

Princess Sumaya University for Technology
King Abdullah II Faculty of Engineering
Electrical Engineering Department



Princess Sumaya جامعة
University الأميرة سميرة
for Technology للتكنولوجيا

COMPUTER ARCHITECTURE 2 / 22321
FLOATING POINT ARITHMETIC PIPELINE SYSTEM
FINAL REPORT

Authors:

Taima Darwish
Mariam Murad

20200415
20210798

Comp. Eng.
Comp. Eng.

Supervisor:

Dr. Awos Kanan

16/05/2023

Abstract

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as pipeline processing. Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Also pipelining increases, the overall instruction throughput.

TABLE OF CONTENTS

1	Introduction	1
1.1	Objectives	1
1.2	Theory	1
1.2.1	IEEE 754 floating-point standard.....	1
1.2.2	Pipelined processor.....	1
2	Algorithm.....	2
3	Proposed Design.....	3
3.1	Instruction Fetch stage	3
3.2	Instruction Decode stage.....	3
3.3	Execution one Stage	4
3.4	Execution two Stage	4
3.5	Data Memory stage	4
3.6	Write BACK STAGE	5
4	Design Analysis	5
4.1	EXE stage.....	5
4.2	Register File	5
4.3	Instruction Memory	5
4.4	Forward unit	5
5	Stages Results	6
5.1	Stage 1 (Instruction Fetch).....	6
5.2	Stage 2 (Instruction Decode).....	7
5.3	Stage 3 (Execution).....	8
5.4	Stage 4 (Execution 2).....	9
5.5	Stage 5.....	11
5.6	Stage 6.....	11
6	PERFORMANCE Analysis	12
7	Diagram.....	15
8	CONCLUSION	13
9	References.....	14

1 INTRODUCTION

1.1 OBJECTIVES

The main objective of this project is to design a simple arithmetic floating-point processor that works on a set of instructions, the design will include the pipeline stages, in addition to the code that runs this pipeline. [6]

In this project we will focus on a specific case such as;

- All numbers are positive.
- No load use hazard.
- Forward will be from the end of the second stage of ALU to the first stage.

So, the processor should receive an instruction, decode it, execute it properly and saves the data calculated in its proper register.

1.2 THEORY

1.2.1 IEEE 754 floating-point standard

The Institute of Electrical and Electronics Engineers created the IEEE Standard for Floating-Point Arithmetic (IEEE 754) in 1985 as a technical standard for floating-point calculation (IEEE). The standard addressed several issues encountered in various floating point representations that made them difficult to use effectively and limited their applicability. IEEE Standard 754 floating point is the most widely used representation of real numbers on computers today, including Intel-based PCs, Macs, and most Operating systems. The IEEE single format consists of three sections: the highest-order bit 31 contains the sign bit, bits 23:30 contain the 8-bit biased exponent and a 23-bit fraction. These fields are stored in one 32-bit word, as shown in Figure 11.2.1 [1]. [2]

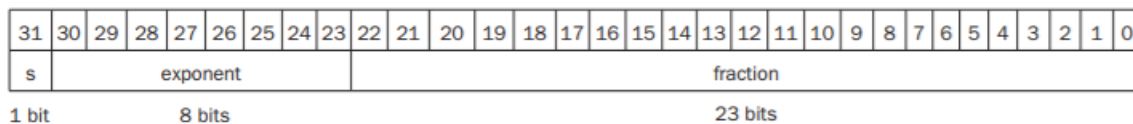


Figure 1: IEEE single precision format

1.2.2 Pipelined processor

A method for running several instructions is called pipelining. The benefit of this strategy is that it allows for a higher throughput. Pipelining usually divides the execution process into phases. Depending on how the project is carried out, different stages may be required. In our situation, the pipelined structure consists of six stages. The instructions can be executed simultaneously, one instruction before the previous one is finished, because they are divided into six sections that can run simultaneously. [3]

The six operations included in the single precision floating-point arithmetic pipelined system are addition, subtraction, multiplication, division, load and store.

