

Design of MIPS Functional Simulator using a High-Level Language (Web-based Implementation using ReactJs)



National Institute of Technology, Silchar
Department of Computer Science and Engineering
Session: Jan-June 2020

A Project Report Submitted By

Ujjawal Jain		1815077
Divesh	1	1815078
E. Joythish Reddy		1815079
P Harshavardhini		1815080
Brangong Sisen Singpho		1815081

Subject: - Computer Architecture & Organization

Subject Code: - CS-205

Group No.: - 16

Contents

Introduction	4
MIPS Architecture	4
Overview of MIPS CPU	4
Overview of MIPS Memory	5
Overview of MIPS Registers	6
Uses of Various General-Purpose Registers.....	7
Types of MIPS Instructions	7
Addressing Modes of MIPS	8
The MIPS Simulator	9
MIPS as 3-address machine.....	10
MIPS Instruction Pipelining	11
MIPS Instructions	11
# Arithmetic Instructions	12
# Comparison Instructions	14
# Constant Manipulating Instruction	15
# Logical Instructions	15
# Shift Operations	16
# Branch Instructions.....	17
# Jump Instructions.....	18
# Load Instructions	18
# Store Instructions	18
# Data Movement Instructions	18
# Exception and Interrupt Instructions	19
MIPS ALU Simulation: Using Circuit Verse Simulator	20
Designing a 1-bit AND, ADD and OR circuit	20
1-Bit Subtractor:	20
1-Bit NOR:	21
4-Bit ALU:.....	21
Block Simulation of a 32-bit MIPS	22
Examples on the designed MIPS Simulator	23
Multi-cycle Implementation of MIPS	25
The Five Cycles of MIPS:	26
Conclusion	27

Appendix	28
A-1 The user interface of the Simulator	28
A-2 The Registers and storage	29
A-3 The main functions for particular instructions	31
A-4 List of Instructions Supported by the Simulator	33
References	33

Entire Simulator Code	https://github.com/ujjawal-1999/MIPS-Simulator
The Simulator	https://5f1083804f6deee0fa75a21a--mips-simulator.netlify.app/
Circuit Simulations	https://circuitverse.org/simulator/4-bit-mips-alu#
Other Simulations	https://circuitverse.org/users/22081

Introduction

One of the major goals of computer science is to use abstraction to insulate the users from how the computer works. Abstraction is a very positive goal, but at some level a computer is just a machine. While High Level Languages (HLL) abstract and hide the underlying hardware, they must be translated into assembly language to use the hardware.

Assembly Language is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Because assembly depends on the machine code instructions, every assembler has its own assembly language which is designed for exactly one specific computer architecture.

MIPS (Microprocessors without Interlocked Pipelined Stages) is such a hardware processor. Manipulating its hardware requires the programmer to write and implement code in assembly language that can be converted to machine language by an assembler. Here, we will explore the assembly language designed for the MIPS architecture and implement the same by designing a web-based simulator on our own.

MIPS Architecture

MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Computer Systems. MIPS is an abbreviation for two different computing terms; millions of instructions per second and microprocessor without interlocking pipeline stages. The first use is a common method of determining a computer's processor speed. Generally, the more MIPS it can perform, the faster it operates. The second is a variety of reduced instruction set computer (RISC), a design that reduces the complexity of its processor in order to speed up the system.

MIPS is a modular architecture supporting up to four coprocessors (CP0/1/2/3). In MIPS terminology, CP0 is the System Control Coprocessor (an essential part of the processor that is implementation-defined in MIPS I–V), CP1 is an optional floating-point unit (FPU) and CP2/3 are optional implementation-defined coprocessors (MIPS III removed CP3 and reused its opcodes for other purposes).

MIPS is a load/store architecture (also known as a register-register architecture); except for the load/store instructions used to access memory, all instructions operate on the registers. MIPS-I has thirty-two 32-bit general-purpose registers (GPRs).

Overview of MIPS CPU

MIPS CPU architectures contain 3 main units.

1. The first is the ALU, which performs all calculations such as addition, multiplication, subtraction, division, bit-shifts, logical operations, etc.

2. Registers are a limited amount of memory which exists on the CPU. No data can be operated on in the CPU that is not stored in a register. Data from memory or disk drives must first be loaded into a register before the CPU can use it. In the MIPS CPU, there are only 32 registers, each of which can be used to store a single 32-bit values.
3. A CU (Control Unit) controls the mechanical settings on the computer so that it can execute the commands.

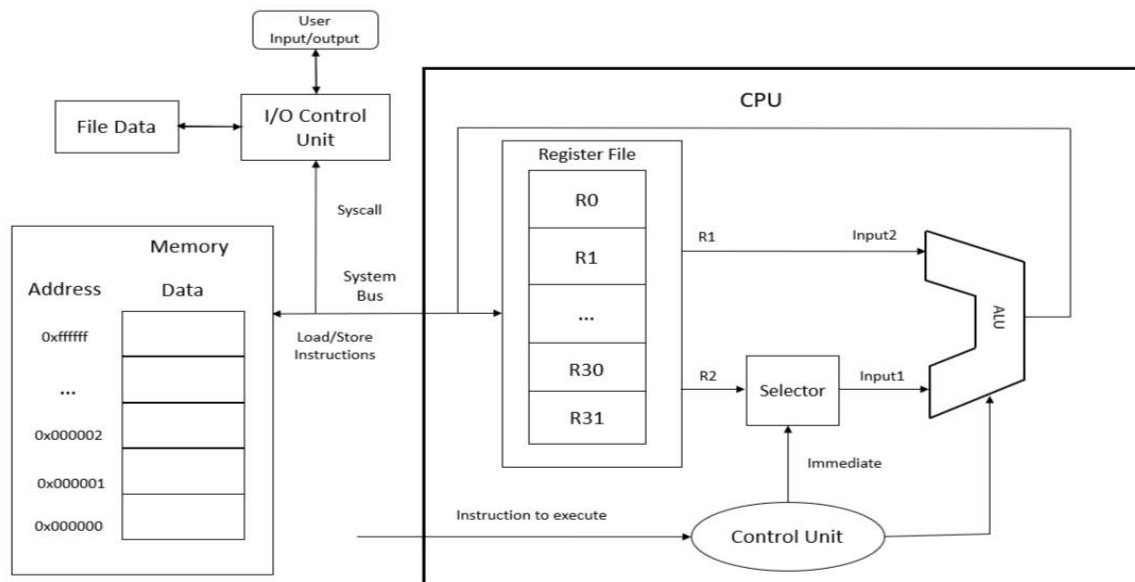


Fig. The MIPS Architecture

Overview of MIPS Memory

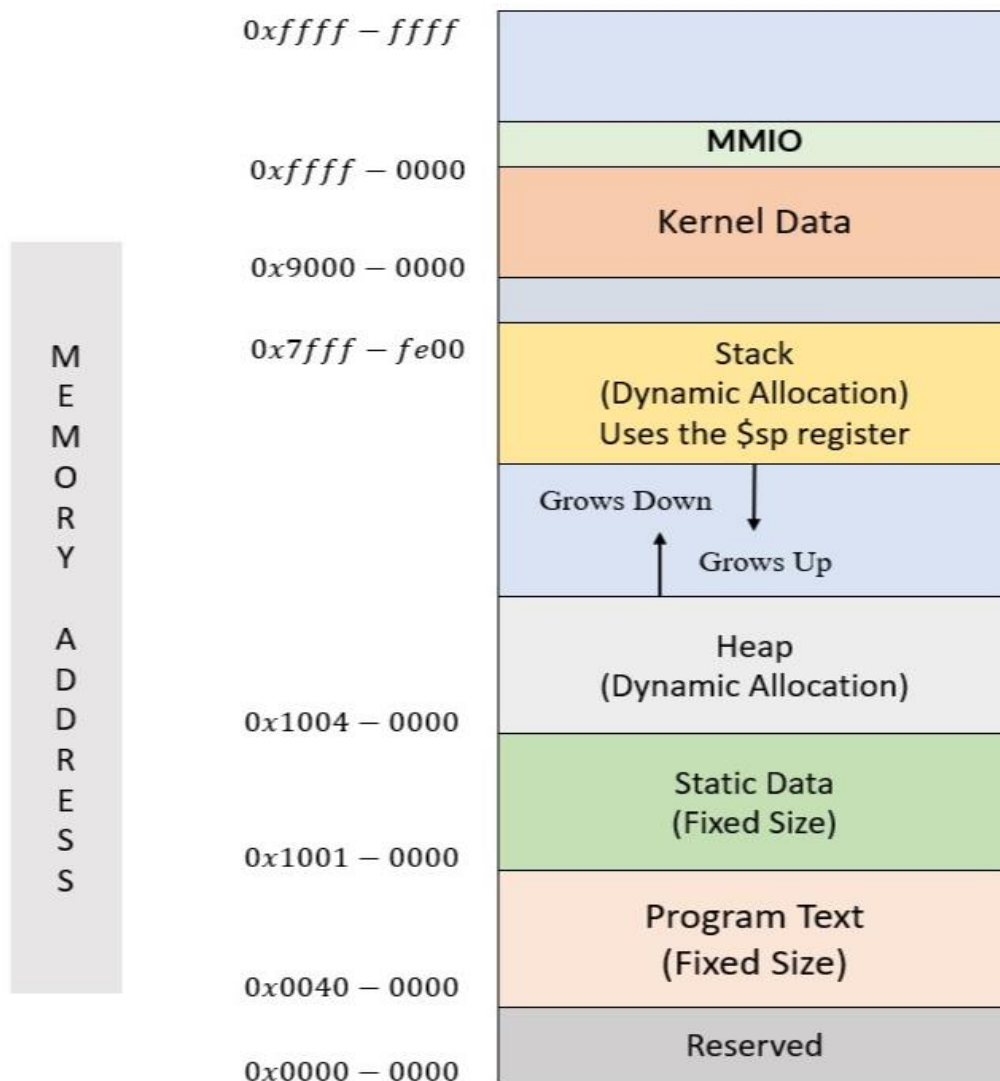
MIPS implements a 32-bit flat memory model. Memory on a MIPS computer starts at address 0x00000000 and extends in sequential, contiguous order to address 0xffffffff.

A 32-bit flat memory model says that a program can *address* (or find) 4 Gigabytes (4G) of data. Some of that memory is used up by the operating system (called *kernel data*), some of it used by the I/O subsystem, etc.

