

# 1 Design

A MIPS, single-instruction, multi-cycle processor (1) was implemented in C++ via the CLion IDE.

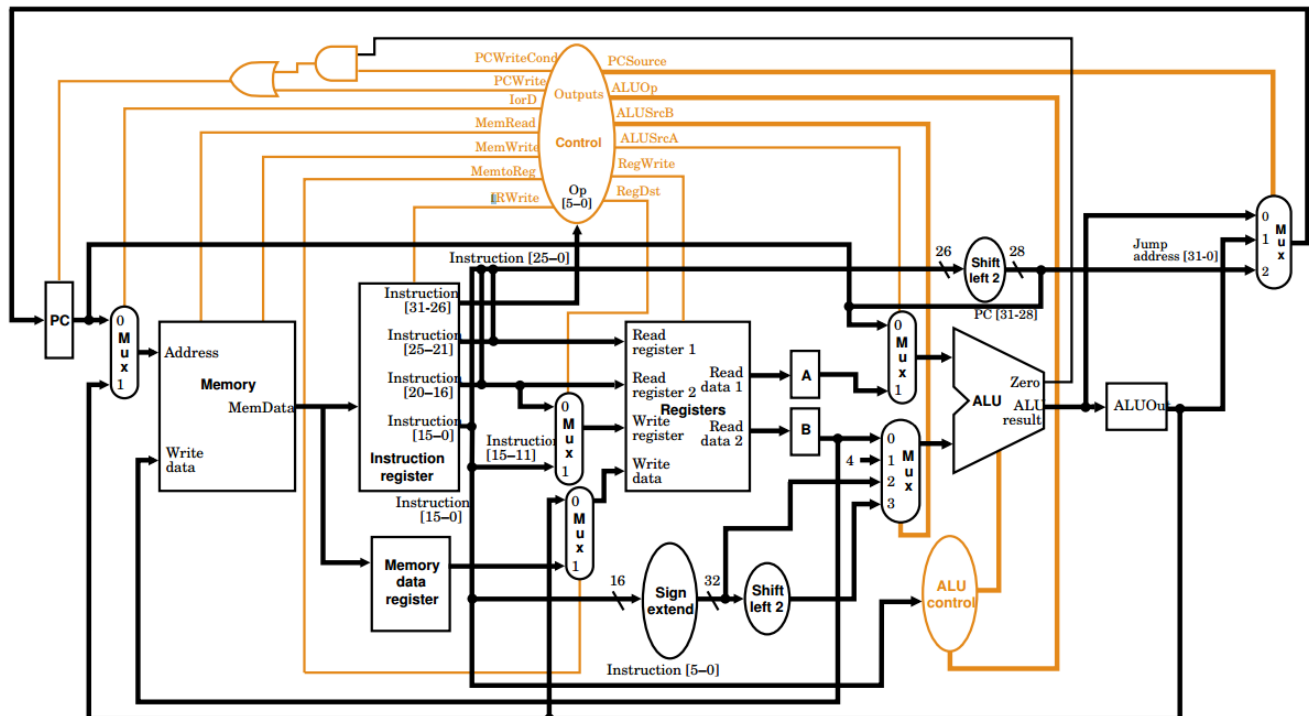


Figure 1: Multi-Cycle, single instruction MIPS processor schematic diagram.

The register topology (figure 2) used in a MIPS processor, was implemented. Only the register numbers are in the simulator, not the register names. `\$fp` and `\$gp` (`\$28` and `\$30`) were not implemented and `\$sp` was set to the maximum value in the PC (slightly changing its true function). All values in the register file are converted to decimal, for debugging purposes.

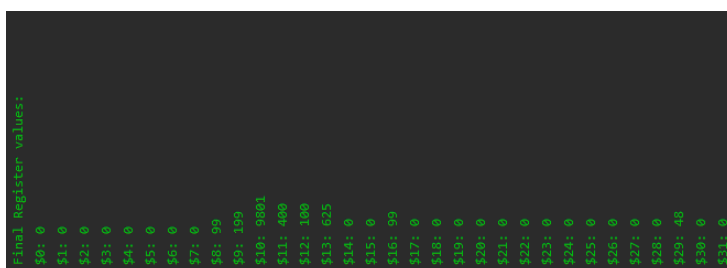


Figure 2: Register file structure.

