ENGG*3380 - Computer Organization and Design

Project Phase 2 - Datapath and Control Design

Professor:
Dr. Mohamed Hassan

Group: 6

Granic Anthony - ID: 0994824

Ayyache Bilal - ID: 0988616

Bello Mustapha - ID: 0998244

Submission Date: Friday, March 15, 2019

School Of Engineering

University of Guelph

Contents

List of Figures

List of Tables

1 Introduction

The main purpose of this report is to highlight the approach of designing and building a MIPS Assembly Multicycle Architecture. The CPU designed includes ALU's, registers, memory, and multiplexers. The main objective of this projects is to build a multicycle architecture using a microprogrammed Control path and pipelining to optimize the performance of the CPU. The architecture will be tested using a MIPS Assembly program that executes the Fibonacci sequence. This project will be implemented using VHDL. Design will be simulated on ISE Design Suite and tested on a Spartan FPGA board.[1]

2 Architecture Description

2.1 Architecture General Description

The main purpose of phase one report is to highlight the approach of designing and building a MIPS Assembly Multicycle Architecture. The CPU designed includes ALU's, registers, memory, and multiplexers. The main objective of this projects is to build a multicycle architecture using a microprogrammed Control path and pipelining to optimize the performance of the CPU. The architecture will be tested using a MIPS Assembly program that executes the Fibonacci sequence. This project will be implemented using VHDL. Design will be simulated on ISE Design Suite and tested on a Spartan FPGA board.

2.2 Assembly Language

Name	**Syntax**	**RTL**	**Machine format**	**Addressing Mode**	**Applicable Status Flags**
add	add rd, rt, rs	R[d] <- R[t] + R[s]	R	Register	OVF
Sub	sub rd, rt, rs	R[d]<- R[t] – R[s]	R	Register	OVF
Sll	sll rd, rt, rs	R[d]<- R[s] <<r[t]	R	Register	OVF
J	j offset	PC <- nPC	J	PC Offset	
Lw	lw, rt, offset(rs)	R[t] <- MEM[R[s] + offset]	I	Memory and Offset	
Sw	sw rt, offset(rs)	MEM[R[s] + offset] <- R[t]	I	Memory and Offset	
addi	addi, rd, rt, immi	R[d] <- R[t] + #immi	I	Immediate	OVF
Beq	beq, rt, rs, loc	If(R[s] == R[t]) Branch to loc	I	PC Offset	ZERO
Out	out	DR <- Mem[s]	S	Segment type	
Ls	ls $t0, s	R[d] <- s	S	Segment type	

Table 1: Assembly Language Table

As mentioned in the introduction section, a MIPS Assembly code will be used to test the multicycle architecture designed. R type, J type, I type, and S type instructions will be sent to the architecture to

execute the Fibonacci Sequence. To be able to execute these instructions, the programmer should implement the instructions into the Instruction register first. The CPU should be able to recognise the name of the instruction, syntax, RTL, Machine format, addressing mode, and the applicable status flags. The table above describes the instructions implemented into the instruction Register.

2.3 MIPS Assembly Program

The MIPS assembly program provided in lab 2 was not the most efficient was to execute the Fibonacci sequence as it included a lot of different instructions that could have been replaced by an instruction that was already implemented. To keep the architecture simple, a new MIPS assembly program was programmed which included less instructions. This optimizes the CPU as less instructions need to be executed to achieve the goal of the project. The new MIPS Assembly code is mentioned below:

```
    addi $s2, $zero, d          #load address of array
    addi $s5, $zero, size       #load address of size variable
    lw $s5, 0($s5)              #load array size
    addi $s0, $zero, 10         #A
    addi $s1, $zero, 5          #B
    addi $s0, $zero, 0          #i
    addi $s0, $zero, 0          #j

loop1:
    beq $s0, $t0 Exit           #if i(t0) is less than A(s0), exit
    addi $t0, $t0, 1            #increment i
    addi $t1 $zero, 0           #reset j
    j loop2

loop2:
    beq $s1, $t1 loop1          #if j(t1) is less than B(s1), go to loop 1
    addi $t3, $t0, -1           #decrement i once so we get original
    add $t4, $t1, $t3           #adds j and temp i
    sll $t5, $t1, 4             #finding offset 4 * j
    add $t5, $t5, $s2           #adding offset to base array location
    sw $t4, 0($t5)              #storing i + j to final location
    addi $t1, $t1, 1            #increment j
    j loop2
```

(To run the program above make sure an exit function that calls syscall is included in the assembly file. To set the address of the array replace d by the number of space in the array and replace size by a value that describes the size of the array. For this project D will be set to 12 and size will be set to 12.)

2.4 Addressing Modes

The table below describes the 3 different addressing modes and introduces an example of how an instruction is used in that specific mode. The RTL and Description of the different addressing modes is also included below:

Addressing	Example	RTL	Description
Immediate	slti $s2, $s3, 4	R[d] <- R[s] + value	Operand encoded in instruction and also found in register
Register	add $s2, $s3, $s4	R[d] <- R[s] + R[t]	Required operand found in registers
Base Offset	lw $s1, 9($s3)	R[d] <- $t0 + 4	Operand found in a base address that has been offset

Table 2: Addressing Modes

2.5 Conditional Branch Instructions

The conditional branch instructions that will be executed by the program are described below. The software implementation and description of every instruction used is described in the table below:

Instruction	Interpretation	Software Implementation
beq	Branches if two registers values are equal	beq $t0, $t1, jump
blt	Branches if register one contents are less than register two contents	slt $t0, rs, rt beq $t0, $t1, jump
bge	Branches if register one contents are greater than register two contents	slt $t0, rs, rt beq $t0, $t0, jump

Table 3: Conditional Branch Instructions

2.6 Machine Format

The machine format of the MIPS Assembly instruction will be used to help the architecture determine what state to jump to. The OP code of each instruction will be read in the control path module to signal other modules in the system to run, and send other module to idle module depending on the instruction requested. Machine format includes R type, I type, J type, and S type format. Every format is different in terms of how the registers is set. The machine format of each type is described below:

R: Op code is 4 bits long, the first/second/Destination registers are 4 bits long

Op code	Destination Register	First Register	Second Register

I: the opcode is 4 bits long, the first/second registers are 4 bits long, and the offset is 4 bits long

Op code	First Register	Second Register	Offset

J: the opcode is 4 bits long, while the address is 12 bits long

Op code	Address

S: Opcode is 4 bits long, Read Register Address is 4 bits, rest is don't care. For in/out instructions

Op code	Read Register	XXXX XXXX

2.7 OP Code Decoder

As described in the section above the OP code has to be decoded to inform the control path of what instruction needs to be excuted. The OPCode will allow the control path to enable other module in the design. The diagram below describes how an Opcode decoder determine which ALU to run:

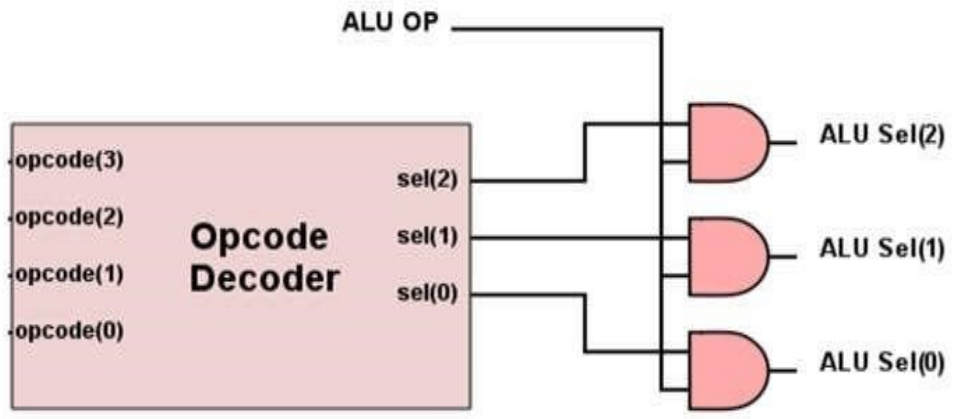

Figure 1: Datapath Block Diagram

2.8 Control Path Block Diagram

The control unit will decide which state to execute depending on the OP code provided to the control unit. For phase one, the control path has not been finalized but a rough block diagram was designed to help understand the approach to implement the control unit into the system. The block diagram below describes a rough idea of how the control path will look like in phase 2 (The signals provided below are subject to change):

