Assembly Language Programming

Prof. John McLeod

ECE9047/9407, Winter 2021

This lesson provides an introduction to assembly language, with specific emphasis to the code base for the ARM®Cortex-A9 processor. The basics of data movement, data manipulation, and conditional execution are provided. Future lessons will extend these basic concepts.

Low-Level Programming

Fans of high-level programming languages enjoy coding in syntax that is easily human-readable. Even if you are not familiar with Python, for example, you probably still have a pretty good idea what this code does:

```
x_list = [4*y for y in y_list if y > 2]
```

This kind of coding may be quicker and (possibly) less error prone than low-level programming, but it does generate a huge amount of overhead in terms of unnecessary usage of processor cycles and memory. Lower-level coding is necessary to write efficient programs for a memory- and processing speed-limited microcontroller. [1] C is lower level, and can be used to program a microcontroller, but it is still less-efficient than assembly language.

- The online simulator used in the first laboratory/homework assignment allows you to code in C rather than assembly.
- After the code is compiled, you can read the assembly code that your C program was converted to — and enjoy seeing how many redundant expressions and useless loops are used.
- Please note that I do *not* want you to code in C for the laboratory/homework assignment. Please use assembly language directly.

Part of the reason goes back to the discussion about **numbers** from last week. As an example, imagine that you need to store an array of one thousand arbitrary $64\,\mathrm{b}$-floating point numbers.

[1] I guess technically this isn't true, as some state-of-the-art compilers can speed up high level programs by a considerable amount. But these optimizing compilers are only "optimal" for the specific test-cases they were optimized for — matrix diagonalization being a good example — and are not always good substitutes for a well-thought out low-level program.

- The minimum amount of memory needed is $1000 \times 64\,\mathrm{b} = 8000\,\mathrm{B}$ of memory (just a bit less than $8\,\mathrm{kB}$).
- But you can *only* do this if you know exactly how these numbers are stored.
- Otherwise, you may need to add "spacer" bytes between each number (possibly all zeros, or some other pre-defined bit-battern) to help identify where one number stops and the other starts.
- Alternatively (or in combination with the above) you may need to add "header" bytes at the start of the array defining the number of digits that follow and the size of each. This header probably also needs "spacer" bytes to identify that it is a header and to separate it from the following data.

The basic point is this: making data and/or code more recognizable necessarily means using more memory. When programming a microcontroller, this tradeoff is not always acceptable.
Low-level programming can be more efficient because you only provide the required instructions in code. This is also the downside: everything the program does needs to be described in specific, incremental steps.

Machine Code and Mnemonics

ARM USES 32 bit words

The **opcodes** that a microprocessor understands are a set of predefined binary codes that often occupy the most-significant bits of a **word**, while the least-significant bits are used for context-dependent information (such as addresses or data values).

- For example, to initialize a register in the microprocessor with a small number, the opcode `0xe3a01005` tells the microprocessor to *initialize register* r_1 *with the value* 5 (in hexadecimal).
- The first four hex digits (`0xe3a0`) identify the instruction ("initialize a register"). There is a hard-coded table of these instruction codes in the logic circuit of the processor.
- The next hex digit (`0x1`) identifies the destination register (in this case, r_1). The connection between these register identifiers and the physical registers on the chip is hardwired.
- The final three hex digits (`0x005`) identify the data to be written to the register (in this case, the number 5). This obviously cannot be hard-coded, but it is part of the opcode.
- In this sense, the number 5 is a constant, so it arguably does not count as data, and when the microcontroller is programmed this entire opcode will be written into ROM.

Programming with opcodes directly can be done, but is extremely difficult. Instead, we use the slightly-less-difficult method of coding in **mnemonics**.