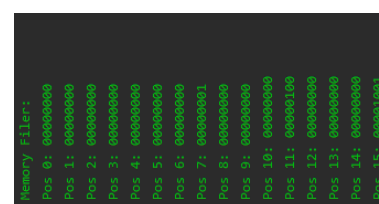


Figure 3: A section of the memory file.

The memory layout (figure 3) used did not mimic that used in a MIPS processor and instead simply started at 0 (with a maximum address value of 1000 although this could be increased by simply increasing the array size of `memoryFile`). The memory is still byte addressed and word aligned. The limited instruction set implemented include the following instructions:

- `beq`
- `add`
- `jr`
- `j`
- `jal`
- `sw`
- `lw`
- `addi`

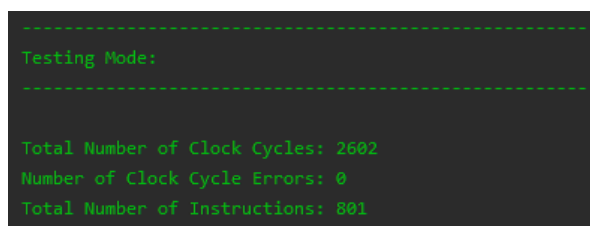
These instructions are sufficient to implement a fully functioning MIPS processor which can process the showcase program, along with leafed/non-leafed procedures and arrays.

## 2 Functionality

The system has an automated MIPS assembly to machine code translator. This is implemented by a class (**Translator**), which uses multiple methods to differentiate between the **IS** (Instruction Set) and produce the binary string instruction (machine code). The MIPS assembly `.txt` file is inputted into the **Translator** class.

The menu of the program allows you to pick from 3 different running modes, with each running mode being selected by entering the accompanying number into the terminal and pressing enter.

0. Picking option 0 will run through the program cycle by cycle. When an instruction is completed after a certain number of cycles, the instruction will be outputted to the console, letting the user know what instruction has just been performed. The values of all the registers in the register file is outputted after each clock cycle is executed. This could be very useful to the user when debugging their own MIPS assembly code.
1. Picking option 1 will simply run through the entire program and output the final values in memory along with the final values in the register file.
2. Picking option 2 will take the user to the testing suite (figure 4). This suite does not test their code but instead tests the simulator and shows information about the code that was just executed. This information includes
  - (a) The total number of clock cycles executed by the simulator
  - (b) The number of clock cycle errors
    - i. These would arise if an instruction (for example a j) would take more/less cycles than expected from the instruction
    - ii. During correct operation of the simulator, this value should be 0
  - (c) The total number of instructions executed



```
-----  
Testing Mode:  
-----  
  
Total Number of Clock Cycles: 2602  
Number of Clock Cycle Errors: 0  
Total Number of Instructions: 801
```

Figure 4: A snippet of the testing suite.

The output of the simulator consists of the contents inside the memory file, followed by a list containing the contents of the register file. The contents of the register file is what the user will usually use.

The processor consists of modules, and each module (**ALU**, **ALU out**, **Control**, **Instruction register**, **Memory file**, **Memory data register**, **PC**, **Register File**, **registers A** and **B**) has its own class, whose

job is to filter inputs and carry out any operations the specific module may do. This includes multiplexer operations (implemented by `switch`, `case` statements), arithmetic, binary sorting etc. Each of these modules are called by the `runInstructions` function. This incrementally steps through the instructions (based on the value in the **PC**) and calls other nested functions which carry out a single clock cycle/stage. At the beginning of each nested function, the control signals for that specific clock cycle are setup, ensuring that the correct dataflow occurs through the Datapath. These control signals are shown in appendix A. All instructions do stages 1 (`runIF`) and 2 (`runID`), the instruction fetch, and instruction decode stages. `runIF` ensures that the instruction register stores the instruction at the **PC** address. It then increments the **PC** by 4.

$$PC = PC + 4 \quad (1)$$

`runID` ensures that the temporary registers, **A** and **B**, store the **rs** and **rt** values for the next clock cycle. This also ensures that the **ALU Out** temporary register, stores the sign-extended, shifted left by 2, **PC** referenced value to also be used in the next clock cycle.

$$A = rs \quad B = rt \quad (2)$$

$$ALUOut = PC + (\text{sign-extend}(\text{imm}) \ll 2) \quad (3)$$

From here on, the cycles become instruction dependant. The opcode is used to differentiate between different instructions and from there, the correct nested functions can be called.

