ESE 345: Computer Architecture

Pipelined Multimedia SIMD Unit

Final Project Report

Luis Pagan
Shrujan Shashank Mutheyboyina

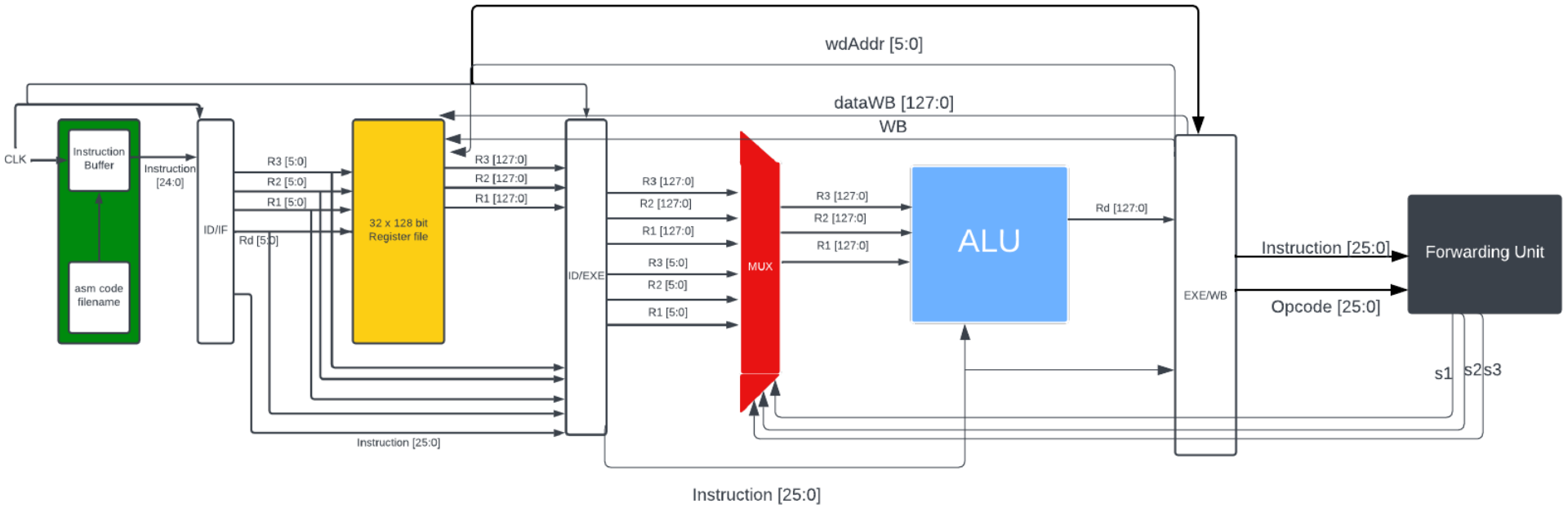
Goals

The project's primary goal is to construct a four-stage pipelined multimedia unit using VHDL/Verilog and contemporary CAD tools while grasping the essence of multimedia instructions akin to those found in the Sony Cell SPU and Intel SSE architectures.

The project entails several steps to achieve this: initially delving into multimedia processing concepts, particularly focusing on subword parallelism as seen in MMX architecture for Intel processors, then refreshing and applying VHDL/Verilog knowledge for digital circuit design. The development phase involves creating behavioral HDL code and verifying multimedia ALU operations, constructing the four-stage multimedia unit and its modules, and rigorously verifying individual modules and the final assembly through comprehensive test benches.

The project's requirements include designing a pipelined structure with concurrently functioning modules such as Multimedia ALU, Register File, Instruction Buffer, Forwarding Unit, and the Four-Stage Pipelined Multimedia Unit, implementing these modules in VHDL/Verilog with specified behavioral or structural criteria, creating an Assembler to convert assembly files to the binary format for the Instruction Buffer, and producing a Results File demonstrating the pipeline status during program execution, encapsulating essential details for each cycle. The emphasis lies on structural/RTL design, verification, and module development to establish a functional pipelined multimedia unit capable of executing instructions across multiple cycles.

Multimedia Unit Block Diagram



Design Procedure

Assembler

The assembler to convert the MIPS assembly instructions to binary code was written in Python. It converts the list of supported instructions as per the project specification to a 25-bit binary representation.

The program uses built-in Python data types and methods, namely a dictionary containing all the available instructions with their types and opcodes. It reads the instruction set from a file (assumes the instruction set is valid to reduce complexity) line by line, parsing out the different components of the instruction, such as the operation and the associated registers. Each instruction is converted to its binary representation based on the instruction type. All the instructions are written to an output file containing only the binary instruction for the instruction buffer.

ALU

The Arithmetic Logic Unit (ALU) is a fundamental building block of any digital processing system and is responsible for performing arithmetic and logic operations on binary data. The ALU proposed in this project is tailored to fit the needs of a SIMD multimedia unit design. Unlike conventional ALUs that typically operate on two operands, this ALU has three operands for more complex operations, such as multiply-add, which require three inputs. The module decodes a 25-bit opcode (OPCODE) to determine the specific arithmetic or logic operation to be executed. This wide opcode allows for a broad spectrum of operations, including specialized functions not commonly found in standard ALUs. The ALU is capable of performing operations on segments of the input data. For instance, it can load immediate values into specific segments or execute arithmetic on individual halfwords or words of the input. Several operations in the ALU_unit support saturation arithmetic. In cases where results exceed the representable range, they are capped to the maximum (or minimum) allowable value, ensuring that overflows wrap around in a controlled manner.

Instruction Buffer

The instruction buffer contains enough space for 64 25-bit instructions that are read in from the assembler output. The instruction buffer takes in a string for the location of the binary instruction file and a clock signal. The buffer is loaded with the binary instructions at the specified string address during the first positive clock and the program counter is set to 0. During the following rising clock edges, the instruction at the specified program counter value is outputted and the program counter is incremented.

Register File

The register file contains an array of 32 128-bit registers used to represent the memory for the pipelined multimedia SIMD unit. The file is independent of the clock signal and can read/write to a specified register given a specified 5-bit address and writeEnable signal (0 - read, 1 - write). The register file is capable of reading up to three registers and writing to one register in a single executed clock cycle.

Forwarding Unit

The forwarding unit handles any data hazards and necessary forwarding actions needed to ensure a stable pipelined processing unit. The unit is responsible for handling data hazards that occur when an instruction reads or writes from a register that is utilized by a previous

instruction. Instructions that do not modify registers are not passed through the forwarding unit, one such instruction being the NOP instruction.

IF/ID, ID/EX, EX/WB Register

The IF/ID (Instruction Fetch/Instruction Decode), ID/EX (Instruction Decode/Execute), and EX/WB (Execute/Write Back) registers are integral components in a pipelined CPU architecture. Each of these registers serves as a critical interface between different stages of the instruction processing pipeline. These registers, functioning as clocked entities, capture values from one state and transfer them to the subsequent state in synchrony with the rising edge of the clock cycle.

The CPU

The CPU unit uses all the above-built units. The CPU unit takes a string file name and a clock input. We will pipeline the unit in the CPU unit file. We have created different signals to feed the I/O ports of each device.

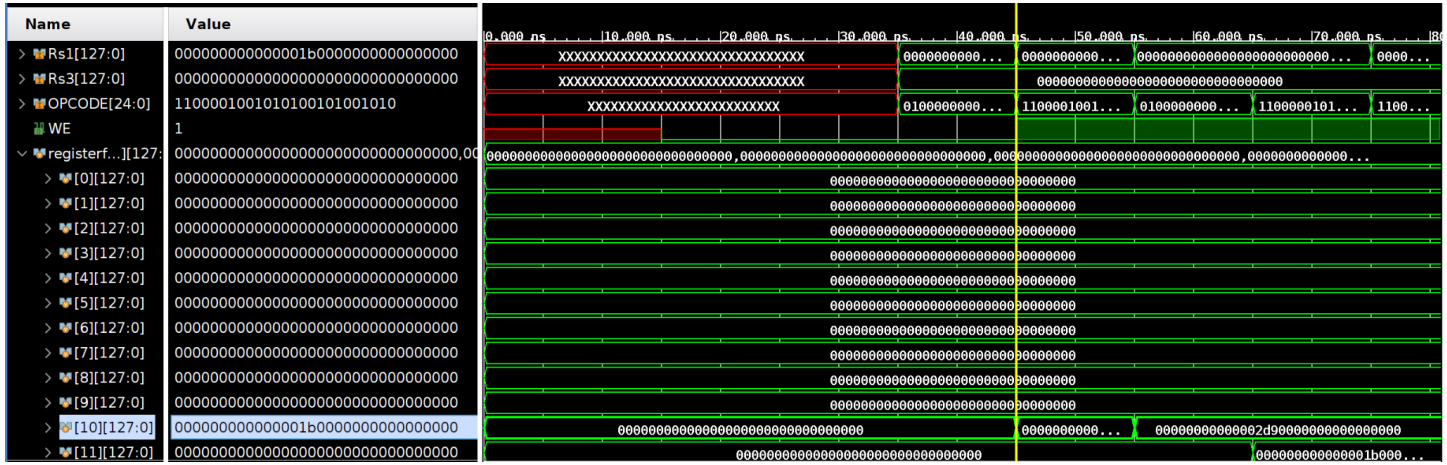
Testbench

The team tested the pipelined multimedia unit, the instruction buffer, and the ALU during the whole design process. The report for part one included the ALU test bench. The instruction buffer test bench and the CPU test bench shared many similarities. The test bench that was utilized to evaluate the CPU is included at the conclusion of the paper.

Simulation Results

Load

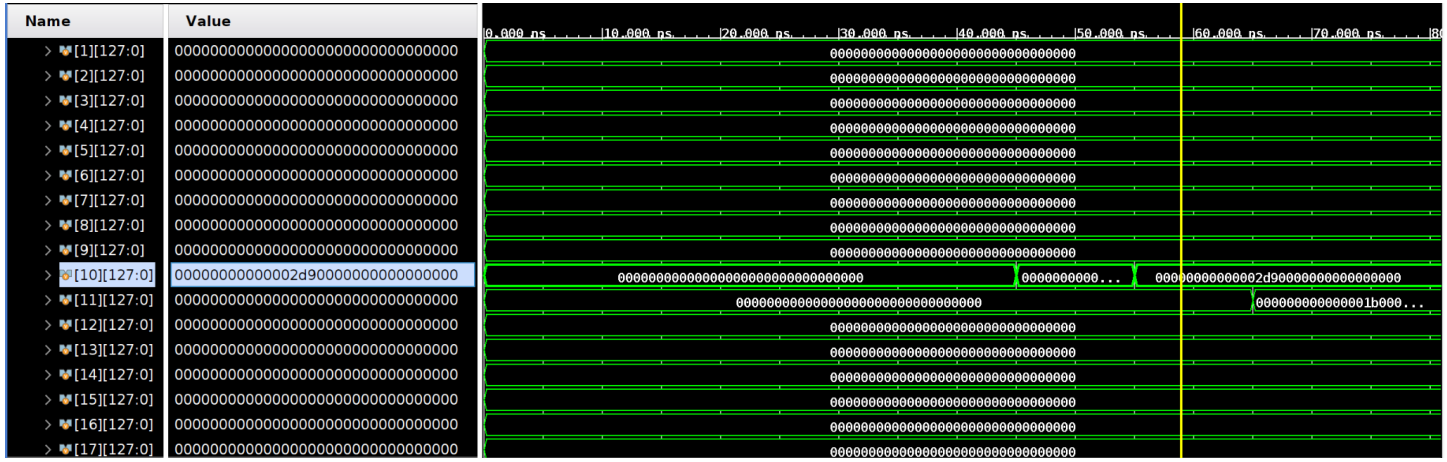
Load a 16-bit Immediate value from the [20:5] instruction field into the 16-bit field specified by the Load Index field [23:21] of the 128-bit register rd.



Our code was LI 10, 4(27), Which means we are loading 27 into the fourth position of the 127-bit value, so the final value of reg 10 should be 0000000000000001b0000000000000000, which matches with the stimulation.

MLHU

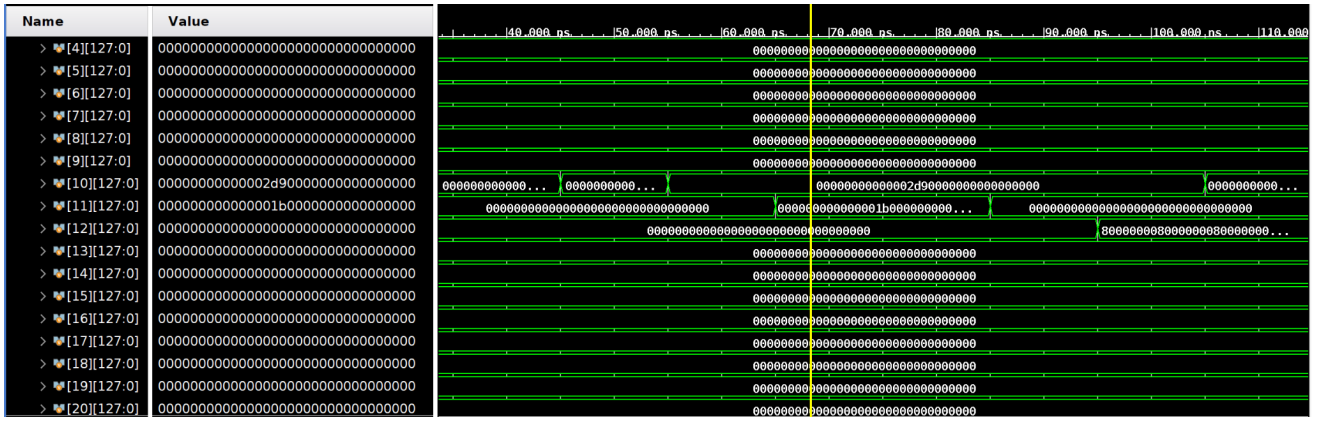
The 16 rightmost bits of each of the four 32-bit slots in register rs1 are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register rs2, treating both operands as unsigned.