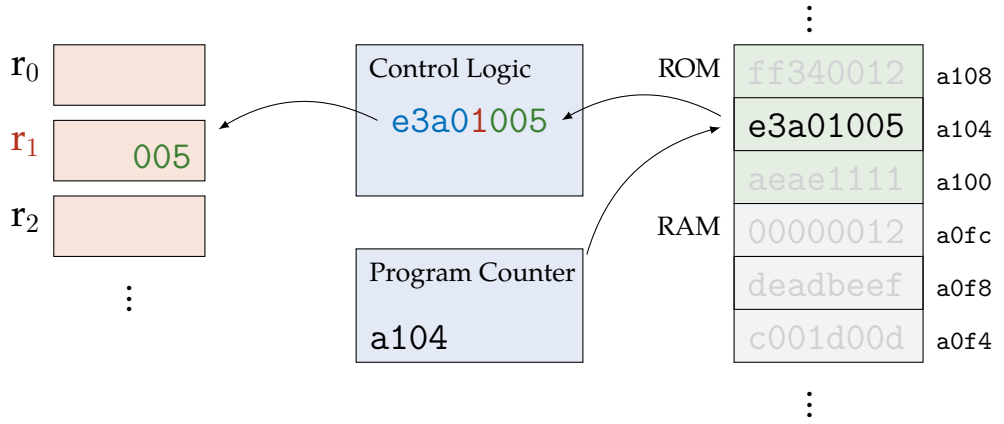

Figure 1: A sketch of the process within a microprocessor when the opcode `0xe3a01005` is executed. A special register called the *program counter* keeps track of where to find the next instruction. This is fetched from ROM and parsed through the control logic, where the opcode is intepreted and the value `0x5` is placed in register r_1. The program counter will then increment and the next opcode (apparently, `0xff340012`) will be fetched. Note that the memory addresses shown here increment by 4 (bytes), as each is schematically shown as containing a 32-bit number. In the actual device, each 32-bit number would be spread across 4 adjacent 1-byte memory cells.

Definition: A **mnemonic** is a "human-readable-esque" sequence of letters that has a one-to-one mapping with an opcode.

So instead of writing `0xe3a01005` in order to perform the operation of *initializing register* r_1 *with the value* 5, we would write:

```
mov r1, #5
```

This mnemonic parses exactly the same way as the opcode: the compiler interprets `mov` as `0xe3a0`, the register name `r1` as `0x1`, and `#5` as `0x005`. [2] This process is schematically shown in Figure 1.

[2] Although it is not always the case that a mnemonic has a *sequential* mapping like this example — but it is always the case that a mnemonic has *some* direct mapping to a single opcode.

Assembly Language and Microcontrollers

Assembly language is intimately related to the opcodes of a microprocessor, and consequently (and unfortunately for students!) there are many variants of assembly language, each unique to a particular system: the opcode `0xe3a01005` for an ARMv7-type microprocessor (like the ARM®Cortex-A9) won't have the same effect on an Intel x86_64 or a Motorola 68k system.

^ 64 bit system ^16 bit system

- There is usually some common ground with the mnemonics: although `mov r1, #5` won't work on an Intel x86_64 system, it is very close to the correct code. [3].

- Still, you can't expect code written for one microprocessor architecture to successfully compile and run on a different architecture.

[3] In fact, just `mov r1, 5` should work — drop the "#" in front of the decimal literal.

Although each microcontroller has its own particular set of attributes — they are unique snowflakes, just like you [4] — they all have common components. Every microprocessor has:

1. A set of **registers** for holding data and addresses that are being manipulated by the program. The ARM®Cortex-A9 has thirteen 32-bit general-purpose registers, called `r0`, `r1`, ..., `r12`.

2. A special **program counter** register that contains the address in ROM of the next instruction. This is called `pc` in the The ARM®Cortex-A9.

[4] And using a C-to-Assembler compiler to club these unique microprocessors into running the same standard program is just like how the education system forces conformity on you! (Insert unhinged rant about modern society here...)

will increment by 4

3. One or more special **program status registers** that contain the control bits returned from various operations. These generally cannot be accessed directly by user-written code. These are called `spsr` and `cpsr` in the ARM®Cortex-A9.

4. A **link register** that contains an address in ROM. This is used for coding simple branch-and-return loops. This is called `lr` in the The ARM®Cortex-A9.

5. A **stack pointer** register that holds the address of the stack (in RAM). This is used for more complex or nested loops. This is called `sp` in the ARM®Cortex-A9.

Assembly language generally consists of lines of code of the form:

label : *mnemonic operand1* [, *operand2* , *operand3*] [@ *comment*]

The concept of a mnemonic was already discussed, and most of the remaining code is probably self-explanatory. But anyway:

- Inline comments can be written as *@ comment*, or over several lines as */* long, multiline comment */*. Of course comments are always optional, but often a good idea — especially as assembly is so hard to read.