1.2.2.1 Pipelining hazards

- Load-use hazard is already solved using reordering instruction. [8]
- We will be dealing with ALU-ALU forwarding. [8]

1.2.2.2 Implementation of the pipelined floating-point processor

32-bit processor architecture is implemented. Six stages of pipelined architecture are as follows: [6]

1. Instruction Fetch (IF):
Fetch the instructions from the memory.
2. Instruction Decode (ID):
Decode the instruction that was fetched from the memory and register read.
3. 2 stages (Three and four)-Instruction Execute (EXE):
The arithmetic operation is performed on the operands to execute the instruction.
4. Memory:
To avoid structural hazard, we have a memory for the data separated from the instruction memory.
The data memory is RAM 32x8 (32 bits since we are dealing with single floating point).
5. Write Back:
Store the final result back to the registers.

2 ALGORITHM

These steps will be followed in order to ensure a successful execution of the operations: [6]

- Checking the opcode in the control unit and pass it to forward unit and comparator unit to prepare mux signals.
- Based on the opcode, the correct operation will take place.
- The source registers will go through the comparator unit, after comparing their values the result will be in the control signal of mux 3.
- Below is the description of the op code operation.

OP Code	Operation executed
000	Addition
001	Subtraction
010	Multiplication
011	Division
100	Load
101	Store

Table 1: OP Code [6]

3 PROPOSED DESIGN

3.1 INSTRUCTION FETCH STAGE

A module with the name stage_1 was made to make the instruction fetch this module consists of instances of 2 other modules:

1. PCADDER Module

This module will take the PC as an input and add to it one, which is the difference between each instruction and the one after it in the Instruction Memory. And then output the PC+1.

2. Instruction Memory Module

A 32 X 8 ROM memory was created in this module it stores pre-defined values and provides the corresponding data based on the address input.

Both modules will save their output on the negative edge of the clock. [7]

After the 2 modules take place, the results are saved into IF/ID buffer so the values could be used in the next stage.

3.2 INSTRUCTION DECODE STAGE

In this stage, instruction is decoded, putting the bits of the instruction received into their proper place, then passing it to the following stage and the register file is accessed to get the values from the registers used in the instruction. [3]

A module with the name Stage2_TopModule was made to make the instruction fetch this module consists of instances of 3 other modules:

1. InstructionSeperation Module

In this module it will take the instruction fetched in stage 1 as an input and then separate the bits depending on the formatting of the instruction as shown below

RS2	RS1	RD	OP-CODE
11-9	8-6	5-3	2-0

Figure 2: Instruction Format

2. Control unit

It takes the opcode of the instruction as input and generates control signals for various operations such as register write, register read, memory write, memory read, etc. It uses a case statement to determine the control signals based on the opcode.

3. Register file Module

It has inputs such as source registers (Rs1, Rs2), destination register (Rd), and write data (Rd_data). It also has an input for the write enable signal (WriteEnable). It stores the data in registers based on the write enable signal and provides the data from the source registers as outputs.

3.3 EXECUTION ONE STAGE

The Ex1 module uses combinational logic to determine the appropriate operation based on the value of Ex1Signal and performs the necessary calculations using the input numbers. It also handles the shifting of fractional values and determines the exponent value for the output. The module assigns the calculated results to the appropriate output signals.

This stage is responsible for the exponent and sign of the final output. [7]

3.4 EXECUTION TWO STAGE

Ex2_stage module uses several registers and variables to perform the arithmetic calculations. It extracts the fractional parts of input1 and input2 and assigns them to frac1 and frac2, respectively. [7]

On each negative edge of the CS clock signal, the module performs the selected arithmetic operation based on the value of Ex2:

- If Ex2 is 10000, it performs addition by adding frac1 and frac2 and assigns the result to ansFrac.
- If Ex2 is 10001 it performs subtraction by subtracting the smaller fraction from the larger fraction and assigns the result to ansFrac.
- If Ex2 is 01010, it performs multiplication by multiplying frac1 and frac2 and assigns the result to ansFrac.
- If Ex2 is 01011, it performs division by dividing frac1 by frac2 and assigns the result to ansFrac.

3.5 DATA MEMORY STAGE

- This stage contains multiple modules such as (Mem_WB_buffer, RAM, testbench, EXE_Mem_buffer, b1_stage5_b2, MUX_3.
- RAM: The RAM module represents a memory component. It initializes an array (memory) with specific values and supports read and write operations. It has input signals (addr, write_data, memread, memwrite, CS) and an output signal (read_data). The module responds to control signals (memread and memwrite) to read or write data from/to the specified memory address.

- This multiplexer takes two input results and selects one of them based on a control signal coming from the comparator.

