

**HaoRan Yuan 113800771**

**Hunter Zhong 113003581**

## **1. Goal**

### **1.1 Purpose**

The purpose of this project is to explore the application of VHDL/Verilog hardware description languages for the design of a four-stage pipelined multimedia unit. The focus is on implementing a SIMD (Single Instruction, Multiple Data) architecture with a reduced set of multimedia instructions. The design draws inspiration from prominent architectures such as the Sony Cell SPU (Synergistic Processing Unit) and Intel SSE (Streaming SIMD Extensions).

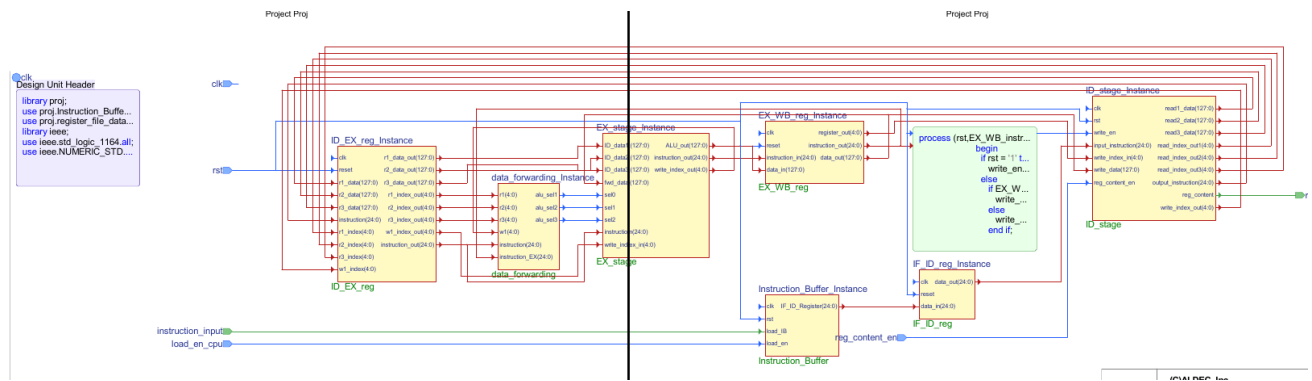
### **1.2 Background**

In the ever-evolving landscape of multimedia processing, the demand for efficient and high-performance solutions is paramount. SIMD architectures have proven to be instrumental in addressing this demand by enabling parallel processing of multiple data elements using a single instruction. The Sony Cell SPU and Intel SSE architectures are exemplary in this regard, showcasing the potential of SIMD in multimedia applications.

### **1.3 Objectives**

1. **Learn VHDL/Verilog:** Gain proficiency in VHDL/Verilog for hardware description and synthesis.
2. **Explore CAD Tools:** Utilize modern CAD tools for structural and behavioral design, simulation, and synthesis.
3. **Implement SIMD Architecture:** Design and implement a SIMD architecture with a focus on multimedia processing.
4. **Pipelined Design:** Develop a four-stage pipelined structure for efficient parallelism.
5. **Instruction Set Design:** Create a reduced set of multimedia instructions inspired by Sony Cell SPU and Intel SSE architectures.

## **2. Block Diagram and Design Procedure**



The design procedure of this CPU is:

CPU

|IF Stage

|IF/ID register

|ID Stage

|ID/EX stage

|EX stage

|EX/WB register

|Data Forwarding Unit

|Simple control logic for write enable on register file

For better structure, we put the ALU and Muxes for ALU into one unit called EX\_stage, similarly, we put the IF stage which contains PC and **instruction buffer** , ID stage which contains a decoder and **register file** into one unit.

