Chapter 1: Intro, What is an Assembly Language?

This tutorial is designed to teach an absolute Beginner about the *ARMv8 AArch64* Assembly Language. It is assumed that said Beginner knows absolutely nothing about Assembly Languages in general. This is a pure Beginner-to-Pro tutorial.

Your desktop/laptop/video-game-console/etc is controlled by a **CPU** (Central Processing Unit). A CPU doesn't understand any Human Language, but a CPU can understand two things. **0**'s and **1**'s. 0 meaning no voltage. 1 meaning voltage. These 0's and 1's are what is known as *Binary numbers*, which will be later explained in next Chapter. A CPU will perform tasks based on receiving fixed or variable length blocks that are made up of certain combinations of 0's and 1's. These blocks are known as CPU Instructions. It goes without saying that attempting to make a program writing in blocks of 0's and 1's is insanity. Therefore, we use what is called ***Assembly Language***.

Assembly Language is a human readable form of these blocks of 0's and 1's (CPU instructions). We can write out Assembly Language in a program to have the CPU execute instructions. Therefore Assembly Language is a Computer Programming Language. A tool called an *Assembler* will take in a source text file written in a CPU's Assembly language, and will translate it into a hexadecimal representation (more on Hex in Chapter 2) of the CPU's Instructions (blocks of 0's and 1's). The CPU instructions will be written to an executable file that the CPU can launch and run it as program.

As CPU's get more advanced, they will use different/improved Assembly Languages. Some group or production line of CPUs may use the same Assembly Language. Or a new line of CPUs can use the same Assembly Language as a previous line but with subtle differences for improvements. Either way, understand that if two different CPUs use a different Assembly Language, they **cannot** run each other's instructions.

This problem is solved by "higher level" Computer Programming Languages, such as **C**, **C++**, etc. A tool called a *Compiler* takes in a source text file written in C/C++/etc, and translates it to a desired Assembly Language (which is changed to the actual CPU instructions) so the targeted/desired CPU can run the program. Therefore, this allows the Developer to not have to learn 100's of Assembly Languages, he/she can just learn one of the higher level languages.

With that being said, there are still some use cases where Assembly is necessary to learn. Such as...

- Writing out the Boot/Reset Sequence for a CPU
- Writing out CPU-specific specialized tasks
- Further enhance performance
- Understanding a specific CPU as much as possible
- Exploits/Hacks for a specific CPU (such as for Video Game Consoles)
- A better Understanding of "under the hood" stuff (such as Memory Management, Cache, etc)

Chapter 2: Basic Vocabulary, Data/Number Types

As mentioned earlier, the **0**'s and **1**'s that a CPU can understand are called *Binary Numbers*. Binary Numbers are the simplest/lowest number form. A Binary value is also known as a **Bit**. Bits can only be 0 or 1, nothing else. Before you can understand Binary, you will need to learn Hexadecimal. First off, regular Decimal is something you already know. It's the basic numbers that everyone uses on a regular basis. Such as 3, 16, 2057, 5168430, etc. The regular Decimal number system uses what is called Base 10. You start at 0 and go to 9. After 9, you must start a new "base of 10". This new base starts at 10 and goes thru 19. After 19, you have to start another new base of 10 at the number 20..

Hexadecimal uses a Base **16** system. Similar to decimal, you start with 0. However, once you get to 9, you proceed to **A**, the first letter of the English Alphabet. You keep proceeding through the Alphabet til you hit **F**. 0 thru F are 16 total numbers/values. This is the first "base of 16" After F, you can then go to 10. 10 thru 1F is the next "Base of 16". After that would be 20 thru 2F, etc etc.

Here's a basic decimal to hex conversion chart~

Decimal	Hex
0	0
1	1
..	..

Decimal	Hex
9	9
10	A
11	B
12	C
..	..
15	F
16	10
17	11
..	..
31	1F
32	20
33	21
..	..
159	99
160	A0
161	A1
..	..
255	FF
256	100
257	101

Once numbers become pretty large, trying to manually convert Decimal to Hex is silly. Instead, here's a Decimal-to-Hex converter - > HERE

You can also use it to convert in the opposite fashion. The knowledge of Hex numbers is required because many values of a CPU are usually shown in Hex form within a debugging tool. Also, you will need to learn Binary which Hex is required beforehand.

Speaking of Binary, we can now dive into it!

Every Hexadecimal digit can be represented by 4 consecutive binary values (bits). Like this...

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110

Hex	Binary
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

As you can see, the chart is pretty easy to remember. Trying to convert Decimal to Binary is more difficult. Now what about going beyond the binary value 1111? Simple, like this..

Hex = Binary
10 = **0001 0000**
11 = **0001 0001**
12 = **0001 0010**
.. ..
19 = **0001 1001**
1A = **0001 1010**
.. ..
1F = **0001 1111**
20 = **0010 0000**

I've separated the binary values via pairs of four bits, since every hex digit can be converted to its 4-digit Binary value. The first 4-bit pair is **blue** in color. The second pair is **violet** in color. Binary is crucial in Assembly for what is known as Logical Operations (a family/type of CPU instructions). Logical Operations are preformed on a bit by bit basis. But let's not get ahead of ourselves here, we still have a lot of other Basics to cover before going into something such as Logical Operations.

One Final Note about Hex:
Hex values present in an Assembly Source file are always designated via a "0x". For example, the Hex Value BC needs to be written as **0x**BC.

We got Hex done, check. Binary done, check. Now let's move on into some key essential Vocabulary every Assembly Developer must know. For starters, the term "null" simply means zero. You will come across that term on a frequent basis in this tutorial. For Assembly Language, special terms are used to describe certain lengths of Binary/Bit values. Here's the list...

8 Bits = **Byte** (2 Hex digits) Example: 0x44
16 Bits = **Halfword** (4 Hex digits) Example: 0xB0C8
32 Bits = **Word** (8 Hex digits) Example: 0xDEDD0020
64 Bits = **Double-Word** (16 Hex digits)
128 Bits = **Quadword** (32 Hex digits)

We need to discuss some common terminology regarding portions of values within values. For example, lets say we have the following value...

0x8045CD0A

The "8045" portion (lefthand 16-bits) of the word value is known as the **upper** 16-bits. The "CD0A" portion (righthand 16-bits) is known as the **lower** 16-bits. This same concept can be applied to any value length. For example, we have the following double-word value...

0xFFFFFFFF80007774

The "FFFFFFFF" portion is known as the upper 32-bits while the "80007774" portion is known as the lower 32-bits. What's important is that you understand the lefthand = upper, and righthand = lower.

Final Chapter NOTE: ARMv8 AArch64 Instruction Length
ARMv8 AArch64 is an Assembly language that uses the same sized Bit blocks for all of its instructions. **All** instructions are **32**-bits (word) in length.

Chapter 3: Navigating through Memory

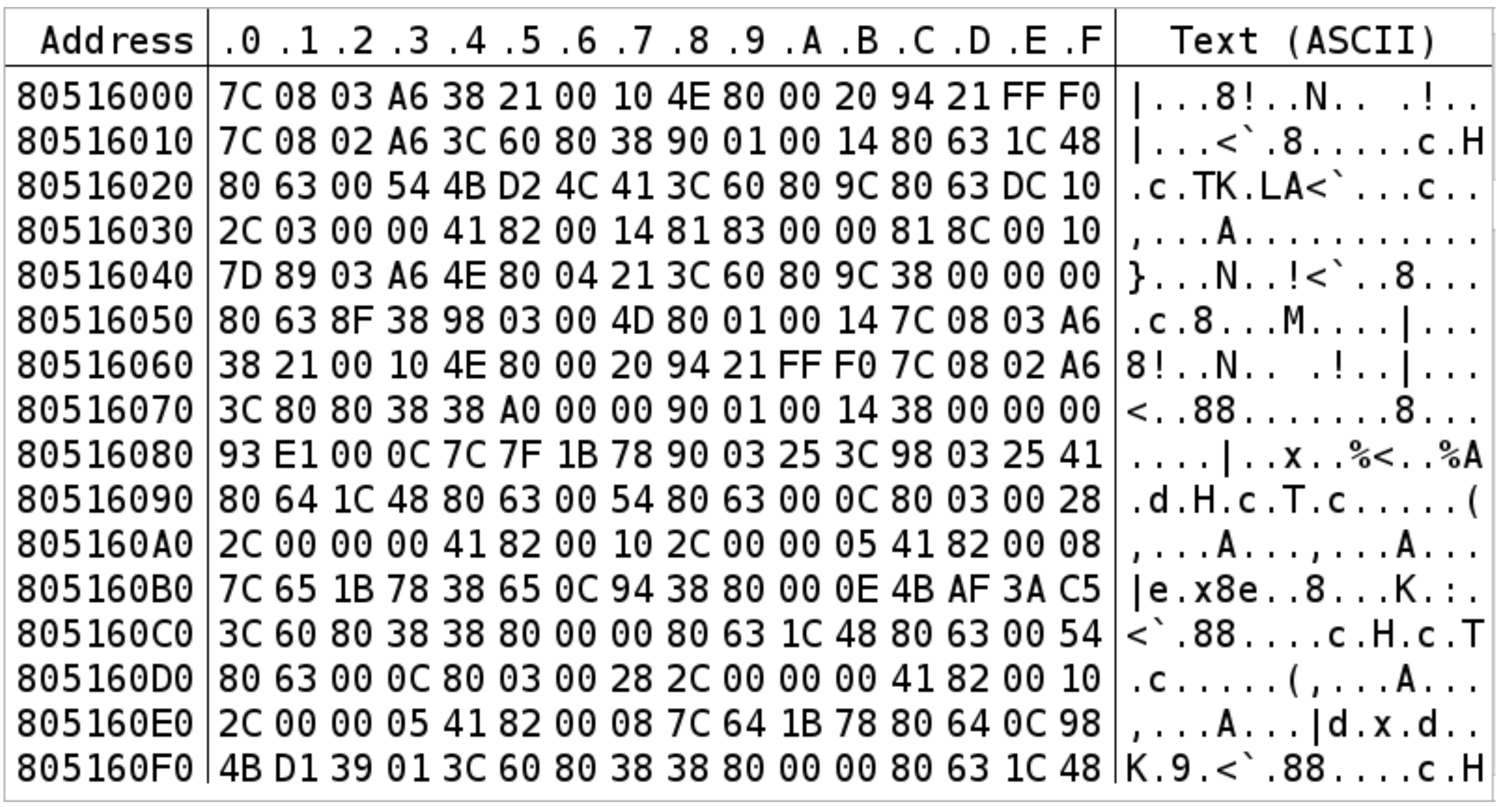
Memory memory memory. Also known as *RAM*. A CPU's instructions will reside in Memory. Also, any data that the CPU needs to read/write to will also reside in Memory. Memory is usually broken up into segments/regions depending on certain intended purpose(s). The region of Memory where a CPU's instructions reside at is usually called **Static** or **Main** memory. It's named *Static* because every time a specific program is executed by the CPU, the CPU instructions are usually placed at the same locations within Static Memory every single time.

The region of Memory where Data is kept at is usually called **Dynamic** or **Heap** memory. It's called *Dynamic* because every time a specific program is executed/launched by the CPU, the Data may **not** be at the same location it was present at the time before. There are of course exceptions to how Static & Dynamic Memory operate. I am speaking in general terms.

Locations in Memory are called Memory Addresses. Memory Addresses are usually a fixed length when referenced or used by a CPU. The determined fixed length is dependent on the CPU itself and some special properties of the CPU that are determined/written by a program shortly after said CPU has booted or been reset.

Example 32-bit Memory Address:
0x80516074

Understanding values located in Memory:
Data residing within Memory (using some sort of Memory Viewer debugging tool/device) is usually broken up into groups of bytes. There are usually 16 bytes per row.

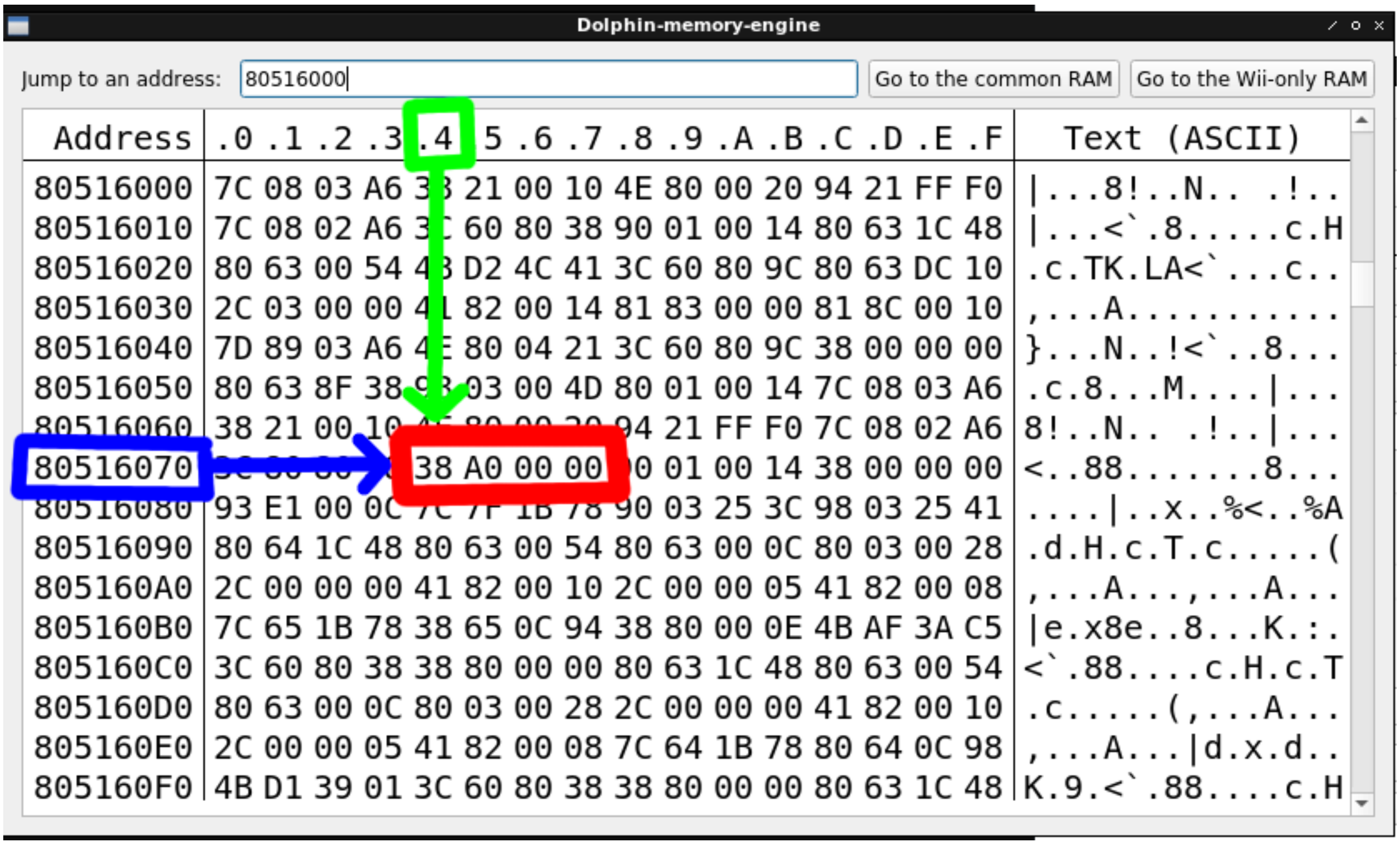


```
 Address  .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F   Text (ASCII)
80516000  7C 08 03 A6 38 21 00 10 4E 80 00 20 94 21 FF F0  |...8!..N.. .!..
80516010  7C 08 02 A6 3C 60 80 38 90 01 00 14 80 63 1C 48  |...<`.8.....c.H
80516020  80 63 00 54 4B D2 4C 41 3C 60 80 9C 80 63 DC 10  .c.TK.LA<`...c..
80516030  2C 03 00 00 41 82 00 14 81 83 00 00 81 8C 00 10  ,...A...........
80516040  7D 89 03 A6 4E 80 04 21 3C 60 80 9C 38 00 00 00  }...N..!<`..8...
80516050  80 63 8F 38 98 03 00 4D 80 01 00 14 7C 08 03 A6  .c.8...M....|...
80516060  38 21 00 10 4E 80 00 20 94 21 FF F0 7C 08 02 A6  8!..N.. .!..|...
80516070  3C 80 80 38 38 A0 00 00 90 01 00 14 38 00 00 00  <..88.......8...
80516080  93 E1 00 0C 7C 7F 1B 78 90 03 25 3C 98 03 25 41  ....|..x..%<..%A
80516090  80 64 1C 48 80 63 00 54 80 63 00 0C 80 03 00 28  .d.H.c.T.c.....(
805160A0  2C 00 00 00 41 82 00 10 2C 00 00 05 41 82 00 08  ,...A...,...A...
805160B0  7C 65 1B 78 38 65 0C 94 38 80 00 0E 4B AF 3A C5  |e.x8e..8...K.:.
805160C0  3C 60 80 38 38 80 00 00 80 63 1C 48 80 63 00 54  <`.88....c.H.c.T
805160D0  80 63 00 0C 80 03 00 28 2C 00 00 00 41 82 00 10  .c.....(,...A...
805160E0  2C 00 00 05 41 82 00 08 7C 64 1B 78 80 64 0C 98  ,...A...|d.x.d..
805160F0  4B D1 39 01 3C 60 80 38 38 80 00 00 80 63 1C 48  K.9.<`.88....c.H
```

The above is a random pic of Memory of a random Program. (Side Note: This picture is from a PowerPC program, not ARM. However we can still use it for teaching you how to navigate thru Memory as the concept of navigation is the same for any CPU)

Most Memory Viewers within Debugging Tools will always display the data in Hex form. Since Hex is a Base-16 number system, this is the reason as to why each row is 16 bytes. Memory Viewers are very helpful and give you a good visual input of what the CPU is 'looking' at. One of the toughest aspects for Beginners is being able to "Visualize" Code. If you can "visualize" it, you can write code for it.

An ability that every Assembler Programmer must possess is being able to navigate thru Memory. Let's go over the example Memory Address of **0x80516074**. To navigate to this Address, you first want to look at the Address values on the left hand side of the picture below. You will see the Address for each row increments by 0x10 (16). You will first go down to Address 0x80516070. Now you simply go over to the right by 4 to end up at 0x80516074.

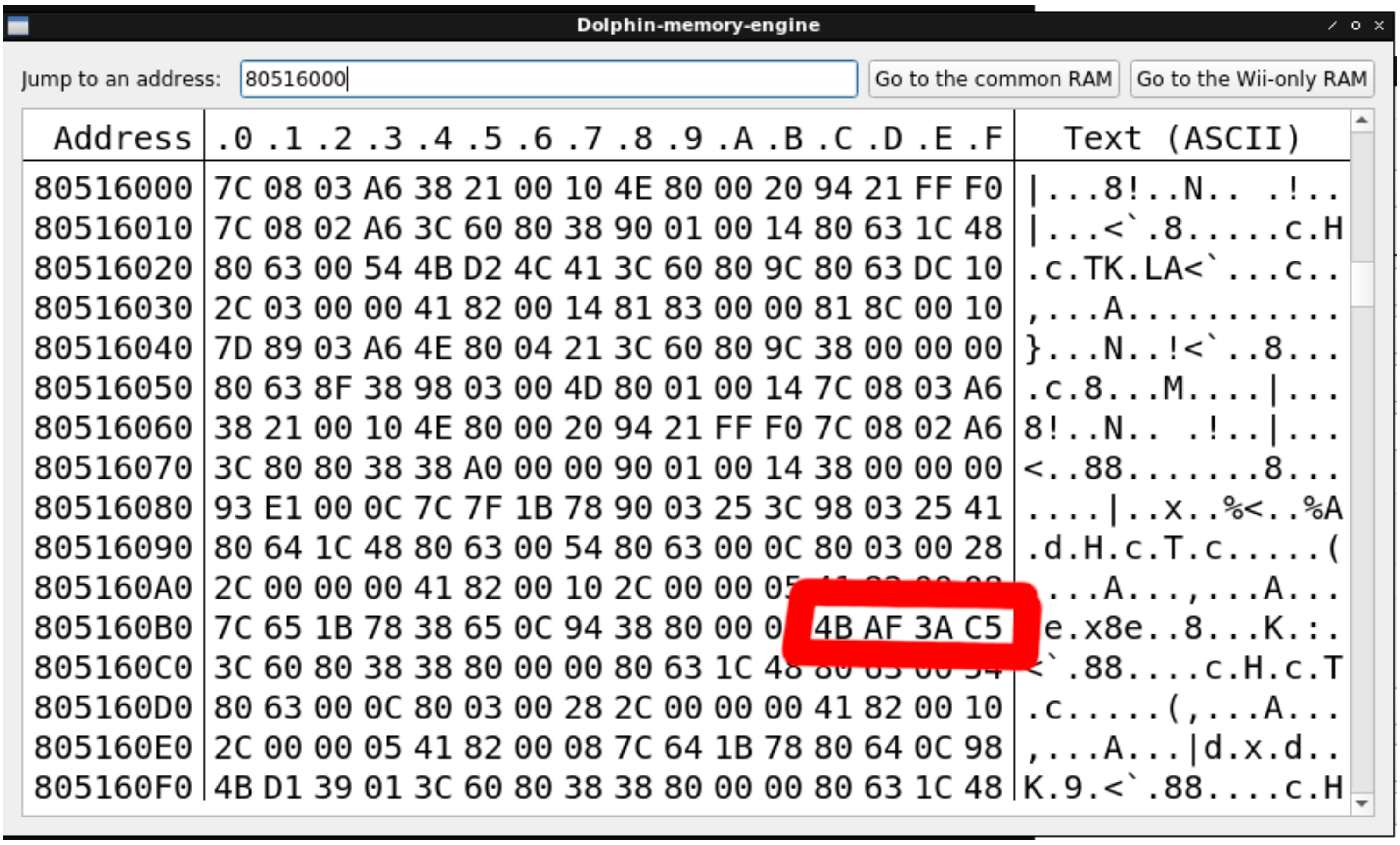


```
80516000

 Address  .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F  Text (ASCII)
80516000  7C 08 03 A6 38 21 00 10 4E 80 00 20 94 21 FF F0  |...8!..N.. .!..
80516010  7C 08 02 A6 3C 60 80 38 90 01 00 14 80 63 1C 48  |...<`.8.....c.H
80516020  80 63 00 54 4B D2 4C 41 3C 60 80 9C 80 63 DC 10  .c.TK.LA<`...c..
80516030  2C 03 00 00 41 82 00 14 81 83 00 00 81 8C 00 10  ,...A...........
80516040  7D 89 03 A6 4E 80 04 21 3C 60 80 9C 38 00 00 00  }...N..!<`..8...
80516050  80 63 8F 38 98 03 00 4D 80 01 00 14 7C 08 03 A6  .c.8...M....|...
80516060  38 21 00 10 4E 80 00 20 94 21 FF F0 7C 08 02 A6  8!..N.. .!..|...
80516070  3C 00 00 38 38 A0 00 00 90 01 00 14 38 00 00 00  <..88.......8...
80516080  93 E1 00 0C 7C 7F 1B 78 90 03 25 3C 98 03 25 41  ....|..x..%<..%A
80516090  80 64 1C 48 80 63 00 54 80 63 00 0C 80 03 00 28  .d.H.c.T.c.....(
805160A0  2C 00 00 00 41 82 00 10 2C 00 00 05 41 82 00 08  ,...A...,...A...
805160B0  7C 65 1B 78 38 65 0C 94 38 80 00 0E 4B AF 3A C5  |e.x8e..8...K.:.
805160C0  3C 60 80 38 38 80 00 00 80 63 1C 48 80 63 00 54  <`.88....c.H.c.T
805160D0  80 63 00 0C 80 03 00 28 2C 00 00 00 41 82 00 10  .c.....(,...A...
805160E0  2C 00 00 05 41 82 00 08 7C 64 1B 78 80 64 0C 98  ,...A...|d.x.d..
805160F0  4B D1 39 01 3C 60 80 38 38 80 00 00 80 63 1C 48  K.9.<`.88....c.H
```

As you can see, the **blue** boxed address shows how to navigate to the example address using the numbered address rows. The **green** boxed number represents the column to navigate to which is number 4. A simple equation of the address row value + the column number will navigate to the desired memory address. (0x80516070 + 0x4 = **0x80516074**).

The hex number outlined in **red** shows the first 4 bytes (*word*) that is located starting at Memory Address 0x80516074. Thus, you can state that the word at Address 0x80516074 is 0x38A00000.