Our code was $MLHU\ \ 10,\ 10,\ 10$ which means we are basically squaring them and storing the 127-bit value into the reg10, so the final value of reg 10 should be 0000000000000001b0000000000000000, which matches with the stimulation.02d9 is 729, which is 27*27.

OR

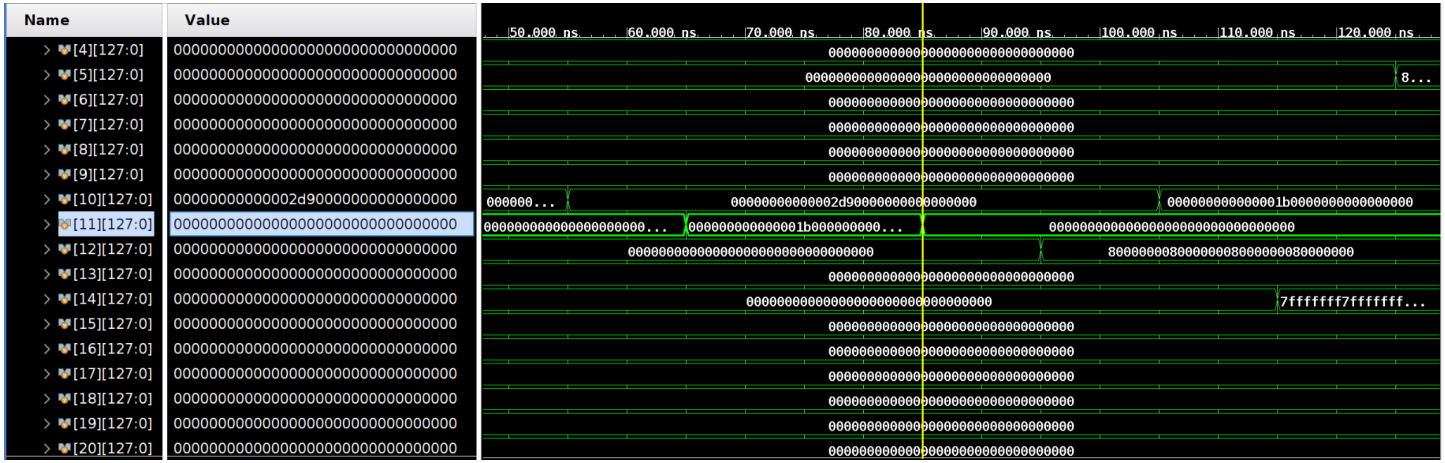
Bitwise logical or of the contents of registers rs1 and rs2



Our code was `OR 12, 0, 3,` Which means we are performing an OR operation between reg3 and reg0 and storing the 127-bit value into the reg12, As there were no values loaded to reg3 and reg0, the or of 0+0 will be 0, so the final value of reg 12 should be 00000000000000000000000000000000, which matches with the stimulation.

MLHU

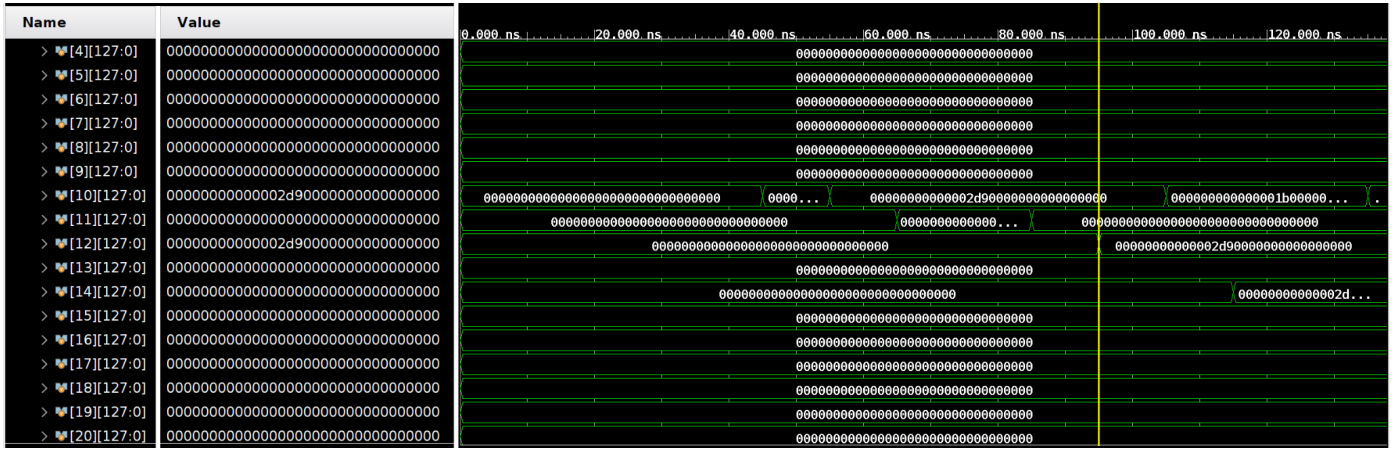
Multiply low unsigned: the 16 rightmost bits of each of the four 32-bit slots in register rs1 are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register rs2, treating both operands as unsigned. The four 32-bit products are placed into the corresponding slots of register rd.



Our code was `MLHU 11, 11, 12` Which means we are performing an MLHU operation between reg12 and reg11 and storing the 127-bit value into the reg11. We just loaded 0 to reg12, so multiplying with zero will make the reg11 0. So the final value of reg 11 should be 00000000000000000000000000000000, which matches with the stimulation.

SIMAL

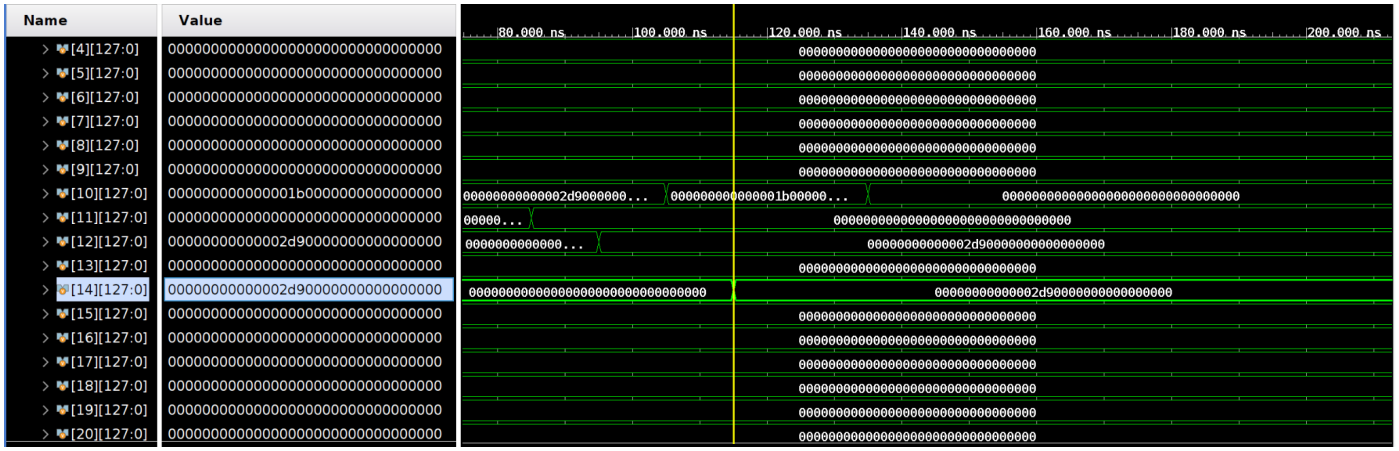
Signed Integer Multiply-Add Low with Saturation: Multiply low 16-bit-fields of each 32-bit field of registers rs3 and rs2, then add 32-bit products to 32-bit fields of register rs1, and save the result in register rd.



Our code was `SIMAL  12, 10, 11`. This means we are performing a SIMAL operation between reg10, reg,11, and reg0 and storing the 127-bit value into the reg12. We are doing reg0 * reg11 +reg10. As reg0 is zero essentially, we are storing the value of reg10 in reg12. So the final value of reg 12 should be 00000000000002d90000000000000000, which matches with the stimulation.

SIMAH

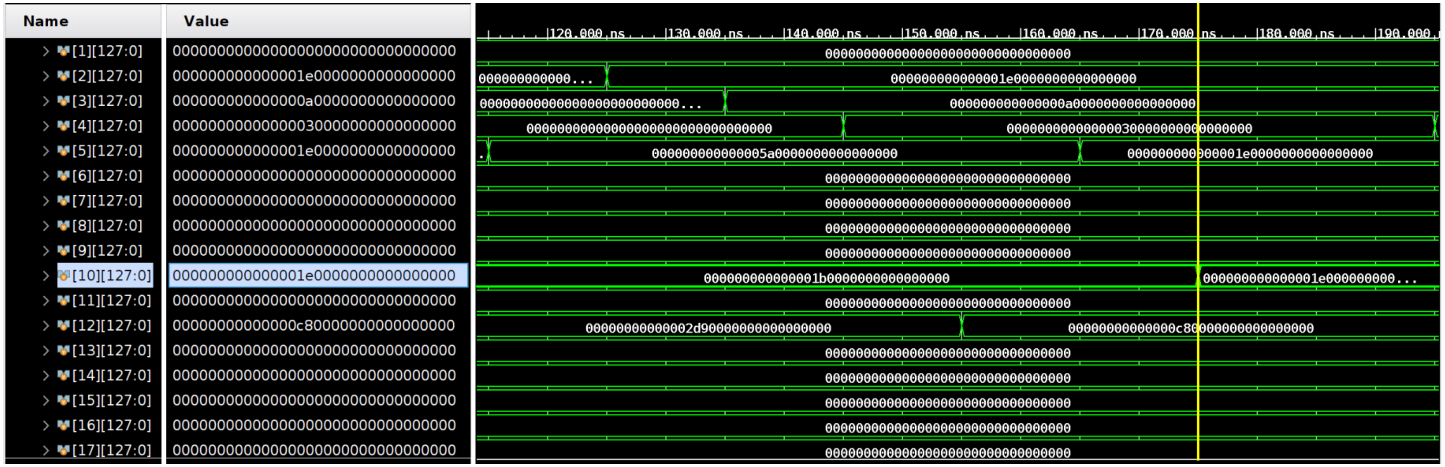
Signed Integer Multiply-Add High with Saturation: Multiply high 16-bit-fields of each 32-bit field of registers rs3 and rs2, then add 32-bit products to 32-bit fields of register rs1, and save the result in register rd.



Our code was `SIMAH   5, 2, 3, 4`. This means we are performing a SIMAL operation between reg10, reg,11, and reg0 and storing the 127-bit value into the reg12. We are doing reg0 * reg11 +reg10. As reg0 is zero essentially, we are storing the value of reg10 in reg12. So the final value of reg 12 should be 00000000000002d90000000000000000, which matches with the stimulation.

SIMSL

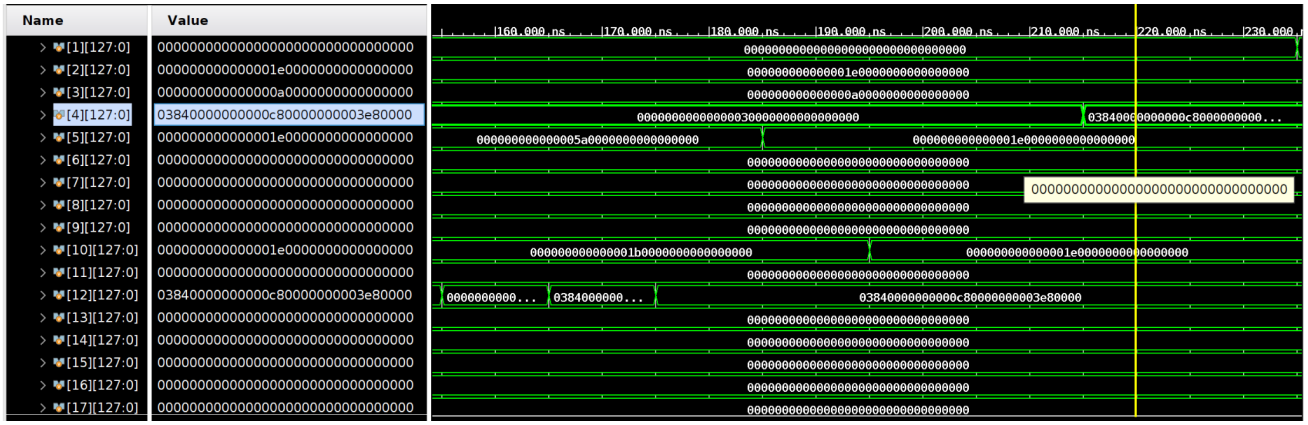
Signed Integer Multiply-Subtract Low with Saturation: Multiply low 16-bit-fields of each 32-bit field of registers rs3 and rs2, then subtract 32-bit products from 32-bit fields of register rs1, and save the result in register rd



Our code was `SIMSL  10, 5, 11, 12`. This means we are performing an SIMSL operation between reg5, reg11, and reg12 and storing the 127-bit value in the reg10. We are doing reg12 * reg11 +reg5. As reg11 is zero essentially, we are storing the value of reg10 in reg5. So the final value of reg 10 should be 000000000000001e0000000000000000, which matches with the stimulation.

