COMPUTER SYSYTEMS ASSIGNMENT 2 – ARMLITE STOPOWATCH

Name: Pulkit Pannu
Student ID: 104093910
Unit Code: COS10004
Lab Session: Thursday 8:30am – 10:30am
Lab Tutor: Kafil Uddin

INTRODUCTION

The aim of this assignment was to implement the Digital Stopwatch as specified in the assignment 1, as closely as possible in the context of functionality, in the ARMlite ARM Assembler Simulator. I have completed till stage 4 of this assignment, i.e., Stopwatch with Buffering, with as precise functionality of the stopwatch as I was able to comprehend from the assignment description.

To achieve this, I have used memory addresses to store and retrieve the display values as and when needed, according to the requirements of the program. The program takes keyboard inputs to interact with the stopwatch.

DESIGN OUTLINE

With respect to the final stage of my assignment, the directions to use the stopwatch are as follows:
The following key presses will allow the user to interact with the stopwatch.

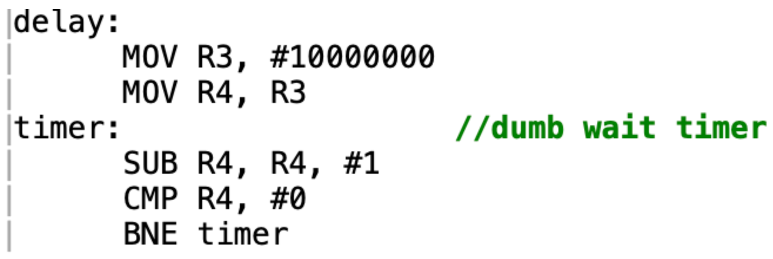
- ‘P’ will play/pause the stopwatch,
- ‘S’ will store the split times,
- When the stopwatch is in the paused state, pressing ‘S’ will allow the user to see the stored split times and to loop through the last 5 stored split times,
- ‘R’ will reset the stopwatch display back to 0, just like it did in assignment 1,
- ‘E’ will end the program/the stopwatch, i.e., it will halt the program.

It must be noted that the user can use either lower-case or upper-case keyboard alphabets, as the code takes this point into consideration.

Stage 1 – The aim for this stage was to display the seconds of the stopwatch incrementing in the text output display area of the simulator and to allow the user to pause and un-pause the timer as required. To achieve this, I used registers R0 and R1 to store in them the seconds one’s position and the seconds ten’s position respectively. I used these registers to display their values and increment them by 1 after every “second”. The values stored in R0 are rolled over to 0 after it hits 9 and only then the value in R1 is incremented by 1. This is just in accordance with assignment part 1.

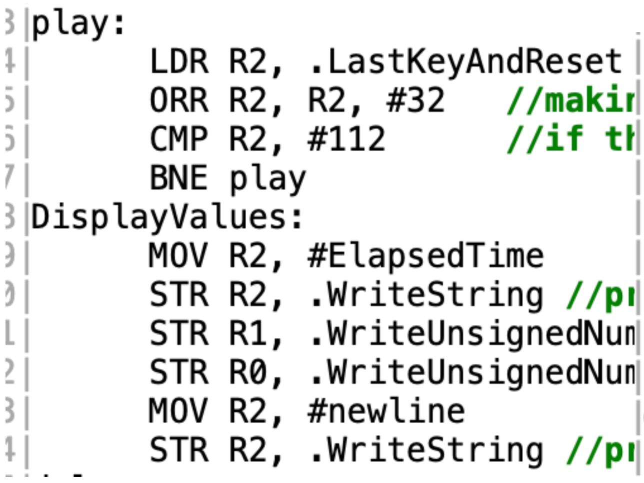
```
ADD R0, R0, #1
CMP R0, #10
BNE skip
MOV R0, #0
ADD R1, R1, #1
CMP R1, #6
BNE skip
MOV R1, #0
```

To implement the “seconds” delay, I used a ‘Dumb Busy Wait Timer’ to make the code wait for a “second” before printing the next value.

