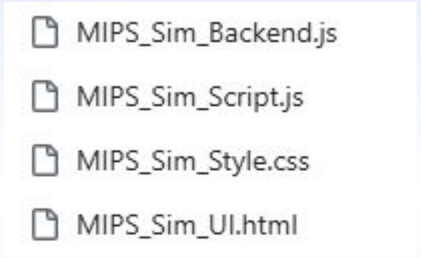
Course Project: MIPS Simulator

Team Members:
Daniel Cabezas, Wyatt Churchman, Jackson Berdou, Srikar Andhavarapu

Project Overview

The goal of the project was to develop a simulator that will execute MIPS instructions and the results can be seen by looking in the register file and memory to check that the program has executed as intended.



MIPS_Sim_Backend.js

MIPS_Sim_Script.js

MIPS_Sim_Style.css

MIPS_Sim_UI.html

We constructed the script in javascript, the UI was made in html, and the back end was constructed in javascript

Describe the simulator

- This project demonstrates how MIPS instructions execute at a low level.
- It helps visualize computer architecture concepts such as the register file, memory addressing, and the program counter.
- The simulator provides insight into how assembly instructions translate into hardware-level operations.

```
fflush(stdin);
scanf("%[^\n]",R.note);
if(fwrite(&R,sizeof(R),1,fp)){
    gotoxy(5,12);
    puts("Note is saved sucessfully");
    fclose(fp);
}else{
    gotoxy(5,12);
    SetColor(12);
    puts("\aFail to save!!\a");
    ClearColor();
}
gotoxy(5,15);
printf("Press any key............");
getch();
fclose(fp);
```

Structure of our Program

1. Register File

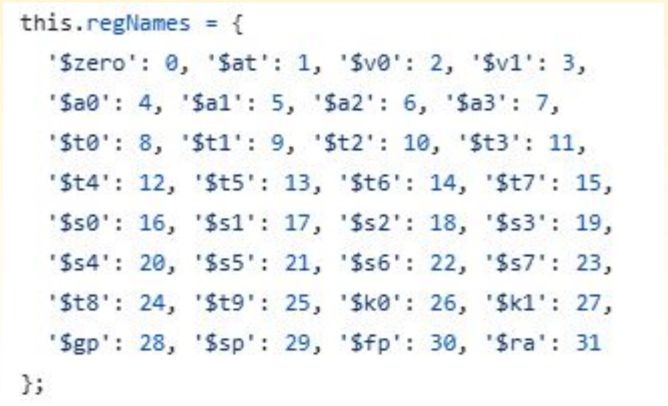
Our simulator keeps a JavaScript object that maps register names like `$zero`, `$t0`, `$s0`, `$ra` to their numeric indices.
This lets the backend decode instructions quickly, update the correct register, and display the results in the UI.

2. ALU

The ALU is implemented in JavaScript and handles all arithmetic and logical operations used by MIPS instructions.
For example: ADD, SUB, ADDI, MULT, and the operations that update HI and LO.

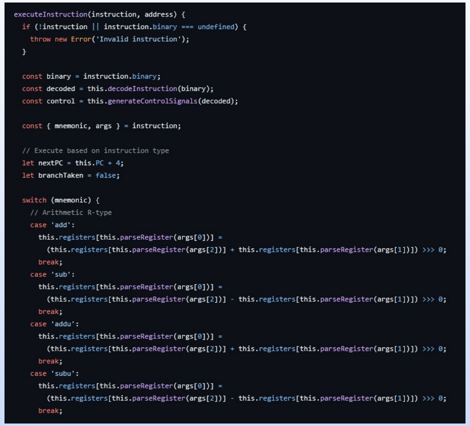
The ALU returns results that the backend writes into the register file or into memory.

