Computer Organization and Assembly Language Lab (EE-3461L)

Open Ended Lab
Spring 2024

Submitted by:

Name:______Syeda Aishah Asim______________
DSU ID:________EE211027________________

Submitted to:
Engr. Shahnila Badar

Written using Latex Overleaf

Department of Electrical Engineering
DHA SUFFA University
DG-78, Off Khayaban-e-Tufail, Ph-VII (Ext.),
DHA, Karachi-75500

Implementing a Program using a Pre-Designed Single Cycle RISC-V Architecture

Syeda Aishah Asim
EE211027
ee211027@dsu.edu.pk
Electrical Engineering Department
DHA Suffa University, Karachi

***Abstract*— This report details the implementation of a program using a pre-designed single cycle RISC-V architecture, specifically focusing on R-type instructions (add, sub, and, or, slt), memory instructions (lw, sw), and branch instructions (beq), using Verilog Code—**

Keywords—Verilog, R-type, l-type, S-type, B-type, Instruction Memory, Single Cycle, RISC-V, AU, Data Memory, Register File.

I. INTRODUCTION

The implementation of a program using a pre-designed single cycle RISC-V architecture is a fundamental exercise in understanding the interplay between performance, cost, and complexity in microprocessor design. This includes proficiency in arithmetic and memory circuits, as well as a solid grasp of the RISC-V architecture.

This report aims to show the process of implementing a single cycle RISC-V microarchitecture, which, despite its simplicity compared to more advanced pipelined designs, provides a robust framework for understanding the core operations of a microprocessor. The single cycle architecture executes each instruction in a single clock cycle, making it an ideal model for educational purposes and initial design exploration.

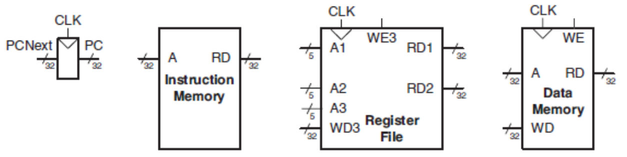
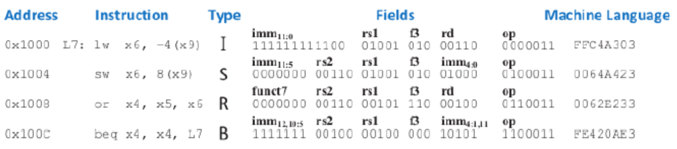


Fig.1 State Elements of a RISC-V Processor

Throughout this report, we will explore the specific arrangement of registers, arithmetic logic units (ALUs), instruction memories, and other logic components required to realize a single cycle RISC-V microarchitecture. Emphasis will be placed on a subset of the RISC-V instruction set, specifically R-type instructions (add, sub, and, or, slt), memory instructions (lw, sw), and branch instructions (beq). These instructions have been selected for their ability to demonstrate the fundamental operations necessary to execute meaningful programs.

By the end of this report, the reader will gain a comprehensive understanding of how to implement and utilize a single cycle RISC-V microarchitecture, appreciating the trade-offs involved in terms of performance, cost, and complexity. This foundational knowledge will serve as a stepping stone for further exploration into more sophisticated microarchitectural designs.

II. THEORETICAL BACKGROUND

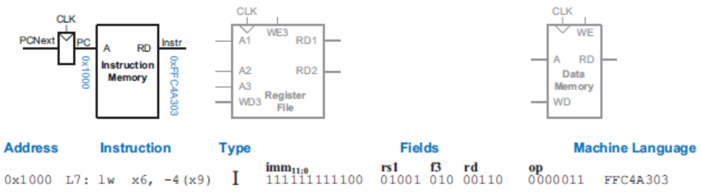


Address	Instruction	Type	Fields						Machine Language
			$imm_{11:0}$		rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100		01001	010	00110	0000011	FFC4A303
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Fig.2 Sample program exercising different types of instructions

III. METHODOLOGY AND SIMULATION

Implementing a program using a pre-designed single cycle RISC-V architecture involves understanding the development and integration of a microprocessor's datapath components. This report details the step-by-step construction of the single cycle datapath, emphasizing new connections and control signals while illustrating the execution of specific instructions.



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

Fig.3 Fetch Instruction from Memory

The process begins with the program counter (PC), which holds the address of the instruction to be executed. The instruction memory reads the instruction from the specified address, as shown in Figure 3. For instance, in our sample program, the PC is set to 0x1000, indicating the starting address for instruction fetching.

