	ILLOP
010100	000000	0	0	0	0	011	0	1	00	1	ILLOP
010101	000000	0	0	0	0	011	0	1	00	1	ILLOP
010110	000000	0	0	0	0	011	0	1	00	1	ILLOP
010111	000000	0	0	0	0	011	0	1	00	1	ILLOP
011000	010000	0	1	1	0	000	0	0	10	1	LD
011001	010000	0	1	0	1	000	1	0	00	0	ST
011010	000000	0	0	0	0	011	0	1	00	1	ILLOP
011011	000000	0	0	0	0	010	0	0	00	1	JMP
011100	000000	0	0	0	0	000	0	0	00	1	BEQ
011101	000000	0	0	0	0	000	0	0	00	1	BNE
011110	000000	0	0	0	0	011	0	1	00	1	ILLOP
011111	101010	1	0	1	0	000	0	0	10	1	LDR
100000	100000	0	0	0	0	000	0	0	01	1	ADD
100001	100001	0	0	0	0	000	0	0	01	1	SUB
100010	000000	0	0	0	0	011	0	1	00	1	ILLOP
100011	000000	0	0	0	0	011	0	1	00	1	ILLOP
100100	000011	0	0	0	0	000	0	0	01	1	CMPEQ
100101	000101	0	0	0	0	000	0	0	01	1	CMPLT
100110	000111	0	0	0	0	000	0	0	01	1	CMPLE
100111	000000	0	0	0	0	011	0	1	00	1	ILLOP
101000	101000	0	0	0	0	000	0	0	01	1	AND
101001	101110	0	0	0	0	000	0	0	01	1	OR
101010	100110	0	0	0	0	000	0	0	01	1	XOR
101011	101001	0	0	0	0	000	0	0	01	1	XNOR
101100	110000	0	0	0	0	000	0	0	01	1	SHL
101101	110001	0	0	0	0	000	0	0	01	1	SHR
101110	110011	0	0	0	0	000	0	0	01	1	SRA
101111	000000	0	0	0	0	011	0	1	00	1	ILLOP
110000	100000	0	1	0	0	000	0	0	01	1	ADDC
110001	100001	0	1	0	0	000	0	0	01	1	SUBC

Continuación de la tabla LXXIV

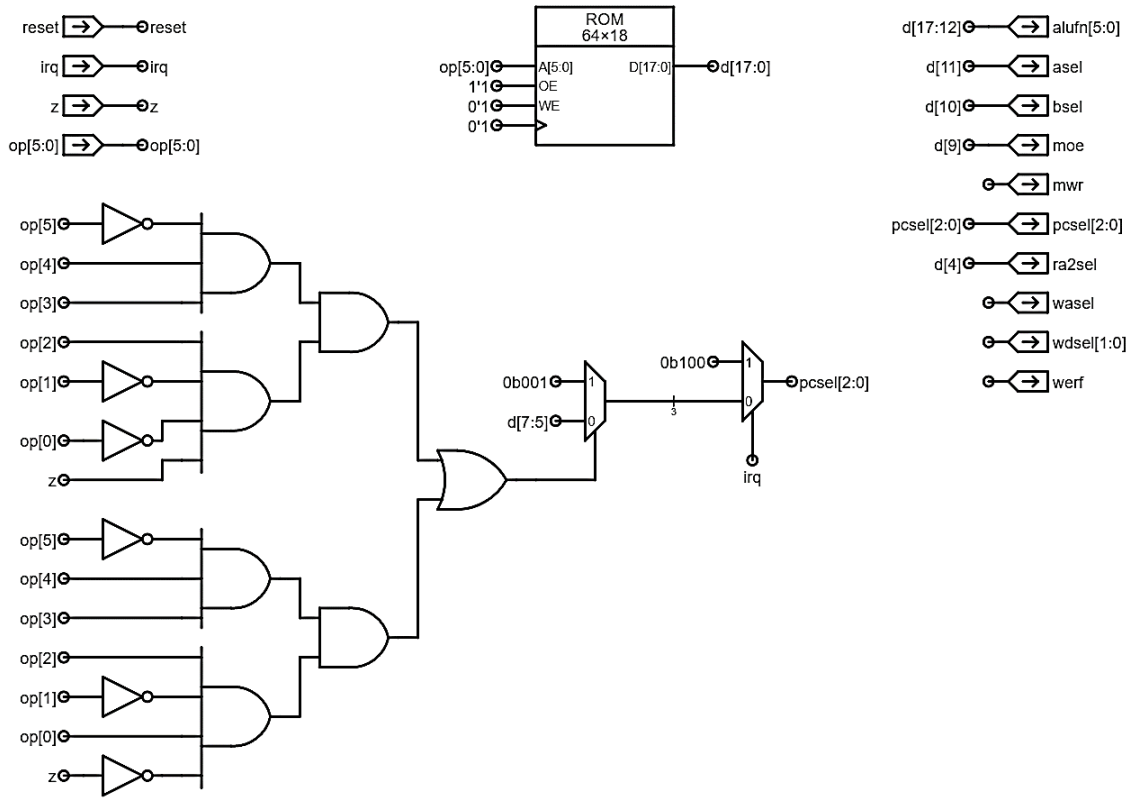
110010	000000	0	0	0	0	011	0	1	00	1	ILLOP
110011	000000	0	0	0	0	011	0	1	00	1	ILLOP
110100	000011	0	1	0	0	000	0	0	01	1	CMPEQC
110101	000101	0	1	0	0	000	0	0	01	1	CMPLTC
110110	000111	0	1	0	0	000	0	0	01	1	CMPLEC
110111	000000	0	0	0	0	011	0	1	00	1	ILLOP
111000	101000	0	1	0	0	000	0	0	01	1	ANDC
111001	101110	0	1	0	0	000	0	0	01	1	ORC
111010	100110	0	1	0	0	000	0	0	01	1	XORC
111011	101001	0	1	0	0	000	0	0	01	1	XNORC
111100	110000	0	1	0	0	000	0	0	01	1	SHLC
111101	110001	0	1	0	0	000	0	0	01	1	SHRC
111110	110011	0	1	0	0	000	0	0	01	1	SRAC
111111	000000	0	0	0	0	011	0	1	00	1	ILLOP

Fuente: elaboración propia.

3.5.1.3. Hardware adicional

Una vez especificados, de manera clara y concisa, los contenidos de la ROM, es momento de remitirse de nuevo a la tabla LXXI, en donde se especifica la codificación de las señales de control dentro de la señal de salida $d[17:0]$ del bloque de memoria. En esta tabla se indica que, las señales $d[8]$, $d[7:5]$, $d[4]$, $d[3]$, $d[2:1]$ y $d[0]$, se deben procesar con lógica adicional para conectarse a las señales de control mwr, $pcsel[2:0]$, $wasel$, $wdsel[1:0]$ y $werf$ respectivamente. Esto debido a la dependencia de estas últimas de los valores de las señales irq, $reset$ y z.

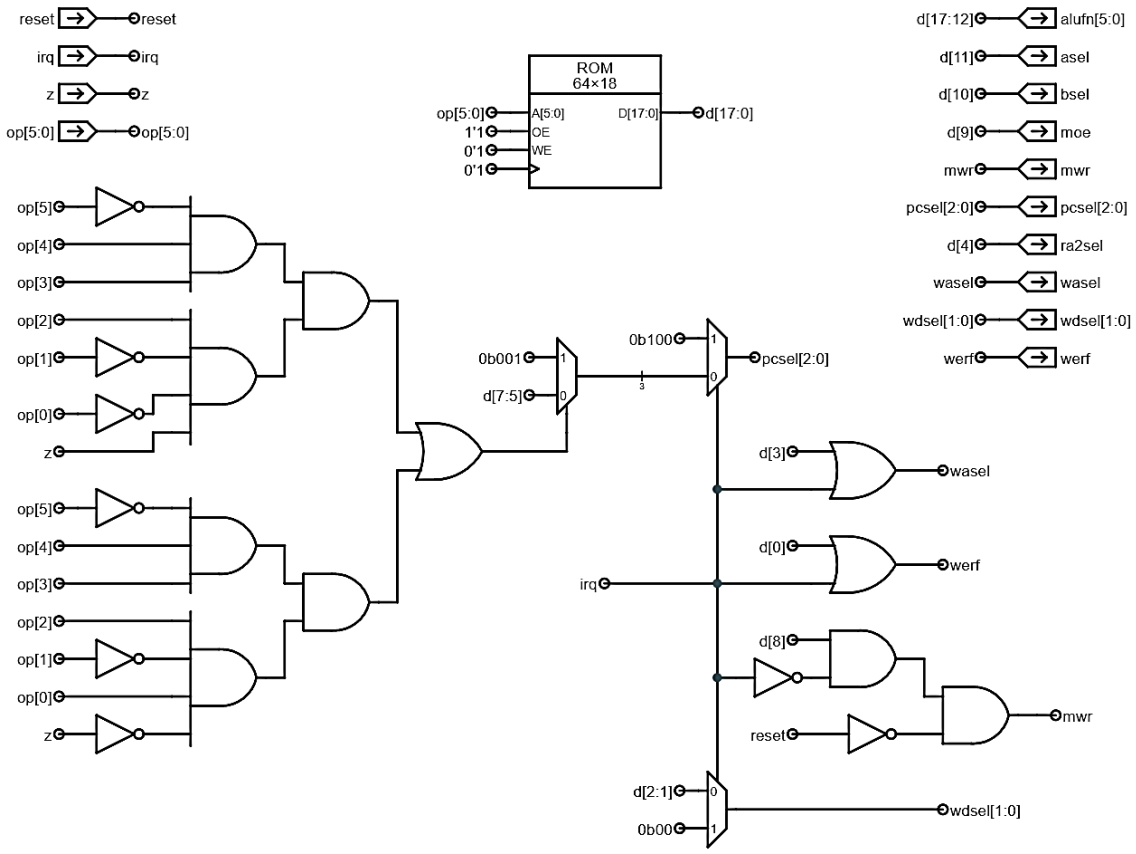
Figura 102. **Hardware adicional para la señal $pcsel[2:0]$**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

El valor de la señal $pcsel[2:0]$ depende de los valores de las señales $d[7:5]$, irq, y z. El valor de $pcsel[2:0]$ debe ser $0b001$ cuando en la presente instrucción se debe tomar un salto condicional, en otras palabras, cuando $op[5:0] = 0b011100$ (instrucción BEQ) y $z = 1$, o cuando $op[5:0] = 0b011101$ (instrucción BNE) y $z = 0$; si no se lleva a cabo el salto condicional, el valor de $pcsel[2:0]$ debe ser igual a $0b000$. En la figura 102 se muestra un circuito combinacional en forma de suma de productos capaz de determinar si se debe o no tomar el salto condicional, la salida de este circuito se conecta a la señal de control del multiplexor de la izquierda para elegir entre las señales de 3 bits $0b001$ y $d[7:5]$.

Figura 103. **Circuito interno de la lógica de control**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

Si la señal de interrupción irq es igual a 1, el valor de $pcsel[2:0]$ debe ser igual a $0b100$, de lo contrario, este debe ser igual al valor de salida del multiplexor de la izquierda. En la figura 102, La señal irq se conecta a la entrada de control del multiplexor de la derecha, esto permite seleccionar entre la salida del primer multiplexor y la constante $0b100$ y mostrar su resultado en la señal $pcsel[2:0]$.

Los valores de las señales $werf$ y $wasel$ dependen no solo de los valores $d[0]$ y $d[3]$ (en ese orden) sino también del valor de la señal irq. Si $irq = 1$, en

otras palabras, si el procesador debe responder a una interrupción, entonces las señales $wasel$ y $werf$ deben ser igual a 1. En la figura 103 se muestra el hardware necesario para cumplir con dicha funcionalidad, se utilizan dos compuertas OR, cada una de las cuales posee una entrada conectada a la señal irq y las entradas restantes, se conectan a las señales $d[0]$ o $d[3]$ para producir las salidas $werf$ y $wasel$, respectivamente.

Además de las señales anteriores, si $irq = 1$, la señal $wdsel[1:0]$ debe ser equivalente a $0b00$, de lo contrario su valor debe ser igual al contenido de la señal $d[2:1]$, el multiplexor de dos líneas de datos, ubicado en la parte inferior de la figura 103, se encarga de elegir entre la constante $0b00$ y la señal $d[2:1]$ mediante la señal irq que se conecta a la señal de control del dispositivo, el resultado se conecta a la señal $wdsel[1:0]$.

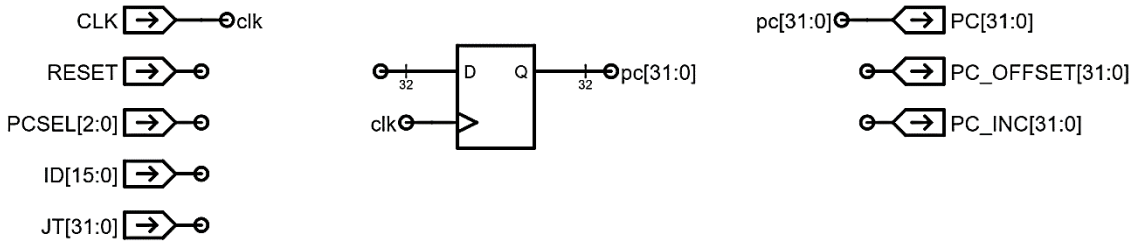
La señal mwr debe ser igual a 0 cuando $irq = 1$, este requerimiento se implementa a través de una compuerta AND de dos entradas, a las que se conectan las señales $d[8]$ e $\overline{irq}$, como se muestra en la figura 103. Sin embargo, cuando $reset = 1$, mwr se debe forzar a 0. Esta acción tiene mayor precedencia sobre los valores determinados por la señal irq y, por lo tanto, el valor producido por esta última señal, se conecta a la entrada de una segunda compuerta AND cuya entrada restante se conecta a la señal $\overline{reset}$; la salida de esta última compuerta se conecta a la señal mwr en el circuito.

3.5.2. Contador de programa, PC

El contador de programa o módulo PC de la figura 73 consta, principalmente, de un registro de 32 bits encargado de almacenar la dirección de memoria en la que se encuentra la instrucción a ejecutar en el presente ciclo de reloj, es decir, el valor $PC[31:0]$. El bit más significativo de dicha dirección,

$PC[31]$, se utiliza como el bit de supervisor. Cuando el bit de supervisor es igual a 0, el procesador se encuentra en modo usuario, ejecutando programas normalmente y con las interrupciones habilitadas. Cuando el bit de supervisor es igual a 1, el procesador se encuentra en modo supervisor, o modo kernel, ejecutando código del sistema operativo y con las interrupciones deshabilitadas.

Figura 104. **Registro de 32 bits del módulo PC**

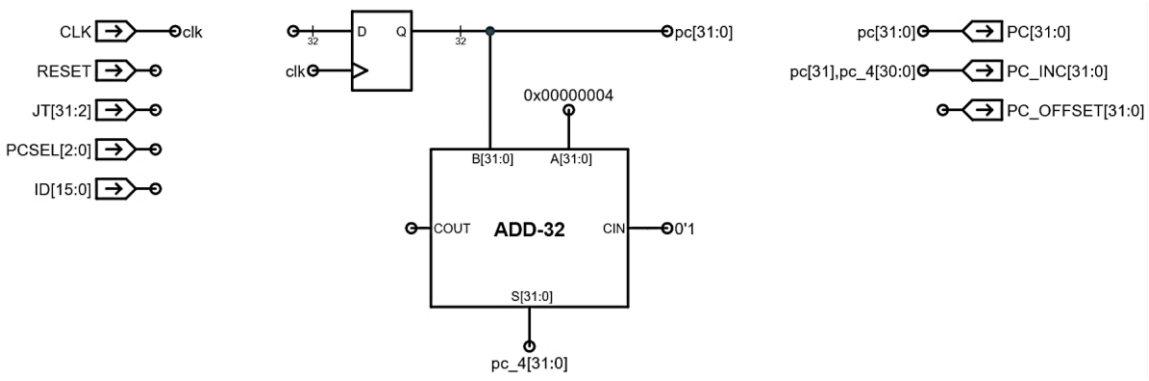


Fuente: elaboración propia, empelando Jade Circuit Simulator.

En la figura 104 se muestra la representación compacta del registro de 32 bits junto con las terminales de entrada y salida del módulo PC, obsérvese que la salida del registro se conecta a la señal $pc[31:0]$, por lo tanto, esta señal es la que contiene el valor actual del registro. Se debe aclarar que, por conveniencia, en términos de representación, se ha utilizado sólo un flip-flop cuya entrada y salida es de 32 bits, para representar los 32 flip-flops de los que se compone el registro.

El módulo contiene 2 sumadores de rizo de 32 bits, idénticos al que se utilizó para el diseño de la ALU de la subsección 3.4.1. Un sumador se encarga de calcular el valor $PC + 4$ mientras que el otro calcula la dirección especificada por los saltos condicionales, es decir, el valor $(PC + 4) + 4 \cdot SEXT(constante)$.

