



**Faculty of Engineering and Technology
Electrical and Computer Engineering Department**

**Computer Architecture
ENCS4370**

Project #2 report
Multicycle Processor Implementation

Prepared by:

Klarein Wassaya **1210279**

Fatima Dawabsheh **1210827**

Instructor: Ayman Hroub

Instructor: Aziz Qaroush

Section: 1, 2

Date: June 18, 2024

Abstract

The aim of this project is to design a specialized multicycle computer processor that operates on 16-bit instructions. The processor is designed with a 5-stage consisting of fetch, decode, execute (ALU), memory access, and write-back stages. The architecture supports four instruction types: R-type, I-type, J-type, and S-type. The instruction set architecture (ISA) includes basic arithmetic, logic, load/store operations, and conditional branching.

This project provides a comprehensive approach to designing a multicycle processor, covering both the Datapath and control path. The design is validated through testing to ensure the correct execution of the defined instruction set.

Table of Contents

Abstract.....	I
Table of Contents	II
Table of Figures	III
Table of Tables	IV
1. Design Specifications	1
1.1 Motivation	1
1.2 Instruction Formats	2
1.1.1 R-Type.....	2
1.1.2 I-Type.....	2
1.1.3 J-Type.....	2
1.1.4 S-Type	2
1.3 Instruction Set	3
2. RTL Design	4
3. Components of Datapath.....	7
3.1 Multiplexer	7
3.2 Instruction Memory	8
3.3 Data Memory	9
3.4 Register File.....	10
3.5 Arithmetic Logic Unit	11
4. Control Path	12
4.1 PC control unit	12
4.2 ALU control unit	13
4.3 Main control unit	14
5. Full Datapath	16
6. Testing	17
Load instructions (LW, LBU, LBs)	18
R-type, I-type (branch instructions), JMP.....	19
SW, Sv	20
Call, Ret	21

Table of Figures

<i>Figure 1 Example of multiplexer use in the Datapath</i>	<i>7</i>
<i>Figure 2 Instruction Memory</i>	<i>8</i>
<i>Figure 3: Data Memory</i>	<i>9</i>
<i>Figure 4: Register File</i>	<i>10</i>
<i>Figure 5: ALU symbol</i>	<i>11</i>
<i>Figure 6: Internal Structure of ALU</i>	<i>11</i>
<i>Figure 7 Full Datapath</i>	<i>16</i>
<i>Figure 8 Load instructions testing</i>	<i>18</i>
<i>Figure 9 R-type, I-type (branch instructions), JMP testing</i>	<i>19</i>
<i>Figure 10 SW, Sv instructions testing.....</i>	<i>20</i>
<i>Figure 11 Call, Ret instructions testing.....</i>	<i>21</i>

Table of Tables

Table 1: Instruction Set	3
Table 2: PC control unit table	12
Table 3: ALU control unit table	13
Table 4: Main control unit table	14
Table 5 Load instructions testing code	18
Table 6 R-type, I-type (branch instructions), JMP testing code	19
Table 7 SW, Sv instructions testing code	20
Table 8 Call, Ret instructions testing code	21

1. Design Specifications

1.1 Motivation

This project is inspired by the MIPS architecture, this design is part of the RISC architecture. Our goal is to create a set of instructions for a Multicycle processor. The key features of this instruction set include:

1. Each instruction in the set is 16 bits in length.
2. The design includes 8 16-bit general-purpose registers, labeled from R0 to R7.
3. Program Counter (PC) is a 16-bit special-purpose register.
4. R0 is hardwired to zero.
5. Byte addressable memory
6. Little endian byte ordering
7. four distinct instruction types – R-type, I-type, J-type, and S-type.
8. The processor is designed with two separate physical memory units: one for storing instructions and the other for data.
9. Arithmetic Logic Unit is also used to decide the outcomes of certain instructions, like whether to take a branch or not.

1.2 Instruction Formats

The instruction set has the following instruction formats.

1.1.1 R-Type

Opcode (4 bits)	Rd (3 bits)	Rs1 (3 bits)	Rs2 (3 bits)	Unused (3 bits)
-----------------	-------------	--------------	--------------	-----------------

Where:

Opcode: specifies the type of the operation that should be performed by the processor.

Rd: is the destination register where the result of an operation should be stored.

Rs1: is the register of the first operand.

Rs2: is the register of the second operand.

1.1.2 I-Type

Opcode (4 bits)	M (1 bit)	Rd (3 bits)	Rs1 (3 bits)	Immediate (5 bits)
-----------------	-----------	-------------	--------------	--------------------

Where:

Opcode: specifies the type of the operation that should be performed by the processor.

M: is the mode bit, used with load (1 → LBs (signed extension), 0 → LBU (zero extension)) and branch (1 → compare Rd with R0, 0 → compare Rd with Rs1) instructions.

Rd: is the destination register where the result of an operation should be stored.

Rs1: is the register of the first operand.

1.1.3 J-Type

The following format is used for three instructions: Jump, Call and Return, except that in the return instruction, the offset part is not used, since in the return instruction the next PC gets its value from the register file from the seventh register R7, this register has its value stored from the call instruction.

Opcode (4 bits)	Jump Offset (12 bits)
-----------------	-----------------------

1.1.4 S-Type

This format supports only one instruction, Sv, this instruction takes the immediate value and stores it in the data memory in Rs location [# M[rs] = imm].