Since the control logic is very simple, instead of a separate unit, it was implemented as process inside the top level unit CPU.

```
process(rst, EX_WB_instruction_out)
begin
    if rst = '1' then
        -- Reset condition
        write_en_logic <= '0';
    else
        -- Your pipeline stage detection logic goes here
        if EX_WB_instruction_out(24 downto 23) = "11" and EX_WB_instruction_out(18
downto 15) = "0000" then --NOP statement
            write_en_logic <= '0';
        else
            write_en_logic <= '1';
        end if;
    end if;
end process;
```

## 2.2 Forwarding Unit Design

```

process(instruction, r1, r2, r3, w1)
begin
    if instruction_EX(24 downto 23) = "11" and instruction(18 downto 15) = "0000" then
        alu_sel1 <= '0';
        alu_sel2 <= '0';
        alu_sel3 <= '0';

    elsif instruction(24) = '0' or instruction_EX(24) = '0' then
        -- Compare only r1 with w1
        alu_sel1 <= '1' when r1 = w1 else '0';
        alu_sel2 <= '0';
        alu_sel3 <= '0';
    elsif instruction(24 downto 23) = "10" then
        -- Compare all three registers with w1
        alu_sel1 <= '1' when r1 = w1 else '0';
        alu_sel2 <= '1' when r2 = w1 else '0';
        alu_sel3 <= '1' when r3 = w1 else '0';
    elsif instruction(24 downto 23) = "11" then
        -- Compare only the first two registers with w1
        if instruction(18 downto 15) = "0000" then --nop
            alu_sel1 <= '0';
            alu_sel2 <= '0';
            alu_sel3 <= '0';
        elsif instruction(18 downto 15) = ("0001" or "0011" or "0110" or "1100") then --SHRHI, CNT1H, BCW, INVB
            alu_sel1 <= '1' when r1 = w1 else '0';
            alu_sel2 <= '0';
            alu_sel3 <= '0';
        else
            alu_sel1 <= '1' when r1 = w1 else '0';
            alu_sel2 <= '1' when r2 = w1 else '0';
            alu_sel3 <= '0';
        end if;
    else
        -- Handle other cases as needed
        alu_sel1 <= '0';
        alu_sel2 <= '0';
    end if;
end process

```

Shown in the snippet above is the implementation of data forwarding. What the data forwarder does is that it outputs a '1' when the identity of the writeback register and the identity of an ALU register is the same. The '1' is sent as a mux-select signal to a multiplexer that decides whether to take data of the register from the register file or from the ALU output in the writeback stage, in this case, a '1' would mean that the input be the writeback data.

```

and $t1, $1, $2
invtb $t2, $t1

```

As proof that the forwarding unit works, a dependency is written into the instruction buffer in the form of the two instructions above. The 'and' instruction will have \$21 as the Rd register and the following 'invtb' instruction will use this register as an input to calculate its own Rd data.

reg_content[22]	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
reg_content[21]	000000000000000000000000000000

Content of register 22 contains the inverted data of register 21, which has been forwarded from the 'and' instruction to be inverted and outputted to register 22.

## 2.3 Registers and Instruction Buffer

These two components will need to support the input from file and output to the file, thus we have to design a package for customize.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

package register_file_data_types is
    type register_set is array (31 downto 0) of std_logic_vector(127 downto 0);
end package register_file_data_types;

package body register_file_data_types is
end package body register_file_data_types;

-- Register File Module
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.register_file_data_types.all;
entity Register_File is
    Port (
        clk      : in std_logic;           -- Clock input
        rst      : in std_logic;
        write_en : in std_logic;

        -- Reset input
        read1 : in std_logic_vector(4 downto 0);           -- Read port 1 (address)
        read2 : in std_logic_vector(4 downto 0);
        read3 : in std_logic_vector(4 downto 0); -- Read port 2 (address)
        write : in std_logic_vector(4 downto 0);           -- Write port (address)
        write_data : in std_logic_vector(127 downto 0); -- Write data input
        read1_data : out std_logic_vector(127 downto 0); -- Read port 1 output
        read2_data : out std_logic_vector(127 downto 0);
        read3_data : out std_logic_vector(127 downto 0);
        -- Read port 2 output
        reg_content_en : in std_logic;
        reg_content : out register_set
    );
end entity Register_File;

architecture Behavioral of Register_File is
    --type Register_Array_Type is array (0 to 31) of std_logic_vector(127 downto 0);
```

