Enhancing RISC and MSAP Computer Architectures by Interfacing I/O Ports

Michael Judrick Aringo
Department of Electronics and Computer Engineering
De La Salle University
2401 Taft Avenue, Manila, Philippines
michael_aringo@dlsu.edu.ph

John Matthew Enciso
Department of Electronics and Computer Engineering
De La Salle University
2401 Taft Avenue, Manila, Philippines
john_enciso@dlsu.edu.ph

Louie Que
Department of Electronics and Computer Engineering
De La Salle University
2401 Taft Avenue, Manila, Philippines
louie_que_a@dlsu.edu.ph

Sean Patrick Tan
Department of Electronics and Computer Engineering
De La Salle University
2401 Taft Avenue, Manila, Philippines
sean_patrick_y_tan@dlsu.edu.ph

***Abstract* - This project seeks to increase the number of input/output ports in the MSAP architecture and thereby improve performance and usability with the aid of Vivado software. The modified architecture was tested on the operations of sorting and arithmetic, concentrating on new I/O ports coupled with changes in the control level. From the numerous tests and analyses of the data, it was a success, as the system passed the testing parameters. The efficiencies of the architecture were enhanced, and further improvements could be targeted for the application of these same concepts to other instruction set architectures, such as CISC or MIPS.**

Index Terms - MSAP architecture, computer architecture, RISC-V, interfacing ports, architecture enhancement;

I. Introduction

A. Introductory Information

With the Reduced Instruction Set Computer (RISC) architecture, each instruction performs a single, simple operation [1]. The datapath, control unit, program memory, and data memory are the usual components of a RISC model [2]. When using Vivado to program and simulate the results of the Modified Simple as Possible Instruction Set, it is important that the proper version and installation of Vivado are used.

The addition of input/output ports to the board will be done by declaring the needed or specified input and output devices to be used in the topmost module in the program code. Then, to properly interface the ports, the constraint file of the board must be modified so that the board will be able to accept input and display output [3]. Thus, it is expected that the addition of input/output ports to the computer will allow users to better interface and interact with the computer as opposed to the absence of the ports. These additional features are beneficial as they make the computer more efficient, versatile, and multifunctional.

This project aimed to enhance the performance and usability of the MSAP architecture by adding additional input/output ports through the use of the program used during the laboratory class. The group has done this by first modifying the control unit to take into account the additional input/output ports, along with adding the requisite code to the other parts if needed.

B. Objectives

The general objective of this study is to design the MSAP architecture with more input/output ports to enhance its performance and usability using Vivado software. The specific objectives of this study are as follows:

- To modify the control unit of the MSAP architecture in a way that will be able to accommodate newly added input/output ports.
- To simulate in Vivado to test the functionality of the enhanced MSAP architecture with the additional ports.
- To assess the performance of the system with regard to performance improvement and usability enhancements through the integration of new input/output ports.
- To test the capability of the system using the specified test parameters

C. Implemented Features and Test Parameters

The implemented features of this project are additional I/O ports. Other than the specified, PORTA, PORTB, PORTC, and PORTD, an additional PORTE and PORTF were added to the architecture set. This was done by modifying several components of the MSAP Instruction Set. The control unit was modified to include the additional ports. Simulating the results by inspecting the timing diagram was done to see if the ports were properly added.

The performance of the newly modified MSAP was tested by ensuring it could handle sorting operations. A variety of sorting algorithms were tested to indicate the capabilities of the instruction set. The three sorting algorithms used were bubble sort, insertion sort, and selection sort.

Additionally, the ability of the system was also tested using multiplication and division operations. The implementation of this was based on the RISC-V architecture. It was noted that the system should be able to handle operations on unsigned integers and fractions. This test was done to determine whether the system could carry out more complex arithmetic operations.

Furthermore, more tests were conducted on the architecture with regard to specific enhancements, among which were the addition of new I/O ports and changes in the control unit. These have ranged from systems that interface with external devices to those that process data from the new ports or merely handle control signals to accommodate new functionalities.

Lastly, the data acquired was statistically analyzed by comparing the execution time of the modified architecture based on this group's enhancements. Additionally, the performance will be tested by comparing the amount of time or clock cycles it takes to complete a certain task or sorting algorithm in comparison to other architectures.

II. Computer Architecture Specifications and Applications

A. *Application Area*

As the project's main focus was to modify MSAP and RISC Architecture by supplying them with additional I/O ports, it is important to note that this was done to make the computer more efficient, versatile, and multifunctional.

The integration of the additional ports was done by making modifications to the entire architecture, mainly by modifying the control unit and memory unit. Since the control unit is where the ports were initialized, new control logic was done to accommodate the newly added ports. To ensure that the ports were added, the architecture was simulated through Vivado to verify the presence of the ports in the system. Since the base program followed a modular set-up, the group continued this practice and maintained the modularity of the system by keeping each component in different modules or files for easier management. To meet expectations, various testing parameters were used to verify the capabilities of the system. As mentioned above, which will also be further explained in the subsequent sections, testing parameters include sorting, arithmetic ability, and port verification.

Several alternative design modifications could be made to the instruction set architecture; however, this project mainly focuses on the addition of ports. There are other modifications and additions available in order to make the system more efficient, capable, and robust.

B. *Instruction Set Architecture*

<u>RISC Architecture</u>

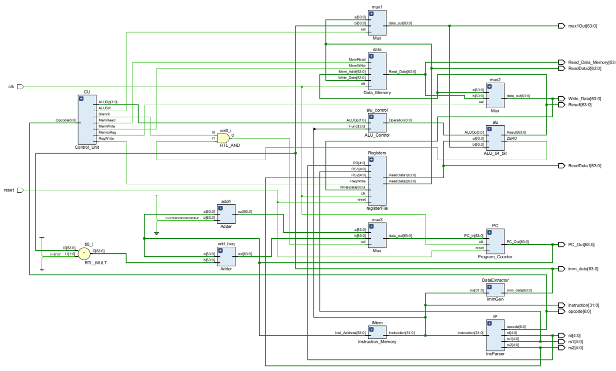


Fig. 1. Block diagram of the RISC architecture

The figure above shows that the RISC architecture is mainly comprised of a control unit, a data memory, ALU, register file module, instruction memory module, a program counter module, and multiple adders and multiplexers. The RISC-V for Sorting project, based on the Pipelined-RISC-V-Processor by Samiya Ali Zaidi and Javeria Azfar, features a pipelined architecture with a 64-bit data path, leveraging the flexibility of the RISC-V instruction set architecture (ISA). Key improvements include the addition of advanced ALU operations and expanded sorting options, enhancing the processor's functionality and performance [4].

