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

Project Report: Part I

Jin Yuan Chen
115702978

November 1, 2025

Abstract

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

The complete 4-stage 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 **Part I of the Final Report**, focusing exclusively on the architecture of the Multimedia ALU during the **execution stage**. At this phase, no prior knowledge of the complete pipeline design is required for implementing the Multimedia ALU module. The MMU is assumed to received all the correct and necessary input signals from the previous module up to the stage following the forwarding unit.

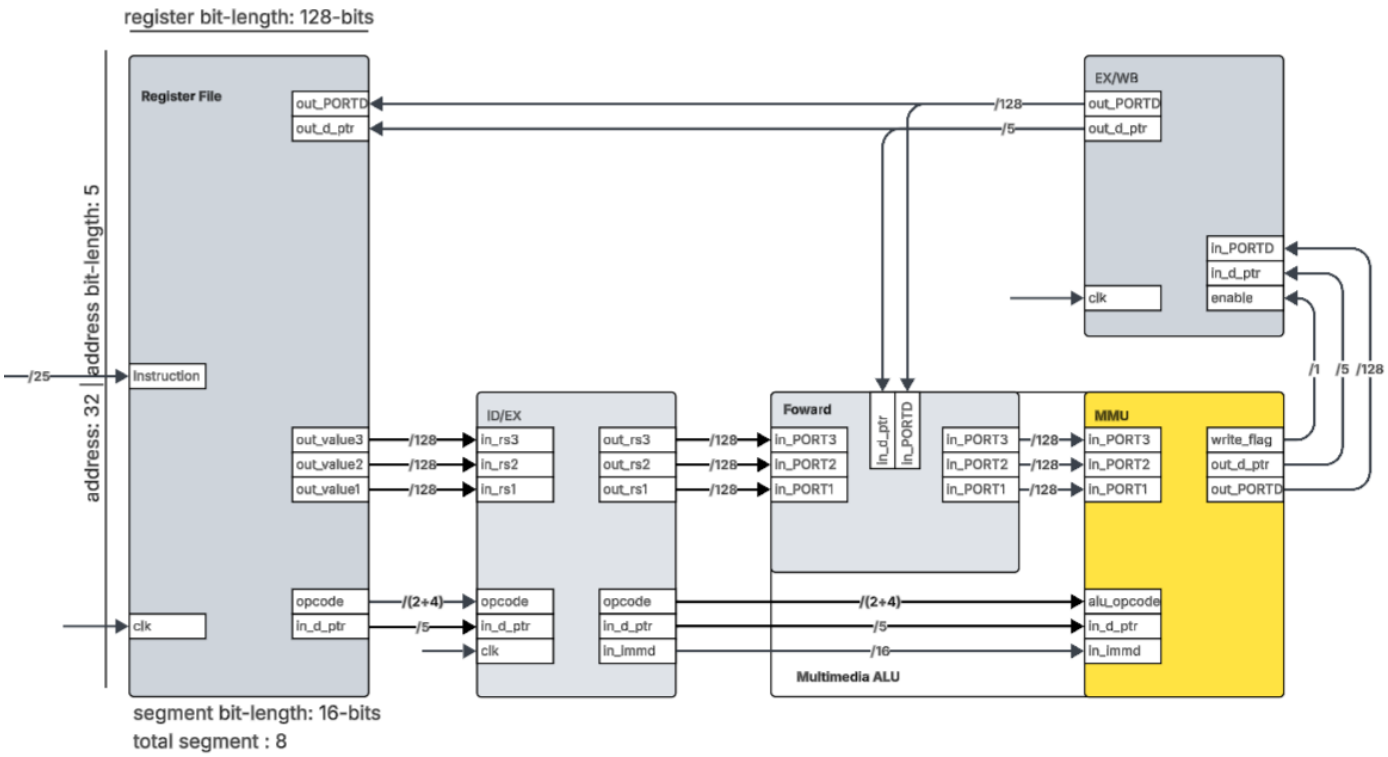


Figure 1: The RTL diagram highlight the execution stage of the Multimedia ALU.

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

2 Multimedia ALU

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

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

```
entity MMU_ALU is
    port (
        opcode      : in  std_logic_vector(OPCODE_LENGTH-1 downto 0);

        in_PORT3    : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);
        in_PORT2    : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);
        in_PORT1    : in  std_logic_vector(REGISTER_LENGTH-1 downto 0);

        in_immed    : in  std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
        in_d_ptr    : in  std_logic_vector(ADDRESS_LENGTH-1 downto 0);

        out_PORTD   : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
        out_d_ptr   : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
        wback_flag  : out std_logic
    );
end entity;
```

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

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

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

5 … 4	3 … 0
10	-000

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

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

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

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

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

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

```
architecture behavior of MMU_ALU is
begin
  main : process(
    opcode,
    in_PORT3, in_PORT2, in_PORT1, in_immed,
    in_d_ptr
    )
  begin
    out_d_ptr <= in_d_ptr;
    case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
      when "00" | "01" =>
        LDI_memory(     --ref. procedure_package/load_immediate.vhd
          opcode,
          in_PORT3,
          in_immed,

          out_PORTD,
          wback_flag
        );

      when "10" =>
        STM_main(     --ref. procedure_package/saturate_math.vhd
          opcode,
          in_PORT3,
          in_PORT2,
          in_PORT1,

          out_PORTD,
          wback_flag
        );

      when "11" =>
        RSI_main(     --ref. procedure_package/rest_instruction.vhd
          opcode,
          in_PORT2,
          in_PORT1,
          in_immed,

          out_PORTD,
          wback_flag
        );

      when others =>
        out_PORTD  <= (others => '0');
        out_d_ptr  <= (others => '0');
        wback_flag <= '0';
    end case;
  end process;
end architecture;
```

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

3 Instruction Set Description and Verfication

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:

$$\text{in\_PORT3} \leftarrow \text{FILE[rd]}$$

$$\text{in\_PORT3[index]} \leftarrow \text{immediate}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{in\_PORT3}$$

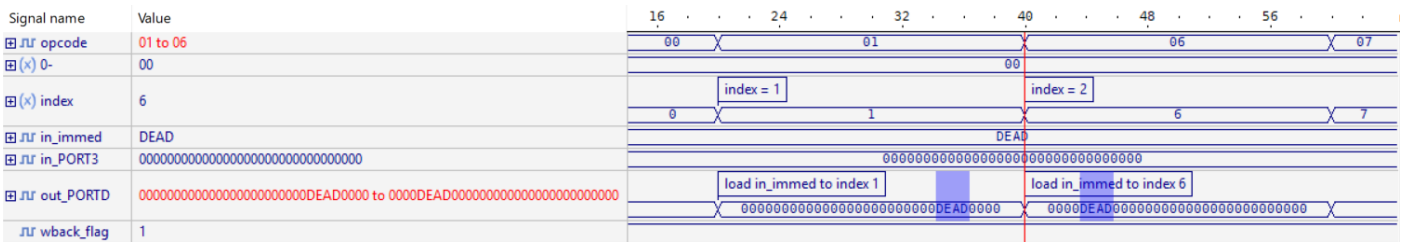
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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 hexdecimal 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:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret32} \leftarrow \text{SignExt(in\_PORT3[16-low])} \times \text{SignExt(in\_PORT2[16-low])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[32-field]} + \text{ret32)}$$

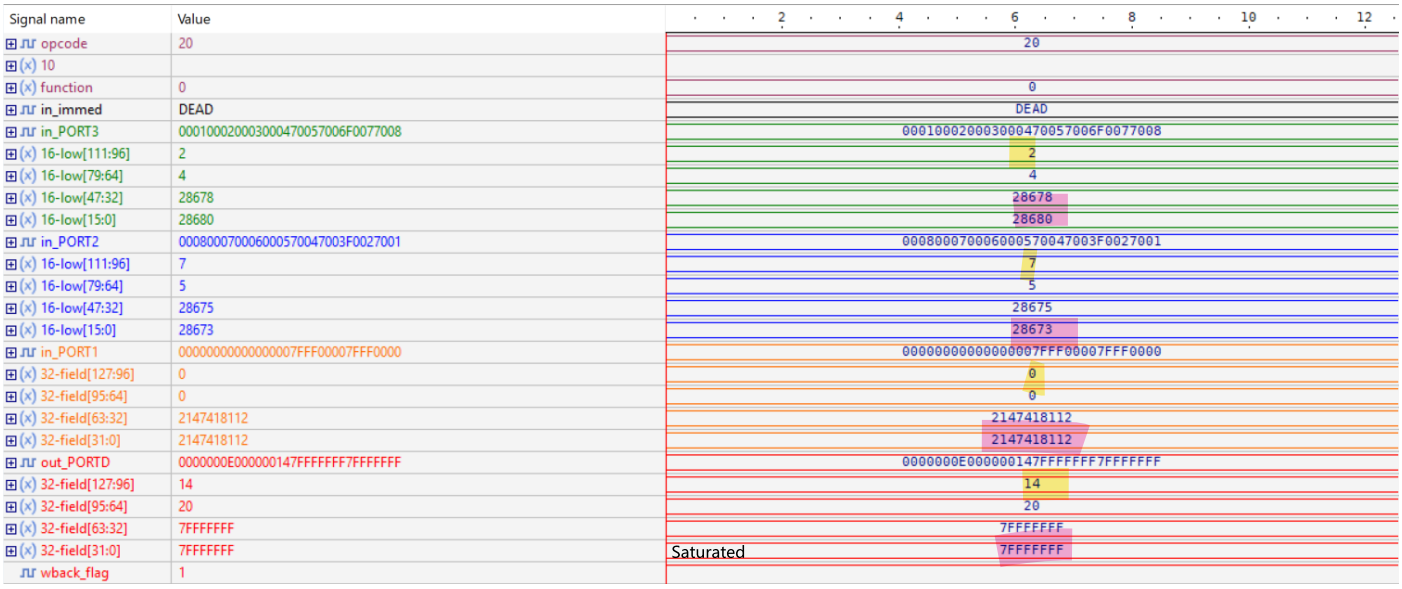
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

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



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:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret32} \leftarrow \text{SignExt(in\_PORT3[16-high])} \times \text{SignExt(in\_PORT2[16-high])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[32-field]} + \text{ret32)}$$

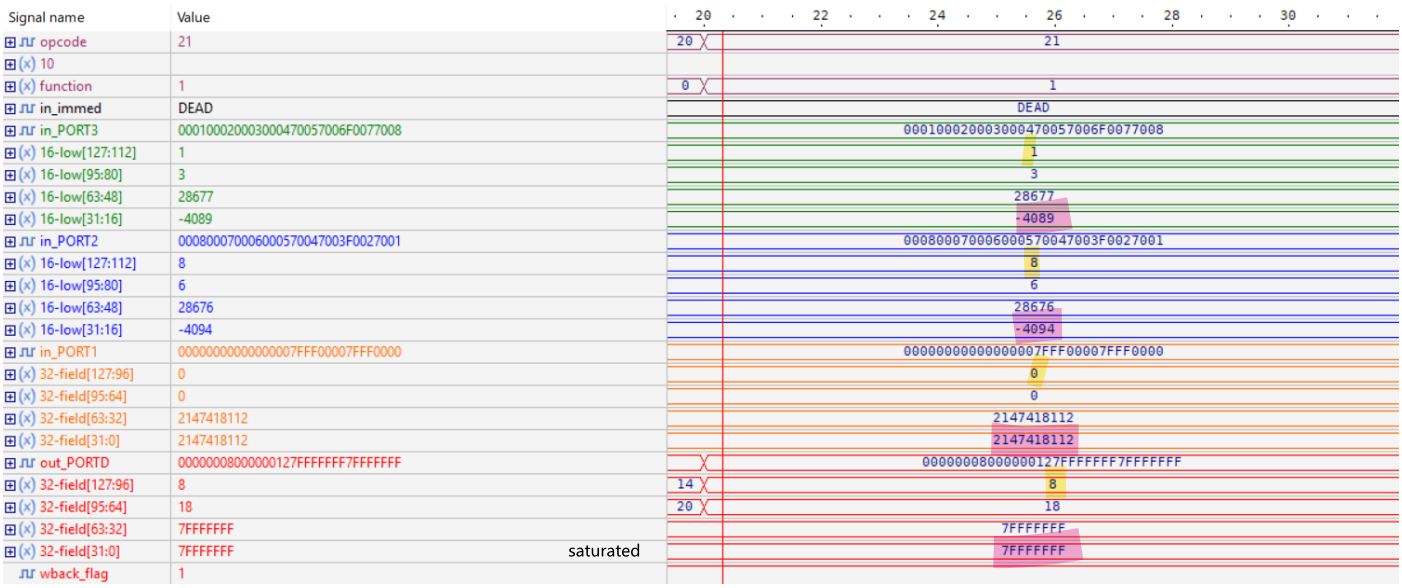
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