```

    signal registers : register_set;
begin
    process(clk, rst)
    begin
        if rst = '1' then
            -- Reset the registers to zeros
            registers <= (others => (others => '0'));
        elsif rising_edge(clk) then
            -- Read data from registers
            read1_data <= registers(to_integer(unsigned(read1)));
            read2_data <= registers(to_integer(unsigned(read2)));
            read3_data <= registers(to_integer(unsigned(read3)));

            -- Write data to a register if the write signal is valid
            if write_en = '1' then
                registers(to_integer(unsigned(write))) <= write_data;
            end if;
        end if;
    end process;

    output_to_file:process(reg_content_en, reg_content)
    begin
        if reg_content_en = '1' then
            reg_content <= registers;
        end if;
    end process;

end architecture Behavioral;

```

In both instruction buffer and register file, a custom type is declared in a package, both unit have a loading or file writing enable pin for FILE I/O which will be controlled by the test bench.

## Instruction Decoder

For outputting register index to the register file from instruction, we needed a simple decoder. This unit receives input from IF/ID pipeline register and output to the register file.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity Instruction_Decoder is
    Port (
        input_instruction : in std_logic_vector(24 downto 0);
        reg_index1       : out std_logic_vector(4 downto 0);
        reg_index2       : out std_logic_vector(4 downto 0);
        reg_index3       : out std_logic_vector(4 downto 0);
        write_index       : out std_logic_vector(4 downto 0)
    );
end entity Instruction_Decoder;

architecture Behavioral of Instruction_Decoder is
begin
    process(input_instruction)
    begin
        case input_instruction(24 downto 23) is
            when "00" =>
                reg_index1 <= input_instruction(4 downto 0);
                reg_index2 <= (others => '0');
                reg_index3 <= (others => '0');
                write_index <= input_instruction(4 downto 0);

            when "01" =>
                reg_index1 <= input_instruction(4 downto 0);
                reg_index2 <= (others => '0');
                reg_index3 <= (others => '0');
                write_index <= input_instruction(4 downto 0);

            when "10" =>
                reg_index3 <= input_instruction(19 downto 15);
                reg_index2 <= input_instruction(14 downto 10);
                reg_index1 <= input_instruction(9 downto 5);
                write_index <= input_instruction(4 downto 0);

                when "11" =>
                    reg_index2 <= input_instruction(14 downto 10);
                    reg_index1 <= input_instruction(9 downto 5);
                    reg_index3 <= (others => '0');
                    write_index <= input_instruction(4 downto 0);

            when others =>
                reg_index1 <= (others => '0');
                reg_index2 <= (others => '0');
                reg_index3 <= (others => '0');
                write_index <= input_instruction(4 downto 0);

        end case;
    end process;
end architecture Behavioral;

```

```
end architecture Behavioral;
```

## 3.Verification

### Assembly Code for testing

```
li $0, 0, 0
li $0, 1, 32768
li $0, 2, 0
li $0, 3, 32768
li $0, 4, 0
li $0, 5, 32768
li $0, 6, 0
li $0, 7, 32768
li $1, 0, 16383
li $1, 1, 8191
li $1, 2, 16383
li $1, 3, 8191
li $1, 4, 16383
li $1, 5, 8191
li $1, 6, 16383
li $1, 7, 8191
li $2, 0, 7
li $2, 1, 8192
li $2, 2, 7
li $2, 3, 8192
li $2, 4, 7
li $2, 5, 8192
li $2, 6, 7
li $2, 7, 8192
imal $3, $1, $2, $0
imah $4, $1, $2, $0
imsl $5, $1, $2, $0
imsh $6, $1, $2, $0
lmal $7, $1, $2, $0
lmah $8, $1, $2, $0
lmsl $9, $1, $2, $0
lmsh $10, $1, $2, $0
shrhi $11, $1, $2
```

```
au $12, $1, $2
cnt1h $13, $1
ahs $14, $1, $2
or $15, $1, $2
bcw $16, $1
maxws $17, $1, $2
minws $18, $1, $2
mlhu $19, $1, $2
mlhss $20, $1, $2
and $21, $1, $2
invb $22, $21
rotr $23, $1, $2
sfwu $24, $1, $2
sfhs $25, $1, $2
invb $26, $21
invb $27, $21
invb $28, $21
invb $29, $21
invb $30, $21
invb $31, $21
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
```

