

Pipelined SIMD multimedia unit Design with the VHDL/Verilog Hardware description language

Project Report: Part I

Jin Yuan Chen
115702978

November 1, 2025

Abstract

This project focuses on the structural and behavioral design of a four-stage pipelined Multimedia Unit (MMU). The design is implemented using VHDL, a hardware description language, to model the MMU with a reduced subset of multimedia instructions, similar to those in Sony Cell SPU and Intel SSE architectures.

The complete 4-stage pipeline is designed at the register transfer level (RTL) developed in a structural manner with several modules operating simultaneously. Each stage of the pipeline is defined by a module that is developed behaviorally with inter-stage register. Verification of each module will be done individually with their respective self-checking test benches. This will ensure the functional correctness of all stages of the pipeline prior to full system integration of the 4-stage MMU.

The complete top-level MMU model is then instantiated with another test bench to validate the completeness of the four-stage pipeline, where each instruction will cycle through all stages of the pipeline. The resulting outputs will demonstrate the operational behavior and status of each pipeline stage during execution.

1 Introduction

This paper presents **Part I of the Final Report**, focusing exclusively on the architecture of the Multimedia ALU during the **execution stage**. At this phase, no prior knowledge of the complete pipeline design is required for implementing the Multimedia ALU module. The MMU is assumed to received all the correct and necessary input signals from the previous module up to the stage following the forwarding unit.

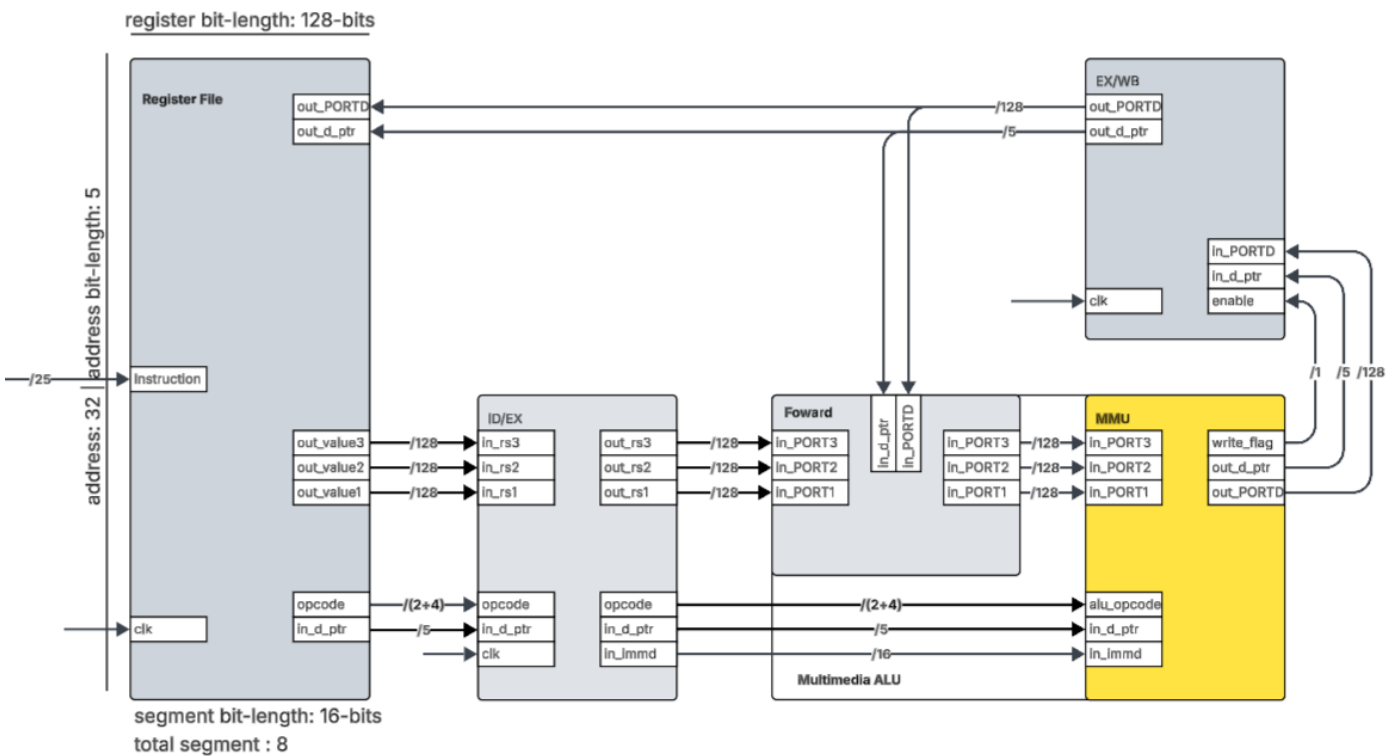


Figure 1: The RTL diagram highlight the execution stage of the Multimedia ALU.

A behavioral model approach is employed to design the test bench for the MMU during execution. The expected results for each test case are predetermined to facilitate the self-checking aspect of the test bench design. A direct comparison between the MMU’s outputs and the expected result ensure accurate functional verification of the instruction sets.

2 Multimedia ALU

The MMU input and output signals has been carefully chosen and simplify for efficient data flow and control. The idea is that the MMU should immediately be able to perform any ALU operation given the available signal. The MMU should not parse or decode any instruction other than the ALU specific opcode.

The entity declaration in VHDL of the Multimedia ALU after the forwarding stage is shown as follow:

```

1  entity MMU_ALU is
2      port (
3          opcode      : in  std_logic_vector(OPCODE_LENGTH-1 downto 0);
4
5          in_PORT3     : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);
6          in_PORT2     : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);
7          in_PORT1     : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);
8
9          in_immed     : in  std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
10         in_d_ptr      : in  std_logic_vector(ADDRESS_LENGTH-1 downto 0);
11
12         out_PORTD     : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
13         out_d_ptr     : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
14         wback_flag    : out std_logic
15     );
16 end entity;

```

in_PORT3; in_PORT2; in_PORT1: Three 128-bits inputs, corresponding to in_PORT3, in_PORT2, and in_PORT1 from the register file. This mean the MMU can read 3 full row of 128-bits from the register file as specify by the design description.

opcode: The 25-bits instruction is parsed and reduce to 6-bits as the MMU opcode. The 6-bits Opcode contain enough information for the MMU to perform the specified operation or function on the given inputs. This is because the reduced subset of MMU instruction is chosen to have 2-bits that define the instruction type and maximum of 4-bits that encodes the operations/function. Instruction type that has less than 4-bits of encode operation, the extra bits are ignored or don't cares. In other word, the first 2 most significant bits of the opcode encodes the instruction type and the last 4 less significant bits of the opcode encodes the instruction operation/function.

Example of MMU 6-bits Opcode with don't cares:

5	4	3	0
10	-000		

R4-Instruction Opcode format with 2-bits instruction type, 3-bits encoding, and 3rd bit is don't care.

in_immed: The 16-bits immediate line is connected for instructions that require MMU operation with an immediate or constant value. Since the reduce instruction set does not utilize a signed immediate field, signed extension for immediate values is not implemented. And all immediate values shorter than 16-bits will flow through the same 16-bits immediate input to the MMU with no signed extension, i.e all bits to the right of the immediate field are zeros.

in_d_ptr: The 5-bits destination register pointer directs the output of the MMU to the specify address in the register file. In this case, the destination register address is always specified in the instruction and never modified in the MMU, the output register address pointer, **out_d_ptr** will be the same value as **in_d_ptr** in this pipeline.

out_PORTD: The single 128-bits output that can write back to the register file. Thus the MMU, by design, can only perform a single write to the register file.

wback_flag: A 1-bit control signal to indicate if the output of the MMU should be write back to the register file.

The Multimedia ALU's architecture body is shown below, utilizing procedures store in package file. For complete version, see Appendix 4.3.

```

1  architecture behavior of MMU_ALU is
2  begin
3      main : process(
4          opcode,
5          in_PORT3, in_PORT2, in_PORT1, in_immed,
6          in_d_ptr
7      )
8      begin
9          out_d_ptr <= in_d_ptr;
10         case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
11             when "00" | "01" =>
12                 LDI_memory(      --ref. procedure_package/load_immediate.vhd
13                     opcode,
14                     in_PORT3,
15                     in_immed,
16
17                     out_PORTD,
18                     wback_flag
19                 );
20
21             when "10" =>
22                 STM_main(      --ref. procedure_package/saturate_math.vhd
23                     opcode,
24                     in_PORT3,
25                     in_PORT2,
26                     in_PORT1,
27
28                     out_PORTD,
29                     wback_flag
30                 );
31
32             when "11" =>
33                 RSI_main(      --ref. procedure_package/rest_instruction.vhd
34                     opcode,
35                     in_PORT2,
36                     in_PORT1,
37                     in_immed,
38
39                     out_PORTD,
40                     wback_flag
41                 );
42
43             when others =>
44                 out_PORTD <= (others => '0');
45                 out_d_ptr <= (others => '0');
46                 wback_flag <= '0';
47         end case;
48     end process;
49 end architecture;

```

The architecture body divides the MMU ALU operation into 3 instruction types LDI, R4, and R3 by calling 3 separate procedures that are packaged in their respective file names: load_immediate, saturated_math, and rest_instruction. For the context of these procedure and the full source code, see in Appendix 4.4, 4.5, 4.6.

3 Instruction Set Description and Verification

LI - Load Immediate

Description:

Load a 16-bit Immediate value from the [20:5] instruction field into the 16-bit field specified by the Load Index field [23:21] of the 128-bit register **rd**. Other fields of register **rd** are not changed. Register **rd** is both a source and destination register in memory.

Instruction:

24	23	21	20	5	4	0
0	index	immediate	rd			

Opcode:

5	4	3	0
0-	index		

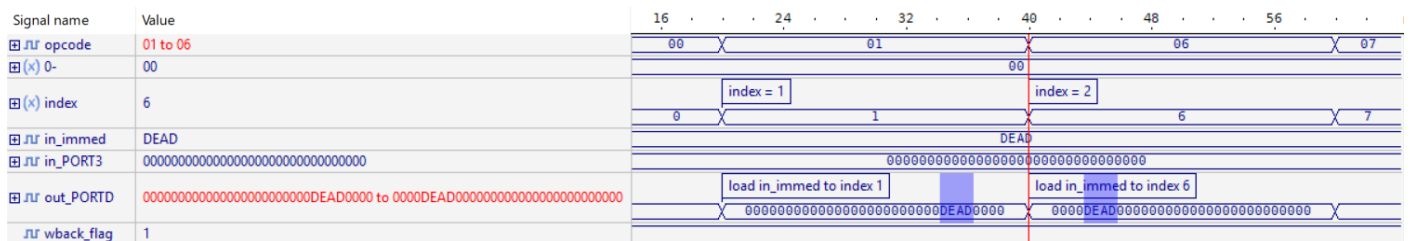
Operation:

$$\begin{aligned} \text{in_PORT3} &\leftarrow \text{FILE}[\text{rd}] \\ \text{in_PORT3}[\text{index}] &\leftarrow \text{immediate} \\ \text{out_portD}[31\text{-fields}] &\leftarrow \text{in_PORT3} \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD} \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

The 128-bit register load from Register File is always send through in_PORT3, as the designated read line for load immediate instruction.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.



The 16-bit in_immed, **in_PORT3**, out_portD are represent in hexadecimal. The in_immed = xDEAD is loaded based on the **index**[0:7] of out_portD. Each index section is allocated 16-bits or 4 hexadecimal value.

SIMAL - Saturated Signed Integer Multiply-Add Low

Description:

Multiply low 16-bit-fields of each 32-bit field of registers rs3 and rs2, then add 32-bit products to 32-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10		0000		rs3		rs2		rs1		rd	

Opcode:

5			4	3							0
10				-000							

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$$

$$\text{ret32} \leftarrow \text{SignExt}(\text{in_PORT3}[16\text{-low}]) \times \text{SignExt}(\text{in_PORT2}[16\text{-low}])$$

$$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[32\text{-field}] + \text{ret32})$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$$

$$\text{wback_flag} \leftarrow 1$$

Operates on 4 separate 16-bit low multiplication and 32-bit field addition in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	2	4	6	8	10	12
in_opcode	20						
in_10	10						
in_function	0						
in_in_immed	DEAD						
in_in_PORT3	000100020003000470057006F0077008						
$\text{in_16-low}[111:96]$	2						
$\text{in_16-low}[79:64]$	4						
$\text{in_16-low}[47:32]$	28678						
$\text{in_16-low}[15:0]$	28680						
in_in_PORT2	000800070006000570047003F0027001						
$\text{in_16-low}[111:96]$	7						
$\text{in_16-low}[79:64]$	5						
$\text{in_16-low}[47:32]$	28675						
$\text{in_16-low}[15:0]$	28673						
in_in_PORT1	00000000000000007FFF00007FFF0000						
$\text{in_32-field}[127:96]$	0						
$\text{in_32-field}[95:64]$	0						
$\text{in_32-field}[63:32]$	2147418112						
$\text{in_32-field}[31:0]$	2147418112						
out_out_PORTD	0000000E000000147FFFFFFFF7FFFFFFFF						
$\text{out_32-field}[127:96]$	14						
$\text{out_32-field}[95:64]$	20						
$\text{out_32-field}[63:32]$	7FFFFFFF						
$\text{out_32-field}[31:0]$	7FFFFFFF						
wback_flag	1						

The 4 separate 16-bit low of both in_PORT3 and in_PORT2 are represented in decimal. The

32-field[127:64] and 32-field[95:64] of in_PORT1 is represented in decimal, while 32-field[63:32] and 32-field[31:0], showing saturated over-flow = 0x7FFFFFFF, is represent in hexadecimal.