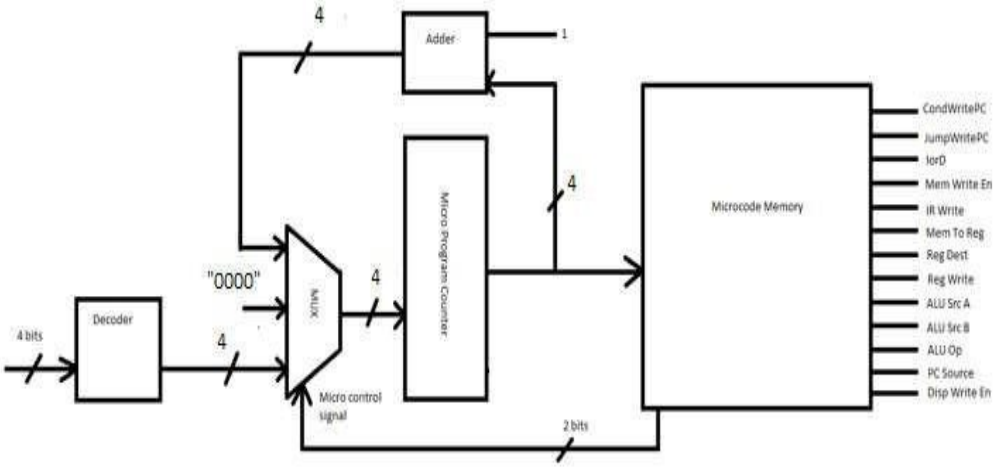

Figure 2: Datapath Block Diagram

3 Datapath Description

3.1 Datapath Block Diagram

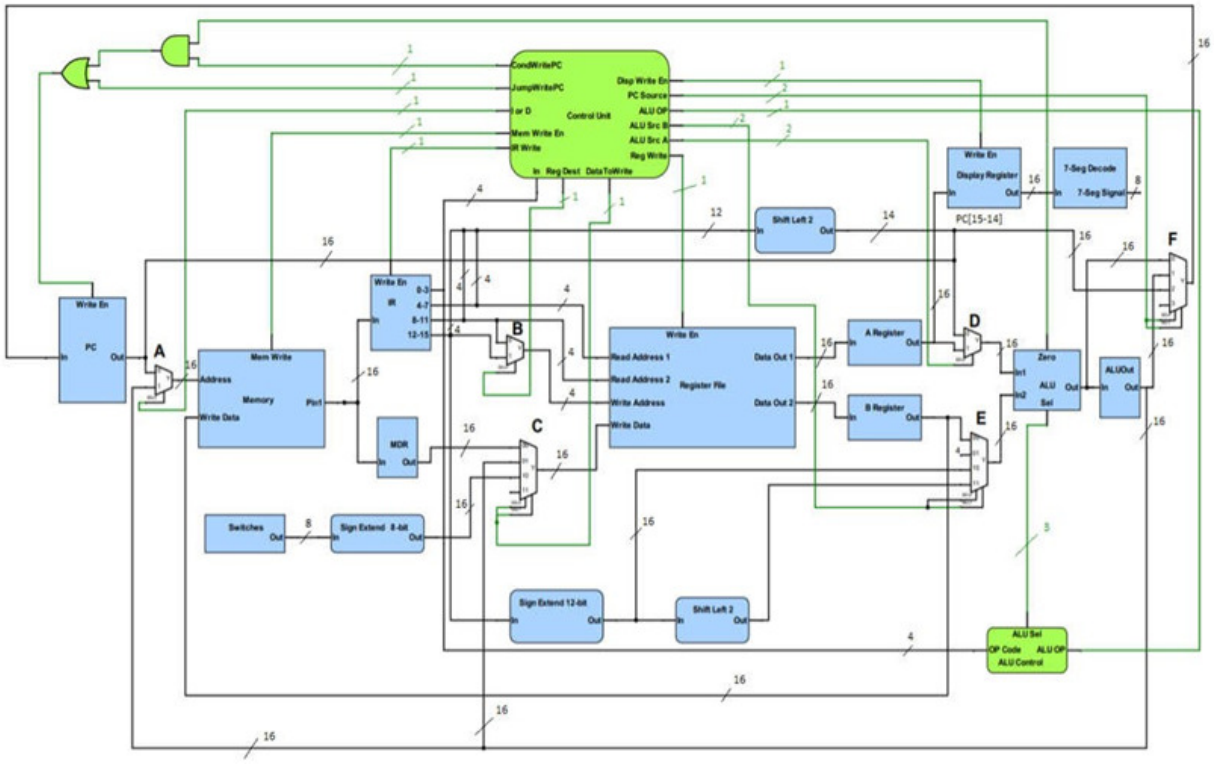

Figure 3: Datapath Block Diagram

3.2 Datapath Components

- **Program Counter (PC)** – a register in the processor which stores the address of the desired instruction in memory. Every Instruction this value is incremented by four unless a jump or branch occurs as indicated by the counter write signal (JumpWritePC) or program counter conditional signal (CondWritePC).
- **MUX A** – a 2 to 1 mux that selects an output based on the input or data signal (I or D). The two inputs to the mux comes from the program counter, which points to the instruction to be performed, and the output of ALU. This determines whether the memory is being used for data or for fetching instructions.
- **Memory** – takes in an address, accesses the address and outputs the data stored at that address. It

also writes data that is inputted from ALU out based on the memory write signal (Mem write En) to the memory address provided.

- **Instruction Register (IR)** – holds the instruction currently being executed based on the Instruction Register write (IR Write) signal.
- **Memory Data Register (MDR)** – holds the data fetched from the memory.
- **MUX B** – a 2 to 1 mux that selects an output based on the register destination signal (Reg Dest). The two inputs are the address to second register and destination register. This is used for Rtype and I-type instructions respectively.
- **MUX C** – a 4 to 1 mux that selects an output based on the memory to register file signal (DataToWrite). The three inputs are the output from ALU out, the output from MDR, and the input from the switches. The output of this component is passed to write data of the register file.
- **Register File** – component that holds an array of registers. Using the inputs, and the register write signal (Reg Write), accesses and writes to the registers.
- **A Register** – register that stores the data out 1 from the register file.
- **B Register** – register that stores the data out 2 from the register file.
- **MUX D** – a 2 to 1 mux that takes in data stored in A register and the instruction shifted left 2 bits. Based on the signal, ALU Src A from control unit, it selects the appropriate input and sends it to the ALU.
- **MUX E** – a 4 to 1 mux that takes in data stored in B register, 4, sign extended offset, and offset shift left by 2 bits. Based on the signal, ALU Src B from control unit, it selects the appropriate input and sends it to the ALU.
- **ALU** – component that performs arithmetic and logic operations using the data provided by MUX D and MUX E. It is controlled by an ALU control unit which instructs the ALU to perform a certain instruction.
- **ALU out** – register that stores the output of the ALU.
- **MUX F** – a 4 to 1 mux that takes in the output from ALU, the output from ALU out, and the instruction shifted left by 2 bits. Based on the signal PC Source from the control unit, it selects the proper input and sends it to the program counter.
- **Display Register** – register that stores the output out of the A register. When the Display write enable signal (Disp Write En) is on, that stored data is sent to seven segment display.
- **7-seg Decode** – component that takes in an input, deciphers what the seven-segment display should display based on the input and outputs it as an 8 bit number.

3.3 Justification of Datapath Design

The Datapath designed is based off the MIPS architecture. MIPS architecture is known for its simple architecture as it is based on a reduced instruction set computer (RISC). Some changes where made compared to the typical MIPs architecture, some changes were made to keep the design process simple. Changes include using a 16-bit implementation instead of a 32 bits architecture. Architecture processes I, J, and R type instructions just like a typical MIPS architecture. An S type instruction which consists of 4-bit opcode, a 4-bit register address, along with 12 bits don't care bits was implemented to display data to the seven-segment display. Another major difference between the design implemented in this project and a typical implementation lies behind the setup of the R-type instruction. Shamt and Function was not included in this implementation as only 12 instructions are needed to implement the Fibonacci sequence after code simplification. This keeps the design simple and easy to use.

4 Control Unit Design

4.1 Block Diagram

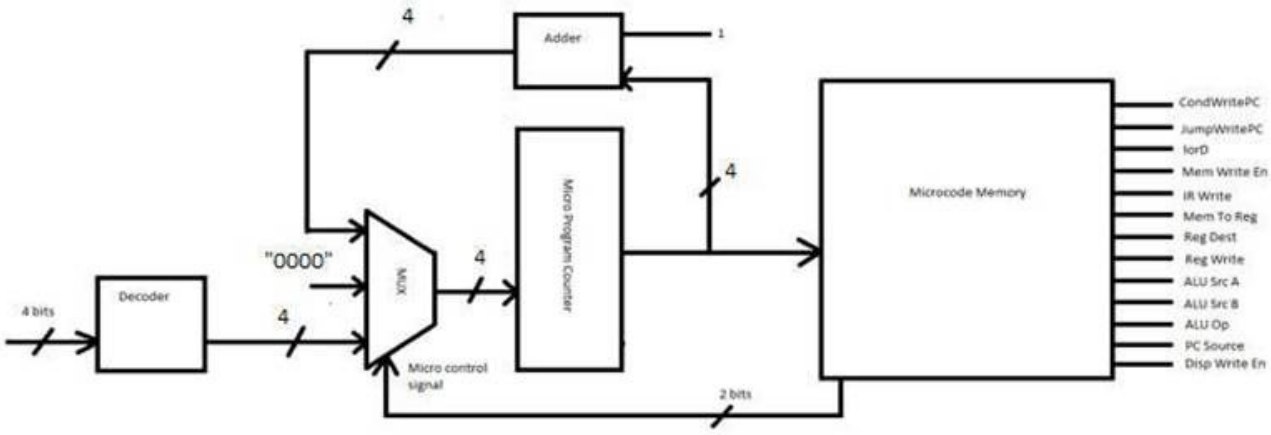

Figure 4: Control Unit Block Diagram

(The control unit proposed in phase one will be used. No changes will be made to previous proposal)

4.2 Control Unit Description

A Micro Control unit will be used to implement this project. The Micro Control controls the output of the mux to the micro counter register based off the first two bits stored in memory.

4.3 Control Unit Signals

1. Conditional write to program counter signal **(CondWritePC)**– signal used to update the program counter when the zero flag is on from the ALU along with a conditional branch
2. Jump write to program counter signal **(JumpWritePC)** – signal used to update the program counter when an unconditional jump instruction is executed and increment by 4 each fetch
3. Instruction or data signal **(I or D)** – signal used in MUX A that selects between instruction memory and data out from ALU out based on the instruction
4. Memory write enable signal **(Mem write En)** – signal that enables or disables the write to memory
5. Write to instruction register signal **(IR Write)** – signal that enables and disables the write to instruction register

6. Register destination selector signal **(Reg Dest)** – acts as a select line for MUX B, determining the destination register
7. Data to write selector signal **(DataToWrite)(**2 bits)** – acts as a select line for MUX C, determining if the data being written is from the memory, the ALU Out or the switches.
8. Register write enable signal **(Reg Write)** – signal that enables or disables the write to register file
9. ALU operand one selector signal **(ALU Src A)** – acts as select lines for MUX D, with the role of obtaining the first source operand
10. ALU operand two selector signal **(ALU Src B) (**2 bits)** – acts as select lines for MUX E, with the role of obtaining the second source operand
11. ALU operation selector signal **(ALU OP)** – signal that determines whether the stage is fetch/decode or another stage. If it is fetch or decode the ALU does signed addition otherwise it looks to the instruction OP-Code for the proper ALU operations.
12. PC source selector signal **(PC Source) (**2 bits)** – acts as select lines for MUX F, with the role of determining which input to use to change the program counter
13. Display to seven segment enable signal **(Disp Write En)** – Determines if the display to the seven segment display is enabled or disabled

