

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

Final Project Report: Draft

Jin Yuan Chen  
115702978

November 30, 2025

## Abstract

The goal of this project involved the designing of the structural and behavioral style 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 pipelines 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 the **Final Report**, focusing the design, implementation, and verification of a 4-stage pipeline architecture of a Multimedia ALU. The concepts of Instruction file, Register file, Multimedia ALU, and Write-back will be covered as the different modules that define each stage of the pipeline.

## 1.1 Procedure

The program counter fetched an Instruction pre-loaded in the Instruction File or Buffer. Then each instruction is parsed and decoded for the register file to feed the necessary opcode, operands, pointer, and control signal before execution. The Multimedia ALU executes all the arithmetic operations corresponding to the decoded input signals from the register file. Finally, the outputs will be written back to the register file and forwarded to the ALU during the write-back stage to avoid hazards. There exist three interstage registers (IF/ID, ID/EX, EX/WB), acting as buffer for each clock cycle. Its important to note that the EX/WB register is also the write-back module during the write-back stage.

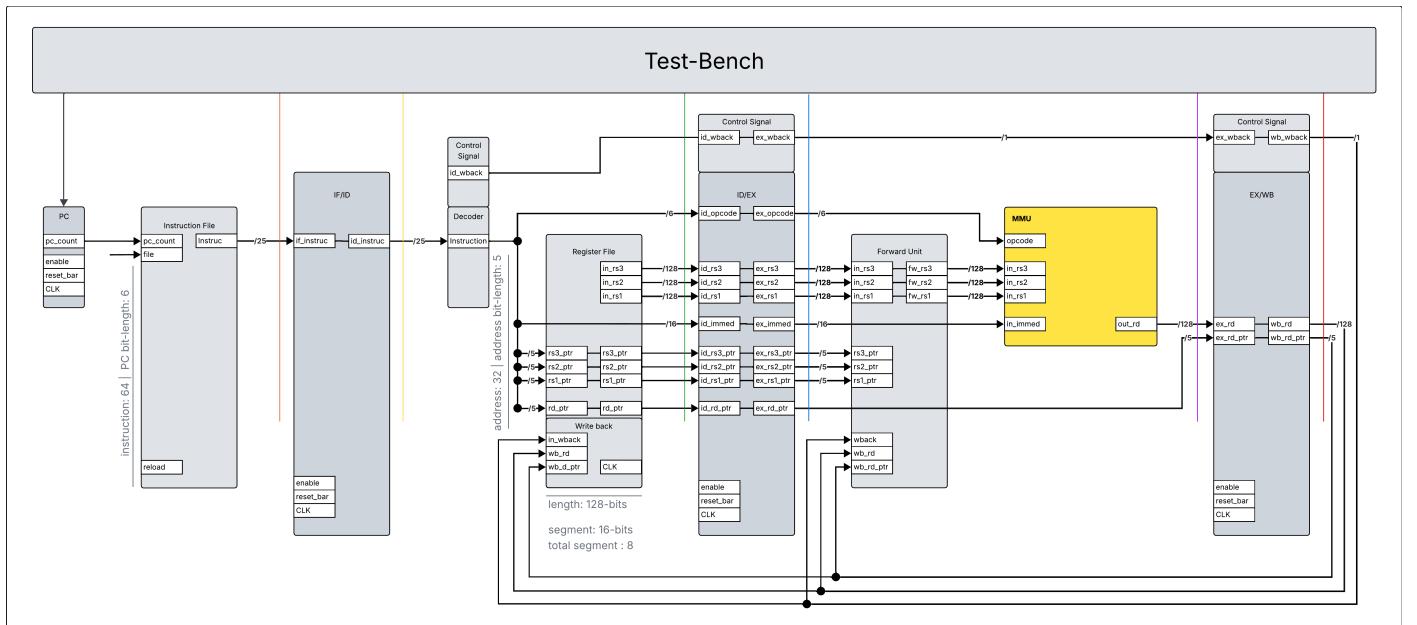


Figure 1: The RTL diagram highlight the complete 4-stage pipeline design of the Multimedia ALU.

A behavioral model approach is used to design the test benches for each module through all stages of the pipeline. 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 outputs and the expected result ensure accurate functional verification of the instruction sets through all stages of the pipeline.

## 2 Stage 1: Instruction Fetch

The instruction fetch stage consists of two units, the Program Counter, PC and Instruction File. The PC is driven by the test-bench to cycle through a counter on a rising edge of the clock. The generated `pc_count` corresponds to the address in the Instruction file, which fetches the 25-bit line of instruction, `Instruc` as the output and routes into IF/ID register as `if_instruc`.

### 2.1 Program Counter

The program counter is a simple synchronous counter that increments on each rising edge of the clock when enabled, otherwise it's zero. The PC value starts counting from zero on the first enabled rising edge cycle. A zero count corresponds to the zero-th address of the Instruction File or the first instruction. When the counter reaches the specified design capacity with a max count of 63 or 64-th instruction, then the counter is stuck at the max counter of 63 and does not overflow back to zero, preventing a loop-back behavior.

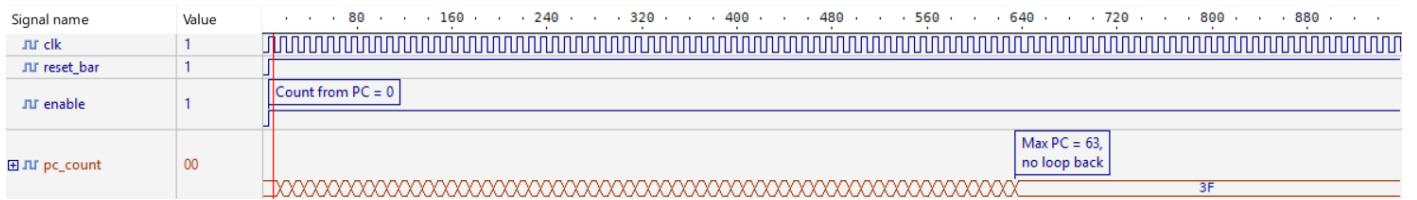


Figure 2: PC starting at count zero to a Max count of 63 or 0x3F

However, since the counter will be stuck at the max count of 63, the last instruction in the file will be continuously fetched. A work-around is to have a 7-bit counter to allow the PC to fetch garbage past the end of file, which is not safe. The better alternative is to sacrifice the last line of instruction file to always be a NOP instruction in order to not corrupt the pipeline with garbage instructions.

White Space Instruction:

24	23	21 20	5 4	0
0	0000	0000000000000000	00000	

In the case of an instruction file containing less than 63 lines of instruction without any NOP fillers at the end, the PC faces another limitation. There is no hardware mechanism to detect the actual end of a short program less than the specified max count. As a result, the program counter continues incrementing until it reaches max count, causing the PC to fetch beyond the last real instruction. The whitespace after the last real instruction is interpreted as a 25-bit zero, which in this design is a valid load-immediate 16-bit all-zero instruction to register zero at index zero. Therefore, to avoid potential whitespace instruction corruption with the zero-th register reserved as the zero register. It is still possible to load values into the zero-th register, just not recommended in case of whitespace instruction.

## 2.2 Instruction File

The instruction file is implemented using one large vector with a size of 25 by 64 or a total of 1600-bits for 64 instructions with each instruction taking 25-bits.

```

1 entity instruction_file is
2   port(
3     pc_count : in std_logic_vector(COUNTER_LENGTH-1 downto 0);
4     in_file : in std_logic_vector(FILE_SIZE-1 downto 0);
5     reload_bar : in std_logic;
6     instruc : out std_logic_vector(INSTRUCTION_LENGTH-1 downto 0) := (others => '-')
7   );
8 end entity;
9
10 architecture behavior of instruction_file is
11   signal INSTRUC_FILE : std_logic_vector(FILE_SIZE-1 downto 0) := (others => '0');
12 begin
13 ...
14 end architecture;

```

The testbench will load a pre-determined set of instructions stored in `instruction_file.txt`, which will be read through a 1600-bit `in_file` input and store in a `INSTRUC_FILE` vector in little endian approach, where the least significant bits is stored at the lowest memory address.

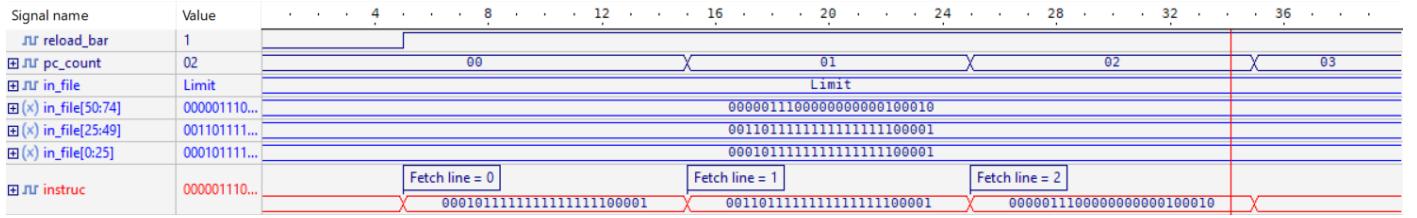


Figure 3: Instruction fetch from the Instruction File with PC

## 3 Stage 2: Instruction Decode

### 3.1 Decoder

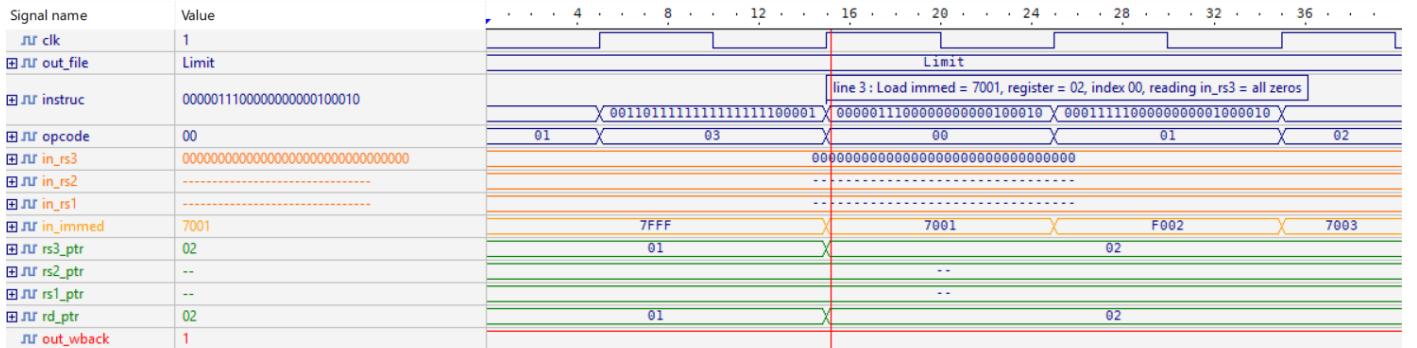


Figure 4: Decoded Instruction and generated wbback control signal

## 3.2 Register File

The Register file is implemented using one large vector with a size of 128 by 32 or a total of 4096-bits for 32 register address with each register store up to 128-bit of data.

```
1 entity register_file is
2 port(
3   clk      : in std_logic;
4   instruc  : in std_logic_vector(INSTRUCTION_LENGTH-1 downto 0);
5
6   wb_rd    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
7   wb_rd_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0); --for forwarding address
8   comparision
9   in_wback : in std_logic;
10
11  out_file  : out std_logic_vector(REGISTER_SIZE-1 downto 0);
12  opcode     : out std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14  in_rs3    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
15  in_rs2    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
16  in_rs1    : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
17
18  in_immed  : out std_logic_vector(IMMEDIATE_LENGTH-1 downto 0) ;
19
20  rs3_ptr   : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
21  rs2_ptr   : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22  rs1_ptr   : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23  rd_ptr    : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
24  out_wback : out std_logic := '0'
25 );
26
27 architecture behavior of register_file is
28   signal REG_FILE : std_logic_vector(REGISTER_SIZE-1 downto 0) := (others => '0');
29 begin
30 ...
31 end architecture;
```

The testbench will read the REG\_FILE vector through the out\_file line to store the value in a `register_file.txt`.

## 4 Stage 3: Execution

The execution stage occur on the next clock cycle of ID/EX register with two unit, the forward unit and the Multimedia Arithmetic Logic Unit, MMU.

The three 128-bit register data value (`ex_rs3`, `ex_rs2`, and `ex_rs1`) is read from the Register file, corresponding to the values of three 5-bit register address pointer (`ex_rs3_ptr`, `ex_rs2_ptr`, and `ex_rs1_ptr`). Both the register data value and register pointer along with write back signal (`wb_wback`, `wb_rd`, and `wb_rd_ptr`) from the previous instructions are routed to the forward unit to compare address for forwarded value (`fw_rs3`, `fw_rs2`, and `fw_rs1`) to be routed to MMU.