The types of memory used by MIPS are the following:

- **Reserved** - This is memory which is reserved for the MIPS platform.
- **Program text** - (Addresses 0x0040 0000 - 0x1000 0000) This is where the machine code representation of the program is stored.
- **Static data** - (Addresses 0x1001 0000 - 0x1004 0000) This is data which will come from the data segment of the program.
- **Heap** - (Addresses 0x1004 0000 - until stack data is reached, grows upward) Heap is dynamic data which is allocated on an as-needed basis at run time.
- **Stack** - (Addresses 0x7fff fe00 - until heap data is reached, grows downward) The program stack is dynamic data allocated for subprograms via push and pop operations. All method local variables are stored here.
- **Kernel** - (Addresses 0x9000 0000 - 0xffff 0000) - Kernel memory is used by the operating system, and so is not accessible to the user.

- MMIO - (Addresses $0xffff\ 0000$ - $0xffff\ 0010$) - Memory Mapped I/O, which is used for any type of external data not in memory, such as monitors, disk drives, consoles, etc.



Overview of MIPS Registers

Registers are a limited number of memory values that exist directly in the CPU. In order to do anything useful with data values in memory, they must first be loaded into registers. The following are the different types of registers present in MIPS along with their uses.

- General Purpose Registers(GPRs): - There are 32 32-bit GPRs in MIPS numbered 0 to 31 and designated as \$0 to \$31. All have designated usage by software.
- Multiply/Divide Registers: - MIPS has two special registers 32-bit HI register and 32-bit LO register which are used for storing the results of Multiply and Divide. They cannot be accessed directly.
- Program Counter(PC): - MIPS has a 32-bit PC which contains the address of the next instruction to be executed. During instruction fetch, the PC is incremented to point to the next instruction.

Uses of Various General-Purpose Registers

Each of the General-Purpose Register has a special use in the processor. So, let us look at them in details before actually executing any of the instructions using simulator.

<u>Mnemonic (Register Number)</u>	<u>Uses</u>
<i>\$zero</i> (\$0)	Always contains a constant value of 0. It can be read but cannot be written.
<i>\$at</i> (\$1)	Reserved for the assembler. If the assembler needs to use a temporary register, it will use <i>\$at</i>
<i>\$v0</i> – <i>\$v1</i> (\$2 – \$3)	Used for return values for subprograms. <i>\$v0</i> is also used to input the requested service to Syscall.
<i>\$a0</i> – <i>\$a3</i> (\$4 – \$7)	Used to pass actual arguments to subprograms
<i>\$t0</i> – <i>\$t9</i> (\$8 – \$15, \$24 – \$25)	Used to store temporary variables. The values can change when a subprogram is called
<i>\$s0</i> – <i>\$s7</i> (\$16 – \$23)	Used to store saved values. The values are maintained across subprogram calls
<i>\$k0</i> – <i>\$k1</i> (\$26 – \$27)	Used by the operating system
<i>\$gp</i> (\$28)	Pointer to global variables. Used with heap allocations.
<i>\$sp</i> (\$29)	Stack Pointer, keeps track of the top of the stack
<i>\$fp</i> (\$30)	Frame Pointer. used with the <i>\$sp</i> for maintaining information about the stack.
<i>\$ra</i> (\$31)	Contains return address: a pointer to the address to use when returning from a subprogram.

Because instructions that call a subroutine overwrite register *\$ra* with their return address, any subroutine that calls another subroutine must save register *\$ra* prior to a nested call on the stack.

Types of MIPS Instructions

All MIPS processor instructions are encoded in a single 32-bit word format. All data operations are register to register; the only memory references are pure load/store operations. MIPS supports the following types of instructions: -

1. I-type (Immediate) Instructions: -

6	5	5	16
Operation Code	rs	rt	Immediate

Load & Store instructions move data between memory and registers are all I-Type.

2. J-Type(Jump) Instructions: -

6	26
Operation Code	Target

Jump instructions are J-Type or R-Type while branch instructions are I-Type.

3. R-Type(Register) Instructions: -

6	5	5	5	5	6
Operation Code	rs	rt	rd	Shift	Function

Computational instructions (arithmetic, logical, shift) operate on registers are both R-Type and I-Type

Addressing Modes of MIPS

Addressing modes define how the operands or memory addresses are specified in an instruction. MIPS supports the following addressing modes: -

1. Register Addressing: - Register addressing is a form of **direct addressing**.

Instructions using registers execute fast because they do not have the delay associated with memory access. The value in the register is an operand instead of being a memory address to an operand.

Example: - *add \$s1,\$s2,\$s3*, where $\$s1 = rd, \$s2 = rs, \$s3 = rt$.

2. Immediate Addressing: - One operand is a constant within the instruction itself.

The advantage of using it is that there is no need to have extra memory access to fetch the operand. However, the size of operand is limited to 16 bits.

The jump instruction format also falls under immediate addressing, where the destination is held in the instruction.

Example: - *addi \$t1,\$zero,10* , which mean $\$t1 \leftarrow 0 + 10$

3. PC-Relative Addressing: - Program Counter addressing is used for conditional branches. The address is the sum of the program counter and a constant in the instruction.

$$\text{The operand address} = PC + \text{an offset}$$

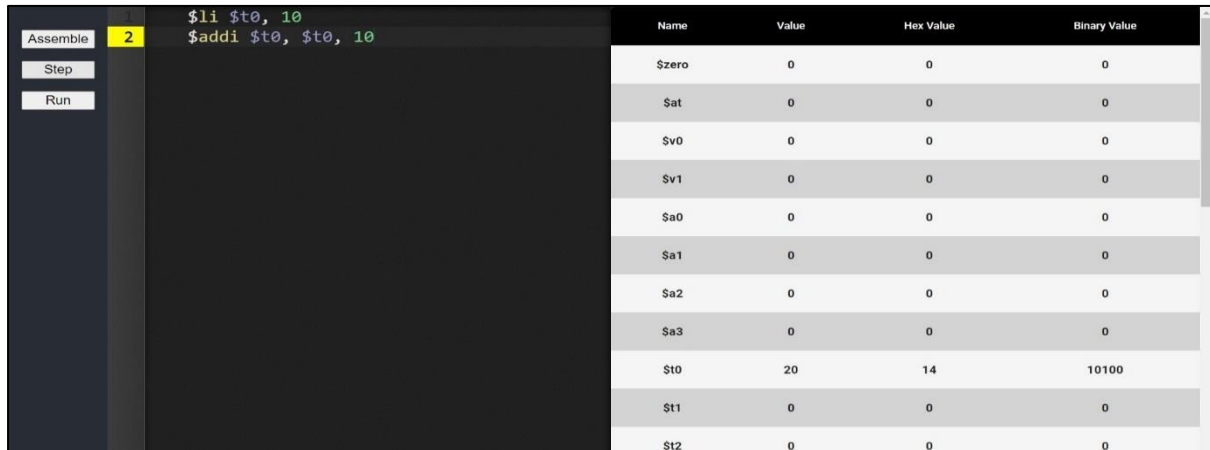
Example: - *beq \$t0 , strEnd*

4. Base Addressing: - Base addressing is also known as indirect addressing, where a register act as a pointer to an operand located at the memory location whose address is in the register. The address of the operand is the sum of the offset value and the base value(rs). However, the size of operand is limited to 16 bits because each MIPS instruction fits into a word.

Example: - *lw \$t1 , 4 (\$t2)*

The MIPS Simulator

The main aim of the project was the designing of a MIPS Simulator. Accordingly, a web-based MIPS Simulator has been designed using the latest React.js Web Framework. An open-source editor 'Ace-editor' has been incorporated for executing the instructions. Further, it uses the MIPS assembler, an open-source tool highlighting codes efficiently. Before we start with executing instructions, let us look at the structure of the entire simulator.



The screenshot shows the MIPS Simulator interface. On the left is a toolbar with buttons: 'Assemble' (highlighted in yellow), 'Step', and 'Run'. The central editor contains two lines of MIPS assembly code: `$li $t0, 10` and `$addi $t0, $t0, 10`. On the right is a table of registers.

Name	Value	Hex Value	Binary Value
\$zero	0	0	0
\$at	0	0	0
\$v0	0	0	0
\$v1	0	0	0
\$a0	0	0	0
\$a1	0	0	0
\$a2	0	0	0
\$a3	0	0	0
\$t0	20	14	10100
\$t1	0	0	0
\$t2	0	0	0

Fig. A web-based MIPS Simulator

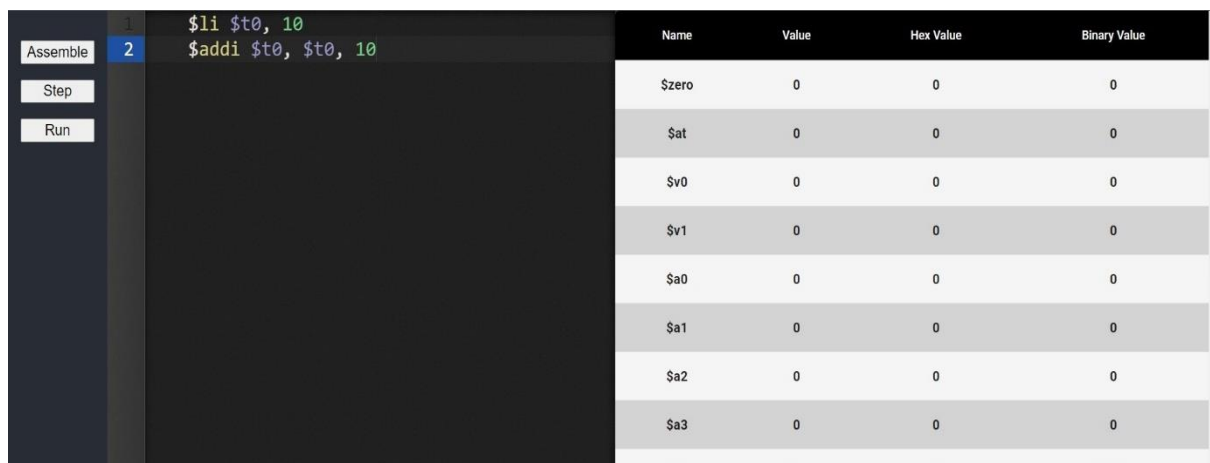
Before assembling, the environment of this simulator can be split to three segments: the *editor* in the middle where all of the code is being written, the toolbar at the left of the editor where all execution buttons are present and the *list of registers* on the right that represent the "CPU" for our program.