Registers



```
this.regNames = {
  '$zero': 0, '$at': 1, '$v0': 2, '$v1': 3,
  '$a0': 4, '$a1': 5, '$a2': 6, '$a3': 7,
  '$t0': 8, '$t1': 9, '$t2': 10, '$t3': 11,
  '$t4': 12, '$t5': 13, '$t6': 14, '$t7': 15,
  '$s0': 16, '$s1': 17, '$s2': 18, '$s3': 19,
  '$s4': 20, '$s5': 21, '$s6': 22, '$s7': 23,
  '$t8': 24, '$t9': 25, '$k0': 26, '$k1': 27,
  '$gp': 28, '$sp': 29, '$fp': 30, '$ra': 31
};
```

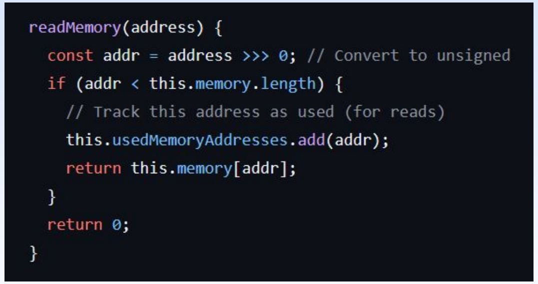
ALU



```
executeInstruction(instruction, address) {
  if (!instruction || instruction.binary === undefined) {
    throw new Error('Invalid instruction');
  }

  const binary = instruction.binary;
  const decoded = this.decodeInstruction(binary);
  const control = this.generateControlSignals(decoded);

  const { mnemonic, args } = instruction;

  // Execute based on instruction type
  let nextPC = this.PC + 4;
  let branchTaken = false;

  switch (mnemonic) {
    // Arithmetic R-type
    case 'add':
      this.registers[this.parseRegister(args[0])] =
        (this.registers[this.parseRegister(args[2])] + this.registers[this.parseRegister(args[1])]) >>> 0;
      break;
    case 'sub':
      this.registers[this.parseRegister(args[0])] =
        (this.registers[this.parseRegister(args[2])] - this.registers[this.parseRegister(args[1])]) >>> 0;
      break;
    case 'addu':
      this.registers[this.parseRegister(args[0])] =
        (this.registers[this.parseRegister(args[2])] + this.registers[this.parseRegister(args[1])]) >>> 0;
      break;
    case 'subu':
      this.registers[this.parseRegister(args[0])] =
        (this.registers[this.parseRegister(args[2])] - this.registers[this.parseRegister(args[1])]) >>> 0;
      break;
```

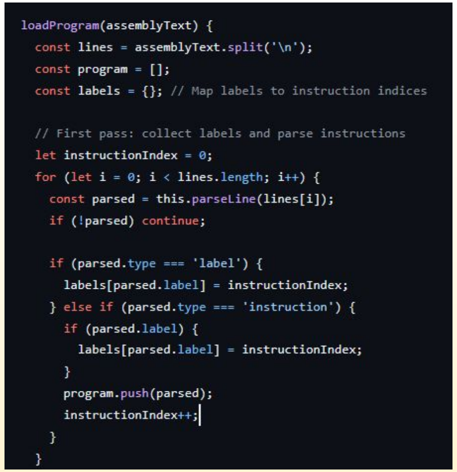
Structure of our Program cont..

Memory



```
readMemory(address) {
  const addr = address >>> 0; // Convert to unsigned
  if (addr < this.memory.length) {
    // Track this address as used (for reads)
    this.usedMemoryAddresses.add(addr);
    return this.memory[addr];
  }
  return 0;
}
```

Instructions



```
loadProgram(assemblyText) {
  const lines = assemblyText.split('\n');
  const program = [];
  const labels = {}; // Map labels to instruction indices

  // First pass: collect labels and parse instructions
  let instructionIndex = 0;
  for (let i = 0; i < lines.length; i++) {
    const parsed = this.parseLine(lines[i]);
    if (!parsed) continue;

    if (parsed.type === 'label') {
      labels[parsed.label] = instructionIndex;
    } else if (parsed.type === 'instruction') {
      if (parsed.label) {
        labels[parsed.label] = instructionIndex;
      }
      program.push(parsed);
      instructionIndex++;
    }
  }
```

