Microprocessors Lab1

Kyriafinis Vasilis 9797, Koro Erika 9707

April 9, 2022

Introduction

The objective of this assignment is to calculate the hash of a given string. The hash function is a given hash table, that matches capital letters to integers, implemented in ARM assembly and called by a main C routine.

Code description

In the main C routine a string is given in order to calculate its hash.

```
// This is the string to caclulate the hash from
char string[] = {'V', 'L', 'a', 'B', ' ', '1', 'c', '@', 'D', '9', '!',
's', 'a', 'T', 'Y', 'O', 'p', '^', '.', 'A', 'Z', '\0'};

```

In the ARM assembly, hash function has three arguments:

1. r0: Contains the hash table's memory address.
2. r1: Contains the given string's memory address.
3. r2: Contains result's memory address.

In the beginning, the first string character is loaded from the memory and then is compared to the value 90(The 'Z' in ASCII). If the character is bigger, it means that it is not a number nor a capital letter, so branch to not_number tag, else compare it to the value 65(the 'A' in ASCII). If it is less than 65, it means that it is not a capital letter and branches to not_capital tag. If none of the comparisons is true, it means that the character is a capital letter.

```
ldrb r3, [r1]     // r3 = mem[r1]  c = string[i]

cmp r3, #90       // r3 == 90
bgt not_number    // if r3 > 90 branch to not_number because it can't
                  // be neither a number nor a capital letter

cmp r3, #65       // r3 == 65      compare char with 65
blt not_capital   // if r3 < 65 branch to not_capital

```

In order to know which number should be selected from the hash table, the subtraction between the character and the 'A' character is used as an offset for the table. The hash table contains integers that are 32 bits, so the offset is multiplied with 4(shift left 2). Afterwards, the hash value is added to the register that holds temporarily the calculated hash value and finally the pointer, that holds the string, is increased to point to the next character of the string to be examined as well.

```
sub r3, r3, #65          // r3 = r3 - 65        c -= 65
lsl r3, r3, #2           // r3 = r3 * 4         reading int from memory
ldr r3, [r0, r3]         // r3 = mem[r0 + r3]   c = hash_value
add r4, r4, r3           // r4 = r4 + r3        h += hash_table[c - 65]

```

```
                        // Here r1 initial value is lost but for this
                        // function this value is not needed

add r1, #1              // r1 = r1 + 1 str points to the next char
b loop                  // There is no need to check if c == 0 because
                        // c was a capital english letter

```

When branching to the not_capital tag the character is compared to 48(0 in ASCII) and then to 57(9 in ASCII). If the character is between those numbers it means that it is a number and its value is subtracted from the hashed value as needed. To get the value, from the character is subtracted the value 48 to convert it to an integer. Eventually, the string pointer increases to point to the next character and the program branches to loop tag. Otherwise, when the above if statement is false the program branches to the not_number tag.

```
not_capital:

cmp r3, #48         // r3 == 48      compare char with 48
blt not_number      // if r3 < 48 branch to not_number

cmp r3, #57         // r3 == 57
bgt not_number      // if r3 > 57 branch to not_number

                    // r3 register is reused since there is no need
                    // to save the char read from memory

sub r3, #48         // r3 = r3 - 48  converts the ascii char to int
sub r4, r4, r3      // r4 = r4 - r3  h -= c - 48
add r1, #1          // r1 = r1 + 1   str points to the next char
b loop              // There is no need to check if c == 0 because
                    // c was a number

```

When branching to the not_number tag, it is checked whether the character is null, in order to exit and store the calculated hash value else it continues to the next loop.

```
not_number:

cmp r3, #0          // r3 == 0       if c == '\0'
beq exit            // if c == '\0' exit

                    // if c was not 0 add 1 to the counter and loop
add r1, #1          // r1 = r1 + 1  str pointer points to the next char
b loop              // There is no need to check if c == 0 because
                    // c was a number

```

Problems Encountered

In an effort to limit the usage of brach commands we tried to use conditional instructions. The program required to execute more than one conditional instructions in a row and an it block was needed. Due to the errors that arose at run time (read permissions) the idea was abandoned.

Testing and Debugging

For testing purposes a C function was created in order to confirm the hash results. The initial string was hardcoded to the code. Many different strings where tested. Also the printf function was called from the assembly function to confirm the results along with the C function.

Regarding the debugging of the assembly code the Keil debugger was used. Especially useful was the memory and register viewer because it allowed to check the validity if every instruction executed.

The memory view windows allows to check the content of the memory on a specific memory address in decimal, hexadecimal and other formats. Finally the register panel displays the value of every register in hexadecimal format.