MIPS Emulator

Lucas Larsson

Spring Term 2022

Implementation

The implementation follows the instructions from the provided document by the professor Johan, below are the three supporting modules plus the program and finally a short test program.

Program

The professor recommendation was to start with the supporting modules first then move to the actual instructions, but i didn't do it the same way, I started as once with the instructions which showed it self pretty straight forward, I basically copied the functions from here.And just wrote them in `Elixir` and continued to the rest f the program, down below a snippet with some examples.

```
{:out, rs} ->
    a = Register.read(reg, rs)
    run(pc + 1, code, mem, reg, Out.put(out, a))

{:add, rd, rs, rt} ->
    a = Register.read(reg, rs)
    b = Register.read(reg, rt)
    reg = Register.write(reg, rd, a + b)
    run(pc + 1 , code, mem, reg, out)

{:beq, rs, rt, imm} ->
    a = Register.read(reg, rs)
    b = Register.read(reg, rt)
    cond do
      a == b ->
        pc = pc + imm
        run(pc, code, mem, reg, out)
      a != b -> pc = pc
      run(pc + 1, code, mem, reg, out)

    end
```

Registers

Registers in a real MIPS processor are reserved for specific cases, as in $29 used as a stack pointers, other registers are preserved for return address or values etc.., this program does not take this into consideration rather a person can write to any of the registers not $0 off course but the other 31 registers. registers are implemented as a single 32 element long tuple initiated with 0 in all the registers. Below are the write and read functions needed to perform the operations on the register tuple. a read can read the content of any register, the write can write to any register besides the $0 register.

.

```
  def read(reg, i) do elem(reg, i) end


  def write(reg, 0, _) do reg end
  def write(reg, i, val) do put_elem(reg, i, val) end
```

Memory

The memory part was by far the hardest part with this lab, the following implementation is due to inspiration from the professor in one of the lectures.

The memory is implemented using the `Map` data structure, where a key is associated with a value, the key is the address inserted and the value is the value. This solution allows for saving data in variant address such as save value x in address 200 and value y in address 400 , without using an actual data-structure with 400 places. Below are read and write functions to the Map data structure, nothing fancy only library methods are used.

.

```
def read({:mem, mem}, i) do
    case Map.get(mem, i) do
      nil -> 0
      val -> val
    end
  end

  def write({:mem, mem}, i, v) do
    {:mem, Map.put(mem, i, v)}
  end
```

Test

The test data is inserted as a `List` and a `List.to_tuple` function is used to turn the instruction to tuples. New instances of `Program, Memory` and `Out` are initiated.

Below is a snippet of the inserted Assembly code and the output. All the mentioned instructions in the assignment are implemented, but only a couple are shown here for simplicity reasons.

```
[{:addi, 0, 4, 678},     # £0 <- £4 + 678
 {:out, 0},              # out £0
 {:addi, 1, 0, 1},       # £1 <- £0 + 1
 {:out, 1},              # out £1
 {:addi, 2, 1, 1},       # £2 <- 1 + 1
 {:out, 2},              # out £2
 {:add, 3, 1, 2},        # £3 <- £1 + £2
 {:out, 3},              # out £3
 ...
```

```
Output

[0, 1, 2, 3, ...
```

Reflection

I do realize that i didn't explain everything in detail, but this is only in an effort to get the report as close to 2 pages as possible, I didn't talk about the `PC` but it is basically incremented by 1 for each instruction, it should be by 4 to be as close to the reality of a MIPS program as possible, but for this program i think it is fine.