3.6 WRITE BACK STAGE

This stage is mostly needed for R-type and load instructions, where the executed result will be written back to its destination register. [6]

4 DESIGN ANALYSIS

4.1 EXE STAGE

The stage's inputs are: the two decoded numbers, the execution one signal indicates which operation is performed. Then, the signs, and exponentials will be extracted from the operands. The exponent of one of the numbers will then be subtracted from the exponent of the greater number, and the fraction portion will be shifted by the amount of the difference to align the numbers.

Following that, the selected operation will take place based on the received signal.

Finally, each component will be placed in its designated location. [7]

4.2 REGISTER FILE

The register file contains 8 registers, 32-bit each, in which the values have are initialized and ready to be tested. it will also read the values that are held in rs1, rs2 and rd.

The register's file cue to write the data in rd would be when the negative edge is triggered and the enable write register is active (write back stage). So, the module would not just change the values of the register on each negative edge of a cycle but also it should wait for an enable to change the values in the reg.

4.3 INSTRUCTION MEMORY

Instruction memory contains a declared memory array that stores the instructions, and because each instruction is 12-bits long, the memory is $2^{12} = 4096$ -bits long. The instruction memory reads the instruction address that the PC is pointing to and sends it to its intended destination.

4.4 FORWARD UNIT

This stage is mostly needed for R-type and load instructions, where the executed result will be written back to its destination register. Rather than waiting for the WB stage to write the register file, the goal is to pass accurate readings from the pipeline registers to the ALU as quickly and efficiently as possible. The designed forwarding unit allows ALU-ALU forwarding where the forwarding unit checks if the destination register in the memory stage is also used as either two source registers in the execution stage, then unit chooses which operand to forward. [4]

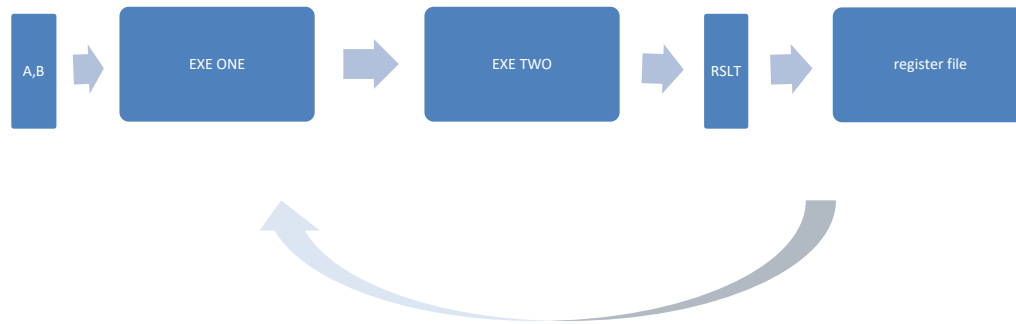


Figure 3: Forwarding Path [3]

5 STAGES RESULTS

In the following section we will run each module and presents the results.

5.1 STAGE 1 (INSTRUCTION FETCH)

As mentioned previously, the instruction fetch stage get the Program counter (PC) as input and add by 1 then fetch the instruction with the PC added by 1 that will be found in the instruction memory. [7]

```

5 );
6
7 reg [31:0] memory [0:7];
8 initial begin
9   // Initialize the ROM contents
10  memory[0] = 32'h00000000;
11  memory[1] = 32'h00000001;
12  memory[2] = 32'h00000010;
13  memory[3] = 32'h00000011;
14  memory[4] = 32'h00000100;
15  memory[5] = 32'h00000101;
16  memory[6] = 32'h00000110;
17  memory[7] = 32'h00000111;
18 end
19
20 always @(negedge CS) begin
21   case (address)
22     3'b000: data <= memory[0];
23     3'b001: data <= memory[1];
24     3'b010: data <= memory[2];
25     3'b011: data <= memory[3];
26     3'b100: data <= memory[4];
27     3'b101: data <= memory[5];
28     3'b110: data <= memory[6];
29     3'b111: data <= memory[7];
30     default: data <= 32'h00000000; // Default value if address is invalid
31   endcase
32 end
  
```

Figure 4: Results of Stage 1

PC result

```

pc in= 001
pc out= 010

pc in= 010
pc out= 011

pc in= 011
pc out= 100

pc in= 100
pc out= 101

pc in= 101
pc out= 110

pc in= 110
pc out= 111

pc in= 111
pc out= 000
  