In this test code, we first loaded three register \$0, \$1, \$2, for later testing of other instruction. Then, some R4 and R3 instruction are tested using these values inside the three register, the results are saved into other register.

The data forwarding is tested in:

```
and $21, $1, $2
invb $22, $21
```



## 3.1 Instruction verification

### LI instruction

For Load Immediate, the assembly language takes the form of:

li \$Rd, load index, immediate

Shown below are examples of load immediate instructions

```
li $0, 0, 0
li $0, 1, 32768
li $0, 2, 0
li $0, 3, 32768
li $0, 4, 0
li $0, 5, 32768
li $0, 6, 0
li $0, 7, 32768
li $1, 0, 16383
li $1, 1, 8191
li $1, 2, 16383
li $1, 3, 8191
li $1, 4, 16383
li $1, 5, 8191
li $1, 6, 16383
li $1, 7, 8191
```

Basically, the assembler will take the immediate as an integer, convert it into a 16-bit binary value, and feed it to the immediate field of the machine code.

The end result will be that it loads 0x80000000800000008000000080000000 to register \$0 and 0x1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF to register, which it does, as shown in the register contents below.

reg_content[2]	20000007200000072000000720000007
reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
reg_content[0]	80000000800000008000000080000000

### R3 instructions

For R3 instructions, the assembly language takes the form of:

xxxxx Rd, \$Rs1, (Rs2)

An example would be:

```
or $15, $1, $2,
```

Which performs a bitwise-or on the contents of \$1 and \$2 and places the results of the operation on \$15. Results shown in the screenshots below.

reg_content[2]	20000007200000072000000720000007
reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF

## R4 instructions

For R4 instructions, the assembly language takes the form of:

xmxx \$Rd, \$Rs2, \$Rs3, \$Rs1

Take this for an example:

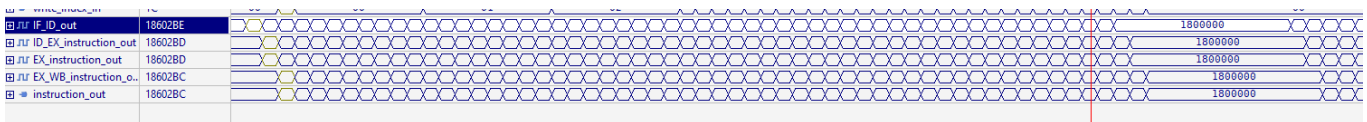
```
imal $3, $1, $2, $0
```

This instruction is the Signed Integer Multiply-Add Low with Saturation instruction, taking the low 16-bit fields on register \$1 and \$2, and adds them to the content of \$0 and puts the result in \$3.

reg_content[3]	1FFF3FFF1FFF00072000000720000007
reg_content[2]	20000007200000072000000720000007
reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
reg_content[0]	80000000800000008000000080000000

This is the hexadecimal result of the instruction, shown in reg\_content[3].