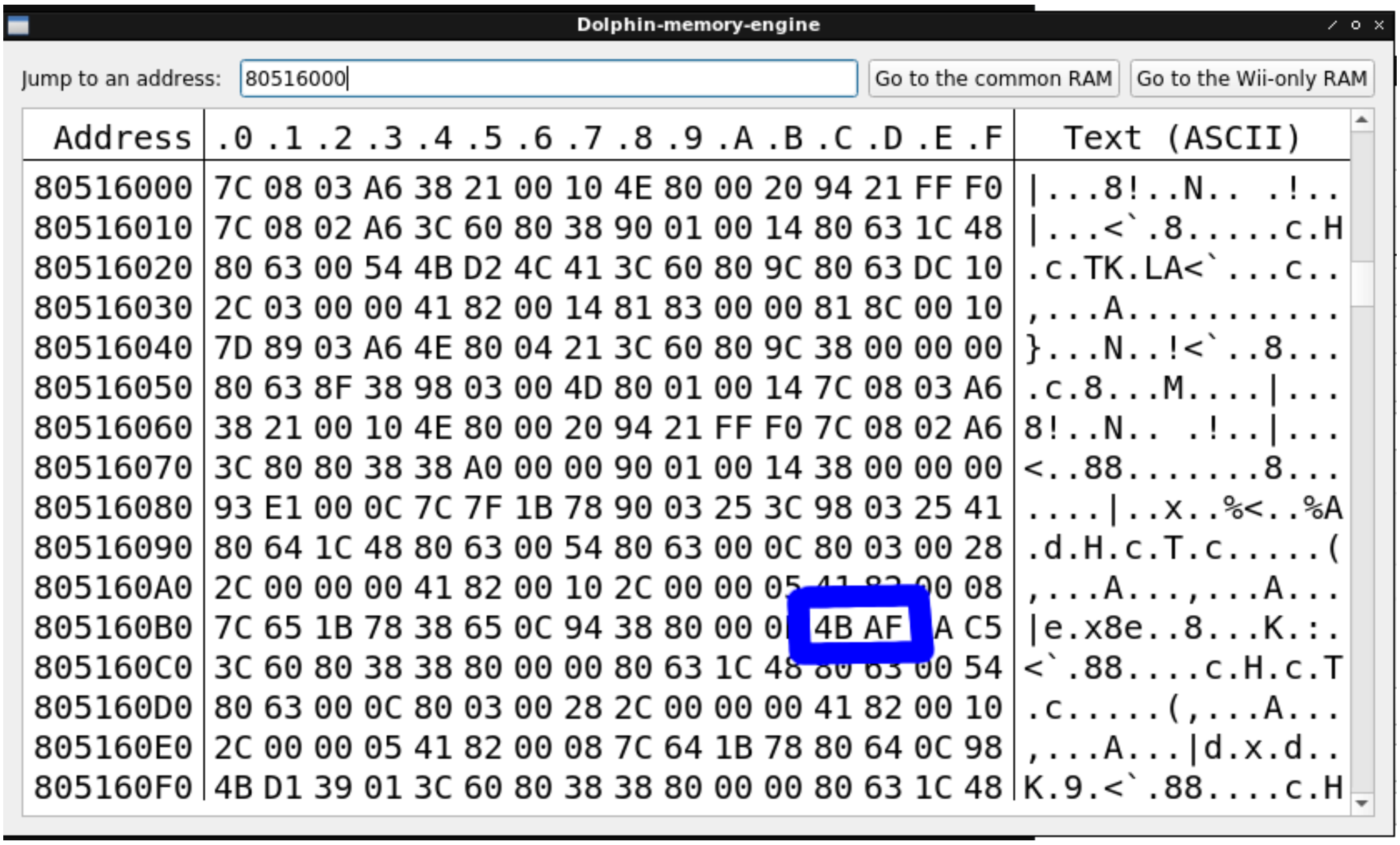
Reiterating the point here, you HAVE to understand how to navigate thru Memory. Let's go over some more examples. Using the picture of Memory that has already been shown, go to Memory Address **0x805160BC**. The Word at this address is **0x4BAF3AC5**. The picture below shows the Word outlined in **red**.


Dolphin-memory-engine

Jump to an address: 80516000 | Go to the common RAM | Go to the Wii-only RAM

```
Address  .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F  Text (ASCII)
80516000 7C 08 03 A6 38 21 00 10 4E 80 00 20 94 21 FF F0 |...8!..N.. .!..
80516010 7C 08 02 A6 3C 60 80 38 90 01 00 14 80 63 1C 48 |...<`.8.....c.H
80516020 80 63 00 54 4B D2 4C 41 3C 60 80 9C 80 63 DC 10 .c.TK.LA<`...c..
80516030 2C 03 00 00 41 82 00 14 81 83 00 00 81 8C 00 10 ,...A...........
80516040 7D 89 03 A6 4E 80 04 21 3C 60 80 9C 38 00 00 00 }...N..!<`..8...
80516050 80 63 8F 38 98 03 00 4D 80 01 00 14 7C 08 03 A6 .c.8...M....|...
80516060 38 21 00 10 4E 80 00 20 94 21 FF F0 7C 08 02 A6 8!..N.. .!..|...
80516070 3C 80 80 38 38 A0 00 00 90 01 00 14 38 00 00 00 <..88.......8...
80516080 93 E1 00 0C 7C 7F 1B 78 90 03 25 3C 98 03 25 41 ....|..x..%<..%A
80516090 80 64 1C 48 80 63 00 54 80 63 00 0C 80 03 00 28 .d.H.c.T.c.....(
805160A0 2C 00 00 00 41 82 00 10 2C 00 00 05 41 82 00 08 ...A...,...A...
805160B0 7C 65 1B 78 38 65 0C 94 38 80 00 0  4B AF 3A C5 e.x8e..8...K.:.
805160C0 3C 60 80 38 38 80 00 00 80 63 1C 48 80 63 00 54 `.88....c.H.c.T
805160D0 80 63 00 0C 80 03 00 28 2C 00 00 00 41 82 00 10 .c.....(,...A...
805160E0 2C 00 00 05 41 82 00 08 7C 64 1B 78 80 64 0C 98 ,...A...|d.x.d..
805160F0 4B D1 39 01 3C 60 80 38 38 80 00 00 80 63 1C 48 K.9.<`.88....c.H
```

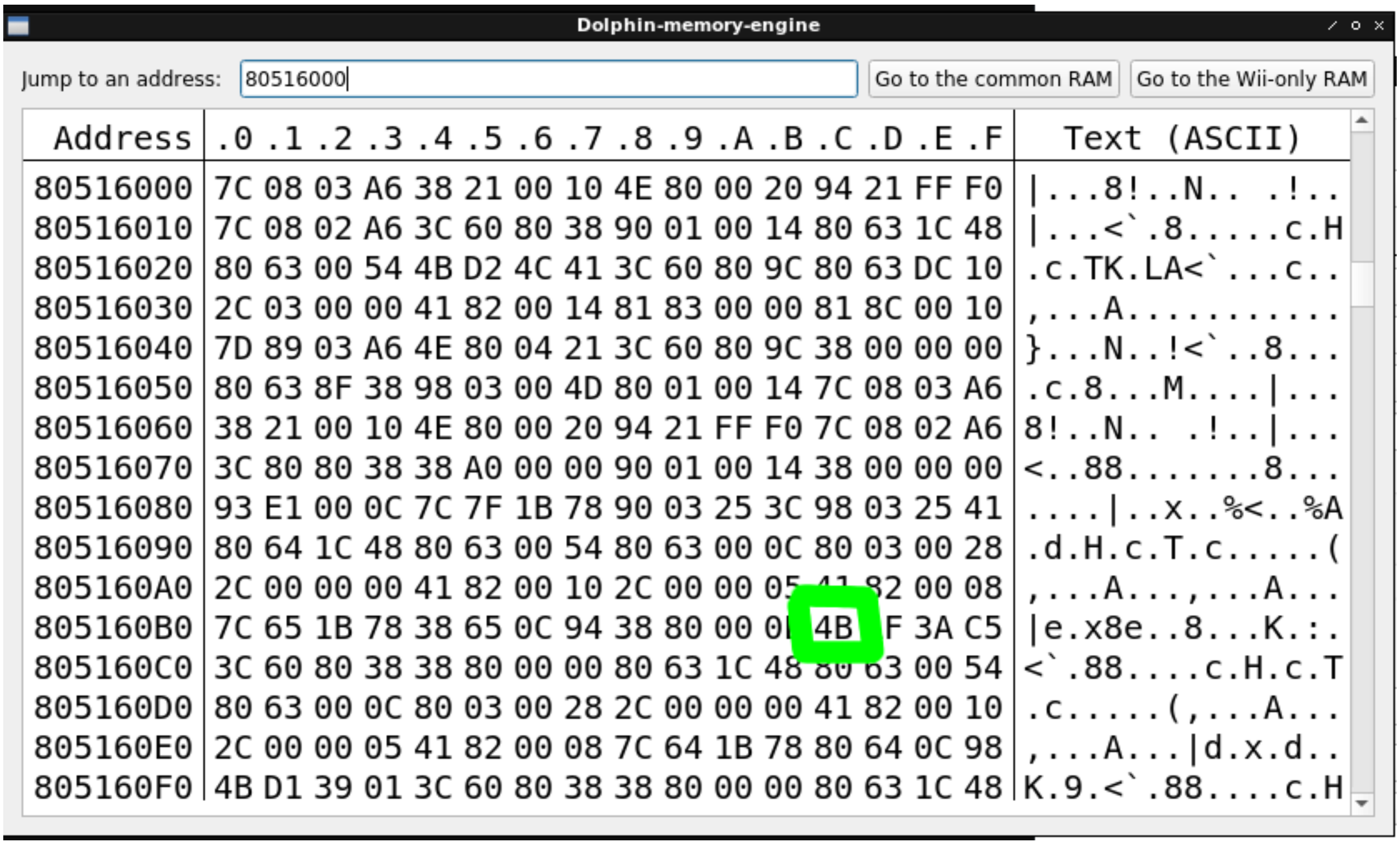
The halfword at that same memory Address is **0x4BAF**. Picture below has the halfword outlined in blue.


Dolphin-memory-engine

Jump to an address: 80516000 | Go to the common RAM | Go to the Wii-only RAM

```
Address  .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F  Text (ASCII)
80516000 7C 08 03 A6 38 21 00 10 4E 80 00 20 94 21 FF F0 |...8!..N.. .!..
80516010 7C 08 02 A6 3C 60 80 38 90 01 00 14 80 63 1C 48 |...<`.8.....c.H
80516020 80 63 00 54 4B D2 4C 41 3C 60 80 9C 80 63 DC 10 .c.TK.LA<`...c..
80516030 2C 03 00 00 41 82 00 14 81 83 00 00 81 8C 00 10 ,...A...........
80516040 7D 89 03 A6 4E 80 04 21 3C 60 80 9C 38 00 00 00 }...N..!<`..8...
80516050 80 63 8F 38 98 03 00 4D 80 01 00 14 7C 08 03 A6 .c.8...M....|...
80516060 38 21 00 10 4E 80 00 20 94 21 FF F0 7C 08 02 A6 8!..N.. .!..|...
80516070 3C 80 80 38 38 A0 00 00 90 01 00 14 38 00 00 00 <..88.......8...
80516080 93 E1 00 0C 7C 7F 1B 78 90 03 25 3C 98 03 25 41 ....|..x..%<..%A
80516090 80 64 1C 48 80 63 00 54 80 63 00 0C 80 03 00 28 .d.H.c.T.c.....(
805160A0 2C 00 00 00 41 82 00 10 2C 00 00 05 41 82 00 08 ,...A...,...A...
805160B0 7C 65 1B 78 38 65 0C 94 38 80 00 0  4B AF  A C5 |e.x8e..8...K.:.
805160C0 3C 60 80 38 38 80 00 00 80 63 1C 48 80 63 00 54 <`.88....c.H.c.T
805160D0 80 63 00 0C 80 03 00 28 2C 00 00 00 41 82 00 10 .c.....(,...A...
805160E0 2C 00 00 05 41 82 00 08 7C 64 1B 78 80 64 0C 98 ,...A...|d.x.d..
805160F0 4B D1 39 01 3C 60 80 38 38 80 00 00 80 63 1C 48 K.9.<`.88....c.H
```

The byte at that same Memory Address is simply **0x4B**. Picture below has the byte outlined in green.



```
 Address  .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F   Text (ASCII)
80516000  7C 08 03 A6 38 21 00 10 4E 80 00 20 94 21 FF F0  |...8!..N.. .!..
80516010  7C 08 02 A6 3C 60 80 38 90 01 00 14 80 63 1C 48  |...<`.8.....c.H
80516020  80 63 00 54 4B D2 4C 41 3C 60 80 9C 80 63 DC 10  .c.TK.LA<`...c..
80516030  2C 03 00 00 41 82 00 14 81 83 00 00 81 8C 00 10  ,...A...........
80516040  7D 89 03 A6 4E 80 04 21 3C 60 80 9C 38 00 00 00  }...N..!<`..8...
80516050  80 63 8F 38 98 03 00 4D 80 01 00 14 7C 08 03 A6  .c.8...M....|...
80516060  38 21 00 10 4E 80 00 20 94 21 FF F0 7C 08 02 A6  8!..N.. .!..|...
80516070  3C 80 80 38 38 A0 00 00 90 01 00 14 38 00 00 00  <..88.......8...
80516080  93 E1 00 0C 7C 7F 1B 78 90 03 25 3C 98 03 25 41  ....|..x..%<..%A
80516090  80 64 1C 48 80 63 00 54 80 63 00 0C 80 03 00 28  .d.H.c.T.c.....(
805160A0  2C 00 00 00 41 82 00 10 2C 00 00 05 41 82 00 08  ,...A...,...A...
805160B0  7C 65 1B 78 38 65 0C 94 38 80 00 0[illegible] 4B [illegible]F 3A C5  |e.x8e..8...K.:.
805160C0  3C 60 80 38 38 80 00 00 80 63 1C 48 80 63 00 54  <`.88....c.H.c.T
805160D0  80 63 00 0C 80 03 00 28 2C 00 00 00 41 82 00 10  .c.....(,...A...
805160E0  2C 00 00 05 41 82 00 08 7C 64 1B 78 80 64 0C 98  ,...A...|d.x.d..
805160F0  4B D1 39 01 3C 60 80 38 38 80 00 00 80 63 1C 48  K.9.<`.88....c.H
```

Final tests:
If I was to say.. what is the halfword at 0x80516086? The correct answer is 0x1B78.
If I was to say.. what is the byte at 0x8051600E? The correct answer is 0xFF.

Chapter 4: Basic Registers

Other than memory, there's one more place a CPU will keep data at. This place is known as the ***Registers***. The Registers are important because the data here is what is used for the calculation of many Instructions.

In a modern CPU, there are 1000's of Registers. However, most of these we don't need to worry about. Registers can be categorized into 3 main groups..

- General (General Purpose Registers, aka GPRs)
- Float (Floating Point Registers, aka FPRs)
- Special (Special Purpose Registers aka SPRs)

Before we continue this thread further, I need to go off-topic and state that this tutorial will focus a little bit on the ARM **Cortex-A57** CPU for its examples and applications. The Cortex A-57 obviously uses the ARMv8 AArch64 Assembly Language. This is the same CPU that is used in the Nintendo Switch (4-cores). There is tons of information, documentation, and debugging tools for this specific CPU.

Going back to the Registers, we will focus on the General Purpose Registers. There are ***31*** GPRs. Register #0 thru Register #30, or also known as **r0** thru **r30**. **Each GPR is 64-bits in length.** Therefore a double-word value is the maximum size data type that can fit into a GPR.

All of the GPRs can be modified (read and/or write) by the CPU. Sometimes a GPR will contain a Memory Address (to use as a reference for a task), or it can contain a basic value (to represent something such as item count). Values in Registers are usally displayed in Hex form by an integer. No fractional integer numbers are able to exist within a GPR. Fractional numbers have to be in Floating Point form and are used in the FPRs.

Two of the GPRs have alternate names. r29 is known as the Frame Pointer (FP). r30 is known as the Link Register (LR). Frame Pointer is too complicated for a beginner to discuss as of now. The Link Register is a register that is used as a method for the CPU to execute sub-routines within a program. It will usually contain a Memory Address of some sorts.

There is a unique register (that is to be used as a GPR when needed) known as the Zero Register. You cannot write to it. It always contains the value of zero. We will discuss this register more in Chapter 6.

There are 2 more Registers I want to discuss very quickly before moving on. Those are the Program Counter (PC) and the Stack Pointer (SP). The PC is a read-only register. It simply keeps track of where the CPU is currently executing at within the program (Memory Address). The SP is used in conjunction with LR for subroutine use. SP will always contain a Memory Address.

Chapter 5: Assembler Installation & Quick Overview

We must install an Assembler. We will use the GNU AAarch64 Assembler. It's a great Assembler with the abilities to specify the Cortex A-57 CPU (Nintendo Switch CPU). Therefore, if there's anything Cortex A-57 specific that you need to test, it can be done. Let's install the Assembler. Unfortunately I only use Linux Machines, so I do not know how to do the following install via Windows. I'm sure Google can find it for you....

Linux only~

```
sudo apt-get update
sudo apt-get install gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu binutils-aarch64-linux-gnu-dbg
```

To make sure your installation of GNU worked, test the following terminal command...

```
aarch64-linux-gnu-as --help
```

You should see an array of information pertaining to assistance in various Assembler commands.

The Assembler itself is actually just one specific tool (out of 3) that were installed. The 3 tools are...

- **aarch64-linux-gnu-as** (The actual Assembler, what we use to make an object file out of a source text file written in ARMv8 Assembly)
- **aarch64-linux-gnu-ld** (Called the Linker. In the most basic terms, it creates an executable file out of an object file)
- **aarch64-linux-gnu-objdump** (What we use to see the Assembly Instructions contained within an Object file)

The executable file is the actual program that will execute the AArch64 instructions that were present in the Source Text File.

There are a slew of symbols that can be used in the Source File for the Assembler (commas, hashtags, asterisks, etc). Some of these symbols are required to write out actual Assembly Instructions in the Source file. There are also what is called *Assembler Directives* which are Assembler specific "commands" to tell the Assembler how to assemble the file. We're not going to dive into all of that for now as you will need to learn basic Instruction Format first (covered in the next Chapter).

For now, let's just go over two topics. Writing numerical values correctly, and writing notes/comments.

Writing Numerical Values in a Source File:
When writing decimal values, there are no special requirements. You can simply write out the number. For negative numbers, simply make sure the decimal value is appended with a minus symbol (i.e. -100).

When writing hexadecimal values, you ***must*** pre-pend the number with "0x". Example: 0xAC. Regarding negative hex numbers, that will be covered in the next Chapter.

You have the ability to write values in binary form. To do this, you ***must*** pre-pend the number with "0b". Example: 0b0101

Hex vs Decimal for writing Values:
When writing Instructions in a Source File for the Assembler, some instructions require a numerical value to be written out. If the value is small enough to be shown as a hex byte, then it's easier in my opinion to write it in Decimal form. For the case you need to write out Negative numbers, writing in Decimal form is fine too. Other than those situations, it's easier (visually) to use Hex over Decimal. However, at the end of the day, this is User Preference. Choose what's easier for you.

Writing notes/comments in the Assembler:
There are two different methods to write notes/comments.

Examples:
```
// This is a comment.
/* This is a comment. */
```

The double slash method requires the double slash on every line where a comment is placed. The 2nd method allows you some flexibility like this....
```
/*This
is a
comment.*/
```

You can use notes/comments adjacent to instructions or in their own line within a Source File to help explain why certain instructions were utilized or what purpose they serve.

Final NOTE: In something like C/C++ tutorials, the Reader is shown a "Hello World" program or something similar. Then the author explains a brief summary of what every line of code does. For learning Assembly, in my opinion, this isn't the best approach. You need to learn the "under the hood" stuff first before we can even tackle a "Hello World" program. Therefore, we won't be revisiting the Assembler until Chapter 19.

AArch64/ARM64 Tutorial

Chapter 6: Instruction Format

Now let's move onto basic Instruction Format. When writing Instructions in a Source File, certain Format(s) must be followed or else the Assembler will output an error. We have 1 major factor to discuss first.... the GPRs. The GPRs have two different "*modes*". **Extended** and **Non-Extended**. Extended is when an Instruction will utilize **all** 64-bits of the GPR. When a GPR is in extended use, it is written in a Source File as such...

xD

D = Register's number. For example, GPR 5 in extended form is x**5**. GPR 22 in extended form is x**22**.

Non-Extended is when an Instruction will utilize **only** the lower (righthand-side) 32-bits of the GPR. When a GPR is in non-extended use, it is written as such...

wD

D = Register's number. For example GPR 11 in non-extended form is w**11**.

In every instruction, there is a *Destination Register*. In most instructions, the Destination Register is the Register that holds the result of an executed instruction, while the *Source Register* is the Register that is used to compute the result for the Destination Register. Some instructions will have one source register, while others will have two. Every instruction has only one Destination Register.

For most instructions, there are essentially 4 main formats...

NOTE: "r" can be "x" (extended) or "w" (non-extended).

Format 1:
rD, **rA**, **rB**

rD = Destination Register
rA = 1st Source Register
rB = 2nd Source Register

Keep in mind this is not an actual instruction, or an exact correct format. This is just to show you a very very general view of any instruction that uses two source registers to compute a value for the destination register. Now let's look at an example of an instruction with just one source register..

Format 2:
rD, **rA**

rD = Destination Register
rA = Source Register

Moving onto another Format that also uses 1 source Register, but also comes with something a bit different..

Format 3:
rD, **rA**, **VALUE**

rD = Destination Register
rA = Source Register
VALUE = Immediate Value

Now let's look at an example with zero source registers...

Format 4:
rD, **VALUE**

rD = Destination Register
VALUE = Immediate Value

Immediate Value (**VALUE**) is a numerical value that is ****not**** represented by what's in a Register. Think of it as writing a Value from 'scratch'. The implementation of Immediate Values allows the AArch64 language to have instructions that can provide more flexibility with less register usage.

Before continuing further into Immediate Values, it's vital that you understand **Signed** vs **Unsigned** values. Signed meaning **negative** numbers are possible. So how do we know if a number is negative?

1. The first bit of the value contained in the register is a 1. and...
2. You specify the values to be treated as Signed (this is determined by conditional branches which will be covered in Chapter 10).

Let's look at a value in a non-extended Register (wD). Assuming point 2 from above is true, if the very first bit of the Hex word value is a 1, then the value is negative.

Example of a negative value in wD:
0xFFFFFF18

To confirm this is a negative value, simply take the first hex digit (F) and convert it to binary. Which is 1111. Since the very first (leading) bit of this 4-bit value is 1, this determines that the hex value is negative. Remember!!! This is a negative value <u>if and only if</u> point 2 from above is also True. If not, then the example value is a positive value.

Signed Range in Non-Extended GPRs:
0x80000000 thru 0xFFFFFFFF (Decimal -2,147,483,648 thru -1)
0x00000000 = zero
0x00000001 thru 0x7FFFFFFF (Decimal 1 thru 2,147,483,647)

Unsigned Range in Non-Extended GPRs:
0x00000000 = zero
0x00000001 thru 0xFFFFFFFF (Decimal 1 thru 4,294,967,295)

In regards to the above example of wD (0xFFFFFF18). This hex value (when Signed) is -232 in decimal form.

For an extended register, since it is 64-bits in size, the Signed & Unsigned range is a bit different but follows the same principle...

Signed Range in Extended GPRs:
0x8000000000000000 thru 0xFFFFFFFFFFFFFFFF (Decimal -9,223,372,036,854,775,808 thru -1)
0x0000000000000000 = zero
0x0000000000000001 thru 0x7FFFFFFFFFFFFFFF (Decimal 1 thru 9,223,372,036,854,775,807)

Unsigned Range in Extended GPRs:
0x0000000000000000 = zero
0x0000000000000001 thru 0xFFFFFFFFFFFFFFFF (Decimal 1 thru 18,446,744,073,709,551,615)

Referring back to **VALUE** (Immediate Values aka Values from Scratch), there are some quirks in regard to its number range. **VALUE** comes in all sorts of "flavors". We are going to cover 6 flavors that are easy to dicpher for the beginner. As you continue through this tutorial, you will learn about the other "flavors".

- SIMM32 = Signed 32-bit value
- UIMM32 = Unsigned 32-bit value
- SIMM64 = Signed 64-bit value
- UIMM64 = Unsigned 64-bit value
- SIMM12 = Signed 12-bit value
- UIMM12 = Unsigned 12-bit value

Signed 32-bit values are actually 16-bit halfword values but need to be lengthen to 32-bits in size to represent negative numbers.

Signed Immediate Value 32-bit Range:
0xFFFF8000 thru 0xFFFFFFFF (Decimal -32768 thru -1)
0x0000
0x0001 thru 0x7FFF (Decimal 1 thru 32767)

Signed Immediate Value 64-bit Range: (values follow the same principal as 32-bits in size except for negative numbers; will be 64-bits)
0xFFFFFFFF80000000 thru 0xFFFFFFFFFFFFFFFF (Decimal -2147483648 thru -1)

0x00000000
0x00000001 thru 0x7FFFFFFF (Decimal 1 thru 2147483647)

Unsigned 32-bit and 64-bit values follow the range you would expect.

Unsigned Immediate Value 32-bit Range:
0x00000000
0x00000001 thru 0xFFFFFFFF (Decimal 1 thru 65535)

Unsigned Immediate Value 64-bit Range:
0x0000000000000000
0x0000000000000001 thru 0xFFFFFFFFFFFFFFFF

12-bit Signed values come with some quirks. When negative, 12-bit values will still be a word value in size for when you need to write them as Immediate Values for non-extended usage. For Extended usage, the negative 12-bit values must be written in 64-bit form.

Signed Immediate Value 12-bit Range for Non-Extended Use:
0xFFFFF800 thru 0xFFFFFFFF
0x000
0x001 thru 0x7FF

Signed Immediate Value 12-bit Range for Extended Use:
0xFFFFFFFFFFFFF800 thru 0xFFFFFFFFFFFFFFFF
0x000
0x001 thru 0x7FF

Unsigned Immediate Value 12-bit Range:
0x000
0x001 thru 0xFFF

Let's revisit the Zero Register that was briefly mentioned back in Chapter 4. It is a read only register that is always zero. There are rare occasions where you would need to use to this register to represent zero if an instruction doesn't permit Immmediate Value usage. It doesn't have a GPR number assigned to it. It has the following names...

- wzr // Use this name for Non-Extended purposes
- xzr // Use this name for Extended purposes

You are free to use wzr/xzr in place of writing zero for an Immediate Value.

Clearing the air on some confusing Terminology:

There's some important terminology that we need to discuss before continuing further. Across the web you will see the term "ARM64" or "AArch64" used in place of ARMv8. For starters, AArch64 is a specific instruction set within the entire ARMv8 Architecture. It is the standard/regular/default instruction set of ARMv8. It is the instruction set that you will learn about in this tutorial. ARMv8 contains a mode with the ability to essentially use ARMv**7** instructions. This is known as AArch32 or ARM32. There is also a 3rd instruction set known as Thumb32 (T32 for short). Thumb32 is a more basic instruction set that contains both 16-bit and 32-bit width instructions. Thus, it is a variable-length instruction set.

The term ARM64 is just an alternate name for AArch64. In my opinion, AArch64 can be a confusing term, because it can apply to ARMv**9** as well. ARMv9 has its own AArch64 instruction set that has more features and enhancements than's ARMv8's AArch64 instruction set. ARMv9's AArch64 mode contains some instructions that cannot be used in ARMv8.

So to recap, here are the terms...
AArch64/ARM64 = The alias used for the main instruction set in ARMv8 and ARMv9; but they are not exactly the same
AArch32/ARM32 = 32-bit instruction set present in ARMv8 and ARMv9 that mimics ARMv7 very closely
Thumb32/T32 = Variable width (16 & 32-bit) instruction set present in ARMv8 and ARMv9.