The added code in the ALU_64_bit module enhances the original functionality from the sourced project by incorporating normal integer multiplication, fixed-point multiplication, and division operations. The normal integer multiplication is handled by multiplying the two 64-bit inputs (a and b), with the result being stored directly in the Result output. For fixed-point multiplication, the inputs are first multiplied as a 128-bit intermediate value to accommodate the fractional parts, and then the result is right-shifted by 32 bits to maintain the correct fixed-point format in a 64-bit output. This shift ensures that the result is scaled down appropriately by a factor of 2^32. The division operation checks if the divisor (b) is not zero before performing the division. If b is zero, the result is set to an undefined value (64'hX) to safely handle the division by zero scenario, which is a common concern in hardware design. These enhancements, built upon the original project, enable the ALU to perform a broader range of arithmetic operations, including both basic and more complex fixed-point calculations, with built-in safeguards for potential errors.

<u>MSAP Architecture</u>

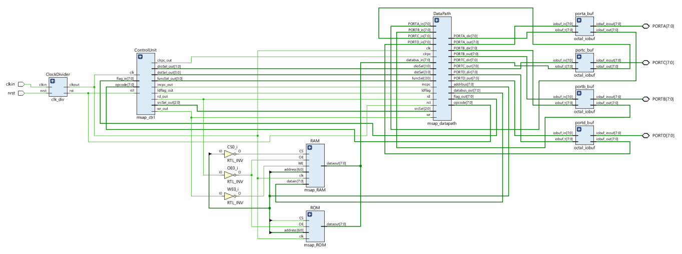


Fig. 2. MSAP architecture block diagram

The MSAP architecture, in accordance with the figure above, is comprised of a clock divider circuit, a control unit, RAM, ROM, a datapath, and four buffer ports.

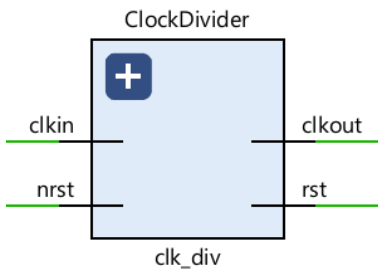


Fig. 3. Clock divider diagram of the MSAP architecture

Firstly, the clock divider circuit is an instance of the 'clk_div' module. This module acts as both a reset synchronizer and a clock divider. The inputs are the 'clkin' and 'nrst' variables representing the input clock and active-low reset signals, respectively. Meanwhile, it's output includes 'clkout' and 'rst' signals representing the divided clock output and synchronized reset signals. The internal registers contained in this modules are two eight-bit register counters for the clock division and reset synchronization, as well as two one-bit registers for the output of the divided clock and reset signal.

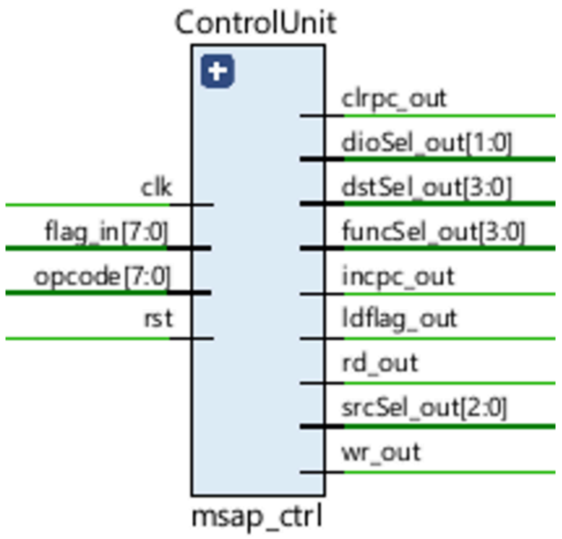


Fig. 4. Control unit of the MSAP architecture

The second submodule of the MSAP architecture is the control unit, which is an instance of the 'msap_ctrl' module. This module is responsible for the control signals and state transitions that are needed to execute several instructions represented by the opcode and other variables, such as input flags. Its inputs include the clock signal, reset signal, and two eight-bit vectors that represent the opcode and the different status flags. On the other hand, its outputs include the likes of output signals manipulating the program counter, memory operations, and status flags. It also contains vectors having a size of two-bits to four-bits, representing the selection of the ALU function, source registers, destination registers, and I/O channel of the data. Internal registers include the current and next state of the state machine, control signals for the mentioned operations, and signals for the selection procedure.

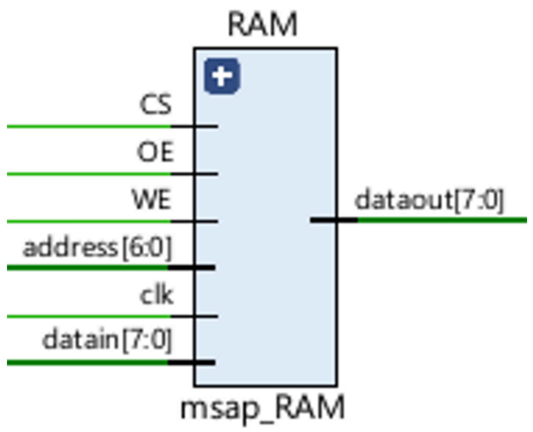


Fig 5. RAM of the MSAP architecture

The next component of the MSAP architecture is the Random Access Memory module, which is designed as the 'msap_RAM' module. This component is utilized for memory read and memory write based from the passed control signals and memory addresses. It consists of six different inputs: a clock signal, control signals for device and read/write enable, two vectors for the memory address and data. A singular output is produced by this module, which represents the output data.

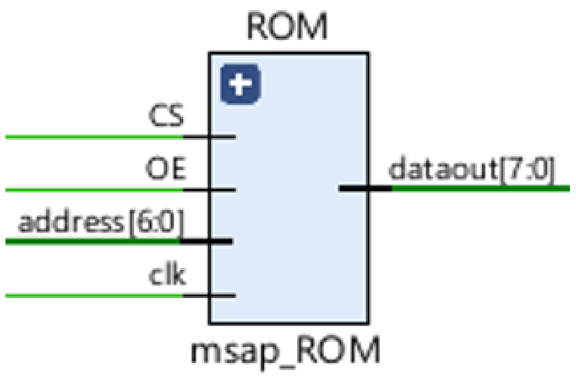


Fig. 6. ROM of the MSAP architecture

A Read-Only Memory module is obtained for the MSAP architecture by obtaining an instance of the 'msap_ROM' module. This module provides the stored instructions and outputs to the system based from the specified memory address. Its inputs are the signals for the clock, chip select, and output enable, as well as the seven-bit that represents the memory address. A singular output named 'dataout' is produced by this module that represents the output data.

