How Does a CPU Work?

A **central processing unit** (CPU) is an electronic machine, and like any machines, it works by repeating the same actions repeatedly in a loop. The actions that a CPU performs are:

1. Load an instruction from memory.
2. Setup the operands for the instruction.
3. Execute the instruction.

The outcome of these three steps depends on what the instruction is that was loaded from memory. A **machine instruction** is a binary string which encodes the information that tells the CPU what to do. To be precise, the zeros and ones represent the electrical signals that travel through the CPU's circuit, resulting in the desire operation being performed. We don't need to pay too much attention to the specifics of the circuit to understand how to program, but it is important to have a general sense of what is happening.

To understand how the CPU interacts with the instructions and data in memory, we must first look at the general anatomy of a CPU.

The Anatomy of the CPU

All modern CPUs have the same general components:

1. A set of general purpose registers
2. An arithmetic and logic unit (ALU)
3. A set of special purpose registers

The **general purpose registers** are a small amount of memory located on the CPU chip. We can store any data that we like in the registers, which the CPU needs to do something with. The reason these registers exist instead of the CPU just using the data in memory directly is that the CPU can interact much more quickly with data in the registers than it can in memory. Unlike the memory, which contains a massive number of bytes, there are very few registers on a CPU (typically around 10, depending on the architecture).

The **arithmetic and logic unit (ALU)** is the circuits in the CPU that performs all of the operations which perform computations on the data. For example, when the CPU adds two integers together, or flips the bits of some binary value around, the ALU handles these operations.

The **special purpose registers** are similar to the general purpose registers, but as the name implies, each one is reserved for a special purpose, rather than to hold general data. The special purpose registers we will pay the most attention to are the:

1. Program Counter (PC)
2. Instruction Register (IR)
3. Processor Status Register (PSR)
4. Stack Pointer (SP)

The **program counter (PC)** keeps track of the address in memory where the next instruction will be loaded from. It is like a bookmark, so that when the CPU is

finished executing its current instruction, it knows where to look next for the next one.

The **instruction register (IR)** is where the actual machine instruction is loaded from memory. It holds the instruction while the rest of the CPU is executing it.

The **Processor Status Register (PSR)** contains several flags, which indicate the status of the previous instruction that was executed. A **flag** is a binary bit which indicates whether something has occurred. For example, there is a flag which indicates whether the output from the instruction is the value zero.

The stack pointer keeps track of the address of the byte in memory on the top of the stack segment of memory. Like the PC, this is a bookmark for when the CPU needs to push or pop to the stack.

CPU Instruction Cycle

Now that we are familiar with the components of the CPU, we can understand more precisely what happens inside of the CPU while it is running.

The first step in the CPUs cycle is loading the instruction from memory. The steps involved are:

1. The CPU references the PC to know where in memory its next instruction is located.
2. It moves the instruction from this location in memory to the IR onboard the CPU.
3. The PC is incremented to the address of the next instruction.

After this, the instruction that the CPU will execute is loaded into the instruction register. In the next step, the operands for the instruction are moved into place. For example, if the instruction is to add together two numbers which are stored in the registers, then the numbers are moved from the registers into the ALU. Every instruction is an operation, and most have one or more operands.

Finally, the last step is that the instruction is executed and the output is moved to its destination.

After the instruction is executed, the CPU starts again from the first step of the next instruction. This simple cycle is repeated indefinitely for the duration that the CPU is running, and that is how it operates.

32-Bit ARM CPU

A 32-bit ARM processor contains the following registers, which can be interacted with:

- 13 general purpose registers
- stack pointer (sp)
- link register (lr)
- program counter (pc)
- current processor status register (cspr)

The names of the thirteen general purpose registers are `r0`, `r1`, ... `r12`. The name of the stack pointer is `sp` but can also be referred to as `r13`. The same is true for the link register (`lr` or `r14`) and the program counter (`pc` or `r15`).

The special purpose registers `sp` and `lr` can be used as general purpose registers, but it is not recommended, since they each serve a purpose of their own. The CPU will look to the program counter whenever it needs to load the next instruction, so `pc` cannot be used as a general purpose register.