In conclusion, from this point forward, this tutorial will use the term "ARM64" when referencing ARMv8's AArch64 Instruction Set.

FInal Notes: Zero can always be written as just 0. Leading zero digits can be omitted if desired. Because leading zero digits can be omitted, this means that the unsigned 12-bit range works for both extended and non-extended use.

Some considerations:
Alternatively, you can write the numbers in Decimal form into your Source File(s). You can also write out negative hex numbers with the minus (-) symbol. For example, you can write out -1 in Hex as -0x1.

Chapter 7: Basic Instructions

We can finally get into some actual Assembly instructions. This will be a pretty hefty chapter. There is a lot to unpack. Please take your time. A great first instruction to learn would be the ***add*** instruction.

Add:

add xD, xA, xB
add wD, wA, wB

The ***add*** instruction has two "versions". One for extended register use and one for non-extended use. Many basic instructions will have this feature of being equipped with two versions.

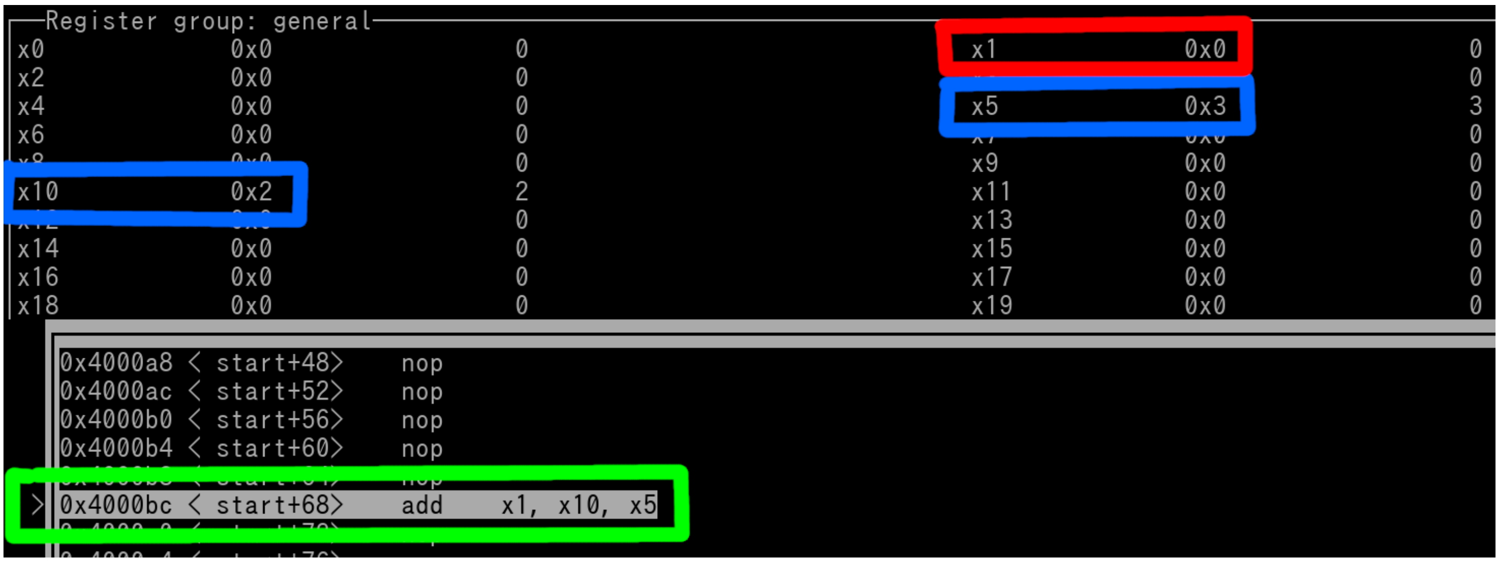
We'll focus on the extended version of ***add***. In the ***add*** instruction, the value in xA is **added** with the value in xB. The result of this addition is **placed** into xD. Whatever value that was in xD beforehand, is **overwritten**.

Example add instruction:
add x1, x10, x5

Value in x10 is added with value in x5. Result placed in x1.

Let's pretend that before the instruction has executed, x10 = 2, and x5 = 3. What's currently in x1 is not a concern for us, as it will be overwritten once the instruction has executed.

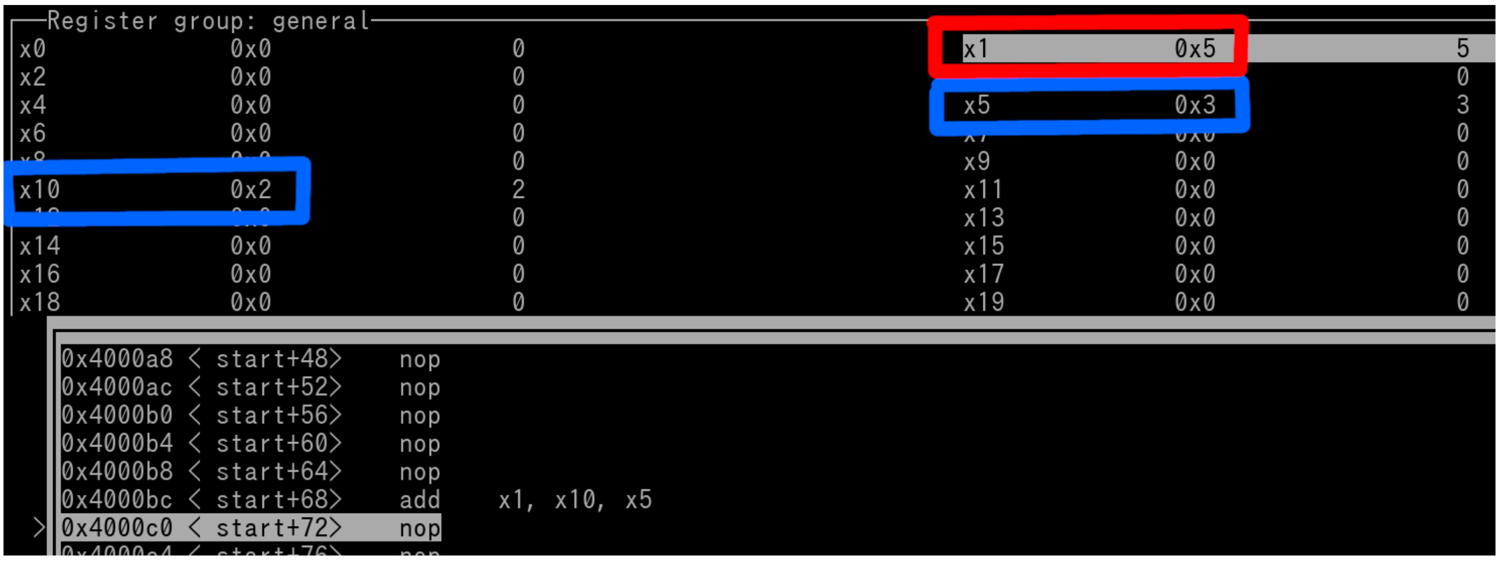
The picture below shows you an instance of this instruction right before it is executed. Both Source registers are outlined in **blue**, and the Destination Register is outlined in **red**. The ***add*** instruction is outlined in **green**.


```
Register group: general
x0             0x0                 0                    x1             0x0            0
x2             0x0                 0                                                  0
x4             0x0                 0                    x5             0x3            3
x6             0x0                 0                    x7             0x0            0
x8             0x0                 0                    x9             0x0            0
x10            0x2                 2                    x11            0x0            0
x12            0x0                 0                    x13            0x0            0
x14            0x0                 0                    x15            0x0            0
x16            0x0                 0                    x17            0x0            0
x18            0x0                 0                    x19            0x0            0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
   0x4000b8 < start+64>    nop
 > 0x4000bc < start+68>    add     x1, x10, x5
   0x4000c0 < start+72>    nop
   0x4000c4 < start+76>    nop
```

<u>Some things to unpack from the above Pic:</u>

The above is a screenshot from a tool called *GNU-Debugger*. I'm using this tool with custom made Assembly Programs/Code to use for various instruction examples. You will using this exact tool in the far future of this Tutorial. All registers are displayed in their Extended form. Each register will have it's Hex and Decimal equivalent values listed. You can see the Hex values are on the near right for each corresponding Register, while the Decimal equivalents are on the far right. Another thing to keep in mind is that the Decimal equivalents are <u>always</u> **64-bit Signed** Decimal equivalents. This is a quirk of the Debugger that can't be changed/modified. Finally let's discuss the instruction outlined in **green**. You will see some information to the left of the instruction. The Hex Value that's incrementing by 4 is the Program Counter. The "start+68" is simply a reference point to in regards to the start of the Program. We'll discuss the ***nop*** instructions shortly.

Anyway, once the instruction gets executed the addition of 2+3 will be performed. The result of **5** is placed into **x1**. View the following picture...


```
Register group: general
x0             0x0                 0                    x1             0x5            5
x2             0x0                 0                                                  0
x4             0x0                 0                    x5             0x3            3
x6             0x0                 0                    x7             0x0            0
x8             0x0                 0                    x9             0x0            0
x10            0x2                 2                    x11            0x0            0
x12            0x0                 0                    x13            0x0            0
x14            0x0                 0                    x15            0x0            0
x16            0x0                 0                    x17            0x0            0
x18            0x0                 0                    x19            0x0            0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
   0x4000b8 < start+64>    nop
   0x4000bc < start+68>    add     x1, x10, x5
 > 0x4000c0 < start+72>    nop
   0x4000c4 < start+76>    nop
```

The result of 5 being placed in x1 is outlined in **red**.

Regarding the ***nop*** instruction, it simply tells the CPU to do nothing, literally. It can serve a variety of purposes. I'm simply inserting ***nop*** everywhere as having other unrelated possible complex instructions everywhere may be "visually cumbersome".

What's great about the ARM64 Assembly Language is that plenty of instructions have multiple formats. For example, in the ***add*** instruction, you can also use 1 source register with an **Immediate Value**. This is known as "Add Immediate".

Add Immediate:

add xD, xA, aimm
add wD, wA, aimm

What is ***aimm***?

aimm stands for Arithmetic Immediate Value. It covers the following Immediate Value Types~

- UIMM12
- UIMM24
- NIMM25

UIMM12 is your typical Unsigned 12-bit Immediate Value Range that you've learned about in Chapter 6. There's also a shift mechanism that can be implented to this to further increase the range, but that will be covered in Chapter 15.

UIMM24 is an Unsigned 24-bit Immediate Value Range but has many restrictions. We won't cover UIMM24 as it includes odd restrictions and is too complicated for a beginner to understand at this time.

NIMM25 is any number from -2^24 minus 1 thru -1). In Hex form for a non-extended register, this is 0xFF000000 thru 0xFFFFFFFF. In Hex form for an extended register, this is 0xFFFFFFFFFF000000 thru 0xFFFFFFFFFFFFFFFF.

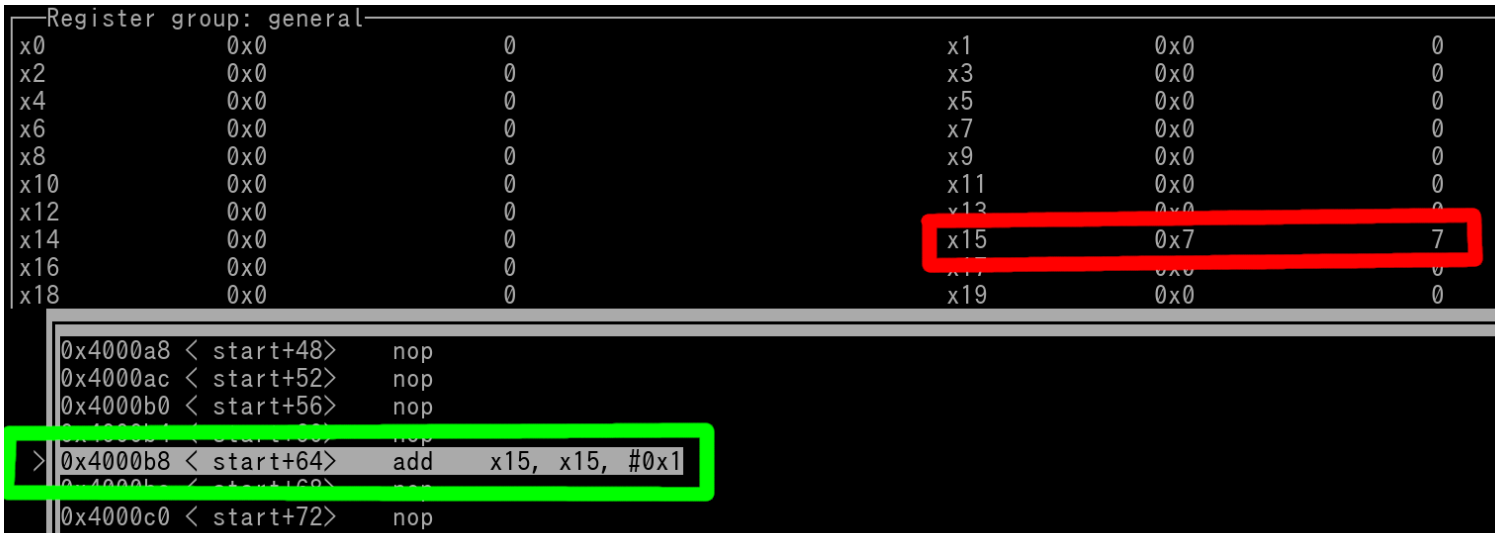
As a Beginner, don't get too caught up in the Immediate Value Ranges. Basically, use whatever value you need to. The Assembler will most likely be able to "alter" your instruction on the fly so it can be assembled. If you are running into too many errors of invalid Immediate Ranges when using the Assembler, then Chapter 8 will explain in detail on how to get around this issue.

Example of an Add Immediate Instruction:
add x15, x15, #0x1

IMPORTANT: As an fyi, there is no such thing as Signed vs Unsigned addition. The addition does not function in that 'manner'. For example: 0xFFFFFFF0 + 0x1 = 0xFFFFFFF1. 0xFFFFFFF1 will **always** be the result. There are indeed a some instructions that are Signed vs Unsigned specific, but don't fret as the naming of such varying instructions differ to let you known the Signage (i.e Signed Multiply vs Unsigned Multiply instructions). Finally, if an addition can't be represented by a proper value (Example: 0xFFFFFFFF + 0x2), then an item known as the Carry Flag wil be set. This is discussed in Chapter 16. Fyi, 0xFFFFFFFF + 0x2 would result as 0x00000001 in a non-extended register.

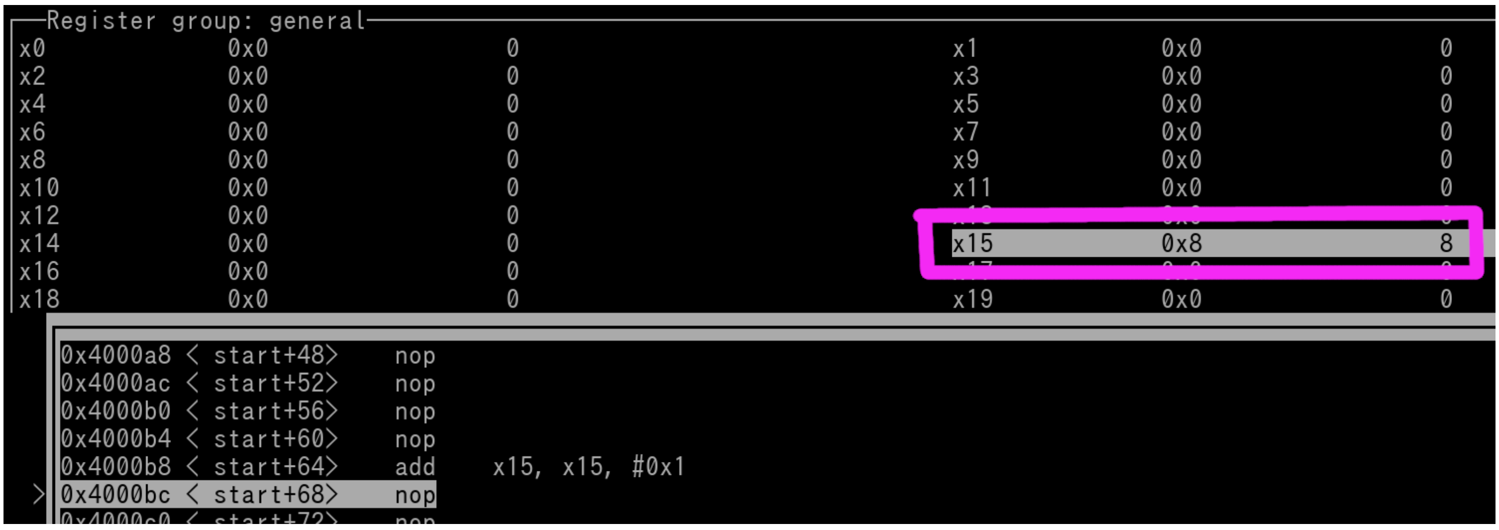
IMPORTANT: For any ARM64 instruction where you use an Immediate Value, you ***SHOULD*** pre-pend said value with a hashtag (#) symbol. The reason being is that some Assemblers may enforce this rule. If it's not followed, the Assembler will refuse to Assemble your Source File. Lucky for us, the GNU Assembler you've installed earlier, is nice and allows you to omit the # symbol. "Official" ARM format requires you to have the value pre-pended with the #.

The value in x15 is added with 1. The result of this addition in **rewritten** back into x15. That is because the register used for the Destination Register is the **same** register used for the Source Register. Thus x15 will simply be incremented by 1. Let's pretend x15 was the value of **7** before the instruction was executed. View the following picture...



```
Register group: general
x0             0x0                 0                  x1             0x0                 0
x2             0x0                 0                  x3             0x0                 0
x4             0x0                 0                  x5             0x0                 0
x6             0x0                 0                  x7             0x0                 0
x8             0x0                 0                  x9             0x0                 0
x10            0x0                 0                  x11            0x0                 0
x12            0x0                 0                  x13            0x0                 0
x14            0x0                 0                  x15            0x7                 7
x16            0x0                 0                  x17            0x0                 0
x18            0x0                 0                  x19            0x0                 0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
>  0x4000b8 < start+64>    add     x15, x15, #0x1
   0x4000bc < start+68>    nop
   0x4000c0 < start+72>    nop
```

We can see x15 (outlined in red) is 7 but the ***add*** instruction (outlined in green) hasn't been executed yet. Now, when the ***add*** instruction does execute, x15 will be incremented by 1 as shown by the following picture...



```
Register group: general
x0             0x0                 0                  x1             0x0                 0
x2             0x0                 0                  x3             0x0                 0
x4             0x0                 0                  x5             0x0                 0
x6             0x0                 0                  x7             0x0                 0
x8             0x0                 0                  x9             0x0                 0
x10            0x0                 0                  x11            0x0                 0
x12            0x0                 0                  x13            0x0                 0
x14            0x0                 0                  x15            0x8                 8
x16            0x0                 0                  x17            0x0                 0
x18            0x0                 0                  x19            0x0                 0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
   0x4000b8 < start+64>    add     x15, x15, #0x1
>  0x4000bc < start+68>    nop
   0x4000c0 < start+72>    nop
```

As you can see, x15 (outlined in magenta) has been incremented by 1 and now contains the value of **8**.

Now that you have learned your first ever ARM64 instruction, we need to cover a small but significant topic. Back in Chapter 5 we quickly discussed about writing notes/comments in a Source File. Well, you can write these notes/comments adjacent to instructions. Like this...

add x15, x15, #1 // Increment x15 by 1

Subtract & Subtract Immediate:

sub xD, xA, xB //xD = xA - xB
sub wD, wA, wB //wD = wA - wB
sub xD, xA, aimm //xD = xA - aimm
sub wD, wA, aimm //wD = wA - aimm

Example sub immediate instruction:
sub x0, x25, 20 //Value in x25 minus 20 is preformed. Result is written to x0.

Alternatively, you could write the instruction as "sub x0, x25, 0x14" if you prefer to write the Immediate Value in Hex form.

ARM64 also includes typical divide and multiply instructions. There are only 4 divide instructions, they are as follows....
sdiv xD, xA, xB //Signed Division. xD = xA/ xB
sdiv wD, wA, wB //Signed Division. wD = wA / wB
udiv xD, xA, xB //Unsigned Division. xD = xA / xB
udiv wD, wA, wB //Unsigned Division. wD = wA / wB

NOTE: If a division by zero occurs, then the result is always 0.
NOTE: If a result is fractional, then standard rounding is applied (5.7 rounds to 6)
NOTE: There are no forms of the Divide instructions that permits Immediate Value implementation

Here is the most basic multiply instruction...
*mul wD, wA, wB //Signed Multiplication. wA * wB = wD*
*mul xD, xA, xB //Signed Multiplication. xA * xB = xD*

This will do a basic signed multiplication of the two source registers. The result is written to the Destination Register. If the result exceeds 32-bits (for non-extended use) or 64-bits (for extended use), then those lower 32/64-bits of the result is what is written to the Destination Register.

Some other multiply instructions~
smull xD, wA, wB //Signed multiplication Long. Multiply two non-extended register values, Result in extended register
umull xD, wA, wB //Unsigned multiplication Long. Multiply two non-extended register values, Result in extended register

For the case that the values you are multiplying in non-extended registers will produce a value that must be expressed in more than 32-bits, then you can use "Long" style multiplications above.

NOTE: There are plenty of other multiply instructions, I just wanted to cover the basics. ****ALL**** ARM64 multiply instructions do *NOT* allow Immediate Value implementation.

Now let's cover another basic instruction...

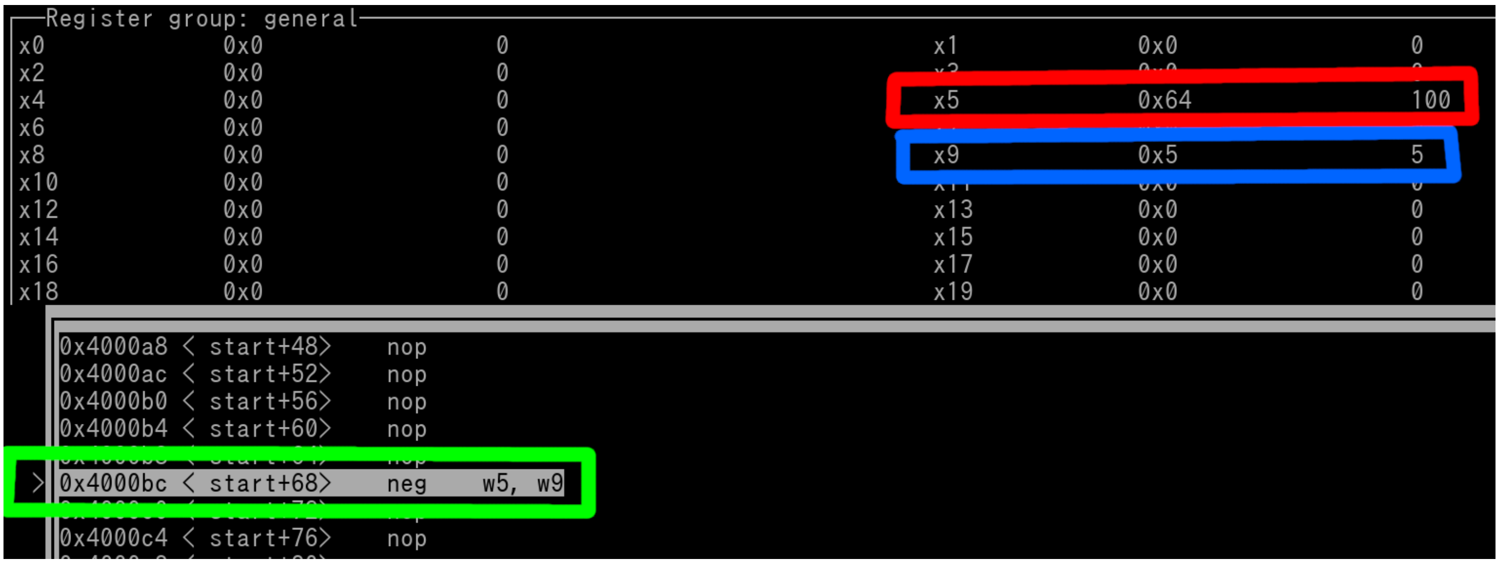
Negate:

neg wD, wA
neg xD, xA

This will simply flip a positive number to be negative or vice-versa. Instances of zero remain as zero.