- Labels are used for the assembler equivalent of `GOTO` commands. While this type of coding is discouraged in higher-level programming, [5] it is common in assembly. Labels can be alphanumeric, and are optional.

[5] Or even made impossible by not having a `GOTO` command.

- Operands are generally registers, or literal numbers (in decimal or hex) or addresses (always in hex). For some commands a label may also be used as an operand. At least one operand is needed for every mnemonic, but some require more.

Assembly language programs also using include special code called *directives*. These are instructions for the compiler, rather than the microprocessor.

- A commonly-used directive in ARMv7, for example, is `.data`, used to define a section of the program that provides a set of numbers that should be pre-loaded into the microcontroller's memory.

With this in mind, we will now look at the types of instructions used in assembly language, and provide specifics for ARMv7 assembly language. The basic categories of assembly language instructions are:

1. Data movement instructions.
2. Data manipulation instructions.
3. Conditionals and test instructions.
4. Branch instructions.
5. Subroutine instructions.
6. Interrupt instructions.

The last three categories will be examined in a future lesson.

Data Movement Instructions

These are the type of instructions that those who are only familiar with high-level computer languages may not even think of as instructions. If you want to create a variable x with the value $y + 5$, then you would just enter the instruction `x=y+5`, right? In assembly it is not so simple: [6]

- First you have to fetch the value of y from memory and store it in a register.
- Then you have to put the literal 5 into a register.
- Then you need to run both registers through the arithmetic logic unit to add the values together, storing the result in another register.
- Then you need to write the sum to memory in the appropriate location (pointed to by the variable x).

There are three basic instructions for data movement: moving data between registers, from memory into a register, and from a register into memory.

Mnemonic: To **move** data between registers, use the `mov` mnemonic seen previously. Two operands are always required, the first is the register the data is moved *into*, the second is the register the data is moved *from*.

mov [to: register], [from: register or literal]

[6] Actually, for ARMv7 several of these steps can be combined, so it isn't quite this tedious.

Mnemonic: To **load** data into a register from memory, use the `ldr` mnemonic. Two operands are always required, the first is the register the data is moved *into*, the second is address in memory that the data is moved *from*.

Mnemonic: To **store** data from a register into memory, use the `str` mnemonic. Two operands are always required, the first is the register the data is moved *from*, the second is address in memory that the data is moved *into*.

As shown above, the second operand does not always have to be a register, it can be a decimal literal, or it can be the *label* of a pre-value defined value. Some examples:

```
.text
    mov r0, #10 @moves 10 into r0
    mov r1, r0   @moves r0 into r1
    mov r1, big_word @moves value at big_word into r1

.data
    big_word: .word 53400524
```

Note the use of directives: `.text` is used to identify the main code, and `.word` is used to identify a piece of data as a 32-bit word (rather than a `.byte` or some other size).
Using a register as the second operand will always use the value in that register as **data**. To use it as an **address**, enclose it in square brackets. This is needed for `str` and `ldr`. For example, to store the value in `r0` into the memory address contained in `r1`:

value of r0 and store it in the address contained in r1

```
str r0, [r1]    @this works
str r0, r1      @this will not compile
str [r0], r1    @this also will not compile
str [r0], [r1] @now we are just being silly
```

Initializing a register with a literal (as in `mov r1, #12`) only works with "small" numbers. [7]

- Use a "variable" (a label defined with a `.word` of data, as in the example above, for example) to get larger values into registers.

Another trick is to use `ldr` with a literal as the second operand. The syntax is a bit different from using a literal with `mov`, and the literal must be a hexadecimal number:

```
ldr r1, =0xff200020
```

register expecting an adress with ldr but a number is a number

As previously mentioned, the program counter is a 32-bit register called `pc`. This *can* be read and written in code, but probably *shouldn't*. For example:

```
mov r1, pc
```

will move the address of the next instruction in the program, stored in `pc`, into the general register `r1`. This is a safe thing to do, although it may not be very useful. You can also have "fun" and try something like this:

```
mov r1, #140
mov pc, r1      @wow, this is a stupid thing to do!
mov r1, r2      @this line will never execute!
```

[7] Refer back to the construction of the opcode to see why: there are limited bits available to store this literal data

As the second line of code overwrites the program counter to the address `0x140`, the last line of code will (probably) never run. [8] Instead, whatever mystery number was stored at address `0x140` in memory (assuming that spot is memory mapped) will be executed as an opcode — probably with terrible results.