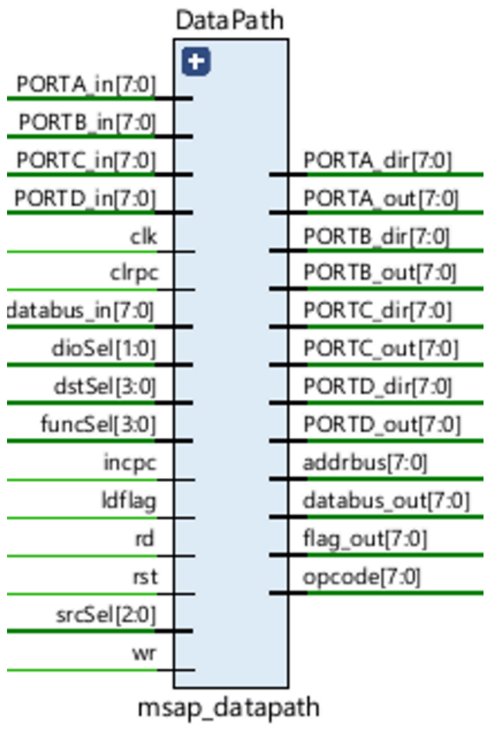


Fig. 7. Datapath of the MSAP architecture

One of the most important modules of the MSAP architecture is the datapath, which, in this project, is an instance of the 'msap_datapath' module. This component is developed by utilizing other submodules, including program counter, memory data register, memory address register, accumulator, instruction register, ALU, I/O ports, and B register. This is the reason why it consists of numerous inputs like input vectors for the four ports, a clock signal, program counter manipulators, operation selection, and a status flag loader. Similarly, its outputs include the opcode, status flags, data bus, and address bus, which are all expressed as eight-bit vectors, and the data and direction outputs of the four ports.

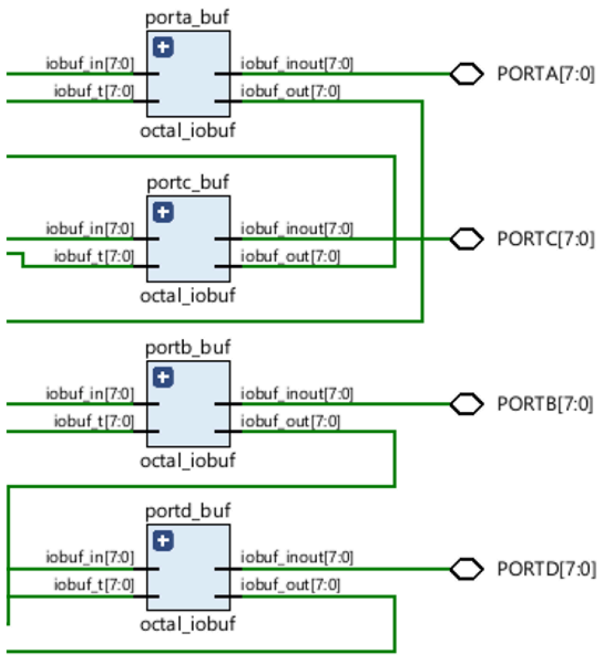


Fig 8. I/O ports of the MSAP architecture

To interface the computer architecture with external devices, four ports are utilized. All of these ports are developed using the 'octal_iobuf' module. This has two inputs and two outputs, where one of the outputs can be easily programmed as an input port or an output port by the architecture.

C. *Computer Performance Testing*

In order to verify and test the parameters created inside the RISC and MSAP architectures, two tests will be made for the experimentation phase. Firstly, the implementation of three algorithms within the architecture was made to gauge and benchmark the implemented variation within the MSAP and RISC architectures. The algorithms that will be used are selection, bubble, and insertion sorting. These three different types of algorithms are made and created as instruction memory to measure and visualize the result inside the timing diagram window. To properly verify the three algorithms, the visual diagram will be looked into and tested to see whether the values were properly sorted with the use of the different types of sorting algorithms. The measurement with respect with the clock pulse is also measured since the total time it takes to finish the sort will be the measurement whether the proposed architecture was successful in performing the desired tasks needed for a computer architecture. The next part is a simulation of multiplication and division operations using integers and unsigned numbers, By looking into the timing diagram, the results will determine if the architecture was successful in completing the arithmetic operations needed.

III. Design Operation, Test Data and Statistical Analysis of Results

A. RISC Architecture

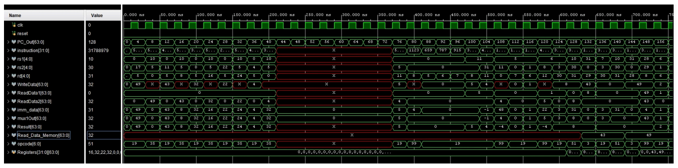


Fig. 9. Selection Sorting Algorithm

The figure above shows the application of the selection sort algorithm. The selection sort algorithm is a type of sorting algorithm that iterates through the values to find the lowest value. Upon detecting the low value, it will then compare the value to the succeeding value, testing which value is much lower. Upon finding the lowest value, it will then be placed at the top of the sorted pile, and the action iterates. In the timing diagram, the algorithm clearly showed the sorting algorithm in action since it properly organized the values within the PC_Out[3:0] variable in ascending order. Hence, it completed its sort and organized the values from lowest to highest.

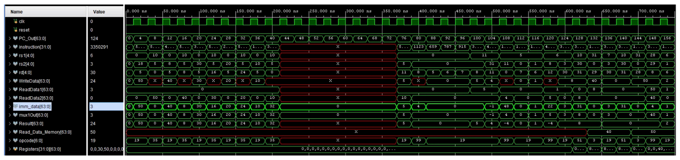


Fig. 10. Bubble Sorting Algorithm

The figure above shows the application of a bubble sort algorithm. Bubble sort is a type of algorithm that starts with the first item and compares it to the second item. If the second item is much lower than the first, it will switch places. This iterates until each and every variable is sorted in ascending order. The timing diagram clearly shows that the outputs were organized in ascending order. Therefore, it is necessary to verify that the algorithm was properly implemented and that the data was organized in ascending order.

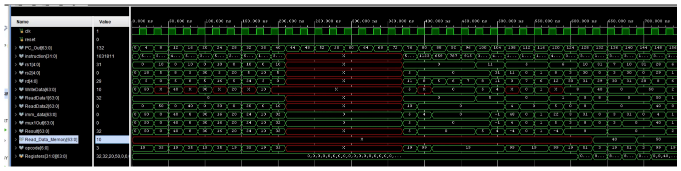
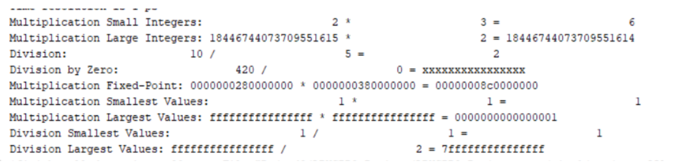


Fig. 11. Insertion Sorting Algorithm

The diagram above shows an application of the insertion sort algorithm. The insertion sort algorithm iteratively inserts each element of an unsorted list into its correct position in a sorted portion of the list. This will iterate until each element is correctly placed in position. The timing diagram clearly shows that the sorting algorithm organized the values from lowest to highest.

