Assignment–2: Mips Processor Design

Course Name: 212 Computer Architecture – Processor design
Student Name: Jayant Sharma, Dedeepya Avancha
Student ID: IMT2023523, IMT2023006

February 2024

1 Introduction

We have simulated the 5-stage MIPS non-pipelined processor. We chose Insertion Sort, Factorial and linear search for our assignment.

Insertion Sort:

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time.
Algorithm: Start from the second element: Assume that the first element is already sorted. Start iterating from the second element in the array. Compare and insert: Compare the current element with the elements before it. Move larger elements to the right and insert the current element into the correct position. Repeat: Continue this process for each element in the array until the entire array is sorted.

Listing 1: Code Snippet: cpp for Loop of insertion sort

```
for (int i = 1; i < n; ++i){
        int value = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > value){
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = value;
    }
```

Factorial:

Factorial of a non-negative number, denoted by 'n!', is a product of all positive integers less than or equal to 'n'. Mathematically, it is defined as $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$
Algorithm: It is a simple iterative algorithm. A variable 'result' is initialized to 1. In each iteration of the for loop, a variable 'i', going from 1 to 'n', is multiplied to 'result'. At the end of the iteration, we get the factorial which is returned to the main.

Listing 2: Code Snippet: cpp for Loop of factorial

```
int result = 1;
    for (int i = 1; i <= n; ++i){
        result *= i;
    }
    return result;
```

Linear Search:

Linear search is the most basic search algorithm, implemented by iterating over each element in the array. It returns the index of the search value in the array. If the value is not found, it returns -1.
Algorithm: Initialize the 'index_result' to -1. Start iterating over each element in 'array'. Compare the current element with the search value. If they are equal, update index_result with the current index and break out of the loop. If the search value does not match any element in the array, -1 is returned.

Listing 3: Code Snippet: cpp for Loop of factorial

```
int index_result = -1;
for (int i = 0; i < array.size(); ++i){
    if (array[i] == search_value){
        index_result = i;
        break;
    }
}
```

2 Explanation of our Implementation

The files uploaded are:
1. MIPS_processor.py
2. MIPS_components.py
3. main.py
4. insertion_sort.asm
5. factorial.asm
6. search.asm
7. insertion_sort.txt
8. factorial.txt
9. search.txt
10. insertion_sort.c
11. factorial.c
12. search.c

- MIPS_processor.py

  This code defines a Python class Processor representing a MIPS processor. It includes components such as memory, register file, ALU, program counter and all of control signals. The class has methods for fetching, decoding, executing, memory access, and write back stages, based on the MIPS instruction set architecture.

  It has a method for the control unit that sets all the control signals according to the instructions it has decoded and calls the appropriate execute function.

  IF stage happens in the fetching method, it takes no parameters and returns a 32-bit binary representation of the fetched instruction. ID stage happens in the decoding method, it returns a dictionary containing the parsed fields of the instruction. EX stage is split into three functions to execute r-type, i-type and j-type instructions. Finally, Mem access and WB stages also have separate functions that run based on the control signals.

  MIPS_processor.py also includes methods for storing instructions from a file in memory and printing the state of the processor. While the processor is running, it prints the PC state, instruction fields, control signals and the instruction it is executing for better understanding of the simulation.

```
PC: 4194336
Instruction returned from fetch cycle is 00010000001000000000000000010101
parsed instruction is {'opcode': '000100', 'rs': '00001', 'rt': '00000', 'rd': '00000', 'shamt': '00000', 'funct': '010101', 'address
': '00000100000000000000010101', 'immediate': '0000000000010101'}
Control Signals: regDst = 0, branch = 1, memRead = 0, memToReg = 0, aluOp = 1, memWrite = 0, aluSrc = 1, regWrite = 1 and jump = 0
 I type instruction
 Executing beq operation

PC: 4194424
Instruction returned from fetch cycle is 00100100000001000000000000001010
parsed instruction is {'opcode': '001001', 'rs': '00000', 'rt': '00010', 'rd': '00000', 'shamt': '00000', 'funct': '001010', 'address
': '00000000100000000000001010', 'immediate': '0000000000001010'}
Control Signals: regDst = 0, branch = 0, memRead = 0, memToReg = 1, aluOp = 0, memWrite = 0, aluSrc = 1, regWrite = 1 and jump = 0
 I type instruction
 Executing addiu operation
```

Figure 1: Mips Simulation

- MIPS_components.py

  This python code defines five classes for simulating components of a MIPS processor.

  Memory class represents memory in a MIPS processor. It allows reading from and writing to memory locations. Register_file class represents the register file in a MIPS processor. It provides methods to read from and write to registers, as well as to get the name of a register. ALU class represents the Arithmetic Logic Unit (ALU) in a MIPS processor. It can perform operations such as addition, subtraction, and set-less-than. SignExtender class represents the sign extension unit used for format the immediate field of I Type instructions. Left Shft by two represents the shifter used during the BTA/JTA calculation.

  Each class is initialized with certain attributes and provides methods to interact with those attributes, simulating the behavior of the respective components in a MIPS processor.