Opcoe (4 bits)	Rs (3 bits)	Immediate (9 bits)
----------------	-------------	--------------------

1.3 Instruction Set

The following table shows the specific instructions that are implemented in the processor design.

Table 1: Instruction Set

<i>R – Type</i>		
<i>AND</i>	Reg[Rd] = Reg[Rs1] & Reg[Rs2]	0000
<i>ADD</i>	Reg[Rd] = Reg[Rs1] + Reg[Rs2]	0001
<i>SUB</i>	Reg[Rd] = Reg[Rs1] - Reg[Rs2]	0010
<i>I – Type</i>		
<i>ADDI</i>	Reg[Rd] = Reg[Rs1] & zero_extended(imm)	0011
<i>ANDI</i>	Reg[Rd] = Reg[Rs1] + sign_extended(imm)	0100
<i>LW</i>	Reg(Rd) = Mem(Reg(Rs1) + sign_extended(Imm))	0101
<i>LBu</i>	Reg(Rd) = Mem(Reg(Rs1) + zero_extended(Imm))	0110
<i>LBs</i>	Reg(Rd) = Mem(Reg(Rs1) + sign_extended(Imm))	
<i>SW</i>	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111
<i>BGT</i>	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000
<i>BGTZ</i>	if (Reg(Rd) > Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	
<i>BLT</i>	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001
<i>BLTZ</i>	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	
<i>BEQ</i>	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010
<i>BEQZ</i>	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	
<i>BNE</i>	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011
<i>BNEZ</i>	if (Reg(Rd) != Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	
<i>J – Type</i>		
<i>JMP</i>	Next PC = {PC[15:10], Immediate}	1100
<i>CALL</i>	Next PC = {PC[15:10], Immediate} R7 = PC + 4	1101
<i>RET</i>	Next PC = r7	1110
<i>S – Type</i>		
<i>Sv</i>	M[rs] = imm	1111

2. RTL Design

R-type (AND, ADD, SUB)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Reg}(\text{RS2})$
Execute operation: $\text{ALU_result} \leftarrow \text{func}(\text{BusA}, \text{BusB})$
Write ALU result: $\text{Reg}(\text{Rd}) \leftarrow \text{ALU_result}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

I-TYPE (ADDI & ANDI)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Extend}(\text{imm5})$
Execute operation: $\text{ALU_result} \leftarrow \text{op}(\text{BusA}, \text{BusB})$
Write ALU result: $\text{Reg}(\text{Rd}) \leftarrow \text{ALU_result}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

LW (Load Word From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Fetch base register: $\text{base} \leftarrow \text{Reg}(\text{RS1})$
Calculate address: $\text{address} \leftarrow \text{base} + \text{sign_extend}(\text{imm5})$
Read memory: $\text{data} \leftarrow \text{MEM}[\text{address}+1, \text{address}]$
Write register : $\text{Reg}(\text{Rd}) \leftarrow \text{data}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

LBs (Load Byte From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Fetch base register: $\text{base} \leftarrow \text{Reg}(\text{RS1})$
Calculate address: $\text{address} \leftarrow \text{base} + \text{sign_extend}(\text{imm5})$
Read memory: $\text{data} \leftarrow \text{MEM}[\text{address}]$
Write register : $\text{Reg}(\text{Rd}) \leftarrow \text{data}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

LBu (Load Byte From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Fetch base register: $\text{base} \leftarrow \text{Reg}(\text{RS1})$
Calculate address: $\text{address} \leftarrow \text{base} + \text{unsign_extend}(\text{imm5})$
Read memory: $\text{data} \leftarrow \text{MEM}[\text{address}]$
Write register : $\text{Reg}(\text{Rd}) \leftarrow \text{data}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

SW (Store From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Fetch registers: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusW} \leftarrow \text{Reg}(\text{Rd})$

Calculate address: $\text{address} \leftarrow \text{BusA} + \text{sign_extend}(\text{imm5})$

Write memory: $\text{MEM}[\text{address}] \leftarrow \text{BusW}$

Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

BGT and BGTZ (From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Reg}(\text{Rd})$

Greater: $\text{NEG} \leftarrow \text{subtract}(\text{BusA}, \text{BusB})[15], \text{ZERO} \leftarrow (\text{subtract}(\text{BusA}, \text{BusB}) == 0)$

Branch: if $(\text{NEG} \&\& \text{ZERO}) \text{ PC} \leftarrow \text{PC} + \text{sign_extend}(\text{offset5})$
else $\text{PC} \leftarrow \text{PC} + 2$

BLT and BLTZ (From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Reg}(\text{Rd})$

LESS: $\text{NEG} \leftarrow \text{subtract}(\text{BusA}, \text{BusB})$

Branch: if $(\text{NEG}) \text{ PC} \leftarrow \text{PC} + \text{sign_extend}(\text{offset5})$
else $\text{PC} \leftarrow \text{PC} + 2$

BEQ and BEQZ (From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Reg}(\text{Rd})$

Equality: $\text{zero} \leftarrow \text{subtract}(\text{BusA}, \text{BusB})$

Branch: if $(\text{zero}) \text{ PC} \leftarrow \text{PC} + \text{sign_extend}(\text{offset5})$
else $\text{PC} \leftarrow \text{PC} + 2$

BNE and BNEZ (From I-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Fetch operands: $\text{BusA} \leftarrow \text{Reg}(\text{RS1}), \text{BusB} \leftarrow \text{Reg}(\text{Rd})$

