Chapter 2: Computer Architecture

Frank Vojtesek

February 11, 2023

Modern computer archictecture is based off of the **Von Neumann Architecture**, which:

- Divides the computer up into two parts: (1) the Central Processing Unit (CPU), and (2) memory
- Specifies that not only computer data should live in memory, but also the programs which control the computers operation

1 Structure of Computer Memory

Computer Memory is one or many physical chips on the computers mother board.

Internally it is organized as a numbered sequence of fixed size containers, each container contains a number.

- E.g., if a computer has 256 megabytes of memory, then your computer contains roughly 256 million fixed size storage locations
- Each storage location contains 1 byte (8 bits) of information
- I.e., a positive integer whose value is between `0` and `255` ($2^8 - 1$), usually represented in binary, e.g., `10011101`

All results of any computation performed on the computer is stored in memory.

All instructions for the CPU to follow are stored in memory also.

The only difference between the data and the functions is the context (information) that the CPU uses to understand the numbers stored in memory.

2 The CPU

The CPU takes care of interpretting the data and instructions stored in memory.

The CPU reads in instructions from memory one at a time and excutes them, this is called the **fetch-execute cycle**.

The CPU consists of the following elements:

- Program Counter
- Instruction Decoder
- Data bus
- General-purpose registers
- Arithemetic and logic unit

The **program counter** is used to tell the computer where to fetch the next instruction.

The program counter holds the memory address of the next instruction to be executed.

The CPU begins by looking at the program counter, and fetching whatever number is stored in memory at the location specified.

This number is then passed on to the **instruction decoder** and figures out what process needs to take place, and what memory locations are going to be involved in this process.

- These actions could include **addition**, **subtraction**, **multiplication**, **data movement**, etc...
- Computer instructions usually consist of both the actual instruction and the list of memory locations that are used to carry it out

Now the computer uses the **data bus** to fetch the memory locations used in the calculation

- The data bus is the connection between the CPU and memory
- It is thye actual wire that connects them
- If you look on the motherboard of a computer, the wires that go in and out from the memory are the data bus

In addition to the memory outside of the processsor, the processor itself has some special, high-speed memory locations called registers

- there are two kinds of registers: (1) **general registers**, and (2) **special-purpose registers**
- General registers are where most of the processing happens: e.g., addition, subtraction, multiplication, comparisons, etc...
- Computers have very few general purpose registers, most information is stored in the main memory, and brought into registers for processing, and then put back into memory when the processing is complete
- Special-purpose registers are registers which have very specific purposes, these will be discussed more when they come up

Once the CPU has retireved all of the data it needs, it passes on the data and decoded instruction to the **arithmetic and logic unit**, where the instruction is actually executed.

After the results of the computation have been calculated, the results are placed on the data bus and sent to the appropriate location in memory or a register, as specified by the instruction.

This description is simplified, and actual processors are quite a bit more complicated, more information can be found in the source text.

3 Some Terminology

Computer memory is a numbered sequence of fixed-size storage locations.

The number attached to each storage location is called it's **memory address**.

The size of a single storage location is called a **byte**.

On `x86` processors, a byte is a number between `0` and `255`.

You may be wondering how computers can display and use text, graphics, and even large numbers when all they can do is store numbers betwen `0` and `255`

The computer has specialized hardware like a graphics card with special interpretations of each number

- When displaying to the screen, the computer uses ASCII code tables to translate the numbers you are sending into letters to display on the screen
- In the case of ASCII, one number translates to exactly on character (unicode makes things a little bit more complicated)
- For example, the capital letter `'A'` is represented by the number `65` (`01000001`), the number character `'1'` is represented by the number `49` (`00110001`)
- In order to print out `"HELLO"`, you would need to give the computer the sequence of numbers `72, 69, 76, 76, 79`
- To print out the number `"100"`, you would need to give the computer the sequence of numbners `49, 48, 48`

In order to represent numbers larger than `255` we use combinations of bytes

- Suppose we wanted to represent the number `55642`, we could do this using two bytes (16 bits)

- The number `55642` is `1101100101011010` in binary, a 16-bit number
- We can represent it with the two 8-bit numbers by storing the top 8-bits in one memory address, and the bottom 8-bits in another
- So, we would be able to represent `55642` with `217` (`11011001`) and `90` (`01011010`)
- These two bytes written next two eachother make up a single 16-bit integer
- The CPU has hardware to handle the mathematics that goes into performing operations on combinations of bytes

We mentioned earlier that in addition to regular memory, the computer has special purpose high-speed memory locations called **registers**.

Registers keep the contents of the numbers that you are currently manipulating.

On the computers we are using, registers are each four bytes long.

The size of a typical register is called the computer's **word size**, `x86` processors have four-byte words.

This means that it is most natural for computers to do computations 4 bytes at a time.

Memory addresses are also four bytes (1 word) long, and therefore also fit into a register.

Notice that this means that we can store addresses the same way we store any other number.

In fact, the computer can't tell the difference between a value that is an address, a value that is a number, a value that is an ASCII code, or a value that you have decided to use for another purpose.

A number becomes an ASCII code when you attempt to display it, and a number becomes a memory address when you attempt to lookup the byte it points to.

