MIPS Emulator

Tomás Belmar da Costa

Spring Term 2022

Introduction

For this assignment we had to build a MIPS emulator that accepted the instructions:
-add $d $s $t
-sub $d $s $t
-addi $d $t imm
-lw $d offset($t)
-sw $s offset($t)
-beq $s $t offset
In addition we have to implement two extra instructions:
-halt -¿ stops the program
-out $s -¿ outputs the value of register $s

Register File

The first part of this assignment required building a register file. The register file was represented as a tuple of 32 items, each representing an individual register. The registers had to have functions to read and write to individual registers, as well as a function to create a new register file. I achieved this by doing the following:

```
defmodule Register do
  def new() do
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
  end

  def read(reg, 0) do
    0
  end

  def read(reg, i) do
```

```
    elem(reg, i)
  end

  def write(reg, 0, _) do
    reg
  end

  def write(reg, i, x) do
    put_elem(reg, i, x)
  end
end
```

This allows for reading and writing to all 32 registers, except for the base case of register $0, which always reads 0 and can't be written to.

Program Module

The program module was in charge of taking a program, represented by the tuple {`:prgrm, prgrm`}, and mapping each label to the point in the program that it's pointing to. There should also be a `read_instruction` function that fetches an instruction following the program counter.

The part of the programming module that actually loads the program is represented as the following:

```
  def load({:prgrm, prgrm}) do
    mem = %{}
    mem = find_labels(mem, prgrm, 0)
    {prgrm, mem}
  end
```

We start by creating an empty map that will serve as the memory. We then call a `find_labels` function that will map all labels and their program counter position to memory like so:

```
  def find_labels(mem, [{:label, label} | tail], pc) do
    write_data(mem, label, pc)
    find_labels(mem, tail, pc + 4)
  end


  def find_labels(mem, [_ | tail], pc) do
    find_labels(mem, tail, pc + 4)
  end
```

```
def find_labels(mem, [], pc) do
  mem
end
```

This function recursively finds all labels in the list and commits each of them to memory, finally returning the updated memory.

Now, we can move on to the actual emulator. This involves building functions that will interpret each MIPS instruction. The first one is simply the `run()` function that will actually run the program.

```
def run(prgm) do
  {code, data} = Program.load(prgm)
  reg = Register.new()
  out = Out.new()
  run(0, code, reg, data, out)
end
```

We then use recursion to continue running the program, this time instruction by instruction.

```
def run(pc, code, reg, mem, out) do
  next = Program.read_instruction(code, pc)
  case next do
    :halt ->
      Out.print(out)
      :ok

    {:add, rd, rs, rt} ->
      pc = pc + 4
      s = Register.read(reg, rs)
      t = Register.read(reg, rt)
      reg = Register.write(reg, rd, s + t)
      run(pc, code, reg, mem, out)

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

    {:addi, rd, rt, imm} ->
      pc = pc + 4
      t = Register.read(reg, rt)
      reg = Register.write(reg, rd, t + imm)
```

```
        run(pc, code, reg, mem, out)

      {:lw, rd, rt, offset} ->
        pc = pc + 4
        t = Register.read(reg, rt)
        result = Program.read_data(mem, t + offset)
        Register.write(mem, rd, result)
        run(pc, code, reg, mem, out)

      {:sw, rs, rt, offset} ->
        pc = pc + 4
        s = Register.read(reg, rs)
        t = Register.read(reg, rt)
        mem = Program.write_data(mem, t + offset, s)
        run(pc, code, reg, mem, out)

      {:beq, rs, rt, {:label, label}} ->
        s = Register.read(reg, rs)
        t = Register.read(reg, rt)
        count = Program.read_data(mem, label)
        if s == t do run(count, code, reg, mem, out)
        else run(pc + 4, code, reg, mem, out) end

      {:out, rs} ->
        pc = pc + 4
        s = Register.read(reg, rs)
        out = Out.put(out, s)
        run(pc, code, reg, mem, out)

      {:label, label} ->
        pc = pc + 4
        run(pc, code, reg, mem, out)
    end
  end
```

The add function reads off the two registers rs and rt using Register.read, adds them together, and finally writes that to rd. The sub function is similar but subtracts instead of adding the two register values.

The addi instruction is also similar to add and sub, but instead of getting the value of register rs, it simply adds the immediate to rt and stores it on rd.

lw and sw make use of the Program module's read_data and write_data to read and write to and from memory.

The beq instruction is a bit more complicated. Once again it collects the values of register rs and rt, and checks if they're equal. If they are, then it finds out through the memory map where the label is pointing to (which we made possible using our find_labels method earlier) and changes the pc to point there. Otherwise, it simply updates the counter by 4.

The out instruction simply calls Out.put(), and feeds it the value of the rs register.

The label instruction can simply be ignored, so the pc is incremented by 4.

Finally, the Out module simply creates a list that can be added to through the out instruction, and prints out the full list when Out.print() is called.

```
defmodule Out do
  def new() do
    []
  end

  def put(list, item) do
    [item | list]
  end

  def print(list) do
    Enum.reverse(list)
  end
end
```