Lab 2

Wednesday, January 26, 2022 12:23 PM

Prelab2

Pre-lab Assignment 2

Due: Fri, 28 Jan 2022 11:30:00 (approximately 1 days from the time this page was loaded)
[40 points possible] [100 penalties possible] [0 penalties graded so far]

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 well-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 breakpoint (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.

The best strategy for the initial problems where you must find memory values is to look at the instruction sequences and decide what you *think* they will do. After that, try them, and check if the results match your expectations. If not, you've found an opportunity to learn something new.

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.

Tzu Yu Chen

Save

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
.balign 4
.global a
a: .word 22
.global b
b: .word 36

.text
.global main
main:
    ldr  r0,=a
    ldr  r1,[r0]
    adds r0,#4
    adds r1,#17
    str  r1,[r0]
    subs r0,#4
    adds r1,#49
    str  r1,[r0]
    bkpt
```

Remember: Decimal values, please.

a:
b:
Save

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
.balign 4
.global a
a: .word 30
.global b
b: .word 14
.global c
c: .word 20

.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:
b:
c:
Save

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 you read a word as 14 (hexadecimal) in the simulator's memory viewer, you should write it as 20 (decimal) below.

```
.data
.balign 4
.global arr
arr: .space 40    // enough for 10 four-byte 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,#19      // r2 = 19
    muls r2,r1
    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]:
arr[1]:
arr[2]:
arr[3]:
arr[4]:
arr[5]:
arr[6]:
arr[7]:
arr[8]:
arr[9]:

Save

Question 4 [5 points]

Implement the following C code as ARM Cortex-M0 assembly language. **For this, and the three following exercises,**

- Allocate space for each global variable in the .data segment using a label and a .word directive.
- Implement the statements as assembly language instructions that follow the "main" label.
- Make sure the assembly language instructions terminate with a "BX LR" instruction. The "LR" register is the "link register", and the "BX" instruction jumps to the contents of the register specified. This treats **`main`** as a *subroutine* that *returns* to its caller, the startup code. The "BX LR" instruction is one typical way of returning from a simple subroutine. You don't need to understand how subroutines work yet. Just be aware that this is what is happening.
- Do your best to follow the rules of the ARM Cortex-M0 Application Binary Interface (ABI) rules. One of those rules is to use only the registers R0, R1, R2, and R3 to hold values. We'll give you a hint later as to how you can be allowed to use R4 - R7.

```
int x = 3;
int y = 7;

void main() {
  if (x < 5)
    y = y + 8;
}
```

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

Save
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 choose a single register to implement x. Remember that each array entry is a four-byte quantity. When you store 32-bit (four-byte) register values to memory, the address must be evenly divisible by four. You must multiply the loop index by four to compute the correct offset.

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

void main() {
  for(int x=0; x<20; x++)
    arr[x] = x*(x+x); // pick a register to hold the intermediate value (x+x)
}
```

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

Save
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`**.

If you look at the result of this code in the memory browser, you should see the lowercase letters 'abcdef...xyz'. They may be displayed in a strange order. Check to make sure that you are looking at them correctly and that you stored them correctly.

```
char arr[26]; // Each char is a single byte

void main() {
  for(int x=0; x<26; x++)
    arr[x] = x + 97;
}
```

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

Save
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. The result of the execution of the instructions should convert all of the uppercase letters in the array to meaningful words. To initialize an array with a zero-terminated string, use the **.string** or **.asciz** directive as described in the lab document. Since you will load and store one byte at a time, and since this byte will not be aligned on a divisible-by-four boundary, you should use the **`ldrb`** and **`strb`** instructions to access memory. They load and store a single byte and have no alignment constraints.

```
char arr[] = "EBIIL, TLOIA!"

void main() {
  for(int x=0; arr[x] != 0; x++) {
    if (arr[x] >= 65 && arr[x] <= 90) {
      arr[x] = arr[x] + 3;
    }
  }
}
```

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

Save
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 take all of the values where the least significant bit is set, increment them by one and store them back. 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. Whenever you see a construct like that, you should say to yourself, "Aha, that's a constant that represents the number of elements in the array," and write that numeric constant down.

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 possible, but difficult, to use only four registers (R0 - R3) to implement this subroutine. Do your best, but if it is too difficult, you should use **`PUSH {R4-R7,LR}`** at the beginning of the subroutine, and use **`POP {R4-R7,PC}`** to return instead of **`BX LR`**. Doing so 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) == 1) {
      arr[x] = arr[x] + 1;
    }
  }
}
```

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

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

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