- For that matter, the stack pointer `sp` and the link register `lr` can be overwritten and used as general-purpose registers if your really want to — but probably you shouldn't.
- You can't touch the program status registers. They cannot be accessed directly using code.

Finally, recall the ARM®Cortex-A9 has thirteen 32-bit general-purpose registers, called `r0`, `r1`, ..., `r12`.

- Some "higher-numbered" registers (in particular, 7, 11, and 12) can be overwritten by the processor for special purposes.
- As a general rule, use the low-numbered registers first to avoid unpleasant surprises.

The same is true if you overwrite the stack pointer and link register to use as general-purpose registers: regular instructions that use those registers can overwrite your data.

[8] Unless `0x140` is actually the correct address for the next line of code, or `0x140` already contained the appropriate opcode to return back to the appropriate spot.

Memory Addressing Modes

Reading and writing ordered lists (or arrays) of data is a common process in computer programming. To simplify this in assembly language, there is more than one method for looking up an address in memory.

- Typically, the memory address for the starting point of some construct will be stored in a register.

Assume that register `r0` stores such an address. In addition to the basic **loading** and **storing** data to/from that address, there are three other possibilities.

Definition: **Addressing with an offset** refers to accessing a memory element that is a given distance from the base address.

Definition: **Pre-indexed addressing** refers to changing the base address in the register *before* looking up the memory element at the new address.

Definition: **Post-indexed addressing** refers to looking up the memory element at the present address in the register, *then* shifting the address stored in that register.

These techniques are useful for reading/writing arrays of data, or for interacting with some memory-mapped I/O peripherals. These techniques work with any instruction that uses a register value as an address. As an example:

```
mov r1, #10
ldr r0, =0x00ff1000
str r1, [r0]        @direct address
str r1, [r0, #4]    @offset
str r1, [r0, #4]!   @pre-indexed
str r1, [r0], #4    @post-indexed
```

The first line initializes register `r1` with the value 10. The second line initializes register `r0` with the memory address `0x00ff1000`. Then the following things happen:

- Direct addressing is used to write the number 10 to the memory address `0x00ff1000`.
- Addressing with an offset is used to write the number 10 to the memory address `0x00ff1004`. The content of register `r0` is still `0x00ff1000`.
- Pre-indexed addressing is used to write the number 10 to the memory address `0x00ff1004` (again!). The content of register `r0` is changed to `0x00ff1004`.
- Post-indexed addressing is used to write the number 10 to the memory address `0x00ff1004` (third time's the charm!). The content of register `r0` is then changed to `0x00ff1008`.

Because pre-indexing and post-indexing change the base address, they can be used to efficiently cycle through an array of data. [9]

[9] Conceptually, it should be possible to combine pre- and post-indexing as: `str r1, [r0 #4]!, #4`, meaning the address is incremented by 4, the data is written to that address, then the address is incremented by 4 again, however ARMv7 won't let you do this.

Memory Alignment

The **memory width** of a microcontroller is often smaller than the **word size** of the microprocessor. This is the case with the ARM®Cortex-A9: the registers are 4 bytes, but the memory cells are only 1 byte wide.

- The data movement instructions discussed above are all based on manipulating a **word** of data. [10]
- Consequently, in the above examples, addresses are always given, and incremented, in multiples of 4 bytes.

[10] If you only wanted to move 1 byte of data to/from memory, use `strb`, `ldrb`. There is no "`movb`" however.

Because of this difference, it is important to keep numbers *aligned* in memory. If you store an array of 32-bit values to memory, but when trying to read those values the base address is accidentally offset by 1 byte, the result is probably useless. To help protect data and instruction sets from becoming misaligned, the following rules must be obeyed:

- Numbers stored as words (32 bits) can only be at addresses that are integer multiples of 4.
- Numbers stored as half-words (16 bits) can only be at addresses that are integer multiples of 2.
- Numbers stored as bytes (8 bits) can be at any address.

