Computer Architecture - Assignment 7 Report

Channa Dias Perera (S4400739)

January 26, 2021

Contents

1 Problem Analysis

1.1 Context

As desired in the specification, the LC-3 can natively only handle from -2^{15} to $2^{15} - 1$. We seek to implement methods to:

- Get a integer from the range $[0, 10^{100}]$[1] from the keyboard.
- Perform addition on `MWN`s
- Display `MWN`s on the terminal

For this task, we will aim to simply have an elegant and simple solution. We will not optimize for time nor memory complexity.

1.2 Analysis

1.2.1 Addition

To perform addition between two `MWN`s, it's easiest to perform simple decimal addition, where we:

1. Take digits of the same significance from both `MWN`s, add them and the carry in.
2. Divide by 10. Since we know that the maximum value of the addition is 18, we simply need to subtract by 10 and check if there was a quotient.
3. Store the remainder in its appropriate position.
4. Pass the quotient to the next most significant addition as a carry in.

1.2.2 Data Representation

As we want to be able to add the decimal digits, it would make sense to store the digits of an `MWN` such that each decimal digit of the `MWN` can be easily obtained.

Due to this, we ruled out storing a `MWN` as a binary value, spread across multiple cells. Even though it would be relatively memory efficient, as it would take only $\lceil \frac{\log_2 10^{100}}{16} \rceil = 21$ memory cells, it would make extracting the decimal digits problematic.

Thus, we will store each decimal digit (as its binary value) in a dedicated space of 4 bits, per digit.

[1] As in the specification, we call these numbers `MWN`s

2 Algorithm and Program Design

2.1 Main Procedure

The main procedure is very simple:

1. We gather both `MWN`s from the terminal
2. We add the two `MWN`s together and store the result in memory.
3. We read the `MWN` in memory.

2.2 Reading input from the keyboard

Due to our simple method of storing `MWN`s, our input method is greatly simplified.
We simply:

1. Set the amount of digits present to 0.
2. Get a character from the keyboard
3. Echo the character
4. Check the value of the character:
    - If the character is a newline, we store the number of digits and leave the subroutine.
    - If the character is not a newline, we:
        (a) Convert the number to binary and store the number in memory.
        (b) Increment our digits counter.
        (c) Repeat steps 2 - 4

2.3 Adding two `MWN`s

This algorithm is more complex (in terms of its implementation).
The algorithm is as follows:

1. Set a `cellOutput` variable to 0.
2. Set two counters (`c1`,`c2`) to the lengths of both `MWN`s.
3. Set a carry to 0
4. Set two pointers (`p1`,`p2`) to the end of the two `MWN`s.
5. Set a pointer (`p3`)to a space allocated for the result and a counter (`c3`) to keep track of the number of digits in the result.
6. While either counter is positive, or the carry is positive:

(a) If `c1` > 0, add the value at `p1` to `cellOutput`. Increment `p1` and decrement `c1`.

(b) If `c2` > 0, add the value at `p2` to `cellOutput`. Increment `p2` and decrement `c2`.

(c) Add the carry to `cellOutput`

(d) Subtract 10 from `cellOutput`

- If the result is negative, set the carry to 0, add 10 to `cellOutput`.
- If the result is positive, set the carry to 1.
- Store `cellOutput` in `p3`. Increment both `p3` and `c3`.

7. Flip the ordering of the digits of the `MWN` stored at the initial value of `p3`.

8. Save the number of digits present in the output.

2.4 Flipping the ordering of a `MWN`

Flipping the order of a `MWN` is a non-trivial task and thus will be described here.

The principle behind the algorithm is to swap the digits mirrored by the center most digit (or two center most) digits. To do this, we simply swap digits on opposite ends, up till we reach the half-way point.

The algorithm is as follows:

1. Keep a pointer (`p1`) to the start of he `MWN`.

2. Keep a pointer (`p2`) to the end of he `MWN`.

3. Keep a counter (`c1`), initialized to $\lfloor \frac{n}{2} \rfloor$, where n is the number of digits in the `MWN`. To acquire this value, one simply needs to right shift the binary value of n by 1 position.

4. While `c1` is positive:

   (a) Swap the value at `p1` and `p2`[2].

   (b) Increment `p1` and decrement `p2`.

   (c) Decrement `c1`

[2]Doing so will require the use of a temporary variable.

3 Implementation Choices

3.1 Data Representation

Now, whilst we stored each digit in its own cell, this is incredibly wasteful. This is because each cell is 16 bits long and we only use 4 of those bits.

Indeed, we could make our memory usage 4 times more efficient, by storing 4 decimal digits in a single memory cell. However, this would:

- Come at a time-performance cost, as we extract the decimal value, possibly requiring a masking and rotation.
- Complicate our implementation

Hence, we stored each digit in a single cell.

There is also the question of whether we should use a sentinel to indicate the end of a `MWN` or to store the number of digits. We shall store the number of digits, since it allows direct access to the end `MWN`, at the cost of a single memory cell (compared to the 100 or so required to store the `MWN`).

3.2 Right Shifting

Our right-shift is incredibly inefficient, since it performs a very slow division by 2, which is equivalent to a right shift.

We could speed this up by dividing by larger powers of 2, adding the powers.

However, a much faster approach would be to:

- Implement a left-rotate
- Left-rotate the value 15 times

We decided against this since it would complicate our design and again, time-performance optimization was not the goal of this project.

3.3 Robustness

We decided to avoid error-checking the input, such that it is a valid `MWN`. This was out of the scope of the project.

3.4 Nested Subroutines

One important consideration in nested subroutine is preserving the state of the program, just before after it was called. For this program, this means saving the register values as the program is called.

For a subroutine directly called from the main program, it is sufficient to store the register values into memory cells dedicated to saving their respective register.

However, this method cannot be used for nested subroutines. This is because the nested subroutine will over-write the saved registers from its calling subroutine.

Thus, for nested subroutines, we allocate a space of 8 memory cells per nested subroutine. This space is used to save the registers.

3.5 Ordering of data labels

It's worth mentioning that there's a key flaw in the scalability of the program. That is, we cannot actually handle decimals with much more than a 100 digits.

As it stands, the many instructions in the program are written such that the address can be loaded by a 9-bit 2's complement offset from the PC. If more space was allocated for larger decimals, the PC would quite quickly require an offset than that which could be immediately included as an operand.

To fix this, one would ideally like to store the values of the starting address of each MWN before allocating space to each MWN. For example:

```
; Declare pointer
PTR     .FILL   MEM_LOC
; ...
MEM_LOC .BLKW   #n
; Allocate Space
```

Our program works to specification, so this was not implemented, but it would not be too difficult to change.