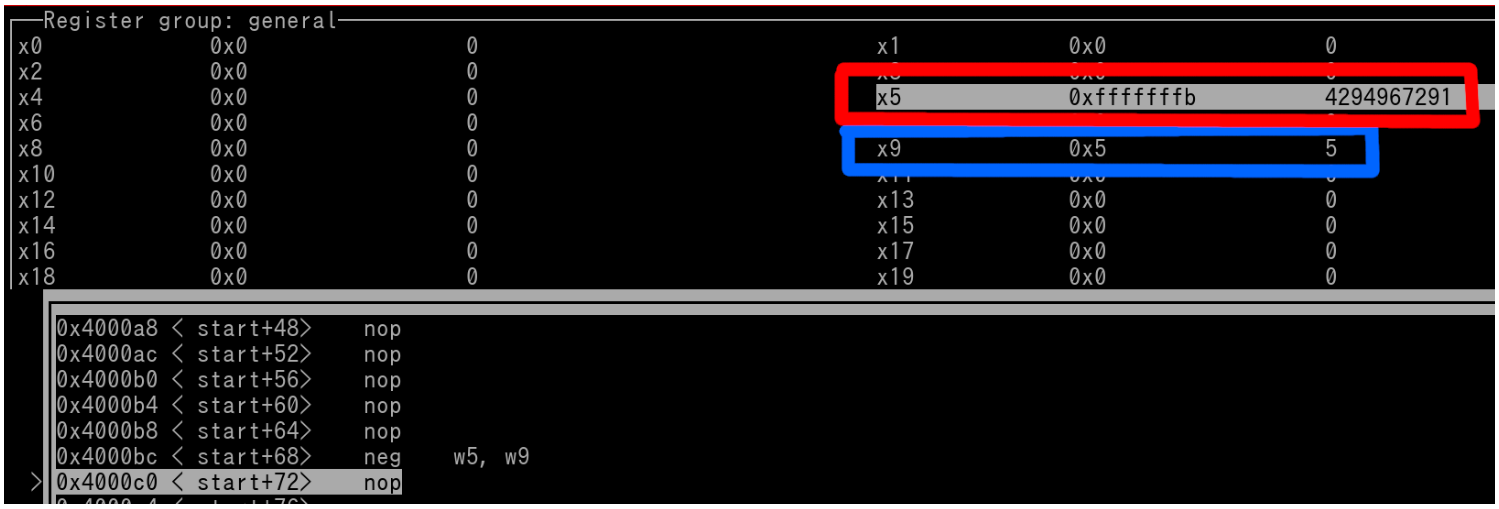
Example:
neg w5, w9

Pretend w9 is 5 before the instruction has executed, what's in w5 isn't a concern because it's value will be overwritten once the instruction gets executed. The following pic shows us the state of the two registers right before the neg instruction is going to be executed...



```
Register group: general
x0             0x0                 0                          x1             0x0                 0
x2             0x0                 0
x4             0x0                 0                          x5             0x64                100
x6             0x0                 0
x8             0x0                 0                          x9             0x5                 5
x10            0x0                 0
x12            0x0                 0                          x13            0x0                 0
x14            0x0                 0                          x15            0x0                 0
x16            0x0                 0                          x17            0x0                 0
x18            0x0                 0                          x19            0x0                 0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
 > 0x4000bc < start+68>    neg    w5, w9
   0x4000c4 < start+76>    nop
```

Source Register (w9) is outlined in blue and Destination Register (w5) is red. w5 currently contains the value of 0x100. However once the instruction executes, that value will be replaced. Let's say the instruction has now executed. View the following pic...



```
Register group: general
x0             0x0                 0                          x1             0x0                 0
x2             0x0                 0
x4             0x0                 0                          x5             0xfffffffb          4294967291
x6             0x0                 0
x8             0x0                 0                          x9             0x5                 5
x10            0x0                 0
x12            0x0                 0                          x13            0x0                 0
x14            0x0                 0                          x15            0x0                 0
x16            0x0                 0                          x17            0x0                 0
x18            0x0                 0                          x19            0x0                 0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
   0x4000b8 < start+64>    nop
   0x4000bc < start+68>    neg    w5, w9
 > 0x4000c0 < start+72>    nop
```

The neg instruction has resulted in w5 being **-5**.

Let's cover one more basic instruction, this one is important...

Move Into Register:

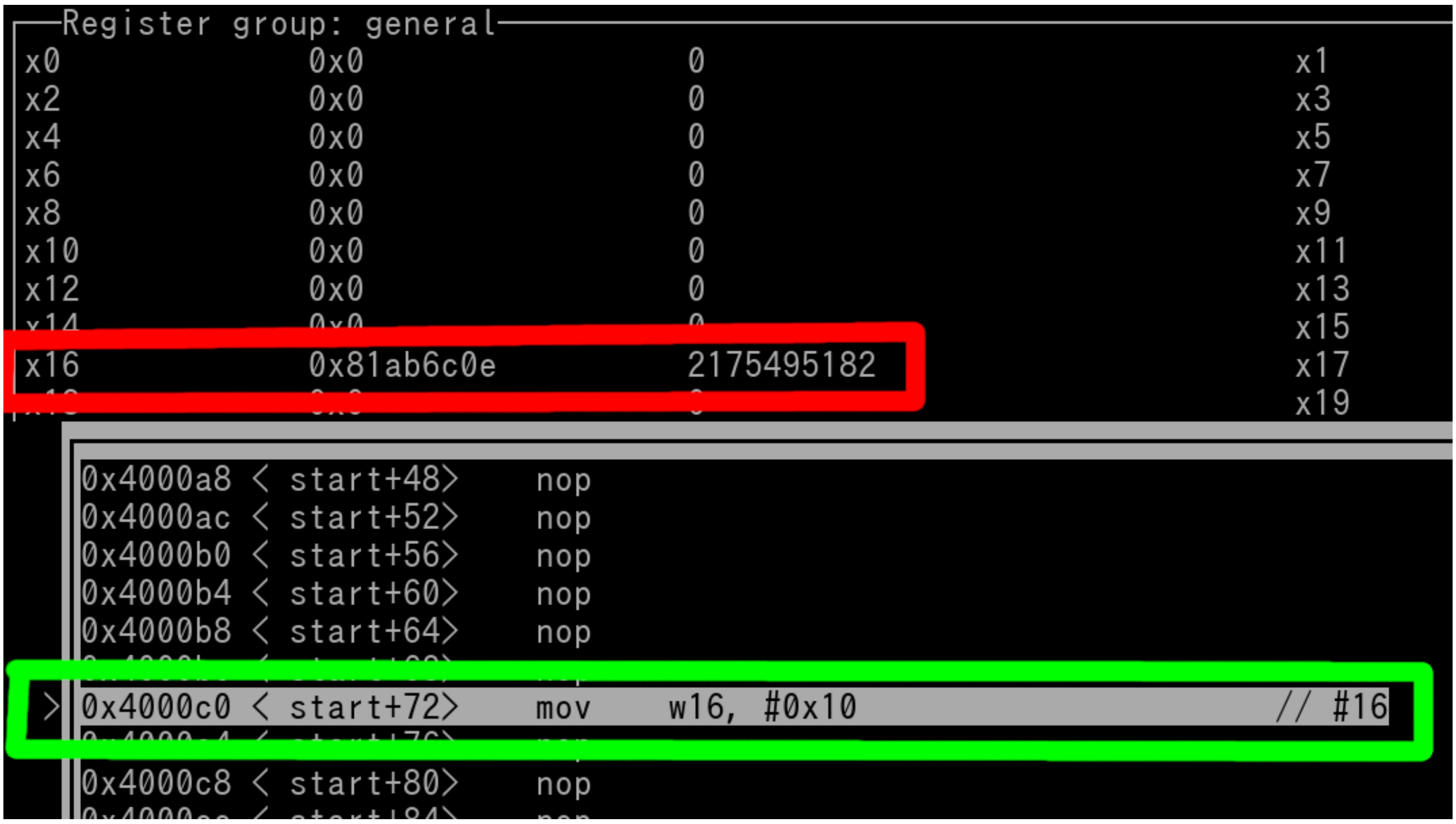
mov xD, SIMM32 //0xFFFFFFFFFFFF8000 thru 0x0000000000007FFF
mov wD, SIMM32 //0xFFFF8000 thru 0x00007FFF
mov xD, xA
mov wD, xA

This instruction (with the Immediate Value option) is what you can use to write values from 'scratch' into a Register. Regarding SIMM32, there's actually more options for Immediate Value Ranges, but those are a bit complex for a beginner to understand. In the next chapter, we will dive into how to write any custom value to a Register.

Let's go into some various examples of ***mov***.

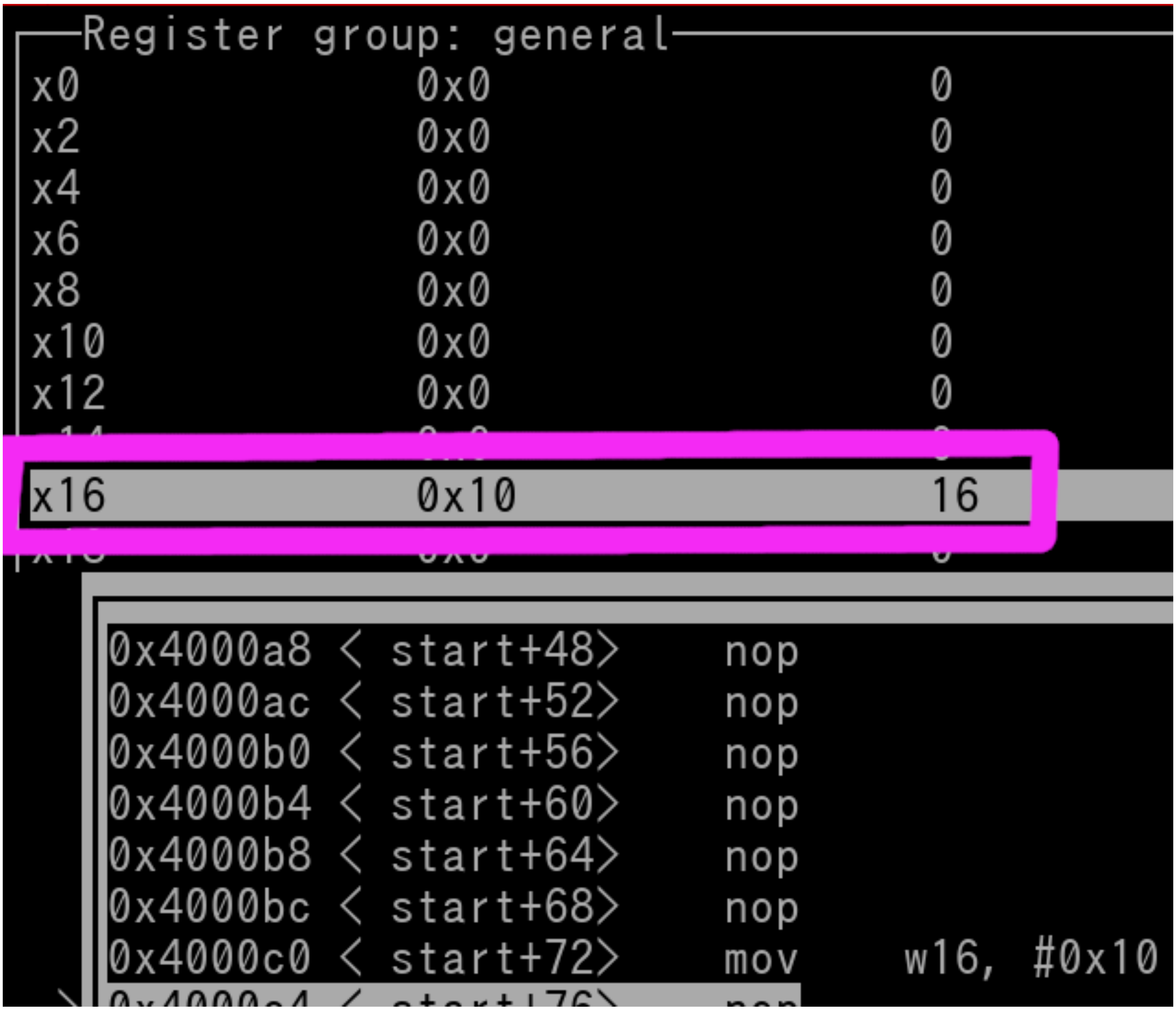
Example non-extended use of ***mov***:
mov w16, #0x0010

The following pic shows us the state of w16 right before the ***mov*** instruction is going to execute (outlined in **Green**). w16 register is outlined in **red**. There is currently a value in w16. It will be overwritten once the ***mov*** instruction executes...



```
┌─Register group: general──────────────────────────────────────────────────
│x0             0x0                 0                               x1
│x2             0x0                 0                               x3
│x4             0x0                 0                               x5
│x6             0x0                 0                               x7
│x8             0x0                 0                               x9
│x10            0x0                 0                               x11
│x12            0x0                 0                               x13
│x14            0x0                 0                               x15
│x16            0x81ab6c0e          2175495182                      x17
                                                                    x19

    0x4000a8 < start+48>    nop
    0x4000ac < start+52>    nop
    0x4000b0 < start+56>    nop
    0x4000b4 < start+60>    nop
    0x4000b8 < start+64>    nop
  > 0x4000c0 < start+72>    mov     w16, #0x10                       // #16
    0x4000c8 < start+80>    nop
```

Now let's have the mov instruction execute....



```
Register group: general
x0             0x0                    0
x2             0x0                    0
x4             0x0                    0
x6             0x0                    0
x8             0x0                    0
x10            0x0                    0
x12            0x0                    0
x16            0x10                   16

0x4000a8 < start+48>    nop
0x4000ac < start+52>    nop
0x4000b0 < start+56>    nop
0x4000b4 < start+60>    nop
0x4000b8 < start+64>    nop
0x4000bc < start+68>    nop
0x4000c0 < start+72>    mov    w16, #0x10
```

The above pic shows us that the ***mov*** instruction has now executed and w16 is now 0x10 (outlined in **magenta**). FYI, you can exclude leading zero(s) in Immediate Values (i.e. write 0x10 for 0x0010). Therefore, you can write the ***mov*** instruction like this...

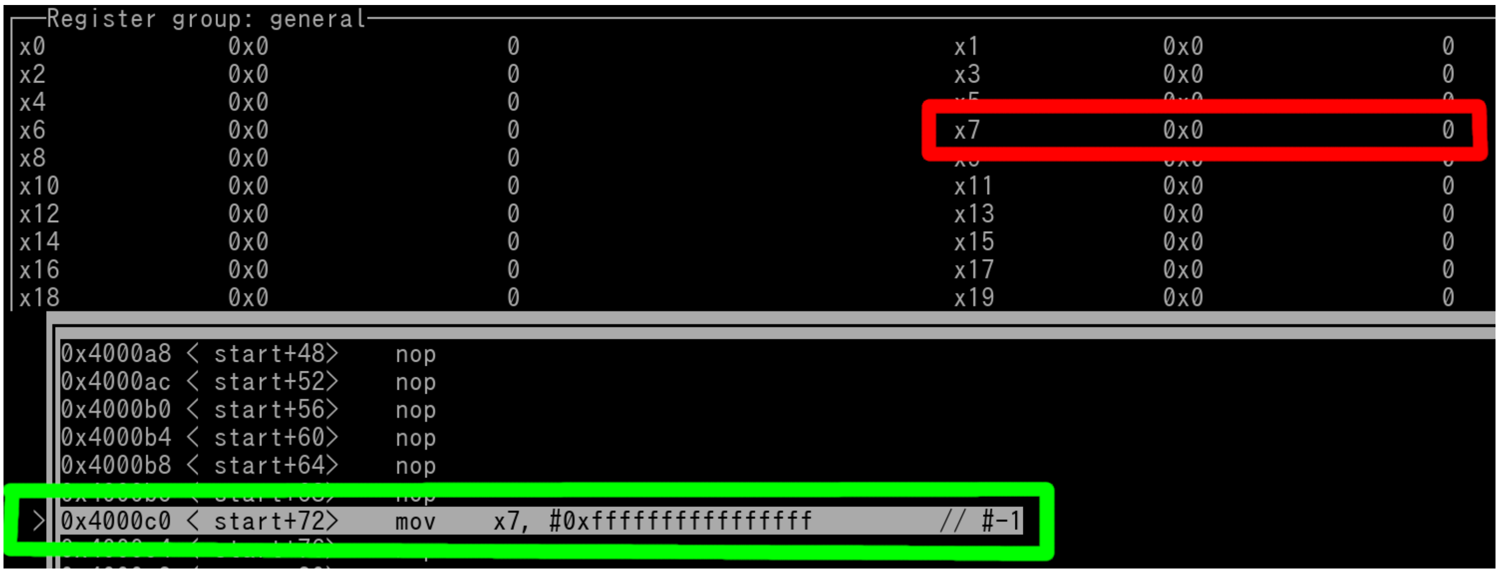
mov w16, #0x10

Another example (extended usage) of mov:
mov x7, #0xFFFFFFFFFFFFFFFF // Decimal form is -1

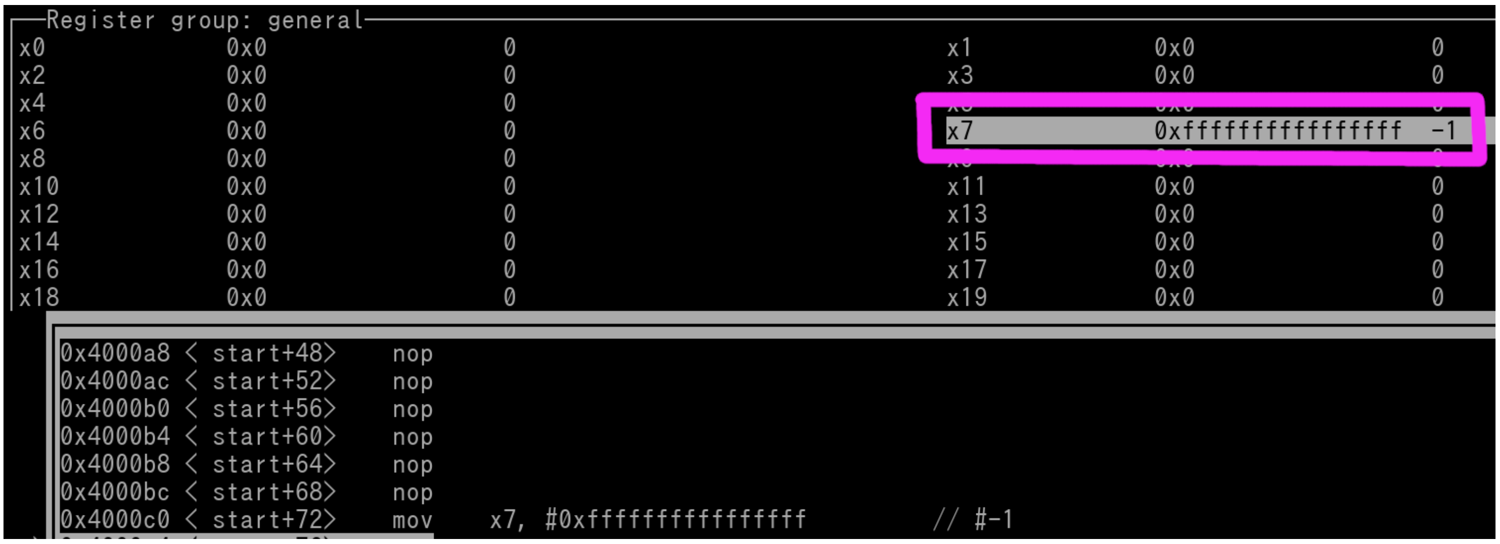
This instruction will set x7 to -1. You don't have to write out this very long hex value, You can use decimal representation for Immediate Values if desired. Like this..

mov x7, #-1

Here's a picture of right before the ***mov*** instruction has executed. x7 outlined in **red**, and the instruction (has not yet executed) is outlined in **green**...


```
Register group: general
x0             0x0                 0                x1             0x0                 0
x2             0x0                 0                x3             0x0                 0
x4             0x0                 0
x6             0x0                 0                x7             0x0                 0
x8             0x0                 0
x10            0x0                 0                x11            0x0                 0
x12            0x0                 0                x13            0x0                 0
x14            0x0                 0                x15            0x0                 0
x16            0x0                 0                x17            0x0                 0
x18            0x0                 0                x19            0x0                 0

0x4000a8 < start+48>    nop
0x4000ac < start+52>    nop
0x4000b0 < start+56>    nop
0x4000b4 < start+60>    nop
0x4000b8 < start+64>    nop
> 0x4000c0 < start+72>    mov    x7, #0xffffffffffffffff          // #-1
```

And here's the picture of once the instruction has executed. x7 is outlined in magenta.


```
Register group: general
x0             0x0                 0                x1             0x0                 0
x2             0x0                 0                x3             0x0                 0
x4             0x0                 0
x6             0x0                 0                x7             0xffffffffffffffff  -1
x8             0x0                 0
x10            0x0                 0                x11            0x0                 0
x12            0x0                 0                x13            0x0                 0
x14            0x0                 0                x15            0x0                 0
x16            0x0                 0                x17            0x0                 0
x18            0x0                 0                x19            0x0                 0

0x4000a8 < start+48>    nop
0x4000ac < start+52>    nop
0x4000b0 < start+56>    nop
0x4000b4 < start+60>    nop
0x4000b8 < start+64>    nop
0x4000bc < start+68>    nop
0x4000c0 < start+72>    mov    x7, #0xffffffffffffffff          // #-1
```

About General Instruction flow; writing multiple Instructions in your Source File:
Instructions (except in the case of exceptions aka crashes, and branches aka jumps) execute in a top to bottom fashion. For example we have the following 3 instructions...

add x1, x10, x5 //This instruction will execute first
mov x7, -1 //This instruction executes second
mul x0, x0, x30 //And this instruction executes third

The top ***add*** instruction executes first. Then the ***mov*** will be executed. After that, the ***mul*** instruction will be executed. When writing instructions out in any Assembler, only one instruction can be present per 'line/row' in your Source File.

Chapter 8: Writing Full Values to Registers

This chapter will teach you how to write in any value (64-bits limited ofc) to a GPR. Knowing this will help you get around invalid Immediate Value ranges as that can be a problem sometimes. There are two different methods that we will discuss.

1. Literal Pools
2. Using "move" type instructions

Literal Pools~

A Literal Pool is basically an Assembler "trick" that allows us to write in values to a register withOUT using a mov-type instruction(s).

Example: We want to load 0x5FFF0FFF into w10
ldr w10, =0x5FFF0FFF

Example: We want to load 0x111122223333CCCC into x16
ldr x16, =0x111122223333CCCC

Take NOTE of the Equals (=) symbol that is attached to the Immediate Value of the above instructions. This will signal to the Assembler that you are using a Literal Pool.

We won't go into the "technicals" of how this works as some of the aspects of this would cover items that you have not learned about yet. Just know that you have this "trick" in your toolbox to use.

Using "move" type instructions~

We've talked about the ***mov*** instruction in the previous Chapter. There is another "move" type instruction called ***movk***.

movk stands for Move Then Keep. The instruction only accepts a **UNSIGNED** 16-bit Immediate Value as it's only input. There is no Source Register usage. In contrast to a typical mov instruction, when the desired Immediate Value is written to the register, all 3 other halfwords present in the Register are **LEFT ALONE**. This may seem confusing at first, this will make sense shortly.

movk has the following options...

movk xD, 0xXXXX, lsl #48
movk xD, 0xXXXX, lsl #32
movk xD, 0xXXXX, lsl #16
movk xD, 0xXXXX //Implies lsl #0

lsl stands for a leftward bit shift. We won't go into detail of what bit shifting is. You will learn about it in Chapter 14. Just understand that...

lsl #48 is to write the Immediate Value of 0xXXXX000000000000
lsl #32 is to write the Immediate Value of 0x0000XXXX00000000
lsl #16 is to write the Immediate Value of 0x00000000XXXX0000
lsl #0 is to write the Immediate Value of 0x000000000000XXXX

Once again, remember that when the Immediate Value is written, the other 3 halfwords are **LEFT ALONE**. As you can see, since other values in the Register are Kept, this gives us the ability to write any value from scratch that we want to.

The ***movk*** instruction has two non-extended variants as well...

movk wD, 0xXXXX, lsl #16
movk wD, 0xXXXX //Implies lsl #0

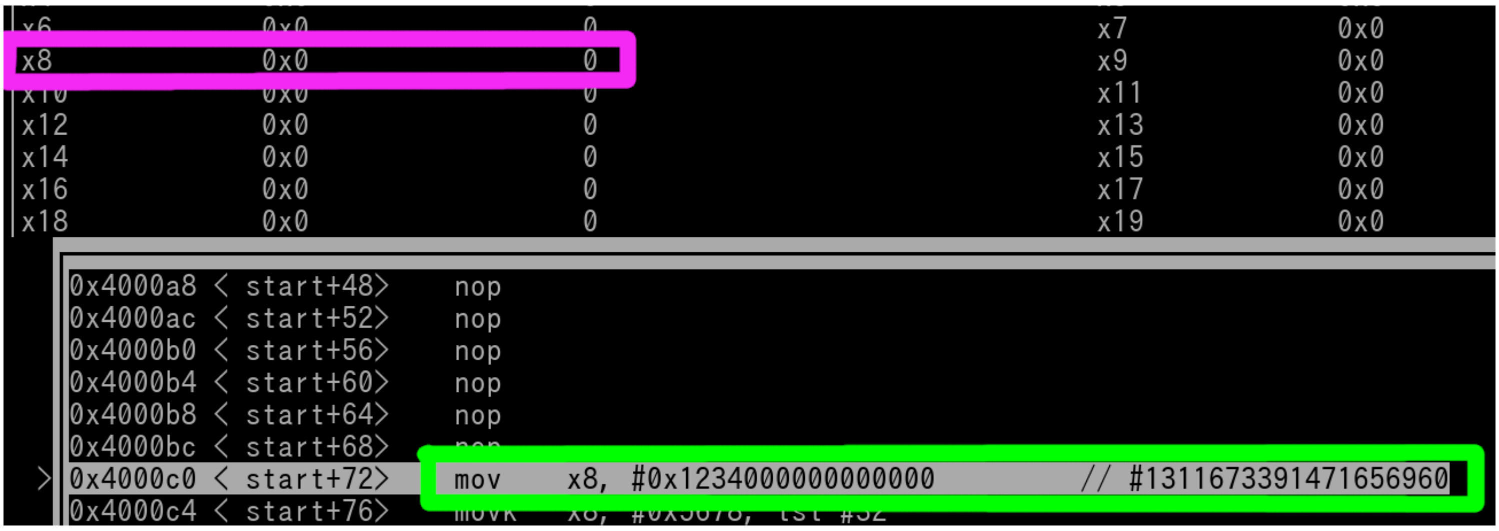
lsl #16 is to write the Immediate Value of 0xXXXX0000
lsl #0 is to write the Immediate Value of 0x0000XXXX

With both ***mov*** and ***movk***, you can use the following template to fill in any custom 64-bit value to a GPR~
// xD = Register
// Value = 0xWWWWXXXXYYYYZZZZ
mov xD, 0xWWWW000000000000
movk xD, 0xXXXX, lsl #32
movk xD, 0xYYYY, lsl #16
movk xD, 0xZZZZ

Using the template, let's say we want to write the value of 0x123456789ABCDEF0 to x8. We can use the template and write it out like this...
mov x8, 0x1234000000000000
movk x8, 0x5678, lsl #32
movk x8, 0x9ABC, lsl #16
movk x8, 0xDEF0

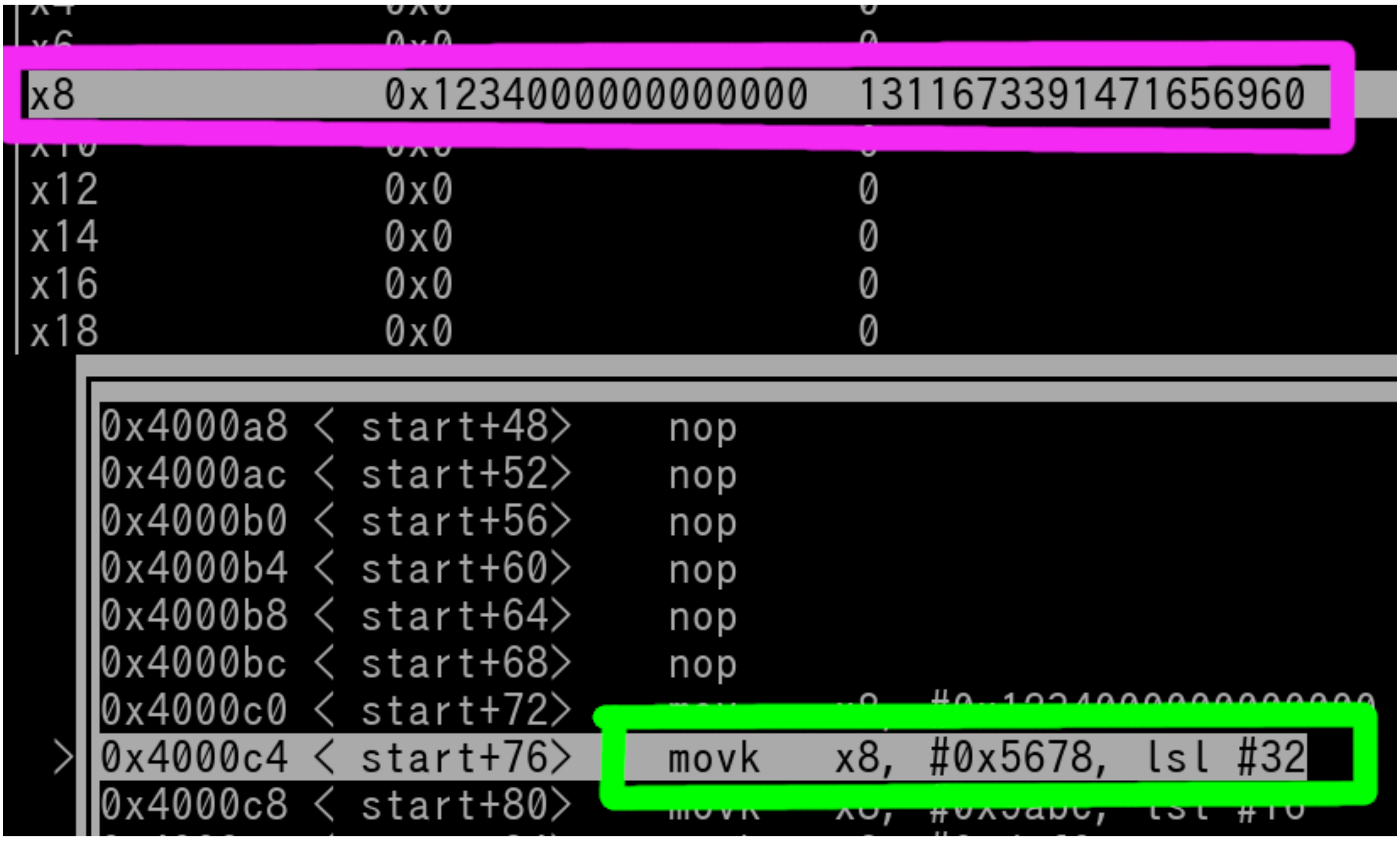
The first instruction in the template has to be a regular ***mov*** as we want any previous data in the Register to be overwritten. The following 5 pics show us what occurs with the above 4 instructions. In the pics below, the Destination Register is outlined in **magenta** and the instruction (that is going to execute) is outlined in **green** except for the final pic.