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



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

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

SIMSL - Saturated Signed Integer Multiply-Subtract Low

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

Instruction:

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

Opcode:

5 4	3 0
10	-010

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret32} \leftarrow \text{SignExt(in\_PORT3[16-low])} \times \text{SignExt(in\_PORT2[16-low])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[32-field]} - \text{ret32)}$$

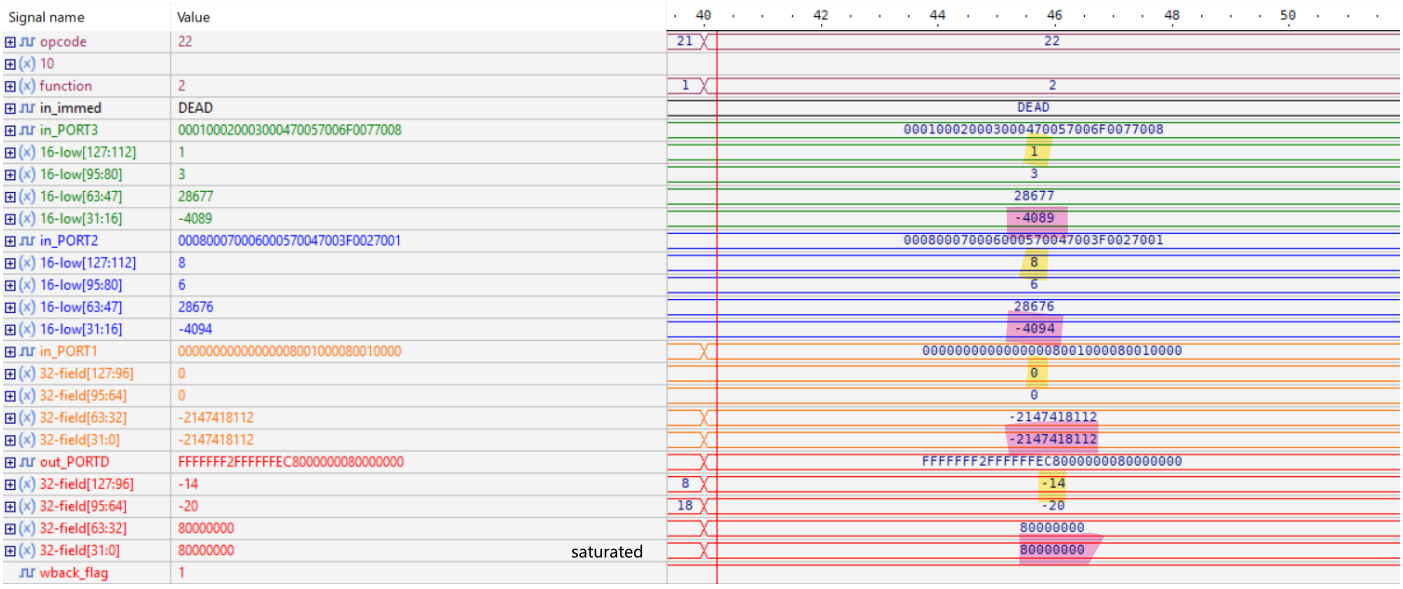
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

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



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

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

SIMSH - Saturated Signed Integer Multiply-Subtract High

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

Instruction:

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

Opcode:

5 4	3 0
10	-011

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret32} \leftarrow \text{SignExt(in\_PORT3[16-high])} \times \text{SignExt(in\_PORT2[16-high])}$$

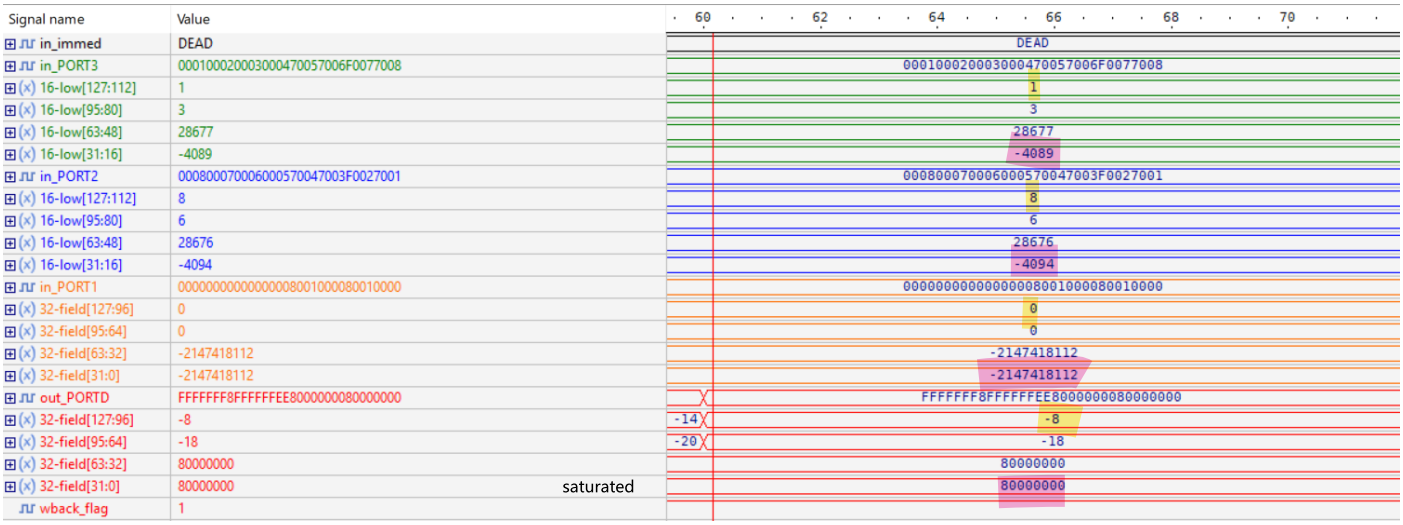
$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[32-field]} - \text{ret32)}$$

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

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

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

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



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

SLMAL - Saturated Signed Long Integer Multiply-Add Low

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

Instruction:

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

Opcode:

5 4	3 0
10	-100

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret64} \leftarrow \text{SignExt(in\_PORT3[32-low])} \times \text{SignExt(in\_PORT2[32-low])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[64-field]} + \text{ret64)}$$

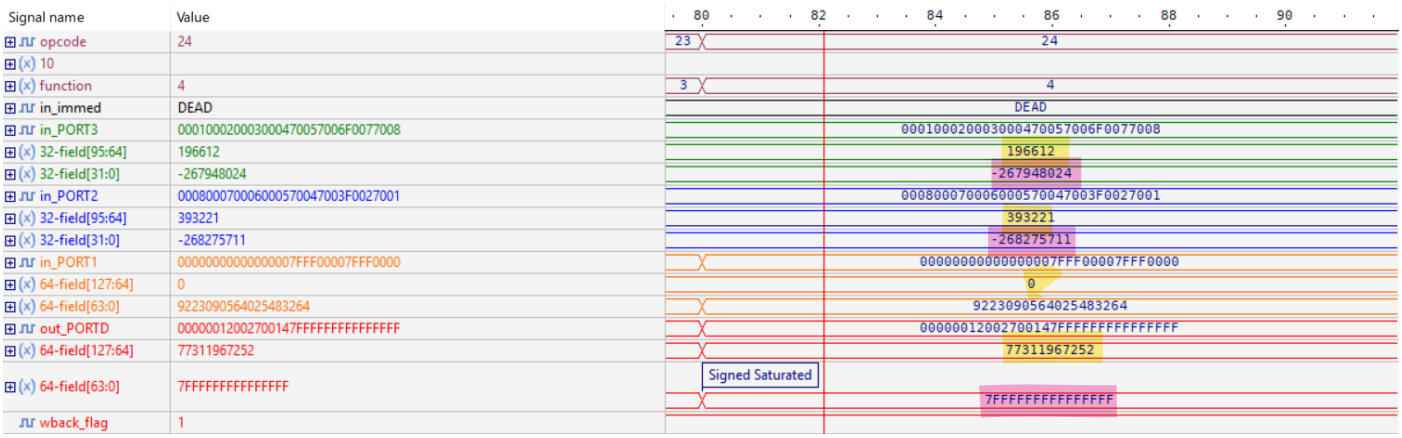
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

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



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 = 0x7FFFFFFFFFFFFFF, is represent in hexadecimal.

SLMAH - Saturated Signed Long Integer Multiply-Add High

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

Instruction:

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

Opcode:

5 4	3 0
10	-101

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret64} \leftarrow \text{SignExt(in\_PORT3[32-high])} \times \text{SignExt(in\_PORT2[32-high])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[64-field]} + \text{ret64)}$$

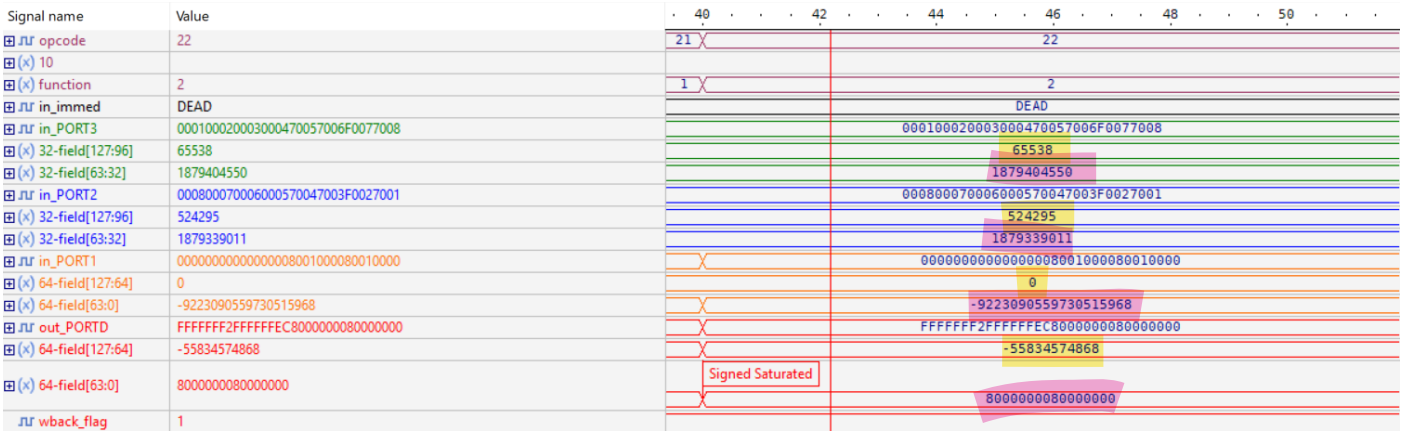
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexdecimal 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 = 0x7FFFFFFFFFFFFFFF, is represent in hexadecimal.

