**Midterm Review Questions (with Answers)**

The midterm exam will be broken into two components:

* A lab-based portion, on August 04, 2020
* A written (lecture-based) portion, on August 05, 2020

The lab-based portion will require you to write assembly code on your laptops or lab machines, which is to be turned in via Canvas by the end of the class. It will be similar in style to the rest of the assignments in the course.

The written portion will require you to:

* Understand number representation and numeric operations (from the first three labs)
* Read and understand assembly code
* Answer short-answer questions related to numeric operations and assembly

The lecture-based portion is heavily biased towards numeric representation, though you should expect some assembly-based questions.

You may bring the following materials into the exam:

* A calculator with exponentiation capabilities.
* The attached handout, which consists of the ARM reference card along with all the SWI codes you may need. A copy of this handout will be distributed at the beginning of the exam.

The review below, **in addition to everything you wrote for your labs**, is intended to be comprehensive. All topics which could potentially be on the exam are somehow covered by this review.

**Questions**

1. The leftmost bit of a 32-bit number is in what position?

31

1. Shifting an unsigned binary number N two positions to the left is equivalent to multipling N by what (in decimal)?

4 (each shift to the left is another multiplication by two, so 22)

1. Shifting an unsigned binary number N four positions to the right is equivalent to performing truncating division (ignoring the remainder) by what (in decimal)?

16 (each shift to the right is another division by two, so 24)

1. For ANY unsigned binary number, which bit must you look at in order to determine if the number is odd or even?

Bit 0 (the rightmost bit)

1. What is -8 in twos complement representation? Represent your solution using 8 bits.

8 in binary: 0000 1000

flip bits: 1111 0111

add 1: 1111 1000

1111 1000

1. What is 1 + 1 with a carry-in bit set?

1

1

+ 1

---

1 with carry-out set

1. What is 1 + 1 without a carry-in bit set?

0

1

+ 1

---

0 with carry-out set

1. What is 1 + 0 without a carry-in bit set?

0

1

+ 0

---

1 without carry-out set

1. What is:

11111101

+ 01000101

Specify if the result has a carry-out set and if the result sets the overflow bit.

1 11111010

11111101

+ 01000101

----------

01000010

Carry-out set, overflow bit not set.

1. What is:

10010110

- 11101010

Specify if the result has a carry-out set and if the result sets the overflow bit.

10010110

- 11101010

...is equivalent to...

10010110

+ (-11101010)

Original: 11101010

Flip bits: 00010101

Instead of adding 1 here, I'll set the carry-in bit for the add

0 00101111

10010110

+ 00010101

----------

10100100

Carry-out not set, overflow bit not set.

1. Consider an unknown binary number N. Using only bitwise operations and bitmasks, give an expression that will produce N, *except* that bit 7 is guaranteed to be one. Express any bitmasks using 2-digit hexadecimal.

We have a binary number that looks like this:

XXXX XXXX

...where X is an unknown bit.

We want to produce a binary number that looks like this:

1XXX XXXX

We'll need to use OR (|) for this, as OR can be forced to produce 1

as a result with an unknown number as with ((X | 1) = 1).

This same reasoning gives us the bitmask to OR with.

We end up with:

XXXX XXXX

| 1000 0000

-----------

1XXX XXXX

1000 0000 in hexadecimal is 0x80.

So overall we have:

N | 0x80

1. While this isn't a review question, be familiar with the [process to convert between binary and decimal floating point representations](https://kyledewey.github.io/comp122-fall17/lecture/week_2/floating_point_interconversions.html).
2. What is wrong with the following code, if anything?

.equ Exit, 0x11

.equ Open, 0x66

.equ Close, 0x68

.equ Read\_Int, 0x6C

.data

filename:

.asciz "myFile.txt"

.text

.global \_start

\_start:

;; open the file

ldr r0, =filename

mov r1, #0

swi Open

;; read an integer from it

swi Read\_Int

;; close the file

swi Close

;; exit the program

swi Exit

.end

swi Read\_Int will overwrite r0 with the integer read in.

r0 contains the filehandle to the file.

As such, the subsequent Close won't use the filehandle from open, but will instead

treat whatever integer that was read in as a filehandle.

1. What is wrong with the following code, if anything?

.equ Write\_Int, 0x6B

.text

.global \_start

\_start:

;; print out 42

mov r0, #1

mov r1, #42

swi Write\_Int

Fails to exit the program, and missing .end

1. Write ARM assembly code which will read two integers from the file myFile.txt and print them out.

.equ Print\_Chr, 0x00

.equ Exit, 0x11

.equ Open, 0x66

.equ Close, 0x68

.equ Write\_Int, 0x6B

.equ Read\_Int, 0x6C

.data

filename:

.asciz "myFile.txt"

.text

.global \_start

\_start:

;; r0, r1: temporaries for SWI instructions

;; r2: filehandle

;; open the file

ldr r0, =filename

mov r1, #0

swi Open

mov r2, r0

;; read the first integer

;; filehandle initially is already in r0

swi Read\_Int

;; print out the first integer

mov r1, r0 ; move integer into place

mov r0, #1

swi Write\_Int

;; print out a newline

mov r0, #'\n

swi Print\_Chr

;; read the second integer

mov r0, r2

swi Read\_Int

;; print out the second integer

mov r1, r0 ; move integer into place

mov r0, #1

swi Write\_Int

;; close the file

mov r0, r2

swi Close

;; exit the program

swi Exit

.end

1. Consider the following code, which sets up a .data section:

.data

label1:

.asciz "Hi"

label2:

.word 1, 2

label3:

.asciz "Bye"

Assuming the .data section starts at address 0, how does this look in memory? Use the following table as a template.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | 'H' | 'i' | '\0' | 0x00 | 0x00 | 0x00 | 0x01 | 0x00 | 0x00 | 0x00 | 0x02 | 'B' | 'y' | 'e' | '\0' | ??? | ??? | ??? | ??? | ??? | ??? |
| **Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly.

if (r0 >= 5) {

r1 = r6;

} else {

r2 = r7;

}

;; many different solutions are possible; this is just one

; r0 >= 5, AKA

; !(r0 < 5)

cmp r0, #5

movpl r1, r6

movmi r2, r7

1. Convert the following Java/C-like code into ARM assembly. **Use branch intructions instead of conditional execution.** The names of the variables reflect which registers must be used for the ARM assembly.

if (r5 < r6) {

r2 = r3;

print\_string("Less");

} else if (r5 == r6) {

r3 = r4;

print\_string("Equal");

} else {

r4 = r5;

print\_string("Greater");

}

.equ SWI\_Print\_String, 0x02

.equ SWI\_Exit, 0x11

.data

less\_string:

.asciz "Less"

equal\_string:

.asciz "Equal"

greater\_string:

.asciz "Greater"

.text

.global \_start

\_start:

cmp r5, r6

bmi less\_branch

beq equal\_branch

;; fallthrough to else

mov r4, r5

ldr r0, =greater\_string

swi SWI\_Print\_String

b program\_exit

less\_branch:

mov r2, r3

ldr r0, =less\_string

swi SWI\_Print\_String

b program\_exit

equal\_branch:

mov r3, r4

ldr r0, =equal\_string

swi SWI\_Print\_String

program\_exit:

swi SWI\_Exit

.end

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly.

for (int r2 = r1; r2 <= 150; r2 += 4) {

int r3 = (r2 - 1) \* (r2 + 1);

print\_int(r3);

print\_char('\n');

}

.equ Write\_Int, 0x6B

.equ Print\_Char, 0x00

.equ Exit, 0x11

mov r2, r1 ; r2 = r1

loop:

;; r2 <= 150 AKA

;; !(r2 > 150) AKA

;; !(150 < r2)

;; r4 isn't used above, so I'm using it as a temp

mov r4, #150

cmp r4, r2

bmi loop\_end

;; r4 = r2 - 1

sub r4, r2, #1

;; r5 isn't used above, so I'm using it as a temp

;; r5 = r2 + 1

add r5, r2, #1

mul r3, r4, r5

;; print out the integer

mov r1, r3

mov r0, #1

swi Write\_Int

;; print out a newline

mov r0, #'\n

swi Print\_Char

add r2, r2, #4

b loop

loop\_end:

swi Exit

.end

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly. Non-register variable names indicate a value that should be stored in memory.

int[] myArray = new int[]{19, 21, -5, 4};

int r2 = 0;

int r3 = 0;

do {

r2 += myArray[r3];

r3++;

} while (r3 < 4);

print\_int(r2);

.equ Exit, 0x11

.equ Write\_Int, 0x6B

.data

myArray:

.word 19, 21, -5, 4

.text

.global \_start

\_start:

mov r2, #0

mov r3, #0

;; r4: myArray

ldr r4, =myArray

loop:

;; r5: temp

ldr r5, [r4, r3, LSL #2]

add r2, r2, r5

add r3, r3, #1

cmp r3, #4

bmi loop

mov r0, #1

mov r1, r2

swi Write\_Int

swi Exit

.end

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly. Non-register variable names indicate a value that should be stored in memory.

int myArray[4] = {19, 21, -5, 4};

int\* r2 = myArray;

int r3 = 4;

int r4 = 0;

do {

r4 += \*r2;

r2++;

r3--;

} while (r3 != 0);

print\_int(r4);

.equ Exit, 0x11

.equ Write\_Int, 0x6B

.data

myArray:

.word 19, 21, -5, 4

.text

.global \_start

\_start:

ldr r2, =myArray

mov r3, #4

mov r4, #0

loop:

;; using r5 as a temp

ldr r5, [r2]

add r4, r4, r5

add r2, r2, #4

sub r3, r3, #1

cmp r3, #0

bne loop

mov r0, #1

mov r1, r4

swi Write\_Int

swi Exit

.end

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly.

if (r2 < r3 && r3 < r4) {

r5 = r6;

} else {

r6 = r5;

}

cmp r2, r3

bpl else\_branch ; if !(r2 < r3)

cmp r3, r4

bpl else\_branch ; if !(r3 < r4)

;; fallthrough to true

mov r5, r6

b after\_if

else\_branch:

mov r6, r5

after\_if:

1. Convert the following Java/C-like code into ARM assembly. The names of the variables reflect which registers must be used for the ARM assembly.

if (r2 < r3 || r3 < r4) {

r5 = r6;

} else {

r6 = r5;

}

cmp r2, r3

bmi true\_branch

cmp r3, r4

bmi true\_branch

;; fallthrough to false

mov r6, r5

b after\_if

true\_branch:

mov r5, r6

after\_if: