Mips Emulator

Atheer Salim

Spring Term 2022

Simple Instructions

In this section lets talk about the more simple instructions that doesn't need memory to work they depends on the registers that we have such as add, sub, addi and some of the custom instructions such as halt and out are included here. Example of sub instruction

```
# reg(rd) := reg(rs) - reg(rt)
{:sub, rd, rs, rt} ->
  pc = pc + 4
  s = Register.read(reg, rs)
  t = Register.read(reg, rt)
  reg = Register.write(reg, rd, s - t)
  # go to next instruction
  run(pc, code, reg, mem, out)
```

Here we increment program counter `pc` and we read the value from the register using read command on Register module and we write to the register the new value in register `rd` then go to next instruction

Register

Register is very important for a MIPS emulator no surprises there, I choose to represent my registers in a binary search tree. Here is what happens when user creates a new register which happens before the first instruction of code is run.

```
def new() do
  regInListForm = [{0, 0}, {1, 0}, {2, 0}, {3, 0}, {4, 0}, {5, 0}, ... {31, 0}]
  createRegisterInTreeFormat(regInListForm, :nil)
end
```

Here we begin by having our register in a list and each register has it own tuple with {`regnumber, regvalue`} we takes this list and create the bst

tree representation a node in tree will have following representing {`:node, key, value, leftTree, rightTree`} note leaf node doesn't have a unique tuple and empty tree is represented by `:nil`

```
defp createRegisterInTreeFormat([], tree), do: tree
defp createRegisterInTreeFormat(regInListForm, tree) do
  rndElem = Enum.random(regInListForm)
  newRegInListForm = List.delete(regInListForm, rndElem)
  {key, value} = rndElem
  newTree = Program.insertElement(key, value, tree)
  createRegisterInTreeFormat(newRegInListForm, newTree)
end
```

We take a random element from list and we make sure to remove then insert into the tree and we continue until list is empty. We insert in random order because this gives a good probability that tree will be balanced to some degree at least which is then good for when reading and modifying nodes in the tree we can perform operations in $O(Log(N))$. Note we perform search via the key and for register the key is regnumber

```
# read(reg, rs)
def read(reg, rs), do: Program.lookup(rs, reg)

# write(reg, rd, s + t) -> reg
def write(_, 0, _), do: raise "You are trying to write into register 0"
def write(reg, rd, value), do: Program.modify(rd, value, reg)
```

Here are read and write operation these functions call functions that are defined in Program module which we will talk later in Memory

Out

Out is a custom command which will add the register you want to output value of into a list and will when program halt print out the list Here is the code for when we output list to screen

```
def close(out) when length(out) > 0 do
  {:Halted, :OutResult, Program.reverse(out)}
end
# handle case where out has no elements inside it
def close(_), do: {:Halted, :OutResult, "Nothing to output"}
```

More complex Instructions

The more complex instructions they require memory to work such as lw, sw and beq instruction we will look into later why beq instruction needs the memory

Memory Representation

This part is very important for this exercises. We will when we compile program have some variables already defined that we know about these are data directive, example `[{:label, :arg}, {:word, 12}]}`. This variable arg is type of word which is a 4 bytes value in this case it has value 12 inside it. When we run the program we will also load from and store to memory.

My memory it is divided into two parts `Data directive memory, Dynamic Data memory` where both is represented in BST the same representation we had for our registers. In Data directive Mem. The key is the label such as `:arg` and the value is the Mem address for that variable in dynamic data Mem. In dynamic data mem the key will be an address and value will be value for that specific memory address. Example representation for arg

`{{:node, 0x0, 12, :nil, :nil} ,{:node, :arg, 0x0, :nil, :nil}}`
Left tuple is dynamic data mem and right is data directive mem note when storing to mem it will be stored into the dynamic data mem expect for beq instruction will talk about later

Creating Memory

When we create memory we will take the data directive which is given to program as list where we have data as we specified above with arg. we will create the data directive mem first in list form then from it also create the dynamic data mem and then convert them to BST like we did with Registers. but we will together with the two memories return the current address we are in memory we start at 0x0000 and when we add to memory the memory address will be increased by 0x4, so mem will be following structure.

```
{{dynamicMemTree, currDynamicMemAddr},
{dataDirectiveMemTree, currDirectiveMemAddr}}
```

Will not add code of how the memory is exactly is created will be to much but essentially is the quite the same as in Register expect we now first make a list form of the memory first where in Register it was already created

SW Instruction

```
# mem[reg(rs) + signext(imm)] := reg(rt)
{:sw, rt, imm, rs} ->
```

```
  pc = pc + 4
  regValue = Register.read(reg, rt)
  mem = Program.writeToMemory(mem, regValue, imm, rs)
  # mem can be the new state of the memory or an error in tuple form
  case mem do
    {:ERROR, _} ->
      {mem, "There was an error in sw instruction at this line: #{div(pc-4,4)}"}
    _ ->
      # go to next instruction
      run(pc, code, reg, mem, out)
  end
```