The opcode and a 16-bit immediate from the instruction decoder is clocked as `ex_opcode` and `ex_immed` respectively and are both directly routed to the MMU, bypassing the forwarding unit.

The `ex_wback` and `ex_rd_ptr` from instruction decoder and not modified in the MMU. Therefore the `ex_wback` control signal and `ex_rd_ptr` desintination address signal clocked

after ID/EX can be directly routed to the corresponding EX/WB register in the pipeline.

## 4.1 Multimedia Arithmetic Logic Unit, MMU

The Multimedia Arithmetic Logic Unit, MMU is design with a careful selection of input and output signals to 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 with the forwarding unit is shown as follow:

```

1  entity mmu is
2    port (
3      --inputs
4      opcode : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
5
6      in_rs3 : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
7      in_rs2 : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
8      in_rs1 : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
9      in_immed : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
10
11     --forwarding
12     rs3_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
13     rs2_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
14     rs1_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
15
16     wb_rd : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
17     wb_rd_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
18
19     in_wback : in std_logic;
20
21     --outputs
22     out_rd : out std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
23   );
24 end entity;

```

**opcode:** The 25-bits instruction is parsed and reduce to 6-bits as the ex\_opcode or simply 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\_rs3; in\_rs2; in\_rs1, wb\_rd:** Three 128-bits inputs: in\_rs3, in\_rs2, and in\_rs1 from the register file to read 3 full row of register file as sepctified by the design description. When forwarding occur, wb\_rd is in place of one of the three values to be fw\_rs3, fw\_rs2, or fw\_rs1. If no forwarding occur, the value of fw\_rs3 = in\_rs3, fw\_rs2 = in\_rs2, and fw\_rs1 = in\_rs1.

**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.

**rs3\_ptr; rs2\_ptr; rs1\_ptr; wb\_rd\_ptr:** The 5-bits register address pointer is used for comparison in the forwarding unit. If any of the rs3\_ptr, rs2\_ptr, rs1\_ptr register address match with the wb\_rd\_ptr from the previous instruction a forwarding will occur. The 128-bits values, wb\_rd corresponding to wb\_rd\_ptr will be routed to the MMU, replacing the value corresponding to one of the three address. Forwarding does not write-back to the register file.

**out\_rd:** 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.

**in\_wback:** The 1-bit wb\_wback control signal to indicate the previous executed instruction should be written back to the register file and more importantly signal to check if forwarding should occur. Not to be confused with ex\_wback which is the write back control signal of the current instruction.

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 is
2  begin
3    main : process(
4      opcode,
5      in_rs3, in_rs2, in_rs1, in_immed,
6      rs3_ptr, rs2_ptr, rs1_ptr,
7      wb_rd, wb_rd_ptr, in_wback)
8
9      variable temp_out_rd : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
10     variable fw_rs3 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
11     variable fw_rs2 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
12     variable fw_rs1 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
13   begin
14     -- no forward pre-set
15     fw_rs3 := in_rs3;
16     fw_rs2 := in_rs2;
17     fw_rs1 := in_rs1;
18
19     -- Apply forwarding if needed
20     if in_wback = '1' then
21       if wb_rd_ptr = rs3_ptr then fw_rs3 := wb_rd; end if;
22       if wb_rd_ptr = rs2_ptr then fw_rs2 := wb_rd; end if;
23       if wb_rd_ptr = rs1_ptr then fw_rs1 := wb_rd; end if;
24     end if;
25
26     case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
27       when "00" | "01" =>
28         LDI_memory(      --ref. procedure_package/load_immediate.vhd
29           opcode,
30           fw_rs3,
31           in_immed,
32
33           temp_out_rd
34         );
35
36       when "10" =>
37         STM_main(        --ref. procedure_package/saturate_math.vhd
38           opcode,

```

```

39      fw_rs3,
40      fw_rs2,
41      fw_rs1,
42
43      temp_out_rd
44  );
45
46  when "11" =>
47    RSI_main(
48      opcode,
49      fw_rs2,
50      fw_rs1,
51      in_immed,
52
53      temp_out_rd
54  );
55
56  when others =>
57    temp_out_rd := (others => '-');
58 end case;
59 out_rd <= temp_out_rd;
60 end process;
61 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.

## 5 Stage 4: Write-Back

### 5.1 Write-Back to Register

```

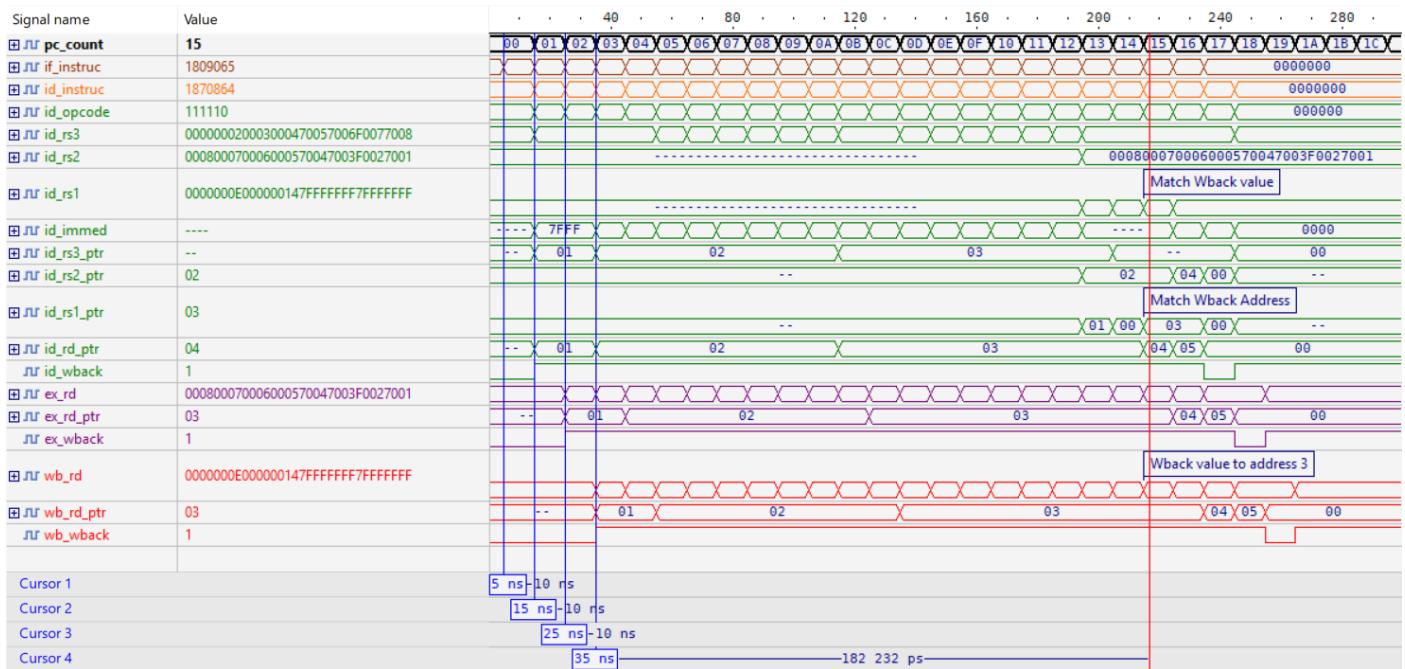
1  architecture behavior of register_file is
2    signal REG_FILE : std_logic_vector(REGISTER_SIZE-1 downto 0) := (others => '0');
3 begin
4   register_file : process(instruc, in_wback)
5     variable var_rs3_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0) := (others => '-');
6     variable var_rs2_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0) := (others => '-');
7     variable var_rs1_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0) := (others => '-');
8
9     variable var_rs3 : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
10    variable var_rs2 : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
11    variable var_rs1 : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
12    variable read_select : std_logic_vector(2 downto 0) := (others => '0');
13 begin
14   -----
15   -- 1. Decode instruction
16   -----
17   ...
18   -----
19   -- Read register values
20   -----
21   if read_select(2) = '1' then
22     var_rs3 := REG_FILE(
23       (to_integer(unsigned(var_rs3_ptr))+1)*(REGISTER_LENGTH)-1
24       downto
25       to_integer(unsigned(var_rs3_ptr))*(REGISTER_LENGTH));
26   end if;
27   if read_select(1) = '1' then
28     var_rs2 := REG_FILE(
29       (to_integer(unsigned(var_rs2_ptr))+1)*(REGISTER_LENGTH)-1
30       downto
31       to_integer(unsigned(var_rs2_ptr))*(REGISTER_LENGTH));
32   end if;
33   if read_select(0) = '1' then
34     var_rs1 := REG_FILE(
35       (to_integer(unsigned(var_rs1_ptr))+1)*(REGISTER_LENGTH)-1

```

```

36         downto
37             to_integer(unsigned(var_rs1_ptr))*(REGISTER_LENGTH));
38         end if;
39
40         -----
41         -- FORWARDING (COMBINATIONAL)
42         -----
43         if in_wback = '1' then
44             if wb_rd_ptr = var_rs3_ptr then
45                 var_rs3 := wb_rd;
46             end if;
47             if wb_rd_ptr = var_rs2_ptr then
48                 var_rs2 := wb_rd;
49             end if;
50             if wb_rd_ptr = var_rs1_ptr then
51                 var_rs1 := wb_rd;
52             end if;
53             REG_FILE(
54                 (to_integer(unsigned(wb_rd_ptr))+1)*REGISTER_LENGTH - 1 downto
55                 to_integer(unsigned(wb_rd_ptr))*REGISTER_LENGTH) <= wb_rd;
56         end if;
57
58         -----
59         -- Drive outputs
60         -----
61         in_rs3 <= var_rs3;
62         in_rs2 <= var_rs2;
63         in_rs1 <= var_rs1;
64
65         rs3_ptr <= var_rs3_ptr;
66         rs2_ptr <= var_rs2_ptr;
67         rs1_ptr <= var_rs1_ptr;
68
69         out_file <= REG_FILE;
70
71     end process;
72 end architecture;

```

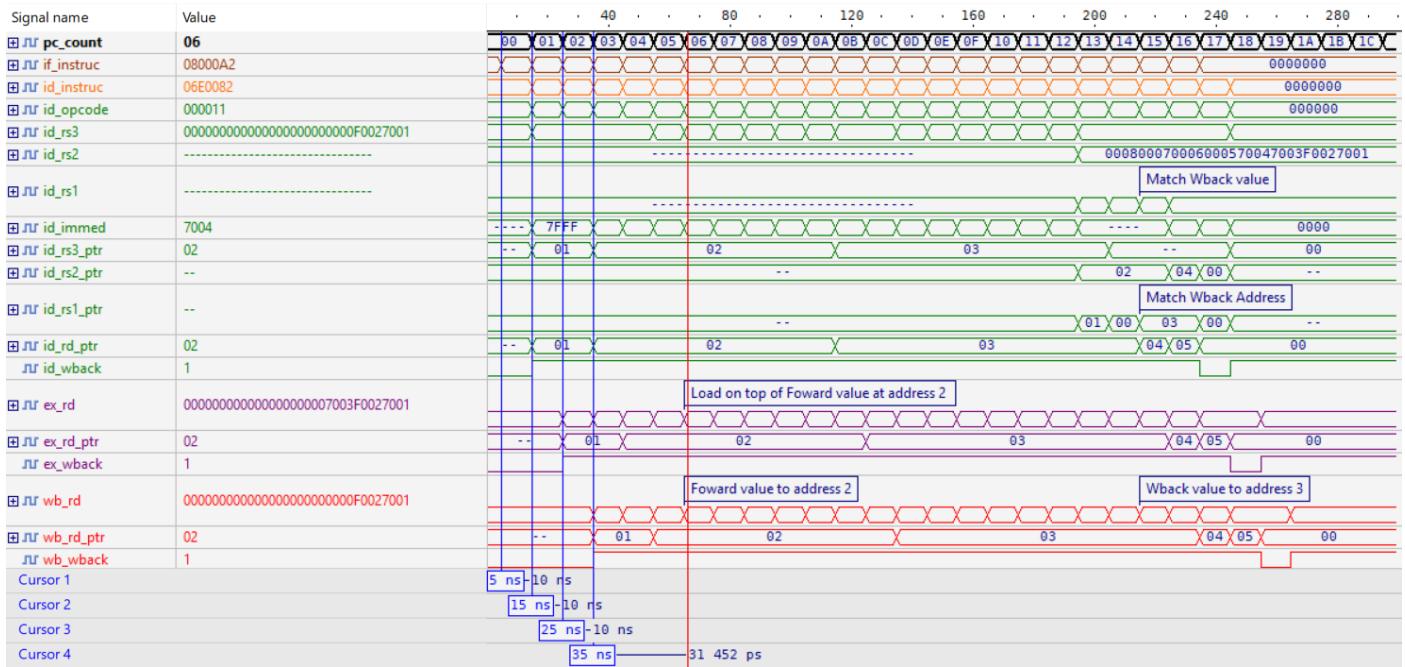


## 5.2 Forward to ALU

```

1 architecture behavior of mmu is
2 begin
3   main : process(
4     opcode,
5     in_rs3, in_rs2, in_rs1, in_immed,
6     rs3_ptr, rs2_ptr, rs1_ptr,
7     wb_rd, wb_rd_ptr, in_wback)
8
9   variable temp_out_rd : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
10  variable fw_rs3 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
11  variable fw_rs2 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
12  variable fw_rs1 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
13 begin
14   -- no forward pre-set
15   fw_rs3 := in_rs3;
16   fw_rs2 := in_rs2;
17   fw_rs1 := in_rs1;
18
19   -- Apply forwarding if needed
20   if in_wback = '1' then
21     if wb_rd_ptr = rs3_ptr then fw_rs3 := wb_rd; end if;
22     if wb_rd_ptr = rs2_ptr then fw_rs2 := wb_rd; end if;
23     if wb_rd_ptr = rs1_ptr then fw_rs1 := wb_rd; end if;
24   end if;
25   ...
26 end process;
27 end architecture;