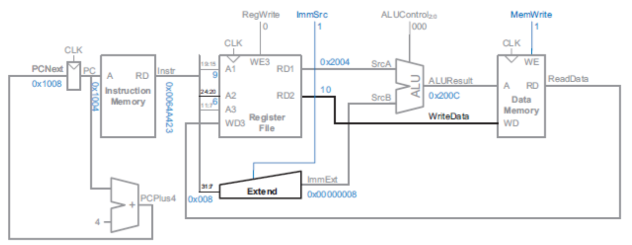
A. I-Type

The `lw` (load word) instruction in RISC-V loads a 32-bit word from memory into a register. As an I-type instruction, it begins by fetching the instruction from memory using the program counter (PC). The base address is read from the source register specified in the `rs1` field (`Instr19:15`) of the instruction. This value is output onto `RD1` from the register file. The instruction's 12-bit immediate value (`Instr31:20`) is then sign-extended to 32 bits to handle potential negative offsets. This extended immediate is added to the base address using an Arithmetic Logic Unit (ALU) to compute the effective memory address. The data at this address is read from memory onto the `ReadData` bus.

Finally, this data is written to the destination register, specified by the `rd` field (`Instr11:7`), on the rising edge of the clock, controlled by the `RegWrite` signal. The process is completed by updating the PC to point to the next instruction.

B. S-Type

The `sw` (store word) instruction in RISC-V stores a 32-bit word from a register into memory. Similar to `lw`, it fetches the instruction using the program counter (PC) and reads the base address from the `rs1` field (`Instr19:15`) of the instruction. The immediate offset for `sw` spans `Instr31:25` and `Instr11:7`, sign-extended to 32 bits by an Extend unit using all bits from `Instr31:7`. The ALU adds this extended immediate to the base address (`RD1`) to compute the memory address.



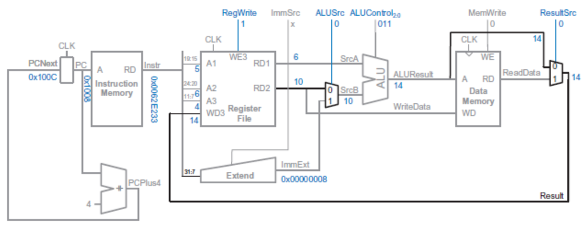
Address	Instruction	Type	Fields						Machine Language
			$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

Fig.4 Write data to memory for sw instruction

Furthermore, `sw` retrieves data from a second register specified in `rs2` (`Instr24:20`) and writes it to memory via the data memory's `WD` input. The memory write operation is controlled by `MemWrite`, set to 1 for `sw`. Unlike `lw`, which updates a register (`RegWrite = 1`), `sw` does not write back to the register file (`RegWrite = 0`). After execution, the PC increments by 4 to prepare for the next instruction.

C. R-Type

For R-type instructions in RISC-V, the datapath is extended to handle operations like add, sub, and, or, and slt. These instructions share a common structure where two source registers (`rs1` and `rs2`) are read from the register file. An ALU performs the specified operation based on the `ALUControl` signal: 000 for addition, 001 for subtraction, 010 for AND, 011 for OR, and 101 for set less than (slt).



Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

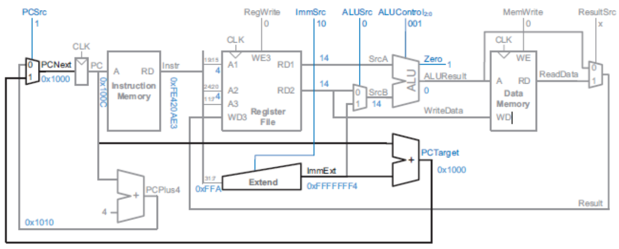
Fig.5 Datapath enhancements for R-type instructions

Figure.5 illustrates the updated datapath for R-type instructions. It includes reading `rs1` and `rs2` from the register file, then selecting `RD2` as `SrcB` using a multiplexer controlled by `ALUSrc`. After computing the result using the ALU, the `Result` multiplexer selects either `ALUResult` (for R-type instructions) or `ReadData` (for lw instructions) based on `ResultSrc` to write back to the register file (`RegWrite`).

In execution, for an instruction like `or` with opcode `0x0062E233`, the PC increments to `0x100C`. The instruction memory fetches `or`, while `x5` and `x6` provide operands `6` and `10` respectively. With `ALUControl` set to `011`, the ALU computes `6 | 10 = 14`. This result (`ALUResult`) is then written back to `x4`.

D. B-type

We've expanded our RISC-V single-cycle processor's datapath to handle the beq (branch if equal) instruction. This instruction compares two registers (`rs1` and `rs2`) fetched from the register file using the ALU. If they are equal, the processor calculates a branch target address by adding a 13-bit signed immediate (branch offset) to the current program counter (PC).