Not Equality: $\text{zero} \leftarrow \text{subtract}(\text{BusA}, \text{BusB})$

Branch: if $(\text{zero}) \text{ PC} \leftarrow \text{PC} + \text{sign_ext}(\text{offset5})$
else $\text{PC} \leftarrow \text{PC} + 2$

JMP (From J-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$

Target PC address: $\text{target} \leftarrow \text{PC}[15:12] \parallel \text{Immediate12}$

Jump: $\text{PC} \leftarrow \text{target}$

CALL (From J-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Target PC address: $\text{target} \leftarrow \text{PC}[15:12] \parallel \text{Immediate}_{12}$
Jump: $\text{PC} \leftarrow \text{target}$
Write return address: $\text{Reg}(\text{R7}) \leftarrow \text{PC} + 4$

RET (From J-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Target PC address: $\text{target} \leftarrow \text{Reg}(\text{R7})$
Jump: $\text{PC} \leftarrow \text{target}$

SV (From S-type)

Fetch instruction: $\text{Instruction} \leftarrow \text{MEM}[\text{PC}+1, \text{PC}]$
Calculate address: $\text{address} \leftarrow \text{extended}(\text{rs})$
Write memory: $\text{MEM}[\text{address}] \leftarrow \text{imm}$
Next PC address: $\text{PC} \leftarrow \text{PC} + 2$

3. Components of Datapath

For implementing a microprocessor supporting the previous instruction set, the following components are required to have a complete design:

3.1 Multiplexer

In this design, three types of multiplexers were required, 2X1_MUX, 3X1_MUX and 4X1_MUX, some of the modules were designed for 16 bits inputs and output while other modules were designed for 3 bits inputs and output. The multiplexer is very important in the design, it works as a selector to determine the correct value to be used in the correct time, avoiding any unexpected assignments to the registers during the process.

Separate modules for 2X1_MUX, 3X1_MUX and 4X1_MUX was implemented avoiding any unexpected values either for the selection input or for the values of the inputs.

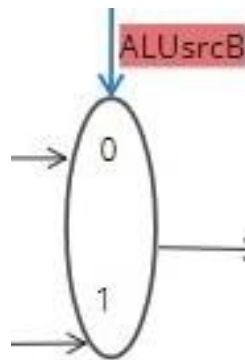


Figure 1 Example of multiplexer use in the Datapath

For example, the multiplexer in the previous figure is used to determine the input to sourceB of the ALU, due to the value of the selection input (ALUsrcB control signal) as follows:

- When ALUsrcB is set to 0, then the value that is passed to sourceB of the ALU is the value taken from the register file that is related to R-type instructions.
- When ALUsrcB is set to 1, then the value that is passed to sourceB of the ALU is the extended immediate value that is related to I-type instructions.

3.2 Instruction Memory

In this processor implementation, memory has been partitioned into two distinct segments: Instruction Memory and Data Memory. This division is intended to avoid conflicts that may arise due to simultaneous operations such as fetching instructions and accessing or modifying data.

The first part to be included in the design is the instruction memory, this memory is byte addressable, then each location in the memory stores one-byte (8 bits) values, in this case the instruction is taken from the input address and the address next to it. Since little endian byte ordering is required, then the value stored in the input address represents the least significant bits, while the value stored in the address next to it represents the most significant bits.

The instruction memory is implemented as a module with PC as input to the address and the instruction as an output as follows:

```
instruction = {memory[address + 1], memory[address]};
```

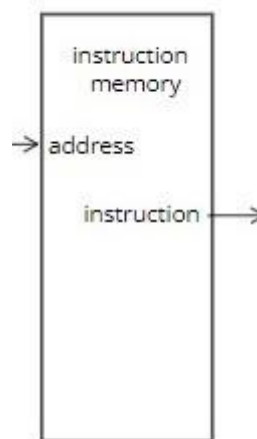


Figure 2 Instruction Memory

3.3 Data Memory

The data memory serves as the storage repository for the processor's data. It interfaces with the rest of the Datapath through an address input and two data lines: one for storing data and another for retrieving data.

Two control signals are connected to this memory to control the reading and writing operations. Here's a breakdown of how it works:

- When the R_control signal is active (1), the data memory retrieves data from the specified address and outputs it. This facilitates data reading.
- When the W_control signal is active (1), the data memory accepts data from the input line and stores it at the specified address. This facilitates data writing.

There are three cases for R_control being active:

- R_control = 1

This case for LW instruction, such that the data out is 2 bytes as the following:

```
data_out = {memory[address + 1], memory[address]};
```

- R_control = 2

This case for LBU instruction, such that the data out is 2 bytes as the following:

```
data_out = {8'b00000000, memory[address]};
```

- R_control = 3

This case for LBU instruction, such that the data out is 2 bytes as the following:

```
data_out = {{(8){memory[address][7]}} ,memory[address]};
```

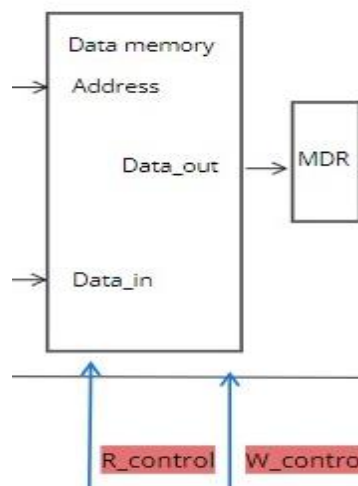


Figure 3: Data Memory

3.4 Register File

The register file is a crucial component, including several registers that serve as temporary storage for data being transferred between the processor's memory and its operational units. To read from the registers, the processor uses two designated registers, Rs1 and Rs2. The data from RA is sent to BusA, and data from RA is sent to BusB. The register labeled RW in the register file is the designated location for writing data; when the control signal RegWrite is set to 1, it indicates that data arriving on BusW would be stored into RW.