<u>Multiplication / Division of integer and fractional (unsigned) numbers</u>


```
Multiplication Small Integers:                    2 *                  3 =                    6
Multiplication Large Integers: 18446744073709551615 *                  2 = 18446744073709551614
Division:                 10 /                  5 =                  2
Division by Zero:                420 /                  0 = xxxxxxxxxxxxxxxx
Multiplication Fixed-Point: 0000000280000000 * 0000000380000000 = 00000008c0000000
Multiplication Smallest Values:                   1 *                  1 =                    1
Multiplication Largest Values: ffffffffffffffff * ffffffffffffffff = 0000000000000001
Division Smallest Values:                   1 /                  1 =                    1
Division Largest Values: ffffffffffffffff /                  2 = 7fffffffffffffff
```
Fig. 12. Output in Console

The output of the ALU testbench shows a series of arithmetic operations performed on 64-bit integers. The multiplication of two small integers, 2 and 3, correctly yields a result of 6. For larger values, multiplying the maximum 64-bit unsigned integer (18446744073709551615) by 2 results in 18446744073709551614, demonstrating an overflow condition where the least significant bit is lost due to exceeding the register size. The division of 10 by 5 gives the expected result of 2, while attempting to divide by zero is handled appropriately, outputting an error representation. When performing fixed-point multiplication, the values 2.5 and 3.5 (represented in hexadecimal) multiply to produce 8.75 in fixed-point format. The smallest integer multiplication, 1 * 1, correctly outputs 1, while multiplying the maximum 64-bit value by itself results in an overflow, yielding a wrapped-around value of 1. Similarly, the division of 1 by 1 outputs 1, and dividing the maximum 64-bit value by 2 produces 9223372036854775807, which is the correct halved value. Overall, the ALU testbench effectively demonstrates the handling of standard operations, overflow, and edge cases like division by zero.

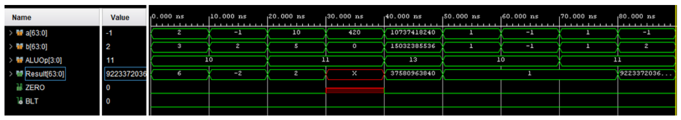


Fig. 13. Waveform of new ALU instructions

The waveform illustrates the results of various ALU operations as specified in the testbench. The key signals shown are a, b, ALUOp, Result, and ZERO, with the operations and results corresponding to different ALUOp codes. Each of these results is reflected in the waveform, showing the corresponding values at different times. The zero flag remains low except during the division by zero, confirming the accuracy of non-zero results.

B. *MSAP Architecture*

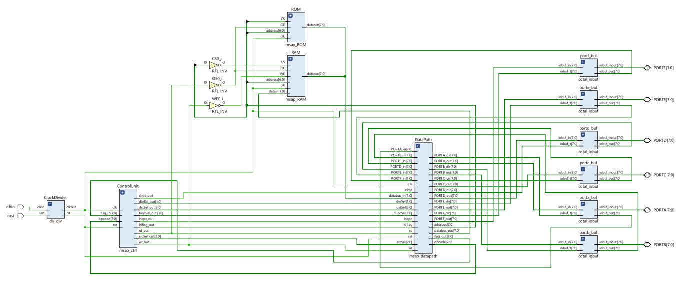


Fig. 14. Block diagram of the modified MSAP architecture with additional ports

The figure above shows the block diagram of the MSAP architecture after the enhancement provided by the additional ports named PORTE and PORTF.

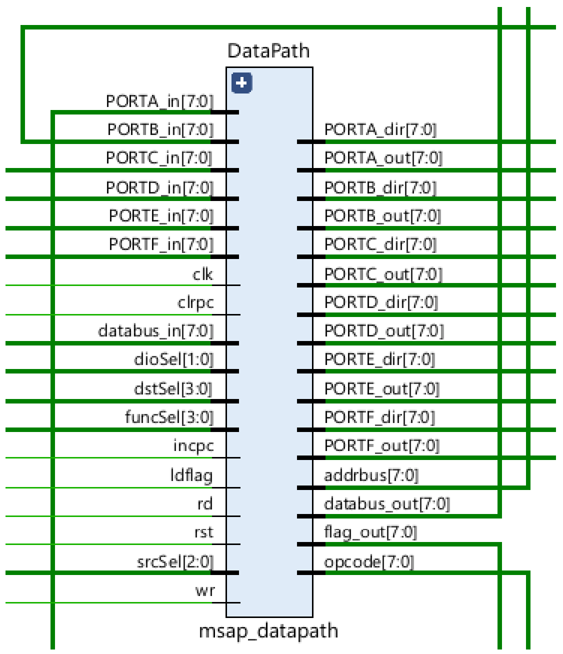


Fig. 15. Block diagram of the modified MSAP datapath

The block diagram of the datapath component of the MSAP architecture is illustrated in the figure above. The datapath is important as this component handles how data is processed and moves between the different components in the computer. As seen, the diagram shows the additional two ports. The figure clearly illustrates the presence of the input, output, and direction data buses for each port from PORTA to PORTF.

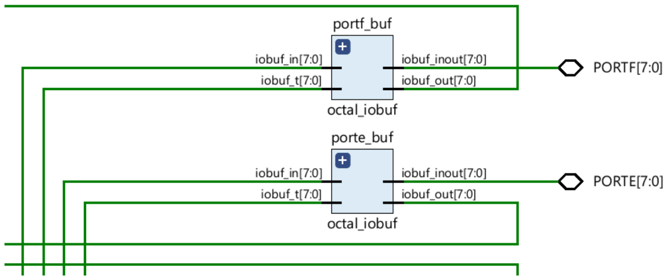


Fig. 16. Block diagram of the additional ports for the MSAP architecture

Two additional ports are found on top of the original ports. This indicates that the adjustments provided by the researchers have been successfully implemented in the MSAP architecture, as seen by the Vivado implemented designer.

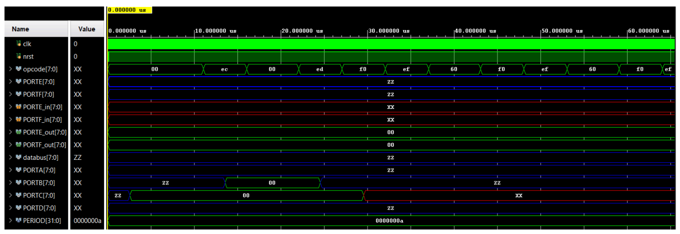


Fig. 17. Simulation waveform of the modified MSAP architecture

To further certify the operation of the additional ports, a simulation was performed to test whether the two execute as instructed by the program. PORTE was utilized as an input port, which will contain the data stored in the accumulator. Meanwhile, PORTF was used as an output port, which displays the passed value by the input port. In the waveform, it can be seen that PORTF has been successfully altered, indicating that the ports are properly added to the original MSAP architecture.

