Hw1

Tuesday, January 18, 2022 7:45 PM

Hw1

Homework 1

Due: Fri, 21 Jan 2022 17:30:00 (approximately 2 days from the time this page was loaded)
[20 points possible] [20 penalties possible] [0 penalties graded so far]
Weight = 1.00

The first part of this homework consists of some simple thought questions for which you can (mostly) look up the answers in the reference manuals or the lecture notes. It's best if you can use the reference manuals. The second part of the homework involves modifying an existing assembly language program to do what we're asking. You can use either the simulator or the actual microcontroller with System Workbench to do this exercise. We will evaluate your work with the simulator so be sure to adhere to the limitations imposed by its wimpy parser:

- If you write comments, use only "//".
- Type all instructions and registers with lowercase letters.

Academic Honesty Statement [0 ... -20 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

Question 1 [1 point]

For the form of assembly language we use for this class, if you want the assembler to create an instruction that adds the contents of two registers and stores the result in a third register like this:

`r6 = r5 + r0`

as well as set the flags, how would you write that instruction so that the assembler would produce the right thing? Are you sure? Did you try it? There's no partial credit for being close. The assembler program will never say "Oh, that's an error, but I'll just forgive you for it." It is important to type exactly the right thing.

adds R6,R5,R0

Save

adds r6, r5, r0
if no s updates the flag
then no update flags

Question 2 [1 point]

For the form of assembly language we use for this class, if you want the assember to create an instruction that adds the contents of two registers and stores the result in a third register like this:

`r1 = r1 + r2`

Tab Actions Menu

but **not** change any flags, how would you write that instruction so that the assembler would produce the right thing?

add r1,r2

Save

add
no s so does not change flag

Question 3 [1 point]

For either form of the add (immedate) instruction described in section A6.7.2 of the Architecture Reference Manual, what registers can be used for the source and destination? Express your answer as a comma-separated list of registers.

r0,r1,r2,r3,r4,r5,r6

Save

because only 3 bits so max # is 4+2+1=7

Question 4 [1 point]

Suppose you want to write an instruction that takes the value stored in register **r5**, add an immediate value to it and store it in **r2**. What is the largest immediate value you can use? Express your answer as a positive decimal number. (Hint: See A6.7.2. Should you use instruction encoding T1 or T2? How do you decide?) Be sure to check your answer. Try setting the immediate value to one larger than what you think the largest answer should be and make sure it produces an error.

7

Save

should use encoding T1 <imm3> ⇒ r2 = r5 + #7 ⇒ so max value for constant is 7

Question 5 [1 point]

List the flags that may be affected by the instruction

`adds r1,r6,r3`

Express your answer as an unpunctuated list of flags in the order they appear in the xPSR. (xPSR means any of the three forms of the Processor Status Register. The top four bits are the ALU flags.) Read the documentation carefully, and try examples in the simulator. If you cannot think of a way to change the value of a flag, maybe it cannot be changed. (Write "none" if the instruction never changes any flags.)

Z

Save

Question 6 [1 point]

List the flags that may be affected by the instruction

`mov r3,r4`

Express your answer as an unpunctuated list of flags in the order they appear in the xPSR. (Write "none" if the instruction never changes any flags.) Read the documentation carefully, and try examples in the simulator.

Z

Save

Question 7 [1 point]

List the flags that may be affected by the instruction

`movs r6,r5`

Express your answer as an unpunctuated list of flags in the order they appear in the xPSR. (Write "none" if the instruction never changes any flags.)

Z

Save

Question 8 [1 point]

Is the following a valid ARM Cortex-M0 instruction?

`mov r4,#191`   mov is only for registers

No

Save

Question 9 [1 point]

Is the following a valid ARM Cortex-M0 instruction?