SIMAH - Saturated Signed Integer Multiply-Add High

Description:

Multiply high 16-bit-fields of each 32-bit field of registers rs3 and rs2, then add 32-bit products to 32-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10	001	rs3	rs2	rs1	rd						

Opcode:

5	4	3	0
10	-001		

Operation:

in_PORT1 \leftarrow FILE[rs1]

in_PORT2 \leftarrow FILE[rs2]

in_PORT3 \leftarrow FILE[rs3]

ret32 \leftarrow SignExt(in_PORT3[16-high]) \times SignExt(in_PORT2[16-high])

out_portD[31-fields] \leftarrow Saturate₃₂(in_PORT1[32-field] + ret32)

FILE[rd] \leftarrow out_PORTD

wback_flag \leftarrow 1

Operates on 4 separate 16-bit high multiplication and 32-bit field addition in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	20	22	24	26	28	30
NJR opcode	21	20			21		
X 10							
X function	1	0			1		
NJR in_immed	DEAD				DEAD		
NJR in_PORT3	000100020003000470057006F0077008				000100020003000470057006F0077008		
X 16-low[127:112]	1				1		
X 16-low[95:80]	3				3		
X 16-low[63:48]	28677				28677		
X 16-low[31:16]	-4089				-4089		
NJR in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
X 16-low[127:112]	8				8		
X 16-low[95:80]	6				6		
X 16-low[63:48]	28676				28676		
X 16-low[31:16]	-4094				-4094		
NJR in_PORT1	00000000000000007FFF00007FFF0000				00000000000000007FFF00007FFF0000		
X 32-field[127:96]	0				0		
X 32-field[95:64]	0				0		
X 32-field[63:32]	2147418112				2147418112		
X 32-field[31:0]	2147418112				2147418112		
NJR out_PORTD	00000008000000127FFFFFFF7FFFFFFF				00000008000000127FFFFFFF7FFFFFFF		
X 32-field[127:96]	8	14			8		
X 32-field[95:64]	18	20			18		
X 32-field[63:32]	7FFFFFFF				7FFFFFFF		
X 32-field[31:0]	7FFFFFFF				7FFFFFFF		
NJR wback_flag	1						

The 4 separate 16-bit high of both in_PORT3 and in_PORT2 are represented in decimal.

The 32-field[127:64] and 32-field[95:64] of in_PORT1 is represented in decimal, while 32-field[63:32] and 32-field[31:0], showing saturated over-flow = 0x7FFFFFFF, is represent in hexadecimal.

SIMSL - Saturated Signed Integer Multiply-Subtract Low

Description:

Multiply low 16-bit-fields of each 32-bit field of registers rs3 and rs2, then subtract 32-bit products from 32-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10	010	rs3	rs2	rs1	rd						

Opcode:

5	4	3	0
10	-010		

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$

$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$

$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$

$\text{ret32} \leftarrow \text{SignExt}(\text{in_PORT3}[16\text{-low}]) \times \text{SignExt}(\text{in_PORT2}[16\text{-low}])$

$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[32\text{-field}] - \text{ret32})$

$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$

$\text{wback_flag} \leftarrow 1$

Operates on 4 separate 16-bit low multiplication and 32-bit field subtraction in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	40	42	44	46	48	50
in_opcode	22	21			22		
in_10	10	1			2		
in_function	2				DEAD		
in_immed	DEAD				000100020003000470057006F0077008		
in_PORT3	000100020003000470057006F0077008				1		
$\text{in_PORT3}[16\text{-low}[127:112]]$	1				3		
$\text{in_PORT3}[16\text{-low}[95:80]]$	3				28677		
$\text{in_PORT3}[16\text{-low}[63:47]]$	28677				-4089		
$\text{in_PORT3}[16\text{-low}[31:16]]$	-4089				000800070006000570047003F0027001		
in_PORT2	000800070006000570047003F0027001				8		
$\text{in_PORT2}[16\text{-low}[127:112]]$	8				6		
$\text{in_PORT2}[16\text{-low}[95:80]]$	6				28676		
$\text{in_PORT2}[16\text{-low}[63:47]]$	28676				-4094		
$\text{in_PORT2}[16\text{-low}[31:16]]$	-4094				00000000000000008001000080010000		
in_PORT1	00000000000000008001000080010000				0		
$\text{in_PORT1}[32\text{-field}[127:96]]$	0				0		
$\text{in_PORT1}[32\text{-field}[95:64]]$	0				-2147418112		
$\text{in_PORT1}[32\text{-field}[63:32]]$	-2147418112				-2147418112		
$\text{in_PORT1}[32\text{-field}[31:0]]$	-2147418112				FFFFFFFF2FFFFFFEC80000000800000000		
out_PORTD	FFFFFFFF2FFFFFFEC80000000800000000				8		
$\text{out_PORTD}[32\text{-field}[127:96]]$	-14				-20		
$\text{out_PORTD}[32\text{-field}[95:64]]$	-20				80000000		
$\text{out_PORTD}[32\text{-field}[63:32]]$	80000000				80000000		
$\text{out_PORTD}[32\text{-field}[31:0]]$	80000000				1		
wback_flag	1						

The 4 separate 16-bit low of both in_PORT3 and in_PORT2 are represented in decimal. The

32-field[127:64] and 32-field[95:64] of in_PORT1 is represented in decimal, while 32-field[63:32] and 32-field[31:0], showing saturated under-flow = 0x80000000, is represent in hexadecimal.

SIMSH - Saturated Signed Integer Multiply-Subtract High

Description:

Multiply high 16-bit- fields of each 32-bit field of registers rs3 and rs2, then subtract 32-bit products from 32-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10		011		rs3		rs2		rs1		rd	

Opcode:

5			4	3							0
10				-011							

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$

$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$

$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$

$\text{ret32} \leftarrow \text{SignExt}(\text{in_PORT3}[16\text{-high}]) \times \text{SignExt}(\text{in_PORT2}[16\text{-high}])$

$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[32\text{-field}] - \text{ret32})$

$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$

$\text{wback_flag} \leftarrow 1$

Operates on 4 separate 16-bit high multiplication and 32-bit field subtraction in each 128-bit register.

Verification: The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	60	62	64	66	68	70
in_immed	DEAD						
in_PORT3	000100020003000470057006F0077008						
$\text{16-low}[127:112]$	1						
$\text{16-low}[95:80]$	3						
$\text{16-low}[63:48]$	28677						
$\text{16-low}[31:16]$	-4089						
in_PORT2	000800070006000570047003F0027001						
$\text{16-low}[127:112]$	8						
$\text{16-low}[95:80]$	6						
$\text{16-low}[63:48]$	28676						
$\text{16-low}[31:16]$	-4094						
in_PORT1	00000000000000008001000080010000						
$\text{32-field}[127:96]$	0						
$\text{32-field}[95:64]$	0						
$\text{32-field}[63:32]$	-2147418112						
$\text{32-field}[31:0]$	-2147418112						
out_PORTD	FFFFFFFF8FFFFFFFEE8000000080000000						
$\text{32-field}[127:96]$	-8						
$\text{32-field}[95:64]$	-18						
$\text{32-field}[63:32]$	80000000						
$\text{32-field}[31:0]$	80000000						
wback_flag	1						

The 4 separate 16-bit high of both in_PORT3 and in_PORT2 are represented in decimal. The 32-field[127:64] and 32-field[95:64] of in_PORT1 is represented in decimal, while 32-field[63:32] and 32-field[31:0], showing saturated under-flow = 0x80000000, is represent in hexadecimal.

SLMAL - Saturated Signed Long Integer Multiply-Add Low

Description:

Multiply low 32-bit- fields of each 64-bit field of registers rs3 and rs2, then add 64-bit products to 64-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10	100	rs3	rs2	rs1	rd						

Opcode:

5	4	3	0
10	-100		

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$

$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$

$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$

$\text{ret64} \leftarrow \text{SignExt}(\text{in_PORT3}[32\text{-low}]) \times \text{SignExt}(\text{in_PORT2}[32\text{-low}])$

$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[64\text{-field}] + \text{ret64})$

$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$

$\text{wback_flag} \leftarrow 1$

Operates on 2 separate 32-bit low multiplication and 64-bit field addition in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	80	82	84	86	88	90
in opcode	24	23			24		
in 10	10						
in function	4	3			4		
in_inmed	DEAD				DEAD		
in_in_PORT3	000100020003000470057006F0077008				000100020003000470057006F0077008		
in 32-field[95:64]	196612				196612		
in 32-field[31:0]	-267948024				-267948024		
in_in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
in 32-field[95:64]	393221				393221		
in 32-field[31:0]	-268275711				-268275711		
in_in_PORT1	00000000000000007FFF00007FFF0000				00000000000000007FFF00007FFF0000		
in 64-field[127:64]	0				0		
in 64-field[63:0]	9223090564025483264				9223090564025483264		
in out_PORTD	00000012002700147FFFFFFFFFFFFFFFF				00000012002700147FFFFFFFFFFFFFFFF		
in 64-field[127:64]	77311967252				77311967252		
in 64-field[63:0]	7FFFFFFFFFFFFFFF				7FFFFFFFFFFFFFFF		
in wback_flag	1						

The 2 separate 32-bit low of both in_PORT3 and in_PORT2 are represented in decimal. The 64-field[127:64] of in_PORT1 is represented in decimal, while 64-field[63:0], showing saturated over-flow = 0x7FFFFFFFFFFFFFFF, is represent in hexadecimal.

SLMAH - Saturated Signed Long Integer Multiply-Add High

Description:

Multiply high 32-bit- fields of each 64-bit field of registers rs3 and rs2, then add 64-bit products to 64-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10		101		rs3		rs2		rs1		rd	

Opcode:

5		4	3								0
10											-101

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$

$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$

$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$

$\text{ret64} \leftarrow \text{SignExt}(\text{in_PORT3}[32\text{-high}]) \times \text{SignExt}(\text{in_PORT2}[32\text{-high}])$

$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[64\text{-field}] + \text{ret64})$

$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$

$\text{wback_flag} \leftarrow 1$

Operates on 2 separate 32-bit high multiplication and 64-bit field addition in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	40	42	44	46	48	50
nr opcode	22	21			22		
$\text{nr } 10$	10						
nr function	2	1			2		
nr in_immed	DEAD				DEAD		
nr in_PORT3	000100020003000470057006F0077008				000100020003000470057006F0077008		
$\text{nr } 32\text{-field}[127:96]$	65538				65538		
$\text{nr } 32\text{-field}[63:32]$	1879404550				1879404550		
nr in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
$\text{nr } 32\text{-field}[127:96]$	524295				524295		
$\text{nr } 32\text{-field}[63:32]$	1879339011				1879339011		
nr in_PORT1	000000000000000000008001000080010000				000000000000000000008001000080010000		
$\text{nr } 64\text{-field}[127:64]$	0				0		
$\text{nr } 64\text{-field}[63:0]$	-9223090559730515968				-9223090559730515968		
nr out_PORTD	FFFFFFFF2FFFFFFEC80000000800000000				FFFFFFFF2FFFFFFEC80000000800000000		
$\text{nr } 64\text{-field}[127:64]$	-55834574868				-55834574868		
$\text{nr } 64\text{-field}[63:0]$	8000000080000000				8000000080000000		
nr wback_flag	1						

The 2 separate 32-bit high of both in_PORT3 and in_PORT2 are represented in decimal. The 64-field[127:64] of in_PORT1 is represented in decimal, while 64-field[63:0], showing saturated over-flow = 0x7FFFFFFFFFFFFFFFFF, is represent in hexadecimal.