1st pic (right before ***mov*** is executed)



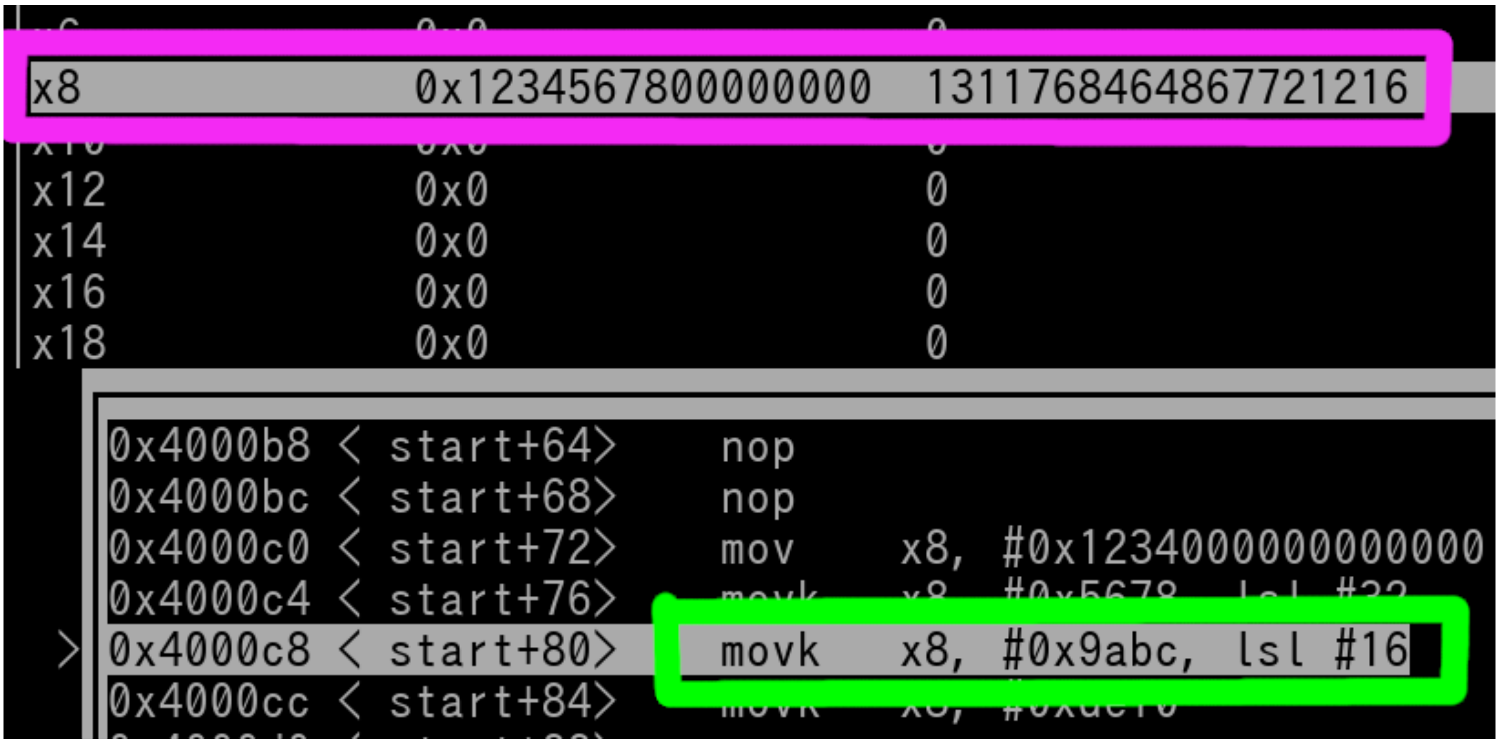
```
x6             0x0                 0                          x7             0x0
x8             0x0                 0                          x9             0x0
x10            0x0                 0                          x11            0x0
x12            0x0                 0                          x13            0x0
x14            0x0                 0                          x15            0x0
x16            0x0                 0                          x17            0x0
x18            0x0                 0                          x19            0x0

   0x4000a8 < start+48>    nop
   0x4000ac < start+52>    nop
   0x4000b0 < start+56>    nop
   0x4000b4 < start+60>    nop
   0x4000b8 < start+64>    nop
   0x4000bc < start+68>    nop
 > 0x4000c0 < start+72>    mov    x8, #0x1234000000000000          // #1311673391471656960
   0x4000c4 < start+76>    movk   x8, #0x5678, lsl #32
```

2nd pic (once ***mov*** executed)



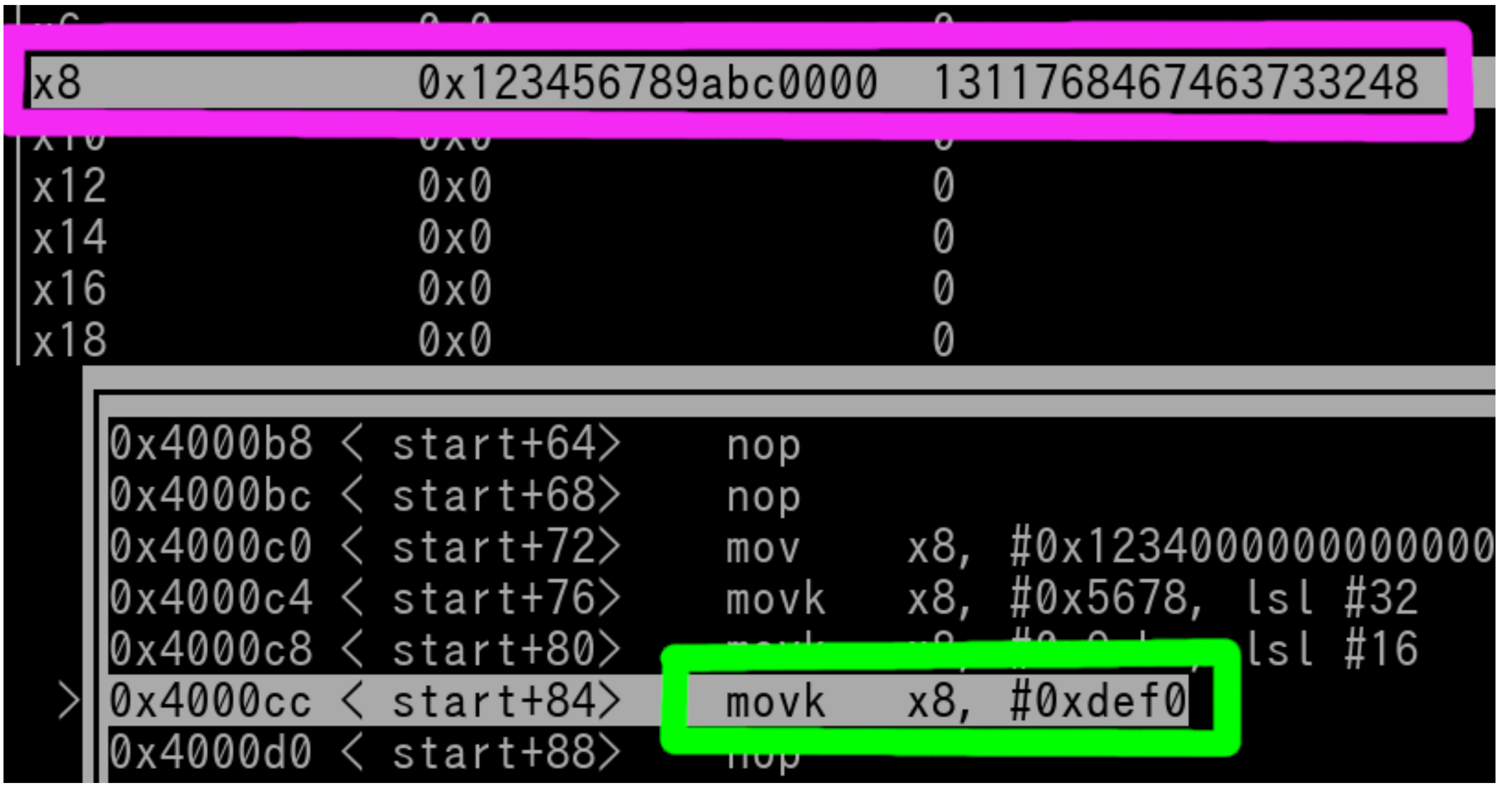
```
x8                0x1234000000000000  1311673391471656960
x12               0x0                 0
x14               0x0                 0
x16               0x0                 0
x18               0x0                 0

0x4000a8 < start+48>    nop
0x4000ac < start+52>    nop
0x4000b0 < start+56>    nop
0x4000b4 < start+60>    nop
0x4000b8 < start+64>    nop
0x4000bc < start+68>    nop
0x4000c0 < start+72>    mov    x8, #0x1234000000000000
>0x4000c4 < start+76>   movk   x8, #0x5678, lsl #32
0x4000c8 < start+80>    movk   x8, #0x9abc, lsl #16
```

3rd pic (once 1st ***movk*** has executed)



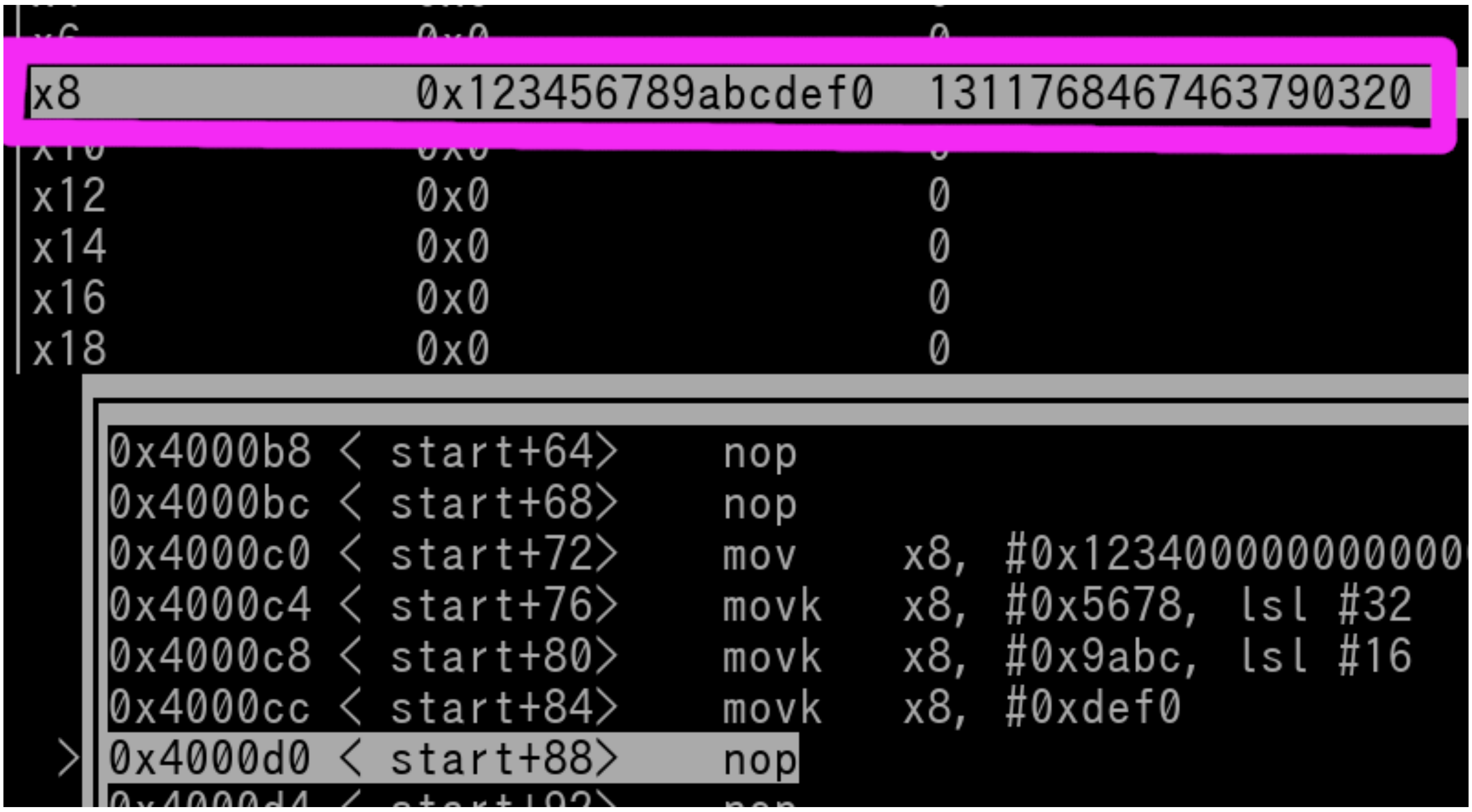
```
x8                0x1234567800000000  1311768464867721216
x12               0x0                 0
x14               0x0                 0
x16               0x0                 0
x18               0x0                 0

0x4000b8 < start+64>    nop
0x4000bc < start+68>    nop
0x4000c0 < start+72>    mov    x8, #0x1234000000000000
0x4000c4 < start+76>    movk   x8, #0x5678, lsl #32
>0x4000c8 < start+80>   movk   x8, #0x9abc, lsl #16
0x4000cc < start+84>    movk   x8, #0xdef0
```

4th pic (once 2nd ***movk*** has executed)



```
x8             0x123456789abc0000   1311768467463733248
x12            0x0                  0
x14            0x0                  0
x16            0x0                  0
x18            0x0                  0

  0x4000b8 < start+64>    nop
  0x4000bc < start+68>    nop
  0x4000c0 < start+72>    mov     x8, #0x1234000000000000
  0x4000c4 < start+76>    movk    x8, #0x5678, lsl #32
  0x4000c8 < start+80>                           lsl #16
> 0x4000cc < start+84>    movk    x8, #0xdef0
  0x4000d0 < start+88>
```

5th pic (once final ***movk*** has executed)



```
x8             0x123456789abcdef0   1311768467463790320
x12            0x0                  0
x14            0x0                  0
x16            0x0                  0
x18            0x0                  0

  0x4000b8 < start+64>    nop
  0x4000bc < start+68>    nop
  0x4000c0 < start+72>    mov     x8, #0x1234000000000000
  0x4000c4 < start+76>    movk    x8, #0x5678, lsl #32
  0x4000c8 < start+80>    movk    x8, #0x9abc, lsl #16
  0x4000cc < start+84>    movk    x8, #0xdef0
> 0x4000d0 < start+88>    nop
```

Writing out the first ***mov*** instruction with 12 appended zeros on the Immediate Value for Extended register usage may be annoying. You can use an alternate instruction instead. That is ***movz***. You have 6 total options with the ***movz*** instruction (first 4 for Extended usage, last 2 for non-Extended)...

movz xD, 0xXXXX, lsl #48
movz xD, 0xXXXX, lsl #32
movz xD, 0xXXXX, lsl #16

movz xD, 0xXXXX //Implies lsl #0
movz wD, 0xXXXX, lsl #16
movz wD, 0xXXXX //Implies lsl #0

This instruction works the same as ***movk*** **ONLY** in regards to the lsl mechanism. Excluding that, the instruction will set the rest of the register (excluding the Immediate Value used ofc) to zero. Therefore, for writing full values to registers, you can replace the usage of the regular ***mov*** instruction with this...

movz xD, 0xXXXX, lsl #48

Thus, you can write out any 64-bit value in an extended register with this updated template that utilizes ***movz***....

// xD = Register
// Value = 0xWWWWXXXXYYYYZZZZ
movz xD, 0xWWWW, lsl #48
movk xD, 0xXXXX, lsl #32
movk xD, 0xYYYY, lsl #16
movk xD, 0xZZZZ

Please note that the Assembler will likely convert the ***movz*** instruction to a ***mov*** instruction upon Assembling. Therefore, on something such as the GNU Debugger, you will see a regular ***mov*** instruction.

Chapter 9: Basic Loads & Stores; Big vs Little Endian

We've mentioned in previous Chapters about how Data can reside in Memory. How do we modify the contents that are residing in Memory? How do we figure out what's currently in Memory? Let's begin.

Simple Vocab Guideline~

- Loaded from Memory into Register = **Read**
- Stored from Register into Memory = **Write**

Let's cover Store instructions first.

Store Double-Word:

str xD, [xA, UIMM12]
*str xD, [xA, sSIMM12]****
str xD, [xA, xB]

***sSIMM12 stands for scaled 12-bit Signed Offset. Scaled means that the Immediate Offset must be a certain multiple. The size of the multiple is dependent on the data size used within the Destination Register. Since the Destination Register is for a Double-Word, the sSIMM12 must have a value that is a multiple of 8.

For ***any*** load or store instruction, a Memory Address must first be calculated via the addition of the two source registers, or the source register and the Immediate Value. The result of this equation is known as the ***Effective Address***.

Effective Address calculation guide (applies to all loads and stores in this Chapter):
xA + UIMM12 = Effective Address
xA + sSIMM12 = Effective Address
xA + xB = Effective Address

Once the Effective Address has been determined, then the act of Storing will take place. In the ***str*** (double-word) instruction, the entire value (64-bits; double-word) in xD is **COPY-PASTED** to the Memory Location determined by the ***Effective Address***. It is important that you understand Store = *Copy-Paste* to Memory.

Example:
str x29, [x3, #0x1B0]

x29 = 0xFFFFFFFF80000001
x3 = 0x0040A0000000

The ***Effective Address*** is 0x0040A0000000 + 0x01B0. Which is 0x0040A00001B0.

Once the instruction has executed, the value of 0xFFFFFFFF80000001 is stored to Memory Address 0x0040A00010B0. Whatever double-word value that was there beforehand, is now overwritten.

Okay, so now that you know how to store a full double-word, how do we store a single word? Well, we still use the same instruction (***str***), but we will use a non-extended Register as the Destination Register.

Store Word:

str wD, [xA, UIMM12]
str wD, [xA, sSIMM12] //Offset must be a multiple of 4
str wD, [xA, xB]

This stores the **ENTIRE** 32-bits of the non-extended register wD to the Effective Address.

In order to Store a halfword, we have to use the strh instruction.

Store Halfword:

strh wD, [xA, UIMM12]
strh wD, [xA, sSIMM12] //Offset must be a multiple of 2
strh wD, [xA, xB]

This stores the **LOWER** 16-bits of the non-extended register wD to the Effective Address.

To store just a Byte, we use the strb instruction.

Store Byte:

strb wD, [xA, UIMM12]
strb wD, [xA, SIMM12] //Scaled offset isn't applicable because it's a multiple of 1
strb wD, [xA, xB]

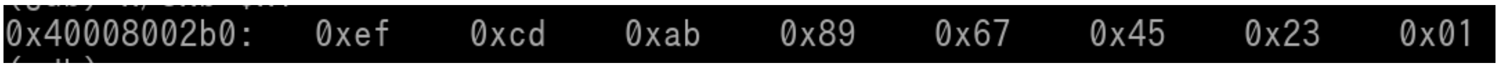
This stores the **LOWER** 8-bits of the non-extended register wD to the Effective Address.

Little Endian annoyances:
Before we discuss load instructions we need to cover Big vs Little Endian. Back in Chapter 3, the method that you have learned to navigate and observe memory with was using Big Endian. Big Endian makes the most intuitive sense. It's exactly like reading a book, left to right, top to bottom.

Little Endian is essentially the opposite of that. Unfortunately, ARM64 uses Little Endian.

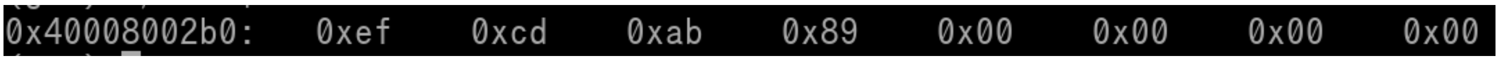
Anyway, lets pretend you have the value of **0x0123456789ABCDEF** in register x0. Let's also pretend that x1 contains the value (Memory Address) of 0x40008002B0.

If we preform the instruction of "str x0, [x1]", memory is now this....


```
0x40008002b0:   0xef    0xcd    0xab    0x89    0x67    0x45    0x23    0x01
```

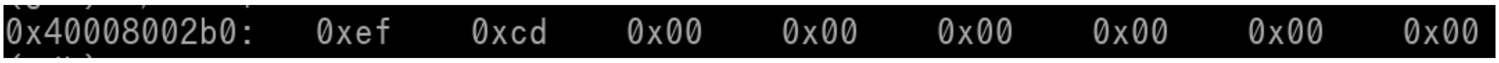
As you can see the value was flipped to store all the bytes within the double-word backwards.

Now pretend memory is re-clear (re-zeroed). If we store just the word of w0 (lower 32-bits of entire GPR which is 0x89ABCDEF) to x1 (instruction would be "str w0, [x1]"), view of memory is this..


```
0x40008002b0:   0xef    0xcd    0xab    0x89    0x00    0x00    0x00    0x00
```

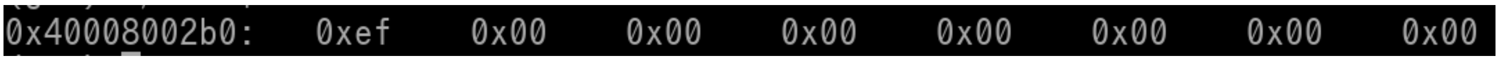
The bytes themselves are still in "intact" but all bytes of the word were flipped from left-right to right-left. **IMPORTANT:** Only the word value (0xEFCDAB89) was written to memory. The zeroes you see in the pic were *NOT* written by the store word instruction.

Okay pretend memory is re-cleared (re-zeroed) again. Now, if we store the halfword of w0 (lower 16 bits of w0 which is **0xCDEF**) to x1 (instruction would be "strh w0, [x1]"), the view of memory is this ..


```
0x40008002b0:   0xef    0xcd    0x00    0x00    0x00    0x00    0x00    0x00
```

Just like with the double-word and word storing, the bytes are "intact" but flipped. **IMPORTANT:** Only the halfword value (0xEFCD) was written to memory. The zeroes you see in the pic were *NOT* written by the ***strh*** instruction.

Bytes for Little Endian load/store exactly how they would in Big Endian. Let's say memory is re-cleared yet again. We store the byte of w0 (lower 8 bits of w0 which is 0xEF) to x1 (instruction would be "strb w0, [x1]"). Memory is this...