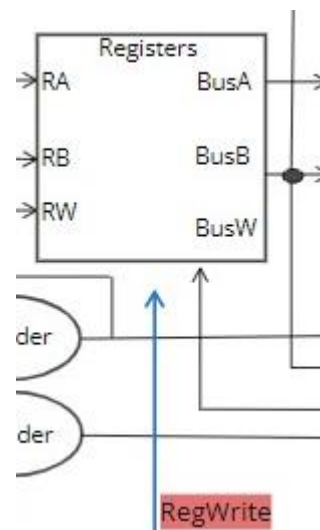


Figure 4: Register File

RA, RB, RW receive their values from Rs1, Rs2, Rd from the instruction respectively.

3.5 Arithmetic Logic Unit

The ALU (Arithmetic Logic Unit) is a critical component of the CPU responsible for executing arithmetic and logic operations, such as addition, subtraction, and logical comparisons.

In our Datapath, the ALU has two primary inputs, each consisting of 16 bits. The ALU can perform various operations on these bits, including logical AND, addition, and two forms of subtraction. A 2-bit control signal determines the operation to be performed. This signal directs a 4-to-1 multiplexer (Mux) to select the appropriate arithmetic or logic operation. The selected result is then outputted from the ALU as the final result.

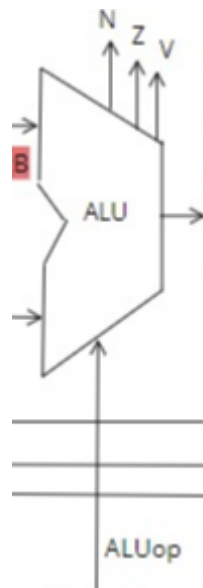


Figure 5: ALU symbol

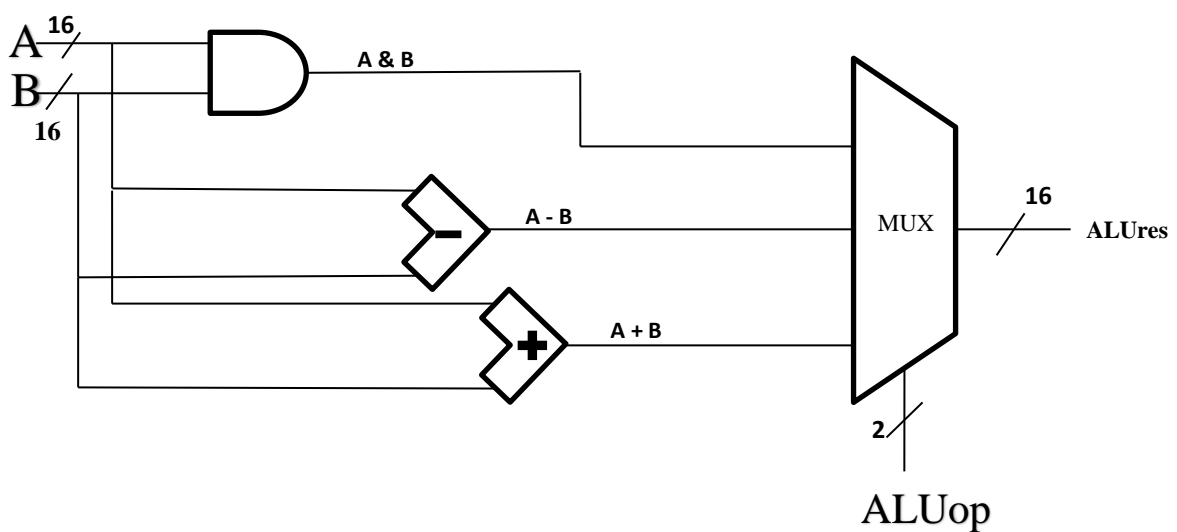


Figure 6: Internal Structure of ALU

4. Control Path

4.1 PC control unit

Table 2: PC control unit table

Op	Zero flag (Z)	Negative flag (N)	Overflow flag (V)	PCsrc
R-type	X	X	X	0 = Increment PC
JMP	X	X	X	2 = Jump Target Address
CALL	X	X	X	2 = Jump Target Address
RET	X	X	X	3 = Return Address
BEQ	0	X	X	0 = Increment PC
BEQ	1	X	X	1 = Branch Target Address
BNE	0	X	X	1 = Branch Target Address
BNE	1	X	X	0 = Increment PC
BGT	0	0	0	1 = Branch Target Address
BGT	0	1	1	1 = Branch Target Address
BGT	1	X	X	0 = Increment PC
BLT	0	0	1	1 = Branch Target Address
BLT	0	1	0	1 = Branch Target Address
BLT	1	X	X	0 = Increment PC
Other	X	X	X	0 = Increment PC

The PC Control Unit determines the value of the PCsrc control signal based on the flag bits (zero Z, negative N and overflow V). For branch instructions; 'Branch if Greater Than' (BGT) needs Z=0 and N=V to set the PCsrc signal to 1. 'Branch if Equal' (BEQ) requires Z=1 to set the PCsrc signal to 1. 'Branch if Not Equal' (BNE) looks for Z=0 in order to set the PCsrc signal to 1. 'Branch if Less Than' (BLT) checks $N \neq V$, if $N \neq V$ and the zero flag Z is 0 then the PCsrc is set to 1. Moreover, this control unit set the PCsrc to 2 for JMP and CALL instructions, as well as setting it to 3 for RET instruction, for other instructions the default value for PCsrc is 0, indicating that the PC will increment by 2 in order to fetch the next instruction normally.