Once the code is written in the editor, we just have to assemble the code using the 'Assemble' button in the toolbar. After assembling the code, the code can be executed in two ways: -

1. Using 'Step' – this will execute one instruction at a time and show the output.
2. Using 'Run' – this will execute the entire code and just produce the final output.

The output can be seen by the change in register values.

Once the entire execution is completed, the highlight on the left side of the editor becomes blue as shown below.



The screenshot shows the MIPS Simulator interface after execution. The 'Run' button is now highlighted in blue. The register table on the right is updated, showing the final values of the registers.

Name	Value	Hex Value	Binary Value
\$zero	0	0	0
\$at	0	0	0
\$v0	0	0	0
\$v1	0	0	0
\$a0	0	0	0
\$a1	0	0	0
\$a2	0	0	0
\$a3	0	0	0
\$t0	20	14	10100
\$t1	0	0	0
\$t2	0	0	0

Fig. MIPS Simulator after the program is executed.

MIPS as 3-address machine

Most of the MIPS operators take 3 registers as parameters in the instructions, and that is because the MIPS CPU is known as 3-address machine.

The three registers are: -

1. The first input to the ALU, always R_s (first source register)
2. The second input to the ALU, either R_t (second source register) or an immediate value
3. The register to write the result to, R_d (destination register)

Most of the instructions follow this format. The ALU here takes two inputs of which the first input is always the R_s register, while the second input depends on the operator. The register operator will always take the input from the R_t register.

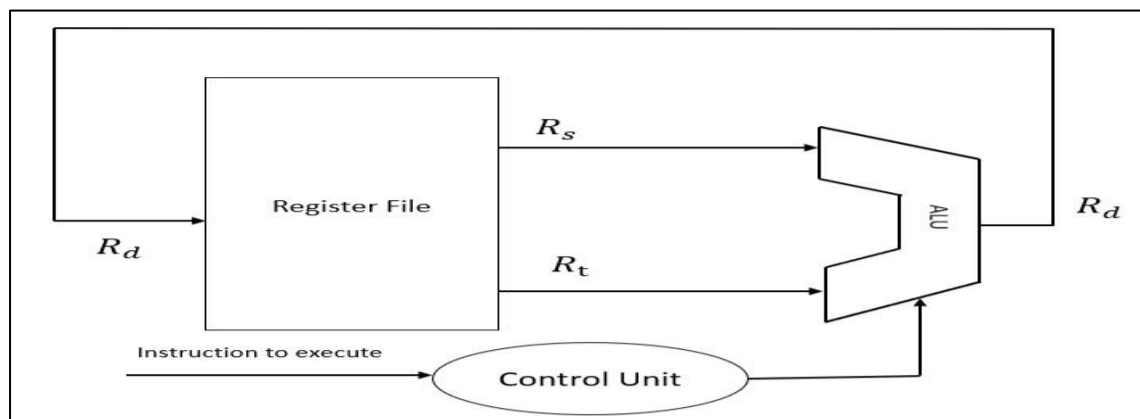


Fig. Basic ALU setup for 3-address instructions
[operator] R_d R_s R_t

This syntax means $R_d \leftarrow R_s$ [operator] R_t

When the second input is an immediate value as in case of Immediate operators, the operator is replaced by appending an 'i' to the operator name and the register R_t is replaced by an immediate value.

[operator]i R_d R_s immediate value

The syntax means $R_d \leftarrow R_s$ [operator]i immediate value

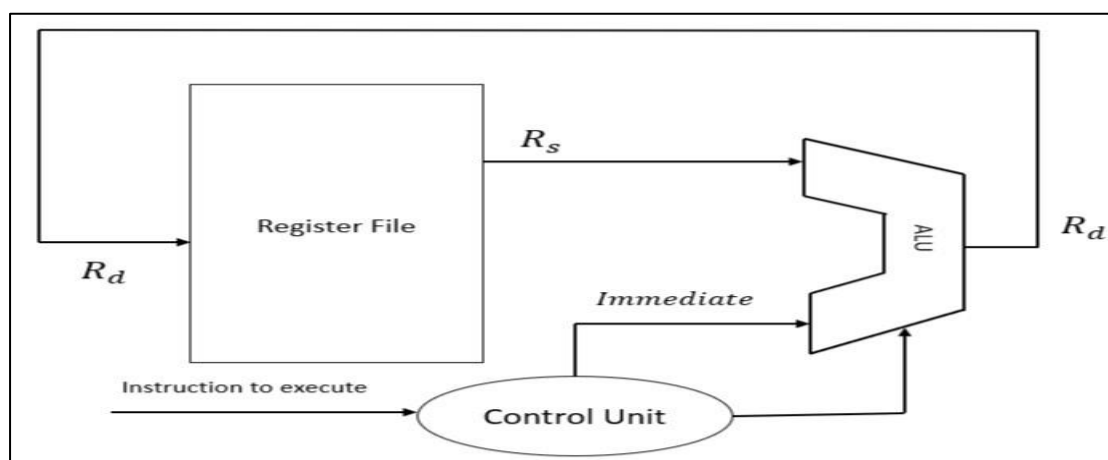


Fig. Basic ALU setup for 3-address with immediate value

This helps explain the difference between a constant and an immediate value. A constant is a value which exists in memory and must be loaded into a register before it is used. An immediate value is defined as part of the instruction, and so is loaded as part of the instruction. No load of a value from memory to a register is necessary, making the processing of an immediate value faster than processing a constant.

MIPS Instruction Pipelining

Instruction pipelining is a technique used in the design of modern microprocessors, microcontrollers and CPUs to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

The main idea is to divide (termed "split") the processing of a CPU instruction, as defined by the instruction microcode, into a series of independent steps of micro-instructions (also called "microinstructions"), with storage at the end of each step. This allows the CPUs control logic to handle instructions at the processing rate of the slowest step, which is much faster than the time needed to process the instruction as a single step.

The following 5-step execution cycle of MIPS explains the concept of pipelining.

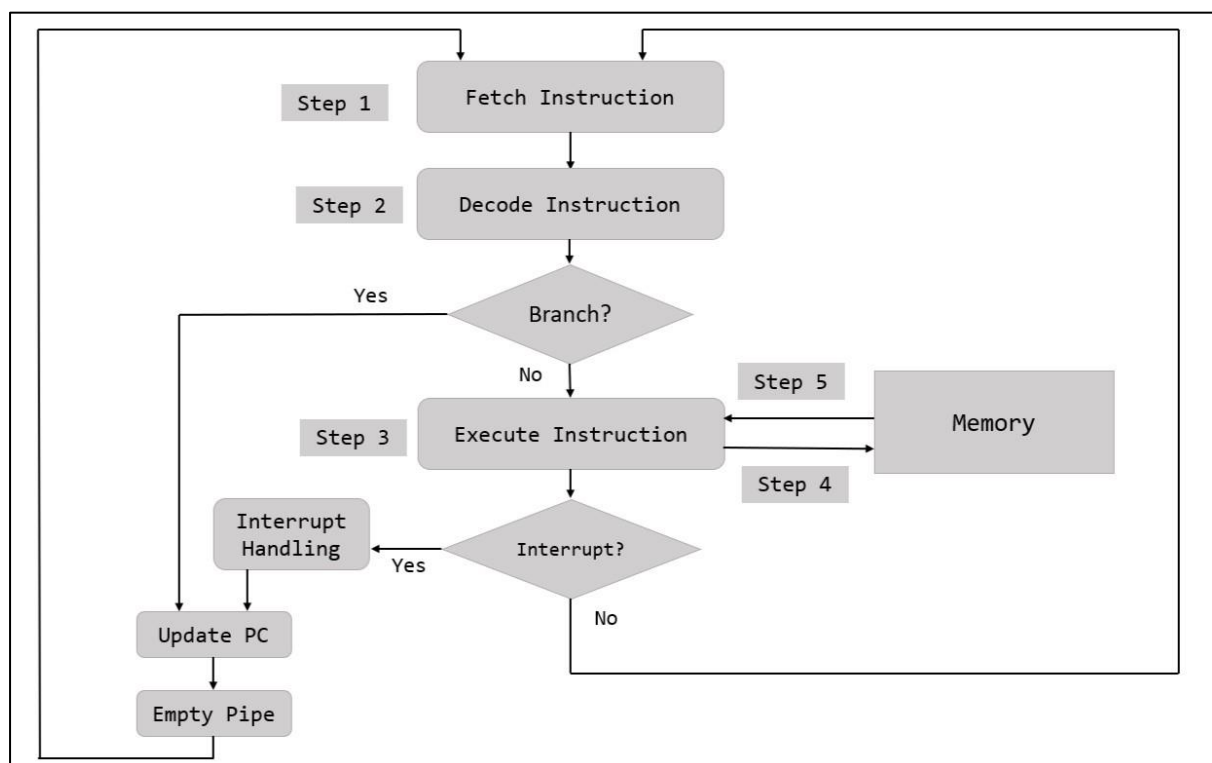


Fig. 5-step instruction execution of MIPS

MIPS Instructions

The MIPS instructions can be broadly divided as arithmetic, logical, comparison, load store, conditional, branch and jump instructions. Let us look at each of them in details.

Arithmetic Instructions

1. **add** instruction

R-type Format: -

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	1 0 0 0 0 0 (0x20)

Assembly format: - $add\ R_d\ R_s\ R_t$

Effect of the instruction: - $R_d \leftarrow [R_s] + [R_t]$; $PC \leftarrow [PC] + 4$

2. **addu** :- $addu\ R_d\ R_s\ R_t$ Addition without overflow

R-Type Format: -

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	1 0 0 0 0 1 (0x21)

Effect of the instruction: - $R_d \leftarrow [R_s] + [R_t]$; $PC \leftarrow [PC] + 4$

3. **addi** :- $addi\ R_t\ R_s\ Immediate$ Addition immediate with overflow

I-Type Format: -

6	5	5	16
0 0 1 0 0 0	R_s	R_t	$Immediate$

Effect of the instruction: - $R_t \leftarrow [R_s] + [I_{15...0}]$; $PC \leftarrow [PC] + 4$

4. **addiu** :- $addiu\ R_t\ R_s\ Immediate$ Addition immediate without overflow

I-Type Format: -

6	5	5	16
0 0 1 0 0 1	R_s	R_t	$Immediate$

Effect of the instruction: - $R_t \leftarrow [R_s] + [I_{15...0}]$; $PC \leftarrow [PC] + 4$

Simulator Example: - Let us demonstrate all the addition instructions on the simulator.

