MIPS Architecture Simulation

Course Name: EG 212, Computer Architecture - Processor Design

Prakrititz Borah IMT2023547
Unnath Chittimalla IMT2023620
Mannat Kaur Bagga IMT2023071

26 February 2024

Abstract

Our Python-based MIPS simulator provides a platform for simulating the execution of MIPS binary code generated by external assemblers such as MARS. The simulator includes a processor and a compiler that can handle binary code, allowing users to write, compile, and execute MIPS programs within the simulated environment.

Introduction

Our simulator mimics the MIPS architecture using key components like the Program Counter (PC), Instruction Memory (IM), Register File (RF), Data Memory (DM), Arithmetic Logic Unit (ALU), and Control Unit (CU). Together, they recreate the behavior of a basic MIPS processor.

We've created a user-friendly MIPS simulation environment where users can work with MIPS binary code. They can input code from external assemblers, and our simulator runs it using the processor model.

We've included sample programs in the simulator, covering arithmetic, control flow, memory access, and data manipulation tasks. These programs showcase MIPS architecture in action and can be run smoothly within the simulator.

Our simulator supports the simulation of MIPS binary code, allowing users to experiment with code generated by external tools. This feature enables users to explore MIPS programming concepts in a controlled setting.

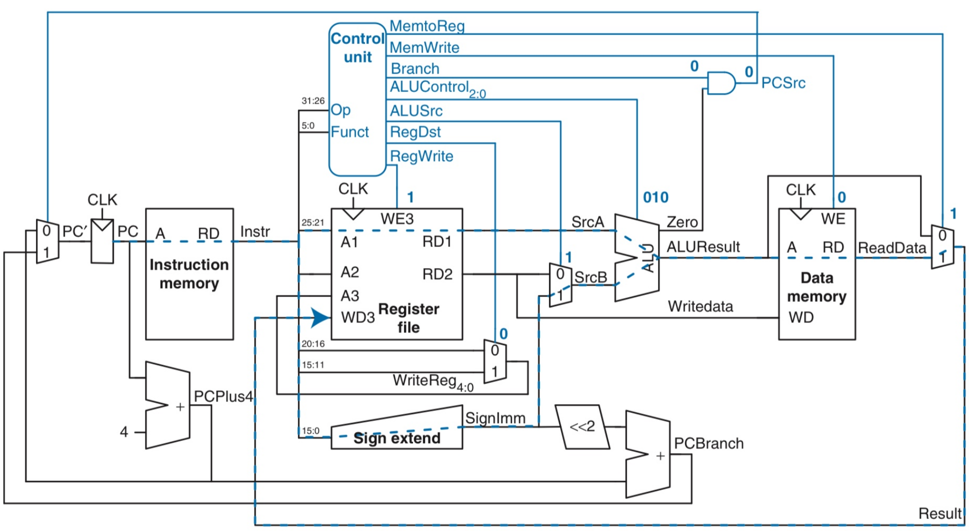

Figure 1: MIPS processor

Processor Implementation

The MIPS simulator is built with different parts that are separate but work together. The main part, known as the central processing unit (CPU), manages how instructions are carried out by coordinating with other important components:

- The Program Counter (PC) keeps track of which instruction is being worked on.
- The Instruction Memory (IM) stores all the instructions to be executed by the processor.
- The Register File (RF) contains several registers where data and temporary results are stored during program execution.
- The Data Memory (DM) serves as the computer's main memory, storing data.
- The Arithmetic Logic Unit (ALU) performs mathematical and logical operations on data.
- The Control Unit (CU) reads instructions from the Instruction Memory and directs the other components accordingly.

By having these components work together, our MIPS simulator functions like a real MIPS processor. It allows users to write, compile, and run MIPS programs in a simulated environment.

Program Counter (PC)

The Program Counter (PC) keeps track of the memory address of the current instruction being executed and controls the flow of instructions. It is responsible for updating the address of the next instruction based on the current program flow.

Here's an implementation of the `PC` class:

```
class PC:

  def __init__(self, value):
    self.value = value
    self.pcSrc = 0
    self.jump = 0

  def update(self, address, branchAddress):
    if not self.jump:
      if (self.pcSrc != 0):
        print('pc is branching')
        self.value = self.value + 4 + branchAddress
        print('pc is now', self.value)
      else:
        self.value += 4
    else:
      self.value = address << 2
```

Code Fig: 1: PC class Implementation

In the above implementation, the `PC` class maintains the current value of the program counter and provides a method to update it based on branching conditions and control signals.

