Homework Assignment 2

Due: Fri, 11 Sep 2020 23:59:59 (approximately 90 days ago)
[Score: 112 / 140 points possible]
Weight = 1.0

For the following questions, assume that each code segment is independent of the others. Each one can be tried with the simulator or System Workbench by adding a global main label to the beginning of the code segment and single stepping through the code.

Academic Honesty Statement [0 ... -140 points]

By typing my name, below, I hereby certify that the work on this homework 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 assignment, a one letter drop in my final course grade, and be subject to possible disciplinary action.

Raghuram Selvaraj 

---

(1) Instruction encodings [60 points]

For each ARM Cortext-M0 instruction below, write its instruction encoding. For instance, the encoding for the **`MOVS R6,R0`** instruction would be **`0x0006`**. For each instruction, also write a C statement to represent the operation it performs in the box below the encoding. For instance, the **`MOVS R6,R0`** instruction might be represented as **`r6 = r0;`**. You should omit the terminating semicolon. You will need only one C statement to implement each instruction. Furthermore, for the purposes of describing the operation, treat registers as though they were variables. You can also assume that the following C variable declarations are in place:

```
signed int r0;
signed int r1;
signed int r2;
signed int r3;
signed int r4;
signed int r5[1000]; // each element is 4 bytes in size
signed short int r6[1000]; // each element is 2 bytes in size
signed char r7[1000]; // each element is 1 byte in size
```

The array size declaration is important. It means, for instance, that the address for r5[7] is four bytes higher than r5[6]. Take this into account with the LDRx and STRx instructions. Also note that the *signed* load instructions do the natural thing and preserve the sign of the loaded 8- or 16-bit value as it is promoted to a 32-bit space in a register. An *unsigned* load will set the extended bits to zero.

If you want to disassemble these instructions in System Workbench or on the simulator, feel free to do so, but **you should be prepared to come up with the encoding all on your own simply by looking at the *ARMv6-M Architecture Reference Manual***. That's what you'll have to do on the midterm lab practical exam.

When recording your answers, use a 4-digit hexadecimal number (with no 0x prefix) for the encoding. Don't get too creative with the C statement. We'll try to automatically grade this work.

A few hints:

When thinking about the difference between LDRB and LDRBS, remember that LDRBS will turn a signed 8-bit inteter into a signed 31-bit integer---just like C will if you say "r0 = r7[n];". LDRB will AND the value assigned to the register with 0xff to zero off the upper 24 bits. Something similar happens with LDRH for the upper 16 bits. See that example? Pay attention to that.

In the LDRSH example below, the remember that C accesses the nth entry in an array with arr[n]. Square brackets in ARM Cortex-M0 assembly language represent a byte offset. That's why the LDRSH example below divides r3 by 2 for the C description. As a TA for some examples about this if you need some help with this concept.

Question	Instruction	Encoding	C statement
example	`MOVS R3,#17`	2311	`r3 = 17`
example	`LDRSH R0,[R6,R3]`	5ef0	`r0 = r6[r3/2]`
example	`LDRH R1,[R6,#12]`	89b1	`r1 = r6[12/2] & 0x0000ffff`
1.01	`MOVS R1,#0x3C`	213C ✓	r1 = 0x3c ✓
1.02	`ADDS R2,R4,#6`	1da2 ✓	r2 = r4 + 6 ✓
1.03	`ADDS R2,#25`	3219 ✓	r2 += 25 ✓
1.04	`SUBS R2,#14`	3a0e ✓	r2 -= 14 ✓
1.05	`LSLS R4,R0,#3`	00c4 ✓	r4 = r0 << 3 ✓
1.06	`ASRS R4,R0,#2`	1084 ✓	r4 = r0 >> 2 ✓
1.07	`RSBS R2,R1,#0`	424a ✓	r2 = 0 - r1 ✓
1.08	`EORS R2,R0`	4042 ✓	r2 ^= r0 ✓
1.09	`ANDS R2,R3`	401a ✓	r2 += r3 ✗
1.10	`BICS R4,R1`	438c ✓	r4 &= ~(1 << r3) ✗
1.11	`MVNS R2,R1`	43ca ✓	r2 = ~r1 ✓
1.12	`MOVS R3,R2`	0013 ✓	r3 = r2 ✓
1.13	`UXTB R1,R4`	b2e1 ✓	r1 = (r4 << 24) ✗
1.14	`MULS R3,R2`	4353 ✓	r3 *= r2 ✓
1.15	`NEGS R0,R1`	4248 ✓	r0 = 0 - r1 ✓
1.16	`LDR R2,[R5]`	682a ✓	r2 = &r5 ✗
1.17	`LDR R2,[R5,#0]`	682a ✓	r2 = r5[0] ✓
1.18	`STR R2,[R5,#4]`	612a ✗	r5[4] = r2 ✗
	`STR R2,[R5,#16]`		