<div>Assemble</div> <div>Step</div> <div>Run</div>	1		Name	Value	Hex Value	Binary Value
	2	$addi\ \$t0,\ $zero,\ 10$				
	3	$add\ $t1,\ $t1,\ $t0$				
	4	$addu\ $t2,\ $t1,\ $t0$				
		$addiu\ $t3,\ $t2,\ 10$				
			\$t0	10	A	1010
			\$t1	10	A	1010
			\$t2	20	14	10100
			\$t3	30	1E	11110

Fig. A Simulator code showing all the addition operations

5. **sub** :- $sub\ R_d\ R_s\ R_t$ Subtraction with overflow

R-Type Format: -

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	1 0 0 0 1 0

Effect of the instruction: - $R_d \leftarrow [R_s] - [R_t]; \quad PC \leftarrow [PC] + 4$
7. *subu* :- *subu* $R_d \ R_s \ R_t$ Subtraction without overflow

R-Type Format: -

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	1 0 0 0 1 1

Effect of the instruction: - $R_d \leftarrow [R_s] - [R_t]; \quad PC \leftarrow [PC] + 4$
Simulator Example

1	\$addi \$t0, \$zero, 10	Name	Value	Hex Value	Binary Value
2	\$li \$t1, 100	\$t0	10	A	1010
3	\$li \$t2, 50	\$t1	90	5A	1011010
4	\$sub \$t1, \$t1, \$t0	\$t2	40	28	101000
5	\$subu \$t2, \$t2, \$t0	\$t3	0	0	0
		\$t4	0	0	0

Fig. Simulator Code showing the subtraction operations

8. *Multiplication Instructions*: -

R-Type Format: -

6	5	5	10	6
0 0 0 0 0 0	R_s	R_t	0 0 0 0 0 0 0 0 0 0	Function code

Signed Multiply

mult $R_s \ R_t$ - Function Code = 0x18

Unsigned Multiply

multu $R_s \ R_t$ - Function Code = 0x19

Effect of the instruction

$$Hi||Lo \leftarrow [R_s] * [R_t]; \quad PC \leftarrow [PC] + 4$$

9. *mul*: *mul* $R_d \ R_s \ R_t$ Multiply without overflow

6	5	5	5	5	6
0x1c	R_s	R_t	R_d	0 0 0 0 0	0 0 0 0 1 0

Effect of the instruction

$$R_d \leftarrow [R_s] * [R_t]; \quad PC \leftarrow [PC] + 4$$

Simulator Example: -

Assemble	1	li \$s7, 10	Name	Value	Hex Value	Binary Value
Step	2	li \$t8, 10	\$s6	0	0	0
Run	3	mul \$t9, \$s7, \$t8	\$s7	10	A	1010
	4	mult \$s7, \$t8	\$t8	10	A	1010
			\$t9	100	64	1100100
			\$k0	0	0	0
			\$k1	0	0	0
			\$gp	0	0	0
			\$sp	0	0	0
			\$fp	0	0	0
			\$ra	0	0	0
			hi	1	1	1
			lo	0	0	0

Fig. Simulator Code showing the multiplication instructions

10. Division Instructions: -

R-Type: -

6	5	5	5	5	6
0x1c	R_s	R_t	R_d	00000	Function Code

$div\ R_s\ R_t$ Division with overflow - Function Code = 0x1a

$divu\ R_s\ R_t$ Division without overflow - Function Code = 0x1b

Effect of the instruction: $Lo \leftarrow [R_s] / [R_t]$; $Hi \leftarrow [R_s] \% [R_t]$; $PC \leftarrow [PC] + 4$;

Simulator Example: -

Assemble	1	li \$t8, 123	Name	Value	Hex Value	Binary Value
Step	2	li \$t9, 10	\$s6	0	0	0
Run	3	div \$t8, \$t9	\$s7	0	0	0
			\$t8	123	7B	1111011
			\$t9	10	A	1010
			\$k0	0	0	0
			\$k1	0	0	0
			\$gp	0	0	0
			\$sp	0	0	0
			\$fp	0	0	0
			\$ra	0	0	0
			hi	3	3	11
			lo	12	C	1100

Fig. Simulator Code showing the division instructions

Comparison Instructions

1. Set Less Than

R-Type Format

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	Function Code

Set Less Than $slt\ R_d\ R_s\ R_t$ Function Code: 0x2a

Set Less Than(Unsigned) $sltu\ R_d\ R_s\ R_t$ Function Code: 0x2b

Effect of the instruction

if ($[R_s] < [R_t]$) $\rightarrow R_d = 1$
 else $R_d = 0$; $PC = [PC] + 4$

2. Set Less Than Immediate

I-Type Format: -

6	5	5	16
Opcode	R_s	R_t	Immediate

Set Less Than Immediate $slti\ R_t\ R_s\ immediate$ Opcode: 0xa

Set Less Than Imm(Unsigned) $sltiu\ R_t\ R_s\ immediate$ Opcode: 0xb

Effect of the instruction

if ($[R_s] < immediate$) $\rightarrow R_t = 1$
 else $R_t = 0$; $PC = [PC] + 4$

Constant Manipulating Instruction

1. **Load Immediate:-** $li\ rdest\ immediate$
 $rdest \leftarrow immediate$; $PC = [PC] + 4$

Simulator Example:

Assemble	1	<code>li \$t0, 30</code>	Name	Value	Hex Value	Binary Value
	2	<code>li \$t1, 20</code>	\$a3	0	0	0
Step	3	<code>slt \$t2,\$t1,\$t0</code>	\$t0	30	1E	11110
Run	4	<code>slti \$t3,\$t1,100</code>	\$t1	20	14	10100
			\$t2	1	1	1
			\$t3	1	1	1

Fig. Simulator code showing the *slt* and *li* instructions

Logical Instructions

- R-Type

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	Function Code

a. AND: - $and\ R_d\ R_s\ R_t$ Function Code: 0x24

b. OR: - $or\ R_d\ R_s\ R_t$ Function Code: 0x25

c. XOR: - $xor\ R_d\ R_s\ R_t$ Function Code: 0x26

d. NOR: - $nor\ R_d\ R_s\ R_t$ Function Code: 0x27

Effect of the instruction

$$R_d \leftarrow R_s [\text{operation}] R_t ; \quad PC = [PC] + 4$$

- I-Type

6	5	5	16
Opcode	R_s	R_t	Immediate

- a. ANDI: - $\text{andi } R_t \ R_s \ \text{immediate}$ Function Code: 0xc
b. ORI: - $\text{ori } R_t \ R_s \ \text{immediate}$ Function Code: 0xd
c. XORI: - $\text{xori } R_t \ R_s \ \text{immediate}$ Function Code: 0xe

Effect of the instruction

$$R_t \leftarrow R_s [\text{operation}] \text{immediate} ; \quad PC = [PC] + 4$$

Simulator Example:

Assemble	1	li \$t0, 10	Name	Value	Hex Value	Binary Value
Step	2	li \$t1, 20	\$a3	0	0	0
Run	3	and \$t2, \$t1, \$t0	\$t0	10	A	1010
	4	or \$t3, \$t1, \$t0	\$t1	20	14	10100
	5	xor \$t4, \$t1, \$t0	\$t2	0	0	0
			\$t3	30	1E	11110
			\$t4	30	1E	11110

Fig. Simulator code showing the logical operations

Shift Operations

→ Without Shamt (Shift Amount)

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	0 0 0 0 0	Function Code

1. Shift Left Logical $\text{sllv } R_d \ R_t \ R_s$ Function Code: 0x04
 $R_d \leftarrow [R_t] \ll [R_s]$ $PC = [PC] + 4$
2. Shift Right Logical $\text{srlv } R_d \ R_t \ R_s$ Function Code: 0x06
 $R_d \leftarrow [R_t] \gg [R_s]$ $PC = [PC] + 4$

→ With Shamt

6	5	5	5	5	6
0 0 0 0 0 0	R_s	R_t	R_d	Shamt	Function Code

1. Shift Left Logical $\text{sll } R_d \ R_t \ \text{shamt}$ Function Code: 0x04
 $R_d \leftarrow [R_t] \ll \text{shamt}$ $PC = [PC] + 4$
2. Shift Right Logical $\text{srl } R_d \ R_t \ \text{shamt}$ Function Code: 0x06
 $R_d \leftarrow [R_t] \gg \text{shamt}$ $PC = [PC] + 4$

Simulator Example:

1	li \$t0, 1	Name	Value	Hex Value	Binary Value
2	li \$t1, 10	\$a3	0	0	0
3	sllv \$t2, \$t1, \$t0	\$t0	1	1	1
4	srlv \$t3, \$t1, \$t0	\$t1	10	A	1010
5	sll \$t4, \$t1, 2	\$t2	20	14	10100
6	srl \$t5, \$t1, 3	\$t3	5	5	101
		\$t4	40	28	101000
		\$t5	1	1	1

Fig. Simulator Code showing the shift operations

Branch Instructions

1. *branch on equal*

6	5	5	16
0 0 0 1 0 0	R_s	R_t	Offset

Assembly format: - *beq* R_s R_t label

Effect of the instruction: - *if*($[R_s] == [R_t]$) then $PC \leftarrow \text{label}$
else $PC = [PC] + 4$

2. *branch on greater than equal zero*

6	5	5	16
0 0 0 0 0 1	R_s	0 0 0 0 1	Offset

Assembly format: - *bgez* R_s label

Effect of the instruction: - *if*($[R_s] \geq 0$) then $PC \leftarrow \text{label}$
else $PC = [PC] + 4$

3. *branch on greater than zero*

6	5	5	16
0 0 0 1 1 1	R_s	0 0 0 0 0	Offset

Assembly format: - *bgtz* R_s label

Effect of the instruction: - *if*($[R_s] > 0$) then $PC \leftarrow \text{label}$
else $PC = [PC] + 4$

4. *branch on less than equal zero*

6	5	5	16
0 0 0 1 1 0	R_s	0 0 0 0 0	Offset

Assembly format: - *blez* R_s label

Effect of the instruction: - *if*($[R_s] \leq 0$) then $PC \leftarrow \text{label}$

$else\ PC = [PC] + 4$

5. *branch on less than zero*

6	5	5	16
0 0 0 0 0 1	R_s	0 0 0 0 0	<i>Offset</i>

Assembly format: - $bltz\ R_s\ label$

Effect of the instruction: - $if([R_s] < 0)\ then\ PC \leftarrow label$
 $else\ PC = [PC] + 4$

