FIT1047 Tutorial 3 – Sample Solution

Topics

- Programming MARIE assembly code
- Circuits for adding and subtracting

Instructions

The tasks are supposed to be done in groups of two or three students. You will need two different *simulators* for this assignment (both are linked from Moodle):

- The *MARIE* simulator lets you write and execute programs for the *MARIE* architecture.
- The *Logisim* tool allows you to draw digital circuits and simulate their behaviour.

Task 0: MARIE basics

Try out the online MARIE simulator with a simple program like the one from the lecture:

```
Load  004     / Load value from address 004 into AC
Add   004     / Add value from address 004 to value in AC
Store 004     / Store AC into address 004
Halt          / Stop execution
DEC 42        / This is address 004, initialise with value 42
```

Task 1: Multiplication

MARIE only has arithmetic instructions for adding and subtracting numbers, but not for multiplication. Your task is to implement a multiplication algorithm. Assume that we want to compute $A \times B$, with A and B stored in some memory location. The result should be stored in memory location C. You can use the following pseudo-code as a starting point.

```
01   C := 0
02   if A = 0 then jump to line 06
03   C := C + B
04   A := A - 1
05   jump to line 02
06   halt
```

- Discuss the pseudo-code for multiplication. It assumes that variables (or storage locations) `A` and `B` hold the two integers that should be multiplied, and the result should be stored in `C`. Which MARIE instructions can you use for each construct in the code (such as the `:=` operator, the `jump to line` instructions, the `if ... then`)?

  The algorithm used is based on the fact that multiplication of two numbers A and B (that is A * B) is actually summing up B, A times, starting from 0 (i. e. B+ B + ... A times). The algorithm first tests whether A is zero, in which case the answer is zero. Otherwise, keep looping until B is summed up A times.

  The following instructions can be used:

  C := 0

  ```
  Load Zero
  Store C
  ```

  (load constant Zero, store at address C)

  go to line: Jump

  if A=0 then: Skipcond 800

- Implement the algorithm using the MARIE simulator. Test your code with several different inputs.

  See file `marie_mult.mas`

- What happens if A is negative before the algorithm starts? What happens if B is negative? Fix the algorithm so that it also works in these cases. First write your solution in pseudo-code, then implement it using MARIE.

  The above algorithm only tests if A is exactly equal to zero at the start and as a result, if A is a negative number, the algorithm gets into an infinite loop. However, if MARIE number system supports negative numbers, the algorithm should work for negative numbers of B.

  To use the algorithm for negative values of A, we will have to use the above conditional test (if statement) on the pure numeric value of A, disregarding the sign of A and then at the end of the calculation if A is negative we will have to adjust the answer appropriately to have the impact similar to multiplying by -1.

  For the code see file `marie_mult_neg.mas`.

Task 2: Adder/Subtractor

You have seen half-adders, full-adders and ripple-carry adders in this week's lecture. You will now use this knowledge to build a 4-bit adder/subtractor in Logisim.

- Your first task is to draw and simulate a simple *half-adder* and *full-adder*. Discuss the truth tables for these two circuits and how they correspond to the circuit diagrams you saw in the lecture. Then implement both in Logisim and try them out.

  Note that the half adder has 2 inputs and 2 outputs while the full adder has 3 inputs and 2 outputs that make it more complex and uses more logic gates. Each bit of the summed output is dependent on the value combinations of the inputs.

  See logisim file **`half_adder.circ`** and **`full_adder.circ`**.

- Now combine four full adders into a 4-bit ripple carry adder, using copy-and-paste. It should have eight inputs (two 4-bit numbers A and B) and five outputs (the 4-bit result and the final carry bit). The carry-in for the first full adder has to be fixed to 0 (use the *Ground* symbol in the *Wiring* folder).

  Use the simulator to enter different numbers and make sure the adder produces the correct result.

  See logisim file **`ripple_carry_adder.circ`**.

- Now let's assume that we are using the *2's complement* representation for negative numbers. Simulate a few computations, such as $5 + (-7)$ and $(-3) + (-4)$.

  See logisim file **`ripple_carry_adder.circ`**.

- Finally, extend the adder to an adder/subtractor. We need an additional input signal N that switches between the two operations: if N is 0, the circuit computes $A + B$, if it is 1, it computes $A + (-B)$.

  Hint: computing $-B$ in 2's complement means negating every bit and adding 1. So for the B input, the circuit negates all bits if N is 1, and it passes the bits through to the ripple-carry adder if N is 0. Now the only thing that's required is to add another 1. Is there something in the existing ripple-carry adder that you can reuse to do that?

  We can use the carry-in to add the extra 1, and negate all B inputs using NOT gates, which would yield a circuit that computes A-B. To make it possible to switch between addition and subtraction, we can use XOR gates: if the switch is 1, they act as NOT gates, if the switch is 0, they just let the other bit pass unchanged.

  See logisim file `add_subt.circ`.

Bonus task: ALU

In case you've finished the first two tasks, implement the ALU from the lecture slides, replacing the negator and adder (which use pre-defined Logisim primitives) with the adder/subtractor you implemented in Task 2.

You need to extend you adder to 8 bits.

See logisim file `alu_ownadder.circ`.