```

Figure 5: PC result

5.2 STAGE 2 (INSTRUCTION DECODE)

Depending on the instruction that was fetched in stage 1, stage 2 will decode it using the instruction separation, register file module and control unit.

```
instruction = 00000000111000111000111000111000
Rs_data =      7 Rs2_data =      0 Rd_out = 7
opcodeOut = 000 RegW = 1 RegR = 1 WB = 1 MemR = 0
MemW = 0 Ex1 = 10000 Ex2 = 00000
extended binary = 0000000000000000000000000000111

instruction = 000000000000000000000000110010010010
Rs_data =      2 Rs2_data =      2 Rd_out = 6
opcodeOut = 010 RegW = 1 RegR = 1 WB = 1 MemR = 0
MemW = 0 Ex1 = 00010 Ex2 = 01010
extended binary = 0000000000000000000000000000010

instruction = 000000000000000000000000111111111100
Rs_data =      7 Rs2_data =      7 Rd_out = 7
opcodeOut = 100 RegW = 1 RegR = 1 WB = 1 MemR = 1
MemW = 0 Ex1 = 10100 Ex2 = 00100
extended binary = 00000000000000000000000000000111
```

Figure 6: Results of Stage 2

Stage 2 and 3 buffer result

```
167 //////////////////////////////////////////////////
168 module ID_Ex1_buffer_tst();
169 reg [31:0]rs1;
170 reg [31:0]rs2;
171 reg [2:0]rd;
172 reg [2:0]Ex1;
173 reg [2:0]Ex2;
174 reg RegR;
175 reg RegW;
176 reg MemW;
177 reg MemR;
178 reg WB;
179 reg CS;
180 wire [31:0]rs1out;
181 wire [31:0]rs2out;
182 wire [2:0]rdout;
183 wire [2:0]Ex1out;
184 wire [2:0]Ex2out;
185 wire RegRout;
186 wire RegWout;
187 wire MemWout;
188 wire MemRout;
189 wire WBout;
190

rs1 = 000000000000000000000000010111001
rs1out = 000000000000000000000000010111001
RegW = 1 RegWout = 1
RegR = 0 RegRout = 0
WB = 1 WBout = 1
MemR = 0 MemRout = 0
MemW = 1 MemWout = 1
Ex1 = 110 Ex1out = 110
Ex2 = 010 Ex2out = 010
```

Figure 7: stage 2 and 3 buffer

Instruction separation Result

[illegible]

Figure 8: instruction separation result

Control unit result

```

29 opcodeIn = 010;
30 #5 // multiplication
31 $display("opcodeOut = %b ", opcodeOut, "RegW = %b ", RegW, "RegR = %b "
32 , RegR, "WB = %b ", WB, "MemR = %b ", MemR, "MemW = %b ", MemW, "Ex1 = %b ", Ex1, "Ex2 = %b
33 \n", Ex2);
34
35 opcodeIn = 011;
36 #5 // division
37 $display("opcodeOut = %b ", opcodeOut, "RegW = %b ", RegW, "RegR = %b "
38 , RegR, "WB = %b ", WB, "MemR = %b ", MemR, "MemW = %b ", MemW, "Ex1 = %b ", Ex1, "Ex2 = %b
39 \n", Ex2);
40
41 opcodeIn = 100;
42 #5 // load
43 $display("opcodeOut = %b ", opcodeOut, "RegW = %b ", RegW, "RegR = %b "
44 , RegR, "WB = %b ", WB, "MemR = %b ", MemR, "MemW = %b ", MemW, "Ex1 = %b ", Ex1, "Ex2 = %b
45 \n", Ex2);
46
47 opcodeIn = 101;
48 #5 // store
49 $display("opcodeOut = %b ", opcodeOut, "RegW = %b ", RegW, "RegR = %b "
50 , RegR, "WB = %b ", WB, "MemR = %b ", MemR, "MemW = %b ", MemW, "Ex1 = %b ", Ex1, "Ex2 = %b
51 \n", Ex2);
52

