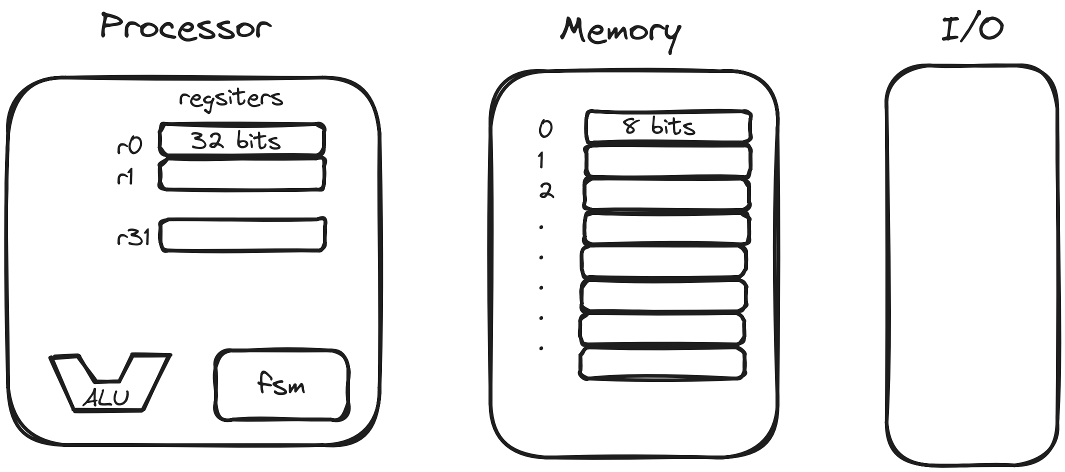
From our current perspectives, we can think about a computer consisting of three things:



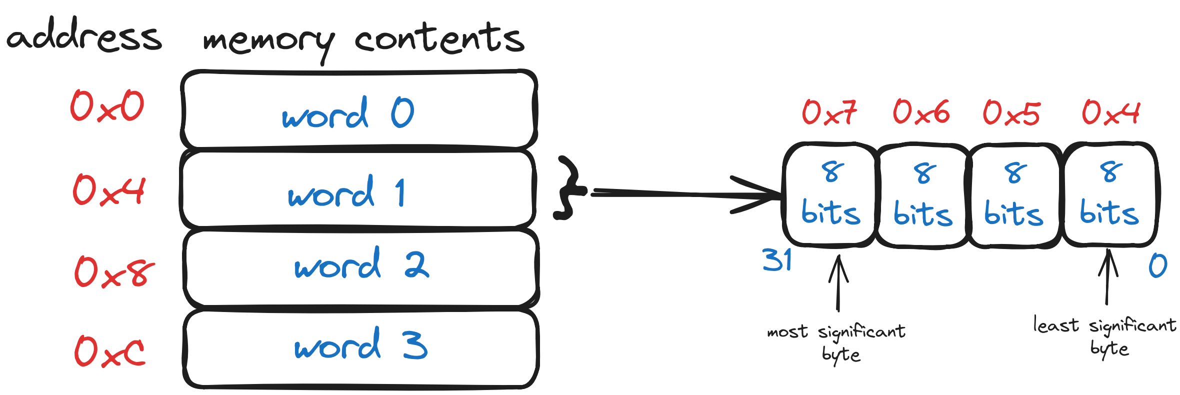
- a processor that has registers, something that can compute things, and something that can control things
- memory that contains that contains the program itself, and the data for the program
- input and output that connect the computer to other things

The Nios II is a **32-bit processor**. This means:

- the registers in the processor are 32 bits wide
- memory addresses are 32 bits wide, so there are $2^{32} - 1 \approx 4 \text{ billion}$ addresses
- the *word size* of the processor is 32 bits, meaning it can access 32 bits, or 4 bytes, of memory at once even though each address of memory holds only 1 byte.

Because the word size of the processor is 32 bits, we typically access memory 4 bytes at a time. This means that the start of "word 1" will be 4 addresses after the start of 'word 0'. See below:

A word comprises 4 bytes.



Note: `0x` is a prefix used to signify a hexadecimal number following it, i.e. $0\text{x}C = (OC)_{16}$.

Note that the least significant byte is at the lowest address. This is a convention used by the Nios II called "little endian".