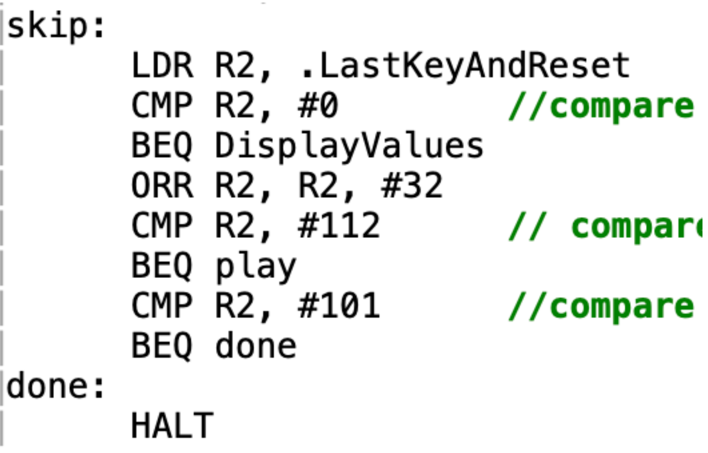


```
delay:
      MOV R3, #10000000
      MOV R4, R3
timer:                        //dumb wait timer
      SUB R4, R4, #1
      CMP R4, #0
      BNE timer
```

For getting the user keyboard input, I used the ARMlite pre-defined function "LastKeyAndReset" to store the ASCII value of the last key pressed in a register and then compare that value to the key we have defined for that purpose. In this case, I store the key value in register R2, and I used the logical ORR operation to convert that ASCII value to a lower-case character if the user pressed an upper-case alphabet. Then the value stored in R2 is checked to see if it is equal to 112, which is the ASCII value for a lower-case 'p'. If the comparison is not true, then the code branches back to the 'play' label again (BNE, Branch if comparison is Not Equal), looking for the user input key. But if the comparison is equal then the code executes the next line of code, i.e., displaying the values.



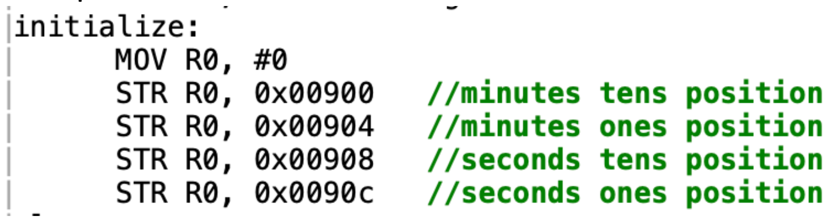
```
play:
      LDR R2, .LastKeyAndReset
      ORR R2, R2, #32    //maki
      CMP R2, #112       //if t
      BNE play
DisplayValues:
      MOV R2, #ElapsedTime
      STR R2, .WriteString //p
      STR R1, .WriteUnsignedNu
      STR R0, .WriteUnsignedNu
      MOV R2, #newline
      STR R2, .WriteString //p
```



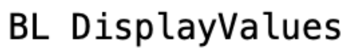
```
skip:
      LDR R2, .LastKeyAndReset
      CMP R2, #0         //compare
      BEQ DisplayValues
      ORR R2, R2, #32
      CMP R2, #112       // compar
      BEQ play
      CMP R2, #101       //compare
      BEQ done
done:
      HALT
```

Similarly, after every "second" wait, the code checks for user input under the skip label and then branches to the label accordingly.

Stage 2 – The aim of this stage was to show the time ticking in "real-time" and to allow the user to reset the stopwatch whenever needed. First, in this stage instead of representing only the seconds for the stopwatch display, I implemented the minutes as well. And to do this, rather than using registers to store the values of displayed time, I used constant memory addresses to store and retrieve those values as and when needed. I used the memory addresses 0x00900, 0x00904, 0x00908, 0x0090c for each of the digit to be represented on the display. At the start, I initialized these addresses with the value 0 because the starting point of a stopwatch is 00:00.

