CS3339 Project 1

Description: In this project, you will create a simple ARM(Stumpyleg)v8 dissasembler. Your dissasembler will read a binary file containing an ARM program in machine code and generate the assembly code for the given ARM program. You will not have to implement exception/interrupt handling and we will not use all of the LEGv8 instructions. The instructions will follow simplifications as outlines in the zybook Ch2. You will use this code in subsequent projects so do a good job and test it competely.

Change Notices: As in real life I reserve the right to add to the project whenever I want to. In real like you would charge more. In this life you just get to spend more (time). Implication: You should make sure your code is constructed to be easy to add additional instructions.

Implementation: You must implement this program in Python 2.7

Details: Refer to theARM instruction set architecture in your book. I will be creating an achitecture document for the instructions we are using just to simplify lookup but its not ready yet. Part 1 - A summary of instructions is attached.

Memory Layout: Memory will be aligned on word. This means that you will start your program on mem location 96 and then increment by four bytes 96, 100, 104, 108. This will become obvious once you see an example output.

Instructions Covered:

You will be given an input file containing a sequence of 32 bit instruction words. Assume that the first instruction is at memory address 96. The final instruction in an instruction sequence is ALWAYS a “BREAK” instruction. Following the break instruction is a sequence of 32 bit 2’s compliment signed integers for the program data if there is program data. These continue until the end of file. The break instruction is a bit of an issue since ARM does not implement this exact instruction so I will describe what I want you to use for "BREAK" in a section below.

Your simulator/dissassembler must support the following ARM instructions:

ADD, ADDI, SUB, SUBI
LSL, LSR, AND, ORR, EOR
LDUR, STUR
CBZ, CBNZ
MOVZ, MOVK

B
NOP

Notes on instructions:

ADDI, SUBI - the immediate value will be treated as just that, a signed integer that needs to be added or subtracted.

SHIFTS - The shamt amount will be integer describing places to shift. The real ARM instruction is more complex.

CNBZ , CBZ - Although the way the actual assembly is written is with a tag, we are going to modify this to have the number of words offset from the current position of the PC. So if the branch is just to the next instruction, there will be a 1 in the appropriate CB instruction field.

BREAK - Use the following coding for BREAK:

1 11111 10110 11110 11111 11111 100111
OPCODE - 2038_{10}
MASK 0x1FFFFF = 2031591_{10}

NOP - 0x0 = 0000 0000 0000 0000 0000 0000 0000 0000

Execution:

Your program must accept command line arguments for execution. The following and only the following arguments must be supported in exactly the way I am showing it:

```
$ python team#_project1.py -i test1_bin.txt -o team#_out
```

Please show some thought and don't enter "team#"! If you have any questions, make sure you understand. I will not be happy if I have to mess around when my automation runs your program.

Your program will produce 1 output file named named team#_out_dis.txt, which contains the disassembled program code for the input ARM machine code.

Your program will be graded both with the sample input and output provided to you, and with input and output that is not provided to you. It is recommended you construct your own input programs for testing.

WARNING: I AM DEAD SERIOUS ABOUT THE FOLLOWING!!!!!

Output: The dissembled output file should contain one line per word in the input file.   It should be separated into 4 columns, each separated by tab character. The columns contain the following information:

1) The binary representation of the instruction word. If the word is an instruction (as opposed to memory data after the BREAK instruction), the instruction should be split into six groups of digits: 8 bits which is the smallest opcode size, three more bits covering the largest opcode, four groups of 5 bits, and a final group of 6 bits.
2) The address of the memory location (in decimal)
3) If it is an instruction, print the operation, followed by a tab character, then print each argument separated by a comma and a space (", ").

Instructions and arguments should be in capital letters. All integer values should be in decimal. Immediate values should be preceded by a # sign. Be careful and consider which instructions take signed values and which take unsigned values. Be sure to use the correct format depending on the context. Output exactness is a goal of the project just as it would be if this was for a real customer!

You output will be graded with the DIFF command. Test your output against the provided sample outputs! Any differences reported by the DIFF command are assumed to be incorrect output! That is why the formatting is so prescribed!!!!!!!!! If you don't follow it you will get hammered on the grading. :(

What to turn in: Your source file. A README.txt file indicating how to run your program on LINUX!! Also, it should contain the names and txstate netIDs of your group members. You can develop on Windows, but your program will be graded on Linux on my mac powerbook. The executable must be named "team#_project1.py". (# is replaced by your team number) I also want a test input file you ran - preferably the one I gave you - and the output for the test file. This is case your code will not run MY final test file.

You should turn in one zipped folder with all the files in it with the folder name "team#_project1". (# is replaced by your team number) I want to be able to download the zipped folder from TRACS in my testing program, unzip it, and run your code.

Misc:
Think about all data structures before you start. If you want me to review your plan, come by my office hours. I will tell you what I would do and its your funeral if you do something different.

The basic idea for the disassembler is a for loop to march through all of the instructions.

We will talk about unit testing. I cannot emphasize enough about developing your own set of tests since the ones I give you are not complete!!

The bulk of your code will be contained in a python class and be executed by a function called *run.*   I will describe the layout of the code in more detail in class so you should be there.

Instruction Summary for Projects

Instruction	OPCODE	OP Size (base 10)	11 bit OPCODE range		Instruction Format
			Start	End	
B	000101	6	160	191	B
AND	10001010000	11	1104		R
ADD	10001011000	11	1112		R
ADDI	1001000100	10	1160	1161	I
ORR	10101010000	11	1360		R
CBZ	10110100	8	1440	1447	CB
CBNZ	10110101	8	1448	1455	CB
SUB	11001011000	11	1624		R
SUBI	1101000100	10	1672	1673	I
MOVZ	110100101	9	1684	1687	IM
MOVK	111100101	9	1940	1943	IM
LSR	11010011010	11	1690		R
LSL	11010011011	11	1691		R
STUR	11111000000	11	1984		D
LDUR	11111000010	11	1986		D
EOR	11101010000	11	1616		R
NOP	00000000000	11	0		R

BREAK 1 11111 10110 11110 11111 11111 100111
OPCODE - 2038_{10}
MASK 0x1FFFFF = 2031591_{10}