Introduction to ARMv7 Assembly Language Programming

Profs. John McLeod & Arash Reyhani

ECE3375, Winter 2023

This lesson provides an introduction to the ARMv7 Assembly language. You need to know ARMv7 Assembly to understand many of the examples that will be used in class, and to answer questions on the midterm and final exams. For the lab exercises and projects you have the option of using ARMv7 Assembly or C (and, at this level, C is almost the same as Java — mostly just some differences in syntax), but we don't have the time to provide full support for learning C. And assembly language is fun! This lesson covers the basics of ARMv7 assembly language, with respect to moving and manipulating data, evaluating tests and conditions, and branching.

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. There are three basic instructions for data movement: moving data between registers, from memory into a register, and from a register into memory. These concepts were already discussed last week, but I want to revisit them here briefly just to make sure you are familiar with ARMv7 syntax.

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*.

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 prevalue defined value.

Some examples:

```
    mov r0, #10
    mov r1, r0
    ldr r1, big_word

    big_word: .word 53400524
```

Note the use of directives: `.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`:

```
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. [1]

- 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:

[1] This is because the opcode for `mov` has some spare bits that can be used to record the literal directly in the instruction.

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

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, #0x140
mov pc, r1      @wow, this is a stupid thing to do!
mov r1, r2      @this line will never execute!
```

As the second line of code overwrites the program counter to the address `0x140`, the last line of code will (probably) never run. [2] 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.

[2] 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. [3]

[3] 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.
- Consequently, in the above examples, addresses are always given, and incremented, in multiples of 4 bytes.

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. [4]

[4] 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.

Words, Half-Words, & Endian-ness

Storing numbers across multiple memory cells also raises the problem of *endian-ness*: in what order do the contents of each memory cell combine to make the large number? Consider storing the number `0x1A2B` in a system with a 16-bit address space and 8-bit memory cells, starting at address `0x1000`. There are two ways to store this number, as shown in Figure 1.

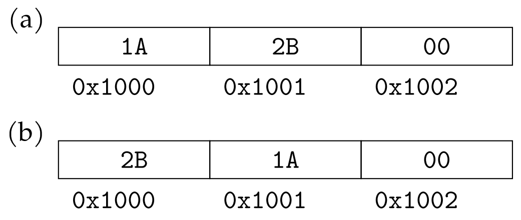

Figure 1: Two ways to order bytes in memory for the half-word `0x1A2B`: (a) big-endian, and (b) little-endian. (The data at address `0x1002` is for the next byte/half-word/word of data.)

Definition: A computer system is **big endian** if the most significant byte in a larger number is stored at the lowest memory address.

Definition: A computer system is **little endian** if the most significant byte in a larger number is stored at the highest memory address.

There are reasonable arguments for and against both methods of storing data.

- As shown in Figure 1, the **big endian** convention is easier for a human to read. If we examine the memory cells sequentially from the lowest address to the highest address, multi-byte numbers are presented in the same way we have learned to read them: from most significant digit to least significant digit.
- However, logically, **big endian** is silly because the largest part of the number is at the lowest address.
- **Little endian** is logically more sensible, because the least significant byte is stored at the lowest memory address, and the most significant byte is stored at the highest memory address.
- However **little endian** is less human readable.

By default, ARM processors are **little endian**, although they can be reconfigured if necessary. In most circumstances, programmers and users never know, or care, about the *endian-ness* of the system. If two microprocessors with *different* conventions need to exchange data, however, then some kind of hardware or software conversion is required to ensure data is read correctly by each system.

Loading and Storing Half-Words & Bytes

As mentioned above, the mnemonics `str` and `ldr` are all based on manipulating a **word** of data. For this course, usually that is sufficient, but sometimes you might want to access the contents of only a single byte or (probably even less frequently) manipulate 16-bit half-words.

Mnemonic: To **load** a single byte from memory, use the `ldrb` mnemonic. All unused bits in the register are filled with zeros. Any integer can be used as the memory address, not just integers evenly divisible by 4. Otherwise this mnemonic is the same as the `ldr` mnemonic discussed previously.

Mnemonic: To **store** a single byte to memory, use the `strb` mnemonic. Any integer can be used as the memory address, not just integers evenly divisible by 4. Otherwise this mnemonic is the same as the `str` mnemonic discussed previously.