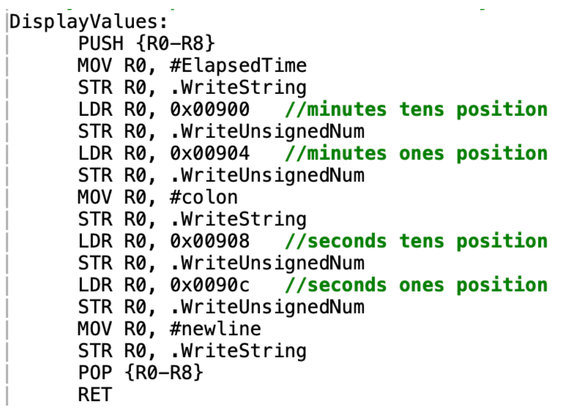


```
initialize:
      MOV R0, #0
      STR R0, 0x00900   //minutes tens position
      STR R0, 0x00904   //minutes ones position
      STR R0, 0x00908   //seconds tens position
      STR R0, 0x0090c   //seconds ones position
```

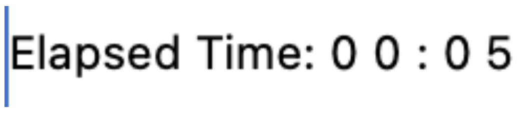


```
BL DisplayValues
```

Then after the user input, the Display Values function is called by using the Branch Link ARMlite function. The Display Values function loads (using LDR) the elapsed time values from the memory address into a register and then stores (or "prints") that value (using STR) on the input-output screen of the simulator.

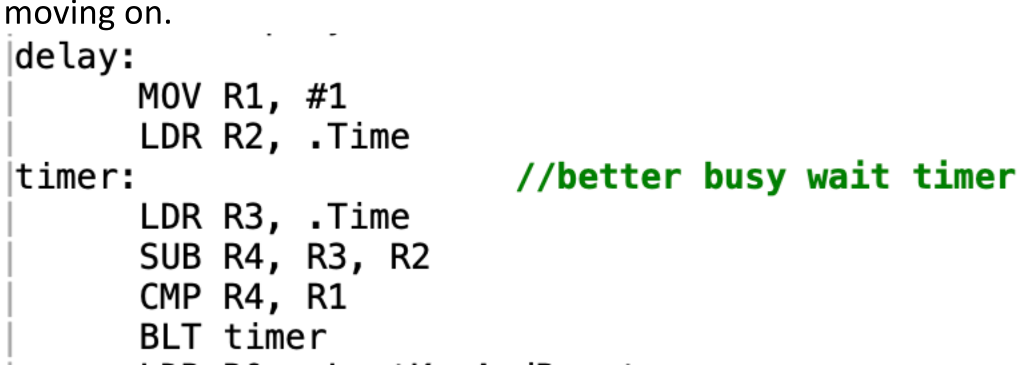


```
DisplayValues:
      PUSH {R0-R8}
      MOV R0, #ElapsedTime
      STR R0, .WriteString
      LDR R0, 0x00900   //minutes tens position
      STR R0, .WriteUnsignedNum
      LDR R0, 0x00904   //minutes ones position
      STR R0, .WriteUnsignedNum
      MOV R0, #colon
      STR R0, .WriteString
      LDR R0, 0x00908   //seconds tens position
      STR R0, .WriteUnsignedNum
      LDR R0, 0x0090c   //seconds ones position
      STR R0, .WriteUnsignedNum
      MOV R0, #newline
      STR R0, .WriteString
      POP {R0-R8}
      RET
```



