Pre-lab Assignment 2

Due: Mon, 14 Sep 2020 17:26:34 (approximately 88 days ago)
[Score: 40 / 40 points possible]

Instructions:

Review the lab 2 background information and lecture 4 to make sure that you understand assembler directives.

For the following questions, assume that each code segment is independent of the others. Each one can be tried with the simulator, but since they are meant to prepare you for the lab experiment, you would be advised to try them in SystemWorkbench and practice using the memory browser. For each problem, insert the code just under the global main label and either single step through it or run it and let it wait at the bkpt.

It is possible to just let the simulator do all the work, but there is still a goal of understanding what is happening. Each problem is presented for a reason, so if you are surprised or confused by a result, you should think about it, do some research, and ask questions.

Several problems involve writing assembly language programs. Expect this to take a while. You will be doing things like this in lab 2.

Academic Integrity Statement [0 ... -100 points]

By typing my name, below, I hereby certify that the work on this prelab is my own and that I have not copied the work of any other student (past or present) while completing it. I understand that if I fail to honor this agreement, I will receive a score of zero for the lab, a one letter drop in my final course grade, and be subject to possible disciplinary action.

Raghuram Selvaraj 

Question 1 [5 points]

Look at the instruction sequence below. What are the values of the words represented by labels **a** and **b** after execution of the instructions. Specify the results as decimal values.

```
.data
.align 4
.global a
a: .word 123
.global b
b: .word 246

.text
.global main
main:
    ldr  r0,=a
    ldr  r1,[r0]
    adds r0,#4
    adds r1,#15
```

```
    str  r1,[r0]
    bkpt
```

a: 123

b: 138

Question 2 [5 points]

What are the values of the words represented by the labels **a**, **b**, and **c** after execution of the instructions below?

```
.data
.align 4
.global a
a: .word 20
.global b
b: .word 30
.global c
c: .word 0

.text
.global main
main:
    ldr  r0,=a
    ldr  r1,[r0]
    ldr  r0,=b
    ldr  r2,[r0]
    adds r3,r1,r2
    ldr  r0,=c
    str  r3,[r0]
    ldr  r0,=b
    str  r1,[r0]
    ldr  r0,=a
    str  r2,[r0]
    bkpt
```

Express the words as decimal integers.

a: 30

b: 20

c: 50

Question 3 [5 points]

You can think of "arr" as a label that represents the start of a global array of 10 integers (4-byte words). What values are found in the 10 array elements after execution of the following program? Note that the simulator's memory viewer shows things as hexadecimal bytes. Specify the values, below, as decimal. In other words, if the value 20 (decimal) was stored in memory, it would appear as 14 (hexadecimal) in the simulator's memory viewer.

```
.data
.align 4
.global arr
arr: .space 40 // 10 words

.text
.global main
main:
    ldr  r0,=arr     // r0 is the address of the start of arr
    movs r1,#0       // r1 = 0
write_loop:
    cmp  r1,#10      // while r1 < 10
    bge  done        // or we're done
    movs r2,r1
    muls r2,r2
    lsls r3,r1,#2    // multiply r1 by 4...
    str  r2,[r0,r3]  // ...to use as an offset for the store
    adds r1,#1       // r1 = r1 + 1
    b    write_loop  // and move on to the next element
done:
    bkpt
```

Please state the contents as decimal numbers.

arr[0]: 0
arr[1]: 1
arr[2]: 4
arr[3]: 9
arr[4]: 16
arr[5]: 25
arr[6]: 36
arr[7]: 49
arr[8]: 64
arr[9]: 81

Question 4 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. Do this in a similar manner as hw2. Define the global variables as labels in the data segment. Implement the statements as assembly language instructions that follow the "main" label. Make sure the assembly language instructions terminate with a "bx lr" instruction so that the **`main`** subroutine returns to the startup loop.

Do your best to follow rules of the ARM Cortex-M0 ABI. Use only the registers R0, R1, R2, and R3 to hold values. If you need to use more registers (you don't) save R4 - R7 with a PUSH and restore them on return with a POP instead of a "bx lr".

```
int x = 5;
int y = 3;

void main() {
```

```
  if (x > 2)
    y = y + 8;
}
```

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
x: .word 5
y: .word 3

.text
.global main
main:
ldr r0, =x
ldr r1, [r0]
adds r0, #4
ldr r2, [r0]
movs r3, #2
b if1
bx lr

if1:
push {lr}
cmp r1, r3
ble endif1
bgt then1
pop {pc}

then1:
push {lr}
movs r3, #8
adds r2, r3
str r2, [r0]
b endif1
pop {pc}

endif1:
bx lr
```

Save first, then click here to try it in the simulator

Remember to copy code updated in the simulator into the box above!

Question 5 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. Here, we specify 'int x' in the for loop so it does not need to be implemented as a global variable. You may use a single register to implement x.