SLIMAL

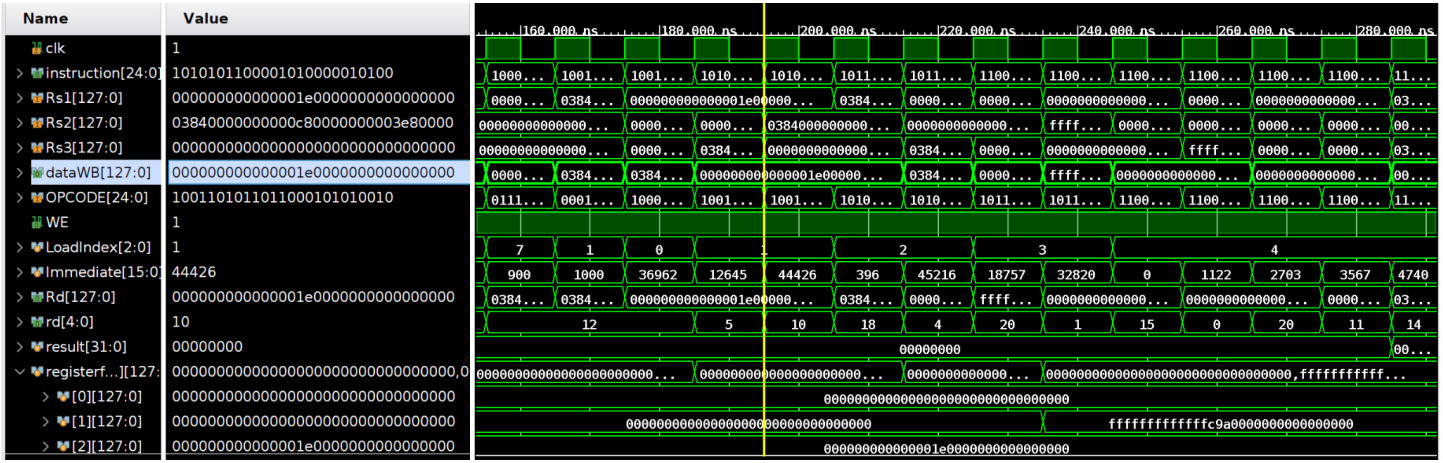
Signed Long Integer Multiply-Add Low with Saturation: Multiply low 32-bit- fields of each 64-bit field of registers rs3 and rs2, then add 64-bit products to 64-bit fields of register rs1, and save the result in register rd.



Our code was `SLIMAL  4, 12, 12, 0`. This means we are performing an SIMSL operation between reg0 and reg12 and storing the 127-bit value in reg4. We are doing reg12 * reg0 +reg12. As reg0 is essentially zero, we are storing the value of reg12 in reg4. So the final value of reg4 should be 03840000000000c80000000003e80000, which matches with the stimulation.

SLIMAH

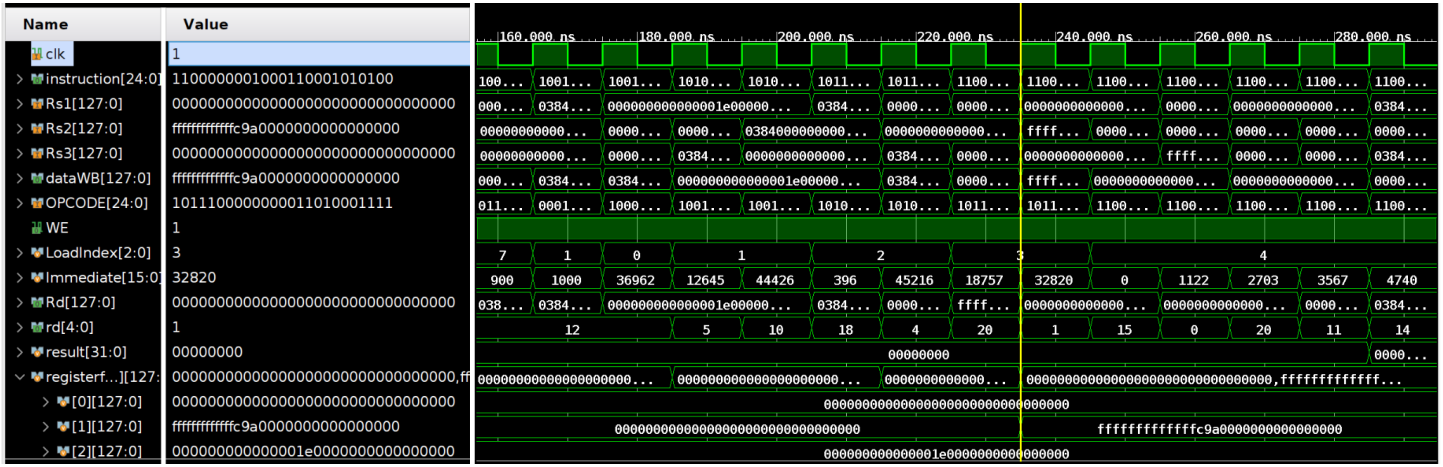
Signed Long Integer Multiply-Add High with Saturation: Multiply high 32-bit- fields of each 64-bit field of registers rs3 and rs2, then add 64-bit products to 64-bit fields of register rs1, and save the result in register rd.



Our code was `SLIMAH  20, 0, 5, 12`. This means we are performing an SLIMAH operation between reg0,reg5, and reg12 and storing the 127-bit value in reg20. We are doing reg5 * reg12 +reg0. As reg0 is essentially zero, we are storing the value of reg12* reg5 in reg20. As the high values for both the registers are 0 their product is zero. So the final value of reg20 should be 00000000000000000000000000000000, which matches with the stimulation.

SLIMSL

Signed Long Integer Multiply-Subtract Low with Saturation: 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 the result in register rd.

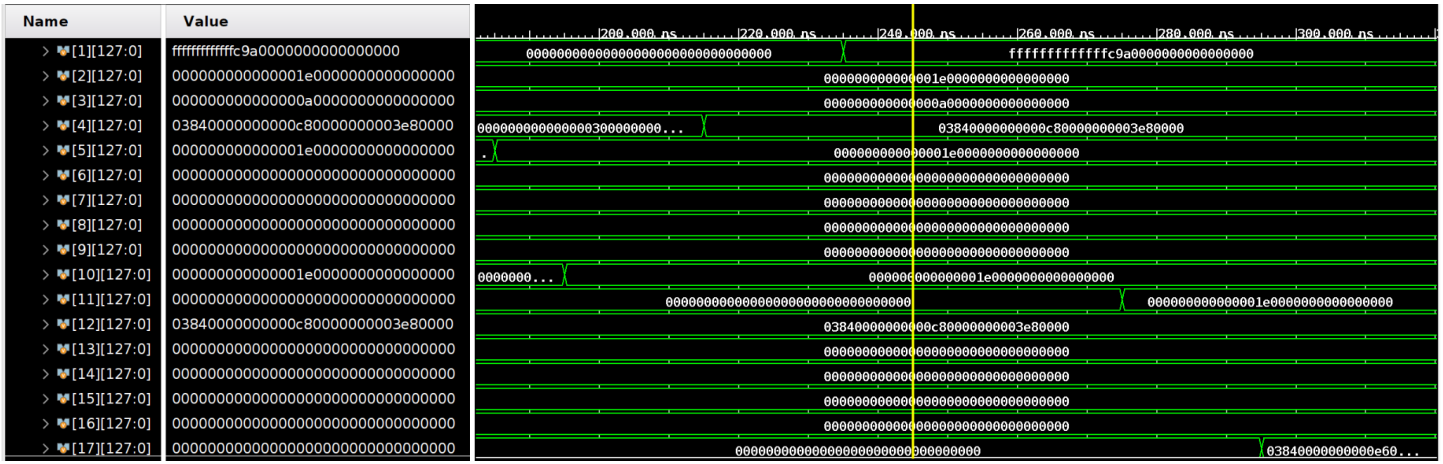


Our code was `SLIMSL  1, 5, 10, 18`. This means we are performing an SLIMSL operation between reg10,reg5, and reg18 and storing the 127-bit value in reg1. We are doing reg18 * reg10 - reg5.In this case, all the registers have the same values (0000000000000001e0000000000000000), so when we

multiply the Low parts of the number, we get (00000000000000900000000000000000), and when we subtract, we get (ffffffffffffc9a0000000000000000). So the final value of reg20 should be ffffffffffffc9a0000000000000000, which matches with the stimulation.

SLIMSH

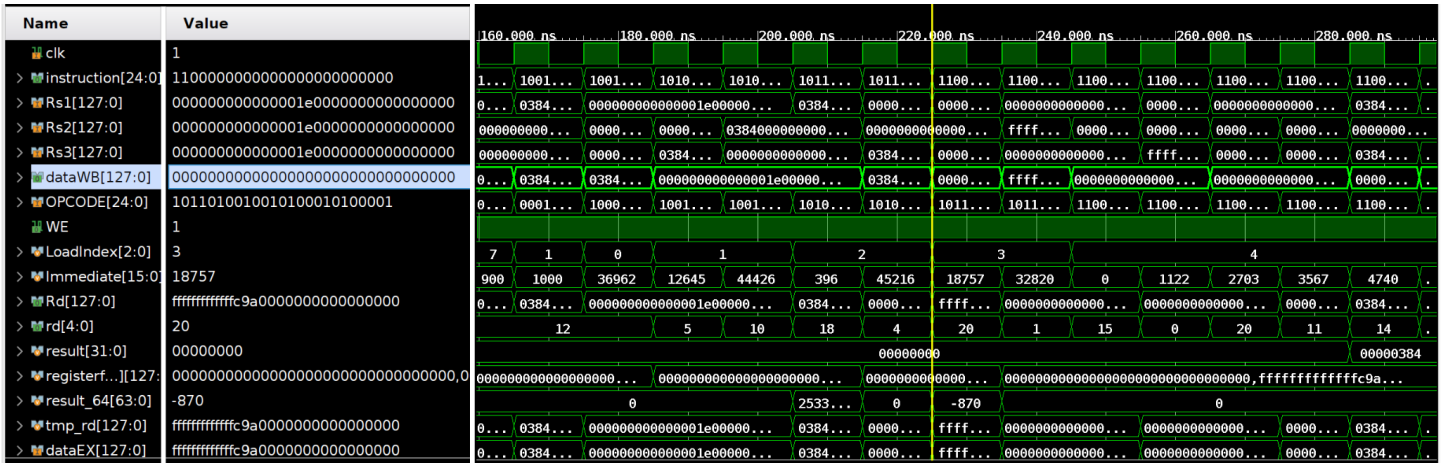
Signed Long Integer Multiply-Subtract High with Saturation: Multiply high 32- bit-fields of each 64-bit field of registers rs3 and rs2, then subtract 64-bit products from 64-bit fields of register rs1, and save the result in register rd.



Our code was `SLIMSH  15, 20, 1, 0`. This means we are performing an SLIMSH operation between reg20,reg1, and reg0 and storing the 127-bit value in reg15. We are doing reg1 * reg0 - reg20.In this case, reg20 and reg0 have 0 values, and reg1 has a value of (ffffffffffffc9a0000000000000000), so when we multiply the high parts of the number, we get (00000000000000000000000000000000), and when we subtract, we get (00000000000000000000000000000000). So the final value of reg20 should be 00000000000000000000000000000000, which matches with the stimulation.