SLMSL - Saturated Signed Long Integer Multiply-Subtract Low

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

Instruction:

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

Opcode:

5 4	3 0
10	-110

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret64} \leftarrow \text{SignExt(in\_PORT3[32-low])} \times \text{SignExt(in\_PORT2[32-low])}$$

$$\text{out\_portD[31-fields]} \leftarrow \text{Saturate}_{32}\text{(in\_PORT1[64-field]} - \text{ret64)}$$

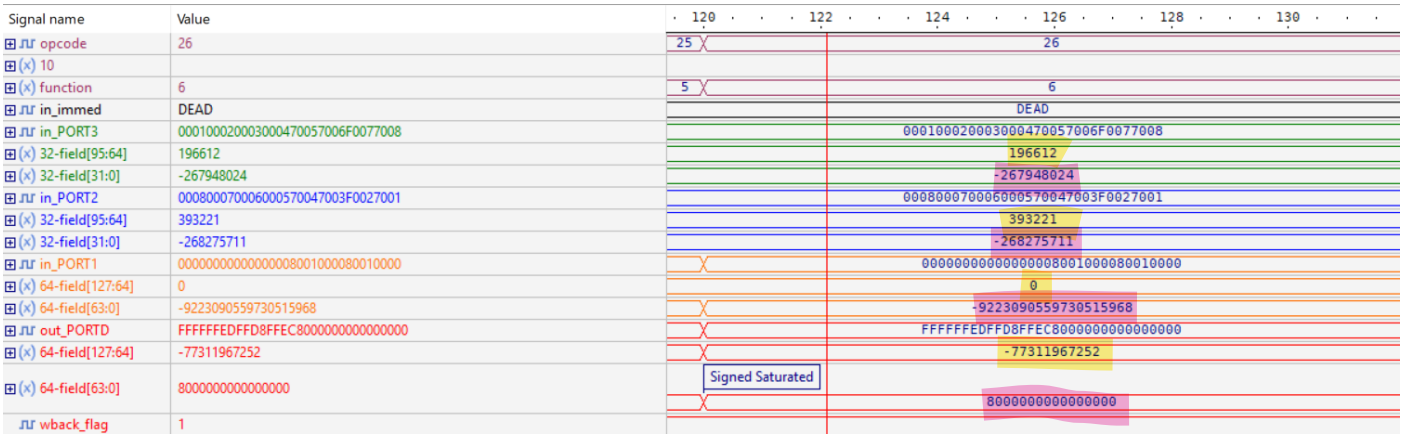
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

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



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 = 0x800000000000000000, is represent in hexadecimal.

SLMSH - Saturated Signed Long Integer Multiply-Subtract High

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

Instruction:

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

Opcode:

5 4	3 0
10	-111

Operation:

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

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

$$\text{in\_PORT3} \leftarrow \text{FILE[rs3]}$$

$$\text{ret64} \leftarrow \text{SignExt(in\_PORT3[32-high])} \times \text{SignExt(in\_PORT2[32-high])}$$

$$\text{out\_portD[64-fields]} \leftarrow \text{Saturate(in\_PORT1[64-field]} - \text{ret64)}$$

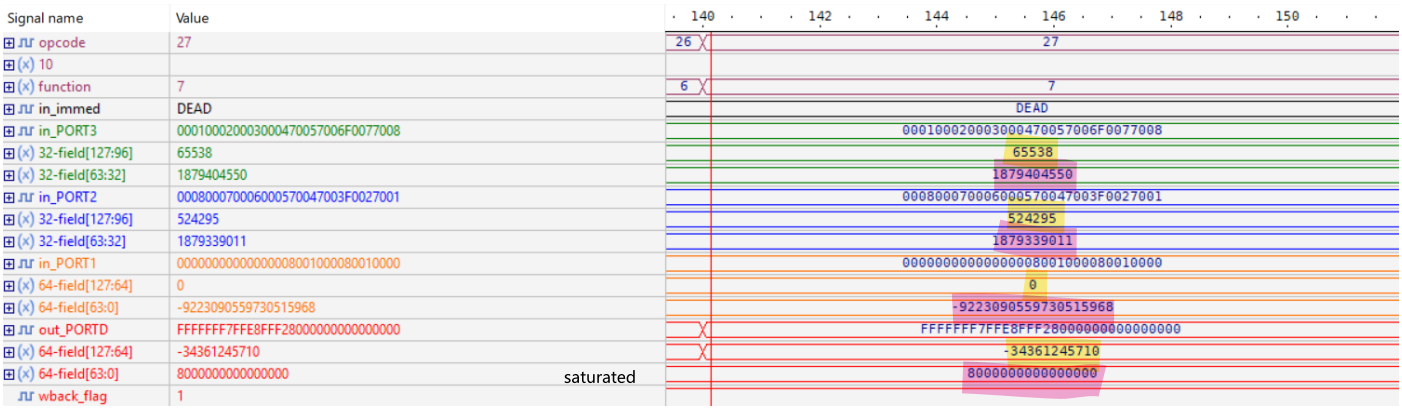
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD}$$

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

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

Verification:

The full 128-bit register values of in_PORT1, in_PORT2, and out_PORTD are represented in 32 hexdecimal 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 under-flow = 0x800000000000000000, is represent in hexadecimal.

SHRHI – Shift Right Halfword Immediate

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

Instruction:

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

Opcode:

5 – 4	3 – 0
11	0001

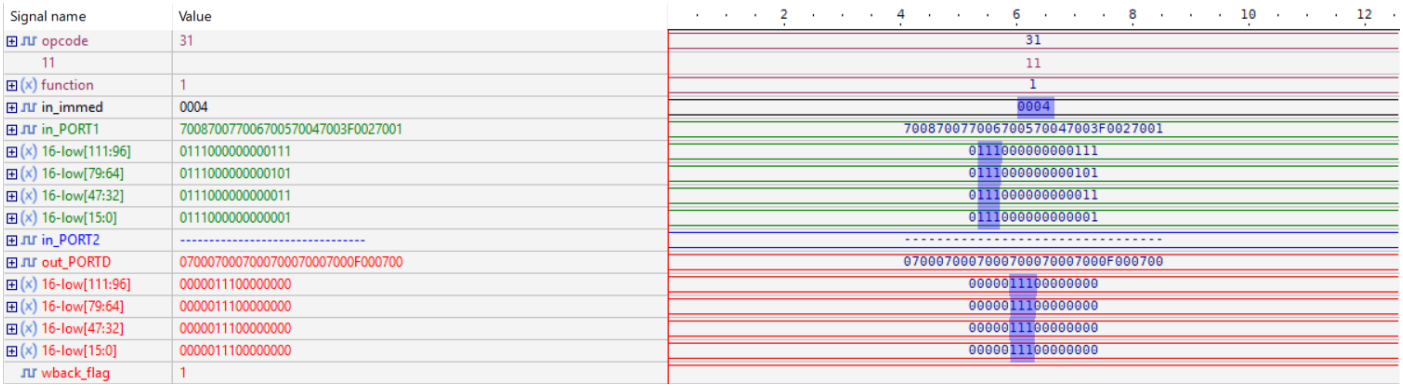
Operation:

$$\begin{aligned} \text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\ \text{in\_immed} &\leftarrow \text{rs2} \\ \text{out\_portD[16-low]} &\leftarrow \text{Logical}\big(\text{in\_PORT1[16-low]} \gg \text{in\_immed}\big) \\ \text{FILE[rd]} &\leftarrow \text{out\_PORTD}, \\ \text{wback\_flag} &\leftarrow 1 \end{aligned}$$

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

Verification:

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



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

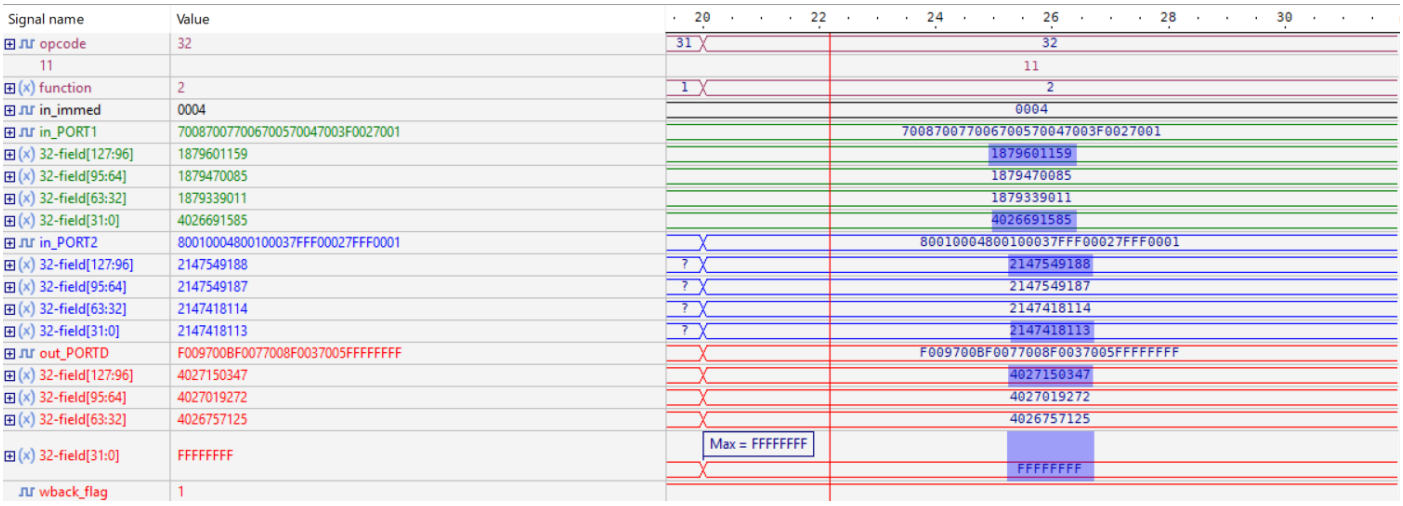
Operation:

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

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

Verification:

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



$$\begin{aligned}
\text{out_PORTD[32-fields]} &\leftarrow \text{Unsigned}\big(\text{in_PORT1[32-fields]} + \text{in_PORT2[32-fields]}\big) \\
\text{out_PORTD[127:96]} &\leftarrow \text{Unsigned}\big(\text{in_PORT1[127:96]} + \text{in_PORT2[127:96]}\big) \\
\boxed{4027150347} &= 1879601159 + 2147549188 \\
\text{out_PORTD[31:0]} &\leftarrow \text{Unsigned}\big(\text{in_PORT1[31:0]} + \text{in_PORT2[31:0]}\big) \\
\boxed{4294967295} &< 4026691585 + 2147418113
\end{aligned}$$

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:

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

$$\text{out\_PORTD[16-low]} \leftarrow \text{Count1s}\big(\text{in\_PORT1[16-low]}\big)$$

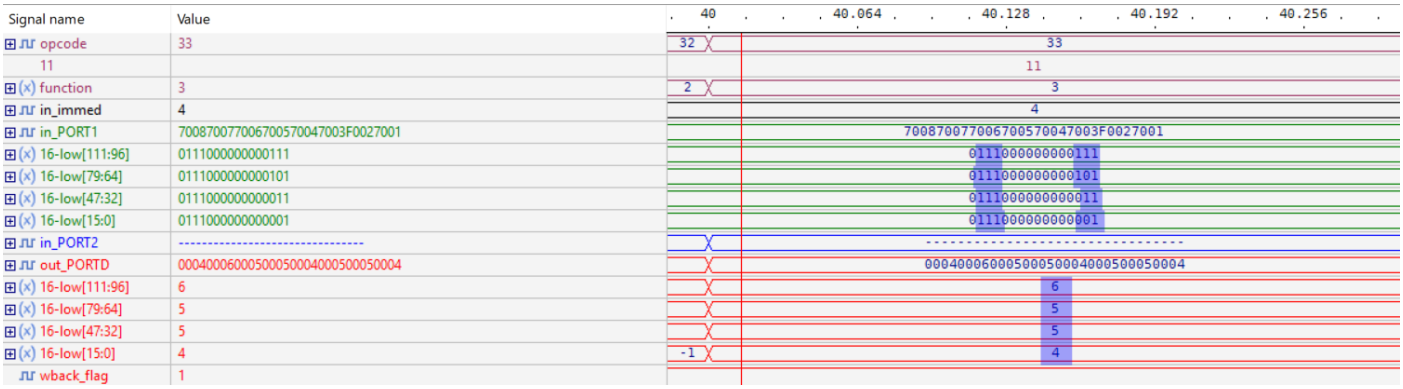
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD},$$

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

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 hexdecimal values.