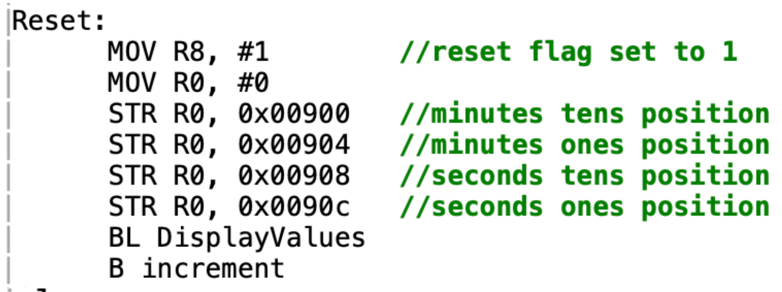
```
Elapsed Time: 0 0 : 0 5
```

As for showing the time ticking in " real-time", I used the 'Better Busy Wait Timer' taught to us by Chris which uses the ARMlite built in ".Time" function to store the value of actual time in a register and then compare the difference of the time of entrance in the function to the time spent inside of the function with the amount of "wait" or delay we need to create before moving on.

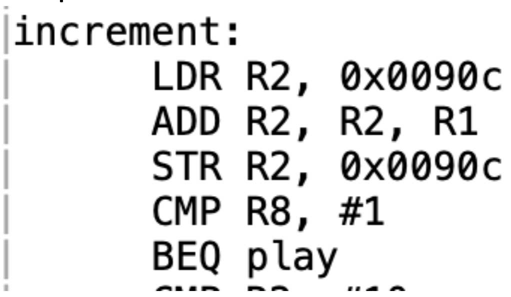


```
delay:
      MOV R1, #1
      LDR R2, .Time
timer:                        //better busy wait timer
      LDR R3, .Time
      SUB R4, R3, R2
      CMP R4, R1
      BLT timer
```

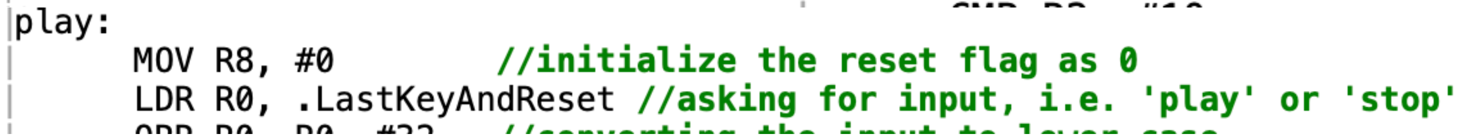
For giving the user the ability to reset the stopwatch, I created a label with the name 'Reset' in which the values stored in the memory addresses are all reset to zero and a reset flag in register R8, which was initially set to 0, is set to 1, indicating that the reset button has been pressed. Then the reset values are displayed by the Display Values function and under the increment label, after incrementing the seconds to 1, it is checked if the reset flag is set to one, which it is, and then the execution branched to the play label where the reset flag is set back to 0 and the code waits for the user input to restart the stopwatch.



```
Reset:
      MOV R8, #1        //reset flag set to 1
      MOV R0, #0
      STR R0, 0x00900   //minutes tens position
      STR R0, 0x00904   //minutes ones position
      STR R0, 0x00908   //seconds tens position
      STR R0, 0x0090c   //seconds ones position
      BL DisplayValues
      B increment
```

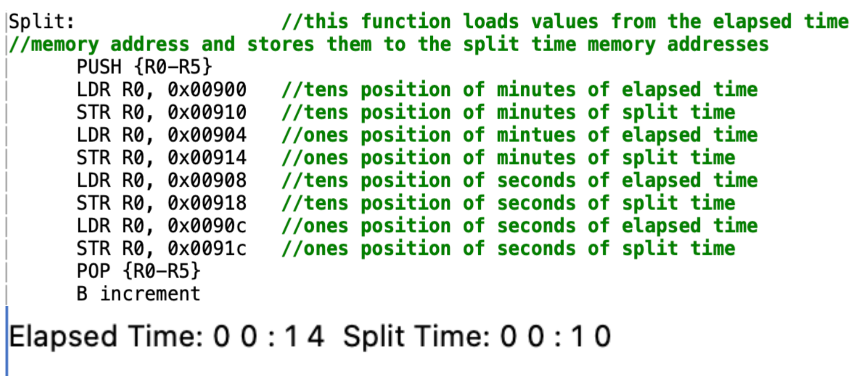


```
increment:
      LDR R2, 0x0090c
      ADD R2, R2, R1
      STR R2, 0x0090c
      CMP R8, #1
      BEQ play
```



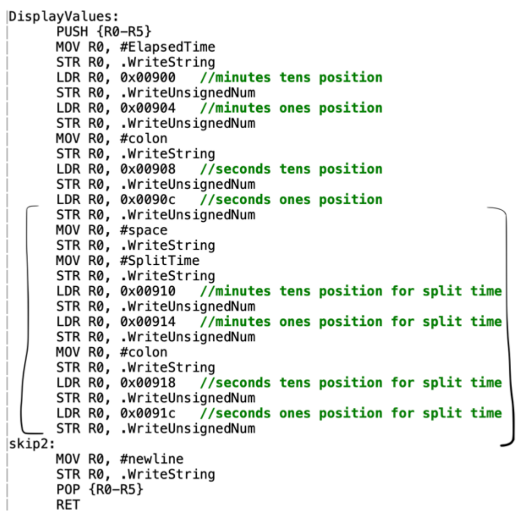
```
play:
      MOV R8, #0        //initialize the reset flag as 0
      LDR R0, .LastKeyAndReset //asking for input, i.e. 'play' or 'stop'
```

Stage 3 – The aim of this stage was to allow the user to store and display split times while also continuing to show the timer. To achieve this, I had to allocate memory addresses for the split time values as well. For them, I used the memory addresses 0x00910, 0x00914, 0x00918 and 0x0091c to store and retrieve the split time values when the split button is pressed. To implement this, I created a label named 'Split', which is called when the stopwatch is not in the paused state and the user presses 'S', in which the value stored in the elapsed time memory addresses, at the time of the button press, are loaded into a register (using LDR) and stored in the respective split time memory address. For displaying the split time alongside the elapsed time, I had to update my Display Values function with the instructions to load the split time values from the memory address and display them on the input-output screen.



```
Split:                  //this function loads values from the elapsed time
//memory address and stores them to the split time memory addresses
      PUSH {R0-R5}
      LDR R0, 0x00900   //tens position of minutes of elapsed time
      STR R0, 0x00910   //tens position of minutes of split time
      LDR R0, 0x00904   //ones position of mintues of elapsed time
      STR R0, 0x00914   //ones position of minutes of split time
      LDR R0, 0x00908   //tens position of seconds of elapsed time
      STR R0, 0x00918   //tens position of seconds of split time
      LDR R0, 0x0090c   //ones position of seconds of elapsed time
      STR R0, 0x0091c   //ones position of seconds of split time
      POP {R0-R5}
      B increment
```