```



## 6 Result: Register File



r31



## 7 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:

```

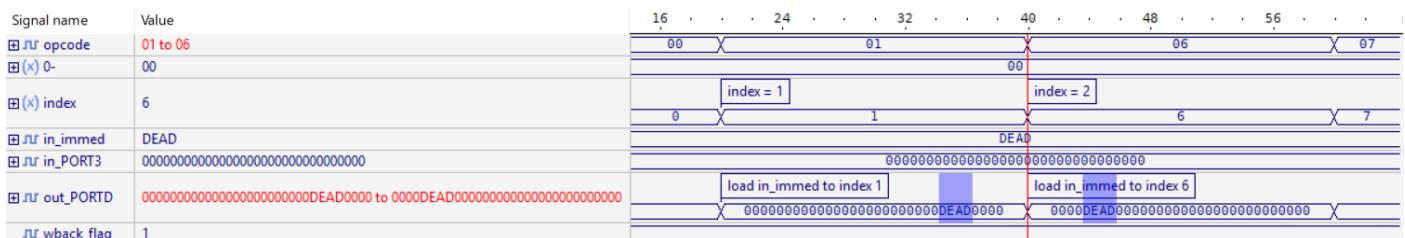
in_PORT3 ← FILE[rd]
in_PORT3[index] ← immediate
out_portD[31-fields] ← in_PORT3
FILE[rd] ← out_PORTD
wback_flag ← 1

```

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 hexdecimal. The in\_immed = xDEAD is loaded based on the index[0:7] of out\_portD. Each index section is allocated 16-bits or 4 hexdecimal 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret32 ← SignExt(in_PORT3[16-low]) × SignExt(in_PORT2[16-low])

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

FILE[rd] ← out_PORTD

wback_flag ← 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
ju opcode	20		20				
ju(x) 10	0		0				
ju(x) function	DEAD		DEAD				
ju in_immed	000100020003000470057006F0077008		000100020003000470057006F0077008				
ju in_PORT3	000100020003000470057006F0077008		2		2		
ju(x) 16-low[111:96]	2		4		4		
ju(x) 16-low[79:64]	4			28678		28678	
ju(x) 16-low[47:32]	28678			28680		28680	
ju(x) 16-low[15:0]	28680			000800070006000570047003F0027001		000800070006000570047003F0027001	
ju in_PORT2	000800070006000570047003F0027001		7		7		
ju(x) 16-low[111:96]	7		5		5		
ju(x) 16-low[79:64]	5			28675		28675	
ju(x) 16-low[47:32]	28675			28673		28673	
ju(x) 16-low[15:0]	28673			00000000000000007FFF00007FFF0000		00000000000000007FFF00007FFF0000	
ju in_PORT1	00000000000000007FFF00007FFF0000		0		0		
ju(x) 32-field[127:96]	0			2147418112		2147418112	
ju(x) 32-field[95:64]	0			2147418112		2147418112	
ju(x) 32-field[63:32]	2147418112			0000000E000000147FFFFFF7FFFFFF		0000000E000000147FFFFFF7FFFFFF	
ju(x) 32-field[31:0]	2147418112			14		14	
ju out_PORTD	0000000E000000147FFFFFF7FFFFFF			20		20	
ju(x) 32-field[127:96]	14			7FFFFFFF		7FFFFFFF	
ju(x) 32-field[95:64]	20			7FFFFFFF		7FFFFFFF	
ju(x) 32-field[63:32]	7FFFFFFF						
ju(x) 32-field[31:0]	7FFFFFFF						
ju 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 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret32 ← SignExt(in_PORT3[16-high]) × SignExt(in_PORT2[16-high])

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

FILE[rd] ← out_PORTD

wback_flag ← 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
rf opcode	21	X			21		
rf 10	10	0	X		1		
rf function	1				DEAD		
rf in_immed	DEAD					000100020003000470057006F0077008	
rf in_PORT3	000100020003000470057006F0077008						
rf 16-low[127:112]	1				1		
rf 16-low[95:80]	3				3		
rf 16-low[63:48]	28677				28677		
rf 16-low[31:16]	-4089				-4089		
rf in_PORT2	000800070006000570047003F0027001				000800070006000570047003F0027001		
rf 16-low[127:112]	8				8		
rf 16-low[95:80]	6				6		
rf 16-low[63:48]	28676				28676		
rf 16-low[31:16]	-4094				-4094		
rf in_PORT1	00000000000000000000000000000000				00000000000000000000000000000000		
rf 32-field[127:96]	0				0		
rf 32-field[95:64]	0				0		
rf 32-field[63:32]	2147418112				2147418112		
rf 32-field[31:0]	2147418112				2147418112		
rf out_PORTD	00000008000000127FFFFFF7FFFFFF	X			00000008000000127FFFFFF7FFFFFF		
rf 32-field[127:96]	8				8		
rf 32-field[95:64]	18				18		
rf 32-field[63:32]	7FFFFFFF				7FFFFFFF		
rf 32-field[31:0]	7FFFFFFF				7FFFFFFF		
rf 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret32 ← SignExt(in_PORT3[16-low]) × SignExt(in_PORT2[16-low])

out_portD[31-fields] ← Saturate32(in_PORT1[32-field] – ret32)

FILE[rd] ← out_PORTD

wback_flag ← 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
JU opcode	22	21	X	22			
JU 10	10	1	X	2			
JU function	2			DEAD			
JU in_immed	DEAD			000100020003000470057006F0077008			
JU in_PORT3	000100020003000470057006F0077008			1			
JU 16-low[127:112]	1			3			
JU 16-low[95:80]	3			28677			
JU 16-low[63:47]	28677			-4089			
JU 16-low[31:16]	-4089				000800070006000570047003F0027001		
JU in_PORT2	000800070006000570047003F0027001			8			
JU 16-low[127:112]	8			6			
JU 16-low[95:80]	6			28676			
JU 16-low[63:47]	28676			-4094			
JU 16-low[31:16]	-4094				00000000000000000000000000000000		
JU in_PORT1	00000000000000000000000000000000	X		0			
JU 32-field[127:96]	0			0			
JU 32-field[95:64]	0						
JU 32-field[63:32]	-2147418112		X	-2147418112			
JU 32-field[31:0]	-2147418112		X	-2147418112			
JU out_PORTD	FFFFFFFFFFE8000000000000000	X		FFFFFFFFFFE8000000000000000			
JU 32-field[127:96]	-14			8	X	-14	
JU 32-field[95:64]	-20			18	X	-20	
JU 32-field[63:32]	80000000			X		80000000	
JU 32-field[31:0]	80000000			X		80000000	
JU 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret32 ← SignExt(in_PORT3[16-high]) × SignExt(in_PORT2[16-high])

out_portD[31-fields] ← Saturate32(in_PORT1[32-field] – ret32)

FILE[rd] ← out_PORTD

wback_flag ← 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 decimal values.

Signal name	Value	60	61	62	63	64	65	66	67	68	69	70	71
ru in_immed	DEAD							DEAD					
ru 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					
ru 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					
ru in_PORT1	00000000000000000000000000000000							00000000000000000000000000000000					
(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					
ru out_PORTD	FFFFFFFFFF8FFFFFE800000080000000							FFFFFFFFFF8FFFFFE800000080000000					
(x) 32-field[127:96]	-8							-8					
(x) 32-field[95:64]	-18							-18					
(x) 32-field[63:32]	80000000							80000000					
(x) 32-field[31:0]	80000000							80000000					
ru wback_flag	1							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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret64 ← SignExt(in_PORT3[32-low]) × SignExt(in_PORT2[32-low])

out_portD[31-fields] ← Saturate32(in_PORT1[64-field] + ret64)

FILE[rd] ← out_PORTD

wback_flag ← 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
JU opcode	24	23 X		24			
JU(10)							
JU(x) function	4	3 X		4			
JU in_immed	DEAD			DEAD			
JU in_PORT3	000100020003000470057006F0077008			000100020003000470057006F0077008			
JU(x) 32-field[95:64]	196612			196612			
JU(x) 32-field[31:0]	-267948024			-267948024			
JU in_PORT2	000800070006000570047003F0027001			000800070006000570047003F0027001			
JU(x) 32-field[95:64]	393221			393221			
JU(x) 32-field[31:0]	-268275711			-268275711			
JU in_PORT1	00000000000000007FFF00007FFF0000	X		00000000000000007FFF00007FFF0000			
JU(x) 64-field[127:64]	0			0			
JU(x) 64-field[63:0]	9223090564025483264	X		9223090564025483264			
JU out_PORTD	00000012002700147FFFFFFFFFFFF	X		00000012002700147FFFFFFFFFFFF			
JU(x) 64-field[127:64]	77311967252	X		77311967252			
JU(x) 64-field[63:0]	7FFFFFFFFFFFFF		Signed Saturated		7FFFFFFFFFFFFF		
JU 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 = 0x7FFFFFFFFFFFFF, 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret64 ← SignExt(in_PORT3[32-high]) × SignExt(in_PORT2[32-high])

out_portD[31-fields] ← Saturate32(in_PORT1[64-field] + ret64)

FILE[rd] ← out_PORTD

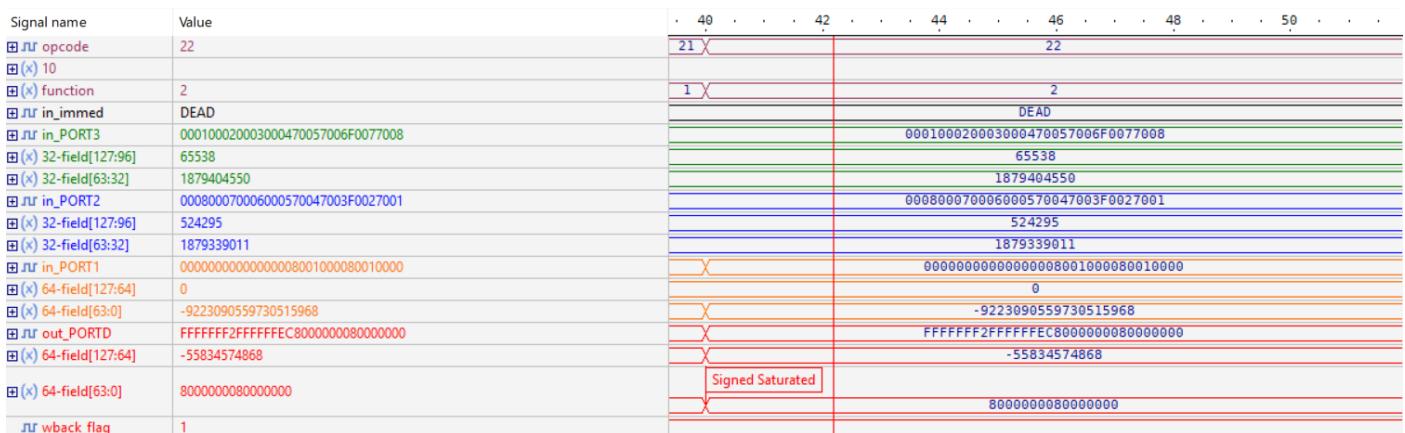
wback_flag ← 1

```

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.



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 = 0x7FFFFFFFFFFFFF, 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret64 ← SignExt(in_PORT3[32-low]) × SignExt(in_PORT2[32-low])

out_portD[31-fields] ← Saturate32(in_PORT1[64-field] – ret64)

FILE[rd] ← out_PORTD

wback_flag ← 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
opcode	26	25	X		26		
(x) 10			5	X		6	
(x) function	6					DEAD	
JU in_immed	DEAD					000100020003000470057006F0077008	
JU in_PORT3	000100020003000470057006F0077008					196612	
(x) 32-field[95:64]	196612					-267948024	
(x) 32-field[31:0]	-267948024					000800070006000570047003F0027001	
JU in_PORT2	000800070006000570047003F0027001					393221	
(x) 32-field[95:64]	393221					-268275711	
(x) 32-field[31:0]	-268275711					00000000000000000000000000000000	
JU in_PORT1	00000000000000000000000000000000	X				00000000000000000000000000000000	
(x) 64-field[127:64]	0					0	
(x) 64-field[63:0]	-9223090559730515968	X				-9223090559730515968	
JU out_PORTD	FFFFFEDFFD8FFEC80000000000000000	X				FFFFFEDFFD8FFEC8000000000000000	
(x) 64-field[127:64]	-77311967252	X				-77311967252	
(x) 64-field[63:0]	8000000000000000		Signed Saturated			8000000000000000	
JU 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 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
in_PORT3 ← FILE[rs3]

ret64 ← SignExt(in_PORT3[32-high]) × SignExt(in_PORT2[32-high])

out_portD[64-fields] ← Saturate(in_PORT1[64-field] – ret64)

FILE[rd] ← out_PORTD

wback_flag ← 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
opcode	27	26	X	27			
(x) 10		6	X	7			
(x) function	7			DEAD			
JU in_immed	DEAD						
JU in_PORT3	000100020003000470057006F0077008			000100020003000470057006F0077008			
(x) 32-field[127:96]	65538			65538			
(x) 32-field[63:32]	1879404550			1879404550			
JU in_PORT2	000800070006000570047003F0027001			000800070006000570047003F0027001			
(x) 32-field[127:96]	524295			524295			
(x) 32-field[63:32]	1879339011			1879339011			
JU in_PORT1	00000000000000000000000000000000			00000000000000000000000000000000			
(x) 64-field[127:64]	0			0			
(x) 64-field[63:0]	-9223090559730515968			-9223090559730515968			
JU out_PORTD	FFFFFFFFFFE8FFF280000000000000000		X	FFFFFFFFFFE8FFF280000000000000000			
(x) 64-field[127:64]	-34361245710		X	-34361245710			
(x) 64-field[63:0]	8000000000000000			8000000000000000			
JU 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:

```

in_PORT1 ← FILE[rs1]
in_immed ← rs2
out_portD[16-low] ← Logical(in_PORT1[16-low] >> in_immed)
FILE[rd] ← out_PORTD,
wback_flag ← 1

```

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
✉ JLR opcode	31			31			
11				11			
✉ (x) function	1			1			
✉ JLR in_immed	0004			0004			
✉ JLR in_PORT1	700870077006700570047003F0027001			700870077006700570047003F0027001			
✉ (x) 16-low[11:96]	01110000000000111			01110000000000111			
✉ (x) 16-low[79:64]	01110000000000101			01110000000000101			
✉ (x) 16-low[47:32]	01110000000000011			01110000000000011			
✉ (x) 16-low[15:0]	01110000000000001			01110000000000001			
✉ JLR in_PORT2	.....			.....			
✉ JLR out_PORTD	07000700070007000700070007000F000700			07000700070007000700070007000F000700			
✉ (x) 16-low[111:96]	0000011100000000			0000011100000000			
✉ (x) 16-low[79:64]	0000011100000000			0000011100000000			
✉ (x) 16-low[47:32]	0000011100000000			0000011100000000			
✉ (x) 16-low[15:0]	0000011100000000			0000011100000000			
JLR 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:

$$\text{in\_PORT1} \leftarrow \text{FILE}[\text{rs1}]$$

$$\text{in\_PORT2} \leftarrow \text{FILE}[\text{rs2}]$$

$$\text{out\_PORTD[32-fields]} \leftarrow \text{Unsigned}(\text{in\_PORT1[32-fields]} + \text{in\_PORT2[32-fields]})$$

$$\text{FILE}[rd] \leftarrow \text{out\_PORTD},$$

$$\text{wback\_flag} \leftarrow 1$$

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	21	22	23	24	25	26	27	28	29	30
JU opcode	32	31	X					32				
JU 11							11					
JU(x) function	2		1	X				2				
JU in_immed	0004						0004					
JU in_PORT1	700870077006700570047003F0027001						700870077006700570047003F0027001					
JU(x) 32-field[127:96]	1879601159						1879601159					
JU(x) 32-field[95:64]	1879470085						1879470085					
JU(x) 32-field[63:32]	1879339011						1879339011					
JU(x) 32-field[31:0]	4026691585						4026691585					
JU in_PORT2	80010004800100037FFF00027FFF0001						80010004800100037FFF00027FFF0001					
JU(x) 32-field[127:96]	2147549188						2147549188					
JU(x) 32-field[95:64]	2147549187						2147549187					
JU(x) 32-field[63:32]	2147418114						2147418114					
JU(x) 32-field[31:0]	2147418113						2147418113					
JU out_PORTD	F009700BF0077008F0037005FFFFFFF						F009700BF0077008F0037005FFFFFFF					
JU(x) 32-field[127:96]	4027150347						4027150347					
JU(x) 32-field[95:64]	4027019272						4027019272					
JU(x) 32-field[63:32]	4026757125						4026757125					
JU(x) 32-field[31:0]	FFFFFFFFFF						Max = FFFFFFFF					
JU wback_flag	1							FFFFFFFFFF				

$$\text{out\_PORTD[32-fields]} \leftarrow \text{Unsigned}(\text{in\_PORT1[32-fields]} + \text{in\_PORT2[32-fields]})$$

$$\text{out\_PORTD[127:96]} \leftarrow \text{Unsigned}(\text{in\_PORT1[127:96]} + \text{in\_PORT2[127:96]})$$

$$4027150347 = 1879601159 + 2147549188$$

$$\text{out\_PORTD[31:0]} \leftarrow \text{Unsigned}(\text{in\_PORT1[31:0]} + \text{in\_PORT2[31:0]})$$

$$4294967295 < 4026691585 + 2147418113$$

## 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:

```

in_PORT1 ← FILE[rs1]
out_PORTD[16-low] ← Count1s(in_PORT1[16-low])
FILE[rd] ← out_PORTD,
wback_flag ← 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	40	40.064	40.128	40.192	40.256
opcode	33	32 X		33		
11				11		
(x) function	3	2 X		3		
in_immed	4			4		
in_PORT1	700870077006700570047003F0027001			700870077006700570047003F0027001		
(x) 16-low[111:96]	0111000000000111			0111000000001111		
(x) 16-low[79:64]	011100000000101			011100000000101		
(x) 16-low[47:32]	011100000000011			011100000000011		
(x) 16-low[15:0]	0111000000000001			0111000000000001		
in_PORT2	-----	X				
out_PORTD	00040006000500050004000500050004	X		00040006000500050004000500050004		
(x) 16-low[111:96]	6	X		6		
(x) 16-low[79:64]	5	X		5		
(x) 16-low[47:32]	5	X		5		
(x) 16-low[15:0]	4	-1 X		4		
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:

$$\text{in\_PORT1} \leftarrow \text{FILE}[rs1]$$

$$\text{in\_PORT2} \leftarrow \text{FILE}[rs2]$$

$$\text{out\_PORTD[16-low]} \leftarrow \text{SignedSaturate}(\text{in\_PORT2[16-low]} + \text{in\_PORT1[16-low]})$$

$$\text{FILE}[rd] \leftarrow \text{out\_PORTD},$$

$$\text{wback\_flag} \leftarrow 1$$

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	61	62	63	64	65	66	67	68	69	70	71
JU opcode	34	33	X			34							
JU 11						11							
(x) function	4	3	X			4							
JU in_immed	0004					0004							
JU in_PORT1	700870077006800570047003F0027001	X				700870077006800570047003F0027001							
(x) 16-low[111:96]	28679					28679							
(x) 16-low[79:64]	-32763		X			-32763							
(x) 16-low[47:32]	28675					28675							
(x) 16-low[15:0]	28673					28673							
JU in_PORT2	80010004800100037FFF7FF27FFFFFF1	X				80010004800100037FFF7FF27FFFFFF1							
(x) 16-low[111:96]	4	?	X			4							
(x) 16-low[79:64]	3	?	X			3							
(x) 16-low[47:32]	32754	?	X			32754							
(x) 16-low[15:0]	-15	?	X			-15							
JU out_PORTD	F009700BF00780087FFF7FFF70016FF2	X				F009700BF00780087FFF7FFF70016FF2							
(x) 16-low[111:96]	28683					Positive Addition							
(x) 16-low[79:64]	-32760	6	X			28683							
(x) 16-low[47:32]	7FFF	5	X			Negative Addition							
(x) 16-low[15:0]	28658	4	X			-32760							
JU wback_flag	1					Signed Saturated							

$$\text{out\_PORTD[16-low]} \leftarrow \text{Unsigned}(\text{in\_PORT1[16-low]} + \text{in\_PORT2[16-low]})$$

$$\text{out\_PORTD[111:96]} \leftarrow \text{SignedSaturate}(\text{in\_PORT1[111:96]} + \text{in\_PORT2[111:96]})$$

$$28683 = 28679 + 4$$

$$\text{out\_PORTD[31:0]} \leftarrow \text{Unsigned}(\text{in\_PORT1[47:32]} + \text{in\_PORT2[47:32]})$$

$$32767 < 28675 + 32754$$

## OR – Bitwise Logical OR

### Description:

Logical OR the contents of registers rs1 and rs2.

### Instruction:

24	23 22	15 14	10 9	5 4	0
11	00000101	rs2	rs1	rd	

### Opcode:

5	4 3	0
11	0101	

### Operation:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
out_PORTD ← in_PORT2 ∨ in_PORT1
FILE[rd] ← out_PORTD,
wback_flag ← 1

```

### 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	81	82	83	84	85	86	87	88	89	90	91
opcode	35	34	X			35							
11	11					11							
(x) function	5	4	X			5							
in_immed	0004					0004							
in_PORT1	700870077006700570047003F0027001	X				700870077006700570047003F0027001							
in_PORT2	0000000000000000FFFFFFFFFF		X			0000000000000000FFFFFFFFFF							
out_PORTD	7008700770067005FFFFFFFFFF			X		7008700770067005FFFFFFFFFF							
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:

```

in_PORT1 ← FILE[rs1]
out_PORTD[32-fields] ← in_PORT1[127:96]
FILE[rd] ← out_PORTD,
wback_flag ← 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	100	102	104	106	108	110
JR opcode	36	35 X		36			
11				11			
JR function	6	5 X		6			
JR in_immed	0004			0004			
JR in_PORT1	700870077006700570047003F0027001			700870077006700570047003F0027001			
JR in_FIELD[127:96]	1879601159			1879601159			
JR in_PORT2	-----	X	-----	-----	-----	-----	
JR out_PORTD	70087007700870077008700770087007		X	70087007700870077008700770087007			
JR 32-field[127:96]	1879601159			1879601159			
JR 32-field[95:64]	1879601159		X	1879601159			
JR 32-field[63:32]	1879601159		-1 X	1879601159			
JR 32-field[31:0]	1879601159		-1 X	1879601159			
JR 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
out_PORTD[31-fields] ← SignedMin(in_PORT2[32-fields], in_PORT1[32-fields])
FILE[rd] ← out_PORTD,
wback_flag ← 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	121	122	123	124	125	126	127	128	129	130	131
opcode	37	36	X					37					
11								11					
(x) function	7	6	X					7					
in_immed	0004							0004					
in_PORT1	700870077006700570047003F0027001							700870077006700570047003F0027001					
(x) 32-field[127:96]	1879601159							1879601159					
(x) 32-field[95:64]	1879470085							1879470085					
(x) 32-field[63:32]	1879339011							1879339011					
(x) 32-field[31:0]	-268275711							-268275711					
in_PORT2	80010004800100037FFF00027FFF0001							80010004800100037FFF00027FFF0001					
(x) 32-field[127:96]	-2147418108							-2147418108					
(x) 32-field[95:64]	-2147418109							-2147418109					
(x) 32-field[63:32]	2147418114							2147418114					
(x) 32-field[31:0]	2147418113							2147418113					
out_PORTD	70087007700670057FFF00027FFF0001							70087007700670057FFF00027FFF0001					
(x) 32-field[127:96]	1879601159							1879601159					
(x) 32-field[95:64]	1879470085							1879470085					
(x) 32-field[63:32]	2147418114							2147418114					
(x) 32-field[31:0]	2147418113							2147418113					
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:

in\_PORT1  $\leftarrow$  FILE[rs1]

in\_PORT2  $\leftarrow$  FILE[rs2]

out\_PORTD[31-fields]  $\leftarrow$  SignedMin(in\_PORT2[32-fields], in\_PORT1[32-fields])