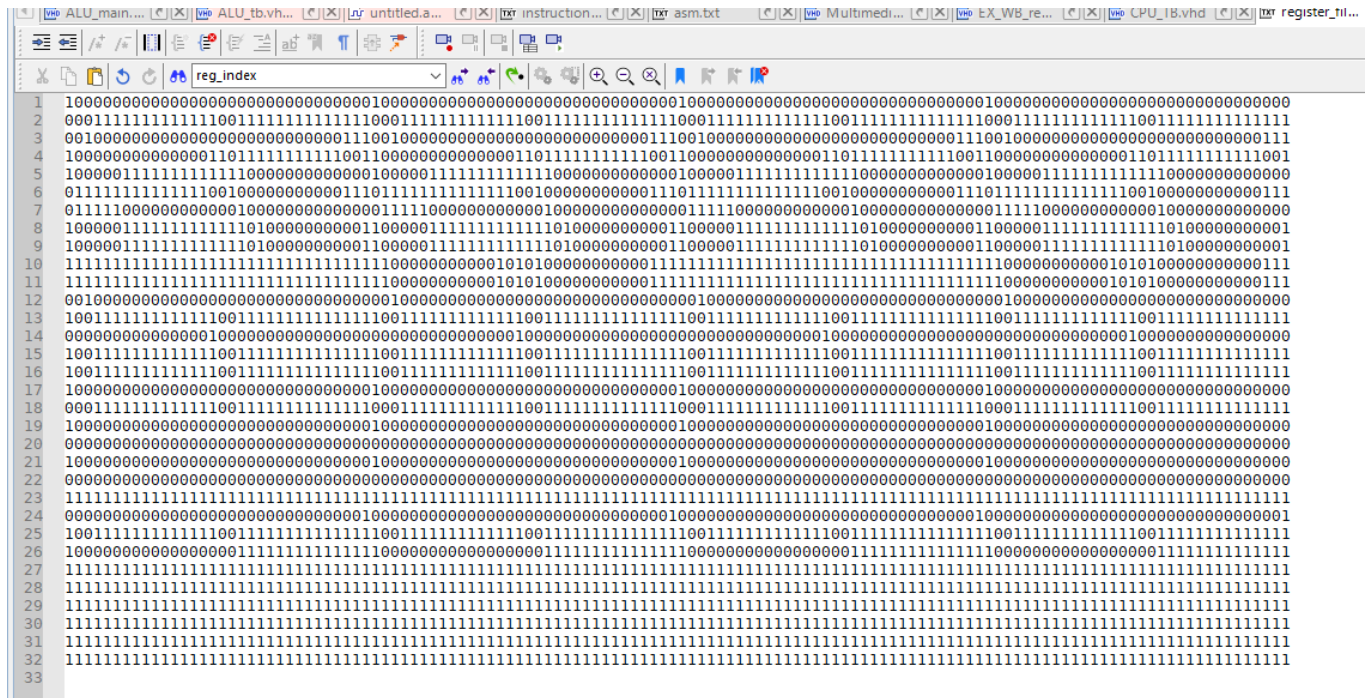
## 3.2 Verification of pipeline



In the output of IF/ID, ID/EX, EX/WB, in the same cycle, we see different instruction being held by the register, this shows that our pipeline design is successful. Note that there are two duplicate signals of EX\_WB and ID\_EX pipeline registers.

## 3.3 Result File comparison

The content of register file will be written to a file at the end of simulation, this results is compared with our expected results:



A simple python program is written to compare the results

```
def compare_files(file1_path, file2_path):
    try:
        # Read the content of the first file
        with open(file1_path, 'r') as file1:
            content1 = file1.read()

        # Read the content of the second file
        with open(file2_path, 'r') as file2:
            content2 = file2.read()

        # Compare the content
        if content1 == content2:
            print("Files have identical content.")
        else:
            print("Files have different content.")

    except FileNotFoundError:
        print("One or both files not found.")

# Example usage:
file_path1 = "register_file_content.txt"
file_path2 = "expected.txt"
```

```
compare_files(file_path1, file_path2)
```

Our results returns that `Files have identical content.`

## 4. Conclusion

In conclusion, this project successfully explored the application of VHDL for the design of a four-stage pipelined multimedia unit. The focus was on implementing a SIMD architecture with a reduced set of multimedia instructions, drawing inspiration from the Sony Cell SPU and Intel SSE architectures. Key objectives included gaining proficiency in VHDL/Verilog, exploring CAD tools, implementing a SIMD architecture, designing a four-stage pipelined structure, and creating a custom instruction set.

