**Part 1: Sets- Symmetric Difference**

The task in this section was to determine the symmetric difference of two given sets of integersA and B of known size. The program is to stor e the set of symmetric difference C, and its size, in memory.

The algorithm I used was as follows:

Length\_C = 0

Boolean AinB

A = {A list of integers}

B = {Another list of integers}

**for** elementA **in** A:

AinB = 0

**for** elementB **in** B:

**if** elementA==elementB:

AinB=1

**if** AinB==0:

C.append elementA

length\_C++

**for** elementB **in** B:

AinB=0

**for** elementA **in** A:

**if** elementA==elementB:

AinB==1

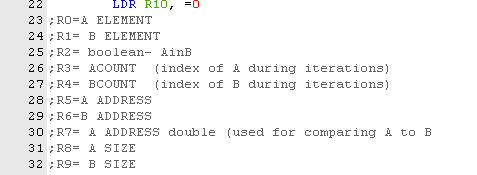
**if** AinB==0:

C.append elementB

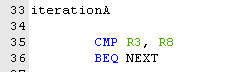
length\_C++

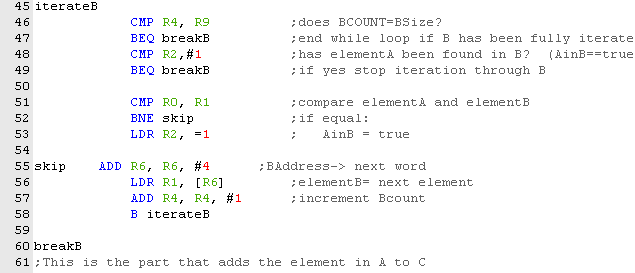
This method checks both sets, ensuring that if there is a unique number in either it is added to C.

Below I outline how I implemented this in ARM assembly code.

*How the registers were used.*

As the pseudo-code shows, the algorithm uses two for loops to iterate through one set, then repeats for the other set. The program begins by comparing an element in A to each element in B, then going on to the next A element.

 Each iteration increments R3 (the ACount), and when this is equal to the size of A (R8), the for loop is done.

Next comes the iteration through each element in B.  BCount (R4) is compared to Blength, and the loop finishes if they are equal (because this signifies that all elements in B have been checked).

AinB is also compared to 1, as this represents true. This means that if, for any iteration through B, AinB is already true, the loop stops, because the A element **will not be added to C.** In pseudo-code:

if BCount==B.length & AinB==true:

break

Otherwise, we compare elementA and elementB. If they are equal, AinB is set to true.

Else the program skips line 53. (this was done using the conditional execution code “NE”, “not equal”.

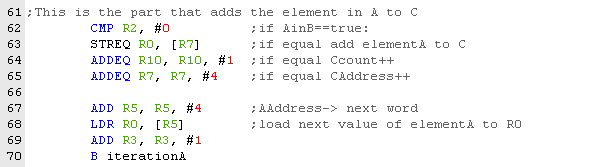
At line 55, the program prepares for the next iteration, changing the address of the B element by one word (R6 = R6 + 4). This is because each number in the list is word sized (32 bits).

R4 is incremented, and the iteration repeats.

Once the iteration stops, the program moves on decides whether or not element A belongs in C. All it needs to do is check the value of AinB.

If AinB is false, this means that elementA was not found in B, so it is a member of C.

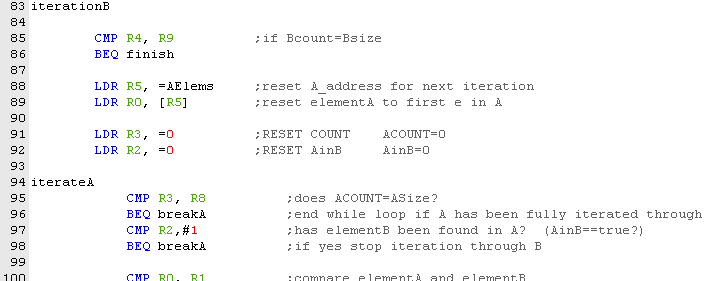
The program stores R0 (elementA) at the current address of C (initialised at start), then increments the address by 4, to allow the next element (if any) to be stored. R10 is also incremented only if AinB == false, as it represents the size of C.

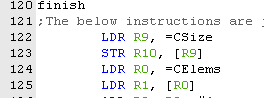


Since we are still in the A for loop (iterating through elements in A), we move the A address to to the next word (see line 67), and the next element of A to R0. Finally, ACount (R3) is incremented, and the program branches back the start of the iterationA.