SLMSL - Saturated Signed Long Integer Multiply-Subtract Low

Description:

Multiply low 32-bit fields of each 64-bit field of registers rs3 and rs2, then subtract 64-bit products from 64-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10	110	rs3	rs2	rs1	rd						

Opcode:

5	4	3	0
10	-110		

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$$

$$\text{ret64} \leftarrow \text{SignExt}(\text{in_PORT3}[32\text{-low}]) \times \text{SignExt}(\text{in_PORT2}[32\text{-low}])$$

$$\text{out_portD}[31\text{-fields}] \leftarrow \text{Saturate}_{32}(\text{in_PORT1}[64\text{-field}] - \text{ret64})$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$$

$$\text{wback_flag} \leftarrow 1$$

Operates on 2 separate 32-bit low multiplication and 64-bit field subtraction in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	120	122	124	126	128	130
in opcode	26	25			26		
in 10	6	5			6		
in function	DEAD				DEAD		
in in_PORT3	000100020003000470057006F0077008				000100020003000470057006F0077008		
in 32-field[95:64]	196612				196612		
in 32-field[31:0]	-267948024				-267948024		
in in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
in 32-field[95:64]	393221				393221		
in 32-field[31:0]	-268275711				-268275711		
in in_PORT1	00000000000000008001000080010000				00000000000000008001000080010000		
in 64-field[127:64]	0				0		
in 64-field[63:0]	-9223090559730515968				-9223090559730515968		
out out_PORTD	FFFFFFFFFD8FFEC80000000000000000				FFFFFFFFFD8FFEC80000000000000000		
in 64-field[127:64]	-77311967252				-77311967252		
in 64-field[63:0]	8000000000000000				Signed Saturated		
in wback_flag	1				8000000000000000		

The 2 separate 32-bit low of both in_PORT3 and in_PORT2 are represented in decimal. The 64-field[127:64] of in_PORT1 is represented in decimal, while 64-field[63:0], showing saturated under-flow = 0x8000000000000000, is represent in hexadecimal.

SLMSH - Saturated Signed Long Integer Multiply-Subtract High

Description:

Multiply high 32-bit fields of each 64-bit field of registers rs3 and rs2, then subtract 64-bit products from 64-bit fields of register rs1, and save result in register rd. Signed addition performed with saturated rounding, using the max values when overflowed.

Instruction:

24	23	22	20	19	15	14	10	9	5	4	0
10	111	rs3	rs2	rs1	rd						

Opcode:

5	4	3	0
10	-111		

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$

$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$

$\text{in_PORT3} \leftarrow \text{FILE}[\text{rs3}]$

$\text{ret64} \leftarrow \text{SignExt}(\text{in_PORT3}[32\text{-high}]) \times \text{SignExt}(\text{in_PORT2}[32\text{-high}])$

$\text{out_portD}[64\text{-fields}] \leftarrow \text{Saturate}(\text{in_PORT1}[64\text{-field}] - \text{ret64})$

$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD}$

$\text{wback_flag} \leftarrow 1$

Operates on 2 separate 32-bit high multiplication and 64-bit field subtraction in each 128-bit register.

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	140	142	144	146	148	150
in opcode	27	26			27		
in function	7	6			7		
in_immed	DEAD				DEAD		
in_PORT3	000100020003000470057006F0077008				000100020003000470057006F0077008		
32-field[127:96]	65538				65538		
32-field[63:32]	1879404550				1879404550		
in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
32-field[127:96]	524295				524295		
32-field[63:32]	1879339011				1879339011		
in_PORT1	00000000000000008001000080010000				00000000000000008001000080010000		
64-field[127:64]	0				0		
64-field[63:0]	-9223090559730515968				-9223090559730515968		
out_PORTD	FFFFFFFF7FFE8FFF280000000000000000				FFFFFFFF7FFE8FFF280000000000000000		
64-field[127:64]	-34361245710				-34361245710		
64-field[63:0]	80000000000000000000000000000000				80000000000000000000000000000000		
wback_flag	1						

The 2 separate 32-bit high of both in_PORT3 and in_PORT2 are represented in decimal. The 64-field[127:64] of in_PORT1 is represented in decimal, while 64-field[63:0], showing saturated under-flow = 0x8000000000000000, is represent in hexadecimal.

SHRHI – Shift Right Halfword Immediate

Description:

Performs a packed 16-bit halfword logical right shift on the contents of register **rs1** by the value of the four least significant bits of register **rs2**. Each resulting 16-bit value is placed into the corresponding halfword position of destination register **rd**. Bits shifted out of each halfword are discarded, and bits shifted in are filled with zeros.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00000001		rs2		rs1		rd	

Opcode:

5	4	3	0
11		0001	

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{in_immed} &\leftarrow \text{rs2} \\ \text{out_portD}[16\text{-low}] &\leftarrow \text{Logical}(\text{in_PORT1}[16\text{-low}] \gg \text{in_immed}) \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Operates on 8 separate 16-bit low within each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	· · · 2 · · · 4 · · · 6 · · · 8 · · · 10 · · · 12 ·
rs1 opcode	31	31
11		11
rs2 function	1	1
rs2 in_immed	0004	0004
rs1 in_PORT1	700870077006700570047003F0027001	700870077006700570047003F0027001
rs1 16-low[111:96]	0111000000000111	0111000000000111
rs1 16-low[79:64]	0111000000000101	0111000000000101
rs1 16-low[47:32]	0111000000000011	0111000000000011
rs1 16-low[15:0]	0111000000000001	0111000000000001
rs1 in_PORT2	-----	-----
rs1 out_PORTD	0700070007000700070007000F000700	0700070007000700070007000F000700
rs1 16-low[111:96]	0000011100000000	0000011100000000
rs1 16-low[79:64]	0000011100000000	0000011100000000
rs1 16-low[47:32]	0000011100000000	0000011100000000
rs1 16-low[15:0]	0000011100000000	0000011100000000
rs1 wback_flag	1	

The 4 out of 8 separate 16-bit low of each in_PORT1, in_PORT2 (don't cares), and out_PORTD are represent in binary. The four least significant bits of register **rs2** in the instructions is treated as immediate. The immediate is loaded into the **in_immed** line to the MMU. The example above have **rs2** = b00004 or decimal 4 is stored in the 16-bit immediate as 0x0004. Therefore performing 4 logical right shift on **rs1**.

AU – Add Word Unsigned

Description:

packed 32-bit unsigned addition of the contents of registers rs1 and rs2.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00000010		rs2		rs1		rd	

Opcode:

5	4	3	0
11		0010	

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{out_PORTD}[32\text{-fields}] \leftarrow \text{Unsigned}(\text{in_PORT1}[32\text{-fields}] + \text{in_PORT2}[32\text{-fields}])$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$$

$$\text{wback_flag} \leftarrow 1$$

Operates on 4 separate 32-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	20	22	24	26	28	30
rs1 opcode	32	31			32		
11					11		
function	2	1			2		
in_immed	0004				0004		
in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
32-field[127:96]	1879601159				1879601159		
32-field[95:64]	1879470085				1879470085		
32-field[63:32]	1879339011				1879339011		
32-field[31:0]	4026691585				4026691585		
in_PORT2	80010004800100037FFF00027FFF0001				80010004800100037FFF00027FFF0001		
32-field[127:96]	2147549188	?			2147549188		
32-field[95:64]	2147549187	?			2147549187		
32-field[63:32]	2147418114	?			2147418114		
32-field[31:0]	2147418113	?			2147418113		
out_PORTD	F009700BF0077008F0037005FFFFFFF				F009700BF0077008F0037005FFFFFFF		
32-field[127:96]	4027150347				4027150347		
32-field[95:64]	4027019272				4027019272		
32-field[63:32]	4026757125				4026757125		
32-field[31:0]	FFFFFFFF				FFFFFFFF		
wback_flag	1						

$$\text{out_PORTD}[32\text{-fields}] \leftarrow \text{Unsigned}(\text{in_PORT1}[32\text{-fields}] + \text{in_PORT2}[32\text{-fields}])$$

$$\text{out_PORTD}[127:96] \leftarrow \text{Unsigned}(\text{in_PORT1}[127:96] + \text{in_PORT2}[127:96])$$

$$4027150347 = 1879601159 + 2147549188$$

$$\text{out_PORTD}[31:0] \leftarrow \text{Unsigned}(\text{in_PORT1}[31:0] + \text{in_PORT2}[31:0])$$

$$4294967295 < 4026691585 + 2147418113$$

CNT1H – Count 1s in Halfword

Description:

Count 1s in each packed 16-bit halfword of the contents of register rs1. The results are placed into corresponding slots in register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00000011		rs2		rs1		rd	

Opcode:

5	4	3	0
11		0011	

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{out_PORTD}[16\text{-low}] &\leftarrow \text{Count1s}(\text{in_PORT1}[16\text{-low}]) \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Operates on 8 separate 16-bit low in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	40	40.064	40.128	40.192	40.256
<i>rs1</i> opcode	33	32			33	
11				11		
<i>function</i>	3	2		3		
<i>rs1</i> in_immed	4			4		
<i>rs1</i> in_PORT1	700870077006700570047003F0027001			700870077006700570047003F0027001		
16-low[111:96]	0111000000000111			0111000000000111		
16-low[79:64]	0111000000000101			0111000000000101		
16-low[47:32]	0111000000000111			0111000000000111		
16-low[15:0]	0111000000000001			0111000000000001		
<i>rs1</i> in_PORT2	-----			-----		
<i>rs1</i> out_PORTD	00040006000500050004000500050004			00040006000500050004000500050004		
16-low[111:96]	6			6		
16-low[79:64]	5			5		
16-low[47:32]	5			5		
16-low[15:0]	4	-1		4		
<i>rs1</i> wback_flag	1					

AHS – Add Halfword Saturated

Description:

Packed 16-bit halfword signed addition with saturation of the contents of registers rs1 and rs2.

Instruction:

24	23	22	25	14	10	9	5	4	0
11	00000100	rs2	rs1	rd					

Opcode:

5	4	3	0
11	0100		

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{out_PORTD}[16\text{-low}] \leftarrow \text{SignedSaturate}(\text{in_PORT2}[16\text{-low}] + \text{in_PORT1}[16\text{-low}])$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$$

$$\text{wback_flag} \leftarrow 1$$

Operates on 8 separate 16-bit low in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	60	62	64	66	68	70
JUI opcode	34	33			34		
11					11		
JUI function	4	3			4		
JUI in_immed	0004				0004		
JUI in_PORT1	700870077006800570047003F0027001				700870077006800570047003F0027001		
$\text{JUI 16-low}[111:96]$	28679				28679		
$\text{JUI 16-low}[79:64]$	-32763				-32763		
$\text{JUI 16-low}[47:32]$	28675				28675		
$\text{JUI 16-low}[15:0]$	28673				28673		
JUI in_PORT2	80010004800100037FFF7FF27FFFFFFF1				80010004800100037FFF7FF27FFFFFFF1		
$\text{JUI 16-low}[111:96]$	4	?			4		
$\text{JUI 16-low}[79:64]$	3	?			3		
$\text{JUI 16-low}[47:32]$	32754	?			32754		
$\text{JUI 16-low}[15:0]$	-15	?			-15		
JUI out_PORTD	F009700BF00780087FFF7FFF70016FF2				F009700BF00780087FFF7FFF70016FF2		
$\text{JUI 16-low}[111:96]$	28683	Positive Addition			28683		
$\text{JUI 16-low}[79:64]$	-32760	Negative Addition			-32760		
$\text{JUI 16-low}[47:32]$	7FFF	Signed Saturated			7FFF		
$\text{JUI 16-low}[15:0]$	28658				28658		
JUI wback_flag	1						

$$\text{out_PORTD}[16\text{-low}] \leftarrow \text{Unsigned}(\text{in_PORT1}[16\text{-low}] + \text{in_PORT2}[16\text{-low}])$$

$$\text{out_PORTD}[111:96] \leftarrow \text{SignedSaturate}(\text{in_PORT1}[111:96] + \text{in_PORT2}[111:96])$$

$$28683 = 28679 + 4$$

$$\text{out_PORTD}[31:0] \leftarrow \text{Unsigned}(\text{in_PORT1}[47:32] + \text{in_PORT2}[47:32])$$

$$32767 < 28675 + 32754$$

OR – Bitwise Logical OR

Description:

Logical OR the contents of registers rs1 and rs2.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00000101		rs2		rs1		rd	

Opcode:

5	4	3	0
11		0101	

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{in_PORT2} &\leftarrow \text{FILE}[\text{rs2}] \\ \text{out_PORTD} &\leftarrow \text{in_PORT2} \vee \text{in_PORT1} \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	80	82	84	86	88	90
rs1 opcode	35	34			35		
11					11		
(x) function	5	4			5		
rs1 in_immed	0004				0004		
rs1 in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
rs1 in_PORT2	0000000000000000FFFFFFFFFFFFFFFF				0000000000000000FFFFFFFFFFFFFFFF		
rs1 out_PORTD	7008700770067005FFFFFFFFFFFFFFFF				7008700770067005FFFFFFFFFFFFFFFF		
rs1 wback_flag	1						

BCW – Broadcast Word

Description:

Broadcast the leftmost 32-bit word of register rs1 to each of the four 32-bit words of register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11	00000110	rs2	rs1	rd					

Opcode:

5	4	3	0
11	0110		

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{out_PORTD}[32\text{-fields}] &\leftarrow \text{in_PORT1}[127:96] \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Operates on 4 separate 32-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	100	102	104	106	108	110
<input checked="" type="checkbox"/> <i>rs</i> opcode	36	35	X		36		
11					11		
<input checked="" type="checkbox"/> <i>function</i>	6	5	X		6		
<input checked="" type="checkbox"/> <i>rs</i> in_immed	0004				0004		
<input checked="" type="checkbox"/> <i>rs</i> in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
<input checked="" type="checkbox"/> 32-field[127:96]	1879601159				1879601159		
<input checked="" type="checkbox"/> <i>rs</i> in_PORT2	-----	X			-----		
<input checked="" type="checkbox"/> <i>rs</i> out_PORTD	70087007700870077008700770087007	X			70087007700870077008700770087007		
<input checked="" type="checkbox"/> 32-field[127:96]	1879601159				1879601159		
<input checked="" type="checkbox"/> 32-field[95:64]	1879601159	X			1879601159		
<input checked="" type="checkbox"/> 32-field[63:32]	1879601159	-1	X		1879601159		
<input checked="" type="checkbox"/> 32-field[31:0]	1879601159	-1	X		1879601159		
<i>rs</i> wback_flag	1						

MAXWS – Max Signed Word

Description:

For each of the four 32-bit word slots, place the maximum signed value between rs1 and rs2 in register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11	00000111	rs2	rs1	rd					

Opcode:

5	4	3	0
11	0111		

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{out_portD}[31\text{-fields}] \leftarrow \text{SignedMin}(\text{in_PORT2}[32\text{-fields}], \text{in_PORT1}[32\text{-fields}])$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$$

$$\text{wback_flag} \leftarrow 1$$

Operates on 4 separate 32-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	120	122	124	126	128	130
rsr opcode	37	36			37		
11					11		
function	7	6			7		
rsr in_immed	0004				0004		
rsr in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
32-field[127:96]	1879601159				1879601159		
32-field[95:64]	1879470085				1879470085		
32-field[63:32]	1879339011				1879339011		
32-field[31:0]	-268275711				-268275711		
rsr in_PORT2	80010004800100037FFF00027FFF0001				80010004800100037FFF00027FFF0001		
32-field[127:96]	-2147418108				-2147418108		
32-field[95:64]	-2147418109				-2147418109		
32-field[63:32]	2147418114				2147418114		
32-field[31:0]	2147418113				2147418113		
rsr out_PORTD	70087007700670057FFF00027FFF0001				70087007700670057FFF00027FFF0001		
32-field[127:96]	1879601159				1879601159		
32-field[95:64]	1879470085				1879470085		
32-field[63:32]	2147418114				2147418114		
32-field[31:0]	2147418113				2147418113		
rsr wback_flag	1						

MINWS – Min Signed Word

Description:

For each of the four 32-bit word slots, place the minimum signed value between rs1 and rs2 in register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11	00001000	rs2	rs1	rd					

Opcode:

5	4	3	0
11	1000		

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{in_PORT2} &\leftarrow \text{FILE}[\text{rs2}] \\ \text{out_PORTD}[31\text{-fields}] &\leftarrow \text{SignedMin}(\text{in_PORT2}[32\text{-fields}], \text{in_PORT1}[32\text{-fields}]) \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Operates on 4 separate 32-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	140	142	144	146	148	150
inr opcode	38	37			38		
11					11		
inr function	8	7			8		
inr in_immed	0004				0004		
inr in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
32-field[127:96]	1879601159				1879601159		
32-field[95:64]	1879470085				1879470085		
32-field[63:32]	1879339011				1879339011		
32-field[31:0]	-268275711				-268275711		
inr in_PORT2	80010004800100037FFF00027FFF0001				80010004800100037FFF00027FFF0001		
32-field[127:96]	-2147418108				-2147418108		
32-field[95:64]	-2147418109				-2147418109		
32-field[63:32]	2147418114				2147418114		
32-field[31:0]	2147418113				2147418113		
inr out_PORTD	800100048001000370047003F0027001				800100048001000370047003F0027001		
32-field[127:96]	-2147418108				-2147418108		
32-field[95:64]	-2147418109				-2147418109		
32-field[63:32]	1879339011				1879339011		
32-field[31:0]	-268275711				-268275711		
inr wback_flag	1						

MLHU – Multiple Low Unsigned

Description:

The 16 rightmost bits of each of the four 32-bit slots in register rs1 are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register rs2, treating both operands as unsigned. The four 32-bit products are placed into the corresponding slots of register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001001		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1001	

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{out_portD}[31\text{-fields}] \leftarrow \text{UnsignedMult}(\text{in_PORT1}[16\text{-fields}], \text{in_PORT2}[16\text{-fields}])$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$$

$$\text{wback_flag} \leftarrow 1$$

Operates 8 separate 16-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	160	162	164	166	168	170
rs1 opcode	39	38			39		
11					11		
function	9	8			9		
rs1 in_immed	0004				0004		
rs1 in_PORT1	700870077006700570047003F0027001				700870077006700570047003F0027001		
16-low[111:96]	28679				28679		
16-low[79:64]	28677				28677		
16-low[47:32]	28675				28675		
16-low[15:0]	28673				28673		
rs1 in_PORT2	80010004800100037FFF7FF27FFFFFFF1				80010004800100037FFF7FF27FFFFFFF1		
16-low[111:96]	4				4		
16-low[79:64]	3				3		
16-low[47:32]	32754	2			32754		
16-low[15:0]	65521	1	unsigned		65521		
rs1 out_PORTD	0001C01C0001500F37FB5FD66FFA6FF1				0001C01C0001500F37FB5FD66FFA6FF1		
32-field[127:96]	114716				114716		
32-field[95:64]	86031				86031		
32-field[63:32]	939220950				939220950		
32-field[31:0]	1878683633		unsigned product		1878683633		
rs1 wback_flag	1						

MLHU – Multiple Low by constant Unsigned

Description:

The 16 rightmost bits of each of the four 32-bit slots in register rs1 are multiplied by a 5-bit value in the rs2 field of the instruction, treating both operands as unsigned. The four 32-bit products are placed into the corresponding slots of register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001010		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1010	

Operation:

$$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in_immed} \leftarrow \text{ZeroExtend}(\text{rs2})$$

$$\text{out_portD}[31\text{-fields}] \leftarrow \text{UnsignedMult}(\text{in_PORT1}[16\text{-fields}], \text{in_immed})$$

$$\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$$

$$\text{wback_flag} \leftarrow 1$$

Operates 4 separate 16-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	180	182	184	186	188	190
inr opcode	3A	39			3A		
11					11		
function	A	9			A		
inr in_immed	20	4			20		
inr in_PORT1	80010004800100037FFF00027FFF0001				80010004800100037FFF00027FFF0001		
16-low[111:96]	4				4		
16-low[79:64]	3				3		
16-low[47:32]	2				2		
16-low[15:0]	1				1		
inr in_PORT2		
inr out_PORTD	000000500000003C0000002800000014				000000500000003C0000002800000014		
32-field[127:96]	80				80		
32-field[95:64]	60				60		
32-field[63:32]	40				40		
32-field[31:0]	20				20		
inr wback_flag	1						

AND – Bitwise Logical And

Description:

Logical AND the contents of registers rs1 and rs2.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001011		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1011	

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{in_PORT2} &\leftarrow \text{FILE}[\text{rs2}] \\ \text{out_portD}[31\text{-fields}] &\leftarrow \text{in_PORT2} \wedge \text{in_PORT1} \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	200	200.064	200.128	200.192	200.256
<i>rtl</i> opcode	3B	3A		3B		
11				11		
<i>rtl</i> function	B	A		B		
<i>rtl</i> in_immed	20			20		
<i>rtl</i> in_PORT1	80010004800100037FFF00027FFF0001			80010004800100037FFF00027FFF0001		
<i>rtl</i> in_PORT2	0000000000000000FFFFFFFFFFFFFFFF			0000000000000000FFFFFFFFFFFFFFFF		
<i>rtl</i> out_PORTD	00000000000000007FFF00027FFF0001			00000000000000007FFF00027FFF0001		
<i>rtl</i> wback_flag	1					

CLZW – Count Leading Zeros in Words

Description:

For each of the four 32-bit word slots in register rs1, count the number of zero bits to the left of the first “1”. If the word slot in register rs1 is zero, the result is 32. The four results are placed into the corresponding 32-bit word slots in register rd.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001100		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1100	

Operation:

$$\begin{aligned} \text{in_PORT1} &\leftarrow \text{FILE}[\text{rs1}] \\ \text{out_portD}[31\text{-fields}] &\leftarrow \text{CLZ}_{32}(\text{in_PORT1}[32\text{-field}]) \\ \text{FILE}[\text{rd}] &\leftarrow \text{out_PORTD}, \\ \text{wback_flag} &\leftarrow 1 \end{aligned}$$

Operates 4 separate 32-bit field in each 128-bit register)

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	
in opcode	3C	3B X 3C
11		11
function	C	8 X C
in_immed	20	20
in_PORT1	070870071006700570047003F0027001	070870071006700570047003F0027001
32-field[127:96]	00000111000010000111000000000111	00000111000010000111000000000111
32-field[95:64]	0001000000001100111000000000101	0001000000001100111000000000101
32-field[63:32]	0111000000001000111000000000011	0111000000001000111000000000011
32-field[31:0]	111100000000100111000000000001	111100000000100111000000000001
in_PORT2	-----	-----
out_PORTD	00000005000000030000000100000000	00000005000000030000000100000000
32-field[127:96]	5	0 X 5
32-field[95:64]	3	0 X 3
32-field[63:32]	1	X 1
32-field[31:0]	0	X 0
wback_flag	1	

ROTW – Rotate Bits in Word

Description:

The contents of each 32-bit field in register rs1 are rotated to the right according to the value of the 5 least significant bits of the corresponding 32-bit field in register rs2. The results are placed in register rd. Bits rotated out of the right end of each word are rotated in on the left end of the same 32-bit word field.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001101		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1101	