Listing 4: Code Snippet: cpp for Loop of factorial

```
class ALU:
    """Class for MIPS ALU with class method for executing operations"""

    def __init__(self):
        self.srcA = 0x00000000
        self.srcB = 0x00000000
        self.Zero = 0b0
        self.ALU_result = 0x00000000

    def execute_operation(self, opcode):
        if opcode == "100000":  # add operation
            self.ALU_result = self.srcA + self.srcB
        elif opcode == "100010":  # sub operation
            self.ALU_result = self.srcA - self.srcB
        elif opcode == "101010":  # slt operation
            if self.srcA < self.srcB:
                self.ALU_result = 1
            else:
                self.ALU_result = 0
        elif opcode == "001101": # or operation
            self.ALU_result = self.srcA | self.srcB
```

- main.py

This Python program imports the Processor class. We can run the Mips simulation by running this program. Input is taken to show the execution of one of the three programs. You can input 1, 2 or 3 to execute insertion sort, factorial and linear search respectively. Based on your input the memory is populated with the machine code corresponding to the program. The processor state before executing the instructions is printed. The execution of each program is shown instruction by instruction. Then based on the programs, their respective outputs are printed.

- insertion_sort.asm, factorial.asm and search.asm

  These files contain the assembly code implementations of our c programs.

- insertion_sort.txt, factorial.txt and search.txt

  The machine code generated by the mars simulator is written in these files.
  The format for the machine code is an 8-bit hexadecimal number indicating the address for the memory location followed by the 32-bit memory location itself.

Listing 5: Machine Code

```
0040005c    10101101000011100000000000000000
00400060    10101101000011001111111111111100
.....
```

- insertion_sort.c, factorial.c and search.c

  These files contain the c programs we have implemented.

3 Results and Discussion

The implementation of the MIPS processor simulator gave successful results when executing the chosen programs: Insertion Sort, Factorial, and Linear Search.
Insertion Sort: Upon executing the insertion sort program, the simulator accurately sorted the input array (which was hard coded into the memory) in ascending order, and stored the results in the data memory.

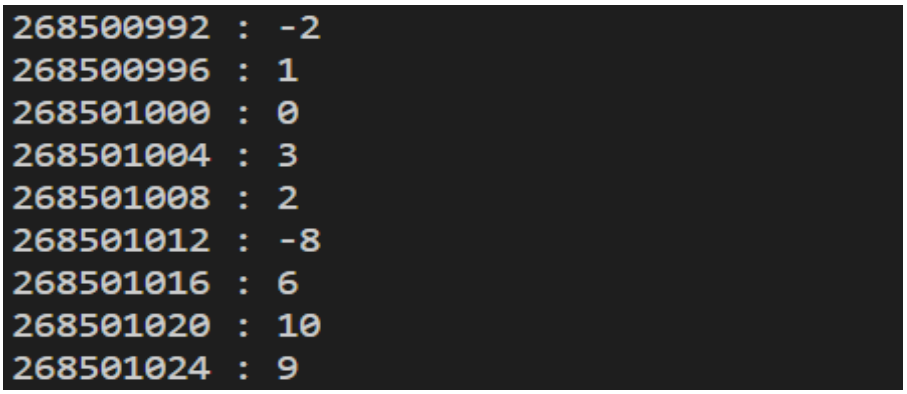

Figure 2: Mips Simulation: Insertion sort Input array

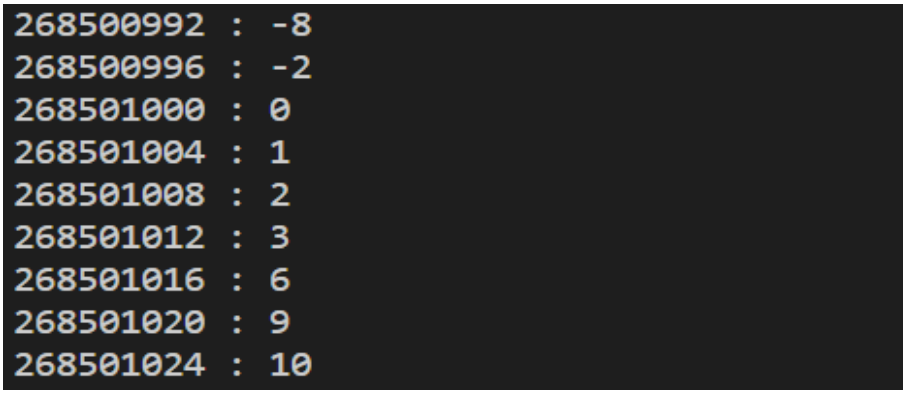

Figure 3: Mips Simulation: Insertion sort Output array

Factorial: The factorial program executed correctly, with the result being stored in the data memory.

Figure 4: Mips Simulation: Factorial result in Data memory

Linear Search: The linear search program successfully located the target value within the input array(which was hard coded), returning the index of the first occurrence, and storing it in $t2 of the register file. In case the target is not present in the input array, t2 was set to -1.

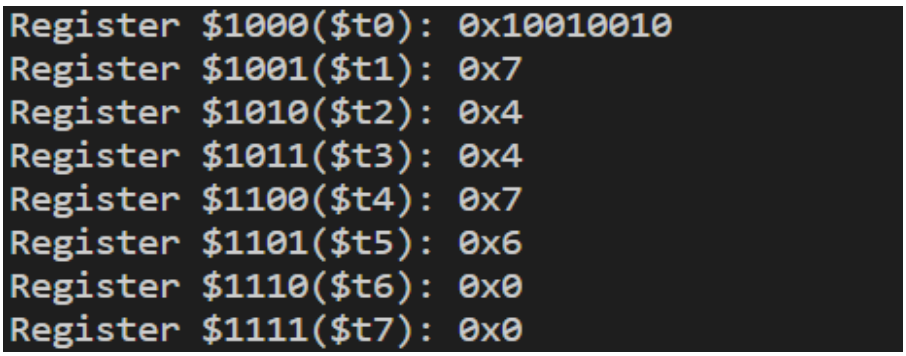

Figure 5: Mips Simulation: Search result in Reg file