```

Log Share

```

opcodeOut = 010 RegW = 1 RegR = 1 WB = 1 MemR = 0 MemW = 0 Ex1 = 00010 Ex2 = 01010

opcodeOut = 011 RegW = 1 RegR = 1 WB = 1 MemR = 0 MemW = 0 Ex1 = 01011 Ex2 = 01011

opcodeOut = 100 RegW = 1 RegR = 1 WB = 1 MemR = 1 MemW = 0 Ex1 = 10100 Ex2 = 00100

opcodeOut = 101 RegW = 0 RegR = 1 WB = 0 MemR = 0 MemW = 1 Ex1 = 10101 Ex2 = 00101

```

Figure 9: control unit result

5.3 STAGE 3 (EXECUTION ONE)

As mentioned previously, depending on the `ExlSignal` this stage compares the exponents and shift the smallest fraction to the left if it was an addition or subtraction operation, adds the exponent if its multiplication and subtract the exponent for division.

```
input1 = 0000101010101111011110000010001
input2 = 00001111101110111011000100000001
output1 = 0000111111100000100010000000000
output2 = 00001111101110111011000100000001
Ex1Signal = 10001
result = 0000111110000000000000000000000

input1 = 0000000000100000000000000000010
input2 = 000000000001100000000000000000011
output1 = 00001111101000000000000000000010
output2 = 000011111011000000000000000000011
Ex1Signal = 01001
result = 000011111000000000000000000000000
```

Figure 10: Results of stage 3

Buffer 2, stage three and buffer 3 result

```

247 module b2_stage3_b3_tb();
248 reg [31:0]rs1;
249 reg [31:0]rs2;
250 reg [2:0]rd;
251 reg [4:0]Ex1;
252 reg [4:0]Ex2;
253 reg MemW;
254 reg MemR;
255 reg [31:0]imm;
256 reg mux1Signal;
257 reg [1:0]mux2Signal;
258 reg WB;
259 reg CS;
260 reg [31:0]resultFW;
261 wire [31:0]rs1out;
262 wire [31:0]rs2out;
263 wire [2:0]rdout;
264 wire [31:0]resultout;
265 wire MemWout;
266 wire MemRout;
267 wire WBout;

```

Figure 11: buffer 2, stage 3 and buffer 3 result

EX1/EX2 buffer result

```
rs1 = 0000004c rs1out = 0000004c
rs2 = 0000001b rs2out = 0000001b
WB = 1 WBout = 1
MemR = 0 MemRout = 0
MemW = 1 MemWout = 1
resultin = 00000008
Result = 00000008

rs1no = 010 rs1noOut = 010
rs2no = 101 rs2noOut = 101
```