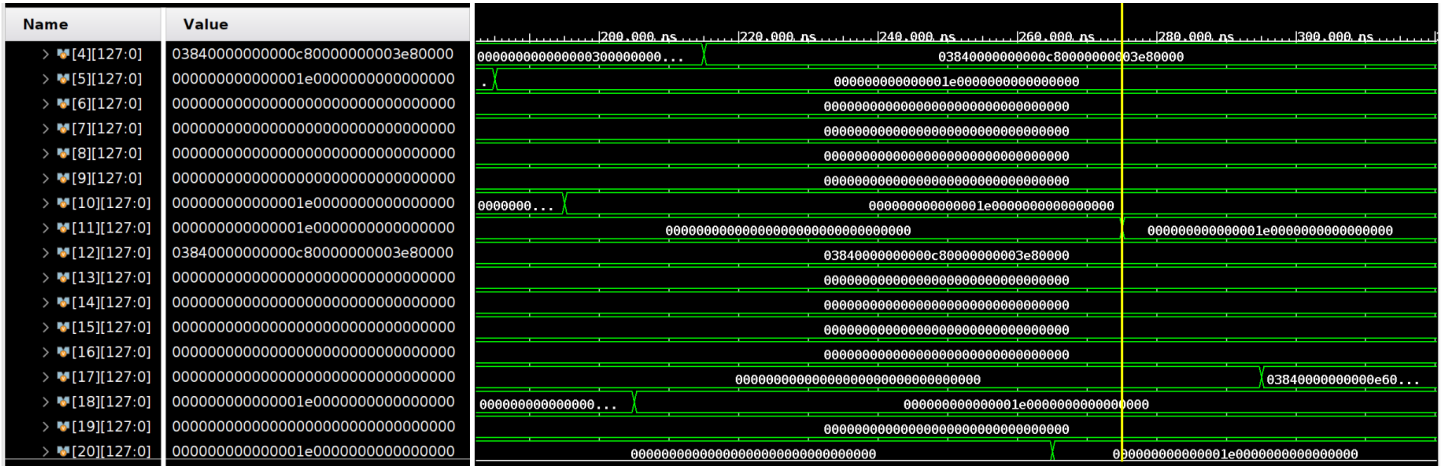
NOP

No operation.



AU

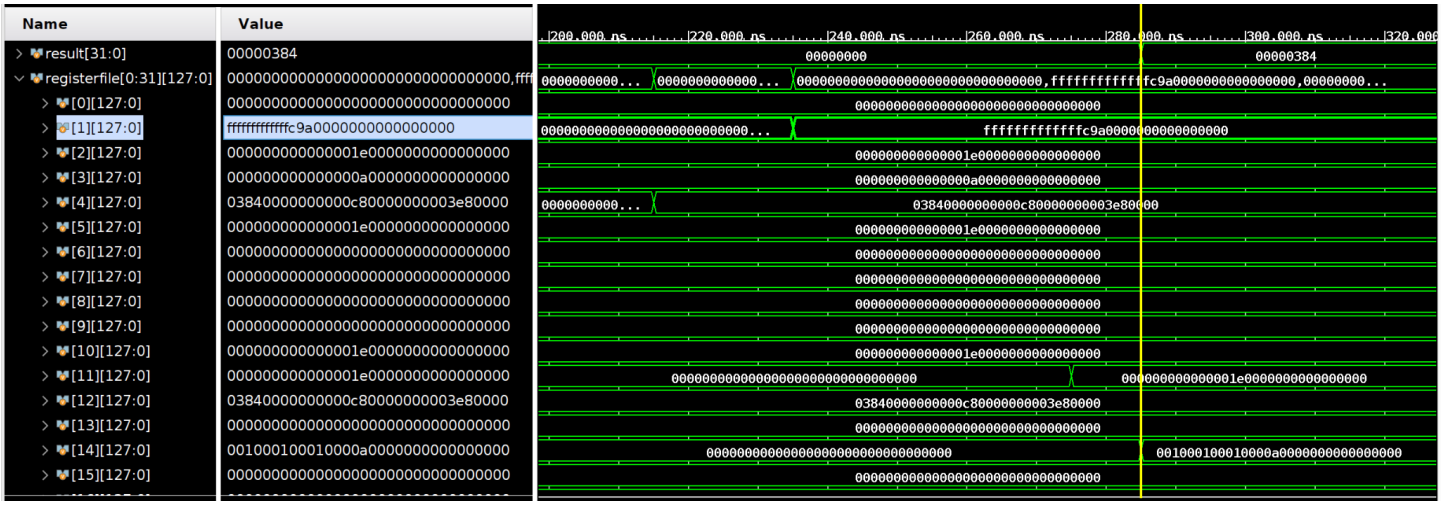
Add word unsigned: packed 32-bit unsigned addition of the contents of registers rs1 and rs2.



Our code was `AU 11, 15, 20`. This means we are performing an AU operation between reg20 and reg15 and storing the 127-bit value in reg11. We are doing reg15 + reg20.In this case, reg15 has 0 values, and reg20 has a value of (0000000000000001e0000000000000000), so when we do addition with zero, we are storing the value of reg20 in reg 11.

CNT1H

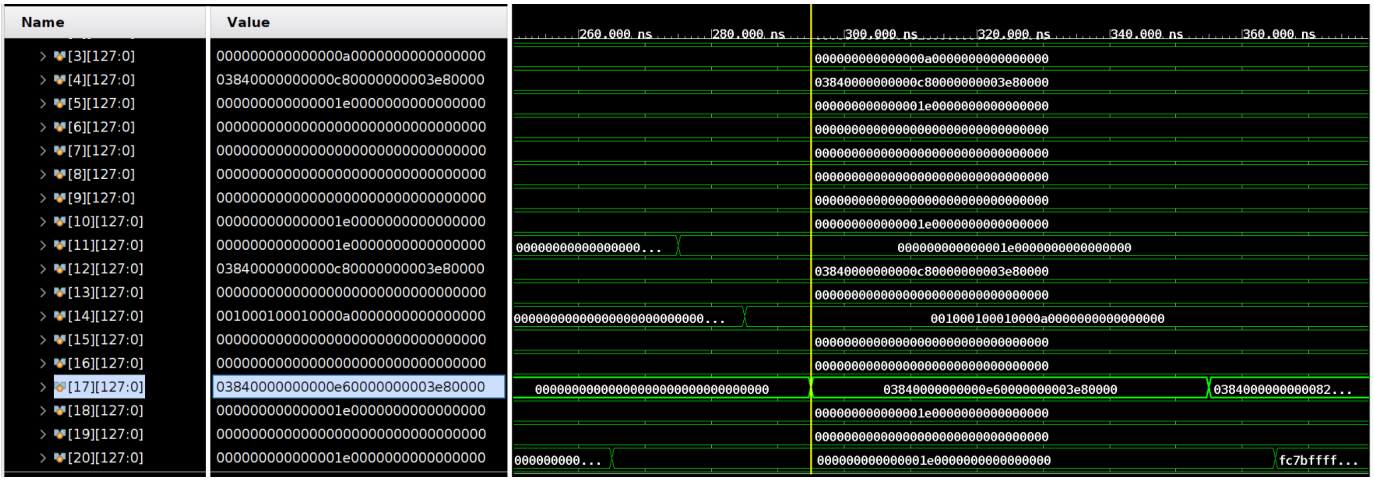
Count 1s in halfword: count 1s in each packed 16-bit halfword of the contents of register rs1. The results are placed into corresponding slots in register rd.



Our code was `CNT1H  14, 1`. This means we are counting the number 1's in halfwords in reg1 and storing the 127-bit value in reg14. Reg1 has a value of 11111111111111111111111111111111111111111111111111111111001001101000000000000000000000000000000000000000000000000000000000000000000000000000000000000000. So the answer is 0010001000100000a0000000000000000

AHS

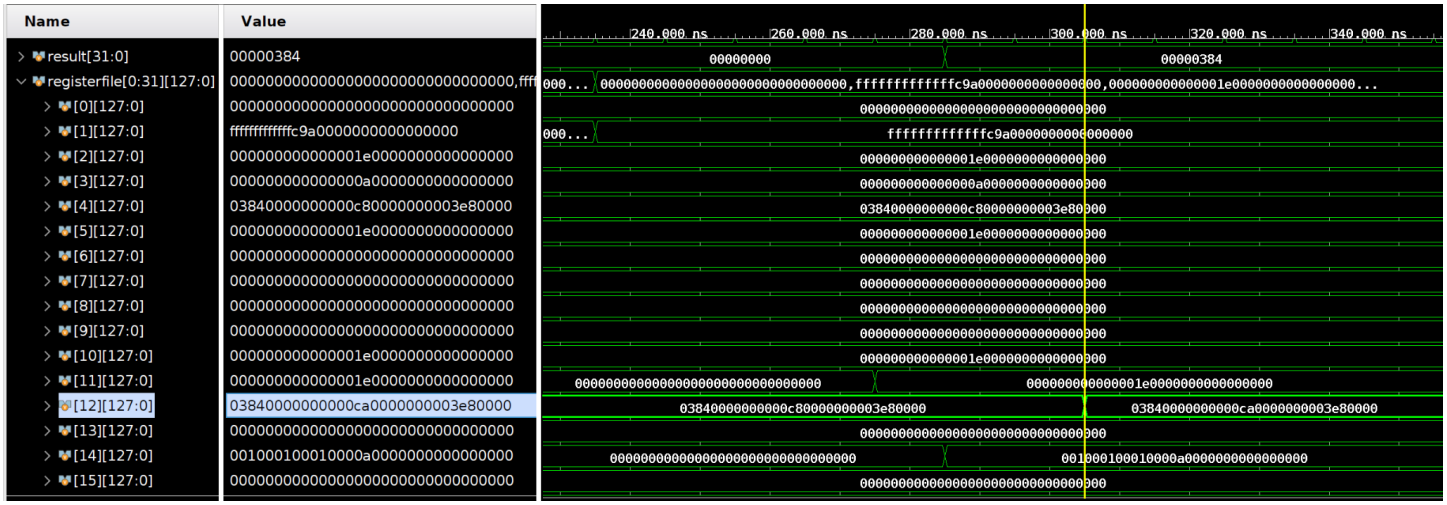
Add halfword saturated: packed 16-bit halfword signed addition with the saturation of the contents of registers rs1 and rs2.



Our code was `AHS 17, 4, 20`. This means we are adding 16-bit halfword signed numbers of reg4 and reg20 and storing the 127-bit value in reg17. reg4 has a value of (03840000000000c80000000003e80000), and reg20 (000000000000001e0000000000000000) . So the answer is 03840000000000e60000000003e80000

OR

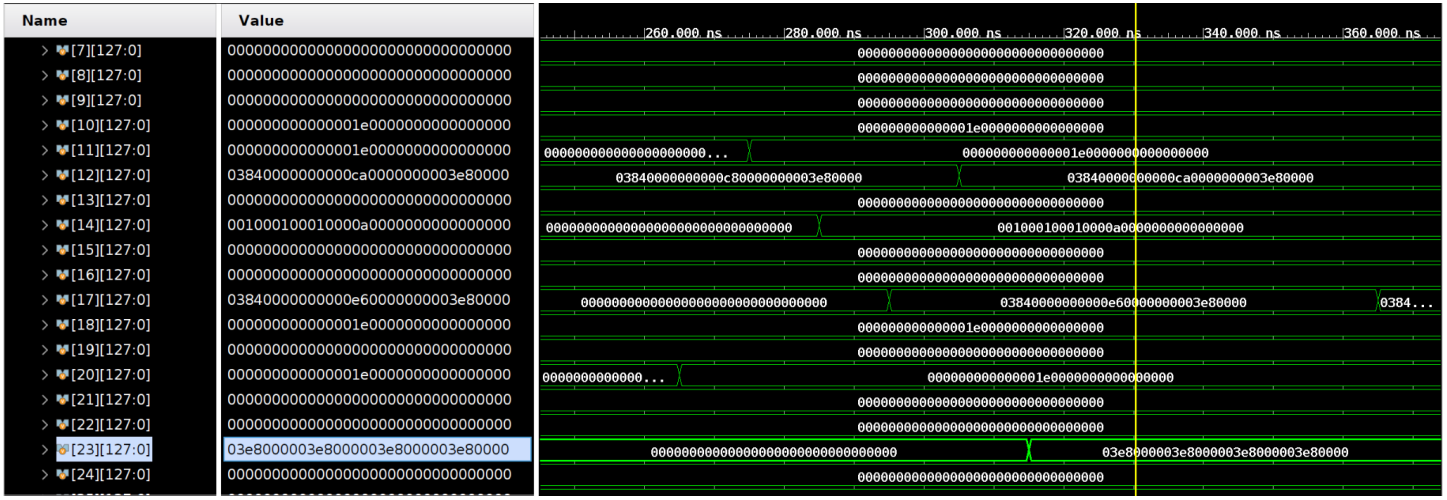
Bitwise logical or of the contents of registers rs1 and rs2.