FILE[rd]  $\leftarrow$  out\_PORTD,

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

$$\text{in\_PORT1} \leftarrow \text{FILE}[rs1]$$

$$\text{in\_PORT2} \leftarrow \text{FILE}[rs2]$$

$$\text{out\_portD[31-fields]} \leftarrow \text{UnsignedMult}(\text{in\_PORT1[16-fields]}, \text{in\_PORT2[16-fields]})$$

$$\text{FILE}[rd] \leftarrow \text{out\_PORTD},$$

$$\text{wback\_flag} \leftarrow 1$$

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
JU opcode	39	38 X		39			
JU 11	11			11			
JU (x) function	9	8 X		9			
JU in_immed	0004			0004			
JU in_PORT1	700870077006700570047003F0027001			700870077006700570047003F0027001			
JU (x) 16-low[111:96]	28679			28679			
JU (x) 16-low[79:64]	28677			28677			
JU (x) 16-low[47:32]	28675			28675			
JU (x) 16-low[15:0]	28673			28673			
JU in_PORT2	80010004800100037FFF7FF27FFFFF1	X		80010004800100037FFF7FF27FFFFF1			
JU (x) 16-low[111:96]	4			4			
JU (x) 16-low[79:64]	3			3			
JU (x) 16-low[47:32]	32754	2 X		32754			
JU (x) 16-low[15:0]	65521		unsigned		65521		
JU out_PORTD	0001C01C0001500F37FB5FD66FFA6FF1	X		0001C01C0001500F37FB5FD66FFA6FF1			
JU (x) 32-field[127:96]	114716	X		114716			
JU (x) 32-field[95:64]	86031	X		86031			
JU (x) 32-field[63:32]	939220950	X		939220950			
JU (x) 32-field[31:0]	1878683633		unsigned product		1878683633		
JU wback_flag	1	X					

## 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:

```

in_PORT1 ← FILE[rs1]
in_immed ← ZeroExtend(rs2)
out_portD[31-fields] ← UnsignedMult(in_PORT1[16-fields], in_immed)
FILE[rd] ← out_PORTD,
wback_flag ← 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
JU opcode	3A	39 X		3A			
JU 11				11			
JU(x) function	A	9 X		A			
JU in_immed	20	decimal 20		20			
JU in_PORT1	80010004800100037FFF00027FFF0001	X		80010004800100037FFF00027FFF0001			
JU(x) 16-low[111:96]	4	X		4			
JU(x) 16-low[79:64]	3	X		3			
JU(x) 16-low[47:32]	2	X		2			
JU(x) 16-low[15:0]	1	X		1			
JU in_PORT2	-----	X		-----			
JU out_PORTD	0000005000000003C000002800000014	X		0000005000000003C000002800000014			
JU(x) 32-field[127:96]	80	X		80			
JU(x) 32-field[95:64]	60	X		60			
JU(x) 32-field[63:32]	40	X		40			
JU(x) 32-field[31:0]	20	X		20			
JU 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:

```

in_PORT1 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
out_PORTD[31-fields] ← in_PORT2 ∧ in_PORT1
FILE[rd] ← out_PORTD,
wback_flag ← 1

```

### 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 . . . .
JR opcode	3B	3B X
11		11
JR function	B	A X B
JR in_immed	20	20
JR in_PORT1	80010004800100037FFF00027FFF0001	80010004800100037FFF00027FFF0001
JR in_PORT2	0000000000000000FFFFFFFFFF	0000000000000000FFFFFFFFFF
JR out_PORTD	00000000000000007FFF00027FFF0001	00000000000000007FFF00027FFF0001
JR wback_flag	1	X

## 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:

$$\text{in\_PORT1} \leftarrow \text{FILE}[rs1]$$

$$\text{out\_portD[31-fields]} \leftarrow \text{CLZ}_{32}(\text{in\_PORT1[32-field]})$$

$$\text{FILE}[rd] \leftarrow \text{out\_PORTD},$$

$$\text{wback\_flag} \leftarrow 1$$

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	220.032	220.096	220.16	220.224
JU opcode	3C	3B X	3C		
JU 11			11		
JU function	C	B X	C		
JU in_immed	20		20		
JU in_PORT1	070870071006700570047003F0027001	X	070870071006700570047003F0027001		
JU (x) 32-field[127:96]	000001100001000011000000000111	X	000001100001000011000000000111		
JU (x) 32-field[95:64]	000100000000110011100000000101	X	000100000000110011100000000101		
JU (x) 32-field[63:32]	011100000000100011100000000011	X	011100000000100011100000000011		
JU (x) 32-field[31:0]	11110000000000100111000000000001	X	11110000000000100111000000000001		
JU in_PORT2	-----	X	-----		
JU out_PORTD	0000000500000003000000100000000	X	0000000500000003000000100000000		
JU (x) 32-field[127:96]	5	0 X	5		
JU (x) 32-field[95:64]	3	0 X	3		
JU (x) 32-field[63:32]	1	X	1		
JU (x) 32-field[31:0]	0	X	0		
JU 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
opcode	3D
11	3C X
(x) function	D
in_immed	20
in_PORT1	80010004800100037FFF00027FFF0001
(x) 32-field[127:96]	10000000000000000000000000000000
(x) 32-field[95:64]	1000000000000000000000000000000011
(x) 32-field[63:32]	01111111111111100000000000000010
(x) 32-field[31:0]	01111111111111100000000000000001
in_PORT2	FFFFF10FFFFFFFFFF00000000100000000
(x) 5-low[100:96]	16
(x) 5-low[68:64]	16
(x) 5-low[36:32]	1
(x) 5-low[4:0]	0
out_PORTD	00048001000380013FFF80017FFF0001
(x) 32-field[127:96]	00000000000000000000000000000001
(x) 32-field[95:64]	00000000000000000000000000000001
(x) 32-field[63:32]	00111111111111100000000000000001
(x) 32-field[31:0]	01111111111111100000000000000001
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 ← FILE[rs1]
in_PORT2 ← FILE[rs2]
out_PORTD[31-fields] ← Unsigned(in_PORT2[32-fields] – in_PORT1[32-fields])
FILE[rd] ← out_PORTD,
wback_flag ← 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	. . . 260.032 . . . 260.096 . . . 260.16 . . . 260.224 . . .
JU opcode	3E	3D X 3E
11	11	11
JU function	E	E
JU in_immed	20	20
JU in_PORT1	700870077006700570047003F0027001	700870077006700570047003F0027001
JU 32-field[127:96]	1879601159	1879601159
JU 32-field[95:64]	1879470085	1879470085
JU 32-field[63:32]	1879339011	1879339011
JU 32-field[31:0]	4026691585	4026691585
JU in_PORT2	80010004800100037FFF00027FFF0001	80010004800100037FFF00027FFF0001
JU 32-field[127:96]	2147549188	2147549188
JU 32-field[95:64]	2147549187	2147549187
JU 32-field[63:32]	2147418114	2147418114
JU 32-field[31:0]	2147418113	2147418113
JU out_PORTD	0FF88FFD0FFA8FFE0FFA8FFF00000000	0FF88FFD0FFA8FFE0FFA8FFF00000000
JU 32-field[127:96]	267948029	267948029
JU 32-field[95:64]	268079102	268079102
JU 32-field[63:32]	268079103	268079103
JU 32-field[31:0]	0	unsigned saturated
JU wback_flag	1	0

## 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:

$in\_PORT1 \leftarrow FILE[rs1]$

$in\_PORT2 \leftarrow FILE[rs2]$

$out\_portD[31\text{-fields}] \leftarrow \text{SignedSaturate}(in\_PORT2[16\text{-fields}] - in\_PORT1[16\text{-fields}])$

$FILE[rd] \leftarrow out\_PORTD,$

$wback\_flag \leftarrow 1$

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
opcode	3F	3E	X	3F		
11	11		11			
(x) function	F	E	X	F		
JU in_immed	20			20		
JU in_PORT1	70080007780600037004FFF3F0027001	X		70080007780600037004FFF3F0027001		
(x) 16-low[111:96]	7	X		7		
(x) 16-low[79:64]	3	X		3		
(x) 16-low[47:32]	-13	X		-13		
(x) 16-low[15:0]	28673			28673		
JU in_PORT2	80010004800100057FFF00067FFF8F01	X		80010004800100057FFF00067FFF8F01		
(x) 16-low[111:96]	4			4		
(x) 16-low[79:64]	5	3	X	5		
(x) 16-low[47:32]	6	2	X	6		
(x) 16-low[15:0]	-28927	1	X	-28927		
JU out_PORTD	8000FFFD800000020FFB00137FFF8000	X		8000FFFD800000020FFB00137FFF8000		
(x) 16-low[111:96]	-3	X		-3		
(x) 16-low[79:64]	2	X		2		
(x) 16-low[47:32]	19				negative subtraction	
(x) 16-low[15:0]	-32768				Signed Saturated	
JU wback_flag	1	0	X			-32768

## NOP – No Operation

### Description:

Instruction does not write anything to the register file.

### Instruction:

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

### Opcode:

5 11	4 3 0000	0
---------	-------------	---

### Operation:

$$\text{wback\_flag} \leftarrow 0$$

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

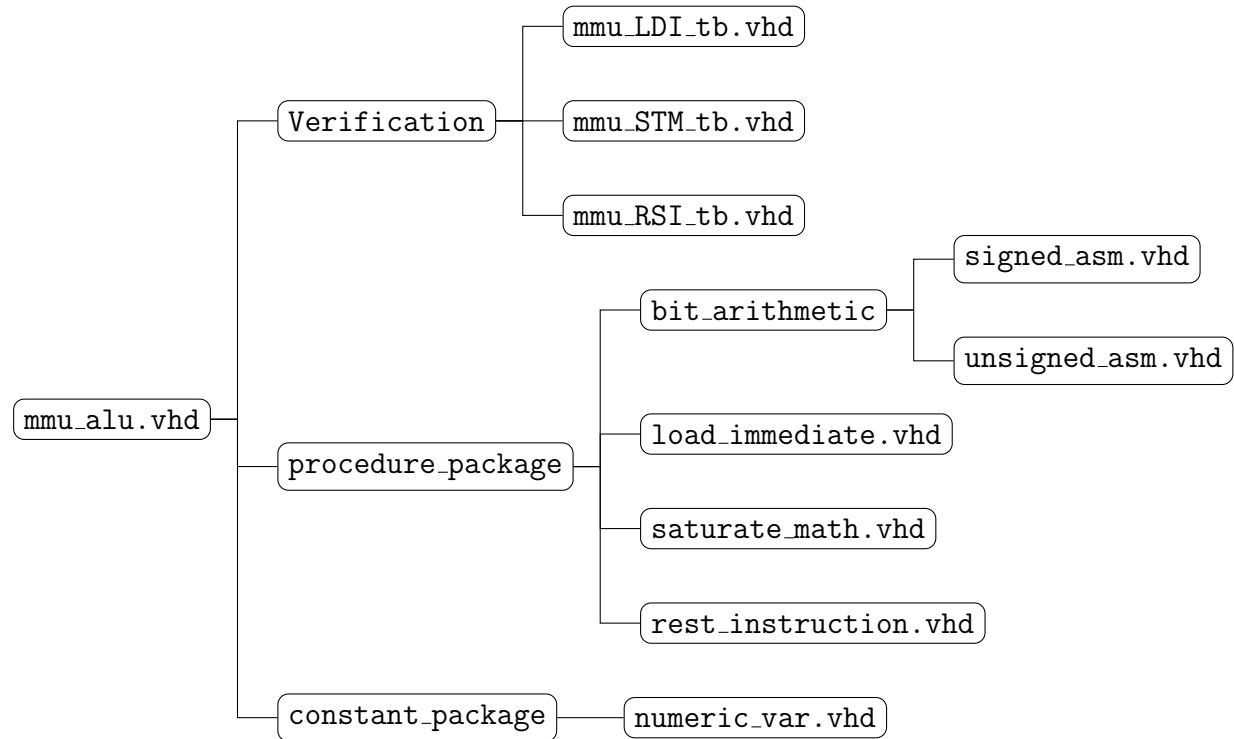
### Verification:

The full 128-bit register values of in\_PORT1, in\_PORT2, and out\_PORTD are represented in 32 hexadecimal values.

Signal name	Value	300	304	308	312	316
JU opcode	3F to 30	3F		30		
JU 11			11			
JU (x) function	0	F		0		
JU in_immed	20			20		
JU in_PORT1	700870077006700570047003F0027001 to .....					
JU in_PORT2	80010004800100037FFF7FF27FFFFF1 to .....					
JU out_PORTD	80008FFD80008FFE0FFB0FEF7FFF8FF0			80008FFD80008FFE0FFB0FEF7FFF8FF0		
JU wback_flag	1 to 0					

# 8 Appendix

## 8.1 File Structure



## 8.2 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 -----PROGRAM_COUNTER_CONSTANT-----
7   constant COUNTER_LENGTH      : integer := 6;
8   constant INCREMENT          : integer := 1;
9
10  constant MAX_COUNT          : integer := 64;
11
12 -----INSTRUCTION_FILE_CONSTANT-----
13  constant INSTRUCTION_LENGTH    : integer := 25;
14  constant FILE_SIZE           : integer := INSTRUCTION_LENGTH * 64; --buffer size of 64, ie
15    64 instruction
16
17  constant IMMEDIATE_LENGTH     : integer := 16;
18  constant INDEX_LENGTH         : integer := 3;
19  constant OPCODE_LENGTH        : integer := 6;
20
21  constant NOP_INSTRUCTION     : std_logic_vector(INSTRUCTION_LENGTH-1 downto 0) := b"
22    11000000000000000000000000000000";
23
24 -----REGISTER_FILE_CONSTANT-----
25  constant REGISTER_LENGTH      : integer := 128;
26  constant REGISTER_SIZE        : integer := REGISTER_LENGTH * 32; --buffer size of 32, ie
27    32 address
28
29  constant VALUE16             : integer := 16;
30  constant ADDRESS_LENGTH       : integer := 5;
31
32 -----DEBUG_CONSTANT-----
33  constant PERIOD              : time := 10ns;
34
35 end package;