1.19		642a ✗	r5[16] = r2 ✗
1.20	STR R2,[R5,#20]	652a ✗	r5[20] = r2 ✗
1.21	STR R2,[R5,#24]	662a ✗	r5[24] = r2 ✗
1.22	LDRB R1,[R7,#3]	78f9 ✓	r1 = r7[3] << 24 ✗
1.23	STRH R2,[R6,#2]	80b2 ✗	r6[2/2] = r2 & 0x0000ffff ✓
1.24	STRB R3,[R7,#2]	70bb ✓	r7[2] = r3 << 24 ✗
1.25	LDR R4,[R5,R0]	a82c ✗	r4 = r5[r0] ✗
1.26	LDRSH R0,[R6,R1]	ae70 ✗	r0 = r6[r1/2] ✓
1.27	LDRSB R1,[R7,R2]	a6b9 ✗	r0 = r6[r1] << 24 ✗
1.28	LDRH R2,[R6,R3]	5af2 ✓	r2 = r6[r3/2] & 0x0000ffff ✓
1.29	STR R3,[R5,R4]	512b ✓	r5[r4] = r3 ✗
1.30	STRH R1,[R6,R0]	5231 ✓	r6[r0/2] = r1 & 0x0000ffff ✓

---

(2) Decoding Instructions [20 points]

For each of the 16-bit hexadecimal numbers below, decode it to the corresponding ARM Cortex-M0 "Thumb2" instruction. Remember that you can look at page 84 to decide what to do with the most significant 6 bits, and then look at the tables on the pages you are referred to.

For instance, the hexadecimal value 0x0006 has the binary representation 0000000000000110. The top six bits are 000000, which matches the pattern 00xxxx, so it is a *Shift (immediate), add, subtract, move, and compare* instruction described on page A5-85.

Table A5.2.1 on page 85 shows a new opcode field which consists of bits 13-9. For this instruction, these are also 00000, which matches the pattern 000xx indicating that it is a Logical Shift Left "LSL (immedate)" instruction described on page A6-150.

Section A6.7.35 on page A6-150 lets us interpret the LDL instruction as having an imm5 field of 00000, an Rm field of 000, and an Rd field of 110. This represents an instruction of "LSLS R6,R0,#0". We know from lecture that this is also known as a "MOVS R6,R0" instruction.

You can, of course, use System Workbench or the simulator to check your work or do it entirely for you by writing the words into memory with assembler directives like this:

```
.global main:
main:
    .hword 0x0006
```

You should feel free to do so, but **you should be prepared to do the decoding all on your own** simply by looking at the *ARMv6-M Architecture Reference Manual*.

Question	Encoding	Instruction
example	0x0006	MOVS R6,R0
2.01	0x4345	MULS r5, r0 ✓
2.02	0x4053	EORS r3, r2 ✓
2.03	0x502b	STR r3, [r5, r0] ✓
2.04	0x7815	LDRB r5, [r2, #0] ✓
2.05	0x5917	LDR r7, [r2, r4] ✓
2.06	0x18b4	adds r4, r6, r2 ✓
2.07	0x1e93	subs r3, r2, #2 ✓
2.08	0x212d	movs r1, #0x2d ✓
2.09	0x46ea	movs r2, SP ✗
2.10	0x7a08	ldrb r0, [r1, #0x4] ✗
2.11	0x4323	orrs r3, r4 ✓
2.12	0x115a	asrs r2, r3, #0x5 ✓
2.13	0xb2da	UXTB r2, r3 ✓
2.14	0x4018	ands r0, r3 ✓
2.15	0x30eb	adds r0, #0xeb ✓
2.16	0x41d9	rors r1, r3 ✓
2.17	0x1d02	adds r2, r0, #0x4 ✓
2.18	0x6833	ldr r3, [r6, #0] ✓
2.19	0x60da	str r2, [r3, #3] ✗
2.20	0x0183	lsls r3, r0, #6 ✓

---

(3) Flags and Conditional Branches [15 points]

It can be difficult to understand the effects that instructions have on the Application Program Status Register (APSR) flags: NZCV. Luckily, we can take a higher-level view of the system by looking at the relationship between instructions and the effect that we expect them to have on subsequent conditional branches.