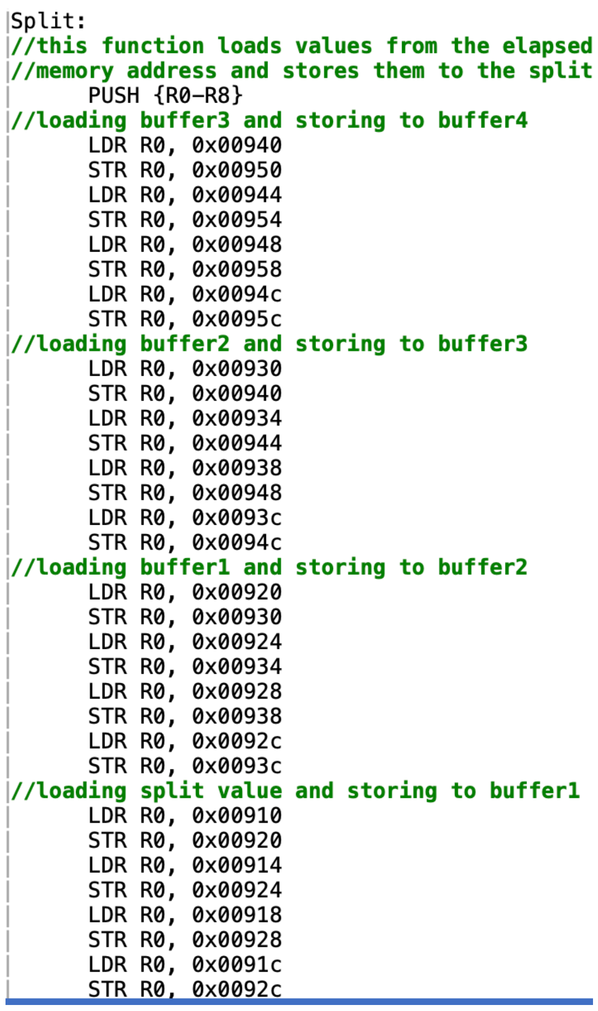
```
Elapsed Time: 0 0 : 1 4  Split Time: 0 0 : 1 0
```



```
DisplayValues:
      PUSH {R0-R5}
      MOV R0, #ElapsedTime
      STR R0, .WriteString
      LDR R0, 0x00900   //minutes tens position
      STR R0, .WriteUnsignedNum
      LDR R0, 0x00904   //minutes ones position
      STR R0, .WriteUnsignedNum
      MOV R0, #colon
      STR R0, .WriteString
      LDR R0, 0x00908   //seconds tens position
      STR R0, .WriteUnsignedNum
      LDR R0, 0x0090c   //seconds ones position
      STR R0, .WriteUnsignedNum
      MOV R0, #space
      STR R0, .WriteString
      MOV R0, #SplitTime
      STR R0, .WriteString
      LDR R0, 0x00910   //minutes tens position for split time
      STR R0, .WriteUnsignedNum
      LDR R0, 0x00914   //minutes ones position for split time
      STR R0, .WriteUnsignedNum
      MOV R0, #colon
      STR R0, .WriteString
      LDR R0, 0x00918   //seconds tens position for split time
      STR R0, .WriteUnsignedNum
      LDR R0, 0x0091c   //seconds ones position for split time
      STR R0, .WriteUnsignedNum
skip2:
      MOV R0, #newline
      STR R0, .WriteString
      POP {R0-R5}
      RET
```

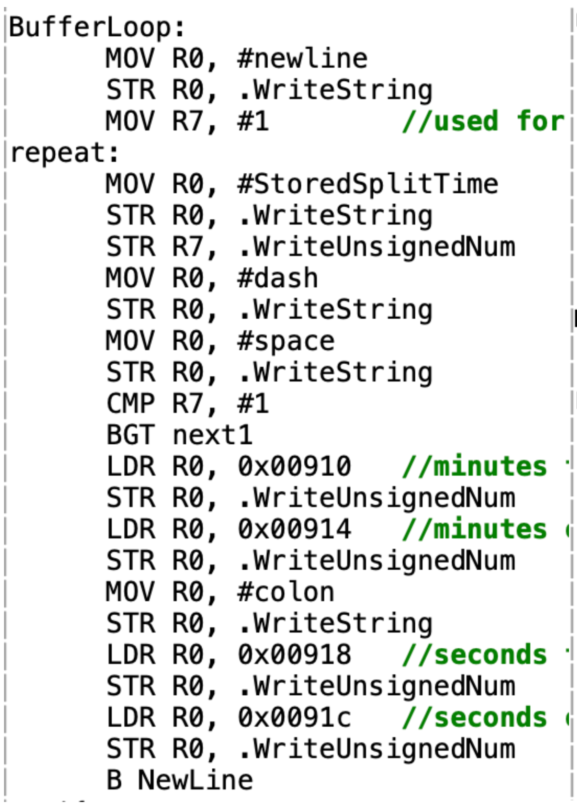
As a consequence to adding the split times, we have to reset the values of split time memory addresses in the 'Reset' label as well and initialize those memory addresses to zero at the start of the program.