The Boolean Expression for the branch instructions is as follows:

$$\text{Branch} = (\text{BEQ}. z) + (\text{BNE}. z') + (\text{BGT}. z'. (N \oplus V)') + (\text{BLT}. (N \oplus V))$$

4.2 ALU control unit

Table 3: ALU control unit table

<i>Instruction</i>	<i>Opcode</i>	<i>4-bits coding</i>	<i>Function</i>	<i>ALUop</i>
<i>AND</i>	AND	0000	Bitwise AND	00
<i>ADD</i>	ADD	0001	Arithmetic ADD	10
<i>SUB</i>	SUB	0010	Arithmetic SUB	01
<i>ADDI</i>	ADDI	0011	Arithmetic ADD	10
<i>ANDI</i>	ANDI	0100	Bitwise AND	00
<i>LW</i>	LW	0101	Arithmetic ADD	10
<i>LBu</i>	LBu	0110	Arithmetic ADD	10
<i>LBs</i>	LBs	0110	Arithmetic ADD	10
<i>SW</i>	SW	0111	Arithmetic ADD	10
<i>BGT</i>	BGT	1000	Arithmetic SUB	01
<i>BGTZ</i>	BGTZ	1000	Arithmetic SUB	01
<i>BLT</i>	BLT	1001	Arithmetic SUB	01
<i>BLTZ</i>	BLTZ	1001	Arithmetic SUB	01
<i>BEQ</i>	BEQ	1010	Arithmetic SUB	01
<i>BEQZ</i>	BEQZ	1010	Arithmetic SUB	01
<i>BNE</i>	BNE	1011	Arithmetic SUB	01
<i>BNEZ</i>	BNEZ	1011	Arithmetic SUB	01
<i>JMP</i>	JMP	1100	X	11
<i>CALL</i>	CALL	1101	X	11
<i>RET</i>	RET	1110	X	11
<i>Sv</i>	Sv	1111	X	11

This control unit determines the value of the opcode for the ALU (ALUop), the instruction set requires three different operations for the various instructions: ADD, SUB and AND operations. Branch and subtract instructions set the ALUop to “01” indicating that the required operation is SUB in order to set the flag bits in the ALU, while AND and ANDI instructions set the ALUop to “00” since the required operation is Bitwise AND operation in the ALU. For J-type and S-type instructions, no need for ALU operations, ALUop is set to “11” telling the ALU to do nothing. The rest of the instructions requires arithmetic ADD operation, setting the ALUop to “10”.

4.3 Main control unit

Table 4: Main control unit table

	<i>Rs1ctrl</i>	<i>Rs2_src</i>	<i>Rdctrl</i>	<i>ALUsrcB</i>	<i>ExtOp</i>	<i>RegWrite</i>	<i>data_in_src</i>	<i>Data_address_src</i>	<i>R_control</i>	<i>W_control</i>	<i>WBdata</i>
<i>AND</i>	2	0	1	0	X	1 or 0	X	X	0	0	0
<i>ADD</i>	2	0	1	0	X	1 or 0	X	X	0	0	0
<i>SUB</i>	2	0	1	0	X	1 or 0	X	X	0	0	0
<i>ADDI</i>	1	X	0	1	1	1 or 0	X	X	0	0	0
<i>ANDI</i>	1	X	0	1	0	1 or 0	X	X	0	0	0
<i>LW</i>	1	X	0	1	1	1 or 0	X	1	1	0	1
<i>LBu</i>	1	X	0	1	1	1 or 0	X	1	2	0	1
<i>LBs</i>	1	X	0	1	1	1 or 0	X	1	3	0	1
<i>SW</i>	1	2	X	1	1	0	1	1	0	1	X
<i>BGT</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BGTZ</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BLT</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BLTZ</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BEQ</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BEQZ</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BNE</i>	0	2	X	0	1	0	X	X	0	0	X
<i>BNEZ</i>	0	2	X	0	1	0	X	X	0	0	X
<i>JMP</i>	X	X	X	X	X	0	X	X	0	0	X
<i>CALL</i>	X	X	2	X	X	1	X	X	0	0	2
<i>RET</i>	X	1	X	X	X	0	X	X	0	0	X
<i>Sv</i>	X	X	X	X	X	0	0	0	0	1	X

The table provides a comprehensive view of how the main control unit handles different instructions, defining the necessary control signals for register selection, ALU operations, memory access, and write-back processes. This information is crucial for implementing the instruction set architecture (ISA) and ensuring correct operation of the CPU for each instruction type.

- **Rs1ctrl:** Control signal for selecting the first source register (Rs1).
- **Rs2_src:** Control signal for selecting the second source register (Rs2).
- **Rdctrl:** Control signal for selecting the destination register (Rd).
- **ALUsrcB:** Determines the second operand for the ALU (register or immediate).
- **ExtOp:** Specifies if the immediate value should be extended.
- **RegWrite:** Control signal to enable writing to a register.
- **data_in_src:** Source of data to be written to memory or registers.
- **Data_address_src:** Source of the address for memory operations.
- **R_control:** Control signal for reading from memory.
- **W_control:** Control signal for writing to memory.
- **WBdata:** Determines the data to be written back to the register.