Mnemonic: To **load** a half-word from memory, use the `ldrh` mnemonic. All unused bits in the register are filled with zeros. Any *even* integer can be used as the memory address, not just integers evenly divisible by 4. Otherwise this mnemonic is the same as the `ldr` mnemonic discussed previously.

Mnemonic: To **store** a half-word to memory, use the `strh` mnemonic. Any *even* integer can be used as the memory address, not just integers evenly divisible by 4. Otherwise this mnemonic is the same as the `str` mnemonic discussed previously.

When loading a word, it is irrelevant whether or not the data is signed or unsigned. However since the CPU registers are all 32-bits, it is important to know whether the data is signed or unsigned when loading a half-word or a byte.

- The representation of a signed number depends on the number of bits available.
- For example, $(-7)_{10}$ is represented in 8-bits as [5] `0b1111 1001`, while in 32-bits it is `0b1111 1111 1111 1001`.
- Clearly for a signed *negative* number, all higher-order, unused bits in the register should be flipped to 1.

[5] Using 2's-complement convention, of course.

For this reason, there are special mnemonics for loading signed bytes and half-words.

Mnemonic: To **load** a single signed byte from memory, use the `ldrsb` mnemonic.

Mnemonic: To **load** a signed half-word from memory, use the `ldrsh` mnemonic.

The mnemonics to load signed bytes and half-words aren't terribly important for this course, but as a conceptual matter hopefully everyone clearly understands *why* it doesn't matter whether or not a word is signed, but it does matter whether or not a byte or a half-word is signed.

The ARMv7 language has mnemonics for storing signed bytes and signed half-words as well (namely, `strsb` and `strsh`) but the user manual states that these function exactly the same as an unsigned store.

Loading and Storing Multiple Words

In complement to loading and storing data smaller than a word, sometimes it is useful to load and store multiple words of data. Each register can only store a single word, so in principle this is just done by applying `ldr` or `str` multiple times to different registers. However to streamline this process there is the option to move words to or from a set of registers and a sequential range of memory.

Mnemonic: To **load** multiple words from a sequential range in memory to a set of registers use the `ldmia`, `ldmda`, `ldmib`, or `ldmdb` mnemonics. Unusually, the first operand is the register containing the base address in memory, and this operand should be preceded by a `!`. The second operand is a set of brace brackets with a list of registers to load data into: such as `{r0,r1,r3}`.

Mnemonic: To **store** multiple words from a sequential range in memory to a set of registers use the `stmia`, `stmda`, `stmib`, or `stmdb` mnemonics. This mnemonic has the same syntax as that for loading multiple words.

Four mnemonics each are listed above for loading or storing, the difference between these mnemonics is in regards to how the base address is incremented for each register. This is summarized in Table 1.

Code	Description
`ia`	Increment After
`da`	Decrement After
`ib`	Increment Before
`db`	Decrement Before

Table 1: Summary of addressing codes used in storing or loading multiple registers.

- "Increment" means the base address is increased by 4 after loading/storing each register, while "decrement" means the base address is decreased by 4 after loading/storing each register.

- "After" means this address shift occurs after loading/storing to each register, "before" means this address shift occurs prior to loading/storing to each register.

The "before" and "after" codes are similar to **pre-indexing** and **post-indexing** discussed above. An example of storing data from multiple registers is given below.

```
ldr r0, =0x2000
mov r1, #12
mov r2, #140
mov r5, #10
stmia r0!, {r5, r1, r2}
```

The first line loads a memory address into `r0`, the next few lines initialize some registers with values. Then these registers are written to memory.

- Interestingly, the ARMv7 assembler always *sorts* the registers in ascending numerical order, so `r1` is stored to memory at the *lowest* address, and `r5` is stored to memory at the *highest* address, regardless of how the registers are listed or whether the store operation is incrementing or decrementing. [6]
- The mnemonic is for incrementing the base address after storing, so `r1` is stored to the address `0x2000`.
- `r2` is then stored to address `0x2004`, and `r5` is stored to address `0x2008`.

[6] I will never stop being peeved about this. What is the point of writing in Assembly if the assembler cleans up all the arbitrary and stupid things you want to do?

- After this operation is completed, r0 holds the value 0x200C.

