Hw2

Wednesday, January 26, 2022 2:00 PM

Hw2

Homework Assignment 2

Due: Fri, 28 Jan 2022 17:30:00 (approximately 2 days from the time this page was loaded)
[150 points possible] [150 penalties possible] [0 penalties graded so far]
Weight = 1.00

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 ... -150 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.

Tzu Yu Chen

Save

(1) Instruction encodings [60 points]

For each ARM Cortex-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. This work will be graded by computer.

A few hints:

When thinking about the difference between LDRB and LDRBS, remember that LDRBS will turn a signed 8-bit integer into a signed 32-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. Ask 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	negs r3, r1	424B	r3 = 0-r1
1.02	muls r1, r3	4359	r1 = r1*r3
1.03	negs r3, r2	4253	r3 = 0-r2
1.04	bics r2, r4	43A2	r2 = r2 & ~r4
1.05	eors r2, r3	405A	r2 = r2 ^ r3
1.06	mvns r2, r4	43e2	r2 = ~r4
1.07	adds r4, r0, #5	1D44	r4 = r0+5
1.08	uxth r2, r3	B29A	r2 = r3 & 0x0000ffff
1.09	adds r1, #8	3108	r1 = r1+8
1.10	ands r3, r1	400B	r3 = r3 & r1
1.11	asrs r3, r2, #4	1113	r3 = r2>>4
1.12	movs r2, #25	2219	r2 = 25
1.13	lsls r2, r1, #3	00CA	r2 = r1<<3
1.14	subs r4, #17	3C11	r4 = r4-17
1.15	movs r4, r3	001C	r4 = r3
1.16	ldr r1, [r5]	6829	r1 = r5[0]
1.17	ldr r1, [r5, #0]	6829	r1 = r5[0]
1.18	ldr r1, [r5, #12]	68e9	r1 = r5[3]
1.19	str r1, [r5, #16]	6129	r5[4] = r1
1.20	str r1, [r5, #32]	6229	r5[8] = r1
1.21	str r1, [r5, #60]	63e9	r5[15] = r1
1.22	strh r1, [r6, r0]	5231	r6[r0/2] = r1
1.23	ldr r3, [r5, r2]	58AB	r3 = r5[r2/4]
1.24	ldrsb r1, [r7, r0]	5639	r1 = r7[r0] & 0x000000ff
1.25	strh r3, [r6, #34]	8473	r3 > r6[17]
1.26	ldrh r2, [r6, r2]	5AB2	r2 = r6[r2/2] & 0x0000ff
1.27	ldrb r2, [r7, #5]	797A	r2 = r7[5] & 0x000000ff
1.28	str r3, [r5, r1]	506B	r5[r1/4] = r3
1.29	ldrsh r2, [r6, r1]	5e72	r2 = r6[r1/2] & 0x0000ffff
1.30	strb r1, [r7, #8]	7239	r7[8] = r1

Save

(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 (immediate)" 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	0x7c4b	0111 1100 0100 1011 ⇒ LDRB r3,[r1,#17]
2.02	0x00ea	0000 0000 1110 1010 ⇒ LSLS r2,r5,#3
2.03	0x8293	1000 0010 1001 0011 ⇒ STRH r3,[r2,#20]
2.04	0x1277	0001 0010 0111 0111 ⇒ ASRS r7,r6,#9
2.05	0x3383	0011 0011 1000 0011 ⇒ ADDS r6,#131
2.06	0x1ce5	0001 1100 1110 0101 ⇒ ADDS r5,r4,#3
2.07	0x628a	0110 0010 1000 1010 ⇒ STR r2,[r1,#40]
2.08	0x245b	0010 0100 0101 1011 ⇒ movs r4,#91
2.09	0x6c48	0110 1100 0100 1000 ⇒ LDR r0,[r1,#68]
2.10	0x1f53	0001 1111 0101 0011 ⇒ SUBS r3,r2,#5
2.11	0x4061	0100 0000 0110 0001 ⇒ EORS r1,r4
2.12	0x434f	0100 0011 0100 1111 ⇒ MULS r7,r1
2.13	0x5959	0101 1001 0101 1001 ⇒ LDR r1,[r3,r5]
2.14	0x51ac	0101 0001 1010 1100 ⇒ STR r4,[r5,r6]
2.15	0x4323	0100 0011 0010 0011 ⇒ ORRS r3,r4
2.16	0x1903	0001 1001 0000 0011 ⇒ ADDS r3,r0,r4
2.17	0xb2da	1011 0010 1101 1010 ⇒ UXTB r2,r3
2.18	0x41ea	0100 0001 1110 1010 ⇒ RORS r2,r5
2.19	0x46f3	0100 0110 1111 0011 ⇒ mov r11,r14
2.20	0x4014	0100 0000 0001 0100 ⇒ ANDS r4,r2