Address	Instruction	Type	Fields						Machine Language
			$imm_{12,10:5}$	rs2	rs1	f3	$imm_{4:1,11}$	op	
0x100C	beq x4, x4, L7	B	1111111	00100	00100	000	10101	1100011	FE420AE3

Fig.6 Datapath enhancements for beq

In Figure.6, we've updated the datapath to accommodate beq instructions. The Extend logic now includes a mode (`ImmSrc = 10`) to select the 13-bit immediate from the instruction bits. An adder computes `PCTarget = PC + ImmExt`, and a multiplexer controlled by `PCSrc` selects `PCTarget` when the ALU's Zero flag indicates equality. This allows the processor to branch accordingly.

For example, executing `beq x4, x4, L7` with opcode `0xFE420AE3` at PC `0x100C`, where both `x4` registers hold the value `14`, the ALU computes `14 - 14 = 0`, setting the Zero flag. The Extend unit extends the 13-bit immediate `0xFFA` (interpreted as -12) to `0xFFFFFFF4`, adding it to PC to yield `PCTarget = 0x1000`. The multiplexer selects `PCTarget` as the next PC, effectively branching back to the start of the code.

This completes the enhancement of our single-cycle processor's datapath to support conditional branching. Next, we will focus on deriving the control signals necessary to control this datapath's operation effectively.