IV. Conclusion and Future Improvements

The Modified Simple as Possible (MSAP) architecture was successfully enhanced by adding two additional ports to the existing instruction set, which improved its ability to interface with external devices, thereby increasing its overall usability. The most significant modifications were made to the control unit, allowing it to accommodate these new ports. The integration was thoroughly tested and validated through simulations using Vivado, which confirmed that the new ports were fully functional and effectively interconnected with other system components. The design changes not only met the project objectives but also enhanced the overall functionality and versatility of the MSAP architecture.

In addition to the MSAP architecture modifications, the project also included enhancements to the RISC architecture, particularly in the implementation of sorting algorithms and arithmetic operations such as multiplication and division. The sorting algorithms—selection sort, bubble sort, and insertion sort—were implemented and tested within the RISC architecture. The timing diagrams and simulations confirmed that the architecture efficiently executed these algorithms, organizing data correctly according to the specified sorting method. This demonstrated the capability of the RISC architecture to handle typical data manipulation tasks effectively. Furthermore, the arithmetic operations involving both integer and fixed-point numbers were successfully integrated into the RISC architecture. The architecture was able to perform multiplication and division operations accurately, even handling edge cases such as division by zero and large integer overflows appropriately. The implementation of these arithmetic operations showcased the architecture’s capability to handle a range of computational tasks, further validating its efficiency and reliability.

Even if the current design meets the objectives declared in this project, there is still a lot of scope for improvement. Addressing existing bugs, particularly timing issues that arise during high-speed data transfer, will be crucial for improving system reliability. Expanding the testing methodology to include more real-world scenarios would provide deeper insights into the architecture's performance under varied conditions. Additionally, enhancing the arithmetic operations by implementing more advanced features like floating-point calculations and optimizing the sorting algorithms for speed and efficiency could significantly improve the architecture's overall performance. Moreover, incorporating advanced features such as dynamic port configuration and support for more sophisticated I/O modules could further enhance the architecture's adaptability, making it suitable for more complex and diverse computing tasks. Applying these concepts to other instruction set architectures, such as Complex Instruction Set Computing (CISC) or MIPS, could demonstrate the broader applicability and potential benefits of this approach, paving the way for future research and development in this area.

By continuing to refine and expand upon the current design, the MSAP and RISC architectures could evolve into robust and flexible computing solutions, well-suited to a wide range of applications.

References

[1] A. Morlan, “Building an FPGA Computer: SAP-2,” Austinmorlan.com, Jan. 15, 2023. https://austinmorlan.com/posts/fpga_computer_sap2/ (accessed Aug. 07, 2024).

[2] Stanford.edu, “RISC vs. CISC,” Stanford.edu, 2024. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ (accessed Aug. 07, 2024).

[3] Digilent, “Nexys A7: FPGA Trainer Board Recommended for ECE Curriculum,” Digilent, 2020. https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/ (accessed Aug. 07, 2024).

[4] Zaidi, S. A., & Azfar, J. (n.d.). Pipelined-RISC-V-Processor. GitHub repository. Retrieved from https://github.com/samiyaalizaidi/Pipelined-RISC-V-Processor

APPENDIX

1. msapMicrocomputer

```
`timescale 1ns / 1ps

module msapMicrocomputer(
   input clkin,
   input nrst,
   inout [7:0] PORTA, PORTB, PORTC, PORTD, PORTE, PORTF
   );

   wire clk, rst;

   wire [7:0] opcode, flag;
   wire clrpc, incpc, ldflag, rd, wr; //incstk, dcrstk, ldstk;
   wire [3:0] funcSel;
   wire [2:0] srcSel;
   wire [3:0] dstSel;
   wire [1:0] dioSel;
   wire [7:0] addrbus;
   wire [7:0] databus_in, databus_out;
   wire [7:0] PORTA_in, PORTB_in, PORTC_in ,PORTD_in, PORTE_in, PORTF_in;
   wire [7:0] PORTA_out, PORTB_out, PORTC_out, PORTD_out, PORTE_out, PORTF_out;
   wire [7:0] PORTA_dir, PORTB_dir, PORTC_dir, PORTD_dir, PORTE_dir, PORTF_dir;

   clk_div ClockDivider(
      .clkin(clkin),
      .nrst(nrst),
      .clkout(clk),
      .rst(rst)
      );

   (* DONT_TOUCH = "true" *) msap_datapath DataPath(
      .clk(clk), .rst(rst),
      .clrpc(clrpc), .incpc(incpc),
      .rd(rd), .wr(wr),
      .funcSel(funcSel),
      .srcSel(srcSel),
      .dstSel(dstSel),
      .dioSel(dioSel),
      .ldflag(ldflag),
      .opcode(opcode),
      .flag_out(flag),
      .addrbus(addrbus),
      .databus_in(databus_in),
      .databus_out(databus_out),
                     .PORTA_in(PORTA_in), .PORTB_in(PORTB_in), .PORTC_in(PORTC_in), .PORTD_in(PORTD_in),
.PORTE_in(PORTE_in), .PORTF_in(PORTF_in),
               .PORTA_out(PORTA_out), .PORTB_out(PORTB_out), .PORTC_out(PORTC_out), .PORTD_out(PORTD_out),
.PORTE_out(PORTE_out), .PORTF_out(PORTF_out),
                 .PORTA_dir(PORTA_dir), .PORTB_dir(PORTB_dir), .PORTC_dir(PORTC_dir), .PORTD_dir(PORTD_dir),
.PORTE_dir(PORTE_dir), .PORTF_dir(PORTF_dir)
      );

   (* DONT_TOUCH = "true" *) msap_ctrl ControlUnit(
      .clk(clk), .rst(rst),
      .opcode(opcode),
      .flag_in(flag),
      .clrpc_out(clrpc), .incpc_out(incpc),
      .rd_out(rd), .wr_out(wr),
      .ldflag_out(ldflag),
   //  .incstk_out(incstk), .dcrstk_out(dcrstk), .ldstk_out(ldstk),
      .funcSel_out(funcSel),
      .srcSel_out(srcSel),
      .dstSel_out(dstSel),
      .dioSel_out(dioSel)
      );

   msap_ROM ROM(
      .clk(clk),
      .CS(addrbus[7]), .OE(!rd),
      .address(addrbus[6:0]),
      .dataout(databus_in)
      );

   msap_RAM RAM(
      .clk(clk),
      .CS(!addrbus[7]), .OE(!rd), .WE(!wr),
      .address(addrbus[6:0]),
      .datain(databus_out), .dataout(databus_in)
      );

   octal_iobuf porta_buf(
      .iobuf_in(PORTA_out),
      .iobuf_out(PORTA_in),
      .iobuf_t(PORTA_dir),
      .iobuf_inout(PORTA)
      );

   octal_iobuf portb_buf(
               .iobuf_in(PORTB_out),
      .iobuf_out(PORTB_in),
      .iobuf_t(PORTB_dir),
      .iobuf_inout(PORTB)
               );

            octal_iobuf portc_buf(
               .iobuf_in(PORTC_out),
      .iobuf_out(PORTC_in),
      .iobuf_t(PORTC_dir),
      .iobuf_inout(PORTC)
               );

            octal_iobuf portd_buf(
               .iobuf_in(PORTD_out),
      .iobuf_out(PORTD_in),
      .iobuf_t(PORTD_dir),
      .iobuf_inout(PORTD)
               );

            octal_iobuf porte_buf(
               .iobuf_in(PORTE_out),
   	.iobuf_out(PORTE_in),
   	.iobuf_t(PORTE_dir),
   	.iobuf_inout(PORTE)
   		);

   		octal_iobuf portf_buf(
   		  .iobuf_in(PORTF_out),
   	.iobuf_out(PORTF_in),
   	.iobuf_t(PORTF_dir),
   	.iobuf_inout(PORTF)
   		);

   endmodule
```