Consider the following sequence of instructions:

```
.global main
main:   MOVS r1, #5
        MOVS r0, #0
        BEQ label       // is this branch taken?
        ADDS r1, r2, r3
        BKPT #1 // not taken
label:  BKPT #2 // taken
```

The BEQ instruction looks at the APSR's Z flag. If the Z flag was set by the most recent flag-setting operation, BEQ will cause the Program Counter (PC) to be set to the address of **`label`**. If the Z flag is false (not set), then execution will continue with the ADDS instruction and anything else that follows the BEQ.

We know that the first instruction, **`MOVS R1,#5`**, modifies the flags because it has an 'S' suffix in its mnemonic. This instruction **is not** moving a zero into R1, so it **clears** the zero flag of the APSR. The second instruction also modifies the flags, and this time it **is** a zero, so it **sets** the Z flag to true. Only the most recent operation that updated the Z flag matters; the first MOVS instruction is now irrelevant. Therefore, this code sequence will cause the BEQ instruction to jump to **`label`**. In this case, we say that this branch was "taken."

If, however, we were to reverse the order of the two MOVS instructions, then the BEQ would continue with the ADDS instruction and anything else that follows it. In that case, we would say that the branch was "not taken."

For the following instruction sequences, look at the Operation section of the instruction descriptions in section A6.7, and in Table A6-1 "Condition Codes" in section A6.3, of the *ARMv6-M Architecture Reference Manual*, and decide if the single conditional branch in the sequence is "**taken**" or "**not taken**." There will be some questions where there is not enough information to know whether the branch is taken or not. These are situations where you don't know the initial flag state and the instructions before the branch either do not modify the flags that the branch depends on or do not modify the flags at all. In these cases, write "**unknowable**." If you want to try these sequences on the STM32 or on the simulator, feel free to do so, but **you should be able to decide what happens only by looking at the instructions' Operation description.**

The default values of the NZCV flags are 0000. You might want to precede each of the code segments with instructions that set them to be non-zero. For instance, you can set the Z, C, and V flags using the following two instructions:

```
LDR R0,=0x80000000
ADDS R0,R0,R0
```

By adding those instructions to the beginning of a code segment, you can initialize the flags to _ZCV.

If you want to clear all of the flags, try:

```
MOVS R0,#0
ADDS R0,#1
```

If you want to set only the N flag try, instead:

```
MOVS R0,#0
SUBS R0,#1
```

Examples:

Instruction Sequence	*Branch Outcome*
`MOVS r0, #0` `BEQ label`	**taken** (because we know MOVS set the Z flag)
`BEQ label`	**unknowable** (because we don't know what the Zero flag is)
	unknowable

`MOVS r0, #0` `BCC label`	(because MOVS does not modify the Carry flag, and we don't know what happened before the MOVS)
`MOVS r0, #0` `BNE label`	**not taken** (because MOVS set the Z flag)

3.01	`B label`	◉ taken ○ not taken ○ unknowable	✓
3.02	`MOVS R0,#2` `BLE label`	○ taken ◉ not taken ○ unknowable	✗
3.03	`MOVS R0,#6` `BCC label`	◉ taken ○ not taken ○ unknowable	✗
3.04	`MOVS R0,#0` `BNE label`	○ taken ◉ not taken ○ unknowable	✓

3.05	`MOVS R0,#19`
	`SUBS R0,#17`
	`BEQ label`
		○ taken ◉ not taken ○ unknowable	✓

3.06	`MOVS R0,#1` `BVC label`	◉ taken ○ not taken ○ unknowable	✗

3.07	`MOVS R0,#2`
	`ADDS R0,#4`
	`BVS label`
		○ taken ◉ not taken ○ unknowable	✓

3.08	`LDR R0,=0x918dc932`
	`CMP R0,#65`
	`BLE label`
		◉ taken ○ not taken ○ unknowable	✓