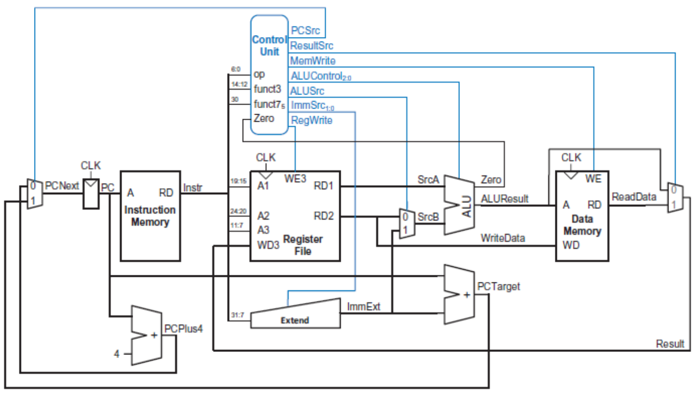


Fig.7 Complete single-cycle processor

IV. RESULTS & CODING

A. *Verilog Design Code: Single Cycle Top*

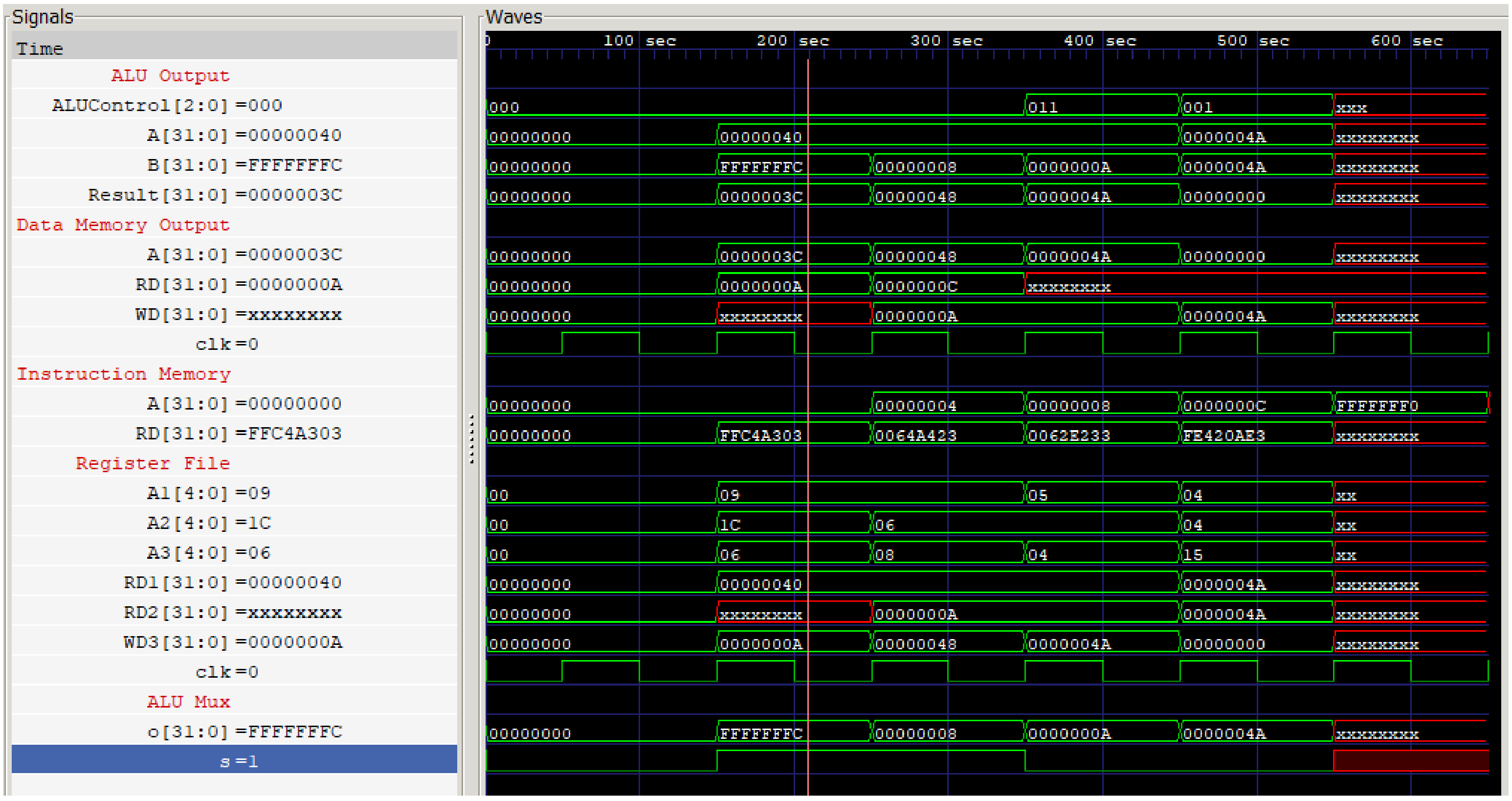
```
`include "PC.v"
`include "Instruction_Memory.v"
`include "Register_File.v"
`include "Sign_Extend.v"
`include "ALU.v"
`include "Data_Memory.v"
`include "PC_Adder.v"
`include "Control_Unit_Top.v"
`include "mux.v"
module Single_Cycle_Top (clk,rst_l);
    input clk,rst_l;

    wire [31:0] PC_Top, RD_Instr, ImmExt_Top,
RD1_Top, WriteResult,
                ReadData, PCPlus4, RD2_Top,
ALU_Result,SrcB,PC_wire,pcnext;
    wire RegWrite, MemWrite;
    wire [1:0] ImmSrc;
    wire [2:0] ALUControl;
    wire ALUSrc, ResultSrc,pcsrc,zro;

    PC_Module PC(
        .clk(clk),
        .rst_l(rst_l),
        .PC(PC_Top),
        .PC_Next(pcnext)
    );

    PC_Adder PC_target(
        .A(PC_Top),
        .B(ImmExt_Top),
        .C(PC_wire)
    );

    Instruction_Memory Instruction_Memory(
        .rst_l(rst_l),
        .A(PC_Top),
        .RD(RD_Instr)
    );

    Register_File Register_File(
        .clk(clk),
        .rst_l(rst_l),
        .A1(RD_Instr[19:15]),
        .A2(RD_Instr[24:20]),
        .A3(RD_Instr[11:7]),
        .RD1(RD1_Top),
        .RD2(RD2_Top),
        .WE3(RegWrite),
        .WD3(WriteResult)
    );

    Sign_Extend Sign_Extend(
        .rst_l(rst_l),
        .ImmSrc(ImmSrc[0]),
        .Instr(RD_Instr),
        .ImmExt(ImmExt_Top)
    );

    Control_Unit_Top Control_Unit(
        .Op(RD_Instr[6:0]),
        .RegWrite(RegWrite),
        .ImmSrc(ImmSrc),