Memory and Instruction Access

In MIPS architecture, memory is byte-addressable, meaning that each byte in memory has a unique address. This allows individual bytes to be accessed and manipulated independently.

Instructions in MIPS are typically stored as 32-bit words. Each word consists of 4 bytes. This byte-addressable nature of memory facilitates precise control over data and instructions at the byte level, enabling efficient memory management and access within the architecture.

Instruction Memory (IM)

The instruction memory (`IM`) in our simulator works with byte-addressable memory for storing instructions. Here's how it operates:

```python
class IM:
    def __init__(self):
        # Initialize instruction memory array with 0s
        self.mem = [0] * 0xffc
        print('Initialized Instruction Memory')
        self.RD = 0

    def RDPort(self, A):
        # Read 4 bytes (32 bits) from memory to form an instruction
        self.RD = int(
            num_to_8bit_binary(self.mem[A + 3]) +
            num_to_8bit_binary(self.mem[A + 2]) +
            num_to_8bit_binary(self.mem[A + 1]) + num_to_8bit_binary(self.mem[A]),
            2)
        print('Instruction fetched from memory: ', self.RD)
        return
```

Code Fig: 2: Instruction Memory Access

The memory is organized so that each element in the `mem` array represents a byte of memory. When fetching instructions, we access 4 bytes (32 bits) at a time to form a single instruction word. This allows us to effectively access and process instructions within the MIPS architecture.

By utilizing byte-addressable memory and accessing instructions as 32-bit words, our simulator accurately represents the memory organization and instruction access in MIPS architecture.

Register File

The Register File initializes control signals and registers, including the stack pointer.

```python
class RF:
    def __init__(self):
        self.WE3 = 0
        self.RD1 = 0
        self.RD2 = 0
        self.file = [0] * 32
        self.file[29] = 16380  # stack pointer location in DM
        self.RegDst = 0
        self.MemtoReg = 0
```

Reading from Register File: These methods read data from specified registers.

```python
    def RD1Port(self, A1):
        self.RD1 = self.file[A1]

    def RD2Port(self, A2):
        self.RD2 = self.file[A2]
```

Writing to Register File: This method writes data to specified registers based on control signals.

```python
    def WD3Port(self, rt, rd, ALUResult, RD):
        if (self.MemtoReg == 0):
            value = ALUResult
        elif (self.MemtoReg == 1):
            value = RD
        if (self.WE3):
            if (self.RegDst == 0):
                self.file[rt] = value
            elif (self.RegDst == 1):
                self.file[rd] = value
```

Code Fig: 3: Pseudo code of our implementation

Execute and Writeback

In this part, we take a closer look at how the processor works when it runs programs and handles data in its memory. We'll focus on two key parts: the Control Unit (CU) and the Arithmetic Logic Unit (ALU). These parts are like the brain and muscles of the processor, working together to process instructions and manage information

Control Unit (CU)

The Control Unit (CU) manages control signals and sets them based on the opcode and function code of the instruction being executed.

```
class CU:
  def __init__(self, RF, DM, ALU, PC):
    self.ALUControl = 0
    self.RF = RF
    self.DM = DM
    self.ALU = ALU
    self.PC = PC
    self.branch = 0

  def set_signals(self, opcode, funct):
    #set signal to different select lines
      ...
```

Code Fig: 4: Pseudo-code for Control Unit

The ALU decoder, as portrayed in Figure 2, plays a crucial role in interpreting standardized control signals. Its function ensures the adherence of any binary code to the MIPS architecture standards, thereby validating the integrity and compatibility of the instructions.

The tabular representation provided Figure 3 serves to elucidate the intricacies of the control unit signals. This comprehensive breakdown facilitates a deeper understanding of the control mechanisms governing the processor's operations.

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

Figure 2: ALU control signals

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Figure 3: Control Unit signals

The compiler, along with the memory initialization, is a crucial step in preparing the IAS computer for code execution. The provided Python script utilizes the `write_code` function to load machine code from `Machine.out` into memory. Additionally, specific memory locations are initialized to set the stage for program execution.

