Computer stores memory with “bits”, and 1 bit only has 2 situations: 0 (Clear), 1(Set).

We always group 4 bits, just like “0100”, that group is called “Nibble(Half a Byte)”.

And we can combine 4 number in group with 16 different situations like below:

***0000***

***0001***

***0010***

***0011***

***0100***

***0101***

***0110***

***0111***

***1000***

***1001***

***1010***

***1011***

***1100***

***1101***

***1110***

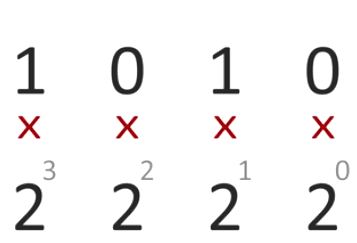
***1111***

And by the way, Here is a tutorial for Binary numbers’ calculating:

Firstly, we need to know how to convert a Binary to Decimal or a Hexadecimal number.

***For example,***

***Binary 1010 means,***



So we can compute that formula:

***Thus Binary “1010” = Decimal “10”***

***For example,***

***Hexadecimal 9F means,***

***Thus Hexadecimal “9F” = Decimal “159”***

But we can see that is very hard to remember all the 16 Binary combination, so we can make them easier by using different Hexadecimal digit represent them respectly, just like below table:

|  |  |
| --- | --- |
| Binary | Hexadecimal |
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

And you can easily see that remember a Hexadecimal digit “A” is mush easier that a binary nibble “1010”, but actually they are same.

In our life, we use Decimal System. So if we want to count numbers, we probably read like: 1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12………

But you see, counting in Hexadecimal is a little bit different, as shown below:

***1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, F;***

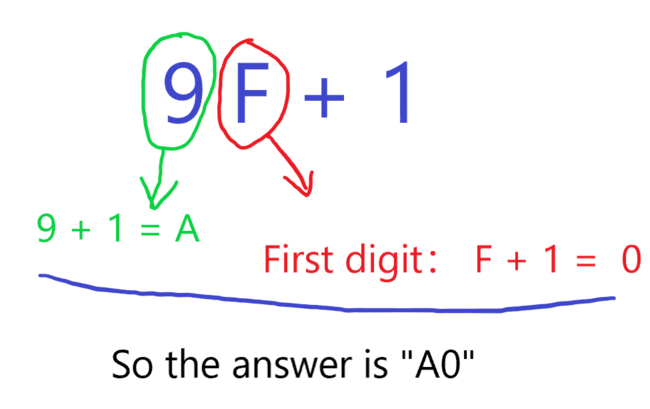
***10, 11, 12, 13, 14, 15, 1A, 1B, 1C, 1D, 1F;***

***20, 21, 22, 23, 24,25, 2A, 2B, 2C, 2D, 2F……***

And we all know how to calculate 9 + 1 in Decimal, 9 + 1 = 10. But in Hexadecimal,

9F + 1 do not equal to 100, just check the steps below, it will shows how to compute that expression.

***9F + 1***



And we already know that Hexadecimal “9F” = Decimal “159”, we can compute 159 + 1 = 160. So we can do a little verification, let us transfer Hexadecimal “A0” to a Decimal number and the process is as follows expression:

Of course, they are same, A0 = 9F + 1 = 159 + 1 = 160.

After so many strenuous steps to convert numbers from different system, we can sure about that, All the number system are just varieties way to represent numbers, they are exactly same.

We’ll be working with Hexadecimal more in this tutorial, and I’ll be calling it “Hex” from now on, as it’s much smaller than Hexadecimal.

We already know 1 Nibble means “0000”, which has 4 bits. And we still need another sizes of data,

***2 Nibbles(e.g. 0000 1111) = 1 Byte = 8 Bits***

***2 Bytes(e.g. 0000 1000 0001 1111) = 1 Word***

***2 Words = 1 Long-Word***

And there must are many longer data, but what we exploring in M68K, and the 68K handles data only up to the size of “Long-Word”, and no higher. So we don’t really to care about the beyond data size.

Unlike human, Computer has a specific way to access or store memory, and it’s done through the “addressing”. As the below table shows, which is a example of how memory is stored in computer.

The white “bytes” we see in the table, is the “data” (just our memory). The purple digits at the top and left side are the “Offset”. A computer will use the Offset as an “address”, to locate a byte in memory that it wants to read, or even change.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000010 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

For example, we can see the below table, every memory is “00”, except one.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000010 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | F2 | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

We all noticed the byte in (Column B, Row 00000020) is “F2”, if computer want to access this byte, it will go to Offset 0000002B.

Look the highlight cube 0000000A, we can write the “long-word” 20 4F F3 A0.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 20 | 4F | F3 | A0 | 00 | 00 |
| 00000010 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | F2 | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

And you can see the long-word“20 4F F3 A0” has been split up to 4 cube. “20” is in 0000000A, and “4F” is in 0000000B, “F3” is in 0000000C, and “A0” is in 0000000D.

They are highlighted. Then we can know the long-word not only can be read at 0000000A, but also at 0000000B, 0000000C, 0000000D.

The Motorola 68000 (sometimes shortened to Motorola 68k or m68k and usually pronounced "sixty-eight-thousand") is a 16/32-bit complex instruction set computer (CISC) microprocessor, introduced in 1979 by Motorola Semiconductor Products Sector.