Our code was `OR 12, 12, 3`. This means we are performing an OR operation between reg12 and reg3 and storing the 127-bit value in reg12. reg12 has a value of (03840000000000c80000000003e80000), and reg3 (000000000000000a0000000000000000) . So the result would be 03840000000000ca0000000003e80000

BCW

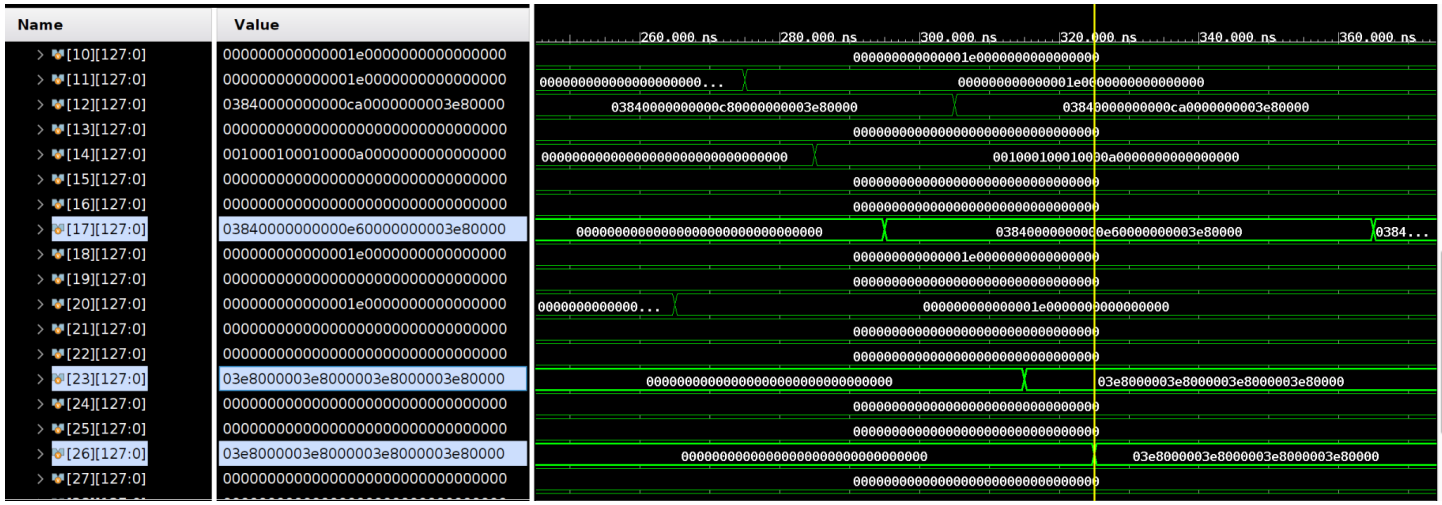
Broadcast word: broadcast the rightmost 32-bit word of register rs1 to each of the four 32-bit words of register rd.



Our code was `BCW 23, 4`. This means we have broadcast the rightmost 32-bit word of register reg4 to each of the four 32-bit words of reg23. reg23 has an initial value of (00000000000000000000000000000000), and reg4 (03840000000000c80000000003e80000). So the result would be 03e8000003e8000003e8000003e80000

MAXWS

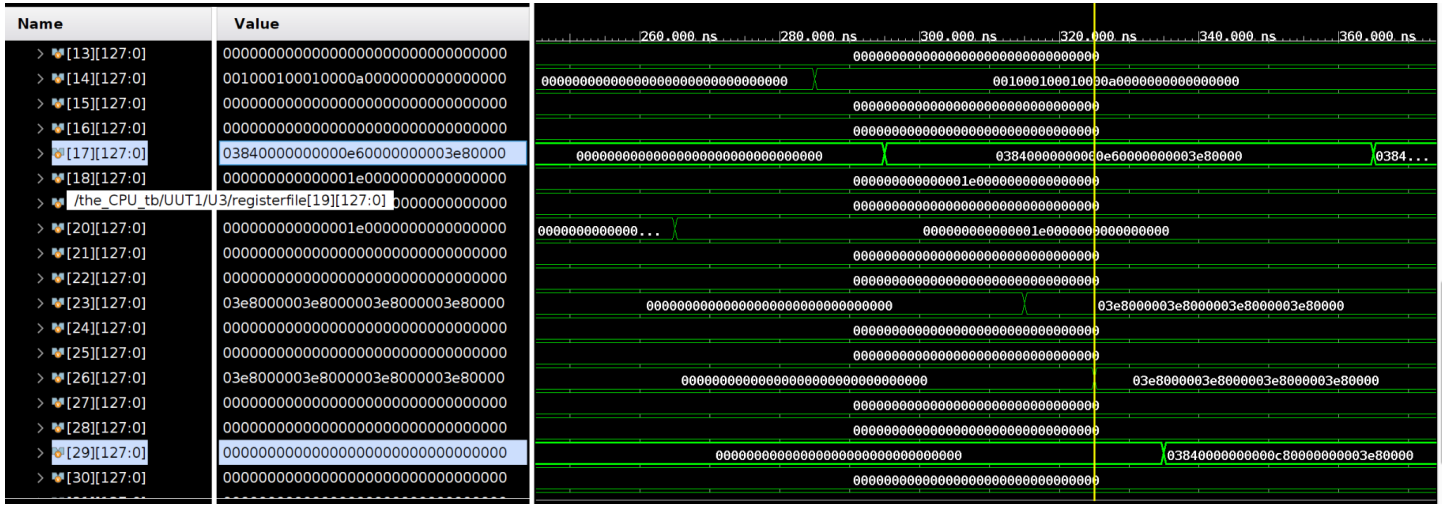
Max signed word: for each of the four 32-bit word slots, place the maximum signed value between rs1 and rs2 in register rd.



Our code was `MAXWS 26, 17, 23`. This means we have to load the max value into reg26. reg17 has an initial value of (03840000000000e60000000003e80000), and reg23 (03e8000003e8000003e8000003e80000). So the max result would be 03e8000003e8000003e8000003e80000

MINWS

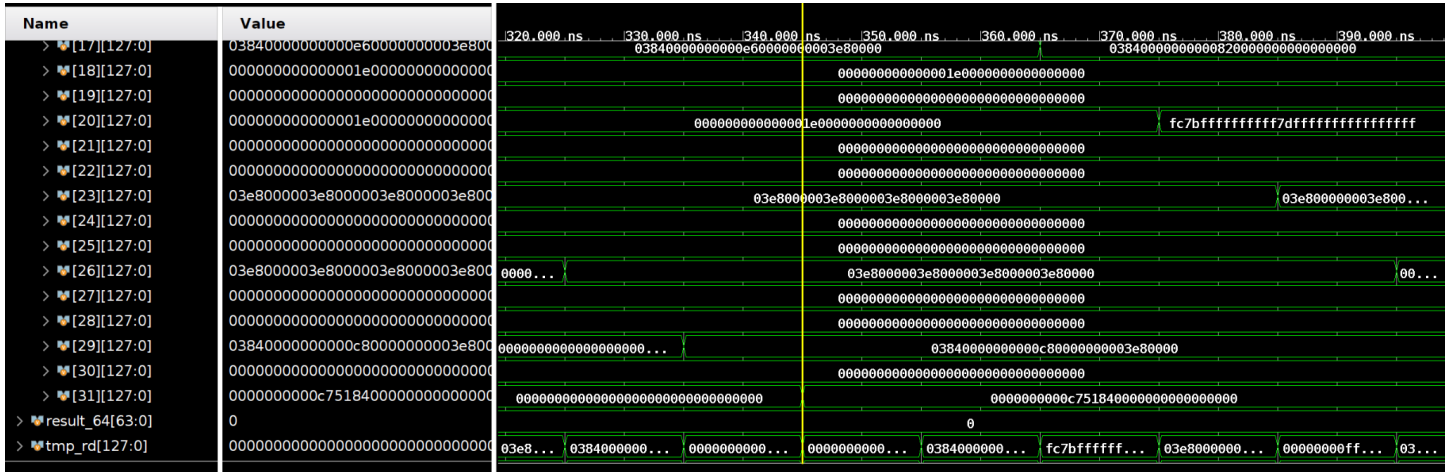
Min signed word: for each of the four 32-bit word slots, place the minimum signed value between rs1 and rs2 in register rd.



Our code was `MINWS  29, 17, 4`. This means we have to load the max value into reg29. reg17 has an initial value of (03840000000000e60000000003e80000), and reg23 (03840000000000c80000000003e80000). So the max result would be 03840000000000c80000000003e80000

MLHU

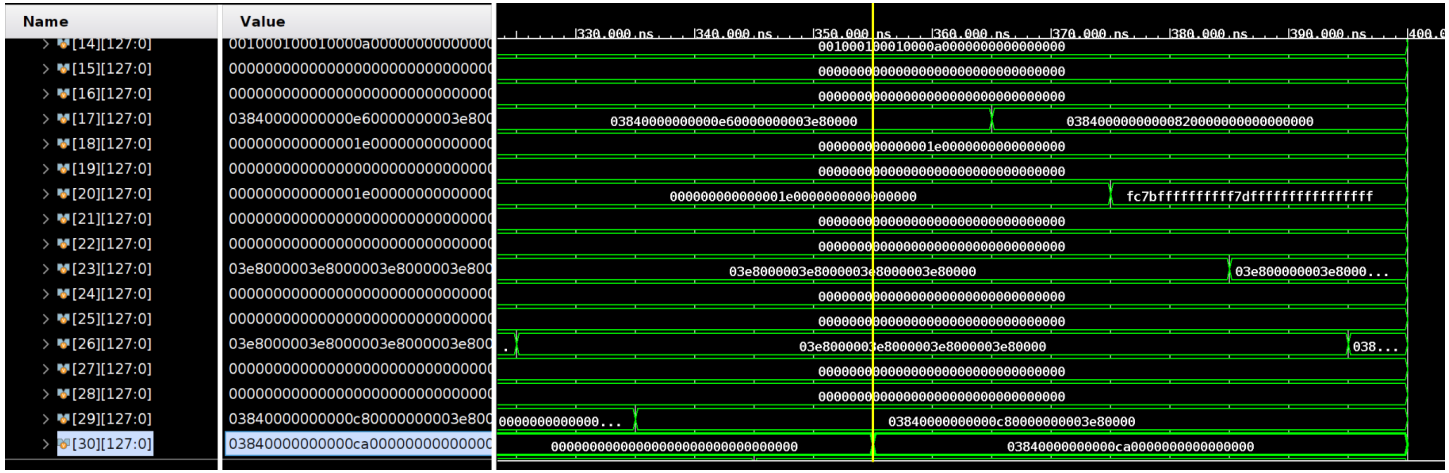
Multiply low unsigned: the 16 rightmost bits of each of the four 32-bit slots in register rs1 are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register rs2, treating both operands as unsigned. The four 32-bit products are placed into the corresponding slots of register rd.



Our code was `MLHU 31, 12, 1`. This means we Multiply the 16 rightmost bits of each of the four 32-bit slots in register reg12 by the 16 rightmost bits of the corresponding 32-bit slots in register reg1. So the result would be 0000000000c751840000000000000000000.

MLHSS

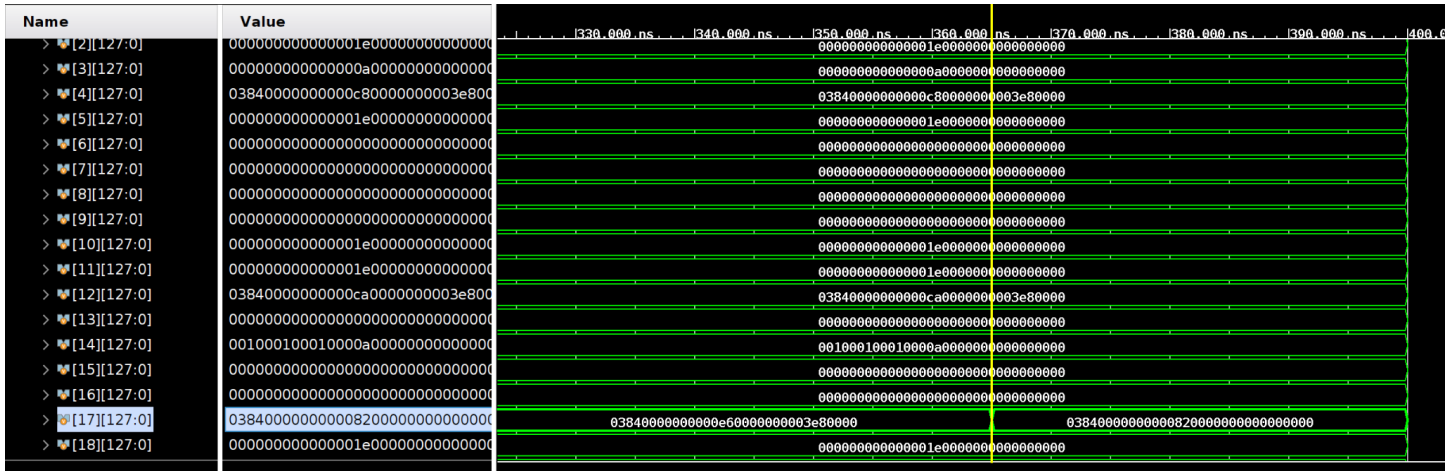
Multiply by sign saturated: each of the eight signed 16-bit halfword values in register rs1 is multiplied by the sign of the corresponding signed 16-bit halfword value in register rs2 with saturation, and the result is placed in register rd. If a value in a 16-bit register rs2 field is zero, the corresponding 16-bit field in rd will also be zero.



Our code was `MLHSS 30, 12, 1`. This means we Multiply the 16 bits of each of the four 32-bit slots in register reg12 by the signed 16 bits of the corresponding 32-bit slots in register reg1. So the result would be 0384000000000ca0000000000000000.