The program status register `cpsr` is unique in that it cannot be written to directly. The CPU updates the `cpsr` itself after each instruction to indicate the various flags that pertain to the output of that instruction.

The processor does contain an instruction register, but it is internal to the CPU and cannot be addressed by a user.

Assembly instructions

As previously discussed, the machine instructions are written in binary and stored in memory. The CPU uses the binary to determine how to pass the electricity through its circuits in order to carry out the desired operation.

While the binary instructions are natural for the CPU to understand and work with, the same is not true for a human programmer. For example, here is the binary string which represents the instruction to move the number 10 into register `r2`:

```
11100011101000000010000000001010
```

Clearly, the machine code is quite cryptic. In order to work with it directly, we would need to use reference tables to keep track of which binary strings do what, and that would mean that writing the code would be extremely slow and cumbersome. Additionally, since we can't easily look at the string and visually understand what it does, then it would be easy to make mistakes and hard to detect them.

An assembly language is a set of mnemonics for the instructions to make them easier for us to read/write and generally understand. Each assembly instruction corresponds directly to a machine instruction. For example, the instruction to move the number 10 into register r2, whose binary was given above, is:

```
mov r2, #10
```

As you can see, the assembly instruction is much easier to visually parse and understand. However, the assembly instruction is just an ASCII string and not the machine instruction itself. A program called an **assembler** is used to translate a program from assembly language into machine code.

Assembler Instruction Syntax

The syntax of assembler instructions are quite simple and include an:

1. operation code
2. destination register
3. one or two operands.

The operation code (opcode) describes what operation the instruction is encoding. The kinds of instructions that a CPU has falls into three categories:

1. Data movement operations
2. Arithmetic and Logic
3. Flow control

For most of the instructions, the destination register is listed immediately after the opcode in the assembler instruction. After that, zero, one or two more operands are listed, depending on the operation. Therefore, the general syntax of an instruction is

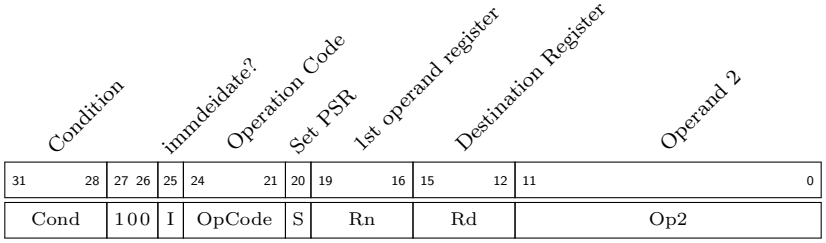
```
opcode rd, op2
```

with a destination register `rd` (where `d` here represents a number), and one input operand `op2`, which can be a register value, a memory location, or a numerical value, or

```
opcode rd, rn, op2
```

when there are two input operands, the first of which is a register `rn` and the second is `op2`.

Each machine instruction in a 32-bit ARM processor is 32 bits, and has a specific structure. As an example, most of the instructions that deal with data processing have the format:



If we put the machine code for the move instruction from above in this structure, we get:

Notice that the four bits in the destination register are `0010` which is 2, and the 12 bits in operand 2 is `000000001010`, which is the 12-bit binary number 10.

We largely don't need to know the details of the machine code in order to write in assembler, but it does help to understand that there is a general structure in each instruction, and that all of the information about the instruction is encoded into that structure.

For example, notice that there are only 12 bits for operand 2, which means we cannot use this to move numbers larger than what can be represented by 12 bits with only a single instruction.

Directives

In addition to instructions, an assembler program file also contains directives to the assembler, which indicate how the program should be constructed. We can use directives to allocate and initialize memory, identify certain labels to be shared globally, indicate which segments of memory are being written in the file and where the assembly file ends. All ARM directives start with a '.'

A **label** is an identifier placed at the beginning of a line in an assembler file to bookmark that line, so it can be referred to somewhere else in the program. For example, if you allocate some memory, then you may include a label so that you can reference that memory in your program. One important note is that a label is always a mnemonic for an address in memory. Use of labels is convenient, so that we don't have to think about specific addresses by value when writing code.