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

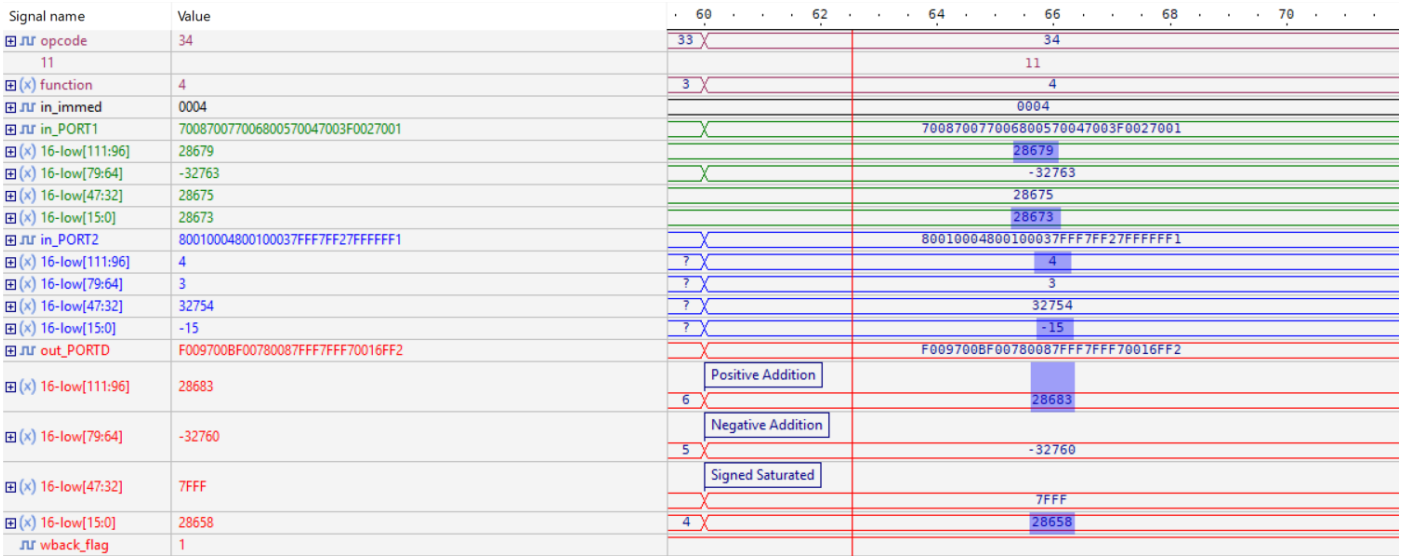
Operation:

$$\begin{aligned}
\text{in_PORT1} &\leftarrow \text{FILE[rs1]} \\
\text{in_PORT2} &\leftarrow \text{FILE[rs2]} \\
\text{out_PORTD[16-low]} &\leftarrow \text{SignedSaturate}\big(\text{in_PORT2[16-low]} + \text{in_PORT1[16-low]}\big) \\
\text{FILE[rd]} &\leftarrow \text{out_PORTD}, \\
\text{wback_flag} &\leftarrow 1
\end{aligned}$$

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

Verification:

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



$$\begin{aligned}
\text{out_PORTD[16-low]} &\leftarrow \text{Unsigned}\big(\text{in_PORT1[16-low]} + \text{in_PORT2[16-low]}\big) \\
\text{out_PORTD[111:96]} &\leftarrow \text{SignedSaturate}\big(\text{in_PORT1[111:96]} + \text{in_PORT2[111:96]}\big) \\
\boxed{28683} &= 28679 + 4 \\
\text{out_PORTD[31:0]} &\leftarrow \text{Unsigned}\big(\text{in_PORT1[47:32]} + \text{in_PORT2[47:32]}\big) \\
\boxed{32767} &< 28675 + 32754
\end{aligned}$$

OR – Bitwise Logical OR

Description:
Logical OR the contents of registers rs1 and rs2.

Instruction:

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

Opcode:

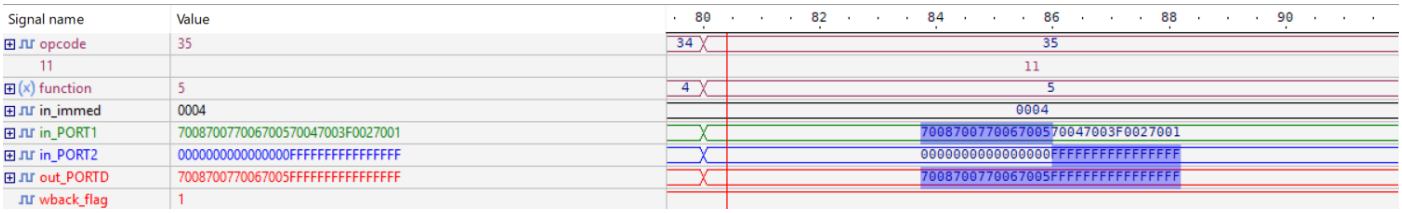
5 – 4	3 – 0
11	0101

Operation:

$$\begin{aligned} \text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\ \text{in\_PORT2} &\leftarrow \text{FILE[rs2]} \\ \text{out\_PORTD} &\leftarrow \text{in\_PORT2} \vee \text{in\_PORT1} \\ \text{FILE[rd]} &\leftarrow \text{out\_PORTD}, \\ \text{wback\_flag} &\leftarrow 1 \end{aligned}$$

Verification:

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



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:

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

$$\text{out\_PORTD[32-fields]} \leftarrow \text{in\_PORT1[127:96]}$$

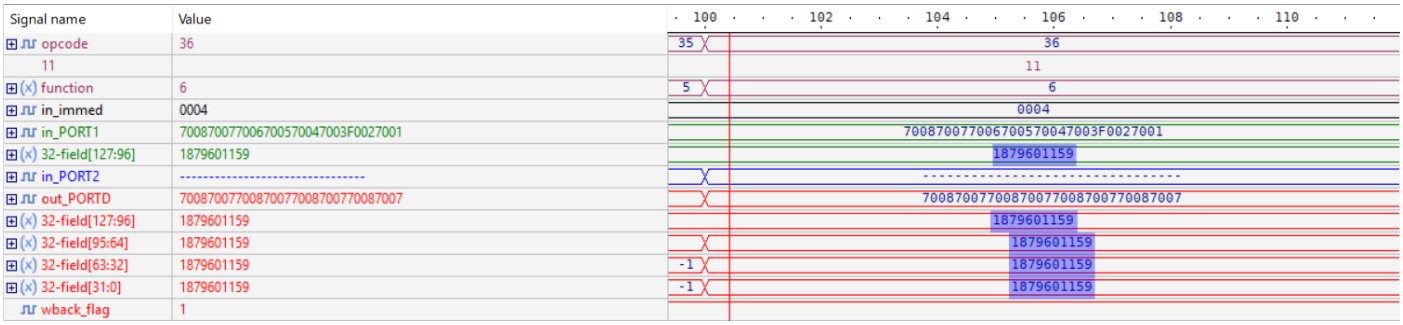
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD},$$

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

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 hexdecimal values.



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

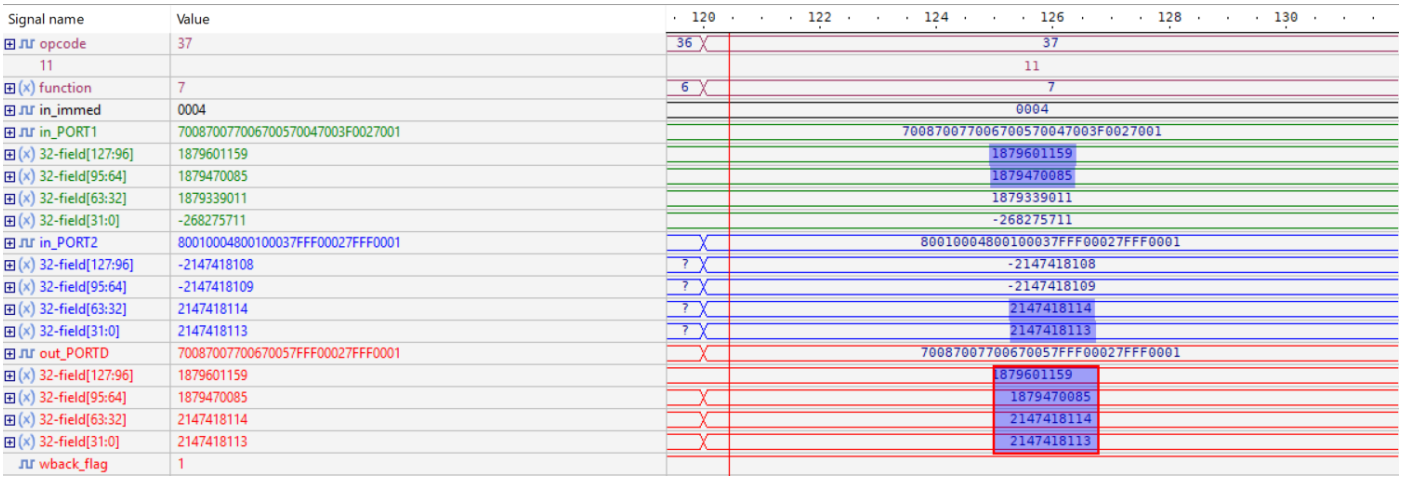
Operation:

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

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

Verification:

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



MINWS – Min Signed Word

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

Instruction:

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

Opcode:

5 – 4	3 – 0
11	1000

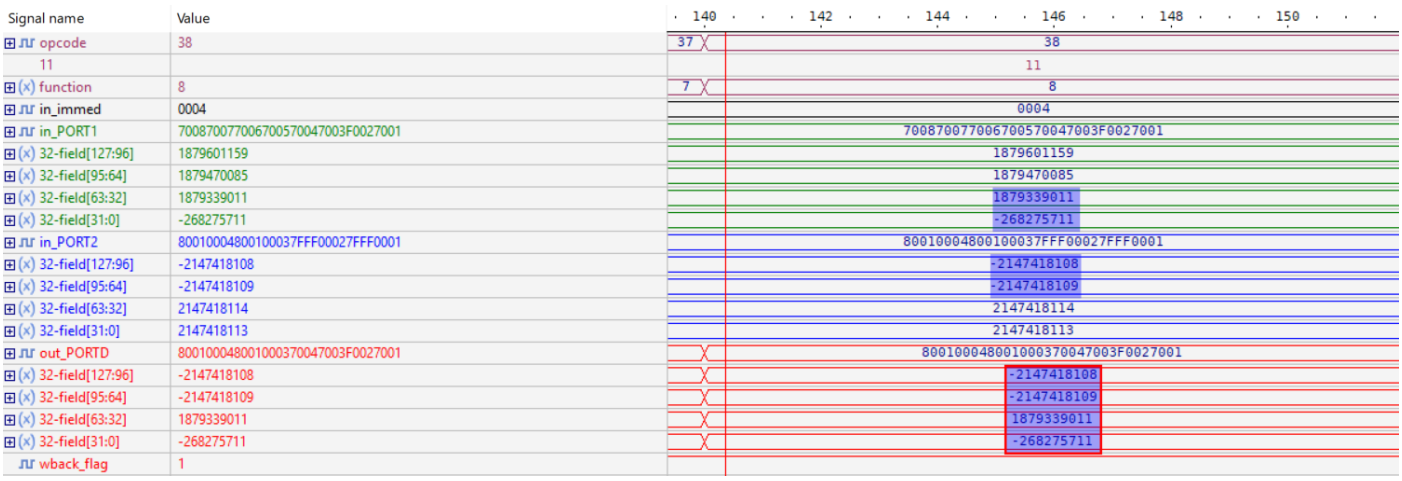
Operation:

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

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

Verification:

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



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

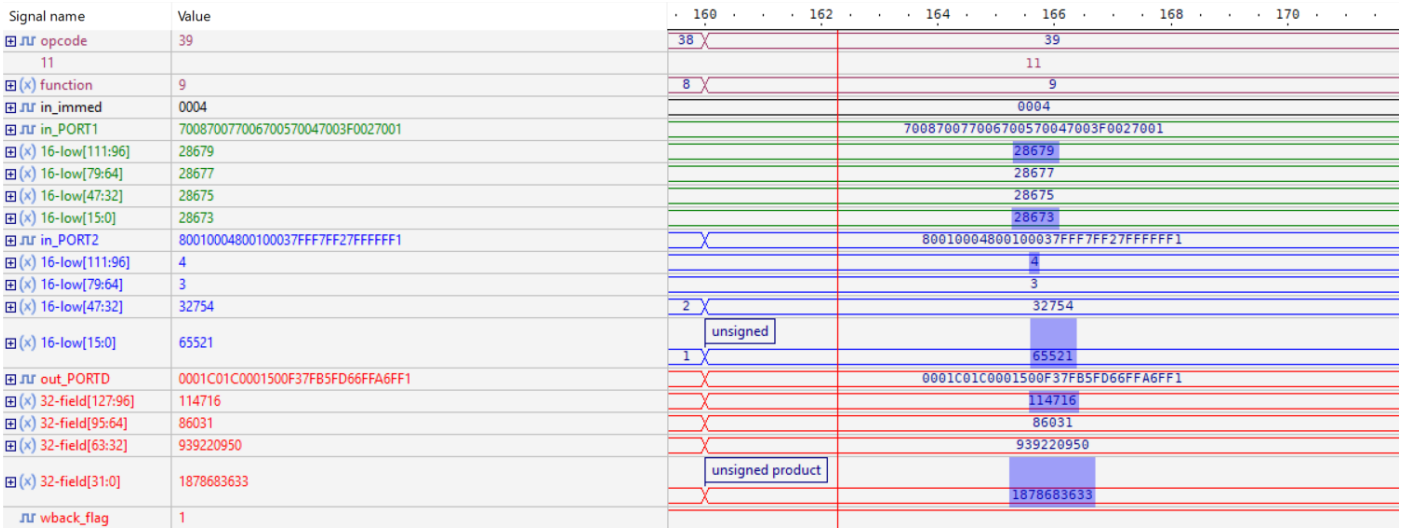
Operation:

$$\begin{aligned}
\text{in_PORT1} &\leftarrow \text{FILE[rs1]} \\
\text{in_PORT2} &\leftarrow \text{FILE[rs2]} \\
\text{out_portD[31-fields]} &\leftarrow \text{UnsignedMult}\big(\text{in_PORT1[16-fields]}, \text{in_PORT2[16-fields]}\big) \\
\text{FILE[rd]} &\leftarrow \text{out_PORTD}, \\
\text{wback_flag} &\leftarrow 1
\end{aligned}$$

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 hexdecimal values.