Figura 105. **Hardware para el cálculo de $PC + 4$**

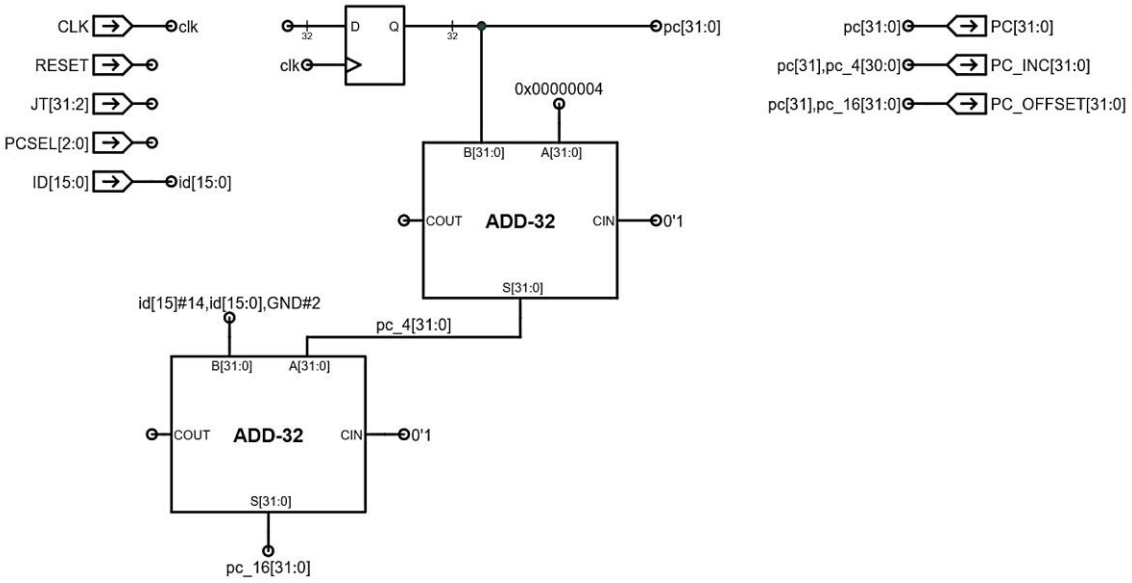


Fuente: elaboración propia, empleando Jade Circuit Simulator.

El primer sumador, encargado de calcular el valor $PC + 4$, únicamente se encarga de sumar la señal $pc[31:0]$ con la constante $0x00000004$, calculando así, la dirección de la instrucción que le sigue a la instrucción actual. En la figura 105 se muestra un sumador de 32 bits idéntico al que se utilizó en la figura 93 para la unidad aritmética, la señal $pc[31:0]$ se conecta a la entrada $B[31:0]$ y la constante $0x00000004$, a la entrada $A[31:0]$. La entrada CIN se conecta a la constante $0'1$, debido a que no existe acarreo de entrada. Una vez realizado el cálculo, el resultado de la suma ($PC + 4$) se conecta a la señal $pc\_4[31:0]$.

Debido a que el sumador, al incrementar el valor del registro PC, puede pasar de modo usuario ($pc\_4[31] = 0$) a modo supervisor ($pc\_4[31] = 1$), no es posible conectar la señal $pc\_4[31:0]$ con la terminal de salida $pc\_inc[31:0]$; para evitar este inconveniente, se conecta la señal compuesta $pc[31], pc\_4[30:0]$, a la terminal de salida $pc\_inc[31:0]$, asegurándose de que el bit de supervisor se mantiene sin cambios cuando esta señal compuesta se elige como el siguiente valor para el registro PC.

Figura 106. **Hardware para el cálculo de $(PC + 4) + 4 \cdot SEXT(constante)$**



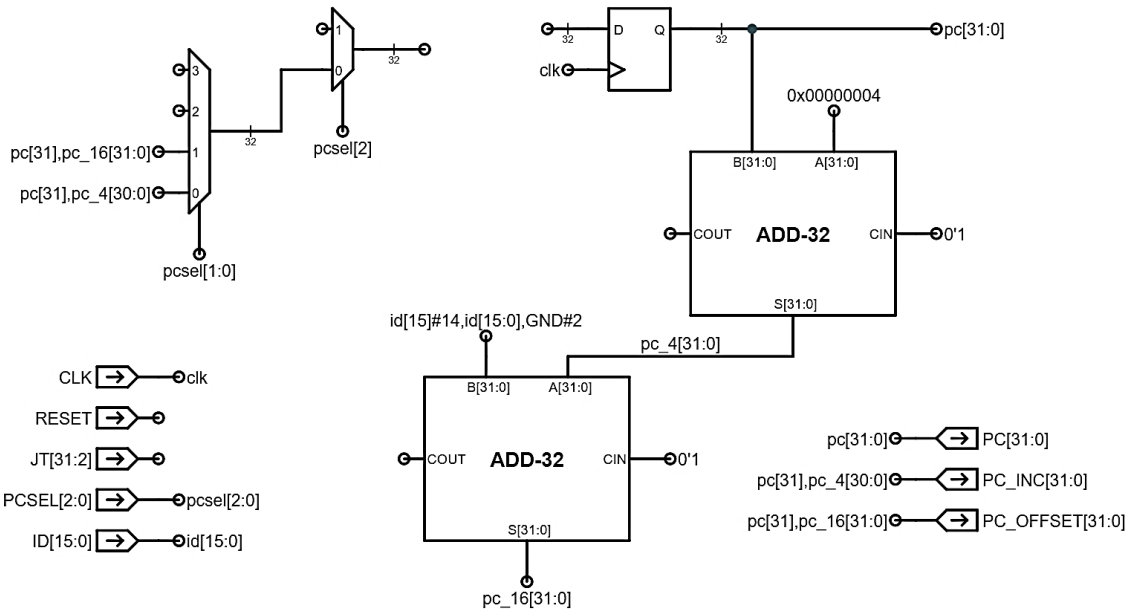
Fuente: elaboración propia, empleando Jade Circuit Simulator.

El segundo sumador es el encargado de calcular la dirección especificada por la entrada $id[15:0]$ para los saltos condicionales en las instrucciones BEQ y BNE, en otras palabras, realiza el cálculo de la expresión $(PC + 4) + 4 \cdot SEXT(id[15:0])$. La extensión con signo y la multiplicación en la expresión anterior se pueden implementar sin la necesidad de lógica adicional, es decir, realizando únicamente la conexión de señales correspondiente; la expresión $SEXT(id[15:0])$ se puede implementar a través de la señal compuesta $id[15]\#16, id[15:0]$ y, la multiplicación por 4 se implementa a través de la señal compuesta $id[15]\#14, id[15:0], GND\#2$, asumiendo que GND es igual a 0, en donde se ha realizado un desplazamiento lógico de 2 unidades a la izquierda.

En la figura 106 se muestra un sumador de 32 bits adicional, cuyas entradas $A[31:0]$ y $B[31:0]$ se conectan con las señales $pc\_4[31:0]$ y $id[15]\#16, id[15:0]$

respectivamente; de nuevo, como no existe acarreo, la entrada CIN se conecta con la constante $0'1$. El resultado de la suma se conecta a la señal $pc\_16[31:0]$ y, exactamente por las mismas razones descritas anteriormente, se conecta la señal compuesta $pc[31], pc\_16[30:0]$ a la terminal de salida $pc\_offset[31:0]$.

Figura 107. **Multiplexor de 5 entradas controlado por $pcsel[2:0]$**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

La señal $pcsel[2:0]$ controla el multiplexor compuesto, de 5 entradas, que se muestra en la figura 107, este dispositivo se encarga de controlar el valor que se cargará al registro de 32 bits en el siguiente ciclo de reloj; la señal $pcsel[1:0]$ controla el multiplexor de cuatro entradas cuya salida se conecta a una de las dos entradas del multiplexor controlado por la señal $pcsel[2]$.

Recordar que, cuando $pcsel[2:0] = 0$, la siguiente instrucción a ejecutar se encuentra justo después de la instrucción en ejecución, por lo que la señal

$pc[31], pc\_4[30:0]$ se conecta a la entrada 0 del multiplexor. Cuando $pcsel[2:0] = 1$, se debe tomar un salto debido a una instrucción de salto condicional y, por lo tanto, la señal $p[31], pc\_16[30:0]$ se conecta a la entrada 1 del multiplexor.

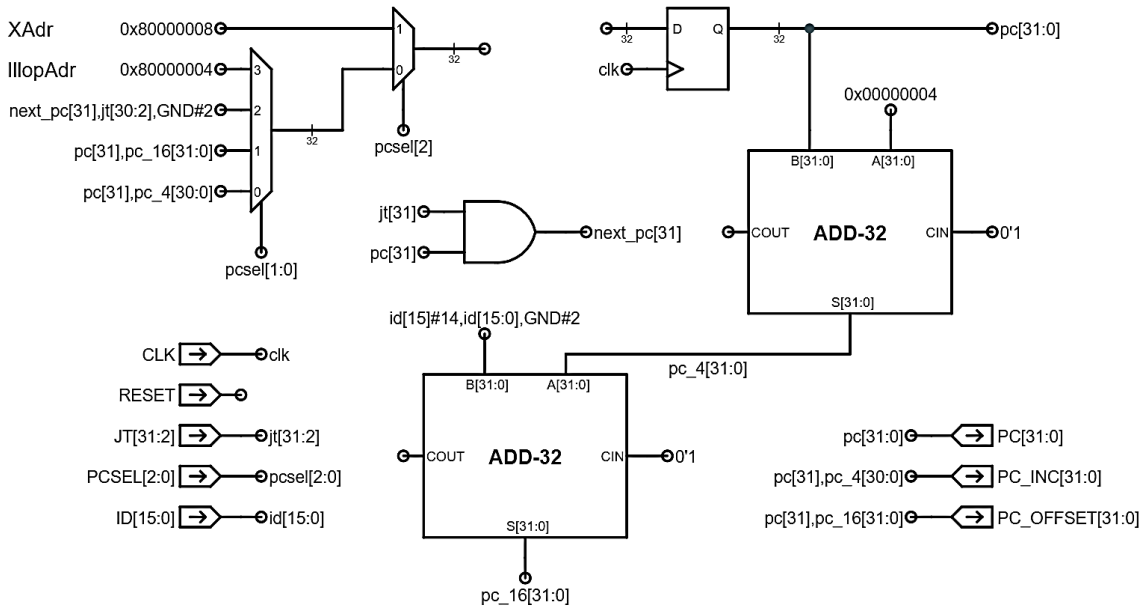
Tabla LXXV. **Especificación funcional para el bit $\boldsymbol{next\_pc}[\mathbf{31}]$**

$\boldsymbol{pc}[\mathbf{31}]$	$\boldsymbol{jt}[\mathbf{31}]$	$\boldsymbol{next\_pc}[\mathbf{31}]$
0	X	0
1	0	0
1	1	1

Fuente: elaboración propia.

Cuando $pcsel[2:0] = 2$, el siguiente valor del contador de programa proviene de una versión ligeramente modificada del registro Ra, instrucción JMP, la cual proviene del bloque de registros, y se encuentra dentro de la señal $jt[31:2]$. El contenido de la señal $jt[31:2]$, es idéntico al del registro Ra, con la diferencia de que los 2 bits menos significativos de este último se omiten, con el objetivo de formar la señal compuesta $jt[31:2], GND\#2$, garantizando así, que la dirección a utilizar siempre es un múltiplo de 4.

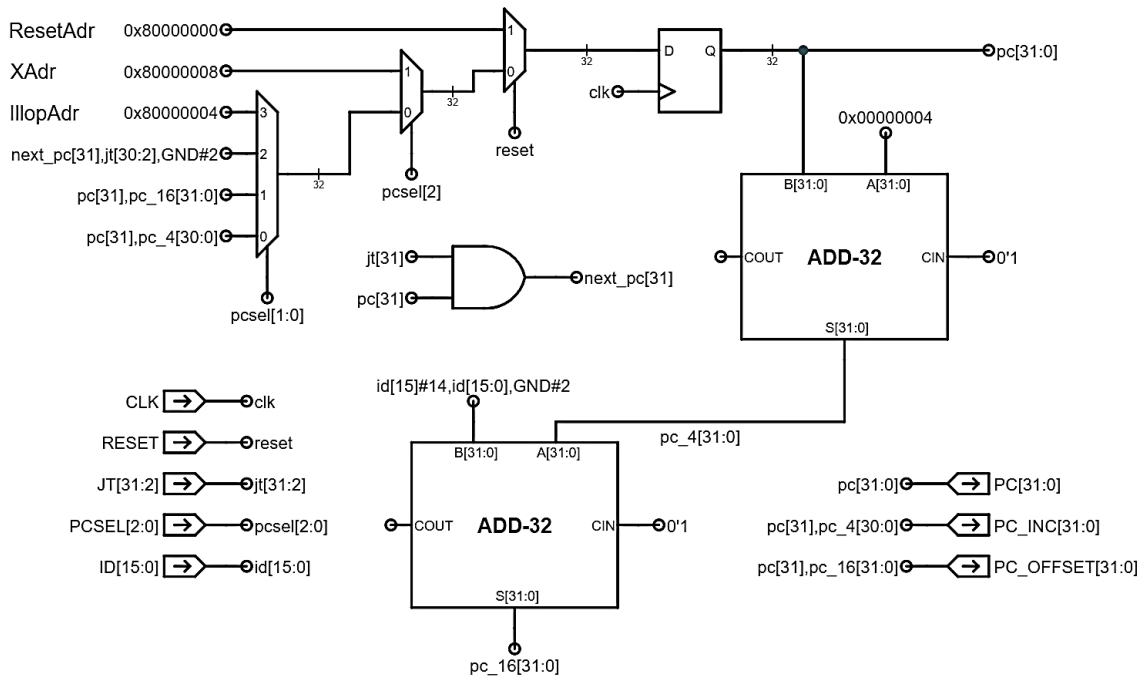
Figura 108. **Entradas restantes del multiplexor PCSEL**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

El bit $jt[31]$ es el nuevo valor propuesto para el bit de supervisor. Se debe añadir lógica para asegurarse de que la instrucción JMP únicamente pueda limpiar o dejar intacto este bit (ver subsección 2.6.5). En la tabla LXXV se muestra la especificación funcional para el bit $next\_pc[31]$, el cual es el bit que sustituye el valor de $jt[31]$, la implementación de dicha especificación se muestra en la figura 108, únicamente consiste en una compuerta AND de dos entradas. El valor de la señal $next\_pc[31]$ se utiliza para formar la señal compuesta $next_{pc[31]}, jt[30:2], GND\#2$, la cual se conecta a la entrada 2 del multiplexor.

Figura 109. **Circuito interno del módulo PC**



Fuente: elaboración propia, empleando Jade Circuit Simulator.

En la figura 108 también se indican las conexiones restantes para los casos en los que ocurren excepciones. La entrada $pcsel[2:0] = 3$ se selecciona cuando la instrucción actual contiene un opcode ilegal, por lo que se debe asignar el siguiente valor del registro PC a la dirección $0x800000004$, dicha constante se conecta a la entrada 3 del multiplexor. La entrada $pcsel[2:0] = 4$ se selecciona cuando el procesador debe responder ante una interrupción, por lo que se conecta la dirección $0x800000008$ a la entrada 5 del multiplexor compuesto, o la entrada 1 del multiplexor de la derecha.

Por último, y para completar el diseño del módulo PC, es necesario añadir lógica para forzar el registro PC a la dirección $0x80000000$, en caso de que la señal $reset$ sea igual a 1. En la figura 109 se muestra un multiplexor adicional,

controlado por la señal $reset$, permite seleccionar entre la salida del multiplexor de 5 entradas compuesto, y la constante $0x80000000$; la salida de este último multiplexor se conecta a la entrada del registro PC. Nótese que la señal $reset$ tiene mayor precedencia a la señal de control $pcsel[2:0]$, por lo que el valor de este último resulta irrelevante cuando $reset = 1$.

4. IMPLEMENTACIÓN Y SIMULACIÓN DEL DISEÑO

En este capítulo se presentan los resultados de la implementación de la arquitectura descrita en el capítulo 3, para la FPGA Xilinx Spartan 3E-1200, utilizando el entorno de desarrollo Xilinx ISE Design Suite 14,6 y el lenguaje de especificación VHDL.

Para cada uno de los niveles de abstracción presentes en la arquitectura, se proporciona el código fuente, el diagrama RTL producido por la síntesis del código y, para los niveles de mayor relevancia, se incluyen muestras representativas de los resultados de simulación en los que se utilizan diferentes testbench.

4.1. Repositorio remoto

Toda la información presentada en las secciones posteriores se encuentra disponible en detalle dentro de un repositorio remoto de acceso público y gratuito que contiene todos los archivos de proyecto necesarios para que el lector pueda estudiar, modificar, y utilizar el contenido libremente con cualquier fin y redistribuirlo con cambios o mejoras o sin ellas. Este repositorio se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits.

Conforme se presentan los resultados para cada uno de los módulos de la arquitectura, se hace referencia a URLs de archivos específicos dentro del repositorio con la finalidad de que el lector pueda localizar de manera más eficiente el archivo o directorio fuente de la información en discusión.