To observe how memory works, lets look at a program to add three numbers 10, 20, 30, that are placed into memory as part of the program.

In C, this would look like:

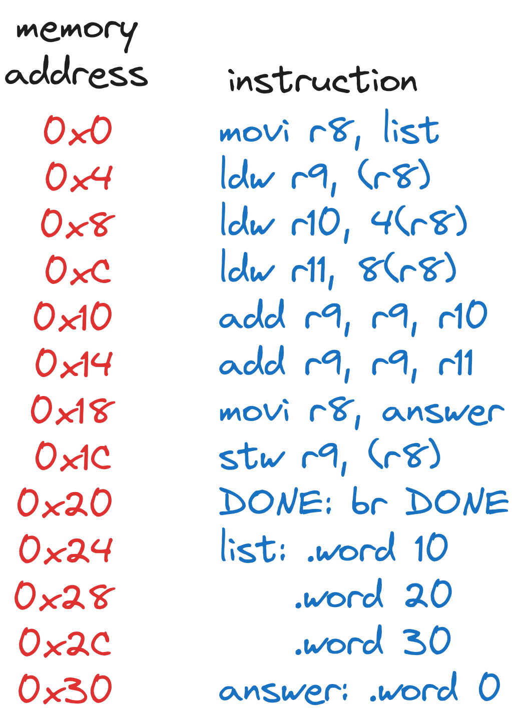
```c
int list[3] = {10, 20, 30};
int answer = list[0] + list[1] + list[2];
```

It would look like this in assembly (Nios II):

```
movi r8, list
ldw r9, (r8)
ldw r10, 4(r8)
ldw r11, 8(r8)
add r9, r9, r10
add r9, r9, r11
movi r8, answer
stw r9, (r8)
DONE: br DONE

list: .word 10
        .word 20
        .word 30
answer: .word 0
```

Lets break it down. Firstly, we need to understand that every single assembly instruction is stored in memory. Before the program **executes**, it **assembles** (similar to "compiling" in C/C++) where the instructions are "translated" and placed in memory. Here is an example:



First, its crucial to understand what `DONE`, `list`, and `answer` are. They are labels we can use in assembly language programming to refer to a memory address. In the above example, `list` is simply the place in memory that holds the instruction `.word 10`. In this example, `list` is equivalent to `0x24`. However since we cannot actually know the address when we are writing assembly, we use `list`.

This also explains the `br` instruction, which for now we can think of as telling to go to the instruction indicated by the label. `DONE: br DONE` tells the program to go to the instruction in `DONE`, which is itself, hence causing an infinite loop.

Next, its important to understand what the `.word` directive does. `.word 10` tells the computer to allocate a word sized piece of memory (4 bytes) and initialize it with the specified value, `10`. During assembly, *before execution*, the computer will translate this instruction and, for the example above, simply place the integer value `10` in the address `0x24`. Because this is done before execution, it does not matter that this instruction will never be executed. In fact, it is not executable, **they define data, not executable instructions.**

Finally, lets go through the code sequentially:

1. `movi r8, list`: places the address indicated by the label `list`, which is `0x24`, into register `r8`.
2. `ldw r9, (r8)`: `ldw` stands for "load word", and is used to load a word from memory. Keep in mind, `r8` is storing `0x24`. We would like to load `10`, which is stored in `0x24`, and not `0x24` itself. For this, we use the parentheses `()` around `r8`. This is similar to *dereferencing* a pointer in C/C++.
3. `ldw r10, 4(r8)`: loads another word from memory into register `r10`. `4(r8)` tells the computer to find the memory contents in the memory location `4` after `r8`, i.e. the next memory location (`0x24 + 4 = 0x28`). This will load `20` into `r10`.
4. `ldw r11, 8(r8)`: loads word from memory at a location 8 addresses after the value of `r8`. This will load `30` into `r11`.
5. `add r9, r9, r10`: stores `r9 + r10` in `r9`.
6. `add r9, r9, r11`: stores `r9 + r11` in `r9`.
7. `movi r8, answer`: places the address indicated by the label `answer`, which is `0x30`, into register `r8`.
8. `stw r9, (r8)`: `stw` stands for "store word". This will store the value of `r9`, which is now `60`, into the memory location of `r8`, which is `0x30`.