ENEL260 CA3: Architecture and Memory

Ciaran Moore

"...computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1.5 tons." – Popular Mechanics, 1949

A "computer" is a device that performs logical operations on binary logical values. The order that the operations are performed in, and the values that are operated on, are all stored in memory or derived from the outside world. To understand what a computer is, we need to understand how the computing logic we've already learned about is combined with data storage and input/output.

Computers are unlike any other machine because, instead of being designed to perform a single task, they're flexible and can be re-tasked as needed. Computers come by their flexibility because their operation is determined by a *stored program* (i.e., software) that determines what computations the computer performs and what data the computer operates on. As a result, the same hardware can be used to perform many different tasks by changing either the program or the data. But what does that hardware actually look like?

Aside: The *architecture* of a building describes the structure of the spaces that make up a building, and how those spaces are related to each other. When talking about system design, the term *architecture* means a set of *components*, and the way in which those components are *connected*. Changing the number or type of components, or the connections between the components, results in a different architecture.

Computer Architecture[1] deals with the components that make up a computer, and the connections between those components (Fig. 1). At a high level, a computer is made up of:

[1] https://en.wikipedia.org/wiki/Computer_architecture

- Something to do computations: one or more *Central Processing Units*.
- Somewhere to store programs and intermediate data: one or more *Memory* components.
- Some way to get data in and out of the computer: *Input/Output Ports* that allow the computer to communicate with the outside world.

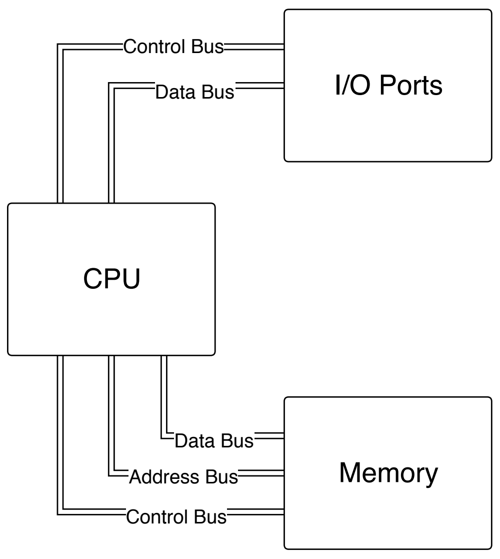

Figure 1: Major components and connections that make up a computer architecture.

Aside: The word 'bus' is a contraction of the word *busbar*. It refers to the set of interconnecting "wires" used to connect two or more devices.

The major connections between components within the computer are implemented using *buses*: sets of parallel wires. A computer contains 3 different types of buses:

1. *Address Bus*: an output from the CPU specifying locations (addresses) for memory or I/O transfers. The smallest referencable ("addressable") unit in most computer architectures is the *byte*: a sequence of 8 bits [2]. The width of the address bus in bits defines the size of the *address space* (that is, the number of bytes that the CPU can address): the size of the address space is 2^M bytes, where M = the width of the address bus in bits.
2. *Data Bus*: a (usually) bi-directional bus that carries the data to be written to a location specified by the address bus, or the data read from a location specified by the address bus.
3. *Control Bus*: specifies which way the data transfer occurs over the data bus, and when.

[2] Historically the word "byte" could refer to other numbers of bits, but in modern usage a *byte* is 8 bits.

Now let's consider some of these components and connections in greater detail.

CPU

A CENTRAL PROCESSING UNIT (CPU) is the core of a computer. A CPU consists of at least the following components (Fig. 2):

1. An **arithmetic logic unit** (ALU)[3]
   - Performs operations like: arithmetic functions (add, subtract, shift), comparisons (equal, not equal, greater than, less than), logical functions (NOT, OR, AND, XOR) and bitwise operations.
   - The two inputs (or for some operations a single input) are from the general purpose registers
   - The computed output is put back into a general purpose register.
2. A set of **general purpose registers** (sometimes called a GPR *file*)
   - These are memory locations situated *physically close*[4] to the other CPU components, particularly the ALU.
   - All ALU operations take their data inputs from GPRs and put the computed results back into a GPR.
3. A set of **control registers**
   - Specialist registers where the values are normally set as the result of actions undertaken by the machine.
   - Programmers and programs don't usually set these values directly but rather read or check their values.