3.09	`MOVS R0,#9`
	`SUBS R0,#13`
	`CMP R0,#4`
	`BLS label`
		○ taken ◉ not taken ○ unknowable	✓

3.10	`MOVS R0,#6`
	`SUBS R0,#10`
	`CMP R0,#4`
	`BLT label // Note: not the same as BLS`
		◉ taken ○ not taken ○ unknowable	✓

3.11	`EORS R0,R0`
	`ADDS R1,R0,#6`
	`SUBS R0,#3`
	`CMP  R1,R0`
	`BHI label`
		○ taken ◉ not taken ○ unknowable	✓

3.12	`MOVS R0,#9` `SUBS R0,#35`	○ taken ◉ not taken ○ unknowable	✓

```
BPL label
```

3.13
```
EORS R0,R0
ADDS R1,R0,#3
SUBS R0,#7
CMP  R1,R0
BGT label
```
taken not taken unknowable 

3.14
```
MOVS R0,#255    // i.e., r0 = 0x000000ff
MOVS R1,#8
RORS R0,R1      // i.e., r0 = 0xff000000
LSLS R0,R0,#8   // i.e., r0 = 0x00000000
CMP R0,#2
BCC label
```
taken not taken unknowable 

3.15
```
LDR  R0,=0xffffffff
MOVS R1,R0
ANDS R0,R1
BVC label
```
taken not taken unknowable 

---

(4) You are a C compiler.

For each of the following C code stanzas, translate it into ARM Cortex-M0 instructions. Append a BKPT instruction at the end of your translation to pause the simulator or debugger at the end of the code. **If you edit your code in the simulator, remember to copy it back into the submission text boxes.**

Example:

C Code	*Example ARM Cortex-M0 translation*

`int x = 7;`
`int y = 6;`
`int z = 1;`
 	
`...`
 	
`x = z;`
`do {`
`  x = x + y;`
`  y = y - z;`
`} while (y != 0);`
	`.cpu cortex-m0`
	`.thumb`
	`.syntax unified`
	`.fpu softvfp`
	
	`.data`
	`x: .word 7`
	`y: .word 6`
	`z: .word 1`
	
	`.text`
	`.global main`
	`main:`
	`  // let r0 be the address of the start of the variables`
	`  ldr r0, =x  // load the address of the x variable`
	
	`  // x = z;`
	`  ldr  r1, [r0, #8]  // load from z`
	`  str  r1, [r0]  // store to x`
	
	`label0: // do`
	
	`  // x = x + y`
	`  ldr  r1, [r0]  // load x`
	`  ldr  r2, [r0, #4]  // load y`
	`  adds r1, r1, r2  // x = x + y`
	`  str  r1, [r0]  // store x`

```
// y = y - z
ldr r1, [r0, #4]   // load y
ldr r2, [r0, #8]   // load z
subs r1, r1, r2    // y = y - z
str r1, [r0, #4]   // store y

// while (y != 0);
ldr r1, [r0, #4]   // load y
// need a cmp because ldr does not set flags.
cmp r1, #0
bne label0

// when we're done with the loop, we fall through...

bkpt #0
```

(4.1) Simple data variable manipulation [15 points]

```
// Let's start with something not too hard.
// You can mostly use the example as a template.
// It's linear code.  There are no loops.
//
// You do have to write the code for the calculations rather
// than just setting d to the value 14 and call it a day.
// We can change the a,b,c input variables to test your code.
int a = 2;
int b = 4;
int c = 6;
int d = 8;

...

d = a + b;
d = d * a;
d = d - b;
d = d + c;
```

Enter your ARM Cortex-M0 code below:

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

.data
// Write your data statements below in the order the variables were defined.
a: .word 2
b: .word 4
c: .word 6
d: .word 8
// Write your data statements above in the order the variables were defined.