4.2. Bloque de registros de múltiples puertos, REGFILE

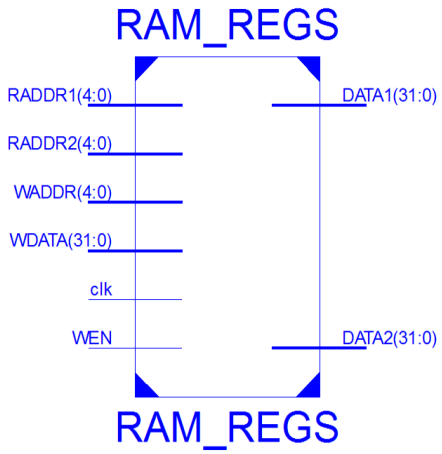
El módulo REGFILE se implementa jerárquicamente y con base en las especificaciones del capítulo 3. Primero se implementa una RAM de múltiples puertos y luego, se agrega la lógica de control.

4.2.1. RAM de múltiples puertos

El archivo que contiene el código VHDL fuente de la implementación del módulo de la RAM de múltiples puertos recibe el nombre de *RAM_REGS.vhd* dentro del directorio del proyecto. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/RAM_REGS.vhd.

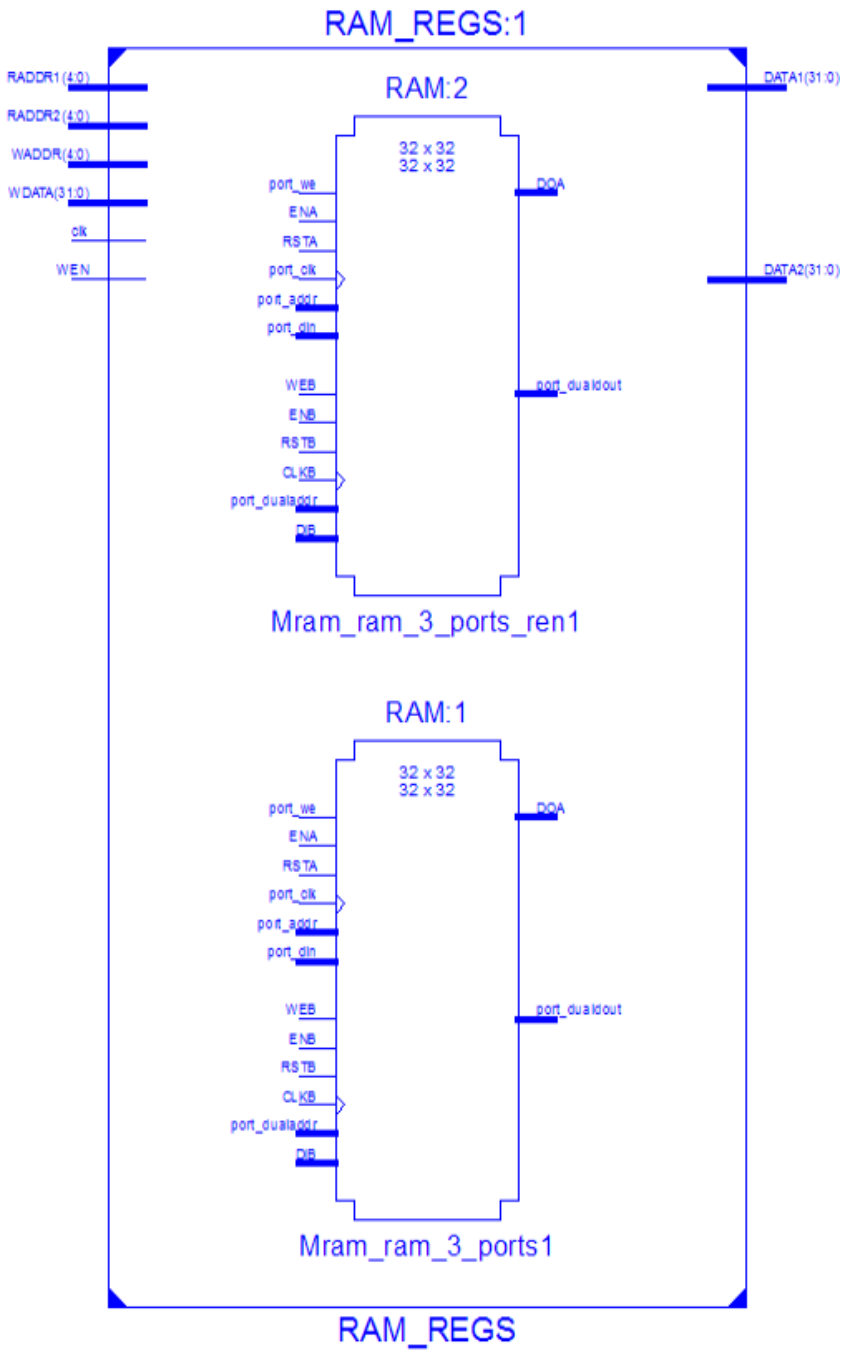
En las figuras 110 y 111 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente.

Figura 110. **Diagrama RTL de alto nivel RAM_REGS**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 111. **Diagrama RTL del módulo RAM_REGS**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

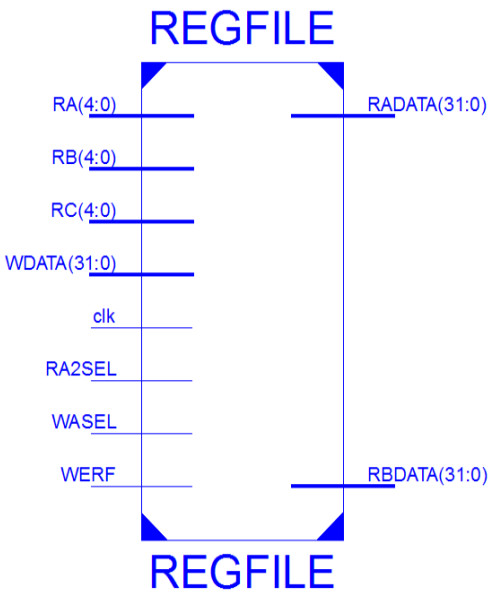
4.2.2. Circuito general

El archivo que contiene el código VHDL fuente de la implementación del bloque de registros de múltiples puertos recibe el nombre de *REGFILE.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/REGFILE.vhd

4.2.2.1. Diagramas esquemáticos RTL

En la figura 112 se muestra el diagrama RTL de alto nivel, posterior a la síntesis del código fuente.

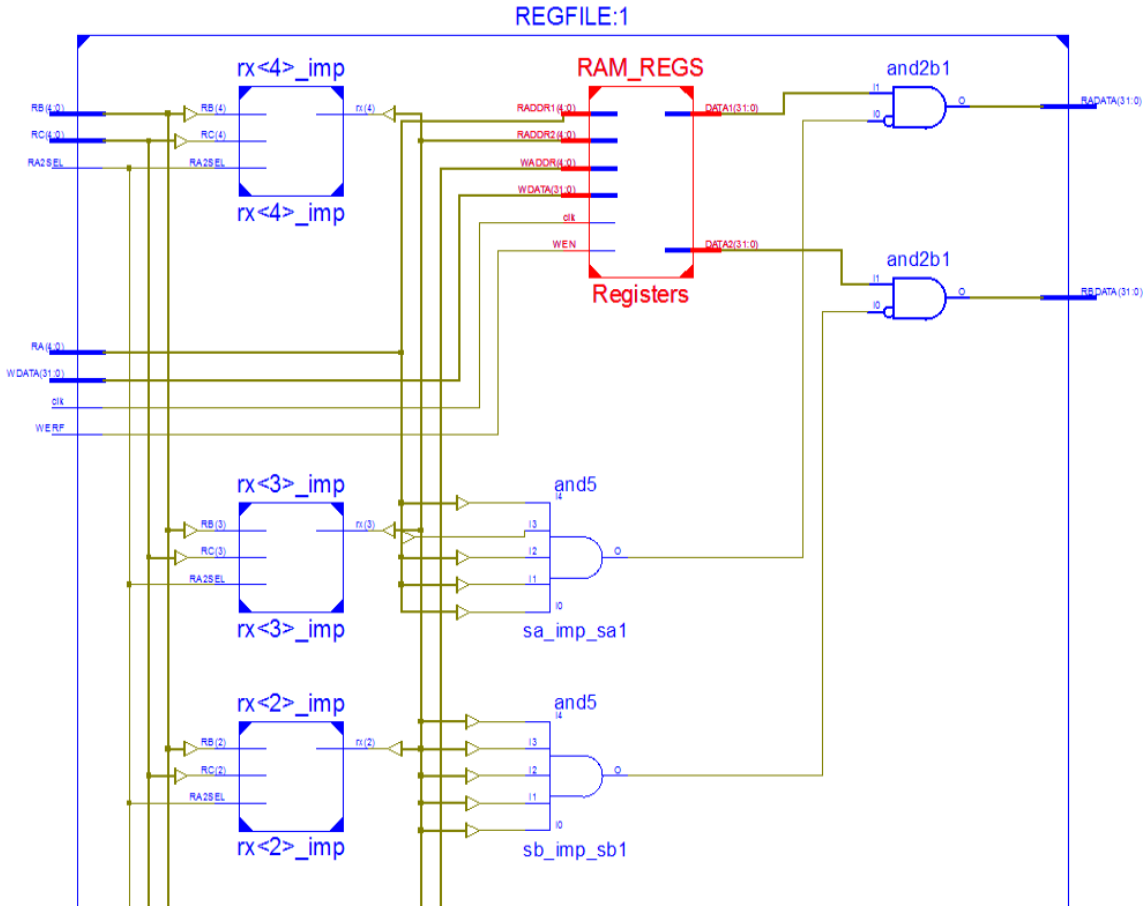
Figura 112. **Diagrama RTL de alto nivel REGFILE**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En la figura 113 se muestra una parte del diagrama RTL interno del módulo, nótese la interconexión del módulo RAM_REGS, en rojo, con el hardware adicional encargado de cumplir la funcionalidad descrita en el código fuente.

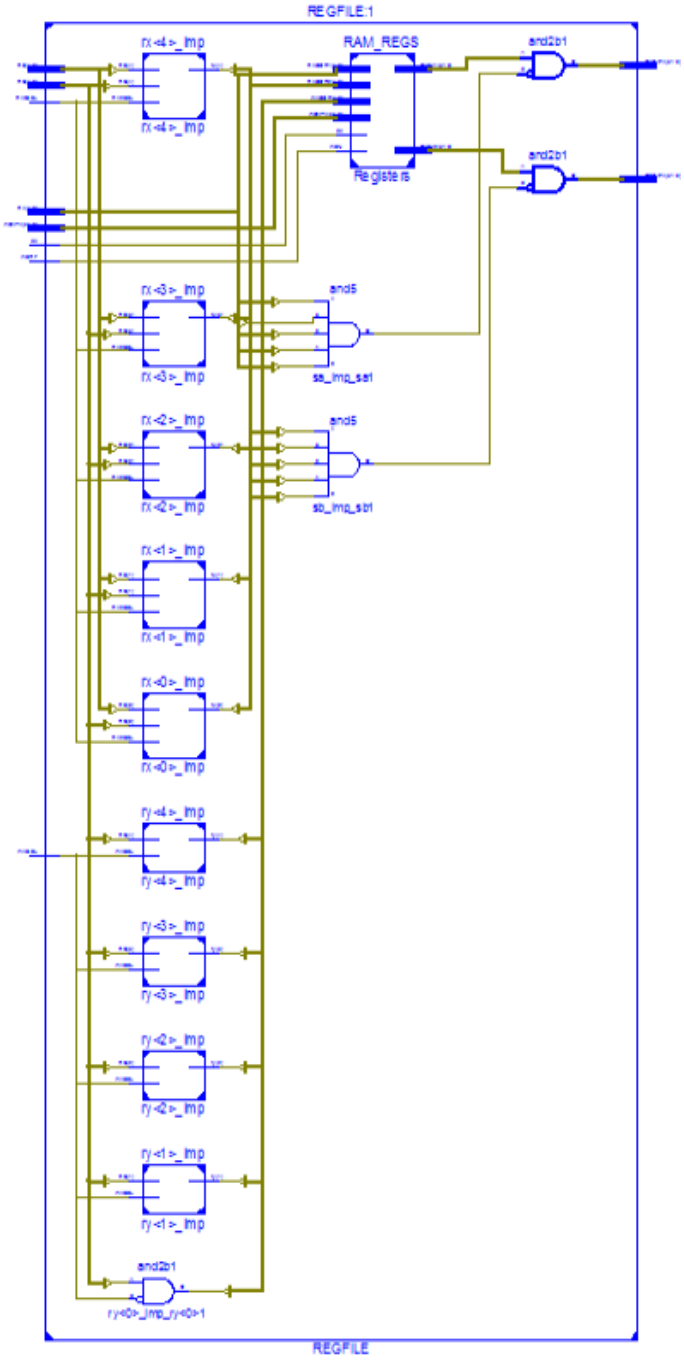
Figura 113. **Diagrama RTL parcial del módulo REGFILE**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Por último, para completar los resultados, en la figura 114 se muestra el diagrama RTL completo del módulo.

Figura 114. **Diagrama RTL del módulo REGFILE**



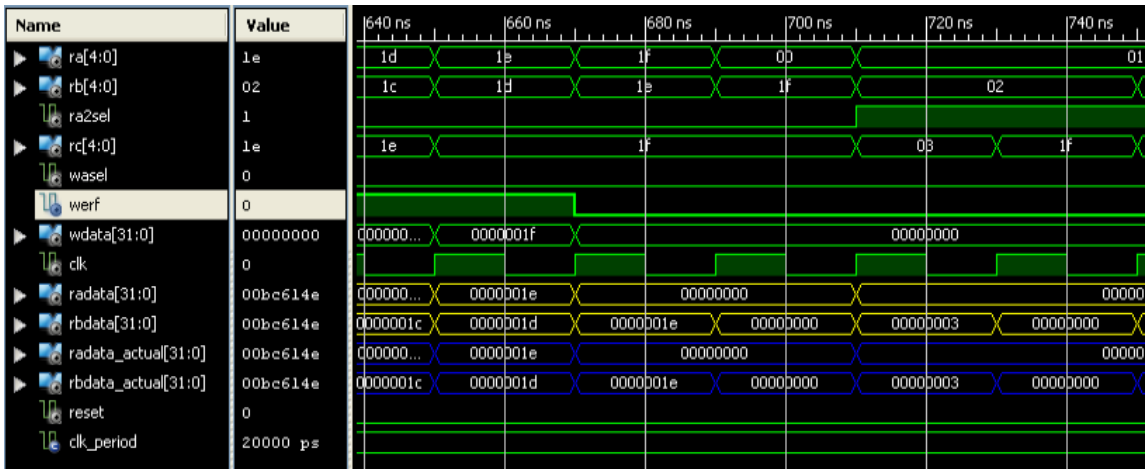
Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.2.2.2. Simulación

El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo REGFILE.vhd recibe el nombre de *REGFILE_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/REGFILE_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *regfile_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/regfile_tests.csv.

Figura 115. **Simulación del módulo REGFILE**



Fuente: elaboración propia, empleando ISim Simulator.

En la figura 115 se muestra una parte de los resultados de simulación, las señales de color azul representan las salidas esperadas mientras que las señales

de color amarillo representan las salidas actuales para las diferentes combinaciones de entradas.

4.3. Unidad aritmética lógica, ALU

El módulo ALU se implementa jerárquicamente, siguiendo un enfoque ascendente, y con base en las especificaciones del capítulo 3. Se comienza con la implementación de cada uno de los bloques encargados de realizar las operaciones de 32 bits y luego se añade la lógica de control correspondiente.

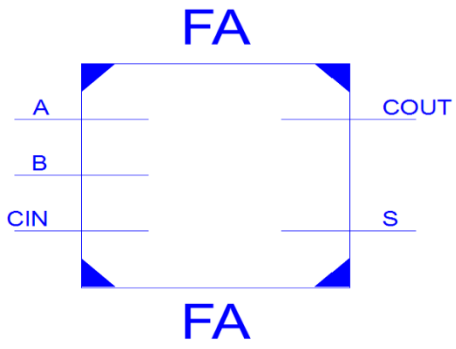
4.3.1. Sumador completo

El archivo que contiene el código VHDL fuente de la implementación del módulo sumador completo recibe el nombre de *FA.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/FA.vhd.