In the internal design of the main control unit, a signal called lock_R7 is used to manage the write process to register R7 between CALL and RET instructions. This signal ensures that R7, which holds the return address, is preserved until the RET instruction is executed. During a CALL instruction, R7 is updated with the return address and then locked by lock_R7 to prevent any further changes. This lock remains in place until the RET instruction is reached, ensuring the return address remains intact and allowing the program to proceed correctly.

5. Full Datapath

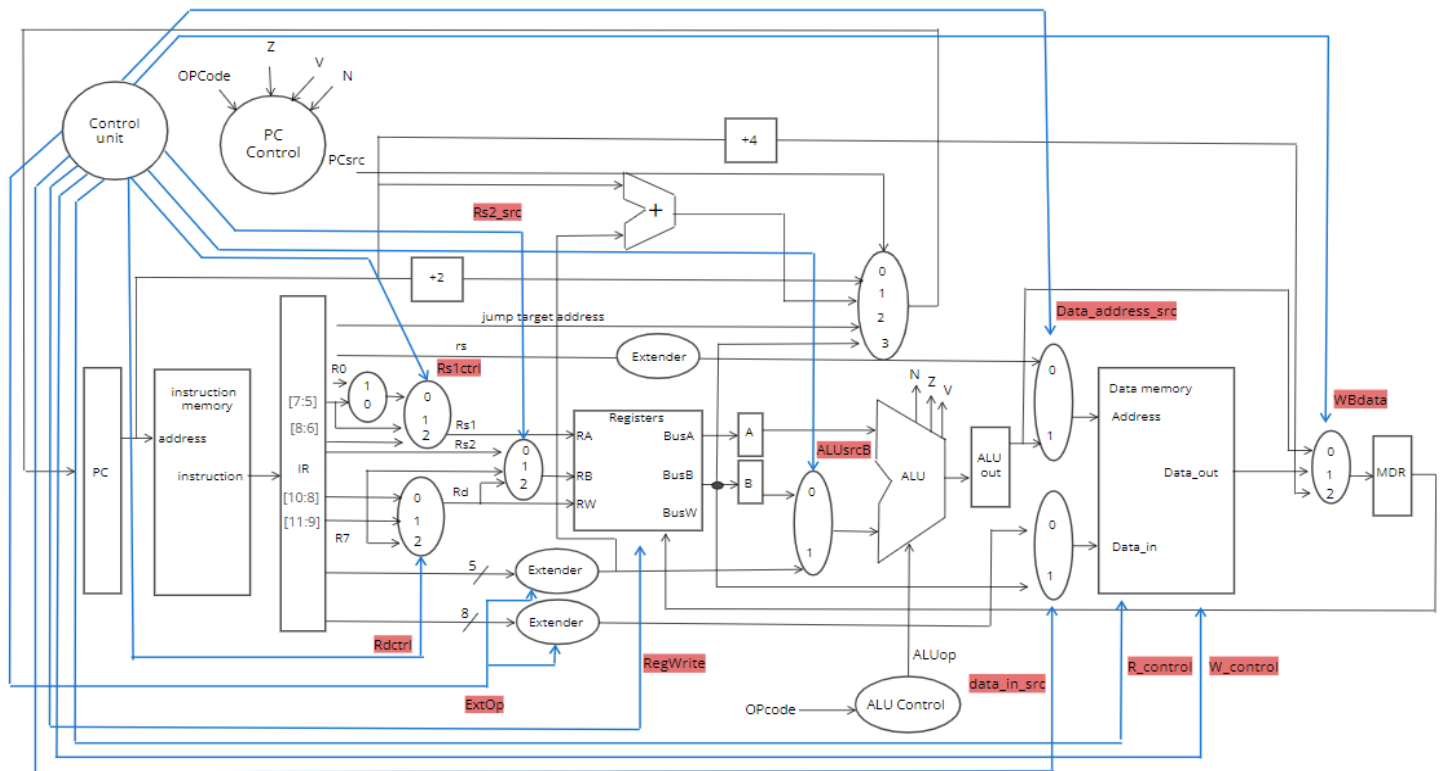


Figure 7 Full Datapath

6. Testing

Initial values in the 8 16-bits registers:

```
initial begin
// Initial values for the registers
    registers[0] = 16'h0000;
    registers[1] = 16'h0120;
    registers[2] = 16'h1032;
    registers[3] = 16'h0040;
    registers[4] = 16'h0156;
    registers[5] = 16'h0341;
    registers[6] = 16'h0C4D;
    registers[7] = 16'hBE86;
end
```

Initial values in some locations of data memory:

```
initial begin
// Initial values for the memory
    memory[0] = 8'h84;
    memory[1] = 8'h02;
    memory[2] = 8'h58;
    memory[3] = 8'h14;
    memory[4] = 8'hE0;
    memory[5] = 8'h2A;
    memory[6] = 8'h40;
    memory[7] = 8'h75;
    memory[8] = 8'h3F;
    memory[9] = 8'h31;
    memory[10] = 8'h0C;
    memory[11] = 8'h89;
    memory[12] = 8'h00;
    memory[13] = 8'hC0;
    memory[14] = 8'h01;
    memory[15] = 8'hC0;
end
```

Four testing codes were used to test the various instructions of the instruction set supported by this processor.