The above is repeated until ACount = = A.length. Once this is true, the program has iterated through all elements in A, and added any members of C to C.

Next, the same process is carried out, with the A and B switched (so that elements in B that are not in A can be added to C). (below is a snippet of that section: you can see it is the same with A swapped for B).



Once this is done, all elements which are in A or B but not both will have been added to C, and C’s size will also be stored in memory.

This is done using load and store instructions.

**Testing Sets**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | C length correct |
| **1** | **4,8,15,16,23,42** | **13,9,1,9,5,8** | **4,15,16,23,42,13,9,1,9,5** | **yes** |
| **2** | **-2,-1,0** | **-2,1** | **-1,0,1** | **yes** |
| **3** | **null** | **1,2,3,4,5** | **N/A** | **N/A** |
| **4** | **1000000000** | **1,2,3,4,5** | **N/A** | **N/A** |
|  |  |  |  |  |

1. Program behaved as expected, producing C and giving the correct length.

2. Program worked even for negative numbers and zero.

3.Program did not build, there was no way to represent an empty set.

4.Program did not build, numeric overflow error was given.

This testing shows that sets performs very well, and is not easily broken. If the input is correct, it will practically always do its job.

**Part 2: Countdown – testing for anagrams**

The task for this section was to write a program which would determine if a given word (stored in memory, of known size) could be made from a given set of letters (also stored in memory, of known length).

Here is the algorithm I used in pseudo-code:

for element in word:

for letter in letters:

if element==letter:

letter\_balance++

for proceeding\_element in word:

if letter\_balance==0:

break

if word.index(proceeding\_element) >word.index(element):

if proceeding element==element:

letter\_balance—

if letter\_balance==0:

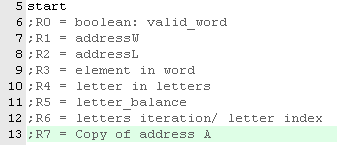
break

if letter\_balance ==0:

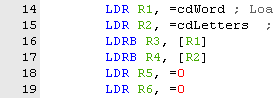
valid\_word = false

Below I will show how I implemented this algorithm using ARM assembly code.

Here is a key to the registers used:

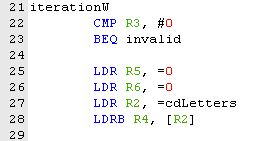


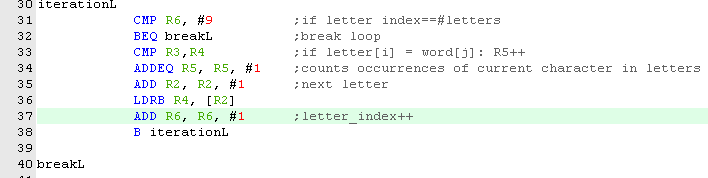
**Initialise variables.**

Addresses of the word and the letters are set at the beginning of the program. These will be incremented in order to retrieve subsequent elements from memory, using the load function.

The counts R5 and R6 are also set to zero, so that they can be used to determine if the for loops are finished. Note that lines 20 and 21 check R0. If it is equal to ‘NULL’, this means the word has no characters, and so is invalid.

**Start of iteration through word**

The program begins to iterate through the letters of the word. The first/next word element is already stored in R3 at this point. The variables used to iterate through Letters are reset, for the next iteration.

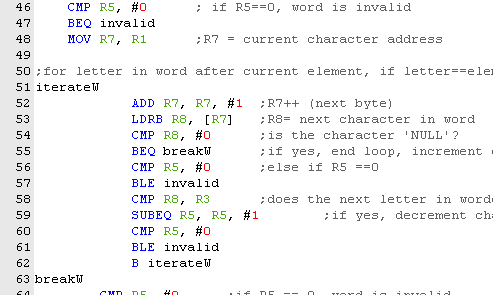
**Iteration through letters.**

This loop increments R6 each time, so that when it has been run through 9 times it breaks (lines 37 and 31-32).

Else, we compare the letter in LETTERS to the element in word. If they are the same, letter\_balance (R5) is incremented (lines 33-34).

The next letter of letters is loaded to R4 by incrementing the address in R2 (line35) and loading the byte at that address using LDRB.