MLHU – Multiple Low by constant Unsigned

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

Instruction:

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

Opcode:

5 4	3 0
11	1010

Operation:

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

$$\text{in\_immed} \leftarrow \text{ZeroExtend(rs2)}$$

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

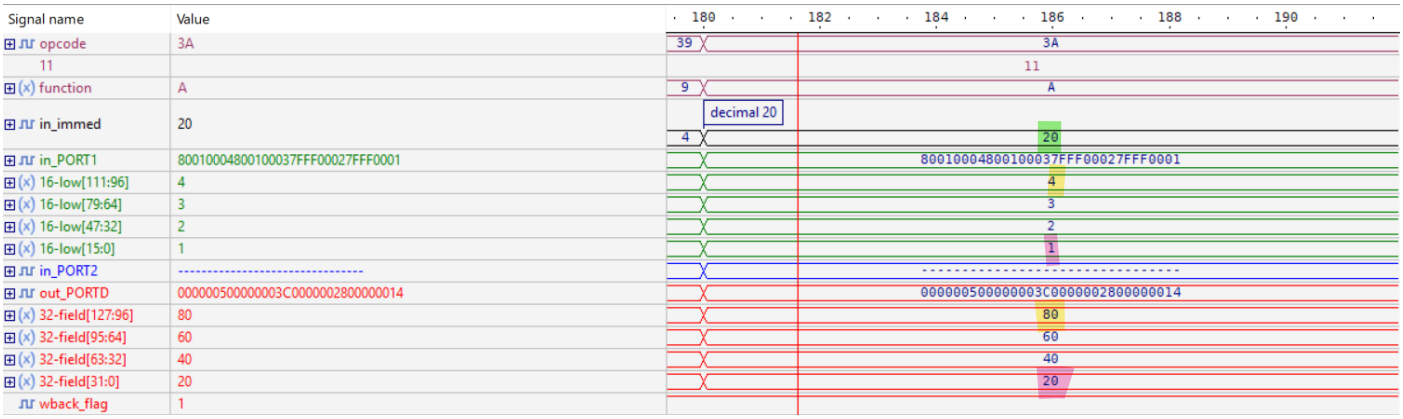
$$\text{FILE[rd]} \leftarrow \text{out\_PORTD},$$

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

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

Verification:

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



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:

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

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

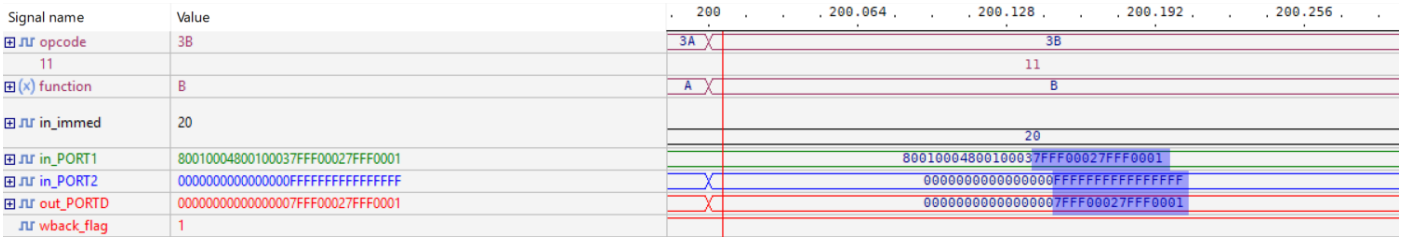
$$\text{out\_portD[31-fields]} \leftarrow \text{in\_PORT2} \wedge \text{in\_PORT1}$$

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

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

Verification:

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



CLZW – Count Leading Zeros in Words

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

Instruction:

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

Opcode:

5 4	3 0
11	1100

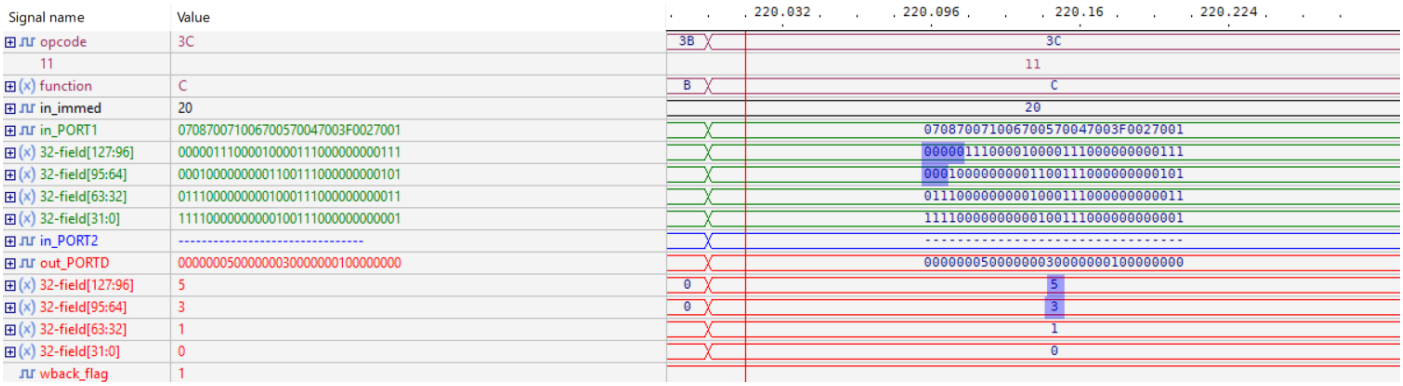
Operation:

$$\begin{aligned} \text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\ \text{out\_portD[31-fields]} &\leftarrow \text{CLZ}_{32}\big(\text{in\_PORT1[32-field]}\big) \\ \text{FILE[rd]} &\leftarrow \text{out\_PORTD}, \\ \text{wback\_flag} &\leftarrow 1 \end{aligned}$$

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

Verification:

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



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

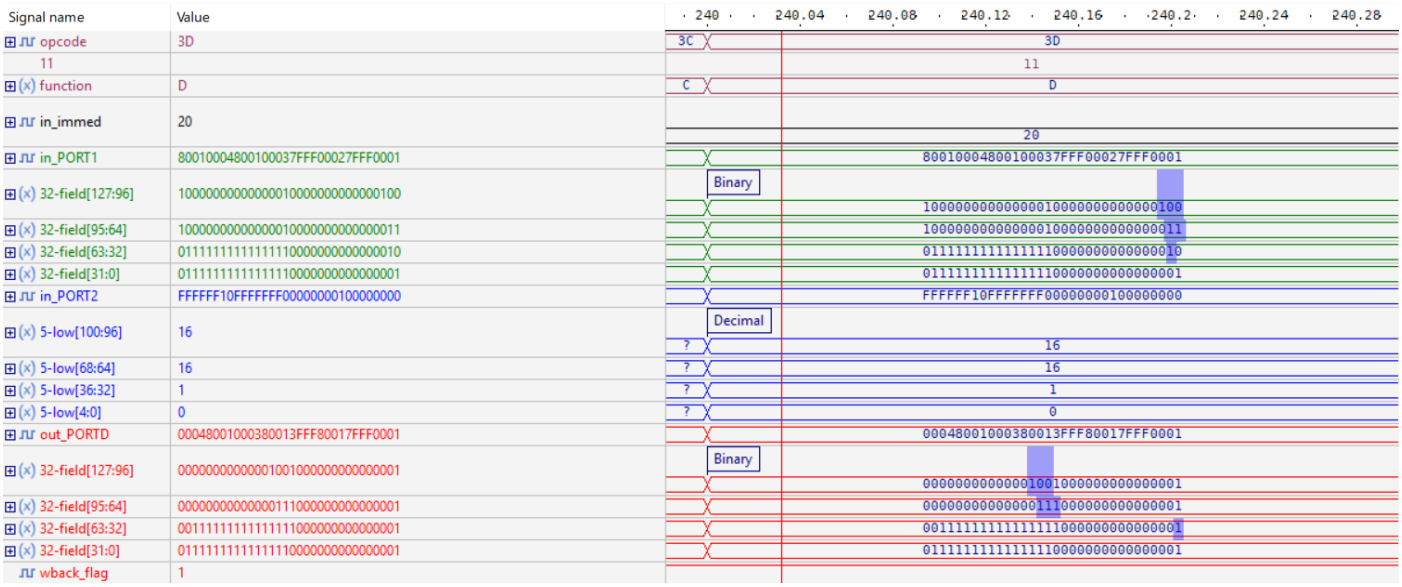
Operation:

$$
\begin{aligned}
\text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\
\text{in\_PORT2} &\leftarrow \text{FILE[rs2]} \\
\text{int\_var} &\leftarrow \text{Unsigned(in\_PORT2[5-low])} \bmod 32 \\
\text{out\_portD[31-fields]} &\leftarrow \text{RotateRight}\big(\text{in\_PORT1[32-field, int\_var}\big) \\
\text{FILE[rd]} &\leftarrow \text{out\_PORTD,} \\
\text{wback\_flag} &\leftarrow 1
\end{aligned}
$$

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 hexdecimal values.



SFWU – Subtract from Word Unsigned

Description:
Packed 32-bit word unsigned subtract of the contentsof 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

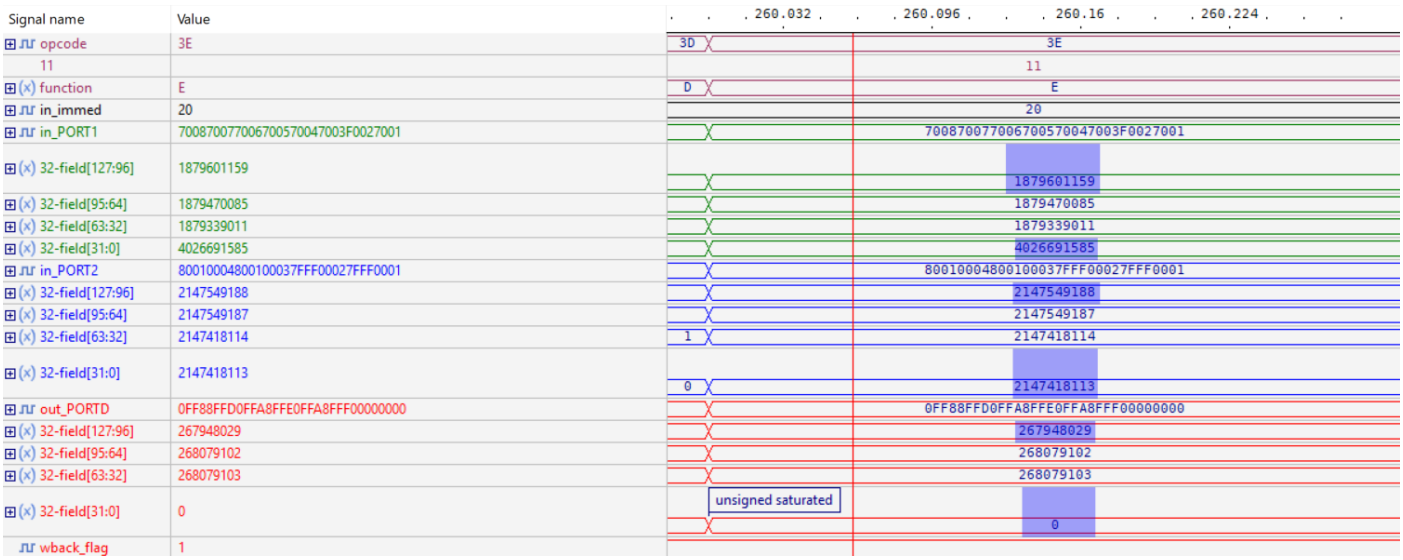
Operation:

$$
\begin{aligned}
\text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\
\text{in\_PORT2} &\leftarrow \text{FILE[rs2]} \\
\text{out\_portD[31-fields]} &\leftarrow \text{Unsigned}\big(\text{in\_PORT2[32-fields]} - \text{in\_PORT1[32-fields]}\big) \\
\text{FILE[rd]} &\leftarrow \text{out\_PORTD}, \\
\text{wback\_flag} &\leftarrow 1
\end{aligned}
$$

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 hexdecimal values.



