Spring 2022 Haroutun Haroutunian

California State University Northridge
Department of Electrical and Computer Engineering

Experiment 7
Loops and Branches

Professor Flynn
18 March, 2022

Introduction

The objective of this experiment is to familiarize students with the use-cases of loops and branches. Loops are utilized to repeat an operation a certain amount of times. Branches are utilized to branch a program off to another operation in the meantime, and branch back to where it stopped in the original program. Loops are generally used as delays or with counters to count how many times an operation has been performed. On the other hand, branches are generally used to organize a program by separating functions from each other to make the code more legible for the programmer.

Procedure

Task 1 requires students to develop an ARM assembly program which adds every number starting from 1 to a user-defined "num" variable. For instance, if num was initialized to 3, the program would find the sum from 1 to 3, which would result in 6. In figure 7.1, directive DCD is utilized to manipulate the starting address easily without the use of ADD or SUB.

```
4   SUMADR   DCD 0    ;RESERVES 4BIT ADRESS
```

Figure 7.1: DCD directive for SUMADR

Since the program needs to repeatedly add and increment by one from the previous number, a loop would be optimal in this situation. A counter must be declared initially in order for the algorithm to know how many times the loop should be performed. As shown in the figure below, "num" is utilized as a "find & replace" for 9, which represents the amount of times the addition would be performed. Register 0 is loaded with num to specify the end point. Registers 1 and 2 are initialized to 0, which represents the starting points. Register 1 is the current while Register 2 acts like the next number to be added. Register 4 is loaded with the address of the resultant after the addition is performed.

```
19  num     EQU     0X9;sum will be from 1 to count
20              LDR     R0, =num
21              MOV     r1, #0x0
22              MOV     r2, #0x0
23              LDR     r4, =SUMADR
```

Figure 7.2: Counter initialization

In the figure below, on line 26, the sum label is placed at the starting point of the loop, which the end of the loop will point back to in the situation where we would need to keep incrementing. Register 2 is incremented by 1 initially to begin the addition, since register 2 points to the number needed to be added into Register 1, which occurs on line 27. ADD mnemonic is utilized rather than ADDS since no flags are needed to be set, currently. In line 28, Register 1 result is stored into Register 4, then points to the next address after Register 4's current address. In line 29, although SUBS mnemonic sets all the NZCV flags, the program only needs to look at the zero flag to determine whether the addition needs to be continued or not. The

counter is decremented and sets the flags accordingly. The counter is then updated to its current value, which is checked whether it is equivalent to be zero, which would branch out of the “sum” loop, or branched back to line 26 if not equal to zero.

```
26  sum         ADD     r2, r2, #0x1 ;Increment R2 (R2++)
27              ADD     r1, r1, r2 ;UPDATE R1 WITH R2 N+1(reccurance relation)
28              STR     r1, [r4],#4
29              SUBS    r3, r0, #0x1;DECREMENT COUNTER AND SET FLAGS
30              MOV     R0, r3  ;update counter
31              BNE     sum ;Loop if counter is NOT ZERO; EXIT if counter = 0(Z-flag)
```

Figure 7.3: Addition loop

In task 2, students are to run, debug and observe the program flow of the CPSR in the mystartup file. As seen in figure 7.4 CPSR is currently at address 0xD3, which is then changed to USER mode with interrupts enabled. This causes the program to initialize a stack before running the main program, which occurs in full descending order. Register 15 Program Counter is loaded with the starting address of 0x68, as shown in figure 7.5, which then jumps to 0xA0 when the main program finally begins to run to identify the starting address of the experiment.

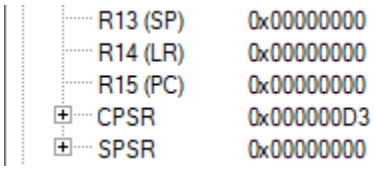

Figure 7.4: CPSR initial

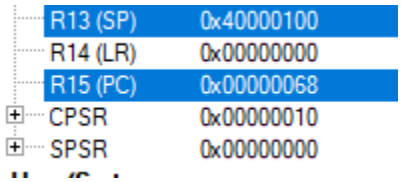

Figure 7.5: CPSR current

Task 3 asks to count how many times a character appears in a string. The DCB directive is utilized instead of DCD since each character in a string is represented by a Byte. When loading the string into a register, LDRB mnemonic is used for the same reason. myString is used as the label for the string, as shown in figure 7.6. The ALIGN directive makes sure that the string ends on a 4-byte boundary by appending 0’s if necessary.

```
80  myString    DCB "Professor Flynn is the best professor in Northridge",0
81              ALIGN
```