**Iteration through proceeding word elements**

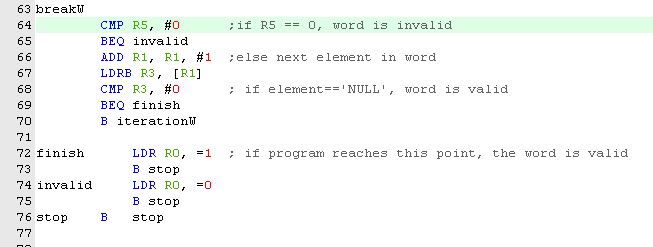


After the number of letters in letters equal to the current word element has been determined (letter balance stored in R5), the program continues. It could be that there are no similar letters in the pool, telling us right away that the word is invalid. In this case, R5 was not incremented at all in the previous loop, so lines 46 and 47 tell the computer to branch to the invalid function if R5==0.

If this is not the case, letter\_balance is greater than zero, however it could be that subsequent letters in the word itself are the same as the current letter. If there are more of these repeating letters than there are in the letters list, the word is invalid.

So the next section of the program subtracts one from the letter\_balance for every similar element in the word(lines 58,59). The address of the current word element is copied to R7 (line 48), and iterateW checks all elements *after* that address (we don’t want to include the current letter). If the character is NULL, this loop breaks as the rest of the word has been tested/end reached.

Since the letter\_balance R5 represents the number of the current element in the list, if it reaches zero this means there are not enough letters, and the loop breaks (lines 60,61).

**End of word iteration**

Once the above processes have been executed, the program prepares the variables for the next iteration, moving the next element in word to R1. However, if this next value is NULL, the loop has reached the end of the word without finding an invalid letter, and so branches to finish (lines 66-69). Else, we branch back up to iterationW, to repeat the above process.

The finish section sets R0 to true, then stops the program, invalid sets it to false.

**Testing Countdown**

For each of the following tests, the letters string was “daetebzsb”.

word is valid

WORD = “date”

R0 =1

As expected, program works for simple correct input.

word is invalid

WORD = “invalid”

R0 =0

As expected, program works for simple incorrect input.

word contains capital letters

WORD = “DAETEBZSB”

R0 = 0

This is one limitation, the program cannot correct for capital letters entered.

word contains numbers

WORD = “b4”

R0 =0

This is a good property, as a word cannot contain a number (the letters string would be incorrect to contain numbers).

word contains spaces

WORD = “date beb”

R0 =0

This is a good property, as a word cannot contain a space.

word is longer than 9 characters

WORD = “”

R0 =0

Program behaves as expected, as there cannot be enough letters in the 9 character long string to make a word longer than 9.

word is 0 characters long

WORD = “”

R0= 0

This is good, as the above is an invalid word.

The above shows that the program performs under nearly all scenarios. One way to break it may be to enter a string longer than the max integer, but that would take a rather long time, so I’ll leave that to you.

**Part 3: Lotto- checking n tickets for 4,5 and 6 matching numbers**

The task was to design a program which would check if given ‘tickets’ containing 6 numbers between 1 and 32 had either 4, 5 or 6 numbers matching a given ‘draw’, another set of 6 numbers. These values were stored in memory as well as the number of tickets. The algorithm I used is shown below:

winning\_tickets\_list={0,0,0}

match\_numbers= {4,5,6}

for match\_number in match\_numbers:

winning\_tickets==0

while tickets checked <number of tickets:

ticket= tickets[tickets checked]

matching elements == 0

for D\_element in Draw:

for T\_element in ticket:

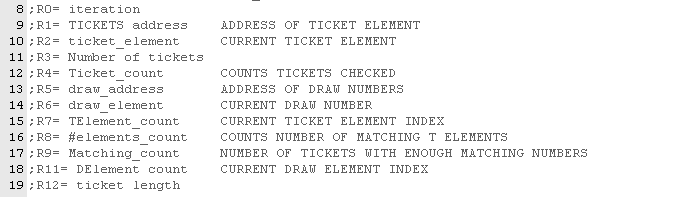
if T\_element == D\_element:

matching\_elements++

if matching\_elements == match\_number:

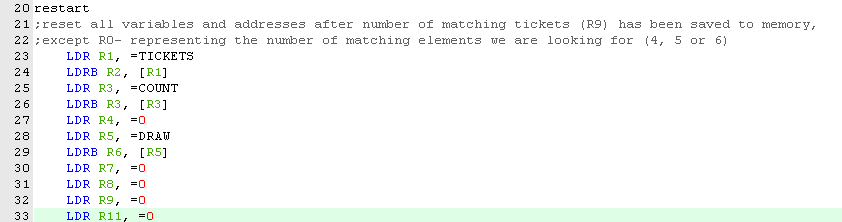
winning\_tickets++

winning\_tickets\_list[match\_number- 4] == winning\_tickets

Below I will outline how I implemented this in ARM assembly. Below is a key of how the registers were used.