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

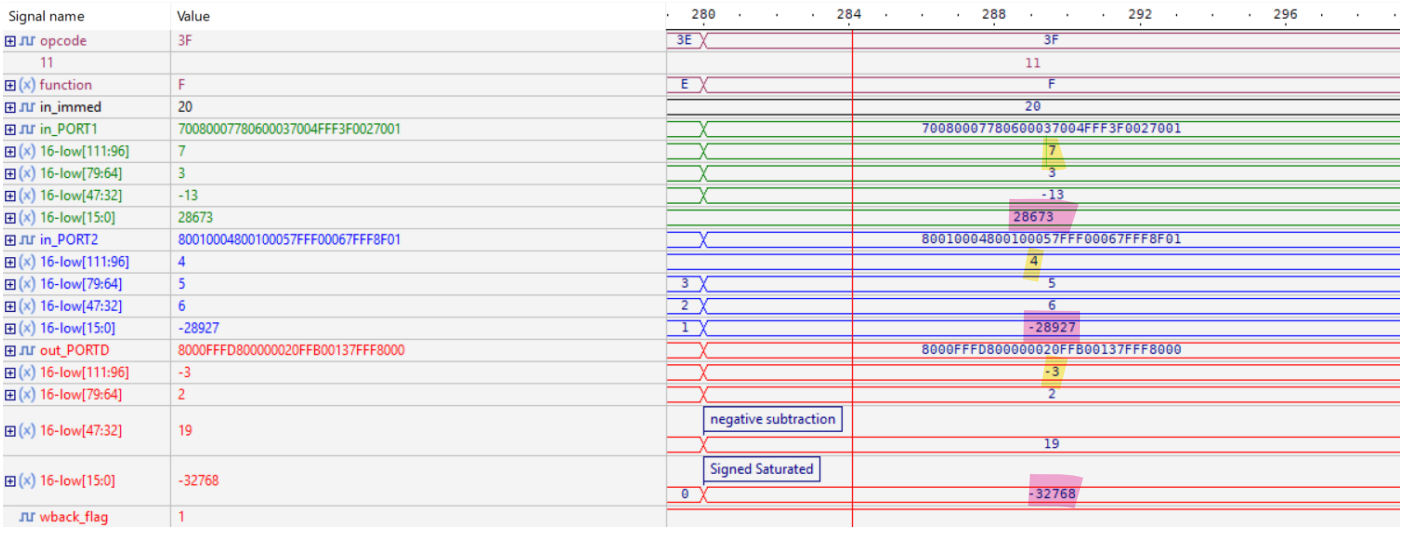
Operation:

$$
\begin{aligned}
\text{in\_PORT1} &\leftarrow \text{FILE[rs1]} \\
\text{in\_PORT2} &\leftarrow \text{FILE[rs2]} \\
\text{out\_portD[31-fields]} &\leftarrow \text{SignedSaturate}\big(\text{in\_PORT2[16-fields]} - \text{in\_PORT1[16-fields]}\big) \\
\text{FILE[rd]} &\leftarrow \text{out\_PORTD}, \\
\text{wback\_flag} &\leftarrow 1
\end{aligned}
$$

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 hexdecimal values.



NOP – No Operation

Description:
Instruction does not write anything to the register file.

Instruction:

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

Opcode:

5 – 4	3 – 0
11	0000

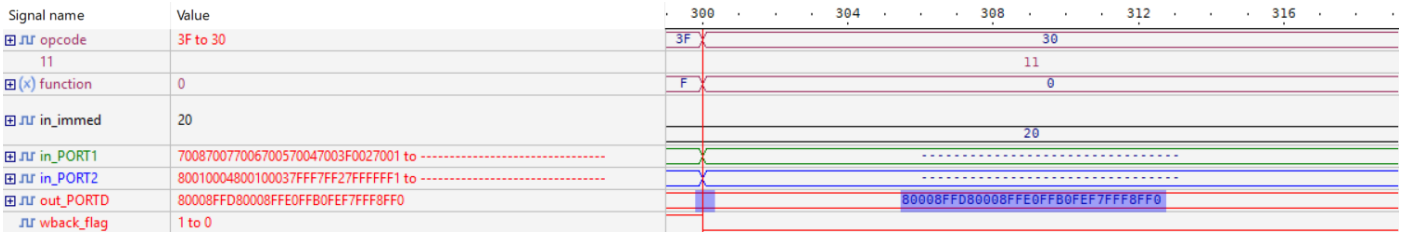
Operation:

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

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

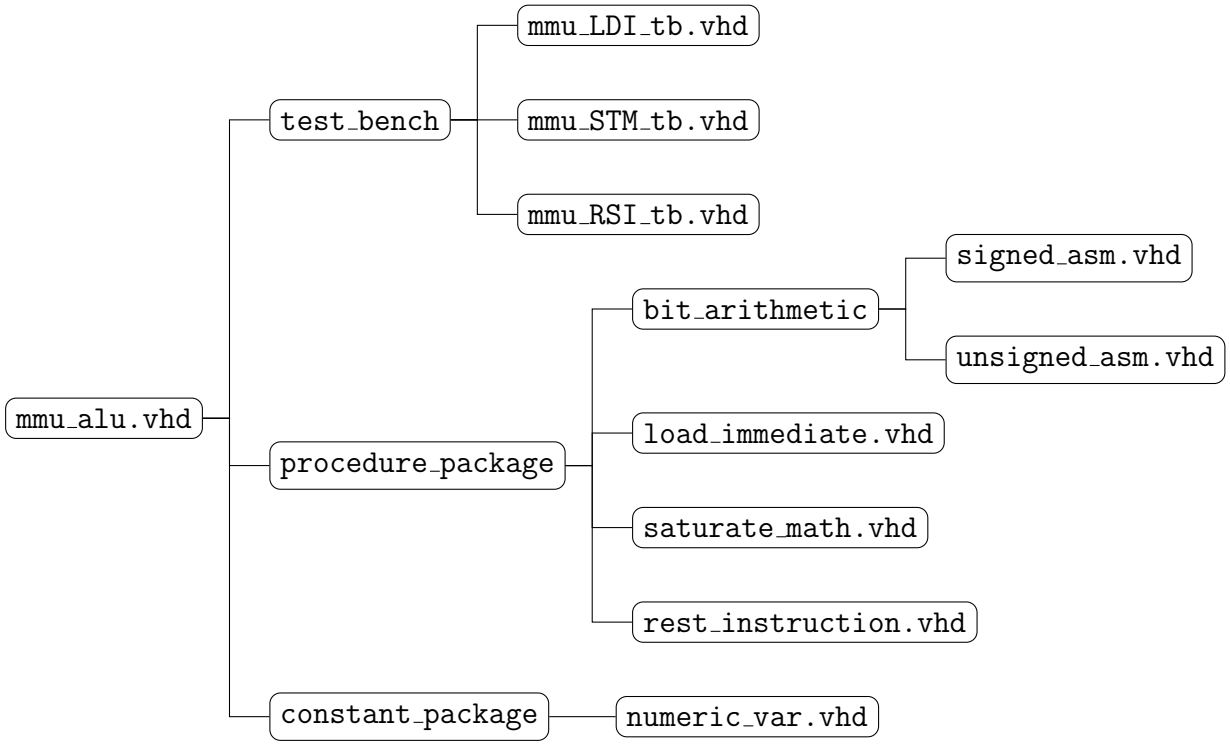
Verification:

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



4 Appendix

4.1 File Structure



4.2 Source File: constant_package/numeric_var

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package numeric_var is

--------INSTRUCTION_FIELD_CONSTANT-----------------------------------------------------
  constant IMMEDIATE_LENGTH           : integer := 16;
  constant INDEX_LENGTH       : integer := 3;
  constant OPCODE_LENGTH        : integer := 6;
  constant INSTRUCTION_LENGTH     : integer := 25;

--------REGISTER_FILE_CONSTANT---------------------------------------------------------
  constant VALUE16          : integer := 16;
  constant ADDRESS_LENGTH       : integer := 5;
  constant REGISTER_LENGTH      : integer := 128;
end package;
```

4.3 Source File: mmu_alu.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.numeric_var.all;
use work.load_immediate.all;
use work.saturate_math.all;
use work.rest_instruction.all;


entity MMU_ALU is
  port (
  opcode    : in std_logic_vector(OPCODE_LENGTH-1 downto 0);

  in_PORT3  : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
  in_PORT2  : in std_logic_vector(REGISTER_LENGTH-1 downto 0);
  in_PORT1  : in std_logic_vector(REGISTER_LENGTH-1 downto 0);

  in_immed  : in std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
  in_d_ptr  : in std_logic_vector(ADDRESS_LENGTH-1 downto 0);

  out_PORTD : out std_logic_vector(REGISTER_LENGTH-1 downto 0);
  out_d_ptr   : out std_logic_vector(ADDRESS_LENGTH-1 downto 0);
  wback_flag  : out std_logic
  );
end entity;

architecture behavior of MMU_ALU is
begin
  main : process(
    opcode,
    in_PORT3, in_PORT2, in_PORT1, in_immed,
    in_d_ptr
    )
  begin
    out_d_ptr <= in_d_ptr;
    case opcode(OPCODE_LENGTH-1 downto OPCODE_LENGTH-2) is
      when "00" | "01" =>
        LDI_memory(     --ref. procedure_package/load_immediate.vhd
          opcode,
          in_PORT3,
          in_immed,

          out_PORTD,
          wback_flag
        );

      when "10" =>
        STM_main(     --ref. procedure_package/saturate_math.vhd
          opcode,
```

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

4.4 Source File: procedure_package/load_immediate.vhd

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

4.5 Source File: procedure_package/saturated_math.vhd

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

```
            in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
            ret32,
            temp_out(16*(i*2+2)-1 downto 16*(i*2))
          );
        end loop;

      when "011" =>
        for i in 3 downto 0 loop --high 16-bit integer mult-sub
          mult_16(
            in_PORT3(16*(i*2+2)-1 downto 16*(i*2+1)),
            in_PORT2(16*(i*2+2)-1 downto 16*(i*2+1)),
            ret32
          );
          sub_32(
            in_PORT1(16*(i*2+2)-1 downto 16*(i*2)),
            ret32,
            temp_out(16*(i*2+2)-1 downto 16*(i*2))
          );
        end loop;

      when "100" =>
        for i in 1 downto 0 loop --low 32-bit integer mult-add
          mult_32(
            in_PORT3(32*(i*2+1)-1 downto 32*(i*2)),
            in_PORT2(32*(i*2+1)-1 downto 32*(i*2)),
            ret64
          );
          add_64(
            in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
            ret64,
            temp_out(32*(i*2+2)-1 downto 32*(i*2))
          );
        end loop;

      when "101" =>
        for i in 1 downto 0 loop --high 32-bit integer mult-add
          mult_32(
            in_PORT3(32*(i*2+2)-1 downto 32*(i*2+1)),
            in_PORT2(32*(i*2+2)-1 downto 32*(i*2+1)),
            ret64
          );
          add_64(
            in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
            ret64,
            temp_out(32*(i*2+2)-1 downto 32*(i*2))
          );
        end loop;

      when "110" =>
        for i in 1 downto 0 loop --low 32-bit integer mult-sub
          mult_32(
            in_PORT3(32*(i*2+1)-1 downto 32*(i*2)),
            in_PORT2(32*(i*2+1)-1 downto 32*(i*2)),
            ret64
          );
          sub_64(
            in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
            ret64,
            temp_out(32*(i*2+2)-1 downto 32*(i*2))
          );
        end loop;

      when "111" =>
        for i in 1 downto 0 loop --high 32-bit integer mult-sub
          mult_32(
            in_PORT3(32*(i*2+2)-1 downto 32*(i*2+1)),
            in_PORT2(32*(i*2+2)-1 downto 32*(i*2+1)),
            ret64
          );
          sub_64(
            in_PORT1(32*(i*2+2)-1 downto 32*(i*2)),
            ret64,
            temp_out(32*(i*2+2)-1 downto 32*(i*2))
```

