COEN 311
Lab 4: Flow control
T. Obuchowicz
2023

OBJECTIVES

• To gain an understanding of the flow control statements in assembly language (unconditional and conditional branches, the compare instruction, and looping constructs).

• To apply these concepts in order to write a complete ARM assembly language program which converts a string of ASCII characters into all upper case characters.

INTRODUCTION

Branch instructions

The previous labs all contained an *unconditional branch* instruction as the last instruction of the program:

```
stop:   b stop
```

When executed, this instruction would transfer program execution to the address specified by label `stop`, creating an infinite loop. The following C++ program contains a goto statement which pef-forms in a similar manner:

```
#include <iostream>
using namespace std;

int main()
{

stop: goto stop ;

return 0 ;  // this statement will never be executed
}
```

The general form of the branch instruction is:

`b{condition_code}   some_label` where:

condition_code is the specified condition (`al` (always), `eq` (equal) , `ne` (not equal), `gt` (greater), `ge` (greater or equal), etc.). If left unspecified, the default condition is `al` (always). Thus, the following is the same as the original unconditional branch:

```
stop:   bal stop
```

Consider the following C++ `if` statement:

```
if ( r0 > r1 )
{
 mov r2, #1 // this is the 'true' part of the if
}
else
{
 mov r2, #0  // this is the 'false' part of the if
}
mov r3, #5    // this is the 'next' statement , always
             // performed regardless of the outcome of the if
```

The program flow control and logic expressed by this if statement can be equivalently performed in assembly language as:

```
           mov r0, #5
           mov r1, #2
           cmp r0, r1    @ performs r0 - r1 and updates flags
           bhi r0_bigger @ branch to the 'true' part if
                         @ condition true
r1_bigger: mov r2, #0    @ this is the 'false' part
           b next        @ skip over the 'true' part
                         @ if the 'false' portion was executed
r0_bigger: mov r2, #1    @ this is the 'true' part

next:      mov r3, #5    @ this is the 'next' statement
```

Some comments on the code: we first load some random values into registers r0 and r1. The `cmp r0, r1` subtracts r1 from r0 and updates the processor flags accordingly. It should be noted that neither register is modified by the `cmp` instruction. A `conditional branch` instruction (`bhi branch higher`) is then used to either branch to the specified label if r0 > r1, or if the condition is false, the branch is **not** taken and program flow continues in a "straightline manner" and the value of 0 is loaded into register r2. An unconditional branch then "skips over" the 'true' part of the code. The conditional branch instruction examines the values of specific bits within the flags register of the processor. There are bits to indicate whether a carry or overflow has occurred during an arithmetic operation, or whether the result of an arithmetic operation is negative or positive, etc.

Loops

The `cmp` instruction together with a conditional branch can be combined to create a program loop. Consider the following C++ program which uses a loop to find the sum of an array of five integers:

```
#include <iostream>
using namespace std;
int main()
```

```
{
int num[5] = { 1,2,3,4,5 } ;
int sum = 0;
int count = 5;
int i = 0 ;

while ( count != 0)
{
 sum = sum + num[i];
 i++;
 count--;
}


cout << “Sum is: “ << sum << endl;

return 0 ;
}
```

The assembly language version of this program is:

```
@ Ted Obuchowicz
@ May 29, 2023
@ simple_loop.s
@ adds a list of 5 numbers stored in memory

.syntax unified
.cpu cortex-m4
.thumb

.word 0x20000400
.word 0x800000ed
.space 0xe4

.data
@ declare your data here
num:   .byte  1,2,3,4,5 @ declare 5 bytes of data starting
                        @ at address  num
sum:   .space 0x01      @ reserve 1 byte of RAM  without
                        @any initialization
.text
start:
mov r0, #0   @ r0 used to hold running sum, clear it to zero
ldr r1, =num @ load starting address of num into r1
mov r2, #5   @ r2 used as a counter to keep track of how
             @ many numbers from the list have been added up
```

```
@ we make use of register indirect addressing and conditional jumps
@ to setup a loop to add up the 5 numbers

top: ldrb r3, [r1]   @ get number from memory, store in r3
     add r0, r0, r3  @ sum = sum + number from list
      add r1, r1, #1  @ make r1 point to the next number in the list
     sub r2, r2, #1  @ decrement the loop counter
     cmp r2, #0      @ check to if end of list
      bne top        @ branch to top of loop if not at end of list
     ldr r1, =sum    @ load r1 with address of sum and store
                     @ the sum into memory
     strb r0, [r1]

stop:   b stop
.end
```

The code together with the comments are self-explanatory. Note that there are other ways of writing the program (perhaps making use of some other addressing mode which would require explicitly adding to to register r1).