.global

The first directive that you are likely to encounter is the global declaration. Every program needs a starting point (e.g. the first line of `main()` is the starting point for any C program). The first instruction that an ARM program executes is located at a label named `_start`. In order to declare that your file contains the `_start` label to any other files which will be linked into the program, your file will begin with

```
1    .global _start
2
3 _start:
```

Notice that the `.global` directive does not begin on the first column of the line it is on. The only thing that should ever start on the first column is a label. Everything else, including instructions and directives should be indented.

A note about assembling the assembler file into an executable. If you use the programs `as` and `ld` to build the executable, then you must create the `_start` label yourself, because these assembler programs won't add anything else to your code. If you use `gcc` however, you should not define `_start` yourself, because `gcc` will add it and write its own instructions to set up the environment.

.text and .data

There are two segments of memory which are initialized when a program starts: the *text* segment and the *data* segment. The text segment contains all of the machine code, and the data segment contains any data variables with a global scope.

The `.text` directive indicates that the lines that follow it in the assembler file are instructions that should be loaded into the text section. Similarly, the `.data` directive indicates that the lines below it in the file are allocating memory in the data segment of the memory.

By default, the beginning of the file is assumed to be in the text segment, so you can omit the initial `.text` directive, if your assembly code it the first thing in the file.

We can use the `.text` and `.data` directives multiple times in a assembler file in order to alternate between writing instructions and allocating memory.

Allocating Memory in the Data segment

There are several directives to allocate memory, depending on the type of data to be stored in the memory, the size of the data and whether it is initialized or not.

Uninitialized Bytes

The simplest way to allocate memory is to simply allocate `n` bytes of unallocated memory. The directive for this is:

```
.skip n
```

where `n` is replaced by the number of bytes. Here is an example, where allocate 4 bytes of memory

```
1    .global _start:
2 _start
3
4    .data
5 x: .skip 4
```

Note the label `x` on line 5, which is a label that can be used to refer to the address of the allocated 4 bytes of memory in the program.

Initialized Integers

To allocate one byte of memory and initialize it with the number `m`, we can use

```
.byte m
```

where `m` is a 1-byte number. This is the equivalent of a `char` in C.

All of the directives for declaring initialized data have the same format (directive code and a single argument specifying the value which the memory should be initialized to).

For 1-byte integers,there are a couple of directive codes that can be used (they are all equivalent):

`.byte`

`.1byte`

For a 2-byte integer,the options for the directive code is

`.hword`

`.2byte`

`.short`

The directive codes for a 4-byte integer are:

`.word`

`.4byte`

`.int`

Alignment

Remember that a n-byte value stored in memory needs to be aligned so that its address is divisible by n. In order to make sure that our variables are properly aligned, we can use the directive

```
.align n
```

where the `n` here is a power of 2, to make sure that the next allocation has an address that is divisible by 2^n. For 2-byte integers, we would use `.align 1` to make sure that its address is divisible by 2, and similarly for 4-byte integers, we would use `.align 2`. Here is an example of using the directives:

```
1   .global _start
2 _start:
3
4   .data
5 x: .byte -15
6   .align 1
7 y: .hword 1000
8 z: .skip 1
9   .align 2
10 w: .word 500000
```

By default, the first allocation will be at an address divisible by 4. In the example above, that means we can be sure that the address `x` will be divisible by 4 (but since `x` is 1-byte, it doesn't matter anyway). However, `y` will be allocated right after `x`, and so without alignment in this case, we know that the address of `y` will not be divisible by 2, which is a problem. Therefore, the `.align 1` directive will move the allocation of `y` down to the next even address. Next we have `z`, which is the address of a single byte of uninitialized data. This will be at an even address, since it follows `y`, which is an even address and takes two bytes. However, this means that `w` will be an odd address (so not divisible by four). To ensure it is moved to the next address divisible by four, we use the `.align 2` directive.

Note that in the above example, we can allocate these in a smarter, so that we don't need as many alignments:

```
1   .global _start
2 _start:
3
4   .data
5 w: .word 500000
6 y: .hword 1000
7 x: .byte -15
8 z: .skip 1
```

However, this may not always be possible.

