



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

**DISEÑO E IMPLEMENTACIÓN DE UN SOC EN UN FPGA BASADO EN EL
ISA DE RISC-V**

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

GIANLUCA VINCENZO D'AGOSTINO MATUTE

PROFESOR GUÍA:
FRANCISCO RIVERA SERRANO

MIEMBROS DE LA COMISIÓN:
ANDRÉS CABA RUTTE
ÁLVARO SILVA MADRID

SANTIAGO, CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO
POR: GIANLUCA VINCENZO D'AGOSTINO MATUTE
FECHA: 2021
PROF. GUÍA: FRANCISCO RIVERA SERRANO

DISEÑO E IMPLEMENTACIÓN DE UN SOC EN UN FPGA BASADO EN EL ISA DE RISC-V

En este trabajo de título se presenta el problema de desarrollo, diseño e implementación de un *System on a Chip* utilizando el juego de instrucciones libre de *RISC-V*, el cual se espera que tenga un gran impacto en el desarrollo tecnológico a nivel mundial, al facilitar el diseño de dispositivos digitales a nuevas empresas por no requerir el pago de licencias de uso, ser altamente escalable y de diseño modular. El *System on a Chip* se implementa en una tarjeta *FPGA* y dispone del juego de instrucciones base de 32 bits, las extensiones de multiplicación/división y de punto flotante de precisión simple de *RISC-V*. Además, el diseño obtenido tiene una orientación pedagógica, es decir que, tiene una función ilustrativa para que futuros estudiantes de diseño digital aprendan sobre el diseño e implementación de sistemas/arquitecturas digitales sobre un *FPGA*. Para el desarrollo, implementación y pruebas se utiliza *System Verilog* junto al *software Vivado* y para la etapa de verificación se utiliza *RARS*, un simulador de *RISC-V*, y la generación de *test benches* con números pseudoaleatorios.

*A las preguntas de nuestra niñez, que se convierten
en el motor de nuestras acciones en el presente.*

El deseo de saber

Agradecimientos

Todos mis agradecimientos a mi familia que ha sido mi apoyo durante toda mi formación académica, por sobre todo a mi mamá que siempre ha sido una voz de consejo y de apoyo fundamental en mi formación profesional y personal. A mi nonno que siempre me inculcó la importancia de la educación y del deseo de hacer lo que realmente quieres con tu vida, que en paz descanse. A mi abuela que ha estado físicamente presente durante todo este periodo tan importante de mi vida, pero que tristemente se ha desvanecido día a día. A mi papá que siempre que lo necesité me ayudó y estuvo para lo que necesitase. A mis profesores y en particular a mi profesor guía, quién motivó aún más mi pasión por el diseño de sistemas digitales. A mis compañeros y amigos que siempre estuvieron durante este proceso. A mi ex pareja quién me acompañó e hizo muy feliz durante 4 años de este proceso universitario.

Tabla de Contenido

1	Introducción	1
1.1	Motivación	1
1.2	Objetivos del trabajo	2
2	Marco teórico y estado del arte	4
2.1	Conceptos elementales	4
2.1.1	Sistemas digitales	4
2.1.2	Arquitectura de computadores	5
2.1.3	Representación binaria en punto flotante	10
2.2	RISC-V	11
2.2.1	Visión general al <i>ISA</i> de <i>RISC-V</i>	12
2.2.2	La Memoria en <i>RISC-V</i>	13
2.2.3	Codificación del largo de instrucciones	14
2.2.4	Excepciones, trampas e interrupciones	15
2.2.5	El <i>ISA</i> base y extensiones implementadas, <i>RV32IMF</i>	16
2.2.5.1	Instrucciones enteras, <i>RV32IM</i>	19
2.2.5.2	Unidad de punto flotante (<i>FPU</i>), <i>RV32F</i>	22
2.3	Revisión del estado del arte	25
2.3.1	<i>A RISC-V Instruction Set Processor-Micro- architecture Design and Analysis</i>	25
2.3.2	<i>Open-Source RISC-V Processor IP Cores for FPGAs – Overview and Evaluation</i>	29
2.3.3	<i>RISC-V based implementation of Programmable Logic Controller on FPGA for Industry 4.0</i>	30
2.3.4	<i>Design and Verification Environment for RISC-V Processor Cores</i>	31
2.3.5	<i>Co-verification design flow for HDL Languages</i>	31
2.3.6	Ejemplo de una implementación real: <i>HiFive Rev B</i>	32
2.4	Tarjeta <i>FPGA</i> y <i>software</i> utilizado	34
2.4.1	<i>Nexys 4</i>	34
2.4.2	<i>Vivado Design Suite</i> de <i>Xilinx</i>	35
2.4.3	<i>RARS 1.5</i>	36
3	Metodología de trabajo	37
3.1	Metodología de diseño <i>Top-Down</i>	37
3.2	Flujo de trabajo	38
3.3	Metodología de verificación	39
4	Diseño propuesto	41
4.1	<i>Instruction Decoder</i>	43
4.2	<i>Register Files</i>	45

4.3	Unidades de memoria	46
4.3.1	<i>Cache Controller</i>	47
4.4	<i>Arithmetic Logic Unit (ALU)</i>	50
4.5	<i>Floating Point Unit (FPU)</i>	51
4.5.1	<i>FP Arithmetic Unit</i>	55
4.5.1.1	<i>FP Arithmetic Unit: ADDER</i>	63
4.5.1.2	<i>FP Arithmetic Unit: MULTIPLIER</i>	64
4.5.1.3	<i>FP Arithmetic Unit: DIVISOR</i>	65
4.5.1.4	<i>FP Arithmetic Unit: significands_divisor</i>	66
4.5.1.5	<i>FP Arithmetic Unit: SQRT</i>	71
4.5.1.6	<i>FP Arithmetic Unit: significand_sqrt</i>	72
4.5.1.7	<i>FP Arithmetic Unit: Exceptions and trivial cases checker</i>	76
4.5.1.8	<i>FP Arithmetic Unit: NORMALIZER</i>	78
4.5.1.9	<i>FP Arithmetic Unit: ROUNDER</i>	81
4.5.2	<i>FP Converter Unit</i>	81
4.5.2.1	<i>FP Converter Unit: exceptions_checker</i>	85
4.5.2.2	<i>FP Converter Unit: fp_to_integer_unit</i>	87
4.5.2.3	<i>FP Converter Unit: Integer rounder</i>	87
4.5.2.4	<i>FP Converter Unit: integer_to_fp_unit</i>	88
4.5.2.5	<i>FP Converter Unit: NORMALIZER</i>	89
4.5.2.6	<i>FP Converter Unit: ROUNDER</i>	90
4.6	<i>Core Complex</i>	90
4.6.1	<i>Single-Cycle Version</i>	93
4.6.2	<i>Pipelined Version</i>	100
4.7	<i>System on a Chip</i>	109
4.7.1	<i>debounce module</i>	115
5	Verificación, pruebas e implementación final	116
5.1	Etapas de verificación	116
5.1.1	Testeo de: <i>memory.sv</i>	116
5.1.2	Testeo de: <i>ALU.sv</i>	118
5.1.3	Testeo de: <i>fp_converter.sv</i>	120
5.1.4	Testeo de: <i>fp_arithmetic_unit.sv</i>	124
5.1.5	Testeo de: <i>FPU.sv</i>	128
5.1.6	Testeo de: <i>riscv32imf_singlecycle.sv</i> y <i>riscv32imf_pipeline.sv</i>	128
5.1.7	Testeo de: <i>TOP.sv</i>	132
5.2	Pruebas en <i>hardware</i>	133
5.2.1	<i>fibonacci_inputs.s</i>	133
5.2.2	<i>operating_fp.s</i>	135
5.3	Detalles de la implementación	138
6	Conclusiones	141

Bibliografía **143**

Anexos **146**

1	Código fuente del <i>SoC</i>	146
2	<i>Test Benches</i>	228
3	Códigos generadores de <i>Test Benches</i>	242

4	Códigos <i>Assembler</i>	261
4.1	<i>Script</i> conversor de palabras	267
5	Manual de uso del <i>SoC</i>	269
5.1	Configurando el proyecto en <i>Vivado</i>	269
5.2	Compilación de programas para el <i>SoC</i>	278

Índice de Tablas

2.1.	Representación binaria en punto flotante para un número X . Fuente: [13]. . .	10
2.2.	Casos especiales para la representación en punto flotante de precisión simple según la <i>IEEE 754</i> . Fuente: [13]	10
2.3.	Bits de redondeo. Fuente: [12]	11
2.4.	Características de las <i>trampas</i> . Notas: 1) el término puede ser solicitado, 2) trampas fatales imprecisas pueden ser observadas por <i>software</i> . Fuente: [8]. .	16
2.5.	Codificación de los distintos tipos de instrucciones consideradas. Fuente: [8]. .	19
2.6.	Instrucciones implementadas de <i>RV32I</i> . Fuente: [8].	20
2.7.	Instrucciones implementadas de <i>RV32M</i> . Fuente: [8].	22
2.8.	Modos de redondeo (<i>rm</i>) de <i>RV32F</i> . Fuente: [8].	23
2.9.	Instrucciones implementadas de <i>RV32F</i> . Fuente: [8].	24
2.10.	Posibles formatos para <i>fclass.s</i> . Fuente: [8].	25
4.1.	Codificación para <i>format_type</i> según el tipo de la instrucción.	43
4.2.	Codificación para <i>sub_format_type</i> según el subtipo de la instrucción.	44
4.3.	Codificación para <i>funct7_out</i> según el <i>funct7</i> de la instrucción.	44
4.4.	Codificación para <i>reg_access_option</i> según el acceso a registros de la instrucción. <i>I</i> indica acceso a registros enteros y <i>FP</i> a registros flotantes. Nota: <i>rs3</i> siempre es flotante.	45
4.5.	Codificación de <i>alu_option</i> para cada tipo de operación de la <i>ALU</i>	51
4.6.	Codificación de <i>fpu_option</i> para cada operación de la <i>FPU</i>	52
4.7.	Resumen del criterio de redondeo utilizado según el <i>rounding mode (RM)</i> , el bit de signo (<i>sign</i>) y los bits <i>guard (g)</i> , <i>round (r)</i> y <i>sticky (s)</i> . 1 indica que se redondea el resultado (sumar uno a <i>fraction</i>) y 0 que se trunca (no se modifica <i>fraction</i>). Nota: <i>lsb=fraction[0]</i>	55
4.8.	Codificación de la entrada <i>op</i> del módulo <i>fp_arithmetic_unit</i>	55
4.9.	Codificación de la salida <i>status</i> del módulo <i>fp_arithmetic_unit</i>	56
4.10.	<i>Lookup-Table</i> propuesto por [27].	74
4.11.	Codificación de la salida <i>sel_out</i> del módulo <i>excep_triv_checker</i> y del <i>FSM</i> del módulo <i>fp_arithmetic_unit</i> (ver figura 4.9).	76
4.12.	Codificación de la salida <i>status</i> del módulo <i>NORMALIZER</i>	78
4.13.	Codificación de la salida <i>exit_status</i> del <i>core complex</i> . Nota: <i>program/data error</i> ocurren cuando se trata de acceder a una dirección de memoria fuera del rango de memoria de la <i>BRAM</i> . Además, solo en el caso <i>Single-Cycle</i> las instrucciones no válidas generan <i>program error</i> , mientras que en el caso <i>Pipelined</i> estas se tratan como <i>nops</i>	92

- 4.14. Cuando solo se menciona un color quiere decir que ambos *LED* tricolor tienen el mismo color, en caso contrario el *LED* derecho se corresponde con el color de la derecha y el izquierdo con el color de la izquierda. Cuando ocurre un *get** solo uno de los *LED* se enciende, según la mitad de la palabra que se esté ingresando. 110

Índice de Ilustraciones

2.1.	Ejemplo de <i>datapath single-cycle</i> . Fuente: [12].	6
2.2.	Ejemplo de <i>datapath pipelined</i> . Fuente: [12].	7
2.3.	<i>Endianness</i>	8
2.4.	Codificación de longitud de instrucción <i>RISC-V</i> . Solo las codificaciones de 16 bits y 32 bits se consideran “congeladas” en este momento. Fuente: [8].	15
2.5.	<i>Register Files</i> para <i>RV32I unprivileged</i> . Fuente: [8].	17
2.6.	<i>Register Files</i> para <i>RV32F unprivileged</i> . Fuente: [8].	18
2.7.	Arquitectura de nivel superior del procesador <i>RISC-V</i> . Fuente: [15].	26
2.8.	microarquitectura de la <i>fetch unit</i> . Fuente: [15].	27
2.9.	Diagrama del decodificador de instrucciones. Fuente: [15].	27
2.10.	Microarquitectura del decodificador de instrucciones. Fuente: [15].	28
2.11.	Organización del <i>Register File</i> . Fuente: [15].	28
2.12.	Número de menciones en las publicaciones de diferentes <i>CPU IP cores</i> . Fuente: [16].	30
2.13.	La estructura de la arquitectura PLC propuesta. Fuente: [17].	31
2.14.	Proceso de desarrollo típico. Fuente: [19].	32
2.15.	Proceso de desarrollo de co-verificación. Fuente: [19].	32
2.16.	Tarjeta <i>HiFive1 Rev B</i> . Fuente: [20].	32
2.17.	<i>Pinout</i> de la tarjeta <i>HiFive1 Rev B</i> . Fuente: [20].	33
2.18.	Diagrama de bloques de nivel superior del chip <i>SiFive FE310-G002</i> . Fuente: [21].	33
2.19.	Tarjeta utilizada, la <i>Nexys 4</i> . Fuente: [23].	35
3.1.	Enfoque Diseño Contemporáneo (<i>Top-Down</i>). Fuente: [10].	37
3.2.	Diagrama del flujo de trabajo empleado.	38
4.1.	Diagrama de bloques de nivel superior propuesto en primera instancia para el <i>SoC</i>	41
4.2.	Diagrama de bloques simplificado <i>core complex pipelined</i>	42
4.3.	Diagrama de bloques de nivel superior (simplificado) propuesta final para el <i>SoC</i>	42
4.4.	Diagrama de bloques detallados del <i>Register Files</i>	45
4.5.	Diagrama de bloques de la memoria cache implementada. Fuente: [12].	48
4.6.	<i>FSM</i> del <i>Cache Controller</i> implementado. Fuente: [12].	49
4.7.	Diagrama <i>MDS</i> del <i>FSM</i> de la <i>FPU</i>	53
4.8.	Diagrama de flujo simplificado del módulo <i>fp_arithmetic_unit</i>	57
4.9.	Diagrama de bloques detallado del módulo <i>fp_arithmetic_unit</i>	58
4.10.	Diagrama de flujo detallado del módulo <i>fp_arithmetic_unit</i>	61
4.11.	Diagrama <i>MDS</i> del módulo <i>fp_arithmetic_unit</i>	62
4.12.	Diagrama de flujo simplificado del submódulo <i>ADDER</i>	63

4.13.	Diagrama de bloques detallado del submódulo <i>ADDER</i>	64
4.14.	Diagrama de flujo simplificado del submódulo <i>MULTIPLIER</i>	65
4.15.	Diagrama de bloques detallado del submódulo <i>MULTIPLIER</i>	65
4.16.	Diagrama de flujo simplificado del submódulo <i>DIVISOR</i>	66
4.17.	Diagrama de bloques detallado del submódulo <i>DIVISOR</i>	66
4.18.	Diagrama de flujo simplificado del submódulo <i>significands_divisor</i>	67
4.19.	Diagrama de bloques detallado del submódulo <i>significands_divisor</i>	68
4.20.	Diagrama de flujo detallado del submódulo <i>significands_divisor</i>	69
4.21.	Diagrama <i>MDS</i> del submódulo <i>significands_divisor</i>	70
4.22.	Diagrama de flujo simplificado del submódulo <i>SQRT</i>	71
4.23.	Diagrama de bloques detallado del submódulo <i>SQRT</i>	72
4.24.	Diagrama de flujo simplificado del submódulo <i>significand_sqrt</i>	73
4.25.	Diagrama de bloques detallado del submódulo <i>significand_sqrt</i>	74
4.26.	Diagrama de flujo detallado del submódulo <i>significand_sqrt</i>	75
4.27.	Diagrama <i>MDS</i> del submódulo <i>significand_sqrt</i>	75
4.28.	Diagrama de flujo del submódulo <i>excep_triv_checker</i>	77
4.29.	Diagrama de flujo simplificado del submódulo <i>NORMALIZER</i>	78
4.30.	Diagrama de bloques detallado del submódulo <i>NORMALIZER</i>	79
4.31.	Diagrama de flujo detallado del submódulo <i>NORMALIZER</i>	80
4.32.	Diagrama <i>MDS</i> del submódulo <i>NORMALIZER</i>	80
4.33.	Diagrama de bloques detallado del submódulo <i>ROUNDER</i>	81
4.34.	Diagramas de flujo simplificado del módulo <i>fp_converter</i>	82
4.35.	Diagrama de bloques detallado del módulo <i>fp_converter</i>	83
4.36.	Diagrama <i>MDS</i> del módulo <i>fp_converter</i>	84
4.37.	Diagrama de flujo del submódulo <i>converter_exceptions_checker</i> del módulo <i>fp_converter</i>	86
4.38.	Diagrama de bloques detallado del submódulo <i>fp_to_integer_unit</i> del módulo <i>fp_converter</i>	87
4.39.	Diagrama de bloques detallado del submódulo <i>integer_to_fp_unit</i> del módulo <i>fp_converter</i>	88
4.40.	Diagrama de bloques detallado del submódulo <i>normalizer_integer2fp</i> del módulo <i>fp_converter</i>	89
4.41.	Diagrama <i>MDS</i> del submódulo <i>normalizer_integer2fp</i> del módulo <i>fp_converter</i>	89
4.42.	Diagrama de flujo simplificado para el <i>core complex Single-Cycle</i>	93
4.43.	Diagrama de bloques detallado para el <i>core complex Single-Cycle</i> parte 1. . .	94
4.44.	Diagrama de bloques detallado para el <i>core complex Single-Cycle</i> parte 2. . .	95
4.45.	Diagrama de bloques detallado para el <i>core complex Single-Cycle</i> parte 3. . .	96
4.46.	Diagrama de bloques detallado para el <i>core complex Single-Cycle</i> parte 4. . .	97
4.47.	Diagrama <i>MDS</i> para el <i>core complex Single-Cycle</i>	99
4.48.	Diagrama de flujo simplificado para el <i>core complex Pipelined</i>	100
4.49.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 1. . .	101
4.50.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 2. . .	102
4.51.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 3. . .	103
4.52.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 4. . .	104
4.53.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 5. . .	105
4.54.	Diagrama de bloques detallado para el <i>core complex Pipelined</i> parte 6. . .	106

4.55.	Diagrama <i>MDS</i> para el <i>core complex Pipelined</i>	109
4.56.	Diagrama de flujo simplificado para el <i>SoC</i>	111
4.57.	Diagrama de bloques detallado para el <i>SoC</i>	112
4.58.	Diagrama <i>MDS</i> para el <i>SoC</i>	113
4.59.	Diagrama <i>MDS</i> para el <i>debounce</i>	115
5.1.	Resultados de tb_memory.sv parte 1.	117
5.2.	Resultados de tb_memory.sv parte 2.	118
5.3.	Resultados de tb_ALU.sv	119
5.4.	Resultados de tb_fp_converter.sv parte 1.	121
5.5.	Resultados de tb_fp_converter.sv parte 2.	122
5.6.	Resultados de tb_fp_converter.sv parte 3.	123
5.7.	Resultados de tb_fp_arithmetic_unit.sv parte 1.	125
5.8.	Resultados de tb_fp_arithmetic_unit.sv parte 2.	126
5.9.	Resultados de tb_fp_arithmetic_unit.sv parte 3.	127
5.10.	Resultados de tb_fp_arithmetic_unit.sv parte 4.	128
5.11.	Ejemplo del resultados obtenido al ejecutar tb_FPU.sv	128
5.12.	Estado final de los registros para la simulación de testing_code_imf.s en <i>RARS</i>	129
5.13.	Estado final de los registros para la simulación de testing_code_imf.s en <i>Vivado</i> . Captura de <i>registers.mem</i>	130
5.14.	Memoria de datos resultante para la simulación de testing_code_imf.s en <i>RARS</i>	131
5.15.	Memoria de datos resultante para la simulación de testing_code_imf.s en <i>Vivado</i> . Captura de <i>data_out.mem</i>	131
5.16.	Resultado obtenido en la consola de <i>Vivado</i> al ejecutar tb_riscv32imf_top.sv	131
5.17.	Resultado de <i>fibonacci_simple.s</i> en <i>RARS</i>	132
5.18.	Resultado de <i>fibonacci_simple.s</i> para la señal <i>info_out[31:0]</i> al ejecutar <i>tb_TOP.sv</i> en <i>Vivado</i>	132
5.19.	Resultado de <i>fibonacci_simple.s</i> en la consola de <i>Vivado</i> al ejecutar <i>tb_TOP.sv</i>	132
5.20.	Resultado de <i>fibonacci_inputs.s</i> en <i>RARS</i>	133
5.21.	Resultado de <i>fibonacci_inputs.s</i> en la <i>Nexys 4</i>	134
5.22.	Resultado de <i>operating_fp.s</i> en <i>RARS</i>	135
5.23.	Estado final de los registros de punto flotante para la simulación de operating_fp.s en <i>RARS</i>	136
5.24.	Representación binaria para las entradas 5.7 y 91.26 del programa <i>operating_fp.s</i> . Fuente: https://www.h-schmidt.net/FloatConverter/IEEE754.html	136
5.25.	Resultado de <i>operating_fp.s</i> en la <i>Nexys 4</i>	137
5.26.	Resumen de los recursos de <i>hardware</i> utilizados, <i>post-synthesis</i> y <i>post-implementation</i> , de la tarjeta <i>FPGA Nexys 4</i> en la implementación.	139
5.27.	Resumen del consumo de energía de la implementación realizada.	139
5.28.	Resumen del reloj implementado.	140
5.29.	Resumen de las características del <i>timing</i> del diseño implementado.	140
.1.	Configurando el nombre y directorio del nuevo proyecto de <i>Vivado</i>	269
.2.	Selección del tipo de proyecto en <i>Vivado</i>	270

.3.	Ventana para agregar el código fuente del proyecto de <i>Vivado</i>	271
.4.	Selección de los archivos a importar cómo código fuente del proyecto en <i>Vivado</i>	271
.5.	Confirmación de los archivos fuente importados al proyecto de <i>Vivado</i>	272
.6.	Ventana para agregar las <i>constraints</i> del proyecto de <i>Vivado</i>	273
.7.	Selección del archivo con las <i>constraints</i> a importar al proyecto de <i>Vivado</i>	273
.8.	Confirmación de las <i>constraints</i> importadas al proyecto de <i>Vivado</i>	274
.9.	Selección de la tarjeta <i>FPGA</i> , <i>Nexys 4</i> , para el proyecto de <i>Vivado</i>	275
.10.	Última ventana, que presenta un resumen de las configuraciones realizadas, de la creación del proyecto en <i>Vivado</i>	276
.11.	Ejecución del comando <i>set_param pwropt.maxFaninFanoutToNetRatio 1000</i> en <i>Vivado</i>	276
.12.	Visión de <i>Vivado</i> con el proyecto configurado.	277
.13.	Visión de los módulos del proyecto en <i>Vivado</i>	277
.14.	Apertura de “ <i>fibonacci_inputs.s</i> ” en <i>RARS</i>	278
.15.	Visión de “ <i>fibonacci_inputs.s</i> ” en <i>RARS</i>	278
.16.	Configuración de memoria en <i>RARS</i>	279
.17.	Compilación exitosa de “ <i>fibonacci_inputs.s</i> ” en <i>RARS</i>	279
.18.	Ventana para exportar la memoria de programa y de datos.	280
.19.	Selección del directorio y del nombre del archivo con la memoria de programa.	280
.20.	Ejecución de <i>singleWord2fourWords.py</i>	281
.21.	Se destacan los archivos <i>text_in.mem</i> y <i>data_in.mem</i> que contienen la memoria de programa y de datos, respectivamente, del <i>SoC</i>	281

1 Introducción

En este trabajo de título se presenta el proceso de investigación, diseño e implementación de un *System on a Chip* (*SoC*). Para ello se realiza la investigación bibliográfica pertinente, el desarrollo de todos los diseños y simulaciones de los sistemas digitales que conforman el *SoC* obtenido. Finalmente, se realiza la integración/síntesis final de todos los componentes en una tarjeta *FPGA*, obteniéndose un *SoC* bastante simple, pero, completamente funcional para fines pedagógicos.

Este trabajo y el *SoC* obtenido están pensados para que futuros estudiantes del área se puedan familiarizar con el proceso de diseño, la metodología de desarrollo y el prototipado de sistemas digitales complejos sobre una tarjeta *FPGA*. Por otro lado, se busca explorar las posibilidades del uso de un conjunto de instrucciones (*ISA*) libre como *RISC-V*, el cual ha ganado gran relevancia recientemente.

A continuación, se presenta la motivación del trabajo realizado y los objetivos planteados, para continuar con el marco teórico y la investigación del estado del arte, la metodología de trabajo empleada, los diseños digitales obtenidos, las pruebas realizadas e implementación obtenida y, finalmente, se presentan las conclusiones.

1.1 Motivación

En el mundo actual, cada vez más informatizado y conectado, el número de dispositivos electrónicos digitales ha crecido exponencialmente, en concordancia por lo predicho según la ley de Moore [1], la cual indica/incita el ritmo exponencial en el desarrollo de circuitos integrados (*IC*). Lo anterior, trae como consecuencia la aparición de un mayor número de dispositivos pequeños y de bajo consumo con el deber de realizar tareas simples de forma rápida y eficiente. Este fenómeno es potenciado con la implementación a nivel mundial del *Internet of things* (*IoT*), como se indica en [2]. Los sistemas *IoT* tienen un mercado en crecimiento y la complejidad de dichos sistemas, la cual es soportada a base de *SoCs* para diferentes elementos, también ha aumentado.

Para satisfacer esta necesidad, de dispositivos pequeños y de bajo consumo con tareas específicas, la solución es el desarrollo e implementación de un *SoC* que cumpla la tarea deseada [3]. Un *SoC* es, básicamente, un sistema digital basado en un único chip en el cual se integran todos los componentes necesarios, digitales y/o analógicos, para el correcto funcionamiento de dicho sistema.

Según [1], a comienzos del nuevo milenio se redefine la Ley de Moore, debido a que ya no se logra disminuir de forma considerable el tamaño de los transistores de un chip, esto provoca el desarrollo de nuevas arquitecturas en sistemas digitales, basados en *multi-cores*, diseños *3-D*, etc... Bajo este contexto, en la última década, se vuelven comunes los procesadores de filosofía *RISC* (*Reduced Instruction Set Computer*).

Los procesadores *RISC* se diferencian de los procesadores *CISC* (*Complex Instruction Set Computer*), debido a que su *ISA* (*Instruction Set Architecture*) solo se compone de instrucciones, comparativamente, simples, por lo que la complejidad de la arquitectura para

soportar dicho juego de instrucciones es menor, logrando mayores velocidades y un mejor consumo energético [4]. La relevancia de los procesadores *RISC* es tal que permite el auge de los *Smartphones* y el de múltiples dispositivos de bajo consumo y de tareas específicas, es decir, son la base de muchos *SoCs* hoy en día.

Actualmente, un procesador, más bien un *ISA*, de este tipo que despierta el interés, incluso de grandes empresas como *Nvidia* y *Western Digital Corp*, es el *RISC-V* [5]. Lo que hace interesante a este *ISA*, es su naturaleza de *Hardware Libre*, lo cual quiere decir que puede ser utilizado por cualquiera, sin necesidad de pagar derechos o licencias de uso, lo que genera un gran impacto en la industria al facilitar que una empresa pueda desarrollar su propia tecnología basada en este *ISA* de forma libre [5].

Lo anterior deja en claro la importancia de *RISC-V* y de los *SoCs* en la industria. Esto hace interesante, en el ambiente académico y de desarrollo tecnológico, el perfeccionamiento de diseños, prototipado e implementación de este tipo de sistemas. Por ello, el objetivo de este trabajo de memoria de título tiene un enfoque pedagógico al centrarse en el cómo se desarrolla un *SoC* y la metodología empleada para abordar el diseño de un sistema digital complejo.

En la implementación y realización de pruebas del *SoC*, hace falta una plataforma de prototipado. Para esto se emplea una tarjeta basada en un chip *FPGA* (*Field Programmable Gate Arrays*), tecnología con más de 30 años de desarrollo y de gran utilidad para el diseño, prueba, desarrollo e incluso implementación final de sistemas digitales [6]. Además, como se expone en [7], esta tecnología puede tener un gran impacto en la implementación del *5G* y en el *IoT* en un futuro cercano, por lo que es interesante aprovechar las capacidades y versatilidad de este tipo de dispositivos.

1.2 Objetivos del trabajo

El objetivo de este proyecto es el diseño, implementación y prueba de un *SoC*, con fines pedagógicos, en una tarjeta *FPGA*, utilizando el *ISA* libre de *RISC-V*. Como requisito mínimo se implementa el módulo base de instrucciones enteras, *RV32I*, junto a sus extensiones *M*, multiplicación y división de enteros, y *F*, el co-procesador para el cálculo de punto flotante de precisión simple [8].

Para resolver el problema planteado, se debe proponer un diseño para el *core* del *SoC*, basado en los módulos de *RISC-V* ya mencionados, *RV32IMF*. Luego se define el entorno de ejecución para dicho *core*. Este entorno define el comportamiento que tendrá un programa escrito en *RISC-V*, además de manejar los múltiples *cores* y *threads* disponibles en implementaciones más avanzadas. Debido al esfuerzo que conlleva el diseño de un *core* que soporta *RV32IMF*, el entorno de ejecución implementado es bastante simple.

Dicho entorno se limita a una interacción básica con el usuario mediante pulsadores e interruptores como entradas al sistema, y *displays* de 7 segmentos y LED como salidas. Lo que permite ingresar y visualizar enteros o números flotantes (ambos de 32 bits) en representación binaria y hexadecimal, respectivamente. Esto que quiere decir que dicho entorno, o *execution environment*, es de tipo "*Bare metal*" o, en otras palabras, una implementación

simple y directa en *hardware* [8].

Con lo anterior, solo se cubre la parte de diseño, pero también hace falta testear su funcionamiento, para ello se realizan simulaciones del *core* obtenido y de distintos sub-módulos del mismo, *FPU* y *ALU*, y del *SoC* completo. Una vez realizadas estas simulaciones se sintetiza el *SoC* en un *FPGA* para corroborar su correcto funcionamiento. Esto se explica con mayores detalles en la secciones 3 y 5.

De lo anterior se desprenden los siguientes objetivos específicos:

1. Implementar y diseñar la arquitectura de un *CPU-core* basada en el conjunto de instrucciones libre de *RISC-V*.
 - Incluyendo los módulos *RV32I*, *RV32M* y *RV32F* de *RISC-V*.
 - Analizar y comprender las posibilidades que ofrece el *ISA* libre de *RISC-V*.
2. Asociado al punto anterior, está el diseño e implementación del *hardware* necesario para el correcto funcionamiento del *SoC* y del entorno de ejecución.
3. Poner en práctica la metodología de diseño *Top-Down* en el desarrollo de sistemas digitales.
4. Utilizar una placa *FPGA* como plataforma de prototipado y para la implementación final.
5. Dominio y uso de *HDL* (*Hardware Description Language*), el cuál se utiliza para sintetizar todos los diseños digitales realizados sobre el *FPGA* y para realizar las simulaciones.
6. Diseño de pruebas de funcionamiento de los diseños obtenidos, previo a la implementación en *hardware* sobre el *FPGA*.
7. Obtener un *SoC* y corroborar su correcto funcionamiento.
8. Finalmente, comentar sobre cómo se aborda un trabajo de este tipo, las complicaciones relacionadas y las mejoras propuestas para el *SoC* obtenido.

A continuación, en la sección 2, se presentan los antecedentes teóricos y prácticos pertinentes al problema estudiado y el estado del arte sobre este tema.

2 Marco teórico y estado del arte

En esta sección se presentan los fundamentos teóricos básicos para abordar el problema, junto a una investigación que abarca desde la documentación necesaria para la implementación del *CPU-core* y del resto del *hardware*, hasta los entornos y tarjetas de desarrollo necesarias para el proyecto. Además, se revisa el trabajo de otros autores y el estado del arte como punto de referencia/partida para la realización de la metodología de trabajo y de diseño, de la implementación y de las pruebas de funcionamiento; temas que se explican en las secciones 3, 4 y 5.

2.1 Conceptos elementales

Para entender este proyecto se deben manejar conceptos básicos de sistemas digitales y arquitectura de computadores, puesto que un *SoC* es, básicamente, un sistema computacional completo, tal vez más o menos complejo, que cuenta con todos componentes básicos: *CPU* (*central processing unit*), sistemas de memoria, conjuntos de instrucciones (*ISA*), *hardware* controlador y controlado, *software* que controla las tareas del dispositivo, lógica binaria, entorno de ejecución, dispositivos de entrada/salida (E/S), etc... Por esto, es importante dar ciertas nociones fundamentales.

2.1.1 Sistemas digitales

Un sistema digital es un dispositivo electrónico que utiliza lógica binaria, junto al álgebra booleana, para ejecutar tareas. Estas pueden ser algoritmos complejos o muy simples, controlar algún sistema o entregar resultados matemáticos. Prácticamente, cualquier función que pueda realizar una máquina puede ser controlada mediante un sistema digital. Estos pueden ser tan complejos como un supercomputador, hasta tan simples como indicar la hora. Por lo mismo, hay una gran variedad de diseños digitales, según lo estudiado en los cursos [9] y [10] estos pueden ser:

1. **Combinacionales o secuenciales**, la diferencia radica en que el último es controlado por un reloj (*clock*) y posee elementos de memoria. Mientras que los circuitos combinacionales simplemente responden a las señales que reciben (sus salidas dependen solo de las entradas) sin estar controlados por un reloj. Es importante destacar que los circuitos combinacionales forman parte de los secuenciales.
 - En un sistema secuencial hay que considerar el *setup time* y el *hold time* del reloj para el correcto desempeño del mismo: el primero corresponde al tiempo mínimo que las entradas/memoria del sistema deben ser estables antes de que se active el flanco del reloj y el segundo al tiempo mínimo que deben estar estables después de que dicho flanco se active
2. Una **máquina de estados** (*Finite-state machine* o **FSM**), esta puede ser de *Moore*, donde las salidas solo dependen únicamente del estado actual, o de *Mealy*, donde también dependen de las entradas. Una máquina de estados es *secuencial*.
3. Ser de propósito general, como un microcontrolador, o específico, como un transreceptor WiFi.

4. Implementados en un único chip (*SoC*), en conjunto con múltiples circuitos integrados o, incluso, solo con transistores (casos muy simples).
5. Basados en una arquitectura de computador o no. Es decir que un sistema digital no debe contar necesariamente con un *CPU-core*.

Por otro lado, es importante destacar los siguientes aspectos de los sistemas digitales:

1. Estos sistemas manejan la información que se reciben del mundo real de forma digital, es decir, en forma de unos y ceros. Por lo que cualquier información continua pasa a ser discreta.
2. Están compuestas por compuertas lógicas (AND, NAND, OR, XOR, NOR, NOT, etc...), estas a su vez se componen de transistores dispuestos físicamente de tal forma que al entrar/recibir cierto nivel de voltaje, alto o bajo, en su(s) entrada(s), se retorna el correspondiente nivel de voltaje, según la lógica de la(s) compuerta(s), en la(s) salida(s).
3. Dependen de la tecnología en la que se implementan los transistores/compuertas (CMOS, TTL, n-MOS, p-MOS, etc...) y/o la decisión del diseñador, si para el sistema el nivel de voltaje alto corresponde al uno lógico y el bajo al cero lógico o viceversa.
4. Para el diseño de sistemas digitales se recurren a mayores niveles de abstracción, trabajando con compuertas y módulos típicos en la implementación de estos (MUX, DECODER, ENCODER, ALU, etc...) y no a nivel de transistor propiamente tal, se realiza un diseño a nivel de bloques funcionales.

2.1.2 Arquitectura de computadores

Un computador es un sistema digital muy complejo, puesto que es un dispositivo de propósito general, programable y que debe interactuar con el usuario, ojalá de forma amigable. Según lo estudiado en el curso [11], el cual a su vez se basa en el libro [12], este tipo de sistemas se componen de los siguientes elementos:

1. **La CPU:** se encarga de procesar toda la información del sistema.
 - La arquitectura de la *CPU* se construye a partir de un conjunto de instrucciones (*ISA*), las cuales el procesador deberá ser capaz de leer y ejecutar. Dichas instrucciones se codifican en binario y se almacenan en memoria.
 - El *datapath* de una *CPU* es una colección de unidades funcionales (como unidades lógicas aritméticas, multiplicadores, muxes, decoders, etc...) que realizan operaciones de procesamiento de datos, acceso a registros temporales y manejo de buses de datos. Una unidad de control está encargada de manejar el *datapath*. El *datapath* se compone de diferentes etapas de ejecución de la instrucción, siendo las siguientes 5 etapas las más típicas:
 - **Fetch:** etapa en la que se busca la instrucción en la memoria.
 - **Decode:** etapa en la que se decodifica la instrucción y lee los registros involucrados en la misma.
 - **Execute:** etapa en la que se ejecuta la operación aritmética/lógica de la instrucción.

- **Memory:** etapa en la que se accede a la memoria si la instrucción lo necesita.
 - **Write Back:** etapa en la que se guarda, si corresponde, el resultado de la instrucción en el registro.
- Presentan diferentes filosofías de diseño que repercuten en la complejidad de la arquitectura final de la *CPU*, estas son:
 - **RISC** (*Reduced instruction set computing*): basada en un conjunto de instrucciones reducido, menor complejidad. Busca simplificar al máximo el conjunto de instrucciones requeridas.
 - **CISC** (*Complex instruction set computing*): basada en un conjunto de instrucciones complejo, mayor complejidad. Busca acercar instrucciones a los lenguajes de alto nivel.
 - Pueden ser *Single-Cycle*, *Multi-Cycle* o *Pipelined*:
 - Un procesador *Single-Cycle* (ver figura 2.1) funciona de tal manera que la instrucción “cruza” una sola vez el *datapath* de la *CPU* (es decir en un solo ciclo de reloj), pero requiere una unidad funcional diferente para cada etapa de la ejecución de dicha instrucción.
 - El segundo tipo de procesador, *Multi-Cycle*, presenta un *datapath* simplificado, esto gracias a que las unidades funcionales para cada etapa de la ejecución de la instrucción son la misma dado a que la instrucción se ejecuta en múltiples ciclos de reloj.

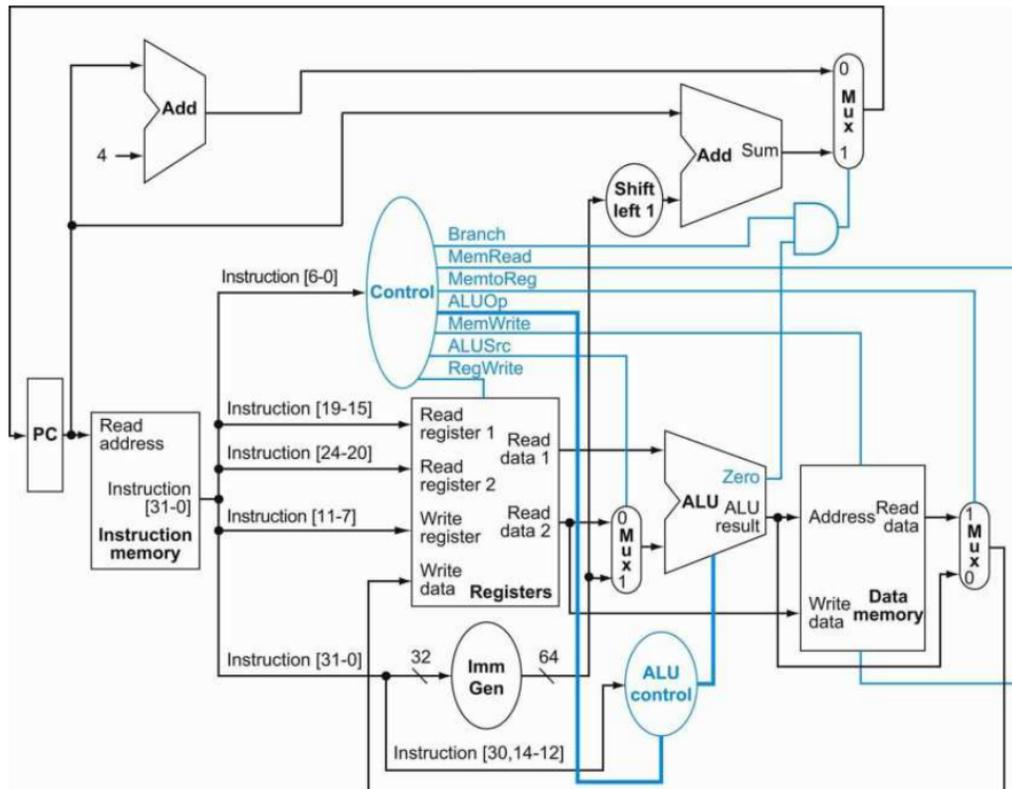


Figura 2.1: Ejemplo de *datapath single-cycle*. Fuente: [12].

- Y el último tipo de procesador, *Pipelined* (ver figura 2.2), es la norma en los procesadores de computadores de mayor rendimiento. Está basado en el diseño del *datapath* de un procesador *Single-Cycle* pero se introducen registros (memorias que almacenan los resultados de la etapa anterior para el siguiente ciclo de reloj) entre cada una de las etapas de ejecución de la instrucción, de esta forma se ejecutan de forma paralela más de una instrucción a la vez (en el mismo ciclo de reloj). Esto gracias a que se utilizan todas las unidades funcionales del *datapath* en cada ciclo de reloj pero por diferentes instrucciones. Con este diseño es imposible tener una instrucción ejecutada en un solo ciclo de reloj, como pasa en el *Single-Cycle*, pero permite elevar la frecuencia de funcionamiento, obteniendo un mayor rendimiento del sistema en comparación a la versión *Single-Cycle*, donde el periodo de reloj queda definido por el *datapath* completo, mientras que en el caso *Pipelined* queda definido por la unidad funcional (*Fetch*, *Decode*, *Execute*, *Memory* o *Write Back*) más lenta. En este tipo de arquitecturas suelen ocurrir dependencias de datos entre las etapas (*Data Hazard*) y fallas en la predicción de la siguiente instrucción cuando ocurre alguna bifurcación (*Control Hazard*).

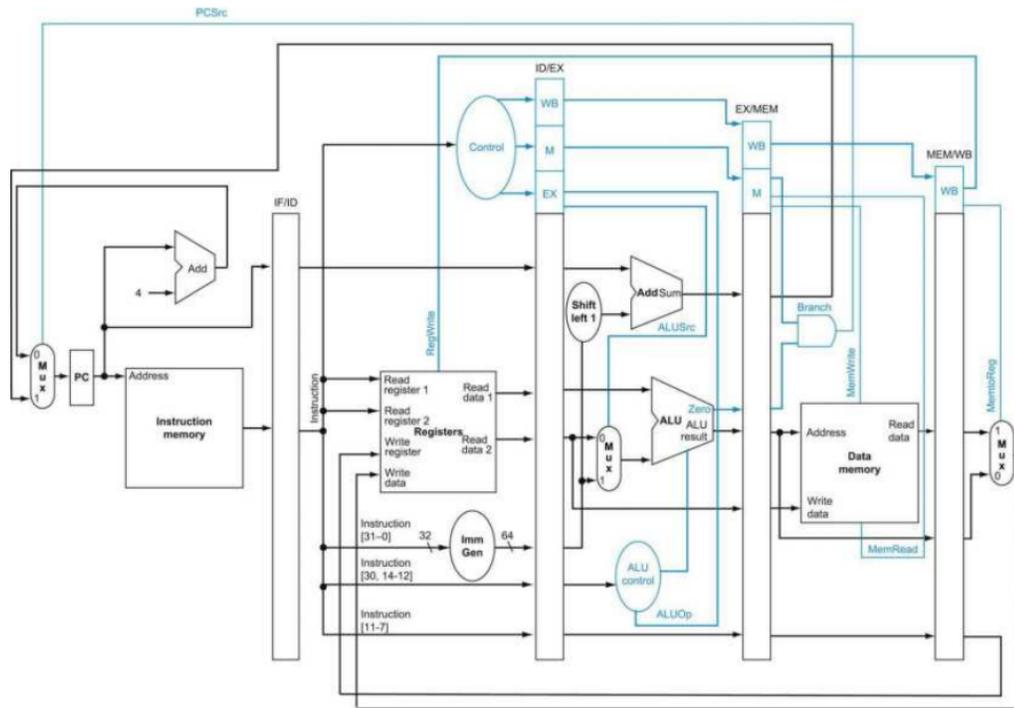


Figura 2.2: Ejemplo de *datapath pipelined*. Fuente: [12].

- Actualmente, debido a que cada vez es más difícil aumentar el rendimiento de un procesador se han desarrollado arquitecturas *multi-core* con *multiple issue* (paralelismo a nivel de instrucción, no confundir con *pipelined*), los cuales ejecutan múltiples instrucciones en el mismo ciclo de reloj y *core*, entre muchas otras.

- Según la forma en que direcciona la memoria se clasifican entre *Big-endian* y *Little-endian*, la diferencia radica en a que byte, más (*MSB*) o menos (*LSB*) significativo, respectivamente, se direcciona. Esto se aprecia claramente en la figura 2.3.

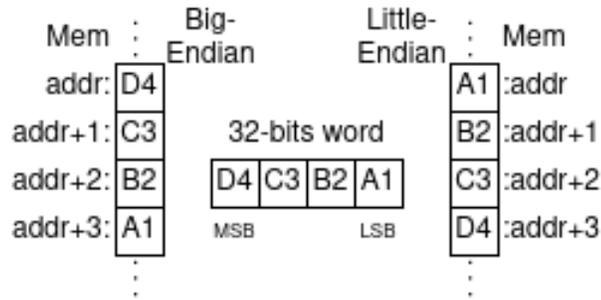


Figura 2.3: *Endianness*.

- Existen dos subdivisiones según cómo se comunica el procesador con la memoria de datos e instrucciones, estas son:

- *Von Neumann*: donde la memoria de datos e instrucciones es la misma o, visto desde otro ángulo, se acceden mediante el mismo bus de datos. Por esto, solo permite leer una instrucción o un dato en memoria a la vez, no permite realizar ambas operaciones al momento.

- *Harvard*: en este caso la memoria de datos e instrucciones se encuentran separadas o, en otras palabras, el procesador puede acceder mediante buses distintos a la memoria de programa/instrucciones y a la de datos, lo que permite un acceso simultaneo. Nótese que no es necesario diseñar de forma completamente aislada las memorias, basta con añadir dos puertos de lectura.

2. **Sistema de memoria:** la memoria se encarga de almacenar la información con la que trabaja todo el sistema, ya sean datos y/o instrucciones. La memoria generalmente se jerarquiza, debido a que los dispositivos de almacenamiento suelen ser más lentos que la *CPU*, por lo que se almacena la información con la que se está trabajando en el momento en una memoria más rápida. Siendo la más rápida la memoria caché integrada en la *CPU* misma. Esto se debe a que las memorias veloces son muy costosas de implementar, por ello también suelen ser más pequeñas (como la memoria caché), mientras que las memorias RAM ya son más económicas de implementar y cuentan con mayor capacidad de almacenamiento, después un disco duro mecánico ya es bastante más económico por GB de almacenamiento pero es el más lento de todos; es decir, a mayor almacenamiento se suele perder velocidad de acceso, por ello, este apartado suele ser un limitante en el rendimiento de un sistema computacional.

3. **Dispositivos de entrada/salida (E/S):** estos son el medio por el cual el sistema recibe información e interactúa con el mundo real.

- Ejemplos de entradas: teclado, ratón, pantalla táctil, sensores, dispositivos de almacenamiento, etc...

- Ejemplos de salidas: la imagen en un monitor (generalmente se requiere de un co-procesador gráfico para esta tarea), actuadores, dispositivos de almacenamiento, altavoces, etc...

4. • **Entorno de ejecución:** o *Execution Environment* [8], es el encargado de que cada uno de los *harts* (hilo de ejecución a nivel de *hardware*, no de *software*, como lo son los *threads*) ejecuten las instrucciones que se le asignan (*forward progress*), si un *hart* no

está ejecutando ninguna tarea entra en un modo de espera por algún evento y el entorno de ejecución no se debe preocupar de este hasta que le asigne alguna tarea nueva al *hart* (el entorno suspende su responsabilidad). La responsabilidad del entorno de ejecución para con el *hart* finaliza cuando este último termina su ejecución.

- Es interesante notar la diferencia entre un *thread* y un *hart*, los primeros son ejecutados por un *software* el cual se encuentra dentro de un entorno de ejecución mientras que los *harts* son manejados por dicho entorno. Por lo tanto, un *thread* no decide como administra los recursos del sistema, si no que le pide al entorno de ejecución que administre el uso de los *harts* disponibles. Nótese que un *core* puede tener más de un *hart*.

- Por otro lado, el estado inicial de un programa, el número y tipos de *harts* disponibles y sus privilegios dentro de sistema quedan definidos por el *Execution Environment Interface (EEI)*, el cual también define la accesibilidad y los atributos de la memoria, las regiones de E/S, el comportamiento de todas las instrucciones legales ejecutadas en cada *hart* (el *ISA* es un componente de la *EEI*), y el manejo de cualquier interrupción o excepción que surja durante la ejecución, incluidas las llamadas al entorno. La implementación del *EEI* puede ser puramente *software*, *hardware* o una mezcla de ambas. A continuación, se presentan algunos ejemplos:

- Puramente *hardware (Bare Metal)*, el/los *hart/s* están directamente implementados en hilos de procesamiento físico y sus instrucciones tienen acceso completo al espacio físico de direcciones de memoria. Esta plataforma define un entorno de ejecución al encenderse.

- Sistemas operativos que proporcionan múltiples entornos de ejecución a nivel de usuario mediante la multiplexación de *harts* a nivel de usuario en subprocesos en cada uno de los procesadores físicos disponibles y controlando el acceso a la memoria a través de la memoria virtual.

- *Hypervisors* los cuales brindan múltiples entornos de ejecución a nivel de supervisor para sistemas operativos invitados.

- Emuladores.

Por otro lado, el desempeño de un procesador se puede resumir como “el tiempo que tarda en ejecutar algún programa en particular”. Por ello, se define el *Performance* de un procesador como $Performance = 1/CPUtime$, se puede decir que un procesador *X* es *n* veces más rápido que *Y* en ejecutar dicho programa si:

$$\frac{Performance_X}{Performance_Y} = \frac{CPUtime_Y}{CPUtime_X} = n \quad (2.1)$$

Donde se define el tiempo de CPU como:

$$CPUtime = totalClockCycles * clockPeriod = \frac{totalClockCycles}{clockFrequency} \quad (2.2)$$

Considerando:

$$totalClockCycles = \frac{ClockCycles}{instruction} * \frac{instructions}{program} \quad (2.3)$$

Definiendo *CPI* (*cycles per instruction*) como:

$$CPI = \frac{ClockCycles}{instruction} \quad (2.4)$$

Se obtiene:

$$CPUtime = \frac{instructions}{program} * \frac{CPI}{clockFrequency} \quad (2.5)$$

Es importante destacar que el *CPI* no es el mismo para todos los tipos de instrucciones, pero por simplicidad se asume lo contrario en las ecuaciones recién expuestas.

2.1.3 Representación binaria en punto flotante

En esta sección se presenta la notación binaria para números punto flotante de precisión simple, la cual está definida por la *IEEE 754* [13]. Esta representación utiliza 32 bits y se compone de 3 campos, *sign* (1 bit), *exponent* (8 bits) y *fraction* (23 bits), tal como se puede ver en tabla 2.1.

31	30	23	22	0	
sign	exponent	fraction			X

Tabla 2.1: Representación binaria en punto flotante para un número *X*.
Fuente: [13].

En esta representación, *sign* indica el signo del número, 1 para negativo y 0 para positivo. Por otro lado, *exponent* es un número sin signo el cual indica el exponente sesgado del número representado, esto quiere decir que el exponente real viene dado por $exponent - bias$, donde *bias* es igual a 127 para el caso de precisión simple. Luego, para el caso común con números normalizados, *fraction* es la fracción de la mantisa (*significand*) normalizada, cumpliéndose que $significand = 1.fraction$ (bit silencioso). Finalmente, el número viene dado por:

$$X = (-1)^{sign} * significand * 2^{exponent - bias} = (-1)^{sign} * 1.fraction * 2^{exponent - 127} \quad (2.6)$$

Hay ciertos casos especiales en esta notación, éstos son notaciones reservadas para números fuera del rango que permite la precisión simple (la cual es desde $\pm 1.18 * 10^{-38}$ hasta $\pm 3.4 * 10^{38}$), dichos casos se resumen en la tabla 2.2.

<i>exponent</i>	<i>fraction</i>	<i>X</i>	Descripción
= 0	= 0	<i>Zero</i>	El cero es indiferente al bit de signo
= 0	≠ 0	<i>Denormalized</i>	Número desnormalizado: $significand = 0.fraction$
= 255	= 0	\pm <i>Infinity</i>	El bit de signo indica hacia donde tiende el número
= 255	≠ 0	<i>Signaling NaN</i>	<i>Not a Number</i> con $fraction[22] = 0$
= 255	≠ 0	<i>Quiet NaN</i>	<i>Not a Number</i> con $fraction[22] = 1$

Tabla 2.2: Casos especiales para la representación en punto flotante de precisión simple según la *IEEE 754*. Fuente: [13]

Si durante alguna operación aritmética se decrece el campo *exponent* hasta llegar a cero, se empieza a desplazar hacia la derecha el *significand* (originalmente normalizado). Si se realizan 23

desplazamientos o menos, el *significand* no llega a ser cero, obteniéndose un número desnormalizado (sin bit silencioso). En el caso que llegase a ser cero ocurre un *underflow* y el resultado será cero. Por otro lado, si dicha operación aritmética aumenta el *exponent* y llega a 255, ocurre un *overflow* y, automáticamente, el resultado pasa a ser $\pm Infinity$. Finalmente, si la operación no entrega un número válido se devuelve un *NaN*.

Cuando se realizan operaciones se consideran 3 bits extras, a la derecha del campo *fraction*. Estos se explican en la tabla 2.3 y se utilizan, según el modo de redondeo empleado, para decidir si truncar o redondear el resultado de alguna operación.

Bit	Descripción
$Guard = g = significand[-24]$	Primer bit a la derecha de $fraction[0]$.
$Round = r = significand[-25]$	Segundo bit a la derecha de $fraction[0]$.
$Sticky = s = significand[-26]$	Tercer bit a la derecha de $fraction[0]$. Este bit se mantiene igual a 1 siempre que haya algún bits igual a 1 a la derecha de r .

Tabla 2.3: Bits de redondeo. Fuente: [12]

2.2 RISC-V

El corazón de este proyecto es la *CPU*, el encargado de gestionar los datos, acciones y todas las funciones del *SoC*. En este caso la implementación se realizará utilizando la arquitectura e *ISA* del *RISC-V*, un procesador de *Hardware Libre* desarrollado en la *University of California, Berkeley* [8].

Los manuales disponibles sobre *RISC-V* se dividen en dos volúmenes, el primero es [8] y se centra en las instrucciones *unprivileged*, mientras que el segundo volumen, [14], se centra en las instrucciones *privileged*. Las instrucciones sin privilegios (*unprivileged*) son aquellas que generalmente se pueden usar en todos los modos de privilegio y en todas las arquitecturas privilegiadas (*privileged*), aunque el comportamiento puede variar según el modo de privilegio y la arquitectura de privilegio. El concepto de privilegio se explica en [8], pero para el nivel de la implementación realizada basta considerar, únicamente, el modo sin privilegios presentado en [8].

A continuación, se expondrán las principales características y consideraciones del *ISA* de *RISC-V*. Toda esta información, y más, se puede hallar en el manual oficial [8].

Ahora, se presentan los principales objetivos del proyecto *RISC-V*:

- El desarrollo de un *ISA* completo, abierto y gratuito a la disposición, tanto de la industria como de la academia.
- El desarrollo de un *ISA* útil para la implementación directa de *hardware* nativo, no solo para simulación o traducción binaria.
- El desarrollo de un *ISA* sencillo que evite el desarrollo de arquitecturas demasiado complicadas, con el fin de que sea útil para implementar algún estilo particular de micro-arquitecturas o para la implementación de tecnología (por ejemplo, tecnologías personalizables como *ASIC* (*Application-Specific Integrated Circuit*) o *FPGA*). Este *ISA* debe permitir una implementación eficiente en cualquiera de dichos casos.
- Desarrollar un *ISA* segmentado, es decir, compuesto por un *ISA* básico con las instrucciones más simples, pero completamente funcional por si solo y útil como base en el diseño de ace-

leradores de *hardware* (*accelerators*), y por extensiones estandarizadas de este *ISA* base, que son útiles para el desarrollo de *software* de propósito general.

- Soporte para el estándar *2008 IEEE-754* para el cálculo de punto flotante.
- Desarrollar un *ISA* que admite amplias extensiones y variantes especializadas.
- El desarrollo de las variantes de 32 y 64 bits.
- Desarrollar un *ISA* con soporte para implementaciones altamente paralelas de *multi-cores* o *many-cores*, incluidos los multiprocesadores heterogéneos (sistemas donde los diferentes *cores* no son idénticos).
- El desarrollo de instrucciones opcionales de longitud variable para expandir el espacio de codificación de instrucciones disponible y para admitir una codificación de instrucción densa, opcional, para un mejor rendimiento, tamaño de código estático y eficiencia energética.
- Desarrollar un *ISA* completamente virtualizable con el fin de facilitar el desarrollo de un *Hypervisor* (un *Hypervisor* no es más que un software que crea y ejecuta máquinas virtuales).
- Desarrollar un *ISA* que simplifica los experimentos con nuevos diseños de arquitectura privilegiada.

El diseño de una plataforma de hardware basada en *RISC-V* puede contener bancos de memoria, dispositivos de entrada salida, etc. . . , pero por sobre todo es capaz de contener múltiples *cores* compatibles, por definición un *core* debe tener integrada una unidad independiente de *instruction fetch* (unidad encargada de encontrar la siguiente línea de comando a ejecutar por el *core*). A su vez, a un *core* se le pueden agregar extensiones del juego de instrucciones o algún co-procesador añadido (también añadiéndose extensiones al juego de instrucciones). Otro elemento que puede contener un diseño de este tipo es un *accelerator*, la cual es una unidad de función fija no programable o un *core* que puede funcionar de forma autónoma pero que está especializado para ciertas tareas.

2.2.1 Visión general al *ISA* de *RISC-V*

RISC-V se define mediante un *ISA* completo base, que debe estar presente en cualquier implementación, más extensiones opcionales para este. Dicha *ISA* base se parece bastante al de las primeras arquitecturas *RISC* y se restringe a las instrucciones mínimas necesarias para el funcionamiento de compiladores, *assemblers* (programa que convierte el lenguaje ensamblador en código máquina), *linkers* (programa que toma uno o más archivos objeto (generados por un compilador o ensamblador) y los combina en un solo archivo ejecutable, archivo de biblioteca u otro archivo “objeto”), y sistemas operativos (añadiendo ciertas operaciones con privilegios), por lo que se provee un *ISA* conveniente y un *software toolchain*, “esqueleto”, sobre el cual se pueden construir *ISAs* más personalizadas.

RISC-V es, en realidad, una familia de *ISAs* relacionadas entre sí; hay 4 *ISAs* base diferentes. Cada una de estas se caracteriza por el ancho de los *integer registers*, el tamaño correspondiente del espacio de direcciones, y por el número de *integer registers*. Hay dos variantes primarias, el *RV32I* y el *RV64I*, los cuales tienen un espacio de direcciones de 32 y 64 bits, respectivamente. Se utiliza el termino *XLEN* para referirse al ancho del *integer register* en bits (32 o 64 bits generalmente). Hay otras variantes secundarias, conocidas como *RV32E* (variante del *RV32I* con soporte para microcontroladores; tiene la mitad de *integer registers*) y *R128I* (aún no está completo y pretende trabajar con un $XLEN = 128$). Los conjuntos de instrucciones utilizan una representación en complemento a dos para valores enteros con signo.

Por otro lado, las extensiones de *RISC-V* se pueden clasificar en 3 categorías; *standard*, *reserved* y *custom*. *Standard* se usa para referirse a las extensiones definidas por la fundación *RISC-V*, *reserved* son las extensiones que aún no están definidas pero que en un futuro se pueden volver *standard*.

non-standard se utiliza para referirse a las extensiones no definidas por la fundación. Las extensiones *custom* no se deben usar con una *standard* y están disponibles para extensiones *non-standard* específicas del proveedor. Una extensión *non-conforming* describe una extensión *non-standard* que puede usar una codificación *standard* o *reserved* (por ejemplo, las extensiones personalizadas son *non-conforming*). Las extensiones del juego de instrucciones, generalmente, son compartidas, pero pueden haber ligeras diferencias de funcionalidad dependiendo del *ISA* base utilizado.

El nombre asignado al *ISA* base es *I* (con el prefijo RV32 o RV64, según la variante elegida), y se compone por instrucciones de cálculo de números enteros, cargas de números enteros, almacenamiento de números enteros e instrucciones de flujo de control. A continuación, se listan las extensiones *standar* más importantes:

- *M*: extensión de multiplicación y división de enteros, agrega instrucciones para multiplicar y dividir valores contenidos en los *integer registers*.
- *A*: extensión de instrucciones *atomic*, agrega instrucciones que leen, modifican y escriben de forma “atómica” (sin división, evita la pérdida de información) la memoria para la sincronización entre *harts*.
- *F*: extensión de punto flotante de precisión simple, agrega registros de punto flotante, instrucciones de precisión simple, y cargas y almacenamiento de precisión simple.
- *D*: extensión de punto flotante de precisión doble, expande los registros de punto flotante y agrega instrucciones computacionales de precisión doble, y cargas y almacenamiento de precisión doble.
- *C*: la extensión de instrucción comprimida proporciona formas más estrechas (16 bits) de instrucciones comunes.

2.2.2 La Memoria en *RISC-V*

Un *hart* basado en *RISC-V* tiene un espacio de direcciones de 2^{XLEN} bytes y es direccionable a nivel de byte. En este ámbito se define como *word* de memoria a un conjunto de 32 bits (4 bytes), *halfword* serían 16 bits (2 bytes), *doubleword* son 64 bits (8 bytes) y *quadword* 128 bits (16 bytes). El espacio de direcciones de memoria es circular, es decir que el byte en la dirección $2^{XLEN} - 1$ es adyacente al de la dirección 0. Gracias a esto, para el cálculo de las direcciones de memoria realizadas por el *hardware* se ignoran los *overflows*, simplemente se realiza la operación módulo 2^{XLEN} .

El entorno de ejecución es el encargado de determinar y mapear los recursos de *hardware* en el espacio de direcciones del *hart*. Diferentes rangos de direcciones del espacio de direcciones de un *hart* pueden:

1. Estar vacíos, libres.
2. Contener la memoria principal.
3. Contener dispositivos de Entrada/Salida (E/S).

Las lecturas y escrituras de dispositivos de E/S pueden tener efectos secundarios visibles, mientras que los accesos a la memoria principal no. Es posible, para el entorno de ejecución, que todo el espacio de direcciones del *hart* sean dispositivos de E/S, pero usualmente hay una porción específica para la memoria principal.

Cuando una plataforma *RISC-V* tiene varios *harts*, los espacios de direcciones de dos *harts* pueden ser completamente iguales o completamente diferentes, o pueden ser parcialmente diferentes, pero compartiendo algún subconjunto de recursos, mapeados en el mismo o diferentes rangos de direcciones.

La ejecución de cada instrucción de máquina desencadena uno o más accesos a la memoria, subdivididos en accesos “implícitos” y “explícitos”. En cada instrucción ejecutada hay una lectura “implícita” en memoria, conocida como *instruction fetch*, con el fin de obtener la instrucción codificada a ejecutar; en muchas instrucciones es el único acceso a memoria. El acceso “explícito” se da cuando las instrucciones específicas de carga (*load*) y almacenamiento (*store*) realizan un acceso de la memoria, lectura (*read*) o escritura (*write*), respectivamente, en una dirección determinada por la instrucción. El entorno de ejecución puede dictar que la ejecución de la instrucción realice otros accesos implícitos a la memoria (como implementar la traducción de direcciones), además de los documentados para la *ISA* sin privilegios.

El entorno de ejecución determina qué porciones del espacio de direcciones no vacante son accesibles para cada tipo de acceso a la memoria. Para ejemplificar, el conjunto de direcciones que puede ser leído por los accesos de memoria implícitos (*instruction fetch*), puede o no traslaparse con el conjunto de direcciones para los accesos de memoria de lectura explícitos. O el conjunto de direcciones que puede ser explícitamente escrito, por alguna instrucción *store*, puede ser solo un subconjunto de todas las direcciones que se pueden leer. Normalmente, si una instrucción intenta acceder a la memoria en una dirección inaccesible/prohibida, se genera una excepción para la instrucción. Las direcciones de memoria vacantes en el espacio de direcciones nunca son accesibles.

2.2.3 Codificación del largo de instrucciones

El *ISA RISC-V* básico tiene instrucciones de 32 bits de longitud fija que deben alinearse en límites de 32 bits. Sin embargo, el esquema de codificación estándar de *RISC-V* está diseñado para soportar extensiones de *ISA* con instrucciones de largo variable, donde cada instrucción puede tener cualquier número de parcelas (*parcels*) de instrucciones de 16 bits de longitud y las parcelas están alineadas, naturalmente, en límites de 16 bits. La extensión de *ISA* estándar *C*, reduce el tamaño del código al proporcionar instrucciones comprimidas de 16 bits y relajar las restricciones de alineación para permitir que todas las instrucciones (16 y 32 bits) se alineen en cualquier límite de 16 bits, con el fin de mejorar la densidad del código.

Se utiliza el término *IALING* (medido en bits) para referirse a las restricciones de alineamiento de las direcciones de memoria de las instrucciones impuesta por la implementación. *IALING* es de 32 bits para el *ISA* base, pero algunas extensiones relajan el *IALING* a 16 bits, como es el caso de la extensión *C*. 32 y 16 son los únicos dos valores que puede tomar *IALING*.

El término *ILEN* (medido en bits) se utiliza para referirse al máximo *Instruction-Length* de una implementación dada; siempre es múltiplo de *IALING*. En implementaciones donde se utiliza un *ISA* base, el *ILEN* es igual a 32, y en implementaciones donde se utiliza un largo de instrucciones mayor, el *ILEN* también es mayor.

En la figura 2.4 se puede ver el estándar *RISC-V* para la codificación del largo de instrucciones. Todas las instrucciones de 32 bits en el *ISA* base tienen los últimos dos bits menos significativos fijos en 11_{base2} . Las instrucciones opcionales comprimidas de 16 bits (extensión *C*) tienen los últimos dos bits menos significativos fijos en 00_{base2} , 01_{base2} o 10_{base2} .

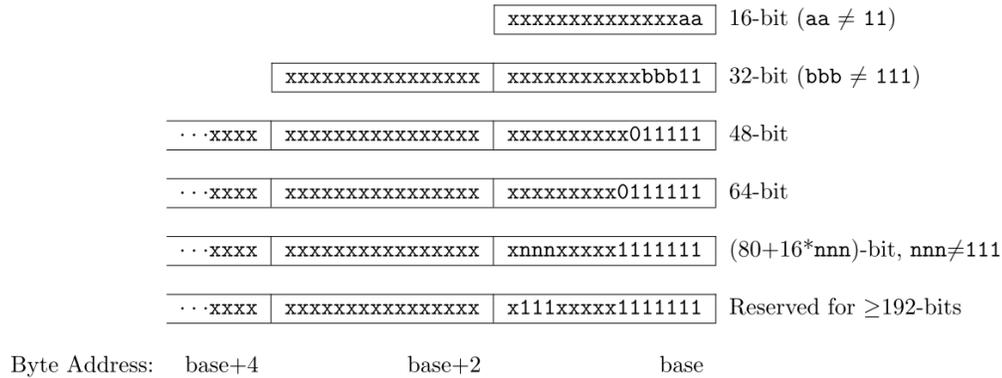


Figura 2.4: Codificación de longitud de instrucción *RISC-V*. Solo las codificaciones de 16 bits y 32 bits se consideran “congeladas” en este momento. Fuente: [8].

Las extensiones del conjunto de instrucciones estándar codificadas con más de 32 bits tienen bits adicionales, de orden inferior, fijos en 1_{base2} , con las convenciones para longitudes de 48 y 64 bits que se muestran en la figura 2.4. Las instrucciones con un largo de entre 80 y 176 bits son codificados usando un campo de 3 bits, [14:12], que entregan el número de *halfwords* (16 bits) agregadas a las primeras 5 *halfwords*. La codificación con los bits [14:12] = 111_{base2} está reservado para codificación de instrucciones aún más largas en un futuro.

Codificaciones con los bits [15:0] iguales a cero se consideran instrucciones “ilegales”. Se considera que estas instrucciones tienen una longitud mínima: 16 bits si está presente alguna extensión de conjunto de instrucciones de 16 bits, de lo contrario 32 bits. La codificación con los bits [*ILEN*-1:0] iguales a uno también se considera “ilegal”; esta instrucción se considera que tiene una longitud de bits *ILEN*.

Endianness de *RISC-V*

La *ISA* base de *RISC-V* puede trabajar con sistemas de memoria tanto *Big-Endian* como *Little-Endian* (definido por el *EEI*); la arquitectura privilegiada fomenta aún más las operaciones *Bi-Endian*. Las instrucciones se guardan en memoria en forma de secuencias de 16 bits, en parcelas *Little-Endian*, sin importar si el sistema de memoria es *Big* o *Little Endian*. Las parcelas que forman una instrucción se almacenan en direcciones de memoria de 16 bits (*halfword*) crecientes, y la parcela con la dirección más baja contiene los bits menos significativos de la instrucción.

2.2.4 Excepciones, trampas e interrupciones

Se utiliza el término “*excepción*” para referirse a una condición inusual que ocurre durante la ejecución de alguna otra instrucción en el *hart* actual del *RISC-V*. El término “*interrupción*” se refiere a un evento externo, asíncrono, que puede causar que un *hart* del *RISC-V* experimente una inesperada transferencia de control. Finalmente, se usa la palabra “*trampa*” para referirse a la transferencia de control a un manejador de *trampas* a causa de una *excepción* o *interrupción*.

El comportamiento general de la mayoría de los *EEIs* de un *RISC-V* es que acciona una *trampa* para algún manejador cuando se señala una excepción en alguna instrucción. La manera en que las *interrupciones* son generadas, enrutadas y habilitadas por un *hart* depende del *EEI*.

El cómo las *trampas* son manejadas y se hacen visibles en el *software* en ejecución, en el *hart*, depende del entorno de ejecución. Desde el punto de vista del *software* corriendo dentro de un

entorno de ejecución, las *trampas* detectadas por un *hart* durante el tiempo de ejecución pueden tener cuatro efectos diferentes:

- **Trampa Contenida (*Contained Trap*):** la *trampa* es visible y manejada por el *software* que está corriendo dentro del entorno de ejecución. Por ejemplo, en un *EEI* que provee ambos modos en el *hart*, *supervisor* y *usuario*, un *ECALL* (*System Call* o, como se denomina en la nomenclatura de *RISC-V*, *Executive Call*) por un *hart* en *modo-usuario*, generalmente, resulta en una transferencia de control a un manejador en el *modo-supervisor*, que se ejecuta en el mismo *hart*. De igual modo, en el mismo entorno de ejecución, cuando un *hart* es interrumpido, un manejador de interrupciones va a ejecutarse en el *modo-supervisor* en el *hart*.
- **Trampa Solicitada (*Requested Trap*):** la *trampa* es una excepción asíncrona, la cual es una llamada explícita para el entorno de ejecución y solicita una acción en nombre del *software* que se está corriendo en el entorno de ejecución. Un ejemplo es un *System Call*. En este caso, la ejecución puede o no reanudarse en el *hart* después de que la acción solicitada es tomada por el entorno de ejecución. Por ejemplo, un *System Call* puede eliminar el *hart* o provocar una terminación ordenada de todo el entorno de ejecución.
- **Trampa Invisible (*Invisible Trap*):** la *trampa* es manejada de forma transparente por el entorno de ejecución y la ejecución se reanuda normalmente, después de que se maneja la *trampa*. Los ejemplos incluyen emular instrucciones faltantes, manejar fallas de páginas no residentes en un sistema de memoria virtual paginado por demanda o manejar interrupciones del dispositivo para un trabajo diferente en una máquina multiprogramada. En esos casos, el *software* corriendo dentro del entorno de ejecución no se entera de la *trampa* (se ignoran los efectos en el “*timing*” en estas definiciones).
- **Trampa Fatal (*Fatal Trap*):** esta *trampa* representa una falla fatal y hace que el entorno de ejecución finalice la ejecución. Los ejemplos incluyen fallar en una verificación de protección de página de memoria virtual o permitir que expire un *watchdog timer*. Cada *EEI* debe definir como una ejecución finaliza y se reporta con el medio externo.

En la tabla 2.4 se puede apreciar un resumen de las principales características de los tipos de *trampas*.

	Contenida	Solicitada	Invisible	Fatal
¿Terminó la ejecución?	No	No ₁	No	Si
¿El <i>software</i> es ajeno?	No	No	Si	Si ₂
¿Manejado por el entorno?	No	Si	Si	Si

Tabla 2.4: Características de las *trampas*. Notas: 1) el término puede ser solicitado, 2) trampas fatales imprecisas pueden ser observadas por *software*. Fuente: [8].

El *EEI* define para cada *trampa* si se maneja con precisión, aunque la recomendación es mantener la precisión siempre que sea posible. Las *trampas contenidas* y *solicitadas* pueden observarse que son imprecisas por *software* dentro del entorno de ejecución. Las *trampas invisibles*, por definición, no pueden ser vistas si son imprecisas o no por el *software* que está corriendo dentro del entorno de ejecución. Las *trampas fatales* pueden ser observadas como imprecisas por el *software* corriendo dentro del entorno de ejecución, si las instrucciones erróneas conocidas no causan la terminación inmediata del entorno.

2.2.5 El *ISA* base y extensiones implementadas, *RV32IMF*

A continuación, se presenta el juego de instrucciones consideradas, *RV32IMF*, para la implementación del *SoC*, el cual se conforma por el *ISA* base, *RV32I*, y dos de sus extensiones *standard*, *M*

y F . Donde, se cumple $XLEN = ILEN = IALING = 32bits$.

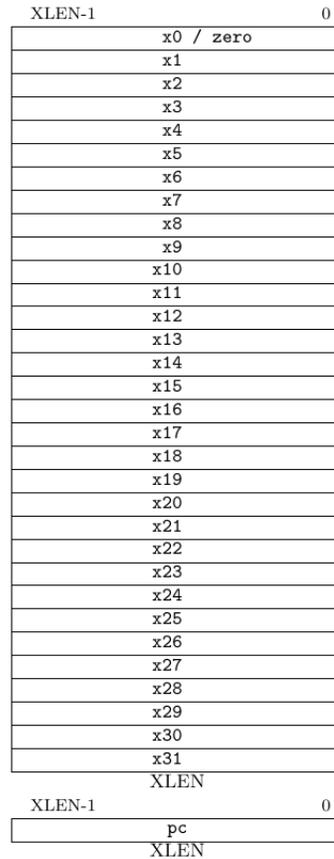


Figura 2.5: *Register Files* para *RV32I unprivileged*. Fuente: [8].

RV32I añade 32 registros de 32 bits, $x0 - x31$, de propósito general (*Register Files*), con $x0$ siempre igual a cero (*hardwired to zero*). Los registros de uso general pueden mantener valores que muchas instrucciones interpretan como una colección de valores *booleanos*, como enteros binarios en complemento de dos o como enteros binarios sin signo. Adicionalmente, hay un registro *unprivileged* llamado *Program Counter (PC)*, el cual almacena la dirección de la instrucción actual. Ver figura 2.5.

RV32I tiene 4 formatos de instrucciones básicos (*R/I/S/U*), donde se mantienen los registros de la fuente (*source*), $rs1$ y $rs2$, y el de destino (*destination*), rd , en la misma posición en todos los formatos para facilitar/simplificar la decodificación. A excepción de los datos inmediatos de 5 bits utilizados en las instrucciones *CSR (Control and Status Register)*, extensión no implementada, los datos inmediatos (*imm*) siempre están extendidos por el signo y, generalmente, se ubican en los bits disponibles a la izquierda en la instrucción que se han asignado para reducir la complejidad del *hardware*. En particular, el bit de signo, para todos los datos inmediatos, siempre está en el bit 31 de la instrucción para acelerar el circuito de extensión de signo. Por otro lado, *RV32M* solo añade instrucciones de tipo *R* con soporte para multiplicación y división de enteros. Es importante destacar que no se incluyen instrucciones especiales para verificaciones de *overflow* de operaciones aritméticas enteras, pero estas verificaciones se pueden realizar mediante una, dos o tres instrucciones simples según el caso.

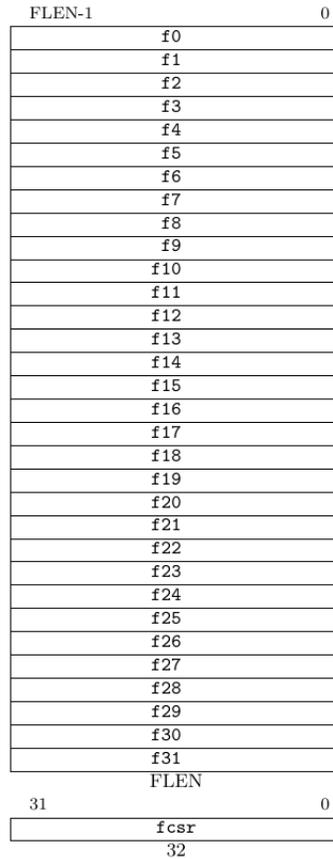


Figura 2.6: *Register Files* para *RV32F unprivileged*. Fuente: [8].

Para dar soporte a operaciones de punto flotante de precisión simple, bajo el estándar *IEEE 754-2008*, a nivel de *hardware* se añade *RV32F*, el cual integra 32 registros, $f0 - f31$, de 32 bits (lo que quiere decir que $FLEN = 32$, este valor indica el ancho de los registros de la extensión F) exclusivos para almacenar registros de operaciones de punto flotante. Además, añade un registro de *control and status*, *FCSR*, pero como no se implementa la extensión *CSR* de *RISC-V*, tampoco se implementa *FCSR*. Ver figura 2.6.

A su vez, *RV32F* añade instrucciones de tipo *R* e introduce las de tipo *R4* (las cuales agregan una tercera fuente, $rs3$), además de añadir una de tipo *S* y otra de tipo *I*, correspondientes al *store* y *load*, respectivamente, para palabras de punto flotante, ya que utilizan un espacio de registros distinto al de las operaciones enteras.

En la tabla 2.5 se puede apreciar la codificación y los campos que componen a cada tipo de instrucción considerada:

31		27	26	25	24	20	19	15	14	12	11	7		6	0	
funct7				rs2		rs1		funct3		rd		opcode				R
rs3		funct2			rs2		rs1		funct3		rd		opcode			R4
imm[11:0]							rs1		funct3		rd		opcode			I
imm[11:5]				rs2		rs1		funct3		imm[4:0]			opcode			S
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]			opcode			B
imm[31:12]											rd		opcode			U
imm[20 10:1 11 19:12]											rd		opcode			J

Tabla 2.5: Codificación de los distintos tipos de instrucciones consideradas.
Fuente: [8].

Las instrucciones de tipo *R* y *R4* son operaciones *register-to-register*, las de tipo *I* son *immediate-to-register* o *load*, las tipo *S* son operaciones *store*, la instrucción de tipo *J* corresponde a un *jump*, las tipo *B* son *branch* y las tipo *U* también son operaciones *immediate-to-register*.

2.2.5.1 Instrucciones enteras, *RV32IM*

En la tabla 2.6 se aprecian todas las instrucciones de *RV32I* implementadas, junto al detalle de cada uno de los campos presentados en la tabla 2.5 y una breve descripción, escrita en pseudocódigo, de la operación que realizan. Nótese, que en la tabla 2.5 los campos *rs1*, *rs2* y *rd* tienen 5 bits y se refieren a la dirección en el *Register Files* y el campo *imm* se encuentra sin extender; mientras que en la tabla 2.6, *rs1*, *rs2* y *rd* se refieren a los valores de *XLEN* bits almacenados (o por almacenar en el caso de *rd*) en el *Register Files* y con *imm* ya extendido.

RV32I - Integer instructions					
Inst.	FMT	Opcode	funct3	extra	Descripción
add	R	0110011	0x0	funct7=0x00	rd = rs1 + rs2
sub	R	0110011	0x0	funct7=0x20	rd = rs1 - rs2
xor	R	0110011	0x4	funct7=0x00	rd = rs1 ^ rs2
or	R	0110011	0x6	funct7=0x00	rd = rs1 rs2
and	R	0110011	0x7	funct7=0x00	rd = rs1 & rs2
sll	R	0110011	0x1	funct7=0x00	rd = rs1 << rs2
srl	R	0110011	0x5	funct7=0x00	rd = rs1 >> rs2
sra	R	0110011	0x5	funct7=0x20	rd = rs1 >> rs2 (msb-extends)
slt	R	0110011	0x2	funct7=0x00	rd = (rs1 < rs2) ? 1 : 0
sltu	R	0110011	0x3	funct7=0x00	rd = (rs1 < rs2) ? 1 : 0 (unsigned)
addi	I	0010011	0x0		rd = rs1 + imm
xori	I	0010011	0x4		rd = rs1 ^ imm
ori	I	0010011	0x6		rd = rs1 imm
andi	I	0010011	0x7		rd = rs1 & imm
slli	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 << imm[4:0]
srli	I	0010011	0x5	imm[11:5]=0x00	rd = rs1 >> imm[4:0]
srai	I	0010011	0x5	imm[11:5]=0x20	rd = rs1 >> imm[4:0] (msb-extends)
slti	I	0010011	0x2		rd = (rs1 < imm) ? 1 : 0
sltiu	I	0010011	0x3		rd = (rs1 < imm) ? 1 : 0 (zero-extends)
lb	I	0000011	0x0		rd = M[rs1+imm][7:0]
lh	I	0000011	0x1		rd = M[rs1+imm][15:0]
lw	I	0000011	0x2		rd = M[rs1+imm][31:0]
lbu	I	0000011	0x4		rd = M[rs1+imm][7:0] (zero-extends)
lhu	I	0000011	0x5		rd = M[rs1+imm][15:0] (zero-extends)
sb	S	0100011	0x0		M[rs1+imm][7:0] = rs2[7:0]
sh	S	0100011	0x1		M[rs1+imm][15:0] = rs2[15:0]
sw	S	0100011	0x2		M[rs1+imm][31:0] = rs2[31:0]
beq	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	B	1100011	0x5		if(rs1 >= rs2) PC += imm
bltu	B	1100011	0x6		if(rs1 < rs2) PC += imm (zero-extends)
bgeu	B	1100011	0x7		if(rs1 >= rs2) PC += imm (zero-extends)
jal	J	1101111			rd = PC+4; PC += imm
jalr	I	1100111	0x0		rd = PC+4; PC = rs1 + imm
lui	U	0110111			rd = imm << 12
auipc	U	0010111			rd = PC + (imm << 12)
ecall	I	1110011	0x0	imm=0x0	Transfer control to OS
ebreak	I	1110011	0x0	imm=0x1	Transfer control to debugger

Tabla 2.6: Instrucciones implementadas de *RV32I*. Fuente: [8].

A continuación, se explican las instrucciones de la tabla 2.6 con consideraciones que no se aprecian claramente en la misma. La diferencia entre *sra/srai* y *srl/srli* radica en que las primeras extienden el signo durante el desplazamiento de *rs1* (siempre el operando a desplazar), las segundas no lo hacen. Para *sra*, *srl* y *sll* se codifica el número de desplazamientos en los 5 bits menos significativos de *rs2*, mientras que *srai*, *srli* y *slli* codifican dicho número en los 5 bits menos significativos del campo *imm*. Luego, *slt/slti* y *sltu/sltiu* realizan la comparación con y sin signo, respectivamente. Por otro lado, las instrucciones *sll* y *slli* rellenan con ceros los bits menos significativos resultantes del desplazamiento. Las instrucciones de tipo *U* llenan con ceros los 12 bits menos significativos del campo *imm*. La instrucción *nop* se implementa como *addi x0, x0, 0* y, simplemente, incrementa el *PC*.

RV32I provee dos tipos de instrucciones de transferencia de control: “saltos incondicionales” (*unconditional jumps, jal/jalr*) y “bifurcaciones condicionales” (*conditional branches, todas las tipo B*). En la instrucción *jal* los saltos pueden apuntar a un rango de $PC \pm 1\text{MiB}$. En la ejecución de *jalr* se obtiene la dirección objetivo, *PC*, sumando *imm* a *rs1* y luego establece el bit menos significativo de la suma igual cero. Las instrucciones *jal/jalr* generan una excepción de “dirección de instrucción desalineada” (*instruction-address-misaligned*) si la dirección de destino no está alineada con un “límite de cuatro bytes” (*four-byte boundary*). En las instrucciones de tipo *B* el rango de la bifurcación condicional es de $PC \pm 4\text{KiB}$ y generan una excepción de *instruction-address-misaligned* si la dirección objetivo no está alineada con un límite de cuatro bytes (*four-byte boundary*) y la condición de bifurcación evaluada es verdadera. Por otro lado, la diferencia entre las instrucciones *blt/bge* y *bltu/bgeu* es que las últimas dos realizan las comparación sin signo y las primeras con.

En la tabla 2.6 se representa a la memoria como $M[\text{dirección de acceso}]$, donde las instrucciones *load-I* leen un dato en memoria y lo guardan en *rd*, mientras que las de tipo *store-S* almacenan la información del registro *rs2* en memoria. En ambos casos, se utiliza el registro *rs1* y el campo *imm* para el direccionamiento. Luego, la diferencia entre *lb/lh* y *lbu/lhu* es que los últimos no extienden el signo del *byte/halfword* leído desde memoria a 32 bits, mientras que los primeros sí lo hacen.

Independiente del *EEI*, los *loads* y *stores* cuyas direcciones efectivas están alineadas naturalmente, no generan excepción de dirección desalineada (*address-misaligned exception*) y, si no es el caso, el comportamiento depende del *EEI*. Un *EEI* puede garantizar que los desalineamientos de *loads* y *stores* sean totalmente compatibles, por lo que el *software* que se ejecuta dentro del entorno de ejecución nunca experimentará una trampa de dirección desalineada contenida o fatal. En este caso, los *loads* y *stores* desalineados se pueden manejar mediante *hardware*, o mediante una trampa invisible en la implementación del entorno de ejecución o, posiblemente, una combinación de *hardware* y trampa invisible dependiendo de la dirección. Es posible que un *EEI* no garantice que los *loads* y *stores* desalineados se manejen de manera invisible. En este caso los *loads* y *stores* que no están alineados de forma natural pueden completar la ejecución con éxito o generar una excepción. La excepción generada puede ser un *address-misaligned exception* o un *access-fault exception*. Para un acceso a la memoria que de otro modo podría completarse pero por la desalineación no es capaz, se puede generar un *access exception* en lugar de un *address-misaligned exception*, sino se debe emular el acceso desalineado, por ejemplo, si los accesos a la región de la memoria tienen efectos secundarios. Cuando un *EEI* no garantiza el manejo invisible de los desalineamientos generados por los *loads* y *stores*, el *EEI* debe definir si las excepciones causadas por la desalineación de direcciones dan como resultado una trampa contenida (permitiendo que el *software* que se ejecuta dentro del entorno de ejecución maneje la trampa) o una trampa fatal (*terminating execution*). Incluso, cuando los *loads* y *stores* desalineados se completan correctamente, estos accesos pueden ejecutarse muy lentamente según la implementación (por ejemplo, cuando se implementa a través de una trampa invisible). Además, mientras que se garantiza que los *loads* y *stores* alineados de forma natural se ejecuten de forma atómica, los *loads* y *stores* desalineados pueden no estarlo y, por lo tanto, requieren una sincronización adicional para asegurar la atomicidad.

Finalmente, las instrucciones *ecall/ebreak* provocan una trampa solicitada precisa para el entorno de ejecución. *ecall* se utiliza para realizar una solicitud de servicio, donde el *EEI* definirá cómo se pasan los parámetros para la solicitud de servicio pero, generalmente, estos estarán en ubicaciones definidas en el *Integer Register Files*. Y *ebreak* se utiliza para devolver el control a un entorno de depuración.

Es importante señalar que hay una última instrucción que forma parte de *RV32I* que no se considera relevante para la implementación. Esto es debido a que dicha instrucción, *fence*, se utiliza para funciones de ordenamiento de memoria y dispositivos de E/S, característica propia de implementaciones de mayor complejidad con múltiples *harts* en el entorno de ejecución.

RV32M - Multiply Extension					
Inst.	FMT	Opcode	funct3	extra	Descripción
mul	R	0110011	0x0	funct7=0x01	rd = (rs1 * rs2)[31:0]
mulh	R	0110011	0x1	funct7=0x01	rd = (rs1 * rs2)[63:32]
mulhsu	R	0110011	0x2	funct7=0x01	rd = (rs1 * rs2)[63:32]
mulhu	R	0110011	0x3	funct7=0x01	rd = (rs1 * rs2)[63:32]
div	R	0110011	0x4	funct7=0x01	rd = rs1 / rs2
divu	R	0110011	0x5	funct7=0x01	rd = rs1 / rs2
rem	R	0110011	0x6	funct7=0x01	rd = rs1 % rs2
remu	R	0110011	0x7	funct7=0x01	rd = rs1 % rs2

Tabla 2.7: Instrucciones implementadas de *RV32M*. Fuente: [8].

Las instrucciones *RV32M* implementadas se pueden ver en la tabla 2.7, la cual entrega una visión general de la extensión de la misma forma que la tabla anterior. Las instrucciones *mul/-mulh/mulhsu/mulhu* realizan la multiplicación entre los operandos de 32 bits, *rs1* y *rs2*. Mientras *mul* almacena los 32 bits menos significativos del resultado en *rd*. Las demás instrucciones de multiplicación, *mulh/mulhsu/mulhu*, almacenan los 32 bits más significativos. En *mul/mulh* se realiza la operación considerando ambos operandos con signo, *mulhu* considera ambos sin signo y *mulhsu* considera *rs1* con signo y *rs2* sin. Luego *div* y *divu* realizan la división entera (redondeando hacia cero) con y sin signo, respectivamente. Análogamente, *rem* y *remu* entregan el resto de dicha división, con y sin signo, respectivamente. El signo del resultado de estas últimas dos instrucciones es igual al del dividendo (*rs1*).

2.2.5.2 Unidad de punto flotante (*FPU*), *RV32F*

La unidad de punto flotante, o *FPU* por sus siglas en inglés, es la unidad encargada de realizar las operaciones de punto flotante de precisión simple bajo el estándar *IEEE 754-2008*. A diferencia de las instrucciones descritas en la sección anterior que suelen ser ejecutadas por la *ALU*, siglas en inglés para unidad aritmética lógica, la mayoría de las instrucciones que añade *RV32F* son ejecutadas por la *FPU*. Es importante destacar que por la naturaleza de los números de punto flotante, la *FPU* presenta un alto costo en ciclos de ejecución, en comparación a la *ALU*, lo que penaliza el rendimiento del sistema cuando ejecuta códigos con muchos cálculos de este tipo.

El registro *FCSR* tiene el fin de almacenar los *flags* que genera la *FPU*, estos *flags* son *NV* (operación inválida), *DZ* (división por cero), *OF* (*overflow*), *UF* (*underflow*) y *NX* (resultado

inexacto). Si bien este registro no se implementa, si se consideran dichos *flags* en el diseño de la *FPU*. Otro rol del registro es almacenar el modo de redondeo dinámico (*DYN*).

En *RV32F*, excepto cuando se indique lo contrario, si el resultado de una operación de punto flotante es *NaN*, es el canónico (*0x7FC00000*). Por otra parte, considera los modos de redondeo (*rm*) que se presentan y explican en la tabla 2.8.

Modo	Código	Descripción
RNE	000	Redondear al más cercano, o al par.
RTZ	001	Redondear hacia cero.
RDN	010	Redondear hacia abajo (hacia $-\text{inf}$).
RUP	011	Redondear hacia arriba (hacia $+\text{inf}$).
RMM	100	Redondear al más cercano, o lejos de cero.
DYM	111	Se utiliza el modo especificado en <i>FCSR</i> , el cual puede ser cualquiera de los anteriores.

Tabla 2.8: Modos de redondeo (*rm*) de *RV32F*. Fuente: [8].

En la tabla 2.9 se pueden ver las instrucciones añadidas por *RV32F*. La mayoría de ellas hacen uso exclusivos de los registros flotantes $f0 - f31$, pero hay ciertas excepciones en las que se pueden usar registros enteros $x1 - x31$ y flotantes. A continuación se detallan todas las salvedades que no se aprecian claramente en la tabla 2.9. En la tabla *rm* se refiere al modo de redondeo seleccionado.

Las instrucciones *flw* y *fsw* corresponden al *load* y *store*, respectivamente, para los registros flotantes. En ambos casos se cumple que *rs1* es entero, mientras que *rd* y *rs2* son registros flotantes para *flw* y *fsw*, respectivamente.

En cuanto a *fmin.s* y *fmax.s*, éstas consideran $-0.0 < +0.0$ y todos sus registros son flotantes. Además, si ambos operandos son *NaN* se devuelve el *NaN* canónico, si solo uno es *NaN* el resultado es el operando que no lo es. Finalmente, si algún operando es un *Signaling NaN* se levanta *NV*.

El formato *R4* se introduce para dar cabida a instrucciones que reciben 3 operandos; las instrucciones de este tipo ejecutan un multiplicación y una suma/resta, todos los registros involucrados son flotantes. Estas instrucciones levantan el *flag NV* cuando los multiplicandos son $\pm\text{inf}$ y 0, incluso cuando el sumando es un *Quiet NaN*.

RV32F - Floating-Point Extension						
Inst.	FMT	Opcode	funct3	funct7	extra	Descripción
flw	I	0000111	010			$rd = M[rs1 + imm]$
fsw	S	0100111	010			$M[rs1 + imm] = rs2$
fmadd.s	R4	1000011	rm		funct2=00	$rd = rs1 * rs2 + rs3$
fmsub.s	R4	1000111	rm		funct2=00	$rd = rs1 * rs2 - rs3$
fnmadd.s	R4	1001111	rm		funct2=00	$rd = -rs1 * rs2 + rs3$
fnmsub.s	R4	1001011	rm		funct2=00	$rd = -rs1 * rs2 - rs3$
fadd.s	R	1010011	rm	0000000		$rd = rs1 + rs2$
fsub.s	R	1010011	rm	0000100		$rd = rs1 - rs2$
fmul.s	R	1010011	rm	0001000		$rd = rs1 * rs2$
fdiv.s	R	1010011	rm	0001100		$rd = rs1 / rs2$
fsqrt.s	R	1010011	rm	0101100	rs2=00000	$rd = \text{sqrt}(rs1)$
fsgnj.s	R	1010011	000	0010000		$rd = \text{abs}(rs1) * \text{sgn}(rs2)$
fsgnjs	R	1010011	001	0010000		$rd = \text{abs}(rs1) * -\text{sgn}(rs2)$
fsgnjx.s	R	1010011	010	0010000		$rd = rs1 * \text{sgn}(rs2)$
fmin.s	R	1010011	000	0010100		$rd = \text{min}(rs1, rs2)$
fmax.s	R	1010011	001	0010100		$rd = \text{max}(rs1, rs2)$
fcvt.s.w	R	1010011	rm	1101000	rs2=00000	$rd = (\text{float}) rs1$
fcvt.s.wu	R	1010011	rm	1101000	rs2=00001	$rd = (\text{float}) rs1$
fcvt.w.s	R	1010011	rm	1100000	rs2=00000	$rd = (\text{int32}_t) rs1$
fcvt.wu.s	R	1010011	rm	1100000	rs2=00001	$rd = (\text{uint32}_t) rs1$
fmv.x.w	R	1010011	000	1110000	rs2=00000	$rd = *((\text{int}^*) \&rs1)$
fmv.w.x	R	1010011	000	1111000	rs2=00000	$rd = *((\text{float}^*) \&rs1)$
feq.s	R	1010011	010	1010000		$rd = (rs1 == rs2) ? 1 : 0$
flt.s	R	1010011	001	1010000		$rd = (rs1 < rs2) ? 1 : 0$
fle.s	R	1010011	000	1010000		$rd = (rs1 \leq rs2) ? 1 : 0$
fclass.s	R	1010011	001	1110000	rs2=00000	$rd = 0..9$

Tabla 2.9: Instrucciones implementadas de *RV32F*. Fuente: [8].

Las instrucciones de conversión entre un número en punto flotante y un entero usan ambos tipos de registros. En el caso de *fcvt.s.w* y *fcvt.s.wu* se realiza la conversión del número entero, con y sin signo, respectivamente, del registro entero *rs1* a un número en punto flotante y lo almacena en el registro flotante *rd*. Por otra parte, *fcvt.w.s* y *fcvt.wu.s* convierten el flotante ubicado en el registro flotante *rs1* a un entero, con y sin signo, respectivamente, y lo almacenan en el registro entero *rd*. En cualquier caso, si el resultado no es representable en el formato de destino, se devuelve el valor más cercano y se levanta *NV*. Se realiza el redondeo de acuerdo al *rm* especificado en el campo *funct3* de la instrucción.

Para mover registros desde los registros enteros a los flotantes y viceversa, sin realizar conversión alguna, se utilizan las instrucciones *fmv.w.x* y *fmv.x.w*, respectivamente. En *fmv.w.x* se tiene que *rd* es un registro flotante y *rs1* entero, mientras que para *fmv.x.w* es al revés.

Bit rd	Descripción de $rs1$
0	$-inf$
1	< 0 (<i>Normalizado</i>)
2	< 0 (<i>Desnormalizado</i>)
3	-0
4	$+0$
5	> 0 (<i>Desnormalizado</i>)
6	> 0 (<i>Normalizado</i>)
7	$+inf$
8	<i>Signaling NaN</i>
9	<i>Quiet NaN</i>

Tabla 2.10: Posibles formatos para $fclass.s$. Fuente: [8].

Por otro lado, $feq.s$, $flt.s$ y $fle.s$ realizan la comparación con los registros $rs1$ y $rs2$ flotantes, pero almacenan el resultado en uno entero (rd). Además, $flt.s$ y $fle.s$ realizan una comparación *signaling*, es decir, levantan NV cuando $rs1$ o $rs2$ es NaN . Sin embargo, $feq.s$ realiza una comparación *quiet*, o sea, levanta NV cuando algún operando en un *Signaling NaN*. Basta que algún operando sea NaN para que el resultado sea 0.

Finalmente, $fclass.s$ examina el número en punto flotante, ubicado en el registro flotante $rs1$, y guarda en el registro entero rd una máscara de 10 bits que indica la clase a la que pertenece el operando. En la tabla 2.10 se puede ver qué bit (siempre es sólo uno) se debe “encender” en dicha máscara según la clasificación correspondiente. El resto de instrucciones sólo utilizan registros flotantes para sus operaciones.

2.3 Revisión del estado del arte

Otros autores ya han enfrentado la problemática del diseño e implementación de un sistema basado en *RISC-V*, por ejemplo en [15] se presenta el diseño y el análisis de la microarquitectura de procesador utilizando el conjunto de instrucciones de *RISC-V*, en [16] se estudia la implementación en *FPGAs* de núcleos basados en *RISC-V*. Los autores de [17] exponen la implementación de un *PLC* (*Programmable Logic Controller*) en un *FPGA* para la Industria 4.0 basándose en *RISC-V*. En [18] se presenta la implementación de un entorno de diseño y verificación para núcleos de procesadores *RISC-V* y en [19] se desarrolla una metodología de diseño para lenguajes *HDL*. A continuación se presenta una revisión de los trabajos recién mencionados y un ejemplo comercial de una implementación real basada en *RISC-V* por parte de la empresa *SiFive*.

2.3.1 A *RISC-V* Instruction Set Processor-Micro- architecture Design and Analysis

En [15] se presenta el diseño un procesador de 32 bits *single-core* basado en el *ISA* de *RISC-V*. El diseño se realiza utilizando el simulador *Blue-spec System Verilog* y se implementa en una plataforma *FPGA* y nodos de tecnología de *65nm* y *130nm* para *ASIC* (*Application-specific integrated circuit*).

Los autores adoptan una arquitectura *RISC* típica *single issue*, *pipelined* de 5 etapas (*Fetch*, *Decode*, *Register Select*, *Execute* y *Write Back*), con *in-order fetch*, *out-of order execution* e *in-order completion*. La arquitectura se puede ver en la figura 2.7. Por otro lado, la verificación de la *FPU*

implementada se realiza mediante *test benches* con *pseudo random test vectors* generados en *C*, y el *core* se verifica mediante archivos hexadecimales generados a partir de código ensamblador escrito a mano.

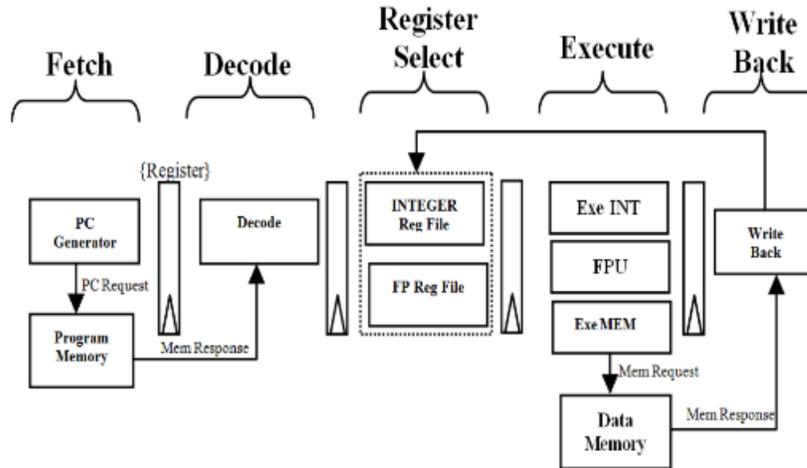


Figura 2.7: Arquitectura de nivel superior del procesador *RISC-V*. Fuente: [15].