Table 1 summarizes the conditions for branches based upon a single flag.

Table 1: branch conditions (single flag)

mnemonic	meaning	comments
`bal`	branch always	
`beg`	branch if equal	checks if zero (Z) flag = 1
`bne`	branch if not equal	checks if zero (Z) flag = 0
`bcs`	branch if carry set	checks if carry (C) flag = 1
`bcc`	branch if carry clear	checks if carry (C) flag = 0
`bmi`	branch if minus	negative result, checks if negative (N) flag = 1
`bpl`	branch if plus	positive or zero result, checks if negative (N) flag = 0
`bvs`	branch if overflow set	checks if overflow (V) flag = 1
`bvc`	branch is overflow clear	checks if overflow (V) flag = 0

When comparing numeric operands, the operands can be interpreted either as signed or unsigned values. Two different sets of conditions are used depending whether the operands are to be signed or unsigned values. Different combinations of processor flags are then tested to determine whether the specified branch condition is true or false. Table 2 summarizes the two sets of branch conditions.

Table 2:

result of comparison	signed operands	unsigned operands
>	`bgt` (greater than)	`bhi` (branch higher)
>=	`bge` (greater or equal)	`bhs` (branch higher or same)
<	`blt` (less than)	`blo` (branch lower)
<=	`ble` (less or equal)	`bls` (branch lower or same)

PROCEDURE

Consider the task of converting an array of ASCII characters from lower case into their upper-case equivalents. Any non-alphabetic characters are remain as is. If the character is already in upper-case form, then it should remain as is. The last character in the array is the NULL character (ASCII code = 0).

The lowercase ASCII characters `a', `b', `c', ... `z' are represented with the ASCII codes 97 through to 122. To convert a lower-case ASCII alphabetic character into its upper-case equivalent, we simply subtract decimal 32 from the lower case ASCII value. For example, For example, `A' = `a' - 32 (65 = 97 - 32).

The following C++ program illustrates the lower-case to upper-case conversion process. It makes use of the infamous goto statement which has considerable debate among computer scientists who who question the utility of the goto statement in high-level programming languages. Its use in this example, however is justified as there is a direct link between the C++ program and its assembly language equivalent.

```
// Author: Ted Obuchowicz
// lower-case to upper-case ASCII conversion

#include <iostream>
using namespace std;

int main()
{
```

```
char array[] = {‘j’,’A’,’c’,’K’,’ ‘,’f’, ‘L’, ‘a’, ‘S’, ‘h’,’#’,
‘1’, ‘\0’};

// print out the array before converting to all UPPER case

cout << array << endl;

int i = 0;

// figure out how to implement a while in assembly language
// hint... do a MOVE of the first element of the array
// and branch to somewhere if it’s zero
// if it’s not equal to 0, then perform the body of the loop


while ( array[i] != ‘\0’)   // while we did not yet reach the end of
                           // the string
{
     if ( array[i] <  ‘a’ ) // figure out how to do an if in assembly
      {
        goto next;  // skip over any non-lowercase letters, figure
                    // out how to do a goto in assembly
      }

     if ( array[i] > ‘z’ ) // figure out how to do another
                            // if in assembly
     {
       goto next;  // skip over any non-lowercase letters
     }

// if we got to here, we know the letter is a lower case one
// so convert it to its uppercase version by subtracting 32
// from its ASCII value.

     array[i] = array[i] - 32 ;

next: i = i + 1; // point to the next letter

}


// print it out to see if we did this correctly...

cout << array << endl;

return 0;
```

```
}


// yes, it works as this is the output

//JACK FLASH#1
```