[3] We saw how to implement most of the functions in previous lectures.

[4] Placing the registers close to the rest of the CPU increases the speed at which data can be stored and retrieved from these memory locations.

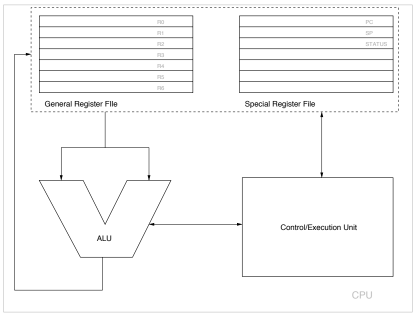

Figure 2: A basic CPU Architecture

- The most important control registers include:
  - *Program Counter* (PC) — holds the memory address of the next instruction to execute
  - *Status Register* (SR) — holds information about the last operation performed
  - *Stack Pointer* (SP) — holds the memory address of the top of the stack (used for function calling and storing *local* variables).

4. A control or execution unit
   - A *finite state machine* (FSM) that controls what happens and when.
   - The inputs to the FSM are the program instructions and the control registers.
   - The FSM outputs control what data (from registers) are presented to the ALU, what ALU function is performed, and where the result goes. It also controls the signaling on the address and control buses. Finally it may also change values stored in the control registers.

We'll take a look at state machines in a later lecture.

The *datapath* comprises the GP registers, the ALU, the data bus connecting them, and the data bus connecting the GP registers to the data memory.

A CPU may also contain other more specialized units:

1. A memory management unit (MMU)
2. A bus control unit (for direct memory access - DMA - transfers)
3. A floating point arithmetic unit (FPU)

The architecture of a CPU depends on the number and size of the registers, the operations supported by the ALU, the way that the execution unit behaves, the type of additional specialized units that are included, and how all of these things are connected.

It's worth noting that although a computer, at a high level, includes both a CPU and Memory, even the CPU itself includes small memory elements (the registers). The ability to store programs and data in various memory components is a large part of what makes a computer the flexible machine that it is. But we also know that computers are built from logic circuits. So, let's look a little more closely at how we might store data in a logic circuit.

Latches

The logic circuits that we've looked at so far have all been *combinational logic*: the output is a combination of the input values. Another form of logic circuit is known as *sequential logic*: logic circuits that have loops (i.e., *feedback*) so the output is not just a combination of the current inputs but *also of previous outputs*. Including feedback in a logic circuit gives us a way to store a state from moment to moment, but it also means that the circuit is harder to analyse (Fig. 3).

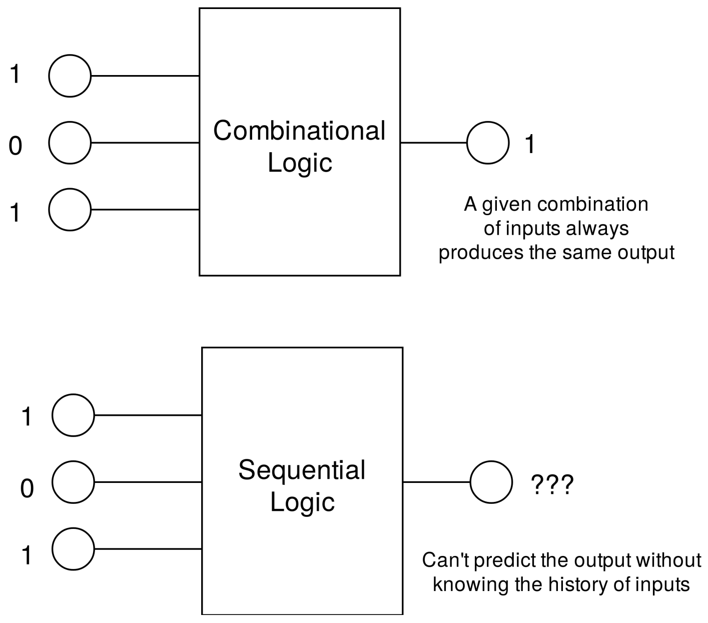
1
0
1
Combinational Logic
1
A given combination of inputs always produces the same output
1
0
1
Sequential Logic
???
Can't predict the output without knowing the history of inputs

Figure 3: Combinational logic and Sequential logic

One of the simplest sequential logic circuits is the **SR latch** (Fig. 4, Fig. 5), which is constructed from two NOR gates. The circuit has two inputs, *'S'* and *'R'*, and two outputs.