If the following code is then immediately executed:

```
ldmda r0!, {r2, r1, r5}
```

The data in memory starting at the address in r0 is loaded into the registers.

- Again, the ARMv7 assembler always *sorts* the registers, but this time in **descending** numerical order, so r5 will be loaded the base address first (as it is at the highest address), and r1 is loaded from memory last (as it is at the lowest address).
- Following from the previous example, this means r5 will read the value stored in memory at 0x200C, r2 the value stored in memory at 0x2008, and r1 the value stored in memory at 0x2004.
- After this operation is completed, r0 again holds the value 0x2000.

The point of this odd-seeming rearrangement of registers in these operations is such that commands like the following:

```
stmia r0!, {r3, r1, r2, r4} @ register order
ldmdb r0!, {r4, r2, r3, r1} @  does not matter
```

Result in the system being in the exact same state as when it started. These commands may seem strange, but they exist for a useful reason.

- In a higher-level code (C, Java, Python, etc.) you can always define extra user variables whenever you want.

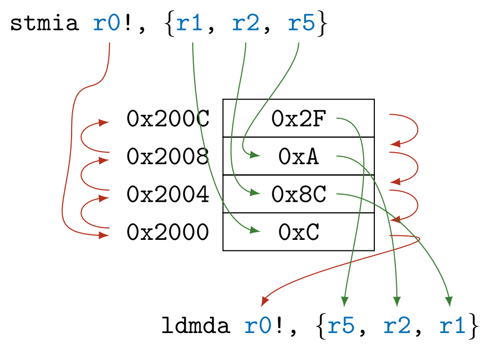

Figure 2: Visual demonstration of storing and loading multiple registers. Here the registers have been written in the appropriate order for each operation.

- In assembly, however, you are limited by the number of registers that physically exist in the CPU architecture.
- If your program needs to perform some complicated calculation, and requires space for temporary variables, you can perform a `stmia r7! {r0 - r5}` (for example) to store the contents of several registers to memory. [7]
- Then your program can use registers `r0` to `r5` for the complicated calculation.
- After obtaining the result, your program can restore it's original state (i.e. overwrite the temporary variables with the original values) using `ldmdb r7! {r0 - r5}`.

[7] As shown in this sample code, you do not need to list every register if they fall in numeric sequence.

This process of temporary storing registers to memory, then restoring them later, will be revisited in a later unit when we discuss **stacks**.