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

4.6 Source File: procedure_package/rest_instruction.vhd

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

```
59          when "0011" => --count 1s in halfword
60              for i in 0 to 7 loop
61                  vector16 := in_PORT1(16*(i+1)-1 downto i*16);
62                  int_var := 0;
63                  for j in 0 to 15 loop
64                      if vector16(j) = '1' then
65                          int_var := int_var + 1;
66                      end if;
67                  end loop;
68                  temp_out(16*(i+1)-1 downto i*16) := std_logic_vector(to_unsigned(int_var,
                        16));
69              end loop;
70
71          when "0100" => --add halfword saturated
72              for i in 0 to 7 loop
73                  add_16(
74                      in_PORT2(16*(i+1)-1 downto i*16),
75                      in_PORT1(16*(i+1)-1 downto i*16),
76                      temp_out(16*(i+1)-1 downto i*16)
77                  );
78              end loop;
79
80          when "0101" => --bitwise logical or
81              temp_out := in_PORT2 or in_PORT1;
82
83          when "0110" => --broadcast word
84              for i in 0 to 3 loop
85                  temp_out(32*(i+1)-1 downto i*32) := in_PORT1(REGISTER_LENGTH-1 downto
                        REGISTER_LENGTH-32);
86              end loop;
87
88          when "0111" => -- max signed word
89              for i in 0 to 3 loop
90                  if signed(in_PORT2(32*(i+1)-1 downto i*32)) > signed(in_PORT1(32*(i+1)-1
                        downto i*32)) then
91                     temp_out(32*(i+1)-1 downto i*32) := in_PORT2(32*(i+1)-1 downto i*32);
92                  else
93                     temp_out(32*(i+1)-1 downto i*32) := in_PORT1(32*(i+1)-1 downto i*32);
94                  end if;
95              end loop;
96
97          when "1000" => --min signed word
98              for i in 0 to 3 loop
99                  if signed(in_PORT2(32*(i+1)-1 downto i*32)) < signed(in_PORT1(32*(i+1)-1
                        downto i*32)) then
100                    temp_out(32*(i+1)-1 downto i*32) := in_PORT2(32*(i+1)-1 downto i*32);
101                 else
102                    temp_out(32*(i+1)-1 downto i*32) := in_PORT1(32*(i+1)-1 downto i*32);
103                 end if;
104             end loop;
105
106         when "1001" => --multiply low unsigned
107             for i in 0 to 3 loop
108                 mult_16_unsigned(
109       in_PORT1(16*(i*2+1)-1 downto 16*(i*2)),
110                     in_PORT2(16*(i*2+1)-1 downto 16*(i*2)),
111                     temp_out(32*(i+1)-1 downto 32*i)
112                 );
113             end loop;
114
115     when "1010" => --multiply low by constant unsigned
116         for i in 0 to 3 loop
117         mult_16_unsigned(
118           in_PORT1(16*(i*2+1)-1 downto 16*(i*2)),
119           "00000000000" & in_immed(4 downto 0),
120           temp_out(32*(i+1)-1 downto 32*i)
121         );
122       end loop;
123
124         when "1011" => --bitwise logical and
125             temp_out := in_PORT2 and in_PORT1;
126
127         when "1100" => --count leading zeroes in words
```

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

4.7 Source File: procedure_package/bit_arithmetic/unsigned.vhd

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

```
  procedure add_32_unsigned(
    a32, b32  : in std_logic_vector(31 downto 0);
    ret32     : out std_logic_vector(31 downto 0));

  procedure sub_32_unsigned(
    a32, b32  : in std_logic_vector(31 downto 0);
    ret32     : out std_logic_vector(31 downto 0));

end package unsigned_asm;

package body unsigned_asm is
  constant MAX_32_unsigned : unsigned(31 downto 0) := (others => '1');
  constant MIN_32_unsigned : unsigned(31 downto 0) := (others => '0');

-----Unsigned-Multiple_16-bits-------------------------------------------------------
  procedure mult_16_unsigned(
    a16, b16  : in std_logic_vector(15 downto 0); -- 32 bits or half-long
    ret32     : out std_logic_vector(31 downto 0)
  ) is
    variable prod : unsigned(31 downto 0);
  begin
    prod := unsigned(a16) * unsigned(b16);
    ret32 := std_logic_vector(prod);
  end procedure;

-----Unsigned-Addition_32-bits-------------------------------------------------------
  procedure add_32_unsigned(
    a32, b32  : in std_logic_vector(31 downto 0);
    ret32     : out std_logic_vector(31 downto 0)
  ) is
    variable sum : unsigned(32 downto 0);
  begin
      sum := unsigned('0' & a32) + unsigned('0' & b32);

      if sum(32) = '1' then
          ret32 := std_logic_vector(MAX_32_UNSIGNED);
      else
          ret32 := std_logic_vector(sum(31 downto 0));
      end if;
  end procedure;

-----Unsigned-Subtraction_32-bits----------------------------------------------------
  procedure sub_32_unsigned(
    a32, b32  : in std_logic_vector(31 downto 0);
    ret32     : out std_logic_vector(31 downto 0)
  ) is
    variable diff : unsigned(32 downto 0);
  begin
    diff :=  unsigned('0' & a32) - unsigned('0' & b32);

      if diff(32) = '1' then
          ret32 := std_logic_vector(MIN_32_UNSIGNED);
      else
          ret32 := std_logic_vector(diff(31 downto 0));
      end if;
  end procedure;
end package body unsigned_asm;
```

4.8 Source File: procedure_package/bit_arithmetic/signed.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package signed_asm is
  procedure mult_16(
    a16, b16  : in std_logic_vector(15 downto 0);
    ret32     : out std_logic_vector(31 downto 0));

  procedure mult_32(
    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;
```

4.9 Test-Bench File: test_bench/mmu_LDI_tb.vhd

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

```
        when 2  => result(num_nibbles - i) := '2';
        when 3  => result(num_nibbles - i) := '3';
        when 4  => result(num_nibbles - i) := '4';
        when 5  => result(num_nibbles - i) := '5';
        when 6  => result(num_nibbles - i) := '6';
        when 7  => result(num_nibbles - i) := '7';
        when 8  => result(num_nibbles - i) := '8';
        when 9  => result(num_nibbles - i) := '9';
        when 10 => result(num_nibbles - i) := 'A';
        when 11 => result(num_nibbles - i) := 'B';
        when 12 => result(num_nibbles - i) := 'C';
        when 13 => result(num_nibbles - i) := 'D';
        when 14 => result(num_nibbles - i) := 'E';
        when 15 => result(num_nibbles - i) := 'F';
        when others => result(num_nibbles - i) := 'X';
      end case;
    end loop;
    return result;
  end function;
    ------------------------------------------------------------------
begin

  UUT : entity work.MMU_ALU
    port map(
        opcode      => opcode,

        in_PORT3    => in_PORT3,
        in_PORT2    => in_PORT2,
        in_PORT1    => in_PORT1,
        in_immed    => in_immed,
        in_d_ptr    => in_d_ptr,

        out_PORTD   => out_PORTD,
        out_d_ptr   => out_d_ptr,
        wback_flag  => wback_flag
    );

  -- stimulus process
  stim_proc : process
  begin
-------------------------------------------------------------------
-- load_immediate TEST w/ indexing
-------------------------------------------------------------------
    in_immed <= x"DEAD";
    in_d_ptr <= b"00000";
      in_PORT3 <= (others => '0');
  ------------------------------------------------------------------
  -- TEST: opcode = 0--000
  ------------------------------------------------------------------
        opcode <= std_logic_vector(to_unsigned(0, OPCODE_LENGTH));
        wait for period;
        assert out_PORTD(15 downto 0) = x"DEAD"
    report "Test failed: 000000, out_PORTD = x" & slv_to_hex(out_PORTD)
        severity error;

  ------------------------------------------------------------------
  -- TEST: opcode = 0--001
  ------------------------------------------------------------------
        opcode <= std_logic_vector(to_unsigned(1, OPCODE_LENGTH));
        wait for period;
        assert out_PORTD(31 downto 16) = x"DEAD"
            report "Test failed: 000001, out_PORTD = x" & slv_to_hex(out_PORTD)
            severity error;

  ------------------------------------------------------------------
  -- TEST: opcode = 0--110
  ------------------------------------------------------------------
        opcode <= std_logic_vector(to_unsigned(6, OPCODE_LENGTH));
        wait for period;
        assert out_PORTD(111 downto 96) = x"DEAD"
            report "Test failed: 100110, out_PORTD = x" & slv_to_hex(out_PORTD)
            severity error;

```

```
    ---------------------------------------------------------------------
    -- TEST: opcode = 0--111
    ---------------------------------------------------------------------
          opcode <= std_logic_vector(to_unsigned(7, OPCODE_LENGTH));
          wait for period;
          assert out_PORTD(127 downto 112) = x"DEAD"
              report "Test failed: 100111, out_PORTD = x" & slv_to_hex(out_PORTD)
              severity error;