Operation:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
int_var ← Unsigned(in_PORT2[5-low]) mod 32
out_portD[31-fields] ← RotateRight(in_PORT1[32-field, int_var)
FILE[rd] ← out_PORTD,
wback_flag ← 1

```

Operates 4 separate 5-bit low to the corresponding 32-bit word fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	240	240.04	240.08	240.12	240.16	240.2	240.24	240.28
rs1 opcode	3D	3C				3D			
11						11			
function	D	C				D			
in_immed	20					20			
in_PORT1	80010004800100037FFF00027FFF0001					80010004800100037FFF00027FFF0001			
32-field[127:96]	10000000000000010000000000000100	Binary				10000000000000010000000000000100			
32-field[95:64]	10000000000000010000000000000011					10000000000000011000000000000011			
32-field[63:32]	01111111111111110000000000000010					01111111111111110000000000000010			
32-field[31:0]	01111111111111110000000000000001					01111111111111110000000000000001			
in_PORT2	FFFFFF10FFFFFF000000001000000000					FFFFFF10FFFFFF000000001000000000			
5-low[100:96]	16	Decimal				16			
5-low[68:64]	16					16			
5-low[36:32]	1					1			
5-low[4:0]	0					0			
out_PORTD	00048001000380013FFF80017FFF0001					00048001000380013FFF80017FFF0001			
32-field[127:96]	00000000000001001000000000000001	Binary				00000000000001001000000000000001			
32-field[95:64]	00000000000000011000000000000001					00000000000000011000000000000001			
32-field[63:32]	00111111111111110000000000000001					00111111111111110000000000000001			
32-field[31:0]	01111111111111110000000000000001					01111111111111110000000000000001			
wback_flag	1								

SFWU – Subtract from Word Unsigned

Description:

Packed 32-bit word unsigned subtract of the contents of rs1 from rs2 ($rd = rs2 - rs1$).

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001110		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1110	

Operation:

$in_PORT1 \leftarrow FILE[rs1]$
 $in_PORT2 \leftarrow FILE[rs2]$
 $out_portD[31:fields] \leftarrow \text{Unsigned}(in_PORT2[32:fields] - in_PORT1[32:fields])$
 $FILE[rd] \leftarrow out_PORTD,$
 $wback_flag \leftarrow 1$

Operates 4 separate 32-bit fields in each 128-bit register

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value		
<input checked="" type="checkbox"/> <i>in</i> opcode	3E	3D	3E
11			11
<input checked="" type="checkbox"/> <i>function</i>	E	D	E
<i>in</i> in_immed	20		20
<i>in</i> in_PORT1	700870077006700570047003F0027001		700870077006700570047003F0027001
<input checked="" type="checkbox"/> 32-field[127:96]	1879601159		1879601159
<input checked="" type="checkbox"/> 32-field[95:64]	1879470085		1879470085
<input checked="" type="checkbox"/> 32-field[63:32]	1879339011		1879339011
<input checked="" type="checkbox"/> 32-field[31:0]	4026691585		4026691585
<i>in</i> in_PORT2	80010004800100037FFF00027FFF0001		80010004800100037FFF00027FFF0001
<input checked="" type="checkbox"/> 32-field[127:96]	2147549188		2147549188
<input checked="" type="checkbox"/> 32-field[95:64]	2147549187		2147549187
<input checked="" type="checkbox"/> 32-field[63:32]	2147418114	1	2147418114
<input checked="" type="checkbox"/> 32-field[31:0]	2147418113	0	2147418113
<i>out</i> out_PORTD	0FF88FFD0FFA8FFE0FFA8FFF00000000		0FF88FFD0FFA8FFE0FFA8FFF00000000
<input checked="" type="checkbox"/> 32-field[127:96]	267948029		267948029
<input checked="" type="checkbox"/> 32-field[95:64]	268079102		268079102
<input checked="" type="checkbox"/> 32-field[63:32]	268079103		268079103
<input checked="" type="checkbox"/> 32-field[31:0]	0	unsigned saturated	0
<i>wback</i> wback_flag	1		

SFHS – Subtract from Halfword Saturated

Description:

Packed 16-bit halfword signed subtraction with saturation of the contents of rs1 from rs2 (rd = rs2 - rs1).

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00001111		rs2		rs1		rd	

Opcode:

5	4	3	0
11		1111	

Operation:

$\text{in_PORT1} \leftarrow \text{FILE}[\text{rs1}]$
 $\text{in_PORT2} \leftarrow \text{FILE}[\text{rs2}]$
 $\text{out_portD}[31\text{-fields}] \leftarrow \text{SignedSaturate}(\text{in_PORT2}[16\text{-fields}] - \text{in_PORT1}[16\text{-fields}])$
 $\text{FILE}[\text{rd}] \leftarrow \text{out_PORTD},$
 $\text{wback_flag} \leftarrow 1$

operates 8 separate 16-bit fields in each 128-bit register)

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	280	284	288	292	296
11	3F	3E			3F	
function	F	E			F	
in_immed	20				20	
in_PORT1	70080007780600037004FFF3F0027001				70080007780600037004FFF3F0027001	
16-low[111:96]	7				7	
16-low[79:64]	3				3	
16-low[47:32]	-13				-13	
16-low[15:0]	28673				28673	
in_PORT2	80010004800100057FFF00067FFF8F01				80010004800100057FFF00067FFF8F01	
16-low[111:96]	4				4	
16-low[79:64]	5	3			5	
16-low[47:32]	6	2			6	
16-low[15:0]	-28927	1			-28927	
out_PORTD	8000FFFD800000020FFB00137FFF8000				8000FFFD800000020FFB00137FFF8000	
16-low[111:96]	-3				-3	
16-low[79:64]	2				2	
16-low[47:32]	19				19	
16-low[15:0]	-32768				-32768	
wback_flag	1					

NOP – No Operation

Description:

Instruction does not write anything to the register file.

Instruction:

24	23	22	25	14	10	9	5	4	0
11		00000000		rs2		rs1		rd	

Opcode:

5	4	3	0
11		0000	

Operation:

$wback_flag \leftarrow 0$

No operation performed. The value from the previous instruction is held in out_PORTD, producing no new output or register update.

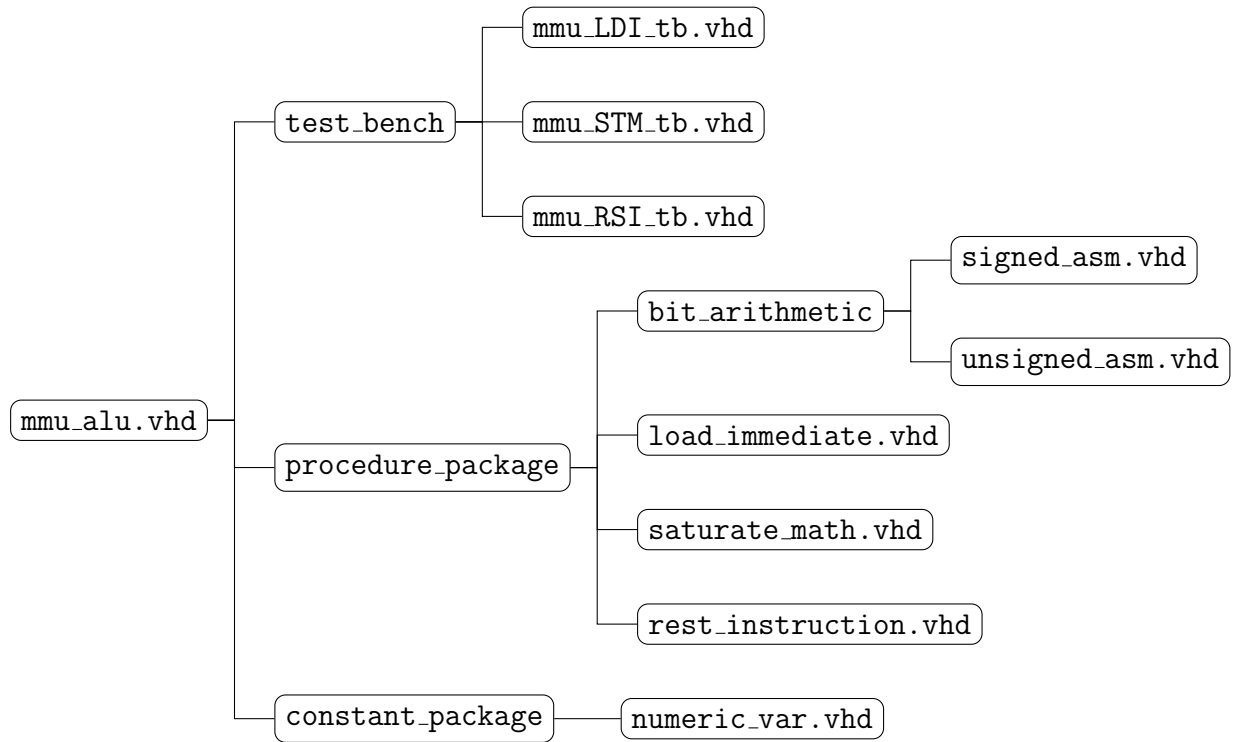
Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexadecimal values.

Signal name	Value	
RV opcode	3F to 30	3F 30
11		11
function	0	F 0
RV in_immed	20	20
RV in_PORT1	700870077006700570047003F0027001 to -----	
RV in_PORT2	80010004800100037FFF7FF27FFFFF1 to -----	
RV out_PORTD	80008FFD80008FFE0FFB0FEF7FFF8FF0	80008FFD80008FFE0FFB0FEF7FFF8FF0
RV wback_flag	1 to 0	

4 Appendix

4.1 File Structure



4.2 Source File: constant_package/numeric_var

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 package numeric_var is
6
7 -----INSTRUCTION_FIELD_CONSTANT-----
8     constant IMMEDIATE_LENGTH      : integer := 16;
9     constant INDEX_LENGTH          : integer := 3;
10    constant OPCODE_LENGTH          : integer := 6;
11    constant INSTRUCTION_LENGTH     : integer := 25;
12
13 -----REGISTER_FILE_CONSTANT-----
14    constant VALUE16                : integer := 16;
15    constant ADDRESS_LENGTH         : integer := 5;
16    constant REGISTER_LENGTH        : integer := 128;
17 end package;
```

4.3 Source File: mmu_alu.vhd

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.numeric_var.all;
5 use work.load_immediate.all;
6 use work.saturate_math.all;
7 use work.rest_instruction.all;
8
9
10 entity MMU_ALU is
11     port (
12         opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14         in_PORT3     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
15         in_PORT2     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
16         in_PORT1     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
17
18         in_immed     : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
19         in_d_ptr     : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
20
21         out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
22         out_d_ptr    : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23         wback_flag   : out std_logic
24     );
25 end entity;
26
27 architecture behavior of MMU_ALU is
28 begin
29     main : process(
30         opcode,
31         in_PORT3, in_PORT2, in_PORT1, in_immed,
32         in_d_ptr
33     )
34     begin
35         out_d_ptr <= in_d_ptr;
36         case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
37             when "00" | "01" =>
38                 LDI_memory( --ref. procedure_package/load_immediate.vhd
39                     opcode,
40                     in_PORT3,
41                     in_immed,
42
43                     out_PORTD,
44                     wback_flag
45                 );
46
47             when "10" =>
48                 STM_main( --ref. procedure_package/saturate_math.vhd
49                     opcode,
```

```

50         in_PORT3,
51         in_PORT2,
52         in_PORT1,
53
54         out_PORTD,
55         wback_flag
56     );
57
58     when "11" =>
59         RSI_main(          --ref. procedure_package/rest_instruction.vhd
60             opcode,
61             in_PORT2,
62             in_PORT1,
63             in_immed,
64
65             out_PORTD,
66             wback_flag
67         );
68
69     when others =>
70         out_PORTD <= (others => '0');
71         out_d_ptr <= (others => '0');
72         wback_flag <= '0';
73     end case;
74 end process;
75 end architecture;

```

4.4 Source File: procedure_package/load_immediate.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.numeric_var.all;
5
6  package load_immediate is
7
8      procedure LDI_memory(
9          signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
10         signal in_PORT3     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
11         signal in_immed     : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
12
13         signal out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
14         signal wback_flag   : out std_logic
15     );
16 end package load_immediate;
17
18 package body load_immediate is
19
20     procedure LDI_memory (
21         signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
22         signal in_PORT3     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
23         signal in_immed     : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
24
25         signal out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
26         signal wback_flag   : out std_logic
27     ) is
28         variable low_bit    : integer;
29         variable high_bit   : integer;
30         variable temp_out   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
31     begin
32         temp_out := in_PORT3;
33         low_bit := to_integer(unsigned(opcode(2 downto 0))) * VALUE16;
34         high_bit := low_bit + VALUE16;
35         temp_out(high_bit-1 downto low_bit) := in_immed;
36
37         out_PORTD <= temp_out;
38         wback_flag <= '1';
39     end procedure;
40 end package body load_immediate;

```

4.5 Source File: procedure_package/saturated_math.vhd

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.numeric_var.all;
5 use work.signed_asm.all;
6
7 package saturate_math is
8   procedure STM_main(
9     signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
10    signal in_PORT3     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
11    signal in_PORT2     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
12    signal in_PORT1     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
13
14    signal out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
15    signal wback_flag   : out std_logic
16  );
17 end package saturate_math;
18
19 package body saturate_math is
20   procedure STM_main (
21     signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
22     signal in_PORT3    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
23     signal in_PORT2    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
24     signal in_PORT1    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
25
26     signal out_PORTD   : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
27     signal wback_flag  : out std_logic
28   ) is
29     variable ret32      : std_logic_vector(31 downto 0);
30     variable ret64      : std_logic_vector(63 downto 0);
31     variable temp_out   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
32     variable wback_var  : std_logic := '1';
33   begin
34     case opcode(2 downto 0) is
35
36       when "000" =>
37         for i in 3 downto 0 loop --low 16-bit integer mult-add
38           mult_16(
39             in_PORT3(16*(i*2+1)-1 downto 16*(i*2)),
40             in_PORT2(16*(i*2+1)-1 downto 16*(i*2)),
41             ret32
42           );
43           add_32(
44             in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
45             ret32,
46             temp_out(16*(i*2+2)-1 downto 16*(i*2))
47           );
48         end loop;
49
50       when "001" =>
51         for i in 3 downto 0 loop --high 16-bit integer mult-add
52           mult_16(
53             in_PORT3(16*(i*2+2)-1 downto 16*(i*2+1)),
54             in_PORT2(16*(i*2+2)-1 downto 16*(i*2+1)),
55             ret32
56           );
57           add_32(
58             in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
59             ret32,
60             temp_out(16*(i*2+2)-1 downto 16*(i*2))
61           );
62         end loop;
63
64       when "010" =>
65         for i in 3 downto 0 loop --low 16-bit integer mult-sub
66           mult_16(
67             in_PORT3(16*(i*2+1)-1 downto 16*(i*2)),
68             in_PORT2(16*(i*2+1)-1 downto 16*(i*2)),
69             ret32
70           );
71           sub_32(
```

```

72         in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
73         ret32,
74         temp_out(16*(i*2+2)-1 downto 16*(i*2))
75     );
76 end loop;
77
78 when "011" =>
79     for i in 3 downto 0 loop --high 16-bit integer mult-sub
80         mult_16(
81             in_PORT3(16*(i*2+2)-1 downto 16*(i*2+1)),
82             in_PORT2(16*(i*2+2)-1 downto 16*(i*2+1)),
83             ret32
84         );
85         sub_32(
86             in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
87             ret32,
88             temp_out(16*(i*2+2)-1 downto 16*(i*2))
89         );
90     end loop;
91
92 when "100" =>
93     for i in 1 downto 0 loop --low 32-bit integer mult-add
94         mult_32(
95             in_PORT3(32*(i*2+1)-1 downto 32*(i*2)),
96             in_PORT2(32*(i*2+1)-1 downto 32*(i*2)),
97             ret64
98         );
99         add_64(
100            in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
101            ret64,
102            temp_out(32*(i*2+2)-1 downto 32*(i*2))
103        );
104    end loop;
105
106 when "101" =>
107     for i in 1 downto 0 loop --high 32-bit integer mult-add
108         mult_32(
109             in_PORT3(32*(i*2+2)-1 downto 32*(i*2+1)),
110             in_PORT2(32*(i*2+2)-1 downto 32*(i*2+1)),
111             ret64
112         );
113         add_64(
114             in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
115             ret64,
116             temp_out(32*(i*2+2)-1 downto 32*(i*2))
117         );
118     end loop;
119
120 when "110" =>
121     for i in 1 downto 0 loop --low 32-bit integer mult-sub
122         mult_32(
123             in_PORT3(32*(i*2+1)-1 downto 32*(i*2)),
124             in_PORT2(32*(i*2+1)-1 downto 32*(i*2)),
125             ret64
126         );
127         sub_64(
128             in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
129             ret64,
130             temp_out(32*(i*2+2)-1 downto 32*(i*2))
131         );
132     end loop;
133
134 when "111" =>
135     for i in 1 downto 0 loop --high 32-bit integer mult-sub
136         mult_32(
137             in_PORT3(32*(i*2+2)-1 downto 32*(i*2+1)),
138             in_PORT2(32*(i*2+2)-1 downto 32*(i*2+1)),
139             ret64
140         );
141         sub_64(
142             in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
143             ret64,
144             temp_out(32*(i*2+2)-1 downto 32*(i*2))

```