Instructions for Moving Data

The CPU spends a lot of its time moving data from one place to another. There are four ways to move data:

1. From one register to another
2. From immediate data to a register
3. Loading from memory to a register
4. Storing from a register to memory

mov instruction

To copy the data from register `rn` to register `rd`, we use the instruction

```
mov rd, rn
```

Notice again that the destination is the first register listed in the instructions (as is the case for most instructions).

The `mov` instruction can also be used to move an immediate value into the register. An **immediate value** is a constant literal typed directly into the assembler code. The syntax for moving an immediate number `m` into register `rd` is

```
mov rd, #m
```

where m is a number (in either decimal or hexadecimal).

An interesting question is, what is the smallest or largest number that can be moved with this instruction? Since the registers in a 32-bit ARM CPU are all 32 bits, it is tempting to conclude that we can move any 32-bit number with the instruction. However, recall that the instruction itself is 32 bits, and we observed earlier that the part of the instruction for the immediate value is only 12 bytes. Therefore, we can only directly move values in the range $-2^{11} \leq m \leq 2^{12} - 1$ or $-2048 \leq m \leq 4095$.

movw and movt

There is another related instruction, **movw**, which is only used to move immediate data, and so has 16-bits designated for immediate values. For example,

```
movw r0, #0xabcd
```

would successfully move the 16-bit value `0xabcd` into register `r0`.

In addition to `movw`, there is another move instruction **movt**, which also moves a 16-bit value, except it moves the value to the upper 2-bytes of the register. If the value in register `r0` is zero, and then we issue the following instruction

```
movt r0, 0xabcd
```

Then 32-bit value in `r0` after the instruction is `0xabcd0000`.

Additionally, `movt` will leave the lower 2 bytes in the register alone, so we can use this to move a 32-bit immediate value into a register using two commands, `movw` followed by `movt`. For example, suppose we want to put the number `0x1234abcd` into register `r1`. The following program will achieve that

```
1    .global _start
2
3 _start:
4    movw r1, #0xabcd
5    movt r1, #0x1234
```

However, the order of the instructions is important. The `movt` instruction will not clear the lower 16 bits in the register, but `movw` will clear the upper 16 bits.

The movw/movt combination is clearly quite useful for moving a 32-bit immediate value into a register, when the value is specified in hexadecimal, since it is easy to split the number into two halves, but what about a number specified in decimal. How do we move the immediate value 1,000,000 into a register?

Once way is to translate the value into hexadecimal ourselves, which for 1,000,000 would be `0xf4240`. However, this requires us to perform manual calculations outside of the program, which isn't ideal.

There is a pair of tags that we can add to an immediate value to get what we want. The tags are `:lower16:` and `:upper16:` to get the lower 16 bits and upper 16 bits of a number, respectively. Using this, we can move the value 1,000,000 to the register `r0` as follows:

```
1    .global _start:
2
3  _start:
4    movw r0, #:lower16:1000000
5    movt r0, #:upper16:1000000
```

Moving Addresses

In order to move data to or from memory, we first need to put the address of the desired memory location in a register. Addresses are 32-bit numbers, and a label which represents an address can be treated just like an immediate value. Consider the following code:

```
1    .global _start:
2
3  _start:
4    movw r0, #:lower16:x
5    movt r0, #:upper16:x
6
7    .data
8  x: .word 500
```

In this example, there is a 4-byte integer initialized with the value 500, and x is the label referencing the address of this memory. That means that x is a 32-bit number representing the address. We can use the movw/movt instructions to move that address using the reference to `x`, and when the code is assembled and executed, the x will be replaced with the actual value of the address.

ldr instruction

The instruction which loads values from memory into a register is the `ldr` instruction. The syntax for the instruction is

```
ldr rd, [rm]
```

where `rd` is the register that the value will be loaded into, and `rm` contains the address of the data in memory to be loaded.

The brackets are a specific syntax choice that let us know we will be using the value in the register to reference memory.