Here is how the sw instruction is defined we will read value from register and then write it into register and this will return the new state of the memory and it could give error or everything is fine and we continue to next line. Here is how we write to Memory.

```
def writeToMemory(mem, rt, _, rs) when is_atom(rs) do
    # deconstruct Mem
    {{dynamicMemTree, currDynamicMemAddr},
    {dataDirectiveMemTree, currDirectiveMemAddr}} = mem
    # check for addr for label in data Directive Mem Tree
    addrLabel = lookup(rs, dataDirectiveMemTree)
    if addrLabel === :no do
      # ! here will assume that we can't store in a place where the
      # data derictive doesn't exist
      {:ERROR, "Trying to store in Mem using a data derictive which doesn't exist"}
    else
      # the label we are trying to store at does exist
      IO.puts("Written in existance place in memory")
      # we modify the value of the address connected to the data directive
      newDynamicMemTree = modify(addrLabel, rt, dynamicMemTree)
      # return our new Mem
      {{newDynamicMemTree, currDynamicMemAddr},
      {dataDirectiveMemTree, currDirectiveMemAddr}}
    end
end
```

Here is when we write to memory where rs is not an address but a data directive that may contain an address so we need to first retrieve the address of the data directive in data directive mem then modify value in dynamic mem using the address as the key to find it. There is also a case where rs is an address the code is quite the same to this expect we now know the

address we want to write so no need to perform lookup in data directive mem.

Beq instruction

we need first to talk about how we handle when we have a label in our code **notice this is not the same as label in data directive part of the program** in the label instruction we will write the label name into both data directive mem and dynamic data mem and this is the exception we mentioned earlier when we talked about data directive mem it can change when a label is met.The label in data directive mem will get it own mem address and in dynamic data mem that address will contain the pc for this label instruction so when when make a jump in beq for example we will retrieve the label form memory and it will point to pc where label is in code and this will be used to direct what next instruction should be. Here is code for label instruction

```
{:label, name} ->
  pc = pc + 4
  # here will write the label into directive memory and in dynamic memory
  # it will include the program counter for the label
  mem = Program.writeToDirectiveDataMemory(mem, name, pc-4)
  IO.puts("label instruction")
  # go to next instruction
  run(pc, code, reg, mem, out)
```

Code for label instruction

Down Below will be code for beq instruction

```
# if reg(rs) == reg(rt) then PC = BTA else NOP
{:beq, rs, rt, label}  ->
  pc = pc + 4
  s = Register.read(reg, rs)
  t = Register.read(reg, rt)
  # check if reg values are the same
  if s === t do
    #IO.puts("Jump")
    value = Program.loadFromMemory(mem, 0, label)
    # value will be pc counter or and error
    case value do
      {:ERROR, _} ->
        # here because we got an error but we know we should but we got error
        # this means the label we tried to load  from memory doesn't yet exist
        # because it's further down in code here will go through the
        # code and try to find that label
```

```
        index = Program.gePcForSpecificInstruction(code, {:label, label})
        # here we will calculate the new pc
        newPc = index*0x4
        # jump to that specific pc
        # ! note here are not jumping to the correct pc that we are supposed
        # ! by mips we are jumping to the start of label
        run(newPc, code, reg, mem, out)
      _ ->
        # no error so the label we have earlier met in code
        # so we know what pc it has
        # ! note here we are correctly jumping to the correct pc
        # if we look at mips we are adding + 0x4
        run(value+0x4, code, reg, mem, out)
      end
  else
    # here the value in register didn't match so we are not going to jump
    # go to next instruction
    run(pc, code, reg, mem, out)
  end
```

Here we retrieve values in register and compare that they are equal **note bne instruction is the same code just the check of register is different** but nonetheless if we don't jump then we just go to next instruction if we jump we could get and error when we get an error this means that the label we looked didn't exist in memory it could be because it don't exist at all or that it exists later down in code ie we are jumping forward so we will try to retieve the pc for that label instruction.
if we don't find the label `gePcForSpecificInstruction` will raise an RuntimeError. Note when we are jumping forward in code the first instruction in code that will be run is the label instruction and not the first instruction after it, this was intentionally done so that the label would be stored in memory because we don't save it otherwise. If no error came when jumping then we have met the label before in code ie we are jumping backwards and notice here we are correctly jumping to the first instruction after label instruction

Byte addressable

Just want to talk byte addressable in emulator the memory does not support byte addressable this is because I save memory 4 bytes at a time what should be in reality is that when saving 4 bytes word for example 0x0000 should save 1 byte and 0x0001 save 1 byte ... 0x03 save 1 byte this would allow us to provide byte addressable. I mainly didn't do it because of time constraints

Examples

```
# program
Program.load({:prgm, [{:lw, 2, 0, :arg}, {:beq, 1, 1, :loop}, {:sw, 1, 0, 0x24},
{:label, :loop}, {:add, 4, 2, 1}, {:out, 4}, {:out, 2} ,:halt], [ {:label, :arg},
{:word, 12}, {:label, :varx}, {:word, 25}]})
# result
{:Halted, :OutResult, [5, 12]}

# program
Program.load({:prgm, [{:add, 4, 2, 1}, {:addi, 1, 1, 0x41}, {:out, 4}, {:out, 1},
:halt], []})
# result
{:Halted, :OutResult, [3, 66]}

# program
Program.load({:prgm, [{:lw, 2, 0, :arg}, {:label, :loop}, {:sw, 1, 0, 0x24},
{:beq, 1, 1, :gg} ,:halt], [ {:label, :arg}, {:word, 12}]})
# result (this is correct label :gg is not defined)
** (RuntimeError) A label in code was not found
   program.ex:182: Program.gePcForSpecificInstruction/3
   emulator.ex:161: Emulator.run/5
```

Conclusion

This exercise at first seemed really hard to do, what data structure should I have which one is correct or better, using a BST is quite a good choice because we have good worst case performance for search and it's also easy to change value in tree without a lot of code, this depends on the tree structure it needs to be balanced. For me this was the most fun exercise we have done so far feel like I have learned a thing or two.