En la etapa *fetch* se obtiene la instrucción desde la “memoria de instrucciones” a partir de la dirección indicada por el *PC*. Luego, se envía la instrucción a la etapa de decodificación (*decode*) para la extracción de información de sus campos. La información extraída se envía a la etapa de *register select* (*RS*). Desde la etapa *RS* el “*pipeline*” se divide en tres unidades *pipelines* simultáneas. Las instrucciones *integer* pasan por el *integer execution pipeline*, las instrucciones de memoria pasan por la *memory pipeline* y las instrucciones de coma flotante por la unidad *FP execution pipeline unit*. La etapa *Write Back* (*WB*) recibe las respuestas de estas tres etapas de ejecución simultáneas y, según la planificación de la instrucción, la etapa *WB* permite confirmar las instrucciones en orden. A continuación se explica cada una de estas etapas.

A- *Fetch Stage*

En la figura 2.8 se presenta la microarquitectura de la *fetch unit*. Esta unidad se encarga de computar/calcular el *PC* actual y de predecir el siguiente valor del *PC*. La unidad tiene un registro de 32 bits, el cual se encarga de almacenar el valor actual del *PC*, y un sumador de 32 bits que calcula el siguiente valor del *PC* sumando 4 al valor actual. El siguiente *PC* se predice mediante $PC + 4$ o utilizando un sofisticado esquema de predicción de bifurcaciones (necesario para el manejo de *Control Hazards*).

El esquema de predicción de bifurcaciones (*branch prediction scheme*) consiste en dos unidades, *Branch Target Buffer* (*BTB*) y *Branch Predication* (*BP*), unidad “búfer de destino de bifurcación” y “predicción de bifurcación”, respectivamente. La unidad *BTB* emite una señal booleana, *valid*, junto con la dirección *PC* de destino. *BP* entrega información booleana (*prediction*) que indica si el valor de *PC* está presente o no. Un multiplexor selecciona entre $PC + 4$ o el *PC* predicho como el siguiente *PC*, en base de las señales *valid* y *prediction*. El *PC* actual y siguiente se entregan a la etapa *Decode Stage* para su posterior procesamiento.

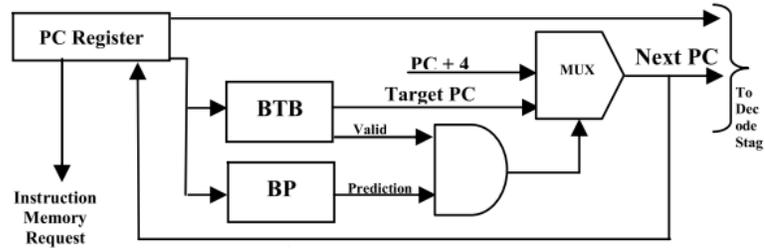


Figura 2.8: microarquitectura de la *fetch unit*. Fuente: [15].

B- Decode Stage

El decodificador de instrucciones recibe la instrucción desde la memoria de programa con su correspondiente *PC* y *PC* predicho desde la etapa de *fetch*. En la figura 2.9 se muestra el diagrama del decodificador de instrucciones. Basándose en los 7 bits menos significativos de la instrucción (*opcode*), las instrucciones se clasifican en uno de los grupos principales de instrucciones (*Branch Inst.*, *Load Inst.*, *Store Inst.*, *Arithmetic and Logic Inst.*, *Atomic Operations*, *System Inst.*, *Special, unconditional jump*, *Synchronization*). Luego, se extrae el sub-grupo de la instrucción a partir del campo de 3 bits, *function field*. Las direcciones de los registros de fuente y destino están codificadas en campos de 5 bits cada uno y su posición es fija, independientemente del tipo de instrucción. Si se ha producido una instrucción de tipo inmediata, los datos inmediatos se extienden por signo a 32 bits para su posterior procesamiento.

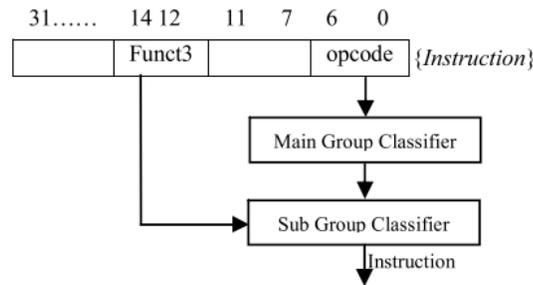


Figura 2.9: Diagrama del decodificador de instrucciones. Fuente: [15].

En la figura 2.10 se presenta la microarquitectura del decodificador de instrucciones. El *Op-code* se proporciona al clasificador de grupo principal, el cual proporciona la clase del grupo principal como salida. El clasificador de grupo principal tiene 7 comparadores en paralelo (*7-bit comparator*) para decodificar la clase del grupo principal. El *Op-code* de la instrucción se compara con 7 categorías de *Op-codes* al mismo tiempo. Si la salida de algún comparador se 'enciende/activa' (*goes high*), se genera la señal del grupo principal correspondiente, de lo contrario, la instrucción que se decodifica se considera ILEGAL. Además, el clasificador de sub-grupos determina la operación exacta que debe realizar la instrucción. El sub-grupo se determina mediante un campo de 3 bits (*function field*) y el grupo principal. Si el campo de 3 bits, *function field*, de la instrucción concuerda con el campo de 3 bits de la clase principal decodificada, el clasificador de sub-grupos genera la categoría de sub-grupo correspondiente bajo la clase principal decodificada, de lo contrario, la instrucción cae bajo la categoría de ILEGAL.

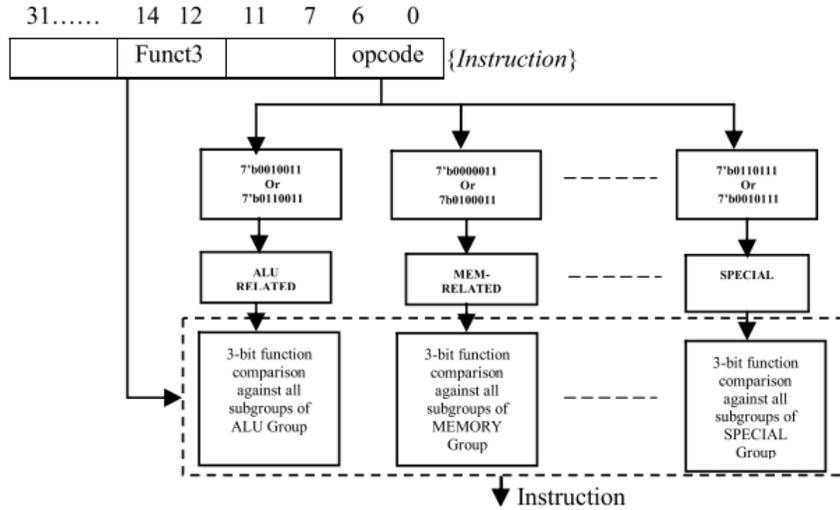


Figura 2.10: Microarquitectura del decodificador de instrucciones. Fuente: [15].

El decodificador genera señales de control para el procesamiento posterior de la instrucción. Las señales de control incluyen: *Register Access*, *Register update* y *Pipeline information* (el *pipeline* en la cual se programará la instrucción). *Register Access* puede ser acceso a archivos de registros enteros (*INT-ACCESS*), acceso a registros de punto flotante (*FP-ACCESS*) o sin acceso (*NO-ACCESS*). *Register update* indica qué archivo de registro debe volver a escribirse después de ejecutar la instrucción. El procesamiento de instrucciones se organiza a través de dos *pipelines* independientes para el cálculo de *ALU* de enteros / direcciones de memoria y *FPU*. La señal *EXPIPE* indica el *pipeline* a través del cual se debe procesar la instrucción.

Además de decodificar instrucciones, en esta etapa se detecta dependencia de datos (*Data Hazard Detection*) entre instrucciones. La información decodificada se reenvía hacia la etapa *Register Select Stage* en forma de un paquete junto el *PC* actual y predicho.

C- Register Select Stage

La etapa *RS* recibe la información codificada de las instrucciones desde la etapa anterior y selecciona los operandos ya sea desde un archivo de registro de punto flotante o entero. Esta etapa contiene un archivo de registros enteros (32 registros de 32 bits de ancho) y de punto flotante (32 registros de 64 bits de ancho). El archivo de registros (*Register file*) se implementa como una *RAM* (*Random Access Memory*), que tiene una latencia de un ciclo de reloj con tres puertos de lectura y un puerto de escritura.

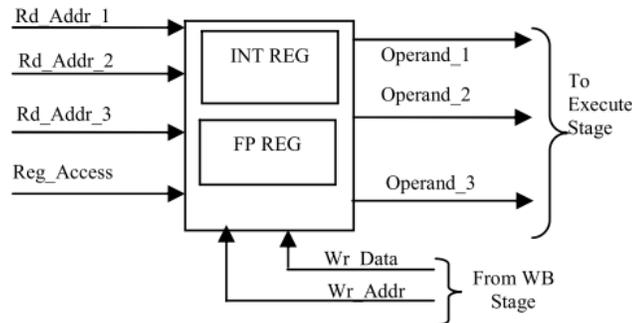


Figura 2.11: Organización del *Register File*. Fuente: [15].

En la figura 2.11 se muestra la organización del *Register file*. La unidad de *Register file* recibe tres direcciones de origen (*Rd_Addr_1*, *Rd_Addr_2* y *Rd_Addr_3*) y una señal de control (*Reg_Access*) que especifica el acceso a los archivos de registro (acceso entero o acceso de punto flotante). La salida del *Register file* es de 64 bits que contiene los datos del *Register file* de punto flotante o entero y tiene un puerto de escritura de 64 bits para el *write back*.

Basado en la información del *pipeline* proveniente de la etapa de decodificación, los operandos después del *Register Select* se entregan al *pipeline* de cálculo de *ALU* de enteros / direcciones de memoria o al *pipeline* de ejecución *FP* (*Floating Point*). Un programador de instrucciones *FIFO* (*First Input First Output*), que es parte de la etapa de ejecución de enteros/memoria/*FPU*, se utiliza para programar las instrucciones en los diversos *pipelines* y confirmar las instrucciones en orden.

D- Execute Stage

Esta etapa consta de tres unidades simultáneas para operaciones aritméticas y lógicas enteras, etc., cálculo de direcciones de memoria de operandos y cálculo de coma flotante. La ejecución de enteros realiza operaciones aritméticas (suma, resta, multiplicación y división) y lógicas (*AND*, *OR*, *XOR* y desplazamiento). Además, la unidad aritmética entera calcula la dirección de destino para instrucciones de salto y bifurcación incondicional o condicional. La unidad de ejecución de enteros (*integer execution unit*) ejecuta las instrucciones relacionadas con el sistema, como las instrucciones *SCALL*, *SBREAK* para el acceso de nivel supervisor de la instrucción. Se utiliza un esquema *forwarding* de envío de datos eficiente para enviar la salida de las unidades de ejecución a la entrada de la misma. Por otro lado, las instrucciones relacionadas con la unidad de memoria calculan la dirección de la memoria de datos de destino para las operaciones de *load* y *store*. La microarquitectura de la unidad de ejecución *FP* sigue un esquema *out-of-order*. Como no se busca una implementación compleja en el caso del trabajo presentado en este informe, no se considera la posibilidad de implementar una *FPU* bajo dicho esquema.

E- Write Back Stage

La etapa *WB* confirma la instrucción desde el *pipeline* y actualiza el archivo de registros con los resultados de las unidades de ejecución. *WB* lee la instrucción desde la parte superior de la instrucción programada *FIFO* y, en base a la información del *pipeline*, lee unidades concurrentes de números enteros, de memoria o de coma flotante.

2.3.2 Open-Source RISC-V Processor IP Cores for FPGAs – Overview and Evaluation

En [16] se plantea la importancia que han adquirido los *FPGAs*, gracias a que cada vez pueden implementarse diseños más complejos debido a la reducción en los tamaños de los transistores que lo conforman y a la mejora de rendimiento con respecto a las frecuencias de reloj que llegan a conseguir. Esto junto a los acotados plazos de desarrollo del mercado actual, impulsa a que cada vez más microprocesadores sean implementados en este tipo de dispositivos a escala comercial.

El trabajo de los autores es un análisis y pruebas, usando un *FPGA*, de *CPUs* de *Hardware Libre*, centrándose en *RISC-V*. En el texto hablan de “*CPU IP core*”, *IP* viene de *intellectual property*, los *IP cores* comerciales representan una metodología para la implementación de tecnología en cortas ventanas de tiempo de desarrollo y son la base de diseño de múltiples *SoCs*, por lo que estudian los alcances de la propiedad intelectual del *Hardware Libre* como tal.

En el texto estudiado también se señalan ciertas consideraciones a tomar en cuenta a la hora de utilizar *CPU IP cores* comerciales:

- Los *CPU cores* comerciales existentes de los proveedores de FPGA suelen depender de la tecnología. Por lo tanto, no es posible trasladar el diseño a dispositivos de otros proveedores de FPGA sin cambiar también el *CPU core*.
- Por otro lado existen familias de *FPGA* con *CPU cores* cableados (*hard-wired*) (por ejemplo, la arquitectura ARM Cortex). Estos *CPU cores* brindan un rendimiento excelente (frecuencias de reloj superiores a 1 GHz), pero los costos se elevan.
- Para el caso anterior, es prácticamente una necesidad agregar una memoria externa al chip *FPGA*, debido a que la RAM incorporada en dicho chip es insuficiente para el *CPU core* cableado del sistema; lo que aumenta los costos.

Los autores mencionan las cualidades del *ISA* libre de *RISC-V* y destacan que realizando búsquedas de texto completo de palabras clave significativas de los últimos 5 años (desde la fecha de la publicación en 2019), ha disminuido la cantidad de menciones de *commercial soft CPU IP core* y aumentado las menciones de *open-source CPU IP core*, en especial las de *RISC-V*. En la figura 2.12 se puede ver la gráfica con estos datos.

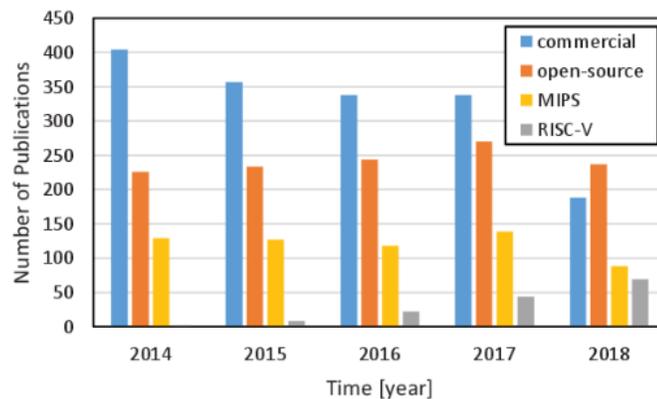


Figura 2.12: Número de menciones en las publicaciones de diferentes *CPU IP cores*. Fuente: [16].

2.3.3 *RISC-V based implementation of Programmable Logic Controller on FPGA for Industry 4.0*

Los autores de [17] implementan un *PLC* (*Programmable Logic Controller*), como producto para la industria 4.0 y el *IIoT* (*Industrial Internet of Things*), basándose en *RISC-V* y utilizando un *FPGA* como plataforma. Un *PLC* es un dispositivo programable enfocado a reemplazar las antiguas técnicas de control industrial basados en relés. Como se indica en el texto, cada vez se les exigen más características y capacidades, pero los productores de *PLCs* no han ahondado en mejorar el mecanismo por el cual ejecutan sus programas; por lo que el foco de su trabajo recae en optimizar dicho mecanismo. Por otro lado, comentan los diversos lenguajes que se utilizan para manejar un *PLC*; *Instruction List (IL)*, *Structured Text (ST)*, *Ladder Diagram (LD)* y *Function Block Diagram (FBD)*.

La industria 4.0 requiere la gestión de las conexiones entre sensores, *PLCs* y los dispositivos computacionales en la nube. Por lo que los autores proponen que al integrar los *PLCs* en un *FPGA* se abra la puerta para poder integrar otros componentes en el mismo chip (osea un *SoC*); esto permitiría el paso a la próxima generación de la industria de fabricación, donde se necesita una

puerta de enlace entre la tecnología de la información (*Information Technology (IT)*) y la tecnología operativa (*Operation Technology (OT)*).

En la figura 2.13 se puede apreciar la arquitectura propuesta para el *PLC*; es un diagrama de bloques que ilustra los componentes básicos propuestos por parte de los autores.

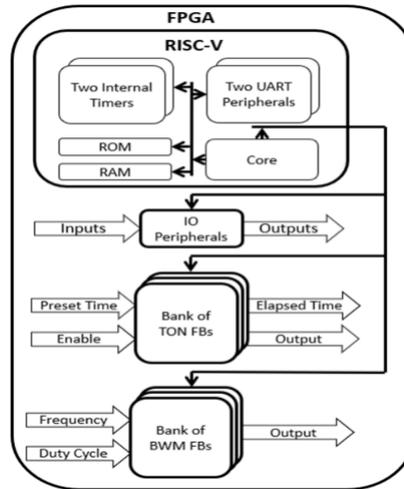


Figura 2.13: La estructura de la arquitectura PLC propuesta. Fuente: [17].

2.3.4 Design and Verification Environment for RISC-V Processor Cores

En [18] se desarrolla un entorno de diseño y verificación orientado a *RISC-V*. El objetivo de estos entornos es comprobar que cada una de las unidades funcionales del procesador se comuniquen correctamente entre si. Esto con el fin de disminuir complejidad y tiempos en la etapa de verificación.

En el texto se comentan las ventajas de *RISC-V* al ser *Hardware Libre* bajo la licencia “*Berkeley Software Distribution (BSD) open-source license*”. También, se destaca el potencial que representa la comunidad de *RISC-V*, la cual provee soporte de *software* como; *compiler toolchains*, *integrated Development Environment (IDA)*, sistemas operativos orientados a sistemas embebidos y un *framework* para la verificación formal (es decir que usa métodos matemáticos para comprobar el correcto funcionamiento del diseño) llamado *riscv-formal*, el cual utiliza el *RISC-V Formal Interface (RVFI)*. Por otro lado, se recalca la estabilidad de *RISC-V*, debido a que sus *ISAs* base están congeladas. Además, se destaca el apoyo a la fundación de *RISC-V*, sin fines de lucro, con miembros como *Google*, *Nvidia* o *Qualcomm* (entre más de 100), lo que indica la estabilidad del *ISA* y el impacto de este en el futuro cercano.

Los autores proponen un modelo que consta de dos unidades: un emulador que actúe como *golden reference model* y un *disassembler* (traduce lenguaje de máquina a lenguaje ensamblador), ambos escritos en *Verilog*. El emulador se conecta con el flujo de instrucciones ejecutadas por el modelo del procesador, si ocurre alguna inconsistencia es reportada por el emulador. Mientras, el *disassembler* traduce y muestra las instrucciones para facilitar el proceso de depuración.

2.3.5 Co-verification design flow for HDL Languages

En [19], los autores presentan una metodología de trabajo enfocada al diseño de sistemas complejos en *HDL* sobre *FPGAs*. Indican que cerca del 60% del tiempo del diseñador se utiliza en realizar pruebas. Por ello, proponen una metodología de diseño y prueba que reduzca el tiempo ocupado en

simulaciones y verificaciones.

En la figura 2.14 se puede apreciar el típico desarrollo de este tipo de diseños. El problema que se presenta es que el modelo del sistema (*System Model*) debe concordar con el modelo *HDL*; pero estos códigos/modelo pueden evolucionar constantemente por lo que se pueden llegar a presentar múltiples fallas o *bugs*, en especial en procesos de desarrollo ágiles.

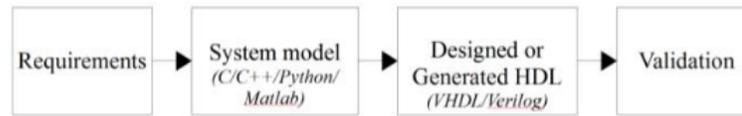


Figura 2.14: Proceso de desarrollo típico. Fuente: [19].

El enfoque propuesto para resolver este problema se puede apreciar en la figura 2.15, consiste en el uso de herramientas y metodologías para simplificar el diseño de código en *HDL* a partir del *System Model* y la co-verificación automática para asegurar que la implementación se comporte correctamente. Sin embargo, se enfocan en diseños *fixed-point*

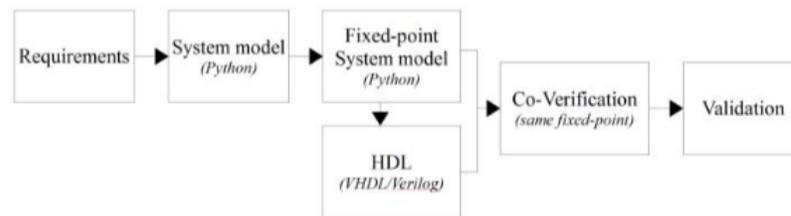


Figura 2.15: Proceso de desarrollo de co-verificación. Fuente: [19].

2.3.6 Ejemplo de una implementación real: *HiFive Rev B*

HiFive Rev B es una tarjeta de desarrollo, similar a *Arduino*, de la compañía *SiFive*. Se puede ver en la figura 2.16. El chip que controla esta tarjeta es el *SiFive FE310-G002* y su procesador está basado en *RISC-V*.

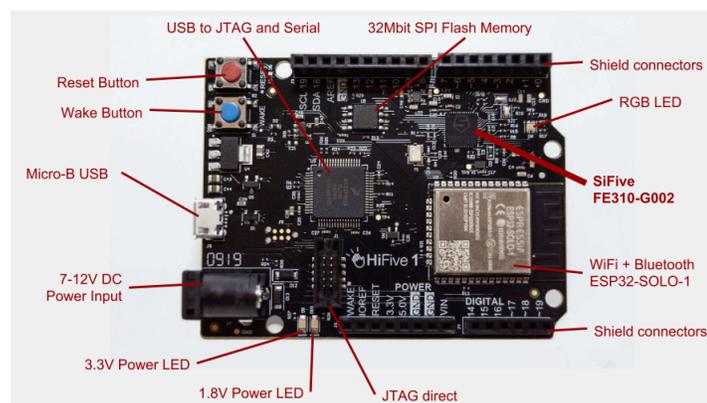


Figura 2.16: Tarjeta *HiFive1 Rev B*. Fuente: [20].

La gracia de esta tarjeta es que posee conectividad inalámbrica (*WiFi* y *Bluetooth*) y *pines* de conexión *serial/GPIO/I2C/SPI* para comunicarse (ver figura 2.17). Sin embargo, esta tarjeta no incluye cálculo de punto flotante mediante *hardware* [20].

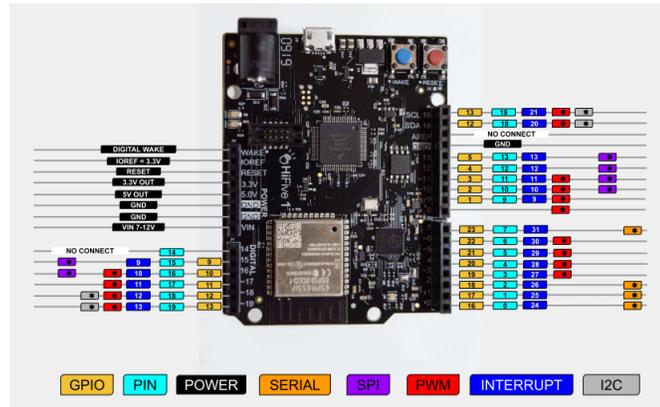


Figura 2.17: *Pinout* de la tarjeta *HiFive1 Rev B*. Fuente: [20].

Por otro lado, en [21] se dispone del *datasheet* del chip *SiFive FE310-G002*, donde se puede apreciar el diagrama de bloques de nivel superior del mismo (ver figura 2.18). Este es un buen ejemplo de implementación de un *SoC*. En [22] se puede verificar que el diseño realizado consta de 5 etapas *pipelined*.

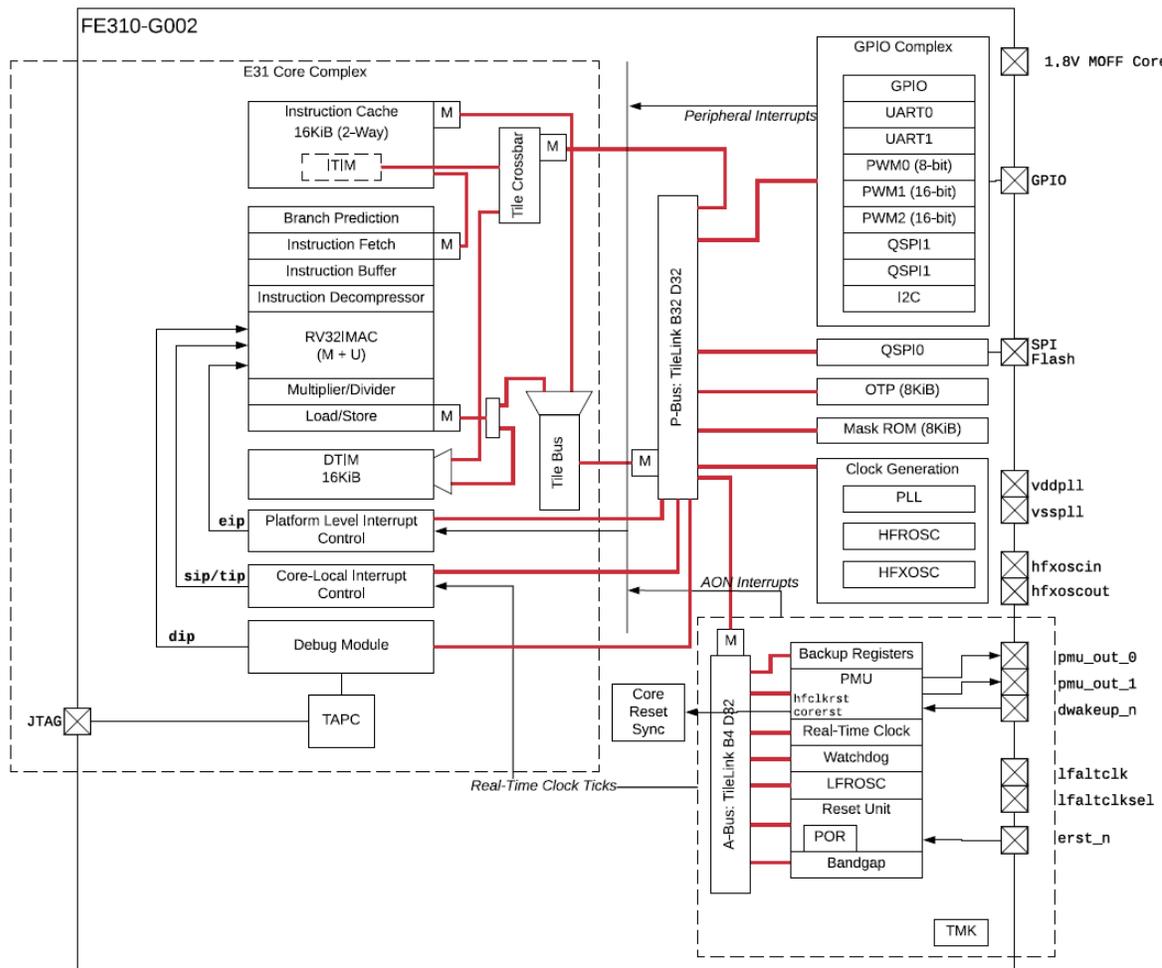


Figura 2.18: Diagrama de bloques de nivel superior del chip *SiFive FE310-G002*. Fuente: [21].

2.4 Tarjeta *FPGA* y *software* utilizado

En esta sección se presentan las herramientas utilizadas en el proceso de desarrollo, verificación e implementación de los diseños digitales obtenidos. Para la implementación, como ya se ha mencionado, se dispone de una tarjeta de desarrollo *FPGA*, en este caso la *Nexys 4* [23]. Los diseños digitales a implementar se desarrollan y verifican utilizando un lenguaje de programación especializado de tipo *HDL* (*hardware description language*), utilizado para describir *hardware*. Para este tipo de lenguajes hace falta un *software* de desarrollo especializado en la síntesis, verificación e implementación de diseños digitales, en este caso se trata de *Vivado Design Suite*, proveído por *Xilinx* [24]. Finalmente, para la escritura y verificación de código ensamblador para *RV32IMF* se dispone de la herramienta *RARS 1.5* [25].

2.4.1 *Nexys 4*

Un *FPGA* (*Field Programmable Gate Arrays*) es, en palabras simples, una pieza de *hardware* fabricada en silicio capaz de reconfigurarse dinámicamente, es decir que es capaz de cambiar su estructura interna (la conexión de los transistores que la componen), lo que permite implementar diferentes diseños de *hardware* en el mismo chip y actualizarlos de ser necesario. Tal es la versatilidad de esta tecnología que en [7] se habla de su importancia y su posible papel revolucionario en el internet de las cosas (*IoT*) junto al venidero 5G, y es gracias a esta misma versatilidad que habilita implementar actualizaciones a nivel de *hardware* para el manejo de los nuevos estándares de comunicaciones y su posibilidad de personalización. También, evita la necesidad de invertir en un nuevo *hardware* específico para el manejo de estas nuevas tecnologías y permite el desarrollo de aceleradores, los cuales se podrían implementar en base al *ISA* libre de *RISC-V*.

La tarjeta utilizada corresponde a la *Nexys 4* (figura 2.19), fabricada por *Digilent* y basada en el chip *FPGA Artix-7 100T* de *Xilinx*. Esta tarjeta es una plataforma de desarrollo de circuitos digitales completa y lista para usar que cuenta con generosas memorias externas y varios periféricos incorporados que permiten el uso de esta tarjeta en una amplia gama de diseños sin necesidad de otros componentes, se destacan [23]:

- 16 interruptores de usuario.
- 16 *LED* de usuario.
- 2 *LED* tri-color.
- 2 pantallas de 4 dígitos de 7 segmentos.
- 5 pulsadores + 1 de reinicio *CPU* + 1 de reinicio de configuración *FPGA*
- Puerto *USB UART/JTAG* compartido.
- Micrófono (*PDM*).
- Conector *VGA* de 12 bits.
- Conector de audio (+ salida de audio *PWM*).
- Conector *Ethernet*.
- Conector de host *USB*.
- Sensor de temperatura.
- Acelerómetro de 3 ejes.
- 16MB de *Cellular RAM*.
- Conector de tarjeta *Mirco SD*.

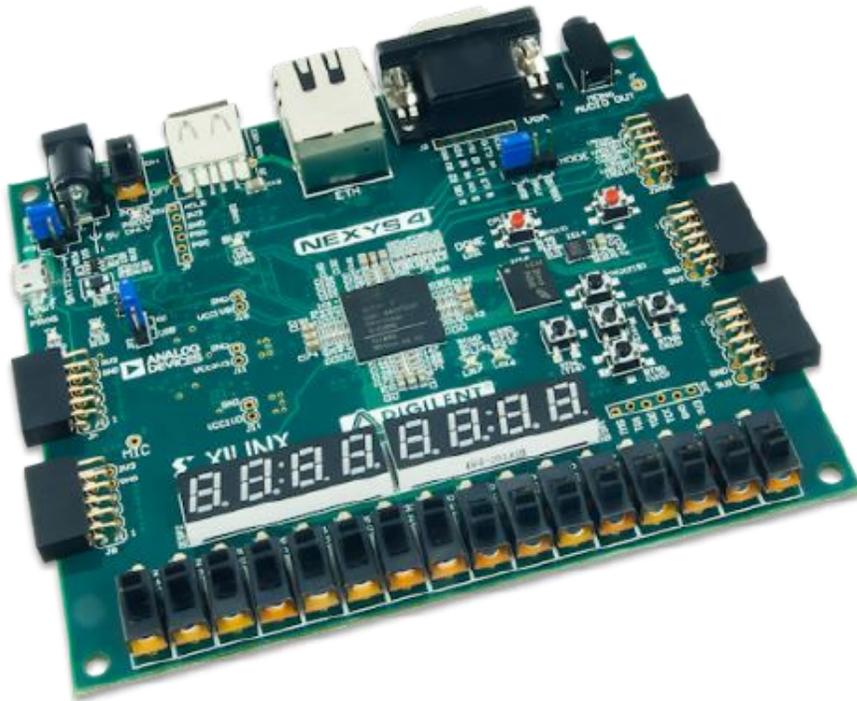


Figura 2.19: Tarjeta utilizada, la *Nexys 4*. Fuente: [23].

Por otra parte, es importante destacar que el *Artix-7 100T* cuenta con:

- 15.850 *logic slides*, cada uno con cuatro *6-input LUTs* (*look-up tables*) y 8 *flip-flops*.
- 4.860Kb de *fast block RAM*.
- 240 *DSP slices* (*Digital Signal Processing*).
- Un *analog-to-digital converter* (*XADC*) en el chip.
- Velocidades de reloj interno superiores a 450MHz.
- 6 *clock management tile* (*CMT*), cada uno con *phase-locked loop* (*PLL*).

2.4.2 *Vivado Design Suite* de *Xilinx*

Tal como se indica en [23], la *Nexys 4* es compatible con la herramienta *Vivado Design Suite*, *software* desarrollado por *Xilinx*, compañía inventora de la tecnología *FPGA* [24]. Esta *suite* de desarrollo, *Vivado*, entrega herramientas de síntesis, simulación e implementación de diseños escritos en *HDL*, por lo que su función es tomar el código fuente con dichos diseños para sintetizarlo en un circuito digital viable. A partir de dicha síntesis, se puede realizar una implementación en la misma *suite* para el chip *FPGA* deseado, garantizando la compatibilidad del diseño sintetizado con el *FPGA* utilizado. Posteriormente, se genera el archivo (*bitstream*) que permite cargar los diseños en el *FPGA*. Por otro lado, para crear pruebas de verificación (*test benches*) se usa el mismo lenguaje *HDL* y *Vivado* permite ejecutar simulaciones junto con la visualización de la evolución de todas las señales del sistema en una línea de tiempo.

Dentro de los lenguajes *HDL* que *Vivado 2020.2* (versión utilizada) admite, se encuentran: *VHDL*, *Verilog* y *System Verilog*. El lenguaje utilizado para el proyecto es *System Verilog*, el cual se encuentra definido por la *IEEE* en [26]. Se usa este lenguaje debido a su simplicidad en comparación con *VHDL* y por ser una actualización de *Verilog*.

2.4.3 *RARS 1.5*

Para la elaboración de código de prueba para el *SoC* se utiliza *RARS*. Esta herramienta, desarrollada por *Pete Sanderson* y *Kenneth Vollmar*, es un simulador de *RISC-V*. La última versión es la 1.5 y se encuentra disponible en *GitHub* en formato “*.jar*”. *RARS* permite escribir instrucciones de máquina, ensamblador o *assembler*, y verificar su funcionamiento a nivel de registros y memoria. Además, presenta una amplia documentación, integrada en la misma aplicación, de las instrucciones de *RISC-V* [25].

Este *software* se utiliza para contrastar el resultado de un código de prueba en las simulaciones de *Vivado* y en *hardware* con el de *RARS*. Gracias a la posibilidad de ejecutar instrucción por instrucción, ver los accesos a memoria y los accesos de registros en *RARS*, se facilita la etapa de depuración de la *CPU* y del *SoC*.

Otra importante función que entrega *RARS* es la posibilidad de compilar código ensamblador en diferentes formatos y exportarlo, como puede ser texto plano en formato hexadecimal o binario. Si bien existen herramientas, *toolchains*, gratuitas proveídas por la fundación *RISC-V* (disponibles en *GitHub*) específicas para compilar código escrito en *RISC-V*, estas están pensadas para implementaciones de mayor complejidad arquitectónica y no resultan útiles ni cómodas para trabajar en la implementación. Por ejemplo, para obtener el código ensamblador compilado en un archivo de texto con cada instrucción en formato hexadecimal o binario se debe escribir un código que ejecute comandos de compilación especiales, además de tener que instalar el *toolchain*, lo cual termina siendo un proceso muy laborioso en comparación a las funciones que ofrece *RARS*. Esto sin mencionar que el *toolchain* suele asumir soporte para sistemas operativos, lo que añade líneas de código extra (innecesarias y molestas en este caso) para el manejo de múltiples *harts* y del entorno de ejecución.

3 Metodología de trabajo

La metodología de trabajo empleada para el desarrollo de los diseños del *SoC* se basa en el enfoque *Top-Down*. Por otra parte, para la metodología de verificación de los diseños se opta por el método presentado en la sección 2.3.1. A continuación, se explica la metodología de diseño *Top-Down* estudiada en [10], el flujo de trabajo y los detalles de la metodología de verificación empleada.

3.1 Metodología de diseño *Top-Down*

Esta es una filosofía/metodología de diseño para sistemas digitales grandes y complejos que permite el trabajo en equipo (factor que no se aprovecha en este trabajo). Entre sus ventajas se destaca [10]:

- Permite el manejo de diseños grandes y complejos.
- Permite esfuerzos simultáneos e independientes.
- Permite eliminar tempranamente problemas serios de la arquitectura.
- Identifica tempranamente módulos reusables.

Y entre sus desventajas se puede destacar que; hay ciertas decisiones que no se pueden tomar sin tener alguna implementación y que podrían significar el doble de esfuerzo.

El enfoque de esta metodología de diseño se puede apreciar en la figura 3.1, donde se propone un modelo compuesto por señales externas de entrada, salida, e internas de control y decisión, más un sistema controlado y uno controlador.

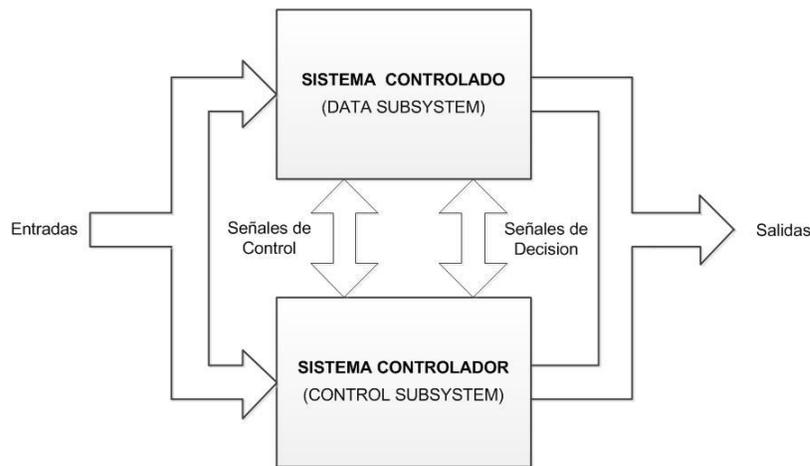


Figura 3.1: Enfoque Diseño Contemporáneo (*Top-Down*). Fuente: [10].

Como se puede desprender de lo anterior, la idea es separar el sistema controlado del controlador y que estos se comuniquen mediante señales internas (control y decisión). La metodología *Top-Down* considera la realización de diagramas simplificados y detallados de la siguiente forma:

1. Primero se realiza un diagrama de bloques simplificado, donde se identifica cada uno de los bloques/componentes necesarios para el funcionamiento del sistema.
2. Luego, a partir del diagrama anterior, se elabora un diagrama de flujo simplificado; este diagrama muestra de forma sencilla la lógica (algoritmo) del sistema controlador y su interacción con el sistema controlado.
3. A partir del diagrama de bloques y flujo simplificados se elabora el diagrama de bloques detallado, donde se especifican cada una de las señales E/S y de control/decisión pertinentes.

4. Una vez detallado el diagrama de bloques se utiliza, junto al diagrama de flujo simplificado, para elaborar un diagrama de flujo detallado, donde se consideran las señales identificadas en el punto anterior.
5. A partir de este último diagrama de flujo detallado se construye un diagrama *MDS* (*Mnemonic Documented State*); este diagrama muestra de forma clara cada uno de los estados posibles del sistema, las señales de transición y las señales que se generan en cada estado de la *FSM* del sistema controlador.
6. Una vez obtenido el diagrama *MDS* se pasa a la fase de implementación, simulación y pruebas mediante el uso de algún lenguaje *HDL*.

3.2 Flujo de trabajo

El flujo de trabajo empleado se puede separar en las siguientes etapas:

- **Etapa de planteamiento** inicial, aquí se define la arquitectura del *SoC* a nivel general. Se identifican los submódulos del mismo, del entorno de ejecución y del procesador.
- **Etapa de diseño** de cada componente identificado anteriormente, utilizando la metodología *Top-Down*.
- **Etapa de verificación** que consta de pruebas/simulaciones para los diseños obtenidos en la etapa anterior.
- **Etapa de rediseño** para mejorar rendimiento y/o solucionar problemas encontrados en la etapa anterior.
- **Etapa de iteración** de las etapas de verificación y rediseño hasta que cada uno de los componentes identificados en la etapa de diseño funcione correctamente.
- **Etapa de integración** de todos elementos del *SoC* y verificación del funcionamiento del *SoC* mismo. De ser necesario, se vuelve a la etapa de iteración considerando los módulos que presenten fallas durante las simulaciones.
- **Etapa de implementación** física en el *FPGA* con sus pruebas de funcionamiento. Si hace falta se vuelve a la etapa de iteración.

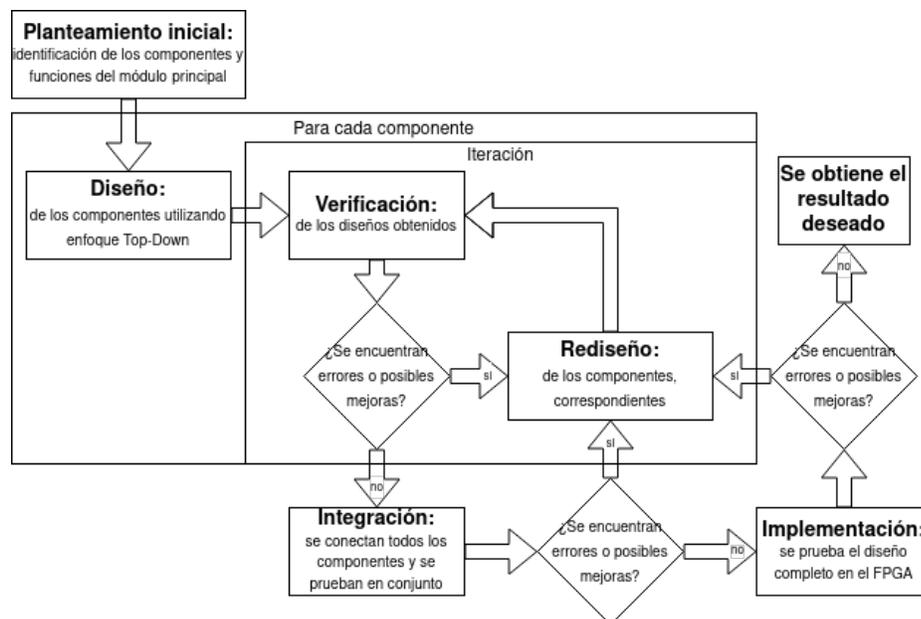


Figura 3.2: Diagrama del flujo de trabajo empleado.

En la figura 3.2 se observa el diagrama que representa/resume el flujo de trabajo empleado para el diseño del *SoC* y del procesador. Es importante destacar que para el diseño final del *SoC* se repite la etapa de planteamiento debido al tiempo consumido por el desarrollo del procesador.

Considerando estos puntos y la metodología de diseño presentada en la sección 3.1: primero se confecciona un diagrama de bloques general (de nivel superior), que contemple la *CPU* y el resto del *hardware* del *SoC*, por lo que hace falta definir las funciones preliminares del mismo. Luego, se comienza con el diseño de los elementos más complejos a los simples, comenzando con el *CPU* y cada uno de los submódulos del mismo, para terminar con el *hardware* del entorno de ejecución y del *SoC*. Todo esto siguiendo el flujo de trabajo antes descrito.

Los detalles del diseño final del *SoC* obtenido se presentan en la sección 4. En la sección 5 se presenta la implementación obtenida y las pruebas realizadas al *SoC* en forma de simulaciones (ya sea sobre submódulos o el *SoC* completo) y sobre *hardware*. Ahora se explica la metodología de verificación empleada para distintos casos de *UUT* (*unit under test*).

3.3 Metodología de verificación

Similar a lo expuesto en la sección 2.3.1, la verificación empleada se basa en *test benches* sobre la *UUT*. Cuando el módulo es muy simple no se realiza verificación. Si tiene un mayor grado de complejidad se escribe el *test bench* directamente en *System Verilog*. Para casos aun más complejos, como la *ALU* o *FPU*, se generan estos *test benches* en *System Verilog* mediante un código escrito en *C* que escribe pruebas con entradas generadas de forma pseudo-aleatoria. Los detalles de las pruebas y simulaciones realizadas a cada módulo relevante con esta metodología de verificación se presentan en la sección 5.1.

Lo anterior solo se refiere a la verificación de los módulos que conforman el *core* y el *SoC*. Sin embargo, para comprobar el funcionamiento de estos últimos se debe ejecutar algún programa de prueba en el *core* del *SoC*. Por ello, se escriben a mano códigos de prueba en lenguaje ensamblador, considerando las llamadas de sistema permitidas por el entorno de ejecución implementado y las instrucciones presentadas en las tablas 2.6, 2.7 y 2.9. Estos códigos de prueba se testean en *RARS* para asegurar su compilación y correcta ejecución. La misma herramienta *RARS* permite exportar el código compilado y el estado inicial (y final) de la memoria de datos en formato de texto plano con representación hexadecimal o binaria. Luego, con un pequeño *script* de *Python* se reordenan las palabras de 8 bytes según la configuración de la memoria implementada en el *SoC*. Con esto se puede precargar el estado inicial de la memoria de programa y datos del *SoC* al momento de encenderse. Los detalles de este procedimiento se presentan en la sección 5.1.

Una vez con el programa cargado en la memoria del procesador se realiza el *test bench* para el comportamiento esperado de dicho programa y comprobar su correcto funcionamiento. En este caso, es especialmente importante la visualización de las señales en la línea de tiempo, por sobre todo los cambios de los registros, los accesos a memoria, el avance del *PC* y de las instrucciones. Todos los datos mencionados y más se contrastan con la simulación del mismo código en *RARS* para la depuración de los diseños (nótese que las llamadas a sistema del entorno implementado y de *RARS* deben concordar para poder realizar esta tarea). Nuevamente, este proceso es similar al del trabajo presentado en la sección 2.3.1, pero aprovechando las bondades de *RARS*.

En cuanto al proceso de verificación sobre *hardware* del *SoC* completo, se limita a comprobar el correcto funcionamiento del programa cargado y la estabilidad del mismo. Los detalles de los resultados obtenidos de esta verificación se presentan en la sección 5.2.

Por otra parte, considerando la metodología de verificación presentada en la sección 2.3.4 se pueden apreciar similitudes con respecto a la utilizada, puesto que *RARS* hace de *disassembler* y *golden reference model* al mismo tiempo. En cuanto a la metodología expuesta en la sección 2.3.5, es desestimada al considerarse mayor la curva de aprendizaje de su implementación que sus beneficios potenciales y poco útil a estar enfocada a diseños *fixed-point*.

4 Diseño propuesto

En esta sección se presenta el proceso de diseño realizado y el resultado del mismo. En primera instancia, tal como se indica en la sección 3.2, se realiza un planteamiento inicial. En este planteamiento se estudian ejemplos de *SoCs*, como el de las figuras 2.13 y 2.18, para luego elaborar el diagrama de bloques de nivel superior propuesto del *SoC* a implementar, el cual se puede ver en la figura 4.1.

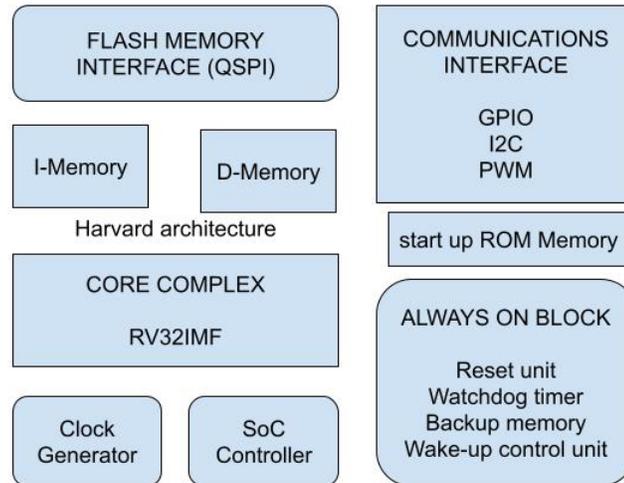


Figura 4.1: Diagrama de bloques de nivel superior propuesto en primera instancia para el *SoC*.

Observando la figura 4.1 se puede apreciar el planteamiento de un procesador con arquitectura de tipo *Harvard* (ver sección 2.1.2). Luego, se centra la atención en el *core complex*, el cual se basa en los juegos de instrucciones *RV32Iv2.1*, *RV32Mv2.0* y *RV32Fv2.2*, las últimas versiones disponibles según [8]. Se propone, en un principio, un diseño de 5 etapas *Pipelined* en base al trabajo de [15]. Sin embargo, se decide implementar el enfoque *Pipelined* de [12] al considerarse más simple e ilustrativo para una primera implementación de un procesador de este tipo. Por lo mismo y siguiendo la metodología *Top-Down*, se propone el diagrama de bloques simplificado de la figura 4.2, donde se identifican las etapas *pipeline* consideradas, estas son: *Instruction Fetch (IF)*, *Instruction Decode & Register Select (DR)*, *Execution or Address calculation (EX)*, *Data Memory Access (MEM)* y *Write Back (WB)*. Otra razón para cambiar al enfoque presentado en [12], es para poder empezar con el diseño de una versión *Single-Cycle* con el objetivo de realizar la conversión a *Pipelined* de forma más sencilla.

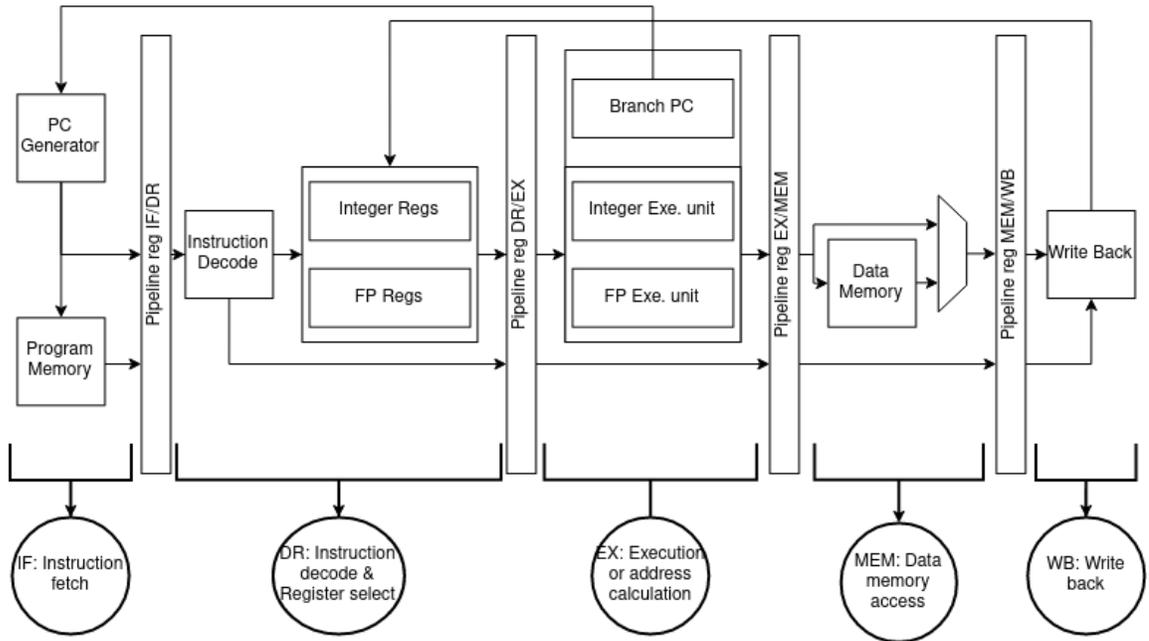


Figura 4.2: Diagrama de bloques simplificado *core complex pipelined*.

En la figura 4.2 ya se pueden apreciar los principales submódulos del *core complex*, como la *ALU*, *FPU*, *Register Files*, *PC generator*, *Instruction Decode*, *Write Back* y las memorias de datos y programa. De todos los módulos recién mencionados, el más complejo de diseñar e implementar es la *FPU*. Debido a estas dificultades en el proceso de diseño, verificación y depuración del mismo (rediseño) y el tiempo requerido en ello, se decide modificar la propuesta de *SoC* de la figura 4.1 y simplificar el mismo.

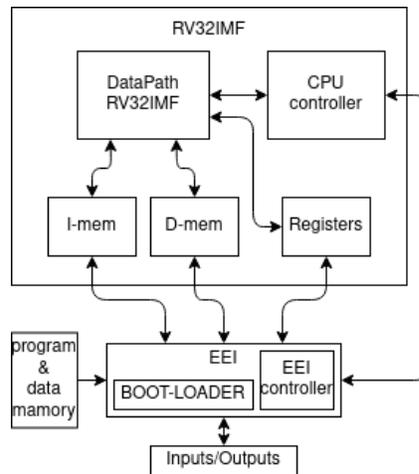


Figura 4.3: Diagrama de bloques de nivel superior (simplificado) propuesta final para el *SoC*.

El diagrama de bloques simplificado del nuevo *SoC* propuesto se puede ver en la figura 4.3. Las únicas diferencias de este diagrama con la implementación final, radica en los bloques “*program & data mamory*” y “*BOOT-LOADER*”. Esto se debe a que no se implementa el bloque *ROM* por temas de tiempo, por ello se carga directamente el código que ejecuta el *SoC* a las *BRAM* (*Block RAM*) del *core complex*. En la figura 4.3 se aprecia que el *EEI* (*Execution Environment Interface*) debe tener acceso a las memorias y al *Register Files*, esto con el fin de cargar programas, leer y

cargar datos en el *hart* e interpretar/recibir llamadas de sistemas. Además, el *EEI* le indica al *core complex* con que *PC* inicial la ejecución y cuando iniciarla, luego espera a que la ejecución termine con alguna llamada de sistema o algún error/excepción (trampas).

Como se indica en la sección 3, la etapa de diseño se basa en la metodología *Top-Down*. Sin embargo, no se aplica la metodología completa para cada submódulo. Se puede aplicar de forma completa, parcial o no aplicar directamente según la complejidad de cada módulo/submódulo.

A continuación, se presenta el diseño de cada módulo y submódulo del *SoC* obtenido a partir del planteamiento inicial recién presentado. Primero se explican las unidades básicas de *Instruction Decoder*, *Register Files* y la *ALU*, para continuar con las más complejas como las unidades de memoria (al implementar *Cache Controller*) y la *FPU*. Finalmente, se presentan los diseños obtenidos del *core complex* y del *EEI*, los cuales representan la implementación completa del *SoC*.

4.1 *Instruction Decoder*

El *Instruction Decoder* es completamente combinacional y tiene la función de recibir la instrucción ($in[31 : 0]$) a ejecutar y retornar la información contenida en cada uno de sus campos (ver tabla 2.5). Además, se encarga de extender a 32 bits el signo del campo *imm* según el tipo de instrucción y activar la señal *is_imm_valid* si dicho campo es válido. Los campos que se retornan directamente son: $funct3[2 : 0] = in[14 : 12]$, $funct2[1 : 0] = in[26 : 25]$ (este campo es ignorado en el resto del diseño porque para el juego de instrucciones implementado no tiene relevancia), $rd_add[4 : 0] = in[11 : 7]$, $rs1_add[4 : 0] = in[19 : 15]$, $rs2_add[4 : 0] = in[24 : 20]$ y $rs3_add[4 : 0] = in[31 : 27]$.

format_type[2:0]	
FMT	code
R	000
R4	001
S	011
B	100
U	101
J	110
I	010
not valid	111

Tabla 4.1: Codificación para *format_type* según el tipo de la instrucción.

Por otro lado, el campo *opcode* de la instrucción se codifica en dos señales: *format_type*[2 : 0] y *sub_format_type*[2 : 0]. La primera señal indica el tipo de instrucción (*R/R4/I/S/B/U/J*) y su codificación se puede apreciar en la tabla 4.1. Mientras que la segunda señal, *sub_format_type*, realiza una segunda clasificación según el subtipo de la instrucción. Esta clasificación se puede apreciar en la tabla 4.2.

sub_format_type[2:0]					
code	R	R4	S	U	I
000	RV32F	fmadd.s	fsw	lui	flw
001	RV32IM	fmsub.s	sb, sh, sw	auipc	addi, xori, ori, andi, slli, srli, srai, slti, sltiu
010	-	fnmsub.s	-	-	lb, lh, lw, lbu, lhu
011	-	fnmadd.s	-	-	jalu
100	-	-	-	-	ecall, ebreak
111	default				

Tabla 4.2: Codificación para *sub_format_type* según el subtipo de la instrucción.

Otro campo que debe analizar este módulo es $funct7[6 : 0] = in[31 : 25]$. La salida para este campo es $funct7_out[3 : 0]$ y está definida en la tabla 4.3.

funct7[6:0]	funct7_out[3:0]
0000000	0000
0000001	0001
0100000	0010
0000100	0011
0001000	0100
0001100	0101
0101100	0110
0010000	0111
0010100	1000
1101000	1001
1100000	1010
1110000	1011
1111000	1100
1010000	1101
not valid	1111

Tabla 4.3: Codificación para *funct7_out* según el *funct7* de la instrucción.

Finalmente, considerando que las instrucciones de *RV32F*, definidas en la tabla 2.9, pueden presentar accesos a registros flotantes y enteros, hace falta una última señal de control de acceso de registros. Esta señal es $reg_access_option[1 : 0]$ y se obtiene a partir del *opcode* (*format_type* y *sub_format_type*) y *funct7* (*funct7_out*); su codificación se presenta en la tabla 4.4.

reg_access_option[1:0]			
code	rd	rs1	rs2
00	I	I	I
01	FP	I	FP
10	I	FP	FP
11	FP	FP	FP

Tabla 4.4: Codificación para *reg_access_option* según el acceso a registros de la instrucción. *I* indica acceso a registros enteros y *FP* a registros flotantes. Nota: *rs3* siempre es flotante.

4.2 Register Files

Esta unidad tiene la función de almacenar, entregar y modificar los registros, 32 enteros y 32 flotantes, con los que trabaja el *core complex*. En la figura 4.4 se aprecia el diagrama de bloques de la implementación realizada junto a las entradas que recibe y sus salidas.

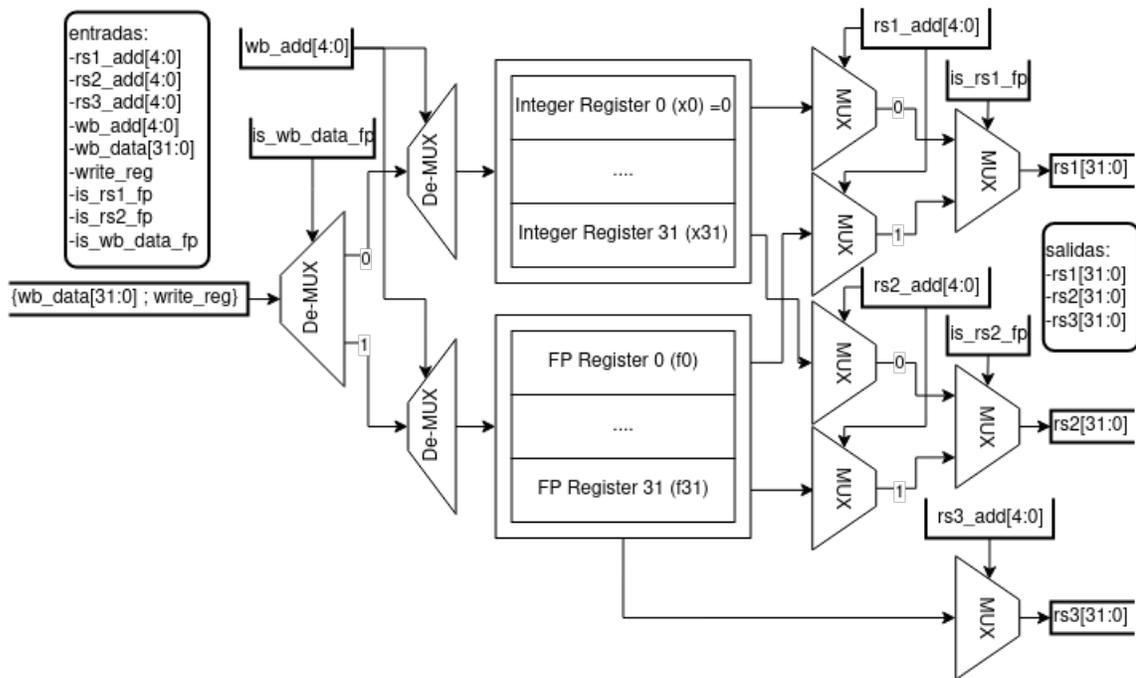


Figura 4.4: Diagrama de bloques detallados del *Register Files*.

Es importante indicar las siguientes consideraciones de este módulo:

- El primer registro entero, *x0*, siempre es cero (*hardwired to zero*).
- Inspirado en [15], posee 3 puertos de lectura combinatoriales, donde *rs1_add*, *rs2_add* y *rs3_add* indican qué registro se desea leer para cada puerto de salida, *rs1*, *rs2* y *rs3*, respectivamente. Para distinguir entre un acceso a los registros flotantes o enteros se utilizan las señales *is_rs1_fp* y *is_rs2_fp*: si son 1 indican que el acceso es a un registro flotante (de lo contrario sería entero) para las salidas *rs1* y *rs2*, respectivamente. El acceso de *rs3* siempre es un registro flotante.
- Posee 1 puerto de escritura síncrono que se activa en el flanco negativo del reloj. Donde *wb_add* direcciona al registro que se desea modificar por el valor ingresado en *wb_data* y

is_wb_data_fp indica si es flotante (= 1) o no (= 0). La señal *write_reg* indica si corresponde realizar la escritura.

- En [8] se definen los roles de los registros enteros y flotantes, entre los que se destacan dos registros enteros: *x2* como el *stack pointer* (*sp*), por lo que se inicializa con el valor *32'h2FFC*, y *x3* como *global pointer* (*gp*), que se inicializa igual a *32'h1800*. Se toman los valores que utiliza *RARS* como referencia en este caso.

4.3 Unidades de memoria

Como ya se ha mencionado, se opta por una arquitectura de tipo *Harvard*, esto quiere decir que el *core complex* accesa mediante buses de datos distintos a las memorias de programa (*I-mem*) y de datos (*D-mem*). Considerando lo anterior, hay distintas formas de implementar la memoria. Por ejemplo, puede ser mediante alguna *BRAM* con dos puertos de lectura/escritura o dos *BRAMs* separadas cada una con un solo puerto de lectura/escritura.

Para ambos casos hay ciertas consideraciones, cuando se implementa solo una *BRAM* al momento de direccionar se debe distinguir claramente qué sección de la memoria corresponde al programa y a los datos, es decir, que alguna de las dos secciones presenta un *offset* y no se direcciona desde cero. Si se utilizan dos *BRAMs*, ambas deben ser direccionadas desde cero (el *offset* antes mencionado entre *I-mem* y *D-mem* no existe), pero puede ser que el diseñador considere conveniente que no sea así, por ejemplo, para compilar programas bajo algún estándar que considere el *offset* entre las memorias. En tal caso, hay diversas opciones, una puede ser añadir un restador o sumador en el *core complex* que ajuste el direccionamiento de la *BRAM*, ya sea *I-mem* o *D-mem*, según el *offset* deseado.

En esta ocasión, por simplicidad, se decide por la implementación de dos *BRAMs* sin ajuste de *offset*. Es importante tener en cuenta esta decisión al momento de escribir y compilar programas en lenguaje ensamblador. En este caso, como se menciona en la sección 2.4.3, se utiliza *RARS* con la configuración de memoria (la cual también define *sp* y *gp*, ver sección 4.2), definiendo que el direccionamiento de datos comience en *0x00000000* y la de programa en *0x00003000*. De esta forma no hay que ajustar el *offset* en el acceso de *D-mem* y, para *I-mem*, no hace falta ajuste alguno puesto a que el *EEI* se encarga de inicializar la ejecución con *PC = 0* y que todos los saltos se calculan en base a dicho *PC* (excepto *jalr*, pero en general recibe algún *PC* anterior a través de *rs1*, ver tabla 2.6).

Debido a lo anterior, se diseña un sólo módulo de memoria que se utiliza para *D-mem* y para *I-mem*. El diseño contempla un direccionamiento *little endian*, una memoria cache con su controlador y la *BRAM*. Además, debe ser capaz de escribir *words*, *halfwords* y *bytes* en memoria, las cuales también se deben poder leer con y sin extensión de signo. Es importante destacar que el direccionamiento es a nivel de *byte*. Por otro lado, se implementa una *BRAM* de 62.496KB, dando un total de 124.992KB al considerar ambas memorias.

El módulo de memoria implementado se llama *memory* y está compuesto por un bloque *always* combinacional, la *BRAM* (llamada *block_ram*) y el submódulo *cache_controller*. Las entradas de *memory* son:

- **clk** y **rst**: reloj y *reset* del sistema, respectivamente.
- **rw**: si es 1 se escribe (*write/store*) en memoria y si es 0 se lee (*read/load*).
- **valid**: si se activa se realiza la operación, ya sea leer o escribir.
- **addr[31:0]**: indica el direccionamiento de memoria a nivel de *byte*.
- **data_in[31:0]**: información por escribir cuando *rw=1*.

- **byte_half_word[1:0]:**
 - 00 o 11: indican que se desea leer/escribir un *word*.
 - 01: indica que se desea leer/escribir un *halfword*.
 - 10: indica que se desea leer/escribir un *byte*.
- **is_load_unsigned:** si es 1 cuando $rw=0$ y *byte_half_word[1:0]* indica *halfword* o *byte*, no se extiende el signo de la respuesta. Si bajo las mismas condiciones es 0, se extiende el signo.

En cuanto a las salidas, estas son: **ready** para indicar que la operación de lectura/escritura se completó, **out_of_range** que indica si la dirección ingresada esta fuera del rango del espacio de direcciones de la memoria física (la *BRAM*) y **data_out[31:0]** con la respuesta para las operaciones de lectura. También, se introduce un parámetro en el módulo, **initial_option**, que se conecta directamente con el parámetro del mismo nombre que recibe el módulo *block_ram* declarado en *memory*.

La función del bloque *always* en *memory* es la de manejar las peticiones provenientes de la *CPU* y dirigir las al *cache_controller* (el cual contiene la memoria cache); este módulo se explica en la sección 4.3.1. Luego el *cache_controller* se comunica con *block_ram* (la *BRAM*).

En concreto, el bloque *always* recibe las entradas de *memory* antes mencionadas y se encarga de verificar si se desea realizar una lectura o escritura. Según sea el caso y la información entregada por *byte_half_word*, se determinan las entradas (de solicitud proveniente de la *CPU*) pertinentes para el *cache_controller*. Luego, espera a que el *cache_controller* entregue el resultado de su ejecución. Al finalizarse, el bloque *always* retorna la información obtenida si era una lectura (considerando la información de *byte_half_word* y realizando la extensión de signo si corresponde según lo indicado en *is_load_unsigned*) e indica que se finaliza la ejecución ($ready=1$). Este bloque, además, se encarga de manejar los accesos a memoria desalineados. Para ello, simplemente ignoran los bits menos significativos de la dirección de acceso solicitada según corresponda: si se accesa un *word*, se ignoran los últimos 2 bits menos significativo, si es un *halfword*, se ignora el último menos significativo y si se accesa un *byte* no se ignora ningún bit.

Como ya se mencionó, la *BRAM* está descrita por el módulo *block_ram* y cuenta con una capacidad de 62.496KB, con 3906 entradas de un ancho de 16 bytes. Las entradas del módulo son:

- **clk:** el reloj del sistema.
- **addr[11:0]:** con 12 bits basta para direccionar las 3906 entradas. Nótese que $addr[11 : 0] = addr2byte[15 : 4]$, donde $addr2byte[31:0]$ sería el direccionamiento a nivel de *byte* del acceso deseado.
- **data_in[127:0]:** es la información a escribir si es el caso.
- **rw:** si es 1 se escribe, en caso de ser 0 se lee.
- **valid:** indica si realizar la operación, ya sea leer o escribir.

Por otra parte, solo hay dos salidas: **ready** para indicar que se realizó la operación y **data_out[127:0]** que retorna el valor de la lectura. Es importante señalar que este módulo recibe el parámetro **initial_option**: si es 1 se precarga la *BRAM* con la memoria de datos, si es 2 con la de programa y en cualquier otro caso se inicializa en blanco (más detalles sobre la implementación en la sección 5).

A continuación, se presenta el diseño del módulo *cache_controller*, el cual se encarga de la comunicación con la *BRAM* y de la gestión de la memoria cache.

4.3.1 Cache Controller

Este módulo contiene la memoria cache y se encarga de manejar la información solicitada por la *CPU* y el acceso a la *BRAM*. La implementación se basa en la presentada en [12] (capítulos 5.9 y 5.12). La memoria cache cuenta con las siguientes características:

- Es directamente mapeada, es decir que, cada locación en memoria principal (la *BRAM*) es mapeada a, exactamente, una locación en la cache.
- Cada bloque tiene un *dirty bit* y un *valid bit*, el primero se activa si se escribe en el bloque, solo se desactiva al cargar un nuevo bloque para lectura. Por otro lado, el *valid bit* indica si el bloque alojado en el cache es válido, o sea, que corresponde a un bloque de la memoria principal.
- El *Write-Back* (actualización de la memoria principal) usa *write allocate*, es decir, si ocurre un *miss* al querer escribir una área de memoria que no está alojada en la cache se verifica el *dirty bit*. Este indica si el bloque alojado actualmente está sucio, si no lo está, se escribe en el cache y se activa el *dirty bit* del bloque, si no, se escribe dicho bloque en la memoria principal y luego se carga el nuevo bloque a escribir en la cache para su escritura.
- Presenta un *block size* (ancho de las entradas de memoria) de 4 palabras (*words*), es decir, 16 *bytes*.
- Un tamaño de 16KiB, es decir que tiene 1024 entradas (*blocks*).
- Direcccionamiento de 32 bits.
- El cache tiene un índice de 10 bits: $2^{10} = 1024$ bloques/entradas de memoria.
- El *block offset* es de 4 bits: $2^4 = 16$ *bytes* por bloque.
- Presenta un *tag size* de $32 - (10 + 4) = 18$ bits: $tag[17 : 0] = addr[31 : 14]$. El *tag* indica que bloque de la memoria principal está alojado en el bloque del cache.

En la figura 4.5 se aprecia el diagrama de bloques del cache implementado.

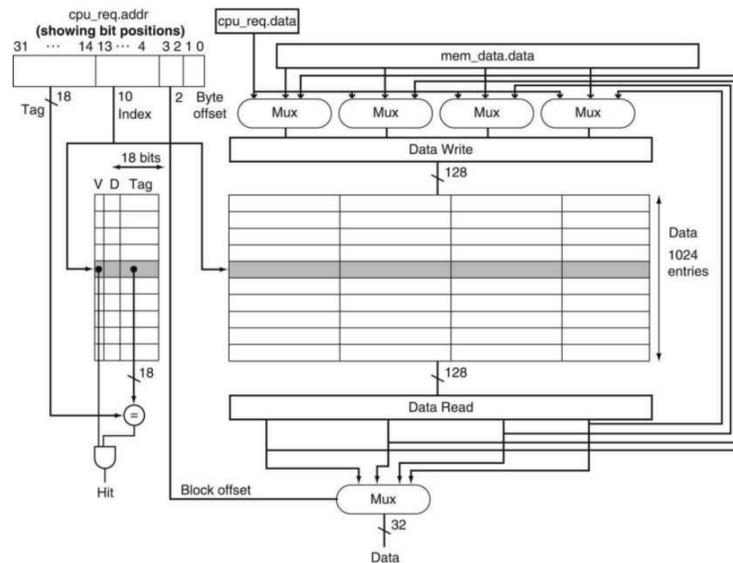


Figura 4.5: Diagrama de bloques de la memoria cache implementada. Fuente: [12].

Para explicar el funcionamiento de este módulo se comienza con las interfaces con la *CPU* y con la memoria principal. Las señales entre el *cache_controller* y la *CPU* son:

- Desde la *CPU* al *cache_controller*:
 - cpu_rw**: si es 1 indica escritura, si es 0 lectura.
 - cpu_valid**: indica que se inicie la operación.
 - cpu_addr[31:0]**: la dirección del acceso deseado.
 - cpu_data_in[31:0]**: datos provenientes de la *CPU*.

- Desde el *cache_controller* a la *CPU*:
 - cpu_ready**: indica a la *CPU* que la operación fue finalizada.
 - cpu_data_out[31:0]**: datos provenientes del *cache_controller*.

Luego las señales entre el *cache_controller* y *block_ram* son:

- Desde el *cache_controller* a la *block_ram*:
 - mem_rw**: si es 1 indica escritura, si es 0 lectura.
 - mem_valid**: indica que se inicie la operación.
 - mem_addr[31:0]**: la dirección del acceso deseado.
 - mem_data_in[127:0]**: datos provenientes del *cache_controller*.
- Desde la *block_ram* al *cache_controller*:
 - mem_ready**: indica al *cache_controller* que la operación fue finalizada.
 - mem_data_out[127:0]**: datos provenientes de la *block_ram*.

Finalmente, se implementa el controlador propuesto en [12], la máquina de estados de dicho controlador se puede ver en la figura 4.6. Solo se realiza una modificación del código utilizado para este controlador, en el estado *compare_tag*, originalmente realiza el *hit* y en el siguiente ciclo vuelve al estado *idle*. Esto se modifica para que se mantenga en *compare_tag* hasta que *cpu_valid* sea 0 de nuevo.

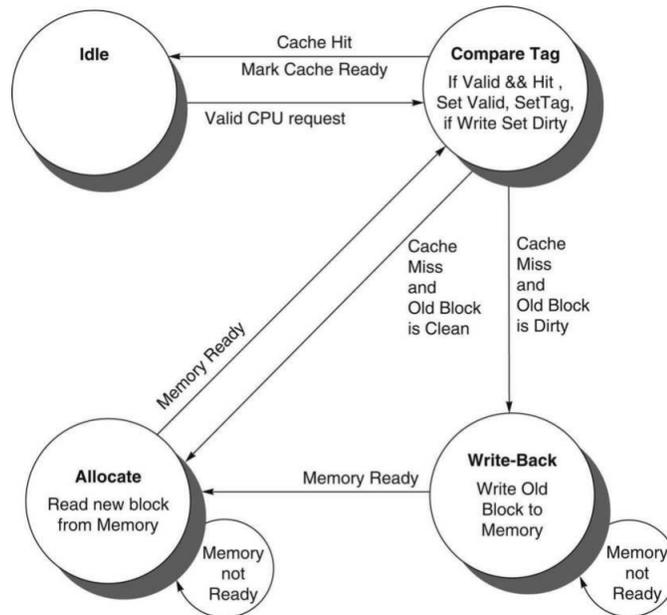


Figura 4.6: *FSM* del *Cache Controller* implementado. Fuente: [12].

A continuación, se explica cada estado del *FSM* de la figura 4.6:

- **Idle**: estado de espera por alguna solicitud de escritura o lectura válida, en tal caso el siguiente estado es **Compare Tag**.
- **Compare Tag**: este estado comprueba si ocurre un *hit*, esto sucede si para el *cache index* igual a $cpu_addr[13 : 4]$, se cumple que $tag[17 : 0] = cpu_addr[31 : 14]$ y que el *valid bit* es 1, en caso contrario es un *miss*. Si ocurre un *hit* se realiza la operación de escritura/lectura en la cache (si es escritura se enciente en *dirty bit* del bloque), indica que la operación se finalizó y, en la versión original, se vuelve al estado *Idle*, pero en este caso, se mantiene en **Compare Tag** hasta que **cpu_valid** sea 0, cuando esto sucede se vuelve a *Idle*. Si por otro lado ocurre un *miss*, se actualiza el *tag* y el siguiente estado es **Write-Back** si el *dirty bit* está activado o **Allocate** si está desactivado.

- **Write-Back:** este estado escribe el bloque de 128 bits de la cache en la memoria principal usando la dirección compuesta por el *tag* y el *cache index* ($cpu_addr[31:4]$). Se mantiene en este estado hasta que la memoria indica que se finaliza la operación de escritura, luego se pasa a **Allocate**.
- **Allocate:** se busca el nuevo bloque en la memoria principal y se actualiza el bloque correspondiente en el cache. Se mantiene en este estado hasta que la memoria indica que se finaliza la operación de lectura, luego se pasa a **Compare Tag**.

4.4 *Arithmetic Logic Unit (ALU)*

Este módulo se encarga de ejecutar las operaciones aritméticas o lógicas de números enteros de la instrucción en ejecución y su implementación es puramente combinacional. Recibe dos operandos, $in1[31:0]$ e $in2[31:0]$, y la operación a realizar está definida por la entrada $operation[4:0]$. Las salidas son dos, $res[31:0]$ y $boolean_res$, la primera es el resultado de la operación y la última salida indica, en las operaciones de comparación ($beq/bne/blt/bltu/slt/sltu/bge/bgeu$), si se cumple la condición ($= 1$) o no ($= 0$).

La misma *ALU* se compone de 3 *sub-ALUs*, una para las operaciones lógicas y aritméticas de *RV32I* (*integer_sub_alu_simple*), otra para las de comparación de *RV32I* (*integer_sub_alu_comparison*) y la última dedicada a las de multiplicación y división de *RV32M* (*integer_sub_alu_mul*). En el caso de *integer_sub_alu_simple* se retorna el resultado directamente, *integer_sub_alu_comparison* retorna el booleano resultante de la comparación, el cual sería el valor de *boolean_res*, y la *ALU* se encarga de retorna $32'b1$ o $32'b0$ según corresponda. En cuanto al submódulo *integer_sub_alu_mul* es importante señalar que si $in2=0$ se retorna $32'hFFFFFFFF$ para las operaciones de división e *in1* en las de módulo.

Para determinar la operación que debe realizar la *ALU* se implementa el módulo combinacional *alu_op_selection*. Las entradas que recibe este módulo son: $imm_11_5[6:0]=imm[11:5]$, $funct7_out[3:0]$, $funct3[2:0]$, $format_type[2:0]$ y $sub_format_type[2:0]$, todas provenientes del *Instruction Decoder*. A partir de dichas entradas y de la información contenida en las tablas 2.6 y 2.7, se determina que operación debe ejecutar la *ALU* para la instrucción en curso, generando la salida $alu_option[4:0]$ que se conecta a la entrada $operation[4:0]$ de la *ALU*. En la tabla 4.5 se aprecia la codificación para todas las operaciones de la *ALU*.

alu_op_selection			
Operation	alu_option[4:0]	Inst.	Note
in1 + in2	00_000	add, addi	default operation
in1 - in2	00_001	sub	-
in1 ^ in2	00_010	xor, xori	-
in1 in2	00_011	or, ori	-
in1 & in2	00_100	and, andi	-
in1 << in2	00_101	sll, slli	-
in1 >> in2	00_110	srl, srli	in1 unsigned
in1 >> in2	00_111	sra, srai	in1 signed
in1 == in2 ?	01_000	beq	-
in1 != in2 ?	01_001	bne	-
in1 < in2 ?	01_010	blt, slt	signed operation
in1 < in2 ?	01_011	bltu, sltu	unsigned operation
in1 >= in2 ?	01_100	bge	signed operation
in1 >= in2 ?	01_101	bgeu	unsigned operation
(in1 * in2)[31:0]	10_000	mul	signed operation
in1 / in2	10_011	div	signed operation
in1 / in2	10_100	divu	unsigned operation
in1 % in2	10_101	rem	signed operation
in1 % in2	10_111	remu	unsigned operation
(in1 * in2)[63:32]	11_000	mulh	signed operation
(in1 * in2)[63:32]	11_001	mulu	unsigned operation
(in1 * in2)[63:32]	11_010	mulsu	in1 signed / in2 unsigned

Tabla 4.5: Codificación de *alu_option* para cada tipo de operación de la *ALU*.

4.5 Floating Point Unit (FPU)

La *FPU* se encarga de ejecutar todas las operaciones relacionadas con el juego de instrucciones definido por la extensión *RV32F*, estas se resumen en la tabla 2.9. Las entradas del módulo son:

- **clk** y **rst**: señales de reloj y *reset*, respectivamente.
- **start**: indica que se inicie el cálculo de la operación solicitada.
- **rm[2:0]**: especifica el modo de redondeo.
- **option[4:0]**: indica la operación que debe realizar la *FPU*.
- **in1[31:0]/in2[31:0]/in3[31:0]**: son los operandos.

Y las salidas son:

- **NV**, **NX**, **UF**, **OF** y **DZ**: indican si ocurre una operación no válida, inexacta, *underflow*, *overflow* y/o división por cero, respectivamente.
- **ready**: indica la finalización de la operación.
- **out[31:0]**: el resultado de la operación.

fpu_op_selection	
Inst.	fpu_option[4:0]
fmadd.s	0_00_00
fmsub.s	0_00_01
fnmsub.s	0_00_10
fnmadd.s	0_00_11
fadd.s	1_00_00
fsub.s	1_00_01
fmul.s	1_00_10
fdiv.s	1_00_11
fsqrt.s	1_01_00
fsgnj.s	0_11_00
fsgnjn.s	0_11_01
fsgnjx.s	0_11_10
fmin.s	0_10_00
fmax.s	0_10_01
fcvt.s.w	0_01_00
fcvt.s.wu	0_01_01
fcvt.w.s	0_01_10
fcvt.wu.s	0_01_11
fle.s	1_10_00
flt.s	1_10_01
feq.s	1_10_10
fclass.s	0_11_11
not valid	1_11_11

Tabla 4.6: Codificación de *fpu_option* para cada operación de la *FPU*.

Para determinar la entrada *option* se añade el módulo *fpu_op_selection*, el cual es combinatorial y se encarga de recibir las señales: **rs2_add[4:0]**, **funct7_out[3:0]**, **funct3[2:0]**, **format_type[2:0]**, **sub_format_type[2:0]** y **rm_from_fcsr[2:0]** (esta última entrada recibe el *rounding mode* proveniente del registro *FCSR*. Sin embargo no fue implementado, por ello siempre es $RNE=000$). Con la información recibida se retorna: **rm2fpu[2:0]**, el modo de redondeo seleccionado para la *FPU* (el cual puede estar definido por *funct3* o por *rm_from_fcsr* si *funct3=111*), y **fpu_option[4:0]**; esta última señal se conecta con la entrada *option* de la *FPU* para indicar la operación deseada. La codificación para cada instrucción se detalla en la tabla 4.6.

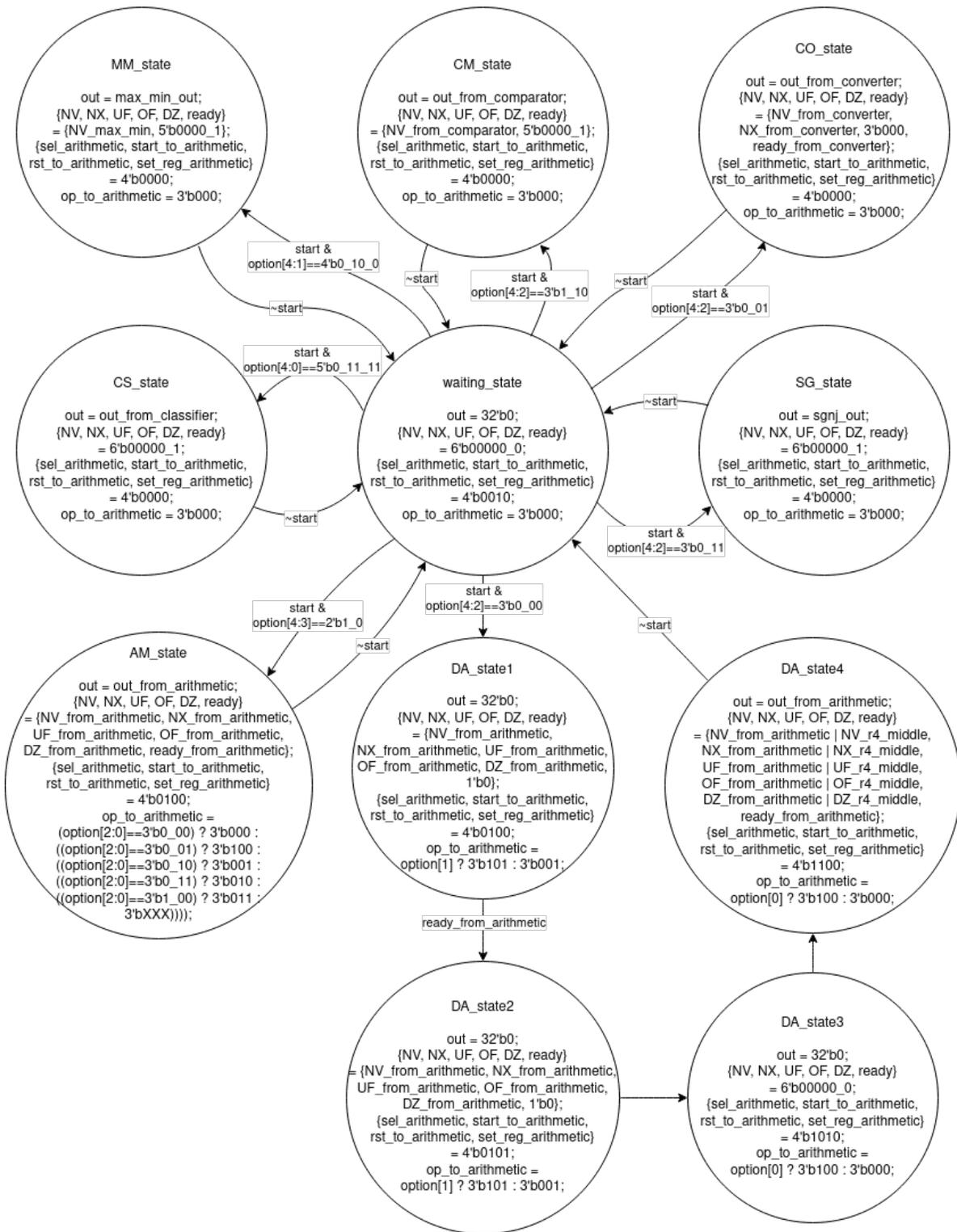


Figura 4.7: Diagrama MDS del FSM de la FPU.

La *FPU* se compone de un *FSM* el cual se encarga de manejar distintas unidades de ejecución según la entrada *option*. En la figura 4.7 se aprecia el diagrama *MDS* del *FSM* con el detalle de las señales de E/S y transición de los estados. Ahora se explica el rol de cada estado y como maneja cada unidad de ejecución:

- **waiting_state**: este estado se encarga de esperar la señal *start*, luego según la opción conte-

nida en *option* (como se aprecia en la figura 4.7) decide cual será el siguiente estado.

- **CS_state:** este estado se encarga únicamente de ejecutar la instrucción *fclass.s* y para ello dispone de un simple submódulo combinacional, *fp_classifier*, encargado de recibir *in1* y retornar la clasificación correspondiente según la tabla 2.10. Activa automáticamente la señal *ready* y vuelve a **waiting_state** cuando *start* se desactiva.
- **MM_state:** aquí se ejecutan las instrucciones *fmin.s* y *fmax.s* considerando los operandos *in1* e *in2*, esto se realiza mediante un *assign* y la ayuda del submódulo *fp_comparator* (el cual a su vez se encarga de ejecutar las instrucciones manejadas por el estado *CM_state*). Activa automáticamente la señal *ready* y vuelve a **waiting_state** cuando *start* se desactiva.
- **CM_state:** este estado realiza las comparaciones de las instrucciones *fle.s*, *flt.s* y *feq.s* para los operandos *in1* e *in2*, para ello se dispone del submódulo combinacional *fp_comparator*, el cual realiza las comparaciones según lo especificado en la sección 2.2.5.2. Activa automáticamente la señal *ready* y vuelve a **waiting_state** cuando *start* se desactiva.
- **CO_state:** aquí se ejecutan las instrucciones *fcvt.s.w*, *fcvt.s.wu*, *fcvt.w.s* y *fcvt.wu.s* considerando *in1*, para esta tarea se implementa el módulo secuencial *fp_converter* (detallado en la sección 4.5.2). Activa *ready* cuando *fp_converter* indica que finalizó la ejecución y vuelve a **waiting_state** cuando *start* se desactiva.
- **SG_state:** realiza las operaciones *fsgnj.s*, *fsgnjn.s* y *fsgnjx.s* de forma combinacional mediante un *assign*, se consideran las entradas *in1* e *in2*. Activa automáticamente la señal *ready* y vuelve a **waiting_state** cuando *start* se desactiva.
- **AM_state:** este estado ejecuta las instrucciones *fadd.s*, *fsub.s*, *fmul.s*, *fdiv.s* y *fsqrt.s* con *in1* e *in2* como operandos. Para esta tarea se implementa el módulo secuencial *fp_arithmetic_unit* (detallado en la sección 4.5.1). Activa *ready* cuando *fp_arithmetic_unit* indica que finalizó la ejecución y vuelve a **waiting_state** cuando *start* se desactiva.
- **DA_state1**, **DA_state2**, **DA_state3** y **DA_state4:** estos estados se encargan de ejecutar las instrucciones de tipo *R4*. Por simplicidad y tiempo, no se implementa una unidad específica para la ejecución de este tipo de instrucciones con tres operandos (*in1*, *in2* e *in3*), por ello se reutiliza el módulo *fp_arithmetic_unit*. En *DA_state1* se realiza la multiplicación considerando *in1* e *in2*, cuando está lista la operación se pasa al estado *DA_state2*, este almacena el resultado en un registro. Luego, en *DA_state3* se modifican las entradas de *fp_arithmetic_unit* por el registro temporal con el resultado anterior e *in3* y se cambia la operación por la suma o resta. Finalmente, en *DA_state4* se espera el fin de la ejecución de *fp_arithmetic_unit* para encender *ready* y volver a **waiting_state** cuando *start* se desactive.

Nótese que no se siguen todos los pasos de la metodología *Top-Down* para la confección del diagrama *MDS*. Esto se debe a que este módulo se desarrolla después que sus principales submódulos, *fp_arithmetic_unit* y *fp_converter*, por ello, la experiencia adquirida ayuda a ahorrar tiempo en los nuevos diseños realizados.

También es importante tener en cuenta el modo de redondeo seleccionado por *rm*, el cual puede ser *RNE*, *RTZ*, *RDN*, *RUP* o *RMM*. En la tabla 4.7 se aprecia la codificación para cada uno de los modos y el criterio de redondeo o truncamiento utilizado según los bits *guard*, *round* y *sticky*.

Los modos de redondeo se consideran en los módulos *fp_arithmetic_unit* y *fp_converter*. A continuación, en las secciones 4.5.1 y 4.5.2, respectivamente, se presenta el diseño desarrollado para estos módulos, los cuales fueron en gran medida el mayor reto de todo el trabajo realizado después de la implementación del *core complex* y del *SoC* como tal. Es importante tener en cuenta para las máquinas de estados que se presentan en las subsecciones venideras, que las mismas se activan en el flanco positivo del reloj y que cuando se habla de algún registro, este almacena su nuevo valor en el flanco negativo.

grs[2:0] 0.grs ₁₀	000 0.000	001 0.125	010 0.250	011 0.375	100 0.500	101 0.625	110 0.750	111 0.875	sign bit	RM Description
RNE (000)	0 0	0 0	0 0	0 0	lsb lsb	1 1	1 1	1 1	0 1	To nearest, ties to even
RTZ (001)	0 0	0 1	Toward zero							
RDN (010)	0 0	0 1	0 1	Toward -inf						
RUP (011)	0 0	1 0	0 1	Toward +inf						
RMM (100)	0 0	0 0	0 0	0 0	1 1	1 1	1 1	1 1	0 1	To nearest, ties away from zero

Tabla 4.7: Resumen del criterio de redondeo utilizado según el *rounding mode* (RM), el bit de signo ($sign$) y los bits *guard* (g), *round* (r) y *sticky* (s). 1 indica que se redondea el resultado (sumar uno a $fraction$) y 0 que se trunca (no se modifica $fraction$). Nota: $lsb=fraction[0]$.

4.5.1 *FP Arithmetic Unit*

Este módulo, *fp_arithmetic_unit* el más complejo de todos los diseñados en este trabajo. Se encarga de las operaciones aritméticas de punto flotante: suma, resta, multiplicación, división y raíz cuadrada. Para ello recibe las siguientes entradas:

- **start**, **rst**, **clk**: señal de inicio de ejecución, de *reset* y reloj, respectivamente.
- **rm[2:0]**: el *rounding mode* seleccionado. Ver tabla 4.7.
- **op[2:0]**: señal que codifica la operación a realizar. Ver tabla 4.8.
- **in1[31:0]/in2[31:0]**: operandos en punto flotante.

op[2:0]		
Code	Pseudocode	Operation
000	out=in1+in2	ADD
001	out=in1*in2	MUL
010	out=in1/in2	DIV
011	out=sqrt(in1)	SQRT
100	out=in1-in2	SUB
101	out=-in1*in2	-MUL
110	out=-in1/in2	-DIV

Tabla 4.8: Codificación de la entrada *op* del módulo *fp_arithmetic_unit*.

Luego, el módulo retorna:

- **ready**: señal que indica que la operación se ha completado.
- **status[2:0]**: señal que indica la bandera que se pudo generar en la ejecución. Ver tabla 4.9.
- **out[31:0]**: el resultado de la operación.

status[2:0]	
Code	Flag
000	none
001	UF
010	OF
011	NV
100	DZ
111	NX

Tabla 4.9: Codificación de la salida *status* del módulo *fp_arithmetic_unit*.

Debido a que este módulo es complejo de implementar, se diseña a partir de la metodología *Top-Down* desde un principio. Sin embargo, no se realiza el planteamiento del diagrama de bloques simplificado debido a que se empieza el diseño desde los algoritmos para el cálculo de cada operación.

Los algoritmos de suma y multiplicación de punto flotante son extraídos del capítulo 3 de [12], el algoritmo de la división es similar al de multiplicación por lo que se infiere a partir de este último. Finalmente, el algoritmo utilizado para el cálculo de la raíz cuadrada fue extraído del trabajo realizado en [27].

Todos los algoritmos tienen en común la fase de redondeo y normalización. Teniendo esto en cuenta, se propone el diagrama de flujo simplificado de la figura 4.8. De esta forma no se duplica el hardware necesario para redondeo y normalización. Además, se determina la necesidad de un módulo encargado de identificar los casos triviales, especiales o excepciones.

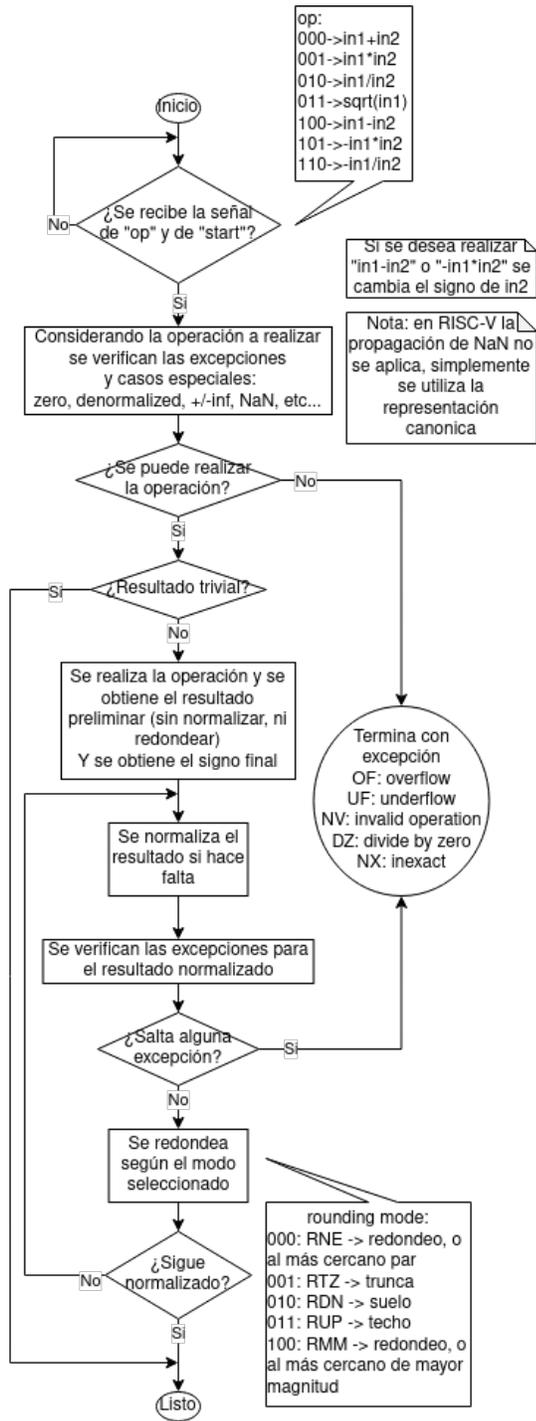


Figura 4.8: Diagrama de flujo simplificado del módulo *fp_arithmetic_unit*.

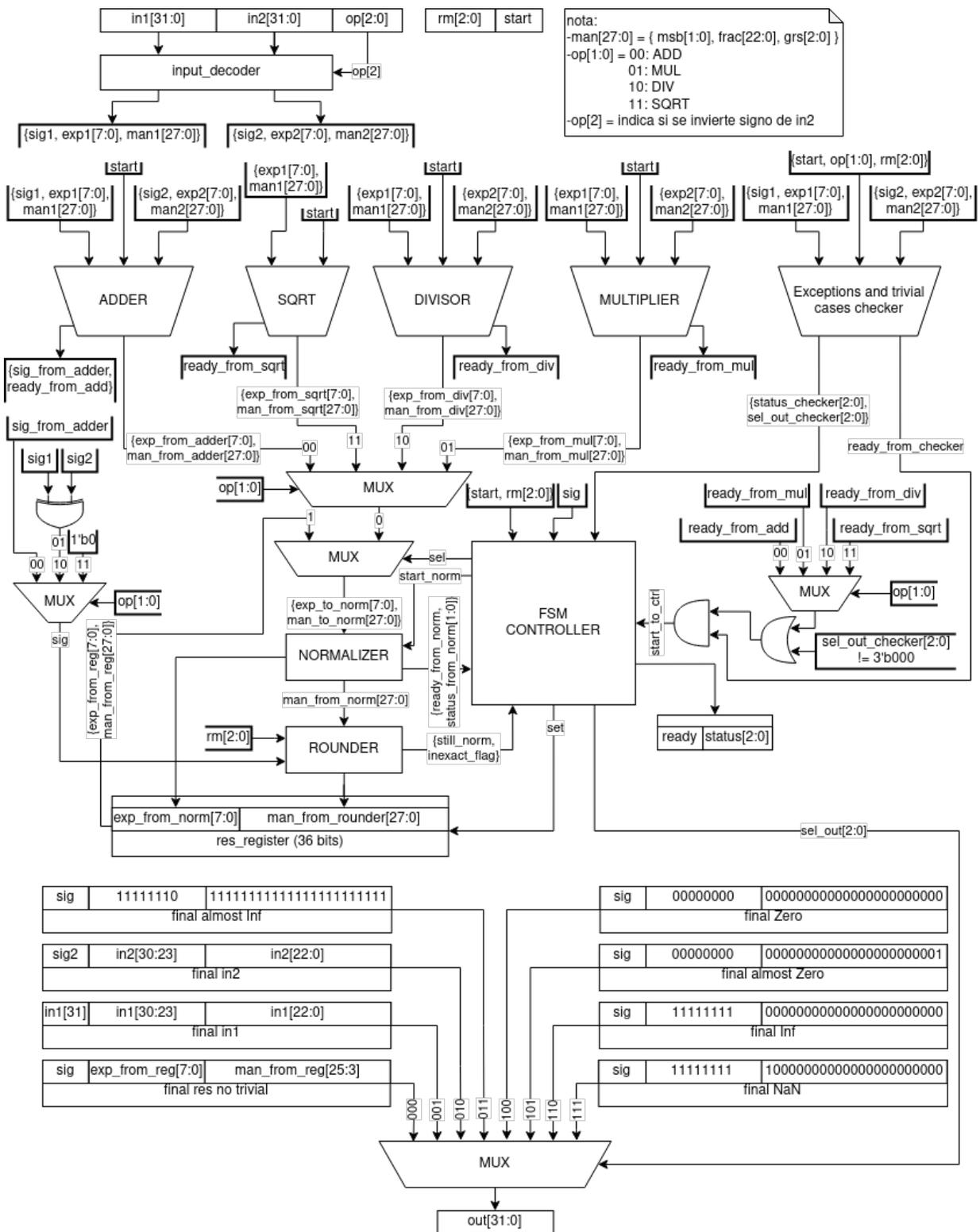


Figura 4.9: Diagrama de bloques detallado del módulo *fp_arithmetic_unit*.

Considerando el diagrama de flujo de la figura 4.8 se elabora el diagrama de bloques detallado (ver figura 4.9) del módulo *fp_arithmetic_unit*, donde se identifican 7 submódulos principales con sus entradas y salidas. A continuación, se detalla el rol de cada uno de estas unidades:

- **ADDER**: encargado de las operaciones de suma y resta. Detallado en la sección 4.5.1.1.
- **MULTIPLIER**: realiza la multiplicación. Detallado en la sección 4.5.1.2.
- **DIVISOR**: realiza la división. Detallado en la sección 4.5.1.3.
 - **significands_divisor**: parte del módulo *DIVISOR*, ejecuta la división de la mantisa. Detallado en la sección 4.5.1.4.
- **SQRT**: calcula la raíz cuadrada. Detallado en la sección 4.5.1.5.
 - **significand_sqrt**: parte del módulo *SQRT*, calcula la raíz cuadrada de la mantisa. Detallado en la sección 4.5.1.6.
- **Exceptions and trivial cases checker**: se encarga de verificar los casos triviales y excepciones. Detallado en la sección 4.5.1.7.
- **NORMALIZER**: normaliza el resultado. Detallado en la sección 4.5.1.8.
- **ROUNDER**: realiza el redondeo del resultado siguiendo la tabla 4.7. Detallado en la sección 4.5.1.9.