Stage 4 (Stopwatch with Buffering) – The aim of this stage was to keep track of the last 5 stored split time in a single activity and allow the user to loop through those split times one at a time and to view them in turn. Again, to achieve this I had to allocate memory addresses for the buffer split time values, meaning that I had to initialize and reset these memory addresses to zero at the start of the program and in the 'Reset' label, respectively. The memory addresses used of the buffer split time are 0x00920 – 0x0095c. The 'Split' label was updated to load the values of the latest split time and store them in the first buffer split time and so on. To avoid having errors in storing these values, I had to write the code in the way that it first stores the last buffer time value with the second last buffer time value and so on. This was done because if I had done this the other way round then only the first stored split time would show the correct split time and the rest would just display the same value because they load in the updated stored time value rather than the original one. This is a fine example of showing that the sequencing of code matters in any programming language, more so than not in assembly language.

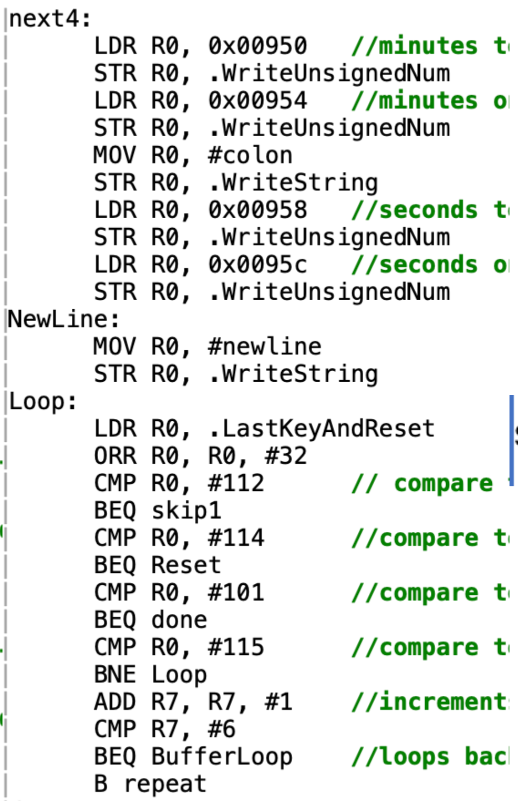


```
Split:
//this function loads values from the elapsed
//memory address and stores them to the split
      PUSH {R0-R8}
//loading buffer3 and storing to buffer4
      LDR R0, 0x00940
      STR R0, 0x00950
      LDR R0, 0x00944
      STR R0, 0x00954
      LDR R0, 0x00948
      STR R0, 0x00958
      LDR R0, 0x0094c
      STR R0, 0x0095c
//loading buffer2 and storing to buffer3
      LDR R0, 0x00930
      STR R0, 0x00940
      LDR R0, 0x00934
      STR R0, 0x00944
      LDR R0, 0x00938
      STR R0, 0x00948
      LDR R0, 0x0093c
      STR R0, 0x0094c
//loading buffer1 and storing to buffer2
      LDR R0, 0x00920
      STR R0, 0x00930
      LDR R0, 0x00924
      STR R0, 0x00934
      LDR R0, 0x00928
      STR R0, 0x00938
      LDR R0, 0x0092c
      STR R0, 0x0093c
//loading split value and storing to buffer1
      LDR R0, 0x00910
      STR R0, 0x00920
      LDR R0, 0x00914
      STR R0, 0x00924
      LDR R0, 0x00918
      STR R0, 0x00928
      LDR R0, 0x0091c
      STR R0, 0x0092c
```