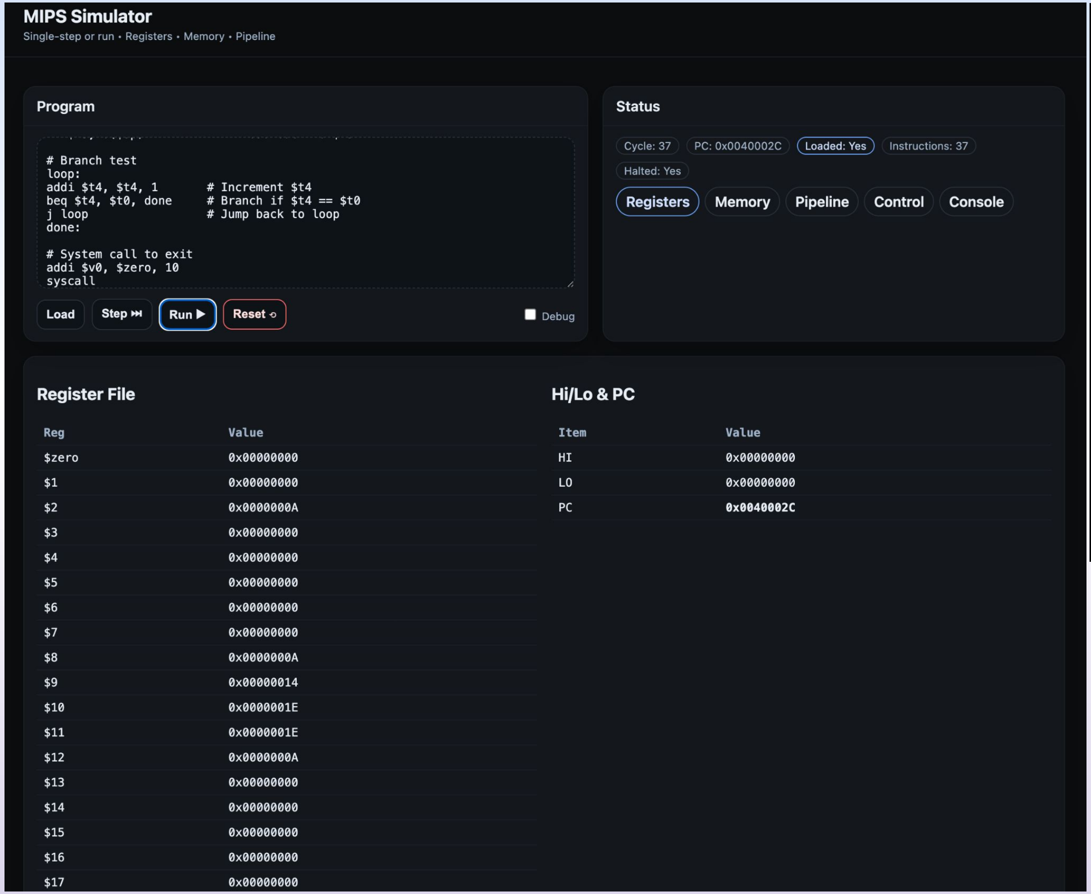
Memory

- Implemented as a 64 KB Uint8Array (byte-addressable)
- Helper functions readWord and writeWord build 32-bit words from four bytes
- We record every address that is read or written so the UI can show the "used memory" range

Instructions

- We first parse the assembly into mnemonics, arguments, and labels
- A second pass encodes each instruction into a 32-bit machine code word and assigns a PC address
- All encoded instructions are stored in an array; the PC index selects which one to execute each cycle

Example Run



MIPS Simulator

Single-step or run • Registers • Memory • Pipeline

Program

```
# Branch test
loop:
addi $t4, $t4, 1       # Increment $t4
beq $t4, $t0, done     # Branch if $t4 == $t0
j loop                 # Jump back to loop
done:

# System call to exit
addi $v0, $zero, 10
syscall
```