        .ALUSrc(ALUSrc),
        .MemWrite(MemWrite),
        .ResultSrc(ResultSrc),
        .Branch(),
        .funct3(RD_Instr[14:12]),
        .funct7(),
        .ALUControl(ALUControl),
        .Zero(zro),
        .PCSrc(pcsrc)

    );

    mux pcmux(
        .a(PCPlus4),
        .b(PC_wire),
        .s(pcsrc),
        .o(pcnext)
    );

    mux alumux(
        .a(RD2_Top),
        .b(ImmExt_Top),
        .s(ALUSrc),
        .o(SrcB)
    );

    mux rsltmux(
        .a(ALU_Result),
        .b(ReadData),
        .s(ResultSrc),
        .o(WriteResult)

    );

    ALU ALU(
        .A(RD1_Top),
        .B(SrcB),
        .Result(ALU_Result),
        .ALUControl(ALUControl),
        .OverFlow(),
        .Carry(),
        .Zero(zro),
        .Negative()
    );

    Data_Memory Data_Memory(
        .clk(clk),
        .rst_l(rst_l),
        .WE(MemWrite),
        .WD(RD2_Top),
        .A(ALU_Result),
        .RD(ReadData)
    );

    PC_Adder PC_Adder(
        .A(PC_Top),
        .B(32'h00000004),
        .C(PCPlus4)
    );
endmodule
```

B. Verilog Testbench Code: Single Cycle Top Testbench

```
module Single_Cycle_Top_Tb ();

    reg clk=1'b1,rst_l;

    Single_Cycle_Top Single_Cycle_Top(
                            .clk(clk),
                            .rst_l(rst_l)
    );

    initial begin
        $dumpfile("Single Cycle.vcd");
        $dumpvars(0);
    end

    always
    begin
        clk = ~ clk;
        #50;

    end

    initial
    begin
        rst_l <= 1'b0;
        #150;

        rst_l <=1'b1;
        #500;
        $finish;
    end
endmodule
```