4.4 Hardware Description

- Decoder: takes opcode as 4-bit input and outputs the corresponding 5-bit address for the microcode memory
- **Mux**: Four to one mux that takes the following three inputs: address 0, next address, and address from decoder. Mux outputs one of the inputs based on the select lines supplied from the microcode memory
- **Micro Program Counter**: A register that holds the current address value being used by the microcode memory
- **Adder**: Component adds one to the address currently held micro program counter and sends the output sum to the mux (represents next address)
- **Microcode Memory**: Holds the string/code that represents which control signal will be on/off for the given instruction

4.5 ALU Control Unit

4.5.1 Block Diagram

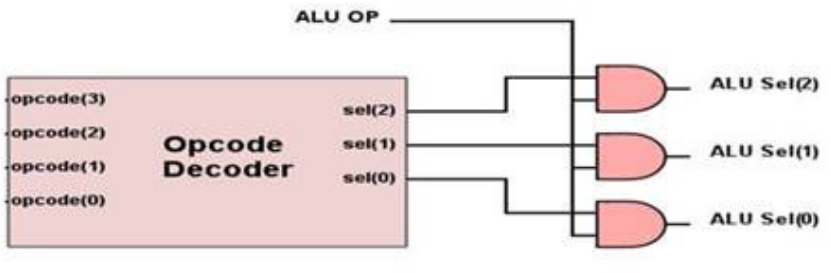

Figure 5: ALU Control Unit Block Diagram

4.5.2 ALU Control Unit Instruction/Truth Table

Table 4: ALU Control Unit Truth Table

Instruction name	Inputs					Outputs			ALU operation based on output
	ALU OP	Opcode(3)	Opcode(2)	Opcode(1)	Opcode(0)	ALU sel (2)	ALU sel (1)	ALU sel (0)	
Fetch/ decode	0	X	X	X	X	0	0	0	Signed add
add	1	0	0	0	0	0	0	1	Unsigned add
lw	1	0	0	0	1	0	0	1	Unsigned add
sw	1	0	0	1	0	0	0	1	Unsigned add
addi	1	0	0	1	1	0	0	1	Unsigned add
sub	1	0	1	0	0	0	1	0	Signed subtraction
slt	1	0	1	0	1	0	1	1	Set less than
sll	1	0	1	1	0	1	0	0	Shift left logical
beq	1	0	1	1	1	1	0	1	Branch equal
X	1	X	X	X	X	X	X	X	X

4.6 Micro Program Instructions

Table 5: Micro Program Instructions

Step Name	Control Unit Address	Micro-Instruction
Fetch	0000	100100100000010000
Decode	0001	000000000000110000
Execute Display	0010	010000000000001000
Execute Jump	0011	010100000000001100
Execute Branch	0100	011000000001001100
Execute R Type	0101	100000000001001000
Write Back R Type	0110	010000010110001000
Execute Save Word	0111	100000000001101000
Memory Access Save Word	1000	010011000000001000
Execute Load Word	1001	100000000001101000
Memory Access Load Word	1010	100010000000001000
Write Back Load Word	1011	010000000010001000
Execute I Type	1100	100000000001101000
Write Back I Type	1101	010000010110001000
Write Back Switches	1110	010000001010001000

The first 2 bits represent the micro program control signal. The next 9 bits represent 9 signals (1 bit represents one signal) and are as follows: Cond Write PC, Jump Write PC, I or D, Mem Write En, IR Write, Reg Dest, Data to Write, Reg Write, and ALU Src A. The last 6 bits represent 3 signals (2 bits for 1 signal) and are as follows: ALU SrcB, ALUOP, and PC Source.

5 Multicycle Implementation

The figure below shows how the control unit moves between states. It begins at fetch, and depending on the Instruction type, finishes at a certain point and then proceeds to fetch the next instruction. Each of the blocks are in the following format: Step Name/Control Unit Address.

5.1 Multicycle State Diagram

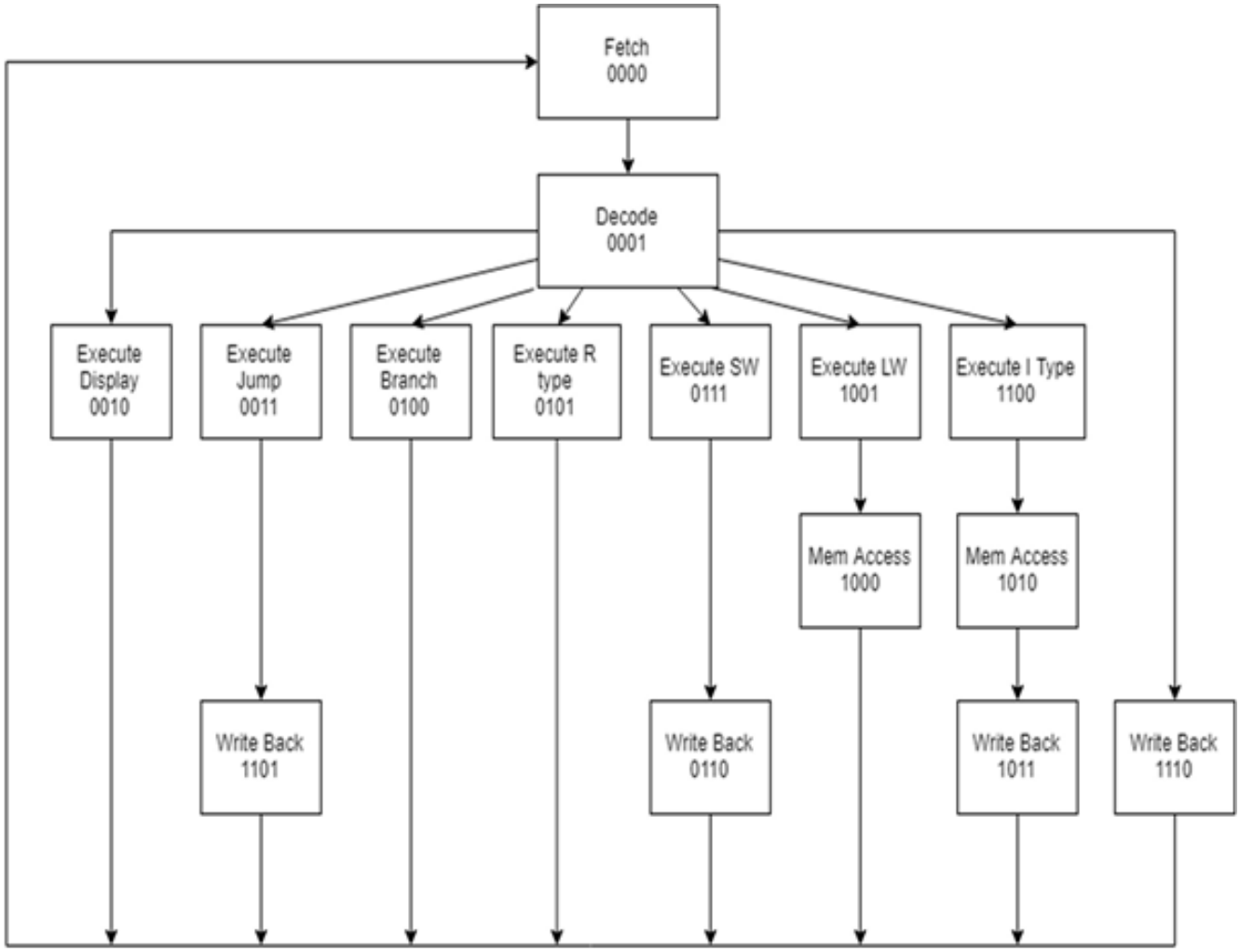

Figure 6: Multicycle State Diagram

5.2 Multicycle State Description

The multicycle processor has five different stages for each instruction: Fetch, Decode, Execute, Memory Access, Write Back.

The following tables and diagrams represent the signal/microinstruction outputs from the control unit for each of the five stages.

5.3 Fetch Stage

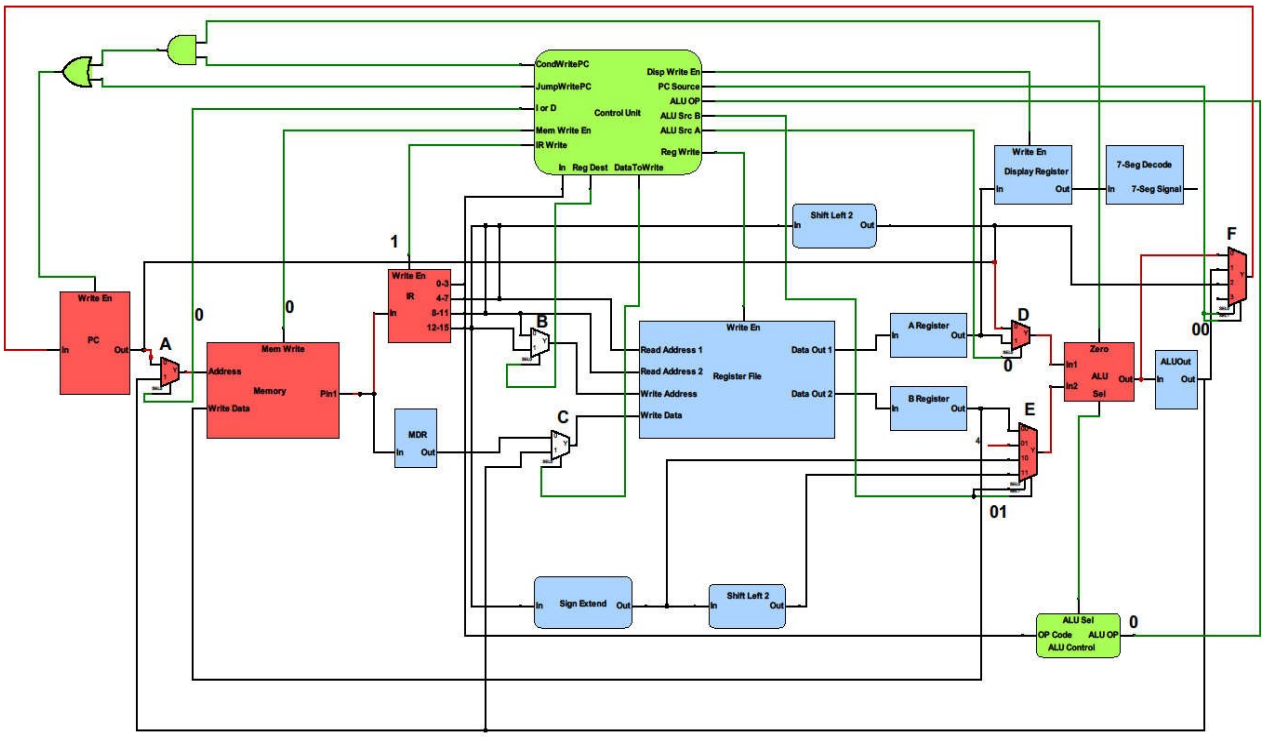

Figure 7: Fetch Stage

During the fetch stage, the instruction is pulled from the instruction register and passed on to the rest of the data path. During this stage, the program counter is also incremented and stored this in the PC register. The instruction is temporarily saved in a register for a multicycle design. This stage is the same for every kind of instruction. The instruction present in the memory corresponding to the value in the program counter should be made available to instruction register. This is achieved by reading the memory at the positive edge of the first clock cycle (memread control signal is asserted) and writing instruction register during the negative edge of the same cycle (IRwrite control signal is asserted).