Consequently, the operation:

```
mov r1, #1
str r1, [r0]
```

uses 4 sequential bytes of memory to store the number 1. Reading from, or writing to, these 4 sequential bytes is handled automatically by `ldr` and `str`. If the address stored in `r0` is not a multiple of 4, executing this code in a simulator will generate an error message. [11]

Storing numbers across multiple memory cells raises the problem of *endian-ness*.

Definition: **Big-endian** data stores the *most* significant bytes at the *lowest* memory address.

Definition: **Little-endian** data stores the *least* significant bytes at the *lowest* memory address.

This is shown schematically in Figure 2. Big-endian storage is slightly more human-readable: when the memory cells are written out left-to-right, data values stored across multiple cells can be read naturally (the most significant digits are on the left side). Little-endian storage makes more sense logically, as the most significant bytes are at the largest memory addresses.

- Really it doesn't matter much which convention is used, especially if you only work with data in increments of words (which is recommended).
- However it becomes extremely important if your microcontroller is reading/writing to some external memory that is

[11] The code will compile just fine, but the simulator will terminate once it tries to write a word to an "unaligned" memory address. What would happen in actual hardware? I don't know, I haven't tested it. But nothing good, I imagine.

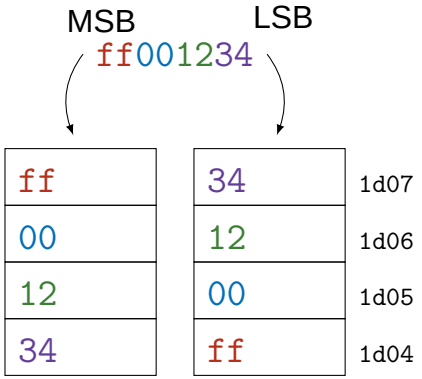

Figure 2: Storing a word of data as a little-endian value (left) and a big-endian value (right).

shared with another microcontroller that uses a different convention!

ARMv7-type microprocessors are **little-endian**. [12] This is important to know if you are inspecting the contents in memory of a program you ran in the simulator. In fact, you can see the contents of memory in the *Memory* tab on the simulator (obviously). But:

- The memory is grouped in words (8 hex digits), the base addresses of the words increase from left-to-right, and top-to-bottom.
- But *inside* each word, the addresses read right-to-left.
- So the "memory-word" address `0x104` is to the right of address `0x100`, but the actual byte `0x100` is the last two digits of this word (right side), and the byte `0x101` is to the left of it.

With this internal inversion in the byte order, an word is still read left-to-right as a number even though the architecture is **little-endian**. This is shown in Figure 3.
Again, as long as you only interact with memory in units of **words**, make sure you always declare memory addresses (or memory address increments) in multiples of 4, and don't share peripherals with a big-endian system (unlikely in practice, and impossible with the simulator), these details are not important.

[12] Actually the endian-ness is configurable, but the default is little-endian.

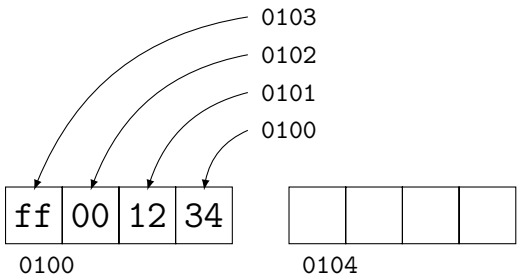

Figure 3: An example of how memory is *displayed* in the ARMv7 simulator. Here the word `0xff001234` is stored starting at address `0x0100`. The word is displayed such that it can be visually read left-to-right, as normal text. However since the architecture is little-endian, that means that the actual bit at address `0x0100` is on the right-side of the word.

Data Manipulation Instructions

Now that you know how to get data into and out of memory, what to do with it? The basic ways of manipulating data are by arithmetic operations, logic operations, and bit shifting.

- Arithmetic and logic operations use the arithmetic logic unit (ALU) in the microprocessor to manipulate the contents of one or two registers.
- Bit shifting can also use the ALU, but in ARMv7 architecture a special piece of hardware called the *barrel shifter* is often used.
- These kinds of data manipulations are obviously important for performing calculations, but they can also be used for making *comparisons* between two numbers.