AND

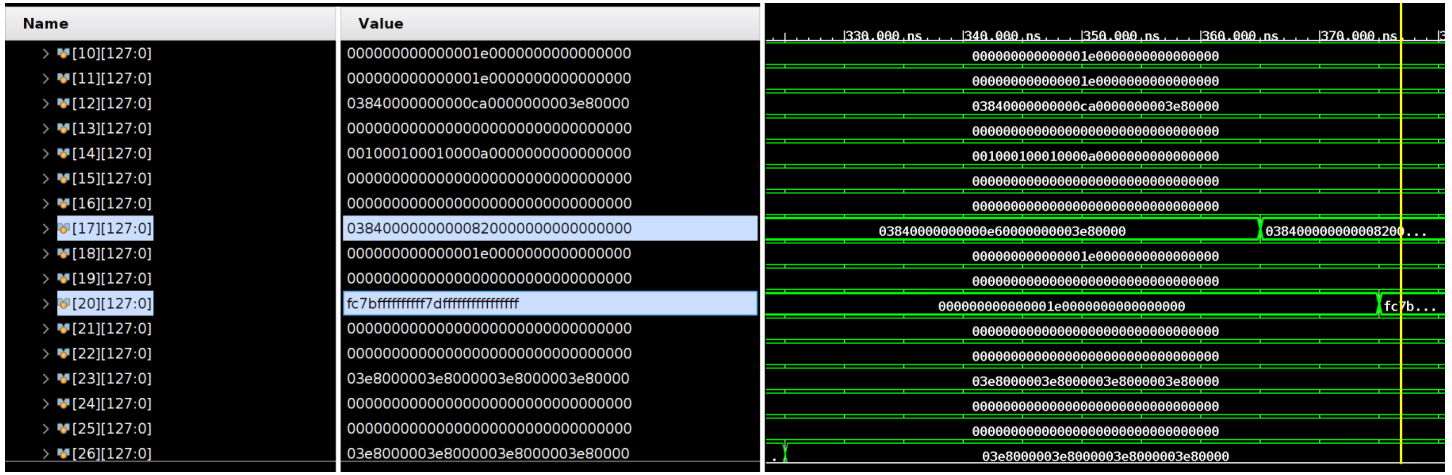
Bitwise logical and of the contents of registers rs1 and rs2.



Our code was `AND 17, 17, 1`. This means we are getting the and of reg17 and reg1 and loading that value in reg17.
(03840000000000e60000000003e80000) & (ffffffffffffc9a0000000000000000) => 03840000000000820000000000000000.

INVB

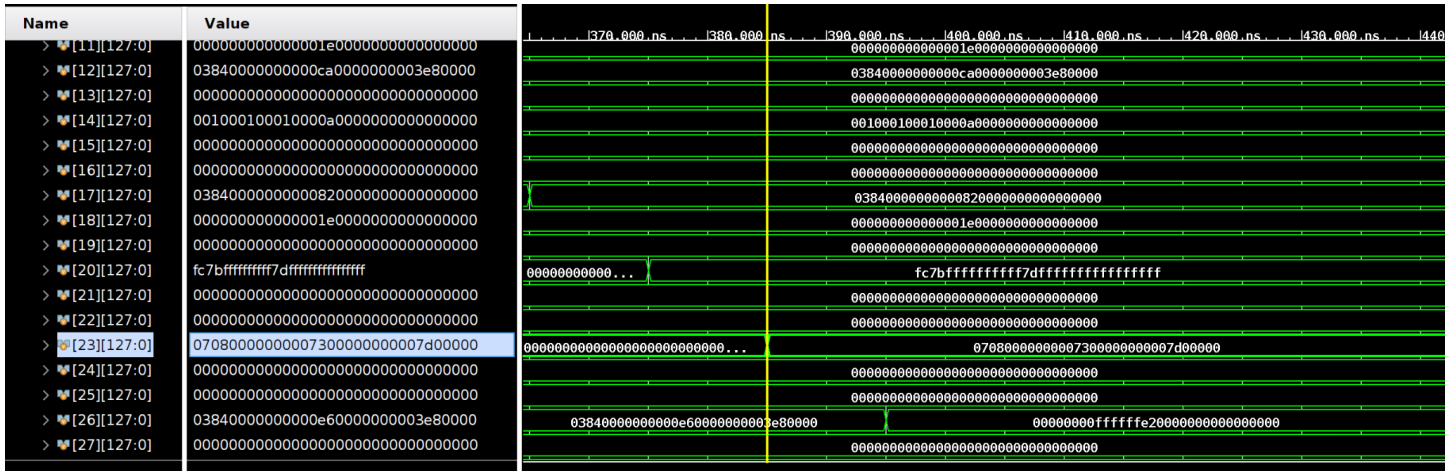
Invert (flip) bits of the contents of register rs1. The result is placed in register rd.



Our code was `INVB 20, 17`. This means we are inverting bits of reg17 and placing them in reg20.
`~(00000011100001000000000000000000000000000000000000000000000100000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000) = 11111100011110111111111111111111111111111111111111111111011111011111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111`

ROTW

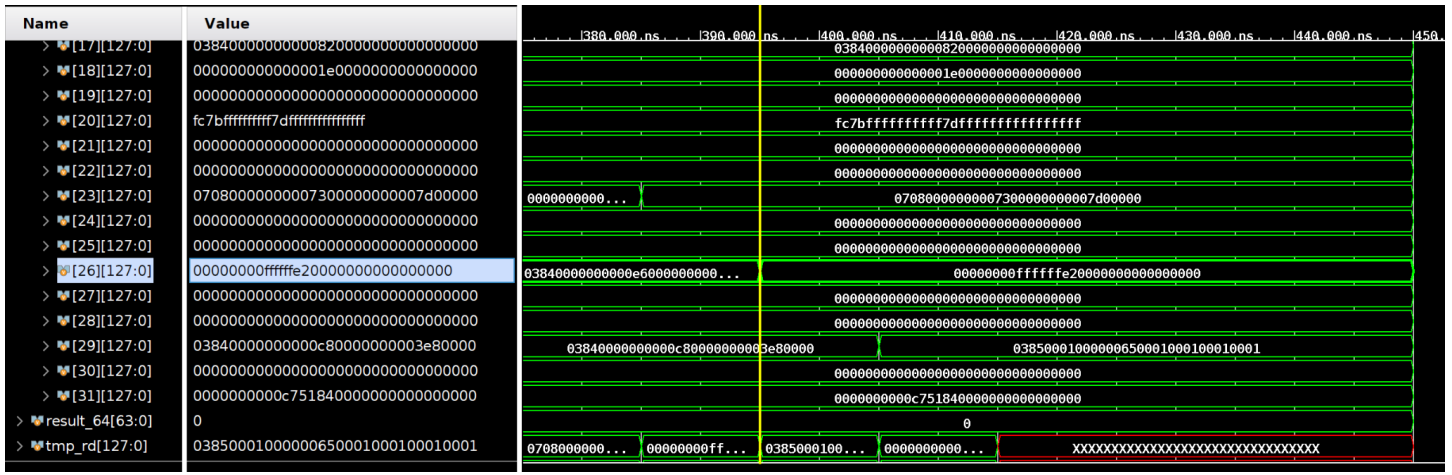
Rotate bits in word: the contents of each 32-bit field in register rs1 are rotated to the right according to the value of the 5 least significant bits of the corresponding 32-bit field in register rs2. The results are placed in register rd. Bits rotated out of the right end of each word are rotated in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



Our code was `ROTW 23, 26, 20`. This means we are rotating bits of reg26 according to the value of the 5 least significant bits of the corresponding 32-bit field in register reg20 and placing them in reg23. 03840000000000e60000000003e80000 is rotated 5 turns; it will result in 07080000000007300000000007d00000

SFWU

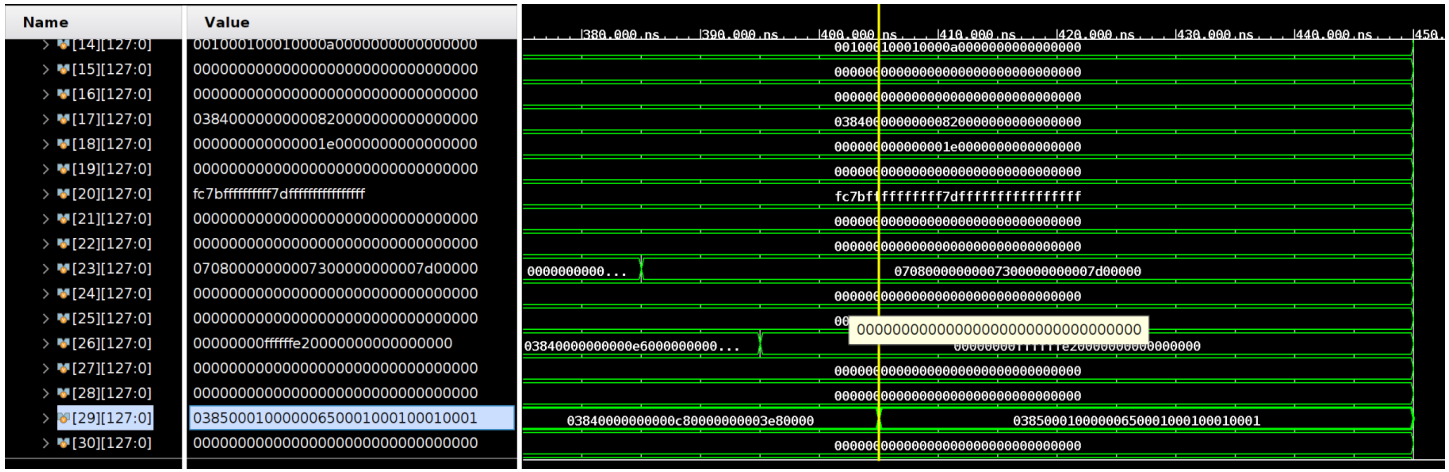
Subtract from word unsigned: packed 32-bit word unsigned subtract of the contents of rs1 from rs2 (rd = rs2 - rs1).



Our code was `SFWU 26, 10, 30`. This means we are sub-bits of reg10 and reg30 and placing the result in reg26.
00000000000000000000000000000000 - 000000000000001e0000000000000000 => 00000000fffffffe2000000000000000

SFHS

Subtract from halfword saturated: packed 16-bit halfword signed subtraction with saturation of the contents of rs1 from rs2 (rd = rs2 - rs1).

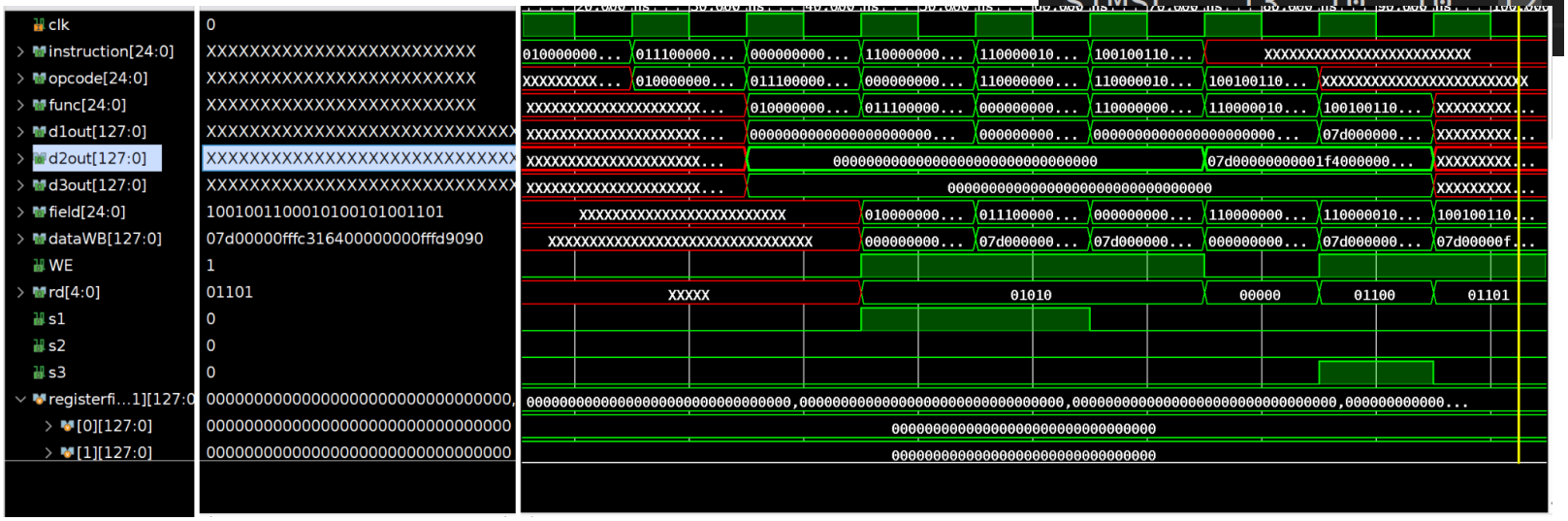


Our code was `SFHS 29, 20, 26`. This means we will perform 16-bit halfword signed subtraction with saturation of the contents of reg26 from reg20 and placing the result in reg29. 00000000ffffffe20000000000000000 - fc7bfffffffff7dfffffffffffffffff => 0385000100000065000100010001 0001

Instruction Progress for each type of instruction/Data Forwarding:

To showcase data forwarding and progress of each State register through the program we have chosen To run the following code. We are first loading reg10 And then performing an R3-type instruction, and then Performing an R4-type instruction.

```
LI      10,4(500)
LI      10,7(2000)
LI      10,0(400)
NOP
OR      12, 0, 10
SIMSI   13, 10, 10, 12
```



Cycle 1				
Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit
The Instruction Buffer reads the binary file LI 10,4(500) and output (01000000001111101000 1010) the instruction to the input of the ID/IF register.				