**Start and restart**

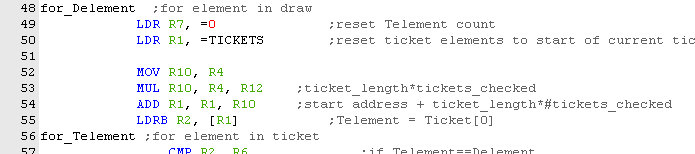




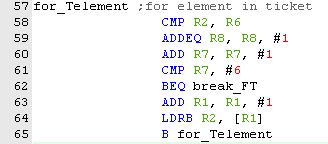
Start and restart are quite self explanatory (see the comment on restart). The ticket length R12 is a constant, only needs to be set once. Initially (line 6), we are looking for 4 matching elements, so R0 is set to 0. At the bottom of the restart loop, it is incremented.

**Resetting the ticket address in Draw element loop**

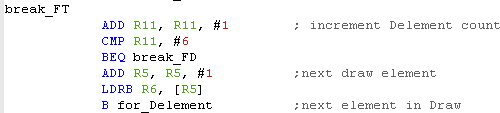
When iterating, the ticket address must be reset every time a ticket has been iterated through, however we don’t always want to reset the address to the start, because this would mean iterating through ticket 1 again. There needs to be a way to load the address of a given ticket. The tickets are of known length, 6. R4 holds the number of tickets already checked for a given iteration through for\_Delement. Using these two facts, the program can arrive at the correct address by resetting it to the start address of TICKETS (line 50), then adding ticket\_length\*tickets\_checked.



**Iterating through the ticket**

The elements match if R2==R6, so we compare them and increment the element\_count R8 if they are equal.

The next ticket element is loaded (lines 63,64), and the loop continues until all 6 elements have been counted (R7 == 6, lines 60-62).

At the end of this loop, the program moves on to the next draw element unless all elements in Draw have been iterated through.

**End of draw loop**

R0 holds the iteration count, which represents how many matching elements we are looking for. So once the matching elements in a ticket have been counted (stored in R8), if this value is equal to the current winning value (ie: 4,5 or 6), R9 is incremented to represent another winning ticket. R4 is incremented (recording another ticket has been checked), and if R4 is equal to the number of tickets, all tickets have thus been checked, and the program jumps to the save\_result section. Else, it jumps back to for\_ticket, and repeats the above for the next ticket .

**When done**

Once all tickets have been checked (R3==R4), the program begins to store the result of the process in memory. R9 holds how many tickets had been determined to have the correct number of matching elements for the current iteration (which in turn is stored in R0). If R0 === 0, we are looking for 4 matches, R0== 1 we are looking for 5, and so on.

The program checks this, and jumps to the relevant save function. Each save function saves R9 in the corresponding memory location, increments R0 (the iteration count) and then jumps back to restart, unless in save 6 where the program stops.

**Testing Lotto**

So long as the correct number of tickets is stored in memory, and the tickets are of the correct length, the program works for all valid values. It even works for values outside the range 1-32, until the maximum integer size is reached.

It works for any number of tickets.

There isn’t really much else to test in this task, it either works or it doesn’t.

For example, the following was stored as the data:

COUNT DCD 6 ; Number of Tickets

TICKETS DCB 1,2,3,4,5,7 ;5

DCB 1,2,3,4,5,6 ;6

DCB 1,2,3,4,5,6 ;6

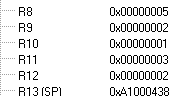
DCB 1,2,3,4,30,31 ;4

DCB 1,2,3,4,9,6 ;5

DCB 1,2,3,4,5,8 ;5

DRAW DCB 1,2,3,4,5,6 ; Lottery Draw

The values of MATCH4/5/6 were loaded to registers R10,R11 and R12 respectively, and as expected gave the following results:



showing that R9 had been 1 when testing for 4 matching elements (1,2,3,4,30,31),

3 when testing for 5 (1,2,3,4,5,7 ; 1,2,3,4,9,6 ; 1,2,,4,5,8)

and 2 when testing for 6 matching elements (1,2,3,4,5,6). This last value of R9 can still be seen in the register after the program had finished.