Brian McIlwain

EE3752 Lab 4 Section 4

Purpose

The purpose of the experiment was to provide practice loading and storing values from memory.

Problems

1) Convert part 1 from Lab 3 to take its inputs from memory and store its sums to memory instead of registers. Declare four words of input data in the same section as your code; they will form an array of four words starting at variable *gold*. You may use pre-indexed or post-indexed addressing. Store your final sums into memory in the read-write data section into two words starting at variable *purple*.

Points will be given for:

• Efficient use of registers

• Efficient use of instructions

Lab 3 part 1:

Add the high 16 bits from r1 and the low 16 bits from r2, place the word-size sum in r5; and (2) add the middle 16 bits from r3 and the middle 16 bits from r4, placing the word-size sum in r6.

2) Write a program that will load an input word from memory and count the number of half-words that are all 0’s, then store this count into memory.

3) For numbers that are 128-bit unsigned integers, write a program that will subtract number *yellow* from number *orange* that are stored in memory and store the difference back into memory at *red*.

Method

1) The addition to Lab 3 that I used can be summed up with the following 2 instructions: LDR and STR. I first used LDR r0, =gold to load the address of label ‘gold’ in r0. I then used LDR r1, [r0] to load the contents of memory pointer r0 into register r1. For storing, I did LDR r7, =purple to store the address of label purple into r7, then STR r5, [r7] to store the contents of r5 into memory at the address pointed to by r7. This was the basis of the modifications needed to complete part 1. For efficiency, I re-worked my solution from last lab to use a mere 3 registers, and depend on bit-shifting only instead of rotations and shifting.

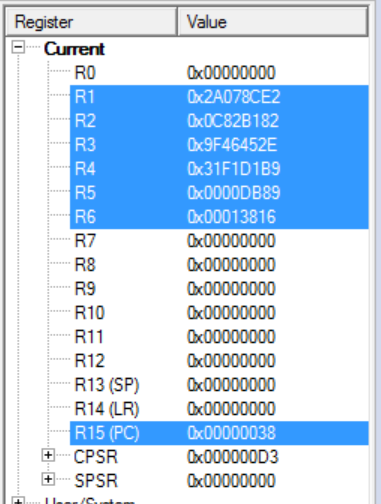
2) The crux of my approach to part 2 depends on two instructions: LDRH r1, [r0] and CMP r1, #0. I load a half word into a register, then add the zero flags. I then repeat for the second half word.

3) For part 3, I depended on SUBS, SBCS, and SBC instructions. SUBS was used to subtract the least significant bits, then SBCS for the middle, and finally SBC was used to subtract the MSBs. I also used LDR and STR to move the input and results in and out of memory. These instructions let me specify which part of the bit string I’m currently subtracting; beginning, middle, or end.

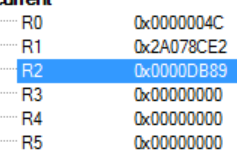
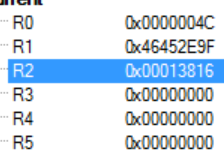
Results

1. The below screenshot shows the answer, with the given input, r5=0x0000DB89, and r6=0x00013816. However, I re-factored the code to compute the same answer using a mere 3 registers.

Old way:



NOW: (Answers are computed in r2, then stored, then the same register is re-used)

1. As we weren’t given any specific inputs, I’ll mention a few examples. If the input is 0x0000 0000, then the result would be 2, as there are 2 half word which are all zero. If given, say, 0x0000 0001, the answer would be 1, and if 0x0010 0101 you would get zero. Since this is so simple, I’ll assume the “solution” looked for is how I went about solving it, so I’ve included my code here.

start LDR r0, =input ; Get address of input

LDRH r1, [r0] ; Get first half word

CMP r1, #0

ADDEQ r3, r1, #1 ; Add zero flags

LDRH r1, [r0, #2] ; Get second half word

CMP r1, #0

ADDEQ r3, r1, #1 ; Add zero flags

LDR r0, =ans ; Store it back in memory

STR r3, [r0]

stop B stop

1. For part three, we were provided the following input:

orange DCD 0xE00700B2, 0x12003E11, 0x00456700, 0x000ABCDE

yellow DCD 0x320002B1, 0x08002177, 0x80111100, 0x00011000

The answer to orange – yellow is given below:

0xAE06FE01, 0x0A001C9A, 0x80345600, 0x7FF9ABDD