5.4 Decode Stage

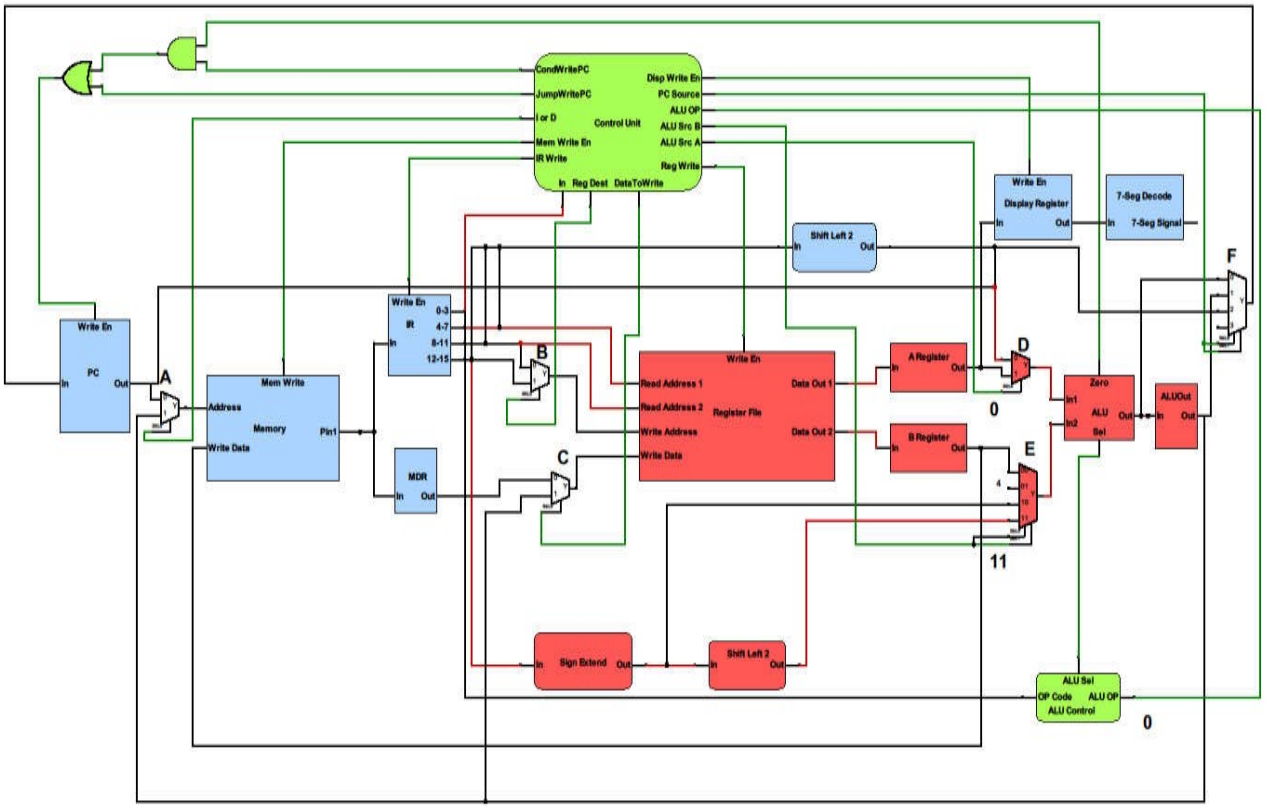

Figure 8: Decode Stage

This stage decides which operation is being run. The instruction stored in IR is decoded by the control unit and the address of the desired micro instruction is stored in a register in the control unit. The data at the rs and rt addresses in the register file are retrieved and stored in the A and B registers respectively in case they are needed later. The ALU is used to calculate the branch value in case the instruction is a branch. The Common to all instruction types. In the instruction decode state, the fetched instruction from the previous state is analyzed to determine which type of instruction has to be executed in the cycles to follow. For this reason, the op-code of the instruction is fed into the controller, and the data from memory corresponding to the address provided by the least significant bits of the instruction is read (memRead is asserted). This data is either a constant operand (offset) required by arithmetic (jump) instructions, or it is an address (pointing to data memory) used by load (store) instructions.

5.5 Execute Stage

The execution stage is where the ALU computes its operation based on the select lines being fed into it (e.g. which operation is being executed). In the instruction execution state, the data and the control lines corresponding to the instruction to be executed are available, and the stage is set for the following processes to take place:

5.5.1 I Type Instructions

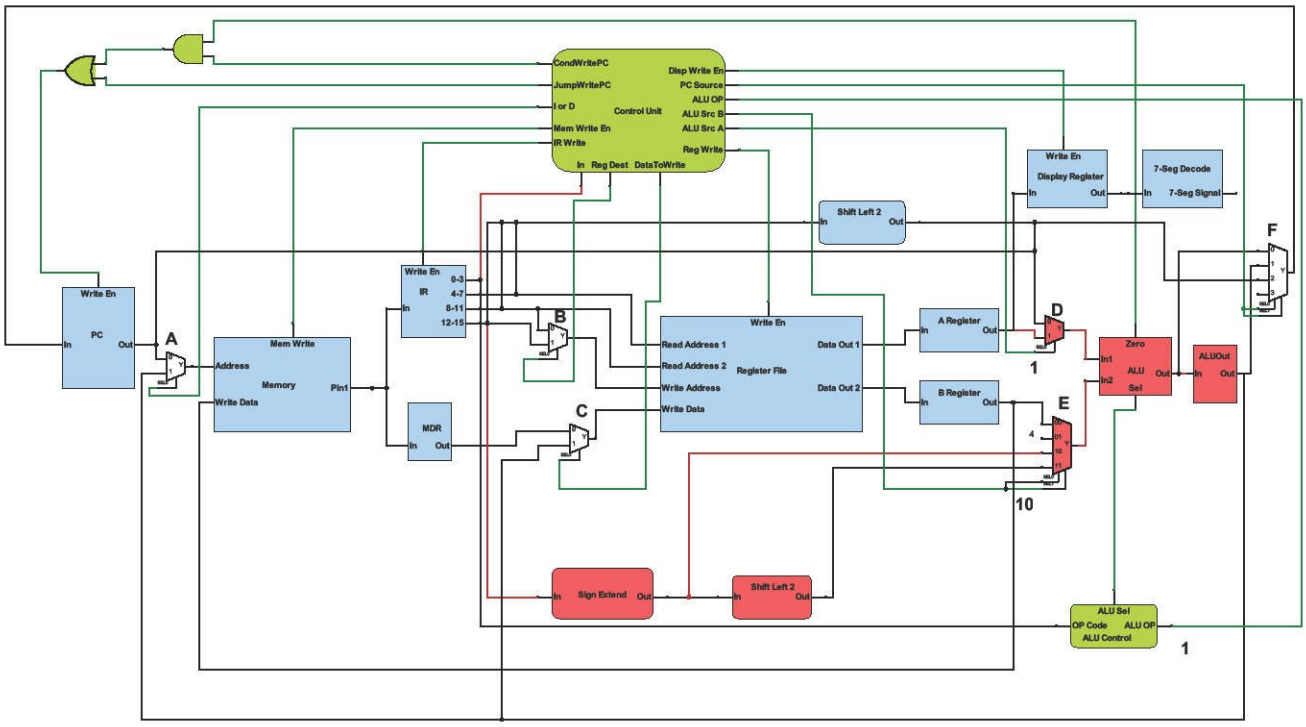

Figure 9: Execute Stage

For I type instructions, the offset is taken from the instruction and is sign extended then an ALU operation is performed on and the value stored in the A register. Which operation is performed depends on the op-code of the instruction. The ALU control takes the op-code and the ALUOP and determines which select lines to send to the ALU. The result is stored in ALUOut.