4.3.1.1. Diagramas esquemáticos RTL

En la figura 116 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

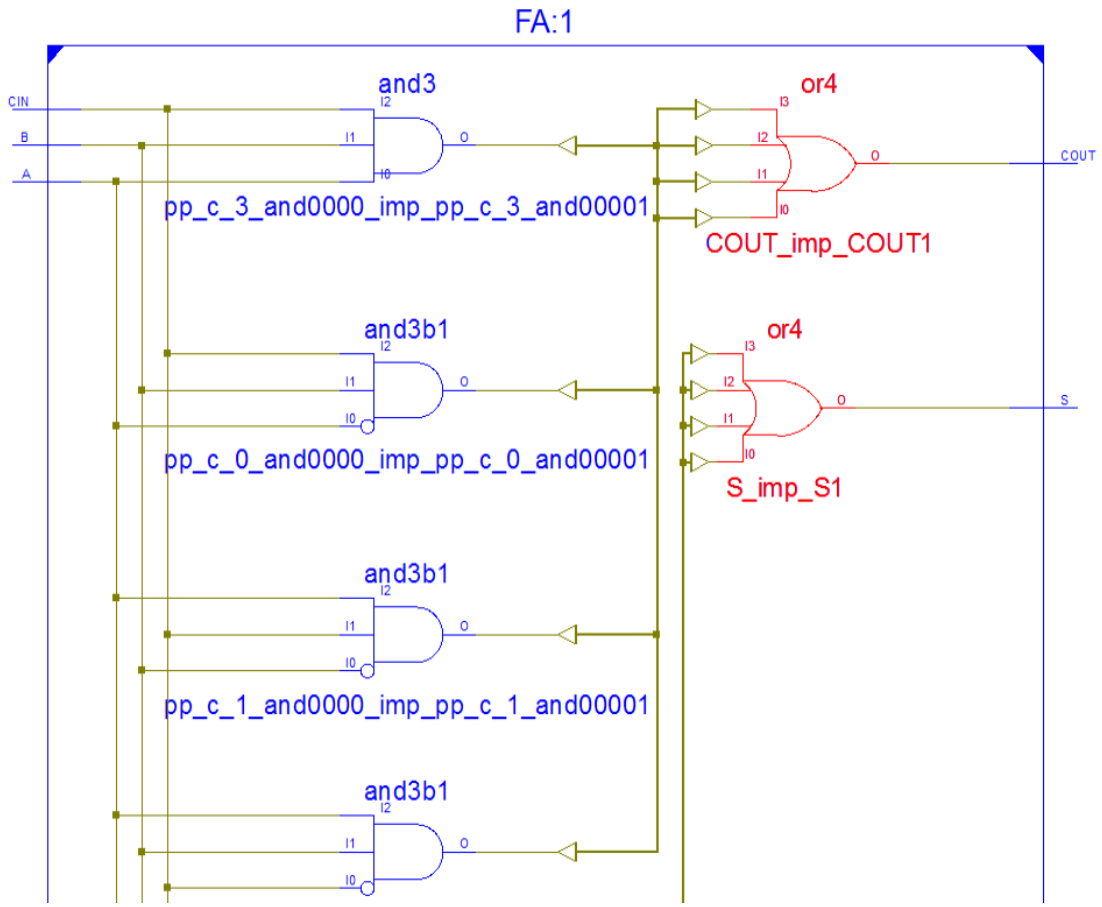
Figura 116. **Diagrama RTL de alto nivel FA**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En la figura 117 se muestra una parte del diagrama RTL interno, en ella se puede apreciar la estructura de un circuito en representación de suma de productos, en donde las compuertas AND representan los productos y las compuertas OR (en rojo) representan la suma de estos.

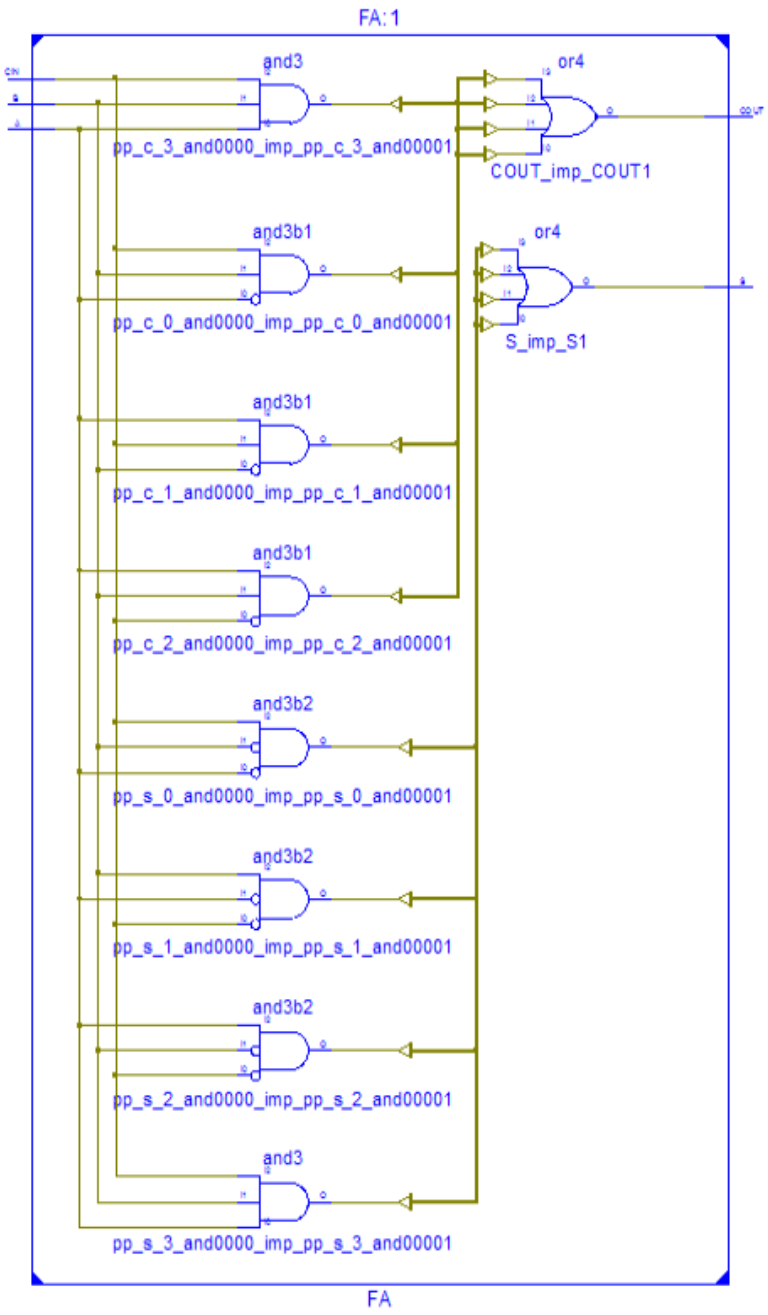
Figura 117. **Diagrama RTL parcial del módulo FA**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Por último y, para completar los resultados de este módulo, en la figura 118 se muestra el diagrama RTL completo.

Figura 118. **Diagrama RTL del módulo FA**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

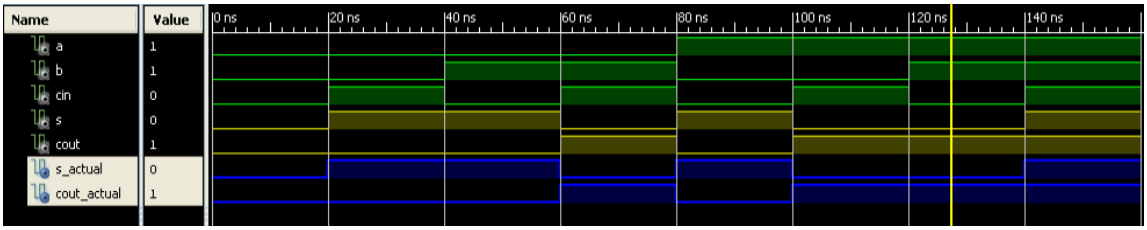
4.3.1.2. Simulación

El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo FA.vhd recibe el nombre de *FA_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/FA_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *fa_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/fa_tests.csv.

En la figura 119 se muestran los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en la sección anterior, nótese que no se necesitan más de ocho combinaciones de entradas para comprobar la funcionalidad del módulo y, por lo tanto, se muestran los resultados completos de la simulación.

Figura 119. **Simulación del módulo FA**



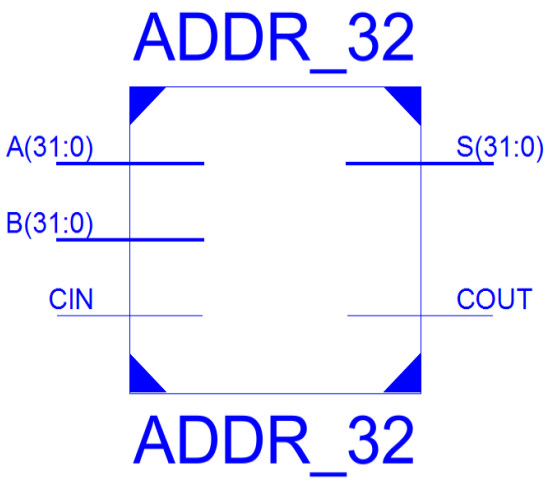
Fuente: elaboración propia, empleando ISim Simulator.

4.3.2. Sumador de 32 bits

El archivo que contiene el código VHDL fuente de la implementación del módulo sumador de 32 bits recibe el nombre de *ADDR_32.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ADDR_32.vhd.

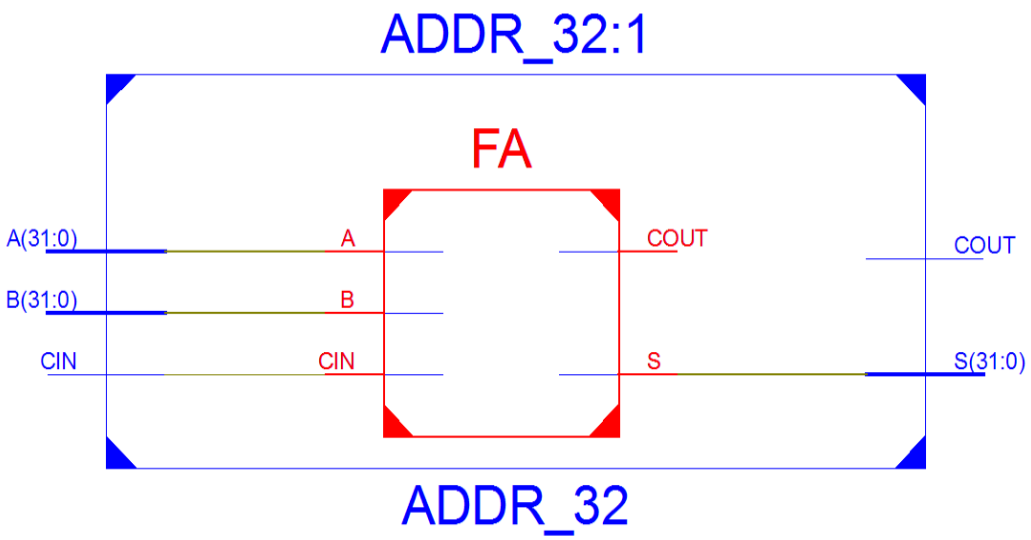
En las figuras 120 y 121 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente. Nótese que, en la figura 121, el entorno de desarrollo representa la conexión en cascada de los 32 módulos FA a través de un solo módulo FA con entradas y salidas representadas como vectores de 32 bits, esto se debe a que el proceso de síntesis utiliza la configuración de optimización de área, o la configuración por defecto.

Figura 120. **Diagrama RTL de alto nivel ADDR_32**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 121. **Diagrama RTL del módulo ADDR_32**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

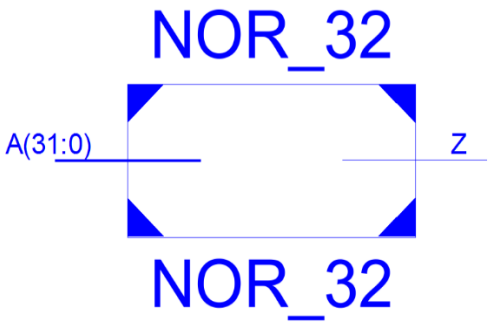
4.3.3. Compuerta NOR de 32 entradas

En este capítulo, la compuerta NOR de 32 entradas se implementa como un módulo separado, el cual se utiliza posteriormente dentro de la implementación de la unidad aritmética y el módulo principal del procesador, esto se realiza con el objetivo de reducir la complejidad del código fuente y los diagramas RTL.

El archivo que contiene el código VHDL fuente de la implementación de la compuerta NOR de 32 entradas recibe el nombre de *NOR_32.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/NOR_32.vhd.

En la figura 122 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

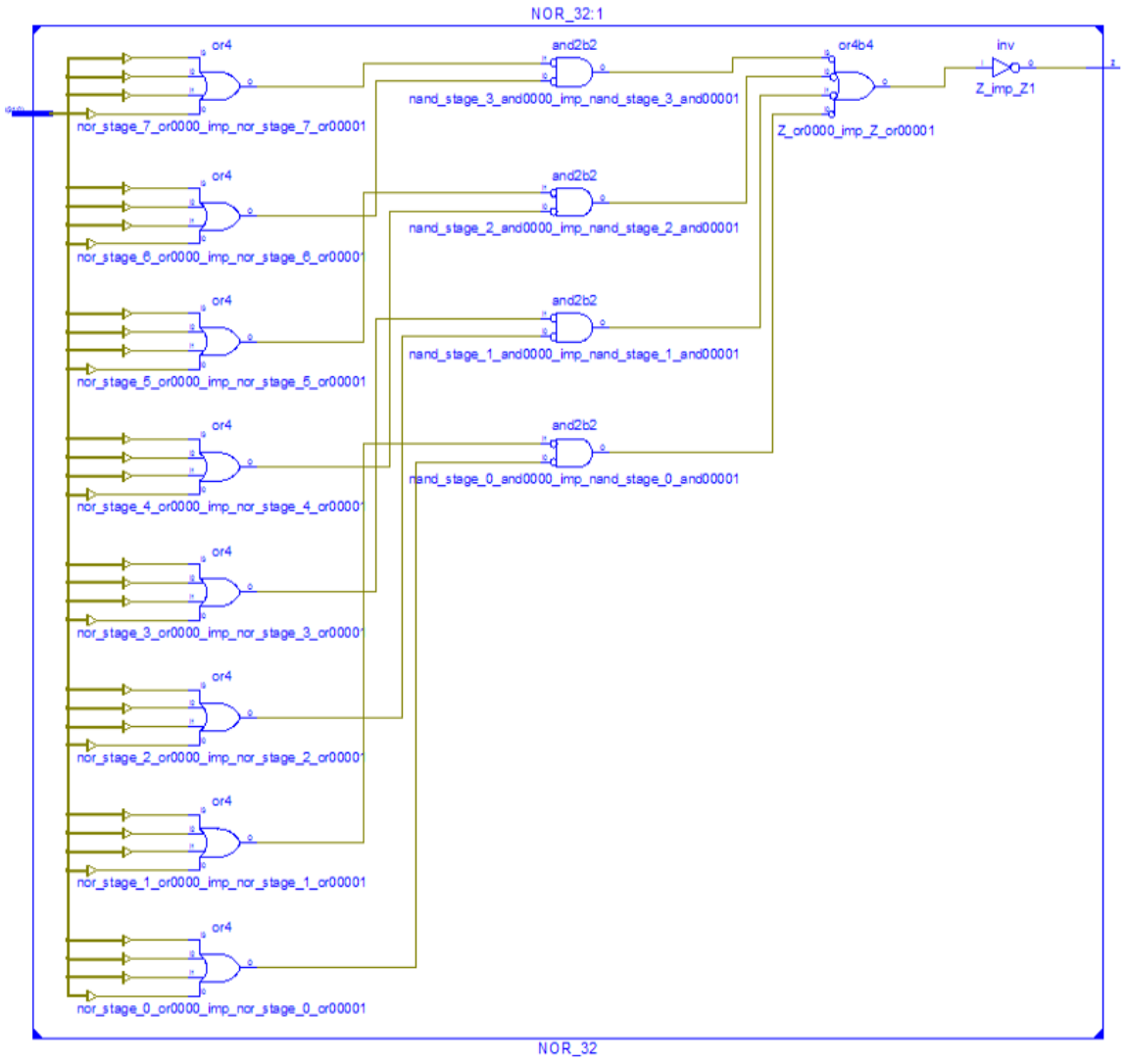
Figura 122. **Diagrama RTL de alto nivel NOR_32**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En la figura 123 se muestra el diagrama interno del módulo NOR_32, en ella se puede identificar las diferentes etapas de compuertas NOR y $NAND$ alternantes.

Figura 123. **Diagrama RTL del módulo NOR_32**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.3.4. Unidad aritmética

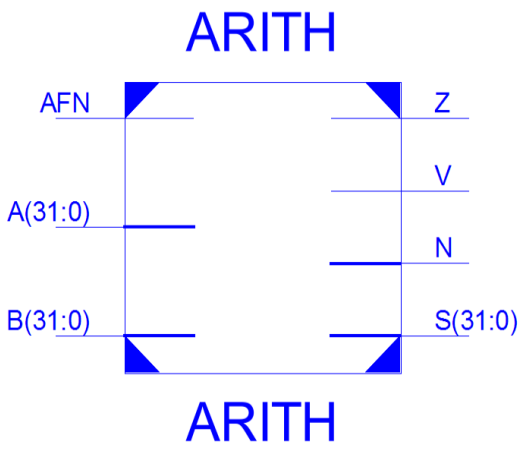
El archivo que contiene el código VHDL fuente de la implementación de la unidad aritmética recibe el nombre de *ARITH.vhd* dentro del directorio del

proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ARITH.vhd.