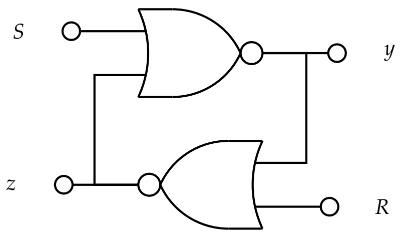

Figure 4: The SR Latch circuit implementation using NOR gates

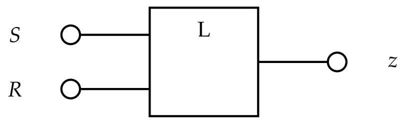

Figure 5: The SR Latch circuit symbol (the gates within don't need to be shown if we know how the latch operates)

Looking Deeper: Attempting to Model Attempting to model an SR Latch in Python using our existing combinational logic gate models gets tricky. Because of the feedback loops in the latch, the outputs are also the inputs. That causes the Python interpreter to complain:

```
def sr_latch(s, r):
    y = nor_gate(s, z)
    z = nor_gate(r, y)
    return (y, z)

>>> sr_latch(1,0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sr_latch
UnboundLocalError: local variable 'z' referenced before
    assignment
```

The example code on Learn includes simple SR Latch model that uses iteration to resolve the circularity. The resulting model is a fairly rough approximation of how a real SR Latch functions, but should give you the idea.

A truth table for the SR-Latch outputs, y and z, is shown in Table 1. The table shows the values of y and z at time steps k and $k+1$, where:

- Time k represents the instant *just before* the inputs shown are asserted.

- Time $k+1$ represents the instant *just after* the inputs shown are asserted and the outputs have settled to the resulting new values.

For this relatively simple version of the latch, the behaviour is undefined if the inputs R and S are both 1 at the same time. Therefore no rows are included in the truth table for those cases. The inversions of the outputs of the NOR gates ensure that y and z always have opposite state, as can be observed in the truth table.

Table 1: The SR Latch truth table

Inputs		Outputs			
S	R	y_k	z_k	y_{k+1}	z_{k+1}
0	0	1	0	1	0
0	0	0	1	0	1
0	1	1	0	1	0
0	1	0	1	1	0
1	0	1	0	0	1
1	0	0	1	0	1

The interesting thing to note in Table 1 is that the SR Latch implements a form of memory (when the inputs are both zero). Indeed this is the whole point of feedback circuits — they depend on *what has happened in the past* as well as *what is happening now*.

Flip-Flops

One major problem with the latch circuit is "transparency": the output changes almost immediately with a change of the inputs, which makes the analysis (and therefore the synthesis) of such circuits difficult. As an alternative to a latch circuit we can use a class of devices called *flip-flops*, for which the output change is controlled to occur only under the influence of a clock signal.

It's important that the inputs do not change during the "decision window" when the clock signal is active.

Flip-flops are "opaque" to changes of the input except when the clock signal is active. Many flip-flop designs are *edge triggered*, so that it's the change of state of the clock signal which causes the flip-flop to react to the state of the inputs.

Fig. 6 shows the symbol for a D-type edge-triggered flip-flop (D for 'data'). The flip-flop has a data input, D, and a clock input. One of the outputs, Q, represents the value of D at the last clock transition. The other output, $\overline{Q}$, is simply the inverse of Q. The step shown next to the clock input indicates that it is the rising edge (the change from 0 to 1) that causes the flip-flop to react. Table 2 shows the truth table for a D-type flip-flop.

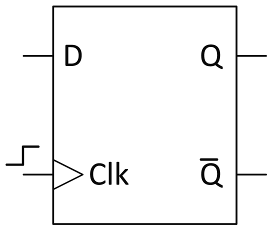

Figure 6: The edge-triggered D-type flip flop symbol

Table 2: The edge-triggered D-type flip flop truth table; k indicates just before the clock event, $k+1$ indicates just after.