`movs r3,#267`

No ⇒ # cant be > 255

Save

Question 10 [1 point]

For the conditional branch instruction, `B`, what 4-bit value in the "cond" field should be used to implement the "Signed less than or equal" branch comparison? For instance, the "branch if equal" instruction would have a "cond" field of 0000. Look at lecture 03 to understand how the conditional branch instruction works. Look at the instruction encoding in A6.7.10, and see the branch condition field encodings on pages A6-98 and A6-99. (Express your answer in four sequential digits that are one or zero.)

1101

Save

Signed less than or equal   cond = 1101
flags: Z == 1 or N != V

Table A6-1 Condition codes

cond	Mnemonic extension	Meaning	Condition flags
0000	EQ	Equal	Z == 1
0001	NE	Not equal	Z == 0
0010	CS [a]	Carry set	C == 1
0011	CC [b]	Carry clear	C == 0
0100	MI	Minus, negative	N == 1
0101	PL	Plus, positive or zero	N == 0
0110	VS	Overflow	V == 1
0111	VC	No overflow	V == 0
1000	HI	Unsigned higher	C == 1 and Z == 0
1001	LS	Unsigned lower or same	C == 0 or Z == 1
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Z == 1 or N != V
1110 [c]	None (AL) [d]	Always (unconditional)	Any

a. HS (unsigned higher or same) is a synonym for CS.
b. LO (unsigned lower) is a synonym for CC.
c. This value is never encoded in any ARMv6-M Thumb instruction.
d. AL is an optional mnemonic extension for always.

Question 11 [1 point]

For the branch described in the previous question, what three-letter instruction mnemonic would be used in an assembly language program to produce the instruction? For instance, if you wanted to "branch if equal", the mnemonic would be "BEQ".

BLE

Save

Question 12 [1 point]

What is the *stated* range, in bytes, for the permitted offset of a conditional branch? For instance:

```
start:  beq  dest
        adds r0,#1
        subs r1,#2
        ...
dest:   muls r0,r1
```

permitted offsets are even numbers in range -256 to 254 for encoding T1

How far apart, in bytes, can the "start" label be from the "dest" label? This may not be quite an accurate way to specify the problem. What we really want you to do is look up the *state* range. Express your answer as a range of negative to positive values like: "-abc to xyz" (Hint: See A6.7.10 instruction encoding T1. It states it right there.)

-256 to 254

Save

Question 13 [1 point]

What is the *stated* range, in bytes, for the permitted offset of an **unconditional** branch? For instance:

`b target`

Express your answer as a range of negative to positive values in a similar manner as the previous question. (Hint: See A6.7.10 instruction encoding T2.)

-2048 to 2046

Save

Question 14 [1 point]

Write the full instruction that *compares* values in the registers, r3 and r0, by subtracting them like (r3 - r0), and sets the flags according to the result of the instruction (and throws away the 32-bit result without updating any of the general-purpose registers). Did you try this? Are you certain you expressed it correctly?

CMP R3,R0

Save

Question 15 [1 point]

What single instruction (that involves an assembler trick described in lecture) can you use to set register r2 to the value 0xd9b3983e?

LDR R2,=0xd9b3983e

Save

Question 16 [1 point]

Write a single (logical) instruction that clears all bits in r2 that are set in r1 regardless of whether they were previously set in r2 or not. For instance, if the initial register values are

```
r2 = 0x00000000
r1 = 0x00008421
```

R2 = R2 &~R1

the resulting value in r2 after execution of the instruction should be 0. If, however, the initial register values are

```
r2 = 0xffffffff
r1 = 0x00008421
```

then the resulting value in r2 after execution of the instruction should be 0xffff7bde.

BICS r2,r1

Save

Question 17 [1 point]

Write a single (logical) instruction that sets the bits in r2 that are also set in r7 regardless of whether those bits were previously set in r2 or not. For instance, if the initial register values are

```
r2 = 0x44444444
r7 = 0x00008210
```

r2 = r2+r7 = 4444 C 654

the resulting value of r2 after execution of the instruction should be 0x4444C654. If, however, the initial register values were

```
r2 = 0x44444444
r7 = 0x00000040
```

then the resulting value of r2 after execution of the instruction should be 0x44444444.