Simple control logic, data forwarding has been implemented and verified. A simple assembler program is been written to simplify the process of writing a machine code input file.

Overall, the project successfully met its objectives, providing valuable insights into hardware description languages, CAD tools, and the design considerations for a pipelined multimedia unit.

**\$R26 - \$R30 : Misc**

64-bit fields: 0x8000000008000000 = -9,223,372,034,707,292,160

64-bit fields: 0x1FFF3FFF1FFF3FFF = 2,305,631,899,223,015,423

64-bit fields: 0x20000000720000007 = 2,305,843,039,815,335,943

reg_content[3]	8001BFF98001BFF98001BFF98001BFF9
----------------	----------------------------------


$$Rs2 = \$1, Rs3 = \$2, Rs1 = \$0$$

$$-2147483648 + (8191 \cdot 8192)$$

Each 32-bit field =

$$= -2.080382976 \times 10^9$$

dec -2,080,382,976  
hex 83FFE000

 `reg_content[4]` 83FFE00083FFE00083FFE00083FFE000

\$R5: Calculated from running **Signed Integer Multiply-Subtract Low with Saturation**

Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

$$-2147483648 - (16383 \cdot 7)$$

Each 32-bit field:

$$= -2.147598329 \times 10^9$$

Value has overflowed in the negative direction, so each 32-bit field will be set to the minimum value of 0x80000000

 `reg_content[5]` 80000000800000008000000080000000

\$R6: Calculated from running **Signed Integer Multiply-Subtract High with Saturation**


Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

$$-2147483648 - (8191 \cdot 8192)$$

Each 32-bit field:

$$= -2.21458432 \times 10^9$$

Value has overflowed in the negative direction, so each 32-bit field will be set to the minimum value of 0x80000000

 `reg_content[6]` 80000000800000008000000080000000

\$R7: Calculated from running **Signed Long Integer Multiply-Add Low with Saturation**

Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

$$\begin{array}{r} -9223372034707292160 + \\ (536821759 \cdot 536870919) \\ \hline -8935168043613765639 \end{array}$$

Each 64-bit field:

dec -8,935,168,043,613,765,639  
hex 83FFE8013FFABFF9

 `reg_content[7]` 83FFE8013FFABFF983FFE8013FFABFF9

\$R8: Calculated from running **Signed Long Integer Multiply-Add High with Saturation**

Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

The low 32-bits and high 32-bits of both partitions of \$R0, \$R1, and \$R2 are identical. So it ends with the same result.

 `reg_content[8]` 83FFE8013FFABFF983FFE8013FFABFF9

\$R9: Calculated from running **Signed Long Integer Multiply-Subtract Low with Saturation**

Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

-9223372034707292160-  
(536821759\*536870919)

-9511576025800818681

The value of the result in decimal is

Which is less than the minimum that a signed 64-bit field can represent. Due to negative underflow, the value is rounded to 0x8000000000000000 for both 64-bit fields.

reg_content[9]	80000000000000008000000000000000
----------------	----------------------------------

\$R10: Calculated from running **Signed Long Integer Multiply-Subtract High with Saturation**

Rs2 = \$1, Rs3 = \$2, Rs1 = \$0

Does the exact computation as the one for \$R9 due to register values being designed to repeat every 32 bits. Numerical underflow.

reg_content[10]	80000000000000008000000000000000
-----------------	----------------------------------

\$R11: Calculated from *shift right halfword immediate*

Rs1 = \$0, Rs2 = \$2

Since the shift amount is decided by the instruction's Rs2 field, register \$2 isn't actually used in this instruction. It's only put as \$2 for Rs2 because it's equal to 0010, which means to shift 2 places to the right.

reg_content[0]	80000000800000008000000080000000
----------------	----------------------------------

reg_content[11]	20000000200000002000000020000000
-----------------	----------------------------------

Shifting 2 places right means division by 4, which is correct.

