FIT1047 Tutorial 4 – Sample Solution

Topics

- Register Transfer Language
- Memory addressing
- Programming MARIE assembly code

Instructions

The tasks are supposed to be done in groups of two or three students. You will need the *MARIE* simulator for this assignment.

Task 1: Conditionals

A **conditional statement** allows the flow of a program to depend on data. In a high-level programming language, this is typically achieved using `if-then-else` statements such as the following (this is Python syntax):

```
if X == Y:
  X = X + Y
else:
  Y = Y + x
```

Implement this piece of code using MARIE assembly. Make use of labels!

Task 2: Extending the instruction set

We are going to extend the MARIE instruction set by the following instruction:

- `SkipIfGreater X` skips the next instruction if the value at memory address `X` is greater than the value in `AC`.

Give the RTL steps needed to implement the instruction.

```
MAR <- X
MBR <- M[MAR]
AC  <- AC - MBR
if AC < 0 then PC <- PC + 1
```

Task 2: Subroutines

An important concept in programming is that of a **procedure**, **function**, or **subroutine**, a piece of code that has a fixed purpose and that needs to be executed over and over again.

As a simple example, you saw last week that MARIE doesn't have an instruction for multiplication. Now imagine a large program: you will probably need the multiplication routine in hundreds of different places!

Of course, you could just copy and paste the piece of code into the place where you need it. **Discuss why that's a bad idea!**

- Copying and pasting the code increases the number of lines of code (the *complexity* of the code), making it harder to write the code in the first place and reducing the readability of the code.
- As a consequence of increased code complexity, if we make a mistake we have to fix it in hundreds of different places. This is error-prone and means that the program will likely contain more bugs. Similarly, if requirements change, there is no single piece of code that needs to be changed. Subroutines therefore also improve the *maintainability* of the code.

Most ISA (Instruction Set Architectures) have some level of support for writing subroutines. In MARIE, there's the `JnS X` instruction ("jump and store"): It stores the value of the `PC` at memory address `X` and then jumps to address `X+1`. The value stored at `X` is called the *return address*, i.e., the address where execution should continue once the subroutine has finished its job.

To **return** from a subroutine, the last instruction in the subroutine should be a jump back to the return address. This can be achieved using the `JumpI X` instruction: it jumps to the address stored at address `X` (compare that to `Jump X` which jumps to the address `X`).

1. Explain how you can use `JnS X` and `JumpI X` to implement subroutines. In particular, think about why `JnS` stores the value of the `PC`, not `PC+1`.

   `JnS X` can be used to jump to the subroutine while saving the PC at address X. `JumpI X` can be used to return from the subroutine.

   `JnS X` saves the PC not PC+1 because the program counter (PC) holds the address of the *next* instruction to be executed in the program. This will be the instruction to be executed after returning from the subroutine.

2. Implement a simple subroutine that computes $2 \times$ `X`, i.e., it takes the value in a memory location `X`, doubles it, and returns to where the original program left off.

   See sample MARIE assembly code `subroutine_double.mas`.

3. Take the multiplication code from last week and convert it into a subroutine.

   See sample MARIE assembly code `subroutine_mult_exp.mas`.

4. Implement a subroutine `Exp` that computes x^y for two numbers x and y. This works almost the same way as multiplication, and since you already have a multiplication subroutine, you can call it to implement `Exp`.

   See sample MARIE assembly code `subroutine_mult_exp.mas`.

5. Bonus task: what if a subroutine wants to call itself? This is called recursion. Discuss the problems with this idea, and what you could do in MARIE to enable recursive calls.

   The "problem" with recursion is that we don't know a priori how often the subroutine will get called. If a subroutine calls itself, it overwrites the return address, so it wouldn't know how to jump back to where it was called from originally. The subroutine must therefore store the previous return address somewhere, then call itself, and upon return restore the original return address.

   This is usually achieved by implementing a *stack*, where each subroutine call pushes the return address on top (so that it doesn't overwrite previous return addresses). You will implement a stack in Assignment 1.