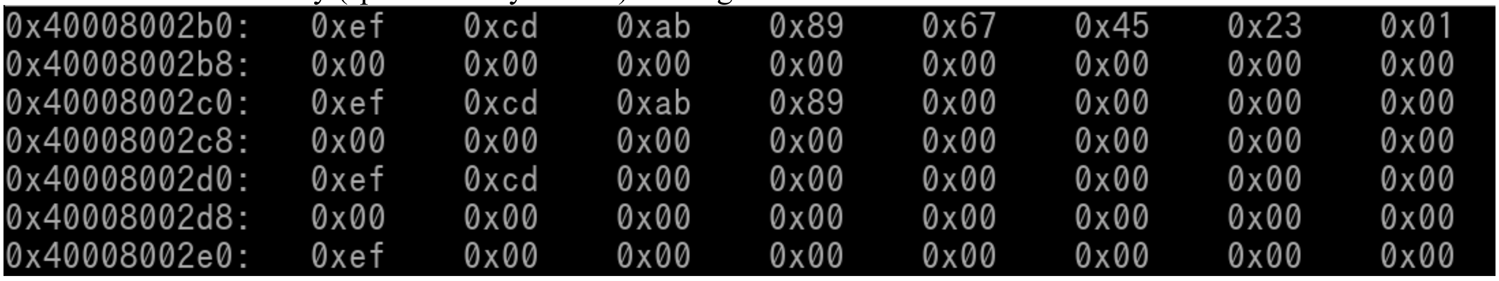

```
0x40008002b0:   0xef    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

IMPORTANT: Only the byte value (0xEF) was written to memory. The zeroes you see in the pic were *NOT* written by the ***strb*** instruction.

Confused still? Let's say we have x0 and x1 that are still the same values from earlier, memory is re-cleared, but we execute the following four instructions..

str x0, [x1] //Store 0x0123456789ABCDEF at 0x40008002B0
str w0, [x1, #0x10] //Store 0x89ABCDEF at 0x40008002C0
strh w0, [x1, #0x20] //Store 0xCDEF at 0x40008002D0
strb w0, [x1, #0x30] //Store 0xEF at 0x40008002E0

Here's a view of Memory (split into 8 byte rows) starting at 0x40008002B0...


```
0x40008002b0:   0xef    0xcd    0xab    0x89    0x67    0x45    0x23    0x01
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002c0:   0xef    0xcd    0xab    0x89    0x00    0x00    0x00    0x00
0x40008002c8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002d0:   0xef    0xcd    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002d8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002e0:   0xef    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Before we cover specific Load instructions, let's discuss what they generally are. Load instructions still calculate the Effective Address via the same method as Store instructions do. Once the Effective Address is calculated, the load will occur. It's **important** that you understand that Load = Copy-Paste from Memory to Register. Whatever was in the Register beforehand is now overwritten. It's simply the opposite of Storing.

Little Endian is still applicable. For example....
0xEFCDAB8967452301 residing in Memory will be loaded into Register as 0x0123456789ABCDEF
0xEFCDAB89 residing in memory will be loaded into Register as 0x89ABCDEF
0xEFCD residing in memory will be loaded into Register as 0xCDEF
0xEF residing in memory will be loaded into Register as 0xEF

Load Double-Word:

ldr xD, [xA, UIMM12]
ldr xD, [xA, sSIMM12] //Offset must be multiple of 8
ldr xD, [xA, xB]

This is essentially the opposite of a Store Double-Word Instruction. The Effective Address must be calculated just like a Store Instruction would. The double-word value located at the Effective Address is **COPY-PASTED** into xD. Whatever value that was in xD beforehand, is now overwritten.

Example:
ldr x11, [x13, #0xFC]

x13 = 0x0041CF000B8

The Effective Address is 0x41CF000B8 + 0xFC = 0x41CF001B4

The double-word value located at 0x41CF001B4 will be copy-pasted into x11.

Load Word:

ldr wD, [xA, UIMM12]
ldr wD, [xA, sSIMM12] //Offset must be multiple of 4
ldr wD, [xA, xB]

To load a Word, we still use the same instruction name (***ldr***), but the Destination Register is now non-extended. It's important to note that in the Load Word instruction, the upper 32-bits (the Extended-only portion) of the Destination Register (xD) is ***ALWAYS*** set to zero. In fact, this is true for ***EVERY*** load instruction where the Destination Register is non-extended (wD).

Example:
ldr w11, [x17, #0x50]

x17 = 0x400C81050

The Effective Address is 0x400C81050 + 0x50 = 0x400C810A0

The word value located at 0x400C810A0 will be copy-pasted into w11. The Extended only portion bits (upper 32-bits of x11) are nulled.

Load Halfword:

ldrh wD, [xA, UIMM12]
ldrh wD, [xA, sSIMM12] //Offset must be multiple of 2
ldrh wD, [xA, xB]

Whenever a ***ldrh*** instruction is executed, the Extended only portion bits (upper 32-bits of xD) are set to zero. Also the upper 16-bits of wD are set to zero as well. Thus, in any ***ldrh*** instruction, wD always results as 0x0000XXXX with XXXX being the halfword value that was loaded.

Load Byte:

ldrb wD, [xA, UIMM12]
ldrb wD, [xA, SIMM12] //Scaled offset isn't applicable because it's a multiple of 1
ldrb wD, [xA, xB]

Whenever a ***ldrb*** instruction is executed, the Extended only portion bits (upper 32-bits of xD) are set to zero. Also the upper 24-bits of wD are set to zero as well. Thus, in any ***ldrb*** instruction, wD always results as 0x000000XX with XX being the byte value that was loaded.

One thing we haven't discussed yet is store and load examples is using load/store instructions that have the same numbered Destination & Source Register

Example 1:
str x5, [x5, #0xF24]

Here we have a store double-word instruction where the Destination and Source register use the same GPR. It operates the same way as like any other basic store instruction. Calculate the Effective Address (x5 + 0xF24), then the instruction simply stores x5 to the Effective Address.

Example 2:
ldrb w3, [x3]

This one is bit less simplistic. Even though the Destination and Source Register use the same GPR, they are using different forms (non-extended vs extended). In this instruction the byte value located at the address in x3 is then loaded into w3. Thus, x3 is no longer its original value.

Okay so at this point in your Assembly Journey, you have learned basic integer instructions (i.e. ***add***) and basic load+store instructions. Now let's go over a simple exercise in which you will use your newly learned skills.

We want to write a Source of code that will do the following...

1. Loads a word value from memory
2. Increment that value by 2
3. Store the updated word value back to where we've loaded it from

We will pretend that x5 + 0x4000 is the exact address of where our word value resides in memory. We will use w0 as our register that will be part of the incrementing. Our first instruction will be a ***ldr*** instruction, which is this...

ldr w0, [x5, #0x4000]

After this instruction has executed, w0 now contains our word value. Let's increment it by 2. This will require a simple ***add*** instruction that uses an Immediate Value of 2....

add w0, w0, #2

In the ***add*** instruction, we've used the same register for both the Destination Register & Source Register. Obviously, because we simply wanted to increment the loaded value by 2, this explains why we need our ***add*** result to rewrite itself back into w0. Okay great, our value has been incremented, now we just need to store it back to where we've loaded it from...

str w0, [x5, #0x4000]

The ***str*** instruction will store the word value back to memory. Here's the entire Source with comments...

ldr w0, [x5, #0x4000] //Load our value from Memory
add w0, w0, #2 //Increment value by 2
str w0, [x5, #0x4000] //Store value back to Memory

Final Note:
For the case where an Immediate Value is 0, you do not need to include it when writing out the instruction. For example...
str x9, [x22, #0]

...Can be written as...
str x9, [x22]

Chapter 10: Compares and Branches

Sometimes we may have some instructions in a program that we only want to be executed under certain conditions. Such as the condition of a register being equal to a certain value. What we essentially need to do is add in alternate "routes/paths" of the execution of the CPU within our program.

In order to create these alternative routes, we need two types of instructions. Compare instructions and Branch instructions. To start let's go over the most basic Branch instruction..

Unconditional Branch:
b label

label? What is that? Branch Instructions do indeed use Immediate Values, but these Values would need to be manually calculated by hand. So to get around this, we use labels. The Assembler will read the labels and automatically know what Immediate Value to apply to the branch instruction.

Example:
b jump_here

add w13, w14, w15

jump_here:
mov x1, #1

In the above example, when the branch instruction gets executed, the execution flow of the CPU will jump over (**skip**) the ***add*** instruction. It will "land" at the ***mov*** instruction and thus execute the ***mov***.

As you can see with the use of the labels, we can name the labels pretty much whatever we want (with some restrictions). When you use a label in a branch instruction, its "landing spot" must contain the same exact label name but be appended with a colon.

Moving onto Conditional Branches... Conditional branches are branches that only execute base on an 'if'. For example let's look at the 'branch if equal' instruction...

Branch If Equal:

beq the_label

the_label will only be 'jumped' to if and only if the conditional branch is true. In order to set up this 'if' for a conditional branch, we need to make a comparison. There are only two different compare instructions for ARM64. Let's go over the most common of the two..

Compare:

cmp xD, aimm
cmp wD, aimm
cmp xD, xD
cmp wD, wD

Example compare and conditional branch code:
cmp x0, #0
beq jump_here
add w1, w2, w3
jump_here:
str x7, [x7]

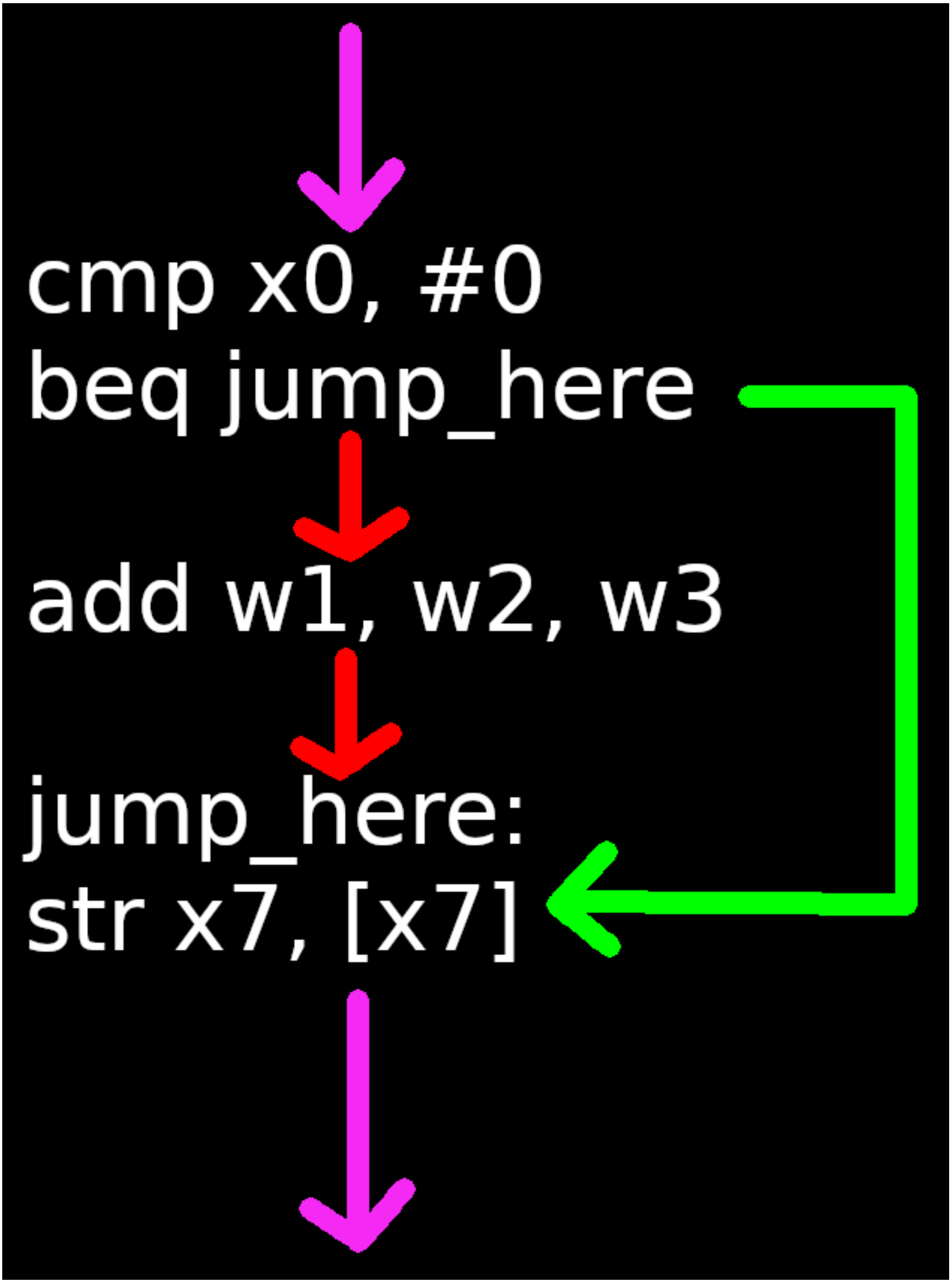
What occurs if x0 **is** 0

1. The compare instruction gets executed. Certain condition flags are set to let any future conditional branch know on how to proceed.
2. Conditional Branch (beq) gets executed. It sees that x0 is indeed equal to 0.
3. Since x0 = 0, the Conditional branch gets executed.
4. Execution of the CPU jumps (skips) over the add instruction and "lands" at the str instruction
5. str instruction is now executed.

What occurs if x0 is ***not*** 0

1. The compare instruction gets executed. Certain condition flags are set to let any future conditional branch know on how to proceed.
2. Conditional Branch gets executed and sees the conditions (x0 = 0?) is *NOT* met. Therefore, nothing happens, there is no jump. Execution of the CPU continues directly downward.
3. add instruction gets executed
4. str instruction gets executed.

Here is a picture of the above example of code. The **magenta** arrows show the general execution flow of the CPU (downward). Once the conditional branch is reached, the execution of the CPU splits into two paths. The **green** path shows the path for when x0 = 0. The **red** path shows the path for when x0 =/= 0.



```
cmp x0, #0
beq jump_here
add w1, w2, w3
jump_here:
str x7, [x7]
```

Let's look at another example of code that implements conditional branching:

```
cmp x0, #0
beq condition_met
mov x1, #1
b the_end
condition_met:
strh w1, [x2]
the_end:
str x3, [x4, #0x40]
```

In the above code we can see a conditional branch present and an unconditional branch present. Why is there an unconditional branch? Well we want the ***strh*** instruction to only execute when x0 = 0. Not only that, we want the ***mov*** instruction to only execute when x0 =/= 0. We want the ***str*** instruction to be executed last in both scenarios. Therefore to satisfy those requirements, an unconditional branch had to be placed in to make sure both paths don't collide with each other.

Let's follow each path.....

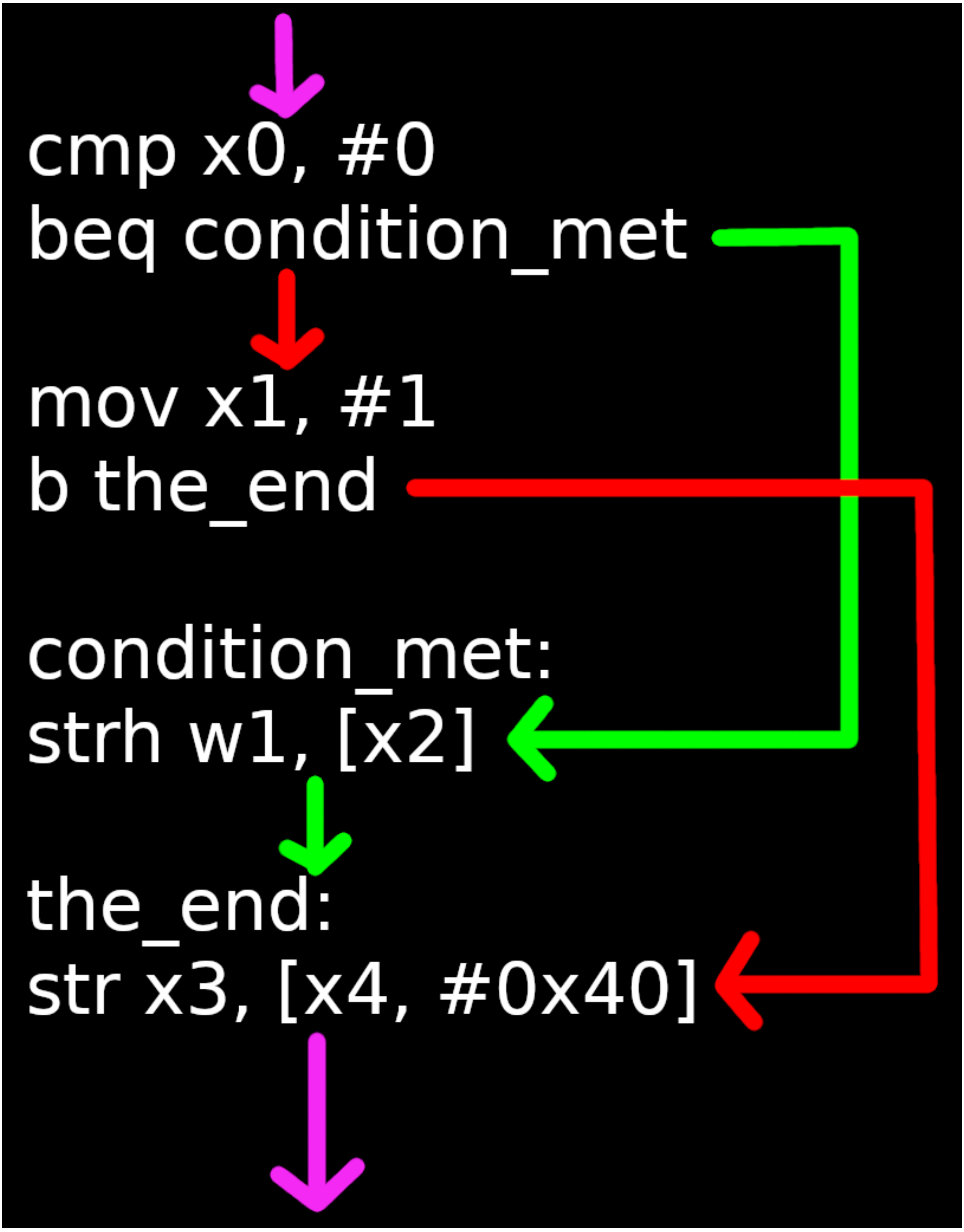
What occurs when x0 = 0

1. Compare instruction gets executed.
2. beq instruction is executed. The conditional branch will be taken since x0 does indeed equal 0
3. Execution the CPU hops over the mov and "b the_end" instructions to "land" at the strh instruction
4. strh gets executed
5. Execution of CPU continues in normal downward fashion to then execute the final (str) instruction

What occurs when x0 =/= 0

1. Compare instruction gets executed
2. beq gets executed but the branch is *NOT* taken since x0 =/0. Therefore beq acts like a nop (do nothing).
3. Execution of CPU continues downward, the mov instruction gets executed.
4. Unconditional Branch (b the_end) gets executed, execution of CPU hops over the strh instruction to skip it and "lands" at the final (str) instruction.
5. Final (str) instruction gets executed.

Here is a picture of the above snippet of code. The **magenta** arrows show the general execution flow of the CPU (downward). Once the conditional branch is reached, the execution of the CPU splits into two paths. The **green** path shows the path for when x0 = 0. The **red** path shows the path for when x0 =/= 0.



```
cmp x0, #0
beq condition_met

mov x1, #1
b the_end

condition_met:
strh w1, [x2]

the_end:
str x3, [x4, #0x40]
```

Signed vs Unsigned Treatment of Values:
Back in Chapter 6, I've mentioned about a point where compares+branches determine if a value in a Register is to be treated as Signed, or treated as Unsigned. What I was referring to is enforcing Signed vs Unsigned based on what kind of conditional branch is used after the compare. This will allow you to choose how to treat Register and/or Immediate Values in the Compare instruction.

Example:
bhs label // Branch if Greater than or Equal to (Unsigned).

Here are all the conditional branch instruction options~

- beq = Equal
- bne = Not Equal
- bgt = Greater Than (signed)
- blt = Less Than (signed)
- bge = Greater Than (signed) or Equal
- ble = Less Than (signed) or Equal
- bhs = Unsigned Higher or Same (aka Unsigned Greater Than or Equal)
- blo = Unsigned Lower Than (aka Unsigned Less Than)
- bmi = Negative
- bpl = Positive or Zero (aka Not Negative)
- bvs = Signed Overflow
- bvc = Not Signed Overflow
- bhi = Unsigned Higher (aka Unsigned Greater Than)
- bls = Unsigned Lower or Same (aka Unsigned Less Than or Equal)
- bcs = Same thing as bhs (branch Carry Set)
- bcc = Same thing as blo (branch Carry Clear)
- bal = Always (same as just a unconditional branch)

So for example. Let's say we have two registers. w7 and w19. Pretend that...

- w2 = 0xFFFFFFFF
- w19 = 0x000000A0

Let's preform a compare on the two values with the following instruction...

cmp w2, w19

Now remember we can determine how the values are treated via what kind of conditional branch we are using. Let's say we want the values to be treated as Signed. We will also pretend, we want the execution of this code to take a branch if and only if w2 is greater than w19. Since we want the values treated as Signed and we want a Branch for when w2 > w19, we have the following two instructions

cmp w2, w19 //Compare w2 vs w19
bgt somewhere //If w2 > w19, take the branch

Now in the above code, the conditional branch (***bgt***), will **NOT** be taken. w2 contains the value of -1 because 0xFFFFFFFF is being treated as Signed. w19 contains the value of 160 (0xA0). -1 is **NOT** greater than 160. Thus, the conditional branch is skipped.

What if we used a ***bhi*** (Unsigned Greater Than) conditional Branch?

cmp w2, w19 //Compare w2 vs w19
bhi somewhere //If w2 > w19, as unsigned, take the branch

Would the branch be taken? YES, it would. Confused? Let's go over it. The ***bhi*** will "treat" the values in w2 and w19 as Unsigned. This will cause w2 to be the value of 4,294,967,295 instead of having -1. w19 is still 160.

Okay, you've now learned about compares and branches. Let's do a quick exercise using your new skills. We want to write a Source of code that does the following...

- Load a word value from memory
- Multiply that word value by 13 **if and only if** the loaded value was 0 or positive
- After multiplication (if it was performed), store word value back to where we've loaded it from

Let's pretend the word value in question is located at the address in x12. We will use w6 & w7 for our multiplication operation. Now we need to start off with a ***ldr*** instruction ofc. Let's write it out...

ldr w6, [x12] //Load our word value from memory into w6

Our word value is now in w6. Remember we can only perform the multiplication if it's 0 or positive. Well we know we need a comparison instruction + conditional branch instruction. We should be using a compare instruction against the value of 0. Why? Well since the multiplication can only be allowed to perform if the loaded value is 0 or positive, it would make sense to use 0 as the "reference point" or "middle ground" for our compare instruction.

What kind of conditional branch though? Well the phrase "0 or positive" is just an alternative saying for "greater than equal to 0". This implies that negative values are indeed possible. Since that is the case, we should be treating the value in w6 as Signed.

Therefore, we should be using the ***blt*** branch. As an fyi, the ***blo*** branch is the Unsigned version. Let's apply the ***blt*** branch with the attached label name "done" and see what our code looks like now.

ldr w6, [x12] //Load our word value from memory into w6
cmp w6, #0 //Compare value in w6 against Zero
blt done //If w6 is less than Zero (signed), branch to the "done:" label

Now the question is, where do we make the conditional branch land at? Well that's simple, we will have it at the very end of our Source. To do that, the "done:" label will be at the very bottom of our Source. Before we place that in and view our current progress, let's cover the multiplication operation.

We will use the standard/basic ***mul*** instruction. Remember that Immediate Value usage is prohibited in this instruction. Our multiple (13) will need to be placed into a different register beforehand. We will use w7 for that.

mov w7, #13 //Place the multiple into w7

Now we can perform the multiplication.

mul w6, w6, w7 //w6 x w7 = w6

In the ***mul*** instruction, we've set w6 as the Destination Register to overwrite the old w6 value. You could of course use a different register as a scratch/temp register, but there's no need to waste an unnecessary use of a another register.

Okay we got our multiplication completed. We need to store it back to where we've loaded it from.

str w6, [x12] //Store new value back to where we've loaded it from

Alright, remember that "done:" label that we need? Well since it has to be the last item of our Source, and the ***str*** instruction is our last ARM64 instruction, the label must be underneath the ***str*** instruction. With all of that being stated, here is our complete source...

```
ldr w6, [x12] //Load our value from memory into w6
cmp w6, #0 //Compare value in w6 against Zero
blt done //If w6 is less than Zero (signed), branch to the "done:" label
mov w7, #13 //Place the multiple into w7
mul w6, w6, w7 //w6 x w7 = w6
str w6, [x12] //Store new value back to where we've loaded it from
done:
```

Sweet. We did it. Our Source is complete.

Final Notes:
It's important to understand the conditional branch instruction "analyzes" the results from the **MOST RECENT** compare instruction. Also "official" ARM format of conditional branch instructions requires a dot (.) to be placed within the instruction name, after the "b" (i.e. b.ne). This is absolutely silly in my opinion. Lucky for us, every ARM64 Assembler that I know of allows you to omit this silly requirement. Debugging tools, such as the GNU Debugger, will include this dot when displaying conditional branches in programs.

Chapter 11: Pre and Post Index of Loads/Stores

Loads and Stores can have additional operations done to them to update the Source Register's Address immediately ***after*** the instruction has executed. This can come in handy for something such as a Loop which you will learn about in the next Chapter.

- Pre-Indexing = Load/Store to the **Effective Address**, **then** increase/decrease the Source Register Address/Value by the Immediate Value present in the Instruction
- Post-Indexing = Load/Store to the **Source Register Address**, **then** increase/decrease the Source Register Address/Value by the Immediate Value present in the instruction

Example Pre-Index Store:
str x0, [x1, #0x4]! //Take note of the exclamation point at the end of the instruction highlighted in yellow. This indicates the store is a pre-index store.

What occurs on the above instruction is...