From our previous example, if we want to load the value 500 stored in memory at x, we can use the ldr instruction to do that:

```
1    .global _start:
2
3  _start:
4    movw r0, #:lower16:x
5    movt r0, #:upper16:x
6    ldr r1, [r0]
7
8    .data
9  x: .word 500
```

After the movw/movt instructions, r0 contains the address x. Then after that, `[r0]` is a reference to the memory location at x, which contains 500. The `ldr` instruction then moves the value 500 from memory at x to register r1.

Suffixes and Data Sizes

The `ldr` instruction above always moves four bytes from memory into the register. In the previous example, this is exactly what we wanted because x is a 4-byte integer. However, what if we want to move a 2-byte or 1-byte integer instead?

The `ldr` instruction has a few suffixes, in order to select the size, and whether the integer is signed or not. The opcodes are

`ldr`	4-byte (word) value
`ldrh`	2-byte (halfword) unsigned value
`ldrsh`	2-byte (halfword) signed value
`ldrb`	1-byte unsigned value
`ldrsb`	1-byte signed values

The reason that the 2-byte and 1-byte values have a signed and unsigned value is because the value loaded from memory will be extended to fill the entire 32-bit register. Depending on whether the value is negative or positive, the leading bits in the register will be filled with 1's or 0's, respectively. For example, the signed 1-byte value `0xff` is -1, and the extension of this value to 32 bits is `0xffffffff`. However, the unsigned value of `0xff` is 255, and its extension to a 32 bit number is `0x000000ff`.

str instruction

To store a value from a register in memory, we use the `str` instruction with the syntax

```
str rn, [rm]
```

where `rn` contains the value we want to store, and `rm` contains the address is memory where we want to store the value. Notice here that this is a rare ARM instruction in which the destination is not the first operand specified.

Like the `ldr` instruction, the data size of the `str` instruction is four bytes. The instructions to move a 2-byte or 1-byte value is

`str`	4-byte (word) value
`strh`	2-byte (halfword) value
`strb`	1-byte value

The signs don't matter when storing values, because the value won't be extended.

An Example from C

Using the instructions from this section to move data around, we can write an assembler program equivalent to the following C program

```
1  char c = 40;
2  short s;
3
4  int main(){
5      s = c;
6      c = -120;
7  }
```

The assembly for this code is

```
    .global _start

_start:
    @ load c into r1
    movw r0, #:lower16:c
    movt r0, #:upper16:c
    ldrsb r1, [r0]

    @ store the value from r1 in s
    movw r2, #:lower16:s
    movt r2, #:upper16:s
    strh r1, [r2]

    @ mov -120 into r1
    mov r1, #-120

    @ store value from r1 in c
    strb r1, [r0]


    .data
c: .byte 40
    .align 1
s: .skip 2
```

Note that the comments in this code example begin with the @ symbol.

Also, we can start to appreciate the ease of use of the high-level languages, like C. The simple program with two statements in C, takes 8 assembly instructions. This is due to the convenience that we don't have to think about how the statement is carried out in the high-level language. We only need to know the end result.

Arithmetic Instructions

The ALU for an ARM processor contains arithmetic instructions for addition, subtraction and multiplication. Notably, it does not contain an instruction for division. Instead, we will need to implement division using an algorithm (for example the division algorithm).

add instruction

The `add` instruction is used to add the values in two registers and store the result in another register. The syntax is

```
add rd, rn, op2
```

and the result is that the sum `rn + op2` will be stored in `rd`. The same register can be used for the operands and the destination. For example

```
add r0, r0, r1
```

will add registers `r0` and `r1` together, and then put the sum in register `r0`.

Just like with the `mov` instruction, the `op2` can also be a 12-bit immediate value, which can be useful if we want to add a small value to a register value. For example,

```
add r0, r0, #1
```

will increment the value of the register `r0` by 1. In fact, incrementing register values is often a very useful thing to do, so there is a pseudo-instruction to make it a bit easier

```
add r0, #1
```

Notice that it looks simpler and more direct to its purpose. However, if you assemble and look at the resulting instruction, it expands to the actual instruction above that does the same thing.

sub instruction

Similar to addition, the subtraction instruction `sub` performs subtraction of the values in the registers.

```
sub rd, rn, op2
```

The instruction will take the difference `rn - op2` and place it in register `rd`. The second operation can be a register, or a 12-bit immediate value as well, and there is a pseudo instruction

```
sub rd, #imm
```

which will decrement the value in `rd` by the immediate value `imm`. The biggest difference between addition and subtraction is that subtraction is not commutative (it is not the same if we reverse the order of the operands).

rsb instruction

The rsb instruction performs subtraction with the reverse order of the operations. The syntax is

```
rsb rd, rn, op2
```

and the instruction will put the difference `op2 - rn` into register `rd`. This can be especially useful when `op2` is an immediate value.