4.3.4.1. Diagramas esquemáticos RTL

En la figura 124 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

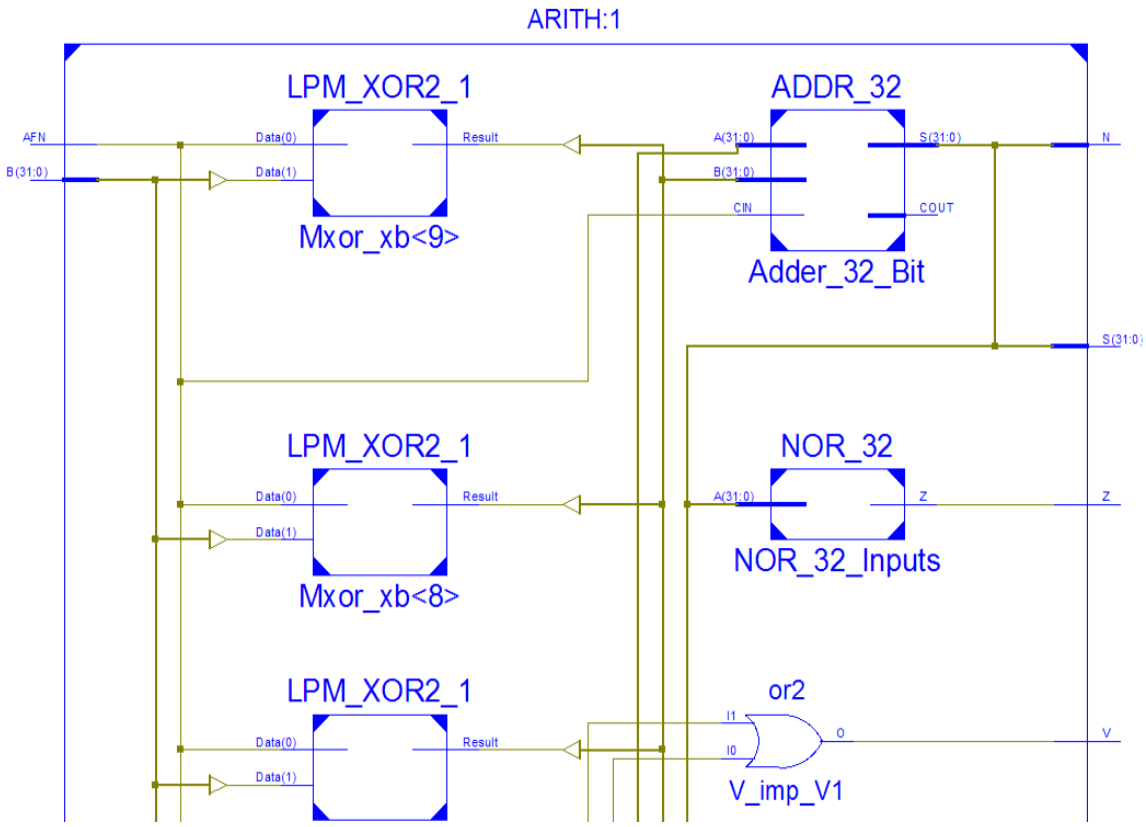
Figura 124. **Diagrama RTL de alto nivel ARITH**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

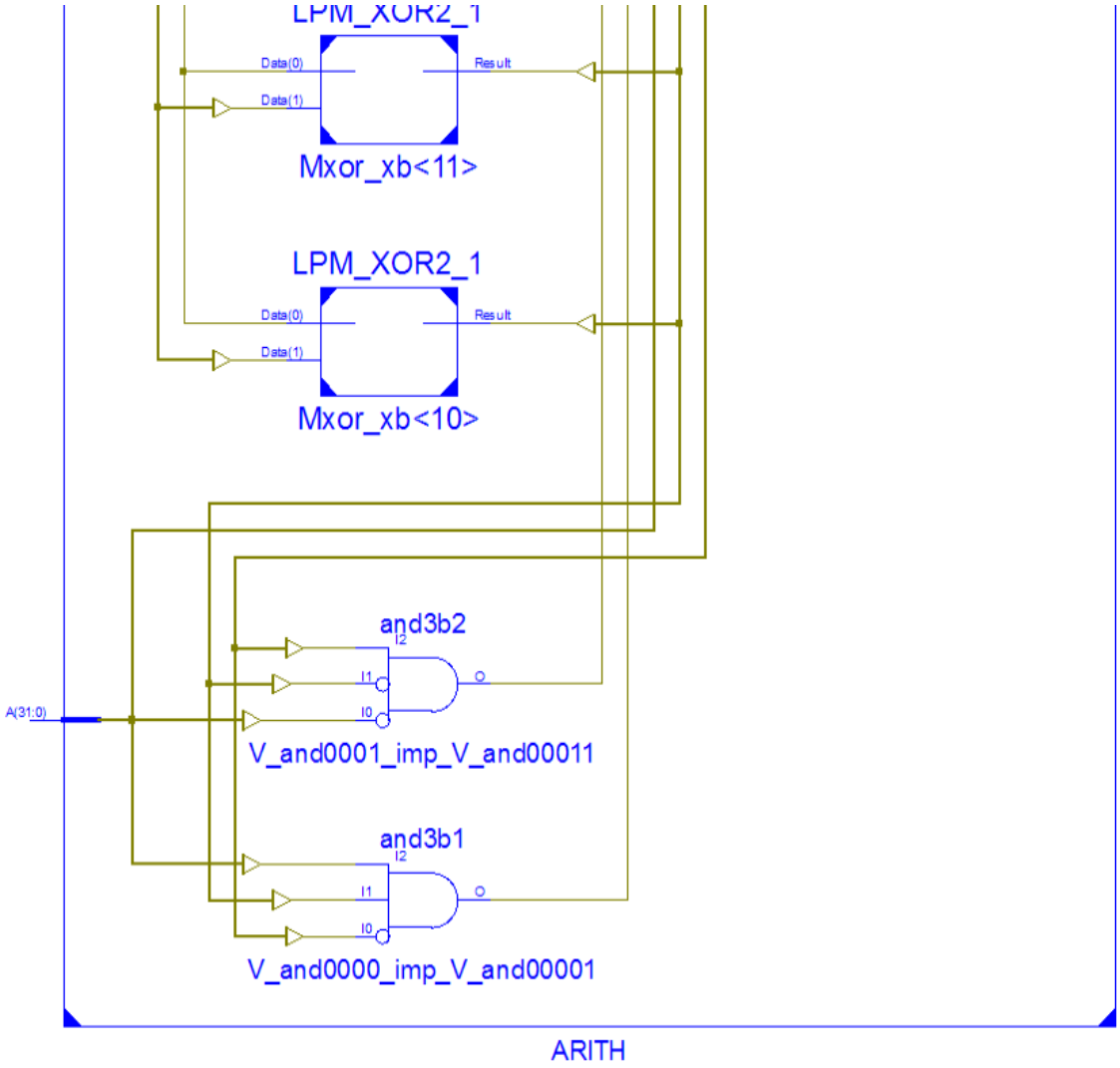
Debido a que el diagrama RTL interno del módulo es demasiado extenso, principalmente por las 32 compuertas XOR encargadas de obtener el complemento a uno del operando $B(31:0)$, únicamente se muestran las partes del diagrama de mayor relevancia. En la figura 125 se muestra la parte superior del diagrama en donde se pueden identificar el uso de los componentes ADDR_32 y NOR_32. Por último, en la figura 126, se muestra la parte inferior del diagrama.

Figura 125. **Diagrama superior del módulo ARITH**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 126. **Diagrama inferior del módulo ARITH**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.3.4.2. Simulación

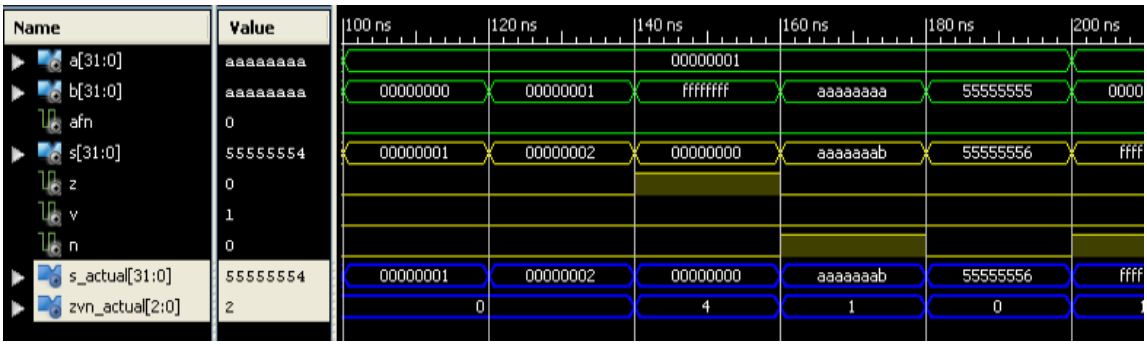
El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo ARITH.vhd recibe el nombre de *ARITH_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en

https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ARITH_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *arith_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/arith_tests.csv.

En la figura 127 se muestra una parte de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores.

Figura 127. **Simulación del módulo ARITH**



Fuente: elaboración propia, empleando ISim Simulator.

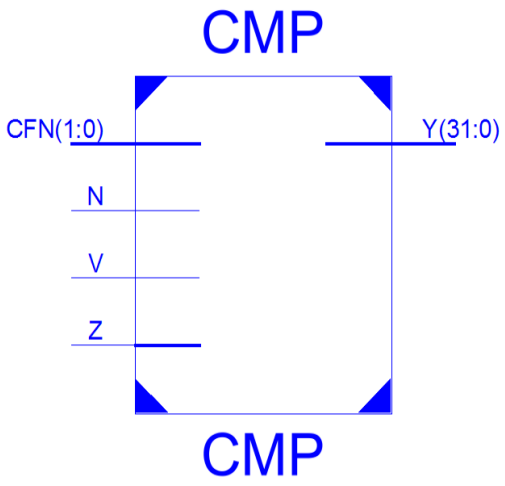
4.3.5. Unidad de comparación

El archivo que contiene el código VHDL fuente de la implementación de la unidad de comparación recibe el nombre de *CMP.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/CMP.vhd.

4.3.5.1. Diagramas esquemáticos RTL

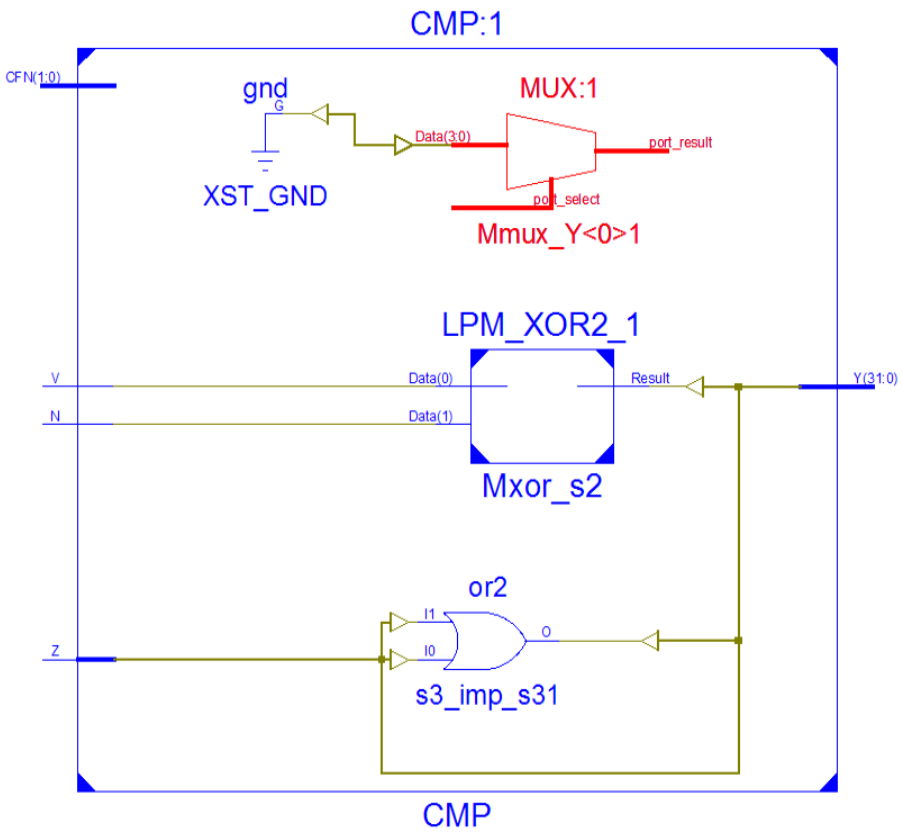
En las figuras 128 y 129 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente. En la figura 129 se puede apreciar el multiplexor, en rojo, encargado de seleccionar una de las diferentes operaciones de comparación.

Figura 128. **Diagrama RTL de alto nivel CMP**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 129. **Diagrama RTL del módulo CMP**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.3.5.2. Simulación

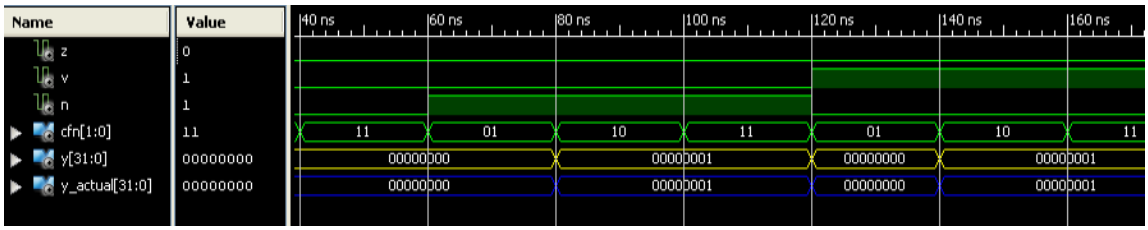
El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo CMP.vhd recibe el nombre de *CMP_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/CMP_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *cmp_tests.csv* dentro de la carpeta

llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/cmp_tests.csv.

En la figura 130 se muestra una parte de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores.

Figura 130. **Simulación del módulo CMP**



Fuente: elaboración propia, empleando ISim Simulator.

4.3.6. Unidad booleana

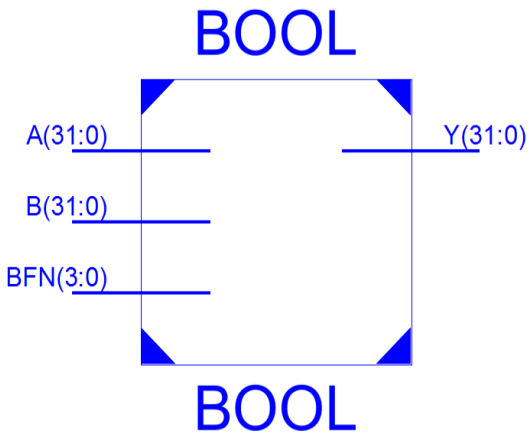
El archivo que contiene el código VHDL fuente de la implementación de la unidad de booleana recibe el nombre de *BOOL.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/BOOL.vhd.

4.3.6.1. Diagramas esquemáticos RTL

En las figuras 131 y 132 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente. Nótese que sólo se incluye la parte superior del diagrama interno, esto debido a

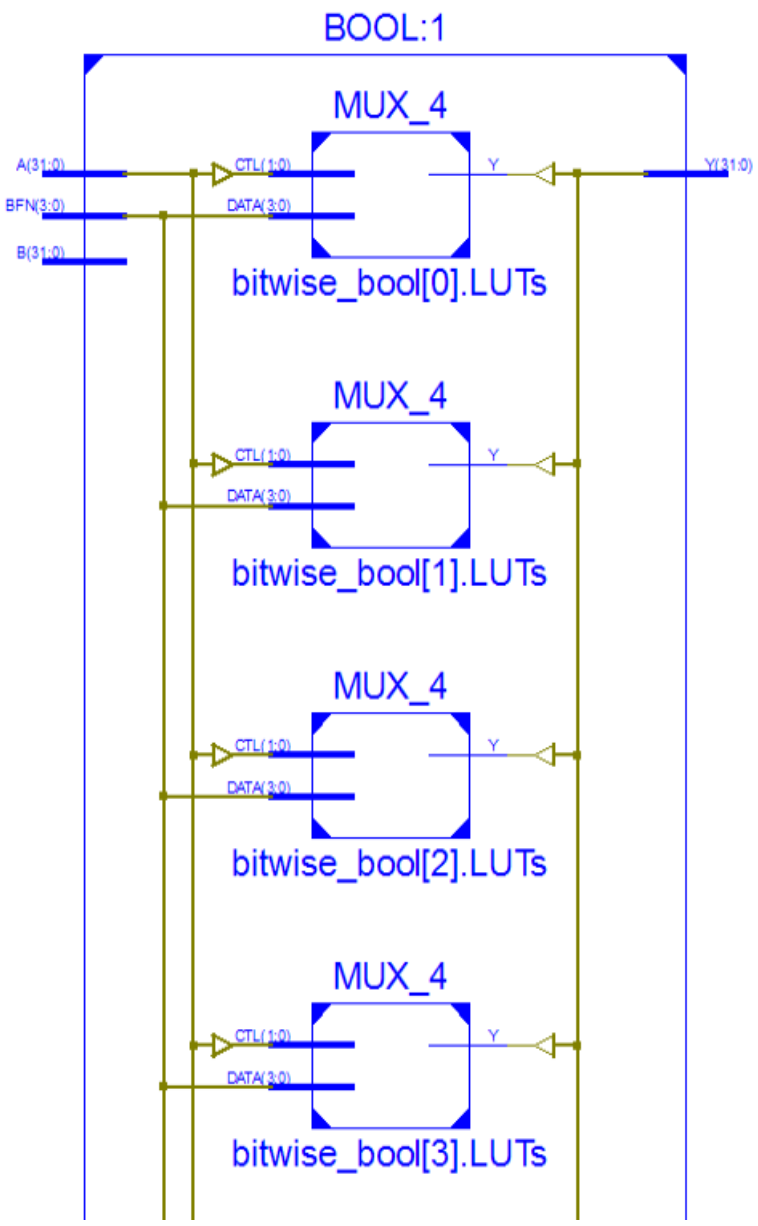
que únicamente consta de 32 multiplexores de dos líneas encargados de realizar la operación lógica bit a bit entre los operandos.