Es importante señalar el rol del bloque identificado como “*input_decoder*” en la figura 4.9. Su tarea es recibir las entradas $in1[31:0]$ e $in2[31:0]$ para separar los campos de signo, exponente y mantisa. Si $op[2]=1$, se invierte el signo de $in2$, esto permite realizar la resta o cambiar el signo del resultado de la multiplicación o división (útil en operaciones de tipo $R4$). Con respecto a la mantisa, esta se obtiene agregando 5 bits al campo $fraction[22:0]$, 3 a la derecha inicializados en cero para considerar los bits *guard*, *round* y *sticky*. Por otro lado, el primer bit a la izquierda de $fraction[22:0]$ es el bit silencioso, se inicia 1 si el exponente es distinto de cero, en caso contrario se inicia en cero y el segundo a la izquierda siempre se inicia en cero.

Teniendo en consideración el diagrama de la figura 4.9 se confecciona el diagrama de flujo detallado del controlador (ver figura 4.10), este diagrama muestra las señales de salida para cada estado y las preguntas o señales de decisión consideradas para las transiciones de estado. A partir del diagrama de flujo detallado se elabora el diagrama *MDS* de la figura 4.11. A continuación, se detalla el rol de cada estado:

- **waiting_state**: espera la señal *start_to_ctrl* para iniciar la ejecución, cuando esto sucede el siguiente estado depende del valor de *check_flag*, si es 1 significa que la unidad *excep_triv_checker* determina que la operación se puede realizar y el siguiente estado es **initial_state**, de ser 0 se pasa a **exception_state_1**.
- **initial_state**: inicia la ejecución del *NORMALIZER* y espera a que finalice su ejecución (*ready_from_norm=1*).
 - Si *NORMALIZER* indica ($is_normal = 0 \iff status_from_norm \neq 00$) que ocurre *overflow*, *underflow* o que el exponente del resultado llega a cero (es decir que el resultado final es desnormalizado) el siguiente estado es **exception_state_2**.
 - Si por otro lado, *NORMALIZER* determina ($is_normal = 1 \iff status_from_norm = 00$) que el resultado es válido y *ROUNDER* que está normalizado ($still_norm = 1$) aún después del redondeo se pasa a **pre_final_state_1**, si no es así ($still_norm = 0$) el siguiente estado es **pre_iteration_state**.
- **pre_iteration_state**: este estado guarda el resultado inicial (con $sel = 0$) y pasa a **iteration_state_1**.
- **iteration_state_1**: desactiva el *NORMALIZER* y cambia la entrada del mismo por el resultado actual ($sel = 1$), el siguiente estado es **iteration_state_2**
- **iteration_state_2**: inicia la ejecución del *NORMALIZER* y espera a que finalice su ejecución (*ready_from_norm=1*).

- Si *NORMALIZER* indica ($is_normal = 0 \Leftrightarrow status_from_norm \neq 00$) que ocurre *overflow*, *underflow* o que el exponente del resultado llega a cero (es decir que el resultado final es desnormalizado) el siguiente estado es **exception_state_3**.

- Si por otro lado, *NORMALIZER* determina ($is_normal = 1 \Leftrightarrow status_from_norm = 00$) que el resultado es válido y *ROUNDER* que está normalizado ($still_norm = 1$) aún después del redondeo se pasa a **pre_final_state_2**, si no es así ($still_norm = 0$) el siguiente estado es **iteration_state_3**.

- **iteration_state_3**: guarda el resultado actual (con $sel = 1$) y pasa a **iteration_state_1**.
- **exception_state_1**: activa la señal *ready* y selecciona como resultado la respuesta trivial y la bandera definida por *excep_triv_checker*. Si se desactiva *start* pasa a **waiting_state**.
- **exception_state_2**: guarda el resultado inicial (con $sel = 0$) y pasa a **final_exception_state**.
- **exception_state_3**: guarda el resultado actual (con $sel = 1$) y pasa a **final_exception_state**.
- **final_exception_state**: activa la señal *ready* y desactiva el *NORMALIZER*. Este estado espera a que se desactive *start* para volver a **waiting_state**. Luego hay 3 escenarios posibles, según el resultado del *NORMALIZER* para la selección de la bandera por levantar y el resultado final:
 - Si $is_desnorm = 1 \Leftrightarrow status_from_norm = 11$ el resultado final es válido y desnormalizado, por ello se levanta la bandera *UF*.
 - Si $is_of = 1 \Leftrightarrow status_from_norm = 10$ ocurre un *overflow*, se levanta *OF* y el resultado final puede ser *inf* o *almost inf* (la representación de mayor magnitud antes de *inf*) según el modo de redondeo y el signo del resultado.
 - Si $is_uf = 1 \Leftrightarrow status_from_norm = 01$ ocurre un *underflow*, se levanta *UF* y el resultado final puede ser *zero* o *almost zero* (la representación de menor magnitud después de *zero*) según el modo de redondeo y el signo del resultado.
- **pre_final_state_1**: guarda el resultado inicial (con $sel = 0$) y pasa a **final_state**.
- **pre_final_state_2**: guarda el resultado actual (con $sel = 1$) y pasa a **final_state**.
- **final_state**: desactiva el *NORMALIZER*, activa *ready* y si el *ROUNDER* levanta la bandera *NX* ($inexact_flag = 1$) se retorna dicha bandera, si no, simplemente no se levanta bandera. El resultado actual es el final y retorna a **waiting_state** si *start* se desactiva.

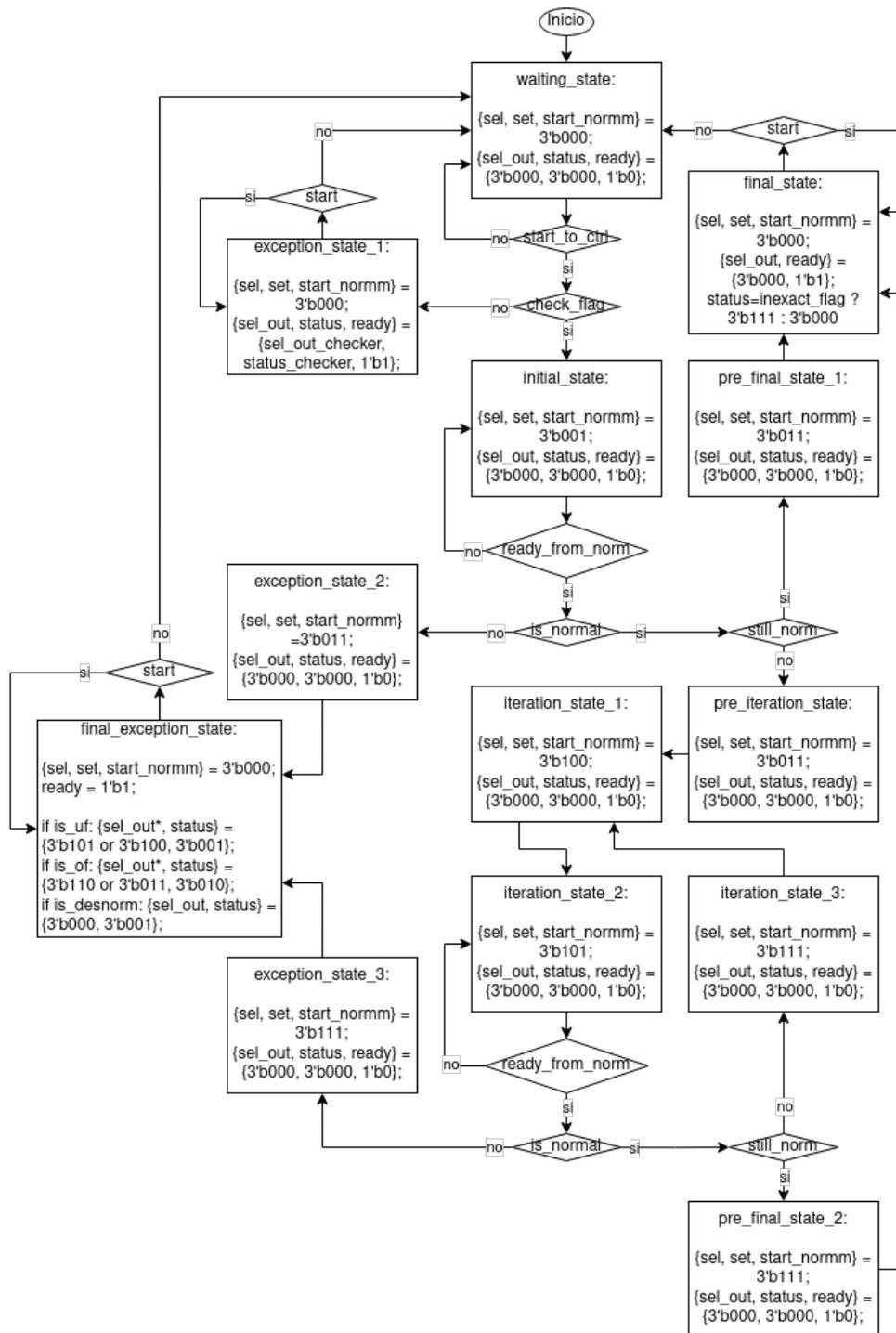


Figura 4.10: Diagrama de flujo detallado del módulo *fp_arithmetic_unit*.

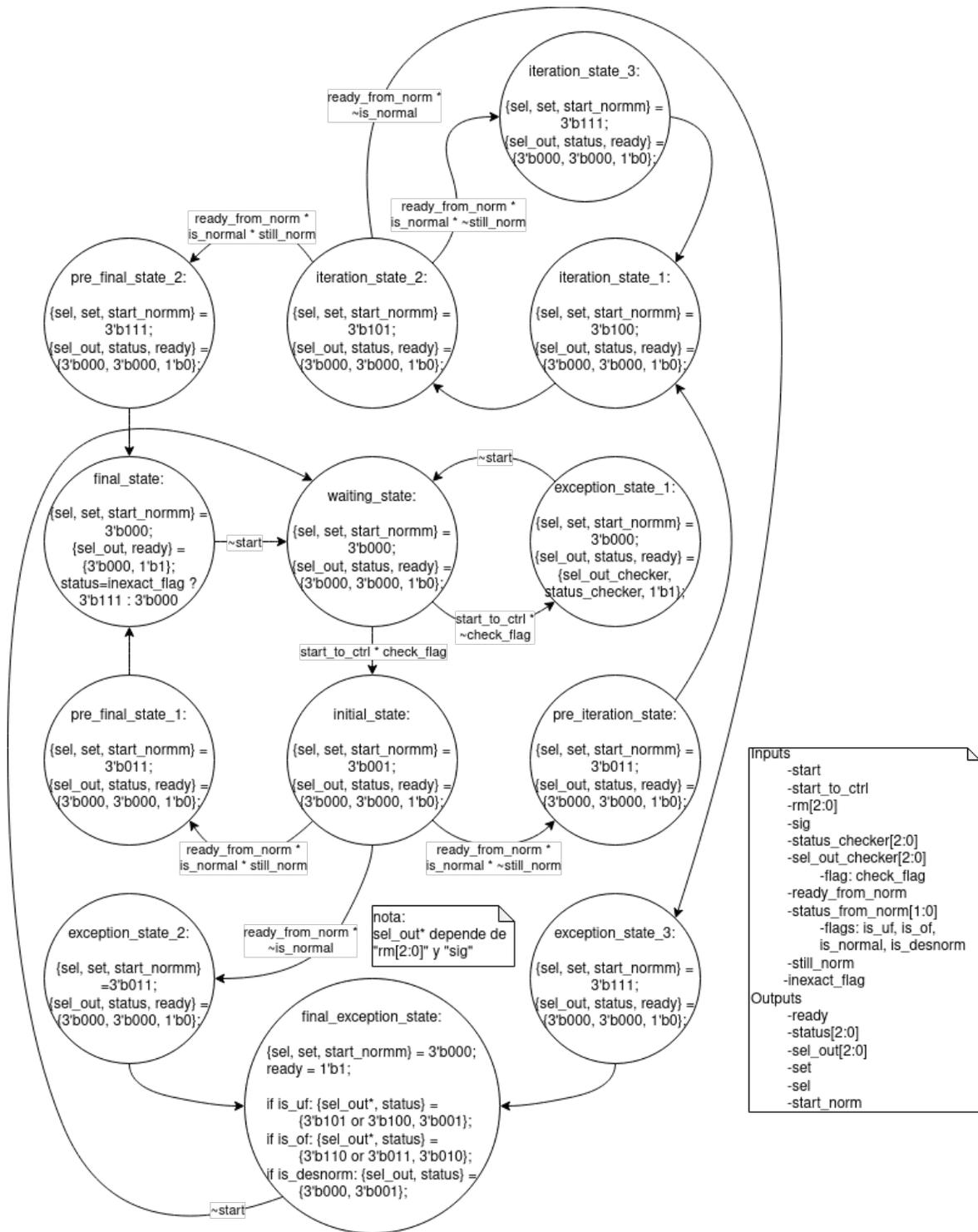


Figura 4.11: Diagrama MDS del módulo *fp_arithmetic_unit*.

A continuación, se detalla el diseño de los principales submódulos que conforman la unidad aritmética de punto flotante. Para el desarrollo de los mismo se vuelve a la metodología de diseño *Top-Down*. Es importante señalar que en algunos de estos módulos se repite la unidad “*inputs normalizer*”, tal como se puede ver en los diagramas de las figuras 4.13, 4.15, 4.19 y 4.23. Esta unidad corresponde a un simple módulo secuencial, *normalizer4significands_inputs*, encargado de tomar la mantisa de entrada y normalizarla en caso de presentar el bit silencioso (el primero a la izquierda de *fraction*) igual a cero (número desnormalizado). Las salidas de *normalizer4significands_inputs*

son la mantisa normalizada y un *offset*. Este ultimo indica el exponente negativo del número en cuestión (cuántos desplazamientos de posición se realizaron). Este módulo se implementa al detectar problemas con el cálculo de números desnormalizados y por razones de tiempo no se realiza una modificación mayor del diseño completo de *fp_arithmetic_unit*. Por ello se deja propuesto como mejora añadir este bloque en el módulo principal *fp_arithmetic_unit* y no en los submódulos para evitar duplicar *hardware*.

4.5.1.1 *FP Arithmetic Unit: ADDER*

Este módulo se encarga de calcular el resultado inicial de la suma de números de punto flotante, sin considerar la etapa de normalización o redondeo. Para ello, recibe las señales:

- **start, clk**: señal de inicio de ejecución y reloj, respectivamente.
- **sig1, sig2**: signo de cada operando.
- **exp1[7:0], exp2[7:0]**: exponente de cada operando.
- **man1[27:0], man2[27:0]**: mantisa de cada operando.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **sig_out**: signo del resultado.
- **exp_out[7:0]**: exponente del resultado.
- **man_out[27:0]**: mantisa del resultado.

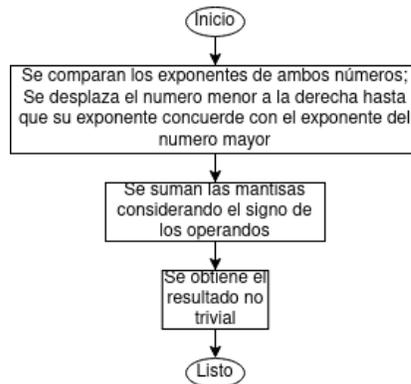


Figura 4.12: Diagrama de flujo simplificado del submódulo *ADDER*.

En la figura 4.12 se aprecia el diagrama de flujo considerado, el cual se basa en el diagrama de flujo para el sumador de punto flotante presentado en [12]. A partir de este diagrama se diseña el diagrama de bloques detallado de la figura 4.13, donde se aprecia que la unidad es completamente combinatorial exceptuando las unidades *“inputs normalizer”*. Con esto, se obtiene el resultado preliminar listo para pasar a la fase de normalización y redondeo.

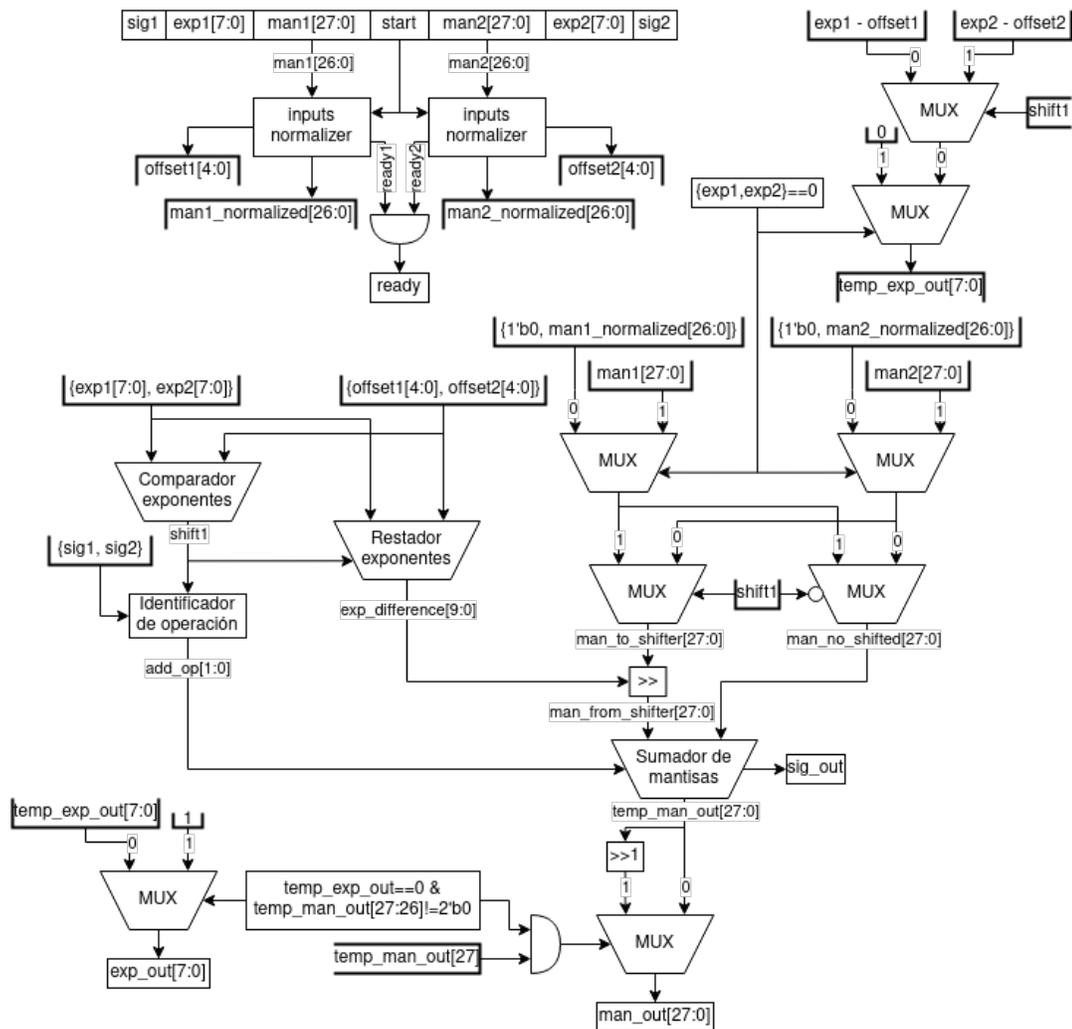


Figura 4.13: Diagrama de bloques detallado del submódulo *ADDER*.

4.5.1.2 *FP Arithmetic Unit: MULTIPLIER*

Este módulo se encarga de calcular el resultado inicial de la multiplicación de números de punto flotante, sin considerar la etapa de normalización o redondeo. Para ello, recibe las señales:

- **start, clk**: señal de inicio de ejecución y reloj, respectivamente.
- **exp1[7:0], exp2[7:0]**: exponente de cada operando.
- **man1[27:0], man2[27:0]**: mantisa de cada operando.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **exp_out[7:0]**: exponente del resultado.
- **man_out[27:0]**: mantisa del resultado.

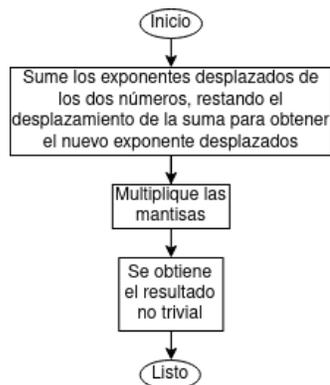


Figura 4.14: Diagrama de flujo simplificado del submódulo *MULTIPLIER*.

En la figura 4.14 se aprecia el diagrama de flujo considerado, el cual se basa en el diagrama de flujo para el multiplicador de punto flotante presentado en [12]. A partir de este diagrama de flujo se diseña el de bloques detallado de la figura 4.15, donde se aprecia que la unidad es completamente combinacional, exceptuando las unidades “*inputs normalizer*”. Es importante indicar que los bloques “Selector de exponente final” y “Selector de mantisa final” (este además requiere la señal $temp_man1[55]$), consideran el resultado del “Sumador de exponente”, $temp_exp[9:0]$, para determinar si ocurre un *overflow*, resultado desnormalizado o normal y así retornar el resultado preliminar correcto para pasar a la fase de normalización y redondeo.

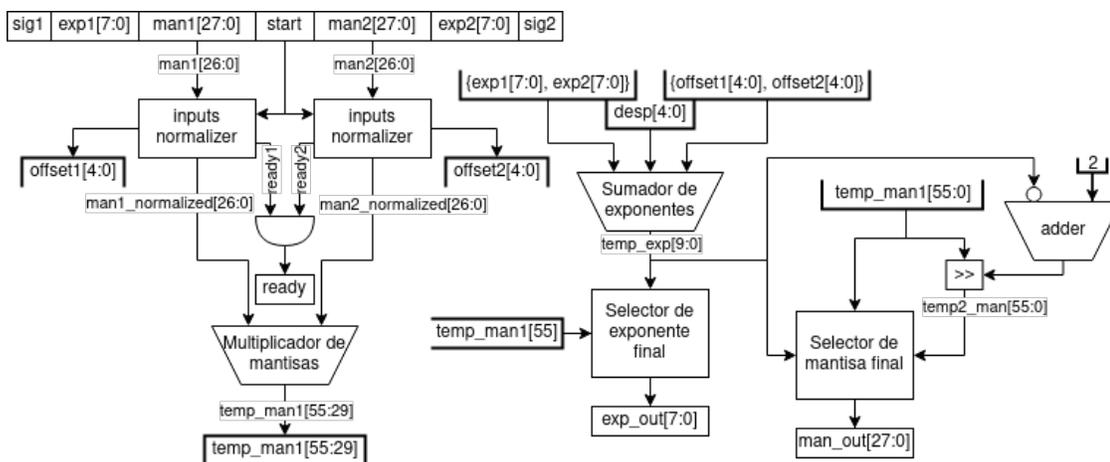


Figura 4.15: Diagrama de bloques detallado del submódulo *MULTIPLIER*.

4.5.1.3 *FP Arithmetic Unit: DIVISOR*

Este módulo se encarga de calcular el resultado inicial de la división de números de punto flotante, sin considerar la etapa de normalización o redondeo. Para ello, recibe las señales:

- **start, rst, clk**: señal de inicio de ejecución, *reset* y reloj, respectivamente.
- **exp1[7:0], exp2[7:0]**: exponente de cada operando.
- **man1[27:0], man2[27:0]**: mantisa de cada operando.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **exp_out[7:0]**: exponente del resultado.

- **man_out[27:0]**: mantisa del resultado.

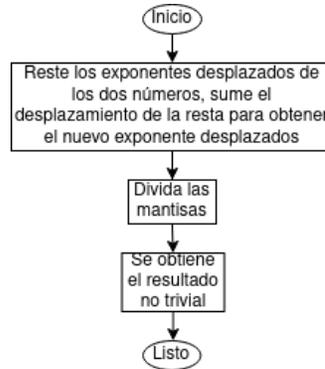


Figura 4.16: Diagrama de flujo simplificado del submódulo *DIVISOR*.

En la figura 4.16 se aprecia el diagrama de flujo considerado, el cual se infiere a partir del diagrama 4.14. A partir de este diagrama de flujo se diseña el de bloques detallado de la figura 4.17, donde se aprecia que la unidad es completamente combinacional, exceptuando la unidad “*Divisor de mantisas*”, el cual se explica en la sección 4.5.1.4. Es importante indicar que los bloques “Selector de exponente final” (este además requiere la señal $temp_man1[54]$) y “Selector de mantisa final” consideran el resultado del “Sumador de exponente”, $temp_exp[9:0]$, para determinar si ocurre un *overflow*, resultado desnormalizado o normal y así retornar el resultado preliminar correcto para pasar a la fase de normalización y redondeo.

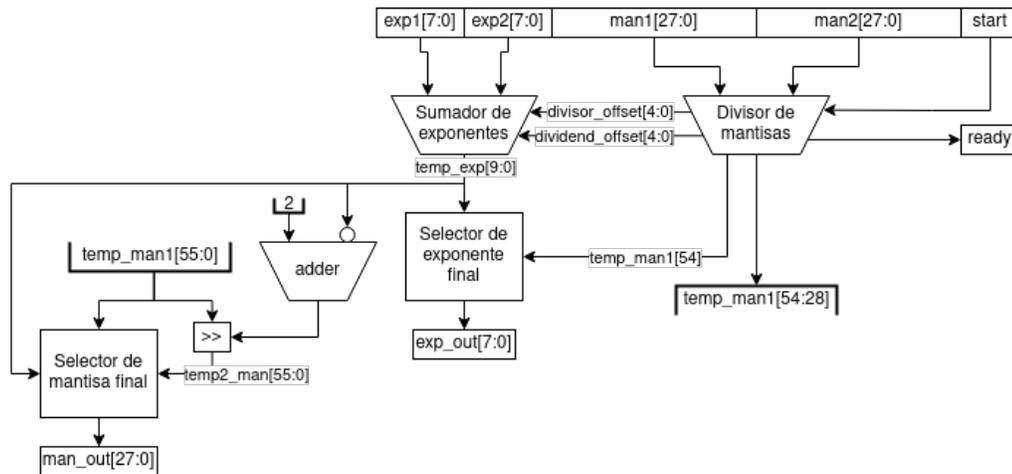


Figura 4.17: Diagrama de bloques detallado del submódulo *DIVISOR*.

4.5.1.4 *FP Arithmetic Unit: significands_divisor*

Este módulo se encarga de realizar la división de las mantisas. Para ello, recibe las entradas:

- **start**, **rst**, **clk**: señal de inicio de ejecución, *reset* y reloj, respectivamente.
- **dividend[26:0]**, **divisor[26:0]**: mantisas de dividendo y divisor, respectivamente.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **man_output[26:0]**: mantisa del resultado.

- **dividend_offset[4:0]**, **divisor_offset[4:0]**: son los *offsets* retornados por las unidades “*inputs normalizer*” del dividendo y divisor, respectivamente.

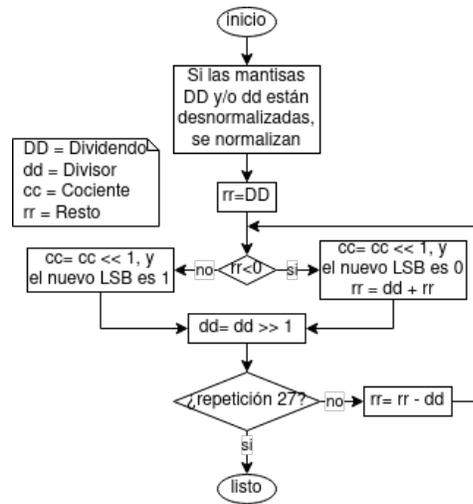


Figura 4.18: Diagrama de flujo simplificado del submódulo *significands_divisor*.

En la figura 4.18 se visualiza el diagrama de flujo simplificado con el algoritmo para realizar la división de mantisas, el cual se basa en el algoritmo clásico de la división (tal como se presenta en el capítulo 3.4 de [12]). Con este diagrama se elabora el de bloques detallado de la figura 4.19 donde se identifican las señales de control para el *FSM*.

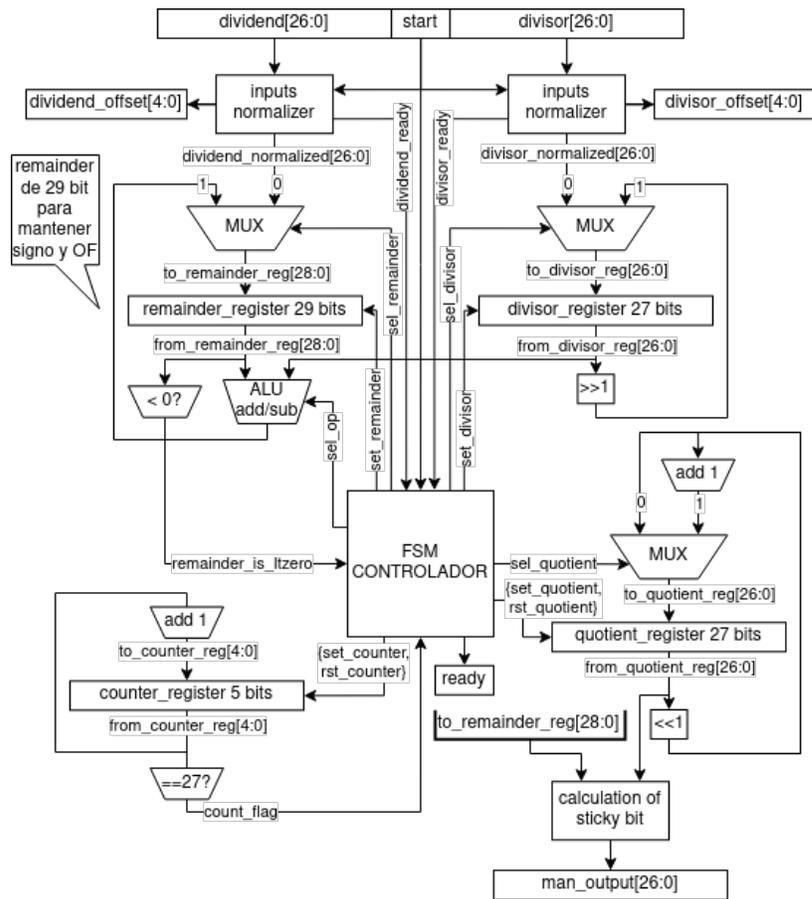


Figura 4.19: Diagrama de bloques detallado del submódulo *significands_divisor*.

En la figura 4.20 se presenta el diagrama de flujo detallado elaborado a partir de los diagramas de las figura 4.18 y 4.19. En este diagrama se identifican los estados necesarios para el controlador junto con el flujo de las señales de control y decisión. Considerando este último diagrama se confecciona el diagrama *MDS* de la figura 4.21.

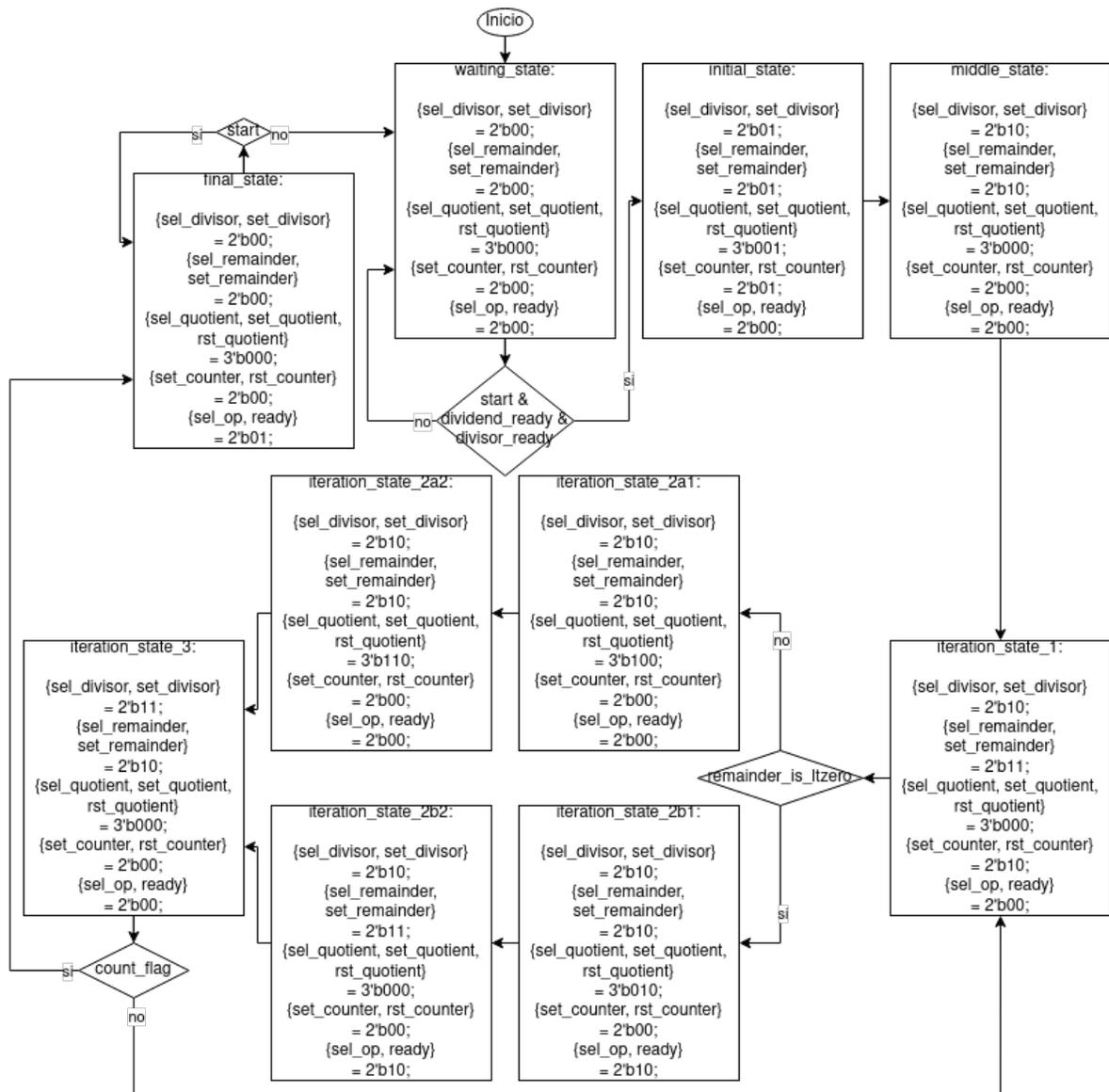


Figura 4.20: Diagrama de flujo detallado del submódulo *significands_divisor*.

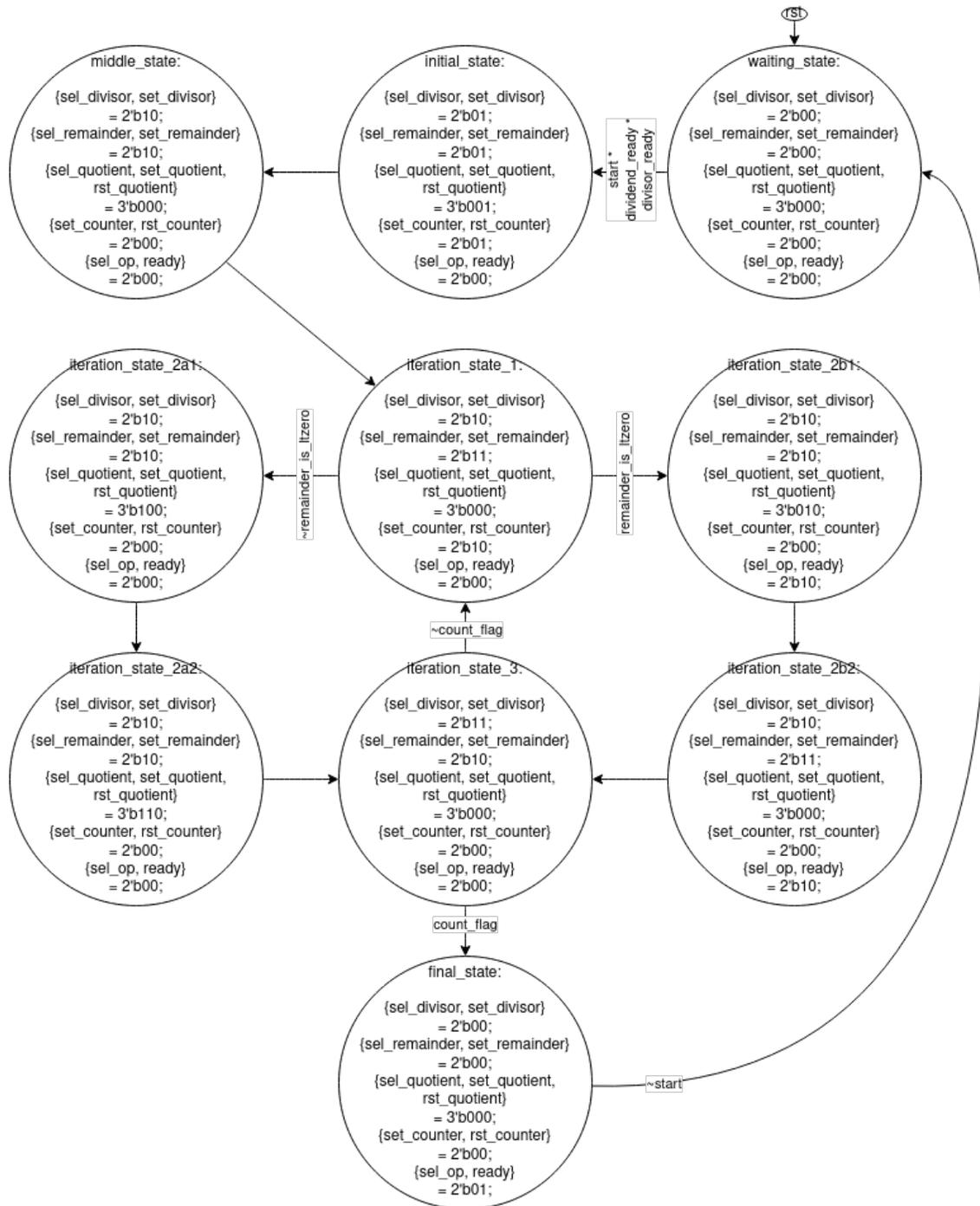


Figura 4.21: Diagrama MDS del submódulo *significands_divisor*.

A continuación, se detalla el rol de cada uno de los estados del controlador:

- **waiting_state:** espera la señal de *start* y que ambos operandos estén normalizados, cuando esto sucede el siguiente estado es **initial_state**.
- **initial_state:** guarda el divisor y resto inicial en los registros temporales e inicializa en cero los registros del cociente y del contador, luego pasa a **middle_state**.
- **middle_state:** cambia las entradas de los registros del divisor y del resto por la salida del nuevo resultado temporal, luego el siguiente estado es **iteration_state_1**.
- **iteration_state_1:** guarda el nuevo resto e incrementa en uno el contador, luego si el resto

es mayor o igual cero el siguiente estado es **iteration_state_2a1**, si no, pasa al estado **iteration_state_2b1**.

- **iteration_state_2a1**: cambia la entrada del cociente (del registro que lo contiene) para que reciba en su entrada el mismo cociente desplazado en una posición a la izquierda, llenando con 1. El siguiente estado es **iteration_state_2a2**.
- **iteration_state_2a2**: guarda el nuevo cociente y pasa a **iteration_state_3**.
- **iteration_state_2b1**: modifica la operación que realiza la pequeña *ALU* por una suma (por defecto es resta). No modifica la entrada del cociente, por ello recibe en su entrada el mismo cociente desplazado en una posición a la izquierda, llenando con 0. Guarda el nuevo cociente y pasa a **iteration_state_2b2**.
- **iteration_state_2b2**: guarda el nuevo resto y pasa a **iteration_state_3**.
- **iteration_state_3**: fija la operación de la *ALU* en resta nuevamente y guarda el nuevo divisor. Si el contador llega a 27 el siguiente estado es **final_state**, en caso contrario se vuelve a **iteration_state_1**.
- **final_state**: activa *ready* y vuelve a **waiting_state** cuando *start* se desactiva.

4.5.1.5 *FP Arithmetic Unit: SQRT*

Este módulo se encarga de calcular el resultado inicial de la raíz cuadrada de un número de punto flotante, sin considerar la etapa de normalización o redondeo. Para ello, recibe las señales:

- **start**, **rst**, **clk**: señal de inicio de ejecución, *reset* y reloj, respectivamente.
- **exp_in[7:0]**: exponente del operando.
- **man_in[27:0]**: mantisa del operando.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **exp_out[7:0]**: exponente del resultado.
- **man_out[27:0]**: mantisa del resultado.

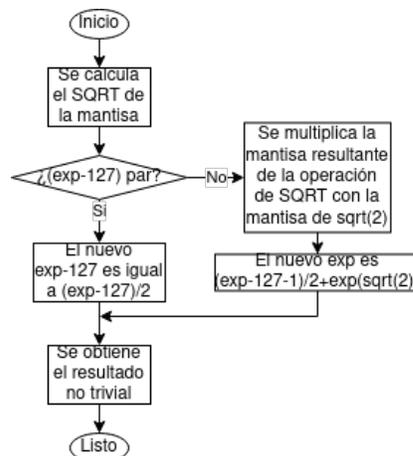


Figura 4.22: Diagrama de flujo simplificado del submódulo *SQRT*.

El algoritmo del diagrama de flujo de la figura 4.22 se basa en el trabajo realizado en [27], donde se propone una arquitectura para el cálculo de la raíz cuadrada de un número. A continuación,

se presenta el desarrollo matemático que determina este algoritmo. Sea X un número en punto flotante, Y su mantisa y K su exponente (sin sesgo, es decir que $K = exponent - 127$), se cumple:

$$\sqrt{X} = \sqrt{Y} \sqrt{2^K} = \begin{cases} \sqrt{Y} 2^{\frac{K}{2}} & \text{si } K \text{ es par} \\ \sqrt{Y} 2^{\lfloor \frac{K}{2} \rfloor} \sqrt{2} & \text{si } K \text{ es impar} \end{cases} \quad (4.1)$$

Luego, para hallar la raíz cuadrada del número basta hallar el de la mantisa, esta tarea es del módulo *significand_sqrt* descrito en la sección 4.5.1.6.

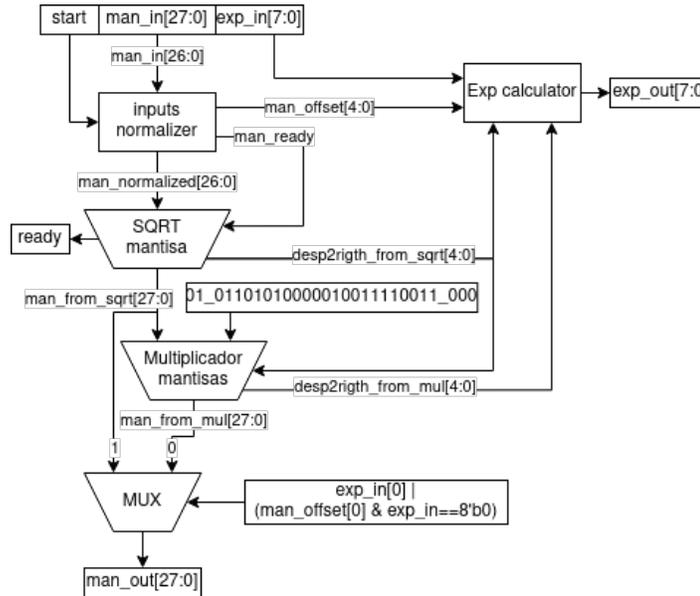


Figura 4.23: Diagrama de bloques detallado del submódulo *SQRT*.

A partir del diagrama de la figura 4.22 se confecciona el diagrama de bloques detallado de la figura 4.23. En este diagrama se identifican dos unidades secuenciales, el “*inputs normalizer*” y el “*SQRT mantisa*” (*significand_sqrt*), cuando finaliza la ejecución del primero comienza la del segundo. Por otra parte, el multiplicador de mantisas recibe la mantisa de $\sqrt{2}$ y el resultado de *significand_sqrt*. Nótese que el exponente sin sesgo de $\sqrt{2}$ es 0, con sesgo sería 127, por ello no hace falta considerarlo en el cálculo del exponente final. Además, *significand_sqrt* y el multiplicador de mantisas retornan *desp2righth_from_sqrt[4:0]* y *desp2righth_from_mul[4:0]*, respectivamente. Estas señales indican cuanto se debe desplazar a la derecha la mantisa resultante para que el exponente inicial concuerde con el resultado de las multiplicaciones realizadas en estos módulos. También, basta dejar la mantisa tal cual y sumar este desplazamiento al exponente.

4.5.1.6 *FP Arithmetic Unit: significand_sqrt*

Este módulo se encarga de calcular la raíz cuadrada de la mantisa. Para ello, recibe las entradas:

- **start, rst, clk**: señal de inicio de ejecución, *reset* y reloj, respectivamente.
- **frac_in[22:0]**: recibe, únicamente, el campo *fraction* del número en punto flotante.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **man_output[26:0]**: mantisa del resultado.
- **desp2righth_out[4:0]**: representa cuanto debe aumentar el exponente del resultado o cuanto se debe desplazar a la derecha el resultado para no tener que modificar el exponente.

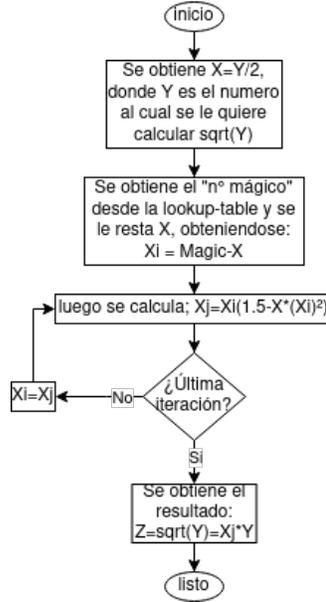


Figura 4.24: Diagrama de flujo simplificado del submódulo *significand_sqrt*.

En la figura 4.24 se aprecia el diagrama de flujo simplificado del módulo, este se basa en el algoritmo presentado en [27]. A continuación, se explica dicho algoritmo; en [27] se propone el método de *Newton-Raphson* para calcular la raíz cuadrada (y su inversa): para esto se comienza de un *guess*, g , inicial sobre el cual iterar para encontrar la función deseada, $g = (\sqrt{Y})^{-1}$, donde Y es la mantisa. Para aplicar *Newton-Raphson* se debe iterar $new_g = g - \frac{f(g)}{f'(g)}$, para esto se define la función:

$$error(g) = (g)^{-2} - ((\sqrt{Y})^{-1})^{-2} = \frac{1}{g^2} - Y = g^{-2} - Y \quad (4.2)$$

Luego se obtiene:

$$new_g = g - \frac{error(g)}{error'(g)} = g + \frac{g - Yg^3}{2} = \frac{3}{2}g - \frac{1}{2}Yg^3 = g(1.5 - 0.5Yg^2) \quad (4.3)$$

A partir de esta última ecuación se obtiene el algoritmo de la figura 4.24. Para la implementación, en [27], se propone un *Look-Up-Table* que determina el primer valor de g , esta tabla recibe Y y retorna el número mágico, *MagicNumber*, según lo indicado en la tabla 4.10. En concreto, se calcula $x = Y/2 = 0.5Y$ y $x_i = MagicNumber - x$, con esto se iteran las siguientes ecuaciones un número determinado de veces:

$$x_j = x_i(1.5 - xx_i^2) \quad (4.4)$$

$$x_i = x_j \quad (4.5)$$

Donde x_j sería new_g y x_i sería g . Al finalizar, $x_j \approx (\sqrt{Y})^{-1}$ y $\sqrt{Y} \approx x_j Y$, obteniendose el resultado deseado.

Considerando el algoritmo planteado y la arquitectura propuesta en [27], se confecciona el diagrama de bloques detallado de la figura 4.25. Este diagrama difiere del diseño original de [27], donde se realizan dos iteraciones simplemente duplicando el *hardware* (permitiendo un diseño puramente combinacional). En este caso, se realizan las iteraciones reutilizando *hardware*, por lo mismo, requiere de un controlador y más ciclos de reloj para su ejecución, tal como se puede apreciar en la figura 4.25.

Lookup-Table		
From	To	MagicNumber
1.000000000000000000000000	1.001111111111111111111111	1.10000001100001001111010
1.010000000000000000000000	1.010111111111111111111111	1.10000100110111010000111
1.011000000000000000000000	1.011111111111111111111111	1.10001010111111001100111
1.100000000000000000000000	1.100111111111111111111111	1.10010001011001000011000
1.101000000000000000000000	1.101111111111111111111111	1.10011001000100100100010
1.110000000000000000000000	1.110111111111111111111111	1.10100010100010100001111
1.111000000000000000000000	1.111111111111111111111111	1.10110101011111011100101

Tabla 4.10: *Lookup-Table* propuesto por [27].

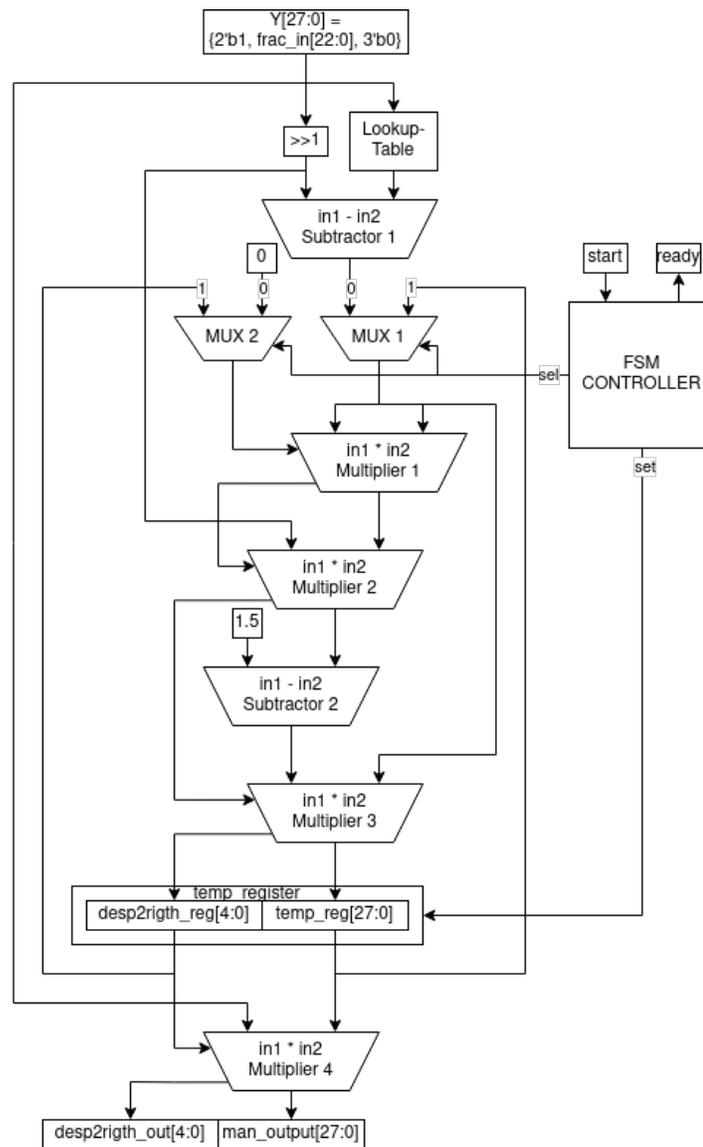


Figura 4.25: Diagrama de bloques detallado del submódulo *significand_sqrt*.

Finalmente, la máquina de estados resulta bastante simple. Se podría añadir un contador para determinar el número de iteraciones. Sin embargo, por simplicidad solo se añaden más estados al *FSM*. En la figura 4.26 se observa el diagrama de flujo detallado, a partir del cual se confecciona el diagrama *MDS* de la figura 4.27.

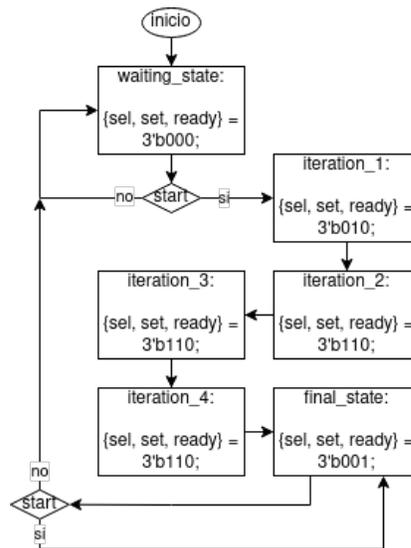


Figura 4.26: Diagrama de flujo detallado del submódulo *significand_sqrt*.

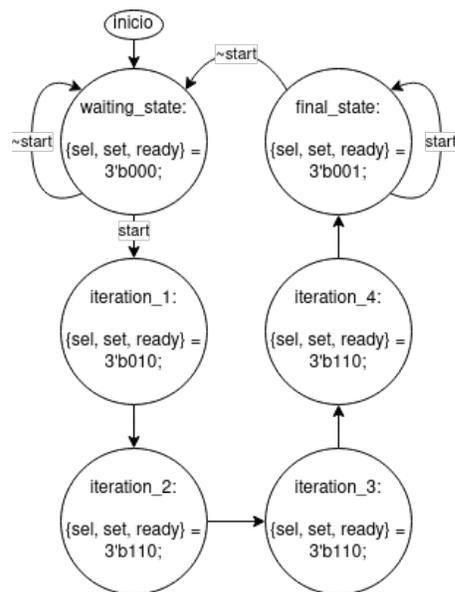


Figura 4.27: Diagrama *MDS* del submódulo *significand_sqrt*.

A continuación, se explica el rol de cada uno de los estados del controlador:

- **waiting_state:** espera la señal de *start*, luego el siguiente estado es **iteration_1**.
- **iteration_1:** guarda el resultado inicial, luego pasa a **iteration_2**.
- **iteration_2**, **iteration_3** y **iteration_4:** todos estos estados guardan el nuevo resultado considerando la mantisa obtenida en la iteración anterior. El estado **iteration_2** pasa a **iteration_3**, **iteration_3** pasa a **iteration_4** e **iteration_4** a **final_state**.
- **final_state:** activa *ready* y vuelve a **waiting_state** cuando *start* se desactiva.

4.5.1.7 *FP Arithmetic Unit: Exceptions and trivial cases checker*

Este módulo, *excep_triv_checker*, se encarga de verificar las entradas de la operación a realizar por la *fp_arithmetic_unit*. Esto con el fin de identificar de forma inmediata cuando los operandos no son válidos, presentan alguna excepción o el resultado es trivial. Para esta tarea recibe las siguientes señales:

- **start**, **clk**: señal de inicio de ejecución y reloj, respectivamente.
- **sig1**, **sig2**: signo de cada operando.
- **exp1[7:0]**, **exp2[7:0]**: exponente de cada operando.
- **frac1[23:0]**, **frac2[23:0]**: recibe, únicamente, el campo *fraction* y el bit silencioso de cada operando.
- **rm[2:0]**: el *rounding mode* seleccionado. Ver tabla 4.7.
- **op[1:0]**: señal que codifica la operación a realizar: 00 para la suma, 01 la multiplicación, 10 la división y 11 para el cálculo de raíz cuadrada.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **status_out[2:0]**: señal que indica la bandera que se genera. Su codificación es la misma que la presentada en la tabla 4.9.
- **sel_out[2:0]**: señal que indica cual es la salida. Esta señal se codifica según lo indicado en la tabla 4.11 y comparte codificación con la señal de control del mismo nombre del *FSM* de la figura 4.9.

sel_out[2:0]	
Code	Out. selected
000	No trivial
001	In1
010	In2
011	Almost inf
100	Zero
101	Almost zero
110	Inf
111	NaN

Tabla 4.11: Codificación de la salida *sel_out* del módulo *excep_triv_checker* y del *FSM* del módulo *fp_arithmetic_unit* (ver figura 4.9).

El módulo es completamente combinacional, exceptuando dos unidades “*inputs normalizer*” consideradas para las entradas *frac1* y *frac2*. La lógica combinacional está definida por el diagrama de flujo de la figura 4.28.

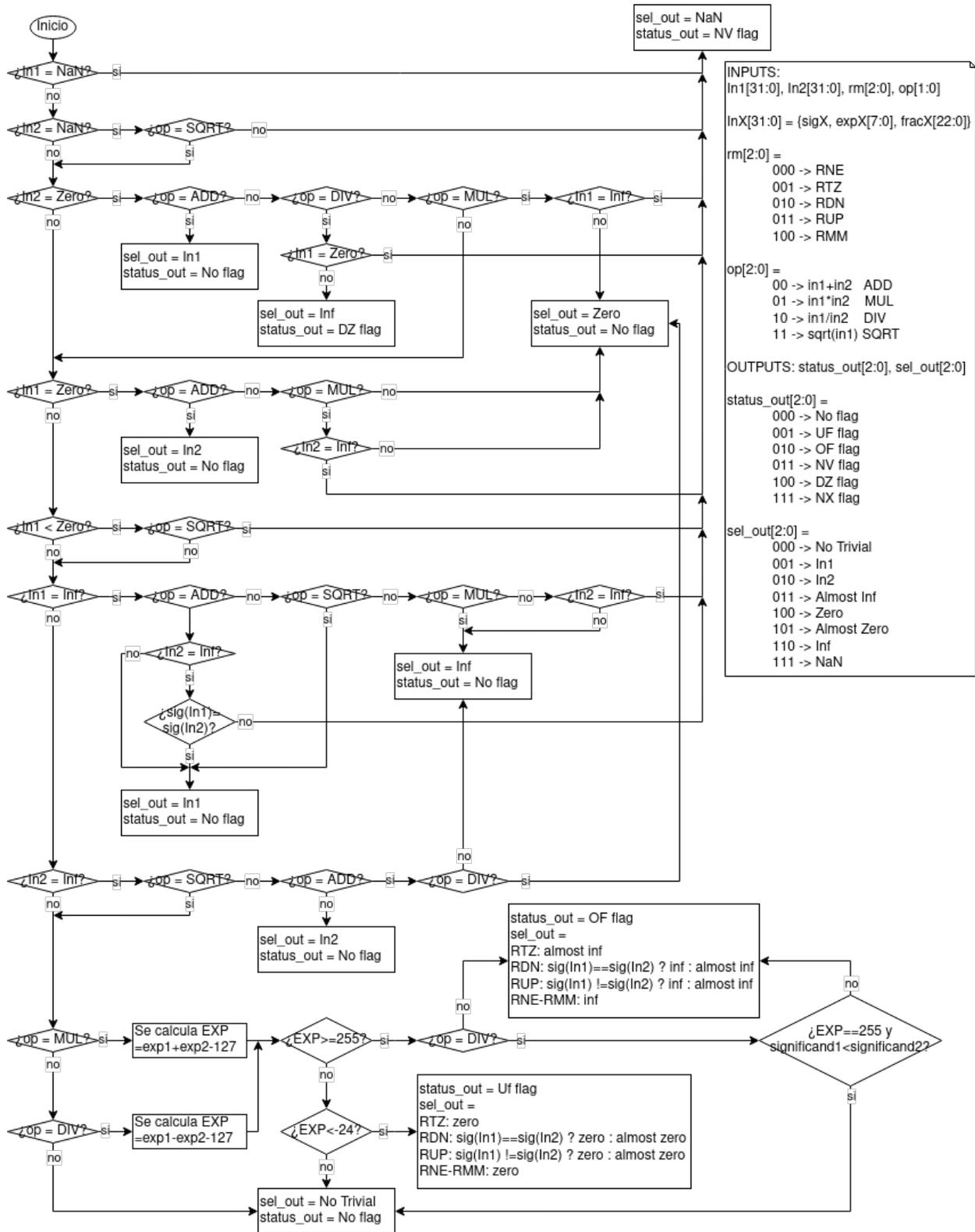


Figura 4.28: Diagrama de flujo del submódulo *except_triv_checker*.

4.5.1.8 *FP Arithmetic Unit: NORMALIZER*

Este módulo se encarga de recibir el resultado preliminar del módulo *fp_arithmetic_unit* y lo normaliza si corresponde. Para esta tarea recibe las siguientes señales:

- **start, rst, clk**: señal de inicio de ejecución, *reset* y reloj, respectivamente.
- **exp_in[7:0]**: exponente del operando.
- **man_in[27:0]**: mantisa del operando.

Y retorna:

- **ready**: señal de finalización de ejecución.
- **status[1:0]**: señal que indica si el resultado final está correcto y normalizado o si es desnormalizado, o si ocurre un *underflow* u *overflow*. Ver tabla 4.12.
- **exp_out[7:0]**: exponente del resultado.
- **man_out[27:0]**: mantisa del resultado.

status[1:0]	
Code	Status
00	Normalized
01	Underflow
10	Overflow
11	Desnormalized

Tabla 4.12: Codificación de la salida *status* del módulo *NORMALIZER*.

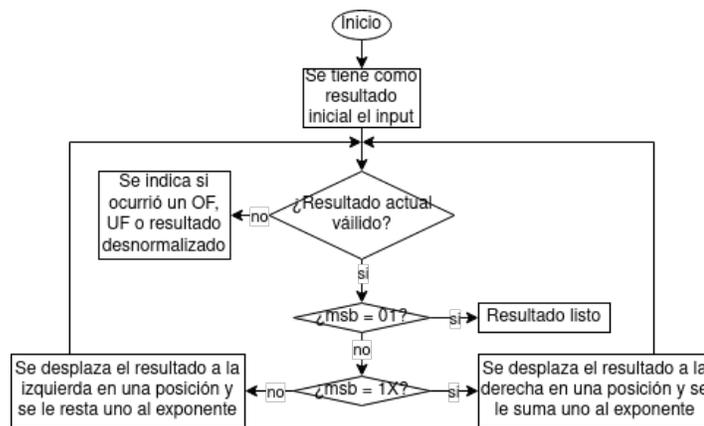


Figura 4.29: Diagrama de flujo simplificado del submódulo *NORMALIZER*.

En la figura 4.29 se observa el diagrama de flujo simplificado con el algoritmo utilizado para la normalización, donde $msb = man_in[27 : 26]$. A partir de este diagrama de flujo se confecciona el diagrama de bloques de la figura 4.30.

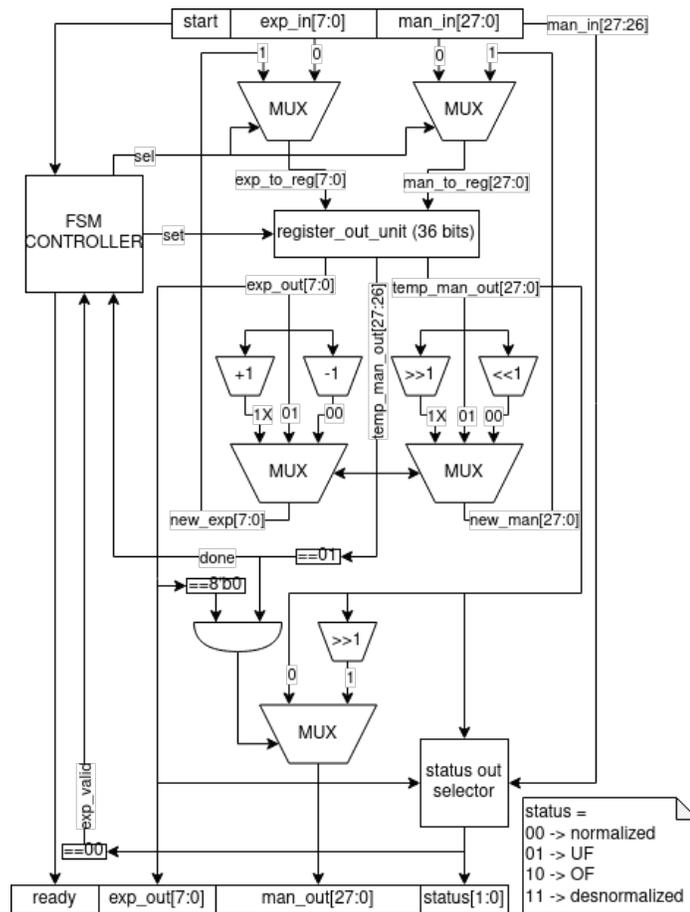


Figura 4.30: Diagrama de bloques detallado del submódulo *NORMALIZER*.

En la figura 4.30 se pueden apreciar las señales de decisión para la máquina de estados, las cuales son: *done* para indicar que el resultado está normalizado y *exp_valid*, que indica si *status* = 00, para informar que el resultado actual es válido. Por otra parte, la unidad “*status out selector*” recibe las señales necesarias para determinar el valor de *status*.

Con la ayuda de los diagramas de las figuras 4.29 y 4.30, se construye el diagrama de flujo detallado de la figura 4.31 y, posteriormente, el diagrama *MDS* de la figura 4.32. Estos últimos definen el funcionamiento de la máquina de estados que rige la ejecución del módulo. A continuación, se explica el rol de cada estado:

- **waiting_state**: espera la señal de *start*, luego el siguiente estado es **initial_state**.
- **initial_state**: guarda el resultado inicial y pasa al estado **iteration_state** si aún no está normalizado pero sigue siendo un resultado válido. Si ya está normalizado o deja de ser válido pasa a **final_state**.
- **iteration_state**: en cada ciclo actualiza al nuevo resultado considerando el anterior, si ya está normalizado o deja de ser válido pasa a **final_state**.
- **final_state**: activa *ready* y vuelve a **waiting_state** cuando *start* se desactiva.

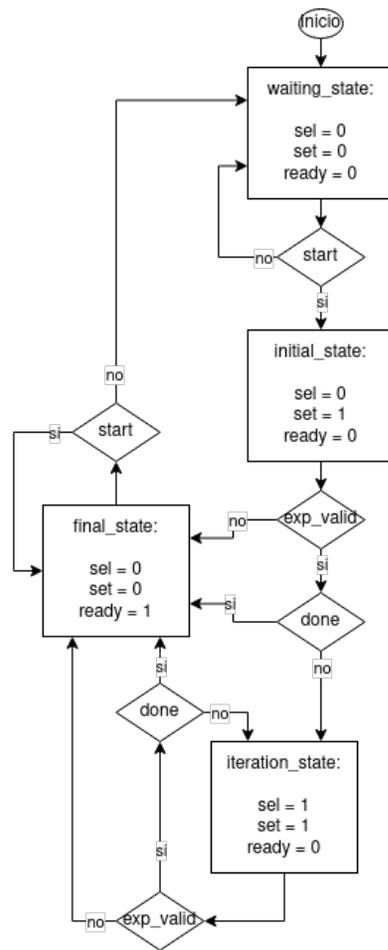


Figura 4.31: Diagrama de flujo detallado del submódulo *NORMALIZER*.

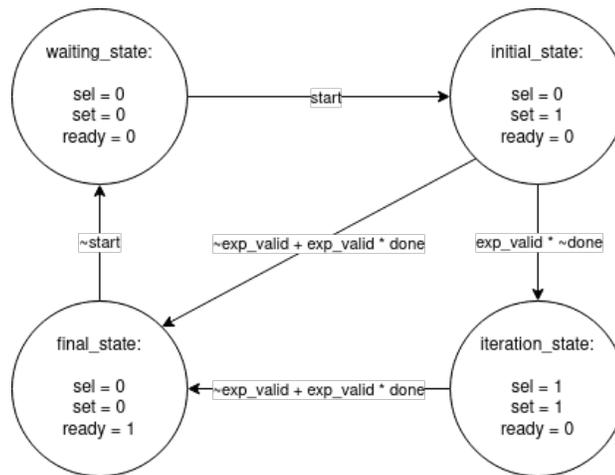


Figura 4.32: Diagrama *MDS* del submódulo *NORMALIZER*.

4.5.1.9 FP Arithmetic Unit: *ROUNDER*

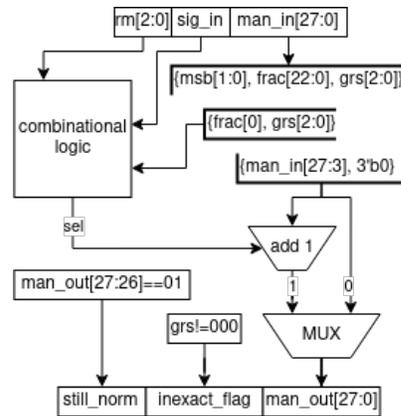


Figura 4.33: Diagrama de bloques detallado del submódulo *ROUNDER*.

Este módulo es completamente combinacional y se encarga de recibir la salida del *NORMALIZER* para aplicar el modo de redondeo seleccionado. Para esta tarea recibe las entradas:

- **rm[2:0]**: el modo de redondeo.
- **sig_in**: signo del resultado sin redondear.
- **man_in[27:0]**: mantisa del resultado sin redondear.

Y retorna:

- **still_norm**: bandera que indica que el resultado sigue estando normalizado.
- **inexact_flag**: bandera que indica que el resultado es inexacto.
- **man_out[27:0]**: mantisa del resultado después del redondeo.

En la figura 4.33 se visualiza el diagrama de bloques detallado del módulo, donde se puede ver el bloque “*combinational logic*”. Este bloque se encarga de decidir, según lo indicado en la tabla 4.7, si truncar o redondear el resultado. Luego, el módulo utiliza las banderas *still_norm* y *inexact_flag* para informar a *fp_arithmetic_unit* el estado del resultado, si sigue normalizado y/o es un resultado inexacto, respectivamente.

4.5.2 FP Converter Unit

Este módulo, *fp_converter*, se encarga de realizar las conversiones de números enteros, con y sin signo, a números en punto flotante y viceversa. Para ello, recibe las siguientes entradas:

- **start, rst, clk**: señal de inicio de ejecución, de *reset* y reloj, respectivamente.
- **rm[2:0]**: el *rounding mode* seleccionado. Ver tabla 4.7.
- **option**: señal que indica la operación a realizar. Si es 1 se realiza la conversión de entero a punto flotante y viceversa si es 0.
- **integer_is_signed**: si es 1 indica que el entero de la conversión es con signo, en caso de ser 0 es sin signo.
- **in[31:0]**: número de entrada, ya sea entero o punto flotante, para la conversión.

Y retorna:

- **ready**: señal que indica que la conversión se ha completado.
- **NV, NX**: banderas de conversión no válida e inexacta, respectivamente.

- `out[31:0]`: el resultado de la conversión.

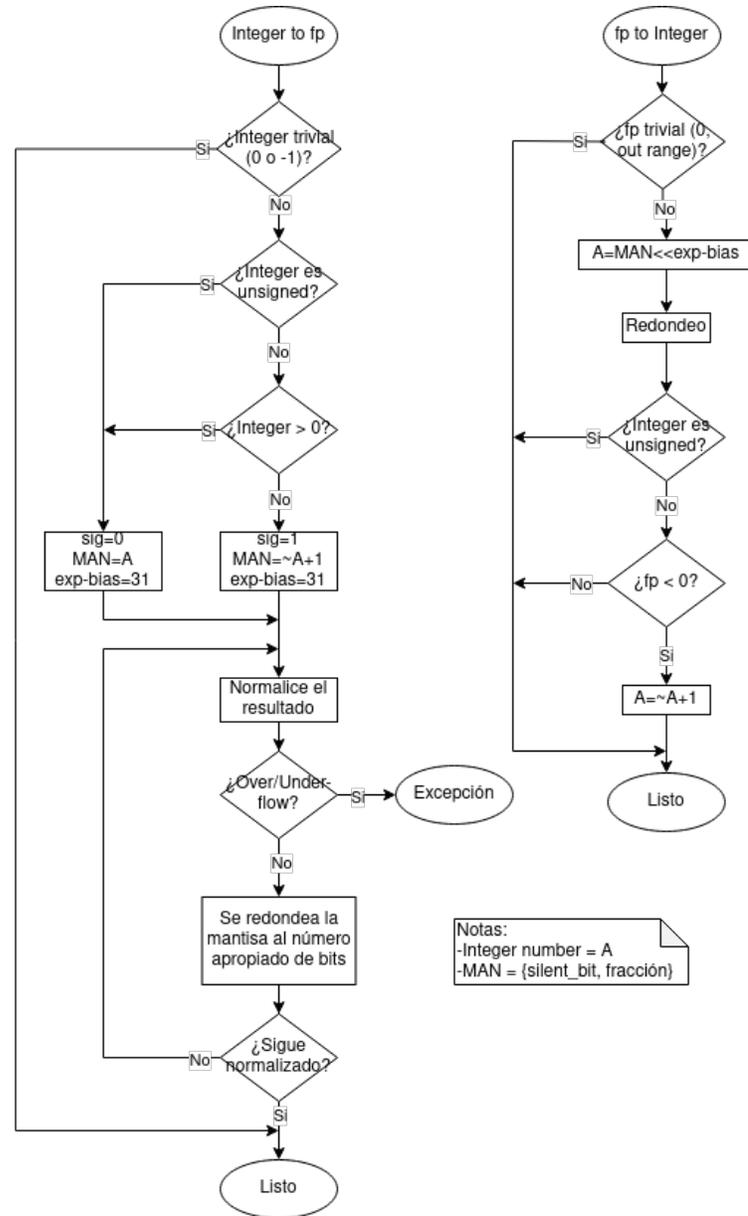


Figura 4.34: Diagramas de flujo simplificado del módulo `fp_converter`.

Para este módulo se confeccionan dos diagramas de flujo simplificado, los cuales se pueden ver en la figura 4.34, uno para cada tipo de conversión. Ambos algoritmos son la aplicación directa de los conceptos básicos presentados en la sección 2.1.3. Además, en estos diagramas se aprecia que la conversión de número entero a punto flotante es más compleja que la de punto flotante a entero, esto se debe a la necesidad de incluir las fases de normalización y redondeo similares a las requeridas por la unidad `fp_arithmetic_unit`.

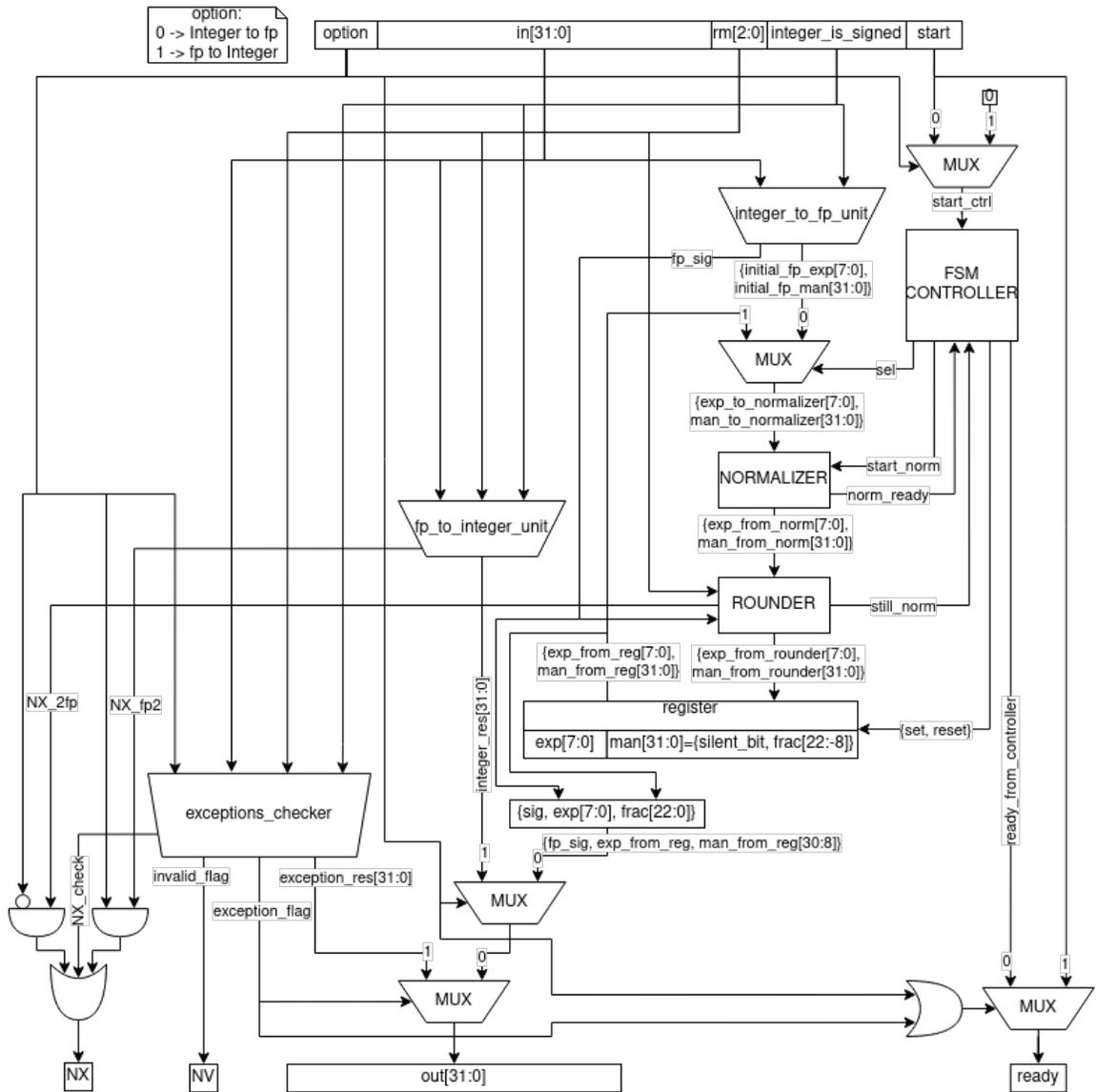


Figura 4.35: Diagrama de bloques detallado del módulo *fp_converter*.

Considerando los diagramas de la figura 4.34 se elabora el diagrama de bloques detallado de la figura 4.35, donde se identifican los principales 5 submódulos funcionales del módulo:

- **exceptions_checker**: unidad encargada de identificar los casos triviales y excepciones en las conversiones. Detallado en la sección 4.5.2.1.
- **fp_to_integer_unit**: unidad que realiza la conversión de número de punto flotante a entero. Detallado en la sección 4.5.2.2.
- **integer_to_fp_unit**: unidad que realiza la conversión de número entero a punto flotante. Detallado en la sección 4.5.2.4.
- **NORMALIZER**: normaliza el resultado de la conversión de número entero a punto flotante. Detallado en la sección 4.5.2.5.
- **ROUNDER**: realiza el redondeo del resultado de la conversión de número entero a punto flotante siguiendo la tabla 4.7. Detallado en la sección 4.5.2.6.

Además, en la figura 4.35 se aprecian las señales de control y de decisión del *FSM* que rige la ejecución de la unidad de conversión. Nótese además, que todos los bloques, exceptuando *NORMALIZER*

y *FSM CONTROLLER*, son combinacionales. Por otra parte, es importante señalar que el *FSM* se activa únicamente para la conversión de entero a flotante.

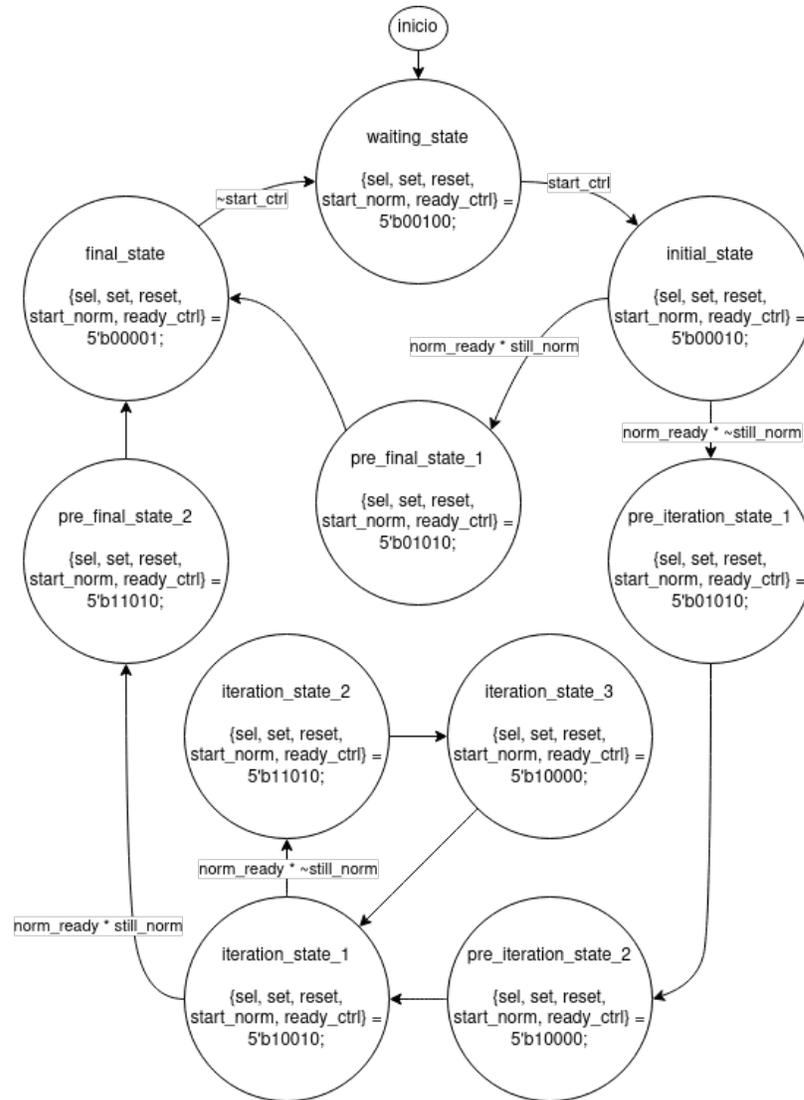


Figura 4.36: Diagrama *MDS* del módulo *fp_converter*.

A partir del diagrama de flujo para la conversión de entero a punto flotante (figura 4.34) y diagrama de bloques detallado (figura 4.35) se confecciona, directamente, el diagrama *MDS* (figura 4.36) para la máquina de estados del módulo. A continuación se explica cada estado de la misma:

- **waiting_state:** mantiene el registro temporal con la respuesta en 0 (*reset* = 1) y espera la señal *start_ctrl* para iniciar la ejecución, cuando esto sucede el siguiente estado es **initial_state**.
- **initial_state:** inicia la ejecución del *NORMALIZER* (*start_norm* = 1) y espera a que finalice su ejecución (*norm_ready*=1). Cuando esto sucede, depende de la señal *still_norm* el siguiente estado, si es 1 se pasa a **pre_final_state_1**, de ser 0 a **pre_iteration_state_1**.
- **pre_iteration_state_1:** guarda el resultado inicial en el registro temporal (*set* = 1) y pasa al estado **pre_iteration_state_2**.
- **pre_iteration_state_2:** desactiva el *NORMALIZER* (*start_norm* = 0) y cambia la entrada del mismo (*sel* = 1) por el del resultado en curso. El siguiente estado es **iteration_state_1**.

- **iteration_state_1**: inicia la ejecución del *NORMALIZER* ($start_norm = 1$) y espera a que finalice su ejecución ($norm_ready=1$). Cuando esto sucede, depende de la señal *still_norm* el siguiente estado, si es 1 se pasa a **pre_final_state_2**, de ser 0 a **iteration_state_2**.
- **iteration_state_2**: guarda el resultado actual en el registro temporal ($set = 1$) y pasa al estado **iteration_state_3**.
- **iteration_state_3**: desactiva el *NORMALIZER* ($start_norm = 0$) y pasa al estado **iteration_state_1**.
- **pre_final_state_1**: guarda el resultado inicial en el registro temporal ($set = 1$) y pasa al estado **final_state**.
- **pre_final_state_2**: guarda el resultado actual en el registro temporal ($set = 1$) y pasa al estado **final_state**.
- **final_state**: desactiva el *NORMALIZER* ($start_norm = 0$), activa *ready_from_controller* y retorna a **waiting_state** si *start_ctrl* se desactiva.

A continuación, se presentan los principales submódulos del *fp_converter*.

4.5.2.1 *FP Converter Unit: exceptions_checker*

El módulo *converter_exceptions_checker* es completamente combinacional y es el encargado de verificar la entrada del conversor e identificar de forma inmediata los casos triviales, entradas no válidas o excepciones. Para esta tarea recibe las siguientes señales:

- **option**: si es 1 la conversión es de punto flotante a entero, en caso de ser 0 es la conversión inversa.
- **integer_is_signed**: si es 1 el entero involucrado en la conversión es con signo, de ser 0 es sin signo.
- **rm[2:0]**: el *rounding mode* seleccionado. Ver tabla 4.7.
- **in[31:0]**: número de entrada para la conversión.

Y retorna:

- **invalid_flag, inexact_flag**: banderas de operación inválida e inexacta, respectivamente.
- **exception_flag**: bandera que indica que la respuesta final del módulo *fp_converter* es trivial y viene dada por *exception_res* cuando es 1, en caso de ser 0 la respuesta no es trivial.
- **exception_res[31:0]**: respuesta trivial, válida cuando *exception_flag* = 1, del módulo *fp_converter*.

En la figura 4.37 se aprecia el diagrama de flujo que determina la lógica combinacional del submódulo.

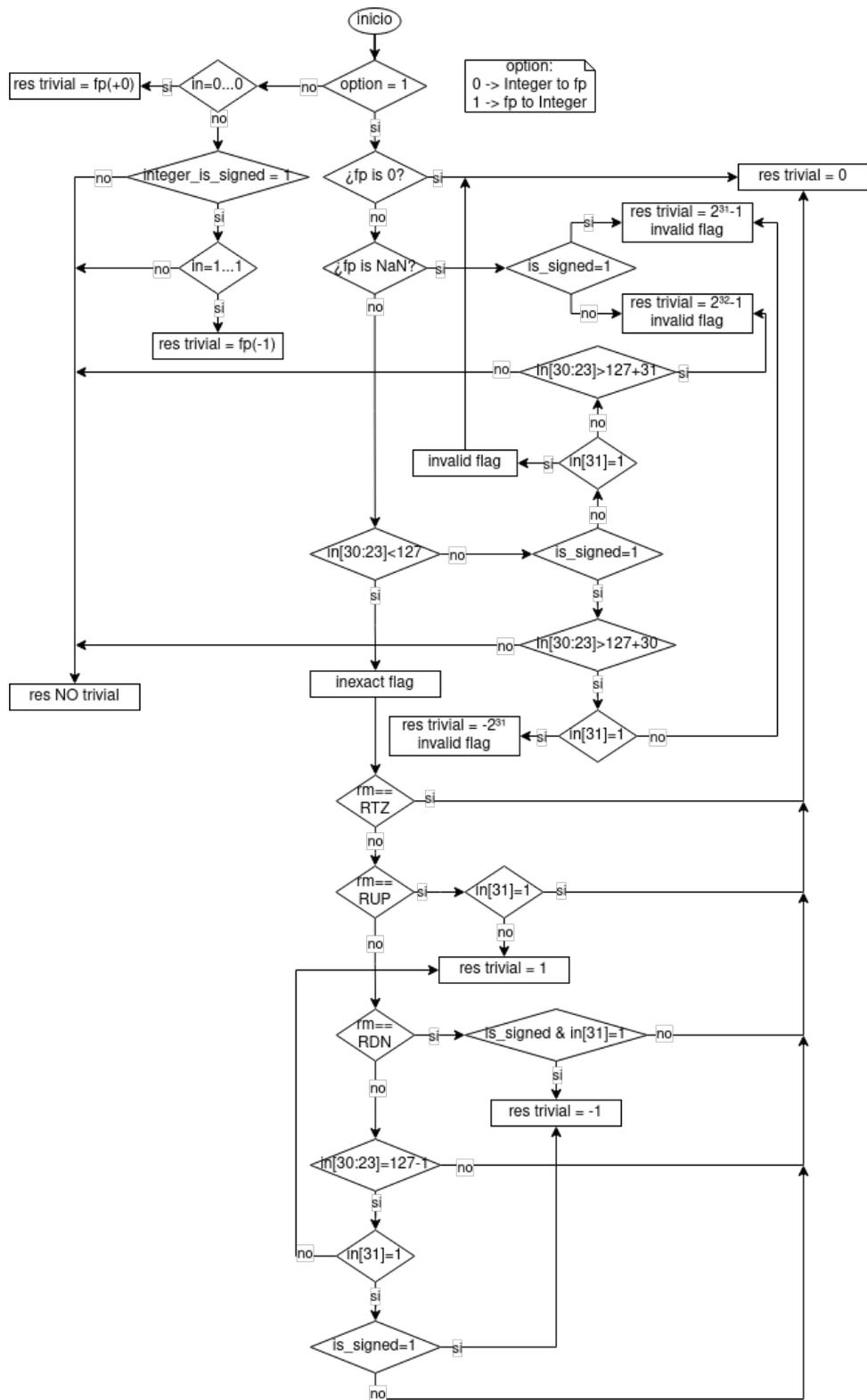


Figura 4.37: Diagrama de flujo del submódulo *converter_exceptions_checker* del módulo *fp_converter*.

4.5.2.2 FP Converter Unit: *fp_to_integer_unit*

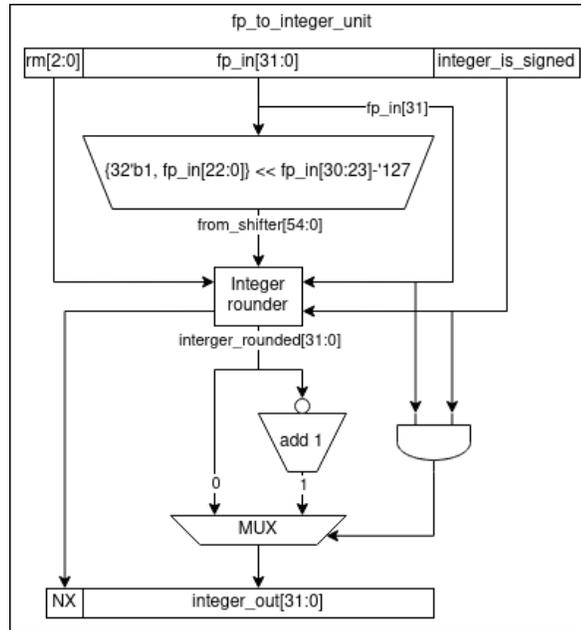


Figura 4.38: Diagrama de bloques detallado del submódulo *fp_to_integer_unit* del módulo *fp_converter*.

Este módulo, *fp_to_integer_unit*, es el encargado de realizar la conversión punto flotante a entero y es completamente combinacional. Para esta tarea recibe las señales:

- **integer_is_signed**: señal que indica que el entero tiene signo.
- **rm[2:0]**: modo de redondeo seleccionado.
- **fp_in[31:0]**: número flotante a convertir.

Y retorna:

- **NX**: bandera de operación inexacta.
- **integer_out[31:0]**: entero resultante.

En la figura 4.38 se aprecia el diagrama de bloques detallado, el cual se confecciona a partir del diagrama de flujo específico para esta conversión de a la figura 4.34. Como se aprecia en la figura este módulo presenta una arquitectura simple basada en los conceptos presentados en la sección 2.1.3. El bloque “*Integer rounder*” se explica a continuación.

4.5.2.3 FP Converter Unit: *Integer rounder*

El módulo combinacional *integer_rounder* realiza el redondeo del número entero resultante de la conversión de punto flotante a entero. Para ello, utiliza el mismo criterio de la tabla 4.7. Las entradas que recibe son:

- **is_signed**: si es 1 el entero es con signo, de ser 0 es sin signo.
- **sig_in**: indica el signo del entero si lo hubiese.
- **rm[2:0]**: modo de redondeo seleccionado.
- **integer_in[54:0]**: entero por redondear, observe que tiene 55 bits (32+23) para no perder ningún bit del campo *fraction* del punto flotante entrante. Esto es importante a la hora de calcular el *sticky* bit a considerar para el redondeo.

Y retorna:

- ***NX***: bandera que de operación inexacta.
- ***integer_out[31:0]***: entero redondeado.

EL entero resultante viene dado por $integer_in[54:23]$, el cálculo de los bits de redondeo se realiza de la siguiente forma: g (*guard*) y r (*round*) corresponden a los bits $integer_in[22]$ e $integer_in[21]$, respectivamente, mientras que s (*sticky*) es igual a 1 solo si se cumple que $integer_in[20 : 0] \neq 0$. Por otra parte, NX es 0 cuando $grs = 000$, en caso contrario es 1. Finalmente, el módulo considera el signo del entero para determinar la salida del módulo y considera la posibilidad de *overflow* (debido a como se realiza la conversión no hace falta manejar el caso de *underflow*).

4.5.2.4 FP Converter Unit: *integer_to_fp_unit*

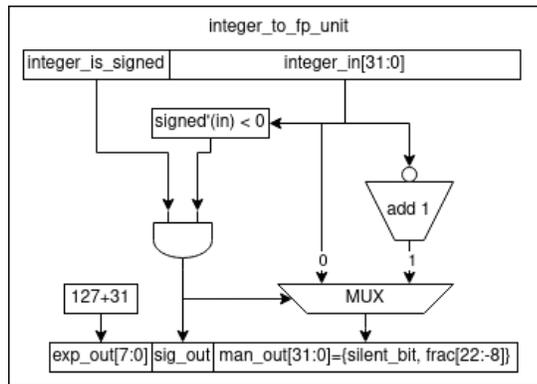


Figura 4.39: Diagrama de bloques detallado del submódulo *integer_to_fp_unit* del módulo *fp_converter*.

El módulo combinacional *integer_to_fp_unit* es el encargado de obtener el resultado inicial de la conversión de entero a punto flotante. Para ello, recibe las entradas:

- ***integer_is_signed***: señal que indica que el entero de entrada tiene signo.
- ***integer_in[31:0]***: entero a convertir.

Para retornar:

- ***sig_out***: signo del punto flotante resultante.
- ***exp_out[7:0]***: exponente del punto flotante resultante.
- ***man_out[31:0]***: mantisa del punto flotante resultante.

En la figura 4.39 se aprecia el diagrama de bloques detallado del submódulo, el cual se confecciona a partir del diagrama de flujo, específico para esta operación, de la figura 4.34. Las salidas de este módulo deben pasar por las fases de normalización y redondeo. A continuación, se explican los módulos encargados de estas tareas.

4.5.2.5 FP Converter Unit: NORMALIZER

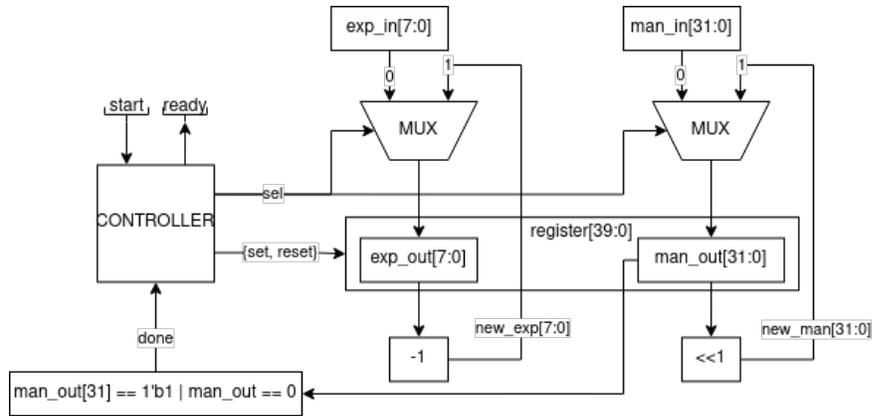


Figura 4.40: Diagrama de bloques detallado del submódulo *normalizer_integer2fp* del módulo *fp_converter*.

El submódulo *normalizer_integer2fp* se encarga de normalizar el resultado preliminar del módulo *integer_to_fp_unit*. Para esta tarea recibe las señales:

- **start**, **rst**, **clk**: señales de inicio de ejecución, *reset*, y reloj del sistema, respectivamente.
- **exp_in[7:0]**: exponente del resultado preliminar en punto flotante.
- **man_in[31:0]**: mantisa del resultado preliminar en punto flotante.

Y retorna:

- **ready**: señal que indica la finalización de la normalización.
- **exp_out[7:0]**: exponente resultante.
- **man_out[31:0]**: mantisa resultante.

En la figura 4.40 se aprecia el diagrama de bloques detallado, el cual se basa en el el diagrama de flujo del normalizador de la unidad aritmética de punto flotante (figura 4.29), pero sin considerar el desplazamiento a la derecha. Por lo tanto, esta es una versión simplificada y suficiente para la conversión de entero a flotante.

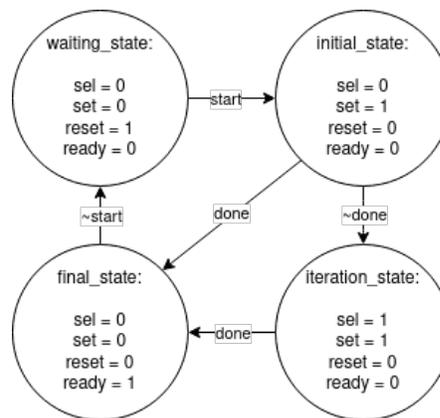


Figura 4.41: Diagrama *MDS* del submódulo *normalizer_integer2fp* del módulo *fp_converter*.

A partir del diagrama de bloques detallado de la figura 4.40 se elabora directamente el diagrama *MDS* (figura 4.41) de la máquina de estados del controlador. A continuación, se explica el rol de cada estado:

- **waiting_state**: mantiene el registro temporal en cero y espera la señal *start*, cuando esto sucede pasa al estado **initial_state**.
- **initial_state**: guarda el resultado inicial y pasa al estado **iteration_state** si aún no está normalizado (*done* = 0), en caso de estarlo (*done* = 1) pasa al estado **final_state**.
- **iteration_state**: en cada ciclo actualiza el nuevo resultado considerando el anterior, si ya está normalizado (*done* = 1) pasa al estado **final_state**.
- **final_state**: activa *ready* y vuelve a **waiting_state** cuando *start* se desactiva.

4.5.2.6 FP Converter Unit: **ROUNDER**

El módulo combinacional *rounder_integer2fp* se encarga de tomar el resultado del módulo *normalizer_integer2fp* para realizar el redondeo siguiendo el criterio de la tabla 4.7. Para esto, recibe las entradas:

- **sig_in**: signo del número en punto flotante a redondear.
- **rm[2:0]**: modo de redondeo seleccionado.
- **exp_in[7:0]**: exponente del número en punto flotante.
- **man_in[31:0]**: mantisa del número en punto flotante.

Y retorna:

- **still_norm**: bandera que indica que el número sigue normalizado después del redondeo.
- **NX**: bandera que indica que el resultado es inexacto.
- **exp_out[7:0]**: exponente del resultado normalizado.
- **man_out[31:0]**: mantisa del resultado normalizado.

El cálculo de los bits de redondeo se realiza de la siguiente forma: *g* (*guard*) y *r* (*round*) corresponden a los bits *man_in[7]* e *man_in[6]*, respectivamente, mientras que *s* (*sticky*) es igual a 1 solo si se cumple que *man_in[5 : 0]* \neq 0. Por otra parte, *NX* es 0 cuando *grs* = 000, en caso contrario es 1.

4.6 Core Complex

El *core complex* corresponde al módulo que contiene el procesador del *SoC*, este es de un solo núcleo (*core*), *single issue* y utiliza el juego de instrucciones *RV32IMF* (ver tablas 2.6, 2.7 y 2.9). Este módulo es el encargado de procesar los programas que se puedan cargar en el *SoC*.

Para este módulo se diseñan dos versiones, una *Single-Cycle* y otra *Pipelined*. Se comienza por la versión *Single-Cycle* (tomando como referencia el procesador del mismo tipo presentado en [12]) debido a su simplicidad. Esta característica ayuda en la etapa de verificación y depuración del *datapath*, donde se utilizan programas escritos en lenguaje ensamblador. Depurar el *datapath* de la versión *Single-Cycle* ahorra tiempo y trabajo en la etapa de verificación del diseño *Pipelined*, el cual se realiza considerando la versión *Single-Cycle* como base.

Ambas versiones del *core complex* hacen uso de todos los módulos ya presentados y reciben las siguientes entradas:

- **start, rst, clk**: señales de inicio de ejecución, *reset* y reloj del sistema, respectivamente.
- **initial_PC**: indica el *Program Counter* inicial para la ejecución.
- **aces_to_registers_files, aces_to_data_mem, aces_to_prog_mem**: señales de acceso para el *Register Files*, memoria de datos y de programa, respectivamente. Cuando alguna de estas señales es 1 y *start* = 0 permite el acceso a las entradas y salidas de su respectivo módulo.