Figure 12: EX1/EX2 buffer

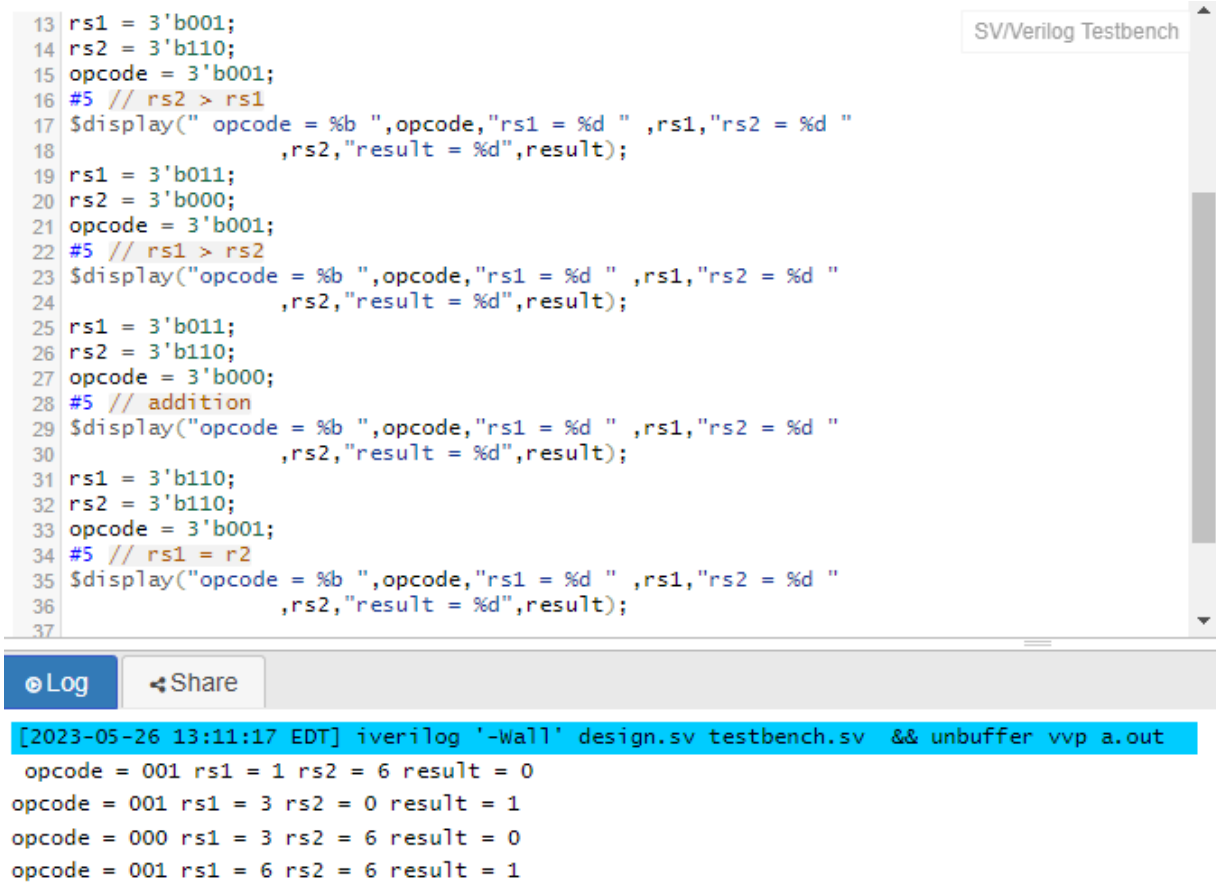
5.4 STAGE 4

This stage shows exactly how we deal with fractions as shown bellow

```
input1 = 00000000000000000000000000000001  
input2 = 000000000000000000000000000000110  
Ex2 = 10000  
resultOut = 010101011000000000000000000000111input1 =          7  
input2 =           5  
Ex2 = 10001  
resultOut = 1434451970
```

Figure 13: Results of stage 4

Comparator result



The screenshot shows a Verilog testbench for a comparator. The testbench sets up four test cases for the comparator's operation based on the opcode and the values of registers rs1 and rs2. The output shows the results of these comparisons.

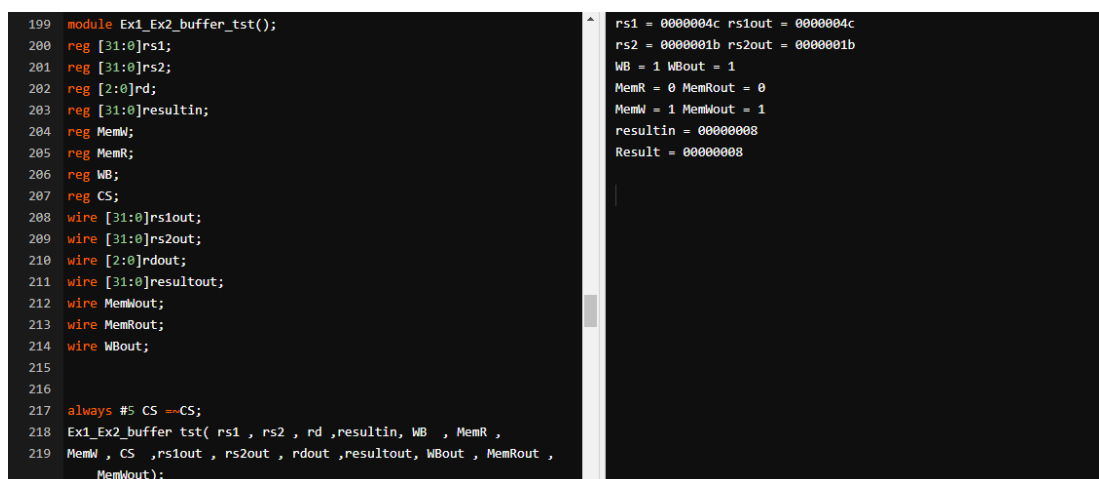
```

13 rs1 = 3'b001;
14 rs2 = 3'b110;
15 opcode = 3'b001;
16 #5 // rs2 > rs1
17 $display(" opcode = %b ",opcode,"rs1 = %d " ,rs1,"rs2 = %d "
18         ,rs2,"result = %d",result);
19 rs1 = 3'b011;
20 rs2 = 3'b000;
21 opcode = 3'b001;
22 #5 // rs1 > rs2
23 $display("opcode = %b ",opcode,"rs1 = %d " ,rs1,"rs2 = %d "
24         ,rs2,"result = %d",result);
25 rs1 = 3'b011;
26 rs2 = 3'b110;
27 opcode = 3'b000;
28 #5 // addition
29 $display("opcode = %b ",opcode,"rs1 = %d " ,rs1,"rs2 = %d "
30         ,rs2,"result = %d",result);
31 rs1 = 3'b110;
32 rs2 = 3'b110;
33 opcode = 3'b001;
34 #5 // rs1 = rs2
35 $display("opcode = %b ",opcode,"rs1 = %d " ,rs1,"rs2 = %d "
36         ,rs2,"result = %d",result);
37
[2023-05-26 13:11:17 EDT] iverilog '-Wall' design.sv testbench.sv && unbuffer vvp a.out
opcode = 001 rs1 = 1 rs2 = 6 result = 0
opcode = 001 rs1 = 3 rs2 = 0 result = 1
opcode = 000 rs1 = 3 rs2 = 6 result = 0
opcode = 001 rs1 = 6 rs2 = 6 result = 1