Figure 7.6: myString declaration/initialization

Similar to task 1, a loop is required to read each character in the string, which would be compared to the character we want to find. ASCII code is used to identify which character the program wants to find within the string, as seen in figure 7.7. In line 38, char is used as the label for the ASCII for the letter “o.” Register 0 is loaded with the myString label and Register 2 is then loaded with the ASCII for ‘o.’ Register 3 is initialized to 0 as it will be utilized as the counter to count how many times ‘o’ appears in the string. Register 1 is loaded with the contents of R0, which is the string. In line 43, CMP mnemonic is used, which subtracts Register 1 to Register 2. CMP sets all NZCV flags, but the relevant flag for this task would be the Zero-flag. When the contents of Register 1 subtracted from Register 2 equals zero, that means the flag would set, which would run line 44, which is dependent on the zero-flag. Line 44 simply increments the counter in Register 3.

```
33  ;TASK 3
34      GLOBAL  user_code
35      AREA    mycode, CODE, READONLY
36
37
38  char        EQU     0X6F        ; ascii for 'o'
39              LDR     R0,=myString    ;BASE ADRESS
40              LDR     R2,=char
41              MOV     R3,#0X0     ;INITIALIZE COUNTER
42              LDRB    R1,[R0]     ;LOADING BYTES
43              CMP     R1,R2       ;CHECKS IF THE CHAR IS A 'o'
44              ADDEQ   R3,#0X1     ;INCREMENTS COUNTER IF TRUE ^^^
45
46  charCount   LDRB    R1,[R0,#0X1]!   ;SHIFTS ADRESS TO NEXT BYTE IN STRING. SAVES RESULT IN R0 MEMORY
47
48              CMP     R1,R2   ;CHECKS IF CHAR IS IN CURRENT BYTE
49              ADDEQ   R3,#0X1 ;INCREMENTS R3 IF TRUE
50              CMP     R1,#0X0 ;CHECKS IF REACHED END OF STRING
51              BNE     charCount
```

Figure 7.7: user_code to count characters in string

Task 4 requires students to flash all the LEDs on the EduBoard 5 times, initially being off then turning toggling with 1 second delay in between. The following symbols are utilized for the task, determining the functionality of the pins.

- PINSEL0 = assigns the pins P0.8-P0.15 as GPIO(General Purpose Input/Output) pins
- IO0DIR = used to set the direction of the pins as with inputs or outputs
- IO0PIN =sends a 0 or 1 to the port pins connected to the LEDs
- IO0SET = used to set a “1” to a given bit in the IO0PIN memory address
- IO0CLR = used to clear a bit in the IO0PIN register, by placing “1” in memory
- FLASH_COUNT = Used to determine how many times the LEDs will flash\

PINSEL0 is the control register, which is configured to be set as GPIO. Within the GPIO are IO0DIR and IO0PIN registers. IO0DIR is the signal direction of the GPIO of each individual pin. Bits 8-15, which corresponds to the LED pins, would be set to “0” to be identified as an input, or “1,” and vice-versa, since the pins run through negative logic. When the bits in IO0DIR

are set to “0” for input, IO0PIN is able to be written. Pins P0.8 to P0.15 correspond to the eight LEDs that will be toggled in this task using IO0SET and IO0CLR. To set certain bits, a mask and IO0SET would be used, where “1” sets the bit and “0” has no effect. Likewise, to clear certain bits, IO0CLR would be used, where “1” clears the bit and “0” has no effect. From lines 63 to 71, this process is repeated in previous experiments, which loads IO0DIR into Register 5, and utilizes the LEDMASK label to declare bits 8-15 as inputs. Then, P0.8 through P0.15 are set off to begin with, then run into the Loop.