```

145         );
146     end loop;
147
148     when others =>
149         wback_var := '0';
150     end case;
151     wback_flag <= wback_var;
152     out_PORTD <= temp_out;
153 end procedure;
154 end package body saturate_math;

```

4.6 Source File: procedure_package/rest_instruction.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.numeric_var.all;
5  use work.unsigned_asm.all;
6  use work.signed_asm.all;
7
8  package rest_instruction is
9      procedure RSI_main(
10         signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
11         signal in_PORT2     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
12         signal in_PORT1     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
13         signal in_immed     : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
14
15         signal out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
16         signal wback_flag   : out std_logic
17     );
18 end package;
19
20 package body rest_instruction is
21     procedure RSI_main(
22         signal opcode      : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
23         signal in_PORT2     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
24         signal in_PORT1     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
25         signal in_immed     : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
26
27         signal out_PORTD    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
28         signal wback_flag   : out std_logic
29     ) is
30         variable temp_out   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
31         variable int_var    : integer := 0;
32         variable unsign16   : unsigned(15 downto 0);
33         variable vector16   : std_logic_vector(15 downto 0);
34         variable vector32   : std_logic_vector(31 downto 0);
35         variable wback_var  : std_logic := '1';
36     begin
37         case opcode(3 downto 0) is
38             when "0000" => --no operation
39                 wback_var := '0';
40
41
42             when "0001" => --shift right halfword immediate
43                 int_var := to_integer(unsigned(in_immed(3 downto 0)));
44                 for i in 0 to 7 loop
45                     unsign16 := unsigned(in_PORT1(16*(i+1)-1 downto i*16));
46                     unsign16 := shift_right(unsign16, int_var);
47                     temp_out(16*(i+1)-1 downto i*16) := std_logic_vector(unsign16);
48                 end loop;
49
50             when "0010" => --add word unsigned
51                 for i in 0 to 3 loop
52                     add_32_unsigned(
53                         in_PORT2(32*(i+1)-1 downto i*32),
54                         in_PORT1(32*(i+1)-1 downto i*32),
55                         temp_out(32*(i+1)-1 downto i*32)
56                     );
57                 end loop;
58

```

```

59     when "0011" => --count 1s in halfword
60         for i in 0 to 7 loop
61             vector16 := in_PORT1(16*(i+1)-1 downto i*16);
62             int_var := 0;
63             for j in 0 to 15 loop
64                 if vector16(j) = '1' then
65                     int_var := int_var + 1;
66                 end if;
67             end loop;
68             temp_out(16*(i+1)-1 downto i*16) := std_logic_vector(to_unsigned(int_var,
69                                     16));
70         end loop;
71     when "0100" => --add halfword saturated
72         for i in 0 to 7 loop
73             add_16(
74                 in_PORT2(16*(i+1)-1 downto i*16),
75                 in_PORT1(16*(i+1)-1 downto i*16),
76                 temp_out(16*(i+1)-1 downto i*16)
77             );
78         end loop;
79     when "0101" => --bitwise logical or
80         temp_out := in_PORT2 or in_PORT1;
81     when "0110" => --broadcast word
82         for i in 0 to 3 loop
83             temp_out(32*(i+1)-1 downto i*32) := in_PORT1(REGISTER_LENGTH-1 downto
84                                     REGISTER_LENGTH-32);
85         end loop;
86     when "0111" => -- max signed word
87         for i in 0 to 3 loop
88             if signed(in_PORT2(32*(i+1)-1 downto i*32)) > signed(in_PORT1(32*(i+1)-1
89                                     downto i*32)) then
90                 temp_out(32*(i+1)-1 downto i*32) := in_PORT2(32*(i+1)-1 downto i*32);
91             else
92                 temp_out(32*(i+1)-1 downto i*32) := in_PORT1(32*(i+1)-1 downto i*32);
93             end if;
94         end loop;
95     when "1000" => --min signed word
96         for i in 0 to 3 loop
97             if signed(in_PORT2(32*(i+1)-1 downto i*32)) < signed(in_PORT1(32*(i+1)-1
98                                     downto i*32)) then
99                 temp_out(32*(i+1)-1 downto i*32) := in_PORT2(32*(i+1)-1 downto i*32);
100             else
101                 temp_out(32*(i+1)-1 downto i*32) := in_PORT1(32*(i+1)-1 downto i*32);
102             end if;
103         end loop;
104     when "1001" => --multiply low unsigned
105         for i in 0 to 3 loop
106             mult_16_unsigned(
107                 in_PORT1(16*(i*2+1)-1 downto 16*(i*2)),
108                 in_PORT2(16*(i*2+1)-1 downto 16*(i*2)),
109                 temp_out(32*(i+1)-1 downto 32*i)
110             );
111         end loop;
112     when "1010" => --multiply low by constant unsigned
113         for i in 0 to 3 loop
114             mult_16_unsigned(
115                 in_PORT1(16*(i*2+1)-1 downto 16*(i*2)),
116                 "000000000000" & in_immed(4 downto 0),
117                 temp_out(32*(i+1)-1 downto 32*i)
118             );
119         end loop;
120     when "1011" => --bitwise logical and
121         temp_out := in_PORT2 and in_PORT1;
122     when "1100" => --count leading zeroes in words

```



```

128         for i in 0 to 3 loop
129             vector32 := in_PORT1(32*(i+1)-1 downto i*32);
130             int_var := 0;
131         if unsigned(vector32) = 0 then
132             int_var := 0;
133         else
134             for bit_index in 31 downto 0 loop
135                 if vector32(bit_index) = '0' then
136                     int_var := int_var + 1;
137                 else
138                     exit;
139                 end if;
140             end loop;
141         end if;
142         temp_out(32*(i+1)-1 downto i*32) := std_logic_vector(to_unsigned(int_var,
143                                     32));
144     end loop;
145
146     when "1101" => --rotate bits in word
147         for i in 0 to 3 loop
148             vector32 := in_PORT1(32*(i+1)-1 downto i*32);
149             int_var := to_integer(unsigned(in_PORT2(32*(i+1)-27-1 downto i*32))) mod
150                 32;
151             if int_var = 0 then
152                 temp_out(32*(i+1)-1 downto 32*i) := vector32;
153             else
154                 temp_out(32*(i+1)-1 downto 32*i) := vector32(int_var-1 downto 0) &
155                     vector32(31 downto int_var);
156             end if;
157         end loop;
158
159     when "1110" => --subtract from word unsigned
160         for i in 0 to 3 loop
161             sub_32_unsigned(
162                 in_PORT2(32*(i+1)-1 downto 32*i),
163                 in_PORT1(32*(i+1)-1 downto 32*i),
164                 temp_out(32*(i+1)-1 downto 32*i)
165             );
166         end loop;
167
168     when "1111" => --subtract from halfword saturated
169         for i in 0 to 7 loop
170             sub_16(
171                 in_PORT2(16*(i+1)-1 downto 16*i),
172                 in_PORT1(16*(i+1)-1 downto 16*i),
173                 temp_out(16*(i+1)-1 downto 16*i)
174             );
175         end loop;
176
177     when others =>
178         wback_var := '0';
179     end case;
180     if wback_var = '1' then
181         out_PORTD <= temp_out;
182     end if;
183     wback_flag <= wback_var;
184
185 end procedure;
186 end package body rest_instruction;

```

4.7 Source File: procedure_package/bit_arithmetic/unsigned.vhd

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 package unsigned_asm is
6     procedure mult_16_unsigned(
7         a16, b16 : in std_logic_vector(15 downto 0); -- 32 bits or half-long
8         ret32 : out std_logic_vector(31 downto 0));
9

```

```

10  procedure add_32_unsigned(
11      a32, b32 : in std_logic_vector(31 downto 0);
12      ret32    : out std_logic_vector(31 downto 0));
13
14  procedure sub_32_unsigned(
15      a32, b32 : in std_logic_vector(31 downto 0);
16      ret32    : out std_logic_vector(31 downto 0));
17
18  end package unsigned_asm;
19
20  package body unsigned_asm is
21      constant MAX_32_unsigned : unsigned(31 downto 0) := (others => '1');
22      constant MIN_32_unsigned : unsigned(31 downto 0) := (others => '0');
23
24  -----Unsigned-Multiple-16-bits-----
25  procedure mult_16_unsigned(
26      a16, b16 : in std_logic_vector(15 downto 0); -- 32 bits or half-long
27      ret32    : out std_logic_vector(31 downto 0)
28  ) is
29      variable prod : unsigned(31 downto 0);
30  begin
31      prod := unsigned(a16) * unsigned(b16);
32      ret32 := std_logic_vector(prod);
33  end procedure;
34
35  -----Unsigned-Addition-32-bits-----
36  procedure add_32_unsigned(
37      a32, b32 : in std_logic_vector(31 downto 0);
38      ret32    : out std_logic_vector(31 downto 0)
39  ) is
40      variable sum : unsigned(32 downto 0);
41  begin
42      sum := unsigned('0' & a32) + unsigned('0' & b32);
43
44      if sum(32) = '1' then
45          ret32 := std_logic_vector(MAX_32_UNSIGNED);
46      else
47          ret32 := std_logic_vector(sum(31 downto 0));
48      end if;
49  end procedure;
50
51  -----Unsigned-Subtraction-32-bits-----
52  procedure sub_32_unsigned(
53      a32, b32 : in std_logic_vector(31 downto 0);
54      ret32    : out std_logic_vector(31 downto 0)
55  ) is
56      variable diff : unsigned(32 downto 0);
57  begin
58      diff := unsigned('0' & a32) - unsigned('0' & b32);
59
60      if diff(32) = '1' then
61          ret32 := std_logic_vector(MIN_32_UNSIGNED);
62      else
63          ret32 := std_logic_vector(diff(31 downto 0));
64      end if;
65  end procedure;
66  end package body unsigned_asm;

```

4.8 Source File: procedure_package/bit_arithmetic/signed.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  package signed_asm is
6      procedure mult_16(
7          a16, b16 : in std_logic_vector(15 downto 0);
8          ret32    : out std_logic_vector(31 downto 0));
9
10     procedure mult_32(
11         a32, b32 : in std_logic_vector(31 downto 0);

```

```

12     ret64    : out std_logic_vector(63 downto 0));
13
14     procedure add_16(
15         a16, b16 : in std_logic_vector(15 downto 0);
16         ret16    : out std_logic_vector(15 downto 0));
17
18     procedure sub_16(
19         a16, b16 : in std_logic_vector(15 downto 0);
20         ret16    : out std_logic_vector(15 downto 0));
21
22     procedure add_32(
23         a32, b32 : in std_logic_vector(31 downto 0);
24         ret32    : out std_logic_vector(31 downto 0));
25
26     procedure sub_32(
27         a32, b32 : in std_logic_vector(31 downto 0);
28         ret32    : out std_logic_vector(31 downto 0));
29
30     procedure add_64(
31         a64, b64 : in std_logic_vector(63 downto 0);
32         ret64    : out std_logic_vector(63 downto 0));
33
34     procedure sub_64(
35         a64, b64 : in std_logic_vector(63 downto 0);
36         ret64    : out std_logic_vector(63 downto 0));
37
38 end package signed_asm;
39
40 package body signed_asm is
41     constant MAX16 : signed(15 downto 0) := not(shift_left(to_signed(1, 16), 15));
42     constant MIN16 : signed(15 downto 0) := shift_left(to_signed(1, 16), 15);
43     constant MAX32 : signed(31 downto 0) := not(shift_left(to_signed(1, 32), 31));
44     constant MIN32 : signed(31 downto 0) := shift_left(to_signed(1, 32), 31);
45     constant MAX64 : signed(63 downto 0) := not(shift_left(to_signed(1, 64), 63));
46     constant MIN64 : signed(63 downto 0) := shift_left(to_signed(1, 64), 63);
47
48 -----Multiple_16-bits-----
49     procedure mult_16(
50         a16, b16 : in std_logic_vector(15 downto 0);
51         ret32    : out std_logic_vector(31 downto 0)
52     ) is
53         variable a_s, b_s : signed(15 downto 0);
54         variable prod     : signed(31 downto 0);
55     begin
56         a_s := signed(a16);
57         b_s := signed(b16);
58         prod := a_s * b_s;
59         ret32 := std_logic_vector(prod);
60     end procedure;
61
62 -----Multiple_32-bits-----
63     procedure mult_32(
64         a32, b32 : in std_logic_vector(31 downto 0);
65         ret64    : out std_logic_vector(63 downto 0)
66     ) is
67         variable prod : signed(63 downto 0);
68     begin
69         prod := signed(a32) * signed(b32);
70         ret64 := std_logic_vector(prod);
71     end procedure;
72
73 -----Addition_16-bits-----
74     procedure add_16(
75         a16, b16 : in std_logic_vector(15 downto 0);
76         ret16    : out std_logic_vector(15 downto 0)
77     ) is
78         variable sum : signed(16 downto 0);
79     begin
80         sum := resize(signed(a16), 17) + resize(signed(b16), 17);
81
82         if sum > resize(MAX16, 17) then
83             ret16 := std_logic_vector(MAX16);
84