Most microprocessors support addition, subtraction, multiplication, and division. Addition and subtraction are straightforward, but multiplication can be a bit tricky because the output can require significantly more bytes than the inputs, [13] and useful division often requires floating-point arithmetic... Anyway, multiplication and division are not discussed in this class.

Mnemonic: To **add** two numbers together, use the `add` mnemonic. Two operands are always required, the first is the register where the sum is stored, the second is the register with a number. If a third operand is provided, it will be used as the second number in the sum, otherwise the first operand will be used as the second number.

[13] For example, to hold the result of $a \times b$ in a register, the number of bits in a and b after leading zeros are removed should add to 32 or less.

Mnemonic: To **subtract** one number from another, use the `sub` mnemonic. Two operands are always required, the first is the register that stores the result, the second is a number. Optionally, a third operand can be provided — if given, the result is the third operand subtracted from the second. Otherwise the result is the second operand subtracted from the first.

Mnemonic: To **reverse subtract** one number from another, use the `rsb` mnemonic. Two operands are always required, the first is the register that stores the result, the second is a number. Optionally a third operand can be provided — if given, the result is the second operand subtracted from the first. Otherwise the result is the first operand subtracted from the second.

These mnemonics are pretty easy to remember and straightforward to use. The most important rule to remember is that if a register is given as the third operand, the result is stored in that register. If only two operands are given, the contents of the *first register are overwritten with the new result*. Some examples:

```
add r1, r2, r3   @compute r1=r2+r3
add r1, r2       @compute r1=r1+r2
add r1, #4       @compute r1=r1+4
```

In the above examples, the last two lines of code cause the contents of `r1` to be overwritten. Remember that the first operand must *always* be a register.

```
add r1, #17      @this works!
add #17, r1      @causes a compiler error
```

Requiring the first operand to be a register is the reason why there is a **reverse subtract** operation.

```
sub r1, #12      @compute r1=r1-12
sub #12, r1      @causes a compiler error
rsb r1, #12      @compute r1=12-r1
sub r1, r2, r3   @compute r1=r2-r3
rsb r1, r3, r2   @compute r1=r2-r3
```

As noted in the comments, the last two lines are identical. The main use for `rsb` is when you are using a literal (like `#12` in the above example) instead of a register for the subtraction.

Data Manipulation and Status Flags

The four MSbs of the **program status register** [14] are flags that can be returned by the ALU.

- Bit 31 of the `cpsr` is the **negative** flag (`N`), which is set to 1 if a signed arithmetic operation returns a negative answer. [15]
- Bit 30 of the `cpsr` is the **zero** flag (`Z`), which is set to 1 if the result of an arithmetic operation is zero.
- Bit 29 of the `cpsr` is the **carry** flag (`C`), which is set to 1 if an unsigned addition overflow, or a unsigned subtraction "underflow" (also called a **borrow**) occurred. [16]
- Bit 28 of the `cpsr` is the **overflow** flag (`V`), which is set to 1 when a signed addition or subtraction caused the sign of the result to be changed improperly.

These status flags are important for determining whether an arithmetic operation produced a valid result. They are also important for performing comparisons between two numbers.

- For example, to test if $A > B$, we can just perform the subtraction $A - B$, ignore the result, and examine the **carry** flag (in this case, acting as a borrow).

However these status flags are not always set by a data manipulation instruction. The mnemonics listed above only set the status flags after performing their particular operation if a "s" is appended to the mnemonic: as `adds`, `subs`, and `rsbs`.

[14] Actually called the *current program status register*, or `cpsr`.

[15] Remember, the CPU does not know what numbers are signed or unsigned. However `movs`, `strs` and `ldrs` can be used to make it more explicit that the value is signed: `movs r1, #-5` is the same as putting `0xfffffffb` into `r1`.

[16] This flag can also be set by some bitwise shifting operations to represent a bit that is "popped off" after the shift.

Comparisons and Conditional Execution

Most (probably all) useful computer programs require some ability to make decisions based on the data provided. This requires data to be compared, and some code to be executed only under certain conditions. In assembly language, data comparison uses a method based on the data manipulation techniques described above.