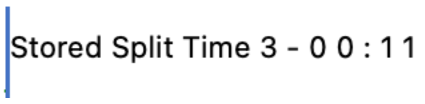
To display the stored split times separately from the main interface and to allow the user to loop through those split times, I had to create another label with the name ‘BufferLoop’ which was called when the stopwatch was in the paused state and the user presses the ‘S’ key. In this label, I basically use R7 as an index to choose and loop through the stored split times. The code basically first prints the latest split time, prints a new line and then checks for the user input to show the next spit time and if the user presses the ‘S’ key it then increments the R7 index by 1 and then loops back to the start of the buffer loop to compare the value of R7 and branch to the correct label for printing the next stored split time. For printing the split times, what’s done is basically loading in the values (using LDR) from the memory address of the required split times and then storing them (using STR) to the input-output screen using the ARMlite ‘.WriteString’ function. After incrementing the R7 index by 1, the code readily checks to see if it is equal to 6, i.e., all the five buffer split times have been displayed, then the index value of R7 is set back to 1 to reinitiate the loop.



```
BufferLoop:
      MOV R0, #newline
      STR R0, .WriteString
      MOV R7, #1         //used for
repeat:
      MOV R0, #StoredSplitTime
      STR R0, .WriteString
      STR R7, .WriteUnsignedNum
      MOV R0, #dash
      STR R0, .WriteString
      MOV R0, #space
      STR R0, .WriteString
      CMP R7, #1
      BGT next1
      LDR R0, 0x00910    //minutes
      STR R0, .WriteUnsignedNum
      LDR R0, 0x00914    //minutes
      STR R0, .WriteUnsignedNum
      MOV R0, #colon
      STR R0, .WriteString
      LDR R0, 0x00918    //seconds
      STR R0, .WriteUnsignedNum
      LDR R0, 0x0091c    //seconds
      STR R0, .WriteUnsignedNum
      B NewLine
```



```
next4:
      LDR R0, 0x00950    //minutes t
      STR R0, .WriteUnsignedNum
      LDR R0, 0x00954    //minutes o
      STR R0, .WriteUnsignedNum
      MOV R0, #colon
      STR R0, .WriteString
      LDR R0, 0x00958    //seconds t
      STR R0, .WriteUnsignedNum
      LDR R0, 0x0095c    //seconds o
      STR R0, .WriteUnsignedNum
NewLine:
      MOV R0, #newline
      STR R0, .WriteString
Loop:
      LDR R0, .LastKeyAndReset
      ORR R0, R0, #32
      CMP R0, #112       // compare
      BEQ skip1
      CMP R0, #114       //compare t
      BEQ Reset
      CMP R0, #101       //compare t
      BEQ done
      CMP R0, #115       //compare t
      BNE Loop
      ADD R7, R7, #1     //increment
      CMP R7, #6
      BEQ BufferLoop     //loops bac
      B repeat
```



```
Stored Split Time 3 - 0 0 : 1 1
```

ASSUMPTIONS MADE

Assumption made for this assignment is that the ‘Dumb Busy Wait Timer’ used in stage 1 provides a 1 second delay for every processor that runs this program.

UNRESOLVED PROBLEMS

There are no unresolved problems with my code to the best of my knowledge.

REFERENCES

References for this report are:

- https://swinburne.instructure.com/courses/49145/files/24432739?wrap=1
- https://www.asciitable.com
- https://metalup.org/al/AL_Student.pdf
- https://peterhigginson.co.uk/ARMlite/Programming%20reference%20manual_v1_2.pdf
- https://swinburne.instructure.com/courses/49145/pages/8-dot-5-busy-wait-timer-flashing-led-part-1?module_item_id=3155642
- https://swinburne.instructure.com/courses/49145/pages/8-dot-6-a-better-busy-wait-timer-flashing-led-part-2?module_item_id=3155644