5.5.2 R Type Instructions

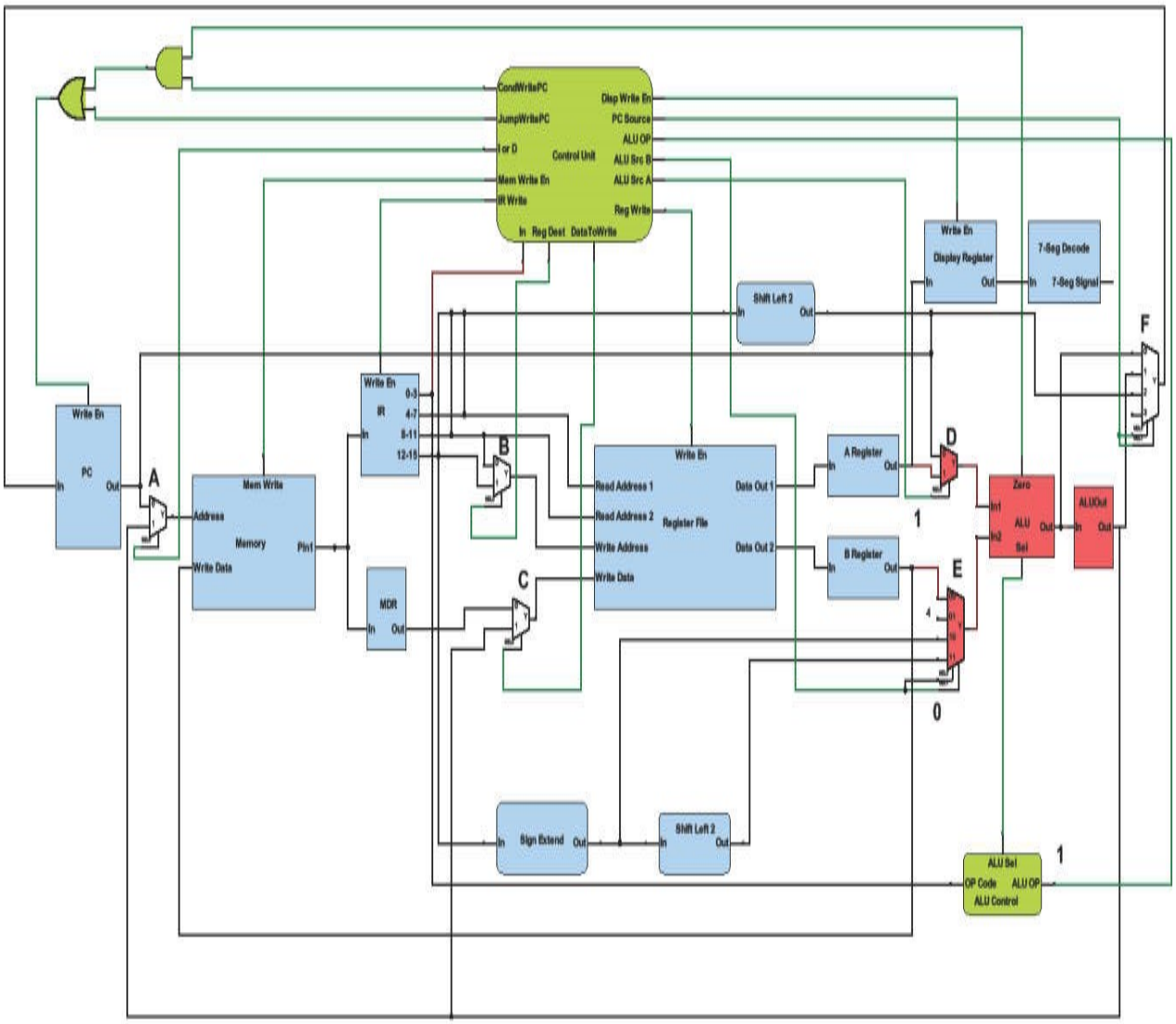

Figure 10: R Type Instructions

For R type instructions, the two operands are taken from register A and B and then are sent to and used in the ALU operation. Once again, the ALU operation performed depends on the OP-Code supplied by the instruction. The result is then stored in the ALUOut register.

5.5.3 J Type Instruction

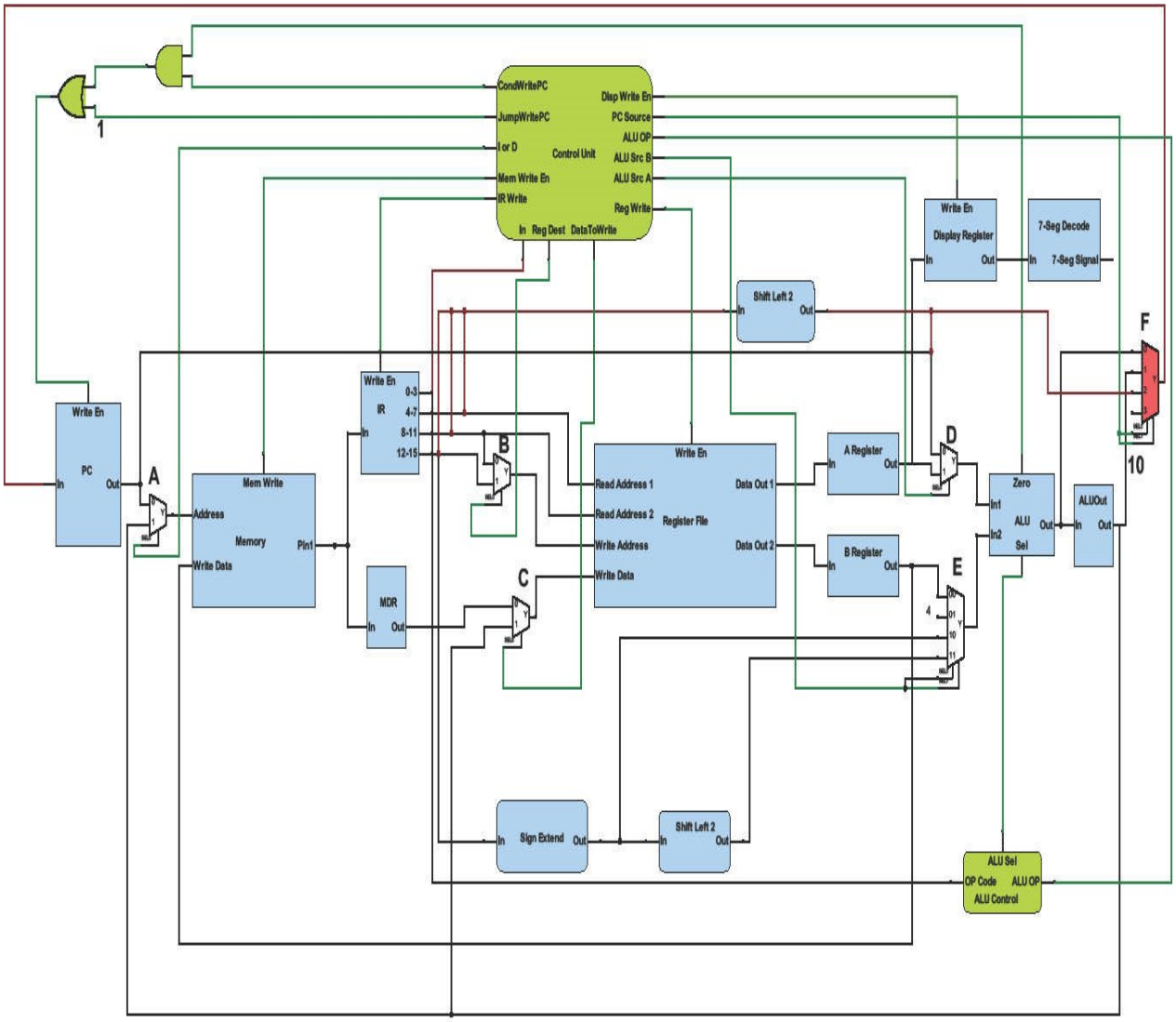

Figure 11: J Type Instruction

For J type instructions, bits 4 through 15 are taken from the instruction register and two zeroes are added to the end (shift left 2). The value is then sent to MUX F, where it is then sent to data in for the PC Register.

5.5.4 Branching Instructions

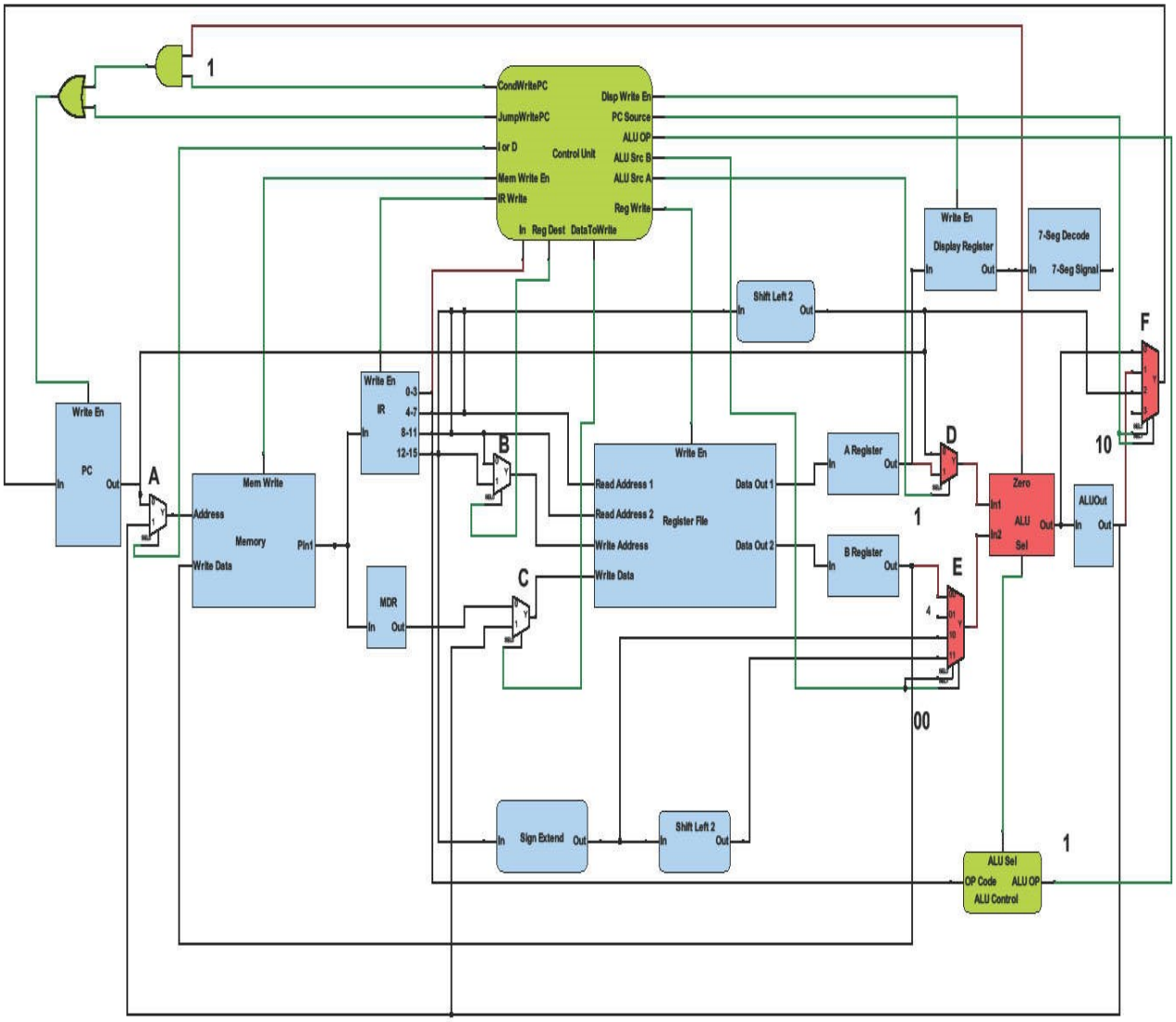

Figure 12: Branching Instructions

For conditional branching instructions, the two operands are subtracted in the ALU, the combinational logic receive the zero flag from the ALU as well as the control signal PC Write Conditional, and the value stored in ALUOut is sent to MUX F where it will be passed to the PC register dataIn.