Figura 131. **Diagrama RTL de alto nivel BOOL**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 132. **Diagrama RTL parcial del módulo BOOL**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

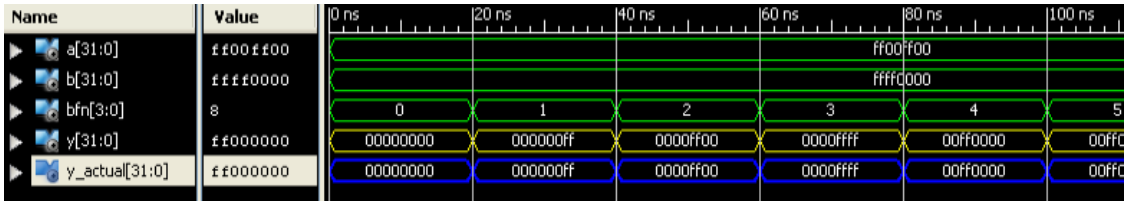
4.3.6.2. Simulación

El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo BOOL.vhd recibe el nombre de *BOOL_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/BOOL_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *bool_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/bool_tests.csv.

En la figura 133 se muestra una parte de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores.

Figura 133. **Simulación del módulo BOOL**



Fuente: elaboración propia, empleando ISim Simulator.

4.3.7. Unidad de desplazamiento

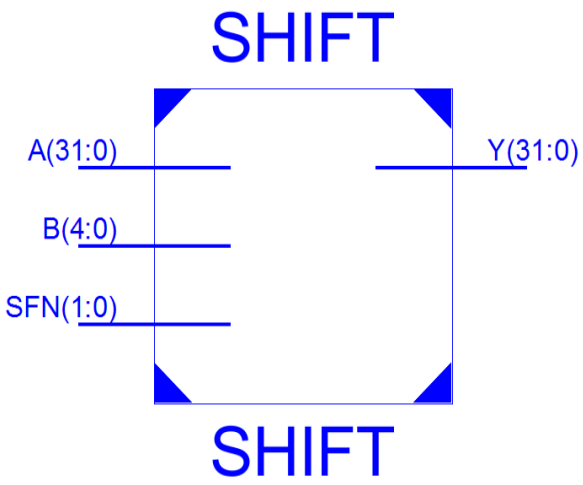
El archivo que contiene el código VHDL fuente de la implementación de la unidad de desplazamiento recibe el nombre de *SHIFT.vhd* dentro del directorio

del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/SHIFT.vhd.

4.3.7.1. Diagramas esquemáticos RTL

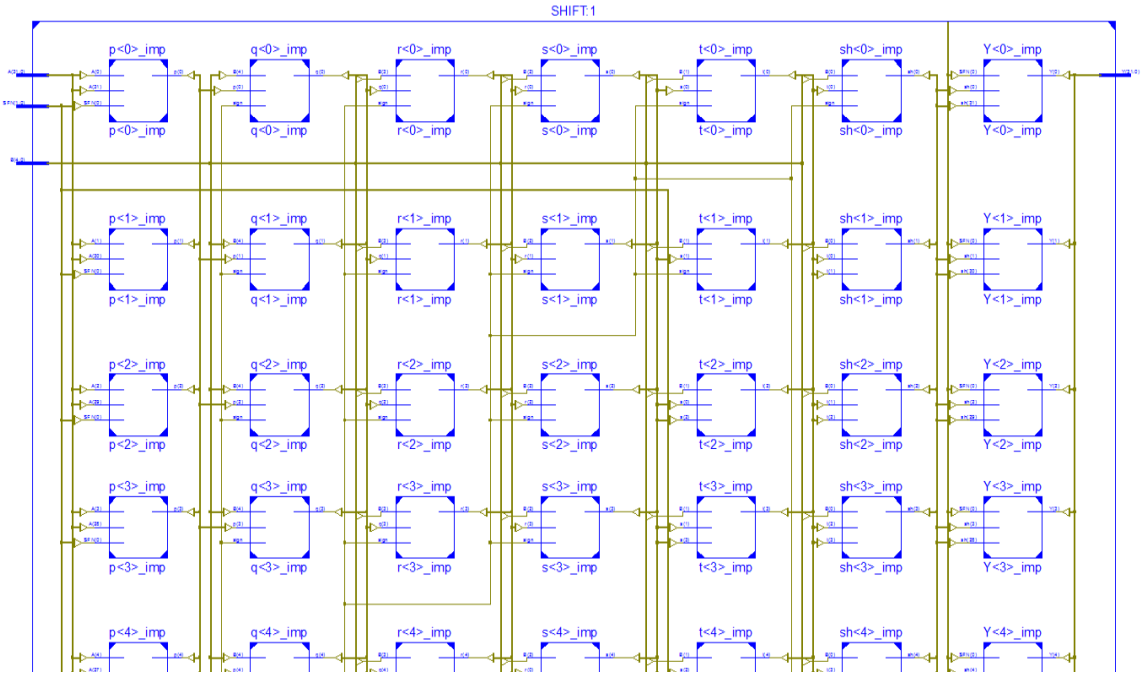
En las figuras 134 y 135 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente. Nótese que sólo se incluye una parte representativa del diagrama interno, esto debido a que la cantidad de multiplexores presente es demasiado grande para ser representada de forma legible.

Figura 134. **Diagrama de alto nivel SHIFT**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 135. **Diagrama RTL parcial del módulo SHIFT**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

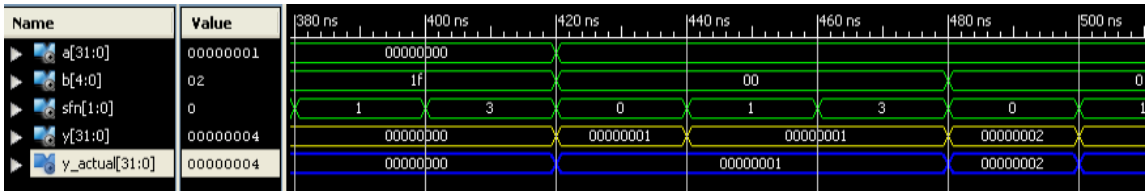
4.3.7.2. Simulación

El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo SHIFT.vhd recibe el nombre de *SHIFT_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/SHIFT_tb.vhd

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *shift_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/shift_tests.csv.

En la figura 136 se muestra un pequeño fragmento de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores.

Figura 136. **Simulación del módulo SHIFT**



Fuente: elaboración propia, empleando ISim Simulator.

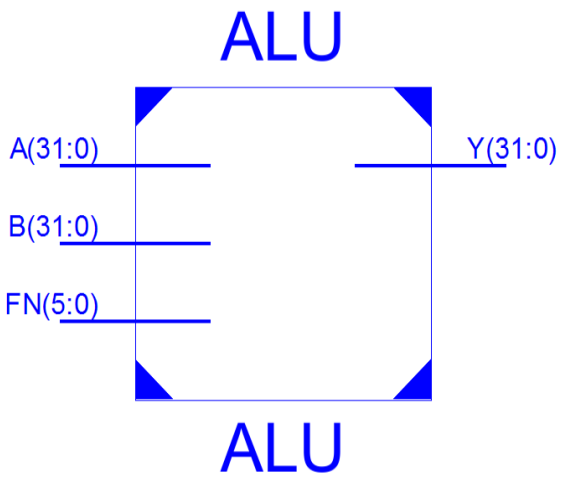
4.3.8. Diseño general

El archivo que contiene el código VHDL fuente de la implementación del diseño general de la ALU recibe el nombre de *ALU.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ALU.vhd.

4.3.8.1. Diagramas esquemáticos RTL

En la figura 137 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

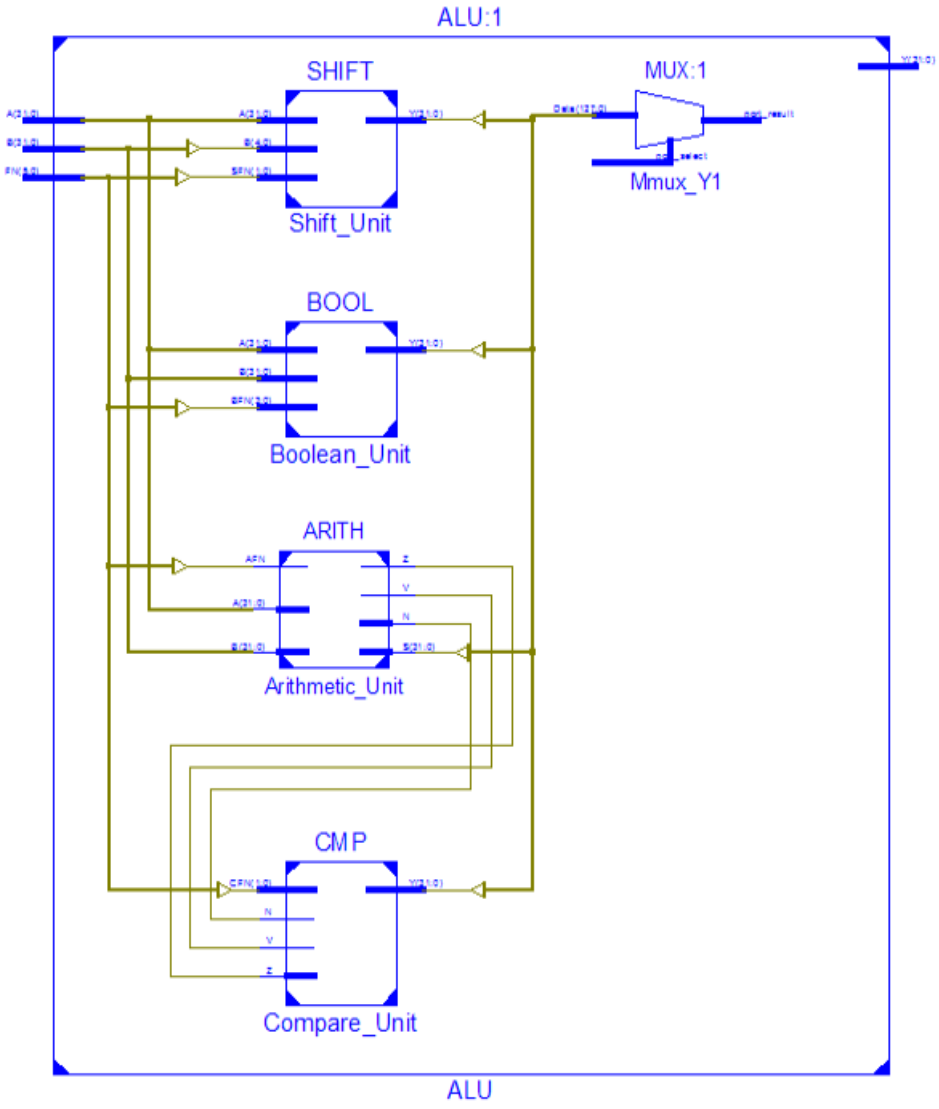
Figura 137. **Diagrama RTL de alto nivel ALU**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

En la figura 138 se muestra el diagrama interno del módulo ALU, en ella se observa la interconexión de los módulos SHIFT, BOOL, ARITH y CMP, cuyas salidas se conectan a un multiplexor de 4 entradas, nótese la estructura similar a la presentada en el capítulo 3.

Figura 138. **Diagrama RTL del módulo ALU**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.3.8.2. Simulación

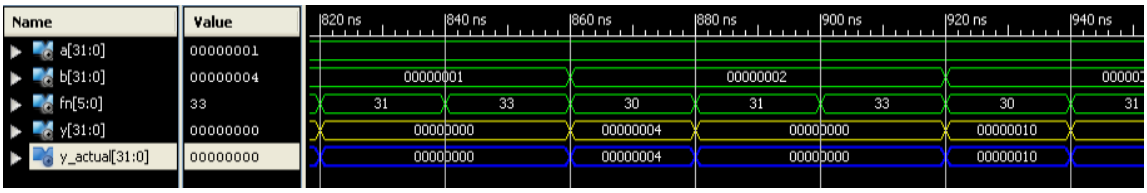
El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo ALU.vhd recibe el nombre de *ALU_tb.vhd* dentro del

directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ALU_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *alu_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/alu_tests.csv.

En la figura 139 se muestra un pequeño fragmento de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores.

Figura 139. **Simulación del módulo ALU**



Fuente: elaboración propia, empleando ISim Simulator.

4.4. Lógica de control, CTL

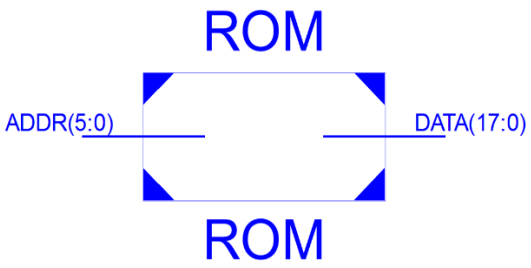
El módulo CTL se implementa jerárquicamente, siguiendo un enfoque ascendente y con base en las especificaciones del capítulo 3. Primero se implementa la ROM de 64x18 y luego se añade la lógica de control correspondiente.

4.4.1. ROM de 64x18

El archivo que contiene el código VHDL fuente de la implementación del módulo de la ROM de 64×18 recibe el nombre de *ROM.vhd* dentro del directorio del proyecto. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/ROM.vhd.

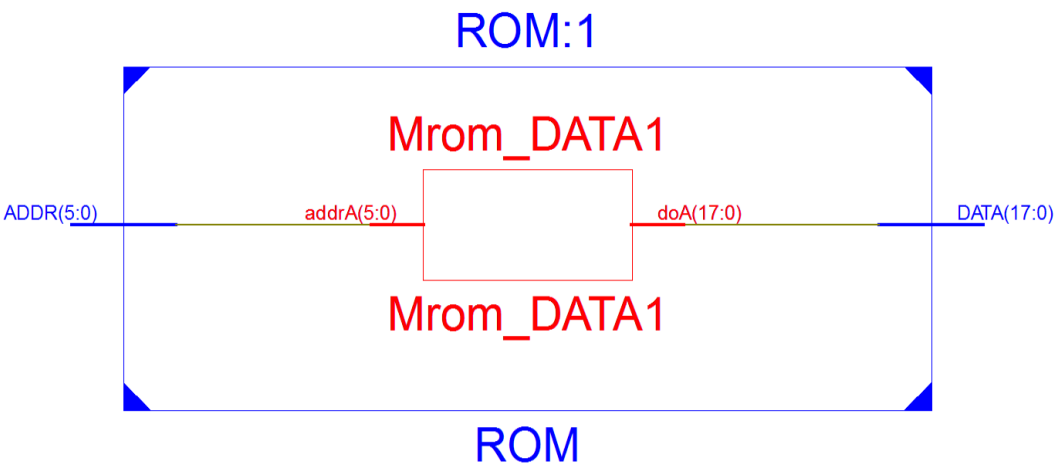
En las figuras 140 y 141 se muestran los diagramas RTL de alto nivel e interno respectivamente, ambos producidos por la síntesis del código fuente.

Figura 140. **Diagrama RTL de alto nivel ROM**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 141. **Diagrama RTL del módulo ROM**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

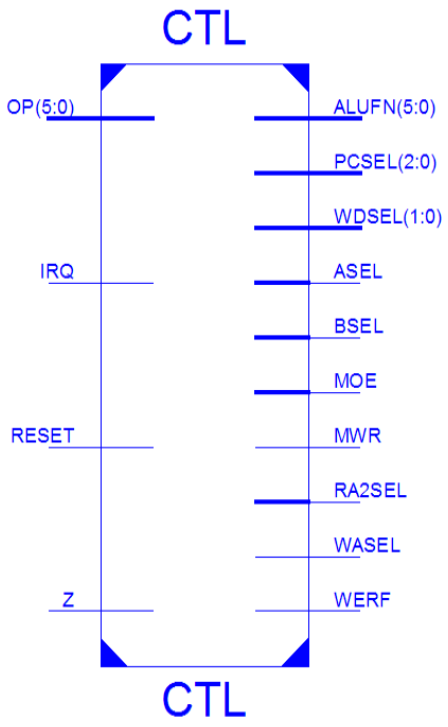
4.4.2. Diseño general

El archivo que contiene el código VHDL fuente de la implementación del diseño general del módulo CTL recibe el nombre de *CTL.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/CTL.vhd.