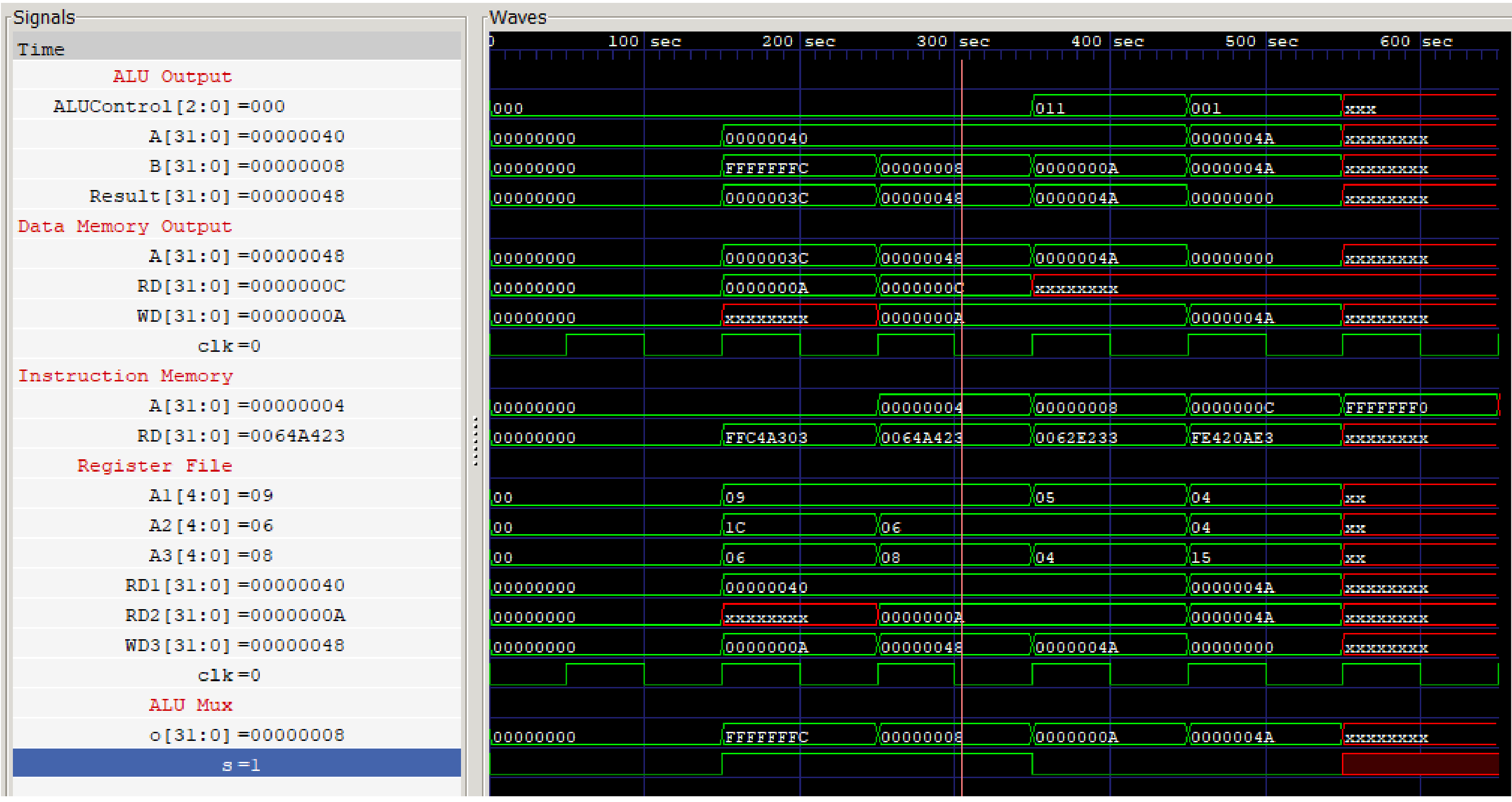
C. GtkWave Output

- ***I-type:***



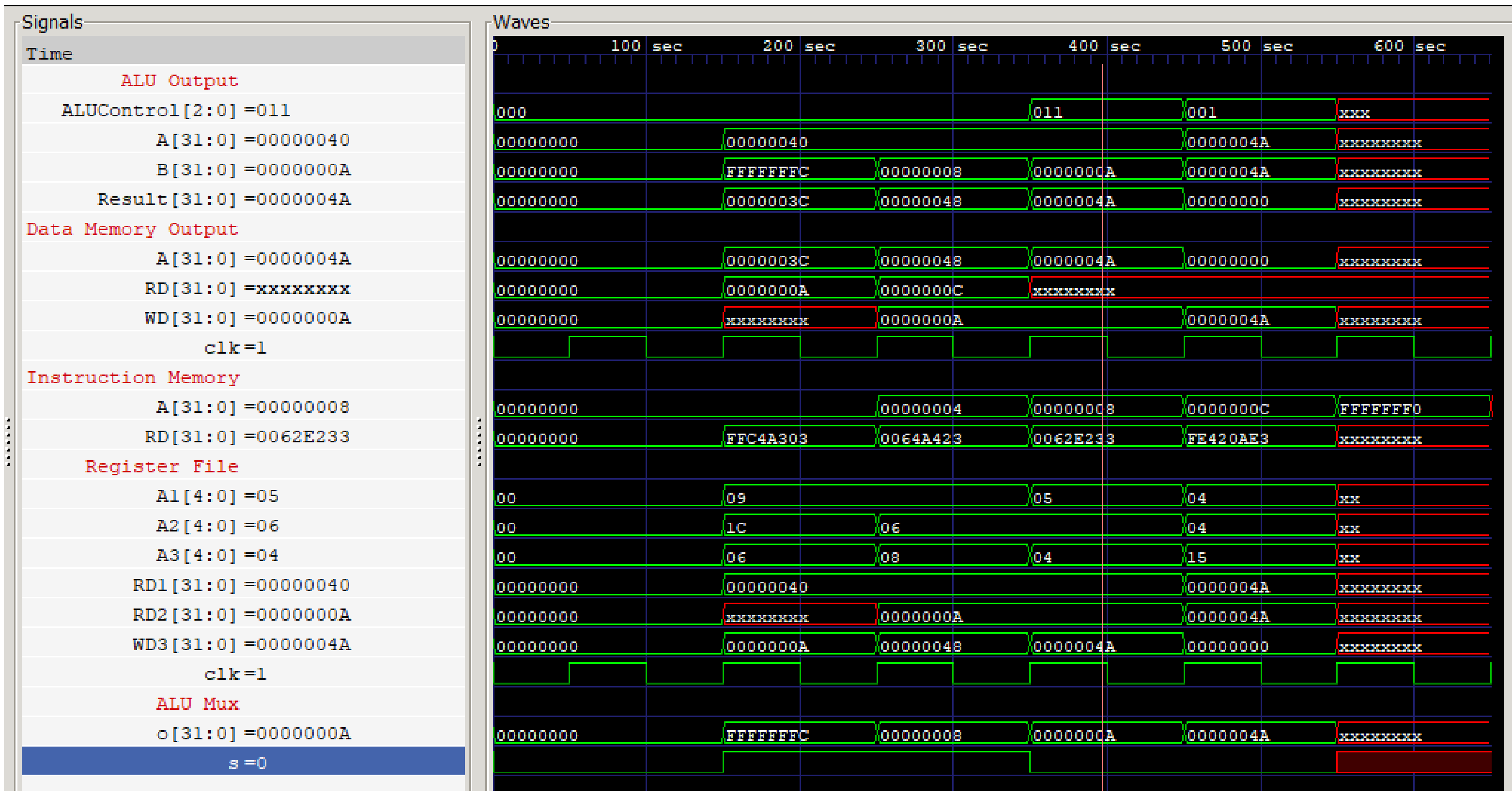
```
Signals
Time
        ALU Output
   ALUControl[2:0]=000
         A[31:0]=00000040
         B[31:0]=FFFFFFFC
    Result[31:0]=0000003C
Data Memory Output
         A[31:0]=0000003C
        RD[31:0]=0000000A
        WD[31:0]=xxxxxxxx
               clk=0
Instruction Memory
         A[31:0]=00000000
        RD[31:0]=FFC4A303
     Register File
          A1[4:0]=09
          A2[4:0]=1C
          A3[4:0]=06
       RD1[31:0]=00000040
       RD2[31:0]=xxxxxxxx
       WD3[31:0]=0000000A
               clk=0
           ALU Mux
         o[31:0]=FFFFFFFC
                 s=1
```