```
int arr[20]; // Each int is a four-byte quantity

void main() {
  for(int x=0; x<20; x++)
    arr[x] = x*x;
```

```
}
```

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
.align 4
arr: .word 0

.text
.global main
main:
    push {lr}
    for1:
        movs r0, #0
    check1:
        movs r1, #20
        cmp r0, r1
        bge endfor1
    body1:
        ldr r1, =arr
        movs r2, #4
        muls r2, r0
        movs r3, r0
        muls r3, r0
        str r3, [r1, r2]
    next1:
        adds r0, #1
        b check1
    endfor1:
    pop {pc}
```

[Save first, then click here to try it in the simulator]

Remember to copy code updated in the simulator into the box above!

Question 6 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. This subroutine will be like the one in Question 5, but you should note that each element in **`arr`** is a single byte rather than a four-byte word. Therefore, you should use the **`STRB`** instruction to store each array element instead of **`STR`**.

```
char arr[20]; // Each char is a single byte

void main() {
  for(int x=0; x<20; x++)
    arr[x] = x + 65;
}
```

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
.align 1
arr: .byte 0

.text
.global main
main:
    for1:
        movs r0, #0
    check1:
        movs r1, #20
        cmp r0, r1
        bge endfor1
    body1:
        movs r1, r0
        adds r1, #65

        ldr r3, =arr

        strb r1, [r3, r0]
    next1:
        adds r0, #1
        b check1
    endfor1:
        bx lr
```

Save first, then click here to try it in the simulator

Remember to copy code updated in the simulator into the box above!

Question 7 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. Implement the 'arr' array with the .string directive if you use the simulator. (We forgot to add .asciz to the simulator.) The result of the execution of the instructions should convert all of the lowercase letters to uppercase.

```
char arr[] = "This is a test."

void main() {
  for(int x=0; arr[x] != 0; x++) {
    if (arr[x] > 96 && arr[x] < 123) {
      arr[x] = arr[x] - 32;
    }
  }
}
```

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
.align 1
arr: .string "This is a test."

.text
.global main
main:
    for1:
        movs r0, #0
    check1:
        ldr r1, =arr
        ldrb r2, [r1, r0]
        movs r3, #0
        cmp r2, r3
        beq endfor1
    body1:

        if1:
            movs r3, #96
            cmp r2, r3
            ble endif1

            movs r3, #123
            cmp r2, r3
            bge endif1
        then1:
            subs r2, #32
            strb r2, [r1, r0]
        endif1:
    next1:
        adds r0, #1
        b check1
    endfor1:
    bx lr
```

Save first, then click here to try it in the simulator

Remember to copy code updated in the simulator into the box above!

Question 8 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. The result of the execution of the instructions should convert all of the odd integers to even integers by ANDing them with 0xfffffffe (or using BICS to AND them with ~1). Since you are being a compiler, you may also assume that "sizeof arr / sizeof arr[0]" can be expressed as a single integer contant in your solution.

Note that you can say something like

```
arr: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

to implement an initialized array of four-byte integers.

Also note that it is difficult to use only four registers (R0 - R3) to implement this subroutine. It is possible. Do your best, but if it is too difficult, you should **`PUSH {R4-R7,LR}`** at the beginning of the subroutine and then use **`POP {R4-R7,PC}`** to return instead of **`BX LR`**. That will allow you to use registers R4 - R7 without affecting the caller. (This will be important for the lab experiment.)

```
int arr[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };

void main() {
  for(int x=0; x < sizeof arr / sizeof arr[0]; x++) {
    if (arr[x] & 1) {
      arr[x] = arr[x] & ~1;
    }
  }
}
```

```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.data
.align 4
arr: .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

.text
.global main
main:
    for1:
        movs r0, #0
    check1:
        movs r1, #16
        cmp r0, r1
        bge endfor1
    body1:
        if1:
            ldr r1, =arr
            movs r2, #4
            muls r2, r0
            ldr r3, [r1, r2]

            movs r1, #1
            ands r1, r3

            cmp r1, #1
            bne endif1
        then1:
            bics r3, r1
            ldr r1, =arr
            str r3, [r1, r2]
        endif1:
    next1:
        adds r0, #1
        b check1
    endfor1:

    bx lr
```

Save first, then click here to try it in the simulator

Remember to copy code updated in the simulator into the box above!