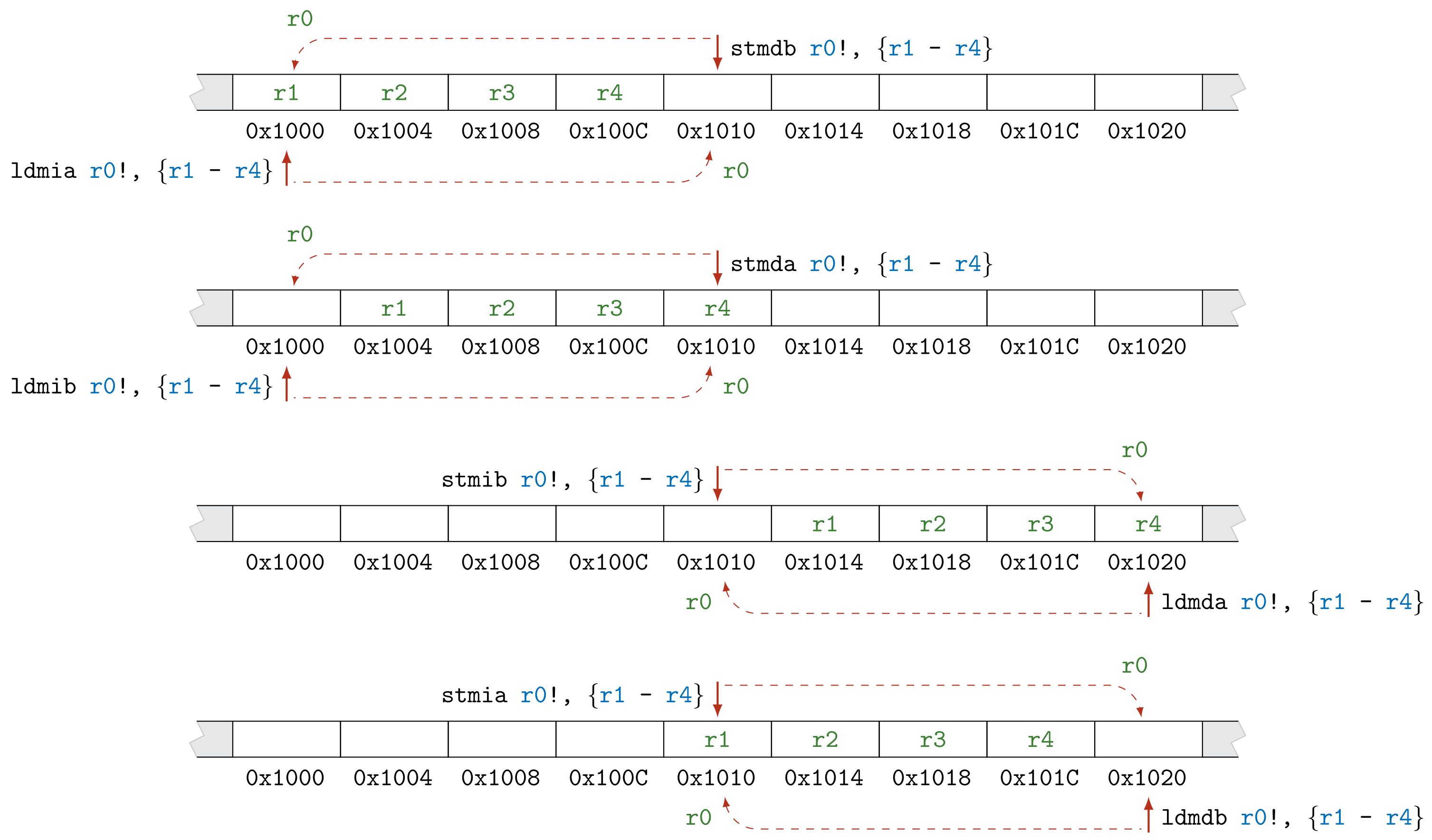

Figure 3: Visual representation of matching pairs of store and load multiple instructions that return the system to the original state.

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. Multiplication and division will not be discussed in this course. Addition and subtraction were introduced last lesson, but just to review:

Mnemonic: To **add** two numbers together, use the `add` mnemonic. Two operands are always required, and the first one must be a register. Three operators may also be given, in which case the first two must be registers. If two operands are given, the two values corresponding to each operand are added together and the result is stored in the first operand (which must be a register, remember), *overwriting* the previous value. If three operands are given, the values of the *last two* operands are added together and stored in the first.

Mnemonic: To **subtract** one number from another, use the `sub` mnemonic. Two or three operands can be provided. `sub` works exactly the same way as `add`, except it performs subtraction instead of addition.

Mnemonic: To **reverse subtract** one number from another, use the `rsb` mnemonic. Two or three operands can be provided, and `rsb` works similarly to `add` and `sub`.

These mnemonics are pretty easy to remember and straightforward to use. The most important rule to remember is that 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** [8] 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. [9]
- 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 occurred, and cleared to 0 if an unsigned subtraction "underflow" (also called a **borrow**) occurred. [10]
- 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`.

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

[9] 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`.

[10] 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. [11]

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

[11] 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`.

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

Condition Code	Type	Description	Status Flags Checked
`eq`	Any	Equal $(x = y)$	`Z=1`
`ne`	Any	Not equal $(x \neq y)$	`Z=0`
`mi`	Any	Negative, or "minus" $(x - y < 0)$	`N=1`
`pl`	Any	Positive, or "plus" $(x - y > 0)$	`N=0`
`vs`	Any	Overflow flag is set	`V=1`
`vc`	Any	Overflow flag is clear	`V=0`
`gt`	Signed	Greater than $(x > y)$	`Z=0` and `N=V`
`ge`	Signed	Greater than or equal $(x \geq y)$	`N=V`
`lt`	Signed	Less than $(x < y)$	`Z=1` or $\mathtt{N}=\overline{\mathtt{V}}$
`le`	Signed	Less than or equal $(x \leq y)$	$\mathtt{N}\neq\mathtt{V}$
`hi`	Unsigned	Greater than $(x > y)$	`C=1` and `Z=0`
`hs`	Unsigned	Greater than or equal $(x \geq y)$	`C=1`
`lo`	Unsigned	Less than $(x < y)$	`C=0`
`ls`	Unsigned	Less than or equal $(x \leq y)$	`C=0` or `Z=1`

Table 2: 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.