```

```

85     elsif sum < resize(MIN16, 17) then
86         ret16 := std_logic_vector(MIN16);
87     else
88         ret16 := std_logic_vector(sum(15 downto 0));
89     end if;
90 end procedure;
91
92 -----Subtraction_16-bits-----
93 procedure sub_16(
94     a16, b16 : in std_logic_vector(15 downto 0);
95     ret16    : out std_logic_vector(15 downto 0)
96 ) is
97     variable sum : signed(16 downto 0);
98 begin
99     sum := resize(signed(a16), 17) - resize(signed(b16), 17);
100
101     if sum < resize(MIN16, 17) then
102         ret16 := std_logic_vector(MIN16);
103     elsif sum > resize(MAX16, 17) then
104         ret16 := std_logic_vector(MAX16);
105     else
106         ret16 := std_logic_vector(sum(15 downto 0));
107     end if;
108 end procedure;
109
110 -----Addition_32-bits-----
111 procedure add_32(
112     a32, b32 : in std_logic_vector(31 downto 0);
113     ret32    : out std_logic_vector(31 downto 0)
114 ) is
115     variable sum : signed(32 downto 0);
116 begin
117     sum := resize(signed(a32), 33) + resize(signed(b32), 33);
118
119     if sum > resize(MAX32, 33) then
120         ret32 := std_logic_vector(MAX32);
121     elsif sum < resize(MIN32, 33) then
122         ret32 := std_logic_vector(MIN32);
123     else
124         ret32 := std_logic_vector(sum(31 downto 0));
125     end if;
126 end procedure;
127
128 -----Subtraction_32-bits-----
129 procedure sub_32(
130     a32, b32 : in std_logic_vector(31 downto 0);
131     ret32    : out std_logic_vector(31 downto 0)
132 ) is
133     variable diff : signed(32 downto 0);
134 begin
135     diff := resize(signed(a32), 33) - resize(signed(b32), 33);
136
137     if diff < resize(MIN32, 33) then
138         ret32 := std_logic_vector(MIN32);
139     elsif diff > resize(MAX32, 33) then
140         ret32 := std_logic_vector(MAX32);
141     else
142         ret32 := std_logic_vector(diff(31 downto 0));
143     end if;
144
145 end procedure;
146
147 -----Addition_64-bits-----
148 procedure add_64(
149     a64, b64 : in std_logic_vector(63 downto 0);
150     ret64    : out std_logic_vector(63 downto 0)
151 ) is
152     variable sum : signed(64 downto 0);
153 begin
154     sum := resize(signed(a64), 65) + resize(signed(b64), 65);
155
156     if sum > resize(MAX64, 65) then
157         ret64 := std_logic_vector(MAX64);

```

```

158     elsif sum < resize(MIN64, 65) then
159         ret64 := std_logic_vector(MIN64);
160     else
161         ret64 := std_logic_vector(sum(63 downto 0));
162     end if;
163 end procedure;
164
165 -----Subtraction_64-bits-----
166 procedure sub_64(
167     a64, b64 : in std_logic_vector(63 downto 0);
168     ret64    : out std_logic_vector(63 downto 0)
169 ) is
170     variable diff : signed(64 downto 0);
171 begin
172
173     diff := resize(signed(a64), 65) - resize(signed(b64), 65);
174
175     if diff < resize(MIN64, 65) then
176         ret64 := std_logic_vector(MIN64);
177     elsif diff > resize(MAX64, 65) then
178         ret64 := std_logic_vector(MAX64);
179     else
180         ret64 := std_logic_vector(diff(63 downto 0));
181     end if;
182 end procedure;
183 end package body signed_asm;

```

4.9 Test-Bench File: test_bench/mmu_LDI_tb.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use work.numeric_var.all;
5  use work.all;
6
7  entity mmu_LDI_tb is
8  end mmu_LDI_tb;
9
10 architecture test_bench of mmu_LDI_tb is
11     -- Inputs to ALU
12     signal opcode      : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13     signal in_PORT3     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
14     signal in_PORT2     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15     signal in_PORT1     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16     signal in_immed     : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
17     signal in_d_ptr     : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
18
19     -- Outputs ALU
20     signal out_PORTD    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
21     signal out_d_ptr    : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22     signal wback_flag   : std_logic;
23
24     constant period: time := 20ns;
25
26     -- Helper: Convert std_logic_vector ? hex string (portable)
27
28     function slv_to_hex(slv : std_logic_vector) return string is
29         variable num_nibbles : integer := (slv'length + 3) / 4;
30         variable padded_slv   : std_logic_vector(num_nibbles * 4 - 1 downto 0);
31         variable result       : string(1 to num_nibbles);
32         variable nibble_val   : integer;
33     begin
34         -- Pad MSBs with zeros if not multiple of 4 bits
35         padded_slv := (others => '0');
36         padded_slv(slv'length - 1 downto 0) := slv;
37
38         for i in 0 to num_nibbles - 1 loop
39             nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
40             case nibble_val is
41                 when 0 => result(num_nibbles - i) := '0';
42                 when 1 => result(num_nibbles - i) := '1';

```

```

43     when 2 => result(num_nibbles - i) := '2';
44     when 3 => result(num_nibbles - i) := '3';
45     when 4 => result(num_nibbles - i) := '4';
46     when 5 => result(num_nibbles - i) := '5';
47     when 6 => result(num_nibbles - i) := '6';
48     when 7 => result(num_nibbles - i) := '7';
49     when 8 => result(num_nibbles - i) := '8';
50     when 9 => result(num_nibbles - i) := '9';
51     when 10 => result(num_nibbles - i) := 'A';
52     when 11 => result(num_nibbles - i) := 'B';
53     when 12 => result(num_nibbles - i) := 'C';
54     when 13 => result(num_nibbles - i) := 'D';
55     when 14 => result(num_nibbles - i) := 'E';
56     when 15 => result(num_nibbles - i) := 'F';
57     when others => result(num_nibbles - i) := 'X';
58 end case;
59 end loop;
60 return result;
61 end function;
62 -----
63 begin
64
65     UUT : entity work.MMU_ALU
66     port map(
67         opcode      => opcode ,
68
69         in_PORT3     => in_PORT3 ,
70         in_PORT2     => in_PORT2 ,
71         in_PORT1     => in_PORT1 ,
72         in_immed     => in_immed ,
73         in_d_ptr     => in_d_ptr ,
74
75         out_PORTD    => out_PORTD ,
76         out_d_ptr    => out_d_ptr ,
77         wback_flag   => wback_flag
78     );
79
80     -- stimulus process
81     stim_proc : process
82     begin
83         -----
84         -- load_immediate TEST w/ indexing
85         -----
86         in_immed <= x"DEAD";
87         in_d_ptr <= b"00000";
88         in_PORT3 <= (others => '0');
89         -----
90         -- TEST: opcode = 0--000
91         -----
92         opcode <= std_logic_vector(to_unsigned(0, OPCODE_LENGTH));
93         wait for period;
94         assert out_PORTD(15 downto 0) = x"DEAD"
95         report "Test failed: 000000, out_PORTD = x" & slv_to_hex(out_PORTD)
96         severity error;
97         -----
98         -- TEST: opcode = 0--001
99         -----
100        -----
101        opcode <= std_logic_vector(to_unsigned(1, OPCODE_LENGTH));
102        wait for period;
103        assert out_PORTD(31 downto 16) = x"DEAD"
104        report "Test failed: 000001, out_PORTD = x" & slv_to_hex(out_PORTD)
105        severity error;
106        -----
107        -----
108        -- TEST: opcode = 0--110
109        -----
110        opcode <= std_logic_vector(to_unsigned(6, OPCODE_LENGTH));
111        wait for period;
112        assert out_PORTD(111 downto 96) = x"DEAD"
113        report "Test failed: 100110, out_PORTD = x" & slv_to_hex(out_PORTD)
114        severity error;
115

```

```

116 -----
117 -- TEST: opcode = 0--111
118 -----
119     opcode <= std_logic_vector(to_unsigned(7, OPCODE_LENGTH));
120     wait for period;
121     assert out_PORTD(127 downto 112) = x"DEAD"
122         report "Test failed: 100111, out_PORTD = x" & slv_to_hex(out_PORTD)
123         severity error;
124
125     report "TEST COMPLETED: load_immediate w/ indexing" severity warning;
126 end process;
127 end test_bench;

```

4.10 Test-Bench File: test_bench/mmu_STM_tb.vhd

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.numeric_var.all;
5 use work.all;
6
7 entity mmu_STM_tb is
8 end mmu_STM_tb;
9
10 architecture test_bench of mmu_STM_tb is
11     -- Inputs to ALU
12     signal opcode      : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13     signal in_PORT3    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
14     signal in_PORT2    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15     signal in_PORT1    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16     signal in_immed    : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
17     signal in_d_ptr    : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
18
19     -- Outputs ALU
20     signal out_PORTD    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
21     signal out_d_ptr    : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22     signal wback_flag   : std_logic;
23
24     constant period: time := 20ns;
25
26     -- Helper: Convert std_logic_vector ? hex string (portable)
27
28     function slv_to_hex(slv : std_logic_vector) return string is
29         variable num_nibbles : integer := (slv'length + 3) / 4;
30         variable padded_slv  : std_logic_vector(num_nibbles * 4 - 1 downto 0);
31         variable result      : string(1 to num_nibbles);
32         variable nibble_val  : integer;
33     begin
34         -- Pad MSBs with zeros if not multiple of 4 bits
35         padded_slv := (others => '0');
36         padded_slv(slv'length - 1 downto 0) := slv;
37
38         for i in 0 to num_nibbles - 1 loop
39             nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
40             case nibble_val is
41                 when 0 => result(num_nibbles - i) := '0';
42                 when 1 => result(num_nibbles - i) := '1';
43                 when 2 => result(num_nibbles - i) := '2';
44                 when 3 => result(num_nibbles - i) := '3';
45                 when 4 => result(num_nibbles - i) := '4';
46                 when 5 => result(num_nibbles - i) := '5';
47                 when 6 => result(num_nibbles - i) := '6';
48                 when 7 => result(num_nibbles - i) := '7';
49                 when 8 => result(num_nibbles - i) := '8';
50                 when 9 => result(num_nibbles - i) := '9';
51                 when 10 => result(num_nibbles - i) := 'A';
52                 when 11 => result(num_nibbles - i) := 'B';
53                 when 12 => result(num_nibbles - i) := 'C';
54                 when 13 => result(num_nibbles - i) := 'D';
55                 when 14 => result(num_nibbles - i) := 'E';
56                 when 15 => result(num_nibbles - i) := 'F';

```

```

57         when others => result(num_nibbles - i) := 'X';
58     end case;
59 end loop;
60 return result;
61 end function;
62 -----
63
64 begin
65     UUT : entity work.MMU_ALU
66     port map(
67         opcode      => opcode,
68         in_PORT3     => in_PORT3,
69         in_PORT2     => in_PORT2,
70         in_PORT1     => in_PORT1,
71         in_immed     => in_immed,
72         in_d_ptr     => in_d_ptr,
73
74         out_PORTD     => out_PORTD,
75         out_d_ptr     => out_d_ptr,
76         wback_flag   => wback_flag
77     );
78
79     -- Stimulus process
80     stim_proc : process
81     begin
82         -----
83         -- saturate_math TEST w/overflow and underflow checks
84         -----
85         in_immed <= x"DEAD";
86         in_d_ptr <= b"00000";
87         in_PORT3 <= x"000100020003000470057006F0077008";
88         in_PORT2 <= x"000800070006000570047003F0027001";
89         in_PORT1 <= x"00000000000000007FFF00007FFF0000";
90
91         -----
92         -- TEST: Opcode 10-000
93         -----
94         opcode <= "100000";
95         wait for period;
96         assert out_PORTD = x"0000000E000000147FFFFFFF7FFFFFFF"
97             report "Test failed: 100000, out_PORTD = x" & slv_to_hex(out_PORTD)
98             severity error;
99
100         -----
101         -- TEST: Opcode 10-001
102         -----
103         opcode <= "100001";
104         wait for period;
105         assert out_PORTD = x"000000080000000127FFFFFFF7FFFFFFF"
106             report "Test failed: 100001, out_PORTD = x" & slv_to_hex(out_PORTD)
107             severity error;
108
109         -----
110         -- TEST: Opcode 10-010
111         -----
112         in_PORT1 <= x"00000000000000008001000080010000";
113         opcode <= "100010";
114         wait for period;
115         assert out_PORTD = x"FFFFFFF2FFFFFFEC8000000080000000"
116             report "Test failed: 100010, out_PORTD = x" & slv_to_hex(out_PORTD)
117             severity error;
118
119         -----
120         -- TEST: Opcode 10-011
121         -----
122         opcode <= "100011";
123         wait for period;
124         assert out_PORTD = x"FFFFFFF8FFFFFFEE8000000080000000"
125             report "Test failed: 100011, out_PORTD = x" & slv_to_hex(out_PORTD)
126             severity error;
127
128         -----
129         -- TEST: Opcode 10-100