5.5.5 S Type Instructions

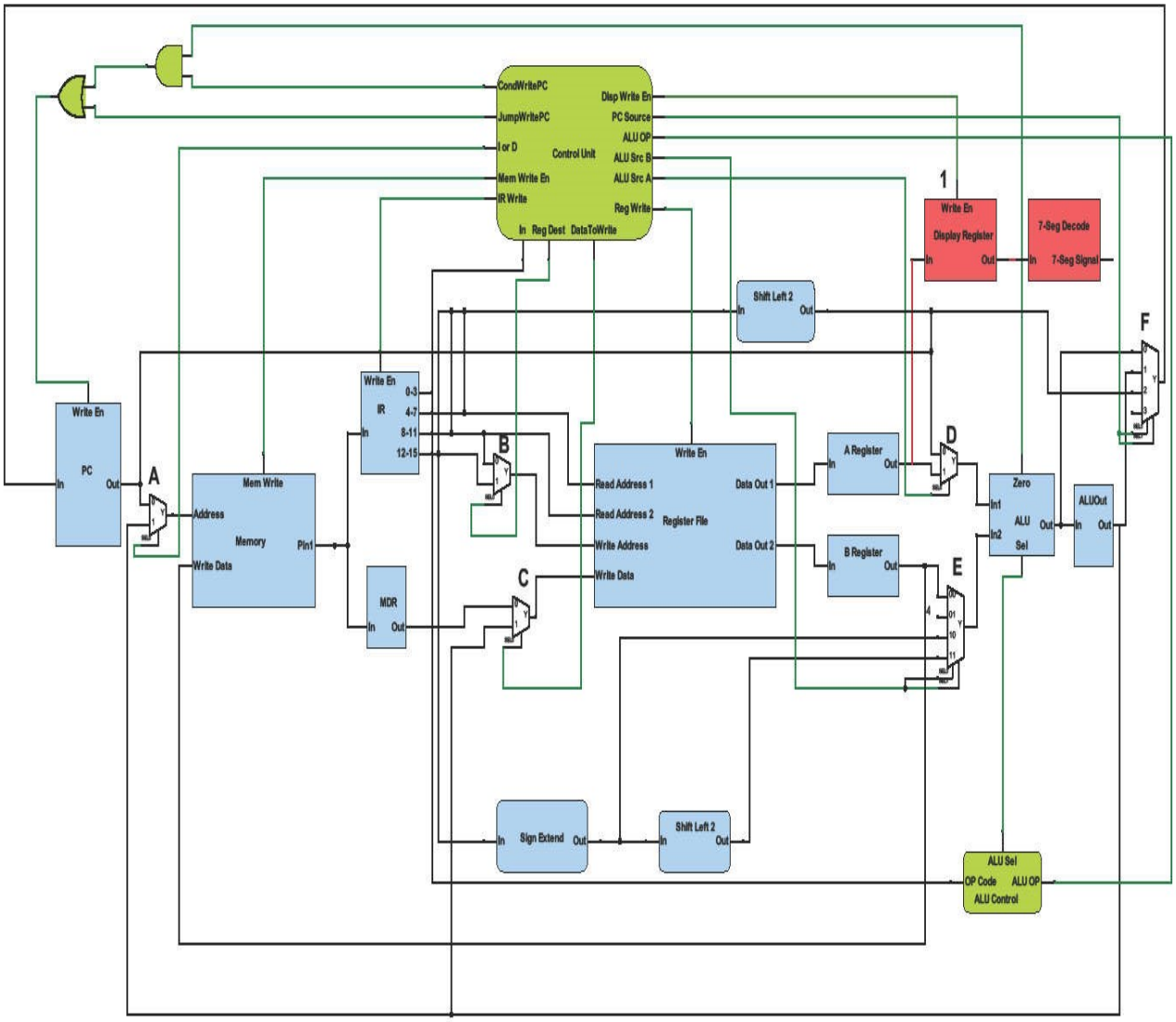

Figure 13: S Type Instructions

Display register receives a write enable from the control unit, taking the value currently stored in register A and storing it in the Display register. The 7-segment decoder then updates the display accordingly.

5.6 Memory Access Stage

In this stage, the memory is accessed and is either read or written to based on which instruction is being executed (load or store).

5.6.1 LW Instructions

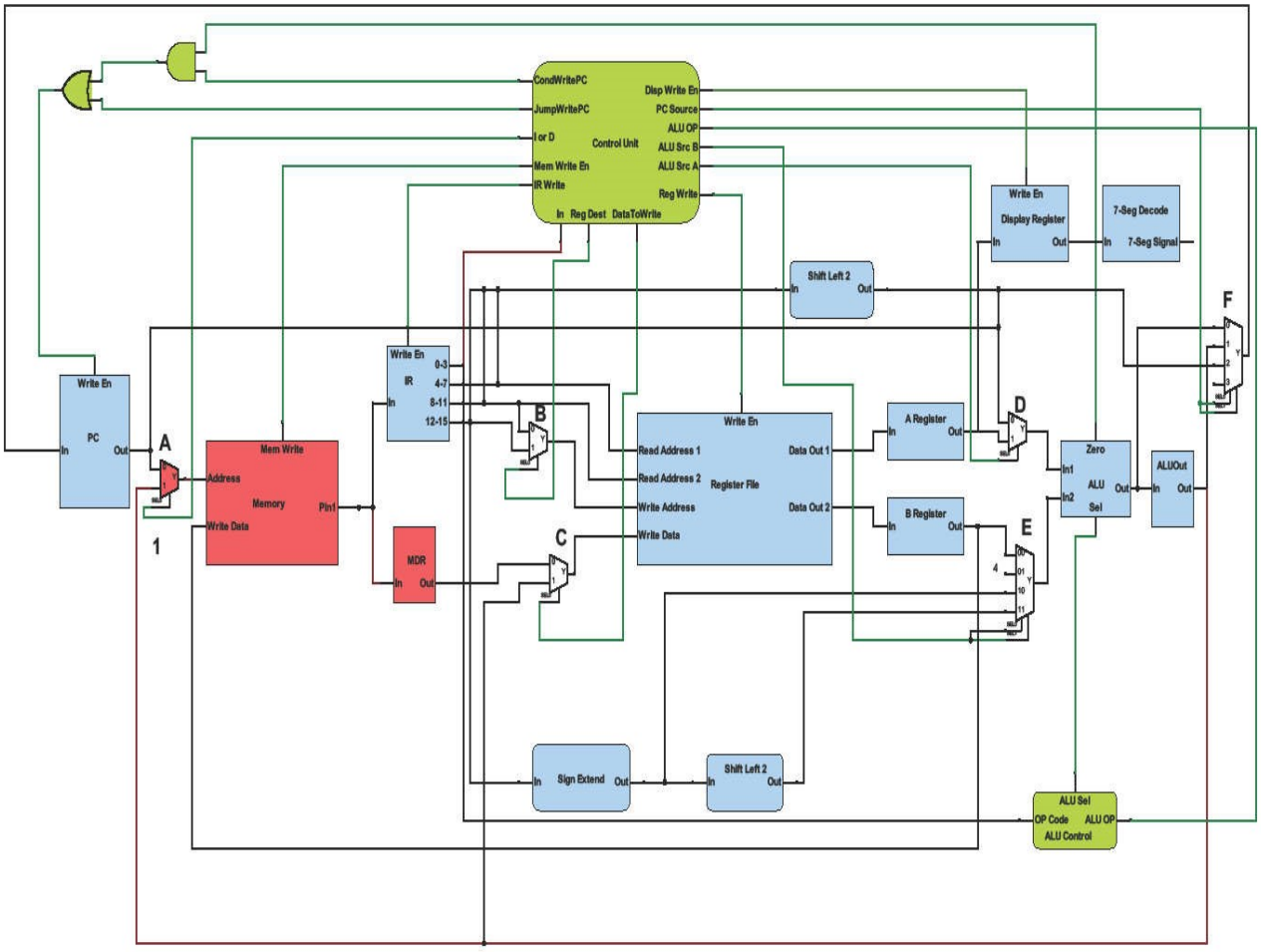

Figure 14: LW Instructions

Mux A receives data from ALUOut, then proceeds to select it and send it as the address for accessing memory. Memory then sends the data at the given address to the Memory Data Register.