Mnemonic: To **compare** two numbers, use the `cmp` mnemonic. This mnemonic requires two operands, the first of which must be a register.

The `cmp` mnemonic is functionally the same as subtracting the two numbers and discarding the result, but updating the status flags described above. [17]

```
cmp r0, r1      @calculate r0-r1, discard answer
subs r2, r0, r1 @same as above but keep answer
```

Once the status flags are set, a two-letter conditional execution code can be appended to almost any mnemonic. This will mean that mnemonic is only executed under certain conditions. To give two exmaples:

Mnemonic: To **conditionally add** two numbers only when a previous comparison demonstrated that two numbers were equal, use the `addeq` mnemonic. This mnemonic is otherwise identical to the `add` mnemonic.

Mnemonic: To **conditionally write** a number to memory only when a previous comparison demonstrated that one number was greater

[17] There are also mnemonics for conditionals based on adding two numbers, or performing the logical AND or OR of two numbers — sometimes using these simplifies a logical expression, but otherwise they are arguably less straightforward than `cmp`.

or equal to

than another, use the `strge` mnemonic. This mnemonic is otherwise identical to the `str` mnemonic.

A complete list of the two-letter conditional execution codes is given in Table 1.

- It is useful to remember that while `cmp` might be the most convenient mnemonic for testing a condition, really we just need the status flags updated to perform a condition test. Therefore sometimes it is more efficient to use a data manipulation step to also update the status flags (with `adds` or `subs`, for example).
- However it is also important to *avoid* updating the status flags when performing conditional execution multiple-cases.

Consider the following example:

```
mov r1, #10
mov r2, #25
ldr r3, =0x100

cmp r1, r2
addhi r1, #2 @ add r1+2 if r1>r2
sublos r1, #2 @whoops!
strlo r1, [r3] @this won't happen
```

Here two registers are initialized with data and a third with an address. The two data values are compared:

- If `r1` is greater than `r2`, then `r1` is further increased by 2.

Signed	Unsigned	Description
`eq`	`eq`	$x = y$
`ne`	`ne`	$x \neq y$
`gt`	`hi`	$x > y$
`ge`	`hs`	$x \geq y$
`lt`	`lo`	$x < y$
`le`	`ls`	$x \leq y$
`mi`	`mi`	$x - y < 0$
`pl`	`pl`	$x - y > 0$
`vs`	`vs`	`V` $= 1$
`vc`	`vc`	`V` $= 0$

Table 1: List of conditional execution codes, which will be true when the condition for two numbers x and y given in the description is valid. Many conditionals have different codes depending on whether the test is with signed or unsigned numbers. The last two codes test whether the "overflow is set" or the "overflow is clear", here `V` is the overflow status flag.

- If `r1` is less than `r2`, then `r1` is further decreased by 2 — and the *status flags are reset*!
- Now the value in `r1` is stored in memory, but only if the value in `r1` is less than 2!

As $10 < 25$, the addition will not occur but the subtraction will. As we used `sublos` instead of `sublo`, this subtraction will overwrite the status flags from the original `cmp`. Therefore, although we (probably) intended to write `r1` to memory as $10 < 25$, instead this will not occur: the condition flags are based on the subtraction $10 - 2$, and as $10 > 2$, the "less than condition for unsigned numbers" will evaluate as false.

Bitwise Operations

There are a variety of bitwise operations that can be performed on a value in a register. Most of you should be familiar with the logical AND and OR operations.

Mnemonic: To compute the **bitwise and** of two numbers, use the mnemonic `and`. This requires at least two, and optionally three operands, the first of which must be a register. The use of these three operands is the same as for `add` and `sub`.

Mnemonic: To compute the **bitwise or** of two numbers, use the mnemonic `orr`. This requires at least two, and optionally three operands, the first of which must be a register. The use of these three operands is the same as for `and`.