Within the Loop, Register 7 is loaded with 5, since the experiment asks students to toggle the LEDs 5 times. Immediately after, Register 7 is decremented and the LEDMASK which will be used to toggle the LEDs, is loaded into Register 4. Register 4 is then stored into Register 5 + IO0CLR, which turns the LEDs on. The program then runs through a delay loop which decrements a calculated number for 1 second, then proceeds to turn the LEDs back off. In line 84, the program then branches back to Loop if the content of Register 7 is not equivalent to 0, which continues to toggle the LEDs.

```
52  ;task 4
53  PINSEL0 EQU 0xE0020000
54  IO0DIR  EQU 0x8
55  IO0PIN  EQU 0xE0028000
56  IO0SET  EQU 0x4
57  IO0CLR  EQU 0xC
58  LEDMASK EQU 0x0000FF00
59
60  CLOCK   EQU 12000000
61  DELAY1S EQU (CLOCK/4)
62
63          MOV r4, #0
64          LDR r5, =PINSEL0
65          STR r4, [r5]
66          LDR r5, =IO0DIR
67          LDR r6, [r5, #IO0DIR]
68          LDR r4, =LEDMASK
69          ORR r6,r6,r4
70          STR r6, [r1,#IO0DIR]
71
72  LOOP    LDR r7, =5
73          SUBS r7,r7,#1
74          MOV r4, #LEDMASK
75          STR r4, [r5,#IO0CLR]
76          LDR r8, =DELAY1S
77  DELAY1  SUBS r8,r8,#1
78          BNE DELAY1
79          STR r4, [r5,#IO0SET]
80          LDR r8, =DELAY1S
81  DELAY2  SUBS r8,r8,#1
82          BNE DELAY2
83          BNE LOOP
```

Figure 7.8: LED loop

Discussion

Within the compiler, the address location of SUMADR in task 1 is automatically initialized as 0x40000000. Next to the address are the 2 LSB changes that occur during the program, as seen below in figure 7.9.

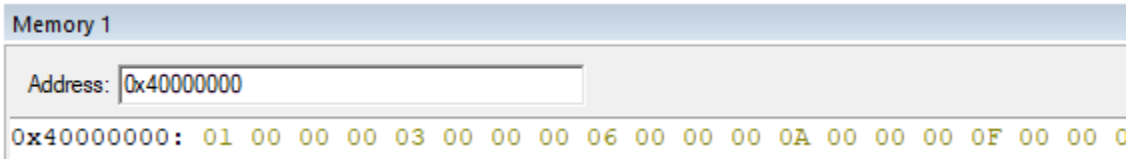

Figure 7.9: SUMADR address location

In task 3, the string "Professor Flynn is the best professor in Northridge" is utilized when counting the number of times the character 'o' occurs within the string in Register 2. The ALIGN directive makes sure that the string ends on a 4-byte boundary by appending 0's if necessary. A loop was used to repeatedly compare the character 'o' with the current character within the string using the CMP mnemonic. CMP subtracts 2 Registers and sets all NZCV flags, but the zero-flag was the only flag that was used within the experiment. Within the string, the character 'o' appears 5 times, as seen in figure 7.10.

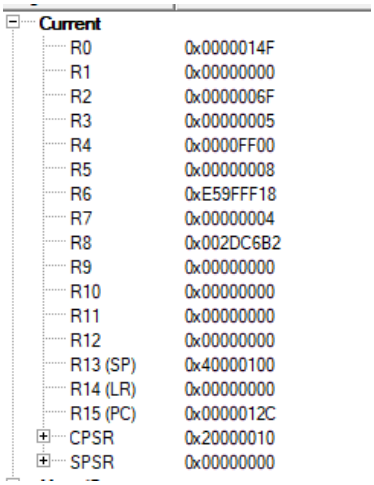

Figure 7.10: Register 3 character counter

Conclusion

Loops and branches not only allow programmers to organize their code, but to cause delays within the code or repetitions, such as counting. Throughout all tasks within the experiment, loops were utilized to create an additional loop, character counter and to toggle all LEDs on the EduBoard. EQU mnemonics also allow effective programmers to change a specific value within the code without searching through the entire code for a specific number, which can be confused for different purposes. Flags were also utilized in this experiment to branch back to a point of the program, as well as to determine whether the character 'o' was found anywhere within the string through the CMP mnemonic.