Status

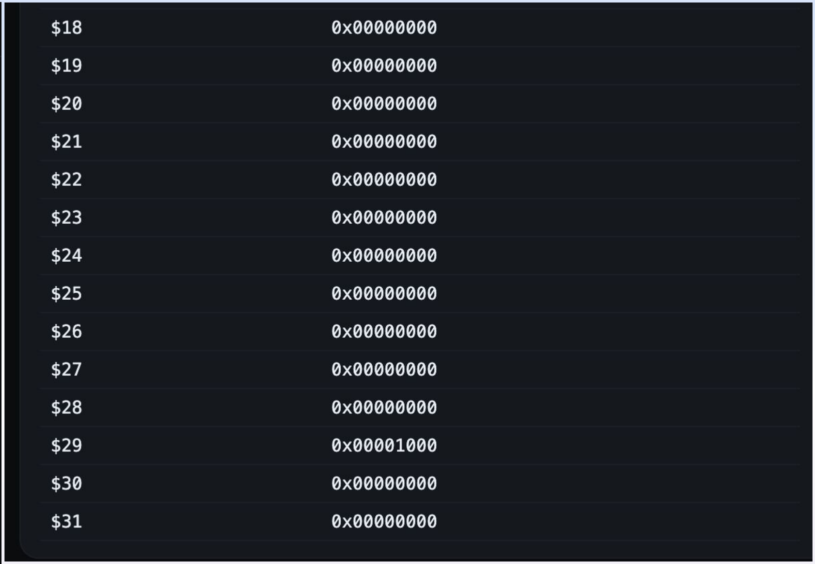
Cycle: 37 | PC: 0x0040002C | Loaded: Yes | Instructions: 37 | Halted: Yes

Register File

Reg	Value
$zero	0x00000000
$1	0x00000000
$2	0x0000000A
$3	0x00000000
$4	0x00000000
$5	0x00000000
$6	0x00000000
$7	0x00000000
$8	0x0000000A
$9	0x00000014
$10	0x0000001E
$11	0x0000001E
$12	0x0000000A
$13	0x00000000
$14	0x00000000
$15	0x00000000
$16	0x00000000
$17	0x00000000