Input	Current		Next	
D	Q	$\overline{Q}$	Q_{k+1}	$\overline{Q}_{k+1}$
0	0	1	0	1
0	1	0	0	1
1	0	1	1	0
1	1	0	1	0

Registers

One flip-flop can remember 1 bit of information. To remember a byte or word in a computer therefore requires an *array* of flip-flops. Such an array is called a *register*. An example 4-bit register is shown in Fig. 7.

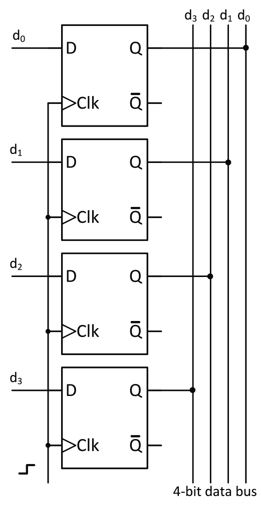

Figure 7: A 4-bit register comprising 4 × edge-triggered flip flops with outputs connected to a 4-bit data bus.

Memory

The registers in a CPU are located close to the rest of the CPU for performance reasons. However, there are only relatively few registers in a CPU, able to hold only a small amount of program code and data. The majority of the program and data storage occurs in a separate memory that's external to the CPU.

Program and data memory can be implemented in a variety of different ways[5], each having different performance characteristics and different capabilities:

[5] See https://barrgroup.com/Embedded-Systems/How-To/Memory-Types-RAM-ROM-Flash

- Bits may be stored as data in latches, as charge in capacitors, as magnetic patterns on a disk, or as electrons trapped in a transistor gate, or some other way. Individual bit-stores are organized into arrays that store bytes or words.
- The memory may retain data through power cycles, or be wiped clean of data when power is removed.
- Read or write times can range from a few nanoseconds to milliseconds.

The *memory* in a computer is a collection of bytes. Each byte is assigned a unique number, called its *address*[6], that can be used to access (read or write) the data contained by that byte. You can think of memory as a giant *array* of bytes. The address of each byte is like an *index* into that array.

[6] As you might guess from the name, the *addresses* of bytes are analogous to the numbers of the houses in a street (a rather long street).

Although it's possible to refer to stored data directly using the address of the bytes where the data is stored, when we write a program it's often easier to use meaningful variable names to identify pieces of data. Behind the scenes, the compiler can convert these names to the addresses of the bytes where the data is actually stored.

Example: Here's an example of an area of memory, with ascending addresses on the left. Each box represents a byte of data. The names of some variables are on the right.

If we think of the memory as a giant array called `memory`, then `varA` is equivalent to `memory[0x100]`, `varB` is equivalent to `memory[0x101]`, and so on. That isn't *actually* how we directly access memory (there isn't an array called `memory`), but it's conceptually similar.

`0x100`	byte 0	varA
`0x101`	byte 1	varB
`0x102`	byte 2	temperature[0]
`0x103`	byte 3	
`0x104`	byte 4	temperature[1]
`0x105`	byte 5	
`0x106`	byte 6	temperature[2]
`0x107`	byte 7	
`0x108`	byte 8	
...	...	

In this example, two 8-bit variables, `varA` and `varB`, and a 3-element array, `temperature`, of 16-bit values are stored in the first 8 bytes. Thus a reference in the program to `varB` means the byte at address 0x101, while a reference in the program to `temperature[2]` means the two bytes at addresses 0x106 and 0x107, treated as a 16-bit value.

The different kinds of memory and their implementation is covered in greater detail in ENCE361.

Moving Data: The Datapath

It's all very well to store data in memory. But in order to perform computations, we need to move data and combine data items within the computer, as well as accepting inputs and issuing outputs. Data is moved by means of a *bus*.

The *datapath* comprises the GP registers, the ALU, the data bus connecting them, and the data bus connecting the GP registers to the data memory. The width of the datapath in a computer is the *word length* (or *word size*), and this is always an integer multiple of the byte size. Thus there are 8-bit, 16-bit, 32-bit and 64-bit wordlength computers. One *word* at a time is moved about the datapath.

For efficiency, reading (fetching) from or writing (storing) to memory is always done in words, i.e., potentially multiple bytes at a time. The address is used to determine which word of memory to read or write. *Reading* means setting the bits of the datapath to be the same as the bits of the word in memory. *Writing* means setting the bits of the word in memory to be the same as the bits of the datapath. Each word in memory can be accessed as fast as any other word; this is called *random access*[7].