```

Figure 14: comparator result

Stage 3 and 4 buffer



The screenshot shows a Verilog testbench for a stage 3 and 4 buffer. The testbench sets up various registers and wires to simulate the buffer's operation. The output shows the values of these registers and wires after the simulation.

```

199 module Ex1_Ex2_buffer_tst();
200 reg [31:0]rs1;
201 reg [31:0]rs2;
202 reg [2:0]rd;
203 reg [31:0]resultin;
204 reg MemW;
205 reg MemR;
206 reg WB;
207 reg CS;
208 wire [31:0]rs1out;
209 wire [31:0]rs2out;
210 wire [2:0]rdout;
211 wire [31:0]resultout;
212 wire MemWout;
213 wire MemRout;
214 wire WBout;
215
216
217 always #5 CS <= CS;
218 Ex1_Ex2_buffer tst( rs1 , rs2 , rd ,resultin, WB , MemR ,
219 MemW , CS ,rs1out , rs2out , rdout ,resultout, WBout , MemRout ,
220 MemWout);

```

```

rs1 = 0000004c rs1out = 0000004c
rs2 = 0000001b rs2out = 0000001b
WB = 1 WBout = 1
MemR = 0 MemRout = 0
MemW = 1 MemWout = 1
resultin = 00000008
Result = 00000008

```

Figure 15: stage 3 and 4 buffer

5.5 STAGE 5.

As shown below the saved data in register four was 00000000 and after operating store instruction the "Data Out" was updated to the new value (00001010)

```
address = 4 read = 1 write = 0 Data Out = 00000000 Data In = 0

ResultEx1 = 00001110
ResultEx2 = 00000101
result_out = 00001010
rs2_in = 4
rd_in = 001 rdout = 001
MemW = 0 WB = 1 MemR = 1
WBout = 1

address = 4 read = 1 write = 0 Data Out = 00001010 Data In = 00000000
```

Figure 16: Results of stage 5

5.6 STAGE 6.

Write back result

```
[2023-05-26 13:15:36 EDT] iverilog '-Wall' design.sv testbench.sv && unbuffer vvp a.out
read values of rs1 and rs2 and pass rd number
Rs = 1 Rs_data = 1
Rs2 = 7 Rs2_data = 7
Rd = 7
Rd_out = 7

write the value (3) in rd (rd = 7)
Rd = 7
Rd_data = 3

to make sure the value of x7 was updated
Rs = 7 Rs_data = 3
Rs2 = 1 Rs2_data = 1
Rd = 2
Rd_out = 2
```

Figure 17: write back result

Write back/memory buffer

[illegible]

Figure 18: WB/MEM buffer

Stages result

[illegible]

Figure 19: stages result

6 PERFORMANCE ANALYSIS

In the pipelining implementation, each cycle needs to take as long as the longest cycle unlike the single cycle implementations where each instruction needs a full cycle. [6]

In this implementation we have 6 stages and 6 instructions so;