Hi/Lo & PC

Item	Value
HI	0x00000000
LO	0x00000000
PC	0x0040002C

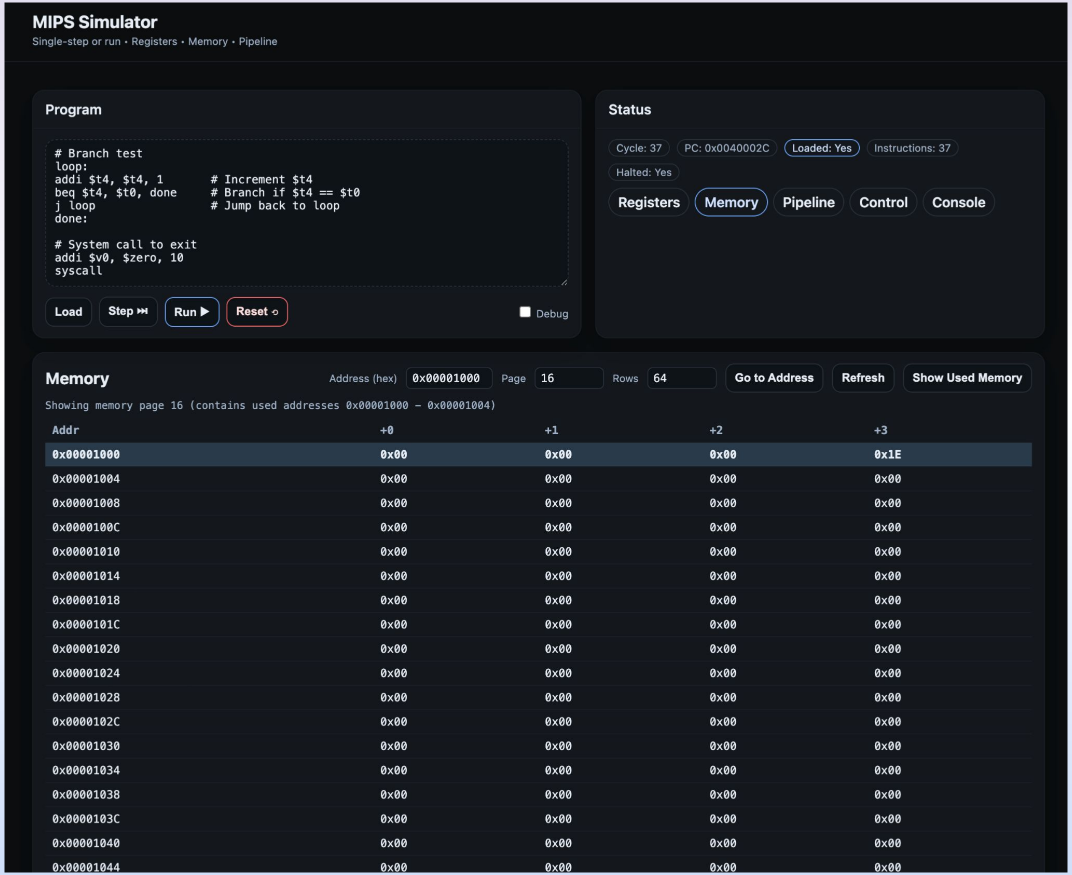


$18	0x00000000
$19	0x00000000
$20	0x00000000
$21	0x00000000
$22	0x00000000
$23	0x00000000
$24	0x00000000
$25	0x00000000
$26	0x00000000
$27	0x00000000
$28	0x00000000
$29	0x00001000
$30	0x00000000
$31	0x00000000

Code that was tested

```
# Simple addition program
addi $t0, $zero, 10    # $t0 (register 8) = 10
addi $t1, $zero, 20    # $t1 (register 9) = 20
add $t2, $t0, $t1      # $t2 (register 10) = $t0 + $t1
# Load/Store test
addi $sp, $zero, 0x1000   # Set stack pointer
sw $t2, 0($sp)            # Store $t2 to memory
lw $t3, 0($sp)            # Load back to $t3
# Branch test
loop:
addi $t4, $t4, 1       # Increment $t4
beq $t4, $t0, done     # Branch if $t4 == $t0
j loop                 # Jump back to loop
done:
# System call to exit
addi $v0, $zero, 10
syscall
```

Example Run cont…



```
# Branch test
loop:
addi $t4, $t4, 1       # Increment $t4
beq $t4, $t0, done     # Branch if $t4 == $t0
j loop                 # Jump back to loop
done:

# System call to exit
addi $v0, $zero, 10
syscall
```

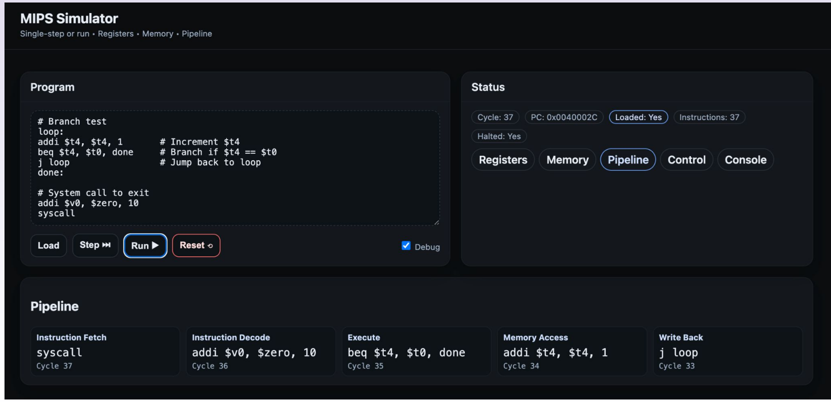
Debug: ☐

Cycle: 37 | PC: 0x0040002C | Loaded: Yes | Instructions: 37 | Halted: Yes

Address (hex): 0x00001000 | Page: 16 | Rows: 64

```
Showing memory page 16 (contains used addresses 0x00001000 – 0x00001004)
```