- ***S-type:***



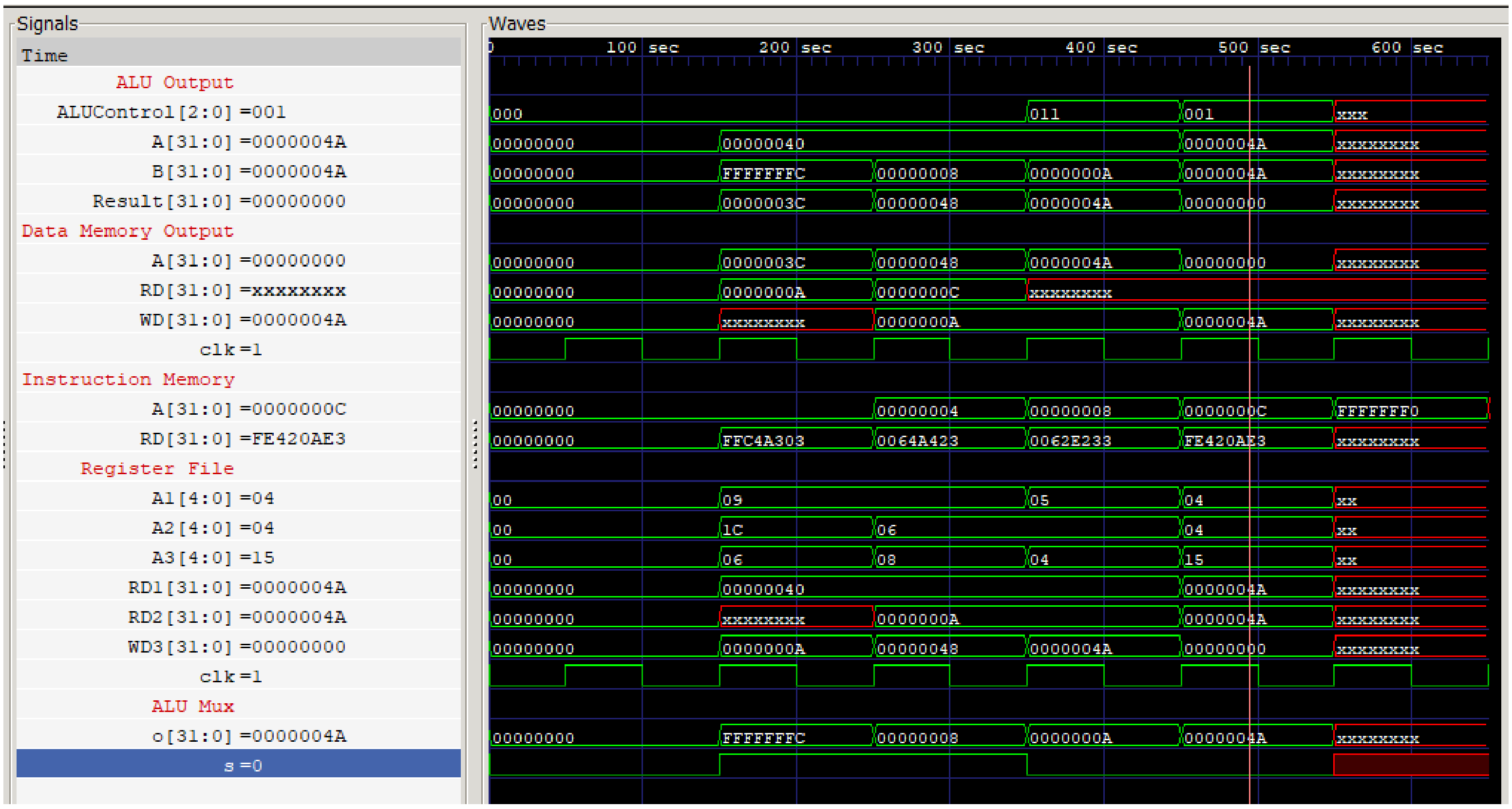
```
Signals
Time
        ALU Output
   ALUControl[2:0]=000
         A[31:0]=00000040
         B[31:0]=00000008
    Result[31:0]=00000048
Data Memory Output
         A[31:0]=00000048
        RD[31:0]=0000000C
        WD[31:0]=0000000A
               clk=0
Instruction Memory
         A[31:0]=00000004
        RD[31:0]=0064A423
     Register File
          A1[4:0]=09
          A2[4:0]=06
          A3[4:0]=08
       RD1[31:0]=00000040
       RD2[31:0]=0000000A
       WD3[31:0]=00000048
               clk=0
           ALU Mux
         o[31:0]=00000008
                 s=1
```