2. Clock Divider

```
`timescale 1ns / 1ps

module clk_div(
    input clkin,
    input nrst,
    output clkout,
    output rst
    );

    reg [7:0] clk_ctr, rst_ctr;
    reg rst_reg, clkout_reg;

    always @(posedge clkin) begin
        if (!nrst) begin
            clk_ctr <= 0;
            clkout_reg <= 0;
            rst_ctr <= 0;
            rst_reg <= 1;
        end else begin
            if (clk_ctr < 49) begin
                clk_ctr <= clk_ctr + 1;
                clkout_reg <= clkout_reg;
            end else begin
                clk_ctr <= 0;
                clkout_reg <= ~clkout_reg;
                if (rst_ctr < 4) begin
                    rst_ctr <= rst_ctr + 1;
                end else begin
                    rst_reg <= 0;
                end
            end
        end
    end

    assign rst = rst_reg;
    assign clkout = clkout_reg;

endmodule
```

3. Datapath

```
`timescale 1ns / 1ps
```

```
module msap_datapath(
   input clk,
   input rst,
   input clrpc, incpc,
   input rd, wr,
   input [3:0] funcSel,
   input [2:0] srcSel,
   input [3:0] dstSel,
   input [1:0] dioSel,
   input ldflag,
   output [7:0] opcode,
   output [7:0] flag_out,
   output [7:0] addrbus,
   input [7:0] databus_in,
   output [7:0] databus_out,
   input [7:0] PORTA_in, PORTB_in, PORTC_in, PORTD_in, PORTE_in, PORTF_in,
   output [7:0] PORTA_out, PORTB_out, PORTC_out, PORTD_out, PORTE_out, PORTF_out,
   output [7:0] PORTA_dir, PORTB_dir, PORTC_dir, PORTD_dir, PORTE_dir, PORTF_dir
   );

   wire [7:0] ibus;
   wire [7:0] pcout, mdrout, aout, bout, aluout, aluflag, portdata;
   wire [7:0] port_a, port_b, port_c, port_d, port_e, port_f;
   wire [7:0] port_a, port_b, port_c, port_d, port_e, port_f;
   wire [18:0] ldout;
   wire ldpc, ldmar, ldmdr, ldir, ldacc, ldb, ldf;
          wire lddioa, lddiob, lddioc, lddiod;
          wire lddira, lddirb, lddirc, lddird;

   // Program Counter
   msap_pc ProgramCounter(
      .clk(clk), .rst(rst),
      .clrpc(clrpc), .incpc(incpc), .ldpc(ldpc),
      .PC_in(ibus), .PC_out(pcout)
      );

   // Memory Address Register
   msap_mar MemoryAddressRegister(
      .clk(clk), .rst(rst),
      .ldmar(ldmar),
      .MAR_in(ibus), .MAR_out(addrbus)
      );

   // Memory Data Register
   msap_mdr MemoryDataRegister(
      .clk(clk), .rst(rst),
      .ldmdr(ldmdr),
      .rd(rd), .wr(wr),
      .MDR_in(ibus), .MDR_out(mdrout),
      .databus_in(databus_in), .databus_out(databus_out)
      );

   // Instruction Register
   msap_ir InstructionRegister(
      .clk(clk), .rst(rst),
      .ldir(ldir),
      .IR_in(ibus), .IR_out(opcode)
      );

   // Accumulator Register
   msap_acc AccumulatorRegister(
      .clk(clk), .rst(rst),
      .ldacc(ldacc),
      .ACC_in(ibus), .ACC_out(aout)
      );

   // B Register
   msap_b BRegister(
      .clk(clk), .rst(rst),
      .ldb(ldb),
      .B_in(ibus), .B_out(bout)
      );

   // Arithmetic Logic Unit
   msap_alu ALU(
      .funcSel(funcSel),
      .A(aout), .B(bout),
      .cin(flag_out[0]),
      .ALU_out(aluout),
      .ALU_flag(aluflag)
      );

   // FLAG Register
   msap_flag FLAG(
      .clk(clk), .rst(rst),
      .ldf(ldflag),
      .FLAG_in(aluflag), .FLAG_out(flag_out)
      );

   // Source Multiplexer
   octal_mux8to1 srcMux(
      .muxSel(srcSel),
      .muxin0(pcout),
      .muxin1(mdrout),
      .muxin2(aout),
      .muxin3(bout),
      .muxin4(aluout),
      .muxin5(),
      .muxin6(),
      .muxin7(portdata),
      .muxout(ibus)
      );

   // Load Control Decoder
   decoder4to16 dstLoadControl(
      .dcdSel(dstSel),
      .dcdOut(ldout)
      );

   assign ldpc = ldout[0];     // Load to program counter
   assign ldmar = ldout[1];    // Load memory address register
   assign ldmdr = ldout[2];    // Load memory data register
   assign ldir = ldout[3];     // Load instruction register
   assign ldacc = ldout[4];    // Load accumulator register
   assign ldb = ldout[5];      // Load to B register
             assign ldf = ldout[6];      //
             assign lddioa = ldout[7];   // Load to port A data
             assign lddiob = ldout[8];   // Load to port B data
             assign lddioc = ldout[9];   // Load to port C data
             assign lddiod = ldout[10];  // Load to port D data
             assign lddioe = ldout[11];  // Load to port E data
             assign lddiof = ldout[12];  // Load to port F data
             assign lddira = ldout[13];  // Load to port A direction
             assign lddirb = ldout[14];  // Load to port B direction
             assign lddirc = ldout[15];  // Load to port C direction
             assign lddird = ldout[16];  // Load to port D direction
             assign lddire = ldout[17];  // Load to port E direction
             assign lddirf = ldout[18];  // Load to port F direction

   // Digital Port A
   msap_port PORTA(
      .clk(clk), .rst(rst),
      .lddir(lddira), .lddio(lddioa),
      .portReg_in(ibus), .portReg_out(port_a),
      .PORTx_in(PORTA_in),
      .PORTx_out(PORTA_out),
      .PORTx_dir(PORTA_dir)
      );

   // Digital Port B
   msap_port PORTB(
      .clk(clk), .rst(rst),
      .lddir(lddirb), .lddio(lddiob),
      .portReg_in(ibus), .portReg_out(port_b),
      .PORTx_in(PORTB_in),
      .PORTx_out(PORTB_out),
      .PORTx_dir(PORTB_dir)
      );

   // Digital Port C
   msap_port PORTC(
      .clk(clk), .rst(rst),
      .lddir(lddirc), .lddio(lddioc),
      .portReg_in(ibus), .portReg_out(port_c),
      .PORTx_in(PORTC_in),
      .PORTx_out(PORTC_out),
      .PORTx_dir(PORTC_dir)
      );

   // Digital Port D
   msap_port PORTD(
      .clk(clk), .rst(rst),
      .lddir(lddird), .lddio(lddiod),
      .portReg_in(ibus), .portReg_out(port_d),
      .PORTx_in(PORTD_in),
      .PORTx_out(PORTD_out),
      .PORTx_dir(PORTD_dir)
      );

   // Digital Port E
```

```
    msap_port PORTE(
        .clk(clk), .rst(rst),
        .lddir(lddire), .lddio(lddioe),
        .portReg_in(ibus), .portReg_out(port_e),
        .PORTx_in(PORTE_in),
        .PORTx_out(PORTE_out),
        .PORTx_dir(PORTE_dir)
        );

     // Digital Port F
    msap_port PORTF(
        .clk(clk), .rst(rst),
        .lddir(lddirf), .lddio(lddiof),
        .portReg_in(ibus), .portReg_out(port_f),
        .PORTx_in(PORTF_in),
        .PORTx_out(PORTF_out),
        .PORTx_dir(PORTF_dir)
        );

    // Port Data Source Multiplexer
    octal_mux8to1 portSelect(
        .muxSel(dioSel),
        .muxin0(port_a),
        .muxin1(port_b),
        .muxin2(port_c),
        .muxin3(port_d),
        .muxin4(port_e),
        .muxin5(port_f),
        .muxin6(),
        .muxin7(),
        .muxout(portdata)
        );

endmodule
```