\$R12: Calculated from *add word unsigned*

Rs1 = \$0, Rs2 = \$1

reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
----------------	----------------------------------

reg_content[0]	80000000800000008000000080000000
----------------	----------------------------------

reg_content[12]	9FFF3FFF9FFF3FFF9FFF3FFF9FFF3FFF
-----------------	----------------------------------

Unsigned addition successful

\$R13: *count 1s in halfword*

Rs1 = \$0

reg_content[0]	80000000800000008000000080000000
----------------	----------------------------------

reg_content[13]	00010000000100000001000000010000
-----------------	----------------------------------

One '1' in the first halfword, none in second, one in third, none in fourth, pattern repeats. Count correctly placed in their respective halfword.

\$R14: *add halfword saturated*

Rs1 = \$0, Rs2 = \$1

reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
----------------	----------------------------------

reg_content[0]	80000000800000008000000080000000
----------------	----------------------------------

reg_content[14]	9FFF3FFF9FFF3FFF9FFF3FFF9FFF3FFF
-----------------	----------------------------------

Value's don't saturate for this one.



\$R15: *bitwise logical or*

Rs1 = \$0, Rs2 = \$1

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [15]	9FFF3FFF9FFF3FFF9FFF3FFF9FFF3FFF

Bitwise or successfully executed

---

\$R16: *broadcast word*

Rs1 = \$0

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [16]	80000000800000008000000080000000

No changes made due to every 32-bits already being the same

---

\$R17: *max signed word*

Rs1 = \$0, Rs2 = \$1

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [17]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF

Max correctly identified since *reg*[0] is literally the minimum signed value

---

\$R18: *min signed word*

Rs1 = \$0, Rs2 = \$1

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [18]	80000000800000008000000080000000

Minimum correctly identified

---

\$R19 *multiply low unsigned*

Rs1 = \$0, Rs2 = \$1

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [19]	00000000000000000000000000000000

Since the low half-word of each 32-bit field have full zeroes for register \$0, the results are all zero.

---

\$R20: *multiply by sign saturated*

Rs1 = \$0, Rs2 = \$1

⊕ <i>reg_content</i> [1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
⊕ <i>reg_content</i> [0]	80000000800000008000000080000000
⊕ <i>reg_content</i> [20]	80000000800000008000000080000000







First 16-bit halfword can't get any lower since it's already minimum to begin with, second halfword stays 0 because one of the multipliers is 0.



---

\$R21: *bitwise logical and*

Rs1 = \$0, Rs2 = \$1





  reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
  reg_content[0]	80000000800000008000000080000000
  reg_content[21]	00000000000000000000000000000000

Bitwise AND of the contents result in all zeroes

---

\$R22: *invert (flip) bits of the contents of register rs1*

Rs1 = \$21







  reg_content[21]	00000000000000000000000000000000
  reg_content[22]	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

From all '0' to all '1's. This also shows that forwarding works.

---

\$R23: *rotate bits in word*

Rs1 = \$0, Rs2 = \$1







  reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
  reg_content[0]	80000000800000008000000080000000
  reg_content[23]	00000001000000010000000100000001

The '1' in the MSB of every 32-bit field is rotated 31 times to the right.

---

\$R24: *subtract from word unsigned*

Rs1 = \$0, Rs2 = \$1







  reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
  reg_content[0]	80000000800000008000000080000000
  reg_content[24]	9FFF3FFF9FFF3FFF9FFF3FFF9FFF3FFF

\$0's contents are subtracted from \$1, but since \$0 is negative, it's added instead. Overflow occurs and the results are now negative.

---

\$R25: *subtract from halfword saturated:*

Rs1 = \$0, Rs2 = \$1

  reg_content[1]	1FFF3FFF1FFF3FFF1FFF3FFF1FFF3FFF
  reg_content[0]	80000000800000008000000080000000
  reg_content[25]	80003FFF80003FFF80003FFF80003FFF

8191 - (-2147483648) = overflow and starting from lowest signed halfword

First halfword doesn't change because it is already saturated, the second halfword has no changes because reg[0] is all zeroes in that field.