[7] As opposed to *sequential access*, in which words need to be accessed in a specific order, making later words take longer to reach than earlier words.

Example: For a computer with a wordlength of 2 bytes (16 bits) a section of the memory could be visualized as:

Note that the memory is still addressed in terms of bytes. The addresses in the diagram increment in steps of two, since each word corresponds to two bytes. The size of the *address space* is 2^M bytes, where M = the width of the address bus.

`0x100`	word 0
`0x102`	word 1
`0x104`	word 2
`0x106`	word 3
`0x108`	word 4
`0x10A`	word 5
`0x10C`	word 6
`0x10E`	word 7
`0x110`	word 8
...	...

Memory Architecture

THE *memory architecture* IS THE PART of the overall computer architecture that defines how buses are used to connect memory to the CPU. The two major classes of memory architecture are the *von Neumann* (also called "Princeton") and the *Harvard* architectures.

The *von Neumann Architecture* originated with John von Neumann, one of the founding fathers of modern computing. Von Neumann's design for the stored program computer (1945) was the most widely used architecture right up through the mid-1990's (Fig. 8). Von Neumann made no distinction between the "bit patterns" that represented data and those that represented instructions to the CPU, so the von Neumann architecture only needs one "monolithic" memory and has a single data bus.

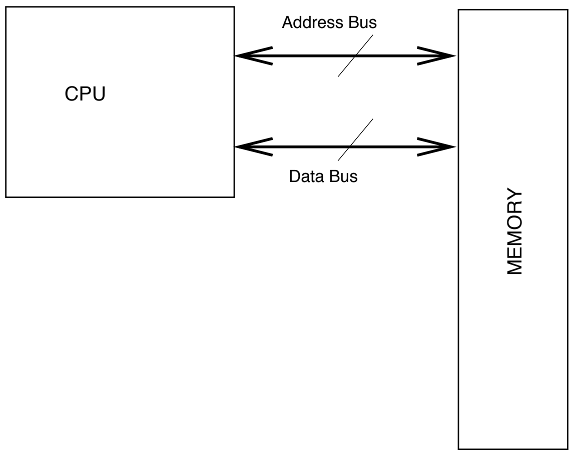

Figure 8: The Princeton architecture has a single monolithic memory.

At around the same time as the von Neumann architecture was created, a rival architecture was developed at Harvard University. The *Harvard Architecture* has separate locations for the data and the program code, thus requiring two memories and two extra buses (Fig. 9).

Until recently, the von Neumann architecture was favoured over

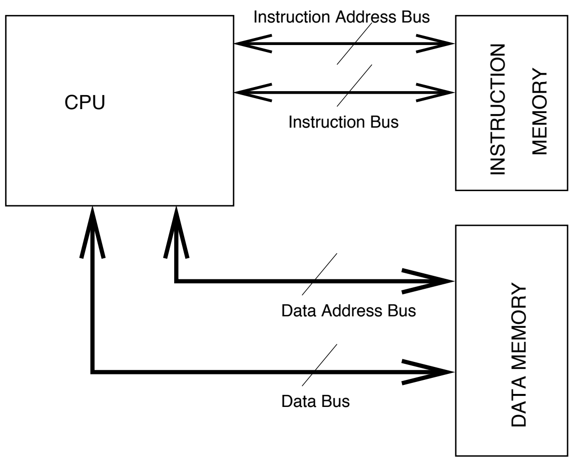

Figure 9: The Harvard architecture has two separate memories for data and instructions.

the Harvard architecture. There were three main reasons for choosing the von Neumann architecture:

1. The Harvard architecture requires more parts and buses (more hardware) than the von Neumann architecture.
2. The Harvard architecture has pre-determined code and data locations, whereas the von Neumann architecture provided the flexibility to allocate memory to data or instruction code differently depending on the needs of the application.
3. The Harvard architecture makes it difficult for programs to modify their own code[8] at runtime, while the von Neumann architecture makes it straightforward to write programs that modify their own code[9].

[8] This can make loading of programs more difficult too.

[9] While useful, this does have some security implications.