6. *branch on not equal*

6	5	5	16
0 0 0 1 0 1	R_s	R_t	<i>Offset</i>

Assembly format: - $bne\ R_s\ R_t\ label$

Effect of the instruction: - $if([R_s] \neq [R_t])\ then\ PC \leftarrow label$
 $else\ PC = [PC] + 4$

Jump Instructions

1. *jump instruction*

6	26
0 0 0 0 1 0	<i>target</i>

Assembly format: - $j\ target$

Effect of the instruction: - $PC \leftarrow target$

Load Instructions

1. *load byte instruction* $lb\ R_t\ address$

6	5	5	16
1 0 0 0 0 0 (0x20)	R_s	R_t	<i>Offset</i>

Store Instructions

1. *store byte instruction* $sb\ R_t\ address$

6	5	5	16
(0x28)	R_s	R_t	<i>Offset</i>

Data Movement Instructions

1. *move*

Assembly format: - $move\ R_{dest}\ R_{src}$

Effect of the instruction: - $R_{dest} \leftarrow [R_{src}];\ PC \leftarrow [PC] + 4$

Exception and Interrupt Instructions

1. *no operation instruction*

6	5	5	5	5	6
0	0	0	0	0	0

Assembly format: - *nop*

Effect of the instruction: - *does nothing*; [PC] – *unchanged*

Simulator Example: In this example, we'll see addition of a number to a register by running a loop. The number is added 5 times by a loop using branch and jump instructions

Assemble	1	<code>addi \$t0, \$zero, 5</code>	<code># Set t0 = 5</code>	Name	Value	Hex Value	Binary Value
Step	2	<code>loop:</code>		\$t0	0	0	0
Run	3	<code>beq \$t0, \$zero, exit</code>	<code># if(t0 == 0) e</code>	\$t1	25	19	11001
	4	<code>nop</code>		\$t2	0	0	0
	5	<code>addi \$t1, \$t1, 5</code>	<code># t1 += 5 (n num</code>	\$t3	0	0	0
	6	<code>addi \$t0, \$t0, -1</code>	<code># t0 -= 1</code>	\$t4	0	0	0
	7	<code>j loop</code>					
	8	<code>nop</code>					
	9	<code>exit:</code>					
	10	<code>addi \$v0, \$zero, 1</code>	<code># Program ended</code>				

Fig. Code displaying the working of branch and jump instructions

Example: Let us take two registers $\$t1 = 10$ and $\$t2 = 20$. We shall swap the two values using the temporary register and using addition method.

Assemble	1	<code>#Code for swapping two numbers</code>	Name	Value	Hex Value	Binary Value
Step	2	<code>li \$t1, 10</code>	\$a0	0	0	0
Run	3	<code>li \$t2, 20</code>	\$a1	0	0	0
	4	<code>move \$t3, \$t1</code>	\$a2	0	0	0
	5	<code>move \$t1, \$t2</code>	\$a3	0	0	0
	6	<code>move \$t2, \$t3</code>	\$t0	0	0	0
	7	<code>li \$t3, 0</code>	\$t1	20	14	10100
	8	<code>nop</code>	\$t2	10	A	1010
	9	<code>#Swapping using addition and subtraction op</code>	\$t3	10	A	1010
	10	<code>li \$t3, 20</code>	\$t4	20	14	10100
	11	<code>li \$t4, 10</code>	\$t5	0	0	0
	12	<code>add \$t5, \$t3,\$t4</code>	\$t6	0	0	0
	13	<code>move \$t3, \$t5</code>	\$t7	0	0	0
	14	<code>li \$t5, 0</code>				
	15	<code>nop</code>				
	16	<code>sub \$t5, \$t3,\$t4</code>				
	17	<code>move \$t4, \$t5</code>				
	18	<code>li \$t5, 0</code>				
	19	<code>nop</code>				
	20	<code>sub \$t5, \$t3,\$t4</code>				
	21	<code>move \$t3, \$t5</code>				
	22	<code>li \$t5, 0</code>				
	23	<code>nop</code>				

Fig. Swapping of two numbers program

The numbers in the two registers are swapped using the temporary register at first and then the same code has been implemented using the addition and subtraction method.

MIPS ALU Simulation: Using Circuit Verse Simulator

Designing a 32-bit ALU Circuit of MIPS and incorporating all the instructions in it is quite difficult, so we are designing a 4-bit ALU that performs the basic operations like addition, subtraction, AND, OR, NOR, set less, etc.

Let us look at the circuit diagram of all of them in details

Designing a 1-bit AND, ADD and OR circuit

Here, we have designed a circuit that makes use of a 4:1 MUX to select the instruction to be executed – AND, ADD, OR and NOP.

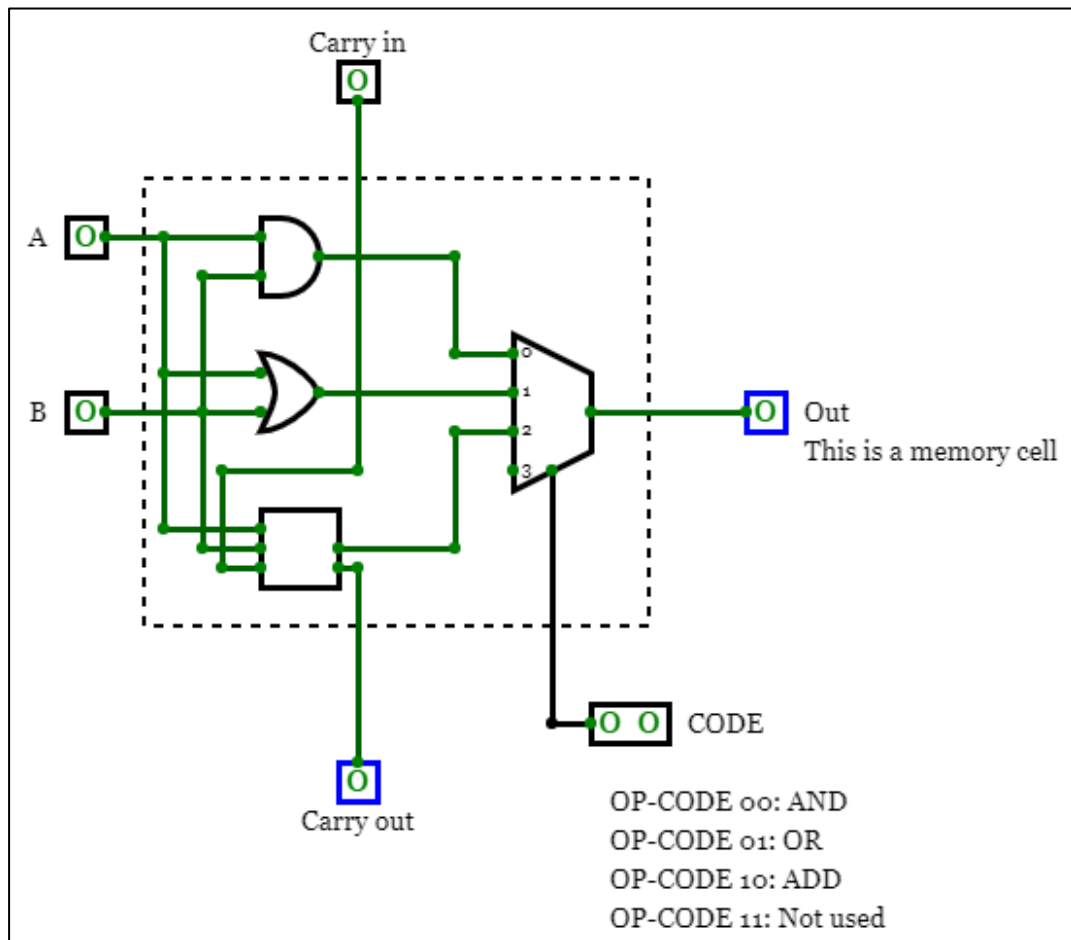


Fig. Designing a circuit performing AND, OR and ADD instructions

Here, we are using a simple And Gate to perform the AND logic, an Or Gate to perform the OR Logic and a Full Adder to perform the addition of numbers. Passing the output of all of them through a MUX and using select lines as the opcode, we get the required output.

1-Bit Subtractor:

A 1-bit which will be used later in designing of 4-bit ALU by combining them.

We know,

$$-B = B' + 1 \text{ (Two's Complement)}$$

$$A - B = A + B' + 1$$

$$\text{Opcode } 0110 + \text{carry in}$$

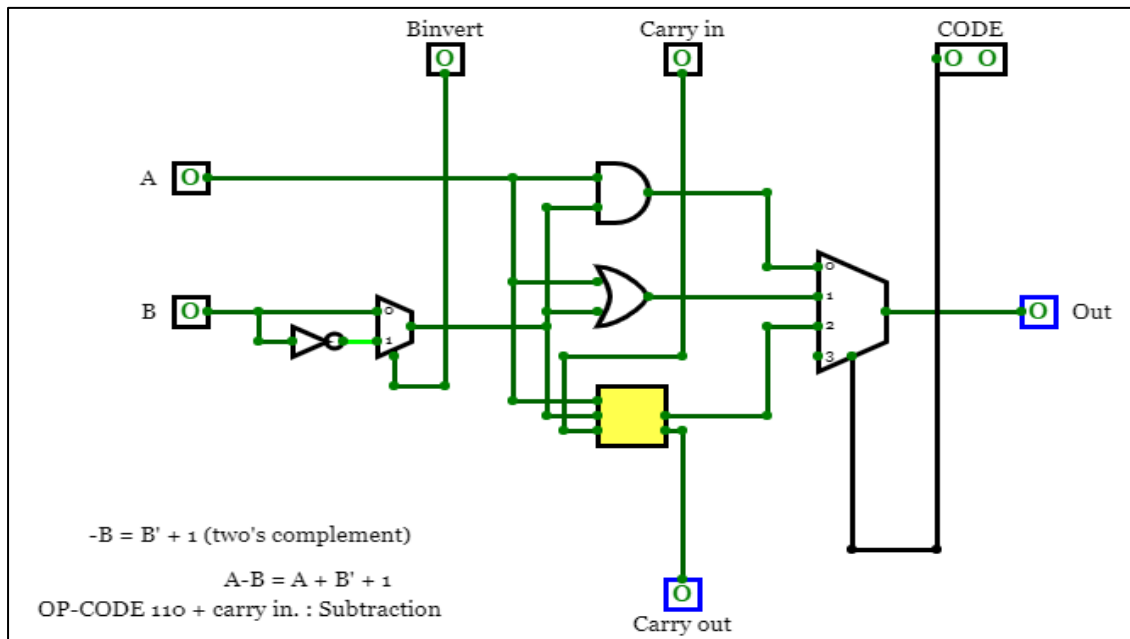


Fig. A 1-bit Subtractor

1-Bit NOR:

We know, $A \text{ Nor } B = A' \text{ And } B'$. So, we use the same concept here.