5.6.2 SW Instructions

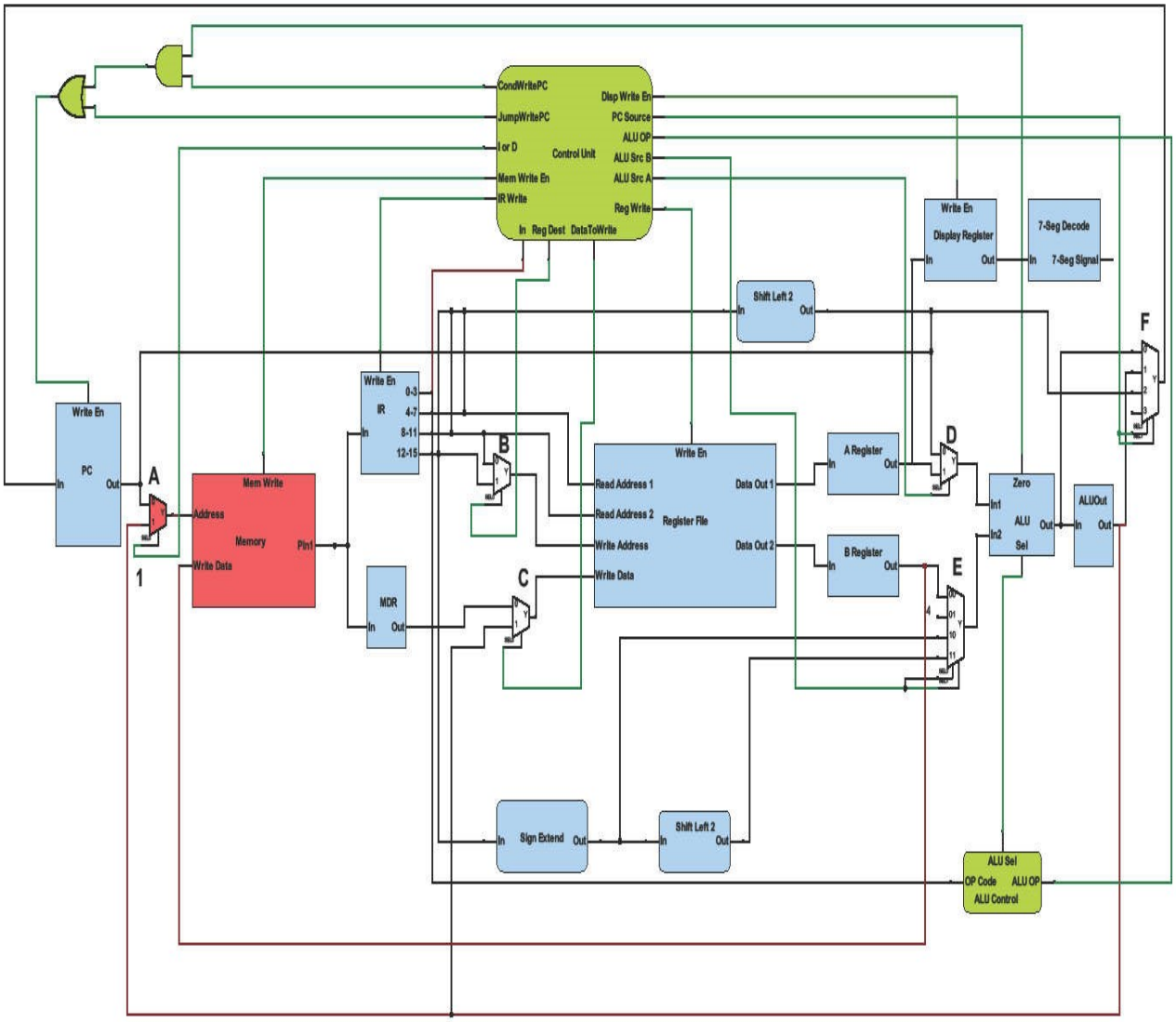

Figure 15: SW Instructions

Mux A receives data from ALUOut while memory receives the data from Register B as Write Data. Memory then stores given data at given address.

5.7 WriteBack Stage

In the writeback stage, the register file is written to after the value to be written was found by the ALU.

5.7.1 R and I Type WriteBack

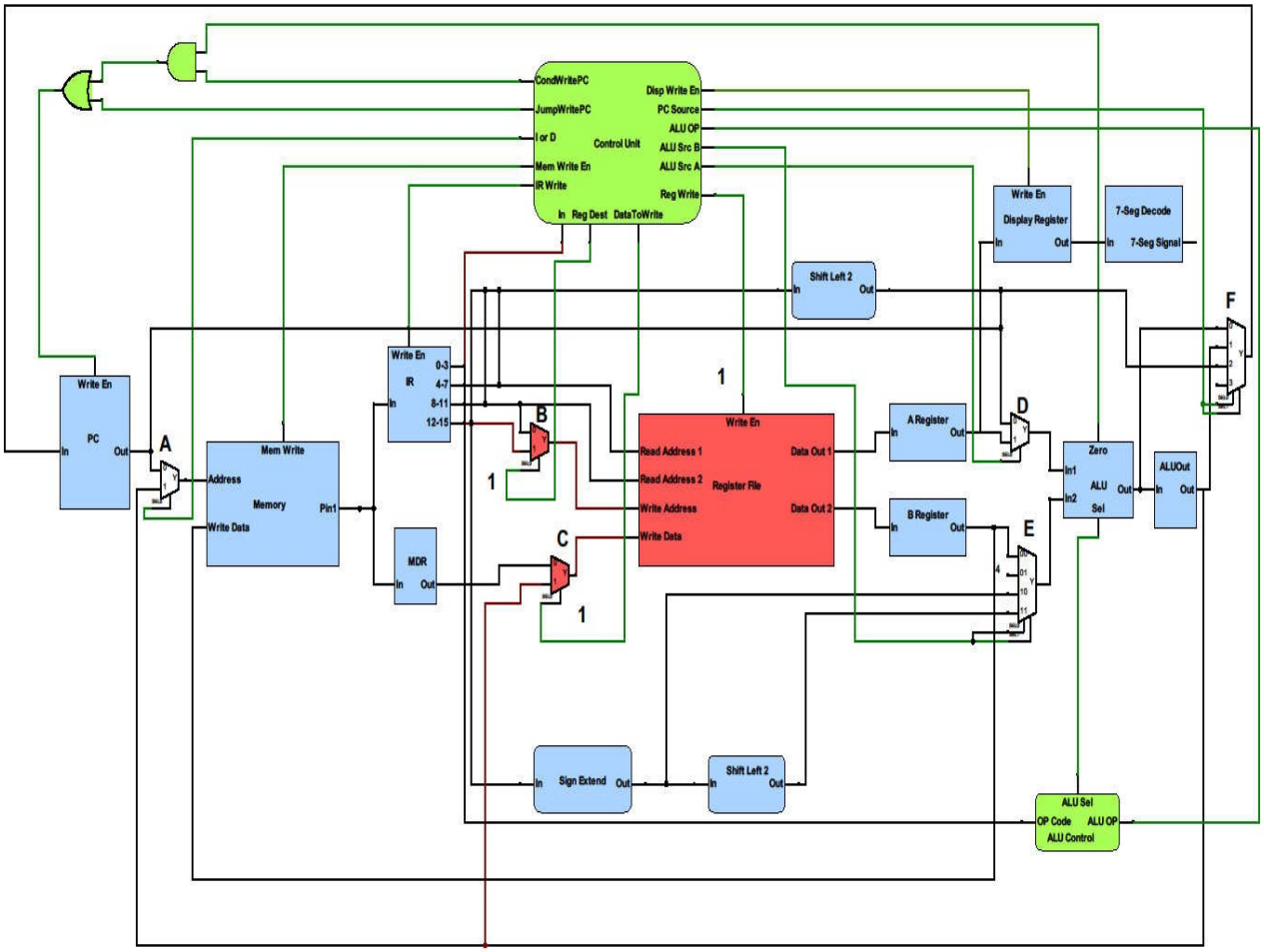

Figure 16: R and I Type WriteBack

Mux C receives the result from ALUOut and sends it as Write Data to the Register File. Also, Mux B receives the address from bits 13-15 from the IR register and proceeds to send it to the Register File as a write Address. The given data is then stored at the given address.