4.4.2.1. Diagramas esquemáticos RTL

En la figura 142 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

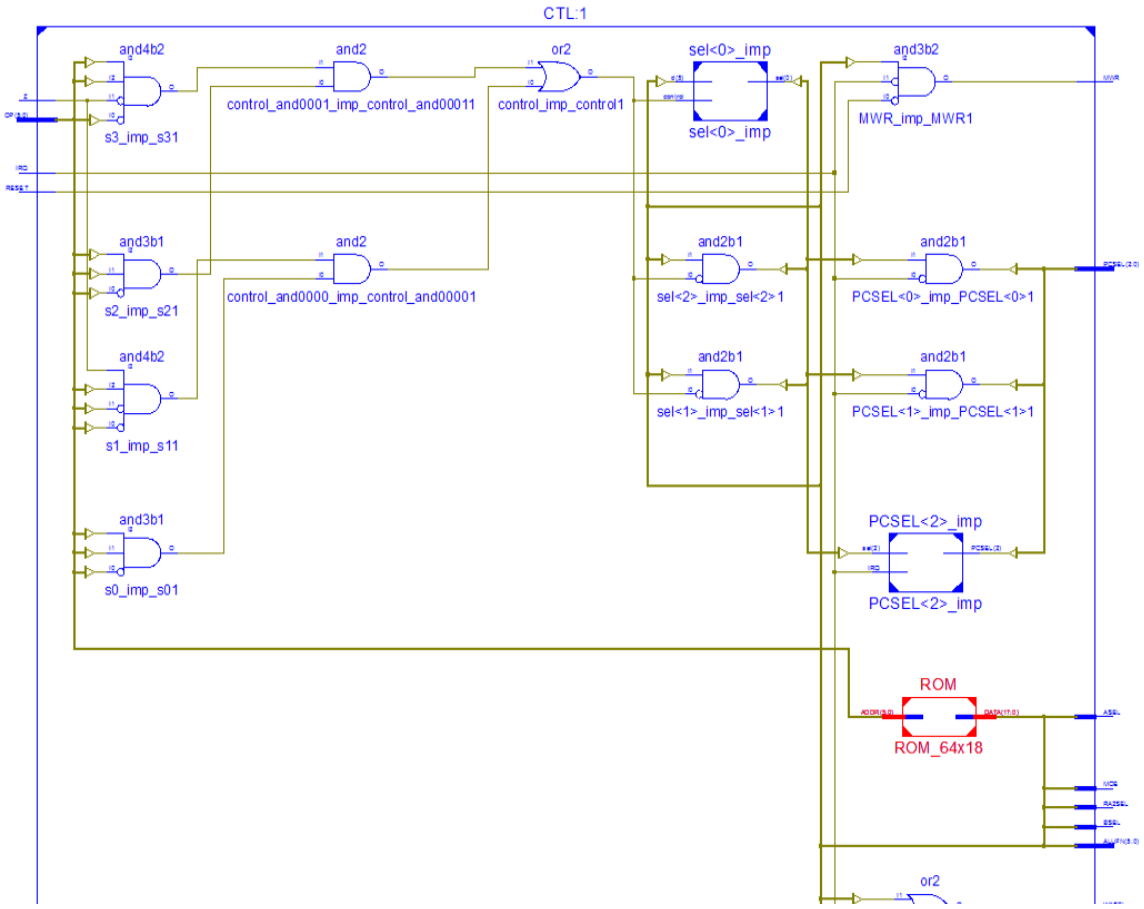
Figura 142. **Diagrama RTL de alto nivel CTL**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

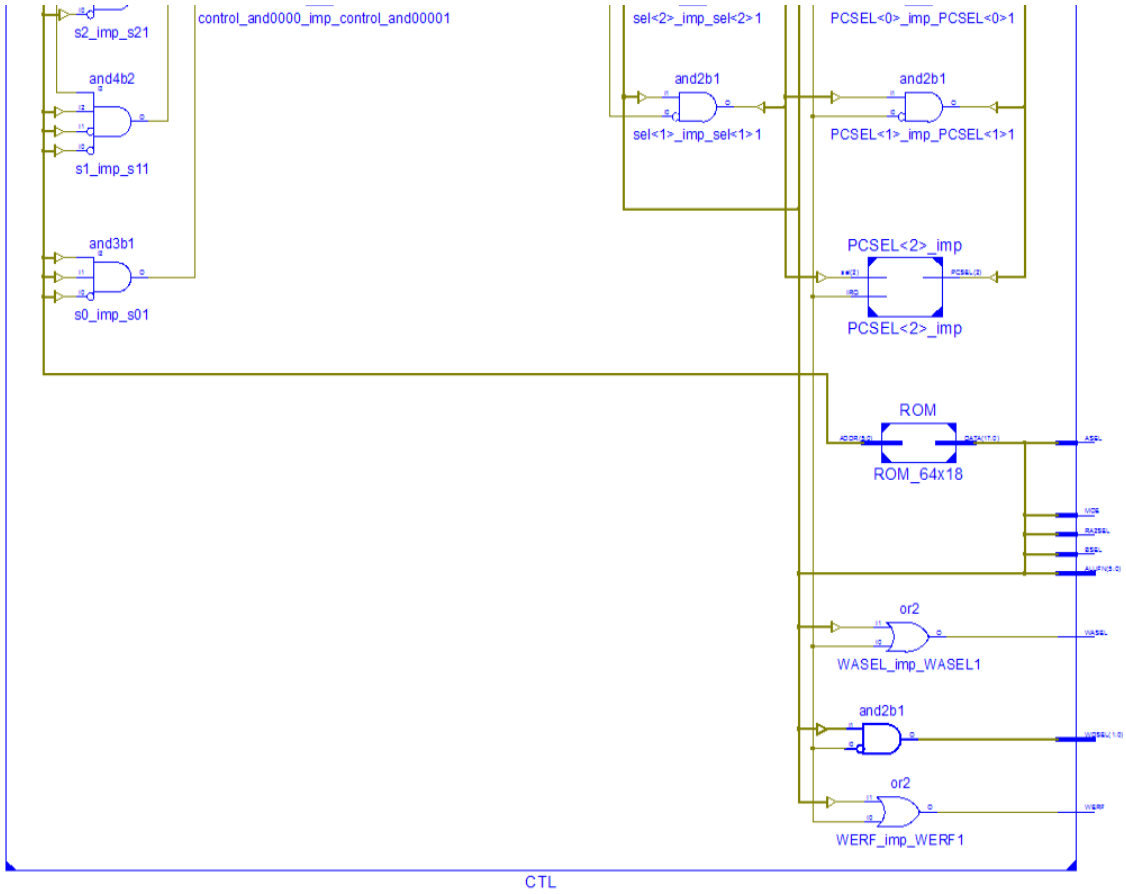
En las figuras 143 y 144 se muestra el diagrama RTL del diseño general del módulo CTL, en ellas se puede observar el componente ROM interconectado con el hardware adicional que implementa la funcionalidad descrita en el código fuente.

Figura 143. **Diagrama RTL superior del módulo CTL**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 144. **Diagrama RTL inferior del módulo CTL**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

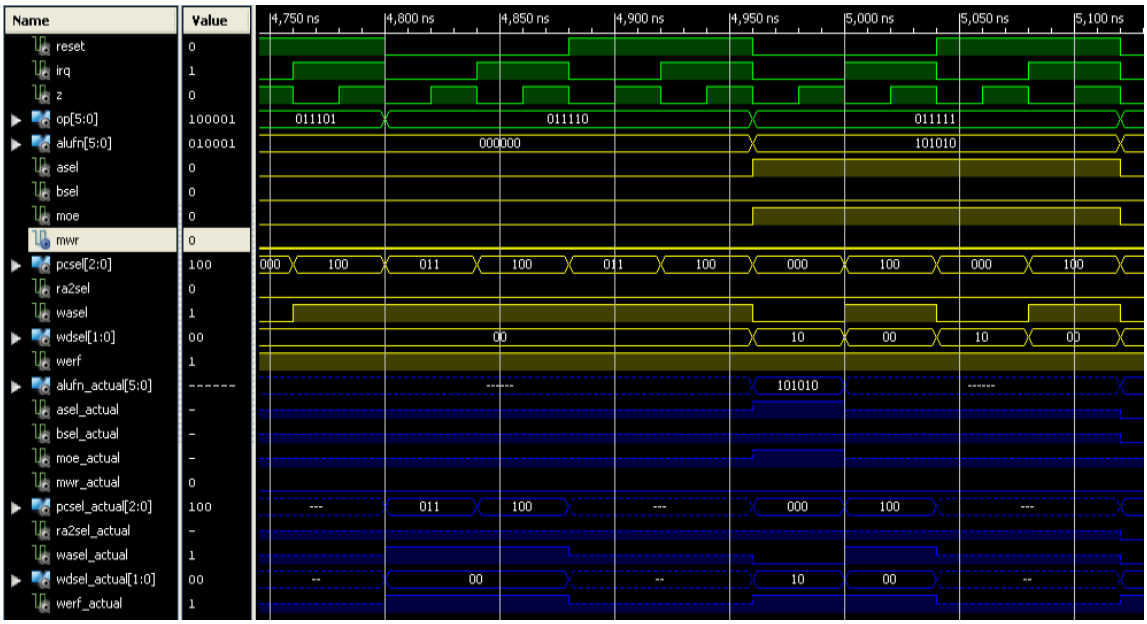
4.4.2.2. Simulación

El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo CTL.vhd recibe el nombre de *CTL_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/CTL_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *ctl_tests.csv* dentro de la carpeta llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/ctl_tests.csv.

En la figura 145 se muestra un pequeño fragmento de los resultados de la simulación del módulo siguiendo el esquema de colores utilizado en las secciones anteriores. Nótese que, para algunas combinaciones de entradas, los valores esperados en algunas salidas son irrelevantes, esto se denota mediante la concatenación de varios caracteres “-”, dependiendo del ancho de la señal de interés.

Figura 145. **Simulación del módulo CTL**



Fuente: elaboración propia, empleando ISim Simulator.

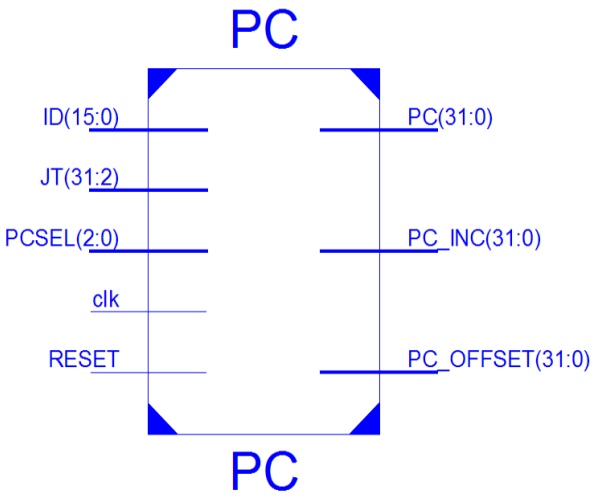
4.5. Contador de programa, PC

El archivo que contiene el código VHDL fuente de la implementación del diseño del módulo PC recibe el nombre de *PC.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/PC.vhd.

4.5.1. Diagramas esquemáticos RTL

En la figura 146 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

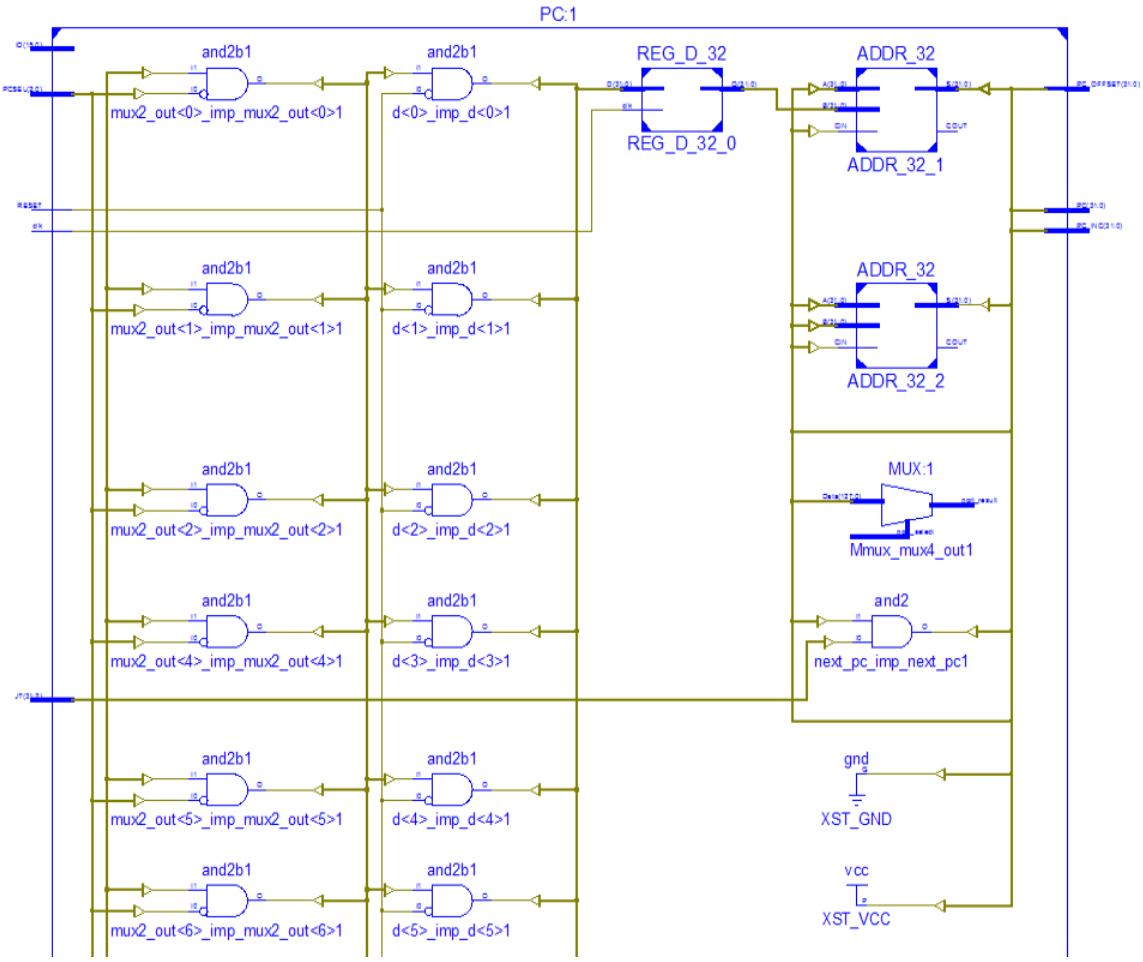
Figura 146. **Diagrama RTL de alto nivel PC**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

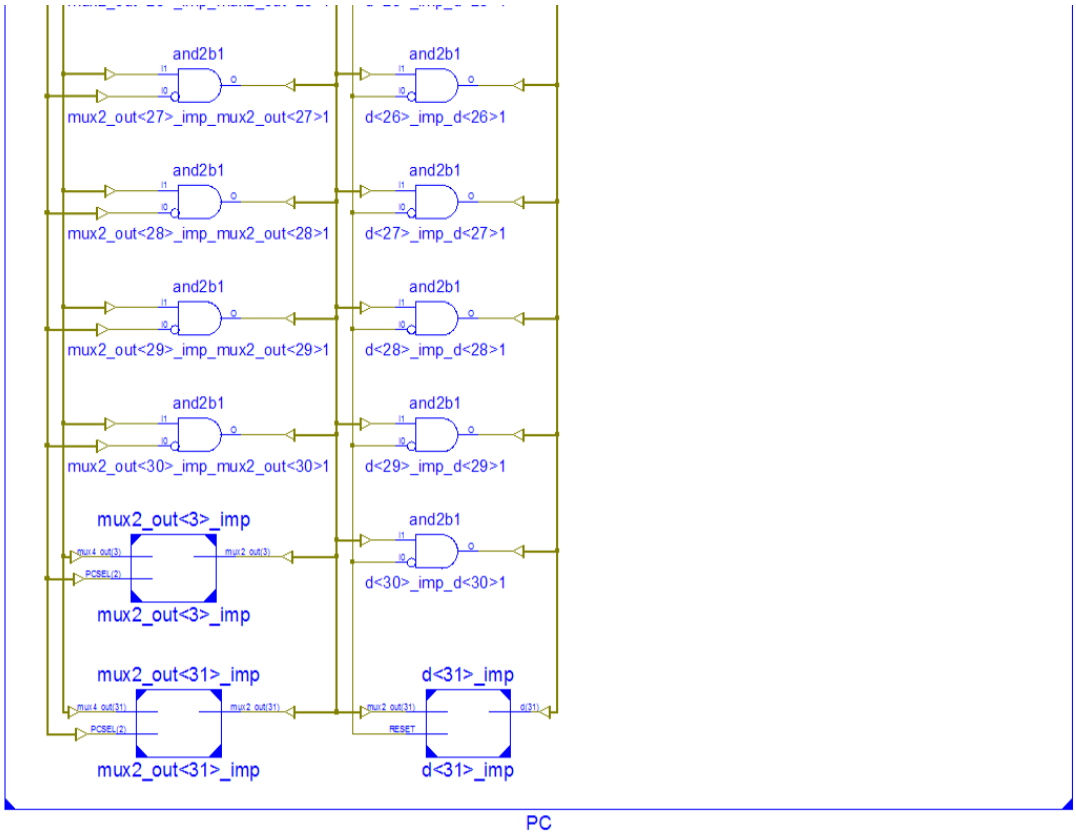
En las figuras 147 y 148 se muestra el diagrama RTL del diseño del módulo PC, en ellas se pueden apreciar los dos módulos sumadores de 32 bits y el registro de 32 bits.

