1 Setup

Visit this link:

to copy the starter repository to your Github account. Then log into an ieng6 machine and clone it.

Next, when logged into ieng6 or the pi-cluster (either should work), run these two commands:

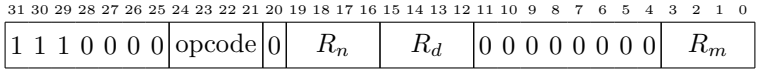
```
$ echo "alias pa1-runner='/home/linux/ieng6/cs30f/public/pa1-runner'" >> ~/.bash_profile
$ source ~/.bash_profile
```

This will set things up so you can directly run the `pa1-runner` command. It won't work unless you perform these steps.

2 ARM Introduction

In this assignment, you'll be writing some short programs directly in machine code. You won't quite write them directly as 1's and 0's—you'll actually type in hexadecimal characters—but you'll directly use the encoding of ARM instructions in binary. This section lays out the instructions you'll need to use for this assignment in detail.

For now, it will suffice to know a few basic formats for data manipulation instructions. Here is the format for manipulating data strictly between registers:



Generally, R_n and R_m are the operands of this operation; opcode selects which operator to apply to them, and the result is stored in R_d. Some math-related opcodes useful for this assignment are:

opcode	mnemonic	expression
0000	AND	$R_d \coloneqq R_n \,\&\, R_m$
0001	EOR	$R_d \coloneqq R_n \hat{} R_m$
0010	SUB	$R_d \coloneqq R_n - R_m$
0011	RSB	$R_d \coloneqq R_m - R_n$
0100	ADD	$R_d \coloneqq R_n + R_m$
1100	ORR	$R_d \coloneqq R_n \mid R_m$
1110	BIC	$R_d \coloneqq R_m \,\&\, \sim R_m$

Another form of data manipulation instructions lets us combine values in registers with constants:

31 30 29 28 27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4 3 2 1 0
1 1 1 0 0 0 1	opcode	0	R_n	R_d	0 0 0 0	Imm

The key difference is the `1` instead of a `0` in bit 25, which makes the instruction operate in immediate mode. Then, Imm is an binary value that is used in place of R_m above. So, for example, if we used `00001111` as bits 0-7 of of an addition instruction, it would add 15 to the value of the R_n register and store it in R_d.

There is another format of instructions which is used for multiply-like instructions:

31 30 29 28 27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 0 0 0 0 0	op	0	R_d	0 0 0 0	R_n	1 0 0 1	R_m

The multiply operation's opcode is `000`, so replacing *op* above with `000` indicates a multiply. You only need this version of multiply for this assignment, you can see the others in tabular form in appendix B of the Harris book if you're interested in seeing more.

3 Support Code and Example

We've provided support code so that you can simply write machine code instructions as hexadecimal, create a binary file from them, and then execute it with certain starting conditions. Let's consider the following problem:

> Assume that `r1` contains x and `r2` contains y. Write machine code instructions that result in the value $x + y$ being stored in `r0`.

Our task is to create a binary file containing the (in this case single) instruction that adds the values and stores the result in `r0`. We can use the specification of add above to get started; we want to take the pattern for data processing instructions, and pick the opcode for ADD (which is `0 1 0 0`) and the appropriate registers. Note that the registers are indicated by their unsigned binary representation in 4 bits, so `r0` is `0 0 0 0`, `r1` is `0 0 0 1`, and so on. Here's the first step

31 30 29 28 27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4	3 2 1 0
1 1 1 0 0 0 0	opcode	0	R_n	R_d	0 0 0 0 0 0 0 0	R_m

31 30 29 28 27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4	3 2 1 0
1 1 1 0 0 0 0	0 1 0 0	0	0 0 1 0	0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1

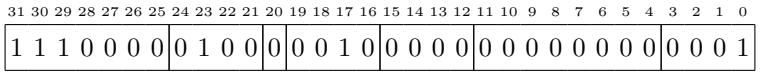
(Here, we picked that the n-labelled source register would hold `r2` (y), and the m-labelled source register would hold `r1` (x). Since addition commutes, we could have picked either order.)

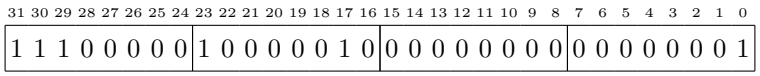
Now we need to take that instruction and turn it into a binary file. Note that a file containing *text* 1's and 0's is not a binary file [REF cmu binary file link]. We don't want a file containing a sequence of bytes representing ASCII; we want a file containing exactly the four bytes of the instruction above, in the right format and order.

One of the most straightforward ways to do this is with the `xxd` command. It is useful for a number of things, where one mode of operation is taking files containing hexadecimal text and converting them to the corresponding binary files. So we can take a file that contains purely hexadecimal ASCII text, and produce a binary file. We'll show an example of that command in a moment, but first, we need to think a little bit about endianness and ordering.

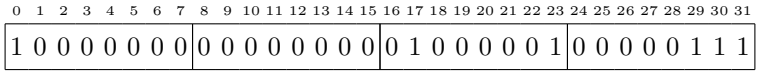
All of the instructions we've shown have had the 0th bit all the way to the right. This is a convention for writing out ARM machine code because it reads well left-to-right – first the opcode, then the arguments. However, the actual order of the bytes is the reverse of this. Further, the bytes themselves are interpreted in little-endian order. Let's do an example to process this conversion.

First, we re-organize the breaks to see the instruction in terms of byte chunks, which will help us think through the hexadecimal encoding.