5.7.2 LW WriteBack

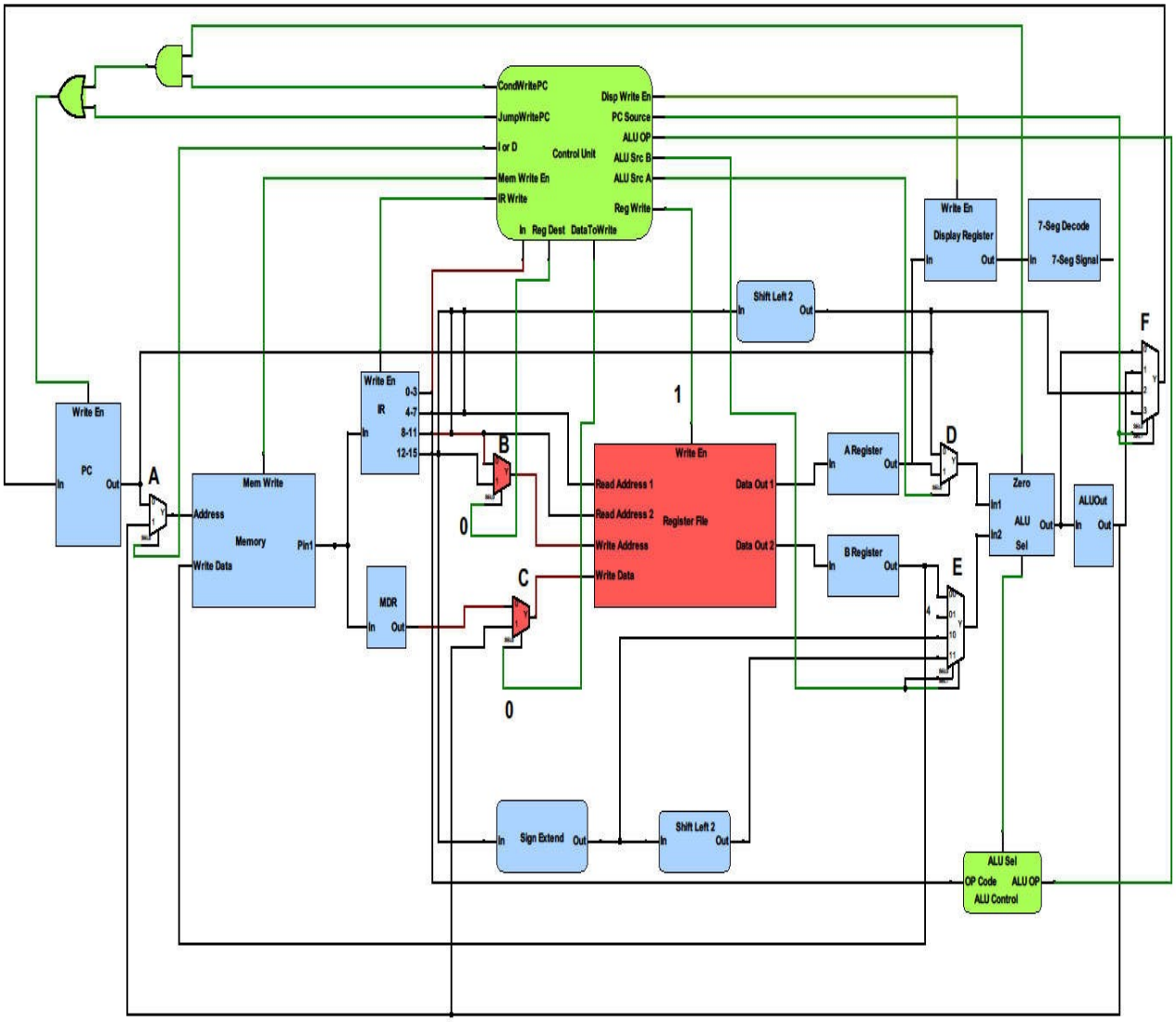

Figure 17: LW WriteBack

Mux C receives the result from Memory Data Register and sends it as Write Data to the Register File. Also, Mux B receives the address from bits 8-11 from the IR register and proceeds to send it to the Register File as a write Address. The given data is then stored at the given address.

5.8 Multicycle Signal Table

Table 6: Multicycle Signal Table

Type	Memory Location	Microprogram Control	Cond Write PC	Jump Write PC	I or D	Mem Write EN	IR Write	Reg Dest	Data To Write	Reg Write	ALU SrcA	ALU SrcB	AL UOP	PC Source	Disp Write En
Fetch	0	10	0	1	0	0	1	0	00	0	0	01	0	00	0
Decode	1	00	0	0	0	0	0	0	00	0	0	11	0	00	0
Display	2	01	0	0	0	0	0	0	00	0	0	00	1	00	1
J	3	01	0	1	0	0	0	0	00	0	0	00	1	10	0
Branch	4	01	1	0	0	0	0	0	00	0	1	00	1	10	0
R Type	5	10	0	0	0	0	0	0	00	0	1	00	1	00	0
	6	01	0	0	0	0	0	1	01	1	0	00	1	00	0
sw	7	10	0	0	0	0	0	0	00	0	1	10	1	00	0
	8	01	0	0	1	1	0	0	00	0	0	00	1	00	0
lw	9	10	0	0	0	0	0	0	00	0	1	10	1	00	0
	10	10	0	0	1	0	0	0	00	0	0	00	1	00	0
	11	01	0	0	0	0	0	0	00	1	0	00	1	00	0
I type	12	10	0	0	0	0	0	0	00	0	1	10	1	00	0
	13	01	0	0	0	0	0	1	01	1	0	00	1	00	0
Switches	14	01	0	0	0	0	0	0	10	1	0	00	1	00	0

6 Pipelining

6.1 Design Overview

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in automobile manufacturing. The first station in an assembly line may prepare the automobile chassis, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one automobile, another group is fitting a body on the chassis of a second automobile, and yet another group is preparing a new chassis for a third automobile. Although it may take hours or days to complete one automobile, the assembly-line operation makes it possible to have a new automobile rolling off the end of the assembly line every few minutes.

6.2 Basic Concept

The five-stage processor organization in Figure 5 and the corresponding datapath in Figure 4 allow instructions to be fetched and executed one at a time. It takes five clock cycles to complete the execution of each instruction. Rather than wait until each instruction is completed, instructions can be fetched and executed in a pipelined manner. The five stages corresponding to those in Figure 4 are labeled as Fetch, Decode, Compute, Memory, and Write. Instruction Ij is fetched in the first cycle and moves through the remaining stages in the following cycles. In the second cycle, instruction Ij+1 is fetched while instruction Ij is in the Decode stage where its operands are also read from the register file. In their third cycle, instruction Ij+2 is fetched while instruction Ij+1 is in the Decode stage and instruction Ij is in the Compute stage where an arithmetic or logic operation is performed on its operands. Ideally, this overlapping pattern of execution would be possible for all instructions. Although any one instruction takes five cycles to complete its execution, instructions are completed at the rate of one per cycle.

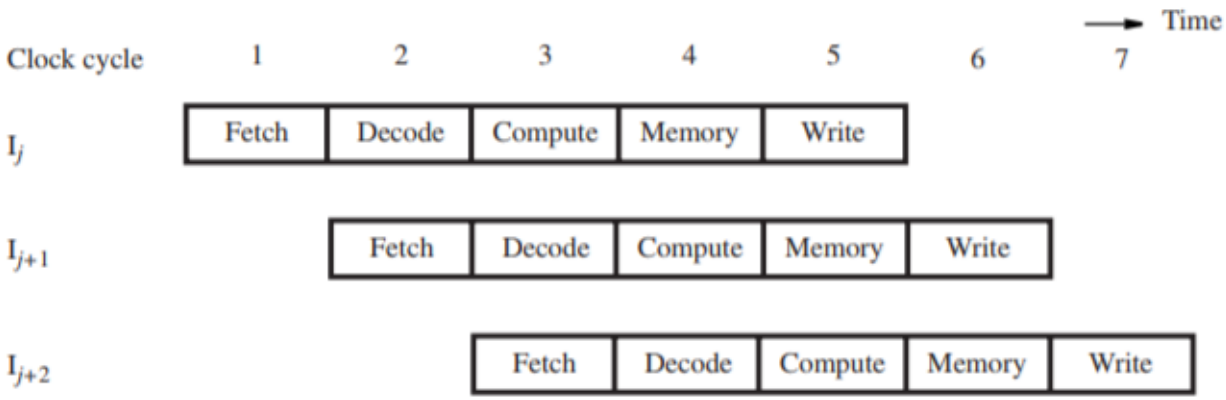

Figure 18: Pipelined exection - The Ideal Case

6.3 Pipeline Organization

Figure 19 indicates how the five-stage organization in Figures 4 and 5 can be pipelined. In the first stage of the pipeline, the program counter (PC) is used to fetch a new instruction. As other instructions are fetched, execution proceeds through successive stages. At any given time, each stage of the pipeline is processing a different instruction. Information such as register addresses, immediate data, and the operations to be performed must be carried through the pipeline as each instruction proceeds from one stage to the next. This information is held in interstage buffers, the IR and PC-Temp registers, and additional storage. The interstage buffers are used as follows:

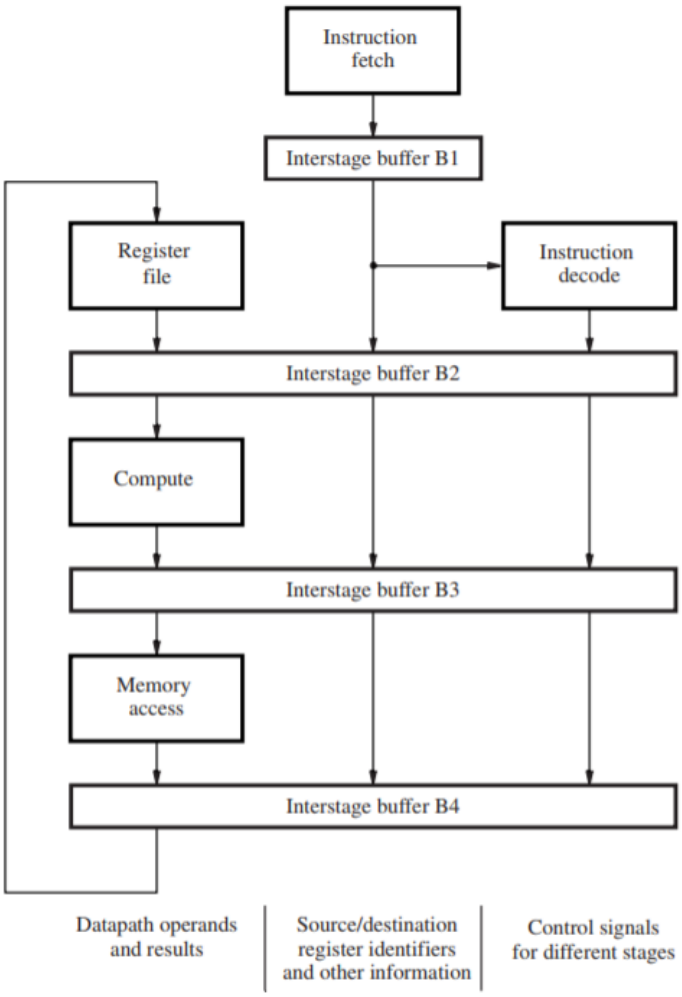

Figure 19: Pipelined exection - Five Stage Pipelining

- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
- Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder. The settings for control signals move through the pipeline to determine the ALU operation, the memory operation, and a possible write into the register file.
- Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage. In the case of a write access to memory, buffer B3 holds the data to be written. These data were read from the register file in the Decode stage. The buffer also holds the incremented PC value passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction.
- Interstage buffer B4 feeds the Write stage with a value to be written into the register file. This value may be the ALU result from the Compute stage, the result of the Memory access stage, or the incremented PC value that is used as the return address for a subroutine-call instruction.

7 Conclusion

The phase two report describes the steps taken before building the multicycle Architecture project. Phase one plans the approach to building the Computer Organization and Architecture course's final project. After phase one have been completed, the group is ready to jump into phase 2 and design the project. Phase 2 will mainly include interfacing the modules created In the labs, pipelining the project to achieve faster speed, and optimizing the project to present to the professor and the teacher assistant of the course.

References

[1] M. Hassan, *ENGG 3380: Computer Organization and Design Laboratory Manual)*, University of Guelph, 2019.