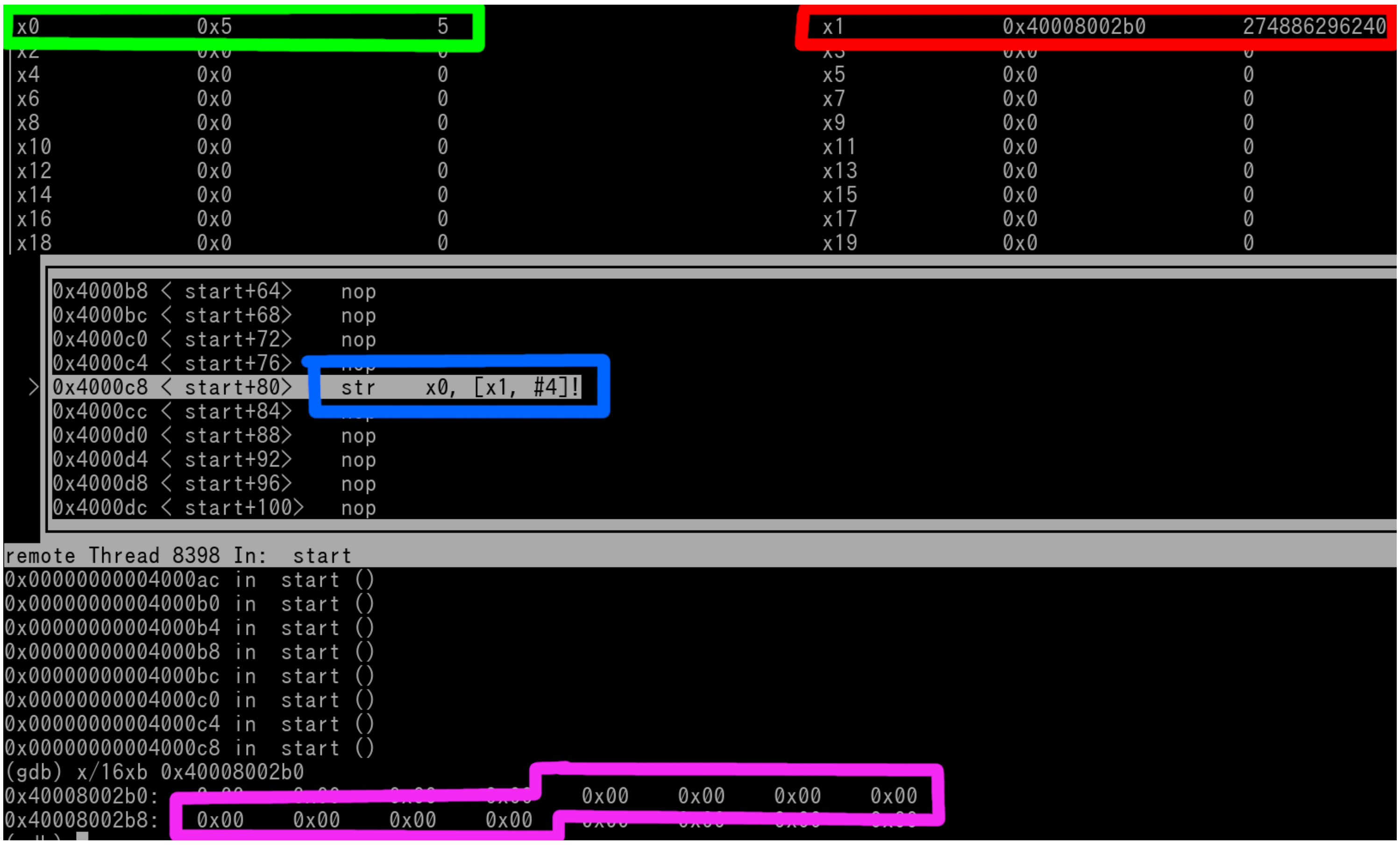
1. x1 + 0x4 = Effective Address
2. x0 (entire double-word) stored at Effective Address
3. x1 is then incremented by 0x4, it now has a new value if the instruction gets executed again

As you can see, to simply add a pre-index feature to a load/store, just append the instruction with a "!".

Confused? Let's go over some pics.

Pretend x0 = 5
Pretend x1 = 0x40008002B0

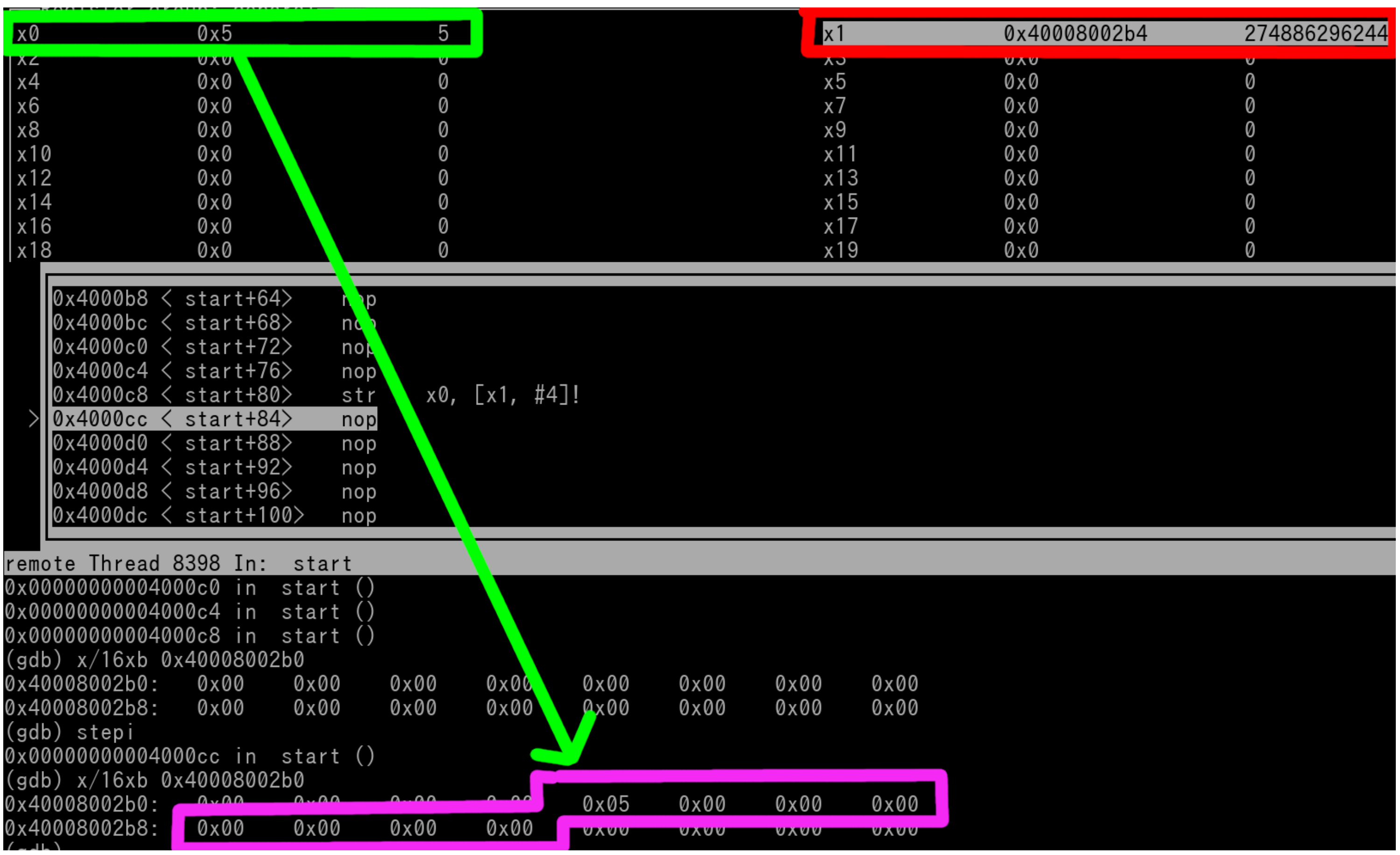
Let's say we execute "str x0, [x1, #0x4]!". Here's a pic of right before the instruction is executed which also includes the view of 16 bytes of Memory starting at 0x40008002B0.



```
x0             0x5                   5             x1             0x40008002b0        274886296240
x2             0x0                   0             x3             0x0                 0
x4             0x0                   0             x5             0x0                 0
x6             0x0                   0             x7             0x0                 0
x8             0x0                   0             x9             0x0                 0
x10            0x0                   0             x11            0x0                 0
x12            0x0                   0             x13            0x0                 0
x14            0x0                   0             x15            0x0                 0
x16            0x0                   0             x17            0x0                 0
x18            0x0                   0             x19            0x0                 0

    0x4000b8 < start+64>    nop
    0x4000bc < start+68>    nop
    0x4000c0 < start+72>    nop
    0x4000c4 < start+76>    nop
  > 0x4000c8 < start+80>    str     x0, [x1, #4]!
    0x4000cc < start+84>    nop
    0x4000d0 < start+88>    nop
    0x4000d4 < start+92>    nop
    0x4000d8 < start+96>    nop
    0x4000dc < start+100>   nop

remote Thread 8398 In:  start
0x00000000004000ac in  start ()
0x00000000004000b0 in  start ()
0x00000000004000b4 in  start ()
0x00000000004000b8 in  start ()
0x00000000004000bc in  start ()
0x00000000004000c0 in  start ()
0x00000000004000c4 in  start ()
0x00000000004000c8 in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002b8:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

You can see that x5 (outlined in **green**) is 5, and x1 (outlined in **red**) is 0x40008002B0. The double-word outlined in **magenta** is where x0 will be stored to (x1 + 4) when the instruction (outlined in **blue**) does execute. We can see in the pic, the **magenta** outlined contents in memory are all zero. Once the instruction executes, those values will be replaced.

Okay let's say we execute the instruction now. Here's the pic.



```
x0             0x5                 5        x1             0x40008002b4        274886296244
x2             0x0                 0        x3             0x0                 0
x4             0x0                 0        x5             0x0                 0
x6             0x0                 0        x7             0x0                 0
x8             0x0                 0        x9             0x0                 0
x10            0x0                 0        x11            0x0                 0
x12            0x0                 0        x13            0x0                 0
x14            0x0                 0        x15            0x0                 0
x16            0x0                 0        x17            0x0                 0
x18            0x0                 0        x19            0x0                 0

   0x4000b8 <start+64>    nop
   0x4000bc <start+68>    nop
   0x4000c0 <start+72>    nop
   0x4000c4 <start+76>    nop
   0x4000c8 <start+80>    str    x0, [x1, #4]!
 > 0x4000cc <start+84>    nop
   0x4000d0 <start+88>    nop
   0x4000d4 <start+92>    nop
   0x4000d8 <start+96>    nop
   0x4000dc <start+100>   nop

remote Thread 8398 In:  start
0x00000000004000c0 in  start ()
0x00000000004000c4 in  start ()
0x00000000004000c8 in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) stepi
0x00000000004000cc in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:   0x00    0x00    0x00    0x00    0x05    0x00    0x00    0x00
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

We can see that x0 was stored to 0x40008002B4 (Effective Address which was x1 + 4). We also see that x1 was updated to have the new value 0x40008002B4 after the store was preformed. The green arrow shows how the contents of x0 (outlined in green) were stored to 0x40008002B4 (outlined in magenta). Because of Little Endian, the value was changed to 0x**05**00000000000000 when the store actually occurred to Memory.

Example Post-Index Store:
str x0, [x1], #0x4 //Take note of how the Source Register alone is enclosed in brackets with the Immediate Value being placed after a second comma

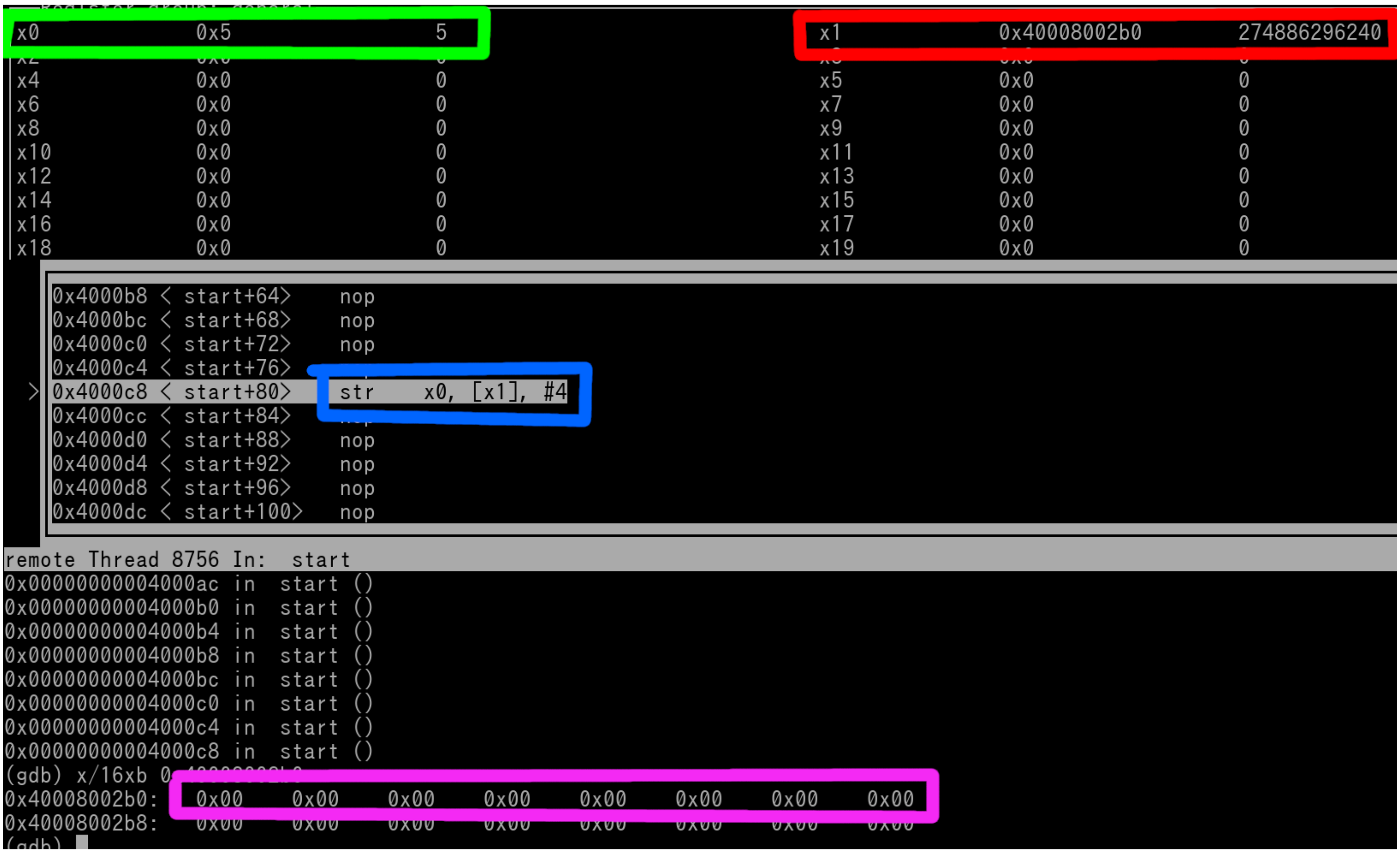
What occurs:

1. x1 = Effective Address
2. x0 (entire double-word) stored at Effective Address
3. x1 is then incremented by 0x4, it now has a new value if the instruction gets executed again

Still confused? Let's go over some pics.

Pretend x0 = 5
Pretend x1 = 0x40008002B0

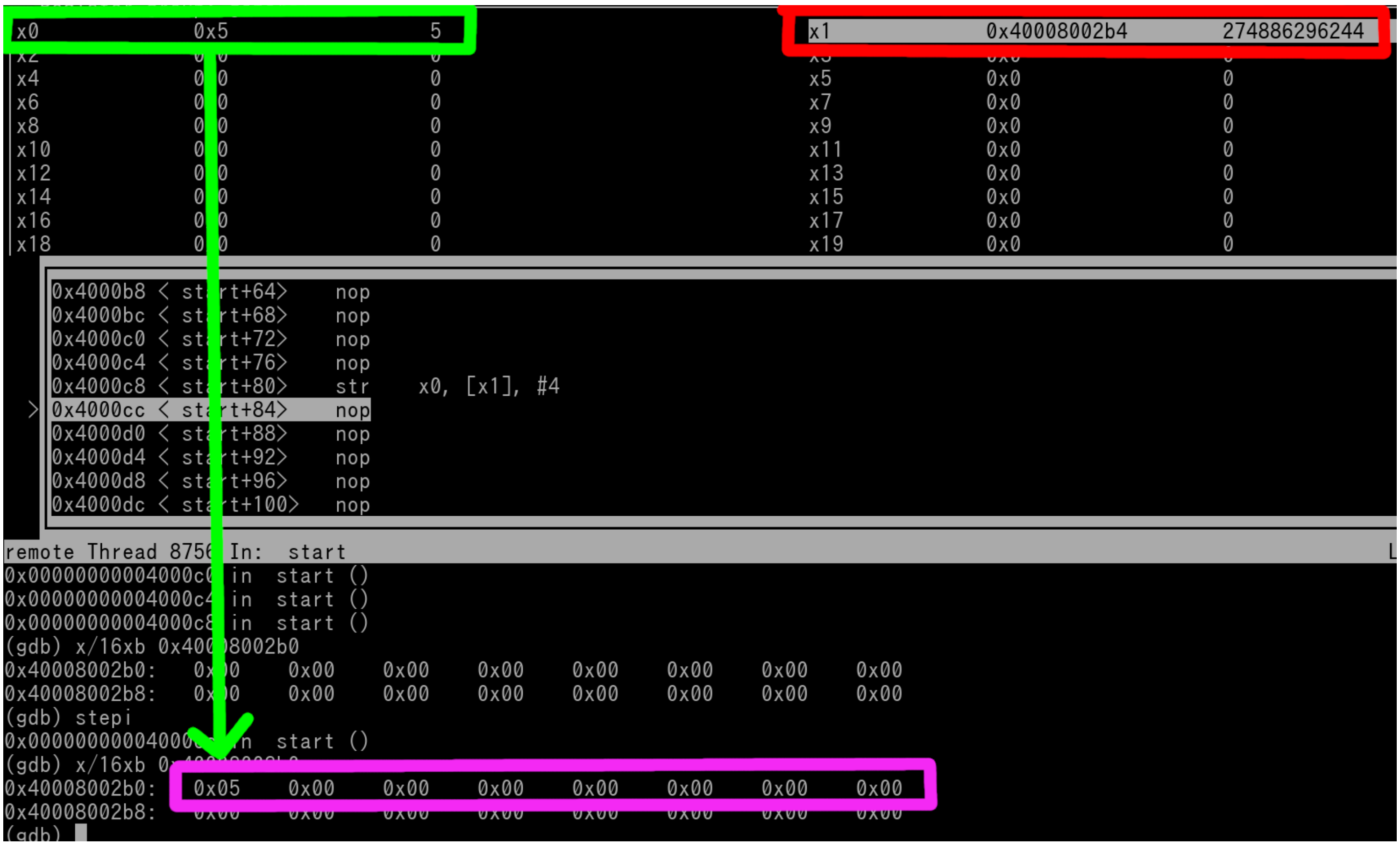
Here's a pic of right before the instruction (str x0, [x1], #0x4) is executed which also includes the view of 16 bytes of Memory starting at 0x40008002B0.



```
x0             0x5                 5                  x1             0x40008002b0          274886296240
x4             0x0                 0                  x5             0x0                   0
x6             0x0                 0                  x7             0x0                   0
x8             0x0                 0                  x9             0x0                   0
x10            0x0                 0                  x11            0x0                   0
x12            0x0                 0                  x13            0x0                   0
x14            0x0                 0                  x15            0x0                   0
x16            0x0                 0                  x17            0x0                   0
x18            0x0                 0                  x19            0x0                   0

   0x4000b8 < start+64>    nop
   0x4000bc < start+68>    nop
   0x4000c0 < start+72>    nop
   0x4000c4 < start+76>
 > 0x4000c8 < start+80>    str     x0, [x1], #4
   0x4000cc < start+84>
   0x4000d0 < start+88>    nop
   0x4000d4 < start+92>    nop
   0x4000d8 < start+96>    nop
   0x4000dc < start+100>   nop

remote Thread 8756 In:  start
0x00000000004000ac in  start ()
0x00000000004000b0 in  start ()
0x00000000004000b4 in  start ()
0x00000000004000b8 in  start ()
0x00000000004000bc in  start ()
0x00000000004000c0 in  start ()
0x00000000004000c4 in  start ()
0x00000000004000c8 in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

You can see that x5 (outlined in **green**) is 5 and x1 (outlined in **red**) is 0x40008002B0. The double-word outlined in **magenta** is where x0 will be stored to when the instruction (outlined in **blue**) does execute. We can see in the pic, the **magenta** outlined contents in memory are all zero. Once the instruction executes, those values will be replaced.

Okay let's say we execute the instruction now. Here's the pic.



```
x0             0x5                 5                  x1             0x40008002b4          274886296244
x4             0x0                 0                  x5             0x0                   0
x6             0x0                 0                  x7             0x0                   0
x8             0x0                 0                  x9             0x0                   0
x10            0x0                 0                  x11            0x0                   0
x12            0x0                 0                  x13            0x0                   0
x14            0x0                 0                  x15            0x0                   0
x16            0x0                 0                  x17            0x0                   0
x18            0x0                 0                  x19            0x0                   0

   0x4000b8 < start+64>    nop
   0x4000bc < start+68>    nop
   0x4000c0 < start+72>    nop
   0x4000c4 < start+76>    nop
   0x4000c8 < start+80>    str     x0, [x1], #4
 > 0x4000cc < start+84>    nop
   0x4000d0 < start+88>    nop
   0x4000d4 < start+92>    nop
   0x4000d8 < start+96>    nop
   0x4000dc < start+100>   nop

remote Thread 8756 In:  start                                                                                L
0x00000000004000c0 in  start ()
0x00000000004000c4 in  start ()
0x00000000004000c8 in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) stepi
0x00000000004000cc in  start ()
(gdb) x/16xb 0x40008002b0
0x40008002b0:   0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x40008002b8:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

We can see that x1 has been incremented by 4 (to 0x40008002B**4**). We also see that x0's contents were stored at 0x40008002B**0** ***BEFORE*** x1 was incremented by 0x4. The **green** arrow shows how the contents of x0 (outlined in **green**) were stored to 0x40008002B**0** (outlined in **magenta**). Because of Little Endian, the value was changed to 0x**05**00000000000000 when the store actually occurred to Memory.

Now let's cover pre and post index loads...

Example Pre-Index Load:
ldr w5, [x16, #0x3C]!

What occurs:

1. x16 + 0x3C = Effective Address
2. Word value located at Effective Address is loaded into w5.
3. x16 is then incremented by 0x3C, it now has a new value if the instruction gets executed again

Example Post-Index Load:
ldr w5, [x16], #0x3C

What occurs:

1. x16 = Effective Address
2. Word value located at Effective Address is loaded into w5.
3. x16 is then incremented by 0x3C, it now has a new value if the instruction gets executed again

Chapter 12: Loops

Loops are a set of instructions used to transfer (copy-paste) a chunk of data from one area of memory to another. You will need 4 items to write a loop~

1. Start Address where Contents are originally at
2. Start Address where you want Contents to be copied to
3. Length of the Contents which will involve a GPR being used as a "loop tracker" (how many times the loop will execute)
4. Conditional Branch

When writing a loop, it depends on *how* you want to transfer it. Do you want to transfer byte per byte, or a double-word at a time, etc etc. Consider the size of the contents. If the size of the contents can be evenly divided by 8, then transferring by double-words makes sense. If the size is 29 bytes in total, then transferring via bytes would make sense.

Let's pretend we need to transfer 29 bytes. Well first we need to define the start address of the where the 29 bytes are currently at, and then we need to define the start address of where the 29 bytes need to be copy-pasted to.

Let's pretend the bytes are currently located at 0x40007F0000. And we need to copy-paste them to starting location of 0x4000800000. 0x40007F0000 is known as the Source Address. 0x4000800000 is known as the Destination Address. Let's place the Source Address in x1, and the Destination Address in x2.

```
// Write x1 (Source) Address,
mov x1, #0x4000000000
movk x1, #0x007F, lsl #16
```

```
// Write x2 (Destination) address
add x2, x1, #0x10000 //This exceeds the 12-bit UIMM range, but due to "Extra Instruction Rotation" feature, this value is possible. More info on this feature in Chapter 15.
```

Okay address's are set, now we need a Register to be used as a Loop iteration/count tracker. Since we are transferring byte-per-byte and the contents is **29** bytes is size, we need to set a register to the value of **29**. We'll use w3.

```
// Set Loop Tracker
mov w3, #29
```

To copy-paste the data from one location to another, we will need load instructions (to load the data into register) and we will need a store instruction (to store the loaded data to the new memory location). We will also need a scrap register that is used for the temporary placement of the data while it is being transferred. We'll just use w4 for that.