## 2.1 BEQ Command

The `beq` command (opcode 000100) is implemented using `runBeqEXE`. This carries out a referenced jump (the reference being the value of the **PC**), depending on whether or not the zero flag has been raised by the **ALU**. The inputs the **ALU** are **A** and **B**, and the **ALU** subtracts them letting us know whether they are equal if the zero flag is raised.

$$\text{if } (A == B), \quad PC = ALUOut \quad (4)$$

This instruction takes a total of 3 clock cycles.

## 2.2 ADD Command

The add command (opcode 000000) uses the `funct` section of its instruction (100000) to setup the correct **ALU** operation. In clock cycle 3, `runAddEXE` is called such that the values in **A** and **B** can added together to give the **ALU Out** temporary register value.

$$A + B = ALUOut \quad (5)$$

`runAddWB` is then called to write **ALU Out** to register **rd**, in the register file.

$$reg[rd] = ALUOut \quad (6)$$

This instruction takes a total of 4 clock cycles.

## 2.3 JR Command

The `jr` command (opcode 000000) also uses the `funct` section of its instruction (001000) to setup the correct **ALU** operation. In clock cycle 3, `runJrEXE` is called such that the values in **A** and **B** can added together to give the **ALU Out** temporary register value.

$$A + B = ALUOut \quad (7)$$

`runJrWB` is then called to write **ALU Out** to the **PC**, as a form of direct addressing.

$$PC = ALUOut \quad (8)$$

This instruction takes a total of 4 clock cycles.

## 2.4 J and JAL Commands

The `j` and `jal` commands (opcodes 000010 and 000011 respectively) **EXE** stage is implemented using the `runJEXE`. This carries out a direct jump, based on the values in the 26 bit immediate.

$$PC = PC[31 - 28] \mid (IR[25 - 0] \ll 2) \quad (9)$$

This instruction takes a total of 3 clock cycles.

## 2.5 Sw, Lw, and Addi Commands

The `sw`, `lw` and `addi` commands (opcodes 101011, 010111 and 001000 respectively) **EXE** stage is implemented using `runSwLwAddiEXE`. This sign-extends the immediate value, adds the value in the temporary register, **A**, and returns it to **ALU Out**.

$$ALUOut = A + sign-extend(imm) \quad (10)$$

Afterwards, the stages become instruction dependant.

### 2.5.1 Sw Command

`sw` uses `runSwMA` to write to memory, the value of register **B**, at memory position **ALU Out**.

$$memory[ALUOut] = B \quad (11)$$

This instruction takes a total of 4 clock cycles.

### 2.5.2 Lw command

`lw` passes the data in memory at the position **ALU Out** to the **MDR** using `runLwMA`.

$$MDR = memory[ALU Out] \quad (12)$$

It then uses `runLwWb` to write the value in the **MDR** to the register file at position, **rt**.

$$reg[rt] = MDR \quad (13)$$

This instruction takes a total of 5 clock cycles.

### 2.5.3 Addi Command

`addi` writes the value in **ALU Out** to the register file, also at position **rt**.

$$reg[rt] = ALU Out \quad (14)$$

## 3 Basic Testing

At the end of every function, there is a commented-out section which starts with ‘proof working’. All of these check that each clock cycle (depending on instruction) is performing the correct operation and producing the correct results. The correct results are all showcased in the above equations and this is what the ‘proof working’ checks for, that the equations are being fulfilled by the function.

Along with this, the clock cycle method (mode 0), was implemented early on to ensure that each clock cycle had the correct ‘proof working’ results, that each instruction had the correct number of clock cycles and (by outputting all of the values in the register file) that each register had the correct value associated with it. Using these in conjunction with the IDE debugger allowed me to fully debug any potential issues that could arise in the simulator.

I created 3 test programs to ensure correct operation of the simulator, all of which were checked with the mentioned methods I implemented to ensure correct operation of the simulator. These test programs checked that every instruction in my IS operated as expected. I initially used mode 2 to ensure that the number of clock cycles was correct for each instruction. I then used mode 0 to ensure that the register file values were correctly updating after each cycle. At the end of all of this, mode 1 could be used to check that the final register values were correct and that the result had been correctly stored. Full details of testing these programs are in appendix B.