.text
.global main
main:
// Write your code here.
ldr r0, =a

ldr r1, [r0, #0]
ldr r2, [r0, #4]
adds r1, r2
str r1, [r0, #12]

ldr r1, [r0, #12]
ldr r2, [r0, #0]
muls r1, r2
str r1, [r0, #12]

ldr r1, [r0, #12]
ldr r2, [r0, #4]
subs r1, r2
str r1, [r0, #12]

ldr r1, [r0, #12]
ldr r2, [r0, #8]
adds r1, r2
str r1, [r0, #12]
// Write your code above.
bkpt // Leave this breakpoint here.
```

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

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

(4.2) A Simple Loop [15 points]

```
// Now, you can try a simple loop.
// This is the sum of every other value.
int sum = 0;
int x = 0; // an index variable
int begin = 7;
int end = 62;

...

x = begin;
```

```
while (x <= end) {
    sum = sum + x;
    x = x + 2;
}
```

Enter your ARM Cortex-M0 code below:

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

.data
// Write your data statements below in the order the variables were defined.
sum:   .word 0
x:     .word 0
begin: .word 7
end:   .word 62
// Write your data statements above in the order the variables were defined.

.text
.global main
main:
// Write your code here.
ldr r0, =sum
ldr r1, [r0, #8]
str r1, [r0, #4]

control:
ldr r1, [r0, #4]
ldr r2, [r0, #12]
cmp r1, r2
BLE loop
B exit

loop:
ldr r1, [r0, #0]
ldr r2, [r0, #4]
adds r1, r2
str r1, [r0, #0]

ldr r1, [r0, #4]
movs r2, #2
adds r1, r2
str r1, [r0, #4]
b control

exit:

// Write your code above.
bkpt // Leave this breakpoint here.
```

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

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

(4.3) A For Loop [15 points]

Note: a couple of values changed since the first time this was published. Now n=24 and the condition of the first 'for' loop is n<28

```
// Try two "for" loops.
// The first one sets the elements of array[] to the Fibonacci sequence.
// The second one finds the sum of the odd elements.
int sum = 0;
int x = 0;
int n = 24;
int array[28];

...

array[0] = 0; // Initialize the first two array elements.
array[1] = 1;

for(x=2; x<28; x++)
    array[x] = array[x-1] + array[x-2];

for (x=0; x <= n; x = x + 1) {
    if ((array[x] & 1) != 0) {
        sum = sum + array[x];
    }
}
```

Enter your ARM Cortex-M0 code below:

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

.data
// Write your data statements below in the order the variables were defined.
sum:    .word 0
x:      .word 0
n:      .word 24
.align 4
array:  .word 28
// Write your data statements above in the order the variables were defined.

.text
.global main
main:
// Write your code here.
    ldr r0, =array
    movs r1, #0
    movs r2, #1
    str r1, [r0, #0]
    str r2, [r0, #4]


for1:
    ldr r0, =x
    movs r1, #2
    str r1, [r0]

check1:
    ldr r0, =x
    ldr r1, [r0]
    movs r2, #28
    cmp r1, r2
    bge endfor1
```

```
    bgt endfor1

body1:
    ldr r0, =array
    ldr r1, =x
    ldr r2, [r1]  // x
    movs r3, r2
    subs r3, #1     // x - 1
    movs r6, #4
    muls r3, r6
    ldr r4, [r0, r3]
    movs r3, r2
    subs r3, #2     // x - 2
    muls r3, r6
    ldr r5, [r0, r3]

    adds r4, r5

    muls r2, r6
    str r4, [r0, r2]

next1:
    ldr r0, =x
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
    b check1
endfor1:

for2:
    ldr r0, =x
    movs r1, #0
    str r1, [r0]

check2:
    ldr r0, =x
    ldr r1, =n
    ldr r2, [r0]
    ldr r3, [r1]
    cmp r2, r3

    bgt endfor2
body2:
if1:
    ldr r0, =array
    ldr r1, =x
    ldr r2, [r1]        // x
    movs r3, #4
    muls r2, r3         // x * 4

    ldr r4, [r0, r2]    // array[x]

    movs r3, #1
    ands r4, r3

    MOVS r5, #0
    cmp r4, r5
    beq endif1
then1:
    ldr r0, =array
    ldr r1, =x
    ldr r2, [r1]        // x

    movs r3, #4
    muls r2, r3         // x * 4
```

```
    ldr r4, [r0, r2]    // array[x]

    ldr r5, =sum
    ldr r6, [r5]

    adds r6, r4
    str r6, [r5]
endif1:
next2:
    ldr r0, =x
    ldr r1, [r0]
    adds r1, #1
    str r1, [r0]
    b check2
endfor2:
// Write your code above.
bkpt // Leave this breakpoint here.
```

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

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