- **do_wb_fromEEI, is_wb_data_fp_fromEEI, wb_add_fromEEI[4:0], wb_data_fromEEI[31:0]**: señales de acceso para el puerto de escritura de registros en el *Register Files*. Si *do_wb_fromEEI* (puerto *write_reg*) es 1, se guarda la entrada *wb_data_fromEEI* (puerto *wb_data*) en el registro ubicado en la dirección indicada por *wb_add_fromEEI* (puerto *wb_add*). La señal *is_wb_data_fp_fromEEI* (puerto *is_wb_data_fp*) indica que es un registro flotante cuando es igual a 1, en caso contrario el registro a modificar es entero. Estas entradas son válidas cuando *aces_to_registers_files* = 1 y *start* = 0.
- **is_rs1_fp_fromEEI, is_rs2_fp_fromEEI, rs1_add_fromEEI[4:0], rs2_add_fromEEI[4:0]**: señales de acceso a los puertos de lectura del *Register Files*. Se pueden acceder dos registros a la vez: *is_rs1_fp_fromEEI* (puerto *is_rs1_fp*) y *rs1_add_fromEEI* (puerto *rs1_add*) indican si el primer registro es flotante y su dirección, respectivamente, mientras que *is_rs2_fp_fromEEI* (puerto *is_rs2_fp*) y *rs2_add_fromEEI* (puerto *rs2_add*) indican lo mismo para el segundo registro. Solo son válidas cuando *aces_to_registers_files* = 1 y *start* = 0.
- **data_rw_fromEEI, data_valid_mem_fromEEI, data_is_load_unsigned_fromEEI, data_byte_half_word_fromEEI[1:0], data_addr_fromEEI[31:0], data_in_fromEEI[31:0]**: señales de acceso a los puertos de la memoria de datos, solo son válidas cuando *start* = 0 y *aces_to_data_mem* = 1. La entrada *data_rw_fromEEI* (puerto *rw*) indica que se desea realizar una escritura en memoria cuando es 1, en caso contrario se realiza una lectura, *data_valid_mem_fromEEI* (puerto *valid*) indica que se comience la ejecución de la lectura/escritura, *data_is_load_unsigned_fromEEI* (puerto *is_load_unsigned*) indica que la lectura es sin signo (aplica cuando no se leen palabras de 32 bits), *data_byte_half_word_fromEEI* (puerto *byte_half_word*) indica si se desea acceder a un *word*, *halfword* o *byte*, *data_addr_fromEEI* (puerto *addr*) indica la dirección de acceso y *data_in_fromEEI* (puerto *data_in*) es la información a escribir si es el caso.
- **prog_rw_fromEEI, prog_valid_mem_fromEEI, prog_is_load_unsigned_fromEEI, prog_byte_half_word_fromEEI[1:0], prog_addr_fromEEI[31:0], prog_in_fromEEI[31:0]**: funcionan exactamente igual que las señales del punto anterior pero para la memoria de programa y solo son válidas cuando *start* = 0 y *aces_to_prog_mem* = 1.

Y retornan las salidas:

- **ready**: señal que indica la finalización de ejecución.
- **exit_status[1:0]**: indica el estado de finalización, esto se detalla en la tabla 4.13.
- **PC[31:0]**: indica el *Program Counter* de la instrucción que termina la ejecución, no el de la siguiente instrucción.
- **rs1_toEEI[31:0], rs2_toEEI[31:0]**: registros accedidos por *rs1_add_fromEEI* y *rs2_add_fromEEI*, respectivamente.
- **data_ready_toEEI, prog_ready_toEEI**: señales que indican que el acceso a la memoria, de datos y programa, respectivamente, ha finalizado.
- **data_out_of_range_toEEI, prog_out_of_range_toEEI**: señales que indican que el acceso de la memoria, de datos y programa, respectivamente, esta fuera del rango válido para la *BRAM* implementada.
- **data_out_toEEI[31:0], prog_out_toEEI[31:0]**: salida de las lecturas de la memoria de datos y de programa, respectivamente.

Estas E/S son parte fundamental del *EEI* y permiten la comunicación entre el *core complex* y el entorno de ejecución. Nótese que la mayoría de las E/S son para tener acceso directo a las E/S del *Register Files* y de las memorias de datos y programa. Esto permite que el entorno de ejecución

en el cual se implemente el núcleo pueda cargar programas, leer/escribir direcciones de memoria e interpretar/ejecutar las llamadas de sistema del núcleo.

exit_status[1:0]	
Code	Status
00	ecall
01	program error
10	data error
11	ebreak

Tabla 4.13: Codificación de la salida *exit_status* del *core complex*. Nota: *program/data error* ocurren cuando se trata de acceder a una dirección de memoria fuera del rango de memoria de la *BRAM*. Además, solo en el caso *Single-Cycle* las instrucciones no válidas generan *program error*, mientras que en el caso *Pipelined* estas se tratan como *nops*.

La única diferencia entre cada *core complex* en cuanto a cómo finalizan su ejecución es que: la versión *Single-Cycle* finaliza con *program error* cuando hay una instrucción no válida, mientras que, la versión *Pipelined* simplemente las trata como *nops*.

Para la aplicación de la metodología de diseño *Top-Down*, se utiliza el mismo diagrama de bloques simplificado de la figura 4.2 en ambas versiones (para el diseño *Single-Cycle* simplemente se ignoran los registros *Pipeline*). A partir de este diagrama de bloques se continua (en la sección 4.6.1 para la versión *Single-Cycle* y en la sección 4.6.2 para la *Pipelined*) con la metodología descrita en la sección 3.1.

4.6.1 Single-Cycle Version

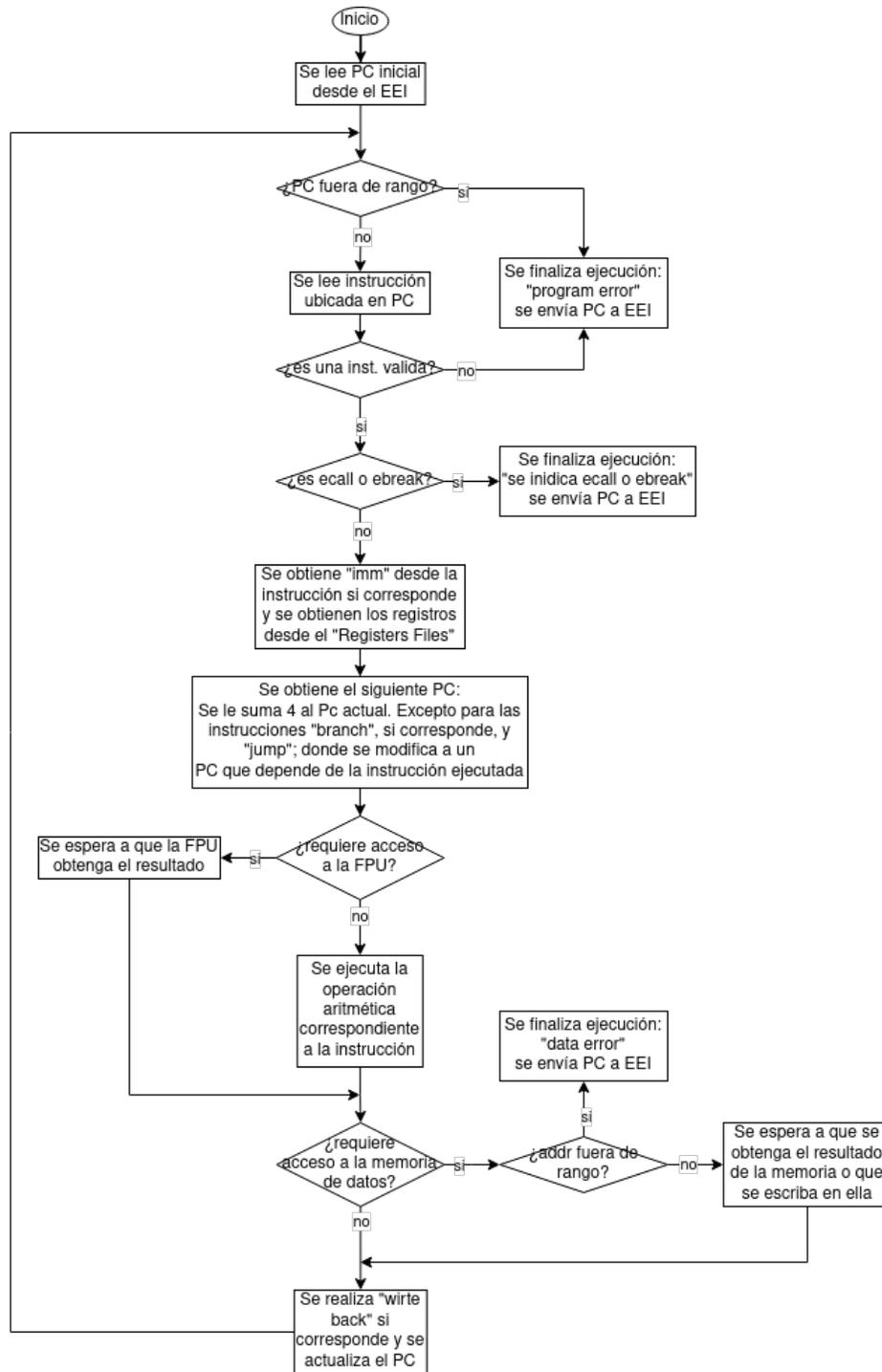


Figura 4.42: Diagrama de flujo simplificado para el *core complex Single-Cycle*.

Para la confección del diseño *Single-Cycle* se plantea el diagrama de flujo simplificado de la figura 4.42, a partir del mismo se obtiene el diagrama de bloques detallado, el cual se puede observar en las figuras 4.43, 4.44, 4.45 y 4.46.

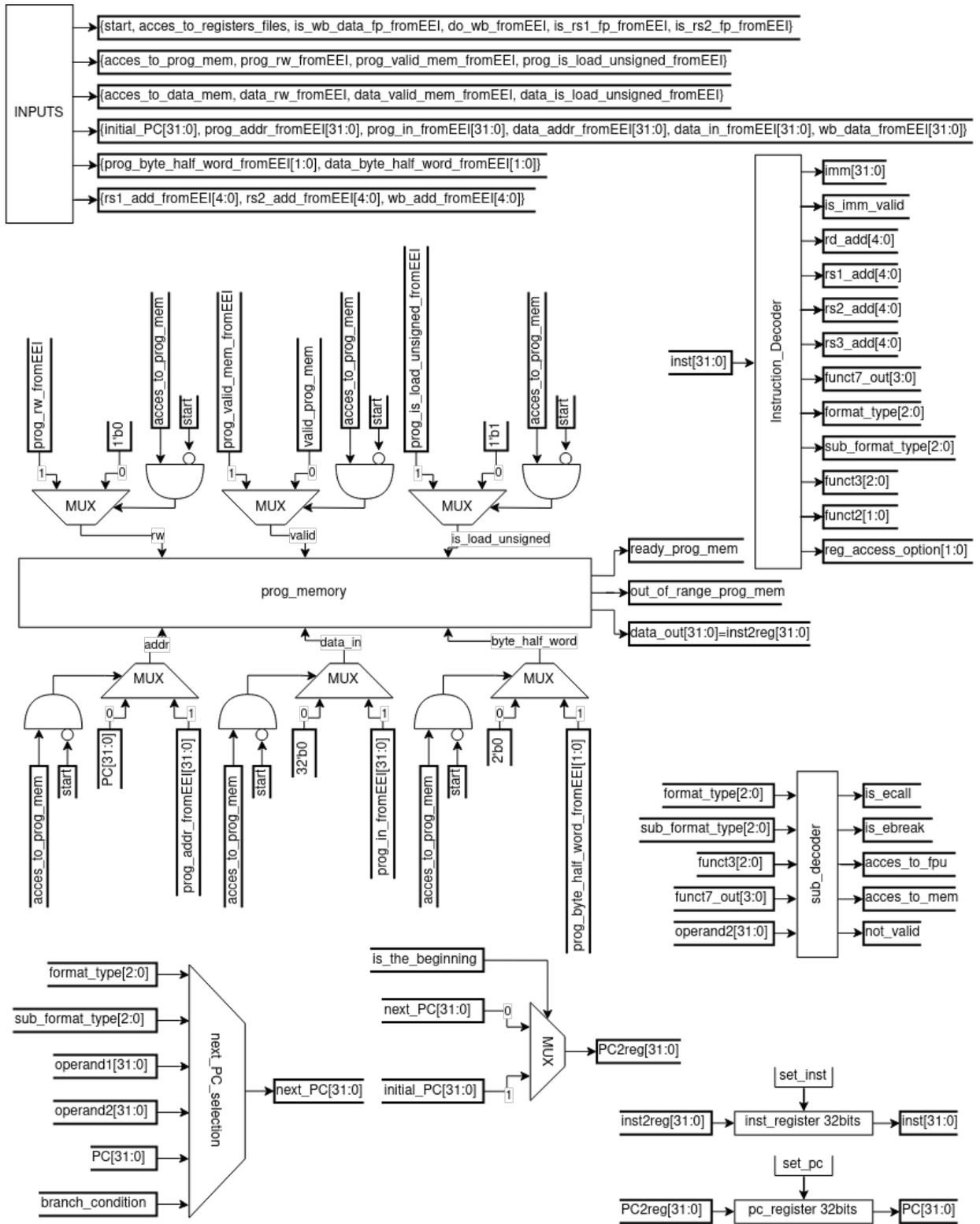


Figura 4.43: Diagrama de bloques detallado para el *core complex Single-Cycle* parte 1.

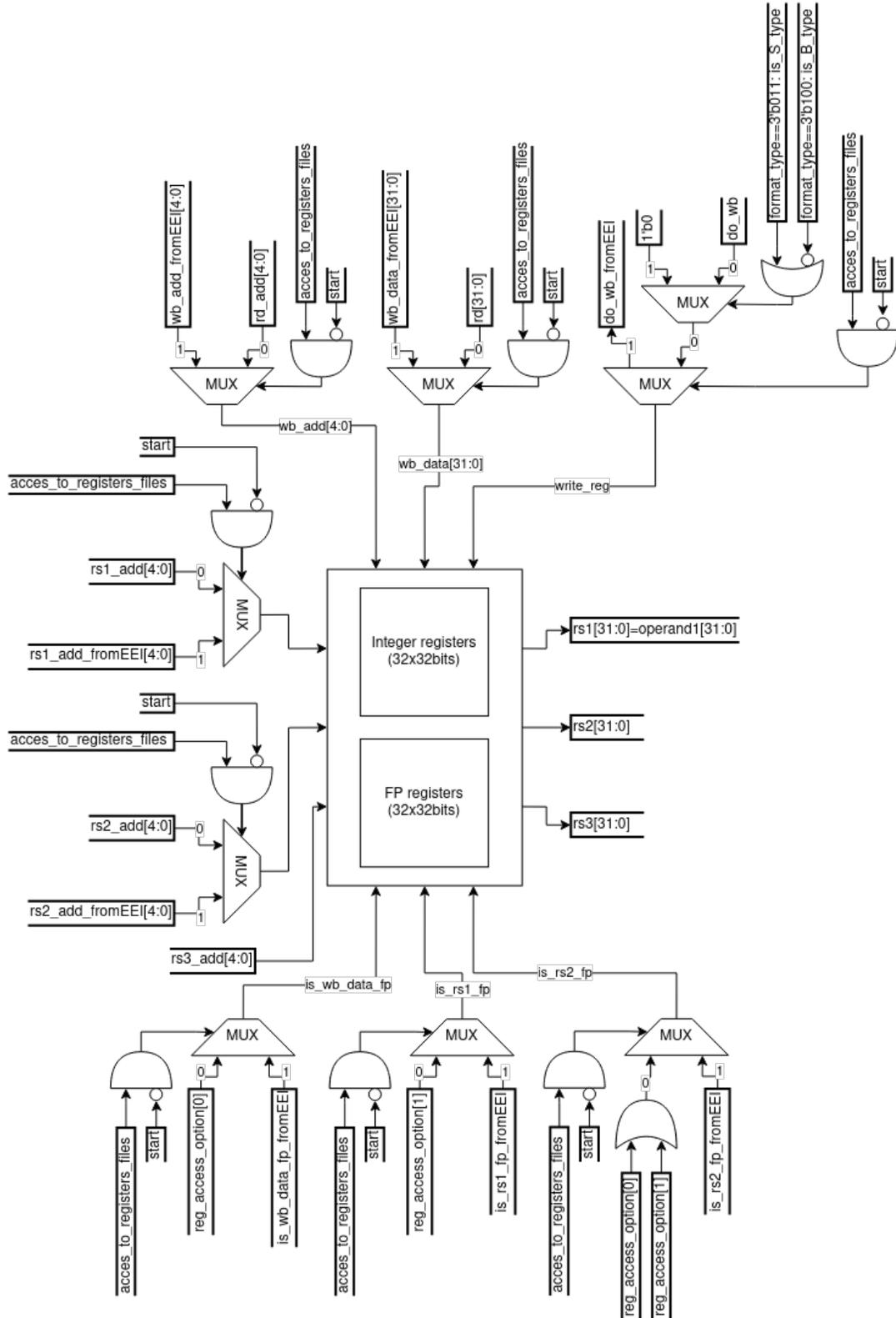


Figura 4.44: Diagrama de bloques detallado para el *core complex Single-Cycle* parte 2.

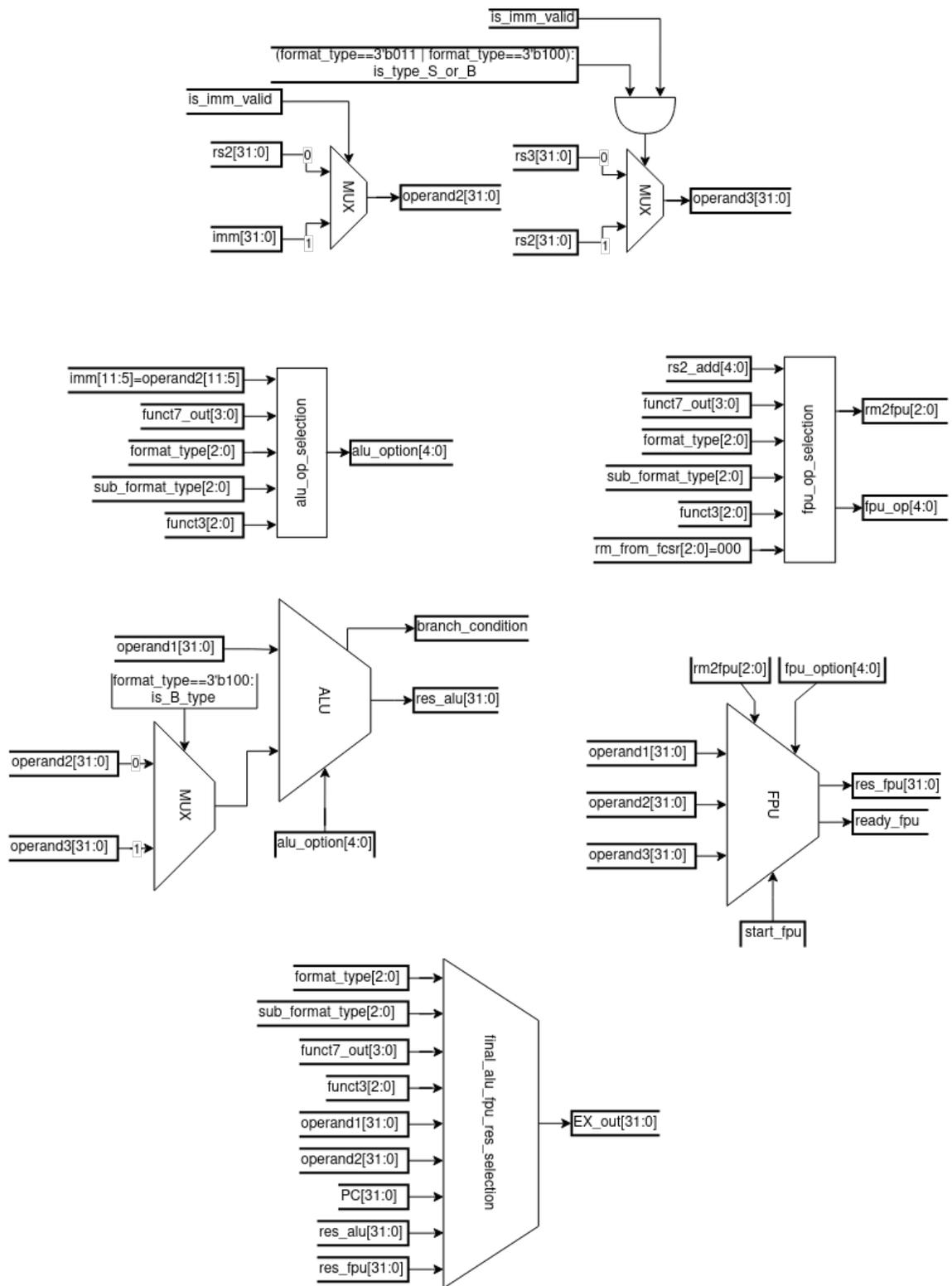


Figura 4.45: Diagrama de bloques detallado para el *core complex Single-Cycle* parte 3.

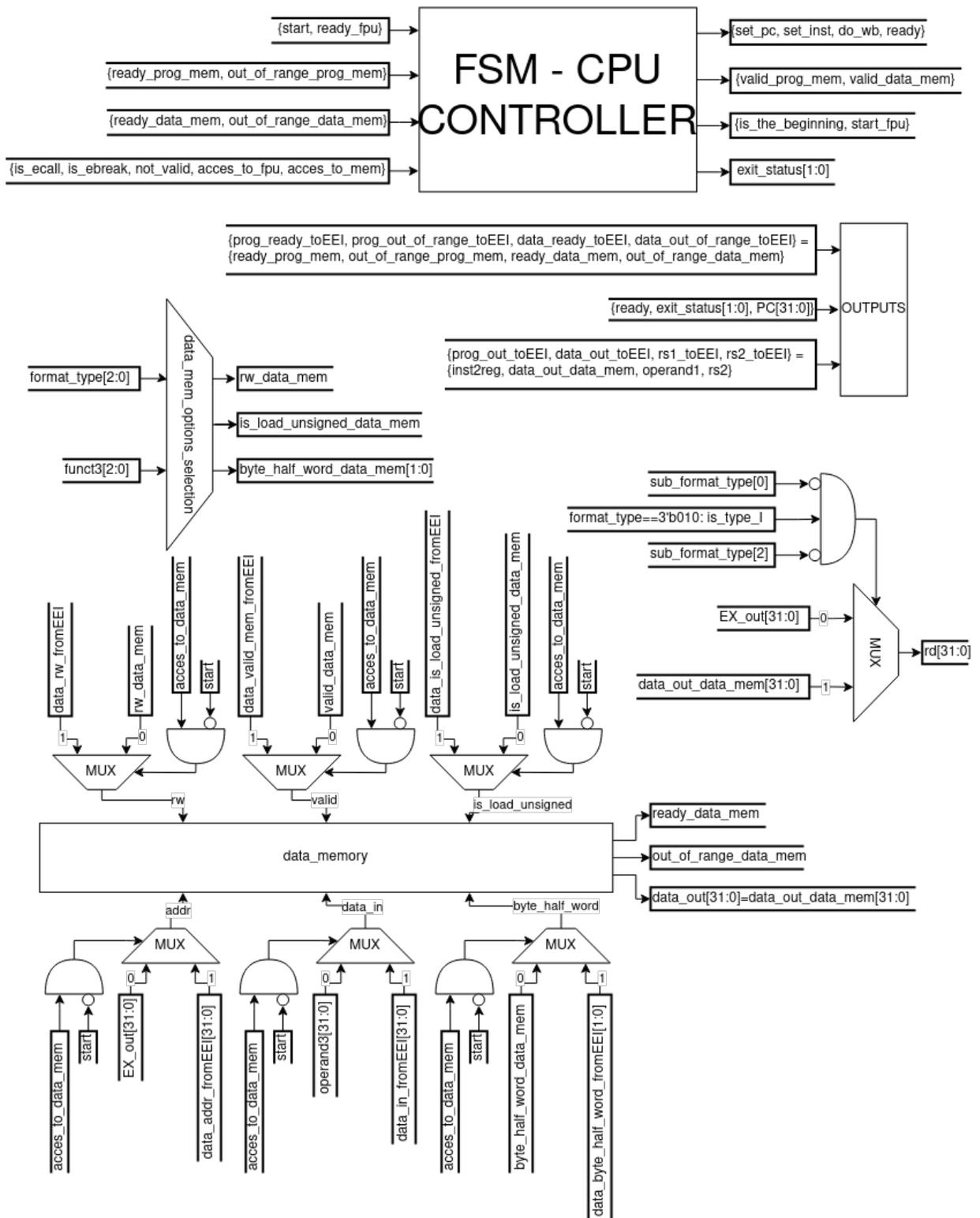


Figura 4.46: Diagrama de bloques detallado para el *core complex Single-Cycle* parte 4.

En las figuras 4.43, 4.44, 4.45 y 4.46 se aprecian las señales entre los distintos bloques funcionales del *datapath*, donde se puede ver la comunicación entre todos los módulos implementados. Sin embargo, hay pequeñas unidades combinatorias que no se han presentado, estas son:

- **pc_register, inst_register**: registros temporales para el *Program Counter* y la instrucción en curso, respectivamente. Ambos actualizan su valor en el flanco negativo del reloj, siempre y cuando reciban la señal *set_pc* y *set_inst*, respectivamente.
- **sub_decoder**: recibe como entradas las salidas de la unidad *Instruction Decoder*, en específico: *format_type*, *sub_format_type*, *funct7_out* y *funct3*. Con dicha información y *operand2* (el campo *imm*) retorna las señales: *is_ecall* para indicar que la instrucción en curso es un *ecall*, *is_ebreak* indica que es un *ebreak*, *acces_to_fpu* indica que la instrucción accede a la *FPU*, *acces_to_mem* para indicar que se accede a la memoria de datos y *not_valid* para indicar que la instrucción es no válida.
- **next_PC_selection**: recibe como entradas las señales: *operand1*, *operand2* y *PC* para el cálculo de nuevo *Program Counter* y *branch_condition*, *format_type* y *sub_format_type* para la selección del mismo. A partir de dichas entradas se obtiene el nuevo *Program Counter*, *next_PC*.
- **final_alu_fpu_res_selection**: esta unidad se encarga de seleccionar el resultado final de la etapa de ejecución. Para esta tarea recibe las señales de decisión: *format_type*, *sub_format_type*, *funct7_out* y *funct3*. Luego, para el cálculo se recibe: *operand1*, *operand2*, *PC*, *res_alu* y *res_fpu*. El resultado de la unidad es *EX_out*.
- **data_mem_options_selection**: recibe las señales *format_type* y *funct3*, con esto retorna las opciones de acceso a la memoria de datos: *rw_data_mem*, *is_load_unsigned_data_mem* y *byte_half_word_data_mem*.

Finalmente, a partir del diagrama de bloques detallado se confecciona directamente el diagrama *MDS* de la figura 4.47. A continuación, se presenta el rol de cada uno de los estados de la máquina de estados, los cuales se actualizan en el flanco positivo del reloj:

- **waiting_state**: espera la señal *start*, cuando esto sucede pasa al estado **initial_state**.
- **initial_state**: guarda el *Program Counter* inicial en el registro temporal y pasa al estado **program_loop_read_inst**.
- **program_loop_read_inst**: desactiva el acceso a la *FPU* o la memoria de datos si corresponde. Luego, lee y guarda la instrucción de la memoria de programa indicada por el *Program Counter*. Si el acceso a la memoria de programa esta fuera de rango (*out_of_range_prog_mem* = 1) se pasa al estado **final_state**, si no es el caso, espera a que la instrucción esté lista (*ready_prog_mem* = 1) y el siguiente estado es **program_loop_main**.
- **program_loop_main**: este analiza las señales de control provenientes del *sub_decoder*. Si alguna de las señales *is_ecall*, *is_ebreak* y *not_valid* está activada pasa al estado **final_state**. Si no es el caso, el siguiente estado depende las señales *acces_to_fpu* y *acces_to_mem*, si se activa la primera se pasa a **program_loop_fpu**, en caso de activarse la segunda señal se pasa a **program_loop_mem** y si ninguna de ellas se activa, se pasa a **program_loop_new_pc_wb**.
- **program_loop_fpu**: inicia la ejecución de la *FPU* y espera a que esté listo (*ready_fpu* = 1), cuando esto sucede pasa al estado **program_loop_new_pc_wb**.
- **program_loop_mem**: si detecta que la dirección de acceso a memoria esta fuera de rango (*out_of_range_data_mem* = 1) pasa directamente al estado **final_state**. Si no es el caso, se inicia el acceso a la memoria de datos y se espera a que esté listo (*ready_data_mem* = 1) para pasar al estado **program_loop_new_pc_wb**.
- **program_loop_new_pc_wb**: realiza el *Write Back* si corresponde y actualiza el *Program Counter*, luego pasa al estado **program_loop_read_inst**.

- **final_state**: estado de finalización de ejecución, activa *ready* y selecciona el *exit_status* pertinente, ver tabla 4.13. Cambia de estado, a **waiting_state** cuando *start* se desactiva.

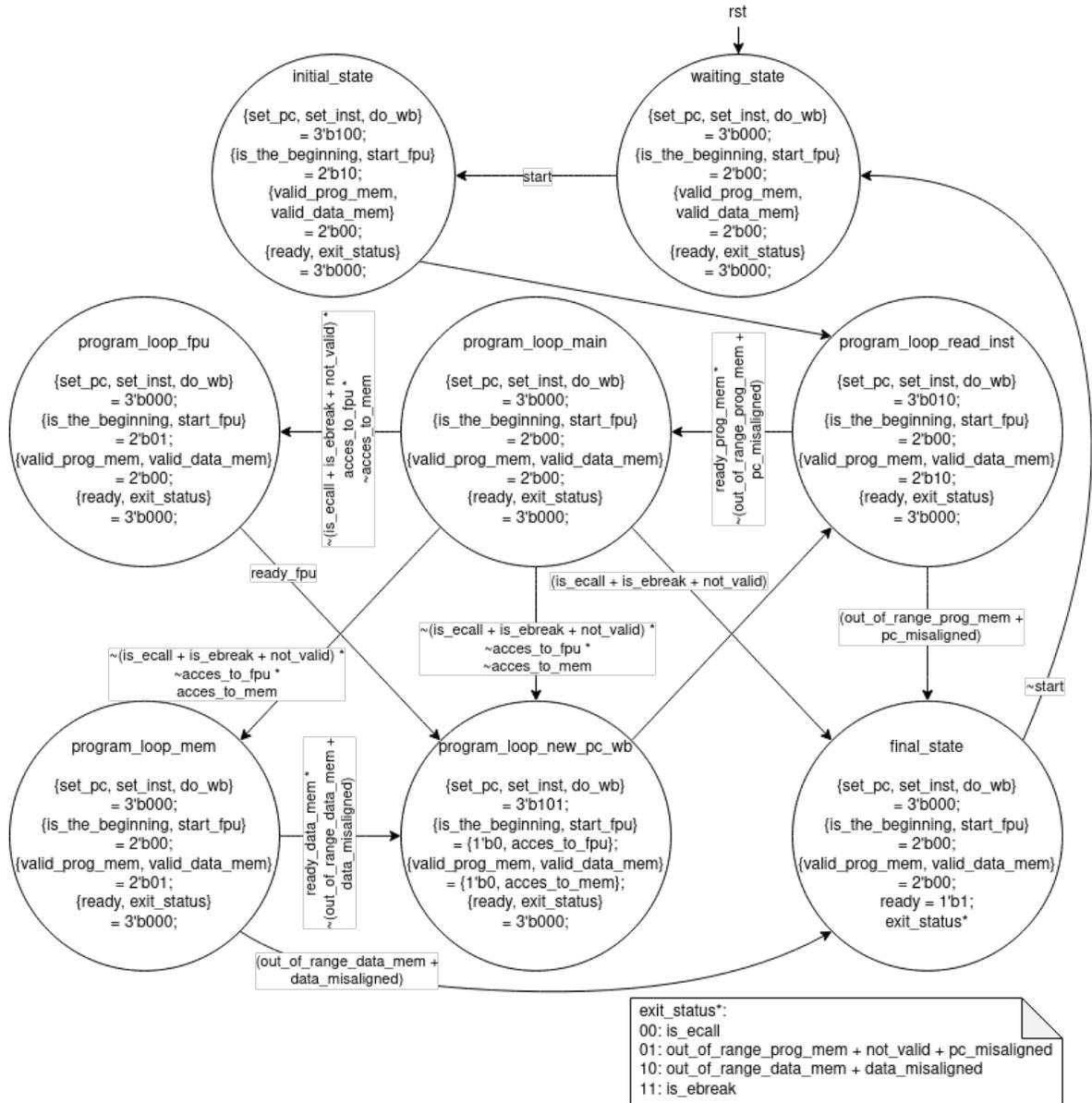


Figura 4.47: Diagrama MDS para el core complex Single-Cycle.

4.6.2 Pipelined Version

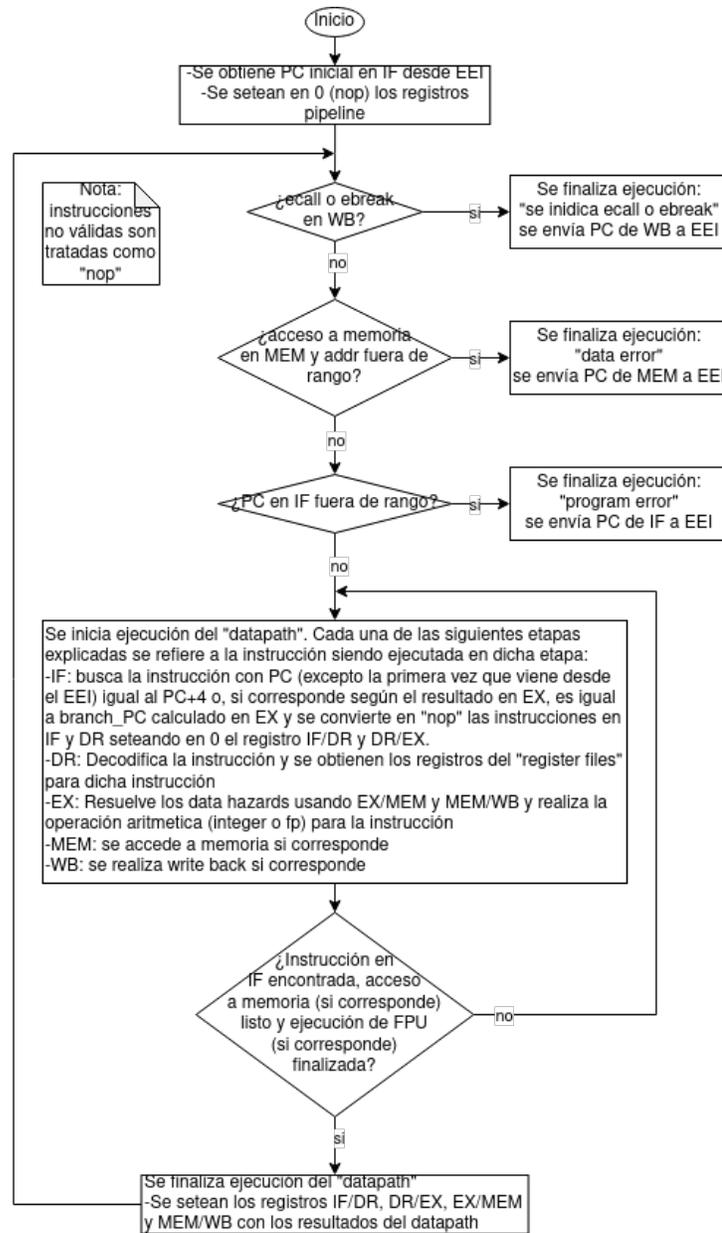


Figura 4.48: Diagrama de flujo simplificado para el *core complex Pipelined*.

Para la confección del diseño *Pipelined* se plantea el diagrama de flujo simplificado de la figura 4.42, a partir del mismo se obtiene el diagrama de bloques detallado, el cual se puede ver en las figuras 4.49, 4.50, 4.51, 4.52, 4.53 y 4.54.

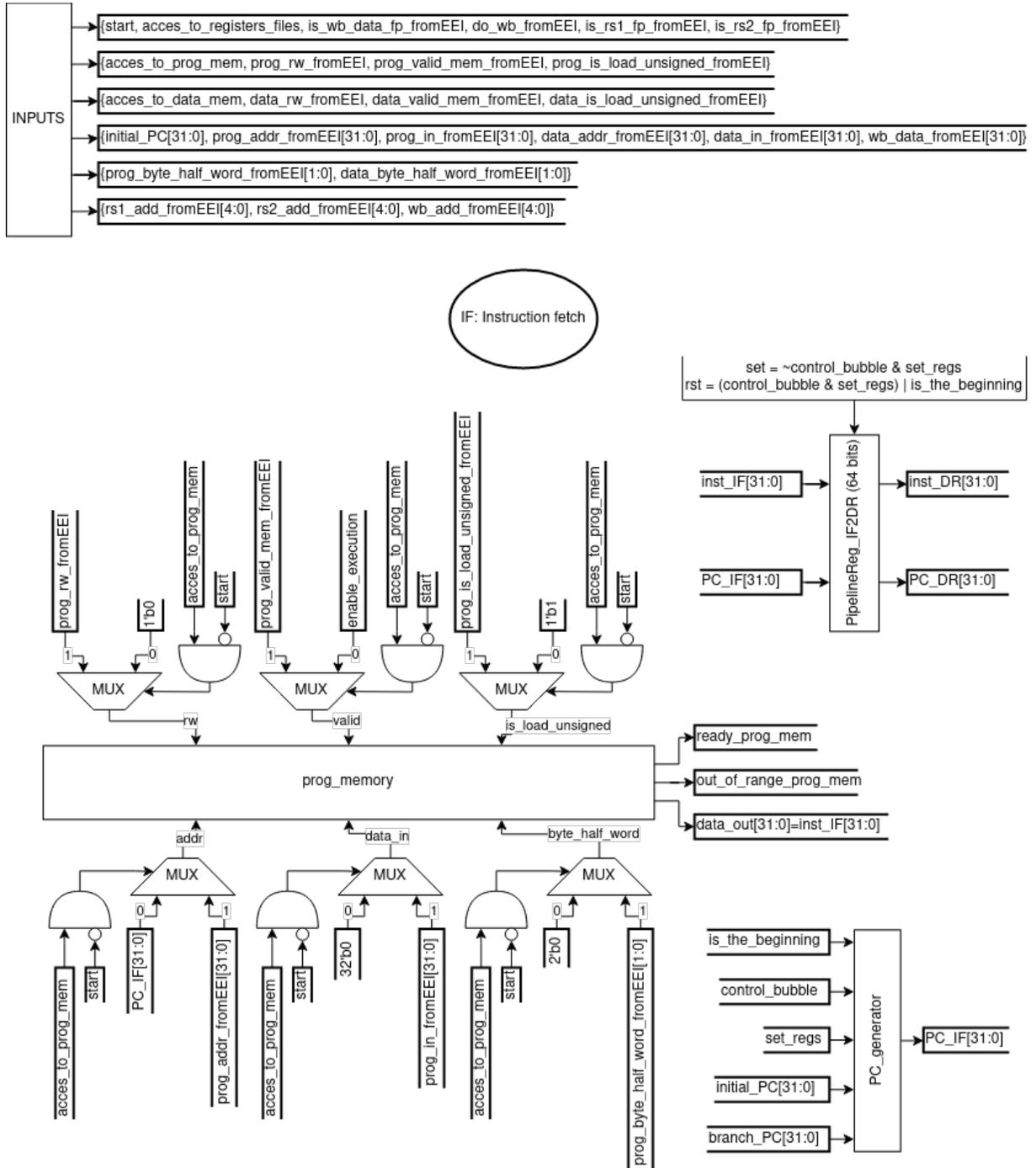


Figura 4.49: Diagrama de bloques detallado para el *core complex Pipelined* parte 1.

DR: Instruction decode

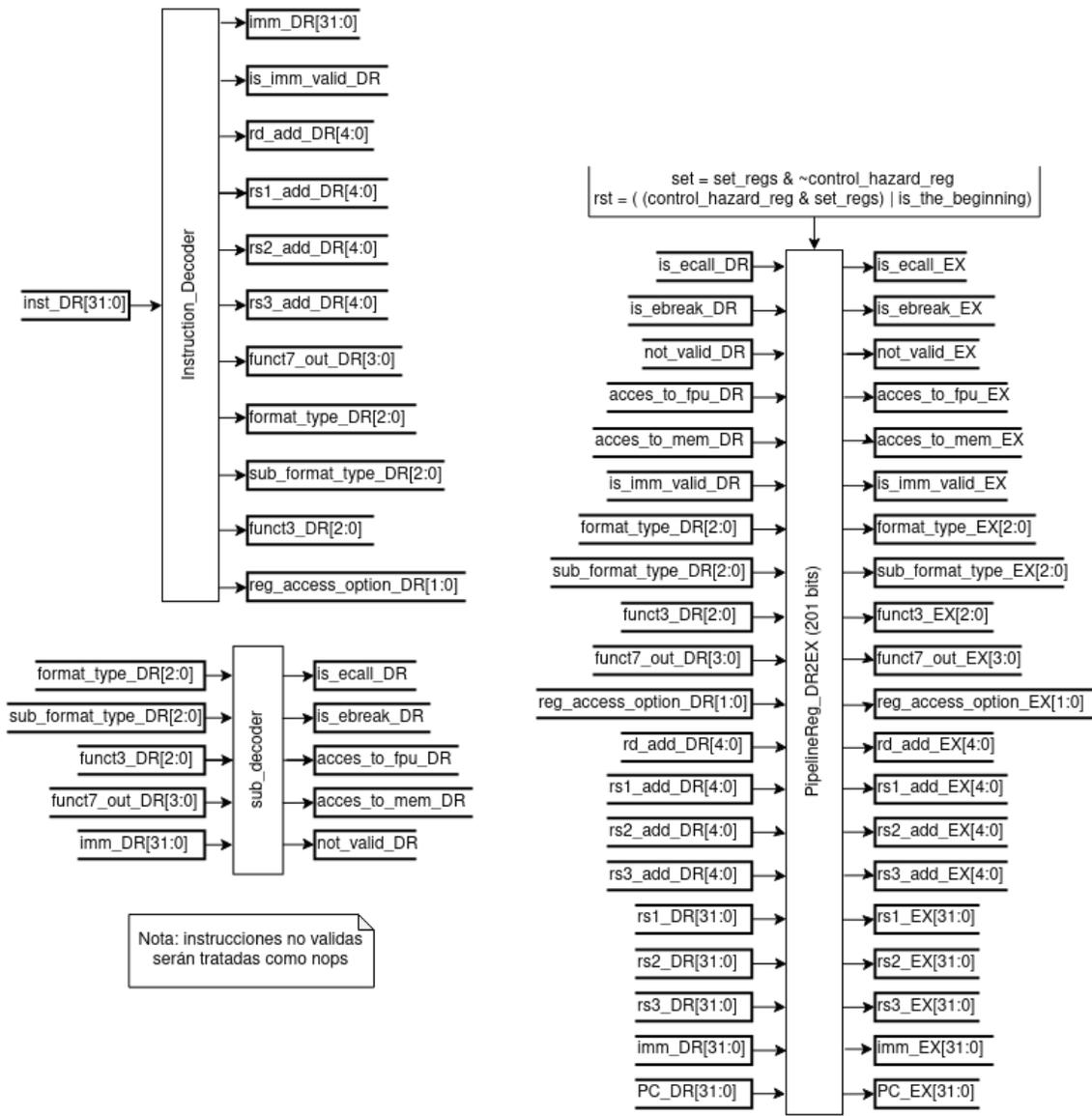


Figura 4.50: Diagrama de bloques detallado para el *core complex Pipelined* parte 2.

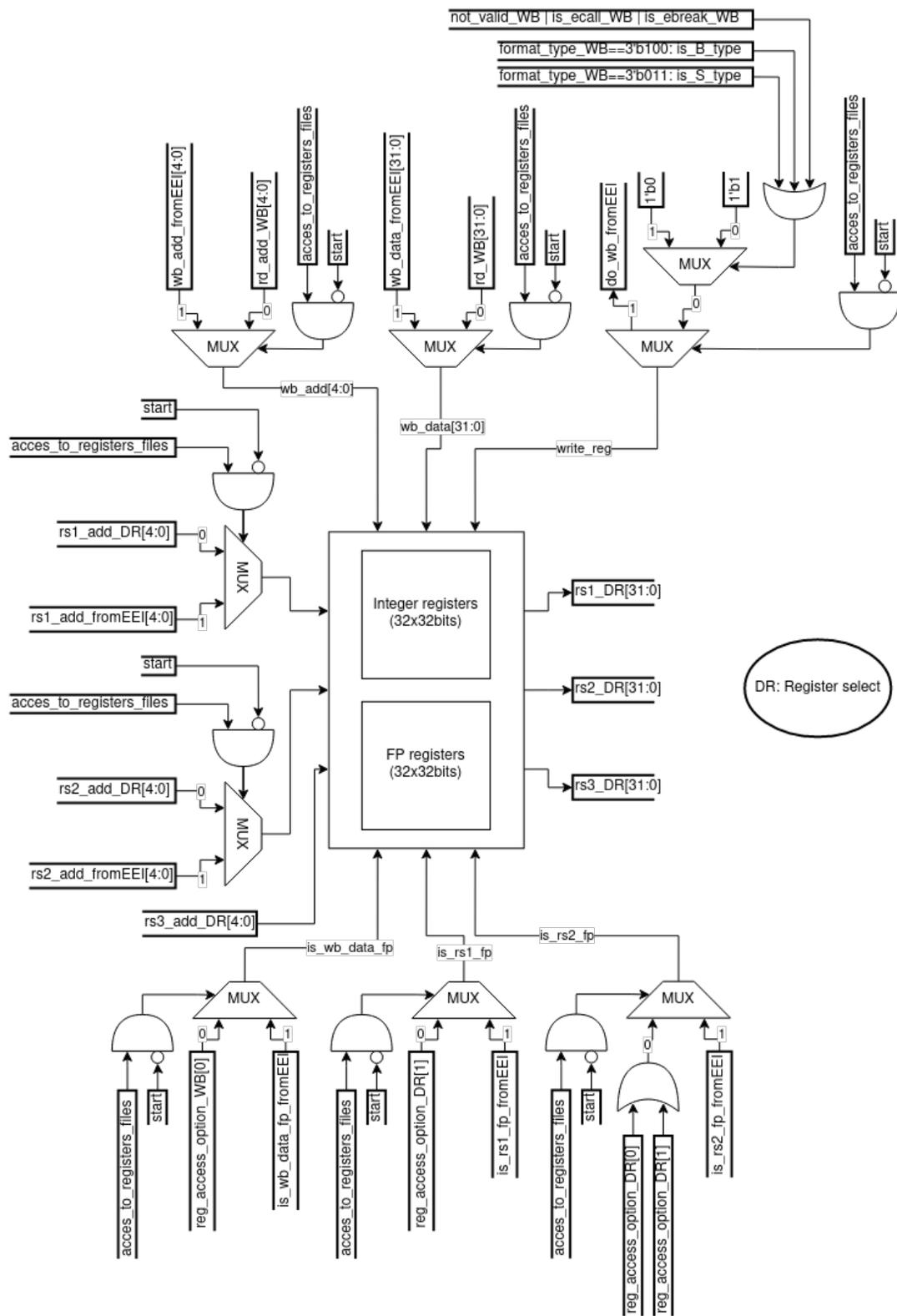


Figura 4.51: Diagrama de bloques detallado para el *core complex Pipelined* parte 3.

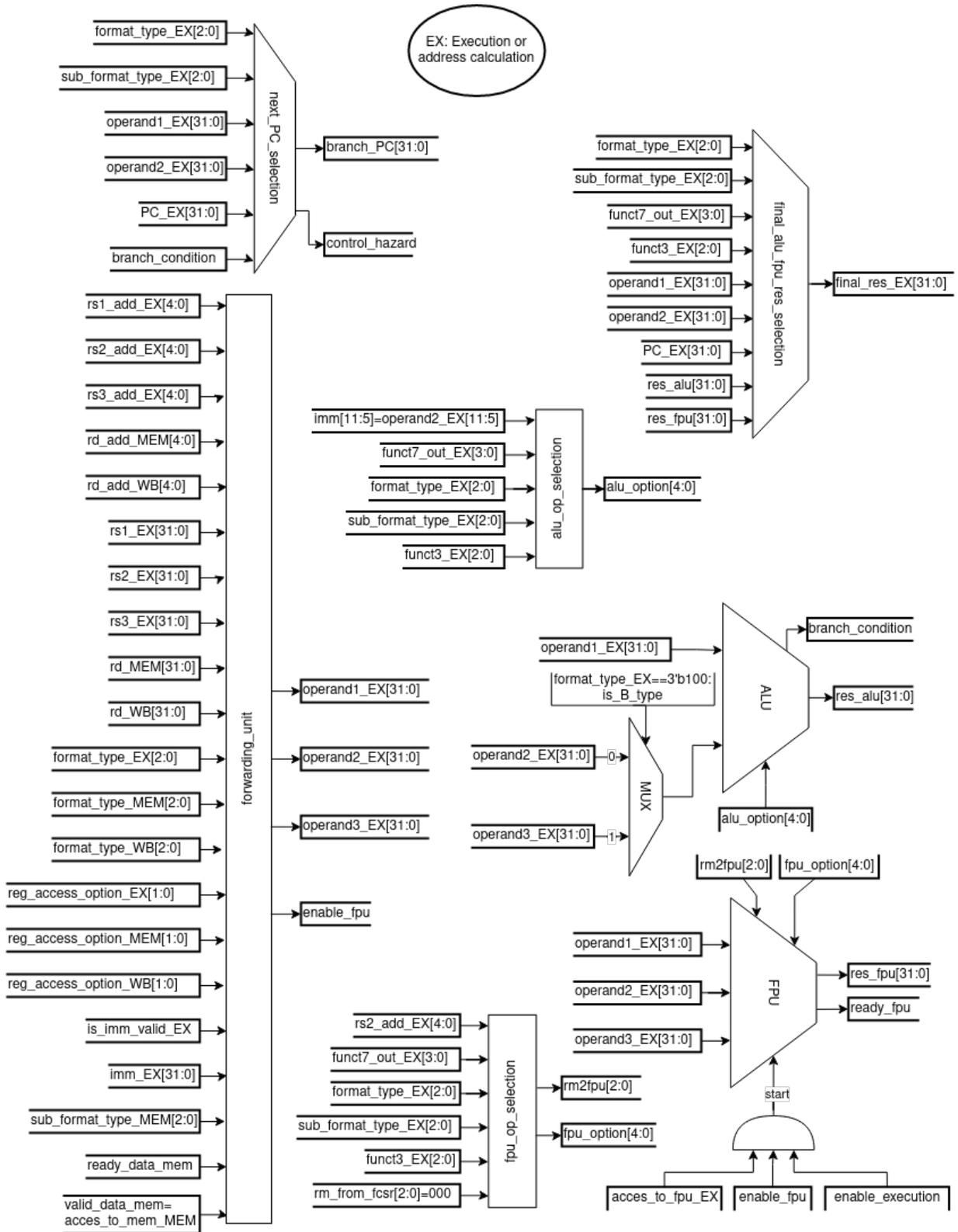


Figura 4.52: Diagrama de bloques detallado para el core complex Pipelined parte 4.

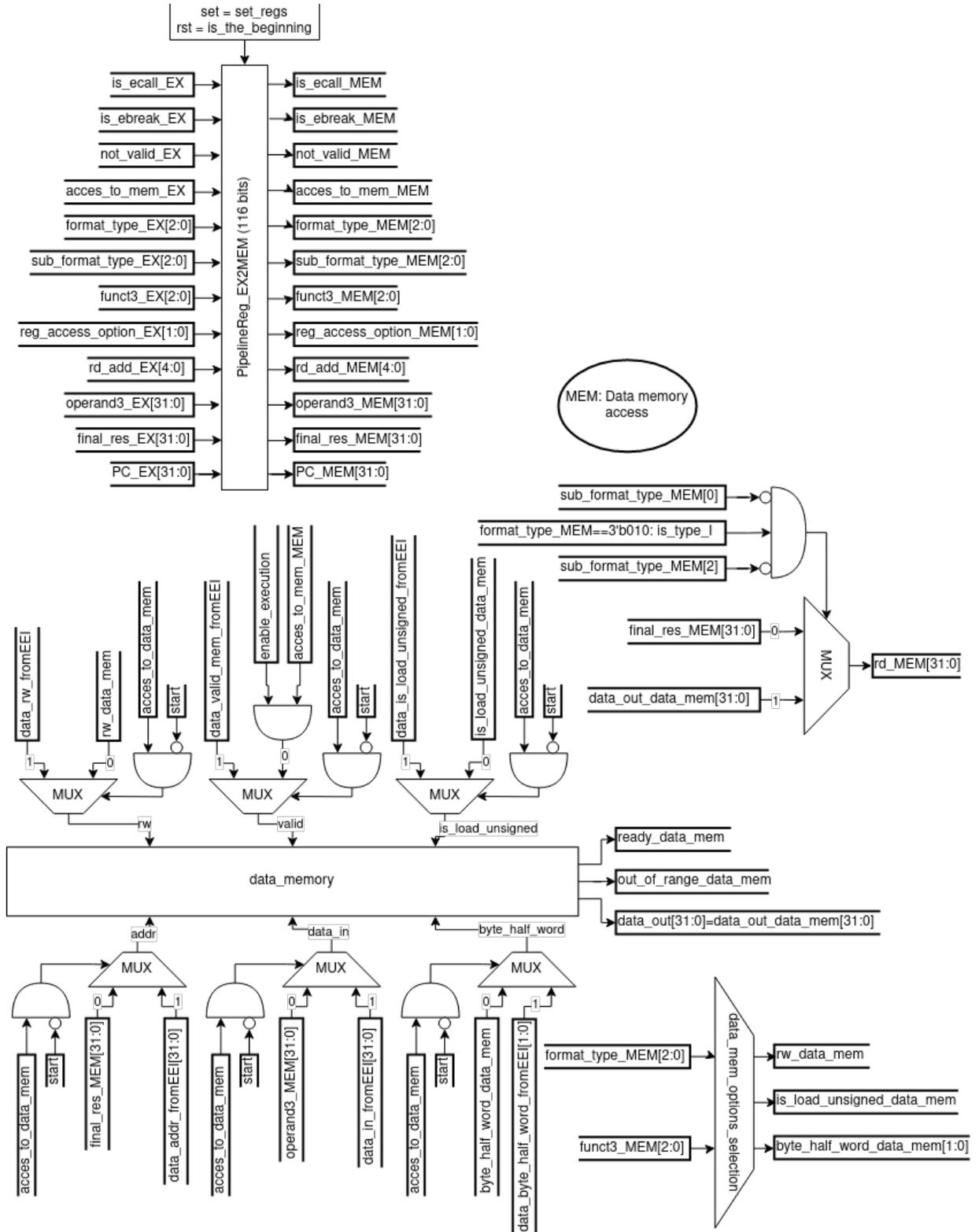


Figura 4.53: Diagrama de bloques detallado para el *core complex Pipelined* parte 5.

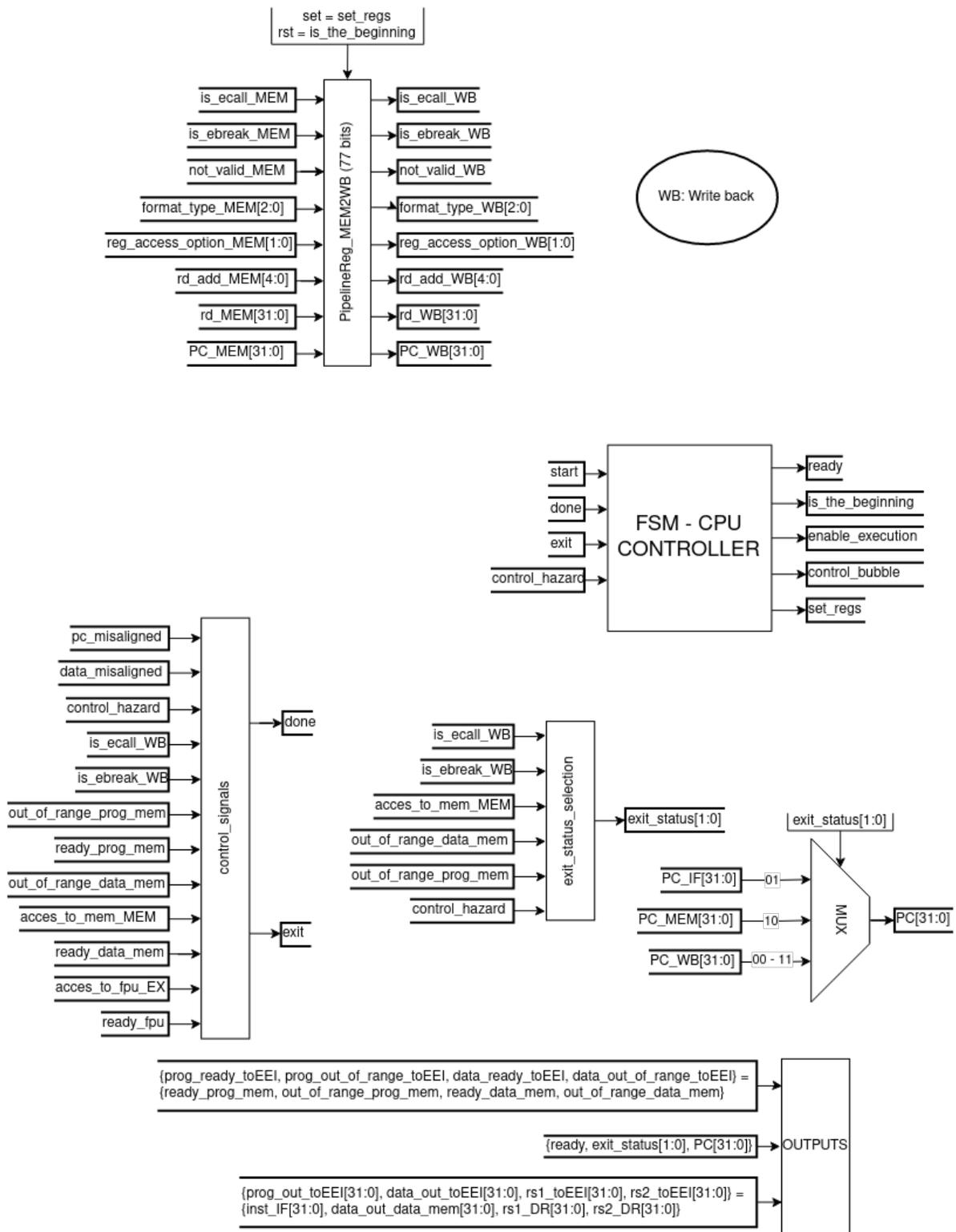


Figura 4.54: Diagrama de bloques detallado para el *core complex Pipelined* parte 6.

Como se puede apreciar en el diagrama de bloques detallado, se implementan 4 registros *pipeline*: *PipelineReg_IF2DR*, *PipelineReg_DR2EX*, *PipelineReg_EX2MEM* y *PipelineReg_MEM2WB*. En un principio se diseñan como 4 registros independientes (cada uno con su propio bloque *always*) con actualización en el flanco negativo del reloj siempre que *set* = 1 (de esta forma se han tratado todos los demás registros). Sin bien esto funciona en las simulaciones, en el *hardware* presenta problemas de metaestabilidad. Finalmente, se implementan todos los registros en un único bloque *always* (actualizado en el flanco positivo del reloj) con asignaciones *blocking*, esto con el fin de asegurar que se actualicen en el siguiente orden: *PipelineReg_MEM2WB* → *PipelineReg_EX2MEM* → *PipelineReg_DR2EX* → *PipelineReg_IF2DR*. En el diagrama de bloques detallado se pueden ver las señales que rigen la actualización de estos registros (*set/reset*).

En las figuras 4.49, 4.50, 4.51, 4.52, 4.53 y 4.54 se aprecian las señales entre los distintos bloques funcionales del *datapath*, donde se puede ver la comunicación entre todos los módulos implementados. Sin embargo, hay pequeñas unidades que no se han presentado, estas son:

- **PC_generator**: unidad secuencial que se activa en el flanco positivo del reloj y se actualiza antes que los registros *pipeline*. Si *is_the_beginning* = 1 se actualiza: $new_PC = initial_PC$, de no ser el caso, el nuevo *Program Counter* depende de las señales *set_regs* y *control_bubble*. Si *set_regs* = 0 se cumple: $new_PC = PF_IF$ (no se actualiza el *Program Counter* de la etapa *IF*), de ser 1, y *control_bubble* = 1 se tiene que: $new_PC = branch_PC$ y si *control_bubble* = 0, se tiene: $new_PC = PF_IF + 4$. Una vez obtenido *new_PC* se actualizan todos los registros *pipeline*, luego se actualiza $PC_IF = new_PC$ solo si $is_the_beginning \mid set_regs = 1$.
- **sub_decoder**: es combinacional y recibe como entradas las salidas de la unidad *Instruction_Decoder*, estas son: *format_type_DR*, *sub_format_type_DR*, *funct7_out_DR*, *funct3_DR* e *imm_DR*. Con dicha información retorna las señales: *is_ecall_DR* para indicar que la instrucción en curso (en la etapa *DR*) es un *ecall*, *is_ebreak_DR* indica que es un *ebreak*, *aces_to_fpu_DR* indica que la instrucción accede a la *FPU*, *aces_to_mem_DR* para indicar que se accede a la memoria de datos y *not_valid_DR* para indicar que la instrucción es no válida.
- **forwarding_unit**: este módulo combinacional se encarga de solucionar los *data hazard* que se presentan en la arquitectura *Pipelined*. Para esta tarea recibe: *is_imm_valid_EX*, *ready_data_mem*, *valid_data_mem*, *reg_access_option_EX*, *reg_access_option_MEM*, *reg_access_option_WB*, *format_type_EX*, *format_type_MEM*, *format_type_WB*, *sub_format_type_MEM*, *rs1_add_EX*, *rs2_add_EX*, *rs3_add_EX*, *rd_add_MEM*, *rd_add_WB*, *rs1_EX*, *rs2_EX*, *rs3_EX*, *rd_MEM*, *rd_WB* e *imm_EX*. Con la información recibida el módulo es capaz de determinar los operandos, *operand1_EX*, *operand2_EX* y *operand3_EX*, para la etapa *EX* cuando ocurre algún *data hazard* redirigiendo los resultados almacenados en el registro *pipeline* de la etapa *MEM* o *WB* si es necesario. Además, si ocurre un *data hazard* en una operación que accede a la *FPU* y tiene dependencia del resultado de la instrucción que se está ejecutando en la etapa *MEM* y esta última accede a memoria, la unidad mantiene desactivado el acceso a la *FPU* hasta que el acceso a memoria esté listo mediante la señal *enable_fpu*.
- **next_PC_selection**: unidad combinacional que recibe como entradas las señales: *operand1_EX*, *operand2_EX* y *PC_EX* para el cálculo de nuevo *Program Counter* en caso de una bifurcación y las señales *branch_condition*, *format_type_EX* y *sub_format_type_EX* para la selección del mismo. A partir de dichas entradas se obtiene el nuevo *Program Counter* en caso de una bifurcación válida, *branch_PC*. También, activa la señal *control_hazard* para indicar que la bifurcación debe ser tomada si es el caso.
- **final_alu_fpu_res_selection**: esta unidad se encarga de seleccionar combinacionalmente el resultado final de la etapa de ejecución (*EX*). Para esta tarea recibe las señales de decisión:

format_type_EX, *sub_format_type_EX*, *funct7_out_EX* y *funct3_EX*. Luego, para la selección se recibe: *operand1_EX*, *operand2_EX*, *PC_EX*, *res_alu* y *res_fpu*. El resultado de la unidad es *final_res_EX*.

- **data_mem_options_selection:** recibe las señales *format_type_MEM* y *funct3_MEM*, con esto retorna las opciones de acceso a la memoria de datos: *rw_data_mem*, *is_load_unsigned_data_mem* y *byte_half_word_data_mem*. Esta unidad es combinacional.
- **control_signals:** unidad combinacional encargada de recibir las señales: *is_ecall_WB*, *is_ebreak_WB*, *control_hazard*, *out_of_range_prog_mem*, *out_of_range_data_mem*, *aces_to_mem_MEM*, *aces_to_fpu_EX*, *ready_prog_mem*, *ready_data_mem* y *ready_fpu*. Con estas señales la unidad retorna: *exit* para indicar a la máquina de estados el fin de la ejecución del programa en curso y *done* para indicar que se finaliza la ejecución de todas las etapas en curso.
- **exit_status_selection:** esta unidad combinacional determina la salida *exit_status* según lo indicado en la tabla 4.13. Para esta tarea recibe las señales: *is_ecall_WB*, *is_ebreak_WB*, *aces_to_mem_MEM*, *out_of_range_data_mem*, *out_of_range_prog_mem* y *control_hazard*.

Finalmente, a partir del diagrama de bloques detallado se confecciona directamente el diagrama *MDS* de la figura 4.55. A continuación, se presenta el rol de cada uno de los estados de la *FSM*, los cuales se actualizan en el flanco positivo del reloj:

- **waiting_state:** espera la señal *start*, cuando esto sucede guarda el *Program Counter* inicial, reinicia los registros *pipeline* en cero (*is_the_beginning = start*) y pasa al estado **loop_state_1**.
- **loop_state_1:** inicia la ejecución del *datapath pipelined* (*enable_execution = 1*). Si *exit = 1* se pasa directamente al estado **final_state**, si no es el caso se espera a que el *datapath* finalice la ejecución (*done = 1*). Luego, si *control_hazard = 1* ocurre un *control hazard* y se pasa al estado **loop_state_2_chazzard**, si no es el caso se pasa a **loop_state_2_normal**.
- **loop_state_2_normal:** mantiene *enable_execution = 1* y actualiza los registros *pipeline* (*set_regs = 1*). El siguiente estado es **loop_state_3**.
- **loop_state_2_chazzard:** mantiene *enable_execution = 1* y actualiza los registros *pipeline* (*set_regs = 1*) considerando el *control hazard* (*control_bubble = 1*). El siguiente estado es **loop_state_3**.
- **loop_state_3:** desactiva *control_bubble* si corresponde y finaliza la ejecución del *datapath* (*enable_execution = 0*). Pasa al estado **loop_state_1**.
- **final_state:** mantiene *enable_execution = 1* y levanta la señal *ready*. Vuelve al estado **waiting_state** solo cuando *start* se desactiva.

Es importante destacar que el estado se actualiza solo después de que se actualizan los registros *pipeline* y *PC_IF*. Esto es fundamental para el correcto funcionamiento del diseño en *hardware*. Por esta razón se implementa en el mismo bloque *always* el *PC_generator*, los registros *pipeline* y la actualización del siguiente estado.

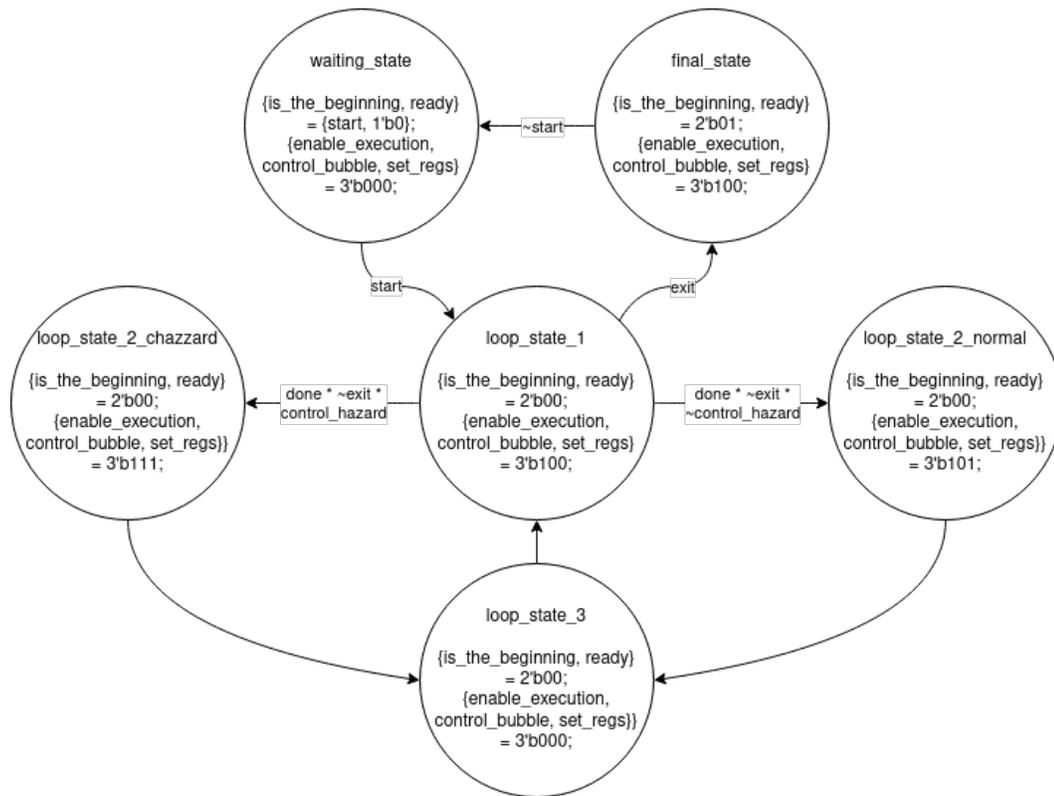


Figura 4.55: Diagrama *MDS* para el *core complex Pipelined*.

4.7 System on a Chip

Finalmente, se diseña el *SoC*, el cual resulta ser bastante simple debido al tiempo requerido para la implementación del *core complex*. Por ello, se consideran los siguientes dispositivos de E/S:

- **Entradas:**

- **3 pulsadores:** *start* (inicia la ejecución), *resume* (reanuda la ejecución) y *reset* (reinicia la ejecución).
- **16 interruptores:** se utilizan para recibir palabras de 32 bits (se utiliza *start* y *resume* para ingresar ambas mitades de la palabra).

- **Salidas:**

- **2 LED tricolor:** muestran el estado de la ejecución. Ver tabla 4.14.
- **16 LED verdes:** sirven como retroalimentación de la palabra ingresada con los interruptores (solo se encienden cuando se esta recibiendo una entrada en una llamada de sistema “*get*”).
- **2 Displays de 7 segmentos:** muestran palabras de 32 bits en formato hexadecimal. Las letras se distinguen de los número porque se prende el punto que acompaña al dígito.

Colors Codification	
Color	SoC state
BLUE	waiting state
CYAN	cpu working
WHITE	ebreak
RED	ecall default
RED/BLUE	prog error
BLUE/RED	data error
PURPLE	print/get* integer
YELLOW	print/get* floating point
GREEN	final state

Tabla 4.14: Cuando solo se menciona un color quiere decir que ambos *LED* tricolor tienen el mismo color, en caso contrario el *LED* derecho se corresponde con el color de la derecha y el izquierdo con el color de la izquierda. Cuando ocurre un *get** solo uno de los *LED* se enciende, según la mitad de la palabra que se esté ingresando.

Teniendo esto en cuenta se establecen 4 llamadas de sistema, estas se realizan cuando el *core complex* finaliza su ejecución con un *ecall*. A continuación, se detalla cada llamada de sistema posible:

- **Print Integer:** cuando $x17 = 1$ se muestra, en formato hexadecimal, la palabra de 32 bits contenida en el registro entero $x10$.
- **Print Float:** cuando $x17 = 2$ se muestra, en formato hexadecimal, la palabra de 32 bits contenida en el registro flotante $f10$.
- **Get Integer:** cuando $x17 = 5$ se recibe, en formato binario, una palabra de 32 bits y la guarda en el registro entero $x10$.
- **Get Float:** cuando $x17 = 6$ se muestra, en formato binario, una palabra de 32 bits y la guarda en el registro flotante $f10$.
- **Exit:** cuando $x17 = 10$ se finaliza el programa.

Si ocurre un *ecall* pero $x17$ no contiene un valor válido, es tratado como un *ebreak*, es decir, simplemente muestra, en formato hexadecimal, el *Program Counter* de la siguiente instrucción. Por otra parte, es importante mencionar que la convención para las llamadas de sistema recién presentada concuerda con las llamadas de sistema *Print Integer*, *Print Float*, *Get Integer*, *Get Float* y *Exit* de *RARS*.

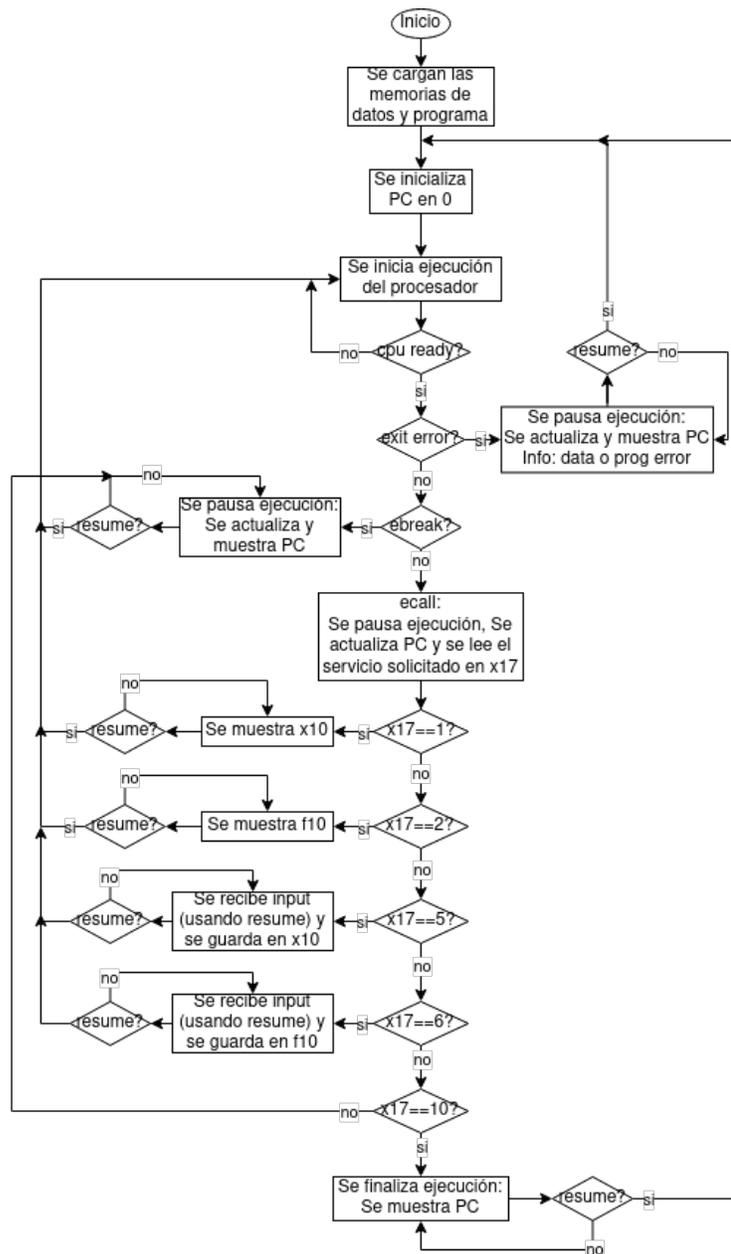


Figura 4.56: Diagrama de flujo simplificado para el SoC.

Considerando los llamados de sistema y las E/S presentadas, se confecciona el diagrama de flujo simplificado para el SoC; este se puede ver en la figura 4.56. Como se aprecia en el diagrama, para el manejo de errores simplemente se finaliza la ejecución del código y se muestra el tipo de error. Por otro lado, este diagrama contempla una fase de carga de programa, un *BOOT-LOADER*, sin embargo, por tiempo y simplicidad, se cargan los programas directamente en las memorias del *core complex*.

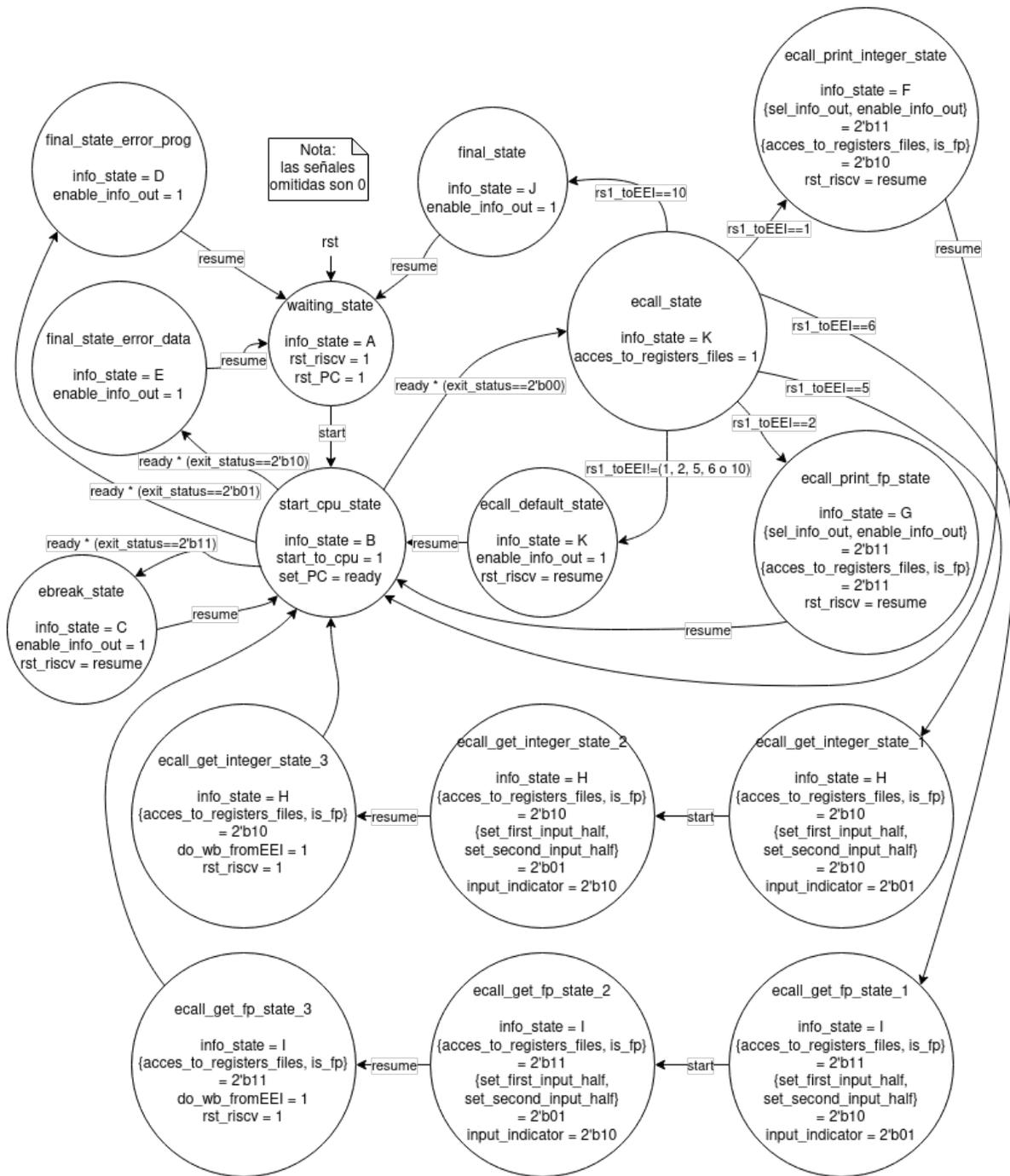


Figura 4.58: Diagrama *MDS* para el *SoC*.

A continuación, se presenta el rol de cada uno de los estados de la *FSM* descrita en la figura 4.58:

- **waiting_state:** *SoC state = waiting state*. Mantiene el *core complex* en *reset* y el registro con el *Program Counter* inicial en 0. Luego, espera la señal *start* para pasar al estado **start_cpu_state**. Esta es la única máquina de estados que simplemente recibe un pulso de la señal *start*, el resto de las máquinas descritas requieren que dicha señal sea 1 durante toda la ejecución.
- **start_cpu_state:** *SoC state = cpu working*. Inicia la ejecución del *core complex*, cuando finaliza dicha ejecución se actualiza el *Program Counter* por el de la instrucción que sigue al

ecall/ebreak ($PC + 4$). Luego, según el *exit_status* se selecciona el siguiente estado:

00: **ecall_state**.

01: **final_state_error_prog**.

10: **final_state_error_data**.

11: **ebreak_state**.

- **ebreak_state**: *SoC state = ebreak*. Detiene la ejecución del *core complex* y muestra $PC + 4$. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **start_cpu_state**.
- **final_state_error_prog**: *SoC state = prog error*. Detiene la ejecución del *core complex* y muestra $PC + 4$. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **waiting_state**.
- **final_state_error_data**: *SoC state = data error*. Detiene la ejecución del *core complex* y muestra $PC + 4$. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **waiting_state**.
- **ecall_state**: *SoC state = ecall default*. Detiene la ejecución del *core complex* y accede al *Register files* para ejecutar la llamada de sistema, por ello el siguiente estado depende de lo indicado en el registro *x17*, si es no válido pasa a **ecall_default_state**, de lo contrario:
 - $x17 = 1$: **ecall_print_integer_state**.
 - $x17 = 2$: **ecall_print_fp_state**.
 - $x17 = 5$: **ecall_get_integer_state_1**.
 - $x17 = 6$: **ecall_get_fp_state_1**.
 - $x17 = 10$: **final_state**.
- **ecall_default_state**: *SoC state = ecall default*. Muestra $PC + 4$. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **start_cpu_state**.
- **ecall_print_integer_state**: *SoC state = print integer*. Detiene la ejecución del *core complex* y muestra *x10*. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **start_cpu_state**.
- **ecall_print_fp_state**: *SoC state = print floating point*. Detiene la ejecución del *core complex* y muestra *f10*. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **start_cpu_state**.
- **ecall_get_integer_state_1**: *SoC state = get* integer* (solo se prende el *LED* tricolor derecho). Se recibe, mediante los interruptores, los primeros 16 bits del entero a ingresar, la entrada se confirma cuando se pulsa *start*. Luego, se pasa al estado **ecall_get_integer_state_2**.
- **ecall_get_integer_state_2**: *SoC state = get** integer* (solo se prende el *LED* tricolor izquierdo). Se recibe, mediante los interruptores, los últimos 16 bits del entero a ingresar, la entrada se confirma cuando se pulsa *resume*. Luego, se pasa al estado **ecall_get_integer_state_3**.
- **ecall_get_integer_state_3**: *SoC state = get integer* (se prenden ambos *LED* tricolor). Guarda el entero ingresado en *x10*, reinicia el *core complex* y vuelve al estado **start_cpu_state**.
- **ecall_get_fp_state_1**: *SoC state = get* floating point* (solo se prende el *LED* tricolor derecho). Se recibe, mediante los interruptores, los primeros 16 bits del punto flotante a ingresar, la entrada se confirma cuando se pulsa *start*. Luego, se pasa al estado **ecall_get_fp_state_2**.
- **ecall_get_fp_state_2**: *SoC state = get** floating point* (solo se prende el *LED* tricolor izquierdo). Se recibe, mediante los interruptores, los últimos 16 bits del punto flotante a ingresar, la entrada se confirma cuando se pulsa *resume*. Luego, se pasa al estado **ecall_get_fp_state_3**.
- **ecall_get_fp_state_3**: *SoC state = get floating point* (se prenden ambos *LED* tricolor). Guarda el punto flotante ingresado en *f10*, reinicia el *core complex* y vuelve al estado **start_cpu_state**.
- **final_state**: *SoC state = final state*. Detiene la ejecución del *core complex* y muestra $PC + 4$. Cuando se recibe la señal *resume* se reinicia el *core complex* y se pasa al estado **waiting_state**.

4.7.1 *debounce module*

Esta unidad se encarga de recibir la señal asíncrona proveniente de los pulsadores, eliminar los rebotes y sincronizarla con el reloj del sistema como un único pulso en un ciclo de reloj. A continuación, se detallan los estados del diagrama *MDS* (figura 4.59) de la *FSM* del módulo:

- **waiting_state:** espera a que se active la señal *signal_in*, luego pasa a **counter_state**.
- **counter_state:** se inicia un contador (de 16 bits), cuando este finaliza (en $0xFFFF$) y la señal *signal_in* sigue activa se pasa a **hold_state**. Si el contador no finaliza antes de que la señal *signal_in* desaparezca, se vuelve al estado **waiting_state**.
- **hold_state:** se mantiene en este estado hasta que la señal *signal_in* desaparezca, cuando esto ocurre se pasa al estado **response_state**.
- **response_state:** levanta la señal *signal_out* solo por un ciclo de reloj y pasa al estado **waiting_state**.

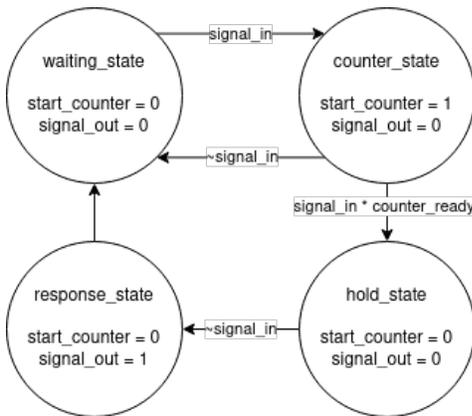


Figura 4.59: Diagrama *MDS* para el *debounce*.

5 Verificación, pruebas e implementación final

Los diseños presentados en la sección 4 se desarrollan y verifican utilizando el *software* recomendado por el fabricante de la tarjeta *FPGA* utilizada, *Digilent*. Dicho *software* es *Vivado* de *Xilinx*, en concreto la versión *2020.2* para *Linux* y la tarjeta utilizada, como se menciona en la sección 2, es la *Nexys 4*. Por otra parte, es importante señalar que todos los diseños digitales están implementados en el lenguaje *HDL System Verilog*. Todos los códigos desarrollados se presentan en los anexos y están disponibles en *GitHub* en el repositorio [GianlucaDagostino/Trabajo_Titulo_SoC-RiscV](#).

A continuación, se presenta la etapa de verificación desarrollada, las pruebas realizadas sobre la *hardware* y los detalles de la implementación final para la versión del *core complex Pipelined*.

5.1 Etapa de verificación

En esta sección se presenta el proceso de verificación realizado para distintos módulos cruciales del diseño, los cuales son: *memory*, *ALU*, *fp_converter*, *fp_arithmetic_unit*, *FPU*, *riscv32imf_pipeline*, *riscv32imf_singlecycle* y *TOP*. Para el desarrollo y verificación de las unidades relacionadas con el cálculo de números de punto flotante es realmente útil tener acceso a un [convertor de punto flotante a binario y viceversa](#).

5.1.1 Testeo de: *memory.sv*

Para este módulo (ver anexo 1) se confecciona el *test bench* **tb_memory.sv** (disponible en el anexo 2). Dicho *test bench* realiza 12 pruebas de funcionamiento, ya sean operaciones de escritura o lectura, y verifica que los resultados sean los esperados. En las figuras 5.1 y 5.2 se aprecian los resultados obtenidos, los cuales son satisfactorios.

```

Último byte direccionable: 0000f41f

test 1: verificar señal "out_of_range"
>> addr: 0000f45f - data_in: 6a70a30c
options: load word
Initial time:          10000
Elapsed time:         0
test 1: passed

test 2: guardar data (word) y leerla
>> addr: 0000941b - data_in: 6a70a30c
options: store word
store - Initial time:      30000
store - Elapsed time:     30000
>> addr: 0000941b - data_in: 6a70a30c
options: load word
load - Initial time:       80000
load - Elapsed time:      10000
data_out: 6a70a30c
test 2: passed

test 3: guardar data (word) y leerla en el mismo bloque cache del test 2 pero otra palabra
>> addr: 0000941f - data_in: c672830c
options: store word
store - Initial time:      110000
store - Elapsed time:     10000
>> addr: 0000941f - data_in: c672830c
options: load word
load - Initial time:       140000
load - Elapsed time:      10000
data_out: c672830c
test 3: passed

test 4: leer la palabra almacenada en el test 2 (para comprobar que se haya almacenado correctamente despues de los tests anteriores)
>> addr: 0000941b - data_in: c672830c
options: load word
load - Initial time:       170000
load - Elapsed time:      10000
data_out: 6a70a30c
test 4: passed

test 5: leer un byte signed
>> addr: 0000941b - data_in: c672830c
options: load byte signed
load - Initial time:       200000
load - Elapsed time:      10000
data_out: 0000006a
test 5: passed

test 6: leer un byte signed
>> addr: 00009419 - data_in: c672830c
options: load byte signed
load - Initial time:       230000
load - Elapsed time:      10000
data_out: ffffffff3
test 6: passed

```

Figura 5.1: Resultados de `tb_memory.sv` parte 1.

```

test 7: leer el mismo byte anterior pero unsigned
>> addr: 00009419 - data_in: c672830c
options: load byte unsigned
load - Initial time:          260000
load - Elapsed time:         10000
data_out: 000000a3
test 7: passed

test 8: ahora se modifica el byte leido en los test 6 y 7, luego se lee
>> addr: 00009419 - data_in: c67283fc
options: store byte
store - Initial time:         290000
store - Elapsed time:        10000
>> addr: 00009419 - data_in: c67283fc
options: load byte unsigned
load - Initial time:          320000
load - Elapsed time:         10000
data_out: 000000fc
test 8: passed

test 9: se comprueba que la palabra donde se almacena el byte modificado en el test 8 mantenga el resto de sus bits en orden
>> addr: 00009419 - data_in: c672830c
options: load word
load - Initial time:          350000
load - Elapsed time:         10000
data_out: 6a70fc0c
test 9: passed

test 10: se cambian los bits del tag c/r a la dirección del test 9 y se guarda un halfword, luego se lee el word
>> addr: 00001419 - data_in: c672830c
options: store halfword
store - Initial time:         380000
store - Elapsed time:        20000
>> addr: 00001419 - data_in: c672830c
options: load word
load - Initial time:          420000
load - Elapsed time:         10000
data_out: 0000830c
test 10: passed

test 11: se vuelve a usar la dirección del test 9 para leer el halfword almacenada en dicha dirección
>> addr: 00009419 - data_in: c672830c
options: load halfword unsigned
load - Initial time:          450000
load - Elapsed time:         20000
data_out: 0000fc0c
test 11: passed

test 12: misma operación del test 11 pero con signo
>> addr: 00009419 - data_in: c672830c
options: load halfword signed
load - Initial time:          490000
load - Elapsed time:         10000
data_out: fffffc0c
test 12: passed

```

Figura 5.2: Resultados de `tb_memory.sv` parte 2.

5.1.2 Testeo de: *ALU.sv*

Este módulo (ver anexo 1) se verifica con la ayuda de un programa escrito en *C* (no se utiliza *Python* debido a que no permite manipular los modos de redondeo deseados. Esto es relevante en las siguientes simulaciones), `tb_ALU_gen.c` (disponible en el anexo 3). Dicho programa al ser ejecutado pregunta al usuario cuántos *set test* se desean escribir y cuántos se desean descartar. A partir de dicha información genera el archivo `tb_ALU.sv` con la cantidad de *set test* pertinentes. Cada uno de estos *sets* se construye a partir de 2 números enteros pseudoaleatorios y escribe un *test* para cada una de las 22 operaciones de la *ALU*. Para el caso de las operaciones de comparación se agregan 2 *tests* extras pues se compara cada número consigo mismo y ambos por separado, esto da un total de 34 *tests* por *set*. Para realizar la mayor cantidad de pruebas posibles en un tiempo razonable, se realizan 10 simulaciones de 1000 *set tests* cada una, esto evita que el computador tenga un consumo excesivo de memoria y sea capaz de realizar las simulaciones sin problemas. En la figura 5.3 se aprecian los resultados obtenidos, los cuales muestran un correcto funcionamiento por parte del módulo.

```

$ gcc tb_ALU ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 1000
Enter n° of initial set test to discard (max input 99999): 0
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 1001010001110000100011000100101
in2      : 0111011011110011000010010100000
Outputs
res: PASSED!!
res      : 00011010110110110000100110000101
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 3000
Enter n° of initial set test to discard (max input 99999): 2000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 2999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 111110110001101111111111000101
in2      : 00001010010111000110101101101000
Outputs
res: PASSED!!
res      : 0000001001000110110110110000101
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 5000
Enter n° of initial set test to discard (max input 99999): 4000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 4999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 001000101010010111101110010101
in2      : 1101101001101010010100110001010
Outputs
res: PASSED!!
res      : 001000101010010111101101110010101
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 7000
Enter n° of initial set test to discard (max input 99999): 6000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 6999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 1011011010111010101000011010001
in2      : 011100000111100000111010101111
Outputs
res: PASSED!!
res      : 010001101100001001100011110010
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 9000
Enter n° of initial set test to discard (max input 99999): 8000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 8999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 00010010001010101010110010111111
in2      : 000101100110100101001101110000
Outputs
res: PASSED!!
res      : 000100100010101010110010111111
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 2000
Enter n° of initial set test to discard (max input 99999): 1000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 1999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 0111110000000100011100111000110
in2      : 0010001110100111100111011011101
Outputs
res: PASSED!!
res      : 0001000100010001001001100010001111
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 4000
Enter n° of initial set test to discard (max input 99999): 3000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 3999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 11111001011101000011110101001001
in2      : 0010001010010111111101000000110
Outputs
res: PASSED!!
res      : 000001101001100011001100011111
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 6000
Enter n° of initial set test to discard (max input 99999): 5000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 5999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 10100110010001100010000010000011
in2      : 00101110011001011001100100101110
Outputs
res: PASSED!!
res      : 000110001001100110001101111001
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 8000
Enter n° of initial set test to discard (max input 99999): 7000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 7999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 111111000110010110000001100101
in2      : 1100010000111101011100011100011
Outputs
res: PASSED!!
res      : 00111001111010000001110000010
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

```

$ ./tb_ALU_gen
Is little endian
Enter n° of set test to write (max input 99999): 10000
Enter n° of initial set test to discard (max input 99999): 9000
N° of tests written: 34000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 9999 - op: modulo (unsigned) <<<
Inputs
operation: 10111
in1      : 0111101001110000110000111111011
in2      : 010000101010010010101011000101
Outputs
res: PASSED!!
res      : 001101111001100000110000110110
boolean_res: PASSED!!
boolean_res : 0
-----
Total errors found      : 0 (0.000000 percentage)
Total out errors found  : 0 (0.000000 percentage)
Total boolean errors found : 0 (0.000000 percentage)
END

```

Figura 5.3: Resultados de tb_ALU.v.

5.1.3 Testeo de: *fp_converter.sv*

Este módulo (ver anexo 1) se verifica de la misma forma que la *ALU* (ver sección 5.1.2). El programa que genera los *set tests* es **tb_fp_converter_gen.c** (ver anexo 3) y retorna el archivo **tb_fp_converter.sv**. Una diferencia con el caso anterior es que solo genera un número de forma pseudoaleatorios por *set test* y es tratado como entero y punto flotante al mismo tiempo (mantienen la misma representación binaria). Cada *set test* realiza las 4 operaciones de conversión para cada uno de los 4 modos de redondeo que permite seleccionar *C* (*RMM* no se puede testear debido a que no está implementado en *C*), lo que da un total de 16 *tests* por *set*. Es importante mencionar que en las conversiones de punto flotante a entero (con o sin signo) se deben tomar ciertas consideraciones en los casos de *overflow* o *underflow* debido a la forma en que el procesador *Intel*, donde se ejecuta **tb_fp_converter_gen.c**, maneja dichas excepciones. Al igual que en la sección 5.1.2 se realizan diversas simulaciones. En este caso son: 20 simulaciones de 500 *set tests* cada una. Los resultados obtenidos se pueden apreciar en las figuras 5.4, 5.5 y 5.6, corroborando un correcto funcionamiento del módulo.


```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 3500
Enter n° of initial set test to discard (max input 99999): 3000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 3499-fp2signed-RUP <<<
in: 01110011011000100101100000111011
integer_is_signed: 1
option: 1
rm: 011
Initial time:          460260000
Elapsed time:         10000
out: PASSED!!
out: 01111111111111111111111111111111
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     170
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 4000
Enter n° of initial set test to discard (max input 99999): 3500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 3999-fp2signed-RUP <<<
in: 0111110000001000111001111000110
integer_is_signed: 1
option: 1
rm: 011
Initial time:          458500000
Elapsed time:         10000
out: PASSED!!
out: 01111111111111111111111111111111
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     150
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 4500
Enter n° of initial set test to discard (max input 99999): 4000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 4499-fp2signed-RUP <<<
in: 01000100110111000001000100001000100
integer_is_signed: 1
option: 1
rm: 011
Initial time:          461260000
Elapsed time:         10000
out: PASSED!!
out: 000000000000000000000000001101110001
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     140
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 5000
Enter n° of initial set test to discard (max input 99999): 4500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 4999-fp2signed-RUP <<<
in: 000100110111110100010000000101
integer_is_signed: 1
option: 1
rm: 011
Initial time:          458820000
Elapsed time:         10000
out: PASSED!!
out: 0000000000000000000000000000000001
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     200
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 5500
Enter n° of initial set test to discard (max input 99999): 5000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 5499-fp2signed-RUP <<<
in: 0010101101111011010101011000001
integer_is_signed: 1
option: 1
rm: 011
Initial time:          458180000
Elapsed time:         10000
out: PASSED!!
out: 0000000000000000000000000000000001
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     160
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 6000
Enter n° of initial set test to discard (max input 99999): 5500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit.sv

```

```

>>>Test 5999-fp2signed-RUP <<<
in: 111110110001101111111111111000101
integer_is_signed: 1
option: 1
rm: 011
Initial time:          461580000
Elapsed time:         10000
out: PASSED!!
out: 1000000000000000000000000000000000
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     130
END

```

Figura 5.5: Resultados de tb_fp_converter.sv parte 2.

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 6500
Enter n° of initial set test to discard (max input 99999): 6000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 6499-fp2signed-RUP <<<
in: 111011110011100110010011100111
integer_is_signed: 1
option: 1
rm: 011
Initial time:          457900000
Elapsed time:         10000
out: PASSED!!
out: 1000000000000000000000000000000000
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     160
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 7000
Enter n° of initial set test to discard (max input 99999): 6500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 6999-fp2signed-RUP <<<
in: 110001110000111010101100111010101
integer_is_signed: 1
option: 1
rm: 011
Initial time:          461740000
Elapsed time:         10000
out: PASSED!!
out: 111111111111111110110000010100111
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     170
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 7500
Enter n° of initial set test to discard (max input 99999): 7000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 7499-fp2signed-RUP <<<
in: 11100110010100001010101110001110
integer_is_signed: 1
option: 1
rm: 011
Initial time:          460540000
Elapsed time:         10000
out: PASSED!!
out: 1000000000000000000000000000000000
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     130
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 8000
Enter n° of initial set test to discard (max input 99999): 7500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 7999-fp2signed-RUP <<<
in: 111110010110100001110101001001
integer_is_signed: 1
option: 1
rm: 011
Initial time:          458820000
Elapsed time:         10000
out: PASSED!!
out: 1000000000000000000000000000000000
NV: PASSED!!
NV: 1
NX: PASSED!!
NX: 0
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     140
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 8500
Enter n° of initial set test to discard (max input 99999): 8000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 8499-fp2signed-RUP <<<
in: 1010101001001000100010101010111
integer_is_signed: 1
option: 1
rm: 011
Initial time:          458460000
Elapsed time:         10000
out: PASSED!!
out: 0000000000000000000000000000000000
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     130
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 9000
Enter n° of initial set test to discard (max input 99999): 8500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 8999-fp2signed-RUP <<<
in: 110000000110111101111000000110
integer_is_signed: 1
option: 1
rm: 011
Initial time:          459060000
Elapsed time:         10000
out: PASSED!!
out: 111111111111111111111111111111111
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     140
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 9500
Enter n° of initial set test to discard (max input 99999): 9000
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 9499-fp2signed-RUP <<<
in: 0100100101101000110001111111011
integer_is_signed: 1
option: 1
rm: 011
Initial time:          461420000
Elapsed time:         10000
out: PASSED!!
out: 0000000000000111101000110010000000
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     140
END

```

```

$ ./tb_fp_converter_gen
Is little endian
Enter n° of set test to write (max input 99999): 10000
Enter n° of initial set test to discard (max input 99999): 9500
N° of tests written: 8000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 9999-fp2signed-RUP <<<
in: 001000101010010111101110010101
integer_is_signed: 1
option: 1
rm: 011
Initial time:          459900000
Elapsed time:         10000
out: PASSED!!
out: 0000000000000000000000000000000001
NV: PASSED!!
NV: 0
NX: PASSED!!
NX: 1
-----
Total errors found:          0 (0.000000 percentage)
Total out_errors errors found: 0 (0.000000 percentage)
Total NV_errors errors found: 0 (0.000000 percentage)
Total NX_errors errors found: 0 (0.000000 percentage)
Minimum elapsed time:      10
Maximum elapsed time:     140
END

```

Figura 5.6: Resultados de tb_fp_converter sv parte 3.

5.1.4 Testeo de: *fp_arithmetic_unit.sv*

Este módulo (ver anexo 1) se verifica con el mismo método presentado para la *ALU* y el *fp_converter*. El código que genera los *test benches* es **tb_fp_arithmetic_unit_gen.c** (disponible en el anexo 3) y retorna **tb_fp_arithmetic_unit.sv**. En este caso se generan 2 números punto flotante de forma pseudoaleatoria por *set test*, cada uno de estos *sets* contiene 5 *tests*, uno por cada operación que realiza la unidad, por modo de redondeo (igual que en el caso anterior, solo se consideran 4 modos). Nuevamente, se realizan 20 simulaciones de 500 *set tests* cada una. Además, **tb_fp_arithmetic_unit_gen.c** se encarga de modificar los resultados que son *NaN* por el *NaN* canónico.

Es importante destacar que los resultados para el cálculo de la raíz cuadrada y la división se verifican de forma especial, esto debido a que en múltiples ocasiones el campo *fraction* del punto flotante resultante varía en ± 1 o ± 2 . Por el tiempo requerido en la implementación de esta unidad se acepta dicho margen de error. A pesar de lo anterior, como se puede ver en los resultados de las figuras 5.7, 5.8, 5.9 y 5.10, se detecta un total de 8 errores en todas las pruebas realizadas, 4 son de *status* (la bandera que levanta una operación) para cada modo de redondeo del *Test 4321-SUB*. Este error es despreciado debido a que finalmente las banderas no juegan un rol importante en el diseño completo ya que no se implementa el registro *FCSR*. Los otros 4 errores están relacionados con el número en punto flotante obtenido en la operación, donde las pruebas: *Test 3678-DIV-RTZ*, *Test 3678-DIV-RDN* y *Test 3678-DIV-RUP* presentan un error en el campo *fraction* de ± 3 y *Test 2758-MUL-RNE* de ± 1 . Estos errores se consideran aceptables debido a las restricciones de tiempo para el desarrollo del *SoC*.

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 500
Enter n° of initial set test to discard (max input 99999): 0
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 499-SUB-RUP <<<
In1: 11101011000101110011010000101001
In2: 1000100100011101110100110010101
Operation: 100
Rounding_Mode: 011
Initial time: 5629610000
Elapsed time: 60000
out: PASSED!!
out : 11101011000101110011010000101000
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 1500
Enter n° of initial set test to discard (max input 99999): 1000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 1499-SUB-RUP <<<
In1: 01000000011000000110110101100110
In2: 1111111010100111111100111001001
Operation: 100
Rounding_Mode: 011
Initial time: 5676820000
Elapsed time: 60000
out: PASSED!!
out : 011111010100111111100111001010
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1210
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 2500
Enter n° of initial set test to discard (max input 99999): 2000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 2499-SUB-RUP <<<
In1: 0001001110111111101000100000101
In2: 10111110000010111111100100111010
Operation: 100
Rounding_Mode: 011
Initial time: 5668410000
Elapsed time: 60000
out: PASSED!!
out : 00111110000010111111100100111011
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1190
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 1000
Enter n° of initial set test to discard (max input 99999): 500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 999-SUB-RUP <<<
In1: 1001010001110001000111000100101
In2: 011101011111001100010010100000
Operation: 100
Rounding_Mode: 011
Initial time: 5680990000
Elapsed time: 60000
out: PASSED!!
out : 111101011111001100010010100000
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1190
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 2000
Enter n° of initial set test to discard (max input 99999): 1500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 1999-SUB-RUP <<<
In1: 011110000001000110011100111000110
In2: 001000111101001111100110111011101
Operation: 100
Rounding_Mode: 011
Initial time: 5609110000
Elapsed time: 60000
out: PASSED!!
out : 01111000000100011100111000110
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 3000
Enter n° of initial set test to discard (max input 99999): 2500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

```

```

>>>Test 2999-SUB-RUP <<<
In1: 11111010001011111111111000101
In2: 000010100101110001010101011000
Operation: 100
Rounding_Mode: 011
Initial time: 5632990000
Elapsed time: 60000
out: PASSED!!
out : 11111010001011111111111000101
status: PASSED!!
status: 111
-----
Total errors found: 1 (0.005000 percentage)
Total out errors found: 1 (0.010000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1180
END
>>>Test 2758-MUL-RNE <<<
In1: 00101000001110101010000101100
In2: 000101110101001010101011011001
Operation: 001
Rounding_Mode: 000
Initial time: 2873690000
Elapsed time: 60000
out: FAILED!!
out should be: 00000000111110010001010100001
out : 00000000111110010001010100000
status: PASSED!!
status: 001

```

Figura 5.7: Resultados de tb_fp_arithmetic_unit sv parte 1.

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 3500
Enter n° of initial set test to discard (max input 99999): 3000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

>>>Test 3499-SUB-RUP <<<
In1: 1100011000011110101100111101101
In2: 011110011010000100111110111101
Operation: 100
Rounding_Mode: 011
Initial time: 5656000000
Elapsed time: 60000
out: PASSED!!
out : 111110011010000100111110111101
status: PASSED!!
status: 111
-----
Total errors found: 0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1220
END

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 4000
Enter n° of initial set test to discard (max input 99999): 3500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

>>>Test 3599-SUB-RUP <<<
In1: 111110011010100001110101001001
In2: 0010001010010111111101000000110
Operation: 100
Rounding_Mode: 011
Initial time: 5656040000
Elapsed time: 60000
out: PASSED!!
out : 11111001011101000011110101001001
status: PASSED!!
status: 111
-----
Total errors found: 3 (0.015000 percentage)
Total out errors found: 3 (0.030000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1180
END

>>>Test 3678-DIV-RTZ <<<
In1: 01001001100001010010100100010010
In2: 01010101100001010101110111011111
Operation: 010
Rounding_Mode: 001
Initial time: 2009120000
Elapsed time: 720000
out: FAILED!!
out should be: 00110011011010101100110100011
out : 00110011011010101100110100110
status: PASSED!!
status: 111

>>>Test 3678-DIV-RDN <<<
In1: 01001001100001010010100100010010
In2: 01010101100001010101110111011111
Operation: 010
Rounding_Mode: 010
Initial time: 2012130000
Elapsed time: 720000
out: FAILED!!
out should be: 001100110110101011001110100011
out : 001100110110101011001110100110
status: PASSED!!
status: 111

>>>Test 3678-DIV-RUP <<<
In1: 01001001100001010010100100010010
In2: 01010101100001010101110111011111
Operation: 010
Rounding_Mode: 011
Initial time: 2015140000
Elapsed time: 720000
out: FAILED!!
out should be: 001100110110101011001110100100
out : 001100110110101011001110100111
status: PASSED!!
status: 111

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 4500
Enter n° of initial set test to discard (max input 99999): 4000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit sv

>>>Test 4499-SUB-RUP <<<
In1: 110000000101101101111000000110
In2: 1000001110100101000111001001000
Operation: 100
Rounding_Mode: 011
Initial time: 5651580000
Elapsed time: 60000
out: PASSED!!
out : 110000000011011101111000000101
status: PASSED!!
status: 111
-----
Total errors found: 4 (0.020000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 4 (0.040000 percentage)
Minimum elapsed time: 20
Maximum elapsed time: 1180
END

>>>Test 4321-SUB-RNE <<<
In1: 1000000001110011110010010110001
In2: 10000000011001010001010001010101
Operation: 100
Rounding_Mode: 000
Initial time: 3629180000
Elapsed time: 70000
out: PASSED!!
out : 1000000000010011010101010000100
status: FAILED!!
status should be: 000
status: 001

>>>Test 4321-SUB-RTZ <<<
In1: 1000000001110011110010010110001
In2: 10000000011001010001010001010101
Operation: 100
Rounding_Mode: 001
Initial time: 3632130000
Elapsed time: 70000
out: PASSED!!
out : 100000000001001101010101010000100
status: FAILED!!
status should be: 000
status: 001

>>>Test 4321-SUB-RDN <<<
In1: 1000000001110011110010010110001
In2: 10000000011001010001010001010101
Operation: 100
Rounding_Mode: 010
Initial time: 3635080000
Elapsed time: 70000
out: PASSED!!
out : 100000000001001101010101010000100
status: FAILED!!
status should be: 000
status: 001

>>>Test 4321-SUB-RUP <<<
In1: 1000000001110011110010010110001
In2: 10000000011001010001010001010101
Operation: 100
Rounding_Mode: 011
Initial time: 3638030000
Elapsed time: 70000
out: PASSED!!
out : 100000000001001101010101010000100
status: FAILED!!
status should be: 000
status: 001

```

Figura 5.8: Resultados de `tb_fp_arithmetic_unit sv` parte 2.

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 5000
Enter n° of initial set test to discard (max input 99999): 4500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 4999-SUB-RUP <<<
In1: 0010001010100101111101110010101
In2: 11011011001101010010100110001010
Operation: 100
Rounding_Mode: 011
Initial time:          5629900000
Elapsed time:         60000
out: PASSED!!
out          : 01011011001101010010100110001011
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 6000
Enter n° of initial set test to discard (max input 99999): 5500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 5999-SUB-RUP <<<
In1: 10100110010001100010000010000011
In2: 0010111100110010111001100101110
Operation: 100
Rounding_Mode: 011
Initial time:          5654180000
Elapsed time:         60000
out: PASSED!!
out          : 101011110011001011100110101111
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 7000
Enter n° of initial set test to discard (max input 99999): 6500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 6999-SUB-RUP <<<
In1: 10110111010111010101000011010001
In2: 01110000011111000001111011011111
Operation: 100
Rounding_Mode: 011
Initial time:          5628880000
Elapsed time:         60000
out: PASSED!!
out          : 111100000111100000111101101111
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 8000
Enter n° of initial set test to discard (max input 99999): 7500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 7999-SUB-RUP <<<
In1: 11111110001100101100000001100101
In2: 11000100001111101011100011100011
Operation: 100
Rounding_Mode: 011
Initial time:          5627520000
Elapsed time:         60000
out: PASSED!!
out          : 11111110001100101100000001100100
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1200
END

```

```

$ ./tb_fp_ari./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 5500
Enter n° of initial set test to discard (max input 99999): 5000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 5499-SUB-RUP <<<
In1: 00000001111100010001100011111111
In2: 11100011101001010010111011100100
Operation: 100
Rounding_Mode: 011
Initial time:          5707980000
Elapsed time:         60000
out: PASSED!!
out          : 01100011101001010010111011100101
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 6500
Enter n° of initial set test to discard (max input 99999): 6000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 6499-SUB-RUP <<<
In1: 10010111100101101100101010110000
In2: 11000010000010101010111000100110
Operation: 100
Rounding_Mode: 011
Initial time:          5651500000
Elapsed time:         60000
out: PASSED!!
out          : 01000010000010101010111000100110
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 7500
Enter n° of initial set test to discard (max input 99999): 7000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 7499-SUB-RUP <<<
In1: 11010001111001001100010100001100
In2: 011110110101110010101001111100
Operation: 100
Rounding_Mode: 011
Initial time:          5671980000
Elapsed time:         60000
out: PASSED!!
out          : 111110110101110010110011111100
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1190
END

```

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 8500
Enter n° of initial set test to discard (max input 99999): 8000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

```

```

>>>Test 8499-SUB-RUP <<<
In1: 11110110111100111010000111111100
In2: 0100001100100001011110110101011
Operation: 100
Rounding_Mode: 011
Initial time:          5653760000
Elapsed time:         60000
out: PASSED!!
out          : 1111011011110011101000011111100
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found:  0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1190
END

```

Figura 5.9: Resultados de tb_fp_arithmetic_unit.v parte 3.

```

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 9000
Enter n° of initial set test to discard (max input 99999): 8500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

>>>Test 8999-SUB-RUP <<<
In1: 00010011000101010101110010111111
In2: 000101100110110010100110110110000
Operation: 100
Rounding_Mode: 011
Initial time:          5627630000
Elapsed time:         60000
out: PASSED!!
out: 10010110011010100101000111111101
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 9500
Enter n° of initial set test to discard (max input 99999): 9000
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

>>>Test 9499-SUB-RUP <<<
In1: 101110010110001011010111011011
In2: 11001000100001000001010000110010
Operation: 100
Rounding_Mode: 011
Initial time:          5671280000
Elapsed time:         60000
out: PASSED!!
out: 01001000100001000001101000110010
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1190
END

$ ./tb_fp_arithmetic_unit_gen
Is little endian
Enter n° of set test to write (max input 99999): 10000
Enter n° of initial set test to discard (max input 99999): 9500
N° of tests written: 10000
Operation ready, file generated: tb_fp_arithmetic_unit.v

>>>Test 9999-SUB-RUP <<<
In1: 0111101001110000110000111111011
In2: 01000001010100100101010111000101
Operation: 100
Rounding_Mode: 011
Initial time:          5645620000
Elapsed time:         60000
out: PASSED!!
out: 0111101001110000110000111111011
status: PASSED!!
status: 111
-----
Total errors found:      0 (0.000000 percentage)
Total out errors found: 0 (0.000000 percentage)
Total status errors found: 0 (0.000000 percentage)
Minimum elapsed time:   20
Maximum elapsed time:  1180
END

```

Figura 5.10: Resultados de `tb_fp_arithmetic_unit.v` parte 4.

5.1.5 Testeo de: *FPU.v*

Este módulo (ver anexo 1) es relativamente simple con respecto a los presentados anteriormente, al incorporar operaciones de punto flotante simple (las cuales se verifican al realizar simulaciones sobre el *core complex*) y coordinar las entradas y salidas de los módulos *fp_converter* y *fp_arithmetic_unit*. Por ello, para su verificación se realizan *test benches* simples basados en el código `tb_FPU.v` (ver anexo 2). Para corroborar que se realicen las operaciones como se espera, simplemente se toma dicho archivo y se realizan modificaciones según la operación a testear. Gracias a esto se comprueba el correcto funcionamiento del módulo. En la figura 5.11 se aprecia un ejemplo de la salida que genera este *test bench*.

```

FPU- out: 110000001100000000000000000000
FPU- NV, NX, UF, OF, DZ: 00000

fpu_op_selection- rm2fpu: 011
fpu_op_selection- fpu_option: 10100

```

Figura 5.11: Ejemplo del resultados obtenido al ejecutar `tb_FPU.v`.

5.1.6 Testeo de: *riscv32imf_singlecycle.v* y *riscv32imf_pipeline.v*

Para testear los módulos *riscv32imf_singlecycle* (ver anexo 1) y *riscv32imf_pipeline* (ver anexo 1) se escribe un programa simple en lenguaje ensamblador que utiliza todas las instrucciones implementadas, `testing_code_imf.s` (ver anexo 4). Además, se crea el *test bench*, compatible para ambos *cores*, `tb_riscv32imf_top.v` (ver anexo 2), específicamente para ser ejecutado con este programa de prueba. Es importante destacar que el *test bench* genera 3 archivos de salida:

prog_out.mem, *data_out.mem* y *registers.mem*, los cuales contienen, respectivamente, el estado final de la memoria de programa (la cual no cambia), la de datos y el estado final de los registros. Nótese además, que la carpeta con el código fuente del diseño debe contener los archivos *text_in.mem* y *data_in.mem*, los cuales contienen la memoria de programa y de datos inicial para la ejecución del programa compilado. Estos últimos archivos deben ordenar las palabras en formato hexadecimal en bloques de 4 palabras de derecha a izquierda. Para esto se utiliza un pequeño *script* de *Python* (detalles en el anexo 4.1).

Registers			Floating Point		
Name	Number	Value	Name	Number	Value
zero	0	0x00000000	ft0	0	0x00001ffc
ra	1	0x80001ffe	ft1	1	0x000017fd
sp	2	0xffffffff	ft2	2	0x000037f9
gp	3	0x00001800	ft3	3	0xbfe00000
tp	4	0x000007ff	ft4	4	0xbfe00000
t0	5	0x00000535	ft5	5	0x000081f0
t1	6	0x0029a800	ft6	6	0x000061f4
t2	7	0xfffffffffa	ft7	7	0x80001ffc
s0	8	0x00000ffe	fs0	8	0x000081f0
s1	9	0x000003fc	fs1	9	0x7fc00000
a0	10	0x00000ffe	fa0	10	0x80001ffc
a1	11	0x00001ffc	fa1	11	0x00001ffc
a2	12	0x000017fd	fa2	12	0xc0440000
a3	13	0x00001002	fa3	13	0xc0440000
a4	14	0x0000007c	fa4	14	0x00001ffc
a5	15	0x00000000	fa5	15	0xc0670000
a6	16	0x00000ffe	fa6	16	0xc0ab8000
a7	17	0x0000000a	fa7	17	0xc0c00000
s2	18	0xffffffff	fs2	18	0x4f800000
s3	19	0x7fffffff	fs3	19	0x00000000...
s4	20	0x00003134	fs4	20	0x00000000...
s5	21	0xfffffffffc	fs5	21	0x00000000...
s6	22	0x0000001f	fs6	22	0x00000000...
s7	23	0x00000000	fs7	23	0x00000000...
s8	24	0x00000020	fs8	24	0x00000000...
s9	25	0x000003fd	fs9	25	0x00000000...
s10	26	0x0000001b	fs10	26	0x00000000...
s11	27	0x00000001	fs11	27	0x00000000...
t3	28	0x00001ffd	ft8	28	0x00000000...
t4	29	0x00001ffc	ft9	29	0x00000000...
t5	30	0x001f4000	ft10	30	0x00000000...
t6	31	0x001f70f0	ft11	31	0x00000000...
pc		0x000031c4			

Figura 5.12: Estado final de los registros para la simulación de `testing_code_imf.s` en *RARS*.

Open + registers.mem 1000 GB Volum...		Open + registers.mem 1000 GB Volum...	
integer_regs		fp_regs	
00:	00000000	00:	00001ffc
01:	80001ffe	01:	000017fd
02:	ffffffffd	02:	000037f9
03:	00001800	03:	bfe00000
04:	000007ff	04:	bfe00000
05:	00000535	05:	000081f0
06:	0029a800	06:	000061f4
07:	ffffffffa	07:	80001ffc
08:	00000ffe	08:	000081f0
09:	000003fc	09:	7fc00000
0a:	00000ffe	0a:	80001ffc
0b:	00001ffc	0b:	00001ffc
0c:	000017fd	0c:	c0440000
0d:	00001002	0d:	c0440000
0e:	0000007c	0e:	00001ffc
0f:	00000000	0f:	c0670000
10:	00000ffe	10:	c0ab8000
11:	0000000a	11:	c0c00000
12:	ffffffffd	12:	4f800000
13:	7ffffffffd	13:	00000000
14:	00000134	14:	00000000
15:	fffffffcc	15:	00000000
16:	0000001f	16:	00000000
17:	00000000	17:	00000000
18:	00000020	18:	00000000
19:	000003fd	19:	00000000
1a:	0000001b	1a:	00000000
1b:	00000001	1b:	00000000
1c:	00001ffd	1c:	00000000
1d:	00001ffc	1d:	00000000
1e:	001f4000	1e:	00000000
1f:	001f40f0	1f:	00000000

Figura 5.13: Estado final de los registros para la simulación de `testing_code_imf.s` en *Vivado*. Captura de `registers.mem`.

En las figuras 5.12 y 5.13 se aprecia el estado final de los registros para la simulación del programa en *RARS* y el resultado del *test bench*, respectivamente, siendo ambos prácticamente idénticos. Las únicas discrepancias ocurren en los registros enteros: *x20* (0x14) y *x31* (0x1f). Estas diferencias se deben a que *RARS* considera como *Program Counter* inicial la dirección 0x3000 y la implementación realizada lo considera desde cero.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+1...	Value (+18)	Value (+1c)
0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x0000ffff	0x00000000	0x00001ffc	0x00000000	0x00000000	0x00000000	0x3fe00000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Figura 5.14: Memoria de datos resultante para la simulación de `testing_code_imf.s` en *RARS*.

```

data_out.mem
1000 GB Volume ~/Docum
00000080: 00000000
00000084: 0000ffff
00000088: 00000000
0000008c: 00001ffc
00000090: 00000000
00000094: 00000000
00000098: 00000000
0000009c: 3fe00000
000000a0: 00000000

```

Figura 5.15: Memoria de datos resultante para la simulación de `testing_code_imf.s` en *Vivado*. Captura de `data_out.mem`.

```

BEGIN

prog_out.mem - OK: -20000
data_out.mem - OK: -19999
registers.mem - OK: -19998

INFO: [USF-XSim-96] XSim co
INFO: [USF-XSim-97] XSim si
launch_simulation: Time (s)
run all

exit_status: 00

PC: 000001c4

END

```

Figura 5.16: Resultado obtenido en la consola de *Vivado* al ejecutar `tb_riscv32imf_top.sv`.

El estado final de la memoria de datos concuerda para ambos casos y se puede apreciar en las figuras 5.14 y 5.15 para la simulación en *RARS* y en *Vivado*, respectivamente. Además, en la figura 5.16 se puede ver la respuesta en consola del *test bench*, donde se observa el *Program Counter* final de la ejecución, nuevamente considerando la discrepancia del mismo entre ambas simulaciones, concuerda con el *PC* que se aprecia en la figura 5.12.

5.1.7 Testeo de: *TOP.sv*

Para verificar este módulo (ver anexo 1) se utiliza otro programa en lenguaje ensamblador, **fibonacci_simple.s** (disponible en el anexo 4). Este código implementa la siguiente función recursiva:

```
1 int fibonacci(int n){
2     if(n <= 2)
3         return 1;
4     else
5         return fibonacci(n-1) + fibonacci(n-2);
6 }
```

Y lo ejecuta de la siguiente forma:

```
1 #include <stdio.h>
2 int main(){
3     for(int i=3; i<=7; i++){
4         printf(" %d", i);
5         printf(" %d", fibonacci(i));
6     }
7     return 0;
8 }
```

El código ensamblador esta escrito de manera formal, considerando los registros *ra* y *sp*. Esto con el fin de simular la ejecución de un código compilado. Luego, para la ejecución de este programa se confecciona el *test bench* **tb_TOP.sv** (ver anexo 2), el cual simula la interacción con el usuario para poder visualizar los resultados.

```
32435568713
-- program is finished running (0) --
```

Figura 5.17: Resultado de *fibonacci_simple.s* en *RARS*.



Figura 5.18: Resultado de *fibonacci_simple.s* para la señal *info_out[31:0]* al ejecutar *tb_TOP.sv* en *Vivado*.

```
BEGIN

color_leds: 001001

INFO: [USF-XSim-96]
INFO: [USF-XSim-97]
launch_simulation: 1
run all

color_leds: 010010

END
```

Figura 5.19: Resultado de *fibonacci_simple.s* en la consola de *Vivado* al ejecutar *tb_TOP.sv*.

En la figura 5.17 se aprecia el resultado de la simulación en *RARS*. Mientras que en la figura 5.18 se puede ver la señal *info_out[31:0]*; esta es la palabra de 32 bits que se muestra en la placa *Nexys 4* mediante los *displays* de 7 segmentos. Considerando como está implementado el entorno de ejecución, esta señal concuerda con los resultados de la simulación en *RARS*. Por otro lado, en la figura 5.19 se muestra el resultado del *test bench* en la consola de *Vivado*. Aquí se aprecia la codificación binaria para los colores de los *LED* que informan el estado inicial y final de la ejecución del programa en el *SoC*. Los resultados obtenidos muestran un correcto funcionamiento del *SoC* y del *core complex* (ambas versiones).

5.2 Pruebas en *hardware*

A continuación, se presentan los resultados experimentales de las pruebas realizadas en la *Nexys 4*. Para esta tarea se dispone de dos programas en lenguaje ensamblador, el primero basado en el programa de verificación presentado en la sección 5.1.7 y el segundo enfocado en testear las operaciones aritméticas de punto flotante.

Es importante mencionar que el botón *start* es el pulsador *BTNU(F15)* de la placa, *resume* corresponde al pulsador *BTND(V10)* y el de *reset* al *CPU RESET*. Por otra parte, los *LED* de tricolor, *LD17* y *LD16* son los *LED* que indican el estado de ejecución.

5.2.1 *fibonacci_inputs.s*

Se escribe el programa **fibonacci_inputs.s** (ver anexo 4) basado en el código, antes mencionado, **fibonacci_simple.s**. Este nuevo programa, nuevamente implementa la función *fibonacci* (presentada en 5.1.7). La diferencia radica en la interacción con el usuario, en este caso se implementa dicha función de la siguiente forma:

```

1 #include <stdio.h>
2 int main(){
3     int a, b;
4     scanf("%d", &a);
5     // Entre cada entrada se agrega un "ebreak", para probar su funcionamiento.
6     scanf("%d", &b);
7     for(int i=a; i<=b; i++)
8         printf("%d", fibonacci(i));
9     return 0;
10 }
```

En la figura 5.20 se aprecia el resultado de la simulación en *RARS* para este programa. Por otro lado, en la figura 5.21 se pueden ver las imágenes de cada estado de la ejecución del programa corriendo en la tarjeta *FPGA*.

```

1
9
112358132134
-- program is finished running (0) --
```

Figura 5.20: Resultado de *fibonacci_inputs.s* en *RARS*.



Figura 5.21: Resultado de *fibonacci_inputs.s* en la *Nexys 4*.

Las imágenes de la figura 5.21 están numeradas y en orden. La imagen número 1 muestra el estado de espera del *SoC*. Para iniciar la ejecución se oprime el botón *start*. En la imagen 2 se introduce la primera mitad de 16 bits del primer entero a ingresar. Para confirmar esta mitad se oprime *start*, luego la segunda mitad se ingresa en la foto número 3 y se confirma con el botón *resume*. En la imagen 4 se aprecia el *ebreak* que separa el ingreso de ambos números enteros y en las imágenes 5 y 6 se ingresa el segundo entero de la misma forma que el primero. Luego, entre las imágenes 7 y 15 se pueden ver las salidas del programa. Para avanzar al siguiente resultado (entero) se oprime *resume*. Nótese que el último dígito de la derecha en la imagen número 13 es el dígito hexadecimal *D* debido a que está acompañado por el punto decimal de la derecha. Para pasar de la pantalla presentada en la imagen 15 a la de fin de ejecución (con los *LED* de estado en verde y mostrando el estado del *Program Counter* al finalizar la ejecución) de la foto 16 se oprime *resume*.

Si se vuelve a oprimir *resume* se retorna al estado de espera (imagen 17). Finalmente, se corrobora el correcto funcionamiento del programa en la tarjeta *FPGA* al obtenerse los resultados esperados.

5.2.2 *operating_fp.s*

Para probar las operaciones aritméticas de punto flotante se escribe el programa ***operating_fp.s*** (ver anexo 4). Este programa, además, guarda 2 palabras en memoria, 0xBBBBBBBB y 0xEEEEEEEE, las cuales se imprimen como enteros en pantalla al inicio y al final de la ejecución, respectivamente. Durante la ejecución, se reciben dos número en punto flotante en los registros *ft1* y *ft2* para luego realizar las siguientes operaciones:

```
1  fmul.s    ft3, ft1, ft2
2  fadd.s    ft4, ft1, ft2
3  fsub.s    ft5, ft3, ft4
4  fdiv.s    ft6, ft4, ft1
5  fsqrt.s   ft7, ft3
6  fmadd.s   ft8, ft3, ft6, ft1
7  fmsub.s   ft9, ft3, ft6, ft1
8  fnmadd.s  ft10, ft3, ft6, ft1
9  fnmsub.s  ft11, ft3, ft6, ft1
```

Después se imprimen en pantalla los números en punto flotante de los registros: *ft3*, *ft4*, *ft5*, *ft6*, *ft7*, *ft8*, *ft9*, *ft10* y *ft11*.

```
-11453246135.7
91.26
520.18296.96423.2220217.01052722.8074998854.278842.87-8854.27-8842.87-286331154
```

Figura 5.22: Resultado de *operating_fp.s* en *RARS*.

En la figura 5.22 se aprecia la salida de la consola del simulador *RARS* para este programa, donde el primer número en punto flotante ingresado es 5.7 y el segundo es 91.26. Por otro lado, -1145324613 corresponde a la palabra 0xBBBBBBBB y -286331154 corresponde a 0xEEEEEEEE. Luego, los números de la tercera línea antes de -286331154 son los resultados de los registros antes mencionados (*ft3* a *ft11*). Sin embargo, resulta más cómodo visualizar los resultados de estos registros en la figura 5.23 ya que se encuentra en formato hexadecimal, el mismo formato de visualización utilizado en la tarjeta *FPGA*.



Figura 5.25: Resultado de *operating_fp.s* en la *Nexys 4*.

En la figura 5.25 se pueden ver las 18 imágenes con los resultados de las pruebas sobre el *hardware* para este programa. La imagen número 1, nuevamente, muestra el estado de espera del *SoC*, la 2 muestra el *print* inicial, como entero, de la palabra `0xB8888888`. Luego, las imágenes 3 y 4 muestran como se ingresa el primer número en punto flotante, 5.7 y las fotos 5 y 6 muestran el ingreso de 91.26 (se ingresan de la misma forma que los enteros del caso anterior, se presiona *start* para ingresar la primera mitad y *resume* para la segunda). Después, para visualizar cada uno de los números en punto flotante de los registros *ft3* a *ft11* se debe ir presionando *resume*, estos se pueden ver en orden entre las fotos 7 y 15. En la imagen número 16 se aprecia la palabra `0xEEEEEEEE` como corresponde, se oprime *resume* y, finalmente, se termina la ejecución en la imagen 17. Presionando *resume* nuevamente se vuelve al estado de espera de la imagen 18.

Si se observa el estado final de los registros de la simulación en *RARS* en la figura 5.23 y se compara con el resultado de las fotos 7 a 15 de la figura 5.25. Se pueden apreciar ciertas discrepancias entre ellos. En concreto, los registros *ft8* y *ft10* difieren en 1 con respecto a los resultados de la simulación y *ft7* en *RARS* es $0x41B675C2 \approx 22.8075$, mientras que en la imagen número 11 de los resultados sobre la tarjeta *FPGA* es $0x41B7DA15 \approx 22.9815$. Estas diferencias concuerdan con lo mencionado en la sección 5.1.4 y el error del cálculo de la raíz cuadrada mencionado en [27], por lo que se consideran aceptables.

5.3 Detalles de la implementación

En esta sección se presentan los detalles de la implementación final del *SoC* obtenido. Además, en el anexo 5, se presenta un ejemplo del modo de uso del mismo (manual de usuario). Para la síntesis del diseño final, etapa en la cual se interpreta el código fuente en una representación a nivel de compuertas, se considera el *core complex Pipelined* y la estrategia por defecto de *Vivado* (*Strategy: Vivado Synthesis Defaults*). Al terminar este proceso de síntesis no se genera señal alguna de error o *warning* en el *software* y una vez finalizado se pueden realizar las simulaciones de comportamiento del sistema digital obtenido.

Después de la síntesis del diseño se debe realizar el proceso de implementación en *Vivado*. Este proceso incluye todos los pasos para colocar y enrutar la lista de conexiones a los recursos del dispositivo *FPGA*, cumpliendo con las limitaciones lógicas, físicas y de tiempo del diseño. En esta etapa es fundamental un archivo de extensión “.*xdc*”; este define los puertos de E/S para el sistema y el reloj del mismo, en este caso dicho archivo es *top_constraints.xdc* (disponible en el anexo 1).

Durante el proceso de implementación pueden surgir 2 *warnings*, el primero es: [*Pwropt 34-321*] *HACOOException: Too many TFIs and TFOs in design, exiting pwropt*. Este se soluciona ejecutando *set_param pwropt.maxFaninFanoutToNetRatio 1000* en la consola de *Vivado*. Por otro lado, el segundo *warning* es: [*Power 33-332*] *Found switching activity that implies high-fanout reset nets being asserted for excessive periods of time which may result in inaccurate power analysis*. Sin embargo, no se ahonda en su resolución ya que no afecta al proyecto y no se considera pertinente invertir tiempo en ello.

Luego de la síntesis e implementación del diseño se procede a generar el *bitstream*. Este corresponde al archivo con el cual se carga el diseño en el *FPGA*. Luego de obtener el *bitstream* se generan 93 *warnings* nuevos, todos ellos relacionados con posibles optimizaciones del diseño enfocados en los multiplicadores implementados. Esto debido a que se utiliza inferencia para todas las operaciones de multiplicación. Estos *warnings* se dividen en 3 tipos:

- 37 son [*DRC DPIP-1*] *Input pipelining*: indica que aplicar un diseño *Pipelined* a la entrada del bloque *DSP48* del *FPGA* (utilizado para las multiplicaciones) mejora el rendimiento (*performance*) del diseño.
- 28 son [*DRC DPOP-1*] *PREG Output pipelining*: indica que aplicar un diseño *Pipelined* a la salida del bloque *DSP48* del *FPGA* (utilizado para las multiplicaciones) mejora el rendimiento (*performance*) y el consumo de energía del diseño.
- 28 son [*DRC DPOP-2*] *MREG Output pipelining*: indica que aplicar un diseño *Pipelined* a la operación de multiplicación mejora el rendimiento (*performance*) y el consumo de energía del diseño.

Nuevamente, por razones de tiempo no se profundiza en solucionar estos problemas debido a que no suponen un problema para el funcionamiento del diseño obtenido pero es un punto a mejorar.

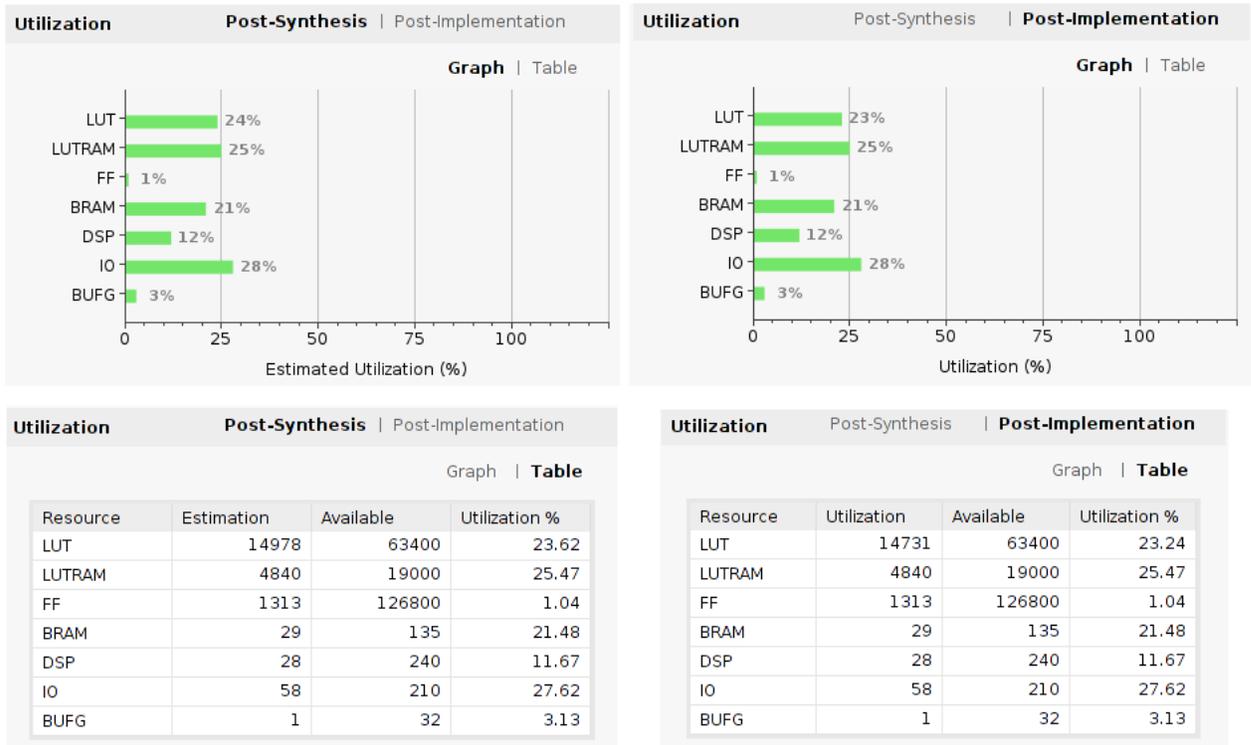


Figura 5.26: Resumen de los recursos de *hardware* utilizados, *post-synthesis* y *post-implementation*, de la tarjeta *FPGA Nexys 4* en la implementación.

En la figura 5.26 se visualizan los gráficos de barras y tablas con el detalle del uso de recursos del *FPGA post-synthesis* y *post-implementation*, donde se aprecia el bajo uso de éstos por parte diseño final obtenido. Por lo tanto, aún quedan muchos recursos del *FPGA* que se pueden utilizar para diseñar un *SoC* más complejo y completo empleando el *core complex* aquí obtenido como base.

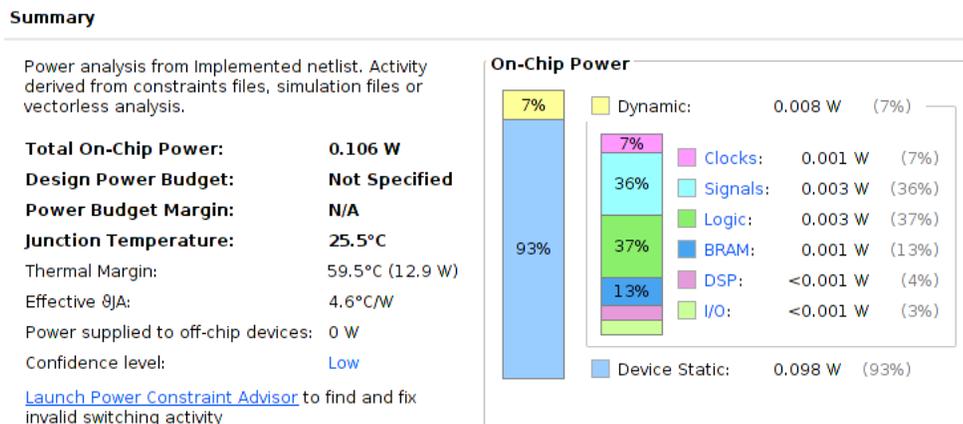


Figura 5.27: Resumen del consumo de energía de la implementación realizada.

El detalle del consumo de energía estimado para el *SoC* implementado se puede apreciar en detalle en la figura 5.27. Sin embargo, debido al *warning* ya mencionado, [Power 33-332], esta información puede resultar inexacta.

Clock Summary			
Name	Waveform	Period (ns)	Frequency (MHz)
CLK	{0.000 135.000}	270.000	3.704

Figura 5.28: Resumen del reloj implementado.

Los detalles del reloj que rige al sistema completo se pueden ver en la figura 5.28, donde se aprecia la frecuencia, el periodo y la forma de onda del mismo. La frecuencia final obtenida resulta ser bastante baja, esto es debido a la poca experiencia previa en el diseño de sistemas digitales del estilo. Por ello, no se aprovechan todas las bondades y técnicas de diseño que ofrece *System Verilog*, sobre todo en la implementación de la unidad de punto flotante, la cual es la que más penaliza el rendimiento del sistema.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 83.461 ns	Worst Hold Slack (WHS): 0.050 ns	Worst Pulse Width Slack (WPWS): 133.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 53136	Total Number of Endpoints: 53136	Total Number of Endpoints: 6304

All user specified timing constraints are met.

Figura 5.29: Resumen de las características del *timing* del diseño implementado.

Finalmente, en la imagen 5.29 se pueden distinguir los detalles de *timing* del diseño, incluyendo los peores casos y detalles para el *setup time* (tiempo mínimo en el cual los datos deben estar estables antes de que se active el flanco del reloj), el *hold time* (tiempo mínimo en el cual los datos deben estar estables después de que se active el flanco del reloj) y el *pulse width* (ancho de pulso del reloj). Si bien se puede lograr una mejor frecuencia sin que *Vivado* arroje algún error relacionado con el *timing* del diseño, en la práctica (con el diseño cargado en el *hardware*) el *SoC* no funciona correctamente.

6 Conclusiones

El trabajo de investigación previo y de diseño convergen a un resultado funcional para propósitos pedagógicos, tal como se propone en la sección 1. En concreto, se obtienen dos diseños para el *core complex*, uno *Single-Cycle* y otro *Pipelined*, basados en el juego de instrucciones *RV32IMF* de *RISC-V*. Luego, este se implementa en un entorno de ejecución básico el cual permite interactuar con el usuario ingresando (en representación binaria) y/o visualizando (en representación hexadecimal) números enteros o en punto flotante. Este resultado, a pesar de su simplicidad, corresponde a un *System on a Chip* al contener todos los elementos necesarios para su funcionamiento en el mismo chip *FPGA*.

El objetivo principal planteado en la sección 1.2 se cumple satisfactoriamente al lograr la implementación exitosa de todo el juego de instrucciones propuesto. Esto incluye las instrucciones base, *RV32I*, de *RISC-V* y sus extensiones *M* y *F*, para el *core complex* e integrarlo en un *FPGA*, obteniéndose el *SoC* final. Gracias a esta tarea se adquiere experiencia en el diseño de procesadores, sus entornos de ejecución, las consideraciones al respecto y su implementación mediante *System Verilog*. También, se logra un mayor entendimiento del área de interés en el diseño de sistemas digitales y se exploran las posibilidades que ofrece *RISC-V* actualmente en el mercado y en la academia de primera mano al tener que analizar el *ISA* oficial, presentado en [8], para la realización del proyecto. Además, se comprueba la eficacia de abordar tareas complejas del estilo con un enfoque de diseño como lo es la metodología *Top-Down* aprendida en [10]. Por otro lado, se destaca la complejidad de la etapa de verificación de diseños digitales grandes debido a que la mayor parte del tiempo de desarrollo se invierte en dicha etapa. Finalmente, es importante mencionar la versatilidad que ofrecen las plataformas de desarrollo y prototipado basados en chips *FPGA*, como la *Nexys 4*, pues permiten testear diseños digitales directamente en *hardware* de forma bastante amigable.

La etapa de verificación es especialmente complicada en este tipo de diseños debido a: la cantidad de simulaciones y *test benches* que se deben realizar en un producto formal para asegurar su correcto funcionamiento y el tiempo invertido en la resolución de los errores hallados en el proceso de depuración. En el trabajo realizado se aplican técnicas válidas de verificación (explicadas en la sección 5.1) e inspiradas en los trabajos presentados en la sección 2.3. Sin embargo, por las limitaciones de tiempo no se logran realizar pruebas más profundas de distintos módulos del diseño, por sobre todo, de la unidad aritmética de punto flotante que no termina de presentar un comportamiento completamente satisfactorio.

El estudio del trabajo previo de otros autores es fundamental para el éxito de desarrollos de este estilo en un tiempo razonable, pues dan una indicación clara de como abordar este tipo de desarrollos y las herramientas/metodologías existentes para ello. Luego, la revisión de trabajos específicos, como el presentado en [27], durante la etapa de diseño también juega un rol crucial pues durante esta etapa se vislumbran dificultades que no se consideran en las etapas tempranas de planteamiento.

El *SoC* obtenido resulta ser bastante limitado para una aplicación comercial/industrial real pero resulta bastante valioso, junto al contenido del presente informe, en el ámbito académico. El objetivo del trabajo es dar un punto de partida para hacer crecer este diseño inicial. Por ello, se busca la simplicidad de manera de reducir también la posibilidad de errores y permitir abarcar el diseño en un tiempo razonable. Como trabajo futuro, para obtener un *SoC* funcional en aplicaciones reales, se propone en primera instancia: optimizar el diseño de la *FPU* y realizar pruebas de verificación más intensivas y rigurosas, en particular se recomienda rediseñar su arquitectura utilizando mejor las bondades que ofrece *System Verilog* y considerar desde un primer momento los casos borde de números en punto flotante desnormalizados para evitar tener que duplicar el *hardware* que normaliza las mantisas de entrada, tal como se comenta en la sección 4.5.1), además de mejorar la precisión del

cálculo de raíces cuadradas. Por otro lado, se puede trabajar en una implementación más compleja del entorno de ejecución del *core complex* y agregar más funciones e interfaces de comunicación al diseño del *SoC*, tal como se propone en el primer planteamiento del diagrama de bloques de nivel superior de la figura 4.1, e integrar nuevas llamadas de sistemas acorde a las nuevas capacidades del mismo, además de añadir un *BOOT LOADER* para no tener que inicializar los diseños con el programa ya cargado directamente en las memorias internas del *core complex*. Esto permitiría aprovechar las memorias externas al *FPGA* que ofrece la *Nexys 4*. Otras posibles mejoras son la integración de los módulos *CSR* (*Control and Status Register*) y *A* (*Atomic*), que permitirían integrar múltiples núcleos en el diseño. También, se pueden añadir estrategias más sofisticadas para el manejo de los *control hazards* y aumentar el nivel de paralelismo en la ejecución de programas añadiendo un esquema *multiple issue* al *core complex*. Sin mencionar las posibles mejoras en el manejo de la memoria cache al agregar un esquema asociativo, ya sea parcial o completo. Hasta se podría llegar a añadir soporte para sistemas operativos, implementado una arquitectura con privilegios. En el diseño de procesadores y sistemas digitales quedan muchas posibilidades de mejora abiertas.

Implementar un *core complex* basado en *RISC-V* es especialmente interesante actualmente y también lo es generar documentación al respecto en español, para así fomentar la investigación a nivel nacional de las posibilidades que ofrece este *ISA* libre tal como sucede a nivel mundial. Como se menciona en la sección 1.1, en el ultimo tiempo *RISC-V* ha hecho eco de sus posibilidades, en concreto: la *European Processor Initiative* acaba de finalizar su primera versión de un procesador basado en *RISC-V* [28], la empresa *Microchip* ya tiene en producción una familia de *SoCs FPGA* con un núcleo basado en *RISC-V* integrado [29]. También, han surgido empresas dedicadas a explotar y explorar las posibilidades de esta tecnología, como: *VentanaMicro* (fundada en 2018) [30] y *SiFive* (fundada en 2015) [31], inclusive empresas con trayectoria como: *Imagination*, con más de 25 años en el mercado de la tecnología, ha puesto sus ojos en *RISC-V* [32] y *Apple* que está en búsqueda de profesionales con experiencia en *RISC-V* [33]. Esto deja en claro el impacto actual y futuro de este *ISA* y las posibilidades que ofrece al ser *hardware* libre. Por ello, es importante fomentar el desarrollo nacional en esta área y este trabajo de título busca acercar este mundo al ámbito académico nacional para poder comenzar a crear tecnología de punta localmente.

Bibliografía

- [1] E. P. DeBenedictis, “It’s Time to Redefine Moore’s Law Again,” in *Computer*, vol. 50, no. 2, pp. 72-75, Feb. 2017, doi: 10.1109/MC.2017.34.
- [2] H. R. Lee, C. H. Lin, K. H. Park, W. J. Kim, and H. J. Cho, “Development of SoC virtual platform for IoT terminals based on OneM2M,” *Proc. - Int. SoC Des. Conf. 2017, ISOCC 2017*, pp. 320–321, 2018, doi: 10.1109/ISOCC.2017.8368917.
- [3] Wei Hwang, “New trends in low power SoC design technologies,” p. 422, 2004, doi: 10.1109/soc.2003.1241559.
- [4] P. Chow, “RISC-(reduced instruction set computers),” in *IEEE Potentials*, vol. 10, no. 3, pp. 28-31, Oct. 1991, doi: 10.1109/45.127642.
- [5] “The rise of RISC - [Opinion],” in *IEEE Spectrum*, vol. 55, no. 8, pp. 18-18, Aug. 2018, doi: 10.1109/MSPEC.2018.8423577.
- [6] S. M. S. Trimberger, “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore’s Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation,” in *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16-29, Spring 2018, doi: 10.1109/MSSC.2018.2822862.
- [7] V. Chamola, S. Patra, N. Kumar, and M. Guizani, “FPGA for 5G: Re-configurable hardware for next generation communication,” *IEEE Wirel. Commun.*, vol. 27, no. 3, pp. 140–147, 2020, doi: 10.1109/MWC.001.1900359.
- [8] A. Waterman and K. Asanovic, “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA - Document Version 20191213,” *RISC-V Found.*, vol. I, 2019.
- [9] “Sistemas Digitales” notas de clases para EL4002-1, Departamento de Ingeniería Eléctrica, Universidad de Chile, Otoño 2018.
- [10] “Seminario de Sistemas Digitales” notas de clases para EL7039-1, Departamento de Ingeniería Eléctrica, Universidad de Chile, Primavera 2019.
- [11] “Arquitectura de Computadores” notas de clases para EL4102-1, Departamento de Ingeniería Eléctrica, Universidad de Chile, Primavera 2018.
- [12] D. A. Patterson and J. L. Hennessy, “Computer Organization and Design, The Hardware Software/Interface: RISC-V Edition”. Cambridge, Estados Unidos: Morgan Kaufmann, 2018.
- [13] “IEEE Standard for Floating-Point Arithmetic,– in IEEE Std 754-2008” , vol., no., pp.1-70, 29 Aug. 2008, doi: 10.1109/IEEESTD.2008.4610935.
- [14] V. Ii, P. Architecture, D. Version, A. Waterman, K. Asanovi, and C. S. Division, “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft,” vol. II, p. 135, 2019, [Online]. Available: <https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-privileged.pdf>.

- [15] A. Raveendran, V. B. Patil, D. Selvakumar and V. Desalpine, “A RISC-V instruction set processor-micro-architecture design and analysis,” 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), Bangalore, 2016, pp. 1-7, doi: 10.1109/VLSI-SATA.2016.7593047.
- [16] R. Höller, D. Haselberger, D. Ballek, P. Rössler, M. Krapfenbauer and M. Linauer, “Open-Source RISC-V Processor IP Cores for FPGAs — Overview and Evaluation,” 2019 8th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 2019, pp. 1-6, doi: 10.1109/MECO.2019.8760205.
- [17] H. Eassa, I. Adly and H. H. Issa, “RISC-V based implementation of Programmable Logic Controller on FPGA for Industry 4.0,” 2019 31st International Conference on Microelectronics (ICM), Cairo, Egypt, 2019, pp. 98-102, doi: 10.1109/ICM48031.2019.9021939.
- [18] A. Oleksiak, S. Cieślak, K. Marcinek and W. A. Pleskacz, “Design and Verification Environment for RISC-V Processor Cores,” 2019 MIXDES - 26th International Conference 'Mixed Design of Integrated Circuits and Systems', Rzeszów, Poland, 2019, pp. 206-209, doi: 10.23919/MIXDES.2019.8787108.
- [19] Beaulieu, L., Weppe, O., Le Ludec, B., & Lebeau, F. (2018). “Co-verification design flow for HDL languages: A complete development methodology”. ICECS 2017 - 24th IEEE International Conference on Electronics, Circuits and Systems, 2018-Janua, 530–533. <https://doi.org/10.1109/ICECS.2017.8292096>
- [20] “SiFive HiFive1 Rev B Getting Started Guide Version 1.2” 2021. https://starfivetech.com/uploads/hifive1b-getting-started-guide_v1.2.pdf
- [21] “SiFive FE310-G002 Preliminary Datasheet”. (n.d.). <https://cdn.sparkfun.com/assets/5/b/e/6/2/fe310-g002-ds.pdf>.
- [22] SiFive. (n.d.). “SiFive FE310-G002 Manual”. https://sifive.cdn.prismic.io/sifive%2F59a1f74e-d918-41c5-b837-3fe01ba7eaa1_fe310-g002-manual-v19p05.pdf.
- [23] “Nexys 4™ FPGA Board Reference Manual”, Digilent, 1300 Henley Court. April 11, 2016. [Online]. <https://digilent.com/reference/programmable-logic/nexys-4/reference-manual>.
- [24] Xilinx, “Vivado Design Suite”, 2021. [Online]. <https://www.xilinx.com/products/design-tools/vivado.html>.
- [25] P. Sanderson, K. Vollmar (2021) “RARS – RISC-V Assembler and Runtime Simulator” [Código fuente]. <https://github.com/TheThirdOne/rars>.
- [26] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language,” in IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , vol., no., pp.1-1315, 22 Feb. 2018, doi: 10.1109/IEEESTD.2018.8299595.
- [27] A. Hasnat, T. Bhattacharyya, A. Dey, S. Halder and D. Bhattacharjee, “A fast FPGA based architecture for computation of square root and Inverse Square Root,” 2017 Devices for Integrated Circuit (DevIC), 2017, pp. 383-387, doi: 10.1109/DEVIC.2017.8073975.
- [28] European Processor Initiative. (2021, Septiembre 22) “EPI EPAC1.0 RISC-V Test Chip Samples Delivered” [Online]. Available: <https://www.european-processor-initiative.eu/epi-epac1-0-risc-v-test-chip-samples-delivered/>
- [29] Microchip “MPFS250T Mid range, low power, with hard RISC-V based” [Online]. Available: <https://www.microchip.com/en-us/product/MPFS250T>
- [30] Ventana Micro “RISC-V Performance Leader Delivering A Family of Data Center Class CPU Cores” [Online]. Available: <https://www.ventanamicro.com/#about>

- [31] SiFive “Leading the RISC-V revolution” [Online]. Available: <https://www.sifive.com>
- [32] D. Harold, J. Jones. (2021, Febrero 22) “Imagination’s GPU selected by StarFive to create high-performance, small and low-cost BeagleV RISC-V AI single board computer” [Online]. Available: <https://www.imaginationtech.com/news/press-release/imaginations-gpu-selected-by-starfive-to-create-high-performance-small-and-low-cost-beaglev-risc-v-ai-single-board-computer/>
- [33] Apple (2021, Septiembre 2) “RISC-V High Performance Programmer” [Online]. Available: <https://jobs.apple.com/es-cl/details/200282667/risc-v-high-performance-programmer>

1 Código fuente del *SoC*

En esta sección se presentan todos los archivos del código fuente del *SoC* diseñado:

TOP.sv: archivo que contiene el módulo *top* del diseño completo. Nótese que en las líneas 212 y 213 se comenta el *core complex Single-Cycle* y se deja como predeterminado el *core complex Pipelined*, pero ambos son compatibles.

```
1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6 // set_param pwropt.maxFaninFanoutToNetRatio 1000
7 module TOP(
8     input clk, rst_asin, start_asin, resume_asin, // rst, start, resume deben ser pulsadores (
9         ↪ OJO, en todo el resto del diseño start se comporta de otra forma)
10    input [15:0] in,
11    output [5:0] color_leds,
12    output [15:0] leds, display
13 );
14 typedef enum {
15     waiting_state,
16     final_state_error_prog, final_state_error_data, start_cpu_state,
17     ebreak_state, ecall_state, ecall_default_state,
18     ecall_print_integer_state, ecall_print_fp_state,
19     ecall_get_integer_state_1, ecall_get_integer_state_2, ecall_get_integer_state_3,
20     ecall_get_fp_state_1, ecall_get_fp_state_2, ecall_get_fp_state_3,
21     final_state
22 } state_type;
23 state_type actual_state, next_state;
24 reg rst_PC, set_PC, set_first_input_half, set_second_input_half,
25     start_to_cpu, sel_info_out, enable_info_out,
26     acces_to_registers_files, is_fp, do_wb_fromEEI, rst_riscv;
27 reg [3:0] info_state;
28 reg [1:0] input_indicator;
29 /*
30     input_indicator =
31         00 o 11 - no input indicator
32         01 - first half indicator
33         10 - second half indicator
34     exit_status[1:0] =
35         00 -> ecall    (se lee registro x17 y según eso se toma decisión)
36         11 -> ebreak  (pausa ejecución)
37         01 -> program out of range
38         10 -> memory out of range
39 */
40 wire ready, prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
41     ↪ data_out_of_range_toEEI, rst, start, resume;
42 wire [1:0] exit_status;
43 wire [15:0] second_input_half, first_input_half;
44 wire [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI, PC, PC_reg,
```

```

    ↪ info_out, input_from_user;
43 assign info_out = enable_info_out ? (sel_info_out ? rs2_toEEI : PC_reg) : 32'b0;
44 assign input_from_user = {second_input_half, first_input_half};
45 assign leds = (input_indicator[1]^input_indicator[0]) ? in : 16'b0;
46 // color_leds_decoder
47 assign color_leds = (input_indicator==2'b01 & info_state==4'h8) ? 6'b000_101 : //
    ↪ get integer (PURPLE)
48     ((input_indicator==2'b10 & info_state==4'h8) ? 6'b101_000 : // get
    ↪ integer (PURPLE)
49     ((input_indicator==2'b01 & info_state==4'h9) ? 6'b000_110 : // get fp (
    ↪ YELLOW)
50     ((input_indicator==2'b10 & info_state==4'h9) ? 6'b110_000 : // get fp (
    ↪ YELLOW)
51     ((info_state==4'h0) ? 6'b001_001 : // A -> --B|--B = BLUE    - waiting
    ↪ state
52     ((info_state==4'h2) ? 6'b011_011 : // B -> -GB|-GB = CYAN    - cpu
    ↪ working
53     ((info_state==4'h3) ? 6'b111_111 : // C -> RGB|RGB = WHITE    -
    ↪ ebreak
54     ((info_state==4'h4) ? 6'b100_001 : // D -> R--|--B = RED/BLUE - prog
    ↪ error
55     ((info_state==4'h5) ? 6'b001_100 : // E -> --B|R-- = BLUE/RED - data
    ↪ error
56     ((info_state==4'h6) ? 6'b101_101 : // F -> R-B|R-B = PURPLE  - print
    ↪ integer
57     ((info_state==4'h7) ? 6'b110_110 : // G -> RG-|RG- = YELLOW  -
    ↪ print fp
58     ((info_state==4'h8) ? 6'b000_000 : // H -> ---|--- = NO COLOR - get
    ↪ integer
59     ((info_state==4'h9) ? 6'b000_000 : // I -> ---|--- = NO COLOR - get fp
60     ((info_state==4'hA) ? 6'b010_010 : // J -> -G-|-G- = GREEN    - final
    ↪ state
61     ((info_state==4'hB) ? 6'b100_100 : // K -> R--|R-- = RED      - ecall
    ↪ default
62         6'b000_000))))))))))))))))); //rgb_rgb
63 // FSM controller
64 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
65 always_comb
66     case (actual_state)
67         default: begin
68             {input_indicator, info_state}          = 6'b00_0000; // A
69             {set_first_input_half, set_second_input_half} = 2'b00;
70             {sel_info_out, enable_info_out}        = 2'b00;
71             {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
72             {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b00_00;
73             next_state = actual_state;
74         end
75         waiting_state: begin
76             {input_indicator, info_state}          = 6'b00_0000; // A
77             {set_first_input_half, set_second_input_half} = 2'b00;
78             {sel_info_out, enable_info_out}        = 2'b00;
79             {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;

```

```

80     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b10_01;
81     next_state = start ? start_cpu_state : waiting_state;
82 end
83 start_cpu_state: begin
84     {input_indicator, info_state}                 = 6'b00_0010; // B
85     {set_first_input_half, set_second_input_half} = 2'b00;
86     {sel_info_out, enable_info_out}              = 2'b00;
87     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
88     {rst_riscv, start_to_cpu, rst_PC} = 3'b01_0; set_PC = ready;
89     next_state = ready ? ((exit_status == 2'b11) ? ebreak_state :
90                          ((exit_status == 2'b01) ? final_state_error_prog :
91                          ((exit_status == 2'b10) ? final_state_error_data :
92                          ecall_state))) : start_cpu_state;
93 end
94 ebreak_state: begin
95     {input_indicator, info_state}                 = 6'b00_0011; // C
96     {set_first_input_half, set_second_input_half} = 2'b00;
97     {sel_info_out, enable_info_out}              = 2'b01;
98     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
99     {rst_riscv, start_to_cpu, set_PC, rst_PC}     = {resume, 3'b0_00};
100    next_state = resume ? start_cpu_state : ebreak_state;
101 end
102 final_state_error_prog: begin
103     {input_indicator, info_state}                 = 6'b00_0100; // D
104     {set_first_input_half, set_second_input_half} = 2'b00;
105     {sel_info_out, enable_info_out}              = 2'b01;
106     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
107     {rst_riscv, start_to_cpu, set_PC, rst_PC}     = 4'b00_00;
108     next_state = resume ? waiting_state : final_state_error_prog;
109 end
110 final_state_error_data: begin
111     {input_indicator, info_state}                 = 6'b00_0101; // E
112     {set_first_input_half, set_second_input_half} = 2'b00;
113     {sel_info_out, enable_info_out}              = 2'b01;
114     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
115     {rst_riscv, start_to_cpu, set_PC, rst_PC}     = 4'b00_00;
116     next_state = resume ? waiting_state : final_state_error_data;
117 end
118 ecall_print_integer_state: begin
119     {input_indicator, info_state}                 = 6'b00_0110; // F
120     {set_first_input_half, set_second_input_half} = 2'b00;
121     {sel_info_out, enable_info_out}              = 2'b11;
122     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b100;
123     {rst_riscv, start_to_cpu, set_PC, rst_PC}     = {resume, 3'b0_00};
124     next_state = resume ? start_cpu_state : ecall_print_integer_state;
125 end
126 ecall_print_fp_state: begin
127     {input_indicator, info_state}                 = 6'b00_0111; // G
128     {set_first_input_half, set_second_input_half} = 2'b00;
129     {sel_info_out, enable_info_out}              = 2'b11;
130     {acces_to_registers_files, is_fp, do_wb_fromEEI} = 3'b110;
131     {rst_riscv, start_to_cpu, set_PC, rst_PC}     = {resume, 3'b0_00};

```

```

132     next_state = resume ? start_cpu_state : ecall_print_fp_state;
133 end
134 ecall_get_integer_state_1: begin
135     {input_indicator, info_state}           = 6'b01_1000; // H
136     {set_first_input_half, set_second_input_half} = 2'b10;
137     {sel_info_out, enable_info_out}         = 2'b00;
138     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b100;
139     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b00_00;
140     next_state = start ? ecall_get_integer_state_2 : ecall_get_integer_state_1;
141 end
142 ecall_get_integer_state_2: begin
143     {input_indicator, info_state}           = 6'b10_1000; // H
144     {set_first_input_half, set_second_input_half} = 2'b01;
145     {sel_info_out, enable_info_out}         = 2'b00;
146     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b100;
147     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b00_00;
148     next_state = resume ? ecall_get_integer_state_3 : ecall_get_integer_state_2;
149 end
150 ecall_get_integer_state_3: begin
151     {input_indicator, info_state}           = 6'b00_1000; // H
152     {set_first_input_half, set_second_input_half} = 2'b00;
153     {sel_info_out, enable_info_out}         = 2'b00;
154     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b101;
155     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b10_00;
156     next_state = start_cpu_state;
157 end
158 ecall_get_fp_state_1: begin
159     {input_indicator, info_state}           = 6'b01_1001; // I
160     {set_first_input_half, set_second_input_half} = 2'b10;
161     {sel_info_out, enable_info_out}         = 2'b00;
162     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b110;
163     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b00_00;
164     next_state = start ? ecall_get_fp_state_2 : ecall_get_fp_state_1;
165 end
166 ecall_get_fp_state_2: begin
167     {input_indicator, info_state}           = 6'b10_1001; // I
168     {set_first_input_half, set_second_input_half} = 2'b01;
169     {sel_info_out, enable_info_out}         = 2'b00;
170     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b110;
171     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b00_00;
172     next_state = resume ? ecall_get_fp_state_3 : ecall_get_fp_state_2;
173 end
174 ecall_get_fp_state_3: begin
175     {input_indicator, info_state}           = 6'b00_1001; // I
176     {set_first_input_half, set_second_input_half} = 2'b00;
177     {sel_info_out, enable_info_out}         = 2'b00;
178     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b111;
179     {rst_riscv, start_to_cpu, set_PC, rst_PC}      = 4'b10_00;
180     next_state = start_cpu_state;
181 end
182 final_state: begin
183     {input_indicator, info_state}           = 6'b00_1010; // J

```

```

184     {set_first_input_half, set_second_input_half} = 2'b00;
185     {sel_info_out, enable_info_out} = 2'b01;
186     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
187     {rst_riscv, start_to_cpu, set_PC, rst_PC} = 4'b00_00;
188     next_state = resume ? waiting_state : final_state;
189 end
190 ecall_state: begin
191     {input_indicator, info_state} = 6'b00_1011; // K
192     {set_first_input_half, set_second_input_half} = 2'b00;
193     {sel_info_out, enable_info_out} = 2'b00;
194     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b100;
195     {rst_riscv, start_to_cpu, set_PC, rst_PC} = 4'b00_00;
196     next_state = (rs1_toEEI==31'b01010) ? final_state :
197         ((rs1_toEEI==31'b00001) ? ecall_print_integer_state :
198         ((rs1_toEEI==31'b00010) ? ecall_print_fp_state :
199         ((rs1_toEEI==31'b00101) ? ecall_get_integer_state_1 :
200         ((rs1_toEEI==31'b00110) ? ecall_get_fp_state_1 :
201         ecall_default_state)))));
202 end
203 ecall_default_state: begin
204     {input_indicator, info_state} = 6'b00_1011; // K
205     {set_first_input_half, set_second_input_half} = 2'b00;
206     {sel_info_out, enable_info_out} = 2'b01;
207     {access_to_registers_files, is_fp, do_wb_fromEEI} = 3'b000;
208     {rst_riscv, start_to_cpu, set_PC, rst_PC} = {resume, 3'b0_00};
209     next_state = resume ? start_cpu_state : ecall_default_state;
210 end
211 endcase
212 //riscv32imf_singlecycle
213 riscv32imf_pipeline
214 riscv32imf(
215     // inputs
216     .clk(clk), .rst(rst | rst_riscv), .start(start_to_cpu), .access_to_registers_files(
217     ↪ access_to_registers_files),
218     .is_wb_data_fp_fromEEI(is_fp), .do_wb_fromEEI(do_wb_fromEEI),
219     .is_rs1_fp_fromEEI(1'b0), .is_rs2_fp_fromEEI(is_fp),
220     .access_to_prog_mem(1'b0), .prog_rw_fromEEI(1'b0),
221     .prog_valid_mem_fromEEI(1'b0), .prog_is_load_unsigned_fromEEI(1'b0),
222     .access_to_data_mem(1'b0), .data_rw_fromEEI(1'b0),
223     .data_valid_mem_fromEEI(1'b0), .data_is_load_unsigned_fromEEI(1'b0),
224     .initial_PC(PC_reg),
225     .prog_addr_fromEEI(32'b0), .prog_in_fromEEI(32'b0),
226     .data_addr_fromEEI(32'b0), .data_in_fromEEI(32'b0),
227     .wb_data_fromEEI(input_from_user),
228     .prog_byte_half_word_fromEEI(2'b00), .data_byte_half_word_fromEEI(2'b00),
229     .rs1_add_fromEEI(5'b10001), .rs2_add_fromEEI(5'b01010), .wb_add_fromEEI(5'
230     ↪ b01010),
231     // outputs
232     .ready(ready),
233     .prog_ready_toEEI(prog_ready_toEEI), .prog_out_of_range_toEEI(
234     ↪ prog_out_of_range_toEEI),
235     .data_ready_toEEI(data_ready_toEEI), .data_out_of_range_toEEI(

```

```

    ↪ data_out_of_range_toEEI),
233     .exit_status(exit_status),
234     .prog_out_toEEI(prog_out_toEEI), .data_out_toEEI(data_out_toEEI), .rs1_toEEI(
    ↪ rs1_toEEI), .rs2_toEEI(rs2_toEEI), .PC(PC)
235 );
236 generic_register #(.width(32)) PC_in_reg(
237     .clk(clk), .reset(rst_PC), .load(set_PC), .data_in(PC+32'b100),
238     .data_out(PC_reg)
239 );
240 generic_register #(.width(16)) first_input_half_reg(
241     .clk(clk), .reset(1'b0), .load(set_first_input_half), .data_in(in),
242     .data_out(first_input_half)
243 );
244 generic_register #(.width(16)) second_input_half_reg(
245     .clk(clk), .reset(1'b0), .load(set_second_input_half), .data_in(in),
246     .data_out(second_input_half)
247 );
248 debounce rst_debounce (.clk(clk), .signal_in(~rst_asin), .signal_out(rst));
249 debounce start_debounce (.clk(clk), .signal_in(start_asin), .signal_out(start));
250 debounce resume_debounce(.clk(clk), .signal_in(resume_asin), .signal_out(resume));
251 full_display full_display_unit(
252     .clk(clk), .enable( (input_indicator[1]^input_indicator[0]) | enable_info_out ),
253     .show_options(input_indicator), .in( (input_indicator[1]^input_indicator[0]) ? {in, in}
    ↪ : info_out ),
254     .out(display)
255 );
256 endmodule
257
258 module generic_register #(parameter width=32)(
259     input clk, reset, load,
260     input [width-1:0] data_in,
261     output reg[width-1:0] data_out
262 );
263 always_ff @(negedge clk) data_out <= load ? data_in : (reset ? 0 : data_out);
264 endmodule
265
266 module debounce(input clk, signal_in, output reg signal_out);
267     typedef enum {
268         waiting_state, counter_state, hold_state, response_state
269     } state_type;
270     state_type actual_state, next_state;
271     reg signal_in_sync, start_counter, counter_ready;
272     reg [15:0] counter;
273     always_ff @(posedge clk) {signal_in_sync, actual_state} <= {signal_in, next_state}; //
    ↪ Señal de entrada sincronizada y actualización de estado
274     always_ff @(posedge clk) // debounce counter
275         if(start_counter) {counter, counter_ready} <= {counter + 16'b1, (counter == 16'
    ↪ hFFFF) ? 1'b1 : 1'b0};
276         else {counter, counter_ready} <= 17'b0;
277     always_comb // FSM controller
278         case (actual_state)
279             default: begin

```

```

280     {start_counter, signal_out} = 2'b00;
281     next_state = actual_state;
282     end
283     waiting_state: begin
284         {start_counter, signal_out} = 2'b00;
285         next_state = signal_in_sync ? counter_state : waiting_state;
286     end
287     counter_state: begin
288         {start_counter, signal_out} = 2'b10;
289         next_state = signal_in_sync ? (counter_ready ? hold_state : counter_state) :
↪ waiting_state;
290     end
291     hold_state: begin
292         {start_counter, signal_out} = 2'b00;
293         next_state = signal_in_sync ? hold_state : response_state;
294     end
295     response_state: begin
296         {start_counter, signal_out} = 2'b01;
297         next_state = waiting_state;
298     end
299     endcase
300 endmodule
301
302 module full_display(
303     input clk, enable,
304     input [1:0] show_options,
305     input [31:0] in,
306     output [15:0] out // enable7, enable6, enable5, enable4, enable3, enable2, enable1, enable0
↪ , CA, CB, CC, CD, CE, CF, CG, DP
307 );
308 /*
309     show_options =
310         00 o 11 -> full: 8 bytes
311         01     -> half: first 4 bytes
312         10     -> half: last 4 bytes
313 */
314 reg [2:0] select;
315 reg [15:0] counter;
316 wire show_first_half, show_last_half;
317 wire [7:0] character0, character1, character2, character3, character4, character5, character6
↪ , character7;
318 assign {show_first_half, show_last_half} = {(show_options==2'b10) ? 1'b0 : 1'b1, (
↪ show_options==2'b01) ? 1'b0 : 1'b1};
319 assign out = enable ?
320     ((select==3'h0) ? {show_first_half ? 8'b11111110 : 8'hFF, character0} :
321     ((select==3'h1) ? {show_first_half ? 8'b111111101 : 8'hFF, character1} :
322     ((select==3'h2) ? {show_first_half ? 8'b11111011 : 8'hFF, character2} :
323     ((select==3'h3) ? {show_first_half ? 8'b11110111 : 8'hFF, character3} :
324     ((select==3'h4) ? {show_last_half ? 8'b11101111 : 8'hFF, character4} :
325     ((select==3'h5) ? {show_last_half ? 8'b11011111 : 8'hFF, character5} :
326     ((select==3'h6) ? {show_last_half ? 8'b10111111 : 8'hFF, character6} :
327     ((select==3'h7) ? {show_last_half ? 8'b01111111 : 8'hFF, character7} :

```

```

328         16'hFFFF))))))))) : 16'hFFFF;
329 always_ff @(posedge clk)
330     if(enable) begin
331         if(counter==16'hFFFF) {select, counter} = { unsigned'(select)+3'b1, 16'b0 };
332         else {select, counter} = { select, unsigned'(counter)+16'b1 };
333     end
334     else {select, counter} = 0;
335 display_character display_character_unit0(
336     .in(in[ 3: 0]), .out(character0) // CA, CB, CC, CD, CE, CF, CG, DP
337 );
338 display_character display_character_unit1(
339     .in(in[ 7: 4]), .out(character1) // CA, CB, CC, CD, CE, CF, CG, DP
340 );
341 display_character display_character_unit2(
342     .in(in[11: 8]), .out(character2) // CA, CB, CC, CD, CE, CF, CG, DP
343 );
344 display_character display_character_unit3(
345     .in(in[15:12]), .out(character3) // CA, CB, CC, CD, CE, CF, CG, DP
346 );
347 display_character display_character_unit4(
348     .in(in[19:16]), .out(character4) // CA, CB, CC, CD, CE, CF, CG, DP
349 );
350 display_character display_character_unit5(
351     .in(in[23:20]), .out(character5) // CA, CB, CC, CD, CE, CF, CG, DP
352 );
353 display_character display_character_unit6(
354     .in(in[27:24]), .out(character6) // CA, CB, CC, CD, CE, CF, CG, DP
355 );
356 display_character display_character_unit7(
357     .in(in[31:28]), .out(character7) // CA, CB, CC, CD, CE, CF, CG, DP
358 );
359 endmodule
360
361 module display_character(
362     input [3:0] in,
363     output [7:0] out // CA, CB, CC, CD, CE, CF, CG, DP
364 ); // Dot On (0 lógico) => letra | Dot Off (1 lógico) => número
365 assign out = (in==4'h0) ? 8'b00000011 : // 0
366             ((in==4'h1) ? 8'b10011111 : // 1
367             ((in==4'h2) ? 8'b00100101 : // 2
368             ((in==4'h3) ? 8'b00001101 : // 3
369             ((in==4'h4) ? 8'b10011001 : // 4
370             ((in==4'h5) ? 8'b01001001 : // 5
371             ((in==4'h6) ? 8'b01000001 : // 6
372             ((in==4'h7) ? 8'b00011111 : // 7
373             ((in==4'h8) ? 8'b00000001 : // 8
374             ((in==4'h9) ? 8'b00001001 : // 9
375             ((in==4'hA) ? 8'b00010000 : // A
376             ((in==4'hB) ? 8'b00000000 : // B
377             ((in==4'hC) ? 8'b01100010 : // C
378             ((in==4'hD) ? 8'b00000010 : // D
379             ((in==4'hE) ? 8'b01100000 : // E

```

```

380         ((in==4'hF) ? 8'b01110000 : // F
381         7'b11111111))))))))))))); // default
382 endmodule

```

riscv32imf_singlecycle.sv: archivo *top* del *core complex Single-Cycle*.

```

1  /*
2  autor: Gianluca Vincenzo D'Agostino Matute
3  Santiago de Chile, Septiembre 2021
4  */
5  `timescale 1ns / 1ps
6
7  module riscv32imf_singlecycle(
8      input clk, rst, start, acces_to_registers_files, is_wb_data_fp_fromEEI, do_wb_fromEEI
9          ↪ , is_rs1_fp_fromEEI, is_rs2_fp_fromEEI,
10     input acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
11         ↪ prog_is_load_unsigned_fromEEI,
12     input acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
13         ↪ data_is_load_unsigned_fromEEI,
14     input [31:0] initial_PC, prog_addr_fromEEI, prog_in_fromEEI, data_addr_fromEEI,
15         ↪ data_in_fromEEI, wb_data_fromEEI,
16     input [1:0] prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI,
17     input [4:0] rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI,
18     output prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
19         ↪ data_out_of_range_toEEI,
20     output reg ready,
21     output reg [1:0] exit_status,
22     output [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI, PC
23     /*
24         exit_status[1:0] =
25             00 -> ecall
26             11 -> ebreak
27             01 -> program out of range or not valid inst
28             10 -> memory out of range
29     */
30     );
31     typedef enum {
32         waiting_state,
33         initial_state,
34         program_loop_main,
35         program_loop_fpu,
36         program_loop_mem,
37         program_loop_new_pc_wb,
38         program_loop_read_inst,
39         final_state
40     } state_type;
41     state_type actual_state, next_state;
42     // Controller signals
43     reg is_the_beginning, valid_prog_mem, valid_data_mem, start_fpu, set_pc, set_inst,
44         ↪ do_wb;
45     wire ready_fpu, ready_prog_mem, out_of_range_prog_mem, ready_data_mem,
46         ↪ out_of_range_data_mem, is_ecall, is_ebreak, acces_to_fpu, acces_to_mem,

```

```

    ↪ not_valid;
40 // Datapath signals
41 wire is_imm_valid, branch_condition, rw_data_mem, is_load_unsigned_data_mem;
42 wire [4:0] rs1_add, rs2_add, rs3_add, rd_add, alu_option, fpu_option, ignored_flags;
43 wire [3:0] funct7_out;
44 wire [2:0] format_type, sub_format_type, funct3, rm2fpu;
45 wire [1:0] reg_access_option, byte_half_word_data_mem, funct2;
46 wire [31:0] jalr_pc, inst, inst2reg, PC2reg, imm, rs2, rs3, rd, operand1, operand2,
    ↪ operand3, res_alu, res_fpu, EX_out, data_out_data_mem, next_PC;
47 // interface con EEI
48 assign {prog_ready_toEEI, prog_out_of_range_toEEI, prog_out_toEEI} = {
    ↪ ready_prog_mem, out_of_range_prog_mem, inst2reg};
49 assign {data_ready_toEEI, data_out_of_range_toEEI, data_out_toEEI} = {
    ↪ ready_data_mem, out_of_range_data_mem, data_out_data_mem};
50 assign {rs1_toEEI, rs2_toEEI} = {operand1, rs2};
51 // PC predictor
52 assign PC2reg = is_the_beginning ? initial_PC : next_PC;
53 // mux for operand2
54 assign operand2 = is_imm_valid ? imm : rs2;
55 // mux for operand3
56 assign operand3 = (is_imm_valid & (format_type==3'b011 | format_type==3'b100)) ?
    ↪ rs2 : rs3;
57 // data_mem_options_selection
58 assign rw_data_mem = (format_type==3'b011) ? 1'b1 : 1'b0; // only 1 in S type
    ↪ instruction
59 assign is_load_unsigned_data_mem = (format_type==3'b010 & funct3[2:1]==2'b10) ?
    ↪ 1'b1 : 1'b0; // only 1 in I type instruction and funct3= 4 o 5
60 assign byte_half_word_data_mem = (funct3[1:0]==2'b00) ? 2'b10 : // funct3= 0 o 4 -
    ↪ byte
61
        ((funct3[1:0]==2'b01) ? 2'b01 : // funct3= 1 o 5 -half
62
            2'b00); // funct3= 2 -word (default)
63 // next_PC_selection
64 assign jalr_pc = operand1+operand2;
65 assign next_PC = ({format_type, branch_condition}==4'b100_1 | format_type==3'
    ↪ b110) ? PC+operand2 : // B | J
66
        (({format_type, sub_format_type}==6'b010_011) ? {jalr_pc
67
            ↪ [31:1], 1'b0} : // I (jalr)
        PC+32'b100); //
    ↪ Deafault
68 // final_alu_fpu_res_selection
69 assign EX_out = (({format_type, sub_format_type}==6'b000_000 & {funct7_out,
    ↪ funct3}!=7'b1011_000 & {funct7_out, funct3}!=7'b1100_000) | // se excluyen fmv.x
    ↪ .w y fmv.w.x
70
        format_type==3'b001) ? res_fpu : // R
    ↪ (fp) o R4
71
        (({format_type, sub_format_type}==6'b000_000 & (funct7_out==4'b1011
    ↪ | funct7_out==4'b1100)) ? operand1 : // fmv.x.w y fmv.w.x
72
        (({format_type, sub_format_type}==6'b010_011 | format_type==3'b110) ?
    ↪ PC+32'b100 : // I (jalr) o J
73
        ((format_type==3'b101) ? (operand2 + (sub_format_type[0] ? PC : 32'
    ↪ b0) ) : // U
74
            res_alu)); // default

```

```

75 // mux rd
76 assign rd = (format_type==3'b010 & ~sub_format_type[2] & ~sub_format_type[0]) ?
    ↪ data_out_data_mem : EX_out;
77 // sub_decoder
78 assign is_ecall = (format_type==3'b010 & sub_format_type==3'b100 & operand2
    ↪ ==32'b0) ? 1'b1 : 1'b0;
79 assign is_ebreak = (format_type==3'b010 & sub_format_type==3'b100 & operand2
    ↪ ==32'b1) ? 1'b1 : 1'b0;
80 assign acces_to_fpu = (
81     format_type==3'b001 | ( // R4
82     {format_type, sub_format_type}==6'b0 & {funct7_out, funct3}!=7'b1011_000 &
    ↪ {funct7_out, funct3}!=7'b1100_000 ) // R (fp) - {fmv.x.w, fmv.w.x}
83     ) ? 1'b1 : 1'b0;
84 assign acces_to_mem = (format_type==3'b011 | (format_type==3'b010 & ~
    ↪ sub_format_type[2] & ~sub_format_type[0])) ? 1'b1 : 1'b0;
85 assign not_valid = format_type==3'b111 ? 1'b1 : 1'b0;
86 // FSM controller
87 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
88 always_comb
89     case (actual_state)
90     default: begin // default
91         {set_pc, set_inst, do_wb} = 3'b000;
92         {is_the_beginning, start_fpu} = 2'b00;
93         {valid_prog_mem, valid_data_mem} = 2'b00;
94         {ready, exit_status} = 3'b000;
95         next_state = actual_state;
96     end
97     waiting_state: begin // se espera señal de inicio "start"
98         {set_pc, set_inst, do_wb} = 3'b000;
99         {is_the_beginning, start_fpu} = 2'b00;
100        {valid_prog_mem, valid_data_mem} = 2'b00;
101        {ready, exit_status} = 3'b000;
102        next_state = start ? initial_state : waiting_state;
103    end
104    initial_state: begin // se setea PC
105        {set_pc, set_inst, do_wb} = 3'b100;
106        {is_the_beginning, start_fpu} = 2'b10;
107        {valid_prog_mem, valid_data_mem} = 2'b00;
108        {ready, exit_status} = 3'b000;
109        next_state = program_loop_read_inst;
110    end
111    program_loop_read_inst: begin // se lee nueva instrucción
112        {set_pc, set_inst, do_wb} = 3'b010;
113        {is_the_beginning, start_fpu} = 2'b00;
114        {valid_prog_mem, valid_data_mem} = 2'b10;
115        {ready, exit_status} = 3'b000;
116        next_state = out_of_range_prog_mem ? final_state : (ready_prog_mem ?
    ↪ program_loop_main : program_loop_read_inst);
117    end
118    program_loop_main: begin // se setea y decodifica instrucción
119        {set_pc, set_inst, do_wb} = 3'b000;
120        {is_the_beginning, start_fpu} = 2'b00;

```

```

121     {valid_prog_mem, valid_data_mem} = 2'b00;
122     {ready, exit_status}           = 3'b000;
123     next_state = (is_ecall | is_ebreak | not_valid) ? final_state :
124                 (acces_to_fpu ? program_loop_fpu :
125                 (acces_to_mem ? program_loop_mem :
↪ program_loop_new_pc_wb));
126     end
127     program_loop_fpu: begin           // se espera FPU
128         {set_pc, set_inst, do_wb}    = 3'b000;
129         {is_the_beginning, start_fpu} = 2'b01;
130         {valid_prog_mem, valid_data_mem} = 2'b00;
131         {ready, exit_status}         = 3'b000;
132         next_state = ready_fpu ? program_loop_new_pc_wb : program_loop_fpu;
133     end
134     program_loop_mem: begin           // se espera mem
135         {set_pc, set_inst, do_wb}    = 3'b000;
136         {is_the_beginning, start_fpu} = 2'b00;
137         {valid_prog_mem, valid_data_mem} = 2'b01;
138         {ready, exit_status}         = 3'b000;
139         next_state = out_of_range_data_mem ? final_state : (ready_data_mem ?
↪ program_loop_new_pc_wb : program_loop_mem);
140     end
141     program_loop_new_pc_wb: begin     // se setea new PC y realiza wb
142         {set_pc, set_inst, do_wb}    = 3'b101;
143         {is_the_beginning, valid_prog_mem} = 2'b0;
144         start_fpu    = acces_to_fpu ? 1'b1 : 1'b0;
145         valid_data_mem = acces_to_mem ? 1'b1 : 1'b0;
146         {ready, exit_status}         = 3'b000;
147         next_state = program_loop_read_inst;
148     end
149     final_state: begin                // se finaliza ejecución
150         {set_pc, set_inst, do_wb}    = 3'b000;
151         {is_the_beginning, start_fpu} = 2'b00;
152         {valid_prog_mem, valid_data_mem} = 2'b00;
153         ready = 1'b1;
154         exit_status = is_ecall                ? 2'b00 : // 00 -> ecall
155                     ( is_ebreak                ? 2'b11 : // 11 -> ebreak
156                     ( (out_of_range_prog_mem | not_valid) ? 2'b01 : // 01 -> program
↪ out of range or not valid
157                     ( out_of_range_data_mem        ? 2'b10 : // 10 -> memory out
↪ of range
158                     2'b00 ))) );
159         next_state = start ? final_state : waiting_state;
160     end
161 endcase
162 generic_register #(.width(32)) pc_register(
163     .clk(clk), .reset(rst), .load(set_pc), .data_in(PC2reg),
164     .data_out(PC)
165 );
166 generic_register #(.width(32)) inst_register(
167     .clk(clk), .reset(rst), .load(set_inst), .data_in(inst2reg),
168     .data_out(inst)

```

```

169 );
170 // prog_memory
171 memory #(.initial_option(2)) prog_memory_unit(
172     .clk(clk), .rst(1'b0),
173     // inputs
174     .rw(          (aces_to_prog_mem & ~start) ? prog_rw_fromEEI          : 1'b0),
175     .valid(       (aces_to_prog_mem & ~start) ? prog_valid_mem_fromEEI    :
↪ valid_prog_mem),
176     .addr(        (aces_to_prog_mem & ~start) ? prog_addr_fromEEI        : PC),
177     .data_in(     (aces_to_prog_mem & ~start) ? prog_in_fromEEI          : 32'b0
↪ ),
178     .byte_half_word( (aces_to_prog_mem & ~start) ? prog_byte_half_word_fromEEI
↪ : 2'b0),
179     .is_load_unsigned((aces_to_prog_mem & ~start) ?
↪ prog_is_load_unsigned_fromEEI : 1'b1),
180     // outputs
181     .ready(ready_prog_mem), .out_of_range(out_of_range_prog_mem), .data_out(
↪ inst2reg)
182 );
183 // data_memory
184 memory #(.initial_option(1)) data_memory_unit(
185     .clk(clk), .rst(1'b0),
186     // inputs
187     .rw(          (aces_to_data_mem & ~start) ? data_rw_fromEEI          :
↪ rw_data_mem),
188     .valid(       (aces_to_data_mem & ~start) ? data_valid_mem_fromEEI    :
↪ valid_data_mem),
189     .addr(        (aces_to_data_mem & ~start) ? data_addr_fromEEI        :
↪ EX_out),
190     .data_in(     (aces_to_data_mem & ~start) ? data_in_fromEEI          :
↪ operand3),
191     .byte_half_word( (aces_to_data_mem & ~start) ? data_byte_half_word_fromEEI
↪ : byte_half_word_data_mem),
192     .is_load_unsigned((aces_to_data_mem & ~start) ?
↪ data_is_load_unsigned_fromEEI : is_load_unsigned_data_mem),
193     // outputs
194     .ready(ready_data_mem), .out_of_range(out_of_range_data_mem), .data_out(
↪ data_out_data_mem)
195 );
196 // Instruction_Decoder
197 Instruction_Decoder Instruction_Decoder_unit(
198     .in(inst),
199     .is_imm_valid(is_imm_valid), .imm(imm),
200     .rd_add(rd_add), .rs1_add(rs1_add), .rs2_add(rs2_add), .rs3_add(rs3_add),
201     .funct7_out(funct7_out), .format_type(format_type), .sub_format_type(
↪ sub_format_type),
202     .funct3(funct3), .funct2(funct2), .reg_access_option(reg_access_option)
203 );
204 // registers_files
205 registers_files registers_files_unit(
206     .clk(clk),
207     // inputs

```

```

208     .rs1_add( (acces_to_registers_files & ~start) ? rs1_add_fromEEI : rs1_add),
209     .rs2_add( (acces_to_registers_files & ~start) ? rs2_add_fromEEI : rs2_add),
210     .rs3_add(rs3_add),
211     .wb_add( (acces_to_registers_files & ~start) ? wb_add_fromEEI : rd_add),
212     .wb_data( (acces_to_registers_files & ~start) ? wb_data_fromEEI : rd),
213     .write_reg( (acces_to_registers_files & ~start) ? do_wb_fromEEI :
214         ((format_type==3'b011 | format_type==3'b100) ? 1'b0 : do_wb) ),
215     .is_wb_data_fp( (acces_to_registers_files & ~start) ? is_wb_data_fp_fromEEI :
↪ reg_access_option[0]),
216     .is_rs1_fp( (acces_to_registers_files & ~start) ? is_rs1_fp_fromEEI :
↪ reg_access_option[1]),
217     .is_rs2_fp( (acces_to_registers_files & ~start) ? is_rs2_fp_fromEEI : (
↪ reg_access_option[1] | reg_access_option[0])),
218     // outputs
219     .rs1(operand1), .rs2(rs2), .rs3(rs3)
220 );
221 // alu_op_selection
222 alu_op_selection alu_op_selection_unit(
223     .imm_11_5(operand2[11:5]), .funct7_out(funct7_out),
224     .format_type(format_type), .sub_format_type(sub_format_type), .funct3(funct3),
225     .alu_option(alu_option)
226 );
227 // ALU
228 ALU ALU_unit(
229     .in1(operand1), .in2( (format_type==3'b100) ? operand3 : operand2), // is B type?
230     .operation(alu_option),
231     .res(res_alu), .boolean_res(branch_condition)
232 );
233 // fpu_op_selection
234 fpu_op_selection fpu_op_selection_unit(
235     .rs2_add(rs2_add), .funct7_out(funct7_out),
236     .format_type(format_type), .sub_format_type(sub_format_type), .funct3(funct3), .
↪ rm_from_fcsr(3'b000),
237     .rm2fpu(rm2fpu), .fpu_option(fpu_option)
238 );
239 // FPU
240 FPU FPU_unit(
241     .start(start_fpu), .rst(rst), .clk(clk), .rm(rm2fpu),
242     .option(fpu_option), .in1(operand1), .in2(operand2), .in3(operand3),
243     .NV(ignored_flags[4]), .NX(ignored_flags[3]), .UF(ignored_flags[2]), .OF(ignored_flags
↪ [1]), .DZ(ignored_flags[0]),
244     .ready(ready_fpu), .out(res_fpu)
245 );
246 endmodule

```

riscv32imf_pipeline.sv: archivo *top* del *core complex Pipelined*.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps

```

```

6
7 module riscv32imf_pipeline(
8     input clk, rst, start, acces_to_registers_files, is_wb_data_fp_fromEEI, do_wb_fromEEI
9         ↪ , is_rs1_fp_fromEEI, is_rs2_fp_fromEEI,
10    input acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
11        ↪ prog_is_load_unsigned_fromEEI,
12    input acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
13        ↪ data_is_load_unsigned_fromEEI,
14    input [31:0] initial_PC, prog_addr_fromEEI, prog_in_fromEEI, data_addr_fromEEI,
15        ↪ data_in_fromEEI, wb_data_fromEEI,
16    input [1:0] prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI,
17    input [4:0] rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI,
18    output prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
19        ↪ data_out_of_range_toEEI,
20    output reg ready,
21    output [1:0] exit_status,
22    output [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI, PC
23    /*
24        exit_status[1:0] =
25            00 -> ecall
26            11 -> ebreak
27            01 -> program out of range
28            10 -> memory out of range
29    */
30    );
31    typedef enum {waiting_state, loop_state_1, loop_state_2_normal,
32        ↪ loop_state_2_chazard, loop_state_3, prog_error_state, final_state} state_type;
33    state_type actual_state, next_state;
34    reg set_regs, is_the_beginning, enable_execution, not_valid_EX, not_valid_MEM,
35        ↪ not_valid_WB, control_bubble, acces_to_fpu_EX,
36        acces_to_mem_EX, acces_to_mem_MEM, is_imm_valid_EX, is_ecall_EX,
37        ↪ is_ecall_MEM, is_ebreak_EX, is_ebreak_MEM, is_ecall_WB, is_ebreak_WB;
38    reg [4:0] rd_add_EX, rd_add_MEM, rd_add_WB, rs1_add_EX, rs2_add_EX,
39        ↪ rs3_add_EX;
40    reg [3:0] funct7_out_EX;
41    reg [2:0] format_type_MEM, sub_format_type_MEM, format_type_EX,
42        ↪ sub_format_type_EX, format_type_WB, funct3_EX, funct3_MEM;
43    reg [1:0] reg_access_option_EX, reg_access_option_MEM, reg_access_option_WB;
44    reg [31:0] new_PC, PC_IF, PC_DR, PC_EX, PC_MEM, PC_WB, inst_DR, imm_EX,
45        ↪ rs1_EX, rs2_EX, rs3_EX, final_res_MEM, operand3_MEM, rd_WB;
46    wire is_ecall_DR, is_ebreak_DR, acces_to_fpu_DR, acces_to_mem_DR,
47        ↪ not_valid_DR, is_imm_valid_DR, ready_fpu, enable_fpu, exit, done,
48        ready_prog_mem, out_of_range_prog_mem, ready_data_mem,
49        ↪ out_of_range_data_mem, rw_data_mem, is_load_unsigned_data_mem,
50        ↪ branch_condition, control_hazard;
51    wire [6:0] ignored_signals;
52    wire [4:0] rd_add_DR, rs1_add_DR, rs2_add_DR, rs3_add_DR, alu_option,
53        ↪ fpu_option;
54    wire [3:0] funct7_out_DR;
55    wire [2:0] format_type_DR, sub_format_type_DR, funct3_DR, rm2fpu;
56    wire [1:0] reg_access_option_DR, byte_half_word_data_mem;
57    wire [31:0] jalr_pc, branch_PC, rs1_DR, rs2_DR, rs3_DR, imm_DR, rd_MEM,

```

```

↪ operand1_EX, operand2_EX, operand3_EX, final_res_EX, inst_IF, res_alu,
↪ res_fpu, data_out_data_mem;
43 // interface con EEI
44 assign {prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
↪ data_out_of_range_toEEI, prog_out_toEEI, data_out_toEEI, rs1_toEEI,
↪ rs2_toEEI} =
45 {ready_prog_mem, out_of_range_prog_mem, ready_data_mem,
↪ out_of_range_data_mem, inst_IF, data_out_data_mem, rs1_DR,
↪ rs2_DR};
46 assign PC = (exit_status[1]^exit_status[0]) ? (exit_status[0] ? PC_IF : PC_MEM) :
↪ PC_WB;
47 // exit_status_selection
48 assign exit_status = is_ecall_WB ? 2'b00 : (
49 is_ebreak_WB ? 2'b11 : (
50 (acces_to_mem_MEM & out_of_range_data_mem) ? 2'b10 : (
51 (out_of_range_prog_mem & ~control_hazard) ? 2'b01 : (
52 2'b00)))));
53 // data_mem_options_selection
54 assign rw_data_mem = (format_type_MEM==3'b011) ? 1'b1 : 1'b0; // only 1 in S type
↪ instruction
55 assign is_load_unsigned_data_mem = (format_type_MEM==3'b010 & funct3_MEM
↪ [2:1]==2'b10) ? 1'b1 : 1'b0; // only 1 in I type instruction and funct3= 4 o 5
56 assign byte_half_word_data_mem = (funct3_MEM[1:0]==2'b00) ? 2'b10 : // funct3= 0
↪ o 4 -byte
57 ((funct3_MEM[1:0]==2'b01) ? 2'b01 : // funct3= 1 o 5 -half
58 2'b00); // funct3= 2 -word (default)
59 // final_alu_fpu_res_selection
60 assign final_res_EX =
61 (({format_type_EX, sub_format_type_EX}==6'b000_000 & {funct7_out_EX,
↪ funct3_EX}!=7'b1011_000 & {funct7_out_EX, funct3_EX}!=7'b1100_000) | // se
↪ excluyen fmv.x.w y fmv.w.x
62 format_type_EX==3'b001) ? res_fpu : // R (fp) o R4
63 (({format_type_EX, sub_format_type_EX}==6'b000_000 & (funct7_out_EX==4'
↪ b1011 | funct7_out_EX==4'b1100)) ? operand1_EX : // fmv.x.w y fmv.w.x
64 (({format_type_EX, sub_format_type_EX}==6'b010_011 | format_type_EX==3'
↪ b110) ? PC_EX+32'b100 : // I (jalr) o J
65 ((format_type_EX==3'b101) ? (operand2_EX + (sub_format_type_EX[0] ?
↪ PC_EX : 32'b0) ) : // U
66 res_alu)); // default // default
67 // mux rd
68 assign rd_MEM = (format_type_MEM==3'b010 & ~sub_format_type_MEM[2] & ~
↪ sub_format_type_MEM[0]) ? data_out_data_mem : final_res_MEM;
69 // sub_decoder
70 assign is_ecall_DR = (format_type_DR==3'b010 & sub_format_type_DR==3'b100 &
↪ imm_DR==32'b0) ? 1'b1 : 1'b0;
71 assign is_ebreak_DR = (format_type_DR==3'b010 & sub_format_type_DR==3'b100
↪ & imm_DR==32'b1) ? 1'b1 : 1'b0;
72 assign acces_to_fpu_DR = (
73 format_type_DR==3'b001 | ( // R4
74 {format_type_DR, sub_format_type_DR}==6'b0 & {funct7_out_DR,
↪ funct3_DR}!=7'b1011_000 & {funct7_out_DR, funct3_DR}!=7'b1100_000 ) // R (
↪ fp) - {fmv.x.w, fmv.w.x}

```

```

75     ) ? 1'b1 : 1'b0;
76 assign acces_to_mem_DR = (format_type_DR==3'b011 | (format_type_DR==3'b010
    ↪ & ~sub_format_type_DR[2] & ~sub_format_type_DR[0])) ? 1'b1 : 1'b0;
77 assign not_valid_DR = format_type_DR==3'b111 ? 1'b1 : 1'b0;
78 // next_PC_selection
79 assign control_hazard = ( ({format_type_EX, branch_condition}==4'b100_1 |
    ↪ format_type_EX==3'b110) | ({format_type_EX, sub_format_type_EX}==6'
    ↪ b010_011) ) ? 1'b1 : 1'b0;
80 assign jalr_pc = operand1_EX+operand2_EX;
81 assign branch_PC = ({format_type_EX, branch_condition}==4'b100_1 |
    ↪ format_type_EX==3'b110) ? PC_EX+operand2_EX : // B | J
82     (({format_type_EX, sub_format_type_EX}==6'b010_011)
    ↪ {jalr_pc[31:1], 1'b0} : // I (jalr)
83     32'b0);
84 // control_signals
85 assign exit = (is_ecall_WB | is_ebreak_WB |
86     (out_of_range_prog_mem & ~control_hazard) |
87     (acces_to_mem_MEM & out_of_range_data_mem) ) ? 1'b1 : 1'b0;
88 assign done = (((ready_prog_mem & ~out_of_range_prog_mem) | control_hazard) &
89     ((acces_to_fpu_EX & ready_fpu) | ~acces_to_fpu_EX) &
90     ((acces_to_mem_MEM & ready_data_mem & ~out_of_range_data_mem) |
    ↪ ~acces_to_mem_MEM)) ? 1'b1 : 1'b0;
91 // FSM controller
92 always_comb
93     case (actual_state)
94     default: begin
95         {is_the_beginning, ready} = 2'b00;
96         {enable_execution, control_bubble, set_regs} = 3'b000;
97         next_state = actual_state;
98     end
99     waiting_state: begin
100        {is_the_beginning, ready} = {start, 1'b0};
101        {enable_execution, control_bubble, set_regs} = 3'b000;
102        next_state = start ? loop_state_1 : waiting_state;
103    end
104    loop_state_1: begin
105        {is_the_beginning, ready} = 2'b00;
106        {enable_execution, control_bubble, set_regs} = 3'b100;
107        next_state = start ? (exit ? final_state :
108            (done ? (control_hazard ? loop_state_2_chazard : loop_state_2_normal)
109                : loop_state_1)) : waiting_state;
110    end
111    loop_state_2_normal: begin
112        {is_the_beginning, ready} = 2'b00;
113        {enable_execution, control_bubble, set_regs} = 3'b101;
114        next_state = loop_state_3;
115    end
116    loop_state_2_chazard: begin
117        {is_the_beginning, ready} = 2'b00;
118        {enable_execution, control_bubble, set_regs} = 3'b111;
119        next_state = loop_state_3;
120    end

```

```

121     loop_state_3: begin
122         {is_the_beginning, ready}           = 2'b00;
123         {enable_execution, control_bubble, set_regs} = 3'b000;
124         next_state = loop_state_1;
125     end
126     final_state: begin
127         {is_the_beginning, ready}           = 2'b01;
128         {enable_execution, control_bubble, set_regs} = 3'b100;
129         next_state = start ? final_state : waiting_state;;
130     end
131 endcase
132 always_ff @(posedge(clk)) begin // el orden SI importa
133     new_PC = is_the_beginning ? initial_PC : (set_regs ? (control_bubble ? branch_PC
134     ↪ : PC_IF+32'b100) : PC_IF); // PC_generator
135     if(is_the_beginning) begin // reset
136         // PipelineReg_MEM2WB
137         {PC_WB, rd_WB, rd_add_WB, format_type_WB, reg_access_option_WB,
138     ↪ not_valid_WB, is_ebreak_WB, is_ecall_WB } = 0;
139         // PipelineReg_EX2MEM
140         {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM, format_type_MEM
141     ↪ , sub_format_type_MEM, funct3_MEM,
142         reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM,
143     ↪ is_ebreak_MEM, is_ecall_MEM} = 0;
144     end
145     else if(set_regs) begin // set
146         // PipelineReg_MEM2WB
147         {PC_WB, rd_WB, rd_add_WB, format_type_WB, reg_access_option_WB,
148     ↪ not_valid_WB, is_ebreak_WB, is_ecall_WB }
149         =
150         {PC_MEM, rd_MEM, rd_add_MEM, format_type_MEM,
151     ↪ reg_access_option_MEM, not_valid_MEM, is_ebreak_MEM, is_ecall_MEM};
152         // PipelineReg_EX2MEM
153         {PC_MEM, final_res_MEM, operand3_MEM, rd_add_MEM, format_type_MEM
154     ↪ , sub_format_type_MEM, funct3_MEM,
155         reg_access_option_MEM, acces_to_mem_MEM, not_valid_MEM,
156     ↪ is_ebreak_MEM, is_ecall_MEM}
157         =
158         {PC_EX, final_res_EX, operand3_EX, rd_add_EX, format_type_EX,
159     ↪ sub_format_type_EX, funct3_EX,
160         reg_access_option_EX, acces_to_mem_EX, not_valid_EX, is_ebreak_EX,
161     ↪ is_ecall_EX };
162     end
163     if((control_bubble & set_regs) | is_the_beginning) begin // reset (c. hazard)
164         // PipelineReg_DR2EX
165         {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX, rs1_add_EX,
166     ↪ rs2_add_EX, rs3_add_EX,
167         funct7_out_EX, funct3_EX, format_type_EX, sub_format_type_EX,
168     ↪ reg_access_option_EX,
169         is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX, not_valid_EX,
170     ↪ is_ebreak_EX, is_ecall_EX} = 0;
171         // PipelineReg_IF2DR
172         {PC_DR, inst_DR} = 0;

```

```

160     end
161     else if(~control_bubble & set_regs) begin // set (c. hazard)
162         // PipelineReg_DR2EX
163         {PC_EX, imm_EX, rs1_EX, rs2_EX, rs3_EX, rd_add_EX, rs1_add_EX,
164 ↪ rs2_add_EX, rs3_add_EX,
165         funct7_out_EX, funct3_EX, format_type_EX, sub_format_type_EX,
166 ↪ reg_access_option_EX,
167         is_imm_valid_EX, acces_to_fpu_EX, acces_to_mem_EX, not_valid_EX,
168 ↪ is_ebreak_EX, is_ecall_EX}
169         =
170         {PC_DR, imm_DR, rs1_DR, rs2_DR, rs3_DR, rd_add_DR, rs1_add_DR,
171 ↪ rs2_add_DR, rs3_add_DR,
172         funct7_out_DR, funct3_DR, format_type_DR, sub_format_type_DR,
173 ↪ reg_access_option_DR,
174         is_imm_valid_DR, acces_to_fpu_DR, acces_to_mem_DR, not_valid_DR,
175 ↪ is_ebreak_DR, is_ecall_DR};
176         // PipelineReg_IF2DR
177         {PC_DR, inst_DR} = {PC_IF, inst_IF};
178     end
179     if(is_the_beginning | set_regs) PC_IF = new_PC; // PC_generator
180     actual_state = rst ? waiting_state : next_state;
181 end
182 // prog_memory
183 memory #(initial_option(2)) prog_memory_unit(
184     .clk(clk), .rst(rst),
185     // inputs
186     .rw(          (acces_to_prog_mem & ~start) ? prog_rw_fromEEI          : 1'b0),
187     .valid(       (acces_to_prog_mem & ~start) ? prog_valid_mem_fromEEI    :
188 ↪ enable_execution),
189     .addr(        (acces_to_prog_mem & ~start) ? prog_addr_fromEEI        :
190 ↪ PC_IF),
191     .data_in(     (acces_to_prog_mem & ~start) ? prog_in_fromEEI          : 32'b0
192 ↪ ),
193     .byte_half_word( (acces_to_prog_mem & ~start) ? prog_byte_half_word_fromEEI
194 ↪ : 2'b0),
195     .is_load_unsigned((acces_to_prog_mem & ~start) ?
196 ↪ prog_is_load_unsigned_fromEEI : 1'b1),
197     // outputs
198     .ready(ready_prog_mem), .out_of_range(out_of_range_prog_mem), .data_out(
199 ↪ inst_IF)
200 );
201 // data_memory
202 memory #(initial_option(1)) data_memory_unit(
203     .clk(clk), .rst(rst),
204     // inputs
205     .rw(          (acces_to_data_mem & ~start) ? data_rw_fromEEI          :
206 ↪ rw_data_mem),
207     .valid(       (acces_to_data_mem & ~start) ? data_valid_mem_fromEEI    :
208 ↪ acces_to_mem_MEM & enable_execution),
209     .addr(        (acces_to_data_mem & ~start) ? data_addr_fromEEI        :
210 ↪ final_res_MEM),
211     .data_in(     (acces_to_data_mem & ~start) ? data_in_fromEEI          :

```

```

↪ operand3_MEM),
197   .byte_half_word( (aces_to_data_mem & ~start) ? data_byte_half_word_fromEEI
↪   : byte_half_word_data_mem),
198   .is_load_unsigned((aces_to_data_mem & ~start) ?
↪   data_is_load_unsigned_fromEEI : is_load_unsigned_data_mem),
199   // outputs
200   .ready(ready_data_mem), .out_of_range(out_of_range_data_mem), .data_out(
↪   data_out_data_mem)
201 );
202 // Instruction_Decoder
203 Instruction_Decoder Instruction_Decoder_unit(
204   .in(inst_DR),
205   .is_imm_valid(is_imm_valid_DR), .imm(imm_DR),
206   .rd_add(rd_add_DR), .rs1_add(rs1_add_DR), .rs2_add(rs2_add_DR), .rs3_add(
↪   rs3_add_DR),
207   .funct7_out(funct7_out_DR), .format_type(format_type_DR), .sub_format_type(
↪   sub_format_type_DR),
208   .funct3(funct3_DR), .funct2(ignored_signals[6:5]), .reg_access_option(
↪   reg_access_option_DR)
209 );
210 // registers_files
211 registers_files registers_files_unit(
212   .clk(clk),
213   // inputs
214   .rs1_add( (aces_to_registers_files & ~start) ? rs1_add_fromEEI : rs1_add_DR),
215   .rs2_add( (aces_to_registers_files & ~start) ? rs2_add_fromEEI : rs2_add_DR),
216   .rs3_add( rs3_add_DR ),
217   .wb_add( (aces_to_registers_files & ~start) ? wb_add_fromEEI : rd_add_WB),
218   .wb_data( (aces_to_registers_files & ~start) ? wb_data_fromEEI : rd_WB),
219   .write_reg( (aces_to_registers_files & ~start) ? do_wb_fromEEI :
220     ((format_type_WB==3'b011 | format_type_WB==3'b100 | not_valid_WB |
↪   is_ecall_WB | is_ebreak_WB) ? 1'b0 : 1'b1) ),
221   .is_wb_data_fp( (aces_to_registers_files & ~start) ? is_wb_data_fp_fromEEI :
↪   reg_access_option_WB[0]),
222   .is_rs1_fp( (aces_to_registers_files & ~start) ? is_rs1_fp_fromEEI :
↪   reg_access_option_DR[1]),
223   .is_rs2_fp( (aces_to_registers_files & ~start) ? is_rs2_fp_fromEEI : (
↪   reg_access_option_DR[1] | reg_access_option_DR[0])),
224   // outputs
225   .rs1(rs1_DR), .rs2(rs2_DR), .rs3(rs3_DR)
226 );
227 // forwarding_unit
228 forwarding_unit forwarding_unit(
229   .is_imm_valid_EX(is_imm_valid_EX), .ready_data_mem(ready_data_mem), .
↪   valid_data_mem(aces_to_mem_MEM),
230   .reg_access_option_EX(reg_access_option_EX), .reg_access_option_MEM(
↪   reg_access_option_MEM), .reg_access_option_WB(reg_access_option_WB),
231   .format_type_EX(format_type_EX), .format_type_MEM(format_type_MEM), .
↪   format_type_WB(format_type_WB), .sub_format_type_MEM(
↪   sub_format_type_MEM),
232   .rs1_add_EX(rs1_add_EX), .rs2_add_EX(rs2_add_EX), .rs3_add_EX(rs3_add_EX
↪   ), .rd_add_MEM(rd_add_MEM), .rd_add_WB(rd_add_WB),

```

```

233     .rs1_EX(rs1_EX), .rs2_EX(rs2_EX), .rs3_EX(rs3_EX), .rd_MEM(rd_MEM), .
↪ rd_WB(rd_WB), .imm_EX(imm_EX),
234     .enable_fpu(enable_fpu), .operand1_EX(operand1_EX), .operand2_EX(operand2_EX
↪ ), .operand3_EX(operand3_EX)
235 );
236 // alu_op_selection
237 alu_op_selection alu_op_selection_unit(
238     .imm_11_5(operand2_EX[11:5]), .funct7_out(funct7_out_EX),
239     .format_type(format_type_EX), .sub_format_type(sub_format_type_EX), .funct3(
↪ funct3_EX),
240     .alu_option(alu_option)
241 );
242 // ALU
243 ALU ALU_unit(
244     .in1(operand1_EX), .in2((format_type_EX==3'b100) ? operand3_EX : operand2_EX
↪ ), // is B type?
245     .operation(alu_option),
246     .res(res_alu), .boolean_res(branch_condition)
247 );
248 // fpu_op_selection
249 fpu_op_selection fpu_op_selection_unit(
250     .rs2_add(rs2_add_EX), .funct7_out(funct7_out_EX),
251     .format_type(format_type_EX), .sub_format_type(sub_format_type_EX), .funct3(
↪ funct3_EX), .rm_from_fcsr(3'b000),
252     .rm2fpu(rm2fpu), .fpu_option(fpu_option)
253 );
254 // FPU
255 FPU FPU_unit(
256     .start(access_to_fpu_EX & enable_fpu & enable_execution), .rst(rst | ~
↪ enable_execution), .clk(clk), .rm(rm2fpu),
257     .option(fpu_option), .in1(operand1_EX), .in2(operand2_EX), .in3(operand3_EX),
258     .NV(ignored_signals[4]), .NX(ignored_signals[3]), .UF(ignored_signals[2]), .OF(
↪ ignored_signals[1]), .DZ(ignored_signals[0]),
259     .ready(ready_fpu), .out(res_fpu)
260 );
261 endmodule
262
263 module forwarding_unit(
264     input is_imm_valid_EX, ready_data_mem, valid_data_mem,
265     input [1:0] reg_access_option_EX, reg_access_option_MEM, reg_access_option_WB,
266     input [2:0] format_type_EX, format_type_MEM, format_type_WB,
↪ sub_format_type_MEM,
267     input [4:0] rs1_add_EX, rs2_add_EX, rs3_add_EX, rd_add_MEM, rd_add_WB,
268     input [31:0] rs1_EX, rs2_EX, rs3_EX, rd_MEM, rd_WB, imm_EX,
269     output enable_fpu,
270     output [31:0] operand1_EX, operand2_EX, operand3_EX
271 );
272 /*
273     reg_access_option:
274     |code| rd | rs1 | rs2 |
275     | 00 | I | I  | I  |
276     | 01 | FP | I  | FP |

```

```

277     | 10 | I | FP | FP |
278     | 11 | FP | FP | FP |
279     note: rs3 is always FP
280 */
281 wire [31:0] final_rs1, final_rs2, final_rs3;
282 // checking rs1, rs2, rs3: se debe checkear MEM primero y despues WB (debido a que
    ↳ MEM el la inst inmediatamente precedente)
283 //           calza la dirección           no se trata de x0           es un
    ↳ write back por realizar           son del mismo tipo (integer/fp)
284 assign final_rs1 = ( rs1_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 & ~
    ↳ reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 & format_type_MEM
    ↳ !=3'b100 & reg_access_option_EX[1]==reg_access_option_MEM[0] ) ? rd_MEM :
    ↳ (
285         ( rs1_add_EX==rd_add_WB & ~( rd_add_WB ==5'b0 & ~
    ↳ reg_access_option_WB [0] ) & format_type_WB !=3'b011 & format_type_WB
    ↳ !=3'b100 & reg_access_option_EX[1]==reg_access_option_WB[0] ) ? rd_WB : (
286         rs1_EX));
287 assign final_rs2 = ( rs2_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 & ~
    ↳ reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 & format_type_MEM
    ↳ !=3'b100 & ((reg_access_option_EX==2'b0 & ~reg_access_option_MEM[0]) | (
    ↳ reg_access_option_EX!=2'b0 & reg_access_option_MEM[0])) ) ? rd_MEM : (
288         ( rs2_add_EX==rd_add_WB & ~( rd_add_WB ==5'b0 & ~
    ↳ reg_access_option_WB [0] ) & format_type_WB !=3'b011 & format_type_WB
    ↳ !=3'b100 & ((reg_access_option_EX==2'b0 & ~reg_access_option_WB[0] ) | (
    ↳ reg_access_option_EX!=2'b0 & reg_access_option_WB[0] )) ) ? rd_WB : (
289         rs2_EX));
290 assign final_rs3 = ( rs3_add_EX==rd_add_MEM & ~( rd_add_MEM==5'b0 & ~
    ↳ reg_access_option_MEM[0] ) & format_type_MEM!=3'b011 & format_type_MEM
    ↳ !=3'b100 & reg_access_option_MEM[0] ) ? rd_MEM : (
291         ( rs3_add_EX==rd_add_WB & ~( rd_add_WB ==5'b0 & ~
    ↳ reg_access_option_WB [0] ) & format_type_WB !=3'b011 & format_type_WB
    ↳ !=3'b100 & reg_access_option_WB [0] ) ? rd_WB : (
292         rs3_EX));
293 // final outputs (análogo al caso single cycle)
294 assign operand1_EX = final_rs1;
295 assign operand2_EX = is_imm_valid_EX ? imm_EX : final_rs2;
296 assign operand3_EX = (format_type_EX==3'b011 | format_type_EX==3'b100) ?
    ↳ final_rs2 : final_rs3;
297 // permite la ejecución de la FPU, es importante en las instrucciones load cuando hay
    ↳ data hazard pues debe esperar a que se obtenga el dato de la memoria
298 assign enable_fpu = ( // si alguno de los operandos depende de rd_MEM
299     ( rs1_add_EX==rd_add_MEM & format_type_EX!=3'b101 & format_type_EX
    ↳ !=3'b110) | // no aplica para U y J
300     ( rs2_add_EX==rd_add_MEM & (~is_imm_valid_EX | format_type_EX==3'
    ↳ b011 | format_type_EX==3'b100)) | // aplica si imm es inválido o para B o S
301     ( rs3_add_EX==rd_add_MEM & format_type_EX==3'b001) // aplica para R4
302     ) & (~sub_format_type_MEM[2] & ~sub_format_type_MEM[0] &
    ↳ format_type_MEM==3'b010 & valid_data_mem) // y es una instrucción de tipo
    ↳ load (en MEM)
303     & ~( rd_add_MEM==5'b0 & ~reg_access_option_MEM[0] ) // y no se trata de x0
    ↳ =zero
304     ) ? ready_data_mem : 1'b1; // se espera a que el dato sea valido

```

305 `endmodule`

Instruction_Decoder.sv: archivo que describe los módulos *Instruction_Decoder* y *registers_files*.

```
1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6
7 module Instruction_Decoder(
8     input [31:0] in,
9     output is_imm_valid,
10    output [31:0] imm,
11    output [4:0] rd_add, rs1_add, rs2_add, rs3_add,
12    output [3:0] funct7_out,
13    output [2:0] format_type, sub_format_type, funct3,
14    output [1:0] reg_access_option, funct2
15 );
16 wire [6:0] opcode, funct7;
17 assign {opcode, funct7, funct3, funct2, rd_add, rs1_add, rs2_add, rs3_add}
18     = {in[6:0], in[31:25], in[14:12], in[26:25], in[11:7], in[19:15], in[24:20], in[31:27]};
19 /*
20     Format Group Classifier:
21     opcode -> Type
22     0110011 -> R
23     1010011 -> R (FP)
24     100 00 11 -> R4 (FP- fmadd.s)
25     100 01 11 -> R4 (FP- fmsub.s)
26     100 11 11 -> R4 (FP- fnmadd.s)
27     100 10 11 -> R4 (FP- fnmsub.s)
28     0100011 -> S
29     0100111 -> S (FP)
30     1100011 -> B
31     1101111 -> J
32     0110111 -> U (lui)
33     0010111 -> U (auipc)
34     0010011 -> I (addi, xori, ori, andi, slli, srli, srai, slti, sltiu)
35     0000011 -> I (lb, lh, lw, lbu, lhu)
36     1100111 -> I (jalr)
37     1110011 -> I (ecall, ebreak)
38     0000111 -> I (FP)
39     format_type[2:0]:
40     000 -> R type
41     001 -> R4 type
42     011 -> S type
43     100 -> B type
44     101 -> U type
45     110 -> J type
46     010 -> I type
47     111 -> Not valid type
48 */
```

```

49 assign format_type = (opcode[4:0]==5'b10011 & (opcode[6]^opcode[5])) ? 3'b000 : // R
50     (({opcode[6:4], opcode[1:0]}==5'b10011) ? 3'b001 : // R4
51     (({opcode[6:3], opcode[1:0]}==6'b010011) ? 3'b011 : // S
52     ((opcode==7'b1100011) ? 3'b100 : // B
53     ((opcode==7'b1101111) ? 3'b110 : // J
54     (({opcode[6], opcode[4:0]}==6'b010111) ? 3'b101 : // U
55     ((opcode[1:0]==2'b11&(opcode[6:2]==5'b100|opcode[6:2]==5'b0|opcode
↪ [6:2]==5'b1
56         |opcode[6:2]==5'b11001|opcode[6:2]==5'b11100)) ? 3'b010 : // I
57         3'b111))))); // Not valid
58 /*
59     Sub-Format Group Classifier
60     sub_format_type[2:0]:
61         if format_type[2:0] = 000
62             -000 -> R (FP)
63             -001 -> R
64         if format_type[2:0] = 001
65             -000 -> R4 (FP- fmadd.s)
66             -001 -> R4 (FP- fmsub.s)
67             -010 -> R4 (FP- fnmsub.s)
68             -011 -> R4 (FP- fnmadd.s)
69         100 00 11 -> R4 (FP- fmadd.s)
70         100 01 11 -> R4 (FP- fmsub.s)
71         100 10 11 -> R4 (FP- fnmsub.s)
72         100 11 11 -> R4 (FP- fnmadd.s)
73         if format_type[2:0] = 010
74             -000 -> I (FP)
75             -001 -> I (addi, xori, ori, andi, slli, srli, srli, srli, srai, srti, srtiu)
76             -010 -> I (lb, lh, lw, lbu, lhu)
77             -011 -> I (jalr)
78             -100 -> I (ecall, ebreak)
79         if format_type[2:0] = 011
80             -000 -> S (FP)
81             -001 -> S
82         if format_type[2:0] = 101
83             -000 -> U (lui)
84             -001 -> U (auipc)
85         else
86             -111
87     */
88 assign sub_format_type = (format_type==3'b000) ? (opcode[6] ? 3'b000 : 3'b001) : // R
89     ((format_type==3'b011) ? (opcode[2] ? 3'b000 : 3'b001) : // S
90     ((format_type==3'b101) ? (opcode[5] ? 3'b000 : 3'b001) : // U
91     ((format_type==3'b001) ? opcode[4:2] : // R4
92     ((format_type==3'b010&opcode[6:2]==5'b1) ? 3'b000 : // I (FP)
93     ((format_type==3'b010&opcode[6:2]==5'b0) ? 3'b010 : // I (lb, lh,
↪ lw, lbu, lhu)
94     ((format_type==3'b010&opcode[6:2]==5'b100) ? 3'b001 : // I (addi,
↪ xori, ori, andi, slli, srli, srai, srti, srtiu)
95     ((format_type==3'b010&opcode[6:2]==5'b11001) ? 3'b011 : // I (jalr)
96     ((format_type==3'b010&opcode[6:2]==5'b11100) ? 3'b100 : // I (
↪ ecall, ebreak)

```

```

97             3'b111))))))));
98  /*
99     Opration Classifier:
100    if Funct7[6:0]=0000000: Funct7-out[3:0]=0000
101    if Funct7[6:0]=0000001: Funct7-out[3:0]=0001
102    if Funct7[6:0]=0100000: Funct7-out[3:0]=0010
103    if Funct7[6:0]=0000100: Funct7-out[3:0]=0011
104    if Funct7[6:0]=0001000: Funct7-out[3:0]=0100
105    if Funct7[6:0]=0001100: Funct7-out[3:0]=0101
106    if Funct7[6:0]=0101100: Funct7-out[3:0]=0110
107    if Funct7[6:0]=0010000: Funct7-out[3:0]=0111
108    if Funct7[6:0]=0010100: Funct7-out[3:0]=1000
109    if Funct7[6:0]=1101000: Funct7-out[3:0]=1001
110    if Funct7[6:0]=1100000: Funct7-out[3:0]=1010
111    if Funct7[6:0]=1110000: Funct7-out[3:0]=1011
112    if Funct7[6:0]=1111000: Funct7-out[3:0]=1100
113    if Funct7[6:0]=1010000: Funct7-out[3:0]=1101
114    else:
115        Funct7-out[3:0]=1111
116  */
117  assign funct7_out = (funct7==7'b0000000) ? 4'b0000 : // 0x00
118                    ((funct7==7'b0000001) ? 4'b0001 : // 0x01
119                    ((funct7==7'b0100000) ? 4'b0010 : // 0x20
120                    ((funct7==7'b0000100) ? 4'b0011 :
121                    ((funct7==7'b0001000) ? 4'b0100 :
122                    ((funct7==7'b0001100) ? 4'b0101 :
123                    ((funct7==7'b0101100) ? 4'b0110 :
124                    ((funct7==7'b0010000) ? 4'b0111 :
125                    ((funct7==7'b0010100) ? 4'b1000 :
126                    ((funct7==7'b1101000) ? 4'b1001 :
127                    ((funct7==7'b1100000) ? 4'b1010 :
128                    ((funct7==7'b1110000) ? 4'b1011 :
129                    ((funct7==7'b1111000) ? 4'b1100 :
130                    ((funct7==7'b1010000) ? 4'b1101 :
131                    4'b1111))))))));
132  /*
133     Registers selection options
134     if format-type[2:0]=000 (R):
135         if sub-format-type[2:0]=000 (FP):
136             if Funct7-out[3:0]=1010 o 1011 o 1101 ( fwt.w(u).s fmv.x.w fclass.s feq.s flt.s fle.s )
137                 reg_access_option[1:0]=10
138             if Funct7-out[3:0]=1001 o 1100 ( fwt.s.w(u) fmv.w.x )
139                 reg_access_option[1:0]=01
140             else:
141                 reg_access_option[1:0]=11
142
143         if sub-format-type[2:0]=001 (no FP):
144             reg_access_option[1:0]=00
145     if format-type[2:0]=010 (I):
146         if sub-format-type[2:0]=000 (FP): reg_access_option[1:0]=01
147         if sub-format-type[2:0]=001 o 010 o 011 o 100 (no FP): reg_access_option[1:0]=00
148     if format-type[2:0]=011 (S):

```

```

149     if sub-format-type[2:0]=000 (FP): reg_access_option[1:0]=01
150     if sub-format-type[2:0]=001 (no FP): reg_access_option[1:0]=00
151     if format-type[2:0]=001 (R4):
152         reg_access_option[1:0]=11
153     if format-type[2:0]=100 (B):
154         reg_access_option[1:0]=00
155     if format-type[2:0]=101 (U) o 110 (J):
156         reg_access_option[1:0]=00 o 10=00
157     reg_access_option[1:0] =
158         00: rd, rs1, rs2 son registros Integer
159         01: rs1 es reg Integer y rd, rs2 son FP
160         10: rd es reg Integer y rs1, rs2 son FP
161         11: rd, rs1, rs2 son registros FP
162     reg_access_option:
163     |code| rd | rs1 | rs2 |
164     | 00 | I | I | I |
165     | 01 | FP | I | FP |
166     | 10 | I | FP | FP |
167     | 11 | FP | FP | FP |
168     note: rs3 is always FP
169 */
170 assign reg_access_option = (format_type[2:1]==2'b10|format_type==3'b110) ? 2'b00 :
    ↪ // B-U-J
171         ((format_type==3'b001) ? 2'b11 : // R4
172         ((format_type==3'b011&sub_format_type==3'b000) ? 2'b01 : // S
    ↪ (FP)
173         ((format_type==3'b011&sub_format_type==3'b001) ? 2'b00 : // S
    ↪ (no FP)
174         ((format_type==3'b010&sub_format_type==3'b000) ? 2'b01 : // I (
    ↪ FP)
175         ((format_type==3'b010&sub_format_type!=3'b111) ? 2'b00 : // I (
    ↪ no FP)
176         ((format_type==3'b000&sub_format_type==3'b001) ? 2'b00 : // R
    ↪ (no FP)
177         ((format_type==3'b000&sub_format_type==3'b000&(funct7_out
    ↪ ==4'b1010|
178         funct7_out==4'b1011|funct7_out==4'b1101 )) ? 2'b10 : // R (
    ↪ FP) ( fwt.w(u).s fmv.x.w fclass.s feq.s flt.s fle.s )
179         ((format_type==3'b000&sub_format_type==3'b000&(
180         funct7_out==4'b1001|funct7_out==4'b1100 )) ? 2'b01 : // R (
    ↪ FP) ( fwt.s.w(u) fmv.w.x )
181         ((format_type==3'b000&sub_format_type==3'b000&(
182         funct7_out!=4'b1111
183         )) ? 2'b11 : // R (FP) ( ~ )
184         2'b00))))))));
185 /*
186     if format-type[2:0] = 010 (I type):
187         imm[31:0] = Inst[31]*ones[19:0] : Inst[30:20]
188         is_imm_valid = 1
189     if format-type[2:0] = 011 (S type):
190         imm[31:0] = Inst[31]*ones[19:0] : Inst[30:25] : Inst[11:7]
191         is_imm_valid = 1
192     if format-type[2:0] = 100 (B type):

```

```

192     imm[31:0] = Inst[31]*ones[19:0] : Inst[7] : Inst[30:25] : Inst[11:8] : 0
193     is_imm_valid = 1
194     if format-type[2:0] = 101 (U type):
195         imm[31:0] = Inst[31:12] : zeros[11:0]
196         is_imm_valid = 1
197     if format-type[2:0] = 110 (J type):
198         imm[31:0] = Inst[31]*ones[11:0] : Inst[19:12] : Inst[20] : Inst[30:21] : 0
199         is_imm_valid = 1
200     else:
201         imm[31:0] = dont_care[31:0]
202         is_imm_valid = 0
203 */
204 assign is_imm_valid = (format_type[2:1]==2'b01|format_type[2:1]==2'b10|
    ↪ format_type==3'b110) ? 1'b1 : 1'b0;
205 assign imm = (format_type==3'b010) ? 32'(signed'(in[31:20]))
    ↪ : // I
206         ((format_type==3'b011) ? 32'(signed'({in[31:25], in[11:7]}))           :
    ↪ // S
207         ((format_type==3'b100) ? 32'(signed'({in[31], in[7], in[30:25], in[11:8], 1'b0}))
    ↪ : // B
208         ((format_type==3'b101) ? {in[31:12], 12'b0}                               : //
    ↪ U
209         ((format_type==3'b110) ? 32'(signed'({in[31], in[19:12], in[20], in[30:21], 1'b0}))
    ↪ : // J
210         32'bxxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx_xxxx));
211 endmodule
212
213 module registers_files(
214     input clk,           // clock del sistema
215     input [4:0] rs1_add, // dirección reg source 1
216     input [4:0] rs2_add, // dirección reg source 2
217     input [4:0] rs3_add, // dirección reg source 3
218     input [4:0] wb_add,  // dirección reg para wb
219     input [31:0] wb_data, // data para wb
220     input write_reg,    // pin que indica si realizar wb
221     input is_wb_data_fp, // indica si el reg de wb es fp o no
222     input is_rs1_fp,    // indica si el reg source 1 es fp o no
223     input is_rs2_fp,    // indica si el reg source 2 es fp o no
224     output [31:0] rs1,  // data de reg source 1
225     output [31:0] rs2,  // data de reg source 2
226     output [31:0] rs3   // data de reg source 3
227 );
228 reg[31:0] int_registers[31:0];
229 reg[31:0] fp_registers[31:0];
230 initial for(int i=0; i<32; i++) begin
231     if(i == 2) {int_registers[i], fp_registers[i]} = {32'h2FFC, 32'b0}; // sp
232     else if(i == 3) {int_registers[i], fp_registers[i]} = {32'h1800, 32'b0}; // gp
233     else {int_registers[i], fp_registers[i]} = '0;
234 end
235 assign rs1 = is_rs1_fp ? fp_registers[rs1_add] : int_registers[rs1_add];
236 assign rs2 = is_rs2_fp ? fp_registers[rs2_add] : int_registers[rs2_add];
237 assign rs3 = fp_registers[rs3_add];

```

```

238     always @ (negedge clk)
239         if(write_reg)
240             if(is_wb_data_fp) fp_registers[wb_add] <= wb_data;
241         else begin
242             if(wb_add == 5'd0) int_registers[wb_add] <= 32'd0;
243             else int_registers[wb_add] <= wb_data;
244         end
245     endmodule

```

`cache_controller.sv`: archivo que describe el `cache_controller`.

```

1  /*
2  autor: Gianluca Vincenzo D'Agostino Matute
3  Santiago de Chile, Septiembre 2021
4
5  THIS CODE WAS EXTRACTED FROM: D. A. Patterson and J. L. Hennessy, "Computer
   ↪ Organization and Design, The Hardware Software/Interface: RISC-V Edition".
   ↪ Cambridge, Estados Unidos: Morgan Kaufmann, 2018.
6  */
7
8  `timescale 1ns / 1ps
9
10 // Data structures for cache tad & data
11
12 parameter int TAGMSB = 31; // tag msb
13 parameter int TAGLSB = 14; // tag lsb
14
15 // Data structure for cache tag
16 typedef struct packed{
17     bit valid;           // valid bit
18     bit dirty;          // dirty bit
19     bit [TAGMSB:TAGLSB]tag; // tag bits
20 } cache_tag_type;
21
22 // Data structure fot cache memory request
23 typedef struct{
24     bit [9:0]index; // 10-bit index
25     bit we;         // write enable
26 } cache_req_type;
27
28 // 128-bit cache line data
29 typedef bit [127:0]cache_data_type;
30
31 // Data structures for CPU <-> Cache controller interface
32
33 // CPU request (CPU -> cache controller)
34 typedef struct{
35     bit [31:0]addr; // 32-bit request addr
36     bit [31:0]data; // 32-bit request data (used when white)
37     bit rw;         // request type: 0=read, 1=write
38     bit valid;     // request is valid
39 } cpu_req_type;

```

```

40
41 // Cache result (cache controller -> CPU)
42 typedef struct{
43     bit [31:0]data; // 32-bit data
44     bit ready; // result is ready
45 } cpu_result_type;
46
47 // Data structures for cache controller <-> memory interface
48
49 // Memory request (cache controller -> memory)
50 typedef struct{
51     bit [31:0]addr; // Request byte addr
52     bit [127:0]data; // 128-bit request data (used when write)
53     bit rw; // request type: 0=read, 1=write
54     bit valid; // request is valid
55 } mem_req_type;
56
57 // Memory controller response (memory -> cache controller)
58 typedef struct{
59     cache_data_type data; // 128-bit read back data
60     bit ready; // data is ready
61 } mem_data_type;
62
63 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
64
65 /* cache: data memory, single port, 1024 blocks */
66 module dm_cache_data(input bit clk,
67     ↪ valid
68     input cache_data_type data_write, // write port (128-bit line)
69     output cache_data_type data_read // read port
70     );
71     timeunit 1ns;
72     timeprecision 1ps;
73
74     cache_data_type data_mem[0:1023];
75
76     initial begin
77         for(int i=0; i<1024; i++)
78             data_mem[i] = '0;
79     end
80
81     assign data_read = data_mem[data_req.index];
82
83     always_ff @(posedge(clk)) begin
84         if(data_req.we)
85             data_mem[data_req.index] <= data_write;
86     end
87
88 endmodule
89
90 /* cache: tag memory, single port, 1024 blocks */

```

```

91 module dm_cache_tag(input bit clk, // write clock
92     input  cache_req_type tag_req, // tag request/command, e.g. RW. valid
93     input  cache_tag_type tag_write, // write port
94     output cache_tag_type tag_read // read port
95 );
96 timeunit 1ns;
97 timeprecision 1ps;
98
99 cache_tag_type tag_mem[0:1023];
100
101 initial begin
102     for(int i=0; i<1024; i++)
103         tag_mem[i] = '0;
104 end
105
106 assign tag_read = tag_mem[tag_req.index];
107
108 always_ff @(posedge(clk)) begin
109     if(tag_req.we)
110         tag_mem[tag_req.index] <= tag_write;
111 end
112
113 endmodule
114
115 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
116
117 /* cache finite state machine */
118 module dm_cache_fsm(input bit clk, input bit rst,
119     input  cpu_req_type  cpu_req, // CPU request input  (CPU -> cache)
120     input  mem_data_type mem_data, // memory response  (memory ->
121         ↪ cache)
122     output mem_req_type  mem_req, // memory request    (cache ->
123         ↪ memory)
124     output cpu_result_type cpu_res // CPU response output (cache -> CPU);
125 );
126 timeunit 1ns;
127 timeprecision 1ps;
128
129 /* write clock */
130 typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;
131
132 /* FSM register */
133 cache_state_type vstate, rstate;
134
135 /* interface signals to tag memory */
136 cache_tag_type tag_read; // tag read result
137 cache_tag_type tag_write; // tag write data
138 cache_req_type tag_req; // tag request
139
140 /* interface signals to cache data memory */
141 cache_data_type data_read; // data read result
142 cache_data_type data_write; // data write data

```

```

141 cache_req_type data_req; // data request
142
143 /* temporary variable for cache controller result */
144 cpu_result_type v_cpu_res;
145
146 /* temporary variable for memory controller request */
147 mem_req_type v_mem_req;
148
149 // Connect to output ports
150 assign mem_req = v_mem_req;
151 assign cpu_res = v_cpu_res;
152
153 always_comb begin
154
155     /*----- default values for all signals -----*/
156
157     /* no state change by default */
158     vstate = rstate;
159     v_cpu_res = '{0, 0};
160     tag_write = '{0, 0, 0};
161
162     /* read tag by default */
163     tag_req.we = '0;
164     /* direct map index for tag */
165     tag_req.index = cpu_req.addr[13:4];
166
167     /* read current cache line by default */
168     data_req.we = '0;
169     /* direct map index for cache data */
170     data_req.index = cpu_req.addr[13:4];
171
172     /* modify correct word (32-bits) based on address */
173     data_write = data_read;
174     case(cpu_req.addr[3:2])
175         2'b00: data_write[31:0] = cpu_req.data;
176         2'b01: data_write[63:32] = cpu_req.data;
177         2'b10: data_write[95:64] = cpu_req.data;
178         2'b11: data_write[127:96] = cpu_req.data;
179     endcase
180
181     /* read out correct word (32-bits) from cache (to CPU) */
182     case(cpu_req.addr[3:2])
183         2'b00: v_cpu_res.data = data_read[31:0];
184         2'b01: v_cpu_res.data = data_read[63:32];
185         2'b10: v_cpu_res.data = data_read[95:64];
186         2'b11: v_cpu_res.data = data_read[127:96];
187     endcase
188
189     /* memory request address (sampled from CPU request) */
190     v_mem_req.addr = cpu_req.addr;
191     /* memory request data (used in write) */
192     v_mem_req.data = data_read;

```

```

193
194 v_mem_req.rw = '0;
195 v_mem_req.valid = '0; // MOD: faltaba
196
197 /*----- cache FSM -----*/
198 case(rstate)
199     /* idle state */
200     idle: begin
201         /* if there is a CPU request, then compare cache tag */
202         if(cpu_req.valid)
203             vstate = compare_tag;
204     end
205     /* compare_tag state */
206     compare_tag: begin
207         /* cache hit (tag match and cache entry is valid) */
208         if(cpu_req.addr[TAGMSB:TAGLSB] == tag_read.tag && tag_read.valid) begin
209             v_cpu_res.ready = '1;
210             /* write hit */
211             if(cpu_req.rw) begin
212                 /* read/modify cache line */
213                 tag_req.we = '1;
214                 data_req.we = '1;
215                 /* no change in tag */
216                 tag_write.tag = tag_read.tag;
217                 tag_write.valid = '1;
218                 /* cache line is dirty */
219                 tag_write.dirty = '1;
220             end
221             /* xaction is finished */
222             vstate = cpu_req.valid ? compare_tag : idle; // MOD: original -> //vstate =
↪ idle;
223         end
224         /* cache miss */
225         else begin
226             /* generate new tag */
227             tag_req.we = '1;
228             tag_write.valid = '1;
229             /* new tag */
230             tag_write.tag = cpu_req.addr[TAGMSB:TAGLSB];
231             /* cache line is dirty if write */
232             tag_write.dirty = cpu_req.rw;
233             /* generate memory request on miss */
234             v_mem_req.valid = '1;
235             /* compulsory miss or miss with clean block */
236             if(tag_read.valid == 1'b0 || tag_read.dirty == 1'b0)
237                 /* wait till a new block is allocated */
238                 vstate = allocate;
239             /* miss with dirty line */
240             else begin
241                 /* write back address */
242                 v_mem_req.addr = {tag_read.tag, cpu_req.addr[TAGLSB-1:0]};
243                 v_mem_req.rw = '1;

```

```

244             /* wait till write is completed */
245             vstate = write_back;
246         end
247     end
248 end
249 /* wait for allocating a new cache line */
250 allocate: begin
251     /* memory controller has responded */
252     if(mem_data.ready) begin
253         /* re-compare tag for write miss (need modify correct word) */
254         vstate = compare_tag;
255         data_write = mem_data.data;
256         /* update cache line data */
257         data_req.we = '1;
258     end
259 end
260 /* wait for writing back dirty cache line */
261 write_back: begin
262     /* write back is completed */
263     if(mem_data.ready) begin
264         /* issue new data memory request (allocating a new line) */
265         v_mem_req.valid = '1;
266         v_mem_req.rw = '0;
267         vstate = allocate;
268     end
269 end
270 endcase
271
272 end
273
274 always_ff @(posedge(clk)) begin
275     if(rst)
276         rstate <= idle; // reset to idle state
277     else
278         rstate <= vstate;
279 end
280
281 /* connect cache tag/data memory */
282 dm_cache_tag ctag(.);
283 dm_cache_data cdata(.);
284
285 endmodule
286
287 ///////////////////////////////////////////////////
288
289 /* TOP MODULE OF THIS FILE */
290 module cache_controller(
291     input clk,
292     input rst,
293     input [31:0] cpu_addr,
294     input [31:0] cpu_data_in,
295     input cpu_rw,

```

```

296 input cpu_valid,
297 input [127:0] mem_data_in,
298 input mem_ready,
299 output [31:0] cpu_data_out,
300 output cpu_ready,
301 output [31:0] mem_addr,
302 output [127:0] mem_data_out,
303 output mem_rw,
304 output mem_valid
305 );
306
307 // Inputs
308 cpu_req_type  cpu_req;
309 mem_data_type mem_data;
310
311 // Outputs
312 mem_req_type  mem_req;
313 cpu_result_type cpu_res;
314
315 // Input connections
316 assign cpu_req = '{addr:cpu_addr, data:cpu_data_in, rw:cpu_rw, valid:cpu_valid};
317 assign mem_data = '{data:mem_data_in, ready:mem_ready};
318
319 // Output connections
320 assign mem_addr    = mem_req.addr;
321 assign mem_data_out = mem_req.data;
322 assign mem_rw      = mem_req.rw;
323 assign mem_valid   = mem_req.valid;
324 assign cpu_data_out = cpu_res.data;
325 assign cpu_ready   = cpu_res.ready;
326
327 // FSM
328 dm_cache_fsm cache_fsm(clk, rst, cpu_req, mem_data, mem_req, cpu_res);
329
330 endmodule

```

memory.sv: archivo *top* de las memorias implementadas que contiene los submódulos *cache_controller* y *block_ram*.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6
7 module memory #(parameter initial_option=0)(
8     input clk,           // global clock
9     input rst,          // reset cache FSM
10    input rw,            // =1 -> write, =0 -> read
11    input valid,         // do operation
12    input [31:0] addr,   // address from cpu
13    input [31:0] data_in, // data to save

```

```

14 input [1:0] byte_half_word, // Options; 00 or 11->word, 01->halfword, 10->byte
15 input is_load_unsigned,    // =1 -> load is unsigned, =0 -> load is signed
16 output reg ready,        // the operation is ready
17 output out_of_range,     // the address is out of range
18 output reg [31:0] data_out // readed data
19 );
20
21 // para manejar el almacenamiento de datos en el always
22 reg lecture_flag;
23
24 // entradas indirectas (que pasan por always combinacional) del cache
25 reg [31:0] data_to_cache;
26 reg rw_to_cache;
27 reg valid_to_cache;
28
29 // conexiones desde la salida del cache al always combinacional
30 wire [31:0] data_from_cache; // salida indirecta a cpu
31 wire ready_from_cache; // salida indirecta a cpu
32 wire [31:0] addr_from_cache; // salida indirecta a bram
33
34 // entrada indirecta de la bram desde el cache
35 wire [11:0] addr_to_bram; // esto es mas bien un indice
36
37 // conexiones desde la salida de cache a la entrada de bram
38 wire [127:0] data_from_cache_to_bram;
39 wire valid_from_cache_to_bram;
40 wire rw_from_cache_to_bram;
41
42 // conexiones desde la salida de bram a la entrada de cache
43 wire [127:0] data_from_bram_to_cache;
44 wire ready_from_bram_to_cache;
45
46 // Se verifica que la dirección está en el rango de la memoria ram (3906*4*4bytes = 62496
47 // ↪ bytes = 62.496KB)
48 assign out_of_range = addr > 32'b1111_0100_0001_1111 ? '1 : '0;
49
50 // Se ajusta dirección recibida por bram (recibe 14 bits de addr)
51 assign addr_to_bram = addr_from_cache[15:4];
52
53 // always necesario para manejar byte, halfword, word
54 always_comb begin
55     lecture_flag = 0;
56     data_out = 32'b0;
57     data_to_cache = 32'b0;
58     rw_to_cache = 0;
59     ready = 0;
60     valid_to_cache = 0;
61     if(~out_of_range*valid) begin
62         valid_to_cache = 1;
63         if(rw) begin // escritura
64             if(ready_from_cache & ~lecture_flag) begin
65                 data_out = data_from_cache;

```

```

65         lecture_flag = 1;
66     end
67     if(lecture_flag) begin
68         valid_to_cache = 0;
69         case(byte_half_word)
70             2'b01: begin // 01->store halfword (little endian)
71                 case(addr[1])
72                     1'b0: data_to_cache = {data_out[31:16],data_in[15:0]};
73                     1'b1: data_to_cache = {data_in[15:0],data_out[15:0]};
74                 endcase
75             end
76             2'b10: begin // 10->store byte (little endian)
77                 case(addr[1:0])
78                     2'b00: data_to_cache = {data_out[31:8],data_in[7:0]};
79                     2'b01: data_to_cache = {data_out[31:16],data_in[7:0],data_out
↪ [7:0]};
80                     2'b10: data_to_cache = {data_out[31:24],data_in[7:0],data_out
↪ [15:0]};
81                     2'b11: data_to_cache = {data_in[7:0],data_out[23:0]};
82                 endcase
83             end
84             default: begin // 00 or 11->store word
85                 data_to_cache = data_in;
86             end
87         endcase
88         rw_to_cache = 1;
89         valid_to_cache = 1;
90         ready = ready_from_cache;
91     end
92 end
93 else begin // lectura is_load_unsigned
94     if(ready_from_cache) begin
95         case(byte_half_word)
96             2'b01: begin // 01->load halfword (little endian)
97                 case(addr[1])
98                     1'b0: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[15:0])) : 32'(signed'(data_from_cache[15:0]));
99                     1'b1: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[31:16])) : 32'(signed'(data_from_cache[31:16]));
100                 endcase
101             end
102             2'b10: begin // 10->load byte (little endian)
103                 case(addr[1:0])
104                     2'b00: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[7:0])) : 32'(signed'(data_from_cache[7:0]));
105                     2'b01: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[15:8])) : 32'(signed'(data_from_cache[15:8]));
106                     2'b10: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[23:16])) : 32'(signed'(data_from_cache[23:16]));
107                     2'b11: data_out = is_load_unsigned ? 32'(unsigned'(
↪ data_from_cache[31:24])) : 32'(signed'(data_from_cache[31:24]));
108                 endcase

```

```

109         end
110         default: begin // 00 or 11->load word
111             data_out = data_from_cache;
112         end
113     endcase
114     ready = 1;
115     valid_to_cache = 1;
116     lecture_flag = 0;
117     data_to_cache = 32'bX;
118     rw_to_cache = 0;
119     end
120 end
121 end
122 else begin
123     lecture_flag = 0;
124     data_out = 32'bX;
125     data_to_cache = 32'bX;
126     rw_to_cache = 0;
127     ready = 0;
128     valid_to_cache = 0;
129 end
130 end
131
132 cache_controller cache_controller_unit(
133     //inputs
134     .clk(clk), // global clock
135     .rst(rst), // reset cache
136     .cpu_addr(addr), // (32 bits) address from cpu
137     .cpu_data_in(data_to_cache), // (32 bits) bus dependiente de always combinacional
138     .cpu_rw(rw_to_cache), // rw dependiente de always combinacional
139     .cpu_valid(valid_to_cache), // valid dependiente de always combinacional
140     .mem_data_in(data_from_bram_to_cache), // (128 bits) conectado directamente con
    ↪ la salida (data_out) de bram
141     .mem_ready(ready_from_bram_to_cache), // conectado directamente con la salida (
    ↪ ready) de bram
142     //outputs
143     .cpu_data_out(data_from_cache), // (32 bits) salida que va a always combinacional
144     .cpu_ready(ready_from_cache), // salida que va a always combinacional
145     .mem_addr(addr_from_cache), // (32 bits) salida que va a always combinacional
146     .mem_data_out(data_from_cache_to_bram), // (128 bits) conectado directamente a
    ↪ la entrada (data_in) de la bram
147     .mem_rw(rw_from_cache_to_bram), // conectado directamente a la entrada (rw) de
    ↪ la bram
148     .mem_valid(valid_from_cache_to_bram) // conectado directamente a la entrada (
    ↪ valid) de la bram
149 );
150
151 block_ram #(initial_option(initial_option)) block_ram_unit(
152     //inputs
153     .clk(clk), // global clock
154     .addr(addr_to_bram), // (14 bits) bus dependiente de always combinacional
155     .data_in(data_from_cache_to_bram), // (128 bits) conectado directamente con la

```

```

156     ↪ salida (mem_data_out) del cache
        .rw(rw_from_cache_to_bram), // conectado directamente con la salida (mem_rw) del
157     ↪ cache
        .valid(valid_from_cache_to_bram), // conectado directamente con la salida (
158     ↪ mem_valid) del cache
        //outputs
159     .data_out(data_from_bram_to_cache), // (128 bits) conectado directamente a la
        ↪ entrada (mem_data_in) del cache
160     .ready(ready_from_bram_to_cache) // conectado directamente a la entrada (
        ↪ mem_ready) del cache
161     );
162
163 endmodule
164
165 // ram 3906*4*4bytes = 62496bytes = 62.496KB
166 module block_ram #(parameter initial_option=0)(
167     input clk,
168     input [11:0] addr,
169     input [127:0] data_in,
170     input rw,
171     input valid,
172     output reg [127:0] data_out,
173     output reg ready
174 );
175 reg[127:0] mem[3905:0];
176 initial
177     if (initial_option==1) $readmemh("data_in.mem", mem);
178     else if (initial_option==2) $readmemh("text_in.mem", mem);
179     else for(int i=0; i<3906; i++) mem[i] = '0;
180 always_ff @ (posedge clk)
181     if (valid) begin
182         if(rw) mem[addr] = data_in;
183         data_out = mem[addr];
184         ready = '1;
185     end
186     else ready = '0;
187 endmodule

```

ALU.sv: archivo que contiene los módulos *ALU* y *alu_op_selection*.

```

1  /*
2  autor: Gianluca Vincenzo D'Agostino Matute
3  Santiago de Chile, Septiembre 2021
4  */
5  `timescale 1ns / 1ps
6
7  module alu_op_selection(
8      input [6:0] imm_11_5, // imm[11:5]
9      input [3:0] funct7_out,
10     input [2:0] format_type, sub_format_type, funct3,
11     output reg [4:0] alu_option
12 );

```

```

13  always_comb
14      if(format_type==3'b000 & sub_format_type==3'b001) // R (no FP)
15          case(funcnt3)
16              3'b000: alu_option = (funcnt7_out==4'b0000) ? 5'b00_000 : // add
17                  ((funcnt7_out==4'b0010) ? 5'b00_001 : // sub
18                  ((funcnt7_out==4'b0001) ? 5'b10_000 : // mul
19                  5'bxx_xxx));
20              3'b001: alu_option = (funcnt7_out==4'b0000) ? 5'b00_101 : // sll
21                  ((funcnt7_out==4'b0001) ? 5'b11_000 : // mulh
22                  5'bxx_xxx);
23              3'b010: alu_option = (funcnt7_out==4'b0000) ? 5'b01_010 : // slt
24                  ((funcnt7_out==4'b0001) ? 5'b11_010 : // mulsu
25                  5'bxx_xxx);
26              3'b011: alu_option = (funcnt7_out==4'b0000) ? 5'b01_011 : // sltu
27                  ((funcnt7_out==4'b0001) ? 5'b11_001 : // mulu
28                  5'bxx_xxx);
29              3'b100: alu_option = (funcnt7_out==4'b0000) ? 5'b00_010 : // xor
30                  ((funcnt7_out==4'b0001) ? 5'b10_011 : // div
31                  5'bxx_xxx);
32              3'b101: alu_option = (funcnt7_out==4'b0000) ? 5'b00_110 : // srl
33                  ((funcnt7_out==4'b0010) ? 5'b00_111 : // sra
34                  ((funcnt7_out==4'b0001) ? 5'b10_100 : // divu
35                  5'bxx_xxx));
36              3'b110: alu_option = (funcnt7_out==4'b0000) ? 5'b00_011 : // or
37                  ((funcnt7_out==4'b0001) ? 5'b10_101 : // rem
38                  5'bxx_xxx);
39              3'b111: alu_option = (funcnt7_out==4'b0000) ? 5'b00_100 : // and
40                  ((funcnt7_out==4'b0001) ? 5'b10_111 : // remu
41                  5'bxx_xxx);
42          default: alu_option = 5'bxx_xxx;
43      endcase
44      else if(format_type==3'b100) // B
45          case(funcnt3)
46              3'b000: alu_option = 5'b01_000; // beq
47              3'b001: alu_option = 5'b01_001; // bne
48              3'b100: alu_option = 5'b01_010; // blt
49              3'b101: alu_option = 5'b01_100; // bge
50              3'b110: alu_option = 5'b01_011; // bltu
51              3'b111: alu_option = 5'b01_101; // bgeu
52          default: alu_option = 5'bxx_xxx;
53      endcase
54      else if(format_type==3'b010) // I
55          if(sub_format_type==3'b001) // (addi, xori, ori, andi, slli, srli, srli, srli, slti, sltiu)
56              case (funcnt3)
57                  3'b000: alu_option = 5'b00_000; // addi
58                  3'b001: alu_option = 5'b00_101; // slli
59                  3'b010: alu_option = 5'b01_010; // slti
60                  3'b011: alu_option = 5'b01_011; // sltiu
61                  3'b100: alu_option = 5'b00_010; // xori
62                  3'b101: alu_option = (imm_11_5==7'b000_0000) ? 5'b00_110 : // srli
63                      ((imm_11_5==7'b010_0000) ? 5'b00_111 : // srli
64                      5'bxx_xxx);

```

```

65         3'b110: alu_option = 5'b 00_011; // ori
66         3'b111: alu_option = 5'b 00_100; // andi
67         default: alu_option = 5'bxx_xxx;
68     endcase
69     else
70         alu_option = 5'b00_000;
71     else
72         alu_option = 5'b00_000; // S(011) & default
73 endmodule
74
75 module ALU(
76     input [31:0] in1,
77     input [31:0] in2,
78     input [4:0] operation,
79     output [31:0] res,
80     output boolean_res
81 );
82 wire [31:0] res_simple, res_mul;
83 wire temp_boolean_res;
84 assign res      = (operation[4:3] == 2'b00) ? res_simple :
85                 ((operation[4] == 1'b1) ? res_mul :
86                 ((operation[4:3] == 2'b01) ? (temp_boolean_res ? 32'b1 : 32'b0) : 32'bX))
87                 ↪ ;
88 assign boolean_res = (operation[4:3] == 2'b01) ? temp_boolean_res : 1'b0;
89 /*
90 over/under-flow se debe controlar mediante software
91 integer_sub_alu_simple:
92     operation=00_000: + add
93     operation=00_001: - sub
94     operation=00_010: ^ xor
95     operation=00_011: | or
96     operation=00_100: & and
97     operation=00_101: << shift left
98     operation=00_110: >> shift righth
99     operation=00_111: >> shift righth (MSB extend)
100 integer_sub_alu_comparison:
101     operation=01_000: == equality
102     operation=01_001: != difference
103     operation=01_010: < is less than
104     operation=01_011: < is less than (unsigned)
105     operation=01_100: >= is bigger or equal
106     operation=01_101: >= is bigger or equal (unsigned)
107 integer_sub_alu_mul:
108     operation=10_000: * mul
109     operation=11_000: * mul high
110     operation=11_001: * mul high (unsigned)
111     operation=11_010: * mul high (signed*unsigned)
112     operation=10_011: / div
113     operation=10_100: / div (unsigned)
114     operation=10_101: % modulo
115     operation=10_111: % modulo (unsigned)
116 */

```

```

116 integer_sub_alu_simple simple_alu (in1, in2, operation[2:0], res_simple);
117 integer_sub_alu_comparison comparison_alu(in1, in2, operation[2:0], temp_boolean_res
↪ );
118 integer_sub_alu_mul mul_alu (in1, in2, operation[2:0], operation[3], res_mul);
119 endmodule
120
121 module integer_sub_alu_simple(
122     input [31:0] in1,
123     input [31:0] in2,
124     input [2:0] sub_option,
125     output reg [31:0] res
126 );
127 always_comb begin
128     case(sub_option)
129         3'b000: // operation=00_000: + add
130             res = in1 + in2;
131         3'b001: // operation=00_001: - sub
132             res = in1 - in2;
133         3'b010: // operation=00_010: ^ xor
134             res = in1 ^ in2;
135         3'b011: // operation=00_011: | or
136             res = in1 | in2;
137         3'b100: // operation=00_100: & and
138             res = in1 & in2;
139         3'b101: // operation=00_101: << shift left
140             res = (unsigned'(in1) <<unsigned'(in2[4:0]));
141         3'b110: // operation=00_110: >> shift righth
142             res = (unsigned'(in1) >>unsigned'(in2[4:0]));
143         3'b111: // operation=00_111: >> shift righth (MSB extend)
144             res = (signed'(in1) >>>unsigned'(in2[4:0]));
145         default:
146             res = 32'bX;
147     endcase
148 end
149 endmodule
150
151 module integer_sub_alu_comparison(
152     input [31:0] in1,
153     input [31:0] in2,
154     input [2:0] sub_option,
155     output reg boolean_res
156 );
157 always_comb begin
158     case(sub_option)
159         3'b000: // operation=01_000: == equality
160             boolean_res = (in1 == in2) ? 1'b1 : 1'b0;
161         3'b001: // operation=01_001: != difference
162             boolean_res = (in1 != in2) ? 1'b1 : 1'b0;
163         3'b010: // operation=01_010: < is less than
164             boolean_res = (signed'(in1) <signed'(in2)) ? 1'b1 : 1'b0;
165         3'b011: // operation=01_011: < is less than (unsigned)
166             boolean_res = (unsigned'(in1) <unsigned'(in2)) ? 1'b1 : 1'b0;

```

```

167     3'b100: // operation=01_100: >= is bigger or equal
168         boolean_res = (signed'(in1) >=signed'(in2)) ? 1'b1 : 1'b0;
169     3'b101: // operation=01_101: >= is bigger or equal (unsigned)
170         boolean_res = (unsigned'(in1) >=unsigned'(in2)) ? 1'b1 : 1'b0;
171     default:
172         boolean_res = 1'b0;
173     endcase
174 end
175 endmodule
176
177 module integer_sub_alu_mul(
178     input [31:0] in1,
179     input [31:0] in2,
180     input [2:0] sub_option,
181     input upper_res,
182     output [31:0] res
183 );
184 reg [63:0] temp_res;
185 assign res = upper_res ? temp_res[63:32] : temp_res[31:0];
186 always_comb begin
187     case(sub_option)
188         3'b000: // operation=10_000: * mul - operation=11_000: * mul high
189             temp_res = 64'( signed'(in1) * signed'(in2) );
190         3'b001: // operation=11_001: * mul high (unsigned)
191             temp_res = 64'( unsigned'(in1) * unsigned'(in2) );
192         3'b010: // operation=11_010: * mul high (signed*unsigned)
193             temp_res = 64'( signed'({in1[31], in1}) * signed'({1'b0, in2}) );
194         3'b011: // operation=10_011: / div
195             temp_res = in2==0 ? 64'b11111111111111111111111111111111 : {32'b0, 32'(
↪ signed'(in1) / signed'(in2) )};
196         3'b100: // operation=10_100: / div (unsigned)
197             temp_res = in2==0 ? 64'b11111111111111111111111111111111 : {32'b0, 32'(
↪ unsigned'(in1) / unsigned'(in2) )};
198         3'b101: // operation=10_101: % modulo
199             temp_res = in2==0 ? in1 : {32'b0, 32'( signed'(in1) % signed'(in2) )};
200         3'b111: // operation=10_111: % modulo (unsigned)
201             temp_res = in2==0 ? in1 : {32'b0, 32'( unsigned'(in1) % unsigned'(in2) )};
202     default:
203         temp_res = 64'bX;
204     endcase
205 end
206 endmodule

```

FPU.sv: archivo que contiene los módulos *FPU* (este a su vez implementa todos los sub-módulos de pertinentes a la unidad de punto flotante, en particular integra: *fp_converter.sv* y *fp_arithmetic_unit.sv*) y *fpu_op_selection*.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps

```

```

6
7 module fpu_op_selection(
8     input [4:0] rs2_add,
9     input [3:0] funct7_out,
10    input [2:0] format_type, sub_format_type, funct3, rm_from_fcsr,
11    output reg [2:0] rm2fpu,
12    output reg [4:0] fpu_option
13 );
14 assign rm2fpu = (funct3 == 3'b111) ? rm_from_fcsr : funct3;
15 always_comb begin
16     if(format_type[2:1]==2'b00)
17         if(format_type[0]) // R4
18             case (sub_format_type)
19                 3'b000: fpu_option = 5'b0_00_00; // fmadd.s
20                 3'b001: fpu_option = 5'b0_00_01; // fmsub.s
21                 3'b010: fpu_option = 5'b0_00_10; // fnmsub.s
22                 3'b011: fpu_option = 5'b0_00_11; // fnmadd.s
23                 default: fpu_option = 5'b1_11_11; // not valid option
24             endcase
25         else // R
26             if(sub_format_type!=3'b000) fpu_option = 5'b1_11_11; // not valid option
27             else
28                 case(funct7_out)
29                     /*-----*/
30                     4'b0000: fpu_option = 5'b 1_00_00; // fadd.s
31                     4'b0011: fpu_option = 5'b 1_00_01; // fsub.s
32                     4'b0100: fpu_option = 5'b 1_00_10; // fmul.s
33                     4'b0101: fpu_option = 5'b 1_00_11; // fdiv.s
34                     4'b0110: fpu_option = 5'b 1_01_00; // fsqrt.s
35                     /*-----*/
36                     4'b0111:case(funct3)
37                         3'b000: fpu_option = 5'b0_11_00; // fsgnj.s
38                         3'b001: fpu_option = 5'b0_11_01; // fsgnjn.s
39                         3'b010: fpu_option = 5'b0_11_10; // fsgnjx.s
40                         default: fpu_option = 5'b1_11_11; // not valid option
41                     endcase
42                     /*-----*/
43                     4'b1000:case(funct3)
44                         3'b000: fpu_option = 5'b0_10_00; // fmin.s
45                         3'b001: fpu_option = 5'b0_10_01; // fmax.s
46                         default: fpu_option = 5'b1_11_11; // not valid option
47                     endcase
48                     /*-----*/
49                     4'b1001:case(rs2_add)
50                         5'b00000: fpu_option = 5'b0_01_00; // fcvt.s.w
51                         5'b00001: fpu_option = 5'b0_01_01; // fcvt.s.wu
52                         default: fpu_option = 5'b1_11_11; // not valid option
53                     endcase
54                     4'b1010:case(rs2_add)
55                         5'b00000: fpu_option = 5'b0_01_10; // fcvt.w.s
56                         5'b00001: fpu_option = 5'b0_01_11; // fcvt.w.u.s
57                         default: fpu_option = 5'b1_11_11; // not valid option

```

```

58         endcase
59         /*-----*/
60         4'b 1101:case(funcnt3)
61             3'b000: fpu_option = 5'b1_10_00; // fle.s
62             3'b001: fpu_option = 5'b1_10_01; // flt.s
63             3'b010: fpu_option = 5'b1_10_10; // feq.s
64             default: fpu_option = 5'b1_11_11; // not valid option
65         endcase
66         /*-----*/
67         4'b1011: fpu_option = (funcnt3==3'b001) ? 5'b0_11_11 : 5'b1_11_11; //
↪ fclass.s
68         /*-----*/
69         default: fpu_option = 5'b1_11_11; // not valid option
70     endcase
71     else fpu_option = 5'b1_11_11; // not valid option
72 end
73 endmodule
74 /*
75 option = f(format_type[2:0], funcnt7_out[3:0], funcnt3[2:0], rs2_option)
76 format_type[2:0]= (only for FPU)
77     000 -> R type
78     001 -> R4 type
79     011 -> S type (mem option, not used here)
80     010 -> I type (mem option, not used here)
81     111 -> Not valid type
82 sub_format_type[2:0]= (only for R4)
83     -000 -> R4 (FP- fmadd.s)
84     -001 -> R4 (FP- fmsub.s)
85     -010 -> R4 (FP- fnmsub.s)
86     -011 -> R4 (FP- fnmadd.s)
87 funcnt7_out[3:0]= (only for FPU)
88     if Funcnt7[6:0]=0000000: Funcnt7-out[3:0]=0000
89     if Funcnt7[6:0]=0000100: Funcnt7-out[3:0]=0011
90     if Funcnt7[6:0]=0001000: Funcnt7-out[3:0]=0100
91     if Funcnt7[6:0]=0001100: Funcnt7-out[3:0]=0101
92     if Funcnt7[6:0]=0101100: Funcnt7-out[3:0]=0110
93     if Funcnt7[6:0]=0010000: Funcnt7-out[3:0]=0111
94     if Funcnt7[6:0]=0010100: Funcnt7-out[3:0]=1000
95     if Funcnt7[6:0]=1101000: Funcnt7-out[3:0]=1001
96     if Funcnt7[6:0]=1100000: Funcnt7-out[3:0]=1010
97     if Funcnt7[6:0]=1110000: Funcnt7-out[3:0]=1011
98     if Funcnt7[6:0]=1111000: Funcnt7-out[3:0]=1100
99     if Funcnt7[6:0]=1010000: Funcnt7-out[3:0]=1101
100 funcnt3[2:0]= 000 | 001 | 010 | rm
101 rm[2:0]=
102     000 -> RNE
103     001 -> RTZ
104     010 -> RDN
105     011 -> RUP
106     100 -> RMM
107     101 -> (Invalid)
108     110 -> (Invalid)

```



```

155     DA_state1, DA_state2, DA_state3, DA_state4, AM_state
156 } state_type;
157 state_type actual_state, next_state;
158 reg sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic;
159 reg [2:0] op_to_arithmetic;
160 wire NX_from_converter, NV_from_converter, NV_from_comparator,
161     ready_from_arithmetic, ready_from_converter, NV_max_min, in1_is_nan,
162     ↪ in2_is_nan,
163     UF_from_arithmetic, OF_from_arithmetic, NV_from_arithmetic,
164     ↪ DZ_from_arithmetic, NX_from_arithmetic,
165     NV_r4_middle, NX_r4_middle, UF_r4_middle, OF_r4_middle, DZ_r4_middle;
166 wire [2:0] status_from_arithmetic;
167 wire [31:0] in1_to_fp_arithmetic_unit, in2_to_fp_arithmetic_unit, r4_middle_out,
168     out_from_arithmetic, out_from_converter, out_from_classifier,
169     ↪ out_from_comparator,
170     sgnj_out, max_min_out;
171 assign in1_is_nan = (in1[30:23] == 8'b11111111 & in1[22:0] != 23'b0) ? 1 : 0;
172 assign in2_is_nan = (in2[30:23] == 8'b11111111 & in2[22:0] != 23'b0) ? 1 : 0;
173 // SG: sgnj
174 assign sgnj_out = {(option[0] ? ~in2[31] : (option[1] ? in1[31]^in2[31] : in2[31])), in1
175     ↪ [30:0]};
176 // MM: max/min /* notese que los inputs op del modulo fp_comparator para ambos
177     ↪ casos (max min) es considerado */
178 /*
179     option[1:0]=01 > fit y max | option[1:0]=00 > fle y min
180
181     option[0]=1 -> lt: in1 < in2 (flag signaling: signaling comparison) flag when Nan in (res
182     ↪ = 0)
183     option[0]=0 -> le: in1 <= in2 (flag signaling: signaling comparison) flag when Nan in (
184     ↪ res = 0)
185 */
186 assign max_min_out = ~(in1_is_nan & in2_is_nan) ? (option[0] ?
187     (out_from_comparator[0] ? (in2_is_nan ? in1 : in2) : (in1_is_nan ? in2 : in1)) : //
188     ↪ max-fit
189     (out_from_comparator[0] ? (in1_is_nan ? in2 : in1) : (in2_is_nan ? in1 : in2)) //
190     ↪ min-fle
191     ) : 32'b0_11111111_100000000000000000000000;
192 assign NV_max_min = ((in1_is_nan&~in1[22]) | (in2_is_nan&~in2[22])) ? 1'b1 : 1'b0;
193 // DA: double arithmetic
194 assign in1_to_fp_arithmetic_unit = sel_arithmetic ? r4_middle_out : in1;
195 assign in2_to_fp_arithmetic_unit = sel_arithmetic ? in3 : in2;
196 assign UF_from_arithmetic = (status_from_arithmetic==3'b001) ? 1'b1 : 1'b0;
197 assign OF_from_arithmetic = (status_from_arithmetic==3'b010) ? 1'b1 : 1'b0;
198 assign NV_from_arithmetic = (status_from_arithmetic==3'b011) ? 1'b1 : 1'b0;
199 assign DZ_from_arithmetic = (status_from_arithmetic==3'b100) ? 1'b1 : 1'b0;
200 assign NX_from_arithmetic = (status_from_arithmetic==3'b111 |
201     status_from_arithmetic==3'b001) ? 1'b1 : 1'b0;
202 /*
203     status =
204     001 -> UF flag
205     010 -> OF flag
206     011 -> NV flag

```

```

198     100 -> DZ flag
199     111 -> NX flag
200 */
201 // FSM
202 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
203 always_comb
204     case (actual_state)
205     default: begin
206         out = 32'b0;
207         {NV, NX, UF, OF, DZ, ready} = 6'b00000_0;
208         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
209         next_state = actual_state;
210     end
211     waiting_state: begin
212         out = 32'b0;
213         {NV, NX, UF, OF, DZ, ready} = 6'b00000_0;
214         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0010, 3'b000};
215         next_state = start ? ((option[4:0]==5'b0_11_11) ? CS_state :
216             ((option[4:1]==4'b0_10_0) ? MM_state :
217             ((option[4:2]==3'b1_10) ? CM_state :
218             ((option[4:2]==3'b0_01) ? CO_state :
219             ((option[4:2]==3'b0_11) ? SG_state :
220             ((option[4:2]==3'b0_00) ? DA_state1 :
221             ((option[4:3]==2'b1_0) ? AM_state :
222             waiting_state)))))) : waiting_state;
223     end
224     CS_state: begin
225         out = out_from_classifier;
226         {NV, NX, UF, OF, DZ, ready} = 6'b00000_1;
227         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
228         next_state = start ? CS_state : waiting_state;
229     end
230     MM_state: begin
231         out = max_min_out;
232         {NV, NX, UF, OF, DZ, ready} = {NV_max_min, 5'b0000_1};
233         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
234         next_state = start ? MM_state : waiting_state;
235     end
236     CM_state: begin
237         out = out_from_comparator;
238         {NV, NX, UF, OF, DZ, ready} = {NV_from_comparator, 5'b0000_1};
239         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
240         next_state = start ? CM_state : waiting_state;
241     end
242     CO_state: begin
243         out = out_from_converter;
244         {NV, NX, UF, OF, DZ, ready} = {NV_from_converter, NX_from_converter, 3'

```

```

↪ b000, ready_from_converter};
245     {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
246     next_state = start ? CO_state : waiting_state;
247     end
248     SG_state: begin
249         out = sgnj_out;
250         {NV, NX, UF, OF, DZ, ready} = 6'b00000_1;
251         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic,
↪ op_to_arithmetic} = {4'b0000, 3'b000};
252         next_state = start ? SG_state : waiting_state;
253     end
254     /*
255     op =
256         000 -> in1+in2  ADD   -1_0 0_00
257         001 -> in1*in2  MUL   -1_0 0_10
258         010 -> in1/in2  DIV   -1_0 0_11
259         011 -> sqrt(in1) SQRT  -1_0 1_00
260         100 -> in1-in2  SUB   -1_0 0_01
261     */
262     AM_state: begin
263         out = out_from_arithmetic;
264         {NV, NX, UF, OF, DZ, ready} = {NV_from_arithmetic, NX_from_arithmetic,
↪ UF_from_arithmetic,
265             OF_from_arithmetic, DZ_from_arithmetic,
↪ ready_from_arithmetic};
266         op_to_arithmetic = (option[2:0]==3'b0_00) ? 3'b000 :
267             ((option[2:0]==3'b0_01) ? 3'b100 :
268             ((option[2:0]==3'b0_10) ? 3'b001 :
269             ((option[2:0]==3'b0_11) ? 3'b010 :
270             ((option[2:0]==3'b1_00) ? 3'b011 :
271             3'bXXX)));
272         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic} =
↪ 4'b0100;
273         next_state = start ? AM_state : waiting_state;
274     end
275     /*
276     fmaddd.s -> option[1:0]=00 -> rd = rs1 * rs2 + rs3
277     fmsub.s  -> option[1:0]=01 -> rd = rs1 * rs2 - rs3
278     fnmsub.s -> option[1:0]=10 -> rd = -rs1 * rs2 + rs3
279     fnmadd.s -> option[1:0]=11 -> rd = -rs1 * rs2 - rs3
280     op =
281         000 -> in1+in2  ADD
282         001 -> in1*in2  MUL
283         100 -> in1-in2  SUB
284         101 -> -in1*in2 -MUL
285     */
286     DA_state1: begin
287         out = 32'b0;
288         {NV, NX, UF, OF, DZ, ready} = {NV_from_arithmetic, NX_from_arithmetic,
↪ UF_from_arithmetic,
289             OF_from_arithmetic, DZ_from_arithmetic, 1'b0};

```

```

290     op_to_arithmetic = option[1] ? 3'b101 : 3'b001;
291     {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic} =
↪ 4'b0100;
292     next_state = ready_from_arithmetic ? DA_state2 : DA_state1;
293     end
294     DA_state2: begin
295         out = 32'b0;
296         {NV, NX, UF, OF, DZ, ready} = {NV_from_arithmetic, NX_from_arithmetic,
↪ UF_from_arithmetic,
297             OF_from_arithmetic, DZ_from_arithmetic, 1'b0};
298         op_to_arithmetic = option[1] ? 3'b101 : 3'b001;
299         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic} =
↪ 4'b0101;
300         next_state = DA_state3;
301     end
302     DA_state3: begin
303         out = 32'b0;
304         {NV, NX, UF, OF, DZ, ready} = 6'b00000_0;
305         op_to_arithmetic = option[0] ? 3'b100 : 3'b000;
306         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic} =
↪ 4'b1010;
307         next_state = DA_state4;
308     end
309     DA_state4: begin
310         out = out_from_arithmetic;
311         {NV, NX, UF, OF, DZ, ready} = {NV_from_arithmetic | NV_r4_middle,
↪ NX_from_arithmetic | NX_r4_middle,
312             UF_from_arithmetic | UF_r4_middle,
↪ OF_from_arithmetic | OF_r4_middle,
313             DZ_from_arithmetic | DZ_r4_middle,
↪ ready_from_arithmetic};
314         op_to_arithmetic = option[0] ? 3'b100 : 3'b000;
315         {sel_arithmetic, start_to_arithmetic, rst_to_arithmetic, set_reg_arithmetic} =
↪ 4'b1100;
316         next_state = start ? DA_state4 : waiting_state;
317     end
318 endcase
319 // AM: arithmetic
320 fp_arithmetic_unit fp_arithmetic_unit_module(
321     //inputs
322     .start(start_to_arithmetic), .rst(rst | rst_to_arithmetic), .clk(clk),
323     .op(op_to_arithmetic), .rm(rm),
324     .in1(in1_to_fp_arithmetic_unit), .in2(in2_to_fp_arithmetic_unit),
325     //outputs
326     .ready(ready_from_arithmetic), .status(status_from_arithmetic), .out(
↪ out_from_arithmetic)
327 );
328 generic_register #(width(32+5)) fp_arithmetic_unit_register(
329     .clk(clk), .reset(1'b0), .load(set_reg_arithmetic), .data_in({NV, NX, UF, OF, DZ,
↪ out_from_arithmetic}),
330     .data_out({NV_r4_middle, NX_r4_middle, UF_r4_middle, OF_r4_middle,
↪ DZ_r4_middle, r4_middle_out})

```

```

331 );
332 // CO: Conversions
333 fp_converter fp_converter_unit(
334     //inputs
335     .start(start), .rst(rst), .clk(clk),
336     .integer_is_signed(~option[0]), .option(option[1]), .rm(rm), .in(in1),
337     /* option: 0 -> Integer to fp | 1 -> fp to Integer */
338     //outputs
339     .NV(NV_from_converter), .NX(NX_from_converter), .ready(ready_from_converter), .
    ↪ out(out_from_converter)
340 );
341 // CS: Class
342 fp_classifier fp_classifier_unit(
343     //inputs
344     .in(in1),
345     //outputs
346     .out(out_from_classifier)
347 );
348 // CM: Comparisson
349 fp_comparator fp_comparator_unit(
350     //inputs
351     .in1_is_nan(in1_is_nan), .in2_is_nan(in2_is_nan),
352     .op((option[0] ? 2'b01 : (option[1] ? 2'b00 : 2'b10))), .in1(in1), .in2(in2),
353     /* op
354     00 -> eq: in1 == in2 (no flag signaling: quiet comparison) no flag when Nan in (res
    ↪ = 0)
355     01 -> lt: in1 < in2 (flag signaling: signaling comparison) flag when Nan in (res = 0)
356     10 -> le: in1 <= in2 (flag signaling: signaling comparison) flag when Nan in (res = 0)
357     11 -> not used but = eq
358     */
359     //outputs
360     .invalid_flag(NV_from_comparator), .out(out_from_comparator)
361 );
362 endmodule
363
364 module fp_classifier(
365     input [31:0] in,
366     output reg [31:0] out
367 );
368 always_comb begin
369     out = 32'b0000_0000_0000_0000_0000_0000_0000;
370     case (in[30:23])
371         default:             if(in[31]) out[1] =1'b1; else out[6] =1'b1;
372         8'b00000000:
373             if(in[22:0]==23'b0) if(in[31]) out[3] =1'b1; else out[4] =1'b1;
374             else if(in[31]) out[2] =1'b1; else out[5] =1'b1;
375         8'b11111111:
376             if(in[22:0]==23'b0) if(in[31]) out[0] =1'b1; else out[7] =1'b1;
377             else if(in[22]) out[9] =1'b1; else out[8] =1'b1;
378     endcase
379 end
380 endmodule

```

```

381
382 module fp_comparator(
383     input in1_is_nan, in2_is_nan,
384     input [1:0] op,
385     input [31:0] in1, in2,
386     output invalid_flag,
387     output [31:0] out
388     /*
389     op =
390     00 -> eq: in1 == in2 (no flag signaling: quiet comparison) no flag when Nan in (res =
    ↪ 0)
391     01 -> lt: in1 < in2 (flag signaling: signaling comparison) flag when Nan in (res = 0)
392     10 -> le: in1 <= in2 (flag signaling: signaling comparison) flag when Nan in (res = 0)
393     11 -> not used but = eq
394     */
395 );
396 reg less_flag;
397 wire equal_flag;
398 assign equal_flag = (in1 == in2) ? 1'b1 : 1'b0;
399 assign invalid_flag = (op[1] ^ op[0]) ? (in1_is_nan | in2_is_nan) : (in1_is_nan & ~in1[22]
    ↪ | in2_is_nan & ~in2[22]);
400 assign out = (op == 2'b01) ? ( (less_flag & ~(in1_is_nan | in2_is_nan))           ?
    ↪ 32'b1 : 32'b0 ) :
401     (op == 2'b10) ? ( ((less_flag | equal_flag) & ~(in1_is_nan | in2_is_nan)) ?
    ↪ 32'b1 : 32'b0 ) :
402     ( (equal_flag & ~(in1_is_nan | in2_is_nan))           ? 32'b1 :
    ↪ 32'b0 ) ;
403 always_comb
404     if(in1[31] == in2[31]) // sig1 = sig2
405         if (unsigned'(in1[30:23]) < unsigned'(in2[30:23])) less_flag = in1[31] ? 1'b0 : 1'b1;
406         else if(unsigned'(in1[30:23]) > unsigned'(in2[30:23])) less_flag = in1[31] ? 1'b1 : 1'b0;
407         else // exp1 = exp2
408             if (unsigned'(in1[22:0]) < unsigned'(in2[22:0])) less_flag = in1[31] ? 1'b0 : 1'b1
    ↪ ;
409             else if(unsigned'(in1[22:0]) > unsigned'(in2[22:0])) less_flag = in1[31] ? 1'b1 : 1'
    ↪ b0;
410             else less_flag = 1'b0; // in1 = in2
411         else // sig1 != sig2
412             less_flag = in1[31];
413 endmodule

```

fp_converter.sv: archivo *top* del módulo *fp_converter* y que incluye todos los submódulos del mismo.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6
7 module fp_converter(
8     input start, rst, clk, integer_is_signed, option, //option: 0 -> Integer to fp | 1 -> fp to

```

```

    ↪ Integer
9  input [2:0] rm,
10 input [31:0] in,
11 output NV, NX, ready, // NV: invalid operation NX: inexact
12 output [31:0] out
13 /*
14 rm =
15     000 -> RNE: To nearest, ties to even
16     001 -> RTZ: Toward 0
17     010 -> RDN: Toward -inf
18     011 -> RUP: Toward +inf
19     100 -> RMM: To nearest, ties away from zero
20 */
21 );
22 typedef enum {waiting_state,
23               initial_state,
24               pre_iteration_state_1,
25               pre_iteration_state_2,
26               iteration_state_1,
27               iteration_state_2,
28               iteration_state_3,
29               pre_final_state_1,
30               pre_final_state_2,
31               final_state
32 } state_type;
33 state_type actual_state, next_state;
34 reg sel, set, reset, start_norm, ready_from_controller;
35 wire start_ctrl, exception_flag, norm_ready, fp_sig, still_norm, NX_fp2, NX_2fp,
    ↪ NX_check;
36 wire [7:0] initial_fp_exp, exp_to_normalizer, exp_from_reg, exp_from_norm,
    ↪ exp_from_rounder;
37 wire [31:0] integer_res, fp_res, exception_res, initial_fp_man, man_to_normalizer,
    man_from_reg, man_from_norm, man_from_rounder;
38 assign start_ctrl = option ? 1'b0 : start;
39 assign ready = (option | exception_flag) ? start : ready_from_controller;
40 assign out = exception_flag ? exception_res : (option ? integer_res : fp_res);
41 assign {exp_to_normalizer, man_to_normalizer} = sel ? {exp_from_reg, man_from_reg
    ↪ } : {initial_fp_exp, initial_fp_man};
42 assign fp_res = {fp_sig, exp_from_reg, man_from_reg[30:8]};
43 assign NX = (NX_fp2 & option) | (NX_2fp & ~option) | NX_check) ? 1'b1 : 1'b0;
44 // FSM
45 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
46 always_comb begin
47     case(actual_state)
48     default: begin
49         {sel, set, reset, start_norm, ready_from_controller} = 5'b00000;
50         next_state = actual_state;
51     end
52     waiting_state: begin
53         {sel, set, reset, start_norm, ready_from_controller} = 5'b00100;
54         next_state = start_ctrl ? initial_state : waiting_state;
55     end
56 end

```

```

57     initial_state: begin
58         {sel, set, reset, start_norm, ready_from_controller} = 5'b00010;
59         next_state = norm_ready ? (still_norm ? pre_final_state_1 :
↪ pre_iteration_state_1) : initial_state;
60     end
61     pre_iteration_state_1: begin
62         {sel, set, reset, start_norm, ready_from_controller} = 5'b01010;
63         next_state = pre_iteration_state_2;
64     end
65     pre_iteration_state_2: begin
66         {sel, set, reset, start_norm, ready_from_controller} = 5'b10000;
67         next_state = iteration_state_1;
68     end
69     iteration_state_1: begin
70         {sel, set, reset, start_norm, ready_from_controller} = 5'b10010;
71         next_state = norm_ready ? (still_norm ? pre_final_state_2 : iteration_state_2
↪ ) : iteration_state_1;
72     end
73     iteration_state_2: begin
74         {sel, set, reset, start_norm, ready_from_controller} = 5'b11010;
75         next_state = iteration_state_3;
76     end
77     iteration_state_3: begin
78         {sel, set, reset, start_norm, ready_from_controller} = 5'b10000;
79         next_state = iteration_state_1;
80     end
81     pre_final_state_1: begin
82         {sel, set, reset, start_norm, ready_from_controller} = 5'b01010;
83         next_state = final_state;
84     end
85     pre_final_state_2: begin
86         {sel, set, reset, start_norm, ready_from_controller} = 5'b11010;
87         next_state = final_state;
88     end
89     final_state: begin
90         {sel, set, reset, start_norm, ready_from_controller} = 5'b00001;
91         next_state = start_ctrl ? final_state : waiting_state;
92     end
93 endcase
94 end
95 converter_exceptions_checker converter_exceptions_checker_module(
96     .integer_is_signed(integer_is_signed), .option(option), .rm(rm), .in(in),
97     .exception_flag(exception_flag), .invalid_flag(NV), .inexact_flag(NX_check), .
↪ exception_res(exception_res)
98 );
99 fp_to_integer_unit fp_to_integer_unit_module(
100     .integer_is_signed(integer_is_signed), .rm(rm), .fp_in(in),
101     .NX(NX_fp2), .integer_out(integer_res)
102 );
103 integer_to_fp_unit integer_to_fp_unit_module(
104     .integer_is_signed(integer_is_signed), .integer_in(in),
105     .sig_out(fp_sig), .exp_out(initial_fp_exp), .man_out(initial_fp_man)

```

```

106 );
107 normalizer_integer2fp normalizer_integer2fp_module(
108     .start(start_norm), .rst(rst), .clk(clk), .exp_in(exp_to_normalizer), .man_in(
109     ↪ man_to_normalizer),
110     .ready(norm_ready), .exp_out(exp_from_norm), .man_out(man_from_norm)
111 );
112 rounder_integer2fp rounder_integer2fp_module(
113     .sig_in(fp_sig), .rm(rm), .exp_in(exp_from_norm), .man_in(man_from_norm),
114     .still_norm(still_norm), .NX(NX_2fp), .exp_out(exp_from_rounder), .man_out(
115     ↪ man_from_rounder)
116 );
117 generic_register #(40) converter_register(
118     .clk(clk), .reset(reset), .load(set), .data_in({exp_from_rounder, man_from_rounder}),
119     .data_out({exp_from_reg, man_from_reg})
120 );
121 endmodule
122
123 module converter_exceptions_checker(
124     input integer_is_signed, option,
125     input [2:0] rm,
126     input [31:0] in,
127     output reg exception_flag, invalid_flag, inexact_flag,
128     output [31:0] exception_res
129 );
130 reg [2:0] sel;
131 wire in_as_fp_is_nan, in_as_fp_is_zero, exp_lt_0, exp_gt_30, exp_gt_31,
132     ↪ in_is_zeros, in_is_ones;
133 assign in_as_fp_is_nan = (in[30:23]==8'b11111111 & in[22:0]!=23'b0) ? 1'b1 : 1'b0;
134 assign in_as_fp_is_zero = (in[30:0]==31'b0) ? 1'b1 : 1'b0;
135 assign exp_lt_0 = (unsigned'(in[30:23])<unsigned'(8'b01111111)) ? 1'b1 : 1'b0;
136 assign exp_gt_30 = (unsigned'(in[30:23])>unsigned'(8'b10011101)) ? 1'b1 : 1'b0;
137 assign exp_gt_31 = (unsigned'(in[30:23])>unsigned'(8'b10011110)) ? 1'b1 : 1'b0;
138 assign in_is_zeros = (in==32'b0) ? 1'b1 : 1'b0;
139 assign in_is_ones = (in==32'b11111111111111111111111111111111) ? 1'b1 : 1'b0;
140 assign invalid_flag = (option & ( in_as_fp_is_nan | (integer_is_signed & exp_gt_30) |
141     ↪ (~integer_is_signed & (exp_gt_31 | in[31]) ) )) ? 1'b1 : 1'b0;
142 assign inexact_flag = option & exp_lt_0;
143 // Opciones de salidas triviales
144 assign exception_res = (sel==3'b000) ? 32'b10000000000000000000000000000000 : // (
145     ↪ signed) -231 = fp(-0)
146     (sel==3'b001) ? 32'b01111111111111111111111111111111 : // (signed) 2
147     ↪ 31-1
148     (sel==3'b010) ? 32'b11111111111111111111111111111111 : // (unsigned
149     ↪ ) 232-1 = (signed) -1
150     (sel==3'b011) ? 32'b101111111100000000000000000000000000000000 : // fp(-1)
151     (sel==3'b100) ? 32'b000000000000000000000000000000000000000000 : // (un/
152     ↪ signed) 0 = fp(+0)
153     (sel==3'b101) ? 32'b000000000000000000000000000000000000000001 : // (un/
154     ↪ signed) 1
155     32'bX;
156 always_comb
157     if(option) // fp2integer

```

```

149     if(in_as_fp_is_zero) {sel, exception_flag} = {3'b100, 1'b1};
150     else if(in_as_fp_is_nan) {sel, exception_flag} = {(integer_is_signed ? 3'b001 : 3'
↪ b010), 1'b1};
151     else if(exp_lt_0)
152         case(rm)
153             3'b010: {sel, exception_flag} = {(integer_is_signed & in[31]) ? 3'b010 : 3'
↪ b100), 1'b1}; // RDN
154             3'b011: {sel, exception_flag} = {(in[31] ? 3'b100 : 3'b101), 1'b1};
↪ // RUP
155             3'b001: {sel, exception_flag} = {3'b100, 1'b1};
↪ // RTZ
156             default: {sel, exception_flag} = {( (unsigned'(in[30:23])==unsigned'(8'
↪ b01111110)) ?
157                 (in[31] ? (integer_is_signed ? 3'b010 : 3'b100) : 3'b101) : 3'b100 ), 1'b1};
↪ // RNE - RMM
158         endcase
159     else if(integer_is_signed) {sel, exception_flag} = exp_gt_30 ? {(in[31] ? 3'b000 : 3'
↪ b001), 1'b1} : {3'bX, 1'b0};
160     else {sel, exception_flag} = in[31] ? {3'b100, 1'b1} : (exp_gt_31 ? {3'b010, 1'b1} :
↪ {3'bX , 1'b0} ); // integer_is_signed=0
161     else begin // integer2fp
162         sel = in_is_zeros ? 3'b100 : ((integer_is_signed & in_is_ones) ? 3'b011 : 3'bX);
163         exception_flag = (in_is_zeros | (integer_is_signed & in_is_ones)) ? 1'b1 : 1'b0;
164     end
165 endmodule
166
167 module fp_to_integer_unit(
168     input integer_is_signed,
169     input [2:0] rm,
170     input [31:0] fp_in,
171     output NX,
172     output [31:0] integer_out
173 );
174 wire [54:0] from_shifter;
175 wire [31:0] interger_rounded;
176 assign from_shifter = {32'b1, fp_in[22:0]} << unsigned'(fp_in[30:23]) - unsigned'(8'
↪ b01111111);
177 assign integer_out = (integer_is_signed & fp_in[31]) ? ~interger_rounded+32'b1 :
↪ interger_rounded;
178 integer_rounder integer_rounder_unit(
179     .is_signed(integer_is_signed), .sig_in(fp_in[31]), .rm(rm), .integer_in(from_shifter),
180     .NX(NX), .integer_out(interger_rounded)
181 );
182 endmodule
183
184 module integer_rounder(
185     input is_signed, sig_in,
186     input [2:0] rm,
187     input [54:0] integer_in,
188     output NX,
189     output [31:0] integer_out
190 );

```

```

191 reg sel;
192 wire sticky, is_overflow;
193 wire [2:0] rounding_bits;
194 wire [32:0] temp_out; // se añade bit en caso de overflow para unsigned integer
195 // Se obtienen rounding bits
196 assign sticky = (integer_in[20:0] != 0) ? 1'b1 : 1'b0;
197 assign rounding_bits = {integer_in[22:21], sticky};
198 // Salida falg NX
199 assign NX = (rounding_bits == 3'b0) ? 1'b0 : 1'b1;
200 // Se redondea
201 assign temp_out = sel ? {1'b0, integer_in[54:23]}+33'b1 : {1'b0, integer_in[54:23]};
202 // Se chequea overflow según si es signed o no
203 assign is_overflow = is_signed ? (temp_out[32:31]!=2'b0 ? 1'b1 : 1'b0)
204 : (temp_out[ 32 ] !=1'b0 ? 1'b1 : 1'b0);
205 // Se obtiene resultado final
206 assign integer_out = is_overflow ?
207 (is_signed ? 32'b01111111111111111111111111111111 : 32'
208 ↪ b11111111111111111111111111111111)
209 : temp_out[31:0];
210 // Logica combinacional de redondeo
211 always_comb
212 case(rounding_bits) // grs
213 3'b000: sel = 1'b0;
214 default: case(rm)
215 3'b000: sel = (rounding_bits[1:0]==2'b0) ? (integer_in[23] ? 1'b1 : 1'b0) : //
216 ↪ RNE
217 (rounding_bits[2] ? 1'b1 : 1'b0);
218 3'b001: sel = 1'b0; // RTZ
219 3'b010: sel = sig_in ? 1'b1 : 1'b0; // RDN
220 3'b011: sel = ~sig_in ? 1'b1 : 1'b0; // RUP
221 3'b100: sel = (rounding_bits[1:0]==2'b0) ? 1'b1 :
222 ↪ RMM
223 (rounding_bits[2] ? 1'b1 : 1'b0);
224 default: sel = 1'b0;
225 endcase
226 endcase
227 endmodule
228
229 module integer_to_fp_unit(
230 input integer_is_signed,
231 input [31:0] integer_in,
232 output sig_out,
233 output [7:0] exp_out,
234 output [31:0] man_out
235 );
236 assign sig_out = (integer_is_signed & (signed'(integer_in)<0)) ? 1'b1 : 1'b0;
237 assign exp_out = 8'b10011110; // = 158 = 127+31
238 assign man_out = sig_out ? ~integer_in+32'b1 : integer_in;
239 endmodule
240
241 module normalizer_integer2fp(
242 // Input

```

```

240 input start, rst, clk,
241 input [7:0] exp_in,
242 input [31:0] man_in,
243 // Output
244 output reg ready,
245 output [7:0] exp_out,
246 output [31:0] man_out
247 );
248 typedef enum {
249     waiting_state, initial_state, iteration_state, final_state
250 } state_type;
251 state_type actual_state, next_state;
252 reg sel, set, reset;
253 wire done;
254 wire [7:0] new_exp;
255 wire [31:0] new_man;
256 wire [39:0] to_reg;
257 assign done = (man_out[31] == 1'b1 | man_out == 0) ? 1'b1 : 1'b0;
258 assign new_exp = unsigned'(exp_out) - unsigned'(8'b1);
259 assign new_man = man_out << 1;
260 assign to_reg = sel ? {new_exp, new_man} : {exp_in, man_in};
261 // FSM
262 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
263 always_comb begin
264     case(actual_state)
265         default: begin
266             {sel, set, reset, ready} = 4'b0000;
267             next_state = actual_state;
268         end
269         waiting_state: begin
270             {sel, set, reset, ready} = 4'b0010;
271             next_state = start ? initial_state : waiting_state;
272         end
273         initial_state: begin
274             {sel, set, reset, ready} = 4'b0100;
275             next_state = done ? final_state : iteration_state;
276         end
277         iteration_state: begin
278             {sel, set, reset, ready} = 4'b1100;
279             next_state = done ? final_state : iteration_state;
280         end
281         final_state: begin
282             {sel, set, reset, ready} = 4'b0001;
283             next_state = start ? final_state : waiting_state;
284         end
285     endcase
286 end
287 // registers
288 generic_register #(40) register_out_unit(
289     .clk(clk), .reset(reset), .load(set), .data_in(to_reg),
290     .data_out({exp_out, man_out})
291 );

```

```

292 endmodule
293
294 module rounder_integer2fp(
295     input sig_in,
296     input [2:0] rm,
297     input [7:0] exp_in,
298     input [31:0] man_in,
299     /*
300     rm =
301         000 -> RNE: To nearest, ties to even
302         001 -> RTZ: Toward 0
303         010 -> RDN: Toward -inf
304         011 -> RUP: Toward +inf
305         100 -> RMM: To nearest, ties away from zero
306     */
307     output still_norm, NX,
308     output [7:0] exp_out,
309     output [31:0] man_out
310 );
311 reg sel;
312 wire sticky;
313 wire [2:0] grs;
314 wire [32:0] temp;
315 assign exp_out = temp[32] ? unsigned'(exp_in)+8'b1 : exp_in;
316 assign sticky = (man_in[5:0] != 6'b0) ? 1'b1 : 1'b0;
317 assign grs = {man_in[7:6], sticky};
318 assign NX = (grs != 3'b0) ? 1'b1 : 1'b0;
319 assign still_norm = (man_out[31]==1'b1 | man_out==0) ? 1'b1 : 1'b0;
320 assign man_out = temp[32] ? temp[32:1] : temp[31:0];
321 assign temp = sel ? {1'b0, man_in}+33'b100000000 : {1'b0, man_in};
322 always_comb
323     case(grs) // grs
324         3'b000: sel = 1'b0;
325         default: case(rm)
326             3'b000: sel = (grs[1:0]==2'b0) ? (man_in[8] ? 1'b1 : 1'b0) : // RNE: To
327                 ↪ nearest, ties to even
328                         (grs[2] ? 1'b1 : 1'b0);
329             3'b001: sel = 1'b0; // RTZ: Toward 0
330             3'b010: sel = sig_in ? 1'b1 : 1'b0; // RDN: Toward -inf
331             3'b011: sel = ~sig_in ? 1'b1 : 1'b0; // RUP: Toward +inf
332             3'b100: sel = (grs[1:0]==2'b0) ? (1'b1) : // RMM: To nearest,
333                 ↪ ties away from zero
334                         (grs[2] ? 1'b1 : 1'b0);
335         default: sel = 1'b0;
336     endcase
337 endcase
338 endmodule

```

fp_arithmetic_unit.sv: archivo *top* del módulo *fp_arithmetic_unit* y que incluye todos los submódulos del mismo.