```



```

130 -----
131 in_PORT1 <= x"0000000000000007FFF0007FFF0000";
132 opcode <= "100100";
133 wait for period;
134 assert out_PORTD = x"00000012002700147FFFFFFFFFFFFFFF"
135     report "Test failed: 100100, out_PORTD = x" & slv_to_hex(out_PORTD)
136     severity error;
137
138 -----
139 -- TEST: Opcode 10-101
140 -----
141 opcode <= "100101";
142 wait for period;
143 assert out_PORTD = x"000000080017000E7FFFFFFFFFFFFFFF"
144     report "Test failed: 100101, out_PORTD = x" & slv_to_hex(out_PORTD)
145     severity error;
146
147 -----
148 -- TEST: Opcode 10-110
149 -----
150 in_PORT1 <= x"0000000000000008001000080010000";
151 opcode <= "100110";
152 wait for period;
153 assert out_PORTD = x"FFFFFFEDFFD8FFEC8000000000000000"
154     report "Test failed: 100110, out_PORTD = x" & slv_to_hex(out_PORTD)
155     severity error;
156
157 -----
158 -- TEST: Opcode 10-111
159 -----
160 opcode <= "100111";
161 wait for period;
162 assert out_PORTD = x"FFFFFFF7FFE8FFF28000000000000000"
163     report "Test failed: 100111, out_PORTD = x" & slv_to_hex(out_PORTD)
164     severity error;
165
166 report "TEST COMPLETED: saturate_math w/o saturating" severity warning;
167 end process;
168 end test_bench;

```

4.11 Test-Bench File: test_bench/mmu_RSI_tb.vhd

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use work.numeric_var.all;
5 use work.all;
6
7 entity mmu_RSI_tb is
8 end mmu_RSI_tb;
9
10 architecture test_bench of mmu_RSI_tb is
11     -- Inputs to ALU
12     signal opcode      : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13     signal in_PORT3     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
14     signal in_PORT2     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15     signal in_PORT1     : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16     signal in_immed     : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
17     signal in_d_ptr     : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
18
19     -- Outputs ALU
20     signal out_PORTD    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
21     signal out_d_ptr    : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22     signal wback_flag   : std_logic;
23
24     constant period: time := 20ns;
25
26     -- Helper: Convert std_logic_vector ? hex string (portable)
27
28     function slv_to_hex(slv : std_logic_vector) return string is
29         variable num_nibbles : integer := (slv'length + 3) / 4;

```

```

30         variable padded_slv : std_logic_vector(num_nibbles * 4 - 1 downto 0);
31         variable result      : string(1 to num_nibbles);
32         variable nibble_val  : integer;
33     begin
34         -- Pad MSBs with zeros if not multiple of 4 bits
35         padded_slv := (others => '0');
36         padded_slv(slv'length - 1 downto 0) := slv;
37
38         for i in 0 to num_nibbles - 1 loop
39             nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
40             case nibble_val is
41                 when 0 => result(num_nibbles - i) := '0';
42                 when 1 => result(num_nibbles - i) := '1';
43                 when 2 => result(num_nibbles - i) := '2';
44                 when 3 => result(num_nibbles - i) := '3';
45                 when 4 => result(num_nibbles - i) := '4';
46                 when 5 => result(num_nibbles - i) := '5';
47                 when 6 => result(num_nibbles - i) := '6';
48                 when 7 => result(num_nibbles - i) := '7';
49                 when 8 => result(num_nibbles - i) := '8';
50                 when 9 => result(num_nibbles - i) := '9';
51                 when 10 => result(num_nibbles - i) := 'A';
52                 when 11 => result(num_nibbles - i) := 'B';
53                 when 12 => result(num_nibbles - i) := 'C';
54                 when 13 => result(num_nibbles - i) := 'D';
55                 when 14 => result(num_nibbles - i) := 'E';
56                 when 15 => result(num_nibbles - i) := 'F';
57                 when others => result(num_nibbles - i) := 'X';
58             end case;
59         end loop;
60         return result;
61     end function;
62     -----
63
64 begin
65     UUT : entity work.MMU_ALU
66     port map(
67         opcode      => opcode,
68         in_PORT3     => in_PORT3,
69         in_PORT2     => in_PORT2,
70         in_PORT1     => in_PORT1,
71         in_immed     => in_immed,
72         in_d_ptr     => in_d_ptr,
73
74         out_PORTD    => out_PORTD,
75         out_d_ptr    => out_d_ptr,
76         wback_flag   => wback_flag
77     );
78
79     -- Clock process
80
81     -- Stimulus process
82     stim_proc : process
83     begin
84         -----
85         -- rest_instruction TEST
86         -----
87         in_d_ptr <= b"00000";
88
89         -----
90         -- TEST: Opcode 110001
91         -----
92         opcode <= "110001";
93         in_immed <= x"0004";
94         in_PORT2 <= (others => '-');
95         in_PORT1 <= x"700870077006700570047003F0027001";
96         wait for period;
97         assert out_PORTD = x"07000700070007000700070007000700"
98         report "TEST FAIL: 110001, out_PORTD =" & slv_to_hex(out_PORTD)
99         severity error;
100         -----
101         -- TEST: Opcode 110010, staturated

```

```

103 -----
104 opcode <= "110010";
105 in_PORT2 <= x"80010004800100037FFF00027FFF0001";
106 in_PORT1 <= x"700870077006700570047003F0027001";
107 wait for period;
108 assert out_PORTD = x"F009700BF0077008F0037005FFFFFFF"
109 report "TEST FAIL: 110010, out_PORTD =" & slv_to_hex(out_PORTD)
110 severity error;
111
112 -----
113 -- TEST: Opcode 110011
114 -----
115 opcode <= "110011";
116 in_PORT2 <= (others => '-');
117 in_PORT1 <= x"700870077006700570047003F0027001";
118 wait for period;
119 assert out_PORTD = x"00040006000500050004000500050004"
120 report "TEST FAIL: 110011, out_PORTD =" & slv_to_hex(out_PORTD)
121 severity error;
122
123 -----
124 -- TEST: Opcode 110100
125 -----
126 opcode <= "110100";
127 in_PORT2 <= x"80010004800100037FFF7FF27FFFFFFF1";
128 in_PORT1 <= x"700870077006800570047003F0027001";
129 wait for period;
130 assert out_PORTD = x"F009700BF00780087FFF7FFF70016FF2"
131 report "TEST FAIL: 110100, out_PORTD =" & slv_to_hex(out_PORTD)
132 severity error;
133
134 -----
135 -- TEST: Opcode 110101
136 -----
137 opcode <= "110101";
138 in_PORT2 <= x"0000000000000000FFFFFFFFFFFFFFFF";
139 in_PORT1 <= x"700870077006700570047003F0027001";
140 wait for period;
141 assert out_PORTD = x"7008700770067005FFFFFFFFFFFFFFFF"
142 report "TEST FAIL: 110101, out_PORTD =" & slv_to_hex(out_PORTD)
143 severity error;
144
145 -----
146 -- TEST: Opcode 110110
147 -----
148 opcode <= "110110";
149 in_PORT2 <= (others => '-');
150 in_PORT1 <= x"700870077006700570047003F0027001";
151 wait for period;
152 assert out_PORTD = x"70087007700870077008700770087007"
153 report "TEST FAIL: 110110, out_PORTD =" & slv_to_hex(out_PORTD)
154 severity error;
155
156 -----
157 -- TEST: Opcode 110111
158 -----
159 opcode <= "110111";
160 in_PORT2 <= x"80010004800100037FFF00027FFF0001";
161 in_PORT1 <= x"700870077006700570047003F0027001";
162 wait for period;
163 assert out_PORTD = x"70087007700670057FFF00027FFF0001"
164 report "TEST FAIL: 110111, out_PORTD =" & slv_to_hex(out_PORTD)
165 severity error;
166
167 -----
168 -- TEST: Opcode 111000
169 -----
170 opcode <= "111000";
171 in_PORT2 <= x"80010004800100037FFF00027FFF0001";
172 in_PORT1 <= x"700870077006700570047003F0027001";
173 wait for period;
174 assert out_PORTD = x"800100048001000370047003F0027001"
175 report "TEST FAIL: 111000, out_PORTD =" & slv_to_hex(out_PORTD)

```

```

176 severity error;
177
178 -----
179 -- TEST: Opcode 111001
180 -----
181 opcode <= "111001";
182 in_PORT2 <= x"80010004800100037FFF7FF27FFFFFFF1";
183 in_PORT1 <= x"700870077006700570047003F0027001";
184 wait for period;
185 assert out_PORTD = x"0001C01C0001500F37FB5FD66FFA6FF1"
186 report "TEST FAIL: 111001, out_PORTD =" & slv_to_hex(out_PORTD)
187 severity error;
188
189 -----
190 -- TEST: Opcode 111010
191 -----
192 opcode <= "111010";
193 in_immed <= x"0014";
194 in_PORT2 <= (others => '-');
195 in_PORT1 <= x"80010004800100037FFF00027FFF0001";
196 wait for period;
197 assert out_PORTD = x"0000005000000003C0000002800000014"
198 report "TEST FAIL: 111010, out_PORTD =" & slv_to_hex(out_PORTD)
199 severity error;
200
201 -----
202 -- TEST: Opcode 111011
203 -----
204 opcode <= "111011";
205 in_PORT2 <= x"0000000000000000FFFFFFFFFFFFFFFF";
206 in_PORT1 <= x"80010004800100037FFF00027FFF0001";
207 wait for period;
208 assert out_PORTD = x"0000000000000007FFF00027FFF0001"
209 report "TEST FAIL: 111011, out_PORTD =" & slv_to_hex(out_PORTD)
210 severity error;
211
212 -----
213 -- TEST: Opcode 111100
214 -----
215 opcode <= "111100";
216 in_PORT2 <= (others => '-');
217 in_PORT1 <= x"070870071006700570047003F0027001";
218 wait for period;
219 assert out_PORTD = x"00000005000000030000000100000000"
220 report "TEST FAIL: 111100, out_PORTD =" & slv_to_hex(out_PORTD)
221 severity error;
222
223 -----
224 -- TEST: Opcode 111101
225 -----
226 opcode <= "111101";
227 in_PORT2 <= x"FFFFFFF10FFFFFFFFF00000000100000000";
228 in_PORT1 <= x"80010004800100037FFF00027FFF0001";
229 wait for period;
230 assert out_PORTD = x"00048001000380013FFF80017FFF0001"
231 report "TEST FAIL: 111101, out_PORTD =" & slv_to_hex(out_PORTD)
232 severity error;
233
234 -----
235 -- TEST: Opcode 111110
236 -----
237 opcode <= "111110";
238 in_PORT2 <= x"80010004800100037FFF00027FFF0001";
239 in_PORT1 <= x"700870077006700570047003F0027001";
240 wait for period;
241 assert out_PORTD = x"0FF88FFD0FFA8FF8FFA8FF8FF00000000"
242 report "TEST FAIL: 111110, out_PORTD =" & slv_to_hex(out_PORTD)
243 severity error;
244
245 -----
246 -- TEST: Opcode 111111
247 -----
248 opcode <= "111111";

```

```

249 in_PORT2 <= x"80010004800100057FFF00067FFF8F01";
250 in_PORT1 <= x"70080007780600037004FFF3F0027001";
251 wait for period;
252 assert out_PORTD = x"8000FFFD800000020FFB00137FFF8000"
253 report "TEST FAIL: 111111, out_PORTD =" & slv_to_hex(out_PORTD)
254 severity error;
255
256 -----
257 -- TEST: Opcode 110000
258 -----
259 opcode <= "110000";
260 in_PORT2 <= (others => '-');
261 in_PORT1 <= (others => '-');
262 wait for period;
263 assert wback_flag = '0'
264 report "TEST FAIL: 110000, wback =" & std_logic'image(wback_flag)
265 severity error;
266
267     report "TEST COMPLETED: rest of the instruction" severity warning;
268 end process;
269 end test_bench;

```