```

## 8.3 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 entity mmu is
10  port (
11    --inputs
12    opcode    : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14    in_rs3    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
15    in_rs2    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
16    in_rs1    : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
17    in_immed  : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
18
19    --fowarding
20    rs3_ptr   : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
21    rs2_ptr   : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22    rs1_ptr   : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23
24    wb_rd     : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
25    wb_rd_ptr : in std_logic_vector(ADDRESS_LENGTH-1 downto 0); --for forwarding address
26      comparision
27    in_wback  : in std_logic;
28
29    --outputs
30    out_rd    : out std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');

```

```

30  );
31 end entity;
32
33 architecture behavior of mmu is
34 begin
35   main : process(
36     opcode,
37     in_rs3, in_rs2, in_rs1, in_immed,
38     rs3_ptr, rs2_ptr, rs1_ptr,
39     wb_rd, wb_rd_ptr, in_wback)
40
41   variable temp_out_rd : std_logic_vector(REGISTER_LENGTH-1 downto 0) := (others => '-');
42   variable fw_rs3 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
43   variable fw_rs2 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
44   variable fw_rs1 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
45 begin
46   -- no forward pre-set
47   fw_rs3 := in_rs3;
48   fw_rs2 := in_rs2;
49   fw_rs1 := in_rs1;
50
51   -- Apply forwarding if needed
52   if in_wback = '1' then
53     if wb_rd_ptr = rs3_ptr then fw_rs3 := wb_rd; end if;
54     if wb_rd_ptr = rs2_ptr then fw_rs2 := wb_rd; end if;
55     if wb_rd_ptr = rs1_ptr then fw_rs1 := wb_rd; end if;
56   end if;
57
58   case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
59     when "00" | "01" =>
60       LDI_memory(      --ref. procedure_package/load_immediate.vhd
61         opcode,
62         fw_rs3,
63         in_immed,
64
65         temp_out_rd
66       );
67
68     when "10" =>
69       STM_main(      --ref. procedure_package/saturate_math.vhd
70         opcode,
71         fw_rs3,
72         fw_rs2,
73         fw_rs1,
74
75         temp_out_rd
76       );
77
78     when "11" =>
79       RSI_main(      --ref. procedure_package/rest_instruction.vhd
80         opcode,
81         fw_rs2,
82         fw_rs1,
83         in_immed,
84
85         temp_out_rd
86       );
87
88     when others =>
89       temp_out_rd := (others => '-');
90   end case;
91   out_rd <= temp_out_rd;
92 end process;
93 end architecture;

```

## 8.4 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    in_rs3      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
11    in_immed    : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
12
13   out_rd      : out std_logic_vector(REGISTER_LENGTH-1 downto 0)
14 );
15 end package load_immediate;
16
17 package body load_immediate is
18
19   procedure LDI_memory (
20     signal opcode : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
21     in_rs3      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
22     in_immed    : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
23
24   out_rd      : out std_logic_vector(REGISTER_LENGTH-1 downto 0)
25 ) is
26   variable low_bit    : integer;
27   variable high_bit   : integer;
28   variable temp_out   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
29 begin
30   temp_out := in_rs3;
31   low_bit := to_integer(unsigned(opcode(2 downto 0))) * VALUE16;
32   high_bit := low_bit + VALUE16;
33   temp_out(high_bit-1 downto low_bit) := in_immed;
34
35   out_rd := temp_out;
36 end procedure;
37 end package body load_immediate;

```

## 8.5 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    in_rs3      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
11    in_rs2      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
12    in_rs1      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
13
14   out_rd      : out std_logic_vector(REGISTER_LENGTH-1 downto 0)
15 );
16 end package saturate_math;
17
18 package body saturate_math is
19   procedure STM_main (
20     signal opcode : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
21     in_rs3      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
22     in_rs2      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
23     in_rs1      : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
24
25   out_rd      : out std_logic_vector(REGISTER_LENGTH-1 downto 0)
26 ) is
27   variable ret32    : std_logic_vector(31 downto 0);
28   variable ret64    : std_logic_vector(63 downto 0);
29   variable temp_out : std_logic_vector(REGISTER_LENGTH-1 downto 0);
30 begin
31   case opcode(2 downto 0) is
32
33   when "000" =>
34     for i in 3 downto 0 loop --low 16-bit integer mult-add
35       mult_16(

```

```

36     in_rs3(16*(i*2+1)-1 downto 16*(i*2)),
37     in_rs2(16*(i*2+1)-1 downto 16*(i*2)),
38     ret32
39   );
40   add_32(
41     in_rs1(16*(i*2+2)-1 downto 16*(i*2)),
42     ret32,
43     temp_out(16*(i*2+2)-1 downto 16*(i*2))
44   );
45 end loop;
46
47 when "001" =>
48   for i in 3 downto 0 loop --high 16-bit integer mult-add
49     mult_16(
50       in_rs3(16*(i*2+2)-1 downto 16*(i*2+1)),
51       in_rs2(16*(i*2+2)-1 downto 16*(i*2+1)),
52       ret32
53     );
54     add_32(
55       in_rs1(16*(i*2+2)-1 downto 16*(i*2)),
56       ret32,
57       temp_out(16*(i*2+2)-1 downto 16*(i*2))
58     );
59   end loop;
60
61 when "010" =>
62   for i in 3 downto 0 loop --low 16-bit integer mult-sub
63     mult_16(
64       in_rs3(16*(i*2+1)-1 downto 16*(i*2)),
65       in_rs2(16*(i*2+1)-1 downto 16*(i*2)),
66       ret32
67     );
68     sub_32(
69       in_rs1(16*(i*2+2)-1 downto 16*(i*2)),
70       ret32,
71       temp_out(16*(i*2+2)-1 downto 16*(i*2))
72     );
73   end loop;
74
75 when "011" =>
76   for i in 3 downto 0 loop --high 16-bit integer mult-sub
77     mult_16(
78       in_rs3(16*(i*2+2)-1 downto 16*(i*2+1)),
79       in_rs2(16*(i*2+2)-1 downto 16*(i*2+1)),
80       ret32
81     );
82     sub_32(
83       in_rs1(16*(i*2+2)-1 downto 16*(i*2)),
84       ret32,
85       temp_out(16*(i*2+2)-1 downto 16*(i*2))
86     );
87   end loop;
88
89 when "100" =>
90   for i in 1 downto 0 loop --low 32-bit integer mult-add
91     mult_32(
92       in_rs3(32*(i*2+1)-1 downto 32*(i*2)),
93       in_rs2(32*(i*2+1)-1 downto 32*(i*2)),
94       ret64
95     );
96     add_64(
97       in_rs1(32*(i*2+2)-1 downto 32*(i*2)),
98       ret64,
99       temp_out(32*(i*2+2)-1 downto 32*(i*2))
100    );
101  end loop;
102
103 when "101" =>
104   for i in 1 downto 0 loop --high 32-bit integer mult-add
105     mult_32(
106       in_rs3(32*(i*2+2)-1 downto 32*(i*2+1)),
107       in_rs2(32*(i*2+2)-1 downto 32*(i*2+1)),
108       ret64

```

```

109      );
110      add_64(
111          in_rs1(32*(i*2+2)-1 downto 32*(i*2)),
112          ret64,
113          temp_out(32*(i*2+2)-1 downto 32*(i*2))
114      );
115      end loop;
116
117      when "110" =>
118          for i in 1 downto 0 loop --low 32-bit integer mult-sub
119              mult_32(
120                  in_rs3(32*(i*2+1)-1 downto 32*(i*2)),
121                  in_rs2(32*(i*2+1)-1 downto 32*(i*2)),
122                  ret64
123              );
124              sub_64(
125                  in_rs1(32*(i*2+2)-1 downto 32*(i*2)),
126                  ret64,
127                  temp_out(32*(i*2+2)-1 downto 32*(i*2))
128              );
129          end loop;
130
131      when "111" =>
132          for i in 1 downto 0 loop --high 32-bit integer mult-sub
133              mult_32(
134                  in_rs3(32*(i*2+2)-1 downto 32*(i*2+1)),
135                  in_rs2(32*(i*2+2)-1 downto 32*(i*2+1)),
136                  ret64
137              );
138              sub_64(
139                  in_rs1(32*(i*2+2)-1 downto 32*(i*2)),
140                  ret64,
141                  temp_out(32*(i*2+2)-1 downto 32*(i*2))
142              );
143          end loop;
144
145      when others =>
146          temp_out := (others => '-');
147      end case;
148      out_rd := temp_out;
149  end procedure;
150 end package body saturate_math;

```

## 8.6 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         in_rs2          : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
12         in_rs1          : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
13         in_immed        : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
14
15         out_rd          : out std_logic_vector(REGISTER_LENGTH-1 downto 0)
16     );
17 end package;
18
19 package body rest_instruction is
20     procedure RSI_main(
21         signal opcode    : in std_logic_vector(OPCODE_LENGTH-1 downto 0);
22         in_rs2          : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
23         in_rs1          : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
24         in_immed        : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
25
26         out_rd          : out std_logic_vector(REGISTER_LENGTH-1 downto 0)

```

```

27  ) is
28      variable temp_out    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
29      variable int_var     : integer := 0;
30      variable unsigned16 : unsigned(15 downto 0);
31      variable vector16   : std_logic_vector(15 downto 0);
32      variable vector32   : std_logic_vector(31 downto 0);
33      variable is_nop     : std_logic := '0';
34  begin
35      case opcode(3 downto 0) is
36          when "0000" => --no operation
37              is_nop := '1';
38          when "0001" => --shift right halfword immediate
39              int_var := to_integer(unsigned(in_immed(3 downto 0)));
40              for i in 0 to 7 loop
41                  unsigned16 := unsigned(in_rs1(16*(i+1)-1 downto i*16));
42                  unsigned16 := shift_right(unsigned16, int_var);
43                  temp_out(16*(i+1)-1 downto i*16) := std_logic_vector(unsigned16);
44              end loop;
45
46          when "0010" => --add word unsigned
47              for i in 0 to 3 loop
48                  add_32_unsigned(
49                      in_rs2(32*(i+1)-1 downto i*32),
50                      in_rs1(32*(i+1)-1 downto i*32),
51                      temp_out(32*(i+1)-1 downto i*32)
52                  );
53              end loop;
54
55          when "0011" => --count 1s in halfword
56              for i in 0 to 7 loop
57                  vector16 := in_rs1(16*(i+1)-1 downto i*16);
58                  int_var := 0;
59                  for j in 0 to 15 loop
60                      if vector16(j) = '1' then
61                          int_var := int_var + 1;
62                      end if;
63                  end loop;
64                  temp_out(16*(i+1)-1 downto i*16) := std_logic_vector(to_unsigned(int_var,
65                                         16));
66              end loop;
67
68          when "0100" => --add halfword saturated
69              for i in 0 to 7 loop
70                  add_16(
71                      in_rs2(16*(i+1)-1 downto i*16),
72                      in_rs1(16*(i+1)-1 downto i*16),
73                      temp_out(16*(i+1)-1 downto i*16)
74                  );
75              end loop;
76
77          when "0101" => --bitwise logical or
78              temp_out := in_rs2 or in_rs1;
79
80          when "0110" => --broadcast word
81              for i in 0 to 3 loop
82                  temp_out(32*(i+1)-1 downto i*32) := in_rs1(REGISTER_LENGTH-1 downto
83                                         REGISTER_LENGTH-32);
84              end loop;
85
86          when "0111" => -- max signed word
87              for i in 0 to 3 loop
88                  if signed(in_rs2(32*(i+1)-1 downto i*32)) > signed(in_rs1(32*(i+1)-1
89                                         downto i*32)) then
90                      temp_out(32*(i+1)-1 downto i*32) := in_rs2(32*(i+1)-1 downto i*32);
91                  else
92                      temp_out(32*(i+1)-1 downto i*32) := in_rs1(32*(i+1)-1 downto i*32);
93                  end if;
94              end loop;
95
96          when "1000" => --min signed word
97              for i in 0 to 3 loop
98                  if signed(in_rs2(32*(i+1)-1 downto i*32)) < signed(in_rs1(32*(i+1)-1
99                                         downto i*32)) then