Figura 147. **Diagrama RTL superior del módulo PC**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 148. **Diagrama RTL inferior del módulo PC**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

4.5.2. Simulación

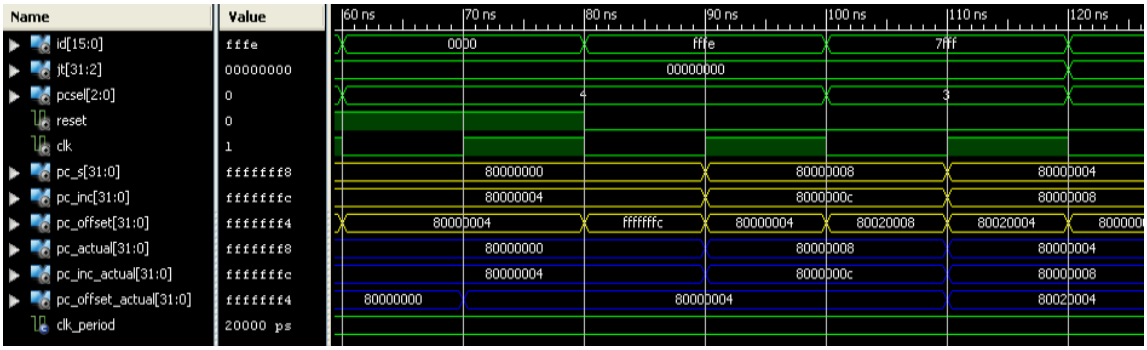
El archivo que contiene el código fuente VHDL del testbench utilizado para la simulación del módulo PC.vhd recibe el nombre de *PC_tb.vhd* dentro del directorio del proyecto y se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/PC_tb.vhd.

El archivo separado por comas que contiene las combinaciones de entrada y salidas esperadas recibe el nombre de *pc_tests.csv* dentro de la carpeta

llamada *Test_files*. Este archivo se encuentra disponible en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/Test_files/pc_tests.csv.

En la figura 149 se muestra un fragmento de los resultados de la simulación del módulo en el que se sigue el mismo esquema de colores para entradas y salidas.

Figura 149. **Simulación del módulo PC**



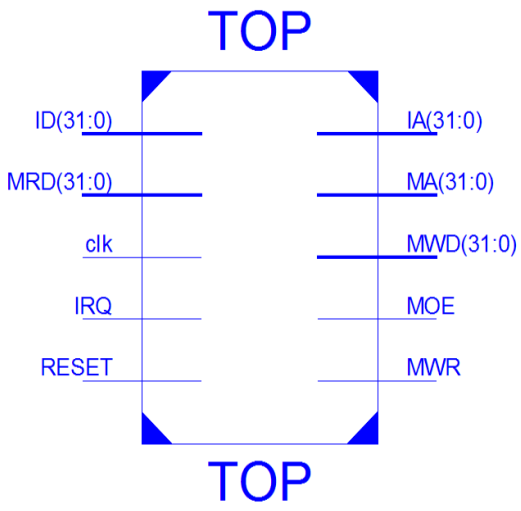
Fuente: elaboración propia, empleando ISim Simulator.

4.6. Módulo TOP

El archivo que contiene el código VHDL fuente de la implementación del diseño general del módulo TOP recibe el nombre de *TOP.vhd* dentro del directorio del proyecto. Dicho archivo se encuentra disponible para su consulta en https://github.com/lemus96/Procesador_RISC_32bits/blob/master/TOP.vhd.

En la figura 150 se muestra el diagrama RTL de alto nivel producido por la síntesis del código fuente.

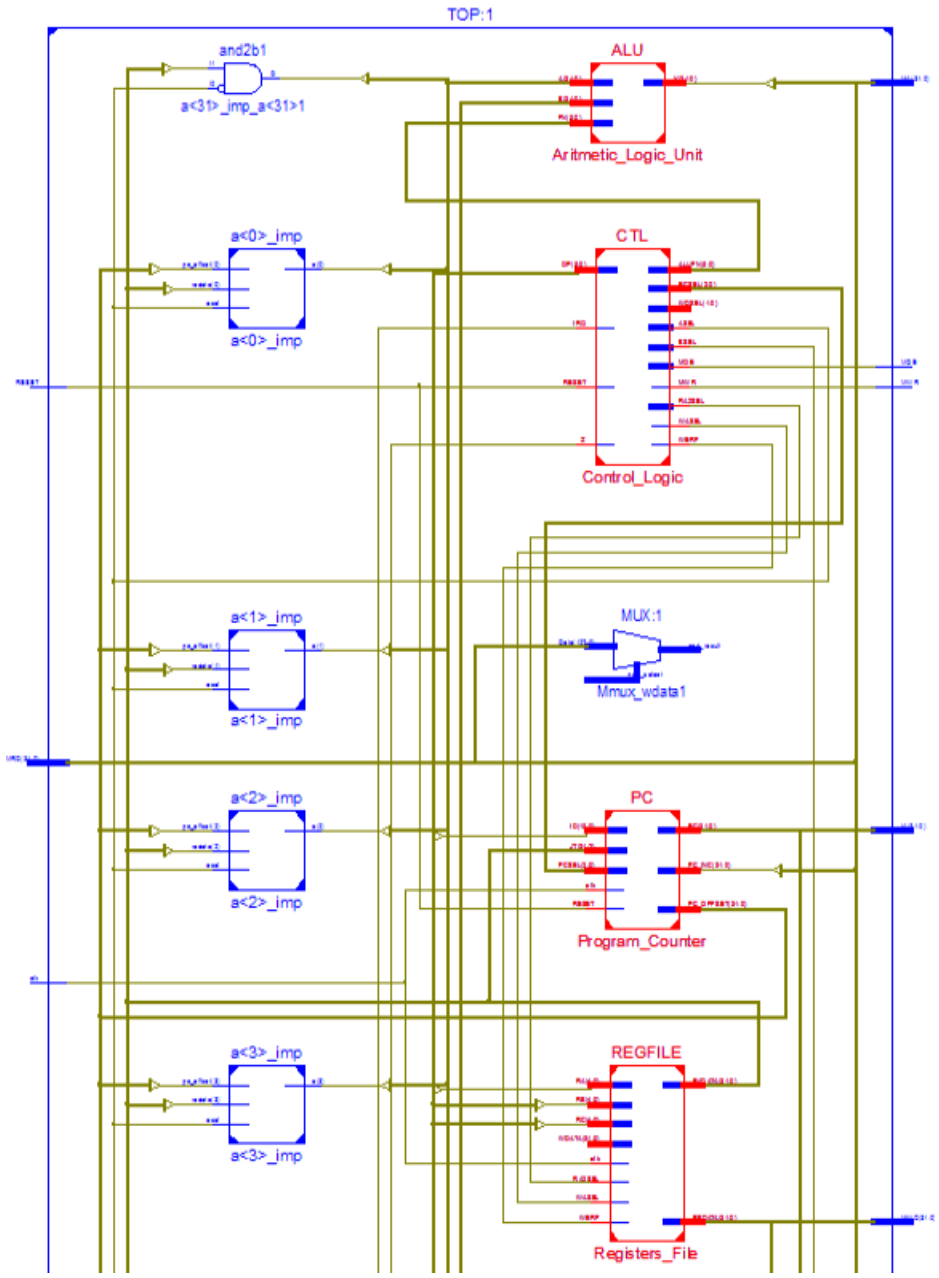
Figura 150. **Diagrama RTL de alto nivel del módulo TOP**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

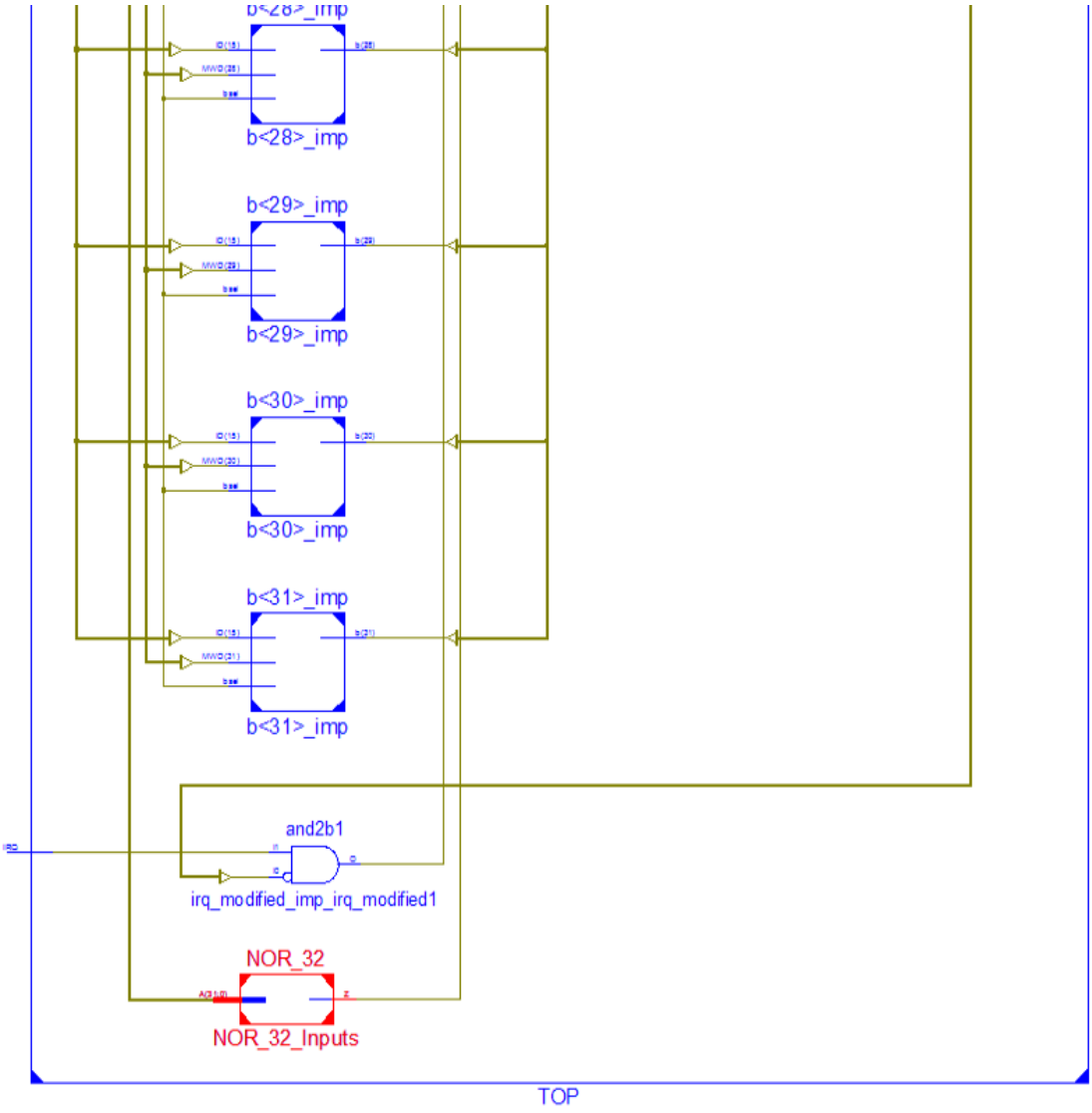
En las figuras 151 y 152 se muestra el diagrama interno del módulo, en ellas se pueden identificar los módulos de las secciones anteriores, como parte del circuito principal que conforma el microprocesador.

Figura 151. **Diagrama RTL superior del módulo TOP**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

Figura 152. **Diagrama RTL inferior del módulo TOP**



Fuente: elaboración propia, empleando ISE Design Suite 14.6.

CONCLUSIONES

1. El conjunto de instrucciones Beta o ISA Beta, es un ejemplo sencillo pero representativo de una arquitectura RISC de 32 bits que permite utilizar estrategias de implementación de hardware basadas en el modelo von Neumann.

2. El diseño del procesador permite ejecutar toda la funcionalidad del ISA Beta utilizando una cantidad relativamente moderada de hardware como consecuencia de la simplicidad del esquema de codificación de instrucciones elegido.

3. La implementación del ISA Beta tiene la capacidad de ejecutar únicamente una instrucción por cada ciclo de reloj.

4. Se describió la funcionalidad del diseño siguiendo un enfoque modular y utilizando VHDL, dentro del entorno de desarrollo ISE Design Suite 14.6.

5. Se realizó la síntesis del código VHDL para la FPGA Spartan 3E-1200 de Xilinx, utilizando el sintetizador configurado para la optimización de área.

6. Se simuló y verificó el correcto funcionamiento de los módulos de mayor relevancia del diseño utilizando diferentes testbench, codificados en VHDL, y el simulador ISim Simulator, contenido dentro del entorno de desarrollo ISE Design Suite 14.6.

RECOMENDACIONES

1. Se debe recordar que, debido a la complejidad del lenguaje VHDL, no es posible realizar una exposición completa del mismo, por lo tanto, se asume que el lector ya posee los conocimientos suficientes para comprender y modificar la implementación del diseño del procesador.

2. Debe darse especial atención a la comprensión de la arquitectura del conjunto de instrucciones y el diseño del hardware del procesador con el fin de comprender y realizar modificaciones a su implementación en VHDL.

3. Utilizar un circuito sumador dedicado que incremente el registro PC en 4 unidades para reducir la cantidad de hardware necesaria en el módulo PC. Esta idea es fácilmente realizable si se considera un sumador cuya segunda entrada es igual a 4 y luego se simplifica la lógica combinacional apropiadamente.

4. Utilizar compuertas lógicas en lugar de la ROM para producir algunas señales de control críticas del módulo CTL. El mayor beneficio de esta optimización consiste en la reducción del retardo de propagación de dichas señales. Sin embargo, no es aconsejable sustituir completamente la ROM con lógica combinacional, debido a que la FPGA realiza su propia optimización de hardware.

5. No es aconsejable utilizar arquitecturas de sumadores de alta velocidad para la ALU y el sumador de offset del módulo PC, debido a que las FPGA

poseen lógica dedicada de acarreo para mejorar el rendimiento de los sumadores. Por lo tanto, es posible que las mejoras en el diseño de los sumadores no sean notables durante de su implementación.

6. Reorganizar el hardware para la etapa de búsqueda de la instrucción, con el objetivo de que el procesador sea capaz de realizar ambos accesos de memoria en paralelo.

7. Es aconsejable que, si se desea aumentar el rendimiento del procesador y así reducir el período de reloj mínimo al que puede operar el procesador, se implemente un pipeline de 2 etapas, con una etapa para la búsqueda de la instrucción y la otra para el resto de operaciones necesarias para su ejecución.

BIBLIOGRAFÍA

1. AGARWAL, Anant; LANG, Jeffrey. *Foundations of analog and digital electronic circuits*. Estados Unidos: Elsevier, 2005. 1009 p.

2. AGUIRRE, Guillermo. *Los principales componentes de un procesador RISC como circuitos simulados*. Argentina: Universidad Nacional de San Luis, 2004. 3 p.

3. DIGILENT. *Nexys-2 Board Reference Manual*. Estados Unidos: Digilent, 2012. 17 p.

4. GOLZE, Ulrich. *VLSI Chip Design with the Hardware Description Language VERILOG: An Introduction Based on a Large RISC Processor Design*. Alemania: Springer, 2013. 347 p.

5. HAUCK, Scott; DEHON, Andre. *Reconfigurable computing: The theory and practice of FPGA-based computation*. Estados Unidos: Elsevier, 2010. 945 p.

6. LUNA VEGA, José Ignacio. et al. *Arquitectura RISC vs CISC*. México: Universidad Autónoma Metropolitana, 2020. 9 p.

7. MAXINEZ, David. *Programación de sistemas digitales con VHDL*. México: Grupo Editorial Patria, 2014. 381 p.

8. MANO, Morris. *Arquitectura de computadoras*. México: Pearson Educación, 1994. 641 p.

9. _____. *Diseño digital*. México: Pearson Educación, 2003. 521 p.

10. NAVABI, Zainalabedin. *VHDL: Analysis and modeling of digital systems*. Estados Unidos: McGraw-Hill, Inc., 1997. 656 p.

11. PARDO, Fernando. VHDL: *Lenguaje para síntesis y modelado de circuitos*. México: Alfaomega, 2000. 310 p.

12. QUIROGA, Patricia. *Arquitectura de computadoras*. Argentina: Alfaomega, 2010. 380 p.

13. SANCHEZ, Marcos. *Introducción a la programación en VHDL*. España: Universidad Complutense de Madrid, 2014. 76 p.

14. SANCHÍS, Enrique. *Sistemas electrónicos digitales: Fundamentos y diseño de aplicaciones*. España: Universitat de València, 2002. 516 p.

15. TERMAN, Chris. *6.004 Computation Structures, Spring 2017 Massachusetts Institute of Technology: MIT OpenCourseWare*. <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/#>. [Consulta: 15 de abril de 2020].

16. VON NEUMANN, John. *First Draft of a Report on the EDVAC*. Estados Unidos: IEEE Annals of the History of Computing, 1993. 49 p.

17. WARD, Stephen A.; HALSTEAD, Robert H. *Computation structures*. Inglaterra: MIT press, 1990. 812 p.

18. XILINX. *Xilinx spartan-3e fpga family: Data sheet*. Estados Unidos: Xilinx, 2009. 227 p.