Task 3 - MIPS Emulator

Casper Kristiansson

2022-02-01

Introduction

The third exercise of the course Programming II consisted of creating an emulator for MIPS. The goal of the task is implementing a handful of distinct functions such as halt, out, add, addi, sub, lw, sw, beq, and labels. Another goal of the task is to choose and implement different data structures to represent the register and memory. The author will be discussing the most vitals parts of the program and present a couple of different MIPS instructions and how they get executed.

Result

The author started off by choosing and implementing the different data structures that would be used to represent the register and memory. For the best data management, the author decided to use a tuple for the register which holds thirty-two different values and a tree data structure to represent the memory where the key is address and value is the value to be stored. This way when searching for a value based on the address the time complexity becomes O(log N). Elixir does have a build in function called "Map," but the author decided to implement the different tree methods manually. The methods to manage the instructions for Out was effortlessly develop by the author. It consisted of three methods, one for creating a Out which consisted of a single list, one which appends to the list and one which returns it.

The register part of the program consists of three different functions; one which creates an empty register, one to read the register and one for writing to the register. Note that to represent 32 0s the author uses (...) in the figure below.

```
def new() do {0,0,0,0,0,0,0,0,0,0,0,0...} end
def read(_, 0) do 0 end
def read(reg, index) do elem(reg, index) end
def write(reg, 0, _) do reg end
def write(reg, index, rs) do put_elem(reg, index, rs) end
```

Figure 1: The three methods of register

The tree structure consisted of four distinct functions, one for creating an empty tree, inserting, lookup, and traversal. The inserting method is used to write to the memory for example when executing the MIPS instruction Store Word. The lookup method is used to find values in the memory using address (key) to find it. The last method, traversal, is a method which finds a key using value, which is used when finding the PC address to jump to for a Branch if equals instruction. The author will only be disusing a few parts of the tree due to limitations of space and next assignment (Tree vs list) where the data structure will be discussed more in detail.

Figure two consists of one of the functions which is used to insert elements into an ordered tree. In figure two below the goal is to insert or update a node. If the key is equal to the key in the tree the goal is to update and insert the new value. If the key is either bigger or less than the new key the function continues to traverse the tree.

```
def tree_insert(k, e, {:node, key, value, left, right}) do
    cond do
      k == key -> {:node, k, e, left, right}
      k < key -> {:node, key, value, tree_insert(k, e, left), right}
      k > key -> {:node, key, value, left, tree_insert(k, e, right)}
    end
end
```

Figure 2: One of the methods for inserting into a tree

Figure three involves the method for finding a value based on the key (address). The method works exactly the same as the function in figure two. The only different is that if the key equals the key in the tree the goal is to return the value.

```
def tree_lookup(k, {:node, key, value, left, right}) do
    cond do
      k == key -> value
      k < key -> tree_lookup(k, left)
      k > key -> tree_lookup(k, right)
    end
end
```

Figure 3: One of the methods for the tree lookup

MIPS Instructions

The author chooses to represent the MIPS program using the structure {:addi, 5, 5, 0}. Using this structure, the author implemented a program which will navigate the different instructions which is aligned using four bytes. This means that when goal is to fetch more instructions the author divides the PC address with four to get the correct index from the instruction tuple.

```
def read_instruction(code, pc) do Enum.at(code, div(pc, 4)) end
```

Figure 4: Reading Instructions

The next part of the program is the actual main function which manages the different MIPS instruction. The author uses simple case condition to oversee the several types of methods. The commands that will be mentioned is add, lw and beq.

Figure five includes the method for the simple MIPS instruction add immediate. The goal of the instruction is to add a register value with an immediate value and store it in a register. For example {x = y + 5}. The program starts of by calling the function for reading a specific register using an index and adding that value to the immediate value. The program than stores the new value to a register.

```
{:addi, rt, rs, imm} ->
    s = Register.read(reg, rs)
    reg = Register.write(reg, rt, s + imm)

    pc = pc + 4
    run(pc, code, reg, mem, out)
```

Figure 5: Addi instruction

Figure six contains off the MIPS instruction Load word. The instruction starts off by reading a register value and adding that value with an immediate value. The program than gets the value that is stored on that address in the memory and stores that in a new register.

```
{:lw, rt, rs, imm} ->
    s = Register.read(reg, rs)
    value = Program.read_value(mem, s + imm)
    reg = Register.write(reg, rt, value)

    pc = pc + 4
    run(pc, code, reg, mem, out)
```

Figure 6: Lw Instruction

The last command that will be discussed is Branch if equal. For the command to work a label needs to be created. If two registers equal each other the program will jump to a specific instruction address (PC). The author had a bit of problem to solve the task because when the label command is being run the program saves the labels address as key and label name as value in the tree data structure. The problem is that because the label is stored as value the only way to find the address (key) is to traverse the entire tree until it finds the desired key which has a time complexity of O(n). Another way to solve this problem could be to store the label as address in the tree. But because the goal is that the memory should be aligned by four bytes the labels might not always work. Therefore, the author chooses to use the traverse method to find the address using a value (label).

```
{:beq, rs, rt, imm} ->
    s = Register.read(reg, rs)
    t = Register.read(reg, rt)

    if s == t do
        address = Program.read_address(mem, imm)
        pc = address + 4
        run(pc, code, reg, mem, out)
    else
        pc = pc + 4
        run(pc, code, reg, mem, out)
    end
```

Figure 7: Beq Instruction