1 /*

```

2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6
7 /***** Top module: begin *****/
8 module fp_arithmetic_unit(
9     input start, rst, clk, // start: Need to be on til the op is ready
10    input [2:0] op, rm,
11    input [31:0] in1, in2,
12    /*
13        op =
14        000 -> in1+in2  ADD
15        001 -> in1*in2  MUL
16        010 -> in1/in2  DIV
17        011 -> sqrt(in1) SQRT
18        100 -> in1-in2  SUB
19        101 -> -in1*in2 -MUL
20        110 -> -in1/in2 -DIV
21        rm =
22        000 -> RNE
23        001 -> RTZ
24        010 -> RDN
25        011 -> RUP
26        100 -> RMM
27    */
28    output reg ready, // On when the operation is ready
29    output reg [2:0] status,
30    output [31:0] out
31    /*
32        status =
33        000 -> no flag
34        101 -> no flag
35        110 -> no flag
36        -----
37        001 -> UF flag
38        010 -> OF flag
39        011 -> NV flag
40        100 -> DZ flag
41        111 -> NX flag
42    */
43    );
44    typedef enum {
45        waiting_state,
46        initial_state,
47        pre_iteration_state,
48        iteration_state_1,
49        iteration_state_2,
50        iteration_state_3,
51        exception_state_1,
52        exception_state_2,
53        exception_state_3,

```

```

54     final_exception_state,
55     pre_final_state_1,
56     pre_final_state_2,
57     final_state
58 } state_type;
59 state_type actual_state, next_state;
60 reg sel, set, start_norm;
61 reg [2:0] sel_out;
62 wire sig1, sig2, sig, sig_from_adder, start_to_ctrl, ready_from_checker,
    ↪ ready_from_norm, ready_from_add, ready_from_mul, ready_from_div,
    ↪ ready_from_sqrt, still_norm, inexact_flag, check_flag, is_normal, is_uf, is_of,
    ↪ is_desnorm;
63 wire [1:0] status_from_norm;
64 wire [2:0] status_checker, sel_out_checker;
65 wire [7:0] exp1, exp2, exp_to_norm, exp_from_reg, exp_from_adder, exp_from_mul,
    ↪ exp_from_div, exp_from_sqrt, exp_from_norm;
66 wire [27:0] man1, man2, man_to_norm, man_from_reg, man_from_adder,
    ↪ man_from_mul, man_from_div, man_from_sqrt, man_from_norm,
    ↪ man_from_rounder;
67 // Obtaining full inputs: input_decoder
68 assign sig2 = op[2] ? ~in2[31] : in2[31];
69 assign {sig1, exp1, exp2, man1[27], man2[27], man1[25:0], man2[25:0]}
70     = {in1[31:23], in2[30:23], 2'b0, in1[22:0], 3'b0, in2[22:0], 3'b0};
71 assign man1[26] = (exp1 == 8'b0) ? 1'b0 : 1'b1;
72 assign man2[26] = (exp2 == 8'b0) ? 1'b0 : 1'b1;
73 // Mux inputs to normalizer
74 assign {exp_to_norm, man_to_norm} = sel ? {exp_from_reg, man_from_reg} :
75     ((op[1:0] == 2'b00) ? {exp_from_adder, man_from_adder} :
76     ↪ // add
77     ((op[1:0] == 2'b01) ? {exp_from_mul, man_from_mul} :
78     ↪ // mul
79     ((op[1:0] == 2'b10) ? {exp_from_div, man_from_div} : //
80     ↪ div
81     ((op[1:0] == 2'b11) ? {exp_from_sqrt, man_from_sqrt} :
82     ↪ // sqrt
83     36'bX)))));
84 // Mux start to controller
85 assign start_to_ctrl = ((op[1:0] == 2'b00) ? ((ready_from_add | ~check_flag) *
86     ↪ ready_from_checker) : // add
87     ((op[1:0] == 2'b01) ? ((ready_from_mul | ~check_flag) *
88     ↪ ready_from_checker) : // mul
89     ((op[1:0] == 2'b10) ? ((ready_from_div | ~check_flag) *
90     ↪ ready_from_checker) : // div
91     ((op[1:0] == 2'b11) ? ((ready_from_sqrt | ~check_flag) *
92     ↪ ready_from_checker) : // sqrt
93     1'bx)))));
94 // Mux sig
95 assign sig = (op[1:0] == 2'b00) ? sig_from_adder : // add
96     ((op[1:0] == 2'b11) ? 1'b0 : // sqrt
97     sig1^sig2); // mul or div
98 // OUT
99 assign out = (sel_out == 3'b000) ? {sig, exp_from_reg, man_from_reg[25:3]} : //

```



```

139     end
140     iteration_state_3: begin
141         {sel, set, start_norm} = 3'b111;
142         {sel_out, status, ready} = {3'b000, 3'b000, 1'b0};
143         next_state = iteration_state_1;
144     end
145     exception_state_1: begin
146         {sel, set, start_norm} = 3'b000;
147         {sel_out, status, ready} = {sel_out_checker, status_checker, 1'b1};
148         next_state = start ? exception_state_1 : waiting_state;
149     end
150     exception_state_2: begin
151         {sel, set, start_norm} = 3'b011;
152         {sel_out, status, ready} = {3'b000, 3'b000, 1'b0};
153         next_state = final_exception_state;
154     end
155     exception_state_3: begin
156         {sel, set, start_norm} = 3'b111;
157         {sel_out, status, ready} = {3'b000, 3'b000, 1'b0};
158         next_state = final_exception_state;
159     end
160     final_exception_state: begin
161         {sel, set, start_norm} = 3'b000; ready = 1'b1;
162         if(is_desnorm) {sel_out, status} = {3'b000, 3'b001}; // desnorm ==> uf ==> NX
163         else if(is_of) begin
164             // OF flag - "inf" or "almost inf"
165             status = 3'b010;
166             sel_out = (rm == 3'b001) ? 3'b011 : // RTZ
167                 ((rm == 3'b010) ? (sig ? 3'b110 : 3'b011) : // RDN
168                 ((rm == 3'b011) ? (~sig ? 3'b110 : 3'b011) : // RUP
169                 3'b110)); // RNE - RMM
170         end
171         else if(is_uf) begin
172             // Uf flag - "zero" or "almost zero"
173             status = 3'b001;
174             sel_out = (rm == 3'b001) ? 3'b100 : // RTZ
175                 ((rm == 3'b010) ? (sig ? 3'b101 : 3'b100) : // RDN
176                 ((rm == 3'b011) ? (~sig ? 3'b101 : 3'b100) : // RUP
177                 3'b100)); // RNE - RMM
178         end
179         else {sel_out, status} = 6'bX;
180         next_state = start ? final_exception_state : waiting_state;
181     end
182     pre_final_state_1: begin
183         {sel, set, start_norm} = 3'b011;
184         {sel_out, status, ready} = {3'b000, 3'b000, 1'b0};
185         next_state = final_state;
186     end
187     pre_final_state_2: begin
188         {sel, set, start_norm} = 3'b111;
189         {sel_out, status, ready} = {3'b000, 3'b000, 1'b0};
190         next_state = final_state;

```

```

191     end
192     final_state: begin
193         {sel, set, start_norm} = 3'b000;
194         {sel_out, ready} = {3'b000, 1'b1};
195         status = inexact_flag ? 3'b111 : 3'b000;
196         next_state = start ? final_state : waiting_state;
197     end
198 endcase
199 end
200 /** Registers **/
201 generic_register #(.width(36)) res_register(
202     .clk(clk), .reset(1'b0), .load(set), .data_in({exp_from_norm, man_from_rounder}),
203     .data_out({exp_from_reg, man_from_reg})
204 );
205 /** Normalizer and Rounder **/
206 NORMALIZER normalizer_unit(
207     .start(start_norm), .rst(rst), .clk(clk), .exp_in(exp_to_norm), .man_in(
208     ↪ man_to_norm),
209     .ready(ready_from_norm), .status(status_from_norm), .exp_out(exp_from_norm), .
210     ↪ man_out(man_from_norm)
211 );
212 ROUNDER rounder_unit(
213     .rm(rm), .sig_in(sig), .man_in(man_from_norm),
214     .still_norm(still_norm), .inexact_flag(inexact_flag), .man_out(man_from_rounder)
215 );
216 /** Exceptions and trivial cases checker **/
217 excep_triv_checker excep_triv_checker_unit(
218     .start(start), .clk(clk), .rm(rm), .op(op[1:0]), .sig1(sig1), .sig2(sig2), .exp1(exp1), .exp2(
219     ↪ exp2), .frac1(man1[26:3]), .frac2(man2[26:3]),
220     .ready(ready_from_checker), .status_out(status_checker), .sel_out(sel_out_checker)
221 );
222 /** Arithmetic Units **/
223 ADDER adder_unit(
224     .start(start), .clk(clk), .sig1(sig1), .sig2(sig2), .exp1(exp1), .exp2(exp2), .man1(man1), .
225     ↪ man2(man2),
226     .ready(ready_from_add), .sig_out(sig_from_adder), .exp_out(exp_from_adder), .
227     ↪ man_out(man_from_adder)
228 );
229 MULTIPLIER multiplier_unit(
230     .start(start), .clk(clk), .exp1(exp1), .exp2(exp2), .man1(man1), .man2(man2),
231     .ready(ready_from_mul), .exp_out(exp_from_mul), .man_out(man_from_mul)
232 );
233 DIVISOR divisor_unit(
234     .start(start), .rst(rst), .clk(clk), .exp1(exp1), .exp2(exp2), .man1(man1), .man2(man2),
235     .ready(ready_from_div), .exp_out(exp_from_div), .man_out(man_from_div)
236 );
237 Sqrt sqrt_unit(
238     .start(start), .rst(rst), .clk(clk), .exp_in(exp1), .man_in(man1),
239     .ready(ready_from_sqrt), .exp_out(exp_from_sqrt), .man_out(man_from_sqrt)
240 );
241 endmodule
242 /** Top module: end *****/

```

```

238
239 /***** Normalizer and Rounder: begin *****/
240 module NORMALIZER(
241     input start, rst, clk,
242     input [7:0] exp_in,
243     input [27:0] man_in,
244     output reg ready,
245     output [1:0] status,
246     output [7:0] exp_out,
247     output [27:0] man_out
248     /*
249         status =
250             00 -> normalized
251             01 -> UF
252             10 -> OF
253             11 -> desnormalized
254     */
255 );
256 typedef enum {
257     waiting_state, initial_state, iteration_state, final_state
258 } state_type;
259 state_type actual_state, next_state;
260 reg sel, set;
261 wire done, exp_valid;
262 wire [7:0] new_exp, exp_to_reg;
263 wire [27:0] new_man, man_to_reg, desp_man_izq, temp_man_out;
264 // flags
265 assign done      = (temp_man_out[27:26] == 2'b01) ? 1 : 0;
266 assign exp_valid = (status      == 2'b00) ? 1 : 0;
267 // new normalized value
268 assign new_exp = (temp_man_out[27] == 1'b1) ? (unsigned'(exp_out)+unsigned'(8'b1)
    ↪ ) :
269         ((temp_man_out[26] == 1'b0) ? (unsigned'(exp_out)-unsigned'(8'b1)) :
    ↪ exp_out);
270 assign new_man = (temp_man_out[27] == 1'b1) ? (temp_man_out[0] ? {(
    ↪ temp_man_out[27:1] >> 1), temp_man_out[0]} : (temp_man_out >> 1)) :
271         ((temp_man_out[26] == 1'b0) ? (temp_man_out << 1) : temp_man_out);
272 // mux to register
273 assign {exp_to_reg, man_to_reg} = sel ? {new_exp, new_man} : {exp_in, man_in};
274 // status out
275 assign status = (exp_out==8'b11111111 & (man_in[27] | man_in[26])) ? 2'b10 :
276         ((exp_out==8'b0) ? ((temp_man_out[27:1]==27'b0) ? 2'b01 : 2'b11) : 2'b00)
    ↪ ;
277 assign man_out = (exp_out==8'b0 & done) ? (temp_man_out>>1) : temp_man_out;
278 // FSM
279 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
280 always_comb begin
281     case(actual_state)
282     default: begin
283         {sel, set, ready} = 3'b000;
284         next_state = actual_state;
285     end

```

```

286     waiting_state: begin
287         {sel, set, ready} = 3'b000;
288         next_state = start ? initial_state : waiting_state;
289     end
290     initial_state: begin
291         {sel, set, ready} = 3'b010;
292         next_state = exp_valid ? (done ? final_state : iteration_state) : final_state;
293     end
294     iteration_state: begin
295         {sel, set, ready} = 3'b110;
296         next_state = exp_valid ? (done ? final_state : iteration_state) : final_state;
297     end
298     final_state: begin
299         {sel, set, ready} = 3'b001;
300         next_state = start ? final_state : waiting_state;
301     end
302 endcase
303 end
304 // register
305 generic_register #(.width(36)) register_out_unit(
306     .clk(clk), .reset(1'b0), .load(set), .data_in({exp_to_reg, man_to_reg}),
307     .data_out({exp_out, temp_man_out})
308 );
309 endmodule
310
311 module ROUNDER(
312     input [2:0] rm,
313     input sig_in,
314     input [27:0] man_in,
315     /*
316         rm =
317         000 -> RNE
318         001 -> RTZ
319         010 -> RDN
320         011 -> RUP
321         100 -> RMM
322     */
323     output still_norm, inexact_flag,
324     output [27:0] man_out
325 );
326 reg sel;
327 assign inexact_flag = (man_in[2:0] != 3'b0) ? 1'b1 : 1'b0;
328 assign still_norm = (man_out[27:26]==2'b01) ? 1'b1 : 1'b0;
329 assign man_out = sel ? {man_in[27:3], 3'b0}+28'b1000 : {man_in[27:3], 3'b0};
330 always_comb
331     case(man_in[2:0]) // grs
332     3'b000: sel = 1'b0;
333     default: case(rm)
334         3'b000: sel = (man_in[1:0]==2'b0) ? (man_in[3] ? 1'b1 : 1'b0) : // RNE
335             ((man_in[2] ? 1'b1 : 1'b0));
336         3'b001: sel = 1'b0; // RTZ
337         3'b010: sel = sig_in ? 1'b1 : 1'b0; // RDN

```

```

338             3'b011: sel = ~sig_in ? 1'b1 : 1'b0;           // RUP
339             3'b100: sel = (man_in[1:0]==2'b0) ? (1'b1) :   // RMM
340                 ((man_in[2] ? 1'b1 : 1'b0));
341             default: sel = 1'b0;
342         endcase
343     endcase
344 endmodule
345 /***** Normalizer and Rounder: end *****/
346
347 /***** Arithmetic Units: begin *****/
348 /*----- ADDER BEGIN -----*/
349 module ADDER(
350     input start, clk,
351     input sig1, sig2,
352     input [7:0] exp1, exp2,
353     input [27:0] man1, man2,
354     output ready,
355     output sig_out,
356     output [7:0] exp_out,
357     output [27:0] man_out
358 );
359     wire ready1, ready2, shift1;
360     wire [1:0] add_op;
361     wire [7:0] temp_exp_out;
362     wire [9:0] exp_difference;
363     wire [4:0] offset1, offset2;
364     wire [26:0] man1_normalized, man2_normalized;
365     wire [27:0] man_to_shifter, man_from_shifter, man_no_shifted, temp_man_out;
366     assign ready = (ready1 & ready2) ? 1'b1 : 1'b0;
367     //Comparador de exponentes
368     assign shift1 = ((signed'({2'b0, exp1})-signed'({5'b0, offset1}))<(signed'({2'b0, exp2})-
        ↪ signed'({5'b0, offset2}))) ? 1 : 0;
369     // Restador de exponentes
370     assign exp_difference = {exp1,exp2}==0 ? 0 : (shift1 ? ((signed'({2'b0, exp2})-signed
        ↪ '({5'b0, offset2}))-signed'({2'b0, exp1})-signed'({5'b0, offset1}))) : ((signed'({2'b0,
        ↪ exp1})-signed'({5'b0, offset1}))-signed'({2'b0, exp2})-signed'({5'b0, offset2})));
371     // Identificador de operación
372     assign add_op = shift1 ? {sig1, sig2} : {sig2, sig1};
373     // Mux to shifter
374     assign man_to_shifter = shift1 ? ({exp1,exp2}==0 ? man1 : {1'b0, man1_normalized})
        ↪ : ({exp1,exp2}==0 ? man2 : {1'b0, man2_normalized});
375     // Man direct to adder
376     assign man_no_shifted = ~shift1 ? ({exp1,exp2}==0 ? man1 : {1'b0, man1_normalized})
        ↪ : ({exp1,exp2}==0 ? man2 : {1'b0, man2_normalized});
377     // Outputs
378     assign temp_exp_out = {exp1,exp2}==0 ? 0 : (~shift1 ? exp1-offset1 : exp2-offset2);
379     assign exp_out = (temp_exp_out==0 & temp_man_out[27:26]!=2'b0) ? 8'b1 :
        ↪ temp_exp_out;
380     assign man_out = (temp_exp_out==0 & temp_man_out[27:26]!=2'b0) ? (
381         temp_man_out[27] ? (temp_man_out>>1) : temp_man_out
382         ) : temp_man_out;
383     shifter_right shifter_right_unit(

```

```

384     .exp_difference(exp_difference), .man_input(man_to_shifter),
385     .man_output(man_from_shifter)
386 );
387 significands_adder significands_adder_unit(
388     .add_op(add_op), .man_in_1(man_from_shifter), .man_in_2(man_no_shifted),
389     .sig_r(sig_out), .man_output(temp_man_out)
390 );
391 normalizer4significands_inputs normalizer4significands_inputs_adder1(
392     .start(start), .clk(clk), .man_in(man1[26:0]),
393     .ready(ready1), .man_out(man1_normalized), .offset(offset1)
394 );
395 normalizer4significands_inputs normalizer4significands_inputs_adder2(
396     .start(start), .clk(clk), .man_in(man2[26:0]),
397     .ready(ready2), .man_out(man2_normalized), .offset(offset2)
398 );
399 endmodule
400
401 module shifter_right(
402     input[9:0] exp_difference,
403     input[27:0] man_input,
404     output[27:0] man_output
405 );
406     wire[49:0] temp;
407     wire sticky;
408     assign temp = {man_input, 22'b0} >> signed'(exp_difference);
409     assign sticky = (temp[22:0] != 0 || exp_difference >= 26) ? 1 : 0;
410     assign man_output = {temp[49:23], sticky};
411 endmodule
412
413 module significands_adder(
414     input [1:0] add_op,
415     input [27:0] man_in_1, man_in_2,
416     output reg sig_r,
417     output [27:0] man_output
418 );
419     reg[27:0] temp;
420     wire sticky;
421     assign sticky = (man_in_1[0] || man_in_2[0] || temp[0]) ? 1 : 0; // Manteniendo el
422     ↪ sticky bit
423     assign man_output = {temp[27:1], sticky}; // Mantisa de salida
424     always_comb
425         case(add_op) // add_op = {signo mantisa desplazada (man_in_1), signo mantisa no
426         ↪ desplazada (man_in_2)}
427             2'b00: begin
428                 temp = unsigned'(man_in_2) + unsigned'(man_in_1);
429                 sig_r = 0;
430             end
431             2'b01:
432                 if(man_in_1 >= man_in_2) begin
433                     temp = unsigned'(man_in_1) - unsigned'(man_in_2);
434                     sig_r = 0;
435                 end
436             end
437         endcase
438 endmodule

```

```

434     else begin
435         temp = unsigned'(man_in_2) - unsigned'(man_in_1);
436         sig_r = 1;
437     end
438 2'b10:
439     if(man_in_2 >= man_in_1) begin
440         temp = unsigned'(man_in_2) - unsigned'(man_in_1);
441         sig_r = 0;
442     end
443     else begin
444         temp = unsigned'(man_in_1) - unsigned'(man_in_2);
445         sig_r = 1;
446     end
447 2'b11: begin
448     temp = unsigned'(man_in_2) + unsigned'(man_in_1);
449     sig_r = 1;
450 end
451 endcase
452 endmodule
453 /*----- ADDER END -----*/
454
455 /*----- MULTIPLIER BEGIN -----*/
456 module MULTIPLIER(
457     input start, clk,
458     input [7:0] exp1, exp2,
459     input [27:0] man1, man2,
460     output ready,
461     output [7:0] exp_out,
462     output [27:0] man_out
463 );
464 wire ready1, ready2;
465 wire [9:0] temp_exp;
466 wire [26:0] man1_normalized, man2_normalized;
467 wire [55:0] temp_man1, temp2_man;
468 wire [4:0] desp, offset1, offset2;
469 assign ready = (ready1 & ready2) ? 1'b1 : 1'b0;
470 assign temp_exp ==signed'({2'b0, exp1})+signed'({2'b0, exp2})-signed'({5'b0, offset1})-
    ↪ signed'({5'b0, offset2})-signed'(10'b0001111111)+signed'({5'b0, desp});
471 assign exp_out = (signed'(temp_exp)<signed'(0)) ? 8'b0 : // Desnorm -
    ↪ UF
472     ((signed'(temp_exp)==signed'(0) & temp_man1[55]) ? 8'b1 : // NX
473     ((signed'(temp_exp)>=signed'(10'b0011111111)) ? 8'b11111111 : // OF
474     temp_exp[7:0])); // normal case
475 assign temp2_man = (unsigned'(temp_man1) >> (~unsigned'(temp_exp)+unsigned'(10'
    ↪ b10)));
476 assign man_out = (signed'(temp_exp)<signed'(0)) ? {temp2_man[55:29], ((temp2_man
    ↪ [28:0] != 29'b0) ? 1'b1 : 1'b0)} : // Desnorm - UF
477     ((signed'(temp_exp)==signed'(0) & temp_man1[55]) ? {1'b0, temp_man1
    ↪ [55:30], ((temp_man1[29:0] != 29'b0) ? 1'b1 : 1'b0)} : // NX
478     ((signed'(temp_exp)>=signed'(10'b0011111111)) ? 28'b0 :
    ↪ // OF
479     {temp_man1[55:29], ((temp_man1[28:0] != 29'b0) ? 1'b1 : 1'b0)});

```

```

480 significands_multiplier man_multiplier(
481     .man_in_1({1'b0, man1_normalized}), .man_in_2({1'b0, man2_normalized}), .
    ↪ previous_desp2righth(5'b0),
482     .man_output(temp_man1[55:28]), .actual_desp2righth(desp)
483 );
484 normalizer4significands_inputs normalizer4significands_inputs_multiplier1(
485     .start(start), .clk(clk), .man_in(man1[26:0]),
486     .ready(ready1), .man_out(man1_normalized), .offset(offset1)
487 );
488 normalizer4significands_inputs normalizer4significands_inputs_multiplier2(
489     .start(start), .clk(clk), .man_in(man2[26:0]),
490     .ready(ready2), .man_out(man2_normalized), .offset(offset2)
491 );
492 endmodule
493
494 module significands_multiplier( // man_in_1 * man_in_2
495     input [4:0] previous_desp2righth,
496     input [27:0] man_in_1, man_in_2,
497     output [4:0] actual_desp2righth,
498     output [27:0] man_output
499 );
500     wire[55:0] temp;
501     wire sticky;
502     assign temp = 56'(unsigned'(man_in_1)*unsigned'(man_in_2));
503     assign sticky = (temp[55] | temp[54]) ? ((temp[28:0] != 29'b0) ? 1 : 0) : ((temp[26:0] !=
    ↪ 27'b0) ? 1 : 0) ;
504     assign actual_desp2righth = (temp[55] | temp[54]) ? (unsigned'(previous_desp2righth)+
    ↪ unsigned'(5'b10)) : previous_desp2righth;
505     assign man_output = (temp[55] | temp[54]) ? ({temp[55:29], sticky}) : ({temp[53:27],
    ↪ sticky});
506 endmodule
507 /*----- MULTIPLIER END -----*/
508
509 /*----- DIVISOR BEGIN -----*/
510 module DIVISOR(
511     input start, rst, clk,
512     input [7:0] exp1, exp2,
513     input [27:0] man1, man2,
514     output ready,
515     output [7:0] exp_out,
516     output [27:0] man_out
517 );
518     wire sticky;
519     wire [9:0] temp_exp;
520     wire [55:0] temp_man1, temp2_man;
521     wire [4:0] dividend_offset, divisor_offset;
522     assign temp_exp = signed'({2'b0, exp1})-signed'({5'b0, dividend_offset})-signed'({2'b0,
    ↪ exp2})+signed'({5'b0, divisor_offset})+signed'(10'b0001111111);
523     assign exp_out = (signed'(temp_exp)<=signed'(0)) ? 8'b0 : //
    ↪ Desnorm - UF
524         ((signed'(temp_exp)>=signed'(10'b0011111111) & temp_man1[54]) ? 8'
    ↪ b11111111 : // OF

```

```

525         temp_exp[7:0]); // normal
    ↪ case
526 assign temp2_man = (unsigned'(temp_man1) >> (~unsigned'(temp_exp)+unsigned'(10'
    ↪ b10)));
527 assign sticky = (temp2_man[28:0] != 29'b0) ? 1'b1 : 1'b0;
528 assign man_out = (signed'(temp_exp)<=signed'(0)) ? {temp2_man[55:29], sticky} :
    ↪ // Desnorm - UF
529     ( (signed'(temp_exp)>=signed'(10'b001111111) & temp_man1[54] ) ? 28'
    ↪ b0 : // OF
530     temp_man1[55:28]); // normal
    ↪ case
531 assign {temp_man1[55], temp_man1[27:0]} = 29'b0;
532 significands_divisor significands_divisor_unit(
533     .start(start), .rst(rst), .clk(clk),
534     .dividend(man1[26:0]), .divisor(man2[26:0]),
535     .ready(ready), .man_output(temp_man1[54:28]),
536     .dividend_offset(dividend_offset), .divisor_offset(divisor_offset)
537 );
538 endmodule
539
540 module significands_divisor (
541     input start, rst, clk,
542     input [26:0] dividend, divisor, // dividend/divisor=quotient or quotient*divisor+
    ↪ remainder=dividend
543     output reg ready,
544     output [26:0] man_output, // bit_0, bit_-1, ..., bit_-23, bit_g, bit_r, bit_s
545     output [4:0] dividend_offset, divisor_offset
546 );
547 typedef enum {
548     waiting_state,
549     initial_state,
550     middle_state,
551     iteration_state_1,
552     iteration_state_2a1,
553     iteration_state_2a2,
554     iteration_state_2b1,
555     iteration_state_2b2,
556     iteration_state_3,
557     final_state
558 } state_type;
559 state_type actual_state, next_state;
560 reg sel_divisor, set_divisor, sel_remainder, set_remainder, sel_op,
561     sel_quotient, set_quotient, rst_quotient, set_counter, rst_counter;
562 wire dividend_ready, divisor_ready, count_flag, remainder_is_ltzero, sticky;
563 wire [4:0] to_counter_reg, from_counter_reg;
564 wire [26:0] dividend_normalized, divisor_normalized,
565     to_quotient_reg, from_quotient_reg,
566     to_divisor_reg, from_divisor_reg;
567 wire [28:0] to_remainder_reg, from_remainder_reg;
568 assign to_divisor_reg = sel_divisor ? (from_divisor_reg>>1) : divisor_normalized;
569 assign to_remainder_reg = sel_remainder ? (
570     (sel_op) ? (signed'(from_remainder_reg)+signed'({2'b0, from_divisor_reg})) : //

```

```

571     ↪ sel_op=1 -> rr=rr+dd
           (signed'(from_remainder_reg)-signed'({2'b0, from_divisor_reg})) // sel_op
572     ↪ =0 -> rr=rr-dd
           ): {2'b0, dividend_normalized};
573 assign to_quotient_reg = sel_quotient ? (unsigned'((from_quotient_reg<<1))+unsigned
574     ↪ '(27'b1)) : (from_quotient_reg<<1);
575 assign to_counter_reg = (unsigned'(from_counter_reg)+unsigned'(5'b1));
576 assign count_flag = (unsigned'(from_counter_reg)==unsigned'(27)) ? 1'b1 : 1'b0;
577 assign remainder_is_ltzero = (signed'(from_remainder_reg)<signed'(29'b0)) ? 1'b1 : 1'b0;
578 assign sticky = ((from_quotient_reg[0]==1'b1 | to_remainder_reg!=29'b0) ? 1'b1 : 1'b0)
579     ↪ ;
580 assign man_output = {from_quotient_reg[26:1], sticky}; // se mantiene sticky
581 // FSM
582 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
583 always_comb
584     case(actual_state)
585     default: begin
586         {sel_divisor, set_divisor}           = 2'b00; // divisor controls
587         {sel_remainder, set_remainder}       = 2'b00; // remainder controls
588         {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
589         {set_counter, rst_counter}           = 2'b00; // counter controls
590         {sel_op, ready}                       = 2'b00; // alu_op and ready signals
591         next_state = actual_state;
592     end
593     waiting_state: begin
594         {sel_divisor, set_divisor}           = 2'b00; // divisor controls
595         {sel_remainder, set_remainder}       = 2'b00; // remainder controls
596         {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
597         {set_counter, rst_counter}           = 2'b00; // counter controls
598         {sel_op, ready}                       = 2'b00; // alu_op and ready signals
599         next_state = (start & dividend_ready & divisor_ready) ? initial_state :
600     ↪ waiting_state;
601     end
602     initial_state: begin
603         {sel_divisor, set_divisor}           = 2'b01; // divisor controls
604         {sel_remainder, set_remainder}       = 2'b01; // remainder controls
605         {sel_quotient, set_quotient, rst_quotient} = 3'b001; // quotient controls
606         {set_counter, rst_counter}           = 2'b01; // counter controls
607         {sel_op, ready}                       = 2'b00; // alu_op and ready signals
608         next_state = middle_state;
609     end
610     middle_state: begin
611         {sel_divisor, set_divisor}           = 2'b10; // divisor controls
612         {sel_remainder, set_remainder}       = 2'b10; // remainder controls
613         {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
614         {set_counter, rst_counter}           = 2'b00; // counter controls
615         {sel_op, ready}                       = 2'b00; // alu_op and ready signals
616         next_state = iteration_state_1;
617     end
618     iteration_state_1: begin
619         {sel_divisor, set_divisor}           = 2'b10; // divisor controls
620         {sel_remainder, set_remainder}       = 2'b11; // remainder controls

```

```

618     {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
619     {set_counter, rst_counter}                = 2'b10; // counter controls
620     {sel_op, ready}                           = 2'b00; // alu_op and ready signals
621     next_state = remainder_is_ltzero ? iteration_state_2b1 : iteration_state_2a1;
622 end
623 iteration_state_2a1: begin
624     {sel_divisor, set_divisor}                 = 2'b10; // divisor controls
625     {sel_remainder, set_remainder}            = 2'b10; // remainder controls
626     {sel_quotient, set_quotient, rst_quotient} = 3'b100; // quotient controls
627     {set_counter, rst_counter}                = 2'b00; // counter controls
628     {sel_op, ready}                           = 2'b00; // alu_op and ready signals
629     next_state = iteration_state_2a2;
630 end
631 iteration_state_2a2: begin
632     {sel_divisor, set_divisor}                 = 2'b10; // divisor controls
633     {sel_remainder, set_remainder}            = 2'b10; // remainder controls
634     {sel_quotient, set_quotient, rst_quotient} = 3'b110; // quotient controls
635     {set_counter, rst_counter}                = 2'b00; // counter controls
636     {sel_op, ready}                           = 2'b00; // alu_op and ready signals
637     next_state = iteration_state_3;
638 end
639 iteration_state_2b1: begin
640     {sel_divisor, set_divisor}                 = 2'b10; // divisor controls
641     {sel_remainder, set_remainder}            = 2'b10; // remainder controls
642     {sel_quotient, set_quotient, rst_quotient} = 3'b010; // quotient controls
643     {set_counter, rst_counter}                = 2'b00; // counter controls
644     {sel_op, ready}                           = 2'b10; // alu_op and ready signals
645     next_state = iteration_state_2b2;
646 end
647 iteration_state_2b2: begin
648     {sel_divisor, set_divisor}                 = 2'b10; // divisor controls
649     {sel_remainder, set_remainder}            = 2'b11; // remainder controls
650     {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
651     {set_counter, rst_counter}                = 2'b00; // counter controls
652     {sel_op, ready}                           = 2'b10; // alu_op and ready signals
653     next_state = iteration_state_3;
654 end
655 iteration_state_3: begin
656     {sel_divisor, set_divisor}                 = 2'b11; // divisor controls
657     {sel_remainder, set_remainder}            = 2'b10; // remainder controls
658     {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
659     {set_counter, rst_counter}                = 2'b00; // counter controls
660     {sel_op, ready}                           = 2'b00; // alu_op and ready signals
661     next_state = count_flag ? final_state : iteration_state_1;
662 end
663 final_state: begin
664     {sel_divisor, set_divisor}                 = 2'b00; // divisor controls
665     {sel_remainder, set_remainder}            = 2'b00; // remainder controls
666     {sel_quotient, set_quotient, rst_quotient} = 3'b000; // quotient controls
667     {set_counter, rst_counter}                = 2'b00; // counter controls
668     {sel_op, ready}                           = 2'b01; // alu_op and ready signals
669     next_state = start ? final_state : waiting_state;

```

```

670     end
671   endcase
672   generic_register #(.width(29)) remainder_register(
673     .clk(clk), .reset(1'b0), .load(set_remainder), .data_in(to_remainder_reg),
674     .data_out(from_remainder_reg)
675   );
676   generic_register #(.width(27)) divisor_register(
677     .clk(clk), .reset(1'b0), .load(set_divisor), .data_in(to_divisor_reg),
678     .data_out(from_divisor_reg)
679   );
680   generic_register #(.width(27)) quotient_register(
681     .clk(clk), .reset(rst_quotient), .load(set_quotient), .data_in(to_quotient_reg),
682     .data_out(from_quotient_reg)
683   );
684   generic_register #(.width(5)) counter_register(
685     .clk(clk), .reset(rst_counter), .load(set_counter), .data_in(to_counter_reg),
686     .data_out(from_counter_reg)
687   );
688   normalizer4significands_inputs normalizer4significands_inputs_divisor_4dividend(
689     .start(start), .clk(clk), .man_in(dividend),
690     .ready(dividend_ready), .man_out(dividend_normalized), .offset(dividend_offset)
691   );
692   normalizer4significands_inputs normalizer4significands_inputs_divisor_4divisor(
693     .start(start), .clk(clk), .man_in(divisor),
694     .ready(divisor_ready), .man_out(divisor_normalized), .offset(divisor_offset)
695   );
696   endmodule
697
698   module normalizer4significands_inputs(
699     input start, clk,
700     input [26:0] man_in,
701     output reg ready,
702     output reg [26:0] man_out,
703     reg [4:0] offset
704   );
705   reg iteration_flag;
706   always_ff @(posedge(clk))
707     if(start)
708       if(~ready)
709         if(iteration_flag)
710           if(man_out[26]) {offset, man_out, iteration_flag, ready} = {offset, man_out,
711             ↪ 2'b11};
712           else {offset, man_out, iteration_flag, ready} = {offset+5'b1,
713             ↪ man_out<<1, 2'b10};
714           else if(man_in==0) {offset, man_out, ready} = 33'b1;
715           else if(man_in[26]) {offset, man_out, iteration_flag, ready} = {5'b0, man_in, 2'
716             ↪ b01};
717           else {offset, man_out, iteration_flag, ready} = {5'b0, man_in<<1, 2'
718             ↪ b10};
719           else {offset, man_out, iteration_flag, ready} = {offset, man_out, 2'b01};
720           else {man_out, iteration_flag, ready} = 29'b0;
721   endmodule

```

```

718 /*----- DIVISOR END -----*/
719
720 /*----- SQRT BEGIN -----*/
721 module Sqrt(
722     input start, rst, clk,
723     input [7:0] exp_in,
724     input [27:0] man_in,
725     output ready,
726     output [7:0] exp_out,
727     output [27:0] man_out
728 );
729 wire man_ready;
730 wire [4:0] desp2righth_from_sqrt, desp2righth_from_mul, man_offset;
731 wire [26:0] man_normalized;
732 wire [27:0] man_from_sqrt, man_from_mul;
733 // exp_in[0] = 1 -> (exp_in-127) is even
734 assign exp_out = (exp_in[0] | (man_offset[0] & exp_in==8'b0)) ? (( ((unsigned'(exp_in
    ↪ )-unsigned'({3'b0, man_offset})-unsigned'(127)) >> 1)+unsigned'(127) )+unsigned
    ↪ '({3'b0, desp2righth_from_sqrt})) :
735     (( ((unsigned'(exp_in)-unsigned'({3'b0, man_offset})-unsigned
    ↪ '(128)) >> 1)+unsigned'(127) )+unsigned'({3'b0, desp2righth_from_mul }));
736 assign man_out = (exp_in[0] | (man_offset[0] & exp_in==8'b0)) ? man_from_sqrt :
    ↪ man_from_mul;
737 significand_sqrt significand_sqrt_unit(
738     .start(man_ready), .rst(rst), .clk(clk), .frac_in(man_normalized[25:3]),
739     .ready(ready), .desp2righth_out(desp2righth_from_sqrt), .man_output(man_from_sqrt)
740 );
741 significands_multiplier significands_multiplier_unit(
742     .man_in_1(man_from_sqrt), .man_in_2(28'b01_01101010000010011110011_000), .
    ↪ previus_desp2righth(desp2righth_from_sqrt),
743     .man_output(man_from_mul), .actual_desp2righth(desp2righth_from_mul)
744 );
745 normalizer4significands_inputs normalizer4significands_inputs_sqrt_unit(
746     .start(start), .clk(clk), .man_in(man_in[26:0]),
747     .ready(man_ready), .man_out(man_normalized), .offset(man_offset)
748 );
749 endmodule
750
751 module significand_sqrt( // = sqrt({1,frac_in})
752     input start, rst, clk,
753     input [22:0] frac_in,
754     output reg ready,
755     output [4:0] desp2righth_out,
756     output [27:0] man_output // bit_1, bit_0, bitexp_out_-1, ..., bit_-23, bit_g, bit_r,
    ↪ bit_s
757 );
758 typedef enum {
759     waiting_state, iteration_1, iteration_2, iteration_3, iteration_4, final_state
760 } state_type;
761 state_type actual_state, next_state;
762 reg set, sel;
763 wire [4:0] desp2righth_to_reg,

```

```

764     desp2righth_from_reg,
765     desp2righth_to_multiplier_1,
766     desp2righth_from_multiplier_1_to_multiplier_2,
767     desp2righth_from_multiplier_2_to_multiplier_3;
768 wire [22:0] magic_number_to_sub_1;
769 wire [27:0] temp_to_reg,
770     temp_from_reg,
771     to_multiplier_1_and_3,
772     from_mul_2_to_sub_2,
773     from_sub_2_to_mul_3,
774     from_mul_1_to_mul_2,
775     res_from_sub_1;
776 // MUX
777 assign {to_multiplier_1_and_3, desp2righth_to_multiplier_1} = sel ? {temp_from_reg,
    ↪ desp2righth_from_reg} : {res_from_sub_1, 5'b0} ;
778 // lookup table
779 assign magic_number_to_sub_1 = ( (frac_in >= 23'b00000000000000000000) & (
    ↪ frac_in <= 23'b00111111111111111111) ) ? 23'b100000011000010011111010 :
780     ( (frac_in >= 23'b01000000000000000000) & (frac_in <= 23'
    ↪ b01011111111111111111) ) ? 23'b100001001101110100001111 :
781     ( (frac_in >= 23'b01100000000000000000) & (frac_in <= 23'
    ↪ b01111111111111111111) ) ? 23'b100010101111110011001111 :
782     ( (frac_in >= 23'b10000000000000000000) & (frac_in <= 23'
    ↪ b10011111111111111111) ) ? 23'b10010001011001000011000 :
783     ( (frac_in >= 23'b10100000000000000000) & (frac_in <= 23'
    ↪ b10111111111111111111) ) ? 23'b10011001000100100100010 :
784     ( (frac_in >= 23'b11000000000000000000) & (frac_in <= 23'
    ↪ b11011111111111111111) ) ? 23'b101000101000101000011111 :
785     ( (frac_in >= 23'b11100000000000000000) & (frac_in <= 23'
    ↪ b11111111111111111111) ) ? 23'b10110101011111011100101 : 23'bX;
786 // FSM
787 always_ff @(posedge(clk)) actual_state <= rst ? waiting_state : next_state;
788 always_comb begin
789     case(actual_state)
790     default: begin
791         {sel, set, ready} = 3'b000;
792         next_state = actual_state;
793     end
794     waiting_state: begin
795         {sel, set, ready} = 3'b000;
796         next_state = start ? iteration_1 : waiting_state;
797     end
798     iteration_1: begin
799         {sel, set, ready} = 3'b010;
800         next_state = iteration_2;
801     end
802     iteration_2: begin
803         {sel, set, ready} = 3'b110;
804         next_state = iteration_3;
805     end
806     iteration_3: begin
807         {sel, set, ready} = 3'b110;

```

```

808         next_state = iteration_4;
809     end
810     iteration_4: begin
811         {sel, set, ready} = 3'b110;
812         next_state = final_state;
813     end
814     final_state: begin
815         {sel, set, ready} = 3'b001;
816         next_state = start ? final_state : waiting_state;
817     end
818 endcase
819 end
820 // sub: |man_in_1 - man_in_2|
821 significands_subtractor sub_1(
822     .man_in_1({3'b1, frac_in, 2'b00}), .man_in_2({2'b01, magic_number_to_sub_1, 3'
823     ↪ b0}),
824     .man_output(res_from_sub_1)
825 );
826 significands_subtractor sub_2(
827     .man_in_1({3'b011, 25'b0}), .man_in_2(from_mul_2_to_sub_2),
828     .man_output(from_sub_2_to_mul_3)
829 );
830 // mul: man_in_1 * man_in_2
831 significands_multiplier mul_1(
832     .man_in_1(to_multiplier_1_and_3), .man_in_2(to_multiplier_1_and_3), .
833     ↪ previus_desp2righth(desp2righth_to_multiplier_1),
834     .man_output(from_mul_1_to_mul_2), .actual_desp2righth(
835     ↪ desp2righth_from_multiplier_1_to_multiplier_2)
836 );
837 significands_multiplier mul_2(
838     .man_in_1(from_mul_1_to_mul_2), .man_in_2({3'b1, frac_in, 2'b00}), .
839     ↪ previus_desp2righth(desp2righth_from_multiplier_1_to_multiplier_2),
840     .man_output(from_mul_2_to_sub_2), .actual_desp2righth(
841     ↪ desp2righth_from_multiplier_2_to_multiplier_3)
842 );
843 significands_multiplier mul_3(
844     .man_in_1(to_multiplier_1_and_3), .man_in_2(from_sub_2_to_mul_3), .
845     ↪ previus_desp2righth(desp2righth_from_multiplier_2_to_multiplier_3),
846     .man_output(temp_to_reg), .actual_desp2righth(desp2righth_to_reg)
847 );
848 significands_multiplier mul_4(
849     .man_in_1(temp_from_reg), .man_in_2({2'b1, frac_in, 3'b00}), .previus_desp2righth(
850     ↪ desp2righth_from_reg),
851     .man_output(man_output), .actual_desp2righth(desp2righth_out)
852 );
853 // register
854 generic_register #(.width(33)) temp_register(
855     .clk(clk), .reset(1'b0), .load(set), .data_in({temp_to_reg, desp2righth_to_reg}),
856     .data_out({temp_from_reg, desp2righth_from_reg})
857 );
858 endmodule
859

```

```

853 module significands_subtractor( // |man_in_1 - man_in_2|
854     input [27:0] man_in_1, man_in_2,
855     output [27:0] man_output
856 );
857 wire sticky;
858 wire [27:0] temp;
859 assign temp = ( unsigned'(man_in_1) >= unsigned'(man_in_2) ) ?
860     ( unsigned'(man_in_1) - unsigned'(man_in_2) ) :
861     ( unsigned'(man_in_2) - unsigned'(man_in_1) );
862 assign sticky = (man_in_1[0] | man_in_2[0] | temp[0]) ? 1 : 0;
863 assign man_output = {temp[27:1], sticky};
864 endmodule
865 /*----- SQRT END -----*/
866 /****** Arithmetic Units: end *****/
867
868 /** Exceptions and trivial cases checker: begin **/
869 module excep_triv_checker(
870     input start, clk,
871     input [2:0] rm,
872     input [1:0] op,
873     input sig1, sig2,
874     input [7:0] exp1, exp2,
875     input [23:0] frac1, frac2,
876     output ready,
877     output reg [2:0] status_out, sel_out
878 /*
879     rm =
880         000 -> RNE
881         001 -> RTZ
882         010 -> RDN
883         011 -> RUP
884         100 -> RMM
885     op =
886         00 -> ADD
887         01 -> MUL
888         10 -> DIV
889         11 -> SQRT
890
891     status_out =
892         000 -> no flag
893         101 -> no flag (not used)
894         110 -> no flag (not used)
895
896         001 -> UF flag
897         010 -> OF flag
898         011 -> NV flag
899         100 -> DZ flag
900         111 -> NX flag (not used here)
901     sel_out =
902         000 -> no trivial
903         001 -> in1
904         010 -> in2

```

```

905         011 -> almost inf
906         100 -> zero
907         101 -> almost zero
908         110 -> inf
909         111 -> nan
910     */
911 );
912 wire in1_is_nan, in2_is_nan, in1_is_zero, in2_is_zero, in1_is_inf, in2_is_inf,
    ↪ sig1_equal_sig2, xor_sigs, ready1, ready2;
913 wire[26:0] man1_normalized, man2_normalized;
914 wire [9:0] mul_exp_initial_res, div_exp_initial_res;
915 wire [4:0] offset1, offset2;
916 assign ready = ((ready1 | in1_is_zero) & (ready2 | in2_is_zero)) ? 1'b1 : 1'b0;
917 assign in1_is_nan = (exp1 == 8'b11111111 & frac1[22:0] != 23'b0) ? 1 : 0;
918 assign in2_is_nan = (exp2 == 8'b11111111 & frac2[22:0] != 23'b0) ? 1 : 0;
919 assign in1_is_zero = (exp1 == 8'b0 & frac1[22:0] == 23'b0) ? 1 : 0;
920 assign in2_is_zero = (exp2 == 8'b0 & frac2[22:0] == 23'b0) ? 1 : 0;
921 assign in1_is_inf = (exp1 == 8'b11111111 & frac1[22:0] == 23'b0) ? 1 : 0;
922 assign in2_is_inf = (exp2 == 8'b11111111 & frac2[22:0] == 23'b0) ? 1 : 0;
923 assign sig1_equal_sig2 = (sig1 == sig2) ? 1 : 0;
924 assign xor_sigs = (sig1 ^ sig2) ? 1 : 0;
925 assign mul_exp_initial_res = signed'({2'b0, exp1}) - signed'({5'b0, offset1}) + signed
    ↪ '({2'b0, exp2}) - signed'({5'b0, offset2}) - signed'(10'b001111111);
926 assign div_exp_initial_res = signed'({2'b0, exp1}) - signed'({5'b0, offset1}) - signed'({2'
    ↪ b0, exp2}) + signed'({5'b0, offset2}) + signed'(10'b001111111);
927 always_comb
928     if(in1_is_nan | (in2_is_nan & (op != 2'b11))) {status_out, sel_out} = {3'b011, 3'
    ↪ b111}; // NV flag - nan
929     else case (op)
930         2'b00: // ADD
931             casex ({in1_is_zero, in2_is_zero, in1_is_inf, in2_is_inf})
932                 4'b1X_0X: {status_out, sel_out} = {3'b000, 3'b010}; // no flag - in2
933                 4'b01_X0: {status_out, sel_out} = {3'b000, 3'b001}; // no flag - in1
934                 4'b00_11: {status_out, sel_out} = xor_sigs ? {3'b011, 3'b111} : // NV flag -
    ↪ nan
935                                     {3'b000, 3'b001}; // no flag - in1
936                 4'b00_10: {status_out, sel_out} = {3'b000, 3'b001}; // no flag - in1
937                 4'b00_01: {status_out, sel_out} = {3'b000, 3'b010}; // no flag - in2
938                 default: {status_out, sel_out} = {3'b000, 3'b000}; // no flag - no trivial
939             endcase
940         2'b01: // MUL
941             casex ({in1_is_zero, in2_is_zero, in1_is_inf, in2_is_inf})
942                 4'b1X_01: {status_out, sel_out} = {3'b011, 3'b111}; // NV flag - nan
943                 4'b1X_00: {status_out, sel_out} = {3'b000, 3'b100}; // no flag - zero
944                 4'b01_10: {status_out, sel_out} = {3'b011, 3'b111}; // NV flag - nan
945                 4'b01_00: {status_out, sel_out} = {3'b000, 3'b100}; // no flag - zero
946                 4'b00_1X: {status_out, sel_out} = {3'b000, 3'b110}; // no flag - inf
947                 4'b00_X1: {status_out, sel_out} = {3'b000, 3'b110}; // no flag - inf
948                 default:
949                     if(signed'(mul_exp_initial_res) >= signed'(10'b001111111)) begin
950                         // OF flag - "inf" or "almost inf" (for DIV or MUL)
951                         status_out = 3'b010;

```

```

952         sel_out = (rm == 3'b001) ? 3'b011 : // RTZ
953             ((rm == 3'b010) ? ( xor_sigs ? 3'b110 : 3'b011) : // RDN
954             ((rm == 3'b011) ? (~xor_sigs ? 3'b110 : 3'b011) : // RUP
955             3'b110)); // RNE - RMM
956     end
957     else if(signed'(mul_exp_initial_res)<signed'(-24)) begin
958         // Uf flag - "zero" or "almost zero" (for DIV or MUL)
959         status_out = 3'b001;
960         sel_out = (rm == 3'b001) ? 3'b100 : // RTZ
961             ((rm == 3'b010) ? ( xor_sigs ? 3'b101 : 3'b100) : // RDN
962             ((rm == 3'b011) ? (~xor_sigs ? 3'b101 : 3'b100) : // RUP
963             3'b100)); // RNE - RMM
964     end
965     else {status_out, sel_out} = {3'b000, 3'b000}; // no flag - no trivial
966 endcase
967 2'b10: // DIV
968     casex ({in1_is_zero, in2_is_zero, in1_is_inf, in2_is_inf})
969         4'b11_00: {status_out, sel_out} = {3'b011, 3'b111}; // NV flag - nan
970         4'b10_0X: {status_out, sel_out} = {3'b000, 3'b100}; // no flag - zero
971         4'b01_X0: {status_out, sel_out} = {3'b100, 3'b110}; // DZ flag - inf
972         4'b00_11: {status_out, sel_out} = {3'b011, 3'b111}; // NV flag - nan
973         4'b00_10: {status_out, sel_out} = {3'b000, 3'b110}; // no flag - inf
974         4'b00_01: {status_out, sel_out} = {3'b000, 3'b100}; // no flag - zero
975     default:
976         if(signed'(div_exp_initial_res)>=signed'(10'b001111111)) begin
977             if((unsigned'(man1_normalized)<unsigned'(man2_normalized))&(signed
↪ '(div_exp_initial_res)==signed'(10'b001111111)))
978                 {status_out, sel_out} = {3'b000, 3'b000}; // no flag -
979             else begin
980                 // OF flag - "inf" or "almost inf" (for DIV or MUL)
981                 status_out = 3'b010;
982                 sel_out = (rm == 3'b001) ? 3'b011 : // RTZ
983                     ((rm == 3'b010) ? ( xor_sigs ? 3'b110 : 3'b011) : // RDN
984                     ((rm == 3'b011) ? (~xor_sigs ? 3'b110 : 3'b011) : // RUP
985                     3'b110));
986             end
987                 // RNE - RMM
988         end
989         else if(signed'(div_exp_initial_res)<signed'(-24)) begin
990             // Uf flag - "zero" or "almost zero" (for DIV or MUL)
991             status_out = 3'b001;
992             sel_out = (rm == 3'b001) ? 3'b100 : // RTZ
993                 ((rm == 3'b010) ? ( xor_sigs ? 3'b101 : 3'b100) : // RDN
994                 ((rm == 3'b011) ? (~xor_sigs ? 3'b101 : 3'b100) : // RUP
995                 3'b100)); // RNE - RMM
996         end
997         else {status_out, sel_out} = {3'b000, 3'b000}; // no flag - no trivial
998     endcase
999     default: // Sqrt
1000         if(sig1) {status_out, sel_out} = {3'b011, 3'b111}; // NV flag - nan
1001         else if(in1_is_zero | in1_is_inf) {status_out, sel_out} = {3'b000, 3'b001}; //
↪ no flag - in1

```

```

1002         else {status_out, sel_out} = {3'b000, 3'b000}; // no flag - no trivial
1003     endcase
1004     normalizer4significands_inputs normalizer4significands_inputs_checker1(
1005         .start(start), .clk(clk), .man_in({frac1[23:0], 3'b0}),
1006         .ready(ready1), .man_out(man1_normalized), .offset(offset1)
1007     );
1008     normalizer4significands_inputs normalizer4significands_inputs_checker2(
1009         .start(start), .clk(clk), .man_in({frac2[23:0], 3'b0}),
1010         .ready(ready2), .man_out(man2_normalized), .offset(offset2)
1011     );
1012 endmodule
1013 /*** Exceptions and trivial cases checker: end *****/

```

top_constraints.xdc: archivo que define las *constraints* del proyecto en *Vivado*.

```

1 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[5]}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[4]}]
3 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[3]}]
4 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[2]}]
5 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[1]}]
6 set_property IOSTANDARD LVCMOS33 [get_ports {color_leds[0]}]
7 set_property PACKAGE_PIN F6 [get_ports {color_leds[0]}]
8 set_property PACKAGE_PIN F13 [get_ports {color_leds[1]}]
9 set_property PACKAGE_PIN K5 [get_ports {color_leds[2]}]
10 set_property PACKAGE_PIN L16 [get_ports {color_leds[3]}]
11 set_property PACKAGE_PIN H6 [get_ports {color_leds[4]}]
12 set_property PACKAGE_PIN K6 [get_ports {color_leds[5]}]
13 set_property IOSTANDARD LVCMOS33 [get_ports {display[15]}]
14 set_property IOSTANDARD LVCMOS33 [get_ports {display[14]}]
15 set_property IOSTANDARD LVCMOS33 [get_ports {display[13]}]
16 set_property IOSTANDARD LVCMOS33 [get_ports {display[12]}]
17 set_property IOSTANDARD LVCMOS33 [get_ports {display[11]}]
18 set_property IOSTANDARD LVCMOS33 [get_ports {display[10]}]
19 set_property IOSTANDARD LVCMOS33 [get_ports {display[9]}]
20 set_property IOSTANDARD LVCMOS33 [get_ports {display[8]}]
21 set_property IOSTANDARD LVCMOS33 [get_ports {display[7]}]
22 set_property IOSTANDARD LVCMOS33 [get_ports {display[6]}]
23 set_property IOSTANDARD LVCMOS33 [get_ports {display[5]}]
24 set_property IOSTANDARD LVCMOS33 [get_ports {display[4]}]
25 set_property IOSTANDARD LVCMOS33 [get_ports {display[3]}]
26 set_property IOSTANDARD LVCMOS33 [get_ports {display[2]}]
27 set_property IOSTANDARD LVCMOS33 [get_ports {display[1]}]
28 set_property IOSTANDARD LVCMOS33 [get_ports {display[0]}]
29 set_property PACKAGE_PIN M4 [get_ports {display[0]}]
30 set_property PACKAGE_PIN L6 [get_ports {display[1]}]
31 set_property PACKAGE_PIN M2 [get_ports {display[2]}]
32 set_property PACKAGE_PIN K3 [get_ports {display[3]}]
33 set_property PACKAGE_PIN L4 [get_ports {display[4]}]
34 set_property PACKAGE_PIN L5 [get_ports {display[5]}]
35 set_property PACKAGE_PIN L3 [get_ports {display[7]}]
36 set_property PACKAGE_PIN N1 [get_ports {display[6]}]
37 set_property PACKAGE_PIN N6 [get_ports {display[8]}]

```

```

38 set_property PACKAGE_PIN M6 [get_ports {display[9]}]
39 set_property PACKAGE_PIN M3 [get_ports {display[10]}]
40 set_property PACKAGE_PIN N5 [get_ports {display[11]}]
41 set_property PACKAGE_PIN N2 [get_ports {display[12]}]
42 set_property PACKAGE_PIN N4 [get_ports {display[13]}]
43 set_property PACKAGE_PIN L1 [get_ports {display[14]}]
44 set_property PACKAGE_PIN M1 [get_ports {display[15]}]
45 set_property IOSTANDARD LVCMOS33 [get_ports {in[15]}]
46 set_property IOSTANDARD LVCMOS33 [get_ports {in[14]}]
47 set_property IOSTANDARD LVCMOS33 [get_ports {in[13]}]
48 set_property IOSTANDARD LVCMOS33 [get_ports {in[12]}]
49 set_property IOSTANDARD LVCMOS33 [get_ports {in[11]}]
50 set_property IOSTANDARD LVCMOS33 [get_ports {in[10]}]
51 set_property IOSTANDARD LVCMOS33 [get_ports {in[9]}]
52 set_property IOSTANDARD LVCMOS33 [get_ports {in[8]}]
53 set_property IOSTANDARD LVCMOS33 [get_ports {in[7]}]
54 set_property IOSTANDARD LVCMOS33 [get_ports {in[6]}]
55 set_property IOSTANDARD LVCMOS33 [get_ports {in[5]}]
56 set_property IOSTANDARD LVCMOS33 [get_ports {in[4]}]
57 set_property IOSTANDARD LVCMOS33 [get_ports {in[3]}]
58 set_property IOSTANDARD LVCMOS33 [get_ports {in[2]}]
59 set_property IOSTANDARD LVCMOS33 [get_ports {in[1]}]
60 set_property IOSTANDARD LVCMOS33 [get_ports {in[0]}]
61 set_property PACKAGE_PIN U9 [get_ports {in[0]}]
62 set_property PACKAGE_PIN U8 [get_ports {in[1]}]
63 set_property PACKAGE_PIN R7 [get_ports {in[2]}]
64 set_property PACKAGE_PIN R6 [get_ports {in[3]}]
65 set_property PACKAGE_PIN R5 [get_ports {in[4]}]
66 set_property PACKAGE_PIN V7 [get_ports {in[5]}]
67 set_property PACKAGE_PIN V6 [get_ports {in[6]}]
68 set_property PACKAGE_PIN V5 [get_ports {in[7]}]
69 set_property PACKAGE_PIN U4 [get_ports {in[8]}]
70 set_property PACKAGE_PIN V2 [get_ports {in[9]}]
71 set_property PACKAGE_PIN U2 [get_ports {in[10]}]
72 set_property PACKAGE_PIN T3 [get_ports {in[11]}]
73 set_property PACKAGE_PIN T1 [get_ports {in[12]}]
74 set_property PACKAGE_PIN R3 [get_ports {in[13]}]
75 set_property PACKAGE_PIN P3 [get_ports {in[14]}]
76 set_property PACKAGE_PIN P4 [get_ports {in[15]}]
77 set_property IOSTANDARD LVCMOS33 [get_ports {leds[15]}]
78 set_property IOSTANDARD LVCMOS33 [get_ports {leds[14]}]
79 set_property IOSTANDARD LVCMOS33 [get_ports {leds[13]}]
80 set_property IOSTANDARD LVCMOS33 [get_ports {leds[12]}]
81 set_property IOSTANDARD LVCMOS33 [get_ports {leds[11]}]
82 set_property IOSTANDARD LVCMOS33 [get_ports {leds[10]}]
83 set_property IOSTANDARD LVCMOS33 [get_ports {leds[9]}]
84 set_property IOSTANDARD LVCMOS33 [get_ports {leds[8]}]
85 set_property IOSTANDARD LVCMOS33 [get_ports {leds[7]}]
86 set_property IOSTANDARD LVCMOS33 [get_ports {leds[6]}]
87 set_property IOSTANDARD LVCMOS33 [get_ports {leds[5]}]
88 set_property IOSTANDARD LVCMOS33 [get_ports {leds[4]}]
89 set_property IOSTANDARD LVCMOS33 [get_ports {leds[3]}]

```

```

90 set_property IOSTANDARD LVCMOS33 [get_ports {leds[2]}]
91 set_property IOSTANDARD LVCMOS33 [get_ports {leds[1]}]
92 set_property IOSTANDARD LVCMOS33 [get_ports {leds[0]}]
93 set_property PACKAGE_PIN T8 [get_ports {leds[0]}]
94 set_property PACKAGE_PIN V9 [get_ports {leds[1]}]
95 set_property PACKAGE_PIN R8 [get_ports {leds[2]}]
96 set_property PACKAGE_PIN T6 [get_ports {leds[3]}]
97 set_property PACKAGE_PIN T5 [get_ports {leds[4]}]
98 set_property PACKAGE_PIN T4 [get_ports {leds[5]}]
99 set_property PACKAGE_PIN U7 [get_ports {leds[6]}]
100 set_property PACKAGE_PIN U6 [get_ports {leds[7]}]
101 set_property PACKAGE_PIN V4 [get_ports {leds[8]}]
102 set_property PACKAGE_PIN U3 [get_ports {leds[9]}]
103 set_property PACKAGE_PIN V1 [get_ports {leds[10]}]
104 set_property PACKAGE_PIN R1 [get_ports {leds[11]}]
105 set_property PACKAGE_PIN P5 [get_ports {leds[12]}]
106 set_property PACKAGE_PIN U1 [get_ports {leds[13]}]
107 set_property PACKAGE_PIN R2 [get_ports {leds[14]}]
108 set_property PACKAGE_PIN P2 [get_ports {leds[15]}]
109 set_property IOSTANDARD LVCMOS33 [get_ports resume_asin]
110 set_property IOSTANDARD LVCMOS33 [get_ports rst_asin]
111 set_property IOSTANDARD LVCMOS33 [get_ports start_asin]
112 set_property PACKAGE_PIN C12 [get_ports rst_asin]
113 set_property PACKAGE_PIN F15 [get_ports start_asin]
114 set_property PACKAGE_PIN V10 [get_ports resume_asin]
115
116 set_property IOSTANDARD LVCMOS33 [get_ports clk]
117 set_property PACKAGE_PIN E3 [get_ports clk]
118
119 set_property CFGBVS VCCO [current_design]
120 set_property CONFIG_VOLTAGE 3.3 [current_design]
121
122 create_clock -period 270.000 -name CLK -waveform {0.000 135.000} -add [get_ports clk]

```

2 Test Benches

En esta sección se presentan *test benches* creados para probar distintos módulos del *SoC* diseñado:

tb_memory.sv: *test bench* encargado de comprobar el correcto funcionamiento del módulo *memory.sv*.

```
1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4 */
5 `timescale 1ns / 1ps
6
7 module tb_memory;
8
9     // Inputs
10    reg clk          = 0;
11    reg rst          = 0;
12    reg rw           = 0;
13    reg valid        = 0;
14    reg [31:0] addr  = 0;
15    reg [31:0] data_in    = 0;
16    reg [1:0] byte_half_word = 0;
17    reg is_load_unsigned = 0;
18    // Outputs
19    wire ready;
20    wire out_of_range;
21    wire [31:0] data_out;
22
23    // Unit under test (UUT)
24    memory uut(
25        // Inputs
26        .clk(clk),
27        .rst(rst),
28        .rw(rw),
29        .valid(valid),
30        .addr(addr),
31        .data_in(data_in),
32        .byte_half_word(byte_half_word),
33        .is_load_unsigned(is_load_unsigned),
34        // Outputs
35        .ready(ready),
36        .out_of_range(out_of_range),
37        .data_out(data_out)
38    );
39
40    always #5 clk = ~clk;
41
42    // Testing
43    initial begin
44        // máxima dirección posible: 32'b0000000000000000_1111_0100_0001_1111
45        int initial_time, elapsed_time;
```

```

46     $display("\nÚltimo byte direccionable: %h\n", 32'
↪ b0000000000000000_1111_0100_0001_1111);
47
48     // test 1: verificar señal "out_of_range"
49     $display("test 1: verificar señal \"out_of_range\");
50     addr    = 32'b0000000000000000_1111_0100_0101_1111;
51     data_in = 32'b01101010011100001010001100001100;
52     // options
53     byte_half_word    = 2'b00;
54     is_load_unsigned  = 0;
55     rw                = 0;
56     $display(">> addr: %h - data_in: %h", addr, data_in);
57     $display("options: load word,");
58     // execution
59     #10 ; initial_time = $time; valid = 1;
60     while(~ready & ~out_of_range) #10;
61     elapsed_time = $time-initial_time;
62     $display("Initial time: %t", initial_time);
63     $display("Elapsed time: %t", elapsed_time);
64     // test
65     if(~out_of_range)
66         $display("test 1: failed\n");
67     else $display("test 1: passed\n");
68     #10 ; valid = 0;
69
70     // test 2: guardar data (word) y leerla
71     $display("test 2: guardar data (word) y leerla");
72     addr    = 32'b0000000000000000_1001_0100_0001_1011;
73     data_in = 32'b01101010011100001010001100001100;
74     // options
75     byte_half_word    = 2'b00;
76     is_load_unsigned  = 0;
77     rw                = 1;
78     $display(">> addr: %h - data_in: %h", addr, data_in);
79     $display("options: store word,");
80     // execution
81     #10 ; initial_time = $time; valid = 1;
82     while(~ready & ~out_of_range) #10;
83     elapsed_time = $time-initial_time;
84     $display("store - Initial time: %t", initial_time);
85     $display("store - Elapsed time: %t", elapsed_time);
86     #10 ; valid = 0;
87     // options
88     byte_half_word    = 2'b00;
89     is_load_unsigned  = 0;
90     rw                = 0;
91     $display(">> addr: %h - data_in: %h", addr, data_in);
92     $display("options: load word,");
93     // execution
94     #10 ; initial_time = $time; valid = 1;
95     while(~ready & ~out_of_range) #10;
96     elapsed_time = $time-initial_time;

```

```

97     $display("load - Initial time: %t", initial_time);
98     $display("load - Elapsed time: %t", elapsed_time);
99     // test
100    $display("data_out: %h",data_out);
101    if(data_in == data_out)
102        $display("test 2: passed\n");
103    else $display("test 2: failed\n");
104    #10 ; valid = 0;
105
106    // test 3: guardar data (word) y leerla en el mismo bloque cache del test 2 pero otra
↪ palabra
107    $display("test 3: guardar data (word) y leerla en el mismo bloque cache del test 2 pero
↪ otra palabra");
108    addr    = 32'b0000000000000000_1001_0100_0001_1111;
109    data_in = 32'b011000110011100101000001100001100;
110    // options
111    byte_half_word    = 2'b00;
112    is_load_unsigned  = 0;
113    rw                = 1;
114    $display(">> addr: %h - data_in: %h", addr, data_in);
115    $display("options: store word",);
116    // execution
117    #10 ; initial_time = $time; valid = 1;
118    while(~ready & ~out_of_range) #10;
119    elapsed_time = $time-initial_time;
120    $display("store - Initial time: %t", initial_time);
121    $display("store - Elapsed time: %t", elapsed_time);
122    #10 ; valid = 0;
123    // options
124    byte_half_word    = 2'b00;
125    is_load_unsigned  = 0;
126    rw                = 0;
127    $display(">> addr: %h - data_in: %h", addr, data_in);
128    $display("options: load word",);
129    // execution
130    #10 ; initial_time = $time; valid = 1;
131    while(~ready & ~out_of_range) #10;
132    elapsed_time = $time-initial_time;
133    $display("load - Initial time: %t", initial_time);
134    $display("load - Elapsed time: %t", elapsed_time);
135    // test
136    $display("data_out: %h",data_out);
137    if(data_in == data_out)
138        $display("test 3: passed\n");
139    else $display("test 3: failed\n");
140    #10 ; valid = 0;
141
142    // test 4: leer la palabra almacenada en el test 2 (para comprobar que se haya
↪ almacenado correctamente despues de los tests anteriores)
143    $display("test 4: leer la palabra almacenada en el test 2 (para comprobar que se haya
↪ almacenado correctamente despues de los tests anteriores)");
144    addr    = 32'b0000000000000000_1001_0100_0001_1011;

```

```

145 data_in = 32'b011000110011100101000001100001100;
146 // options
147 byte_half_word = 2'b00;
148 is_load_unsigned = 0;
149 rw = 0;
150 $display(">> addr: %h - data_in: %h", addr, data_in);
151 $display("options: load word",);
152 // execution
153 #10 ; initial_time = $time; valid = 1;
154 while(~ready & ~out_of_range) #10;
155 elapsed_time = $time-initial_time;
156 $display("load - Initial time: %t", initial_time);
157 $display("load - Elapsed time: %t", elapsed_time);
158 // test
159 $display("data_out: %h",data_out);
160 if(data_out == 32'b01101010011100001010001100001100)
161     $display("test 4: passed\n");
162 else $display("test 4: failed\n");
163 #10 ; valid = 0;
164
165 // test 5: leer un byte signed
166 $display("test 5: leer un byte signed");
167 addr = 32'b0000000000000000_1001_0100_0001_1011;
168 data_in = 32'b011000110011100101000001100001100;
169 // options
170 byte_half_word = 2'b10;
171 is_load_unsigned = 0;
172 rw = 0;
173 $display(">> addr: %h - data_in: %h", addr, data_in);
174 $display("options: load byte signed",);
175 // execution
176 #10 ; initial_time = $time; valid = 1;
177 while(~ready & ~out_of_range) #10;
178 elapsed_time = $time-initial_time;
179 $display("load - Initial time: %t", initial_time);
180 $display("load - Elapsed time: %t", elapsed_time);
181 // test
182 $display("data_out: %h",data_out);
183 if(data_out == 32'b0000000000000000000000000000_01101010)
184     $display("test 5: passed\n");
185 else $display("test 5: failed\n");
186 #10 ; valid = 0;
187
188 // test 6: leer un byte signed
189 $display("test 6: leer un byte signed");
190 addr = 32'b0000000000000000_1001_0100_0001_1001;
191 data_in = 32'b011000110011100101000001100001100;
192 // options
193 byte_half_word = 2'b10;
194 is_load_unsigned = 0;
195 rw = 0;
196 $display(">> addr: %h - data_in: %h", addr, data_in);

```

```

197     $display("options: load byte signed");
198     // execution
199     #10 ; initial_time = $time; valid = 1;
200     while(~ready & ~out_of_range) #10;
201     elapsed_time = $time-initial_time;
202     $display("load - Initial time: %t", initial_time);
203     $display("load - Elapsed time: %t", elapsed_time);
204     // test
205     $display("data_out: %h",data_out);
206     if(data_out == 32'b11111111111111111111111111111111_10100011)
207         $display("test 6: passed\n");
208     else $display("test 6: failed\n");
209     #10 ; valid = 0;
210
211     // test 7: leer el mismo byte anterior pero unsigned
212     $display("test 7: leer el mismo byte anterior pero unsigned");
213     addr = 32'b00000000000000000000_1001_0100_0001_1001;
214     data_in = 32'b011000110011100101000001100001100;
215     // options
216     byte_half_word = 2'b10;
217     is_load_unsigned = 1;
218     rw = 0;
219     $display(">> addr: %h - data_in: %h", addr, data_in);
220     $display("options: load byte unsigned");
221     // execution
222     #10 ; initial_time = $time; valid = 1;
223     while(~ready & ~out_of_range) #10;
224     elapsed_time = $time-initial_time;
225     $display("load - Initial time: %t", initial_time);
226     $display("load - Elapsed time: %t", elapsed_time);
227     // test
228     $display("data_out: %h",data_out);
229     if(data_out == 32'b00000000000000000000000000000000_10100011)
230         $display("test 7: passed\n");
231     else $display("test 7: failed\n");
232     #10 ; valid = 0;
233
234     // test 8: ahora se modifica el byte leído en los test 6 y 7, luego se lee
235     $display("test 8: ahora se modifica el byte leído en los test 6 y 7, luego se lee");
236     addr = 32'b00000000000000000000_1001_0100_0001_1001;
237     data_in = 32'b0110001100111001010000011_11111100;
238     // options
239     byte_half_word = 2'b10;
240     is_load_unsigned = 0;
241     rw = 1;
242     $display(">> addr: %h - data_in: %h", addr, data_in);
243     $display("options: store byte");
244     // execution
245     #10 ; initial_time = $time; valid = 1;
246     while(~ready & ~out_of_range) #10;
247     elapsed_time = $time-initial_time;
248     $display("store - Initial time: %t", initial_time);

```

```

249     $display("store - Elapsed time: %t", elapsed_time);
250     #10 ; valid = 0;
251     // options
252     byte_half_word      = 2'b10;
253     is_load_unsigned    = 1;
254     rw                  = 0;
255     $display(">> addr: %h - data_in: %h", addr, data_in);
256     $display("options: load byte unsigned",);
257     // execution
258     #10 ; initial_time = $time; valid = 1;
259     while(~ready & ~out_of_range) #10;
260     elapsed_time = $time-initial_time;
261     $display("load - Initial time: %t", initial_time);
262     $display("load - Elapsed time: %t", elapsed_time);
263     // test
264     $display("data_out: %h",data_out);
265     if(data_out == 32'b000000000000000000000000_11111100)
266         $display("test 8: passed\n");
267     else $display("test 8: failed\n");
268     #10 ; valid = 0;
269
270     // test 9: se comprueba que la palabra donde se almacena el byte modificado en el test
    ↪ 8 mantenga el resto de sus bits en orden
271     $display("test 9: se comprueba que la palabra donde se almacena el byte modificado en
    ↪ el test 8 mantenga el resto de sus bits en orden");
272     addr  = 32'b0000000000000000_1001_0100_0001_1001;
273     data_in = 32'b011000110011100101000001100001100;
274     // options
275     byte_half_word      = 2'b00;
276     is_load_unsigned    = 1;
277     rw                  = 0;
278     $display(">> addr: %h - data_in: %h", addr, data_in);
279     $display("options: load word",);
280     // execution
281     #10 ; initial_time = $time; valid = 1;
282     while(~ready & ~out_of_range) #10;
283     elapsed_time = $time-initial_time;
284     $display("load - Initial time: %t", initial_time);
285     $display("load - Elapsed time: %t", elapsed_time);
286     // test
287     $display("data_out: %h",data_out);
288     if(data_out == 32'b0110101001110000_11111100_00001100)
289         $display("test 9: passed\n");
290     else $display("test 9: failed\n");
291     #10 ; valid = 0;
292
293     // test 10: se cambian los bits del tag c/r a la dirección del test 9 y se guarda un
    ↪ halfword, luego se lee el word
294     $display("test 10: se cambian los bits del tag c/r a la dirección del test 9 y se guarda
    ↪ un halfword, luego se lee el word");
295     addr  = 32'b0000000000000000_0001_0100_0001_1001;
296     data_in = 32'b011000110011100101000001100001100;

```

```

297 // options
298 byte_half_word    = 2'b01;
299 is_load_unsigned  = 0;
300 rw               = 1;
301 $display(">> addr: %h - data_in: %h", addr, data_in);
302 $display("options: store halfword",);
303 // execution
304 #10 ; initial_time = $time; valid = 1;
305 while(~ready & ~out_of_range) #10;
306 elapsed_time = $time-initial_time;
307 $display("store - Initial time: %t", initial_time);
308 $display("store - Elapsed time: %t", elapsed_time);
309 #10 ; valid = 0;
310 // options
311 byte_half_word    = 2'b00;
312 is_load_unsigned  = 1;
313 rw               = 0;
314 $display(">> addr: %h - data_in: %h", addr, data_in);
315 $display("options: load word",);
316 // execution
317 #10 ; initial_time = $time; valid = 1;
318 while(~ready & ~out_of_range) #10;
319 elapsed_time = $time-initial_time;
320 $display("load - Initial time: %t", initial_time);
321 $display("load - Elapsed time: %t", elapsed_time);
322 // test
323 $display("data_out: %h",data_out);
324 if(data_out == 32'b0000000000000000_1000001100001100)
325     $display("test 10: passed\n");
326 else $display("test 10: failed\n");
327 #10 ; valid = 0;
328
329 // test 11: se vuelve a usar la dirección del test 9 para leer el halfword almacenada en
↪ dicha dirección
330 $display("test 11: se vuelve a usar la dirección del test 9 para leer el halfword
↪ almacenada en dicha dirección");
331 addr = 32'b0000000000000000_1001_0100_0001_1001;
332 data_in = 32'b011000110011100101000001100001100;
333 // options
334 byte_half_word    = 2'b01;
335 is_load_unsigned  = 1;
336 rw               = 0;
337 $display(">> addr: %h - data_in: %h", addr, data_in);
338 $display("options: load halfword unsigned",);
339 // execution
340 #10 ; initial_time = $time; valid = 1;
341 while(~ready & ~out_of_range) #10;
342 elapsed_time = $time-initial_time;
343 $display("load - Initial time: %t", initial_time);
344 $display("load - Elapsed time: %t", elapsed_time);
345 // test
346 $display("data_out: %h",data_out);

```

```

347     if(data_out == 32'b0000000000000000_11111100_00001100)
348         $display("test 11: passed\n");
349     else $display("test 11: failed\n");
350     #10 ; valid = 0;
351
352     // test 12: misma operación del test 11 pero con signo
353     $display("test 12: misma operación del test 11 pero con signo");
354     addr    = 32'b0000000000000000_1001_0100_0001_1001;
355     data_in = 32'b011000110011100101000001100001100;
356     // options
357     byte_half_word    = 2'b01;
358     is_load_unsigned  = 0;
359     rw                = 0;
360     $display(">> addr: %h - data_in: %h", addr, data_in);
361     $display("options: load halfword signed",);
362     // execution
363     #10 ; initial_time = $time; valid = 1;
364     while(~ready & ~out_of_range) #10;
365     elapsed_time = $time-initial_time;
366     $display("load - Initial time: %t", initial_time);
367     $display("load - Elapsed time: %t", elapsed_time);
368     // test
369     $display("data_out: %h",data_out);
370     if(data_out == 32'b1111111111111111_11111100_00001100)
371         $display("test 12: passed\n");
372     else $display("test 12: failed\n");
373     #10 ; valid = 0;
374
375
376     #10 $finish;
377
378 end
379
380 endmodule

```

tb_FPU.sv: *test bench* encargado de comprobar el correcto funcionamiento del módulo *FPU.sv*.

```

1  /*
2  autor: Gianluca Vincenzo D'Agostino Matute
3  Santiago de Chile, Septiembre 2021
4  */
5  `timescale 1ns / 1ps
6
7  module tb_FPU;
8      // FPU inputs
9      reg start = 0; reg rst = 0; reg clk = 0;
10     reg [2:0] rm_to_FPU = 0; reg [4:0] option_to_FPU = 0;
11     reg [31:0] input_1 = 0; reg [31:0] input_2 = 0; reg [31:0] input_3 = 0;
12     // fpu_op_selection inputs
13     reg [4:0] rs2_add = 0; reg [3:0] funct7_out = 0;
14     reg [2:0] format_type = 0; reg [2:0] sub_format_type = 0;
15     reg [2:0] funct3 = 0; reg [2:0] rm_from_fcsr = 0;

```

```

16 // FPU outputs
17 wire ready, NV, NX, UF, OF, DZ;
18 wire [31:0] out;
19 // fpu_op_selection outputs
20 wire [2:0] rm_fpu_op_selection;
21 wire [4:0] option_fpu_op_selection;
22
23 FPU FPU_uut(
24     // Inputs
25     .start(start), .rst(rst), .clk(clk), // start: Need to be on til the op is ready
26     .rm(rm_to_FPU), .option(option_to_FPU), .in1(input_1), .in2(input_2), .in3(
    ↪ input_3),
27     // Outputs
28     .ready(ready), .NV(NV), .NX(NX), .UF(UF), .OF(OF), .DZ(DZ), .out(out)
29 );
30 fpu_op_selection fpu_op_selection_uut(
31     // Inputs
32     .rs2_add(rs2_add), .funct7_out(funct7_out),
33     .format_type(format_type), .sub_format_type(sub_format_type),
34     .funct3(funct3), .rm_from_fcsr(rm_from_fcsr),
35     // Outputs
36     .rm2fpu(rm_fpu_op_selection), .fpu_option(option_fpu_op_selection)
37 );
38
39 always #5 clk = ~clk;
40
41 initial begin
42     #10 start = 0;
43     // FPU
44     option_to_FPU = 5'b0_01_00;
45     rm_to_FPU = 3'b000;
46     input_1 = 32'b1111_1111_1111_1111_1111_1111_1010;
47     input_2 = 32'b11111111111100;
48     input_3 = 32'b0;
49     // fpu_op_selection
50     rs2_add      = 5'b00000;
51     funct7_out   = 4'b0110;
52     format_type  = 3'b000;
53     sub_format_type = 3'b000;
54     funct3       = 3'b111;
55     rm_from_fcsr = 3'b011;
56     ////////////////////////////////////////////////////////////////////
57     #10 start = 1;
58     while(~ready) #10;
59     // FPU
60     $display("FPU- out: %b", out);
61     $display("FPU- NV, NX, UF, OF, DZ: %b \n", {NV, NX, UF, OF, DZ});
62     // fpu_op_selection
63     $display("fpu_op_selection- rm2fpu: %b", rm_fpu_op_selection);
64     $display("fpu_op_selection- fpu_option: %b \n", option_fpu_op_selection);
65     #10 start = 0; #10 $finish;
66 end

```

`tb_riscv32imf_top.sv`: *test bench* encargado de comprobar el correcto funcionamiento de los módulos *core complex*: *riscv32imf_singlecycle* y *riscv32imf_pipeline* (en las líneas 19 y 20 se aprecia como se comenta la versión *Single-Cycle* y se deja por defecto la *Pipelined*). Está diseñado para ser ejecutado con el código ensamblador *testing_code_imf.s* cargado en las memorias del *core complex*.

```

1  /*
2  autor: Gianluca Vincenzo D'Agostino Matute
3  Santiago de Chile, Septiembre 2021
4
5  para: testing_code_imf.s
6  */
7  `timescale 1ns / 1ps
8
9  module tb_riscv32imf_top;
10     reg clk, rst, start, acces_to_registers_files, is_wb_data_fp_fromEEI, do_wb_fromEEI,
        ↪ is_rs1_fp_fromEEI, is_rs2_fp_fromEEI;
11     reg acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
        ↪ prog_is_load_unsigned_fromEEI;
12     reg acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
        ↪ data_is_load_unsigned_fromEEI;
13     reg [31:0] initial_PC, prog_addr_fromEEI, prog_in_fromEEI, data_addr_fromEEI,
        ↪ data_in_fromEEI, wb_data_fromEEI;
14     reg [1:0] prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI;
15     reg [4:0] rs1_add_fromEEI, rs2_add_fromEEI, wb_add_fromEEI;
16     wire ready, prog_ready_toEEI, prog_out_of_range_toEEI, data_ready_toEEI,
        ↪ data_out_of_range_toEEI;
17     wire [1:0] exit_status;
18     wire [31:0] prog_out_toEEI, data_out_toEEI, rs1_toEEI, rs2_toEEI, PC;
19     //riscv32imf_singlecycle
20     riscv32imf_pipeline
21     riscv32imf(
22         // inputs
23         .clk(clk), .rst(rst), .start(start), .acces_to_registers_files(acces_to_registers_files),
24         .is_wb_data_fp_fromEEI(is_wb_data_fp_fromEEI), .do_wb_fromEEI(
        ↪ do_wb_fromEEI),
25         .is_rs1_fp_fromEEI(is_rs1_fp_fromEEI), .is_rs2_fp_fromEEI(is_rs2_fp_fromEEI),
26         .acces_to_prog_mem(acces_to_prog_mem), .prog_rw_fromEEI(prog_rw_fromEEI),
27         .prog_valid_mem_fromEEI(prog_valid_mem_fromEEI), .
        ↪ prog_is_load_unsigned_fromEEI(prog_is_load_unsigned_fromEEI),
28         .acces_to_data_mem(acces_to_data_mem), .data_rw_fromEEI(data_rw_fromEEI),
29         .data_valid_mem_fromEEI(data_valid_mem_fromEEI), .
        ↪ data_is_load_unsigned_fromEEI(data_is_load_unsigned_fromEEI),
30         .initial_PC(initial_PC),
31         .prog_addr_fromEEI(prog_addr_fromEEI), .prog_in_fromEEI(prog_in_fromEEI),
32         .data_addr_fromEEI(data_addr_fromEEI), .data_in_fromEEI(data_in_fromEEI), .
        ↪ wb_data_fromEEI(wb_data_fromEEI),
33         .prog_byte_half_word_fromEEI(prog_byte_half_word_fromEEI), .
        ↪ data_byte_half_word_fromEEI(data_byte_half_word_fromEEI),
34         .rs1_add_fromEEI(rs1_add_fromEEI), .rs2_add_fromEEI(rs2_add_fromEEI), .
        ↪ wb_add_fromEEI(wb_add_fromEEI),

```

```

35 // outputs
36 .ready(ready),
37 .prog_ready_toEEI(prog_ready_toEEI), .prog_out_of_range_toEEI(
↪ prog_out_of_range_toEEI),
38 .data_ready_toEEI(data_ready_toEEI), .data_out_of_range_toEEI(
↪ data_out_of_range_toEEI),
39 .exit_status(exit_status),
40 .prog_out_toEEI(prog_out_toEEI), .data_out_toEEI(data_out_toEEI), .rs1_toEEI(
↪ rs1_toEEI), .rs2_toEEI(rs2_toEEI), .PC(PC)
41 );
42 always #5 clk = ~clk;
43 initial begin
44     int prog_out_file, data_out_file, registers_file;
45     string buff_str;
46     {clk, rst, start, acces_to_registers_files, is_wb_data_fp_fromEEI, do_wb_fromEEI,
↪ is_rs1_fp_fromEEI, is_rs2_fp_fromEEI,
47     acces_to_prog_mem, prog_rw_fromEEI, prog_valid_mem_fromEEI,
↪ prog_is_load_unsigned_fromEEI,
48     acces_to_data_mem, data_rw_fromEEI, data_valid_mem_fromEEI,
↪ data_is_load_unsigned_fromEEI,
49     initial_PC, prog_addr_fromEEI, prog_in_fromEEI, data_addr_fromEEI,
↪ data_in_fromEEI, wb_data_fromEEI,
50     prog_byte_half_word_fromEEI, data_byte_half_word_fromEEI, rs1_add_fromEEI,
↪ rs2_add_fromEEI, wb_add_fromEEI} = 0;
51     $display("\nBEGIN\n");
52     prog_out_file = $fopen("prog_out.mem", "w");
53     data_out_file = $fopen("data_out.mem", "w");
54     registers_file = $fopen("registers.mem", "w");
55     if(prog_out_file) $display("prog_out.mem - OK: %0d", prog_out_file); else $display("
↪ prog_out.mem - FAILED: %0d", prog_out_file);
56     if(data_out_file) $display("data_out.mem - OK: %0d", data_out_file); else $display("
↪ data_out.mem - FAILED: %0d", data_out_file);
57     if(registers_file) $display("registers.mem - OK: %0d", registers_file); else $display("
↪ registers.mem - FAILED: %0d", registers_file);
58     $display("\n");
59 // Se ejecuta el programa
60 #10 start = 1; while(~ready) #10;
61 $display("\nexit_status: %b\n", exit_status);
62 $display("\nPC: %h\n", PC+32'b100);
63 #10 start = 0;
64 // Se guarda el estado de la memoria de programa
65 acces_to_prog_mem = 1'b1;
66 while(1) begin
67     #10;
68     prog_valid_mem_fromEEI = 1'b1;
69     while(~prog_ready_toEEI & ~prog_out_of_range_toEEI) #10;
70     if(prog_out_of_range_toEEI == 1) break;
71     $fwrite(prog_out_file, prog_addr_fromEEI); $fwrite(prog_out_file, ": ");
72     $fwrite(prog_out_file, prog_out_toEEI); $fwrite(prog_out_file, "\n");
73     #10;
74     prog_valid_mem_fromEEI = 1'b0;
75     prog_addr_fromEEI += 4;

```

```

76     end
77     {prog_addr_fromEEI, acces_to_prog_mem, prog_valid_mem_fromEEI} = 0;
78     // Se guarda el estado de la memoria de datos
79     acces_to_data_mem = 1'b1;
80     while(1) begin
81         #10;
82         data_valid_mem_fromEEI = 1'b1;
83         while(~data_ready_toEEI & ~data_out_of_range_toEEI) #10;
84         if(data_out_of_range_toEEI == 1) break;
85         $fwriteh(data_out_file, data_addr_fromEEI); $fwrite(data_out_file, ": ");
86         $fwriteh(data_out_file, data_out_toEEI); $fwrite(data_out_file, "\n");
87         #10;
88         data_valid_mem_fromEEI = 1'b0;
89         data_addr_fromEEI += 4;
90     end
91     {data_addr_fromEEI, acces_to_data_mem, data_valid_mem_fromEEI} = 0;
92     // se guarda el estado del register files
93     acces_to_registers_files = 1'b1;
94     $fwrite(registers_file, "integer_regs\n");
95     {is_rs1_fp_fromEEI, is_rs2_fp_fromEEI} = 2'b00;
96     {rs1_add_fromEEI, rs2_add_fromEEI} = 10'b00000_00001;
97     for(int i = 0; i < 16; i++) begin
98         #10;
99         $fwriteh(registers_file, rs1_add_fromEEI); $fwrite(registers_file, ": ");
100        $fwriteh(registers_file, rs1_toEEI); $fwrite(registers_file, "\n");
101        $fwriteh(registers_file, rs2_add_fromEEI); $fwrite(registers_file, ": ");
102        $fwriteh(registers_file, rs2_toEEI); $fwrite(registers_file, "\n");
103        #10;
104        rs1_add_fromEEI += 5'b10;
105        rs2_add_fromEEI += 5'b10;
106        #10;
107    end
108    $fwrite(registers_file, "fp_regs\n");
109    {is_rs1_fp_fromEEI, is_rs2_fp_fromEEI} = 2'b11;
110    {rs1_add_fromEEI, rs2_add_fromEEI} = 10'b00000_00001;
111    for(int i = 0; i < 16; i++) begin
112        #10;
113        $fwriteh(registers_file, rs1_add_fromEEI); $fwrite(registers_file, ": ");
114        $fwriteh(registers_file, rs1_toEEI); $fwrite(registers_file, "\n");
115        $fwriteh(registers_file, rs2_add_fromEEI); $fwrite(registers_file, ": ");
116        $fwriteh(registers_file, rs2_toEEI); $fwrite(registers_file, "\n");
117        #10;
118        rs1_add_fromEEI += 5'b10;
119        rs2_add_fromEEI += 5'b10;
120        #10;
121    end
122    {rs1_add_fromEEI, rs2_add_fromEEI, is_rs1_fp_fromEEI, is_rs2_fp_fromEEI,
↵ acces_to_registers_files} = 0;
123    // fin
124    $display("\nEND\n");
125    $fclose(prog_out_file); $fclose(data_out_file); $fclose(registers_file); #10 $finish;
126 end

```

127 `endmodule`

`tb_TOP.sv`: *test bench* encargado de comprobar el correcto funcionamiento del módulo `TOP.sv`. Está diseñado para ser ejecutado con el código ensamblador `fibonacci_simple.s` cargado en las memorias del *core complex*.

```
1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4
5 para: fibonacci_simple.s
6 */
7 `timescale 1ns / 1ps
8
9 module tb_TOP;
10     reg clk, rst_asin, start_asin, resume_asin;
11     reg [15:0] in;
12     wire [5:0] color_leds;
13     wire [15:0] leds, display;
14     TOP TOP_uut(
15         .clk(clk), .rst_asin(rst_asin), .start_asin(start_asin), .resume_asin(resume_asin), .in(
16             ↪ in),
17         .color_leds(color_leds), .leds(leds), .display(display)
18     );
19     always #5 clk = ~clk;
20     initial begin
21         {clk, start_asin, resume_asin, in, rst_asin} = 20'b1;
22         $display("\nBEGIN\n"); #10
23
24         $display("\ncolor_leds: %b\n", color_leds);
25
26         #100 start_asin = 1'b1; #1000000 start_asin = 1'b0; // se inicia ejecución
27
28         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
29         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
30
31         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
32         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
33
34         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
35         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
36
37         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
38         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
39
40         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
41         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
42
43         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
44         #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
45
46         while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
```

```

46     #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
47
48     while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
49     #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
50
51     while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
52     #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
53
54     while(color_leds != 6'b101_101) #10; // mientras no printe un entero se espera
55     #100 resume_asin = 1'b1; #1000000 resume_asin = 1'b0; // se reanuda ejecución
56
57     while(color_leds != 6'b010_010) #10; // mientras no finalice
58
59     #1000000
60
61     $display("\ncolor_leds: %b\n", color_leds);
62
63     $display("\nEND\n"); #10 $finish;
64     end
65 endmodule

```

3 Códigos generadores de *Test Benches*

En esta sección se presentan todos los programas generadores de *test benches* para distintos módulos del *SoC* diseñado:

common_functions.h: archivo “header” que contiene los encabezados y funciones comunes para los programas: *tb_ALU_gen.c*, *tb_fp_converter_gen.c* y *tb_fp_arithmetic_unit_gen.c*.

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <math.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #pragma STDC FENV_ACCESS ON
7
8 int rand_range(int n){
9     int limit;
10    int r;
11    limit = RAND_MAX - (RAND_MAX % n);
12    while((r = rand()) >= limit);
13    return r % n; /* uniform random value in the range 0..n-1 */
14 }
15 int is_big_endian_test(void){
16    int num = 1;
17    if(*(char *)&num == 1){
18        printf("Is little endian\n");
19        return 0;
20    }
21    else{
22        printf("Is big endian\n");
23        return 1;
24    }
25 }
26 void get_binary(char str_buffer[37], void *input_ptr){
27    unsigned int *x, max, i;
28    x = (unsigned int*) input_ptr;
29    max = sizeof(*x)*8;
30    strcat(str_buffer, "32'b");
31    for (i = 0u; i < max; i++){
32        char bin_num[1];
33        *bin_num = (char) 48 + (((*x)>>(max-i-1u)) & 1);
34        strcat(str_buffer, bin_num);
35    }
36 }
```

tb_ALU_gen.c: programa generador de *test benches* para el módulo *ALU.sv*.

```
1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4
5 gcc tb_ALU_gen.c -o tb_ALU_gen
```

```

6 ./tb_ALU_gen
7
8 0 operation=00_000: + add
9 1 operation=00_001: - sub
10 2 operation=00_010: ^ xor
11 3 operation=00_011: | or
12 4 operation=00_100: & and
13 5 operation=00_101: << shift left
14 6 operation=00_110: >> shift righth
15 7 operation=00_111: >> shift righth (MSB extend)
16
17 8 operation=01_000: == equality
18 9 operation=01_001: != difference
19 10 operation=01_010: < is less than
20 11 operation=01_011: < is less than (unsigned)
21 12 operation=01_100: >= is bigger or equal
22 13 operation=01_101: >= is bigger or equal (unsigned)
23
24 14 operation=10_000: * mul
25 15 operation=10_001: * mul (unsigned)
26 16 operation=10_010: / div
27 17 operation=10_011: / div (unsigned)
28 18 operation=10_100: % modulo
29 19 operation=10_101: % modulo (unsigned)
30 */
31 #include "./common_functions.h"
32 void fput_test_ALU(FILE* file, char test_number[5], int option, char in1[37], char in2[37],
    ↪ char out[37], int boolean_out){
33 fputs("      $display(\">>>Test ", file); fputs(test_number, file); fputs(" - op: ", file);
34 if(option==0) fputs("+ add <<<\");\n      operation      = 5'b00_000", file);
35 else if(option== 1) fputs("- sub <<<\");\n      operation      = 5'b00_001", file);
36 else if(option== 2) fputs("^ xor <<<\");\n      operation      = 5'b00_010", file);
37 else if(option== 3) fputs("| or <<<\");\n      operation      = 5'b00_011", file);
38 else if(option== 4) fputs("& and <<<\");\n      operation      = 5'b00_100", file);
39 else if(option== 5) fputs("<< shift left <<<\");\n      operation      = 5'b00_101", file);
40 else if(option== 6) fputs(">> shift righth <<<\");\n      operation      = 5'b00_110", file);
41 else if(option== 7) fputs(">> shift righth (MSB extend) <<<\");\n      operation      = 5'
    ↪ b00_111", file);
42 else if(option== 8) fputs("== equality <<<\");\n      operation      = 5'b01_000", file);
43 else if(option== 9) fputs("!= difference <<<\");\n      operation      = 5'b01_001", file);
44 else if(option==10) fputs("< is less than <<<\");\n      operation      = 5'b01_010", file)
    ↪ ;
45 else if(option==11) fputs("< is less than (unsigned) <<<\");\n      operation      = 5'
    ↪ b01_011", file);
46 else if(option==12) fputs(">= is bigger or equal <<<\");\n      operation      = 5'
    ↪ b01_100", file);
47 else if(option==13) fputs(">= is bigger or equal (unsigned) <<<\");\n      operation
    ↪ = 5'b01_101", file);
48 else if(option==14) fputs("* mul <<<\");\n      operation      = 5'b10_000", file);
49 else if(option==15) fputs("* mul high <<<\");\n      operation      = 5'b11_000", file);
50 else if(option==16) fputs("* mul high (unsigned) <<<\");\n      operation      = 5'
    ↪ b11_001", file);

```

```

51 else if(option==17) fputs("* mul high (signed*unsigned) <<<\");\n      operation      =
   ↪ 5'b11_010", file);
52 else if(option==18) fputs("/ div <<<\");\n      operation      = 5'b10_011", file);
53 else if(option==19) fputs("/ div (unsigned) <<<\");\n      operation      = 5'b10_100",
   ↪ file);
54 else if(option==20) fputs("\ % modulo <<<\");\n      operation      = 5'b10_101", file);
55 else if(option==21) fputs("\ % modulo (unsigned) <<<\");\n      operation      = 5'
   ↪ b10_111", file);
56 fputs("\n      input_1      = ", file); fputs(in1, file);
57 fputs("\n      input_2      = ", file); fputs(in2, file);
58 fputs("\n      correct_out  = ", file); fputs(out, file);
59 fputs("\n      correct_boolean = 1'b", file); if(boolean_out) fputs("1;\n", file); else fputs
   ↪ ("0;\n", file);
60 fputs("      #10;\n      $display("\Inputs\");\n      $display("\operation: \%b\",
   ↪ operation);\n      $display("\in1      : \%b\", input_1);\n      $display("\in2      :
   ↪ \%b\", input_2);\n      $display("\Outputs\");\n", file);
61 fputs("      if(out == correct_out) $display("\res: PASSED!!\");\n      else begin\n
   ↪      out_errors += 1;\n      $display("\res: FAILED!!\");\n      $display("\
   ↪ res should be: \%b\", correct_out);\n      end\n      $display("\res      : \%b
   ↪ \", out);\n", file);
62 fputs("      if(boolean == correct_boolean) $display("\boolean_res: PASSED!!\");\n
   ↪      else begin\n      boolean_errors += 1;\n      $display("\boolean_res:
   ↪ FAILED!!\");\n      $display("\boolean_res should be: \%b\", correct_boolean)
   ↪ ;\n      end\n      $display("\boolean_res      : \%b\", boolean);\n", file);
63 fputs("      #10;\n      $display("\-----\");\n", file);
64 }
65 int main(){
66 FILE* file;
67 char percentage_info[5];
68 char str_num_tests[7]; // max input = 99999
69 int num_tests, num_tests_discarded;
70 int is_big_endian = is_big_endian_test();
71 file = fopen ("tb_ALU.sv", "w");
72 fputs("/*\nautor: Gianluca Vincenzo D'Agostino Matute\nSantiago de Chile, Septiembre
   ↪ 2021\nTest Bench for \"tb_ALU.sv\"\n*\n\timescale 1ns / 1ps\n\nmodule tb_ALU
   ↪ ;\n", file);
73 fputs(" reg [31:0] input_1 = 0; reg [31:0] input_2 = 0; reg [4:0] operation = 0;\n wire
   ↪ boolean; wire [31:0] out;\n", file);
74 fputs(" ALU ALU_uut(\n      .in1(input_1), .in2(input_2), .operation(operation),\n
   ↪      .res(out), .boolean_res(boolean)\n      );\n", file);
75 fputs(" initial begin\n      int out_errors = 0; int boolean_errors = 0;\n      reg
   ↪ [31:0] correct_out; reg correct_boolean;\n      #10 $display("\BEGIN\");\n\n", file)
   ↪ ;
76 printf("Enter n° of set test to write (max input 99999): ");
77 fgets(str_num_tests, sizeof(str_num_tests), stdin);
78 num_tests = atoi(str_num_tests);
79 memset(str_num_tests,0,sizeof(str_num_tests));
80 printf("Enter n° of initial set test to discard (max input 99999): ");
81 fgets(str_num_tests, sizeof(str_num_tests), stdin);
82 num_tests_discarded = atoi(str_num_tests);
83 memset(str_num_tests,0,sizeof(str_num_tests));
84 for(int i = 0 ; i < num_tests ; i++){

```

```

85     int in1, in2, out;
86     unsigned int in1_unsig, in2_unsig;
87     long long_out;
88     unsigned long long_out_unsig;
89     unsigned char *rand_byte_0, *rand_byte_1, *rand_byte_2, *rand_byte_3, *
↪ target_add;
90     char out_str[37];
91     char in1_str[37] = "";
92     char in2_str[37] = "";
93     char current_test_str[5] = "";
94     char test_title_begin[50] = " /*----- BEGIN: SET TEST ";
95     char test_title____end[50] = " /*----- END: SET TEST ";
96     // Se asigna espacio en memoria para cada byte aleatorio
97     rand_byte_0 = malloc(sizeof(unsigned char));
98     rand_byte_1 = malloc(sizeof(unsigned char));
99     rand_byte_2 = malloc(sizeof(unsigned char));
100    rand_byte_3 = malloc(sizeof(unsigned char));
101    // Se alinean los bytes
102    rand_byte_1 = rand_byte_0 + 1;
103    rand_byte_2 = rand_byte_1 + 1;
104    rand_byte_3 = rand_byte_2 + 1;
105    // Se determina dirección
106    target_add = is_big_endian ? rand_byte_3 : rand_byte_0;
107    // Se obtienen los bits aleatorios para cada input
108    for(int k = 0 ; k < 2 ; k++){
109        *rand_byte_0 = (unsigned char) rand_range(256);
110        *rand_byte_1 = (unsigned char) rand_range(256);
111        *rand_byte_2 = (unsigned char) rand_range(256);
112        *rand_byte_3 = (unsigned char) rand_range(256);
113        // Se obtienen los resultados aleatorios
114        if(k) {in1 = *( (unsigned int*) target_add ); in1_unsig = *( (unsigned int*)
↪ target_add );}
115        else {in2 = *( (unsigned int*) target_add ); in2_unsig = *( (unsigned int*)
↪ target_add );}
116    }
117    if(i >= num_tests_discarded) {
118        // Se escribe encabezado del test
119        sprintf(current_test_str, "%d", i);
120        strcat(test_title_begin, current_test_str);
121        strcat(test_title____end, current_test_str);
122        strcat(test_title_begin, " -----*/\n");
123        strcat(test_title____end, " -----*/\n\n");
124        fputs(test_title_begin, file);
125        // Se obtiene el string con la representación binaria para cada input
126        get_binary(in1_str, &in1);
127        get_binary(in2_str, &in2);
128        // Se realizan los tests
129
130        // 0 operation=00_000: + add
131        out = (unsigned int) ( ( (unsigned int) in1 ) + ( (unsigned int) in2 ) );
132        memset(out_str,0,sizeof(out_str));
133        get_binary(out_str, &out); // Se obtiene la representación binaria del resultado

```

```

134     fput_test_ALU(file, current_test_str, 0, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

135
136     // 1 operation=00_001: - sub
137     out = (unsigned int) ( ( (unsigned int) in1 ) - ( (unsigned int) in2 ) );
138     memset(out_str,0,sizeof(out_str));
139     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
140     fput_test_ALU(file, current_test_str, 1, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

141
142     // 2 operation=00_010: ^ xor
143     out = (unsigned int) ( ( (unsigned int) in1 ) ^ ( (unsigned int) in2 ) );
144     memset(out_str,0,sizeof(out_str));
145     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
146     fput_test_ALU(file, current_test_str, 2, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

147
148     // 3 operation=00_011: | or
149     out = (unsigned int) ( ( (unsigned int) in1 ) | ( (unsigned int) in2 ) );
150     memset(out_str,0,sizeof(out_str));
151     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
152     fput_test_ALU(file, current_test_str, 3, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

153
154     // 4 operation=00_100: & and
155     out = (unsigned int) ( ( (unsigned int) in1 ) & ( (unsigned int) in2 ) );
156     memset(out_str,0,sizeof(out_str));
157     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
158     fput_test_ALU(file, current_test_str, 4, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

159
160     // 5 operation=00_101: << shift left
161     out = (unsigned int) ( ( (unsigned int) in1 ) << ( (unsigned int) in2 ) );
162     memset(out_str,0,sizeof(out_str));
163     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
164     fput_test_ALU(file, current_test_str, 5, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

165
166     // 6 operation=00_110: >> shift righth
167     out = (unsigned int) ( ( (unsigned int) in1 ) >> ( (unsigned int) in2 ) );
168     memset(out_str,0,sizeof(out_str));
169     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
170     fput_test_ALU(file, current_test_str, 6, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

171
172     // 7 operation=00_111: >> shift righth (MSB extend)
173     out = (unsigned int) ( ( (int) in1 ) >> ( (unsigned int) in2 ) );
174     memset(out_str,0,sizeof(out_str));
175     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
176     fput_test_ALU(file, current_test_str, 7, in1_str, in2_str, out_str, 0); // Se
↪ imprime test

177

```

```

178 // 8a operation=01_000: == equality
179 out = (unsigned int) ( ( (unsigned int) in1 ) == ( (unsigned int) in2 ) );
180 memset(out_str,0,sizeof(out_str));
181 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
182 fputc_test_ALU(file, current_test_str, 8, in1_str, in2_str, out_str, out); // Se
↪ imprime test
183 // 8b operation=01_000: == equality
184 out = (unsigned int) ( ( (unsigned int) in1 ) == ( (unsigned int) in1 ) );
185 memset(out_str,0,sizeof(out_str));
186 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
187 fputc_test_ALU(file, current_test_str, 8, in1_str, in1_str, out_str, out); // Se
↪ imprime test
188 // 8c operation=01_000: == equality
189 out = (unsigned int) ( ( (unsigned int) in2 ) == ( (unsigned int) in2 ) );
190 memset(out_str,0,sizeof(out_str));
191 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
192 fputc_test_ALU(file, current_test_str, 8, in2_str, in2_str, out_str, out); // Se
↪ imprime test
193
194 // 9a operation=01_001: != difference
195 out = (unsigned int) ( ( (unsigned int) in1 ) != ( (unsigned int) in2 ) );
196 memset(out_str,0,sizeof(out_str));
197 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
198 fputc_test_ALU(file, current_test_str, 9, in1_str, in2_str, out_str, out); // Se
↪ imprime test
199 // 9b operation=01_001: != difference
200 out = (unsigned int) ( ( (unsigned int) in1 ) != ( (unsigned int) in1 ) );
201 memset(out_str,0,sizeof(out_str));
202 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
203 fputc_test_ALU(file, current_test_str, 9, in1_str, in1_str, out_str, out); // Se
↪ imprime test
204 // 9c operation=01_001: != difference
205 out = (unsigned int) ( ( (unsigned int) in2 ) != ( (unsigned int) in2 ) );
206 memset(out_str,0,sizeof(out_str));
207 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
208 fputc_test_ALU(file, current_test_str, 9, in2_str, in2_str, out_str, out); // Se
↪ imprime test
209
210 // 10a operation=01_010: < is less than
211 out = (unsigned int) ( ( (int) in1 ) < ( (int) in2 ) );
212 memset(out_str,0,sizeof(out_str));
213 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
214 fputc_test_ALU(file, current_test_str, 10, in1_str, in2_str, out_str, out); // Se
↪ imprime test
215 // 10b operation=01_010: < is less than
216 out = (unsigned int) ( ( (int) in1 ) < ( (int) in1 ) );
217 memset(out_str,0,sizeof(out_str));
218 get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
219 fputc_test_ALU(file, current_test_str, 10, in1_str, in1_str, out_str, out); // Se
↪ imprime test
220 // 10c operation=01_010: < is less than
221 out = (unsigned int) ( ( (int) in2 ) < ( (int) in2 ) );

```

```

222     memset(out_str,0,sizeof(out_str));
223     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
224     fputc_test_ALU(file, current_test_str, 10, in2_str, in2_str, out_str, out); // Se
↳ imprime test

225
226     // 11a operation=01_011: < is less than (unsigned)
227     out = (unsigned int) ( ( (unsigned int) in1 ) < ( (unsigned int) in2 ) );
228     memset(out_str,0,sizeof(out_str));
229     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
230     fputc_test_ALU(file, current_test_str, 11, in1_str, in2_str, out_str, out); // Se
↳ imprime test

231     // 11b operation=01_011: < is less than (unsigned)
232     out = (unsigned int) ( ( (unsigned int) in1 ) < ( (unsigned int) in1 ) );
233     memset(out_str,0,sizeof(out_str));
234     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
235     fputc_test_ALU(file, current_test_str, 11, in1_str, in1_str, out_str, out); // Se
↳ imprime test

236     // 11c operation=01_011: < is less than (unsigned)
237     out = (unsigned int) ( ( (unsigned int) in2 ) < ( (unsigned int) in2 ) );
238     memset(out_str,0,sizeof(out_str));
239     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
240     fputc_test_ALU(file, current_test_str, 11, in2_str, in2_str, out_str, out); // Se
↳ imprime test

241
242     // 12a operation=01_100: >= is bigger or equal
243     out = (unsigned int) ( ( (int) in1 ) >= ( (int) in2 ) );
244     memset(out_str,0,sizeof(out_str));
245     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
246     fputc_test_ALU(file, current_test_str, 12, in1_str, in2_str, out_str, out); // Se
↳ imprime test

247     // 12b operation=01_100: >= is bigger or equal
248     out = (unsigned int) ( ( (int) in1 ) >= ( (int) in1 ) );
249     memset(out_str,0,sizeof(out_str));
250     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
251     fputc_test_ALU(file, current_test_str, 12, in1_str, in1_str, out_str, out); // Se
↳ imprime test

252     // 12c operation=01_100: >= is bigger or equal
253     out = (unsigned int) ( ( (int) in2 ) >= ( (int) in2 ) );
254     memset(out_str,0,sizeof(out_str));
255     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
256     fputc_test_ALU(file, current_test_str, 12, in2_str, in2_str, out_str, out); // Se
↳ imprime test