Despite being more costly in terms of hardware, and less flexible in terms of processing, than the von Neumann architecture, the Harvard architecture is now the predominant design for modern computers. The dominance of the Harvard architecture is due to the enormous performance gains it enables. During computation, instructions operate on data: by splitting instructions and data into separate memory areas the Harvard architecture makes it possible to fetch instructions and data from different memories at the same time (i.e., in parallel), speeding up the overall computation.

It's important to note that the style of architecture used directly affects the design of the CPU.

Input/Output Architecture

I/O refers to *everything that is outside of the CPU*–the "peripherals"–that either supplies external information to the CPU or accepts information from the CPU for display or for use elsewhere.

THE *input/output architecture* IS THE PART of the overall computer architecture that defines how input and output (I/O) are connected to the CPU. As with memory architecture, there are two predominant designs: *memory mapped I/O* and *separately mapped I/O* (Fig. 10, Fig. 11).

In *memory mapped I/O*, no distinction is made between memory and I/O devices. The same address, data, and control buses are used for accessing both. So storing a piece of data to a memory address is the same operation as writing that datum to an output port. Similarly reading a piece of data from a memory address is the same as reading an input port. The only way to tell if an address is pointing to memory or to an I/O Port is to refer to the memory map of the processor.

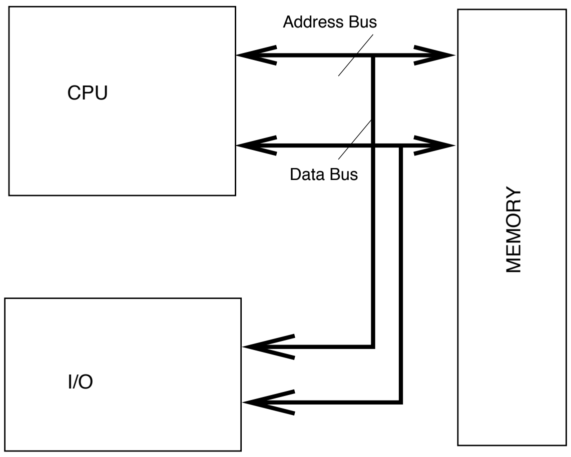

Figure 10: Memory mapped I/O

In *separately mapped I/O*, memory and I/O are accessed using separate control buses, and sometimes using separate address and data buses as well. This separation effectively increases the address space and eases the control of I/O operations.

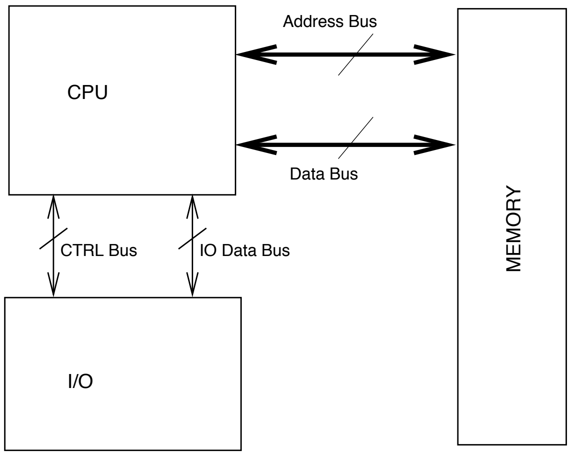

Figure 11: Separately mapped I/O

Storing Addresses $\Longleftrightarrow$ Pointers

Regardless of the memory and I/O architecture, sometimes, rather that wanting to access a *fixed* (i.e., constant) memory address, we may want to access one of a number of addresses, depending on decisions made while the program is running. This means we want to have a *variable* that contains an address. In C, such a variable is called a *pointer*.