One interesting example is

```
rsb r0, r0, #0
```

which will place the value `#0 - r0` into `r0`. If we look a little closer, we can interpret what it does as negating the value in the register. This is another function that is so useful, there is a pseudo-instruction for it:

```
neg rd, rn
```

which negates the value in `rn` and places the value in `rd`. Again, if you assemble this instruction and look at the result, you will see it expanded as

```
rsb rd, rn, #0
```

mul instruction

The `mul` performs multiplication, but the instruction is more limited than the addition and subtraction operations. Recall that the product of two n-bit integers is a $2n$-bit integer. If we multiply two 32-bit numbers together, then the result may need up to 64-bits to store.

The syntax for the `mul` instruction is

```
mul rd, rn, rm
```

where the values in registers `rn` and `rm` are multiplied together, and the lower 32 bits of the result is stored in register `rd`. If the product requires more than 32 bits to store, the upper bits are just discarded (so the value will be incorrect). There are

other instructions to multiply and store the 64 bit product in two separate registers, but we will limit our focus to the simple instruction.

Another limitation of the instruction is that both operands must be registers; we cannot multiply a register with an immediate value with this instruction. Additionally, the destination register `rd` cannot be the same as `rn`.

Flow Control

The normal flow of assembler code, without interruption, is sequential. The CPU begins execution with the first instruction at the `_start` label and moves from one instruction to the next until the end of the program.

There are several ways that the normal flow of code is interrupted, including if-else statements, loops and function calls. These constructs may skip over or repeat blocks of code. Calling functions is more involved than conditional statements and loops because information gets passed back and forth between the calling function and the called function, so we will explore that on its own later and focus on conditional statements and loops in this section.

Recall that the program counter is responsible for keeping track of the address in memory of the next instruction to load and execute. In order to jump to another location in the code, all we need to do is change the value in the program counter.

The break instruction is used to update the program counter, and has syntax:

```
b label
```

where `label` is located at an instruction.

```
1    .global main
2
3  main:
4    mov r0, #5
5  L1:
6    add r0, #1
7    b L1
```

In the above example, the label `L1` has the address of the instruction

```
add r0, #1
```

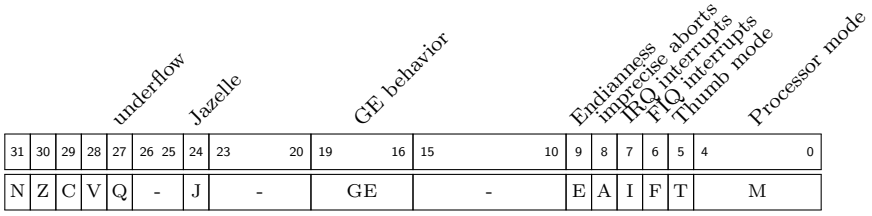
so the break instruction on line 7, will load this instruction's address into the program counter. As a result, the next instruction that the CPU will load is this add instruction. As a result, the code will add 1 to register `r0` repeatedly in a loop. This is an example of how we can use the break instruction to control which instruction gets executed next.

PSR Flags

The processor status register (psr) contains flags that indicate several properties of the previous instruction that was executed. There are four flags, called the **condition flags**, that we will pay attention to, with names `N`, `Z`, `C`, and `V`.

N	negative
Z	zero
C	carry
V	overflow

The flags are located in the highest four bits of the `cpsr` register in an ARM processor, in the order specified. The full content of the `cpsr` register is described in the following diagram.



We will only pay attention to the condition flags.