Try writing code in the simulator to set up the initial register values and the instruction you choose. Make sure it works.

orrs R2,r7

Save

Question 18 [1 point]

Write a single (logical) instruction such that, given the following initial register values:

```
r2 = 0x12345678
r7 = 0xFFFFFFFF
```

the instruction would write the result 0xedcba987 to r2. And for the following initial register values:

```
r2 = 0x11111111
r7 = 0xFFFFFFFF
```

the instruction would write the result 0xeeeeeeee to r2. (Hint: Look at the values 0x12345678 and 0xedcba987 as binary values. How are they related?)

mvns r2,r7

Save

Program Modification [2 points]

Consider the code below that has many typographical errors:

```
.cpu cortext-m0
.thumb2
.syntax unify
.fpu softvfp

.globel main

main:
    movs   r0, #6
    move   r1, #20
    movv   r2, #0 // This is a loop counter
    mov    r3, #0 // This stores a sum.
loop:
    nop    // nop 1
    nop    // nop 2
    nop    // nop 3
    nop    // nop 4
    nop    // nop 5
done:
    bkpt   // Stop the debugger (breakpoint)
    b done
```

Modify this code, one step at a time. You can type it into the simulator and try it.

- First, fix the syntax errors! And don't laugh. These are the kind of errors you will make someday, so get used to finding and correcting them.
- Replace the first "nop" with an instruction that compares the values of registers R2 and R0. (i.e., it should set the flags as if it subtracted R0 from R2: (R2-R0))
- Replace the second "nop" with an instruction that redirects execution to the instruction following the "done:" label if and only if the result of the comparison in the previous instruction is greater than zero.
- Replace the third "nop" with an instruction that adds the value of register R1 to R3 and stores the result in R3.
- Replace the fourth "nop" with an instruction that increments the value of r2 by 1.
- Replace the fifth "nop" with an instruction that will send execution back to the instruction at the "loop:" label.

When the program reaches the

`bkpt`

instruction, the value of the R3 register should be 0x0000008c (decimal 140).

Important note: You should actually type in, run, and test this program. Then you should copy the whole thing (including the .cpu, .thumb, .syntax, and .fpu directives) from System Workbench or the simulator, and paste into the box below. There is a link below the text box that will invoke whatever you submit in the simulator. (Be sure to save it first.) That's what we will use to grade your work. If we can't assemble it, and see it run as specified, you don't get partial credit. It either works or it doesn't.

If you make updates in the simulator, remember to copy it back into the box below.

```
.cpu cortext-m0
.thumb2
.syntax unify
.fpu softvfp

.globel main

main:
    movs   r0, #6
    move   r1, #20
    movv   r2, #0 // This is a loop counter
    mov    r3, #0 // This stores a sum.
loop:
    nop    // nop 1
    nop    // nop 2
    nop    // nop 3
    nop    // nop 4
    nop    // nop 5
done:
    bkpt   // Stop the debugger (breakpoint)
    b done
```

Save

Click here to try it in the simulator

main.s   startup.s

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

.text
.global main

main:
    movs   r0, #6
    movs   r1, #20
    movs   r2, #0 // This is a loop counter
    movs   r3, #0 // This stores a sum.
loop:
    cmp r2, r0    // nop 1
    bgt done    // nop 2
    adds r3, r3, r1    // nop 3
    adds r2, #1    // nop 4
    b loop    // nop 5
done:
    bkpt   // Stop the debugger (breakpoint)
```

Register	Value
R0	0x00000006
R1	0x00000014
R2	0x00000007
R3	0x0000008c
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
SP	0x20008000
LR	0x08000005
PC	0x0800001a
	NZCV
APSR	0010