Load instructions (LW, LBU, LBS)


☐  registers	0284, 0084, FF84..
⊕  registers[7]	0284
⊕  registers[6]	0084
⊕  registers[5]	FF84
⊕  registers[4]	0156
⊕  registers[3]	0040
⊕  registers[2]	1032
⊕  registers[1]	0120
⊕  registers[0]	0000

Figure 8 Load instructions testing

Table 5 Load instructions testing code

Instructions	Binary coding	Hexadecimal coding
<i>LW R7, R0, 0</i>	0101 0 111 000 00000	5700
<i>LBU R6, R0, 0</i>	0110 0 110 000 00000	6600
<i>LBS R5, R0, 0</i>	0110 1 101 000 00000	6D00

The testing involves three load instructions (word, unsigned byte, signed byte), each instruction is used to load data from memory into a register.

- **LW R7, R0, 0:** Loads a 16-bit word from memory address 0 + R0 (address 0) into register R7.
- **LBU R6, R0, 0:** Loads an unsigned byte from memory address 0 + R0 (address 0) into register R6.
- **LBS R5, R0, 0:** Loads a signed byte from memory address 0 + R0 (address 0) into register R5.

Figure 8 shows the values of R7, R6, and R5 after testing the 3 load instructions.

R-type, I-type (branch instructions), JMP

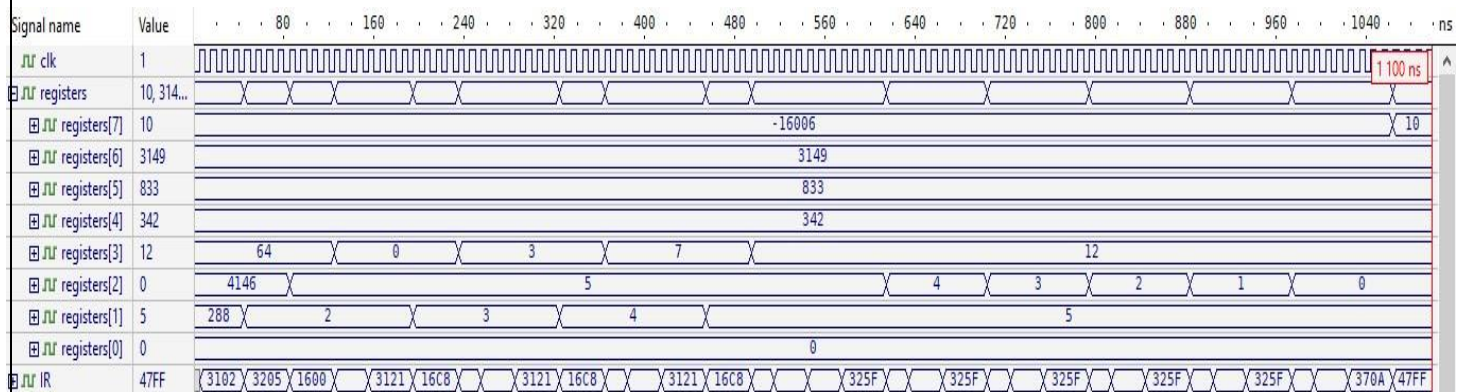


Figure 9 R-type, I-type (branch instructions), JMP testing

Table 6 R-type, I-type (branch instructions), JMP testing code

Instruction	Binary coding	Hexadecimal coding
ADDI R1, R0, 2	0011 0 001 000 00010	3102
ADDI R2, R0, 5	0011 0 010 000 00101	3205
ADD R3, R0, R0	0001 011 000 000 000	1600
loop:		
BEQ R1, R2, loop2	1010 0 001 010 00110	A148
ADDI R1, R1, 1	0011 0 001 001 00001	3121
ADD R3, R3, R1	0001 011 011 001 000	16C8
JMP loop	1100 0000 0000 0110	C006
loop2:		
BEQZ R2, end	1010 1 010 000 00100	AA06
ADDI R2, R2, -1	0011 0 010 010 11111	325F
JMP loop2	1100 0000 0001 0000	C007
end:		
ADDI R7, R0, 10	0011 0 111 000 01010	370A

The sample code in table 6 aims to show how the Branch, jump, R-type and I-type instructions work. It is shown in figure 9, how the registers are updated after each instruction, and how if the branch condition is taken the processor moves to the instruction in the branch target address.

The program works as the following:

- The value of R1 is updated to 2
- The value of R2 is updated to 5
- The value of R3 is updated to 0
- Loop begin; if the value of R1 equals the value of R2 then move to loop2
- Increment the value in R1 by 1
- Add the value of R1 to the value of R3 and store it in R3
- Jump to loop until the branch condition is true
- Loop2 begin; if the value of R2 equals the value 0 then move to end
- Decrement the value of R2 by 1
- Jump to loop2 until the branch condition is true
- end begin; update the value of R7 to 10