The `N` flag will be set for any instruction if the result of the instruction is interpreted as negative. For example, if `r0` contain the value 5, then the instruction

```
sub r0, #10
```

will subtract 10 from the value of `r0`, changing it to -5, and as a result the `N` flag will be set.

Similarly, the `Z` flag is set when the result of the instruction is 0. In the previous example, if `r0` contains 5, the instruction

```
sub r0, #5
```

will change the value of `r0` to zero and hence the `Z` flag will be set.

The carry flag `C` will be set when the instruction results in a 33rd bit which is carried outside of the register. An example of this is if the value in register `r0` is -1, and we add 1, with the instruction

```
add r0, #1
```

Recall that the number -1 in 32-bit two's complement binary is represented with 32 bits which are all 1. When we add 1 to this, the result in 32-bit binary is `0x00000000` with a 1 carried into the 32nd bit. This will result in the `C` flag being set.

Finally, the overflow `V` flag is set when the instruction results in an overflow of the value. An example of overflow is if we have a C variable

```
char x = 127;
```

and then we add 1 to `c`. The value of the signed char variable `x` will be -128, since the value has gone out of the range of the variables (or overflowed). In assembly, here is an example of a program, where the addition will cause the overflow bit to be set

```
1   .global main
2
3   main:
4       movw r0, #0xffff
5       movt r0, #0x7fff
6       @ the value of r0 is now 0x7fffffff
7       add r0, #1
```

Overflow occurs because `0x7fffffff` is the largest 4-byte signed integer.

Conditional Breaks

The break instruction has several suffixes that can be appended in order to condition the execution of the break to only when the condition is satisfied. We will make use of six of these suffixes.

`eq`	equal
`ne`	not equal
`lt`	less than
`le`	less than or equal to
`gt`	greater than
`ge`	greater than or equal to

All of these are comparing the result of the previous instruction to zero, so for example `ge` means "greater than or equal to zero". For example, suppose that the register `r0` contains the number 5, and then the following lines of assembler execute

```
sub r0, #10
blt L1
```

The break to `L1` will occur because the subtraction on the previous instruction resulted in a -5, which is less than zero. However, if the `bge` were used instead, it would not have been executed, since -5 is not greater than or equal to zero.

We can use this to implement the conditions in `if-else` statements or loops, which are generated by a comparison. For example, consider the if statement

```
1    if (x < 7){}
```

Here, we are comparing x to 7, but we can convert this into a comparison with zero by moving everything to one side:

$$x < 7$$
$$x - 7 < 0$$

Then we can implement this in assembly by moving the value of x into a register and subtracting 7 from it.

The condition flags are used for each comparison as follows:

`eq`	`Z` is set
`ne`	`Z` is not set
`lt`	`N` is set
`le`	`N` or `Z` is set
`gt`	`N` and `Z` are not set
`ge`	`N` is not set

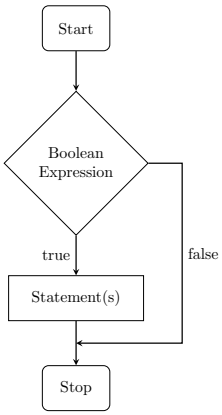
When performing a subtraction like this for the comparison, we arne't interested in storing the value of the subtraction anywhere; we only want to set the comparison flags. There is a specialized instruction called **compare (cmp)** for this purpose, with syntax

```
cmp rn, op2
```

which performs the subtraction `rn - op2` but only sets the condition flags.

If Statements

Now we should consider how to use this to implement the logic of a conditional expression. The basic flow of logic is that there is a block of code that we will run if a certain condition is true. If it is false, then we will jump over the code to the code that comes after the block.



The diagram to the right shows the logic, where the boolean expression is the condition to check. Notice that if the condition is true, then the code does not actually jump anywhere, it continues sequentially to the first statement in the block of statements.

In the example from the previous section, where the condition on the if-statement is `x < 7`, we would perform a jump when the condition is not met, which happens exactly when `x >= 7`, so we would use the `bge` break to conditionally jump over the block of code when the condition is false.