Cycle 2				
Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit
The InstructionBuffer reads the binary file LI 10,7(2000) and output (01110000011111010000010 10) the instruction to the input of the ID/IF register.	The ID/IF register reads the input LI 10,4(500) in binary and outputs the opcode to the reg file.			

Cycle 3				
Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit
The InstructionBuffer reads the binary file LI 10,0(400) and output (00000000001100100000 1010) the instruction to the input of the ID/IF register.	The ID/IF register reads the input LI 10,7(2000) in binary and outputs the opcode to the reg file.	The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function as the instruction is load the register values are zero.		

Cycle 4				
Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit
The InstructionBuffer reads the binary file NOP and output (11000000000000000000 0000) the instruction to the input of the ID/IF register.	The ID/IF register reads the input LI 10,0(400) in binary and outputs the opcode to the reg file.	The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. As the instruction is load the register values are zeros it's the first instruction.	The register EXE/WB gets the output of the ALU. As the instruction is Load, we will enable the write-back and output the rd as (01010 or 10_2) and the write-back value as (00000000000001f400000 00000000000) as we are loading 1f4 in the fourth position of the register	We need to forward data as the next instruction uses the same register. We will forward the value of reg10 to the ALU, which is performing the operation to get the write-back value.

			(ALU will compute the write-back value).	

Cycle 5

Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit

Cycle 6

Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit

Cycle 7

Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit

Cycle 8

Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit

Cycle 9

Stage 1	Stage 2	Stage 3	Stage 4	Forwarding Unit
			The register EXE/WB gets	

			the output of the ALU. As the instruction is SIMSL, we will enable the write-back and output the rd as (01101 or 13_2) and the write-back value as (07d00000fffc316400000000fffd9090) as we performed low half-word multiplication between reg10 and reg12. Add added that result with reg10.	

Cycle 4:

Stage 1: The InstructionBuffer reads the binary file **NOP** and output **(11000000000000000000000000)** the instruction to the input of the ID/IF register.

Stage 2: The ID/IF register reads the input **LI 10,0(400)** in binary and outputs the opcode to the reg file.

Stage 3: The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. As the instruction is load the register values are zeros it's the first instruction.

Stage 4: The register EXE/WB gets the output of the ALU. As the instruction is Load, we will enable the write-back and output the rd as (01010 or 10_2) and the write-back value as (00000000000001f40000000000000000) as we are loading 1f4 in the fourth position of the register (ALU will compute the write-back value).

Forwarding Unit: We need to forward data as the next instruction uses the same register. We will forward the value of reg10 to the ALU, which is performing the operation to get the write-back value.

Cycle 5:

Stage 1: The InstructionBuffer reads the binary file **OR 12, 0, 10** and output **(11000001010101000000001100)** the instruction to the input of the ID/IF register.

Stage 2: The ID/IF register reads the input **NOP** in binary and outputs the opcode to the reg file.

Stage 3: The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. This register will store the r1,r2,r3 just in case to use them, so r1 = (00000000000001f40000000000000000), r2 = 0, r3 = 0.

Stage 4: The register EXE/WB gets the output of the ALU. As the instruction is Load, we will enable the write-back and output the rd as (01010 or 10_2) and the write-back value as (07d00000000001f40000000000000000) as we are loading 07d0 in the seventh position of the register (ALU will compute the write-back value).

Forwarding Unit: We need to forward data as the next instruction uses the same register. We will forward the value of reg10 to the ALU, which is performing the operation to get the write-back value.

Cycle 6:

Stage 1: The InstructionBuffer reads the binary file **SIMSL 13, 10, 12** and output **(10000000000011000101001101)** the instruction to the input of the ID/IF register.

Stage 2: The ID/IF register reads the input **OR 12, 0, 10** in binary and outputs the opcode to the reg file.

Stage 3: The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. This register will store the r1,r2,r3. As the instruction is NOP, we will perform no operation.the values of r1 = 0, r2 = 0, and r3 = 0.
Stage 4: The register EXE/WB gets the output of the ALU. As the instruction is Load, we will enable the write-back and output the rd as (01010 or 10_2) and the write-back value as (07d00000000001f40000000000000190) as we are loading 0190 in the zero position of the register (ALU will compute the write-back value).
Forwarding Unit: —----------

Cycle 7:

Stage 1: —----------
Stage 2: The ID/IF register reads the input **SIMSL 13, 10, 12** in binary and outputs the opcode to the reg file.
Stage 3: The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. This register will store the r1,r2,r3, just in case we need them. As the instruction is OR, we are getting values from reg0 and reg10 values of r1(0) = (00000000000000000000000000000000), r2(10) = (07d00000000001f40000000000000190), and r3 = 0 (As it's an R3 instruction).
Stage 4: The register EXE/WB gets the output of the ALU. As the instruction is NOP, we will not enable the write-back. If right back is not enabled, both rb and write-back data values can be anything the BUP will not be affected.
Forwarding Unit: —----------

Cycle 8:

Stage 1: —----------
Stage 2: —----------
Stage 3: The IF/EXE register outputs the 3 data (r1,r2,r3) values and Opcode function. This register will store the r1,r2,r3, just in case we need them. As the instruction is SIMSL, we are getting values from reg12 and reg10 values of r1(12) = (07d00000000001f40000000000000190), r2(10) = (07d00000000001f40000000000000190), and r3(10) = (07d00000000001f40000000000000190) (it's an R4 instruction).
Stage 4: The register EXE/WB gets the output of the ALU. As the instruction is OR, we will enable the write-back and output the rd as (01100 or 12_2) and the write-back value as (07d00000000001f40000000000000190) as we performed OR operation reg10 and reg0. Reg0 has a 0 value, so essentially, we are storing the value of reg10 in reg12 (ALU will compute the OR operation).
Forwarding Unit: We need to forward data as the next instruction uses the same register. We are fetching the value of reg10, but it takes 4 cycles to write back the value. We will forward the reg10 value to the ALU, which is performing the operation to get the write-back value.

Conclusion

Upon completing our project, we discovered that the four-stage multimedia unit's pipeline design was executed correctly. This conclusion was drawn after thorough testing and validation, where we compared the actual outputs against our predicted results. The consistency of the pipelined unit's outputs with our expectations underscored the importance of understanding pipelining. The project deepened our knowledge of how various components—such as the ALU, instruction buffer, forwarding unit, mux, register files, and stage registers—integrate seamlessly to achieve accurate results. The construction process highlighted the delicate balance required in pipeline design, demonstrating how even minor alterations can lead to incorrect outputs. Reflecting on the project, it is clear that the successful collaboration of different parts culminated in an efficiently functioning multimedia unit. This unit serves as a practical example of how pipelining can enhance CPU efficiency by increasing throughput. Overall,

the project not only reinforced the team's appreciation for the intricacies of pipelining but also revealed the complex nature of what might seem like a straightforward concept.

Source Code

ALU

```
`timescale 1ns / 1ps

module ALU_unit(

    input logic[127:0] Rs1,
    input logic[127:0] Rs2,
    input logic[127:0] Rs3,
    input logic[24:0] OPCODE,
    output logic[127:0] Rd

    );

    bit[1:0] InputCase;

    assign InputCase = OPCODE[24:23];

    localparam OP_LOAD = 2'b00; // Condition for Load Function.
    localparam OP_R4 = 2'b10;   // Condition for R4 Function.
    localparam OP_R3 = 2'b11;   // Condition for R3 Funtion.


    //For Load
    logic [2:0] LoadIndex;
    logic [15:0] Immediate;
    logic [127:0] tmp_rd;
    assign LoadIndex = OPCODE[23:21];
    assign Immediate = OPCODE[20:5];

    //For R3
    logic [3:0] OpCodeR3;
    int result, shift_amount;
    assign OpCodeR3 = OPCODE[15+:4];
```

```
//For R4
logic [2:0] OpcodeR4;
longint result_64;
int result;
assign OpcodeR4 = OPCODE[22:20];



always_comb begin
case(InputCase)
    OP_LOAD,2'b01:
    begin
        // logic for Load Immidiate
        tmp_rd[LoadIndex*16 +: 16] = Immediate;
     end
     OP_R4:
     begin
        // logic for R4
        case(OpcodeR4)
        3'b000:
        begin
            //Signed Integer Multiply-Add Low with Saturation
            for(int i = 0; i < 4; i++) begin
                result = $signed(Rs3[i*32 +: 16]) * $signed(Rs2[i*32 +: 16]) + $signed(Rs1[i*32 +: 32]);
                if(result > 32'h7FFFFFFF) result = 32'h7FFFFFFF;
                else if(result < 32'h80000000) result = 32'h80000000;
                tmp_rd[i*32 +: 32] = result;
            end
        end
        3'b001:
        begin
            //Signed Integer Multiply-Add High with Saturation
            for(int i = 0; i < 4; i++) begin
                result = $signed(Rs3[i*32 + 16 +: 16]) * $signed(Rs2[i*32 + 16 +: 16]) + $signed(Rs1[i*32 +: 32]);
                if(result > 32'h7FFFFFFF) result = 32'h7FFFFFFF;
                else if(result < 32'h80000000) result = 32'h80000000;
                tmp_rd[i*32 +: 32] = result;
```

```
            end
        end
        3'b010:
        begin
            //Signed Integer Multiply-Subtract Low with Saturation
            for(int i = 0; i < 4; i++) begin
                result = $signed(Rs1[i*32 +: 32]) - ($signed(Rs3[i*32 +: 16]) * $signed(Rs2[i*32 +: 16]));
                if(result > 32'h7FFFFFFF) result = 32'h7FFFFFFF;
                else if(result < 32'h80000000) result = 32'h80000000;
                tmp_rd[i*32 +: 32] = result;
            end
        end
        3'b011:
        begin
            //Signed Integer Multiply-Subtract High with Saturation
            for(int i = 0; i < 4; i++) begin
                result = $signed(Rs1[i*32 +: 32]) - ($signed(Rs3[i*32 + 16 +: 16]) * $signed(Rs2[i*32 + 16 +: 16]));
                if(result > 32'h7FFFFFFF) result = 32'h7FFFFFFF;
                else if(result < 32'h80000000) result = 32'h80000000;
                tmp_rd[i*32 +: 32] = result;
            end
        end
        3'b100:
        begin
            //Signed Long Integer Multiply-Add Low with Saturation
            for(int i = 0; i < 2; i++) begin
                result_64 = $signed(Rs3[i*64 +: 32]) * $signed(Rs2[i*64 +: 32]) + $signed(Rs1[i*64 +: 64]);
                if(result_64 > 64'h7FFFFFFFFFFFFFFF) result_64 = 64'h7FFFFFFFFFFFFFFF;
                else if(result_64 < 64'h8000000000000000) result_64 = 64'h8000000000000000;
                tmp_rd[i*64 +: 64] = result_64;
            end
        end
        3'b101:
        begin
            //Signed Long Integer Multiply-Add High with Saturation
            for(int i = 0; i < 2; i++) begin
                result_64 = $signed(Rs3[i*64 + 32 +: 32]) * $signed(Rs2[i*64 + 32 +: 32]) + $signed(Rs1[i*64 +: 64]);
                if(result_64 > 64'h7FFFFFFFFFFFFFFF) result_64 = 64'h7FFFFFFFFFFFFFFF;
                else if(result_64 < 64'h8000000000000000) result_64 = 64'h8000000000000000;
```

```
                tmp_rd[i*64 +: 64] = result_64;
            end
        end
        3'b110:
        begin
            //Signed Long Integer Multiply-Subtract Low with Saturation
            for(int i = 0; i < 2; i++) begin
                result_64 = $signed(Rs1[i*64 +: 64]) - ($signed(Rs3[i*64 +: 32]) * $signed(Rs2[i*64 +: 32]));
                if(result_64 > 64'h7FFFFFFFFFFFFFFF) result_64 = 64'h7FFFFFFFFFFFFFFF;
                else if(result_64 < 64'h8000000000000000) result_64 = 64'h8000000000000000;
                tmp_rd[i*64 +: 64] = result_64;
            end
        end
        3'b111:
        begin
            // Signed Long Integer Multiply-Subtract High with Saturation
            for(int i = 0; i < 2; i++) begin
                result_64 = $signed(Rs1[i*64 +: 64]) - ($signed(Rs3[i*64 + 32 +: 32]) * $signed(Rs2[i*64 + 32 +: 32]));
                if(result_64 > 64'h7FFFFFFFFFFFFFFF) result_64 = 64'h7FFFFFFFFFFFFFFF;
                else if(result_64 < 64'h8000000000000000) result_64 = 64'h8000000000000000;
                tmp_rd[i*64 +: 64] = result_64;
            end
        end

      endcase
    end
    OP_R3:
    begin
        // logic for R3
        case(OpCodeR3)
        4'b0000:
            begin
                //NOP
            end
        4'b0001:
            begin
                //shift right halfword immediate
                for(int i = 0; i < 8; i++) begin
                    tmp_rd[i*16 +: 16] = Rs1[i*16 +: 16] >>> Rs2[0 +: 4];
```

```
            end
        end
     4'b0010:
        begin
            //AU: add word unsigned
            for(int i = 0; i < 4; i++) begin
                tmp_rd[i*32 +: 32] = Rs1[i*32 +: 32] + Rs2[i*32 +: 32];
            end
        end
    4'b0011:
        begin
            //CNT1H: count 1s in halfword
            for(int i = 0; i < 8; i++) begin
                tmp_rd[i*16 +: 16] = $countones(Rs1[i*16 +: 16]);
            end
        end
    4'b0100:
        begin
            //AHS: add halfword saturated
            for(int i = 0; i < 8; i++) begin
                result = $signed(Rs1[i*16 +: 16]) + $signed(Rs2[i*16 +: 16]);
                if(result > 32767) result = 32767;
                else if(result < -32768) result = -32768;
                tmp_rd[i*16 +: 16] = result;
            end
        end
     4'b0101:
        begin
            // OR: bitwise logical or
            tmp_rd = Rs1 | Rs2;
        end
     4'b0110:
        begin
            //BCW: broadcast word
            tmp_rd = {4{Rs1[0 +: 32]}};
        end
    4'b0111:
        begin
             //MAXWS: max signed word
```

```
                    for(int i = 0; i < 4; i++) begin
                        tmp_rd[i*32 +: 32] = ($signed(Rs1[i*32 +: 32]) > $signed(Rs2[i*32 +: 32])) ? Rs1[i*32 +: 32] : Rs2[i*32
+: 32];
                    end
                end
            4'b1000:
                begin
                   //MINWS: min signed word
                    for(int i = 0; i < 4; i++) begin
                        tmp_rd[i*32 +: 32] = ($signed(Rs1[i*32 +: 32]) < $signed(Rs2[i*32 +: 32])) ? Rs1[i*32 +: 32] : Rs2[i*32
+: 32];
                    end
                end
             4'b1001:
                begin
                    //MLHU: multiply low unsigned
                    for(int i = 0; i < 4; i++) begin
                        tmp_rd[i*32 +: 32] = (Rs1[i*32 +: 16]) * (Rs2[i*32 +: 16]);
                    end
                end
            4'b1010:
                begin
                    //MLHSS: multiply by sign saturated
                    for(int i = 0; i < 8; i++) begin
                        if(Rs2[i*16 +: 16] < 0) tmp_rd[i*16 +: 16] = -Rs1[i*16 +: 16];
                        else if(Rs2[i*16 +: 16] > 0) tmp_rd[i*16 +: 16] = Rs1[i*16 +: 16];
                        else tmp_rd[i*16 +: 16] = 0;
                    end
                end
            4'b1011:
                begin
                    //AND: bitwise logical and
                    tmp_rd = Rs1 & Rs2;
                end
             4'b1100:
                begin
                    //INVB: invert (flip) bits
                    tmp_rd = ~Rs1;
                end
```

```
            4'b1101:
                begin
                    //ROTW: rotate bits in word
                    for(int i = 0; i < 4; i++) begin
                        shift_amount = Rs2[i*32 +: 5];
                        tmp_rd[i*32 +: 32] = (Rs1[i*32 +: 32] >>> shift_amount) | (Rs1[i*32 +: 32] << (32-shift_amount));
                    end
                end
            4'b1110:
                begin
                    //SFWU: subtract from word unsigned
                    for(int i = 0; i < 4; i++) begin
                        tmp_rd[i*32 +: 32] = Rs2[i*32 +: 32] - Rs1[i*32 +: 32];
                    end
                end
             4'b1111:
                begin
                    //SFHS: subtract from halfword saturated
                    for(int i = 0; i < 8; i++) begin
                        result = $signed(Rs2[i*16 +: 16]) - $signed(Rs1[i*16 +: 16]);
                        if(result > 32767) result = 32767;
                        else if(result < -32768) result = -32768;
                        tmp_rd[i*16 +: 16] = result;
                    end
                end
            endcase

         end
        endcase
        Rd = tmp_rd;
        end
endmodule
```