## 4 Showcase Testing

The showcase program uses the following algorithm.

$$i_n = i_{n-1} + n, \quad \text{where } n = 1, 3, 5, 7, \dots \quad (15)$$

This program was initially written in C and then translated into MIPS assembly code, shown in figure 5.

```
1      addi $t1, $zero, 1
2      addi $s0, $zero, 99
3      LoopForA:
4      sw $t2, 0($t3)
5      addi $t3, $t3, 4
6      beq $t0, $s0, ExitA
7      add $t2, $t2, $t1
8      addi $t1, $t1, 2
9      addi $t0, $t0, 1
10     j LoopForA
11     ExitA:
12     addi $t4, $zero, 100
13     lw $t5, 0($t4)
```

Figure 5: MIPS assembly showcase program

This program created an array to store the results of the square of all of the values from 0 to 99. This is then stored in memory from the position 0. At the end of the program, is a load word. This was used to test if the correct answer to 25 squared was acquired. This is checked by looking at register **\$t5 (\$13)**, which gives us 625. **\$t2 (\$10)** gives us the final result, and should, therefore, be 99 squared, which is 9801.

I used mode 2 to see how many total clock cycles were required to implement the program, giving me 2602 clock cycles, as shown in figure 6. Further evidence of results is in appendix C.

```
-----
Testing Mode:
-----

Total Number of Clock Cycles: 2602
Number of Clock Cycle Errors: 0
Total Number of Instructions: 801
```

Figure 6: Testing mode showcase program output

## 5 Conclusion

Overall, I am very content with my final simulator. I believe it gives the user an easy to use interface with a lot of functionality. The different modes of use allow for methods of debugging for both the simulator itself along with the MIPS assembly code that has been inputted to it. All of the functionality I planned to achieve by the end of the coursework (with regards to the simulator) were accomplished and the code is presented in an easy to understand format, with comments and commented out extra testing for the user of the code to easily understand how the simulator works.

The simulator can successfully implement all of the testing programs along with the showcase program, while allowing the user to do the following tasks:

- Run the simulator cycle by cycle

- Run the simulator straight to the end
- Run a testing booth of the simulated MIPS code

If I were to continue making the processor, I would expand the IS to include more instructions such as `la`, `sll` etc. and enhance the capability of the translator to a wider IS. The testing booth could also potentially give more data with regards to the simulation, however, none of these improvements are necessities to the simulator and I believe that the finished simulator is more than a sufficient MIPS processor simulator.

# Appendix

## A Control Signals

The control signals used to control the simulator include:

- PCWriteCond
- PCWrite
- IorD
- MemRead
- MemWrite
- MemtoReg
- IRWrite
- PCSource
- ALUOp
- ALUSrcA
- ALUSrcB
- RegWrite
- RegDst



## B Detailed Testing

### B.1 Test Program 1

The output to the program should be in **\$s1 (\$17)** and it should be 125. This program produces 2 variables (x and y) and then sums them. If the *sum* == 25 (which it is), the program will then increment the sum 5 times in a for loop and store the result to **\$17 (125)**.

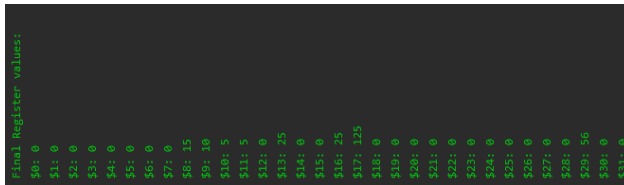


Figure 7: Register file final output for test program 1

```

1  beq $zero, $zero, main
2  If:
3  addi $t3, $zero, 5
4  ForLoop:
5  beq $t3, $t2, ExitFor
6  add $s1, $s1, $s0
7  addi $t2, $t2, 1
8  j ForLoop
9  main:
10 addi $t0, $zero, 15
11 addi $t1, $zero, 10
12 add $s0, $t0, $t1
13 addi $t5, $zero, 25
14 beq $s0, $t5, If
15 ExitFor:

```

Figure 8: Test Program 1 MIPS assembly code

### B.2 Test Program 2

This sets a variable *a* (**\$a0**) to 15, and a variable *b* to -12. It then runs a leaf procedure which increments the sum of the 2 variables.

