An emulator

Programming II

Johan Montelius

Spring Term 2021

Introduction

In this exercise we will combine your knowledge of assembly programming and functional programming to implement an emulator for a subset of MIPS assembler. You should have a basic understanding of assembly programming but we will not do very advanced programming so no need to know all instructions by heart. The idea here is not to implement a complete emulator but to see how we can implement it in Elixir.

1 MIPS assembler

We will only implement a small subset of the MIPS instruction set. If we can emulate one arithmetic operation I think we could easily extend the emulator to handle all. The subset is however chosen to cover the various forms of instructions: arithmetic, load and store, branching etc. We will also include some non-MIPS instructions that might be fun to have when we run our programs.

These are the instructions that we will implement:

- `add $d $s $t` : add the values of register `$s` and `$t` and place result in `$d`.
- `sub $d $s $t` : subtract the values of register `$s` and `$t` and place result in `$d`.
- `addi $d $t imm` : add the values of register `$t` and the immediate value `imm` and place result in `$d`.
- `lw $d offset($t)` : load the value found at address `offset + $t` and place it in `$d`.
- `sw $s offset($t)` : store the value in register `$s` at address `offset + $t`.

- `beq $s $t offset` : branch to `pc + offset` if values at `$s` and `$t` are equal.

We will also implement the two following instructions that do not have any corresponding machine instructions but are very handy for our implementation.

- `halt` : halt the execution (normally implemented as an endless loop but we will actually stop)
- `out $s` : output value at register location `$s`.

When you normally write assembly programs you of course use an editor and write you programs in a text file. A program could for example look like this:

```
        .text
main:
         addi $1  $0  5       # $1 <- 5
         lw   $2  $0  arg     # $2 <- data[arg]
         add  $4  $2  $1      # $4 <- $2 + $1
         addi $5  $0  $1      # $5 <- 1
loop:    sub  $4  $4  $5      # $4 <- $4 - $5
         bne  $4  $0  loop    # branch if not equal

        .data

arg:    .word 12
```

We could write a parser that would read a file and represents the program in a suitable data structure for our emulator but we will skip this step and start from a reasonable data structure that represents the program. We will represent our program in two structures, one that holds the code and one that holds the data. This is not really how things work but it's fine for our needs and it makes things easier.

The program will be a list of instructions where each instruction is a tuple holding the name of the instruction and its arguments. The program above could be represented by the following list:

```
  [{:addi, 1, 0, 5},     # $1 <- 5
   {:lw, 2, 0, :arg},    # $2 <- data[:arg]
   {:add, 4, 2, 1},      # $4 <- $2 + $1
   {:addi, 5, 0, 1},     # $5 <- 1
   {:label, :loop},
   {:sub, 4, 4, 5},      # $4 <- $4 - $5
```

```
{:out, 4},            # out $4
{:bne, 4, 0, :loop},  # branch if not equal
:halt]
```

We have added two extra instructions: `:out` that will be used for output and `:halt` that will terminate the execution.

The data segment will in the same way be represented as a sequence of labels and values.

```
[ {:label, :arg}, {:word, 12} ]
```

We combine the code segment and data segments into one tuple and this is what we will call a program in this tutorial.

```
{:prgm,  code(), data()}
```

This data structure is how the program is presented to us. The structure might not be the best structures for our purposes but that is one of our first tasks of our implementation, find a suitable representation that we can work with.

2 The implementation

We start by doing an overall design of the system. This will give us insight into which modules we will need an how data best is represented.

2.1 the state of the computation

The first thing we should think through is what the state of the computation is. The program itself is of course part of the state but since we have separated the code from the data the code part is static. We will only read from the code using a *program counter*. The program counter is thus something that is part of the state and it will of course change during the execution. The most normal operation is that the program counter is incremented by 4 to index the next instruction but it could also be set by a branch or jump instruction.

The memory is of course also part of the state and it will of course change with each store operation. We should be able to index it using addresses and let's assume that we only use addresses aligned by 4 bytes (you could change this later and allow for byte addressing). So given that we should both be able to read from and write to this data structure we might choose something different from the code area.

The final state of the computation are the registers of the CPU. In a MIPS architecture we have 32 general purpose registers (well, register 0 always holds the value 0) but the MIPS assembler language has a convention

of usage. Register 28 is pointing to the data area, 29 is used as a stack pointer, 30 as the frame base pointer etc. For our purposes the registers are all the same (apart from register zero).

2.2 the execution

When we start our emulation we will have a code area where we can read the next instruction referred to by the program counter. We will retrieve the instruction, interpret it, possibly modify registers and/or memory and then determine how to update the program counter. Let's give it a try:

```
defmodule Emulator do

def run(prgm) do
   {code, data} = Program.load(prgm)
   reg = Registers.new()
   run(0, code, reg, data)
end

def run(pc, code, reg, mem) do
  next = Program.read_instruction(code, pc)
  case next do

    :halt ->
        :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)  # well, almost
       run(pc, code, reg, mem)

     :

   end
end
```

Let's totally ignore the problem of overflow and that negative numbers should be represented as two's complement etc. This is not the course of data architecture, we're only doing this for fun.

It should be rather simple to complete this piece of the emulator. Load and store instructions will simply us a module that does the right thing and branch and jump instructions are simple.

2.3 the output

The `:out` instruction could of course echo the value of the register in the terminal but let's add a feature that collects the output in a list that is returned when the program terminates. We can hide the details of how things are done in a separate module that takes care of everything.

```
defmodule Emulator do

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


def run(pc, code, reg, mem, out) do
  next = Program.read_instruction(code, pc)
  case next do

   :halt ->
      Out.close(out)


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

     :

   end
end
```

That's it, all you have to do now is implement the supporting modules and you have a MIPS emulator up and running.

3 Your implementation

There are four modules that you need to implement, three supporting modules and then the emulator itself. Start with the supporting modules and test them before you implement the final solution. These are the modules:

- `Program:` The module should be able to create a code segment and a data segment given a program description. It should provide functions to read from the code segment and both read and write to a data segment.
- `Register:` The register module should handle all operations for the registers: create a new register structure and, read and write to individual registers.
- `Out:` The module should collect the output from the execution and be able to return it as a list of integers.
- `Emulator:` This module is the heart of the system. It should be able to take a program, transform it to a code and data segment and then execute the program and return the output.

The different modules all handle a state but the requirements on these states are different. The code segments only needs to provide a quick lookup operation given a program counter but does not have to care about changing the code itself. The data segment should of course be easy both to read and write to and the same goes for the registers. The difference between the data segment and the register is that the register is of a fixed and fairly small size. The output structure should support incremental writing but no reading, apart from when the list is returned. These differences should be taken into account in you implementation, choose a data structure that is suitable in each case.