Arithmetic Logic Unit (ALU)

The ALU, short for Arithmetic Logic Unit, is a fundamental component of the processor responsible for performing arithmetic and logical operations on data.

```
class ALU:
  def __init__(self):
    self.ALUResult = 0
    self.Zero = 0
    self.Control = 0
    self.ALUSrc = 0
    self.srcA = 0
    self.srcB = 0

  def calculate(self, srcA, rtVal, imm):
    # Source Decision
    self.srcA = srcA
    if (self.ALUSrc == 0):
      self.srcB = rtVal
    elif (self.ALUSrc == 1):
      self.srcB = imm
    else:
      self.srcB = 0
    self.Zero = self.srcA - self.srcB

    # ALU operation
    if self.Control == -2:
      self.ALUResult = self.srcA * self.srcB
    if self.Control == 0b010:
      self.ALUResult = self.srcA + self.srcB
    elif self.Control == 0b110:
      self.ALUResult = self.srcA - self.srcB
    elif self.Control == 0b101:
      self.ALUResult = self.srcA * self.srcB
    elif self.Control == 0b000:
      self.ALUResult = self.srcA & self.srcB
    elif self.Control == 0b001:
      self.ALUResult = self.srcA | self.srcB
    elif self.Control == 0b111:
      if self.srcA < self.srcB:
        self.ALUResult = 0b1
      else:
        self.ALUResult = 0b0
    else:
      pass
```

Code Fig: 5: Pseudo-code for ALU

The ALU operates based on control signals and performs various arithmetic and logical operations on input data. It handles operations such as addition, subtraction, multiplication, bitwise AND, bitwise OR, and comparison.

For instance, in the provided code snippet:

- If `Control` represents addition (`0b010`), the ALU adds `srcA` and `srcB`.
- If `Control` represents subtraction (`0b110`), the ALU subtracts `srcB` from `srcA`.

The ALU's versatility and efficiency make it a vital component of modern processors, enabling them to perform complex computations and logical operations swiftly and accurately.

MIPS Implementation

This section will focus on the three programs that was written in MIPS assembly code : An array sum program, descending bubble sort and finally, a recursive factorial code that utilizes stack pointers. Note to be taken that 'mul' was implemented as an R-type instruction as MARS supports it without any pseudo instruction (according to MARS, it has a valid 32-bit instruction) **Also, MARS memory was configured to have .text from address 0** (using the Memory Configuration option in the 'Settings')

Array Sum Program

We present an implementation of an array sum program in both C and MIPS assembly language. The program calculates the sum of elements in an array using a loop structure.

```
.text
.globl main

main:
    # Array initialization
    li $t0, 1            # Load first array element
    li $t1, 2            # Load second array element
    li $t2, 3            # Load third array element
    li $t3, 4            # Load fourth array element
    li $t4, 5            # Load fifth array element

    # Calculate sum
    add $s0, $t0, $t1    # Add first and second element
    add $s0, $s0, $t2    # Add third element
    add $s0, $s0, $t3    # Add fourth element
    add $s0, $s0, $t4    # Add fifth element

    # Print sum
    li $v0, 1            # Load system call code for printing integer
    move $a0, $s0        # Move sum to argument register
    syscall              # Perform system call to print sum

    # Exit program
    li $v0, 10           # Load system call code for program exit
    syscall              # Perform system call to exit program
```

Code Fig: 6: MIPS Assembly Code for Array Sum

Initial data

First array element: 1, Second array element: 2, Third array element: 3, Fourth array element: 4, Fifth array element: 5