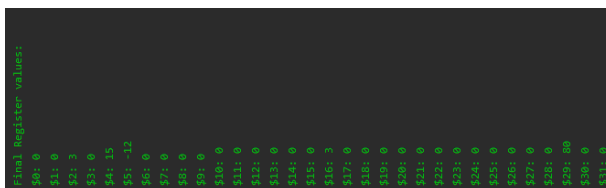


Figure 9: Register file final output for test program 2

```

1  beq $zero, $zero, main
2  procedureFunc:
3  addi $sp, $sp, -4
4  sw $s0, 0($sp)
5  add $s0, $a0, $a1
6  add $v0, $s0, $zero
7  lw $s0, 0($sp)
8  addi $sp, $sp, 4
9  jr $ra
10 DoIf:
11 addi $a1, $zero, 12
12 jal procedureFunc
13 j SkipElse
14 main:
15 addi $a0, $zero, 15
16 addi $t5, $zero, 0
17 beq $a0, $t5, DoIf
18 addi $a1, $zero, -12
19 jal procedureFunc
20 SkipElse:
21 add $s0, $v0, $zero

```

Figure 10: Test Program 2 MIPS assembly code

The expected result is in **\$s0 (\$16)** and should be 3.

### B.3 Test Program 3

This is a very simple program and checks that `addi` and `add` can use negative values.

```
Final Register values:
$0: 0
$1: 0
$2: 0
$3: 0
$4: 0
$5: 0
$6: 0
$7: 0
$8: -15
$9: -12
$10: 0
$11: 0
$12: 0
$13: 0
$14: 0
$15: 0
$16: -27
$17: 0
$18: 0
$19: 0
$20: 0
$21: 0
$22: 0
$23: 0
$24: 0
$25: 0
$26: 0
$27: 0
$28: 0
$29: 0
$30: 0
$31: 0
```

Figure 11: Register file final output for test program 3

```
1      addi $t0, $zero, -15
2      addi $t1, $zero, -12
3      add $s0, $t0, $t1
```

Figure 12: Test Program 3 MIPS assembly code

The expected result in **\$s0 (\$16)** is  $-27$ .

## C Showcase Program

This program gets the squares of all integers from 0 to 99. The final square result (99) is stored in **\$t2** (**\$10** → 9801) and the loaded square of 25 is stored in **\$t5** (**\$13**) and is 625.

This was used in conjunction with modes 0 and 2 for debugging.

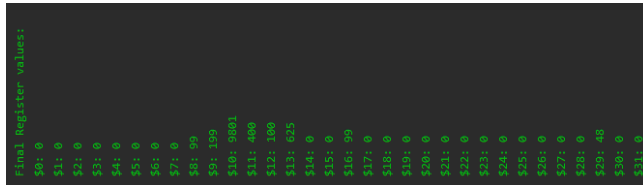


Figure 13: Register file final output for Showcase Program

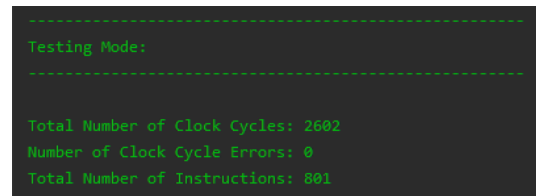


Figure 14: Showcase Program testing mode

```

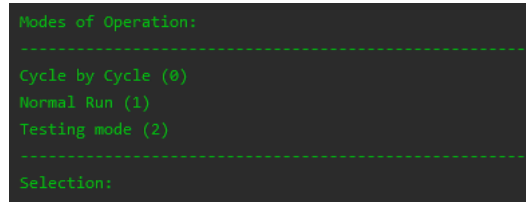
1      addi $t1, $zero, 1
2      addi $s0, $zero, 99
3      LoopForA:
4      sw $t2, 0($t3)
5      addi $t3, $t3, 4
6      beq $t0, $s0, ExitA
7      add $t2, $t2, $t1
8      addi $t1, $t1, 2
9      addi $t0, $t0, 1
10     j LoopForA
11     ExitA:
12     addi $t4, $zero, 100
13     lw $t5, 0($t4)
  
```

Figure 15: Showcase Program MIPS assembly code

## D Simulator Menu

The user can use this menu to input the mode they wish to use the simulator in.

- 0 → cycle-by-cycle
- 1 → normal run
- 2 → testing booth



```
Modes of Operation:
-----
Cycle by Cycle (0)
Normal Run (1)
Testing mode (2)
-----
Selection:
```

Figure 16: Simulator main menu