A complete list of the two-letter conditional execution codes is given in Table 2.

- 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 multiple conditional executions based on the same original test case.

Consider the following example:

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

cmp r1, r2
addhi r1, #2
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.
- 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.

There are also three other commands that can also be used for comparisons.

Mnemonic: To **compare** a number and the negative of another number, use the `cmn` mnemonic. This mnemonic requires two operands, the first of which must be a register.

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

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

The definitions of `cmn`, `tst`, and `teq` probably don't make a lot of sense. These conditions are basically the same as `cmp`, in which an arithmetic or logical operation is applied to both numbers, the status flags are updated, and the result of the operation is discarded.

- The `cmn` mnemonic is functionally the same as adding two numbers and discarding the result. If one of these numbers is the 2's compliment of the other, the sum will add to zero — hence the name "compare negative" for this mnemonic.
- The `tst` mnemonic is functionally the same as performing the bitwise `AND` of the two numbers and discarding the result.
- The `teq` mnemonic is functionally the same as performing the bitwise `OR` of the two numbers and discarding the result.

The `tst` and `teq` mnemonics are used to test the *logical equivalence* of an entire register. A multi-bit value evaluates as "false" only if all bits are zero.

I think you can get by pretty well just using `cmp`. This is the comparison that makes the most sense; the other three variants are more specialized. It is also important to recognize that the *description* of the condition codes in Table 2 are only valid for the `cmp` operation. The *Status Flags Checked* is always appropriate — the condition codes always perform the same checks on the status flags to determine whether or not code should execute — but the *meaning* is different.

As an example, test the following code:

```
ldr r0, =0x88AABEEF
mov r1, r0

cmp r0, r1
moveq r2, #5  @ execute if r0 = r1?

cmn r0, r1
moveq r2, #3  @ execute if r0 = r1?

tst r0, r1
movgt r3, #4  @ execute if r0 > r1?

teq r0, r1
movle r2, #6  @ execute if r0 <= r1?
```

Check the status flags after each comparison. Do the conditional statements execute as expected?

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.

Code	Description
`lsl`	Logical shift left
`lsr`	Logical shift right
`asr`	Arithmetic shift right
`ror`	Rotate right
`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".

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.

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. [12]

12 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.

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.

Addendum to Confession: I tried this in hardware and the device will accept code that rewrites over the memory addresses used by the program code, but this doesn't actually overwrite the program itself. I guess this is an aspect of the "modified Harvard architecture" of the device — when the program writes to memory perhaps it is writing to a physically different device (i.e. RAM) at the same address as the program?

Branching

As previously discussed, the **program counter** (register `pc`) keeps track of the next instruction in memory. Under normal circumstances the program counter automatically increments by 4 after each instruction is completed, so the program is read out sequentially from memory.

- Note that for **conditional exectution**, the instruction is still *read* sequentially, it just may not be *executed* depending on the present condition.

For a fully functional programming language, sequential execution is not sufficient: there must be a way to **branch** to non-sequential regions of code.

- As `pc` acts as a normal register, programmers who are fans of anarchy already know how to create a branch in their programs: just conditionally add or subtract values from the `pc`.
- This is a pretty extreme solution, and as you might have guessed, their are better options.

The preferred way to create branches in your program is to use the equivalent of a `GOTO` command to jump to a label somewhere else in your code.

Mnemonic: To perform a **direct branch** to a known location in your program, use the `b` mnemonic. This requires a single label as an operand.

Mnemonic: To perform an **indirect branch** to a variable location in your program, use the `bx` mnemonic. This requires a single register as an operand, the register should hold the memory address of appropriate instruction.