Instruction Buffer

```
`timescale 1ns / 1ps

module InstructionBuffer (
    input  logic clk,
    input  string asmfile,
    output logic [24:0] instruction
);

    integer read_finish = 0;
    integer PC = 0;
    logic [24:0] instruction_field [0:63];
    integer i = 0;
    integer file_descriptor;
    string line;

    initial begin
        if (read_finish == 0) begin
            file_descriptor = $fopen(asmfile, "r");
            if (file_descriptor) begin
                while (!$feof(file_descriptor)) begin
                    $fgets(line, file_descriptor);
                    instruction_field[i] = line.atobin();
                    i = i + 1;
                end
                read_finish = 1;
                $fclose(file_descriptor);
            end
        end
    end

    always_ff @(posedge clk) begin
        if (read_finish == 1 && PC < 64) begin
             instruction <= instruction_field[PC];
              PC <= PC + 1;
        end
    end

endmodule
```

Register File

```
`timescale 1ns / 1ps
module reg_file (
    input  logic [24:0] opcode,
    input  logic WE,
    input  logic [4:0] rd,
    input  logic [127:0] data,
    output logic [127:0] data1, data2, data3
);

    // Define the registers array
    logic [127:0] registerfile[0:31];

    // Initialize the registers array to zero
    initial begin
        for (int i = 0; i < 32; i++) begin
            registerfile[i] = 128'b0;
        end
    end

    // Write process
    always_comb begin
        if (WE) begin
            registerfile[rd] = data;
        end
    end

    // Read process
    always_comb begin
        data1 = registerfile[opcode[9:5]];
        data2 = registerfile[opcode[14:10]];
        data3 = registerfile[opcode[19:15]];

        // If load instruction, then read address is different
        if (opcode[24] == 1'b0) begin
            data1 = registerfile[opcode[4:0]];
        end
    end

endmodule
```

Mux

```
module mux (
    input  logic s1, s2, s3,
    input  logic [127:0] d1m, d2m, d3m,
    input  logic [127:0] data,
    output logic [127:0] rs1, rs2, rs3
);

    always_comb begin
        // Default assignments
        rs1 = d1m;
        rs2 = d2m;
        rs3 = d3m;

        // Conditional assignments based on select signals
        if (s1) rs1 = data;
        if (s2) rs2 = data;
        if (s3) rs3 = data;
    end

endmodule
```

Forwarding Unit

```
module forward_unit (
    input  logic [24:0] instr,
    input  logic [24:0] opcode,
    output logic s1,
    output logic s2,
    output logic s3
);

    always_comb begin
        // Initialize outputs to 0
        {s1, s2, s3} = 3'b000;

        // Check the opcode and make decisions
        if (opcode[24:15] == 10'b1100000000) begin
            // Do nothing
        end else if ((opcode[24] == 1'b0) || (opcode[24] == 1'b1)) begin
            // For going into load
            if (instr[24] == 1'b0) begin
                if (opcode[4:0] == instr[4:0]) begin
                    s1 = 1'b1;
                end
            end else begin
                // For going into R3/R4 instructions
                if (opcode[4:0] == instr[9:5]) begin  // For rd = rs1
                    s1 = 1'b1;
                end
                if (opcode[4:0] == instr[14:10]) begin // For rd = rs2
                    s2 = 1'b1;
                end
                if (opcode[4:0] == instr[19:15]) begin // For rd = rs3 (only in R4 instructions, R3 ignores rs3 values)
                    s3 = 1'b1;
                end
            end
        end
    end

endmodule
```

IF/ID Register

```
module IfId_reg (
    input  logic clk,
    input  logic [24:0] instr,
    output logic [24:0] opcode
);

    always_ff @(posedge clk) begin
        opcode <= instr;
    end

endmodule
```

ID/IX Register

```
module IdEx_reg (
    input  logic clk,
    input  logic [24:0] opcode,
    input  logic [127:0] d1in, d2in, d3in,
    output logic [24:0] func,
    output logic [127:0] d1out, d2out, d3out
);

    always_ff @(posedge clk) begin
        func <= opcode;
        d1out <= d1in;
        d2out <= d2in;
        d3out <= d3in;
    end

endmodule
```

EX/WB Register

```
`timescale 1ns / 1ps

module ex_wb_stage_reg (
    input  logic clk,
    input  logic [24:0] func,
    input  logic [127:0] dataEX,
    output logic [24:0] field,
    output logic [127:0] dataWB,
    output logic WE,
    output logic [4:0] rd
);

    always_ff @(posedge clk) begin
        field <= func;
        rd <= func[4:0];
        dataWB <= dataEX;
        WE <= 1'b0;

        if (func[24] == 1'b0 || func[24:23] == 2'b10 || func[24:23] == 2'b11) begin
            WE <= 1'b1;
        end

        if (func[24:15] == 10'b1100000000) begin
            WE <= 1'b0;
        end
    end

endmodule
```

CPU

```
`timescale 1ns / 1ps

module the_CPU (
    input  logic clk,
    input  string filename
);

    logic [24:0] InstructionBuff_Out,  twelve, seventeen,IfOpcode_out;
    logic [127:0] three, four, five, six, seven, eight, nine, ten, eleven, thirteen, sixteen;
    logic fourteen, s1, s2, s3;
    logic [4:0] fifteen;

    // Instantiate the sub-modules
    InstructionBuffer U1 (
        .clk(clk),
        .asmfile(filename),
        .instruction(InstructionBuff_Out)
    );

    IfId_reg U2 (
        .clk(clk),
        .instr(InstructionBuff_Out),
        .opcode(IfOpcode_out)
    );

    reg_file U3 (
        .opcode(IfOpcode_out),
        .WE(fourteen),
        .rd(fifteen),
        .data(sixteen),
        .data1(three),
        .data2(four),
        .data3(five)
    );

    IdEx_reg U4 (
        .clk(clk),
        .opcode(IfOpcode_out),
        .d1in(three),
        .d2in(four),
        .d3in(five),
        .func(twelve),
        .d1out(six),
```

```
        .d2out(seven),
        .d3out(eight)
    );

    mux U5 (
        .s1(s1),
        .s2(s2),
        .s3(s3),
        .d1m(six),
        .d2m(seven),
        .d3m(eight),
        .data(sixteen),
        .rs1(nine),
        .rs2(ten),
        .rs3(eleven)
    );

       ALU_unit U6 (
        .Rs1(nine),
        .Rs2(ten),
        .Rs3(eleven),
        .OPCODE(twelve),
        .Rd(thirteen)
    );

    ExWd_reg U7 (
        .clk(clk),
        .func(twelve),
        .dataEX(thirteen),
        .field(seventeen),
        .dataWB(sixteen),
        .WE(fourteen),
        .rd(fifteen)
    );

    forward_unit U8 (
        .instr(twelve),
        .opcode(seventeen),
        .s1(s1),
        .s2(s2),
        .s3(s3)
    );

endmodule
```

Testbench

```
// Testbench for Pipelined MMU

module the_CPU_tb;

    // Parameters and signals
    parameter period = 100; // Time unit is assumed to be nanoseconds in SystemVerilog
    logic clk = 0;

    string filename = "/home/shrujan/Desktop/VHDL_files/asmFile/asmCode.txt"; // File path

    // Instantiate the unit under test (UUT)
    the_CPU UUT1 (
        .clk(clk),
        .filename(filename)
    );

    // Clock generation and test control
    initial begin
        #10;
        forever begin
        #5 clk = ~clk; // Toggle the clock every 10 time units
    end
//        for (int i = 0; i < 128; i++) begin
//            clk = ~clk;
//            # (period / 2);
//        end
        //$finish;
    end

endmodule
```

Assembly Code

```
// Assembly test file
LI      10, 4(27)
MLHU   10, 10, 10
LI      11, 4(27)
OR      12, 0, 3
MLHU   11, 11, 12
SIMAL   12, 10, 11
LI      10, 4(27)
LI      5, 4(90)
LI      2, 4(30)
LI      3, 4(10)
LI      4, 4(3)
LI      12, 4(200)
LI      12, 7(900)
LI      12, 1(1000)
SIMAH   5, 2, 3, 4
SIMSL   10, 5, 11, 12
SIMSH   18, 10, 12, 11
SLIMAL  4, 12, 12, 0
SLIMAH  20, 0, 5, 12
SLIMSL  1, 5, 10, 18
SLIMSH  15, 20, 1, 0
NOP
SHRHI   20, 2, 3
AU      11, 15, 20
CNT1H   14, 1, 1
AHS     17, 4, 20
OR      12, 12, 3
BCW     23, 13, 4
MAXWS   26, 17, 23
MINWS   29, 17, 4
MLHU    31, 12, 1
MLHSS   30, 12, 13
AND     17, 17, 1
INVB    20, 17, 17
ROTW    23, 26, 20
SFWU    26, 10, 30
```

```
SFHS    29, 20, 26
```