SW, Sv

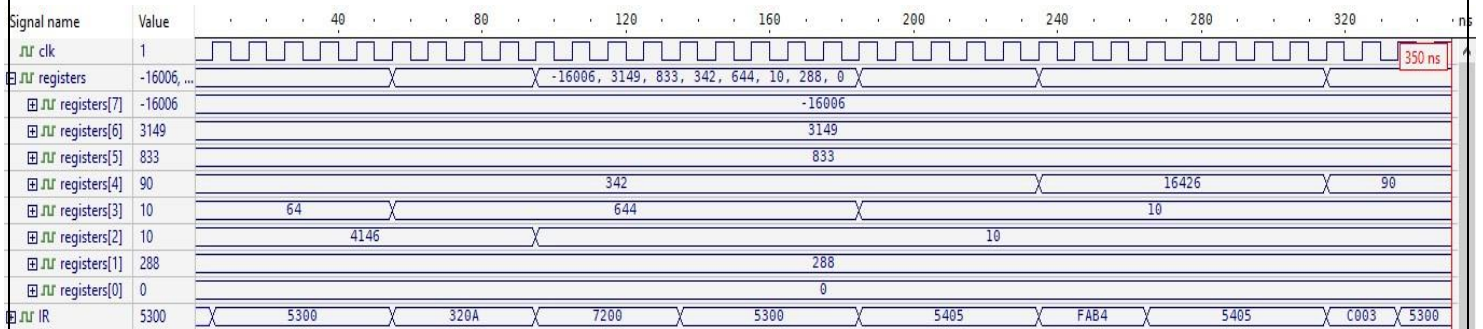


Figure 10 SW, Sv instructions testing

Table 7 SW, Sv instructions testing code

Instruction	Binary coding	Hexadecimal coding
LW R3, R0, 0	0101 0 011 000 00000	5300
ADDI R2, R0, 10	0011 0 010 000 01010	320A
SW R2 R0, 0	0111 0 010 000 00000	7200
LW R3, R0, 0	0101 0 011 000 00000	5300
LW R4, R0, 5	0101 0 100 000 00101	5405
Sv (5), 90	1111 101 01011010 0	FAB4
LW R4, R0, 5	0101 0 100 000 00101	5405

In this testing program, SW and Sv instructions are tested as follows:

- load the value of address $R0 + 0$ in R3
- update the value of R2 to 10
- store the value of R2 in address $R0 + 0$
- load the value of address $R0 + 0$ in R3 again to check if SW works correctly
- load the value of address $R0 + 5$ in R4
- store the immediate value 90 in address 5
- load the value of address $R0 + 5$ in R4 again to check if Sv works correctly

From figure 10, it is shown how the values changed after SW and Sv instructions, this indicates that these instructions work correctly.

Call, Ret

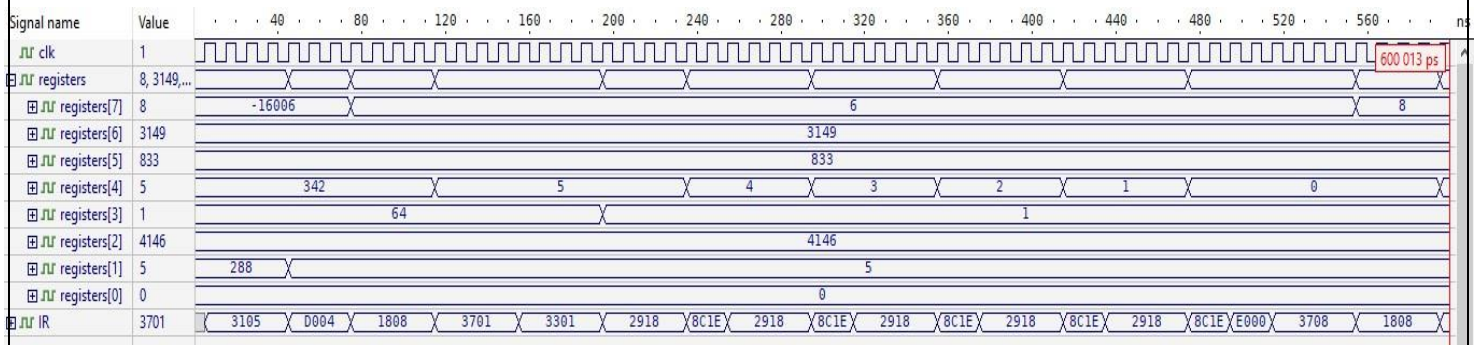


Figure 11 Call, Ret instructions testing

Table 8 Call, Ret instructions testing code

Instruction	Binary coding	Hexadecimal coding
<i>ADDI R1, R0, 5</i>	0011 0 001 000 00101	3105
<i>CALL func</i>	1101 0000 0000 100	D004
<i>ADDI R7, R0, 8</i>	0011 0 111 000 01000	3708
<i>func:</i>		
<i>ADD R4, R0, R1</i>	0001 100 000 001 000	1808
<i>ADDI R7, R0, 1</i>	1100 0 111 000 00001	3701
<i>ADDI R3, R0, 1</i>	0011 0 011 000 00001	3301
<i>loop:</i>		
<i>SUB R4, R4, R3</i>	0010 100 100 011 000	2918
<i>BGTZ R4, loop</i>	1000 1 100 000 11100	8C1E
<i>RET</i>	1110 0000 0000 0000	E000

This sample code aims to test if CALL and RET instructions work correctly, in addition to checking if the lock_R7 signal do its job, the program works as follows:

- update the value of R1 to 5
- move to func
- update R4 with the value of R1
- update R7 with the value 1
- update R3 with the value 1
- loop begin; decrement the value of R4 by the value of R3
- continue the loop until the value in R4 is 0
- return to the address after CALL
- update R7 with the value 8 to indicate that the end of the program reached

Figure 11 shows the registers after each instruction, it is noticed that instruction “ADDI R7, R0, 1” does not update the value of R7, then the return address that is stored in R7 in CALL instruction is preserved.