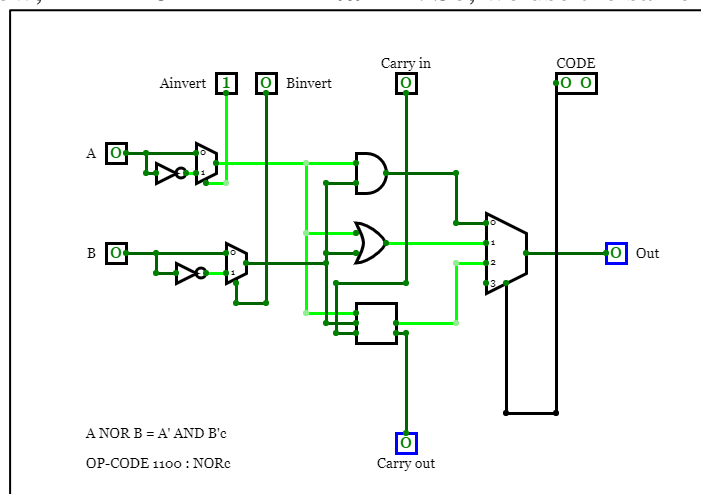


Fig. 1-Bit NOR

OP-Code 1100 $A_{inv} B_{inv} \text{ Code}(2)$

Expanding this concept for the 4-bit ALU we have the 4-bit ALU.

4-Bit ALU:

Let us now design a 4-Bit ALU using the circuits shown above as sub-circuit elements. The ALU is capable of performing three operations, AND, OR and ADD. However, the same concept can be extended to the designing of 16-bit and 32-bit MIPS ALUs.

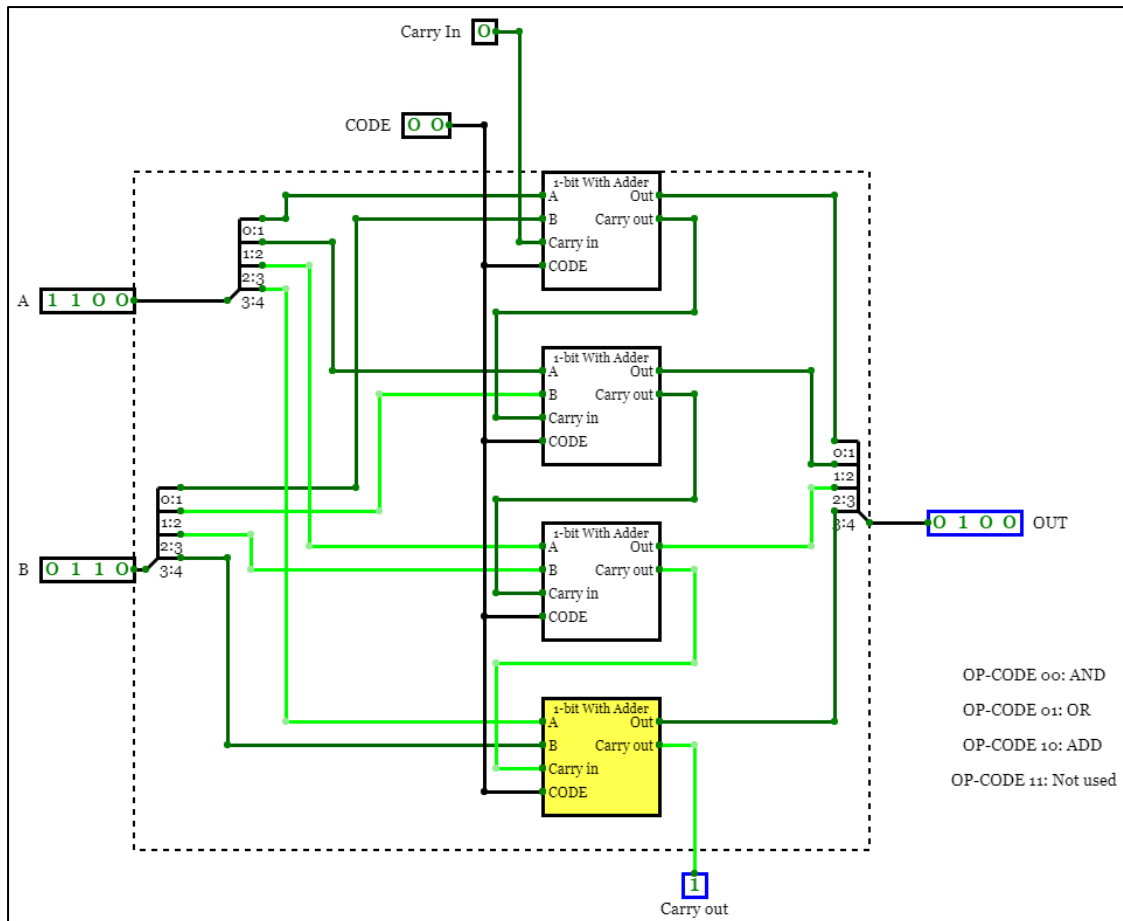


Fig. 4-Bit ALU supporting AND, OR and ADD operations

A simple block doing the same can be made which is demonstrated below: -

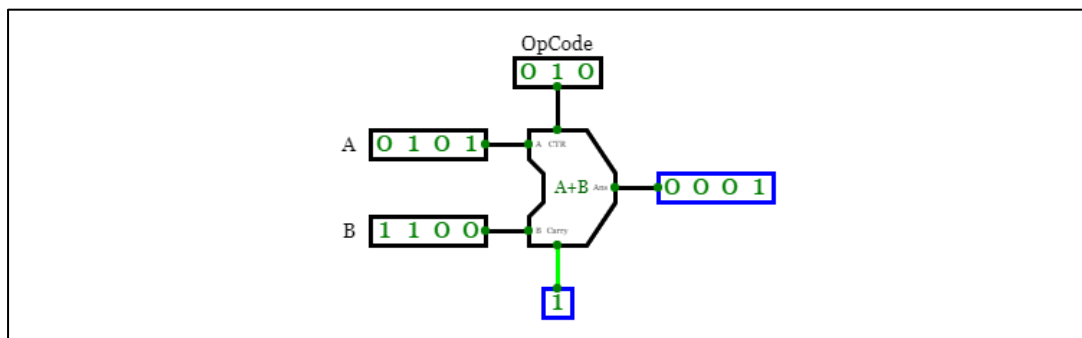


Fig. Block simulation of a 4-bit MIPS ALU

Block Simulation of a 32-bit MIPS

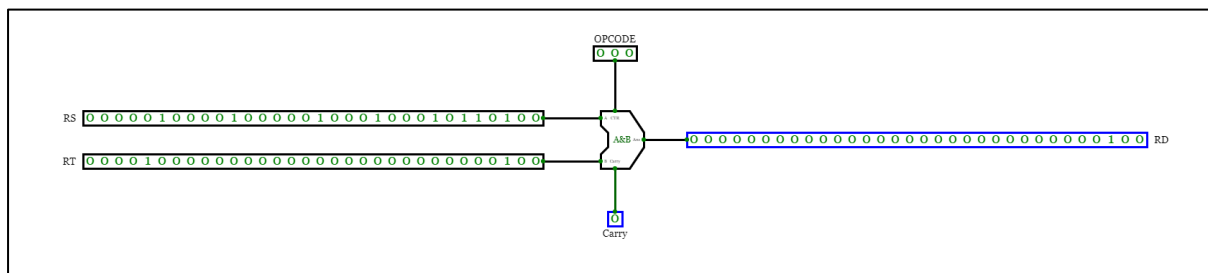


Fig. Simulation of a 32-bit MIPS ALU

All the simulations can be tested using the following link:

<https://circuitverse.org/simulator/4-bit-mips-alu#>

A few more simulations used in the designing of 32-bit or 64-bit MIPS ALUs can be viewed in the following link

<https://circuitverse.org/users/22081>

Examples on the designed MIPS Simulator

Example 1 Write a program in MIPS Simulator to output the sum of all the numbers between 0-10 (inclusive).

```
1  #Program to find the sum of first 10 numbers
2      li $t0, 10
3      li $t3, 1
4  loop:
5      beq $t0, $zero, end
6      nop
7      add $t1, $t1, $t0
8      sub $t0, $t0, $t3
9      j loop
10     nop
11  end:
12     li $v0, 1
```

Output: - \$t1 = 55 \$t0 = 0 \$v0 = 1

Example 2 Write a program to find the factorial of a number using MIPS Simulator.

1 #program to find the factorial of a number	Name	Value	Hex Value	Binary Value
2 li \$t0, 5	\$zero	0	0	0
3 li \$t1, 1	\$at	0	0	0
4 li \$t3, 1	\$v0	1	1	1
5 loop:	\$v1	0	0	0
6 beq \$t0, \$zero, end	\$a0	0	0	0
7 nop	\$a1	0	0	0
8 mul \$t1, \$t1, \$t0	\$a2	0	0	0
9 sub \$t0, \$t0, \$t3	\$a3	0	0	0
10 j loop	\$t0	0	0	0
11 nop	\$t1	120	78	1111000
12 end:				
13 li \$v0, 1				
14 li \$t3, 0				

Input: - \$t0 = 5 Output: - \$t1 = 120

Example 3 Write a MIPS program to find the number of digits in a number.

```
1  #Program find no.of digits in a number
2      li $t0, 123456
3      li $s0, 10
4      li $t1, 0
5  loop:
6      beq $t0, $zero, exit
7      divd $t0, $t0, $s0
8      addi $t1, $t1, 1
9      nop
10     j loop
11 exit:
12     li $v1, 1
13
```

Input \$t0 = 123456 Output \$t1 = 6

Example 4 Write a MIPS program to multiply a number by 7 without using multiplication operator (Using bitwise operator)

```
1  #Program to multiply a number by 7 using
2  #bitwise operator
3      li $t0, 7
4      li $t1, 1
5      sll $t2, $t0, 3
6      sub $t2, $t2, $t0
7      nop
```

Input \$t0 = 7 Output \$t2 = 49

Example 5 Write a MIPS Program to find the sum of all the digits of a number

Input \$t0 = 123456 Output \$t1 = 21


```

1  #Program to find the sum of all the digits
2  # of a given number
3      li $t0, 123456
4      li $t1, 0
5      li $s0, 10
6  loop:
7      beq $t0, $zero, exit
8      rem $t2, $t0, $s0
9      add $t1, $t1, $t2
10     nop
11     divd $t0, $t0, $s0
12     nop
13     j loop
14 exit:
15     li $v0, 1

```

Multi-cycle Implementation of MIPS

As we already saw in the pipelining section that the MIPS uses multi cycle implementation of instructions. But why not one-cycle?