Save

(3) Flags and Conditional Branches [60 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 // If we stop here, it means the branch was not taken
label:  bkpt #2 // If we stop here, it means the branch was 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 moving a **non-zero** value 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.**

In the simulator, the values of the NZCV flags are 0000 at startup. For the code segments below, you should not assume you know what the initial flag values are. You should try setting the flags to different values to see what happens. For instance, 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
```

It is better if you assume all of the initial flag values are unknown. Manually trace through the instructions, one at a time, and identify the changes to the flags as you go. Then consider which flags a particular branch examines. If there are cases that cannot be resolved due to unknown flag values, then you know the branch outcome is "Unknowable".

Examples:

Instruction Sequence	*Flags (NZCV)*	*Branch Outcome*
NZCV ???? movs r0, #0 — 01?? beq label — 01??	01??	**taken** (because we know MOVS set the Z flag)
NZCV ???? beq label — ????	????	**unknowable** (because we don't know what the Zero flag is)
NZCV ???? movs r0, #0 — 01?? bcc label — 01??	01??	**unknowable** (because MOVS does not modify the Carry flag, and we don't know what happened before the MOVS)
NZCV ???? movs r0, #0 — 01?? adds r0, #5 — 0000 subs r0, #2 — 0010 subs r0, #3 — 0110 bne label — 0110	0110	**not taken** (because the last SUBS set the Z flag)

For each of the code segments below, specify what the flag values are at the time the particular branch instruction occurs. Specify them in the order NZCV. Based on that value, say whether the branch is taken, not taken, or unknowable.

Question	Code	NZCV	Outcome (○ taken ○ not taken ○ unknowable)
3.01	`b label`	????	taken
3.02	`movs r0,#0` `movs r0,#5` `beq label`	00??	not taken
3.03	`movs r0,#0xff` `bmi label`	00??	not taken
3.04	`movs r0,#52` `eors r0,r0` `beq label`	01??	taken
3.05	`movs r0,#0x23` `blo label`	00??	not taken
3.06	`movs r0,#5` `bvc label`	00??	not taken
3.07	`movs r0,#29` `eors r0,r0` `bmi label`	01??	not taken
3.08	`movs r0,#12` `subs r0,#18` `beq label`	1000	not taken
3.09	`movs r1,#3` `subs r1,#2` `movs r0,#1` `bhi label`	00?0	taken
3.10	`movs r1,#1` `movs r0,#1` `bhi label`	00??	unknowable
3.11	`eors r0,r0` `movs r0,#5` `bls label`	00??	unknowable
3.12	`movs r0,#1` `bge label`	00??	unknowable
3.13	`movs r0,#1` `adds r1,r0,#1` `bge label`	0000	taken
3.14	`movs r0,#0x45` `eors r0,r0` `bgt label`	01??	not taken
3.15	`eors r1,r1` `adds r1,#1` `movs r0,#1` `bgt label`	0000	taken
3.16	`movs r1,#1` `subs r1,#3` `bgt label`	1000	not taken
3.17	`movs r0,#23` `orrs r0,r0` `bvc label`	1000	not taken
3.18	`movs r0,#9` `adds r0,#8` `bvc label`	0000	taken
3.19	`movs r0,#43` `subs r0,#14` `blo label`	0010	not taken
3.20	`eors r0,r0` `ble label`	01??	taken
3.21	`movs r0,#73` `ands r0,r0` `ble label`	00??	unknowable
3.22	`movs r0,#73` `adds r0,#0` `ble label`	0000	not taken
3.23	`ldr r0,=0xffffff42` `cmp r0,#0x43` `ble label`	1010	taken
3.24	`ldr r0,=0xffffff42` `cmp r0,#0x43` `bls label`	1010	not taken
3.25	`movs r0,#5` `subs r0,#7` `cmp r0,#2` `blt label // Note: not the same as BLS`	1010	taken
3.26	`movs r0,#5` `subs r0,#7` `cmp r0,#2` `bls label // Note: not the same as BLT`	1010	not taken
3.27	`eors r0,r0` `adds r3,r0,#2` `subs r0,#2` `cmp r0,r3` `bhi label`	1010	not taken
3.28	`movs r0,#18` `subs r0,#23` `bpl label`	1000	not taken
3.29	`movs r0,#7` `adds r1,r0,#4` `subs r0,#5` `cmp r0,r1` `bgt label`	1000	taken
3.30	`movs r0,#5 // i.e., r0 = 0x00000005` `movs r1,#1` `rors r0,r1 // i.e., r0 = 0x80000002` `lsrs r0,r0,#1 // i.e., r0 = 0x40000001` `tst r0,r1` `bcs label`	000?	not taken

Save

Did this section seem meaningless and you're just glad it's over? It's probably worthwhile going through it one more time to try to understand it better. Understanding flag outcome is one of the most important concepts in assembly language programming. The better you understand it, the fewer mistakes you will make when writing programs.

(4) Selecting branch instructions [10 points]

Now that you perfectly understand flag generation, you are prepared to use this knowledge to select the right instruction sequences to implement C *control flow* statements. For instance, let's say you need to implement the following C expression in assembly language:

```
int item;
...
if (item == 6) {
    ...
}
```

You would write an instruction to load the 32-bit value of the "item" variable (which is a label, which is the same thing as an address), and then check it to see if it is six. For instance:

```
    ldr r0,=item
    ldr r0,[r0]
    cmp r0,#6
    bne endif1
    ...
endif1: ...
```

Remember that you should always pick the opposite kind of branch for an 'if' statement, so that it takes the branch to the 'endif' when the test fails.

For the tests below, select an appropriate branch instruction to implement the 'if' statement. Pay careful attention to whether variables are *signed* or *unsigned*. (An 'int' in C is signed by default.)

Question	C code	Assembly
4.01	`int abc;` `...` `if (abc < 0) {` `...` `}`	`ldr r0,=abc` `ldr r0,[r0]` `cmp r0,#0` [BGE] `...` `endif1:`
4.02	`int abc;` `...` `if (abc >= 5) {` `...` `}`	`ldr r0,=abc` `ldr r0,[r0]` `cmp r0,#5` [BLT] `...` `endif1:`
4.03	`int abc;` `int xyz;` `...` `if (abc != xyz) {` `...` `}`	`ldr r0,=abc` `ldr r0,[r0]` `ldr r1,=xyz` `ldr r1,[r1]` `cmp r0,r1` [BEQ] `...` `endif1:`
4.04	`int abc;` `int xyz;` `...` `if (abc <= xyz) {` `...` `}`	`ldr r0,=abc` `ldr r0,[r0]` `ldr r1,=xyz` `ldr r1,[r1]` `cmp r0,r1` [BGT] `...` `endif1:`
4.05	`unsigned int val; // NOTE: unsigned` `...` `if (value > 6) {` `...` `}`	`ldr r0,=value` `ldr r0,[r0]` `cmp r0,#6` [BLS] `...` `endif1:`
4.06	`unsigned int val; // NOTE: unsigned` `...` `// Note that I use 8 here...` `if (value > 8) {` `...` `}`	`ldr r0,=value` `ldr r0,[r0]` `cmp r0,#9 // ... but I use a 9 here` [BLS] `...` `endif1:`
4.07	`int fgh;` `...` `if (6 >= fgh) {` `...` `}`	`ldr r0,=fgh` `ldr r0,[r0]` `cmp r0,#6` [BGT] `...` `endif1:`
4.08	`unsigned int jkl;` `...` `if (6 < jkl) {` `...` `}`	`ldr r0,=jkl` `ldr r0,[r0]` `cmp r0,#6` [BLS] `...` `endif1:`
4.09	`unsigned int topbit;` `...` `// How can you tell if the leftmost bit is set?` `if ( topbit & 0x80000000 ) {` `...` `}`	`ldr r0,=topbit` `ldr r0,[r0]` `lsls r0,r0,#1` [BCC] `...` `endif1:`
4.10	`unsigned int bits;` `...` `// Is bit 6 set?` `if ( (bits & (1<<6)) != 0) {` `...` `}`	`ldr r0,=bits` `ldr r0,[r0]` `movs r1,#64` `tst r0,r1` [BEQ] `...` `endif1:`

Save