          report "TEST COMPLETED: load_immediate w/ indexing" severity warning;
    end process;
end test_bench;
```

4.10 Test-Bench File: test_bench/mmu_STM_tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.numeric_var.all;
use work.all;

entity mmu_STM_tb is
end mmu_STM_tb;

architecture test_bench of mmu_STM_tb is
    -- Inputs to ALU
    signal opcode      : std_logic_vector(OPCODE_LENGTH-1 downto 0);
    signal in_PORT3    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
    signal in_PORT2    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
    signal in_PORT1    : std_logic_vector(REGISTER_LENGTH-1 downto 0);
    signal in_immed    : std_logic_vector(IMMEDIATE_LENGTH-1 downto 0);
    signal in_d_ptr    : std_logic_vector(ADDRESS_LENGTH-1 downto 0);

    -- Outputs ALU
    signal out_PORTD   : std_logic_vector(REGISTER_LENGTH-1 downto 0);
    signal out_d_ptr   : std_logic_vector(ADDRESS_LENGTH-1 downto 0);
    signal wback_flag  : std_logic;

  constant period: time := 20ns;
    ---------------------------------------------------------------------
    -- Helper: Convert std_logic_vector ? hex string (portable)
    ---------------------------------------------------------------------
    function slv_to_hex(slv : std_logic_vector) return string is
    variable num_nibbles : integer := (slv'length + 3) / 4;
          variable padded_slv  : std_logic_vector(num_nibbles * 4 - 1 downto 0);
          variable result      : string(1 to num_nibbles);
          variable nibble_val  : integer;
  begin
      -- Pad MSBs with zeros if not multiple of 4 bits
      padded_slv := (others => '0');
      padded_slv(slv'length - 1 downto 0) := slv;

      for i in 0 to num_nibbles - 1 loop
        nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
        case nibble_val is
        when 0  => result(num_nibbles - i) := '0';
        when 1  => result(num_nibbles - i) := '1';
        when 2  => result(num_nibbles - i) := '2';
        when 3  => result(num_nibbles - i) := '3';
        when 4  => result(num_nibbles - i) := '4';
        when 5  => result(num_nibbles - i) := '5';
        when 6  => result(num_nibbles - i) := '6';
        when 7  => result(num_nibbles - i) := '7';
        when 8  => result(num_nibbles - i) := '8';
        when 9  => result(num_nibbles - i) := '9';
        when 10 => result(num_nibbles - i) := 'A';
        when 11 => result(num_nibbles - i) := 'B';
        when 12 => result(num_nibbles - i) := 'C';
        when 13 => result(num_nibbles - i) := 'D';
        when 14 => result(num_nibbles - i) := 'E';
        when 15 => result(num_nibbles - i) := 'F';
```

```
57          when others => result(num_nibbles - i) := 'X';
58        end case;
59      end loop;
60      return result;
61    end function;
62      ---------------------------------------------------------------------

63
64  begin
65    UUT : entity work.MMU_ALU
66      port map(
67        opcode      => opcode,
68        in_PORT3    => in_PORT3,
69        in_PORT2    => in_PORT2,
70        in_PORT1    => in_PORT1,
71        in_immed    => in_immed,
72        in_d_ptr    => in_d_ptr,
73
74        out_PORTD   => out_PORTD,
75        out_d_ptr   => out_d_ptr,
76        wback_flag  => wback_flag
77      );
78
79    -- Stimulus process
80    stim_proc : process
81    begin
82  ------------------------------------------------------------------
83  -- saturate_math TEST w/overflow and underflow checks
84  ------------------------------------------------------------------
85      in_immed <= x"DEAD";
86      in_d_ptr <= b"00000";
87      in_PORT3 <= x"0001000200030004700570060F0077008";
88      in_PORT2 <= x"0008000700060005700470030F0027001";
89      in_PORT1 <= x"0000000000000007FFF00007FFF0000";
90
91      ---------------------------------------------------------------------
92      -- TEST: Opcode 10-000
93      ---------------------------------------------------------------------
94      opcode <= "100000";
95      wait for period;
96      assert out_PORTD = x"0000000E000000147FFFFFFF7FFFFFFF"
97          report "Test failed: 100000, out_PORTD = x" & slv_to_hex(out_PORTD)
98          severity error;
99
100     ---------------------------------------------------------------------
101     -- TEST: Opcode 10-001
102     ---------------------------------------------------------------------
103         opcode <= "100001";
104         wait for period;
105         assert out_PORTD = x"00000008000000127FFFFFFF7FFFFFFF"
106             report "Test failed: 100001, out_PORTD = x" & slv_to_hex(out_PORTD)
107             severity error;
108
109   ---------------------------------------------------------------------
110   -- TEST: Opcode 10-010
111   ---------------------------------------------------------------------
112     in_PORT1 <= x"00000000000000008001000080010000";
113     opcode <= "100010";
114     wait for period;
115     assert out_PORTD = x"FFFFFFF2FFFFFFEC8000000080000000"
116         report "Test failed: 100010, out_PORTD = x" & slv_to_hex(out_PORTD)
117         severity error;
118
119   ---------------------------------------------------------------------
120   -- TEST: Opcode 10-011
121   ---------------------------------------------------------------------
122         opcode <= "100011";
123         wait for period;
124         assert out_PORTD = x"FFFFFFF8FFFFFFEE8000000080000000"
125             report "Test failed: 100011, out_PORTD = x" & slv_to_hex(out_PORTD)
126             severity error;
127
128   ---------------------------------------------------------------------
129   -- TEST: Opcode 10-100
```

```
130   ------------------------------------------------------------------
131     in_PORT1 <= x"00000000000000007FFF00007FFF0000";
132     opcode <= "100100";
133     wait for period;
134     assert out_PORTD = x"00000012002700147FFFFFFFFFFFFFF"
135       report "Test failed: 100100, out_PORTD = x" & slv_to_hex(out_PORTD)
136       severity error;
137
138   ------------------------------------------------------------------
139   -- TEST: Opcode 10-101
140   ------------------------------------------------------------------
141         opcode <= "100101";
142         wait for period;
143         assert out_PORTD = x"0000000800170000E7FFFFFFFFFFFFFF"
144             report "Test failed: 100101, out_PORTD = x" & slv_to_hex(out_PORTD)
145             severity error;
146
147   ------------------------------------------------------------------
148   -- TEST: Opcode 10-110
149   ------------------------------------------------------------------
150     in_PORT1 <= x"00000000000000008001000080010000";
151     opcode <= "100110";
152     wait for period;
153     assert out_PORTD = x"FFFFFFEDFFD8FFEC8000000000000000"
154         report "Test failed: 100110, out_PORTD = x" & slv_to_hex(out_PORTD)
155         severity error;
156
157   ------------------------------------------------------------------
158   -- TEST: Opcode 10-111
159   ------------------------------------------------------------------
160     opcode <= "100111";
161     wait for period;
162     assert out_PORTD = x"FFFFFFF7FFE8FFF280000000000000000"
163         report "Test failed: 100111, out_PORTD = x" & slv_to_hex(out_PORTD)
164         severity error;
165
166     report "TEST COMPLETED: saturate_math w/o saturating" severity warning;
167   end process;
168 end test_bench;
```

4.11 Test-Bench File: test_bench/mmu_RSI_tb.vhd

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

```
30            variable padded_slv  : std_logic_vector(num_nibbles * 4 - 1 downto 0);
31            variable result      : string(1 to num_nibbles);
32            variable nibble_val  : integer;
33    begin
34        -- Pad MSBs with zeros if not multiple of 4 bits
35        padded_slv := (others => '0');
36        padded_slv(slv'length - 1 downto 0) := slv;
37
38        for i in 0 to num_nibbles - 1 loop
39          nibble_val := to_integer(unsigned(padded_slv((i+1)*4 - 1 downto i*4)));
40          case nibble_val is
41          when 0  => result(num_nibbles - i) := '0';
42          when 1  => result(num_nibbles - i) := '1';
43          when 2  => result(num_nibbles - i) := '2';
44          when 3  => result(num_nibbles - i) := '3';
45          when 4  => result(num_nibbles - i) := '4';
46          when 5  => result(num_nibbles - i) := '5';
47          when 6  => result(num_nibbles - i) := '6';
48          when 7  => result(num_nibbles - i) := '7';
49          when 8  => result(num_nibbles - i) := '8';
50          when 9  => result(num_nibbles - i) := '9';
51          when 10 => result(num_nibbles - i) := 'A';
52          when 11 => result(num_nibbles - i) := 'B';
53          when 12 => result(num_nibbles - i) := 'C';
54          when 13 => result(num_nibbles - i) := 'D';
55          when 14 => result(num_nibbles - i) := 'E';
56          when 15 => result(num_nibbles - i) := 'F';
57          when others => result(num_nibbles - i) := 'X';
58        end case;
59      end loop;
60      return result;
61    end function;
62      ---------------------------------------------------------------------
63
64  begin
65    UUT : entity work.MMU_ALU
66      port map(
67        opcode      => opcode,
68        in_PORT3    => in_PORT3,
69        in_PORT2    => in_PORT2,
70        in_PORT1    => in_PORT1,
71        in_immed    => in_immed,
72        in_d_ptr    => in_d_ptr,
73
74        out_PORTD   => out_PORTD,
75        out_d_ptr   => out_d_ptr,
76        wback_flag  => wback_flag
77      );
78
79      -- Clock process
80
81
82    -- Stimulus process
83    stim_proc : process
84    begin
85  ------------------------------------------------------------------
86  -- rest_instruction TEST
87  ------------------------------------------------------------------
88    in_d_ptr <= b"00000";
89
90    ---------------------------------------------------------------------
91      -- TEST: Opcode 110001
92      ---------------------------------------------------------------------
93    opcode <= "110001";
94    in_immed <= x"0004";
95    in_PORT2 <=(others => '-');
96    in_PORT1 <= x"700870077006700570047003F0027001";
97    wait for period;
98    assert out_PORTD = x"070007000700070007000700F000700"
99    report "TEST FAIL: 110001, out_PORTD =" & slv_to_hex(out_PORTD)
100   severity error;
101   ---------------------------------------------------------------------
102     -- TEST: Opcode 110010, stauturated
```

```
    ------------------------------------------------------------------
  opcode <= "110010";
  in_PORT2 <= x"80010004800100037FFF00027FFF0001";
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"F009700BF0077008F0037005FFFFFFFF"
  report "TEST FAIL: 110010, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 110011
    ------------------------------------------------------------------
  opcode <= "110011";
  in_PORT2 <= (others => '-');
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"00040006000500050004000500050004"
  report "TEST FAIL: 110011, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 110100
    ------------------------------------------------------------------
  opcode <= "110100";
  in_PORT2 <= x"80010004800100037FFF7FF27FFFFFF1";
  in_PORT1 <= x"70087007700680057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"F009700BF00780087FFF7FFF70016FF2"
  report "TEST FAIL: 110100, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 110101
    ------------------------------------------------------------------
  opcode <= "110101";
  in_PORT2 <= x"0000000000000000FFFFFFFFFFFFFFFF";
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"7008700770067005FFFFFFFFFFFFFFFF"
  report "TEST FAIL: 110101, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 110110
    ------------------------------------------------------------------
  opcode <= "110110";
  in_PORT2 <= (others => '-');
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"70087007700870077008700770087007"
  report "TEST FAIL: 110110, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 110111
    ------------------------------------------------------------------
  opcode <= "110111";
  in_PORT2 <= x"80010004800100037FFF00027FFF0001";
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"70087007700670057FFF00027FFF0001"
  report "TEST FAIL: 110111, out_PORTD =" & slv_to_hex(out_PORTD)
  severity error;

  ------------------------------------------------------------------
    -- TEST: Opcode 111000
    ------------------------------------------------------------------
  opcode <= "111000";
  in_PORT2 <= x"80010004800100037FFF00027FFF0001";
  in_PORT1 <= x"70087007700670057004700 3F0027001";
  wait for period;
  assert out_PORTD = x"80010004800100037004700 3F0027001"
  report "TEST FAIL: 111000, out_PORTD =" & slv_to_hex(out_PORTD)
```

```
176   severity error;
177
178   -------------------------------------------------------------------
179     -- TEST: Opcode 111001
180     -------------------------------------------------------------------
181   opcode <= "111001";
182   in_PORT2 <= x"80010004800100037FFF7FF27FFFFFF1";
183   in_PORT1 <= x"70087007700670057004700 3F0027001";
184   wait for period;
185   assert out_PORTD = x"0001C01C0001500F37FB5FD66FFA6FF1"
186   report "TEST FAIL: 111001, out_PORTD =" & slv_to_hex(out_PORTD)
187   severity error;
188
189   -------------------------------------------------------------------
190     -- TEST: Opcode 111010
191     -------------------------------------------------------------------
192   opcode <= "111010";
193   in_immed <= x"0014";
194   in_PORT2 <= (others => '-');
195   in_PORT1 <= x"80010004800100037FFF00027FFF0001";
196   wait for period;
197   assert out_PORTD = x"0000005000000003C0000002800000014"
198   report "TEST FAIL: 111010, out_PORTD =" & slv_to_hex(out_PORTD)
199   severity error;
200
201   -------------------------------------------------------------------
202     -- TEST: Opcode 111011
203     -------------------------------------------------------------------
204   opcode <= "111011";
205   in_PORT2 <= x"0000000000000000FFFFFFFFFFFFFFFF";
206   in_PORT1 <= x"80010004800100037FFF00027FFF0001";
207   wait for period;
208   assert out_PORTD = x"00000000000000007FFF00027FFF0001"
209   report "TEST FAIL: 111011, out_PORTD =" & slv_to_hex(out_PORTD)
210   severity error;
211
212   -------------------------------------------------------------------
213     -- TEST: Opcode 111100
214     -------------------------------------------------------------------
215   opcode <= "111100";
216   in_PORT2 <= (others => '-');
217   in_PORT1 <= x"07087007100670057004700 3F0027001";
218   wait for period;
219   assert out_PORTD = x"00000005000000030000000100000000"
220   report "TEST FAIL: 111100, out_PORTD =" & slv_to_hex(out_PORTD)
221   severity error;
222
223   -------------------------------------------------------------------
224     -- TEST: Opcode 111101
225     -------------------------------------------------------------------
226   opcode <= "111101";
227   in_PORT2 <= x"FFFFFF10FFFFFFFF00000000100000000";
228   in_PORT1 <= x"80010004800100037FFF00027FFF0001";
229   wait for period;
230   assert out_PORTD = x"00048001000380013FFF80017FFF0001"
231   report "TEST FAIL: 111101, out_PORTD =" & slv_to_hex(out_PORTD)
232   severity error;
233
234   -------------------------------------------------------------------
235     -- TEST: Opcode 111110
236     -------------------------------------------------------------------
237   opcode <= "111110";
238   in_PORT2 <= x"80010004800100037FFF00027FFF0001";
239   in_PORT1 <= x"70087007700670057004700 3F0027001";
240   wait for period;
241   assert out_PORTD = x"0FF88FFD0FFA8FFE0FFA8FFF00000000"
242   report "TEST FAIL: 111110, out_PORTD =" & slv_to_hex(out_PORTD)
243   severity error;
244
245   -------------------------------------------------------------------
246     -- TEST: Opcode 111111
247     -------------------------------------------------------------------
248   opcode <= "111111";
```

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