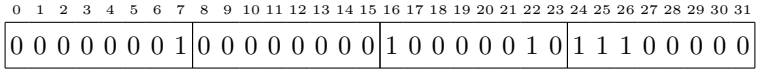




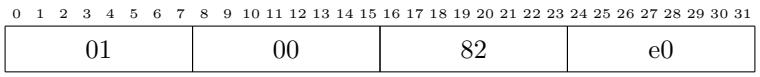
Next we reverse the whole thing to help us see the correct left-to-right order of the whole instruction:



Next, we flip each individual byte, because the bytes are interpreted in little-endian order:



Finally, we convert to hexadecimal:



So the hexadecimal encoding of this instruction is `010082e0`. We can put that into a file, let's call it `add.txt`. The `xxd` command can be used to create a binary from it:

```
$ xxd -p -r add.txt add.bin
```

And then we can run the file using the provided runner, which lets us give arguments as command-line arguments, puts them into the designated registers, and runs our binary code:

```
cs30f@pi-cluster-013:~/pa1/$ pa1-runner add.bin 4 5
f(4, 5):
  r0 = 9
  r1 = 4
  r2 = 5
  r3 = 2126498556
  r4 = 0
  r5 = 0
  r6 = 69808
```

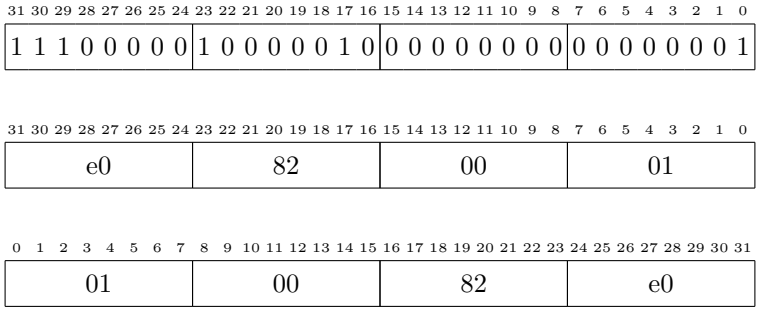
This command *must* be run on `pi-cluster`. Note that the values in `r3` to `r6` aren't set by the sample adding program, so you might see different values show up there. But `r0` has the sum, `r1` has the first input, and `r2` has the second input.

A few extra notes:

- We can use `xxd` to see our instruction in binary again:

  ```
  $ xxd -b add.bin
  00000000: 00000001 00000000 10000010 11100000                    ....
  ```

- An abbreviated process for getting to the appropriate hexadecimal from the conventional binary notation is to convert to hexadecimal, and then simply reverse the order of the four bytes, so the first byte becomes the last, the last the first, and the middle two are swapped:



4 Programs to Write

You will fill in the provided text files, each with a particular formula using the instructions above. You will hand in your plain-text files with instructions

in hexadecimal, using the process described above. In all cases, assume that initially x is stored in `r1`, y is stored in `r2`, and the result should appear in `r0`. You may use any registers from `r0` to `r6` for your computation.

1. In `sub2.txt`, compute $x - y - y$
2. In `submul.txt`, compute $(x - y) * y$
3. In `addsubmul.txt`, compute $(x - y) * (x + y)$
4. In `quad.txt`, compute $2x^2 + yx - 7$

If an operation would produce a number that is too large to represent in 32 bits, you can simply use the part that doesn't overflow (the default for most instructions). For example, the result of multiplying 2,000,000,000 by itself requires more than 32 bits, so if that's an input as x in `quad.txt`, it's fine if your program evaluates it to 0 (since the lower 32 bits of the answer will all be 0).

5 README

Along with your code, please also submit a README file which contains your answers to the following questions:

1. There are three mystery files that contain hexadecimal instructions in your checkout. Your job is to figure out and explain what these mystery binaries do (they are short). You might do so via testing, or by converting the hexadecimal instructions back to binary code, and checking for opcodes. You will probably want to use commands like `xxd -p -c 4 mystery1.bin` to do this. In your explanation, you should explain how you get your final answers. For instance, you can say:
   "the opcode of the first instruction is 0000, which represents a AND operation." or
   "R_d of the second instruction is 0001, which stores the result to register 1"
   The explanation should be as specific as possible.

   Also, you must provide at least three examples for each file with different values of x and y that you pass as arguments to run the program. After you figure out what operations are used in these files, you must explain the meanings of these operations. (Notice that the arguments you passed in and the answer are all in decimal values. It might be helpful to convert them to other bases in order to understand the meaning of these operations)
2. Starting from the instruction format in the textbook and this writeup, we have to do two apparent reversing operations: one on the whole instruction, and one on each byte. Please explain why we need these two reverses in your own words.

3. Consider the machine code corresponding to r0 = r1 + r2. If we wanted to change the instruction to use some other register instead of r1, how would the machine code change and why?

Please limit your answer to each question to no more than 100 words and present them as clearly as possible.

6 Handin and Fetching Files

Commit and push your four text files to the Github repository that was created for you by 11:59PM on Tuesday, October 10. You can push up to one day late for a 20% penalty. After you push, make sure to check on Github that the files are actually there; we will copy all of the repositories for grading a few minutes after midnight and grade precisely what is there.

Your handin should include:

- Four .txt files containing hex values (`addsubmul.txt`, `sub2.txt`, `submul.txt`, and `quad.txt`) that are instructions to calculate the formulae above.
- A single README.txt file that has answers to the open-ended questions above, as specified.