Not only that, every time we transfer a byte we need to update (increment) the address value in both w1 and w2. We can use post-indexed loads and stores for this using Immediate Value of 1. We use the Value of 1 since every consecutive byte in memory is separated by the Address Value of 1. With all that being said here all the two instructions (load & store)

```
ldrb w4, [x1], #1 // Load byte
strb w4, [x2], #1 // Store byte
```

We'll almost done writing the loop, we need to subtract 1 from our Loop tracker (w3) and then place a conditional branch. It can be done like this...

```
sub w3, w3, #1 // Decrement Loop Tracker for every Byte Transferred
cmp w3, 0 // Check when Tracker hits 0
bne loop // When not zero, we still have bytes to transfer
```

We have introduced a label (loop) since we have a conditional branch. Where would the 'landing spot' be for this conditional branch? Well it has to be at the ldrb instruction. We do not want to put it at the very start (where we set the addresses) as this will make the loop execute infinitely (forever) because the post-indexed (updating) addresses will constantly be resetting.

```
loop:
ldrb w4, [x1], #1 // Load byte
strb w4, [x2], #1 // Store byte
sub w3, w3, #1 // Decrement Loop Tracker for every Byte Transferred
cmp w3, #0 // Check when Tracker hits 0
bne loop // When not zero, we still have bytes to transfer.
```

Now there's one more thing we can do, some instructions (such as sub) can have an "**s**" appended to it (***subs***), and we will then get a free use of 'cmp rD, 0'. This allows us to remove the 'cmp w3, 0' instruction.

```
subs w3, w3, #1 //preform the sub instruction and then preform "cmp w3, 0"
bne loop
```

Now let's piece together the entire code...

```
----------
// Write x1 (Source) Address
mov x1, #0x4000000000
movk x1, #0x007F, lsl #16
```

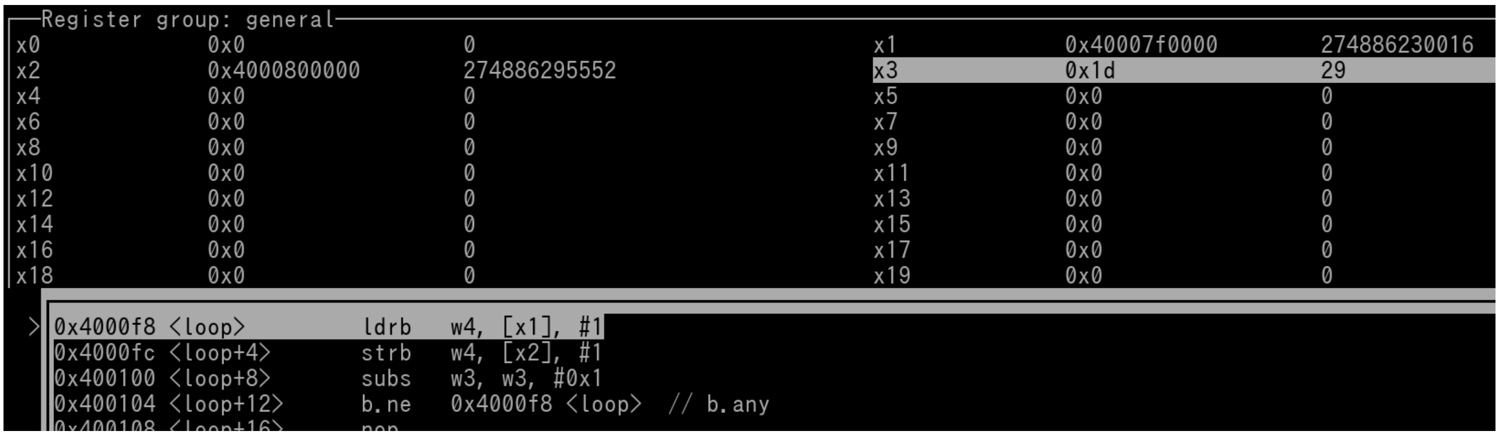
// Write x2 (Destination) Address
add x2, x1, #0x10000 //This exceeds the 12-bit UIMM range, but due to "Extra Instruction Rotation" feature, this value is possible. More info on this feature in Chapter 15.

// Set Loop Tracker
mov w3, #29

// Loop
loop:
ldrb w4, [x1], #1 // Load byte
strb w4, [x2], #1 // Store byte
subs w3, w3, #1 // Decrement Loop Tracker for every Byte Transferred, and Compare w3 to 0
bne loop // When not zero, we still have bytes to transfer.

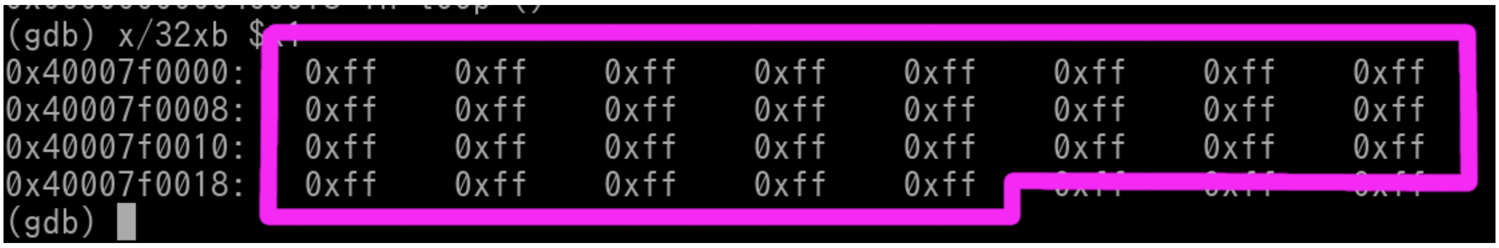
Congratz, you've made your first loop in ARM64.

If this doesn't make much sense to you, visual representations can greatly help. Here's a pic where the 3 mov instructions have already been executed. Therefore the Source Address, Destination Address, and Loop Tracker are all set.

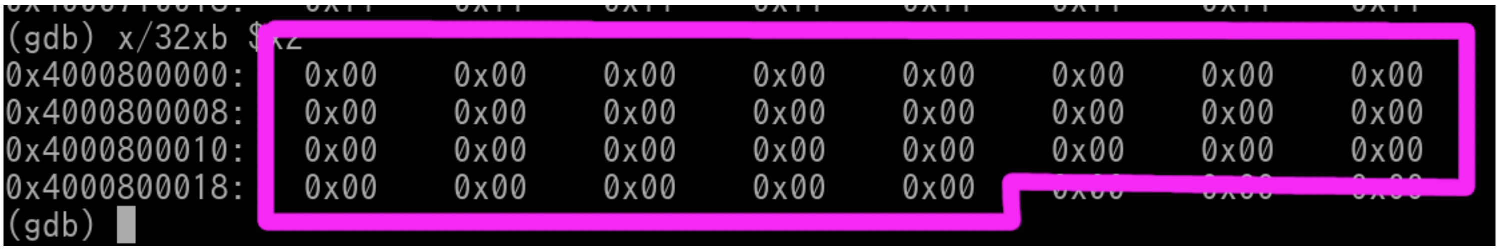

```
┌─Register group: general─────────────────────────────────────────────────────────────────────
│x0             0x0                 0                              x1             0x40007f0000        274886230016
│x2             0x4000800000        274886295552                   x3             0x1d                29
│x4             0x0                 0                              x5             0x0                 0
│x6             0x0                 0                              x7             0x0                 0
│x8             0x0                 0                              x9             0x0                 0
│x10            0x0                 0                              x11            0x0                 0
│x12            0x0                 0                              x13            0x0                 0
│x14            0x0                 0                              x15            0x0                 0
│x16            0x0                 0                              x17            0x0                 0
│x18            0x0                 0                              x19            0x0                 0

  > 0x4000f8 <loop>         ldrb   w4, [x1], #1
    0x4000fc <loop+4>       strb   w4, [x2], #1
    0x400100 <loop+8>       subs   w3, w3, #0x1
    0x400104 <loop+12>      b.ne   0x4000f8 <loop>  // b.any
    0x400108 <loop+16>      nop
```

We can see in the pic that to the left the ldrb instruction is the term "loop". The Debugging Tool used in the pic can recognize branch labels that are from a Source File.

We see that x1, x2, and x3 (w3) are all set. Let's view the 29 bytes of memory at the Source Address.


```
(gdb) x/32xb $
0x40007f0000:   0xff    0xff    0xff    0xff    0xff    0xff    0xff    0xff
0x40007f0008:   0xff    0xff    0xff    0xff    0xff    0xff    0xff    0xff
0x40007f0010:   0xff    0xff    0xff    0xff    0xff    0xff    0xff    0xff
0x40007f0018:   0xff    0xff    0xff    0xff    0xff    0xff    0xff    0xff
(gdb)
```

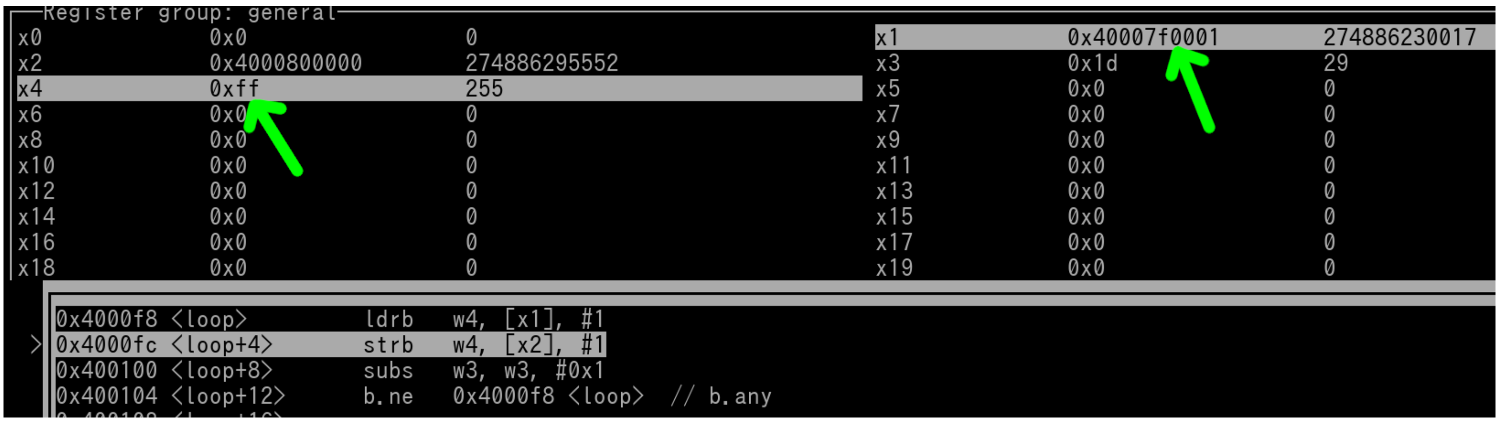
We can see we have a string of 0xFF's (outlined in **magenta**) that will be loaded (byte per byte) into the loop. Let's view the 29 bytes of memory at the Destination Address


```
(gdb) x/32xb $x2
0x4000800000:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800008:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800010:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800018:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

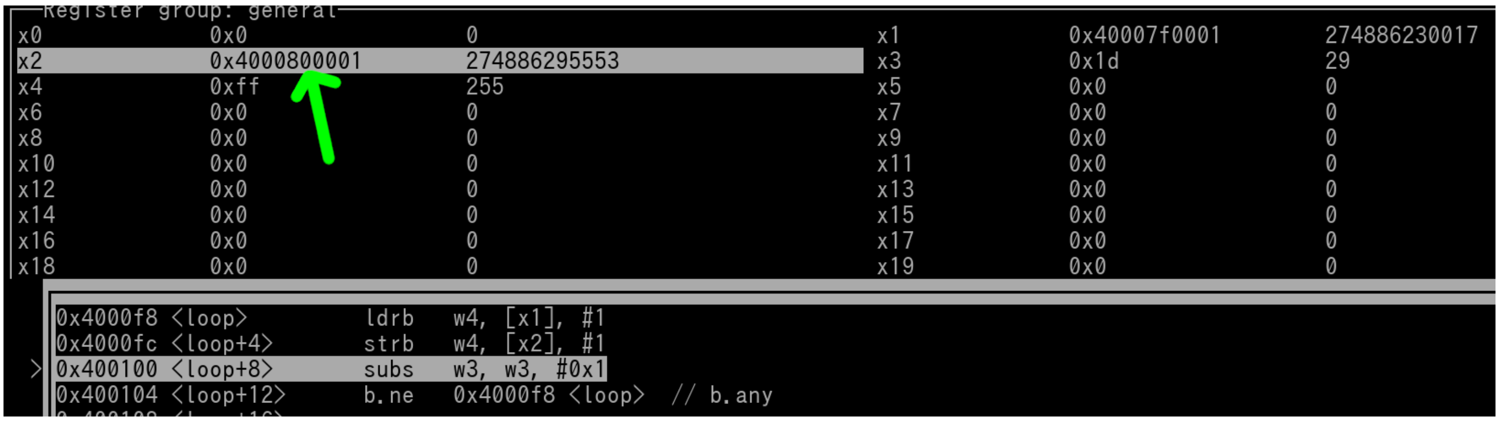
We can see we have a string of zeros (outlined in magenta) there. When the loop has been executed for all 29 times, those zeros will all be replaced with 0xFF's.

The following 4 pics will cover 1 iteration of the loop. In the last pic of the 4, we will be back at the ***ldrb*** instruction ready to execution the loop for a 2nd time.

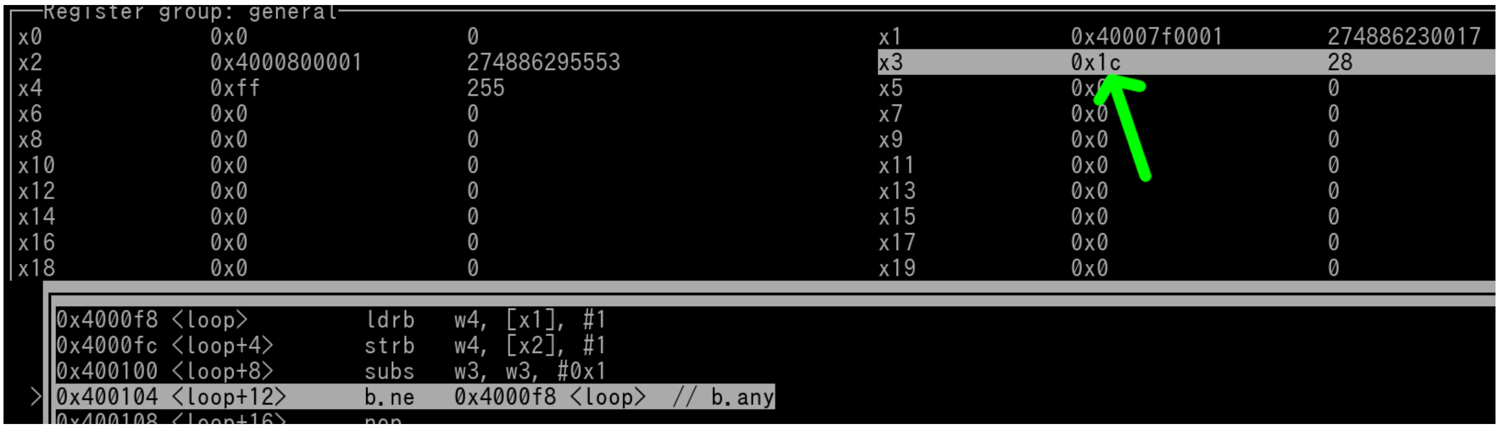
Pic 1/4:
ldrb has been executed, **w4** gets loaded with 0xFF (indicated by the green arrow), since ldrb was a post index load, **x1** gets incremented by 1 after the load occurred which is also indicated by a green arrow):


```
Register group: general
x0             0x0                 0                        x1             0x40007f0001        274886230017
x2             0x4000800000        274886295552             x3             0x1d                29
x4             0xff                255                      x5             0x0                 0
x6             0x0                 0                        x7             0x0                 0
x8             0x0                 0                        x9             0x0                 0
x10            0x0                 0                        x11            0x0                 0
x12            0x0                 0                        x13            0x0                 0
x14            0x0                 0                        x15            0x0                 0
x16            0x0                 0                        x17            0x0                 0
x18            0x0                 0                        x19            0x0                 0

   0x4000f8 <loop>         ldrb   w4, [x1], #1
>  0x4000fc <loop+4>       strb   w4, [x2], #1
   0x400100 <loop+8>       subs   w3, w3, #0x1
   0x400104 <loop+12>      b.ne   0x4000f8 <loop>  // b.any
```

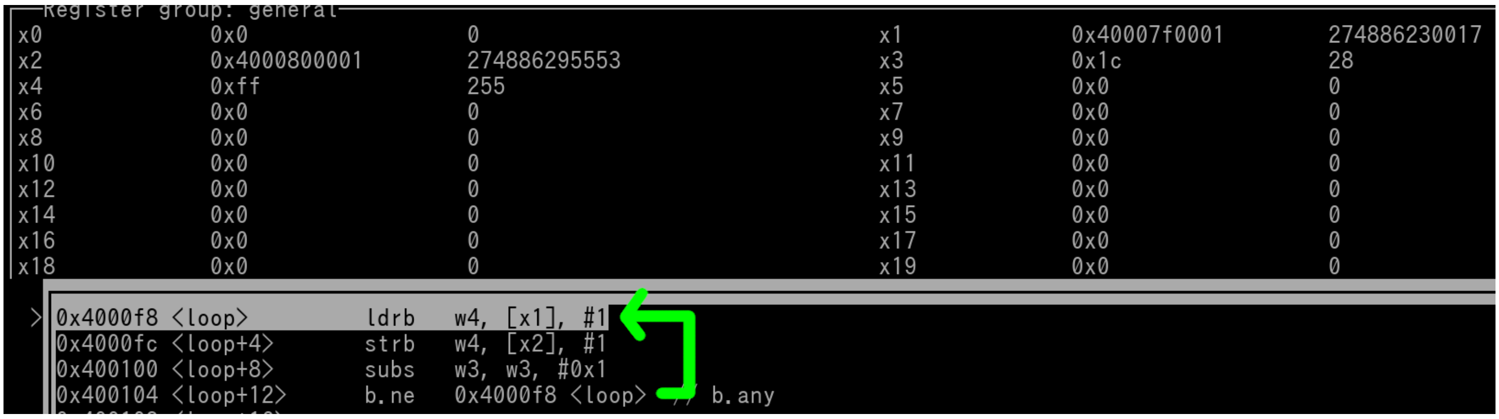
Pic 2/4:
strb has been executed, 0xFF gets stored to 0x4000800000 (memory will be shown later), since strb was a post index store, x2 gets incremented by 1 after the store occurred, which is indicated by a green arrow.


```
Register group: general
x0             0x0                 0                        x1             0x40007f0001        274886230017
x2             0x4000800001        274886295553             x3             0x1d                29
x4             0xff                255                      x5             0x0                 0
x6             0x0                 0                        x7             0x0                 0
x8             0x0                 0                        x9             0x0                 0
x10            0x0                 0                        x11            0x0                 0
x12            0x0                 0                        x13            0x0                 0
x14            0x0                 0                        x15            0x0                 0
x16            0x0                 0                        x17            0x0                 0
x18            0x0                 0                        x19            0x0                 0

   0x4000f8 <loop>         ldrb   w4, [x1], #1
   0x4000fc <loop+4>       strb   w4, [x2], #1
>  0x400100 <loop+8>       subs   w3, w3, #0x1
   0x400104 <loop+12>      b.ne   0x4000f8 <loop>  // b.any
```

Pic 3/4:
subs has been executed, w1 decrements by 1 down to 28 (indicated by green arrow), "cmp w3, #0" is also done

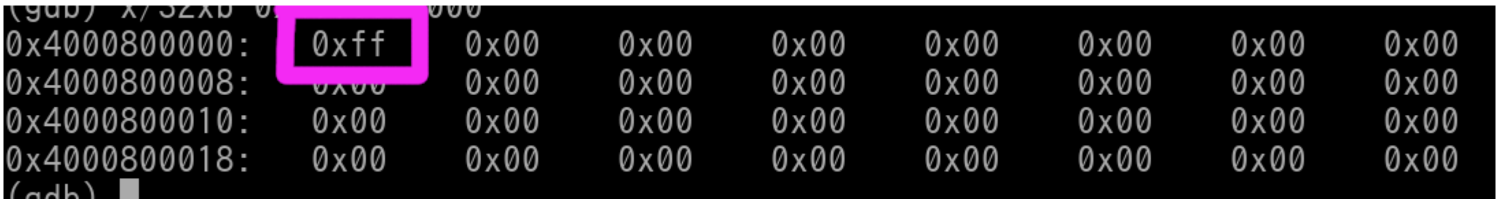

```
Register group: general
x0             0x0                 0                        x1             0x40007f0001        274886230017
x2             0x4000800001        274886295553             x3             0x1c                28
x4             0xff                255                      x5             0x0                 0
x6             0x0                 0                        x7             0x0                 0
x8             0x0                 0                        x9             0x0                 0
x10            0x0                 0                        x11            0x0                 0
x12            0x0                 0                        x13            0x0                 0
x14            0x0                 0                        x15            0x0                 0
x16            0x0                 0                        x17            0x0                 0
x18            0x0                 0                        x19            0x0                 0

   0x4000f8 <loop>         ldrb   w4, [x1], #1
   0x4000fc <loop+4>       strb   w4, [x2], #1
   0x400100 <loop+8>       subs   w3, w3, #0x1
>  0x400104 <loop+12>      b.ne   0x4000f8 <loop>  // b.any
```

Pic 4/4:
bne has been executed, since w1 hasn't hit 0 yet execution of CPU goes back up to the ldrb instruction (indicated by the green arrow). Next iteration of loop is now ready to be performed



```
-Register group: general-
x0             0x0                 0                              x1             0x40007f0001        274886230017
x2             0x4000800001        274886295553                   x3             0x1c                28
x4             0xff                255                            x5             0x0                 0
x6             0x0                 0                              x7             0x0                 0
x8             0x0                 0                              x9             0x0                 0
x10            0x0                 0                              x11            0x0                 0
x12            0x0                 0                              x13            0x0                 0
x14            0x0                 0                              x15            0x0                 0
x16            0x0                 0                              x17            0x0                 0
x18            0x0                 0                              x19            0x0                 0

 > 0x4000f8 <loop>       ldrb   w4, [x1], #1
   0x4000fc <loop+4>     strb   w4, [x2], #1
   0x400100 <loop+8>     subs   w3, w3, #0x1
   0x400104 <loop+12>    b.ne   0x4000f8 <loop>  // b.any
```

Okay we have went through one full iteration of the loop. Let's now take a look at memory at the Destination Address....



```
(gdb) x/32xb 0x4000800000
0x4000800000:   0xff    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800008:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800010:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4000800018:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

We can see that one byte of zero has been changed to 0xFF (outlined in **magenta**). This makes sense since the loop, at this point in time, has only been executed once. It's important to remember that once enough loop iterations have occurred to where w3 = 0, then the conditional branch will no longer be taken. Thus ending the loop.

Alright, at this point in your Assembly Journey, you have learned basic integer, load, store, compare, and branch instructions. You've also learned how to write basic loops. This alone allows you create all sorts of code now. Let's do an exercise where you will perform a basic Fibonacci Sequence. This exercise is a great learning curve for the Beginner. If you are unfamiliar with Fibonacci, it works like this..

0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
etc.. etc.. etc..

As you can see, there is a definitive pattern. First, you start with 0 and add 1 to it. Of course that result is 1. However after this point, you must take the previous result and add it with the latest result. Which would be 1 + 1. This new result is 2. Now let's keep applying this same concept. We take the previous result and add it with the latest result. Therefore the addition is now 1 + 2, which results in 3. Do this again, the addition is now 2 + 3, which is 5. Again, and it's 3 + 5 = 8. So on, and so forth.

We will write a piece of code that will perform this Fibonnaci sequence starting at 0. However, we will put in a twist. We want the Fibonacci sequence to *STOP* once the result of the addition is higher than 1,000. This "twist" now implements a condition environment in our code. Thus meaning we will need a compare and conditional branch instruction at a minimum. Not only that, because we are adding a current result with a previous result, we should be implementing the code into a loop.

We will need at least the following items in our code...

- A register that holds the first variable of the addition. This register will begin as 0.
- A register that holds the second variable of the addition. This register will begin as 1.
- A register that holds the result of the addition.
- A conditional branch that goes backwards (Loop)

Let's view the entire finished Source first, and then dissect it...

```
mov w0, #0 //Variable #1
mov w1, #1 //Variable #2
do_fibonacci:
add w2, w0, w1 //Perform the addition. Result goes into w2
cmp w2, #1000 //Compare result to 1,000
bhi done //If greater than 1000, *stop* the Fibonacci sequence
mov w0, w1 //Previous result now becomes Variable #1
mov w1, w2 //Latest result now becomes Variable #2, therefore the next addition will be Latest Result + Previous
Result.
b do_fibonacci //Do the Fibonacci sequence again
done: //End of code
```

Let's begin the dissection.

```
mov w0, #0
mov w1, #1
```

These first two instructions are the easiest to understand, they are the two variables used in our addition operation. They have to start as 0 and 1. Simple enough.

```
do_fibonacci:
add w2, w0, w1
```

I'll explain the branch label later, let's focus on the ***add*** instruction for now. This is the Fibonacci sequence being performed. We add our two variables and get a result. We need our result in a 3rd register (w2), because when we replace the variables for the next addition, there is a certain procedure to ensure we properly update both variables. Thus we are using w2 as a 'scratch/temp' register.

```
cmp w2, #1000
bhi done
```

Remember that I said we will stop the Fibonacci sequence when the result of the addition yields a number higher than 1000. In order to check for this condition, we need a compare (***cmp***) instruction. Obviously, we will compare the addition result against the value of 1,000 in the ***cmp*** instruction. Below the ***cmp*** instruction is a ***bhi*** conditional branch. Why did we use ***bhi*** instead of ***bgt***? Well for starters, we know for certain our addition result will always be positive since the very first Fibonacci addition results in 1 and can only increase from there. Therefore, since negative numbers are *NOT* possible, we should be treating our values as *Unsigned*. Since we are working with Unsigned values, we should be using ***bhi*** over ***bgt***.

For conditional branches such as bgt, blt, bhi, blo, etc etc, you should *default* to using bhi, blo, etc, *unless* you know negative numbers are possible, *and* you want the negative numbers to be treated as negative.

Alright moving on...

mov w0, w1
mov w2, w1

Now this is the portion of code that may get confusing for the Beginner. Once we have our addition result, we need to find a way for the next addition operation to be the previous result + latest result. If we don't do this, then the Fibonacci sequence won't work and the addition result won't increase every time it's done. The first ***mov*** instruction will make the former latest result become the previous result. The second ***mov*** instruction will make update the latest result (w1 ofc). It also must be done in this order or else w2 (latest result) will be written to both w0 and w1. This will cause us to lose the previous result variable.

b do_fibonacci

Once we have updated both variables, we simply need to execute the Fibonacci sequence again. This is an unconditional branch because we already checked against the value of 1000 earlier in the above conditional branch. Meaning if the execution of our code is at this spot, we already know we HAVE to do the Fibonacci addition again. So the question at hand is... where do we need to unconditional branch to land at? Simple, it needs to land at the ***add*** instruction, so we can start the Fibonacci sequence again. This is why we have this...

do_fibonacci:
add w2, w0, w1

You do *NOT* want the do_fibonacci label at the very start of the source. If so, it will simply reset the Fibonacci variables to 0 & 1 over and over and over again. What happens if this occurs? Well, the w2 register will NEVER become greater than 1000. What happens if that occurs? You get an infinite loop!

Alrighty, let's address the very last item of the Source.

done:

The ***done:*** label is for when w2 is greater than 1000 (unsigned greater than). This is at the very bottom of our source, since the very bottom is the very end. There is no more code left to execute at this spot in the source. We are finished with the Fibonacci sequence, congratulations.