Remember that a *variable* is a location in memory that contains some data. The location of a variable is its *address*. In C, you can get the address of a variable, for example `varA`, by using the `&` operator:

```
varA_address = &varA;
```

A *pointer* is just a special kind of variable: the data it contains is the address of some other variable. If the variable a pointer addresses has a type `T`, then the pointer has the type `T*` (which you can read as "pointer to a variable of type T", or "address of a T"):

Some programmers put a space between the base type and the `*`, i.e., they'll write `T *` instead of `T*`. The compiler doesn't care about spaces though. I prefer to write `T*` so that it's clear I'm talking about a pointer.

```
int varA = 42;
int* varA_address = &varA;
```

To look up the value addressed (or pointed to) by a pointer you can use the `*` operator to "dereference" the pointer. This tells the compiler to take the data contained by the pointer variable and treat it as a memory address:

If we're thinking of memory as an array, then using the `*` operator is like doing an array lookup. So the code in this example is doing something like `memory[varA_address] = memory[varA_address] + 1;`

```
int varA = 42;
int* varA_address = &varA;

*varA_address = *varA_address + 1;
// varA is now equal to 43
```

Example: Here's an example C program that demonstrates different things you can do with pointers:

This program illustrates finding an address and dereferencing a pointer. It also shows how we can use a pointer just like any other variable. So we can pass it as an argument (allowing a function to access the memory at the pointed-to address), and even modify the value (letting us do things like computing one address from another one).

```
/* ENCE260 Pointer Example
 *   A small example of using pointers in C
 *   Allan McInnes / 2018-06-21
 *
 * Build with: gcc -o pointer_example
    ence260_pointer_example.c
 */
#include <stdio.h>

void change_the_value_at_an_address(int* address) {
   *address = 128;
}

int main (int argc, char* argv[]) {
   int varA = 42;
   int varB[2] = {64, 256};
   int* var_address;

   printf("Starting with these variables:\n");
   printf("int varA = 42;\n");
   printf("int varB[2] = {64, 256};\n");
   printf("int* var_address;\n");

   // Getting an address
   var_address = &varA;
   printf("\n--> var_address = &varA;\n");
```

```
    printf("The address of varA is 0x%X\n", &varA);
    printf("The value of var_address is also 0x%X\n",
     var_address);
    printf("The value of varA is %d\n", varA);
    printf("The value of *var_address is also %d\n",
     *var_address);

    // Dereferencing
    *var_address = *var_address + 1;
    printf("\n--> *var_address = *var_address + 1;\n");
    printf("The value of varA is now %d\n", varA);
    printf("The value of var_address is still 0x%X\n",
     var_address);

    // Reassigning a pointer
    var_address = &varB[0];
    printf("\n--> var_address = &varB[0];\n");
    printf("The value of varB[0] is %d\n", varB[0]);
    printf("The address of varB[0] is 0x%X\n", &varB[0]);
    printf("The value of var_address is now 0x%X\n",
     var_address);

    // Passing a pointer as an argument
    change_the_value_at_an_address(var_address);
    printf("\n-->
     change_the_value_at_an_address(var_address);\n");
    printf("The value of varB[0] is now %d\n", varB[0]);
    printf("The value of var_address is still 0x%X\n",
     var_address);

    // Modifying a pointer (pointer arithmetic)
    var_address = var_address + 1;
    printf("\n--> var_address = var_address + 1;\n");
    printf("The value of var_address is now 0x%X\n",
     var_address);
    printf("The address of varB[1] is also 0x%X\n",
     &varB[1]);
    printf("The value of *var_address is %d\n",
     *var_address);
    printf("The value of varB[1] is also %d\n", varB[1]);

    return 0;
}
```

Example: In this example, the function `pointStr` returns a pointer. Each call to the function provides the address of one of 4 strings, the names of the 4 compass points. The variable `northStr` is declared as a 'pointer to char', i.e. with type `char*`, and it is initialised with the address of the string "North". In the loop, the address of each of the 4 strings is fetched in sequence. The address is passed in turn to the `printf` function.

Remember that for a C string, the address is of the first character of the string.

```
/* compass.c
 * Example for ENCE260 - use of pointers
 *    P.J. Bones  UCECE
 *    Last modified:  22.7.2017
 */
#include <stdio.h>

#define NORTH 1
#define EAST  2
#define SOUTH 3
#define WEST  4

char* pointStr (int point) {
    static char* name[] = {"", "North", "East", "South",
     "West"};

    if (point > 0 && point < 5)
        return name[point];
    else
        return name[0];
}

int main(void) {
    int index;
    char* northStr = pointStr (NORTH);

    printf ("In clockwise order the compass points are:\n");
    for (index = 1; index < 5; index++)
        printf ("%s ", pointStr (index));
    printf ("\n\n");

    return 0;
}
```