Like every mnemonic, both b and bx can have appended conditional codes. Consider the following examples:

```
mov r0, #8       @initialize r0
cmp r0, #5       @is r0 < 5?
strlo r0, [r1]   @store to memory if so
```

This piece of code stores the value in register r0 to the memory address in r1 if the value in r0 is less than 5. Pretty clutch piece of code, eh? We can implement the same thing using a branch:

```
mov r0, #8       @initialize r0
cmp r0, #5       @is r0 > 5?
bhi more_code    @skip next line if not
str r0, [r1]
more_code:
```

This piece of code puts the conditional on the branch mnemonic blo, so the str instruction be completely skipped if $8 > 5$. [13] In this example, using a branch is probably a worse idea than just using a conditional execution of str. However if there were multiple lines of code that were conditional, using one branch mnemonic to skip over then entire block rather than write each instruction as a conditional execution may be better.

[13] As opposed to the first example where it would be read but not executed if $8 > 5$.

Finally, just for fun, and to to confuse your enemies, you could implement the same code using indirect branches:

```
mov r0, #8
mov r2, pc
add r2, #12
cmp r0, #5
bxhi r2
str r0, [r1]
more_code:
```

In this example, the program counter is stored in `r2` and then increased by 12, as the line after the `more_code` label is four lines ahead of the point where `pc` is stored in `r2`. [14]

The above examples of branching are a obviously contrived, and also more confusing than just using conditional execution. Some more relevant examples will follow.

[14] Here `pc` is incremented by 4 before it is written to `r2`, and $16 = 12 + 4$.

If... Then... Else

Most programming languages have an *if...then...else* conditional structure.

- This can be implemented without branches, just by having the appropriate conditionals and counter-conditionals on the *then* and *else* statements.

```
cmp r0, #5     @if r0 <= 5
addls r1, r2   @then
subls r0, #1   @then
addhi r1, r3   @else
addhi r0, #1   @else
```

Adding conditional codes makes the mnemonics longer and arguably harder to read. If there is only one or lines of code for each condition, then it is probably appropriate. However your conditional requires several lines of code for the *then* and/or *else* block, using a branch to avoid one or the other can make the code simpler to read.

- Without using branches, you are also unable to update the **status flags** inside an *if...then...else* block — so you can't have any nested conditionals. Using branches fixes this problem.

The following example uses the same basic code as the above example, but uses branches to allow the subtraction step to include a check in the event that `r0` becomes negative.

```
if:     cmp r0, #5     @check r0=5
        bhi else
then:   add r1, r2
        subs r0, #1    @decrement r0
        bmi neg        @is r0 negative?
        b endif
else:   add r1, r3
        add r0, #1
        b endif
neg:    mov r0, #10   @roll r0 back to 10
endif:  str r1, [r4]
```

In additional to the extra functionality and flexibility of branches, the explicit use of labels can help make your code a bit more readable.

Creating Loops

Creating loops is an important functionality for a programming language, and this can be done neatly using conditional branching. There is no explicit *for...next* or *do...while* loops in assembly as there might be in other programming languages, but the same functionality can be obtained by using conditional branches in the correct place. For example, the following code implements a *for* loop that iterates 10 times.

```
         mov r4, #10   @initialize counter
loop:    cmp r4, #0    @exit loop?
         beq endloop
         /*a whole bunch of code here*/
         sub r4, #1    @decrement counter
         b loop        @go back to start
endloop: /*rest of code here*/
```

To make an un-iterated loop is even simpler, just omit the counter.

```
loop:    cmp r4, #17
         beq endloop
         /*More code here.
           I hope somehow r4 will equal 17
           or this loop will never end*/
         b loop
endloop: /*rest of code here*/
```

It is also worth noting that many microcontrollers are designed to run in perpetuity: unless shut down manually by the user they should repeat their programming.

- It is typically good practice to have the entire main program in an endless loop. Once the end of the code is reached, the last line should be `b _start` or something similar.
- It may also be necessary to loop almost endlessly while waiting for user input.

The following example demonstrates a loop that waits unit one of the switches on the DE1-SoC development board is thrown. When using the simulator and stepping through the code line by line, there is plenty of time to toggle switches. However a real ARM®Cortex-A9 operates at clock speeds of at least 800 MHz — millions of processor cycles will have passed before your fat fingers manage to touch one of the switches.

```
ldr r4, =0xff200040 @memory address for switches
ldr r0, [r4]        @get initial state
/*some code here*/
pause_for_input:
   ldr r1, [r4]     @read switches
   cmp r1, r0       @check if switches changed
   beq pause_for_input
mov r0, r1          @new "old state" of switches
/*rest of code*/
```