Addr	+0	+1	+2	+3
0x00001000	0x00	0x00	0x00	0x1E
0x00001004	0x00	0x00	0x00	0x00
0x00001008	0x00	0x00	0x00	0x00
0x0000100C	0x00	0x00	0x00	0x00
0x00001010	0x00	0x00	0x00	0x00
0x00001014	0x00	0x00	0x00	0x00
0x00001018	0x00	0x00	0x00	0x00
0x0000101C	0x00	0x00	0x00	0x00
0x00001020	0x00	0x00	0x00	0x00
0x00001024	0x00	0x00	0x00	0x00
0x00001028	0x00	0x00	0x00	0x00
0x0000102C	0x00	0x00	0x00	0x00
0x00001030	0x00	0x00	0x00	0x00
0x00001034	0x00	0x00	0x00	0x00
0x00001038	0x00	0x00	0x00	0x00
0x0000103C	0x00	0x00	0x00	0x00
0x00001040	0x00	0x00	0x00	0x00
0x00001044	0x00	0x00	0x00	0x00

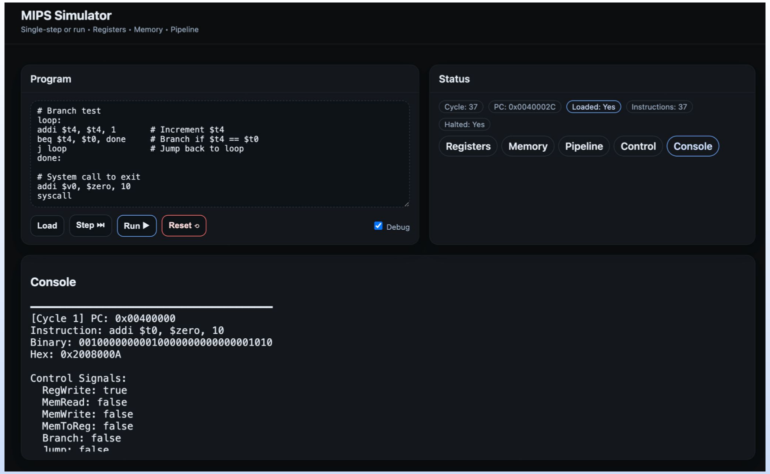


```
# Branch test
loop:
addi $t4, $t4, 1       # Increment $t4
beq $t4, $t0, done     # Branch if $t4 == $t0
j loop                 # Jump back to loop
done:

# System call to exit
addi $v0, $zero, 10
syscall
```

Debug: ☑

Cycle: 37 | PC: 0x0040002C | Loaded: Yes | Instructions: 37 | Halted: Yes

Instruction Fetch	Instruction Decode	Execute	Memory Access	Write Back
syscall	addi $v0, $zero, 10	beq $t4, $t0, done	addi $t4, $t4, 1	j loop
Cycle 37	Cycle 36	Cycle 35	Cycle 34	Cycle 33



```
# Branch test
loop:
addi $t4, $t4, 1       # Increment $t4
beq $t4, $t0, done     # Branch if $t4 == $t0
j loop                 # Jump back to loop
done:

# System call to exit
addi $v0, $zero, 10
syscall
```

Debug: ☑

Cycle: 37 | PC: 0x0040002C | Loaded: Yes | Instructions: 37 | Halted: Yes

```
────────────────────────────────
[Cycle 1] PC: 0x00400000
Instruction: addi $t0, $zero, 10
Binary: 00100000000010000000000000001010
Hex: 0x2008000A

Control Signals:
  RegWrite: true
  MemRead: false
  MemWrite: false
  MemToReg: false
  Branch: false
  Jump: false
```

Showcase of our MIPS Simulator

Q&A