Additionally, one of the claims to fame for ARM-type microprocessors is a special piece of hardware called a *barrel shifter*. This is very fast piece of hardware for rearranging the bits in a binary sequence. It is implemented in assembly during a regular mnemonic code, and these operations can be performed to the *second* operand in a mnemonic. Some of these bitwise operations are listed in Table 4. Each of these operations is followed by a number, indicating the number of bits for the operation. Logical and arithmetic shifts fill in the missing bits with zeros, while rotate wraps around. The only difference between logical and arithmetic shifts is that arithmetic shifts preserve the *most significant bit* of the number, in case that is a sign flag.

Code	Description	
`lsl`	Logical shift left	shift in 0s
`lsr`	Logical shift right	
`asl`	Arithmetic shift left	preserves MSB
`asr`	Arithmetic shift right	
`ror`	Rotate right	wrap around
`rrx`	Rotate right, extend	

Figure 4: Some operations of the barrel shifter. The difference between logical and arithmetic shifts is whether or not the MSb is preserved as a "sign bit".

For example, if an 8-bit register `r0` holds the binary value `0b0110 0111`, then `mov r1, r0, lsl #3` will put the value `0b0011 1000` into register `r1`. [18]

[18] From the number `0b0110 0111`, add three zeros on the right to get `0b011 0011 1000`, then discard the first 3 bits on the left to return to an 8-bit number.

- These bitwise operations are sometimes used to simplify math — as shifting all the bits left or right is a much quicker way to multiply or divide by 2 than actually using the ALU to calculate it.

- Another, possibly more common use for these operations is when the 32-bit ARM registers need to communicate with peripherals that have larger or smaller memory cell sizes. For example, if we read data into `r1` and `r2` as 16-bit numbers (i.e., in the 32-bit registers, the most-significant 16 bits are zeros), we want to combine these into one 32-bit number as:

  ```
  add r1, r2, lsl #16
  ```

  This puts the contents of `r2` as the most-significant 16 bits of `r1`.

These barrel shift operations probably aren't that important to this course, but it is worth having a syntax reference.

Example

The Terasic DE1-SoC has a set of 10 switches and a bank of 10 LEDs. We will write a program to do the following:

- Read in the state of the switches, and represent the first 5 switches as one binary number (read left-to-right), and the last 5 switches as a second binary number (again, read left-to-right).
- Add these two numbers together.
- Subtract the result from the magic number, 789.
- Output the final result to the LED bank.

One way to implement this is as follows:

```
.global _start

.data
@define the magic number
magic_num:   .word 789

.text
_start:
        @load addresses for magic number
        @and i/o hardware
        ldr r3, adr_magic
        ldr r4, adr_led
```

```
        ldr r5, adr_switch

        @read in state of the switches
        ldr r1, [r5]
        @ move the 5-MSbs into
        @ register r2
        mov r2, r1, lsr #5
        @ mask the 5-MSbs from original
        @ note that 31 = 0b11111
        and r1, #31

        @ add and overwrite
        add r1, r2
        @ get magic number
        ldr r2, [r3]
        @ subtract and overwrite
        sub r2, r1
        @ write to LED bank
        str r2, [r4]

@ labels for addresses
adr_magic:  .word magic_num
adr_led:    .word 0xff200000
adr_switch: .word 0xff200040
```

Note the use of directives to define data values (`.data`, where `magic_num` is defined) and the main code (`.text`). Note also the use of labels at the end of the code to identify addresses.

- Like all memory-mapped peripherals, the addresses for the LEDs and switches are visually shown in the online simulator in the top-right corner of the window.
- Technically it is not necessary to define a label for these addresses, they could be loaded directly into registers (as `ldr r3, =0xff200000`).
- It is useful to use a label for the address of the magic number. The data block creates the magic number in memory, but we don't know *where* in memory — that is determined by the compiler. Using a label for the address that is attached to the label for the data value is a way to tell the compiler to remember where it put that value.

Finally, white space and indentation is unimportant in assembly, so feel free to format your code however you like.

Confession: After all the effort last lesson to describe and distinguish between RAM and ROM, it appears that the online simulator makes no such distinction. You are free to read/write data to any memory cell. *Probably* defining data by labels and letting the compiler choose where to put them in memory (like `magic_num` and `adr_magic` in the example) is the "best practice", as in a real-world implementation the microcontroller would know to put that data into RAM. However for the labs/homework in this course you can hardcode memory addresses for data anywhere in the available space if you want.