The length of the clock cycle will always be determined by the slowest operation (*lw*, *sw*) even if the data memory is not used.

Single Cycle

- perform each instruction in 1 clock cycle
- clock cycle must be long enough for slowest instruction; therefore,
- disadvantage: only as fast as slowest instruction

Multi-Cycle

- break fetch/execute cycle into multiple steps
- perform 1 step in each clock cycle
- advantage: each instruction uses only as many cycles as it needs

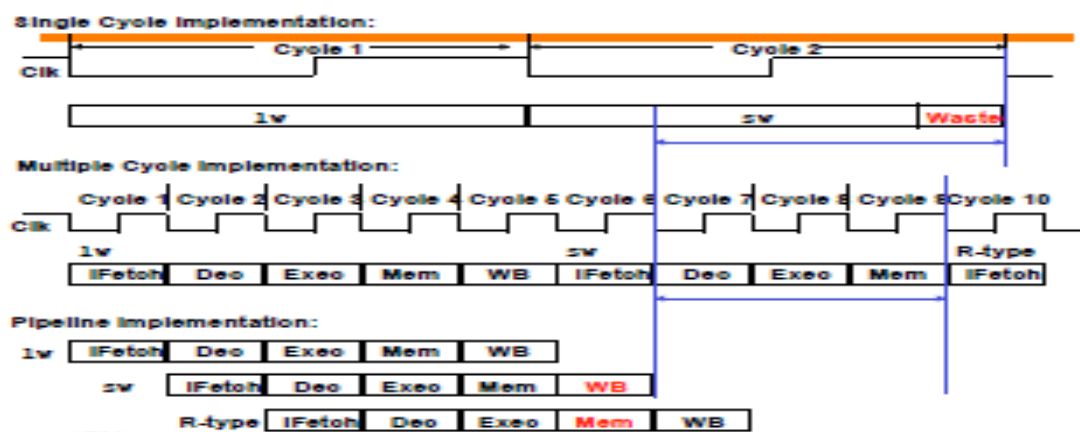


Fig. Comparison between Single and Multi-Cycle execution

Let us look at the steps required in multi-step cycle

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory
Instruction Decode	ID	Read source registers and generate control signals
Execute	EX	Compute an R-type result or a branch outcome
Memory	MEM	Read or write the data memory
Write back	WB	Store a result in the destination register

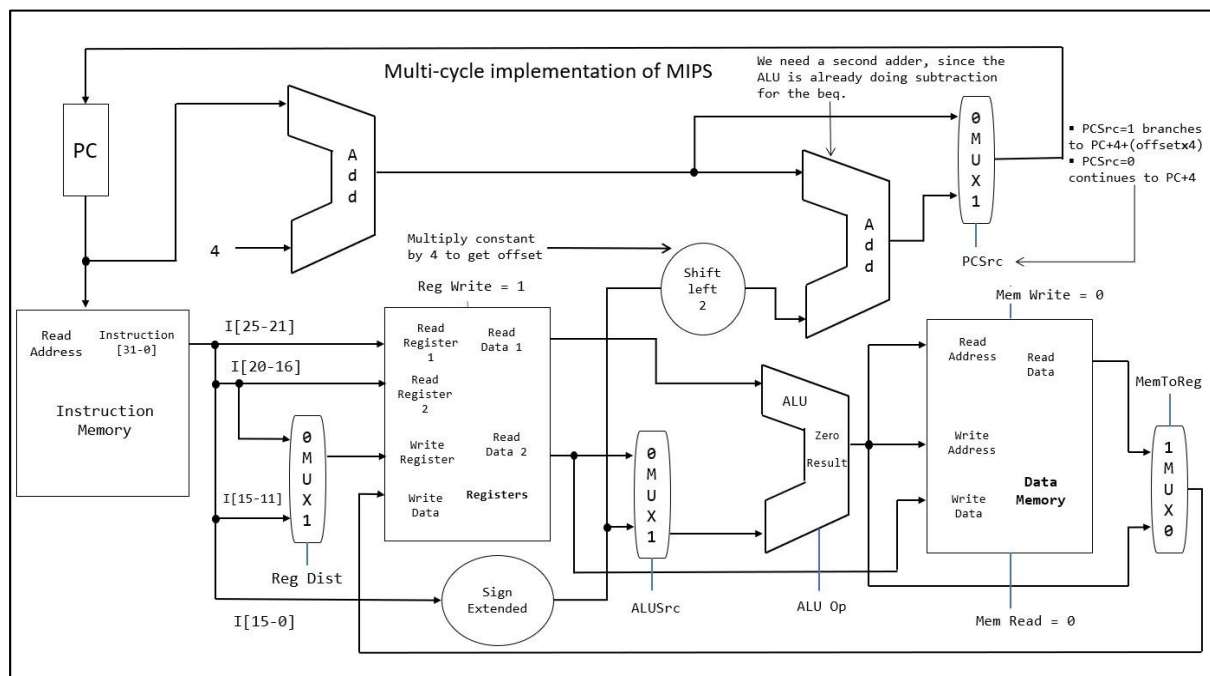


Fig. Multi-Cycle Implementation of MIPS

The Five Cycles of MIPS:

(Instruction Fetch)

$$IR := Memory[PC]$$

$$PC := PC + 4$$

(Instruction decode and Register fetch)

$$A := Reg[IR[25:21]], B := Reg[IR[20:16]]$$

$$ALUout := PC + sign - extend(IR[15:0])$$

(Execute | Memory address | Branch completion)

Memory reference: $ALUout := A + IR[15:0]$

R – type (ALU): $ALUout := A \text{ op } B$

Branch: if $A = B$ then $PC := ALUout$

(Memory access | R-type completion)

LW: $MDR := Memory[ALUout]$

SW: $Memory[ALUout] := B$

R – type: $Reg[IR[15:11]] := ALUout$

(Write back)

LW: $Reg[[20:16]] := MDR$

The 5-steps required for various different instructions are listed below:

Instruction	Steps required					
Beq	IF	ID	EX			
R-type	IF	ID	EX			WB
Sw	IF	ID	EX	MEM		
lw	IF	ID	EX	MEM		WB

Conclusion

The MIPS simulator has been designed to perform almost all the instructions of MIPS keeping the actual architecture and instruction execution in mind. Also, the Simulator provides a good user interface for the users with an in-built IDE and step-by-step execution. The MIPS sums up all about the modern the RISC architecture. The report also contains details about how the architecture of an ALU is designed using an online open-source simulator.

To sum up, the 32-bit MIPS is a very important processor to study for the understanding of RISC and most of its instructions doesn't depend on memory which makes the execution even faster. The pipelining of MIPS makes it better and further reduces the execution time.

Appendix

The whole code can be found in the GitHub repository: -

<https://github.com/ujjawal-1999/MIPS-Simulator>

The deployed link for the simulator is: -

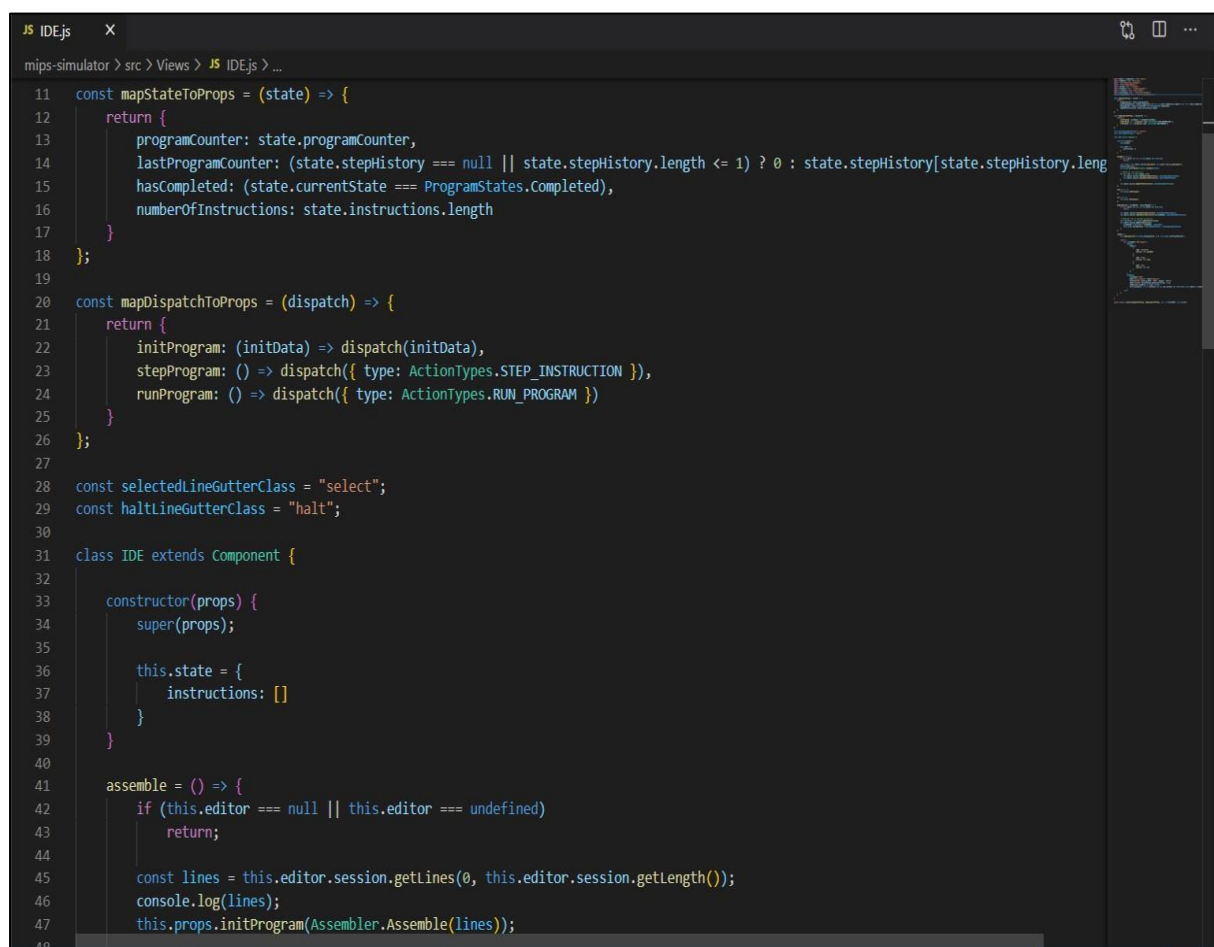
<https://5f1083804f6deee0fa75a21a--mips-simulator.netlify.app/>

A-1 The user interface of the Simulator

The MIPS Simulator user interface is web-based developed on the latest and popular JavaScript framework, ReactJs. The entire interface is designed into three components-

1. The Toolbar containing the execution buttons.
2. The IDE for writing codes.
3. The Register sections which shows all the registers with their updated values.