Speed Up Ratio: Pipelined to Non-pipelined

- A pipeline of K stages
- Number of instructions: n
- Total time required to execute each task: T_n
- Execution time of each pipeline segment: T_p
- Execution time: (number of tasks) x (time/task)
- Non-pipelined execution: $n * T_n$
- Pipelined execution= Execution time of first task ($k * T_p$) + Execution time of remaining tasks $(n-1) * T_p$
- $= (k + n-1) * T_p$
- $$\text{SpeedUp} = \frac{\text{Non-pipelined Execution}}{\text{pipelined Execution}}$$

$$T_n = 6T_p$$

$$\text{Non-pipelined execution} = 6 * T_n = 36 * T_p$$
$$\text{Pipelined execution} = (6 + (6 - 1)) * T_p = 11 * T_p$$

$$\text{SpeedUp} = \frac{16 * T_{pns}}{7 * T_{pns}} = \frac{36}{11} = 3.27273 \text{ [6]}$$

7 DIAGRAM

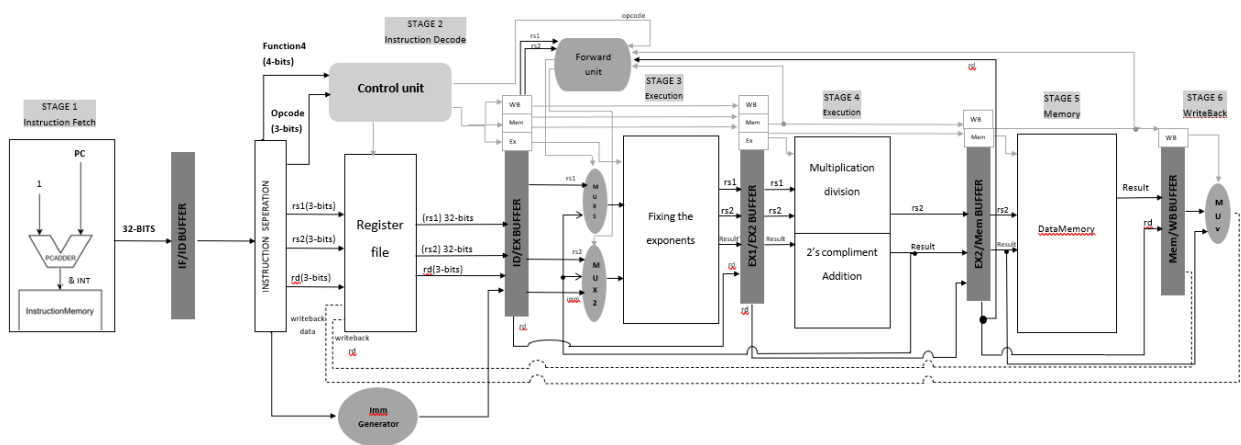


Figure 19: Diagram

8 CONCLUSION

In conclusion, the pipelined arithmetic logic operation project has proven to be a significant advancement in enhancing the efficiency and performance of computational tasks. By implementing a pipelined architecture, we have successfully divided the arithmetic logic operations into multiple stages, allowing for concurrent processing and minimizing idle time. Through this project, we have achieved notable improvements in throughput and latency, enabling faster execution of arithmetic operations. The pipelined design has effectively reduced the critical path delay by breaking down the operations into smaller, independent tasks that can be executed simultaneously. Overall, this pipelined arithmetic logic operation project has showcased the power and benefits of pipelining in achieving higher computational efficiency. It has not only enhanced the speed and throughput of arithmetic operations but also laid the foundation for future advancements in processor design and optimization. [3]

9 REFERENCES

- [1] [Online]. Available: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html.
- [2] [Online]. Available: <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>.
- [3] [Online]. Available: <https://www.geeksforgeeks.org/computer-organization-and-architecture-pipelining-set-1-execution-stages-and-throughput/>.
- [4] "IEEE Editorial Style Manual," 2017. [Online]. Available: https://www.ieee.org/documents/style_manual.pdf.
- [5] [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/800255.810654>.
- [6] <https://ict.iitk.ac.in/wp-content/uploads/CS422-Computer-Architecture-ComputerOrganizationAndDesign5thEdition2014.pdf>
- [7] [Online]. Available: <https://www.bbc.co.uk/bitesize/guides/z2342hv/revision/5>.
- [8] [Online]. Available: <http://www.faadooengineers.com/online-study/post/cse/computer-organization-and-architecture/496/data-hazards>.

We designed our diagram to suit our project.