FOR SORTING ALGORITHM:
Assembly code for:

Selection Sorting:

```
# Store array values in contiguous memory at mem address 0x0:
# {49, 43, 32, 22, 4}
 addi a0, x0, 0

 addi t0, x0, 49
 sd t0, 0(a0)
 addi t0, x0, 43
 sd t0, 8(a0)
 addi t0, x0, 32
 sd t0, 16(a0)
 addi t0, x0, 22
 sd t0, 24(a0)
 addi t0, x0, 4
 sd t0, 32(a0)
```

```
addi a1, x0, 5   # size of 'arr'

beq x0, x0, SEL_SORT
beq x0, x0, EXIT

SEL_SORT:
addi t0, x0, 0 # i
addi t1, x0, 0 # j
addi t2, x0, 0 # min_index

addi s0, a1, 0 # store n.
addi a1, a1, -1 # a1 is the size of array. n - 1

UNSORTED_ARRAY_BOUNDARY_LOOP:
beq t0, a1, END_UNSORTED_ARRAY_BOUNDARY_LOOP # (while i < (n-1))

addi t2, t0, 0   # min_index = i
addi t1, t0, 1   # j = i + 1

SUBARRAY_LOOP:
beq t1, s0, END_SUBARRAY_LOOP # (while j < n)

slli t5, t1, 3  # j * sizeof(int)
add t6, a0, t5
ld t4, 0(t6)           # Load arr[j]

slli t5, t2, 3  # min_index * sizeof(int)
add t6, a0, t5  # arr[min_index]
ld t3, 0(t6)           # Load arr[min_index]

blt t3, t4, MIN_REMAINS_SAME # if (arr[min_index] < arr[j]), don't change the min.
addi t2, t1,0   # min_index = j
MIN_REMAINS_SAME:

addi t1, t1, 1  # j = j + 1
beq x0, x0, SUBARRAY_LOOP
END_SUBARRAY_LOOP:

slli t5, t2, 3          # min_index * sizeof(int)
add t6, a0, t5         # arr[min_index]
ld t3, 0(t6)           # Load arr[min_index]

slli t1, t0, 3   # i * sizeof(int)
add t1, t1, a0        # arr[i]
ld t4, 0(t1)           # Load arr[i]

sd t3, 0(t1)
sd t4, 0(t6)           # swap(&arr[min_index], &arr[i])

addi t0, t0, 1   # i = i + 1
beq x0, x0, UNSORTED_ARRAY_BOUNDARY_LOOP

END_UNSORTED_ARRAY_BOUNDARY_LOOP:
beq x0, x0, EXIT

EXIT:
```