The IDE contains a MIPS assembler that highlights the MIPS instructions on its own so that the user knows if the instruction is correct or not. Moreover, the instruction that is being executed at a particular time is highlighted in yellow which becomes blue when the program comes to an end.

The image shows a screenshot of a VS Code editor window titled 'JS IDE.js'. The editor is displaying a JavaScript file with the following code:

```
11 const mapStateToProps = (state) => {
12   return {
13     programCounter: state.programCounter,
14     lastProgramCounter: (state.stepHistory === null || state.stepHistory.length <= 1) ? 0 : state.stepHistory[state.stepHistory.length - 1],
15     hasCompleted: (state.currentState === ProgramStates.Completed),
16     numberOfInstructions: state.instructions.length
17   }
18 };
19
20 const mapDispatchToProps = (dispatch) => {
21   return {
22     initProgram: (initData) => dispatch(initData),
23     stepProgram: () => dispatch({ type: ActionTypes.STEP_INSTRUCTION }),
24     runProgram: () => dispatch({ type: ActionTypes.RUN_PROGRAM })
25   }
26 };
27
28 const selectedLineGutterClass = "select";
29 const haltLineGutterClass = "halt";
30
31 class IDE extends Component {
32
33   constructor(props) {
34     super(props);
35
36     this.state = {
37       instructions: []
38     }
39   }
40
41   assemble = () => {
42     if (this.editor === null || this.editor === undefined)
43       return;
44
45     const lines = this.editor.session.getLines(0, this.editor.session.getLength());
46     console.log(lines);
47     this.props.initProgram(Assembler.Assemble(lines));
48   }
49 }
```

Fig. Code for the interface of the IDE section

```

JS IDE.js JS Registers.js X
mips-simulator > src > Views > JS Registers.js > Registers > render
12 class Registers extends Component {
13
14   render() {
15     const StyledTableCell = withStyles((theme) => ({
16       head: {
17         backgroundColor: theme.palette.common.black,
18         color: theme.palette.common.white,
19         textAlign: 'center'
20       },
21       body: {
22         fontSize: 16,
23         fontWeight: 'bolder',
24         textAlign: 'center'
25       },
26     }))(TableCell);
27
28     const StyledTableRow = withStyles((theme) => ({
29       root: {
30         '&:nth-of-type(even)': {
31           backgroundColor: 'lightgray'
32         },
33         '&:nth-of-type(odd)': {
34           backgroundColor: theme.palette.action.hover,
35         },
36       },
37     }))(TableRow);
38     const reg = this.props.registers;
39     const rows = reg;
40     return (
41       <>
42       <TableContainer component={Paper}>
43       <Table stickyHeader aria-label="sticky table">
44         <TableHead>
45           <TableRow>
46             <StyledTableCell>Name</StyledTableCell>
47             <StyledTableCell align="right">Value</StyledTableCell>
48             <StyledTableCell align="right">Hex Value</StyledTableCell>
49             <StyledTableCell align="right">Binary Value</StyledTableCell>

```

Fig. Code for the interface of the Registers section

A-2 The Registers and storage

The registers and storage has been implemented using Redux, which is a library for React that helps in passing props as a global variable, so it can be easily accessed by any Component very easily without having the need to worry about passing props every time.

The Actions that are performed internally is listed below

```

class ActionTypes {
  static INSTRUCTION = "Instruction";
  static PROGRAM_ASSEMBLE = "ProgramAssemble";
  static STEP_INSTRUCTION = "StepInstruction";
  static RUN_PROGRAM = "RUN_PROGRAM";
}
export default ActionTypes;

```

Fig. Action Types that are executed internally

Code for the 'StepInstruction' - This code is responsible for performing step-by-step executions.

```
case ActionTypes.STEP_INSTRUCTION:

    // If there are no instructions or the program has completed, don't do anything
    if (
        state.instructions === null
        || state.instructions.length === 0
        || state.currentState === ProgramStates.Completed
    ) {
        return state;
    }
    // console.log(state);
    // If the instruction being executed is out of range, do nothing
    if (state.programCounter >= state.instructions.length) {
        return {
            ...state,
            stepHistory: [...state.stepHistory, "halt"],
            currentState: ProgramStates.Completed
        }
    }

    // The next state is decided by the operation done on the current state by the
    // next instruction being executed
    console.debug(`Executing Instruction: ${state.programCounter}`, state.instructions[state.programCounter]);
    let nextState = (state.instructions[state.programCounter]).payload(state);

    // If program counter wasn't manually changed, go to the next instruction
    if (nextState.programCounter === state.programCounter) {
        nextState.programCounter += 1;
    }

    nextState.stepHistory = [...state.stepHistory, nextState.programCounter];
    nextState.currentState = ProgramStates.Running;

    return nextState;
case ActionTypes.PROGRAM_ASSEMBLE:
    return {
        ...state,
        instructions: action.payload.instructions,
        jumpTable: action.payload.jumpTable,
        currentState: ProgramStates.Assembled,
        programCounter: 0,
        lastProgramCounter: state.programCounter,
        registers: {...defaultRegisters}
    };
};
```

Fig. Code for the step-by-step execution of instructions

A-3 The main functions for particular instructions

The Simulator supports around 50 such instructions, so we will just discuss the codes of three instruction formats.

```
class Operations {
  static New(pattern, stateFunc) {
    pattern = pattern.replace(/#/g, '(.+)')
      .replace(/_/g, '\\s*');
    Operations.Instructions[pattern] = stateFunc;
  }
  static Nop() {
    return Operations.GetInstruction('nop');
  }
  static GetInstruction(userInput) {
    userInput = userInput.trim();
    const patterns = Object.keys(Operations.Instructions);
    let foundIndex = -1; let currentIndex = 0;

    let patternAsRegex;
    while (foundIndex < 0 && currentIndex < patterns.length) {
      patternAsRegex = new RegExp(patterns[currentIndex]);
      if (patternAsRegex.test(userInput)) { foundIndex = currentIndex; }
      currentIndex++;
    }
    // Replace null with a nop to get Nop when cant recognize instruction
    if (foundIndex < 0) { return Operations.GetInstruction('nop'); }
    // Calling this with the appropriate number of arguments will give us the state mutator
    const args = patternAsRegex.exec(userInput).slice(1).map((x) => x.trim());
    // Function that takes in arguments for an operation and returns a function
    // that takes in a state and mutates it based on the operation
    const stateFunc = Operations.Instructions[patterns[foundIndex]];
    ;
    const valueFunc = stateFunc.apply(null, args);

    return {
      type: ActionTypes.INSTRUCTION,
      payload: valueFunc
    }
  }
}
Operations.Instructions = {};
```

Fig. Code that leads to the execution of all the specific functions

R-Type Instruction

```
Operations.New(  
    'add _#_,_#_,_#_',  
    (rd, rs, rt) => {  
        return ((state) => {  
            return {  
                ...state,  
                registers: {  
                    ...state.registers,  
                    [rd]: parseInt(state.registers[rs]) + parseInt(state.regis  
ters[rt])  
                }  
            }  
        })  
    }  
);
```

I-Type Instruction

```
Operations.New(  
    'slti _#_,_#_,_#_',  
    (rd,rs,immediate)=>{  
        return((state)=>{  
            let v1 = parseInt(state.registers[rs]);  
            let v2 = parseInt(immediate);  
            return{  
                ...state,  
                registers:{  
                    ...state.registers,  
                    [rd]: (v1 < v2) ? 1 : 0  
                }  
            }  
        })  
    }  
);
```

J-Type Instruction

```
Operations.New(  
    'j _#_',  
    (label) => {  
        return ((state) => {  
            return {  
                ...state,  
                programCounter: state.jumpTable[label]  
            }  
        });  
    }  
);
```


A-4 List of Instructions Supported by the Simulator

<i>add</i>	<i>divu</i>	<i>and</i>	<i>bgez</i>
<i>addi</i>	<i>divd</i>	<i>andi</i>	<i>bltz</i>
<i>addu</i>	<i>sll</i>	<i>nor</i>	<i>bgtz</i>
<i>addiu</i>	<i>sllv</i>	<i>or</i>	<i>j</i>
<i>sub</i>	<i>srl</i>	<i>ori</i>	<i>jal</i>
<i>subu</i>	<i>srlv</i>	<i>xor</i>	<i>move</i>
<i>mul</i>	<i>slt</i>	<i>xori</i>	<i>li</i>
<i>mult</i>	<i>slti</i>	<i>beq</i>	<i>nop</i>
<i>multu</i>	<i>sltu</i>	<i>bne</i>	<i>rem</i>
<i>div</i>	<i>sltiu</i>	<i>blez</i>	

References

1. William Stallings; Computer Organization and Architecture - Designing for performance; Eight Edition
2. David A. Peterson, John L. Hennessy; Computer Organization and Design- The Hardware/ Software Interface; Fifth Edition.
3. Circuit Verse – Online Electronic Circuit Simulator <https://circuitverse.org/>
4. Prof. James L. Frankel; MIPS Instruction Set. Harvard University
5. MIPS Technologies - MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set
6. Charles W. Kann - Introduction To MIPS Assembly Language Programming. Gettysburg College
7. Wikipedia – MIPS Architecture Processors.
https://en.wikipedia.org/wiki/MIPS_architecture_processors
8. MIPS Instructions - <https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/MIPS%20Instruction%20Set.pdf>
9. React- Redux Documentation - <https://redux.js.org> <https://reactjs.org/docs>
10. Brace – Browser compatible version of ace-editor
<https://github.com/thlorenz/brace>