257
258     // 13a operation=01_101: >= is bigger or equal (unsigned)
259     out = (unsigned int) ( ( (unsigned int) in1 ) >= ( (unsigned int) in2 ) );
260     memset(out_str,0,sizeof(out_str));
261     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
262     fputc_test_ALU(file, current_test_str, 13, in1_str, in2_str, out_str, out); // Se
↳ imprime test

263     // 13b operation=01_101: >= is bigger or equal (unsigned)
264     out = (unsigned int) ( ( (unsigned int) in1 ) >= ( (unsigned int) in1 ) );
265     memset(out_str,0,sizeof(out_str));

```

```

266     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
267     fput_test_ALU(file, current_test_str, 13, in1_str, in1_str, out_str, out); // Se
↳ imprime test
268     // 13c operation=01_101: >= is bigger or equal (unsigned)
269     out = (unsigned int) ( ( (unsigned int) in2 ) >= ( (unsigned int) in2 ) );
270     memset(out_str,0,sizeof(out_str));
271     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
272     fput_test_ALU(file, current_test_str, 13, in2_str, in2_str, out_str, out); // Se
↳ imprime test
273
274     // 14 operation=10_000: * mul
275     out = (unsigned int) ( ( (int) in1 ) * ( (int) in2 ) );
276     memset(out_str,0,sizeof(out_str));
277     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
278     fput_test_ALU(file, current_test_str, 14, in1_str, in2_str, out_str, 0); // Se
↳ imprime test
279
280     // 15 operation=11_000: * mul high
281     long_out = (long) ((long) in1)*((long) in2);
282     long_out = long_out >> 32;
283     memset(out_str,0,sizeof(out_str));
284     get_binary(out_str, &long_out); // Se obtiene la representación binaria del
↳ resultado
285     fput_test_ALU(file, current_test_str, 15, in1_str, in2_str, out_str, 0); // Se
↳ imprime test
286
287     // 16 operation=11_001: * mul high (unsigned)
288     long_out_unsig = (unsigned long) ((unsigned long) in1_unsig)*((unsigned long)
↳ in2_unsig);
289     long_out_unsig = long_out_unsig >> 32;
290     memset(out_str,0,sizeof(out_str));
291     get_binary(out_str, &long_out_unsig); // Se obtiene la representación binaria del
↳ resultado
292     fput_test_ALU(file, current_test_str, 16, in1_str, in2_str, out_str, 0); // Se
↳ imprime test
293
294     // 17 operation=11_010: * mul high (signed*unsigned)
295     long_out = (long) ((long) in1)*((unsigned long) in2_unsig);
296     long_out = long_out >> 32;
297     memset(out_str,0,sizeof(out_str));
298     get_binary(out_str, &long_out); // Se obtiene la representación binaria del
↳ resultado
299     fput_test_ALU(file, current_test_str, 17, in1_str, in2_str, out_str, 0); // Se
↳ imprime test
300
301     // 18 operation=10_011: / div
302     out = (unsigned int) ( ( (int) in1 ) / ( (int) in2 ) );
303     memset(out_str,0,sizeof(out_str));
304     get_binary(out_str, &zout); // Se obtiene la representación binaria del resultado
305     fput_test_ALU(file, current_test_str, 18, in1_str, in2_str, out_str, 0); // Se
↳ imprime test
306

```

```

307 // 19 operation=10_100: / div (unsigned)
308 out = (unsigned int) ( ( (unsigned int) in1 ) / ( (unsigned int) in2 ) );
309 memset(out_str,0,sizeof(out_str));
310 get_binary(out_str, &out); // Se obtiene la representación binaria del resultado
311 fput_test_ALU(file, current_test_str, 19, in1_str, in2_str, out_str, 0); // Se
↪ imprime test
312
313 // 20 operation=10_101: % modulo
314 out = (unsigned int) ( ( (int) in1 ) % ( (int) in2 ) );
315 memset(out_str,0,sizeof(out_str));
316 get_binary(out_str, &out); // Se obtiene la representación binaria del resultado
317 fput_test_ALU(file, current_test_str, 20, in1_str, in2_str, out_str, 0); // Se
↪ imprime test
318
319 // 21 operation=10_111: % modulo (unsigned)
320 out = (unsigned int) ( ( (unsigned int) in1 ) % ( (unsigned int) in2 ) );
321 memset(out_str,0,sizeof(out_str));
322 get_binary(out_str, &out); // Se obtiene la representación binaria del resultado
323 fput_test_ALU(file, current_test_str, 21, in1_str, in2_str, out_str, 0); // Se
↪ imprime test
324
325 // Titulo de cierre del set de pruebas
326 fputs(test_title____end, file);
327 }
328
329 }
330 fputs("      $display(\"Total errors found      : \ %d ( \ %f percentage)\", out_errors
↪ +boolean_errors, (out_errors+boolean_errors)*100.0/", file);
331 sprintf(percentage_info, " %d", (num_tests-num_tests_discarded)*(22+12)*2); fputs(
↪ percentage_info, file); memset(percentage_info,0,sizeof(percentage_info)); sprintf(
↪ percentage_info, " %d", (num_tests-num_tests_discarded)*(22+12));
332 fputs(");\n      $display(\"Total out errors found      : \ %d ( \ %f percentage)\",
↪ out_errors, out_errors*100.0/", file); fputs(percentage_info, file);
333 fputs(");\n      $display(\"Total boolean errors found    : \ %d ( \ %f percentage)\",
↪ boolean_errors, boolean_errors*100.0/", file); fputs(percentage_info, file);
334 fputs(");\n      $display(\"END\"); #10 $finish;\n      end\nendmodule", file);
335 fclose(file);
336 printf("N° of tests written: %d\nOperation ready, file generated: tb_fp_arithmetic_unit.
↪ sv\n", (num_tests-num_tests_discarded)*(22+12));
337 return 0;
338 }

```

tb_fp_converter_gen.c: programa generador de *test benches* para el módulo *fp_converter.sv*.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4
5 gcc tb_fp_converter_gen.c -o tb_fp_converter_gen -lm
6 ./tb_fp_converter_gen
7
8 option: 0 -> Integer to fp / 1 -> fp to Integer

```

```

9  rm =
10  000 -> RNE - code = 0
11  001 -> RTZ - code = 1
12  010 -> RDN - code = 2
13  011 -> RUP - code = 3
14  100 -> RMM - code = 4 (not tested)
15
16  https://www.h-schmidt.net/FloatConverter/IEEE754.html
17
18  */
19
20  #include "./common_functions.h"
21
22  void fput_test_fp_converter(FILE* file, char test_number[4], int option, int
    ↪ integer_is_signed, int rm,
23      char in[37], char out[37], int NV, int NX){
24  fputs("      $display(">>>Test ", file); fputs(test_number, file);
25  if(option) if(integer_is_signed) fputs("-fp2signed", file); else fputs("-fp2unsigned", file);
26  else if(integer_is_signed) fputs("-signed2fp", file); else fputs("-unsigned2fp", file);
27  if(rm==0) fputs("-RNE", file); else if(rm==1) fputs("-RTZ", file); else if(rm==2) fputs("-
    ↪ RDN", file); else if(rm==3) fputs("-RUP", file); else if(rm==4) fputs("-RMM", file);
28  fputs(" <<<");\n      option = 1'b", file);
29  if(option) fputs("1;\n", file); else fputs("0;\n", file);
30  fputs("      integer_is_signed = 1'b", file);
31  if(integer_is_signed) fputs("1;\n", file); else fputs("0;\n", file);
32  fputs("      rm = 3'b", file);
33  if(rm==0)fputs("000", file); else if(rm==1) fputs("001", file); else if(rm==2) fputs("010",
    ↪ file); else if(rm==3) fputs("011", file); else fputs("100", file);
34  fputs(";\n      in = ", file); fputs(in, file); fputs(";\n      correct_out = ", file); fputs(
    ↪ out, file);
35  fputs(";\n      correct_NV = 1'b", file); if(NV) fputs("1", file); else fputs("0", file);
36  fputs(";\n      correct_NX = 1'b", file); if(NX) fputs("1", file); else fputs("0", file);
37  fputs(";\n      $display(\"in: \%b\", in);\n      $display(\"integer_is_signed: \%b\",
    ↪ integer_is_signed);\n      $display(\"option: \%b\", option);\n      $display(\"rm:
    ↪ \%b\", rm);\n", file);
38  fputs("      #10 start = 1;\n      initial_time = $time;\n      while(~ready) #10;\n
    ↪      elapsed_time = $time-initial_time;\n", file);
39  fputs("      $display(\"Initial time: \%t\", initial_time);\n      $display(\"Elapsed time
    ↪ : \%t\", elapsed_time);\n", file);
40  fputs("      if(out == correct_out) $display(\"out: PASSED!!\");\n      else begin\n
    ↪      out_errors += 1;\n      $display(\"out: FAILED!!\");\n", file);
41  fputs("      $display(\"out should be: \%b\", correct_out);\n      end\n
    ↪      $display(\"out: \%b\", out);\n", file);
42  fputs("      if(NV == correct_NV) $display(\"NV: PASSED!!\");\n      else begin\n
    ↪      NV_errors += 1;\n      $display(\"NV: FAILED!!\");\n", file);
43  fputs("      $display(\"NV should be: \%b\", correct_NV);\n      end\n
    ↪      $display(\"NV: \%b\", NV);\n", file);
44  fputs("      if(NX == correct_NX) $display(\"NX: PASSED!!\");\n      else begin\n
    ↪      NX_errors += 1;\n      $display(\"NX: FAILED!!\");\n", file);
45  fputs("      $display(\"NX should be: \%b\", correct_NX);\n      end\n
    ↪      $display(\"NX: \%b\", NX);\n", file);
46  fputs("      if(elapsed_time > max_elapsed_time) max_elapsed_time = elapsed_time;\n

```

```

    ↪ n      if(elapsed_time < min_elapsed_time) min_elapsed_time = elapsed_time;\n
    ↪      #10 start = 0;\n      $display("\n-----\n");\n", file);
47 }
48
49 int main(){
50     FILE* file;
51     char percentage_info[4];
52     char str_num_tests[7]; // max input = 99999
53     int num_tests, num_tests_discarded;
54     int original_rounding = fegetround();
55     int is_big_endian = is_big_endian_test();
56     file = fopen ("tb_fp_converter sv", "w");
57     fputs("/*\nautor: Gianluca Vincenzo D'Agostino Matute\nSantiago de Chile, Septiembre
    ↪ 2021\nTest Bench for \"fp_converter sv\"\n*/\n'timescale 1ns / 1ps\n\nmodule
    ↪ tb_fp_converter;\n", file);
58     fputs("    reg start = 0; reg rst = 0; reg clk = 0;\n    reg option = 0; reg
    ↪ integer_is_signed = 0;\n", file);
59     fputs("    reg [2:0] rm = 0; reg [31:0] in = 0;\n    // option: 0 -> Integer to fp | 1 -> fp to
    ↪ Integer\n", file);
60     fputs("    wire NV, NX, ready; wire [31:0] out;\n    fp_converter uut(\n        .start(start),
    ↪ .rst(rst), .clk(clk),\n", file);
61     fputs("        .integer_is_signed(integer_is_signed), .option(option), .rm(rm), .in(in),\n
    ↪ .NV(NV), .NX(NX), .ready(ready), .out(out)\n", file);
62     fputs("    );\n    always #5 clk = ~clk;\n    initial begin\n", file);
63     fputs("        int out_errors = 0; int NV_errors = 0; int NX_errors = 0;\n        int
    ↪ min_elapsed_time = 100000; int max_elapsed_time = 0;\n", file);
64     fputs("        int initial_time, elapsed_time;\n        reg [31:0] correct_out; reg
    ↪ correct_NV, correct_NX;\n        $display("\nBEGIN\n"); start = 0;\n\n", file);
65     printf("Enter n° of set test to write (max input 99999): ");
66     fgets(str_num_tests, sizeof(str_num_tests), stdin);
67     num_tests = atoi(str_num_tests);
68     memset(str_num_tests,0,sizeof(str_num_tests));
69     printf("Enter n° of initial set test to discard (max input 99999): ");
70     fgets(str_num_tests, sizeof(str_num_tests), stdin);
71     num_tests_discarded = atoi(str_num_tests);
72     memset(str_num_tests,0,sizeof(str_num_tests));
73     for(int i = 0 ; i < num_tests ; i++){
74         int random_signed;
75         unsigned int random_unsigned;
76         float random_float;
77         unsigned char *rand_byte_0, *rand_byte_1, *rand_byte_2, *rand_byte_3, *
    ↪ target_add;
78         char in_str[37] = "";
79         char current_test_str[5] = "";
80         char test_title_begin[50] = "        /*----- BEGIN: SET TEST ";
81         char test_title_end[50] = "        /*----- END: SET TEST ";
82         // Se asigna espacio en memoria para cada byte aleatorio
83         rand_byte_0 = malloc(sizeof(unsigned char));
84         rand_byte_1 = malloc(sizeof(unsigned char));
85         rand_byte_2 = malloc(sizeof(unsigned char));
86         rand_byte_3 = malloc(sizeof(unsigned char));
87         // Se alinean los bytes

```

```

88     rand_byte_1 = rand_byte_0 + 1;
89     rand_byte_2 = rand_byte_1 + 1;
90     rand_byte_3 = rand_byte_2 + 1;
91     // Se obtienen los bits aleatorios
92     *rand_byte_0 = (unsigned char) rand_range(256);
93     *rand_byte_1 = (unsigned char) rand_range(256);
94     *rand_byte_2 = (unsigned char) rand_range(256);
95     *rand_byte_3 = (unsigned char) rand_range(256);
96     if(i >= num_tests_discarded) {
97         // Titulo del set de pruebas
98         sprintf(current_test_str, "%d", i);
99         strcat(test_title_begin, current_test_str);
100        strcat(test_title____end, current_test_str);
101        strcat(test_title_begin, " -----*/\n");
102        strcat(test_title____end, " -----*/\n\n");
103        fputs(test_title_begin, file);
104        // Se determina dirección
105        target_add = is_big_endian ? rand_byte_3 : rand_byte_0;
106        // Se obtienen los enteros y el float aleatorios a partir de los bytes aleatorios
107        random_signed  = *( (int*) target_add );
108        random_unsigned = *( (unsigned int*) target_add );
109        random_float   = *( (float*) target_add );
110        // se obtiene representación binaria
111        get_binary(in_str, target_add);
112        // Ciclo para cada modo de redondeo
113        for(int j = 0 ; j < 4 ; j++){
114            int fp2signed;
115            unsigned int fp2unsigned;
116            float signed2fp, unsigned2fp;
117            int NV, NX;
118            char out_str[37];
119            if(j==0) fesetround(FE_TONEAREST); else if(j==1) fesetround(
↪ FE_TOWARDZERO);
120            else if(j==2) fesetround(FE_DOWNWARD); else if(j==3) fesetround(
↪ FE_UPWARD);
121
122            // {option, integer_is_signed} = {0, 0}: unsigned2fp
123            if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
↪ FAILED!!\n"); // Se limpian las excepciones
124            unsigned2fp = (float) random_unsigned;
125            NV = fetestexcept(FE_INVALID);
126            NX = fetestexcept(FE_INEXACT);
127            memset(out_str,0,sizeof(out_str));
128            get_binary(out_str, &unsigned2fp); // Se obtiene la representación binaria del
↪ resultado
129            fput_test_fp_converter(file, current_test_str, 0, 0, j, in_str, out_str, NV, NX);
↪ // Se imprime test
130
131            // {option, integer_is_signed} = {0, 1}: signed2fp
132            if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
↪ FAILED!!\n"); // Se limpian las excepciones
133            signed2fp = (float) random_signed;

```

```

134     NV = fetestexcept(FE_INVALID);
135     NX = fetestexcept(FE_INEXACT);
136     memset(out_str,0,sizeof(out_str));
137     get_binary(out_str, &signed2fp); // Se obtiene la representación binaria del
↪ resultado
138     fput_test_fp_converter(file, current_test_str, 0, 1, j, in_str, out_str, NV, NX);
↪ // Se imprime test

139
140     // {option, integer_is_signed} = {1, 0}: fp2unsigned
141     if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
↪ FAILED!!\n"); // Se limpian las excepciones
142     fp2unsigned = (unsigned int) random_float;
143     NV = fetestexcept(FE_INVALID);
144     NX = fetestexcept(FE_INEXACT);
145     // Inicio: Manejando los resultados de intel
146     if( (fp2unsigned != (unsigned int) (rintf(random_float))) & rintf(random_float)
↪ >=0 ) fp2unsigned = (unsigned int) (rintf(random_float));
147     if(random_float >= 4294967168.0 | isnanf(random_float)) {fp2unsigned =
↪ 4294967295; NV = 1;}
148     if(random_float < 0.0) {fp2unsigned = 0; NV = 1;}
149     //--- Fin: Manejando los resultados de intel
150     memset(out_str,0,sizeof(out_str));
151     get_binary(out_str, &fp2unsigned); // Se obtiene la representación binaria del
↪ resultado
152     fput_test_fp_converter(file, current_test_str, 1, 0, j, in_str, out_str, NV, NX);
↪ // Se imprime test

153
154     // {option, integer_is_signed} = {1, 1}: fp2signed
155     if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
↪ FAILED!!\n"); // Se limpian las excepciones
156     fp2signed = (int) random_float;
157     NV = fetestexcept(FE_INVALID);
158     NX = fetestexcept(FE_INEXACT);
159     // Inicio: Manejando los resultados de intel
160     if(fp2signed != (int) (rintf(random_float)) ) fp2signed = (int) (rintf(
↪ random_float));
161     if(random_float >= 2147483648.0 | isnanf(random_float)) fp2signed =
↪ 2147483647;
162     //--- Fin: Manejando los resultados de intel
163     memset(out_str,0,sizeof(out_str));
164     get_binary(out_str, &fp2signed); // Se obtiene la representación binaria del
↪ resultado
165     fput_test_fp_converter(file, current_test_str, 1, 1, j, in_str, out_str, NV, NX);
↪ // Se imprime test
166     }
167     // Titulo de cierre del set de pruebas
168     fputs(test_title____end, file);
169     }
170 }
171 /* falta agregar casos borde a mano */
172 fputs("      $display(\"Total errors found: \%d (\%f percentage)\", out_errors+
↪ NV_errors+NX_errors, (out_errors+NV_errors+NX_errors)*100.0/", file);

```

```

173 sprintf(percentage_info, "%d", (num_tests-num_tests_discarded)*4*4*3); fputs(
    ↪ percentage_info, file); memset(percentage_info,0,sizeof(percentage_info)); sprintf(
    ↪ percentage_info, "%d", (num_tests-num_tests_discarded)*4*4);
174 fputs(");\n    $display(\"Total out_errors errors found: \%d (\%f percentage)\",
    ↪ out_errors, out_errors*100.0/", file); fputs(percentage_info, file);
175 fputs(");\n    $display(\"Total NV_errors errors found: \%d (\%f percentage)\",
    ↪ NV_errors, NV_errors*100.0/", file); fputs(percentage_info, file);
176 fputs(");\n    $display(\"Total NX_errors errors found: \%d (\%f percentage)\",
    ↪ NX_errors, NX_errors*100.0/", file); fputs(percentage_info, file);
177 fputs(");\n    $display(\"Minimum elapsed time: %d\", min_elapsed_time);\n", file);
    ↪ fputs("    $display(\"Maximum elapsed time: %d\", max_elapsed_time);\n", file);
178 fputs("    $display(\"END\"); #10 $finish;\n    end\nendmodule", file);
179 fclose(file); fesetround(original_rounding);
180 printf("N° of tests written: %d\nOperation ready, file generated: tb_fp_arithmetic_unit.
    ↪ sv\n", (num_tests-num_tests_discarded)*4*4);
181 return 0;
182 }

```

tb_fp_arithmetic_unit_gen.c: programa generador de *test benches* para el módulo *fp_arithmetic_unit*.

```

1 /*
2 autor: Gianluca Vincenzo D'Agostino Matute
3 Santiago de Chile, Septiembre 2021
4
5 gcc tb_fp_arithmetic_unit_gen.c -o tb_fp_arithmetic_unit_gen -lm
6 ./tb_fp_arithmetic_unit_gen
7
8 op =
9 000 -> in1+in2  ADD                code= 0
10 001 -> in1*in2  MUL                code= 1
11 010 -> in1/in2  DIV                code= 2
12 011 -> sqrt(in1) SQRT (not exactly equal) code= 3
13 100 -> in1-in2  SUB                code= 4
14 101 -> -in1*in2 -MUL (not tested)
15 110 -> -in1/in2 -DIV (not tested)
16 rm =
17 000 -> RNE code= 0
18 001 -> RTZ code= 1
19 010 -> RDN code= 2
20 011 -> RUP code= 3
21 100 -> RMM code= 4 (not tested)
22 status =
23 000 -> no flag (used)    code= 0
24 101 -> no flag (not used)
25 110 -> no flag (not used)
26 -----
27 001 -> UF flag          code= 1
28 010 -> OF flag          code= 2
29 011 -> NV flag          code= 3
30 100 -> DZ flag          code= 4
31 111 -> NX flag          code= 5
32

```

```

33 https://www.h-schmidt.net/FloatConverter/IEEE754.html
34 */
35
36 #include "./common_functions.h"
37
38 int exceptions_checker(void){
39     if(fetestexcept(FE_DIVBYZERO)) return 4;
40     else if(fetestexcept(FE_INVALID)) return 3;
41     else if(fetestexcept(FE_OVERFLOW)) return 2;
42     else if(fetestexcept(FE_UNDERFLOW)) return 1;
43     else if(fetestexcept(FE_INEXACT)) return 5;
44     else return 0;
45 }
46
47 void fput_test_fp_arithmetic_unit(FILE* file, char test_number[5], int op, int rm,
48     char in1[37], char in2[37], char out[37], int status){
49     fputs("      $display("\">>>>Test ", file); fputs(test_number, file);
50     if(op==0) fputs("-ADD", file); else if(op==1) fputs("-MUL", file); else if(op==2) fputs("-
    ↪ DIV", file); else if(op==3) fputs("-SQRT", file); else if(op==4) fputs("-SUB", file);
51     if(rm==0) fputs("-RNE", file); else if(rm==1) fputs("-RTZ", file); else if(rm==2) fputs("-
    ↪ RDN", file); else if(rm==3) fputs("-RUP", file); else if(rm==4) fputs("-RMM", file);
52     fputs(" <<<");\n      operation      = 3'b", file);
53     if(op==0) fputs("000;\n", file); else if(op==1) fputs("001;\n", file); else if(op==2) fputs("
    ↪ 010;\n", file); else if(op==3) fputs("011;\n", file); else if(op==4) fputs("100;\n", file);
54     fputs("      rounding_mode = 3'b", file);
55     if(rm==0) fputs("000;\n", file); else if(rm==1) fputs("001;\n", file); else if(rm==2) fputs(
    ↪ "010;\n", file); else if(rm==3) fputs("011;\n", file); else if(rm==4) fputs("100;\n", file
    ↪ );
56     fputs("      input_1 = ", file); fputs(in1, file); fputs(";\n      input_2 = ", file); fputs(
    ↪ in2, file); fputs(";\n      correct_out = ", file); fputs(out, file);
57     fputs(";\n      correct_status = 3'b", file);
58     if(status==0) fputs("000;\n", file); else if(status==1) fputs("001;\n", file); else if(status
    ↪ ==2) fputs("010;\n", file); else if(status==3) fputs("011;\n", file); else if(status==4)
    ↪ fputs("100;\n", file); else if(status==5) fputs("111;\n", file);
59     fputs("      $display("\In1: \%b\", input_1);\n      $display("\In2: \%b\", input_2);\n
    ↪ n      $display("\Operation: \%b\", operation);\n      $display("\Rounding_Mode:
    ↪ \%b\", rounding_mode);\n", file);
60     fputs("      #200 start = 1;\n      initial_time = $time;\n      while(~ready) #10;\n
    ↪      elapsed_time = $time-initial_time;\n      $display("\Initial time: \%t\",
    ↪      initial_time);\n      $display("\Elapsed time: \%t\", elapsed_time);\n", file);
61
62     if(op==2 | op==3) fputs("      if(out == correct_out | out == {correct_out[31:23],
    ↪ correct_out[22:0]+23'b1} | out == {correct_out[31:23], correct_out[22:0]-23'b1} |
    ↪ out == {correct_out[31:23], correct_out[22:0]+23'b10} | out == {correct_out
    ↪ [31:23], correct_out[22:0]-23'b10} | out[30:0]==31'
    ↪ b11111111100000000000000000000000) $display("\out: PASSED!!");\n", file);
63     else fputs("      if(out == correct_out | out[30:0]==31'
    ↪ b11111111100000000000000000000000) $display("\out: PASSED!!");\n", file);
64
65     fputs("      else begin\n          man_out_errors += 1;\n          $display("\out:
    ↪ FAILED!!");\n          $display("\out should be: \%b\", correct_out);\n          end\n
    ↪ n      $display("\out      : \%b\", out);\n", file);

```

```

66 fputs("    if(status == correct_status) $display(\"status: PASSED!!\");\n    else
    ↪ begin\n        status_out_errors += 1;\n        $display(\"status: FAILED!!\");\n
    ↪ n        $display(\"status should be: \\%b\\\", correct_status);\n        end\n
    ↪ $display(\"status: \\%b\\\", status);\n\", file);
67 fputs("    if(elapsed_time > max_elapsed_time) max_elapsed_time = elapsed_time;\n
    ↪ n    if(elapsed_time < min_elapsed_time) min_elapsed_time = elapsed_time;\n
    ↪    #200 start = 0;\n        $display(\"-----\");\n\", file);
68 }
69
70 int main(){
71     FILE* file;
72     char percentage_info[4];
73     char str_num_tests[7]; // max input = 99999
74     int num_tests, num_tests_discarded;
75     int original_rounding = fegetround();
76     int is_big_endian = is_big_endian_test();
77     file = fopen ("tb_fp_arithmetic_unit.sv", "w");
78     fputs("/*\nautor: Gianluca Vincenzo D'Agostino Matute\nSantiago de Chile, Septiembre
    ↪ 2021\nTest Bench for \"tb_fp_arithmetic_unit.sv\"\n*/\n\timescale 1ns / 1ps\n\
    ↪ nmodule tb_fp_arithmetic_unit;\n\", file);
79     fputs(" //global inputs\n    reg start = 0; reg rst = 0; reg clk = 0;\n\", file);
80     fputs("    reg [31:0] input_1 = 0; reg [31:0] input_2 = 0;\n    reg [2:0] operation = 0; reg
    ↪ [2:0] rounding_mode = 0;\n\", file);
81     fputs(" //global outputs\n    wire ready; wire [2:0] status; wire [31:0] out;\n\n\", file);
82     fputs(" // uut\n    fp_arithmetic_unit uut_fp_arithmetic_unit(\n\", file);
83     fputs("        //input\n            .start(start), .rst(rst), .clk(clk),\n            .op(operation), .rm(
    ↪ rounding_mode),\n\", file);
84     fputs("        .in1(input_1), .in2(input_2),\n        //output\n            .ready(ready), .status(
    ↪ status), .out(out)\n        );\n\n\", file);
85     fputs("    always #5 clk = ~clk;\n\n    initial begin\n        int man_out_errors = 0; int
    ↪ status_out_errors = 0;\n\", file);
86     fputs("        int min_elapsed_time = 100000; int max_elapsed_time = 0;\n        int
    ↪ initial_time, elapsed_time;\n\n\", file);
87     fputs("        reg [31:0] correct_out; reg [2:0] correct_status;\n\n        $display(\"BEGIN
    ↪ \"); start = 0;\n\n\", file);
88     printf("Enter n° of set test to write (max input 99999): ");
89     fgets(str_num_tests, sizeof(str_num_tests), stdin);
90     num_tests = atoi(str_num_tests);
91     memset(str_num_tests,0,sizeof(str_num_tests));
92     printf("Enter n° of initial set test to discard (max input 99999): ");
93     fgets(str_num_tests, sizeof(str_num_tests), stdin);
94     num_tests_discarded = atoi(str_num_tests);
95     memset(str_num_tests,0,sizeof(str_num_tests));
96     for(int i = 0 ; i < num_tests ; i++){
97         float in1, in2;
98         unsigned char *rand_byte_0, *rand_byte_1, *rand_byte_2, *rand_byte_3, *
    ↪ target_add;
99         char in1_str[37] = "";
100        char in2_str[37] = "";
101        char current_test_str[5] = "";
102        char test_title_begin[50] = "        /*----- BEGIN: SET TEST ";
103        char test_title_end[50] = "        /*----- END: SET TEST ";

```

```

104 // Se asigna espacio en memoria para cada byte aleatorio
105 rand_byte_0 = malloc(sizeof(unsigned char));
106 rand_byte_1 = malloc(sizeof(unsigned char));
107 rand_byte_2 = malloc(sizeof(unsigned char));
108 rand_byte_3 = malloc(sizeof(unsigned char));
109 // Se alinean los bytes
110 rand_byte_1 = rand_byte_0 + 1;
111 rand_byte_2 = rand_byte_1 + 1;
112 rand_byte_3 = rand_byte_2 + 1;
113 // Se determina dirección
114 target_add = is_big_endian ? rand_byte_3 : rand_byte_0;
115 // Se obtienen los bits aleatorios para cada input
116 for(int k = 0 ; k < 2 ; k++){
117     *rand_byte_0 = (unsigned char) rand_range(256);
118     *rand_byte_1 = (unsigned char) rand_range(256);
119     *rand_byte_2 = (unsigned char) rand_range(256);
120     *rand_byte_3 = (unsigned char) rand_range(256);
121     // Se obtienen los resultados aleatorios
122     if(k in1 = *(float*) target_add ); else in2 = *(float*) target_add );
123 }
124 if(i >= num_tests_discarded) {
125     // Se escribe encabezado del test
126     sprintf(current_test_str, "%d", i);
127     strcat(test_title_begin, current_test_str);
128     strcat(test_title____end, current_test_str);
129     strcat(test_title_begin, " -----*/\n");
130     strcat(test_title____end, " -----*/\n\n");
131     fputs(test_title_begin, file);
132     // Se obtiene el string con la representación binaria para cada input
133     get_binary(in1_str, &in1);
134     get_binary(in2_str, &in2);
135     // Ciclo para cada modo de redondeo
136     for(int j = 0 ; j < 4 ; j++){
137         int op, status;
138         float out;
139         char out_str[37];
140         if(j==0) fesetround(FE_TONEAREST); else if(j==1) fesetround(
↪ FE_TOWARDZERO);
141         else if(j==2) fesetround(FE_DOWNWARD); else if(j==3) fesetround(
↪ FE_UPWARD);
142
143         // ADD
144         if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
↪ FAILED!!\n"); // Se limpian las excepciones
145         out = (in1+in2); // Se realiza la operación por testear
146         status = exceptions_checker(); // Se obtiene el status correspondiente
147         memset(out_str,0,sizeof(out_str));
148         if(isnanf(out)) {strcat(out_str, "32'b111111111000000000000000000000");
↪ status=3;}
149         else get_binary(out_str, &out); // Se obtiene la representación binaria del
↪ resultado
150         fput_test_fp_arithmetic_unit(file, current_test_str, 0, j, in1_str, in2_str,

```

```

150 ↪ out_str, status); // Se imprime test
151
152 // MUL
153 if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
154 ↪ FAILED!!\n"); // Se limpian las excepciones
155 out = (in1*in2); // Se realiza la operación por testear
156 status = exceptions_checker(); // Se obtiene el status correspondiente
157 memset(out_str,0,sizeof(out_str));
158 if(isnanf(out)) {strcat(out_str, "32'b11111111110000000000000000000000");
159 ↪ status=3;}
160 else get_binary(out_str, &out); // Se obtiene la representación binaria del
161 ↪ resultado
162 fput_test_fp_arithmetic_unit(file, current_test_str, 1, j, in1_str, in2_str,
163 ↪ out_str, status); // Se imprime test
164
165 // DIV
166 if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
167 ↪ FAILED!!\n"); // Se limpian las excepciones
168 out = (in1/in2); // Se realiza la operación por testear
169 status = exceptions_checker(); // Se obtiene el status correspondiente
170 memset(out_str,0,sizeof(out_str));
171 if(isnanf(out)) {strcat(out_str, "32'b11111111110000000000000000000000");
172 ↪ status=3;}
173 else get_binary(out_str, &out); // Se obtiene la representación binaria del
174 ↪ resultado
175 fput_test_fp_arithmetic_unit(file, current_test_str, 2, j, in1_str, in2_str,
176 ↪ out_str, status); // Se imprime test
177
178 // SQRT
179 if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
180 ↪ FAILED!!\n"); // Se limpian las excepciones
181 out = sqrt(in1); // Se realiza la operación por testear
182 status = exceptions_checker(); // Se obtiene el status correspondiente
183 memset(out_str,0,sizeof(out_str));
184 if(isnanf(out)) {strcat(out_str, "32'b11111111110000000000000000000000");
185 ↪ status=3;}
186 else get_binary(out_str, &out); // Se obtiene la representación binaria del
187 ↪ resultado
188 fput_test_fp_arithmetic_unit(file, current_test_str, 3, j, in1_str, in2_str,
189 ↪ out_str, status); // Se imprime test
190
191 // SUB
192 if(feclearexcept(FE_ALL_EXCEPT)) printf("feclearexcept(FE_ALL_EXCEPT)
193 ↪ FAILED!!\n"); // Se limpian las excepciones
194 out = in1 - in2; // Se realiza la operación por testear
195 status = exceptions_checker(); // Se obtiene el status correspondiente
196 memset(out_str,0,sizeof(out_str));
197 if(isnanf(out)) {strcat(out_str, "32'b11111111110000000000000000000000");
198 ↪ status=3;}
199 else get_binary(out_str, &out); // Se obtiene la representación binaria del
200 ↪ resultado
201 fput_test_fp_arithmetic_unit(file, current_test_str, 4, j, in1_str, in2_str,

```

```

    ↪ out_str, status); // Se imprime test
187     }
188     // Titulo de cierre del set de pruebas
189     fputs(test_title____end, file);
190 }
191 }
192 fputs("      $display(\"Total errors found: %d (\ %f percentage)\", man_out_errors+
    ↪ status_out_errors, (man_out_errors+status_out_errors)*100.0/", file);
193 sprintf(percentage_info, "%d", (num_tests-num_tests_discarded)*4*5*2); fputs(
    ↪ percentage_info, file); memset(percentage_info,0,sizeof(percentage_info)); sprintf(
    ↪ percentage_info, "%d", (num_tests-num_tests_discarded)*4*5);
194 fputs(");\n      $display(\"Total out errors found: %d (\ %f percentage)\",
    ↪ man_out_errors, man_out_errors*100.0/", file); fputs(percentage_info, file);
195 fputs(");\n      $display(\"Total status errors found: %d (\ %f percentage)\",
    ↪ status_out_errors, status_out_errors*100.0/", file); fputs(percentage_info, file);
196 fputs(");\n      $display(\"Minimum elapsed time: %d\", min_elapsed_time);\n
    ↪ $display(\"Maximum elapsed time: %d\", max_elapsed_time);\n", file);
197 fputs("      $display(\"END\"); #10 $finish;\n  end\nendmodule", file);
198 fclose(file);
199 fesetround(original_rounding);
200 printf("N° of tests written: %d\nOperation ready, file generated: tb_fp_arithmetic_unit.
    ↪ sv\n", (num_tests-num_tests_discarded)*4*5);
201 return 0;
202 }

```

4 Códigos *Assembler*

En esta sección se presentan todos los programas en lenguaje ensamblador de *Risc-V* utilizados para corroborar el correcto funcionamiento del *core complex* y del *SoC*:

testing_code_imf.s: código ensamblador de prueba específico para verificar el *core complex*, se carga en el mismo para ejecutar el *test bench tb_riscv32imf_top.sv*.

```
1 .text
2   addi    x0, x0, 2047    # x0 = 0 siempre
3   addi    x1, x0, 2047    # x1 = 2047
4   xori    x2, x1, 1024    # x2 = x1 ^ 1024
5   ori     x4, x2, 1563    # x4 = x2 | 1563
6   andi    x5, x4, 1333    # x5 = x4 & 1333
7   slli    x6, x5, 11      # x6 = x5 << 11
8   addi    x7, x0, -78     # x7 = -78
9   srai    x7, x7, 4       # x7 = x7 >> 4 (signed)
10  srli    x7, x7, 1       # x7 = x7 >> 1 (unsigned)
11  beq     x4, x1, label_1  # x4 == x1 -> label_1
12 label_2:
13  sub     x12, x11, x4     # x11 = x11 - x4
14  xor     x13, x12, x4    # x13 = x12 ^ x4
15  or      x14, x6, x2     # x14 = x6 | x2
16  and     x15, x6, x2     # x15 = x6 & x2
17  bge     x4, x7, label_3  # x4 >= x7 -> label_3
18  addi    x9, x0, 1       # x9 = 1
19  sll     x16, x4, x9     # x16 = x4 << x9
20  blt     x4, x7, label_4  # x4 < x7 -> label_4
21 label_1:
22  add     x8, x4, x4       # x8 = x4 + x4
23  addi    x9, x0, 2       # x9 = 2
24  mul     x10, x4, x9     # x10 = x4 * x9
25  sll     x11, x4, x9     # x11 = x4 << x9
26  bge     x8, x10, label_2 # x8 >= x10 -> label_2
27 label_4:
28  sb      x12, 8(x1)      # M[x1+8][7:0] = x12[7:0] -> sb x1, x12, 8
29  sb      x12, 16(x2)     # M[x2+16][7:0] = x7[7:0] -> sb x2, x7, 16
30  sll     x7, x7, x9     # x7 = x7 << x9
31  sra     x18, x7, x9     # x18 = x7 >> x9 (signed)
32  srl     x19, x7, x9     # x19 = x7 >> x9 (unsigned)
33  sub     x14, x14, x6    # x14 = x14 - x6
34  addi    x14, x14, -899   # x14 = x14 - 900
35  sh      x19, 8(x14)     # M[x14+8][15:0] = x19[15:0] -> sb x14, x19, 8
36  sw      x11, 16(x14)    # M[x14+16][31:0] = x11[31:0] -> sb x14, x11, 16
37  jal     x20, label_5    # x20 = PC + 4, PC = label_5
38 label_3:
39  lbu     x21, 16(x14)    # x21[7:0] = M[x14+16][7:0] -> lb x21, x14, 16 (unsigned)
40  lbu     x22, 17(x14)    # x22[7:0] = M[x14+17][7:0] -> lb x22, x14, 17 (unsigned)
41  slli    x22, x22, 8     # x22 = x22 << 8
42  lbu     x23, 18(x14)    # x23[7:0] = M[x14+18][7:0] -> lb x23, x14, 18 (unsigned)
43  slli    x23, x23, 16    # x23 = x23 << 16
44  lbu     x24, 19(x14)    # x24[7:0] = M[x14+19][7:0] -> lb x24, x14, 19 (unsigned)
```

```

45  slli    x24, x24, 24      # x24 = x24 << 24
46  add     x25, x21, x22    # x25 = x21 + x22
47  add     x25, x25, x23    # x25 = x25 + x23
48  add     x25, x25, x24    # x25 = x25 + x24
49  bne     x25, x19, label_5 # x25 != x19 -> label_5
50  bitu    x7, x25, label_2  # x7 < x25 -> label_2 (unsigned)
51  bgeu    x7, x25, label_6  # x7 >= x25 -> label_6 (unsigned)
52 label_7:
53  rem     x25, x25, x23    # x25 = x25 % x23
54  remu    x25, x18, x25    # x25 = x25 % x23 (unsigned)
55  mulhu   x9, x25, x18    # x9 = x25 * x18 (h) (unsigned)
56  mulhsu  x2, x18, x7     # x9 = x18 * x7 (h) (signed*unsigned)
57  jal     x0, label_8      # PC = label_8
58 label_5:
59  lb      x21, 16(x14)     # x21[7:0] = M[x14+16][7:0] -> lb x21, x14, 16
60  lb      x22, 17(x14)     # x22[7:0] = M[x14+17][7:0] -> lb x22, x14, 17
61  lb      x23, 18(x14)     # x23[7:0] = M[x14+18][7:0] -> lb x23, x14, 18
62  lb      x24, 19(x14)     # x24[7:0] = M[x14+19][7:0] -> lb x24, x14, 19
63  add     x26, x21, x22    # x26 = x21 + x22
64  add     x26, x26, x23    # x26 = x26 + x23
65  add     x26, x26, x24    # x26 = x26 + x24
66  jalr    x20, x20, 0     # x20 = PC + 4, PC = x20 + 0
67 label_6:
68  lui     x30, 500         # x30 = label_7 << 12
69  auipc   x31, 500         # x30 = PC + label_7 << 12
70  slt     x27, x12, x22    # x27 = x12 < x22 ? 1 : 0
71  slti    x28, x12, 100    # x27 = x12 < 100 ? 1 : 0
72  add     x27, x27, x28    # x27 = x27 + x28
73  sltiu   x28, x12, -10    # x28 = x12 < -10 ? 1 : 0 (unsigned)
74  add     x27, x27, x28    # x27 = x27 + x28
75  sltu    x28, x18, x7     # x27 = x18 < x7 ? 1 : 0 (unsigned)
76  add     x27, x27, x28    # x27 = x27 + x28
77  lw      x28, 16(x14)     # x28[31:0] = M[x14+16][31:0] -> lb x28, x14, 16
78  add     x28, x28, x27    # x28 = x28 + x27
79  lh      x27, 16(x14)     # x27[15:0] = M[x14+16][15:0] -> lb x27, x14, 16
80  lhu     x29, 16(x14)     # x29[15:0] = M[x14+16][15:0] -> lb x29, x14, 16
81  divu    x27, x27, x29    # x27 = x27 / x29 (unsigned)
82  mulh    x1, x23, x27     # x1 = x23 * x27 (h)
83  mul     x1, x23, x27     # x1 = x23 * x27
84  div     x1, x23, x27     # x1 = x23 / x27
85  jal     x20, label_7     # x20 = PC + 4, PC = label_7
86 label_8:
87  flw     f0, 16(x14)     # f0[31:0] = M[x14+16][31:0] -> flw f0, x14, 16
88  fmv.w.x x1, f12         # x12 -> f1 #----# fmv.w.x f1, x12
89  fadd.s  f2, f0, f1      # f2 = f0 + f1
90  fdiv.s  f3, f2, f0      # f3 = f2 / f0
91  fsw     f3, 32(x14)     # M[x14+32][31:0] = f3[31:0] -> fsw x14, f3, 32
92  flw     f4, 32(x14)     # f4[31:0] = M[x14+32][31:0] -> flw f4, x14, 32
93  fsgnjn.s f4, f4, f0     # f4 = abs(f4) * -sgn(f0)
94  remu    x25, x25, x0    # x25 = x25
95  fmul.s  f6, f3, f2      # f6 = f3 * f2
96  fsgnjn.s f7, f0, f0     # f4 = abs(f0) * -sgn(f0)

```

```

97 fsub.s    f8, f6, f7    # f8 = f6 - f7
98 fmaddd.s f5, f3, f2, f0 # f5 = f3 * f2 + f0
99 fsqrt.s  f9, f7        # f9 = sqrt(f7)
100 fsgnj.s  f10, f7, f4   # f10 = abs(f7) * sgn(f4)
101 fsgnjx.s f11, f7, f4   # f11 = f7 * sgn(f4)
102 fmsub.s  f12, f4, f3, f10 # f12 = f4 * f3 - f10
103 fmin.s   f13, f11, f12  # f13 = min(f11, f12)
104 fmax.s   f14, f11, f12  # f13 = max(f11, f12)
105 fnmadd.s f15, f13, f4, f4 # f15 = -f13 * f4 + f10
106 fnmsub.s f16, f13, f4, f5 # f16 = -f13 * f4 - f10
107 fmv.x.w  x1, f7        # f7 -> x1
108 feq.s    x23, f0, f14   # x23 = (f0 == f14) ? 1 : 0
109 fit.s    x24, f0, f14   # x24 = (f0 < f14) ? 1 : 0
110 add      x23, x23, x24   # x23 += x24
111 fle.s    x24, f11, f14   # x23 = (f0 <= f14) ? 1 : 0
112 add      x23, x23, x24   # x23 += x24
113 add      x1, x23, x1     # x1 += x23
114 fcvt.wu.s x23, f3       # x23 = uint(f3)
115 fsgnjn.s f3, f3, f3     # f4 = abs(f3) * -sgn(f3)
116 fcvt.w.s  x24, f3       # x24 = int(f3)
117 add      x23, x24, x23   # x23 += x24 -> x23 = 0
118 fcvt.s.w  f17, x7       # f17 = fp(int(x7))
119 fclass.s  x24, f8       # x24 = class(f8)
120 fcvt.s.wu f18, x7       # f18 = fp(uint(x7)) ### error en el fp_converter
121
122 addi      x17, x0, 10     # syscall para terminar el programa
123 ecall

```

fibonacci_simple.s: código de prueba específico para verificar el *core complex* y el *SoC*, se carga para ejecutar el *test bench tb_TOP.sv*.

```

1 .eqv INIT_ARG    3
2 .eqv MAX_ARG     7
3
4 .text
5
6 main:
7 sw    s0, (sp)          # se guarda s0 en la dirección apuntada por el stack pointer
8 add   s0, x0, sp        # s0 = sp
9 addi  sp, sp, -16       # se reserva espacio para 4 words (4bytes*4=16): sp -= 16
10 addi  a1, x0, INIT_ARG # a1 = INIT_ARG
11 sw    a1, -4(s0)       # Se guarda a1
12 addi  a2, x0, MAX_ARG  # a2 = MAX_ARG
13 sw    a2, -8(s0)       # Se guarda a2
14 # Hay espacio sin utilizar para otra variable en -12(s0).
15 # Pero el stack pointer debe ser "16-byte aligned" (https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf P3)
16 loop:
17 add   a0, x0, a1        # a0 = a1 -> permite mostrar a1 (el argumento de entrada a
18 # la función fibonacci)
18 li    x17, 1           # syscall 1: PrintInt
19 ecall                  # Print int del argumento, a0, de entrada a la función

```

```

20 jal ra, fibonacci      # Se llama la función, recibe arg en a0
21 lw a1, -4(s0)         # Se recupera el arg de entrada a la función
22 li x17, 1             # syscall 1: PrintInt
23 ecall                 # Print int de la respuesta ya almacenada en a0
24 addi a1, a1, 1        # arg++
25 sw a1, -4(s0)         # se guarda en nuevo argumento
26 ble a1, a2, loop
27 done:
28 li x17, 10           # syscall 10: Exit
29 ecall
30
31 # int fibonacci(int n){ if(n <= 2) return 1; else return fibonacci(n-1) + fibonacci(n-2); }
32 fibonacci:           # Recibe n en a0 y retorna el resultado en el mismo registro
33 # Se almacenan los registros
34 sw s0, (sp)          # se guarda s0 en la dirección apuntada por el stack pointer
35 add s0, x0, sp        # s0 = sp
36 # Se necesita 1 variable local + sp + ra: 3 words en el stack
37 # (recordar: el stack pointer debe ser "16-byte aligned")
38 # 3 words*4 bytes/address = 12 bytes, pero el sp debe apuntar al siguiente espacio
   ↪ disponible: sp-16
39 addi sp, sp, -16
40 sw ra, -4(s0)         # Se guarda: Dirección de retorno
41 sw a0, -8(s0)         # Se guarda: Argumento de entrada
42 addi a1, x0, 2        # a1 = 2
43 ble a0, a1, fib_basecase # if(n <= 2) -> fib_basecase
44 # 1era llamada recursiva: fibonacci(n-1)
45 addi a0, a0, -1       # n-1
46 jal ra, fibonacci     # a0 <- fib(n-1)
47 sw a0, -12(s0)        # -12(s0) <- a0
48 # 2da llamada recursiva: fibonacci(n-2)
49 lw a0, -8(s0)         # a0 <- n
50 addi a0, a0, -2       # n-2
51 jal ra, fibonacci     # a0 <- fib(n-2)
52 # return fibonacci(n-1) + fibonacci(n-2);
53 lw a1, -12(s0)        # a1 <- fib(n-1)
54 add a0, a0, a1        # a0 <- fibonacci(n-1) + fibonacci(n-2)
55 jal x0, fib_ret
56 fib_basecase:        # if(n <= 2) return 1;
57 addi a0, x0, 1        # a0 = 1
58 fib_ret:             # return
59 lw ra, -4(s0)         # Se restaura la dirección de retorno
60 lw s0, (s0)           # Se recupera el stack pointer
61 jalr x0, ra, 0        # Se vuelve a la dirección donde se invocó la función

```

fibonacci_inputs.s: código de prueba para ser ejecutado en *hardware*, se basa en *fibonacci_simple.s* pero añadiendo mayor interacción con el usuario.

```

1 .text
2
3 main:
4 sw s0, (sp)          # se guarda s0 en la dirección apuntada por el stack pointer
5 add s0, x0, sp        # s0 = sp

```

```

6  addi  sp, sp, -16      # se reserva espacio para 4 words (4bytes*4=16): sp -= 16
7  li    x17, 5          # syscall 5: GetInt
8  ecall                # Get int: INIT_ARG en a0
9  add   a1, x0, a0      # a1 = a0
10 sw    a1, -4(s0)      # Se guarda a1
11 ebreak               # Simplemente para ayudar a diferenciar cada input
12 li    x17, 5          # syscall 5: GetInt
13 ecall                # Get int: MAX_ARG en a0
14 add   a2, x0, a0      # a2 = a0
15 sw    a2, -8(s0)      # Se guarda a2
16 # Hay espacio sin utilizar para otra variable en -12(s0).
17 # Pero el stack pointer debe ser "16-byte aligned" (https://riscv.org/wp-content/uploads/
   ↪ 2015/01/riscv-calling.pdf P3)
18 loop:
19 add   a0, x0, a1      # a0 = a1 -> permite mostrar a1 (el argumento de entrada a
   ↪ la función fibonacc)
20 jal   ra, fibonacc    # Se llama la función, recibe arg en a0
21 lw    a1, -4(s0)      # Se recupera el arg de entrada a la función
22 li    x17, 1          # syscall 1: PrintInt
23 ecall                # Print int de la respuesta ya almacenada en a0
24 addi  a1, a1, 1       # arg++
25 sw    a1, -4(s0)      # se guarda en nuevo argumento
26 ble   a1, a2, loop
27 done:
28 li    x17, 10         # syscall 10: Exit
29 ecall
30
31 # int fibonacc(int n){ if(n <= 2) return 1; else return fibonacc(n-1) + fibonacc(n-2); }
32 fibonacc: # Recibe n en a0 y retorna el resultado en el mismo registro
33 # Se almacenan los registros
34 sw    s0, (sp)        # se guarda s0 en la dirección apuntada por el stack pointer
35 add   s0, x0, sp      # s0 = sp
36 # Se necesita 1 variable local + sp + ra: 3 words en el stack
37 # (recordar: el stack pointer debe ser "16-byte aligned")
38 # 3 words*4 bytes/address = 12 bytes, pero el sp debe apuntar al siguiente espacio
   ↪ disponible: sp-16
39 addi  sp, sp, -16
40 sw    ra, -4(s0)      # Se guarda: Dirección de retorno
41 sw    a0, -8(s0)      # Se guarda: Argumento de entrada
42 addi  a1, x0, 2       # a1 = 2
43 ble   a0, a1, fib_basecase # if(n <= 2) -> fib_basecase
44 # 1era llamada recursiva: fibonacc(n-1)
45 addi  a0, a0, -1      # n-1
46 jal   ra, fibonacc    # a0 <- fib(n-1)
47 sw    a0, -12(s0)     # -12(s0) <- a0
48 # 2da llamada recursiva: fibonacc(n-2)
49 lw    a0, -8(s0)      # a0 <- n
50 addi  a0, a0, -2      # n-2
51 jal   ra, fibonacc    # a0 <- fib(n-2)
52 # return fibonacc(n-1) + fibonacc(n-2);
53 lw    a1, -12(s0)     # a1 <- fib(n-1)
54 add   a0, a0, a1      # a0 <- fibonacc(n-1) + fibonacc(n-2)

```

```

55 jal    x0, fib_ret
56 fib_basecase:  # if(n <= 2) return 1;
57 addi   a0, x0, 1      # a0 = 1
58 fib_ret:      # return
59 lw     ra, -4(s0)     # Se restaura la dirección de retorno
60 lw     s0, (s0)      # Se recupera el stack pointer
61 jalr   x0, ra, 0     # Se vuelve a la dirección donde se invocó la función

```

operating_fp.s: código de prueba para ser ejecutado en *hardware* con enfoque en el testeo de las operaciones de punto flotante e interacción con el usuario.

```

1 .data
2
3 begin_msg:    .word  0xBBBBBBBB
4 end_msg:     .word  0xEEEEEEEE
5
6 .text
7
8 main:
9 lw    a0, 0(x0) #begin_msg
10 li   x17, 1 # syscall 1: PrintInt
11 ecall
12 get_fps:
13 fmv.w.x x0, ft0 # ft0=0 por algún motivo en rars los operandos deben estar al revés solo
    ↪ para esta instrucción
14 li   x17, 6 # syscall 1: GetFp
15 ecall
16 fadd.s ft1, fa0, ft0 # ft1 operando 1
17 li   x17, 6 # syscall 1: GetFp
18 ecall
19 fadd.s ft2, fa0, ft0 # ft2 operando 2
20 operations:
21 fmul.s  ft3, ft1, ft2      # ft3 = ft1*ft2
22 fadd.s  ft4, ft1, ft2      # ft4 = ft1+ft2
23 fsub.s  ft5, ft3, ft4      # ft5 = ft3-ft4
24 fdiv.s  ft6, ft4, ft1      # ft6 = ft4/ft1
25 fsqrt.s ft7, ft3          # ft7 = sqrt(ft3)
26 fmadd.s ft8, ft3, ft6, ft1 # ft8 = ft3*ft6+ft1
27 fmsub.s ft9, ft3, ft6, ft1 # ft9 = ft3*ft6-ft1
28 fnmadd.s ft10, ft3, ft6, ft1 # ft10 = -ft3*ft6+ft1
29 fnmsub.s ft11, ft3, ft6, ft1 # ft11 = -ft3*ft6-ft1
30 printing_res:
31 fadd.s fa0, ft0, ft3
32 li   x17, 2 # syscall 2: PrintFp
33 ecall
34 fadd.s fa0, ft0, ft4
35 li   x17, 2 # syscall 2: PrintFp
36 ecall
37 fadd.s fa0, ft0, ft5
38 li   x17, 2 # syscall 2: PrintFp
39 ecall
40 fadd.s fa0, ft0, ft6

```

```

41  li    x17, 2  # syscall 2: PrintFp
42  ecall
43  fadd.s fa0, ft0, ft7
44  li    x17, 2  # syscall 2: PrintFp
45  ecall
46  fadd.s fa0, ft0, ft8
47  li    x17, 2  # syscall 2: PrintFp
48  ecall
49  fadd.s fa0, ft0, ft9
50  li    x17, 2  # syscall 2: PrintFp
51  ecall
52  fadd.s fa0, ft0, ft10
53  li    x17, 2  # syscall 2: PrintFp
54  ecall
55  fadd.s fa0, ft0, ft11
56  li    x17, 2  # syscall 2: PrintFp
57  ecall
58  done:
59  lw    a0, 4(x0)#end_msg
60  li    x17, 1  # syscall 1: PrintInt
61  ecall
62  li    x17, 10 # syscall 10: Exit
63  ecall

```

4.1 *Script* conversor de palabras

Los programas en lenguaje ensamblador presentados anteriormente se ejecutan en *RARS* para contrastar su funcionamiento con el de las simulaciones y el de las pruebas sobre *hardware*. Esta misma herramienta se utiliza para compilar el programa y cargarlo en el *SoC* implementado. Sin embargo, hace falta el siguiente programa en *Python*:

singleWord2fourWords.py: *script* de *Python 2.7* encargado de recibir la memoria de programa y de datos compilados en texto hexadecimal (cada línea del archivo contiene 8 dígitos hexadecimales). Este programa recibe dos archivos, *text_in* y *data_in*, correspondientes a la memoria de programa y datos, respectivamente. Luego retorna una versión convertida de los mismos, *text_in_formatted* y *data_in_formatted*, respectivamente. Esta nueva versión reordena las palabras en 3906 filas de 4 *words* ordenadas de derecha a izquierda, para esto lee línea por línea los respectivos archivos de entrada y rellena con ceros las filas que no se completan.

```

1  # correr con Python 2.7
2  # total_words = words_per_line * number_of_lines
3  def singleWordPerLine2newFormat(words_per_line, number_of_lines, name_file_in,
   ↪ name_file_out):
4  fr = open(name_file_in, 'r')
5  fw = open(name_file_out, 'w')
6  new_line = range(words_per_line)
7  for i in range(number_of_lines):
8      for j in range(words_per_line):
9          new_line[words_per_line-1-j] = fr.read(8)
10         if(new_line[words_per_line-1-j]=='' ):
11             new_line[words_per_line-1-j] = '00000000'

```

```
12     fr.read(1)
13     for j in range(words_per_line):
14         fw.write(new_line[j])
15     fw.write('\n')
16 fr.close()
17 fw.close()
18 # 3906 lineas de 4 palabras => total_words = 3906*4 = 15624
19 singleWordPerLine2newFormat(4, 3906, "text_in", "text_in_formatted")
20 singleWordPerLine2newFormat(4, 3906, "data_in", "data_in_formatted")
```

5 Manual de uso del SoC

En este anexo se presentan los pasos a seguir para configurar el SoC en *Vivado* y el procedimiento para compilar y cargar programas en el mismo.

5.1 Configurando el proyecto en *Vivado*

En primera instancia se debe descargar la carpeta “codigo_fuente_soc” del repositorio de *GitHub*. Además, se instala la librería para la tarjeta *FPGA* deseada, en este caso la *Nexys 4*. En el repositorio de *Digilent*, *vivado-boards* (<https://github.com/Digilent/vivado-boards>), se hallan estas librerías junto con las instrucciones para su instalación en *Vivado*.

Luego, se abre *Vivado 2020.2* y haciendo clic en *Create Project*, en el apartado *Quick Start*, se comienza la creación de un nuevo proyecto. Después de oprimir *Next* en la primera ventana, se selecciona el nombre del proyecto y el directorio del mismo, tal como se aprecia en la figura .1.

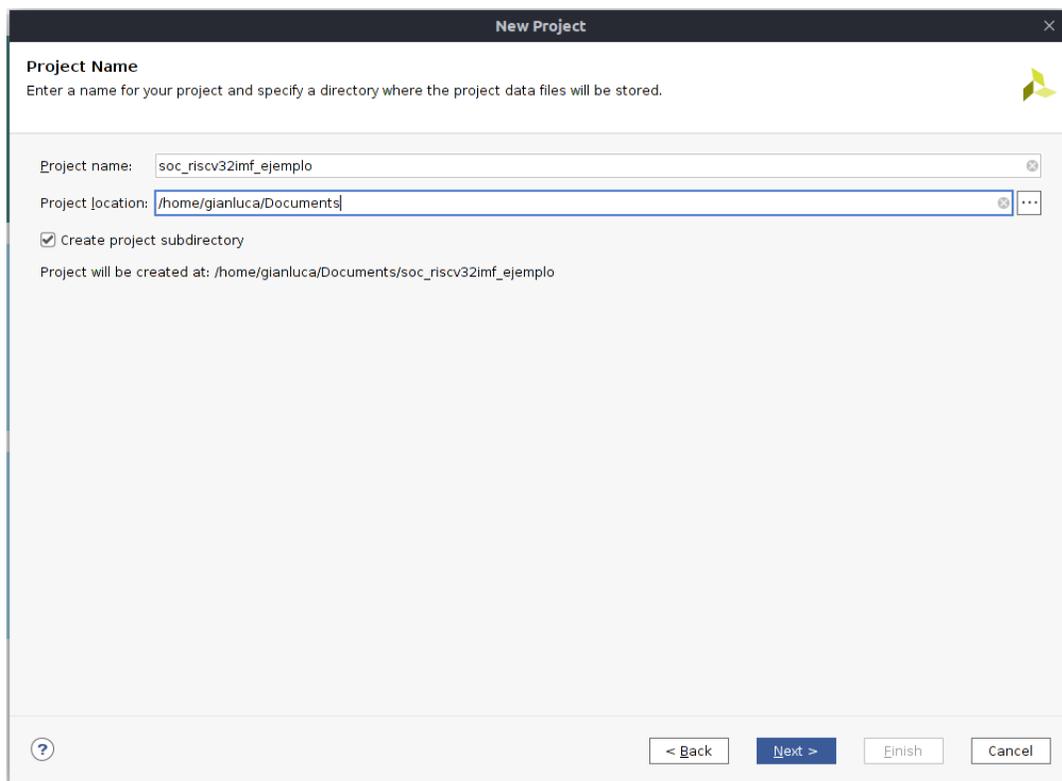


Figura .1: Configurando el nombre y directorio del nuevo proyecto de *Vivado*.

En la figura .2 se aprecia la siguiente ventana, no se modifican las opciones por defecto y se presiona *Next* para continuar con la configuración del proyecto.

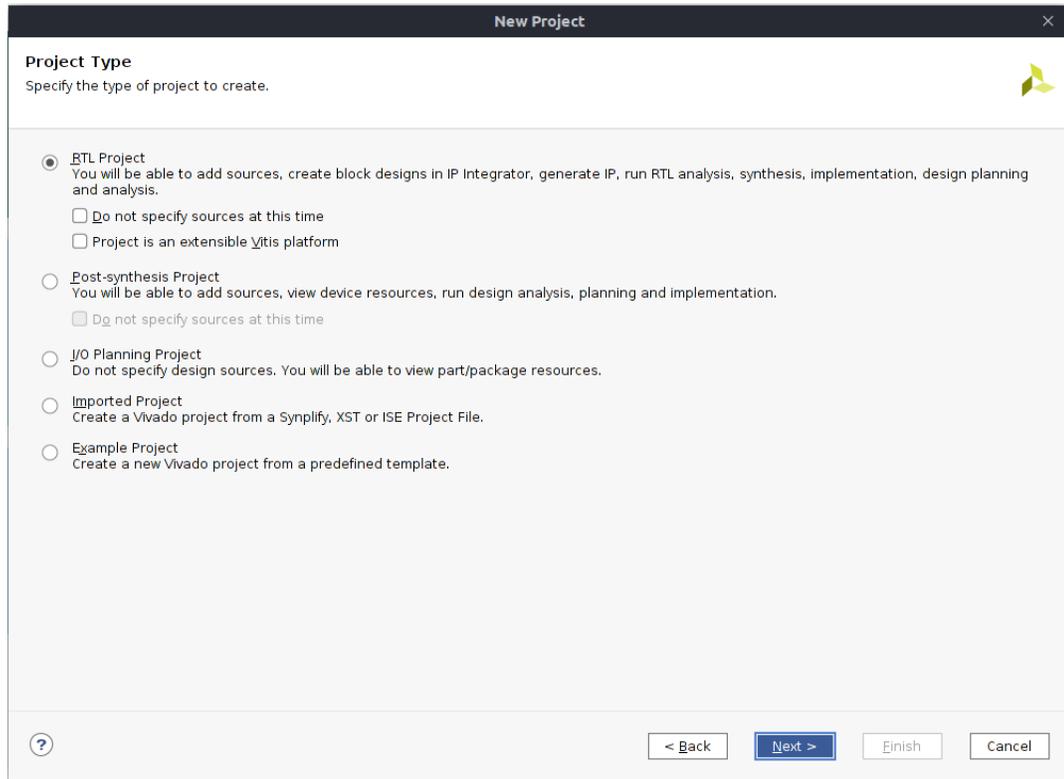


Figura .2: Selección del tipo de proyecto en *Vivado*.

A continuación, se presenta la ventana que permite agregar los archivos fuente del proyecto. Se oprime el botón *Add Files* que se aprecia en la figura .3.

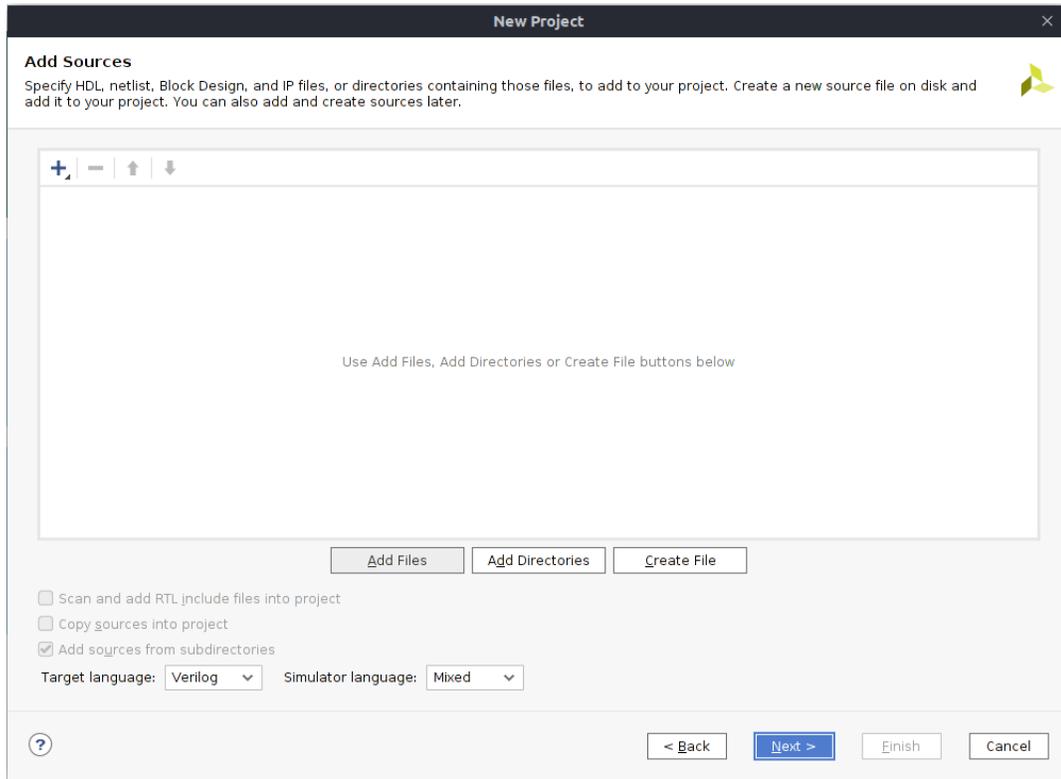


Figura .3: Ventana para agregar el código fuente del proyecto de *Vivado*.

Luego, se busca la carpeta descargada anteriormente, “**codigo_fuente_soc**”, y se añaden todos los archivos (visibles) del mismo al proyecto (ver figura .4).



Figura .4: Selección de los archivos a importar cómo código fuente del proyecto en *Vivado*.

Se vuelve a la ventana anterior pero ahora se puede ver una lista con los nuevos archivos. Es importante activar la opción *Copy sources into project* (ver figura .5).

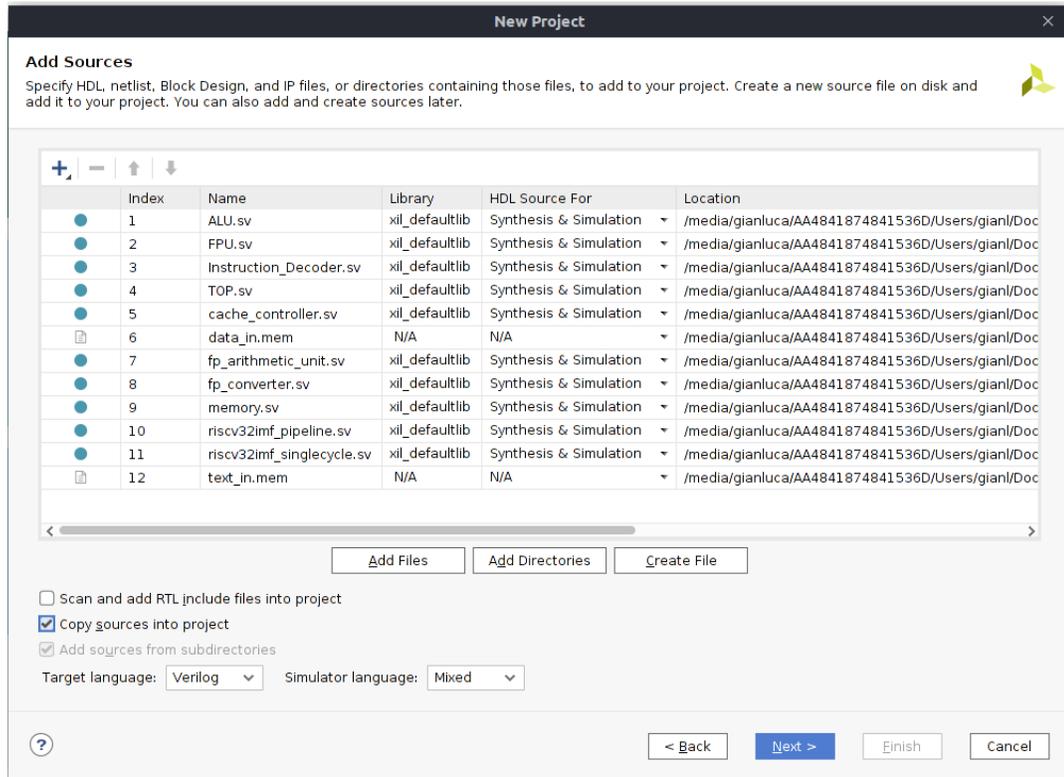


Figura .5: Confirmación de los archivos fuente importados al proyecto de *Vivado*.

Se continua con la creación del proyecto haciendo clic en *Next* y se llega a la ventana de la figura .6.

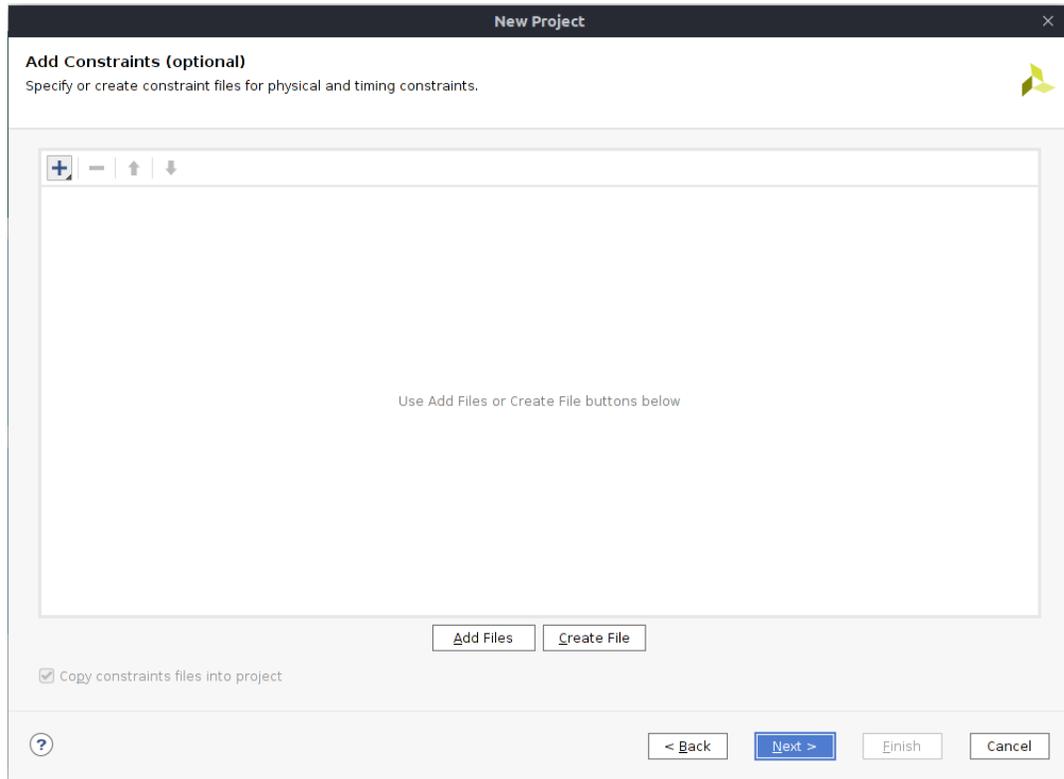


Figura .6: Ventana para agregar las *constraints* del proyecto de Vivado.

Se oprime *Add Files* y se abre la carpeta “**codigo_fuente_soc**”, nuevamente. Después, se añaden las *constraints* del proyecto seleccionando el archivo *top_constraints.xdc* (ver figura .7).

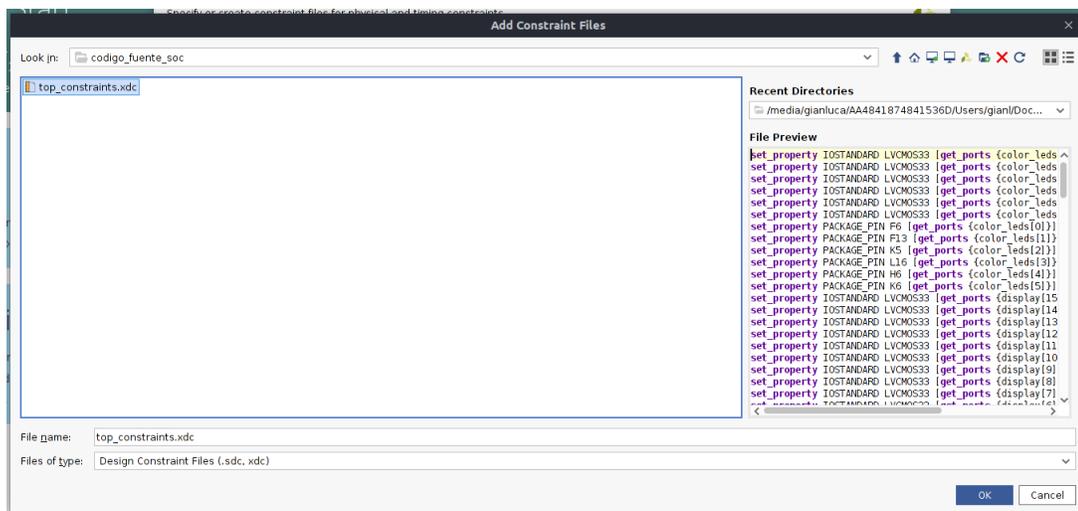


Figura .7: Selección del archivo con las *constraints* a importar al proyecto de Vivado.

Luego de presionar *OK*, se llega a la ventana de la figura .8 donde debe estar activada la opción *Copy constraints files into project*.

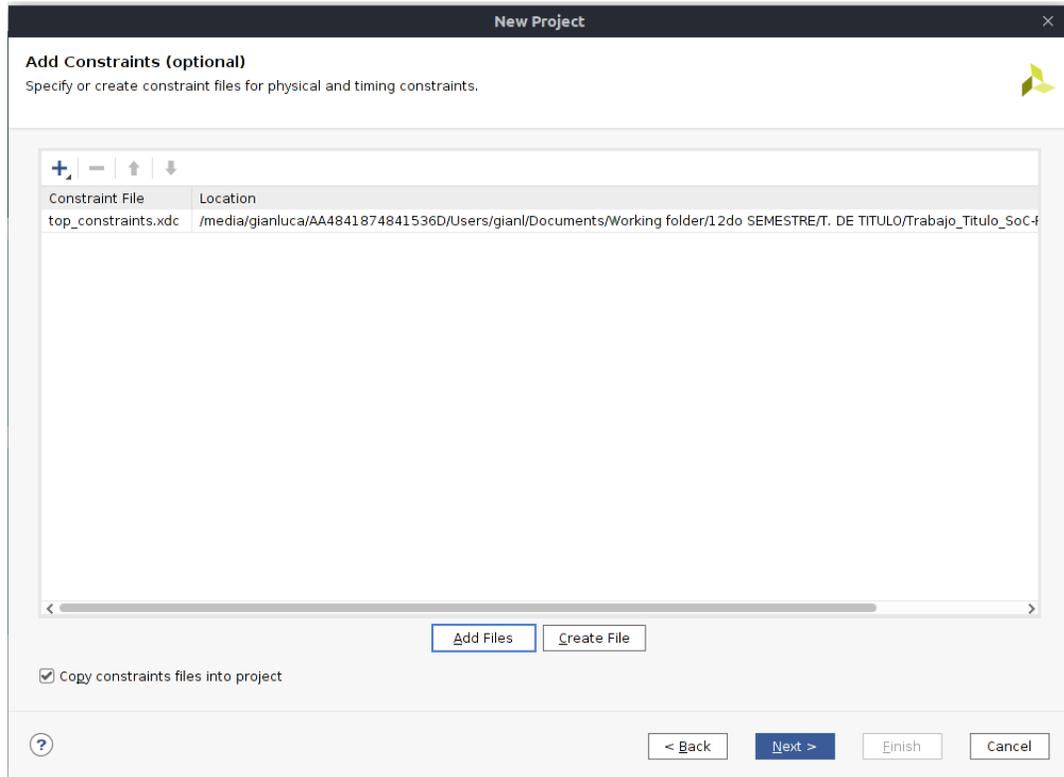


Figura .8: Confirmación de las *constraints* importadas al proyecto de *Vivado*.

Al continuar con la creación del proyecto, después de presionar *Next*, se llega a la selección de la tarjeta *FPGA* objetivo. Tal como se aprecia en la figura .9, se debe ir a la pestaña *Boards* y con la ayuda del buscador se selecciona la tarjeta *Nexys 4* (considerando que las librerías ya están instaladas).

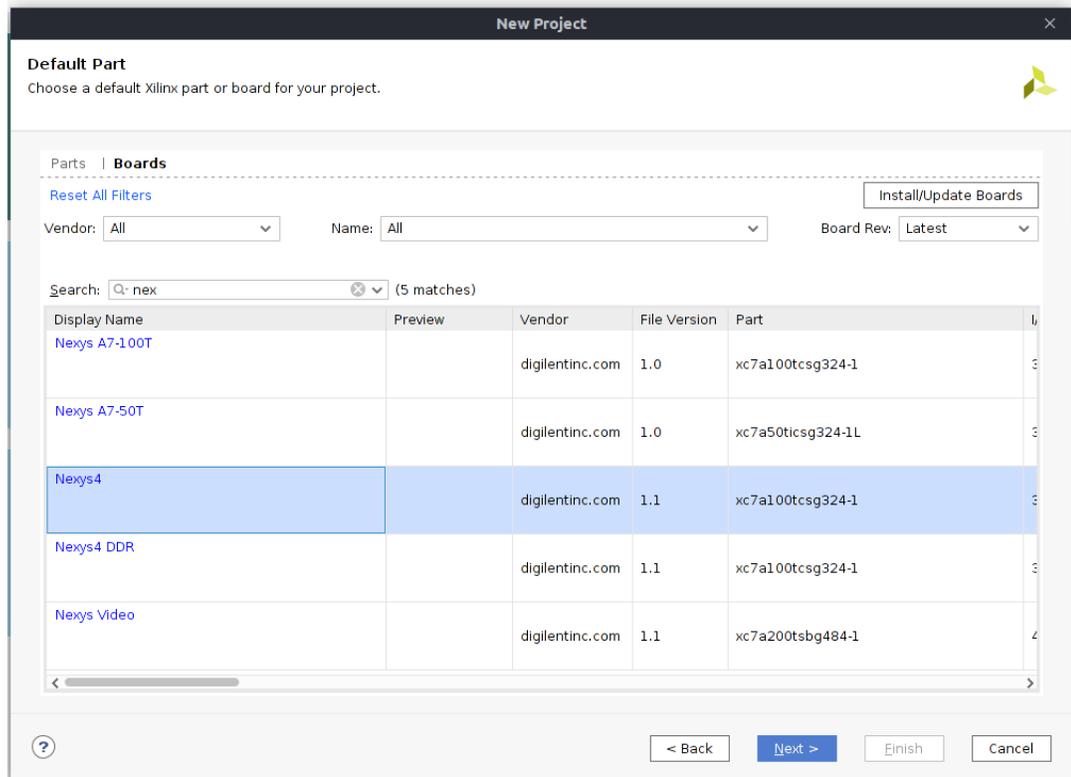


Figura .9: Selección de la tarjeta *FPGA*, *Nexys 4*, para el proyecto de *Vivado*.

Al presionar *Next* se aprecia el cuadro resumen de la configuración inicial del proyecto (ver figura .10). Después de oprimir *Finish* ya se tiene el proyecto creado.

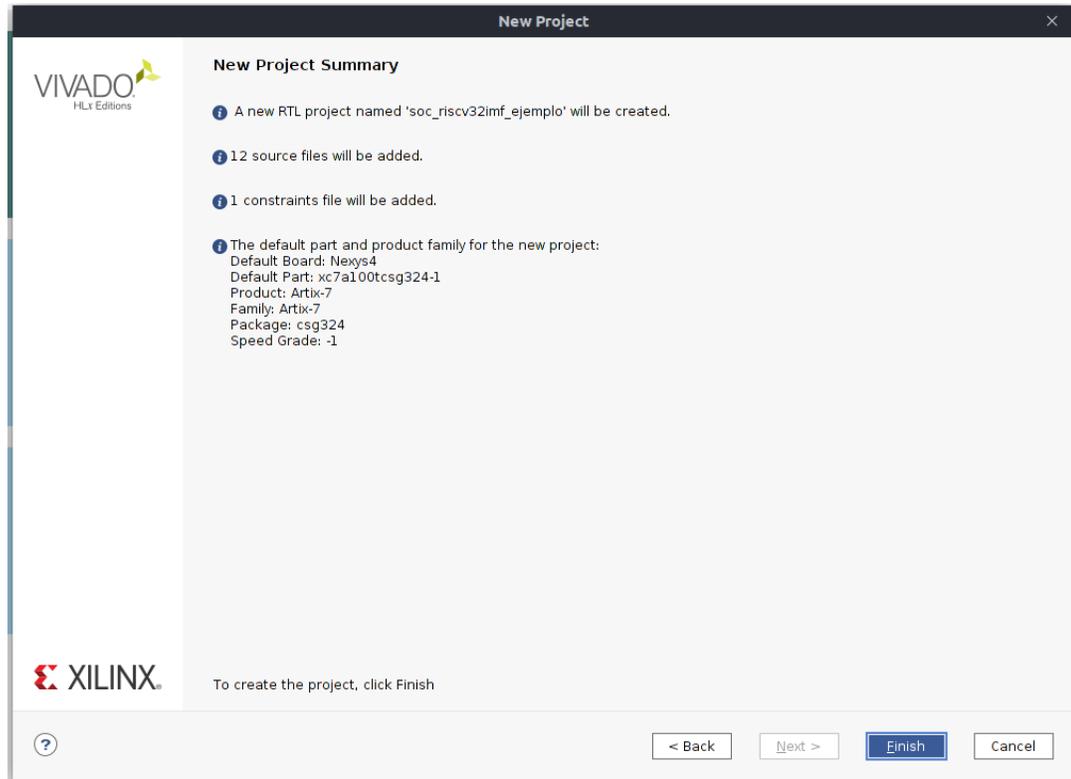


Figura .10: Última ventana, que presenta un resumen de las configuraciones realizadas, de la creación del proyecto en *Vivado*.

Luego, en la consola de *Vivado* se ejecuta el comando `set_param pwropt.maxFaninFanoutToNetRatio 1000` (ver figura .11).

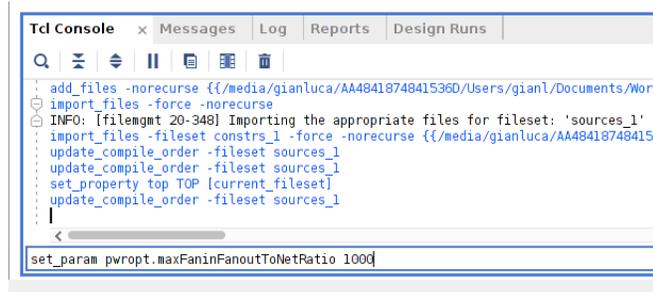


Figura .11: Ejecución del comando `set_param pwropt.maxFaninFanoutToNetRatio 1000` en *Vivado*.

En la figura .12 se muestra una visión general del estado de *Vivado* con el proyecto configurado. Además, es importante definir el módulo *TOP* como el principal (haciendo clic derecho sobre él y oprimiendo *Set as Top*).

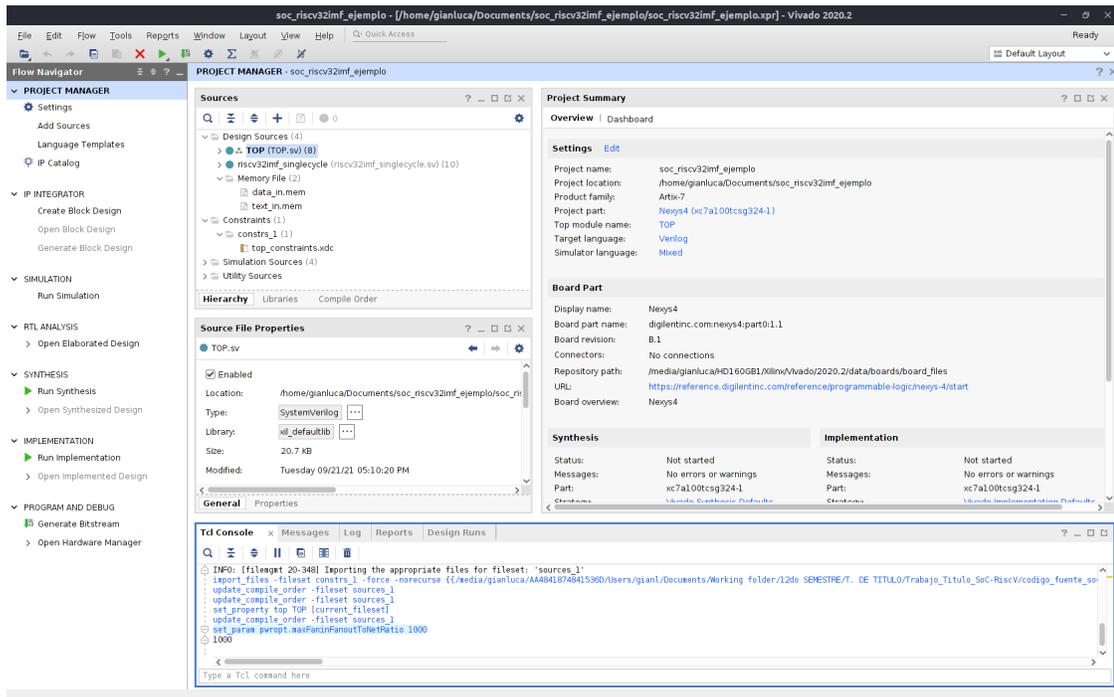


Figura .12: Visión de *Vivado* con el proyecto configurado.

Por otra parte, en la figura .13 se muestra el estado final de las dependencias entre módulos que se debe obtener.

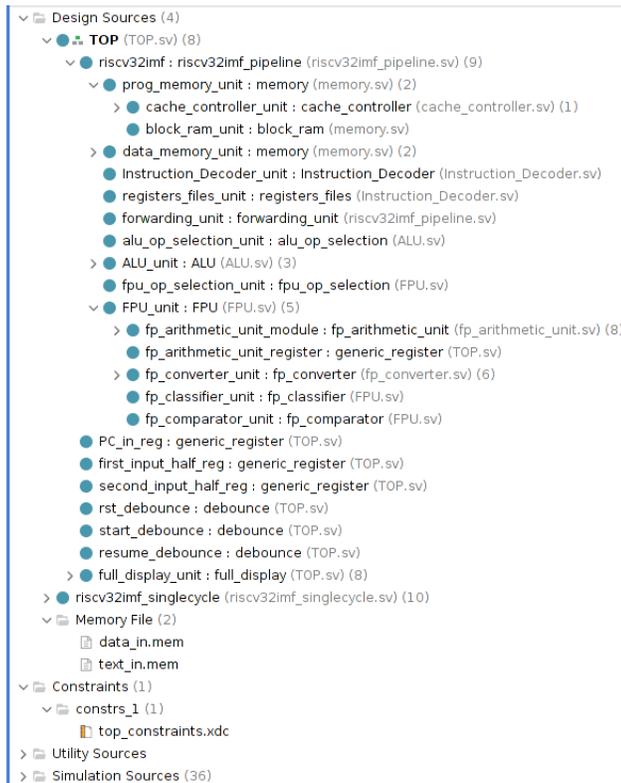


Figura .13: Visión de los módulos del proyecto en *Vivado*.

5.2 Compilación de programas para el SoC

Luego de realizar los pasos de la sección anterior se desea cargar algún programa en el SoC. Para explicar el proceso de compilación de un programa escrito en lenguaje ensamblador se toma como ejemplo a “*fibonacci_inputs.s*” (disponible en el repositorio de [GitHub](#)).

Primero se ejecuta el simulador *RARS 1.5* y se abre el archivo en cuestión (*File* → *Open*). En la figura .14 se aprecia la selección del programa.

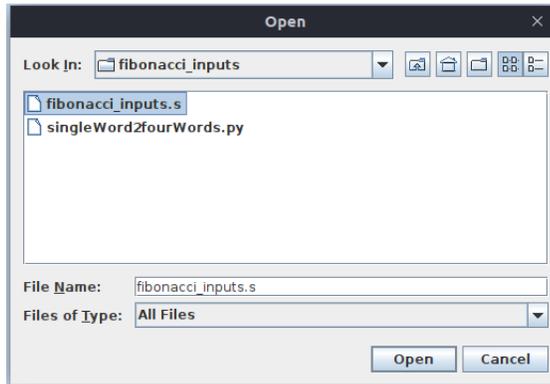


Figura .14: Apertura de “*fibonacci_inputs.s*” en *RARS*.

En la figura .15 se observa una visión de *RARS* con el archivo abierto.

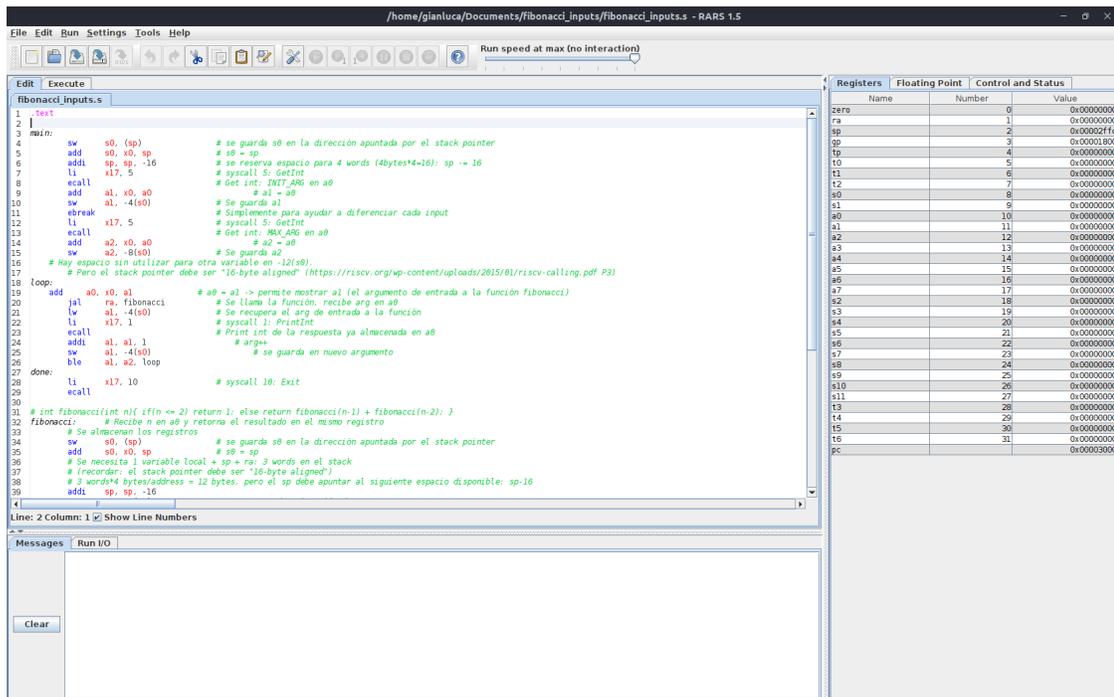


Figura .15: Visión de “*fibonacci_inputs.s*” en *RARS*.

Luego, se debe modificar la configuración de la memoria en el simulador. Para esto se accede a: *Settings* → *Memory Configuration...* y se llega a la ventana de la figura .16. Se debe seleccionar la opción: *Compact, Data at Address 0*, tal como se muestra en dicha figura.

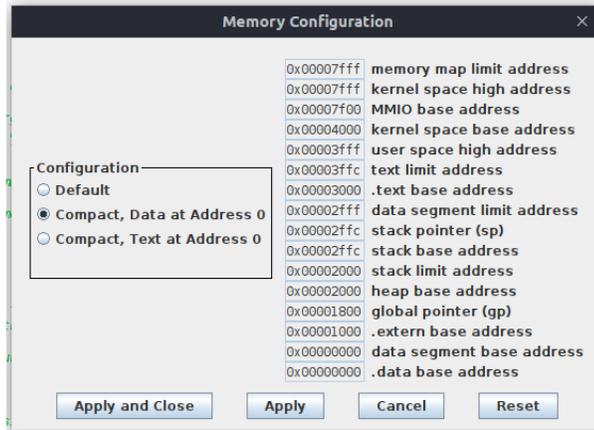


Figura .16: Configuración de memoria en *RARS*.

A continuación, se debe compilar el programa y confirmar que no tenga errores (el programa solo debe contener las instrucciones implementadas, ver tablas 2.6, 2.7 y 2.9). En la figura .17 se muestra el mensaje que arroja *RARS* cuando la compilación es exitosa.

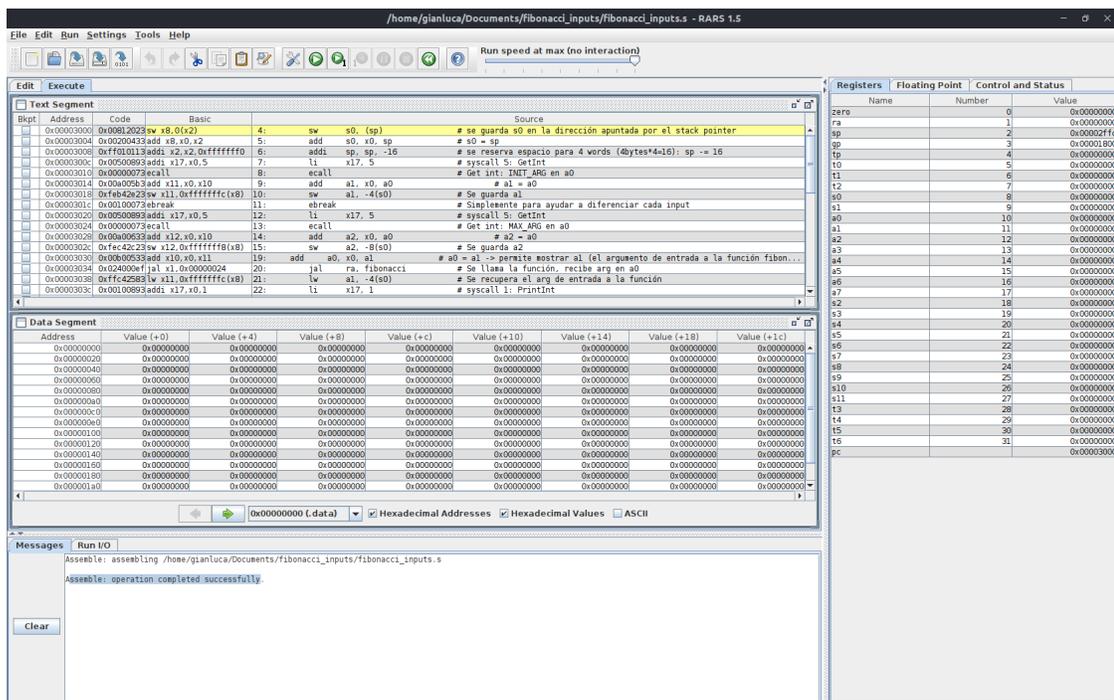


Figura .17: Compilación exitosa de “*fibonacci_inputs.s*” en *RARS*.

Sin ejecutar el programa en el simulador se debe exportar la memoria de programa y la de datos (esta última solo si corresponde). Para esto se accede a: *File* → *Dump Memory...* lo que abre la ventana de la figura .18. En esta ventana se debe seleccionar *.text* en el apartado *Memory Segment* para exportar la memoria de programa y *.data* (cuando corresponde) para la memoria de datos. Luego, en la sección *Dump Format*, se debe seleccionar la opción *Hexadecimal Text* (tanto para la memoria de programa como la de datos).

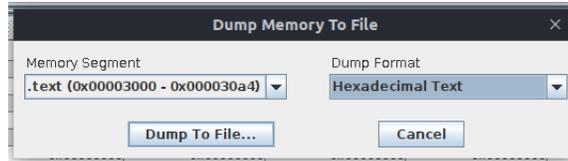


Figura .18: Ventana para exportar la memoria de programa y de datos.

Al oprimir *Dump To File...* se abre la ventana de la figura .19. Se debe guardar la memoria de programa con el nombre *text_in* y la de datos (que en este caso no existe) bajo el nombre de *data_in*. Ambos archivos, *text_in* y *data_in*, deben estar en el mismo directorio que el *script* de *Python 2.7 singleWord2fourWords.py* (disponible en el [GitHub](#)).

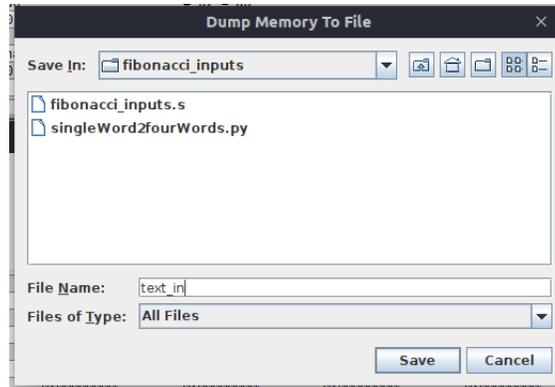


Figura .19: Selección del directorio y del nombre del archivo con la memoria de programa.

Ahora, se debe ejecutar *singleWord2fourWords.py* con el fin de ajustar el formato de las palabras hexadecimales al que requiere el *SoC*. Para esto se crea un archivo vacío con el nombre de *data_in* en el mismo directorio, debido a que no hay memoria de datos pero el *script* lo requiere como entrada. En la figura .20 se aprecia el resultado final en el directorio. El *script* de *Python* crea dos nuevos archivos: *text_in_formatted* y *data_in_formatted*.

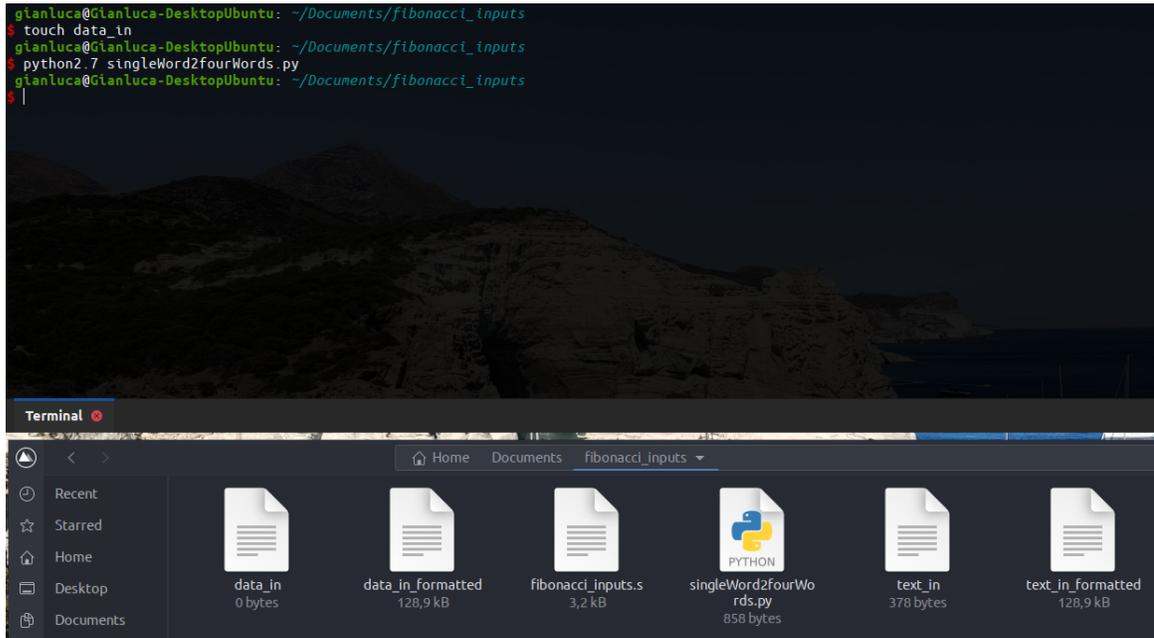


Figura .20: Ejecución de *singleWord2fourWords.py*.

Finalmente, el contenido de los archivos *text_in_formatted* y *data_in_formatted* se debe copiar en los archivos, del proyecto de *Vivado*, *text_in.mem* y *data_in.mem*, respectivamente (ver figura .21). Para esta tarea se recomienda utilizar *Ctrl+A* para seleccionar todo el contenido de un archivo de forma inmediata. Luego, se puede generar el *Bitstream* después del proceso de síntesis e implementación. Cargando dicho *Bitstream* en el *FPGA* se obtiene el *SoC* con el programa compilado y cargado.

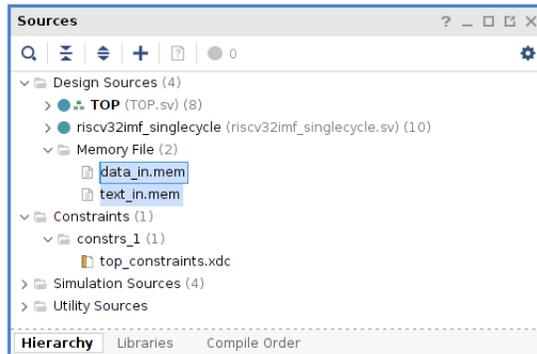


Figura .21: Se destacan los archivos *text_in.mem* y *data_in.mem* que contienen la memoria de programa y de datos, respectivamente, del *SoC*.