The design implements a 32-bit instruction set, with 32-bit registers and a 16-bit internal data bus. The address bus is 24 bits and does not use memory segmentation, which made it easier to program for. Internally, it uses a 16-bit data arithmetic logic unit (ALU) and two more 16-bit ALUs used mostly for addresses, and has a 16-bit external data bus. For this reason, Motorola termed it a 16/32-bit processor.

As one of the first widely available processors with a 32-bit instruction set, large unsegmented address space, and relatively high speed for the era, the 68k was a popular design through the 1980s. It was widely used in a new generation of personal computers with graphical user interfaces, including the Macintosh 128K, Commodore Amiga, Atari ST, and Sharp X68000. The 1988 Sega Genesis/Mega Drive console is powered by a 68000.

Later processors in the Motorola 68000 series, beginning with the Motorola 68020, use full 32-bit ALUs and have full 32-bit address and data buses, speeding up 32-bit operations and allowing full 32-bit addressing rather than the 24-bit addressing of the 68000 and Motorola 68010 or the 31-bit addressing of the Motorola 68012. The original 68k is generally software forward-compatible with the rest of the line despite being limited to a 16-bit wide external bus.

After 44 years in production, the 68000 architecture is still in use.[From Wikipedia]

The CPU we used is Motorola 68000, from now on, we call it “M68K”.M68K has a set of instructions, thus we can use these instructions to rule the CPU.

We all know that Computer work with Binary inside, there is a instruction below for example:

***0011 0000 0011 1100 0000 0100 1111 0000***

And Binary is very difficult for human, not only to read but also to remember, so there is where assembly language comes in. We could use some kinds of “Mnemonics” to simply above Binary code as below shown:

***move.w #$04F0,d0***

And we use different colors to mark the code, their meanings shown as below:

***Command: that means a short word such as, move, add, sub, divu….. And every of them was designed to help user to deal with some mathematical calculation or program flow change.***

***Size: that means the specific size of you selected, you ca have, “.b” for Byte, “.w” for Word, “.l” for Long-Word.***

***Source Operand: where the value is being read from or what value to be used.(Where are they come from?)***

***Destination Operand: Where the Value is being moved or manipulated to.(Where are they going to?)***

As usual, an example can help us to understand, as below shown:

***move.b #$2c,$000001E***

The meaning of above code is “move Byte 2C into memory’s offset 0000001E”, and after M68K processes above code, the memory will be shown as below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000010 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 2c | 00 |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

Of course, M68k will convert the assembly language to Binary then processes it.

By the way, Did we mention there are 2 symbol what we still don’t understand: “#” and “$”.

The “$” symbol means follow digit is a Hexadecimal Digit. If you didn’t add “$” just like below code:

***move.b #32,$000001E***

The assembler will recognize 32 as a Decimal number and convert it to Binary one when assembler assembles it, we know that Decimal “32” = Binary “0010 0000” = Hexadecimal “20”. So you can write below 3 codes:

***move.b #$20,$000001E***

***move.b #32,$000001E***

***move.b #%00100000,$000001E***

All 3 codes above are exactly same, they just use different Number system, “$” means Hexadecimal and “%” means Binary. Though they are same you can use any of them, using Hexadecimal and Binary can help us to understand assembly language well.

And don’t forget symbol “#”, it’s a symbol which can tell assembler follow number is an “immediate” value, and not a value. About “immediate ” value, we will explain that later. Then let us see what happen if there isn’t “#” in code:

***move.b $00000010,$000002D***

This code will read the byte at offset 00000010, and copy it to offset 0000002D, if the byte at offset 00000010 was 49, then 0000002D will now also be 49:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00000000 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000010 | 49 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 49 | 00 | 00 |
| 00000020 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

Then let us explain the “immediate” value, frankly speaking, it’s just a profound name of “raw number”, the symbol “#” tells M68K that number is not an offset/address.

Registers are memory spaces of processer, and different processors have different types of registers for different reasons. They’re used for moving, adding, subtracting, numbers quickly and effectively to and from places. We’re going to look at the common 16 registers of the 68k and what they can do.

M68K has 8 Data Register, their name are: d0, d1, d2, d3, d4, d5, d6, d7.

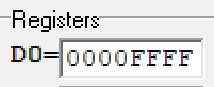
“Data registers” are used for storing and changing numbers mathematically, each register has Long-Word spaces:

|  |  |
| --- | --- |
| Register Name | Space (in Hexadecimal) |
| d0 | 00 00 00 00 |
| d1 | 00 00 00 00 |
| d2 | 00 00 00 00 |
| d3 | 00 00 00 00 |
| d4 | 00 00 00 00 |
| d5 | 00 00 00 00 |
| d6 | 00 00 00 00 |
| d7 | 00 00 00 00 |

Using the previous instruction as an example:

***move.w #$FFFF,d0***

This will move the word “FFFF” into data register d0, after the instruction is processed by the 68k, d0 will contain 00 00 FF FF as below picture.



By the way, “.b” for Byte, “.w” for Word, “.l” for Long-Word.

Also, the M68K has another set of 8 registers called “address” registers, and are named: a0, a1, a2, a3, a4, a5, a6, a7.

As same as Data Register, each Adress Register has Long-Word spaces:

|  |  |
| --- | --- |
| Register Name | Space (in Hexadecimal) |
| a0 | 00 00 00 00 |
| a1 | 00 00 00 00 |
| a2 | 00 00 00 00 |
| a3 | 00 00 00 00 |
| a4 | 00 00 00 00 |
| a5 | 00 00 00 00 |
| a6 | 00 00 00 00 |
| a7 | 00 00 00 00 |

Adress Register can be used in much the same way as Data Register.