Here is another version which makes greater use of the goto statement which leads to ‘spaghetti’ code, but illustrates in a clear manner the assembly language version:

```
// Author: Ted Obuchowicz
// lower-case to upper-case ASCII conversion
// uses the controversial goto statement

#include <iostream>
using namespace std;

int main()
{

char array[] = {‘j’,’A’,’c’,’K’,’ ‘,’f’, ‘L’, ‘a’, ‘S’, ‘h’,’#’,
‘1’, ‘\0’};

// print out the array before converting to all UPPER case

cout << array << endl;

int i = 0;

// figure out how to implement a while in assembly language
// hint... do a MOVE of the first element of the array
// and branch to somewhere if it’s zero
// if it’s not equal to 0, then perform the body of the loop


start: if ( array[i] == ‘\0’)
       {
         goto the_end;
       }
       else
       {
        if ( array[i] <  ‘a’ ) // figure out how to do an
                               //if in assembly
```

```
        {
         goto next;  // skip over any non-lowercase letters, figure
                     // out how to do a goto in assembly
        }

        if ( array[i] > ‘z’ ) // figure out how to do another
                              // if in assembly
        {
         goto next;  // skip over any non-lowercase letters
        }

// if we got to here, we know the letter is a lower case one
// so convert it to its uppercase version by subtracting 32
// from its ASCII value.

        array[i] = array[i] - 32 ;

        next: i = i + 1; // point to the next letter

        }

        goto start;


// print it out to see if we did this correctly...

the_end: cout << array << endl;


return 0;
}


// yes, it works as this is the output

//JACK FLASH#1
```

You are to write an ARM assembly language program which makes use of a loop to access the individual elements of the array containing the ASCII characters. You are to initialize the array with the assembler directive:

```
.data

@ the .ascii directive declares an ASCII string
@ for the purposes of this lab, the NUL character shall be used to
indicate the end of the string
```

```
@ the difference between upper case ASCII and lower case ASCII is 32

message:   .ascii "juMping JAck flaSh #1"
lastchar:  .byte 0
```

The `lastchar` byte with all 0s (the so called NULL character in ASCII) is used to represent the end of the string.

Within the loop, the program is to determine whether the current character represents a lower case character, if the character is lower case, it is to be converted into its upper case version. Non-alphabetic characters are to remain as is.

Assemble, load, and single step through your program using gdb. Examine the contents of memory in which the message is stored using either the `x/22xc` command (examine 22 characters), or the `x/1xs` command (examine 1 string). For example:

```
(gdb) x/22xc &message
0x20000110:      106 'j' 117 'u' 77 'M'  112 'p' 105 'i' 110 'n' 103 'g' 32 ' '
0x20000118:      74 'J'  65 'A'  99 'c'  107 'k' 32 ' '  102 'f' 108 'l' 97 'a'
0x20000120:       83 'S'  104 'h' 32 ' '  35 '#'  49 '1'  0 '\000'
(gdb) x/1xs &message
0x20000110:       "juMping JAck flaSh #1"
```

Single step completely through the program to verify that the characters are correctly converted to their upper case equivalents. Upon completion of the program, the message should contain:

```
(gdb) x/22xc &message
0x20000110:      74 'J'  85 'U'  77 'M'  80 'P'  73 'I'  78 'N'  71 'G'  32 ' '
0x20000118:      74 'J'  65 'A'  67 'C'  75 'K'  32 ' '  70 'F'  76 'L'  65 'A'
0x20000120:       83 'S'  72 'H'  32 ' '  35 '#'  49 '1'  0 '\000'
(gdb) x/1xs &message
0x20000110:       "JUMPING JACK FLASH #1"
```

QUESTIONS

1. Will the following ARM assembly language program produce the same output as that given in the Introduction section of this lab (`simple_loop.s`)?

```
@ Ted Obuchowicz
@ May 29, 2023
@ simple_loop_wrong_order.s
@ adds a list of 5 numbers stored in memory

.syntax unified
.cpu cortex-m4
.thumb
```

```
.word 0x20000400
.word 0x800000ed
.space 0xe4

.data
@ declare your data here
num:   .byte  1,2,3,4,5
sum:   .space 0x01
.text
start:
        @  assembly code goes here
        mov r0, #0
        ldr r1, =num
        mov r2, #5

@ we make use of register indirect addressing and conditional jumps
@ to setup a loop to add up the 5 numbers

top:    ldrb r3, [r1]
        add r0, r0, r3
        subs r2, r2, #1
        add r1, r1, #1
        bne top

        ldr r1, =sum
        strb r0, [r1]

stop:   b stop
.end
```

T. Obuchowicz
May 30, 2023