Bubble Sorting:

```
# {50, 40, 30, 20, 10}
 addi a0, x0, 0

 addi t0, x0, 50
 sd t0, 0(a0)
 addi t0, x0, 40
 sd t0, 8(a0)
 addi t0, x0, 30
 sd t0, 16(a0)
 addi t0, x0, 20
 sd t0, 24(a0)
 addi t0, x0, 10
 sd t0, 32(a0)

addi a1, x0, 5   # size of 'arr'

beq x0, x0, BUBBLE_SORT
beq x0, x0, EXIT

BUBBLE_SORT:
addi t0, x0, 0 # i
addi t1, x0, 0 # j

addi s0, a1, 0 # store n.
addi a1, a1, -1 # a1 is the size of array. n - 1

OUTER_LOOP:
beq t0, a1, END_OUTER_LOOP # (while i < (n-1))

INNER_LOOP:
beq t1, a1, END_INNER_LOOP # (while j < (n-i-1))

slli t5, t1, 3  # j * sizeof(int)
add t6, a0, t5
ld t3, 0(t6)          # Load arr[j]

slli t5, t1, 3
add t5, t5, 8   # (j+1) * sizeof(int)
add t6, a0, t5
ld t4, 0(t6)          # Load arr[j+1]

blt t3, t4, NO_SWAP # if (arr[j] < arr[j+1]), no need to swap
sd t3, 0(t6)          # swap arr[j] and arr[j+1]
sd t4, 0(t1)

NO_SWAP:
addi t1, t1, 1  # j = j + 1
beq x0, x0, INNER_LOOP

END_INNER_LOOP:
addi t0, t0, 1   # i = i + 1
beq x0, x0, OUTER_LOOP

END_OUTER_LOOP:
beq x0, x0, EXIT
```

```
EXIT:


Insertion Sorting:
# {50, 40, 30, 20, 10}
 addi a0, x0, 0

 addi t0, x0, 50
 sd t0, 0(a0)
 addi t0, x0, 40
 sd t0, 8(a0)
 addi t0, x0, 30
 sd t0, 16(a0)
 addi t0, x0, 20
 sd t0, 24(a0)
 addi t0, x0, 10
 sd t0, 32(a0)

addi a1, x0, 5   # size of 'arr'

beq x0, x0, INSERTION_SORT
beq x0, x0, EXIT

INSERTION_SORT:
addi t0, x0, 1 # i (start from the second element)
addi t1, x0, 0 # j

OUTER_LOOP:
beq t0, a1, END_OUTER_LOOP # (while i < size)

slli t5, t0, 3  # i * sizeof(int)
add t6, a0, t5
ld t3, 0(t6)           # key = arr[i]

addi t1, t0, -1 # j = i - 1

INNER_LOOP:
slli t5, t1, 3
add t6, a0, t5
ld t4, 0(t6)           # Load arr[j]

blt t4, t3, NO_SHIFT # if arr[j] <= key, break the loop

slli t5, t1, 3
add t5, t5, 8   # (j+1) * sizeof(int)
sd t4, 0(t6)           # arr[j+1] = arr[j]

addi t1, t1, -1 # j = j - 1
bgez t1, INNER_LOOP # while j >= 0, continue loop

NO_SHIFT:
slli t5, t1, 3
add t6, a0, t5
add t6, t6, 8   # (j+1) * sizeof(int)
sd t3, 0(t6)           # arr[j+1] = key

addi t0, t0, 1   # i = i + 1
```

```
beq x0, x0, OUTER_LOOP

END_OUTER_LOOP:
beq x0, x0, EXIT

EXIT:
```

ALU_64_Bit.v

```
// ALU Module
`timescale 1ns / 1ps
module ALU_64_bit (
		input [63:0] a, b,
		input [3:0] ALUOp,

		output reg [63:0] Result,
		output reg ZERO
);

wire BLT = 4'b1000;
reg [127:0] mul_temp; // to hold the intermediate multiplication result

always @ (*)
begin
		case (ALUOp)
		4'b0000: Result = a & b; // AND
		4'b0001: Result = a | b; // OR
		4'b0010: Result = a + b; // ADD
		4'b0110: Result = a - b; // SUB
		4'b1100: Result = ~(a | b); // NOR
		4'b1000: Result = a < b; // BLT
		4'b1001: Result =  a + b; // ADDI
		4'b1111: Result = a * (2'd2**b); // SLLI
		4'b1010: Result = a * b; // Normal integer multiplication
		4'b1101: // Fixed-point MUL operation
		begin
		mul_temp = a * b; // multiply as 128-bit intermediate value
		Result = mul_temp >> 32; // scale down by 2^32 to get correct fixed-point result
		end
		4'b1011: // DIV operation
		Result = b != 0 ? a / b : 64'hX; // handle division, with safe divide by zero check
		//default: Result = 0;
		endcase

		if (ALUOp == BLT && Result)
		ZERO = 1;
		else
		ZERO = (Result == 0);
end

endmodule
```

test_ALU.v

```
// Testbench Module
module test_ALU;
  reg [63:0] a, b;
  reg [3:0] ALUOp;
  wire [63:0] Result;
  wire ZERO;

  // Instantiate the ALU module
  ALU_64_bit uut (
          .a(a),
          .b(b),
          .ALUOp(ALUOp),
          .Result(Result),
          .ZERO(ZERO)
  );

  initial begin
          // Test multiplication with small integers
          a = 64'd2;
          b = 64'd3;
          ALUOp = 4'b1010; // MUL operation
          #10;
          $display("Multiplication Small Integers: %d * %d = %d", a, b, Result);

          // Test multiplication with large integers
          a = 64'hFFFFFFFFFFFFFFFF; // max 64-bit value
          b = 64'd2;
          ALUOp = 4'b1010; // MUL operation
          #10;
          $display("Multiplication Large Integers: %d * %d = %d", a, b, Result);

          // Test division
          a = 64'd10;
          b = 64'd5;
          ALUOp = 4'b1011; // DIV operation (assuming 1011 is assigned for division)
          #10;
          $display("Division: %d / %d = %d", a, b, Result);

          // Test division by zero
          a = 64'd420;
          b = 64'd0;
          ALUOp = 4'b1011; // DIV operation
          #10;
          $display("Division by Zero: %d / %d = %h", a, b, Result);

          // Test multiplication with fractional values
          // 2.5 * 3.5 in fixed-point representation
          a = 64'h0000000280000000; // 2.5 in fixed-point
          b = 64'h0000000380000000; // 3.5 in fixed-point
          ALUOp = 4'b1101; // Fixed point MUL operation
          #10;
          $display("Multiplication Fixed-Point: %h * %h = %h", a, b, Result); // Expecting 8.75 in fixed-point format

          // Test the smallest multiplication (1 * 1)
          a = 64'd1;
          b = 64'd1;
          ALUOp = 4'b1010; // MUL operation
          #10;
          $display("Multiplication Smallest Values: %d * %d = %d", a, b, Result);

          // Test the largest multiplication (max value * max value)
          a = 64'hFFFFFFFFFFFFFFFF; // max 64-bit value
```

```
            b = 64'hFFFFFFFFFFFFFFFF; // max 64-bit value
            ALUOp = 4'b1010; // MUL operation
            #10;
            $display("Multiplication Largest Values: %h * %h = %h", a, b, Result);

            // Test the smallest division (1 / 1)
            a = 64'd1;
            b = 64'd1;
            ALUOp = 4'b1011; // DIV operation
            #10;
            $display("Division Smallest Values: %d / %d = %d", a, b, Result);

            // Test the largest division (max value / 1)
            a = 64'hFFFFFFFFFFFFFFFF; // max 64-bit value
            b = 64'd2;
            ALUOp = 4'b1011; // DIV operation
            #10;
            $display("Division Largest Values: %h / %d = %d", a, b, Result);
            // Terminate simulation
            $finish;
  end
endmodule
```