Now let's consider a full example. In C, suppose we have the code

```
1     if (x < 7){
2         x++;
3     }
4     y = 10;
```

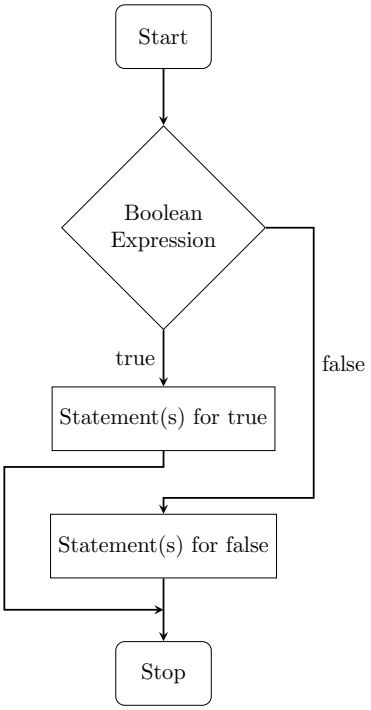
If the value of `x` is less than 7, then it will increment. If `x` is greater than or equal to 7, then it will not. In both cases, `y` will be set to 10 afterwards.

The assembler code for an equivalent program (assuming the variables `x` and `y` are 4-byte integers stored at locations with labels of the same names) is

```
1     movw r0, #:lower16:x
2     movt r0, #:upper16:x
3     ldr r1, [r0] @ r1 = val of x
4
5     cmp r1, 7
6     bge L1
7
8     add r1, #1 @ x++
9     str r1, [r0]
10
11 L1:
12    movw r1, #:lower16:y
13    movt r1, #:upper16:y
14    mov r2, #10
15    str r2, [r1] @ y = 10
```

Line 5 performs the subtraction `x-7` to set the comparison flags, and line 6 follows through by executing a conditional break if $x - 7 \geq 0$ (hence $x \geq 7$). In this case, we will jump past lines 8-9, which increment `x`.

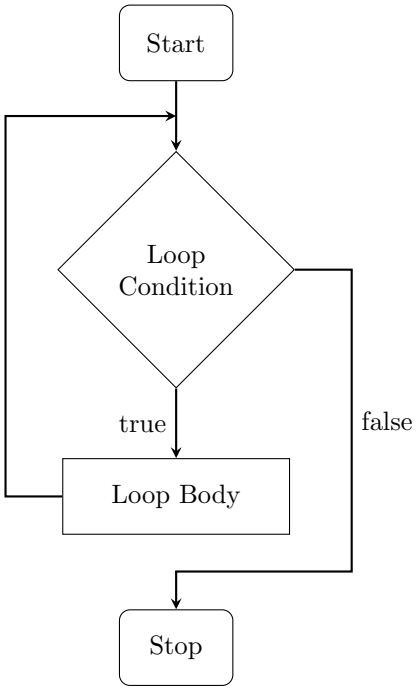
If-Else Statements



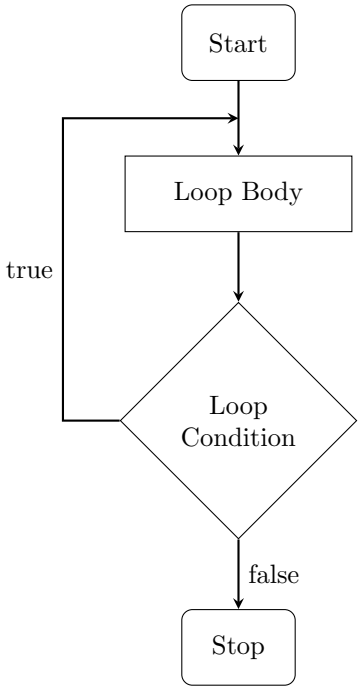
The structure of the code for an if-else statement starts with the boolean expression for the condition check. Next, the block of code for when the condition is true, followed by the block of code for when the condition is false. Finally, you have the rest of the code beyond the if-else statement.

Notice that, after the block of code for when the condition is true, there is an unconditional jump, because we don't want the code to also execute the block of code for when the condition is false. We want to run one block, or the other, but not both, depending on the condition.

While Loops



Do-While Loops



For Loops

Syntax:

```
for(initial action; loop condition; action after iteration)
```