Output Results

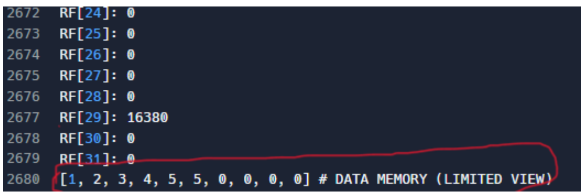

Figure 4: Initial Data Memory

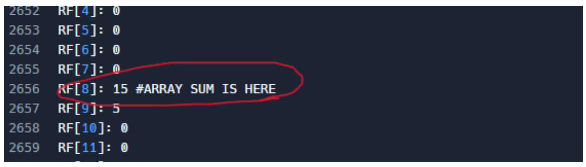

Figure 5: Output Result

Bubble Sort Program

We present an implementation of a bubble sort program in MIPS assembly language. The program sorts an array of integers in descending order using the bubble sort algorithm.

```
.text
.globl main
main:
    # Load array base address
    addi $s0,$zero, 0x2000
    # Load array size (4 times the number of elements in the array)
    addi $s1, $zero, 40
    addi $t0, $zero, 10
    sw $t0, 0($s0)
    addi $t0, $zero, 2
    sw $t0, 4($s0)

    addi $t0, $zero, 5
    sw $t0, 8($s0)

    ...  # Rest of the array initialization
    jal bubble_sort
    # Exit program
    addi $v0,$zero, 10       # syscall code for exit
    syscall

bubble_sort:
    addi $t0, $zero, 0          # i = 0
outer_loop:
    sub $t6, $s1, $t0           # t6 = n - i
    addi $t6, $t6, -4           # t6 = n - i - 4
    beq $t6, $zero, end_bubble # if n - i == 4, exit outer loop

    addi $t1, $zero, 0          # j = 0
inner_loop:
    beq $t1, $t6, next_outer    # if j == n - i - 4, go to next outer iteration

    # Load arr[j] into $t2
    add $t7, $s0, $t1            # base address + offset
    lw $t2, 0($t7)
    # Load arr[j+1] into $t3
    addi $t8, $t1, 4             # j + 4
    add $t8, $s0, $t8            # base address + offset
    lw $t3, 0($t8)

    # Compare arr[j] and arr[j+1]
    slt $at, $t2, $t3
    beq $at, $zero, no_swap

    # Swap arr[j] and arr[j+1]
    sw $t3, 0($t7)
    sw $t2, 0($t8)

no_swap:
    addi $t1, $t1, 4             # j += 4 (increment by word size)
    j inner_loop

next_outer:
    addi $t0, $t0, 4             # i += 4 (increment by word size)
    j outer_loop

end_bubble:
    jr $ra                        # return to caller
```

Code Fig: 7: MIPS Assembly Code for Bubble Sort

Initial Data

First array element: 10, Second array element: 2, Third array element: 5, Fourth array element: 4, Fifth array element: 3, Sixth array element: 6, Seventh array element: 7, Eighth array element: 8, Ninth array element: 4, Tenth array element: 10

Output Results

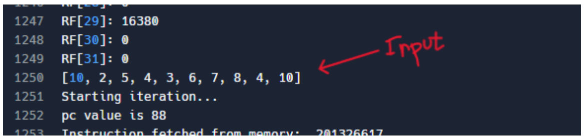

Figure 6: Initial Data Memory

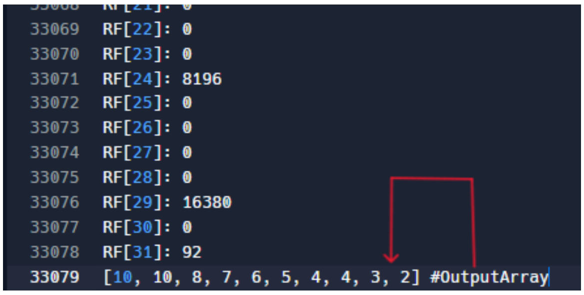

Figure 7: Output Result

Recursive Factorial

The following MIPS assembly code demonstrates the recursive calculation of factorial.

```
.text
addi $t1, $zero, 1
addi $a0, $zero, 6
jal fact
add $s0, $zero, $v0
addi $v0, $zero, 10
syscall
fact:
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
slt $t0, $a0, $t1
beq $t0, $zero, L1
addi $v0, $zero, 1
addi $sp, $sp, 8
jr $ra
L1:
addi $a0, $a0, -1
jal fact
lw $a0, 0($sp)
lw $ra, 4($sp)
addi $sp, $sp, 8
mul $v0, $a0, $v0
jr $ra
```

Code Fig: 8: MIPS Assembly Code for Recursive Factorial

Results

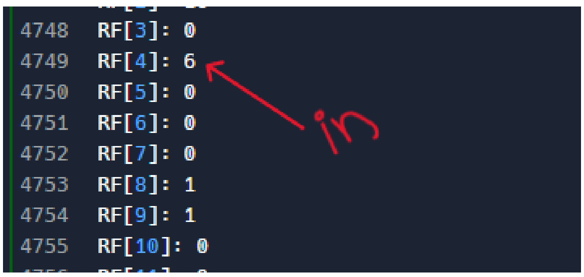

Figure 8: 6 is given as input to factorial function

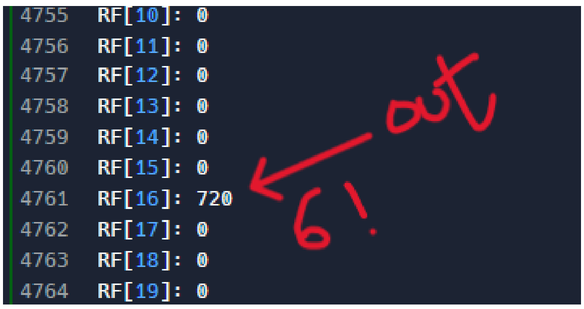

Figure 9: Result 720 can be seen (6! = 720)