- ***R-type:***



```
Signals                              Waves
Time                                 0   100 sec   200 sec   300 sec   400 sec   500 sec   600 sec
        ALU Output
  ALUControl[2:0] =011               000 | 011 | 001 | xxx
         A[31:0] =00000040           00000000 | 00000040 | 0000004A | xxxxxxxx
         B[31:0] =0000000A           00000000 | FFFFFFFC | 00000008 | 0000000A | 0000004A | xxxxxxxx
    Result[31:0] =0000004A           00000000 | 0000003C | 00000048 | 0000004A | 00000000 | xxxxxxxx
Data Memory Output
         A[31:0] =0000004A           00000000 | 0000003C | 00000048 | 0000004A | 00000000 | xxxxxxxx
        RD[31:0] =xxxxxxxx           00000000 | 0000000A | 0000000C | xxxxxxxx
        WD[31:0] =0000000A           00000000 | xxxxxxxx | 0000000A | 0000004A | xxxxxxxx
             clk =1
Instruction Memory
         A[31:0] =00000008           00000000 | 00000004 | 00000008 | 0000000C | FFFFFFF0
        RD[31:0] =0062E233           00000000 | FFC4A303 | 0064A423 | 0062E233 | FE420AE3 | xxxxxxxx
     Register File
         A1[4:0] =05                 00 | 09 | 05 | 04 | xx
         A2[4:0] =06                 00 | 1C | 06 | 04 | xx
         A3[4:0] =04                 00 | 06 | 08 | 04 | 15 | xx
       RD1[31:0] =00000040           00000000 | 00000040 | 0000004A | xxxxxxxx
       RD2[31:0] =0000000A           00000000 | xxxxxxxx | 0000000A | 0000004A | xxxxxxxx
       WD3[31:0] =0000004A           00000000 | 0000000A | 00000048 | 0000004A | 00000000 | xxxxxxxx
             clk =1
         ALU Mux
         o[31:0] =0000000A           00000000 | FFFFFFFC | 00000008 | 0000000A | 0000004A | xxxxxxxx
               s =0
```

- ***B-type:***



```
Signals                              Waves
Time                                 0   100 sec   200 sec   300 sec   400 sec   500 sec   600 sec
        ALU Output
  ALUControl[2:0] =001               000 | 011 | 001 | xxx
         A[31:0] =0000004A           00000000 | 00000040 | 0000004A | xxxxxxxx
         B[31:0] =0000004A           00000000 | FFFFFFFC | 00000008 | 0000000A | 0000004A | xxxxxxxx
    Result[31:0] =00000000           00000000 | 0000003C | 00000048 | 0000004A | 00000000 | xxxxxxxx
Data Memory Output
         A[31:0] =00000000           00000000 | 0000003C | 00000048 | 0000004A | 00000000 | xxxxxxxx
        RD[31:0] =xxxxxxxx           00000000 | 0000000A | 0000000C | xxxxxxxx
        WD[31:0] =0000004A           00000000 | xxxxxxxx | 0000000A | 0000004A | xxxxxxxx
             clk =1
Instruction Memory
         A[31:0] =0000000C           00000000 | 00000004 | 00000008 | 0000000C | FFFFFFF0
        RD[31:0] =FE420AE3           00000000 | FFC4A303 | 0064A423 | 0062E233 | FE420AE3 | xxxxxxxx
     Register File
         A1[4:0] =04                 00 | 09 | 05 | 04 | xx
         A2[4:0] =04                 00 | 1C | 06 | 04 | xx
         A3[4:0] =15                 00 | 06 | 08 | 04 | 15 | xx
       RD1[31:0] =0000004A           00000000 | 00000040 | 0000004A | xxxxxxxx
       RD2[31:0] =0000004A           00000000 | xxxxxxxx | 0000000A | 0000004A | xxxxxxxx
       WD3[31:0] =00000000           00000000 | 0000000A | 00000048 | 0000004A | 00000000 | xxxxxxxx
             clk =1
         ALU Mux
         o[31:0] =0000004A           00000000 | FFFFFFFC | 00000008 | 0000000A | 0000004A | xxxxxxxx
               s =0
```

V. CONCLUSION

In conclusion, we have successfully implemented and extended the datapath of our RISC-V single-cycle processor using Verilog code. This implementation supports a variety of fundamental instructions including lw (load word), sw (store word), R-type (add, sub, and, or, slt), and B-type (beq). Each instruction type utilizes different components of the datapath, such as the ALU, register file, memory units, and control logic, demonstrating the versatility and effectiveness of the single-cycle architecture in executing these operations within a single clock cycle.

By systematically coding and integrating these components in Verilog, we have created a functional processor capable of performing basic arithmetic computations, managing memory access, and executing conditional branches. The next crucial step involves designing the control unit in Verilog to coordinate these datapath elements efficiently, ensuring precise execution of instructions according to the RISC-V architecture specifications.

VI. REFERENCES

[1] Sarah Harris, David Harris - Digital Design and Computer Architecture_ RISC-V Edition-Morgan Kaufmann (2021)

[2] efaidnbmnnnibpcajpcglclefindmkaj/https://ijariie.com/AdminUploadPdf/DESIGN_OF_SINGLE_CYCLE_RISC_V_PROCESSOR_ijariie22774.pdf

[3] efaidnbmnnnibpcajpcglclefindmkaj/https://passlab.github.io/CSE564/notes/lecture08_RISCV_Impl.pdf

[4] https://sirinsoftware.com/blog/inside-risc-v-microarchitecture#:~:text=The%20base%20instruction%20set%20serves,range%20of%20devices%20and%20applications.