- This is crucial to understanding how a computer works
- The idea is based around the difference between data and information
- The number itself stored in memory has no meaning without the context (information, instructions) on how to process it
- For example, consider the number:

```
01001000010001010100110001001100010011111
```

- On it's own it is meaningless, however if I told you that this number represents a string in ASCII, and that every 8 digits (8-bits) represents an ASCII code (a number between `0` and `255`), then we could parse the number as follows
- First we break the number into 8 digit (bit) chunks:

```
01001000 01000101 01001100 01001100 01001111
```

- Then we can work out the ASCII character that corresponds to each number as follows

Binary Number	Decimal Number	ASCII Character
01001000	72	'H'
01000101	69	'E'
01001100	76	'L'
01001100	76	'L'
01001111	79	'O'

- And we see that the number represents the word `"HELLO"` in ASCII that was mentioned earlier

Addresses which are stored in memory are also called **pointers**, because instead of representing a regular value, they point to a different location in memory.

As we've mentioned earlier, computer instructions are alos stored in memory.

In fact, they are stored exactly the same way that other data is stored.

The only way the computer knowsd that a memory location is an instruction is that a special-purpose register called the **instruction pointer** points to them at one point or another.

If the instruction poihnter points to a memory word, it is loaded as an instruction.

Other than that, the computer has no way of knowing the difference between programs and other types of data.

- The source text points out here that there are some types of processers and operating systems that mark the regions of memory that can be executed with a special marker that indicates this.

4 Interpreting Memory

A computer has no idea what your program is supposed to do, it will do exactly what you tell it to do.

If you accidentally print out a regular number instead of the ASCII codes that make up the number's digits, the computer will try to look up what the number represents in ASCII and print that.

If you tell the computer to start executing instructions at a location containing data instead of program instructions, who knows how the computer will interpret that, but it will certainly try.

The computer will execute your instructions in the exact order that you specify, even if it doesn't make sense.

- Consider the previous example with the binary number representing the string `"HELLO"`
- If I told you that you had to first invert every bit in the number, and then every 8 digits would represent and ASCII Character
- Then we would first invert the bits of the number, and then split the digits into 8-bit groups:

```
10110111 10111010 10110011 10110011 10110000
```

- And decode the message as `"·º33°"` which doesn't make any sense

The computer will do exactly what you tell it to, no matter how little sense it makes, so you need to know exactly how to have your data in memory.

Computers can only store numbers, so letters, pictures, music, web pages, documents, and everything else are just long sequences of numbers in the computer, which particular programs know how to interpret.

- For example, say that you wanted to store customer information in memory
- One way to do so would be to set the maximum size for the customer's name and address to 50 ASCII characters each, which would be 50 bytes for each
- Supposethat you represented the customer age and the customer id with an integer (4 bytes each).
- If the pointer to the start of the record was `0x6dfed4` in hexadecimal, then the record could be described as follows:

Field Name	Record Start Pointer
`customerName`	`0x6dfed4`
`customerAddress`	`0x6dfed4 + 50 bytes`
`customerAge`	`0x6dfed4 + 100 bytes`
`customerId`	`0x6dfed4 + 104 bytes`

- This will cause issues if the `customerAddress` for example, is more than 50 characters, to illustrate this issue, let's consider a simpler example
- Consider a simple record of the form:

Field Name	Data Type	Length
`initials`	`ASCII String`	`2 bytes`
`age`	`number`	`1 byte`

- Then, for example the record {`"initials": "TB", "age": 30`}, would be stored in memory as (spaces added for clarity)

```
01010100 01000010 00011110
```

- Where, `01010100` $\mapsto$ `'T'`, `01000010` $\mapsto$ `'B'`, and `00011110` $\mapsto$ `30`
- Suppose we tried to enter a middle initial, then we would expect the following record:

$$\{\texttt{"initials": "TVB", "age": 30}\}$$

- However, in memory this record would be stored as:

`01010100 01010110 01000010 00011110`

- There were only supposed to be 3 bytes written into memory, but 4 bytes were received
- The first two bytes will be read in as the initals `"TV"`, and the byte that was meant to represent the initial `'B'` will be read in as the integer `66`
- The resulting record will be {`"initials": "TV", "age": 66`}, and the byte `00011110` will be written into some memory address that the program wasn't supposed to access
- This could cause an error, if that memory address was being used by some other program
- However, if the memory address was not being used, then it is possible that no error would be thrown
- This could result in the error going unnoticed, and a program which inputs the wrong age whenever a person enters in a middle initial
- This type of error can cause issues later, and would be very difficult to debug
- These types of errors are exploited to create **Buffer Overflow Exploits**, where a nefarious actor uses this undefined behaviour to insert nefarious code in place of some system code that runs with elevated privelages
- In this example, one could enter in an arbitary length string for the initials field followed by some nefarious code
- If the initials string is exactly the right number of bytes, then this could put the byte containing the first instruction of the nefarious code in the memory address where some procedure for a system call is stored
- Then when the system attempts to make the function call, the nefarious code is run instead with root privelages
- Going back to the previous customer record example, if we want to allow for variable length records, then instead of storing the records as a continuous block in memory, we should store pointers to the memory address of each record
- Then we could even specify the length of the field as the first byte (or bytes) of the field, which would allow us to decide the field length at field creation
- If we assume that the memory address of the record is `0x6dfed4`, then the record could be described as follows (recall that a pointer is 4 bytes long)

Field Name	**Record Start Pointer**
`customerNamePointer`	`0x6dfed4`
`customerAddressPointer`	`0x6dfed4 + 4 bytes`
`customerAgePointer`	`0x6dfed4 + 8 bytes`
`customerIdPointer`	`0x6dfed4 + 12 bytes`

- We could also have some variable length data, and some fixed width data, in that case, we would only store the variable length data as a pointer