```

```

96          temp_out(32*(i+1)-1 downto i*32) := in_rs2(32*(i+1)-1 downto i*32);
97      else
98          temp_out(32*(i+1)-1 downto i*32) := in_rs1(32*(i+1)-1 downto i*32);
99      end if;
100     end loop;
101
102    when "1001" => --multiply low unsigned
103        for i in 0 to 3 loop
104            mult_16_unsigned(
105                in_rs1(16*(i*2+1)-1 downto 16*(i*2)),
106                in_rs2(16*(i*2+1)-1 downto 16*(i*2)),
107                temp_out(32*(i+1)-1 downto 32*i)
108            );
109        end loop;
110
111    when "1010" => --multiply low by constant unsigned
112        for i in 0 to 3 loop
113            mult_16_unsigned(
114                in_rs1(16*(i*2+1)-1 downto 16*(i*2)),
115                "000000000000" & in_immed(4 downto 0),
116                temp_out(32*(i+1)-1 downto 32*i)
117            );
118        end loop;
119
120    when "1011" => --bitwise logical and
121        temp_out := in_rs2 and in_rs1;
122
123    when "1100" => --count leading zeroes in words
124        for i in 0 to 3 loop
125            vector32 := in_rs1(32*(i+1)-1 downto i*32);
126            int_var := 0;
127        if unsigned(vector32) = 0 then
128            int_var := 0;
129        else
130            for bit_index in 31 downto 0 loop
131                if vector32(bit_index) = '0' then
132                    int_var := int_var + 1;
133                else
134                    exit;
135                end if;
136            end loop;
137        end if;
138        temp_out(32*(i+1)-1 downto i*32) := std_logic_vector(to_unsigned(int_var,
139                                            32));
140    end loop;
141
142    when "1101" => --rotate bits in word
143        for i in 0 to 3 loop
144            vector32 := in_rs1(32*(i+1)-1 downto i*32);
145            int_var := to_integer(unsigned(in_rs2(32*(i+1)-27-1 downto i*32))) mod
146            32;
147            if int_var = 0 then
148                temp_out(32*(i+1)-1 downto 32*i) := vector32;
149            else
150                temp_out(32*(i+1)-1 downto 32*i) := vector32(int_var-1 downto 0) &
151                vector32(31 downto int_var);
152            end if;
153        end loop;
154
155    when "1110" => --subtract from word unsigned
156        for i in 0 to 3 loop
157            sub_32_unsigned(
158                in_rs2(32*(i+1)-1 downto 32*i),
159                in_rs1(32*(i+1)-1 downto 32*i),
160                temp_out(32*(i+1)-1 downto 32*i)
161            );
162        end loop;
163
164    when "1111" => --subtract from halfword saturated
165        for i in 0 to 7 loop
166            sub_16(
167                in_rs2(16*(i+1)-1 downto 16*i),
168                in_rs1(16*(i+1)-1 downto 16*i),

```

```

166         temp_out(16*(i+1)-1 downto 16*i)
167     );
168   end loop;
169
170   when others =>
171     temp_out := (others => '-');
172   end case;
173   if is_nop = '0' then
174     out_rd := temp_out;
175   end if;
176   end procedure;
177 end package body rest_instruction;

```

## 8.7 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;

```

## 8.8 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
63 -----Multiple_32-bits-----
64  procedure mult_32(
65    a32, b32 : in std_logic_vector(31 downto 0);
66    ret64   : out std_logic_vector(63 downto 0)
67  ) is
68    variable prod : signed(63 downto 0);
69  begin
70    prod := signed(a32) * signed(b32);
71    ret64 := std_logic_vector(prod);
72  end procedure;
73
74 -----Addition_16-bits-----
75  procedure add_16(
76    a16, b16 : in std_logic_vector(15 downto 0);
77    ret16   : out std_logic_vector(15 downto 0)
78  ) is
79    variable sum : signed(16 downto 0);
80  begin
81    sum := resize(signed(a16), 17) + resize(signed(b16), 17);
82
83    if sum > resize(MAX16, 17) then
84      ret16 := std_logic_vector(MAX16);
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;

```

## 8.9 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
12   signal opcode   : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14   signal in_rs3   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15   signal in_rs2   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16   signal in_rs1   : std_logic_vector(REGISTER_LENGTH-1 downto 0);

```

```

17  signal in_immed      : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
18
19 --forwarding
20 signal rs3_ptr       : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
21 signal rs2_ptr       : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22 signal rs1_ptr       : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23
24 signal wb_rd         : std_logic_vector(REGISTER_LENGTH-1 downto 0);
25 signal wb_rd_ptr     : std_logic_vector(ADDRESS_LENGTH-1 downto 0); --for forwarding
   address comparision
26 signal in_wback      : std_logic;
27
28 --outputs
29 signal out_rd        : std_logic_vector(REGISTER_LENGTH-1 downto 0);
30
31 constant period: time := 20ns;
32 -----
33 -- Helper: Convert std_logic_vector ? hex string (portable)
34 -----
35 function slv_to_hex(slv : std_logic_vector) return string is
36 variable num_nibbles : integer := (slv'length + 3) / 4;
37     variable padded_slv : std_logic_vector(num_nibbles * 4 - 1 downto 0);
38     variable result      : string(1 to num_nibbles);
39     variable nibble_val : integer;
40 begin
41     -- Pad MSBs with zeros if not multiple of 4 bits
42     padded_slv := (others => '0');
43     padded_slv(slv'length - 1 downto 0) := slv;
44
45     for i in 0 to num_nibbles - 1 loop
46         nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
47         case nibble_val is
48             when 0 => result(num_nibbles - i) := '0';
49             when 1 => result(num_nibbles - i) := '1';
50             when 2 => result(num_nibbles - i) := '2';
51             when 3 => result(num_nibbles - i) := '3';
52             when 4 => result(num_nibbles - i) := '4';
53             when 5 => result(num_nibbles - i) := '5';
54             when 6 => result(num_nibbles - i) := '6';
55             when 7 => result(num_nibbles - i) := '7';
56             when 8 => result(num_nibbles - i) := '8';
57             when 9 => result(num_nibbles - i) := '9';
58             when 10 => result(num_nibbles - i) := 'A';
59             when 11 => result(num_nibbles - i) := 'B';
60             when 12 => result(num_nibbles - i) := 'C';
61             when 13 => result(num_nibbles - i) := 'D';
62             when 14 => result(num_nibbles - i) := 'E';
63             when 15 => result(num_nibbles - i) := 'F';
64             when others => result(num_nibbles - i) := 'X';
65         end case;
66     end loop;
67     return result;
68 end function;
69 -----
70 begin
71
72 UUT : entity work.mmu
73 port map(
74     opcode      => opcode,
75
76     in_rs3      => in_rs3,
77     in_rs2      => in_rs2,
78     in_rs1      => in_rs1,
79     in_immed    => in_immed,
80
81     rs3_ptr     => rs3_ptr,
82     rs2_ptr     => rs2_ptr,
83     rs1_ptr     => rs1_ptr,
84
85     wb_rd       => wb_rd,
86     wb_rd_ptr   => wb_rd_ptr,
87     in_wback   => in_wback,
88     out_rd      => out_rd

```

```

89      );
90
91      -- stimulus process
92      stim_proc : process
93      begin
94      -----
95      -- load_immediate TEST w/ indexing
96      -----
97      in_immed <= x"DEAD";
98      in_rs3 <= (others => '0');
99      -----
100     -- TEST: opcode = 0--000
101    -----
102     opcode <= std_logic_vector(to_unsigned(0, OPCODE_LENGTH));
103     wait for period;
104     assert out_rd(15 downto 0) = x"DEAD"
105     report "Test failed: 000000, out_rd = x" & slv_to_hex(out_rd)
106     severity error;
107
108     -----
109     -- TEST: opcode = 0--001
110    -----
111     opcode <= std_logic_vector(to_unsigned(1, OPCODE_LENGTH));
112     wait for period;
113     assert out_rd(31 downto 16) = x"DEAD"
114     report "Test failed: 000001, out_rd = x" & slv_to_hex(out_rd)
115     severity error;
116
117     -----
118     -- TEST: opcode = 0--110
119    -----
120     opcode <= std_logic_vector(to_unsigned(6, OPCODE_LENGTH));
121     wait for period;
122     assert out_rd(111 downto 96) = x"DEAD"
123     report "Test failed: 100110, out_rd = x" & slv_to_hex(out_rd)
124     severity error;
125
126     -----
127     -- TEST: opcode = 0--111
128    -----
129     opcode <= std_logic_vector(to_unsigned(7, OPCODE_LENGTH));
130     wait for period;
131     assert out_rd(127 downto 112) = x"DEAD"
132     report "Test failed: 100111, out_rd = x" & slv_to_hex(out_rd)
133     severity error;
134
135     report "TEST COMPLETED: load_immediate w/ indexing" severity warning;
136   end process;
137 end test_bench;

```

## 8.10 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
12   signal opcode    : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14   signal in_rs3    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15   signal in_rs2    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16   signal in_rs1    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
17   signal in_immed  : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
18
19   --fowarding

```

```

20 signal rs3_ptr      : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
21 signal rs2_ptr      : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22 signal rs1_ptr      : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23
24 signal wb_rd       : std_logic_vector(REGISTER_LENGTH-1 downto 0);
25 signal wb_rd_ptr   : std_logic_vector(ADDRESS_LENGTH-1 downto 0); --for forwarding
26             address comparision
27 signal in_wback    : std_logic;
28
29 --outputs
30 signal out_rd      : std_logic_vector(REGISTER_LENGTH-1 downto 0);
31
32 constant period: time := 20ns;
33 -----
34 -- Helper: Convert std_logic_vector ? hex string (portable)
35 -----
36 function slv_to_hex(slv : std_logic_vector) return string is
37 variable num_nibbles : integer := (slv'length + 3) / 4;
38 variable padded_slv : std_logic_vector(num_nibbles * 4 - 1 downto 0);
39 variable result      : string(1 to num_nibbles);
40 variable nibble_val : integer;
41 begin
42     -- Pad MSBs with zeros if not multiple of 4 bits
43     padded_slv := (others => '0');
44     padded_slv(slv'length - 1 downto 0) := slv;
45
46     for i in 0 to num_nibbles - 1 loop
47         nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
48         case nibble_val is
49             when 0 => result(num_nibbles - i) := '0';
50             when 1 => result(num_nibbles - i) := '1';
51             when 2 => result(num_nibbles - i) := '2';
52             when 3 => result(num_nibbles - i) := '3';
53             when 4 => result(num_nibbles - i) := '4';
54             when 5 => result(num_nibbles - i) := '5';
55             when 6 => result(num_nibbles - i) := '6';
56             when 7 => result(num_nibbles - i) := '7';
57             when 8 => result(num_nibbles - i) := '8';
58             when 9 => result(num_nibbles - i) := '9';
59             when 10 => result(num_nibbles - i) := 'A';
60             when 11 => result(num_nibbles - i) := 'B';
61             when 12 => result(num_nibbles - i) := 'C';
62             when 13 => result(num_nibbles - i) := 'D';
63             when 14 => result(num_nibbles - i) := 'E';
64             when 15 => result(num_nibbles - i) := 'F';
65             when others => result(num_nibbles - i) := 'X';
66         end case;
67     end loop;
68     return result;
69 end function;
70 -----
71 begin
72     UUT : entity work.mmu
73     port map(
74         opcode      => opcode,
75
76         in_rs3      => in_rs3,
77         in_rs2      => in_rs2,
78         in_rs1      => in_rs1,
79         in_immed   => in_immed,
80
81         rs3_ptr     => rs3_ptr,
82         rs2_ptr     => rs2_ptr,
83         rs1_ptr     => rs1_ptr,
84
85         wb_rd       => wb_rd,
86         wb_rd_ptr   => wb_rd_ptr,
87         in_wback    => in_wback,
88         out_rd      => out_rd
89     );
90
91     -- Stimulus process

```

```

92  stim_proc : process
93  begin
94  -----
95  -- saturate_math TEST w/overflow and underflow checks
96  -----
97      in_immed <= x"DEAD";
98      in_rs3 <= x"000100020003000470057006F0077008";
99      in_rs2 <= x"000800070006000570047003F0027001";
100     in_rs1 <= x"0000000000000000007FFF00007FFF0000";
101
102     -----
103     -- TEST: Opcode 10-000
104     -----
105     opcode <= "100000";
106     wait for period;
107     assert out_rd = x"0000000E000000147FFFFFF7FFFFFF"
108         report "Test failed: 100000, out_rd = x" & slv_to_hex(out_rd)
109         severity error;
110
111     -----
112     -- TEST: Opcode 10-001
113     -----
114     opcode <= "100001";
115     wait for period;
116     assert out_rd = x"000000080000000127FFFFFF7FFFFFF"
117         report "Test failed: 100001, out_rd = x" & slv_to_hex(out_rd)
118         severity error;
119
120     -----
121     -- TEST: Opcode 10-010
122     -----
123     in_rs1 <= x"00000000000000008001000080010000";
124     opcode <= "100010";
125     wait for period;
126     assert out_rd = x"FFFFFFF2FFFFFFEC8000000080000000"
127         report "Test failed: 100010, out_rd = x" & slv_to_hex(out_rd)
128         severity error;
129
130     -----
131     -- TEST: Opcode 10-011
132     -----
133     opcode <= "100011";
134     wait for period;
135     assert out_rd = x"FFFFFFF8FFFFFFEE8000000080000000"
136         report "Test failed: 100011, out_rd = x" & slv_to_hex(out_rd)
137         severity error;
138
139     -----
140     -- TEST: Opcode 10-100
141     -----
142     in_rs1 <= x"0000000000000000007FFF00007FFF0000";
143     opcode <= "100100";
144     wait for period;
145     assert out_rd = x"00000012002700147FFFFFF7FFFFFF"
146         report "Test failed: 100100, out_rd = x" & slv_to_hex(out_rd)
147         severity error;
148
149     -----
150     -- TEST: Opcode 10-101
151     -----
152     opcode <= "100101";
153     wait for period;
154     assert out_rd = x"000000080017000E7FFFFFF7FFFFFF"
155         report "Test failed: 100101, out_rd = x" & slv_to_hex(out_rd)
156         severity error;
157
158     -----
159     -- TEST: Opcode 10-110
160     -----
161     in_rs1 <= x"00000000000000008001000080010000";
162     opcode <= "100110";
163     wait for period;
164     assert out_rd = x"FFFFFFFEDFFD8FFEC8000000000000000000"

```

```

165     report "Test failed: 100110, out_rd = x" & slv_to_hex(out_rd)
166     severity error;
167
168 -----
169 -- TEST: Opcode 10-111
170 -----
171     opcode <= "100111";
172     wait for period;
173     assert out_rd = x"FFFFFFFFFF7FFE8FFF280000000000000000"
174     report "Test failed: 100111, out_rd = x" & slv_to_hex(out_rd)
175     severity error;
176
177     report "TEST COMPLETED: saturate_math w/o saturating" severity warning;
178 end process;
179 end test_bench;

```

## 8.11 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
12   signal opcode : std_logic_vector(OPCODE_LENGTH-1 downto 0);
13
14   signal in_rs3 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
15   signal in_rs2 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
16   signal in_rs1 : std_logic_vector(REGISTER_LENGTH-1 downto 0);
17   signal in_immed : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
18
19   --fowarding
20   signal rs3_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
21   signal rs2_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
22   signal rs1_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
23
24   signal wb_rd : std_logic_vector(REGISTER_LENGTH-1 downto 0);
25   signal wb_rd_ptr : std_logic_vector(ADDRESS_LENGTH-1 downto 0); --for forwarding
26   address comparision
26   signal in_wback : std_logic;
27
28   --outputs
29   signal out_rd : std_logic_vector(REGISTER_LENGTH-1 downto 0);
30
31   constant period: time := 20ns;
32
33   -- Helper: Convert std_logic_vector ? hex string (portable)
34
35   function slv_to_hex(slv : std_logic_vector) return string is
36     variable num_nibbles : integer := (slv'length + 3) / 4;
37     variable padded_slv : std_logic_vector(num_nibbles * 4 - 1 downto 0);
38     variable result : string(1 to num_nibbles);
39     variable nibble_val : integer;
40
41   begin
42     -- Pad MSBs with zeros if not multiple of 4 bits
43     padded_slv := (others => '0');
44     padded_slv(slv'length - 1 downto 0) := slv;
45
46     for i in 0 to num_nibbles - 1 loop
47       nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
48       case nibble_val is
49         when 0 => result(num_nibbles - i) := '0';
50         when 1 => result(num_nibbles - i) := '1';
51         when 2 => result(num_nibbles - i) := '2';
52         when 3 => result(num_nibbles - i) := '3';
53         when 4 => result(num_nibbles - i) := '4';

```

```

53     when 5 => result(num_nibbles - i) := '5';
54     when 6 => result(num_nibbles - i) := '6';
55     when 7 => result(num_nibbles - i) := '7';
56     when 8 => result(num_nibbles - i) := '8';
57     when 9 => result(num_nibbles - i) := '9';
58     when 10 => result(num_nibbles - i) := 'A';
59     when 11 => result(num_nibbles - i) := 'B';
60     when 12 => result(num_nibbles - i) := 'C';
61     when 13 => result(num_nibbles - i) := 'D';
62     when 14 => result(num_nibbles - i) := 'E';
63     when 15 => result(num_nibbles - i) := 'F';
64     when others => result(num_nibbles - i) := 'X';
65   end case;
66 end loop;
67 return result;
68 end function;
69 -----
70
71 begin
72   UUT : entity work.mmu
73   port map(
74     opcode      => opcode,
75
76     in_rs3      => in_rs3,
77     in_rs2      => in_rs2,
78     in_rs1      => in_rs1,
79     in_immed    => in_immed,
80
81     rs3_ptr     => rs3_ptr,
82     rs2_ptr     => rs2_ptr,
83     rs1_ptr     => rs1_ptr,
84
85     wb_rd       => wb_rd,
86     wb_rd_ptr   => wb_rd_ptr,
87     in_wback   => in_wback,
88     out_rd      => out_rd
89   );
90
91   -- Clock process
92
93
94   -- Stimulus process
95   stim_proc : process
96   begin
97   -----
98   -- rest_instruction TEST
99   -----
100  --
101  -- TEST: Opcode 110001
102  --
103  opcode <= "110001";
104  in_immed <= x"0004";
105  in_rs2 <=(others => '-');
106  in_rs1 <= x"700870077006700570047003F0027001";
107  wait for period;
108  assert out_rd = x"0700070007000700070007000F000700"
109  report "TEST FAIL: 110001, out_rd =" & slv_to_hex(out_rd)
110  severity error;
111  --
112  -- TEST: Opcode 110010, stauturated
113  --
114  opcode <= "110010";
115  in_rs2 <= x"80010004800100037FFF00027FFF0001";
116  in_rs1 <= x"700870077006700570047003F0027001";
117  wait for period;
118  assert out_rd = x"F009700BF0077008F0037005FFFFFFF"
119  report "TEST FAIL: 110010, out_rd =" & slv_to_hex(out_rd)
120  severity error;
121  --
122  -- TEST: Opcode 110011
123  --
124  opcode <= "110011";

```

```

126  in_rs2 <= (others => '-');
127  in_rs1 <= x"700870077006700570047003F0027001";
128  wait for period;
129  assert out_rd = x"00040006000500050004000500050004"
130  report "TEST FAIL: 110011, out_rd =" & slv_to_hex(out_rd)
131  severity error;
132
133  -----
134  -- TEST: Opcode 110100
135  -----
136  opcode <= "110100";
137  in_rs2 <= x"80010004800100037FFF7FF27FFFFFFF1";
138  in_rs1 <= x"700870077006800570047003F0027001";
139  wait for period;
140  assert out_rd = x"F009700BF00780087FFF7FFF70016FF2"
141  report "TEST FAIL: 110100, out_rd =" & slv_to_hex(out_rd)
142  severity error;
143
144  -----
145  -- TEST: Opcode 110101
146  -----
147  opcode <= "110101";
148  in_rs2 <= x"000000000000000000FFFFFFFFFFFFFFFF";
149  in_rs1 <= x"700870077006700570047003F0027001";
150  wait for period;
151  assert out_rd = x"7008700770067005FFFFFFFFFFFF"
152  report "TEST FAIL: 110101, out_rd =" & slv_to_hex(out_rd)
153  severity error;
154
155  -----
156  -- TEST: Opcode 110110
157  -----
158  opcode <= "110110";
159  in_rs2 <= (others => '-');
160  in_rs1 <= x"700870077006700570047003F0027001";
161  wait for period;
162  assert out_rd = x"700870077008700770087007"
163  report "TEST FAIL: 110110, out_rd =" & slv_to_hex(out_rd)
164  severity error;
165
166  -----
167  -- TEST: Opcode 110111
168  -----
169  opcode <= "110111";
170  in_rs2 <= x"80010004800100037FFF00027FFF0001";
171  in_rs1 <= x"700870077006700570047003F0027001";
172  wait for period;
173  assert out_rd = x"70087007700670057FFF00027FFF0001"
174  report "TEST FAIL: 110111, out_rd =" & slv_to_hex(out_rd)
175  severity error;
176
177  -----
178  -- TEST: Opcode 111000
179  -----
180  opcode <= "111000";
181  in_rs2 <= x"80010004800100037FFF00027FFF0001";
182  in_rs1 <= x"700870077006700570047003F0027001";
183  wait for period;
184  assert out_rd = x"800100048001000370047003F0027001"
185  report "TEST FAIL: 111000, out_rd =" & slv_to_hex(out_rd)
186  severity error;
187
188  -----
189  -- TEST: Opcode 111001
190  -----
191  opcode <= "111001";
192  in_rs2 <= x"80010004800100037FFF7FF27FFFFFFF1";
193  in_rs1 <= x"700870077006700570047003F0027001";
194  wait for period;
195  assert out_rd = x"0001C01C0001500F37FB5FD66FFA6FF1"
196  report "TEST FAIL: 111001, out_rd =" & slv_to_hex(out_rd)
197  severity error;
198
```

```

199  -----
200  -- TEST: Opcode 111010
201  -----
202  opcode <= "111010";
203  in_immed <= x"0014";
204  in_rs2 <= (others => '-');
205  in_rs1 <= x"80010004800100037FFF00027FFF0001";
206  wait for period;
207  assert out_rd = x"000000500000003C000002800000014"
208  report "TEST FAIL: 111010, out_rd =" & slv_to_hex(out_rd)
209  severity error;
210
211  -----
212  -- TEST: Opcode 111011
213  -----
214  opcode <= "111011";
215  in_rs2 <= x"0000000000000000FFFFFFFFFF";
216  in_rs1 <= x"80010004800100037FFF00027FFF0001";
217  wait for period;
218  assert out_rd = x"0000000000000007FFF00027FFF0001"
219  report "TEST FAIL: 111011, out_rd =" & slv_to_hex(out_rd)
220  severity error;
221
222  -----
223  -- TEST: Opcode 111100
224  -----
225  opcode <= "111100";
226  in_rs2 <= (others => '-');
227  in_rs1 <= x"070870071006700570047003F0027001";
228  wait for period;
229  assert out_rd = x"0000000500000003000000100000000"
230  report "TEST FAIL: 111100, out_rd =" & slv_to_hex(out_rd)
231  severity error;
232
233  -----
234  -- TEST: Opcode 111101
235  -----
236  opcode <= "111101";
237  in_rs2 <= x"FFFFF10FFFFFFF0000000100000000";
238  in_rs1 <= x"80010004800100037FFF00027FFF0001";
239  wait for period;
240  assert out_rd = x"00048001000380013FFF80017FFF0001"
241  report "TEST FAIL: 111101, out_rd =" & slv_to_hex(out_rd)
242  severity error;
243
244  -----
245  -- TEST: Opcode 111110
246  -----
247  opcode <= "111110";
248  in_rs2 <= x"80010004800100037FFF00027FFF0001";
249  in_rs1 <= x"700870077006700570047003F0027001";
250  wait for period;
251  assert out_rd = x"OFF88FFD0FFA8FFE0FFA8FFF00000000"
252  report "TEST FAIL: 111110, out_rd =" & slv_to_hex(out_rd)
253  severity error;
254
255  -----
256  -- TEST: Opcode 111111
257  -----
258  opcode <= "111111";
259  in_rs2 <= x"80010004800100057FFF00067FFF8F01";
260  in_rs1 <= x"70080007780600037004FFF3F0027001";
261  wait for period;
262  assert out_rd = x"8000FFFD800000020FFB00137FFF8000"
263  report "TEST FAIL: 111111, out_rd =" & slv_to_hex(out_rd)
264  severity error;
265
266  -----
267  -- TEST: Opcode 110000
268  -----
269  opcode <= "110000";
270  in_rs2 <= (others => '-');
271  in_rs1 <= (others => '-');

```

```
272     wait for period;
273
274     report "TEST COMPLETED: rest of the instruction" severity warning;
275 end process;
276 end test_bench;
```