# CM100194: Systems Architecture I

Jim Laird*

Semester I: 2011/12

Disclaimer: These notes are not intended to be a definitive record of the course, but an adjunct to it — they may (do) contain inaccuracies, inconsistencies and irrelevancies.

**Contents - Part I**

## 1 A brief history of computers

Computers have been around for a long time. In the early days, a *computer* was a person who computed, e.g., worked out tables of latitude that ships navigators could use. These days, of course, the word refers to certain kinds of machine that do computations: *number crunchers* is a popular phrase. But computers that crunch numbers (e.g., weather or payroll computations) are in the minority. Most computers these days crunch things like words, images and sounds. You might want to argue that, underneath it all, those things are represented by numbers and we crunch those numbers, but this is a very limited way of looking at things. After all, what does it mean to do arithmetic on sentences? It is not useful to think of the words being represented as numbers. The operations we do on words and other media are certainly not arithmetical.

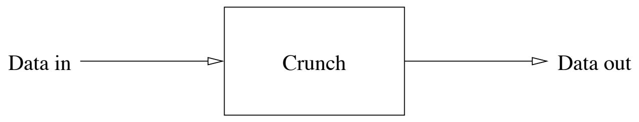


Figure 1: Data Cruncher

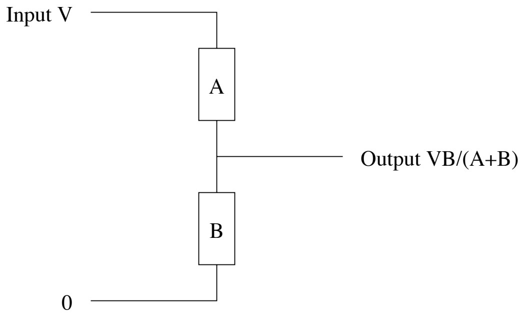


Figure 2: Analogue Computation

Much better is to think of a computer as a *data cruncher*

Data go in, some transformations are made, and data come out.

---

Or, in the famous phrase "Garbage in, garbage out" or *GIGO*. Something that is often forgotten is that quality of the data that come out are only worth as much as the data quality of the that go in.

---

Many types of data are possible. Just a few include

- process control (sensors and controls, cars, toasters)
- numeric data
- word processing
- symbolic processing (manipulating formulae, e.g., $(x + 1)(x - 1) = x^2 - 1$)
- game playing (chess as well as shoot-em-up and interactive)
- modelling (all kinds of things)
- speech (mobile phones)
- vision (DVD, digital TV)

## 1.1 Analogue Computers

And digital computers are a relatively new addition. Early machines used continuously variable *analogues*, such as electric current or voltage to represent values.

*with thanks to N Michael Brooke

A couple of resistors can scale a voltage by a chosen value. Such machines are hard to design and build, and not very programmable!

Analogue techniques are

- **precise** in that a voltage specifies a certain number exactly
- **inaccurate** in that we are not quite sure which number is being represented. Reading off values from the system always incurs errors.

Furthermore, the more operations we do on a value, the less sure we are about the accuracy of the result.

On the other hand a digital representation is

- **imprecise** in that we might not be able to represent certain values (like 1/3 in binary) exactly
- **accurate** in that when we read a number out of the system, we know exactly what value it is we are reading.

The imprecision is called *quantisation*: we have to approximate certain values. A common example of this is encoding music to MP3 format. The precision of the representation can be increased, but always at a cost.

We shall only be considering digital machines.

## 1.2 Timeline

- One of the earliest recognisable counting machines was the abacus, though recording devices, such as knots on strings, came first. The abacus was different in that it aided computation and didn't just make records.
- Blaise Pascal (1642) produced the first adding machine for his father's tax office
- Liebnitz (1671) designed a multiplying machine, but
- Thomas de Colmar (1820s) made the first practical adder and multiplier, the *arithmometer*.
- Greater automation came with Muller's difference engine (1786), designed to build log tables by turning a handle. At this time accurate charts for sea navigation were the highest priority.
- This idea was developed by Charles Babbage into an analytical engine, which had a thousand number store, a mill to do computation and punched card inputs so that instructions could be programmed (Ada, Lady Lovelace — Lord Byron's daughter — was the first programmer).
- George Boole wrote "Investigation of the Laws of Thought" in 1854 and devised an algebra of logic and propositions.
- Herman Hollerith devised punched card machinery for the American census in 1889.
- Burroughs (1892) developed the first commercial adding machine.
- Howard Aiken built the nearest approach to an analytical engine in 1937, the same year that A M Turing specified an automatic computing machine based on mathematical logic.
- John von Neumann in the USA had worked along very similar lines independently.

- In 1938 Claude Shannon's paper, "Symbolic analysis of relay & switching circuits" showed how electronics and Boolean Algebra could be combined to implement computing machines.
- The impetus of the Second World War then led to the first experimental computers, e.g. Flowers and Turing's COLOSSUS, von Neumann's ENIAC, Princeton's EDVAC, NPL's ACE Pilot and DEUCE; Wilkes' EDSAC at Cambridge; and the Manchester MARK I. All of these were operating before 1950. They used relays and valves and could do about 100 operations per second. Paper tapes and cards were used for input and output.
- By the mid 1950s there were more machines for accounting and commercial applications and large-scale numeric processing, the first generation. They included English Electric, Ferranti & Elliott machines. All used valves and were programmed in a machine dependent code.
- By 1960 the second generation machines using transistors were appearing. They were smaller and faster (typically 50,000 operations per second) and had more input and output devices. Commercial interest was widespread and problem oriented languages were being developed for programming.
- Solid state integrated circuitry allowed very compact third generation computers to be made by the mid 1960s. They could perform a million operations per second and had sophisticated hardware with multiple channels, memory sharing, time sharing, visual displays and interrupts. There were by now many problem oriented languages and operating systems were beginning to appear.
- Fourth generation machines in the early 1970s began to use multiple processors (e.g. ILIAC 64) and novel architectures (e.g. CDC STAR with its pipelines).
- With the advent of micro-electronics in the late 1970s, the stage was set for contemporary computing machinery to make its debut.
- The IBM Personal Computer (PC) emerged and the era of mass computing began.

Moore's Law (an observation on existing experience, not a "law" in any real sense) said that the transistor count in microprocessors doubles every two years; this was later updated to a doubling every 18 months. This is often misquoted as a *speed* doubling every 18 months. This speed doubling held true for a long time, but is now faltering, while the count doubling is still holding up. Instead of processors getting faster, they are currently getting larger, like the current *multi-core* processors (with multiple CPUs on one chip).

## 2 The von Neumann architecture

In the early days of programming each program was intimately tied to an individual machine. If you wanted to write a program to solve a problem, you had to know what machine you were going to use before you could write the program. This was because the programmers had to know the individual features and quirks of the machine, such as

- how many bits in a word?
- what kinds of storage does the system have?
- how is the storage accessed?
- what kinds of instructions does the machine have?

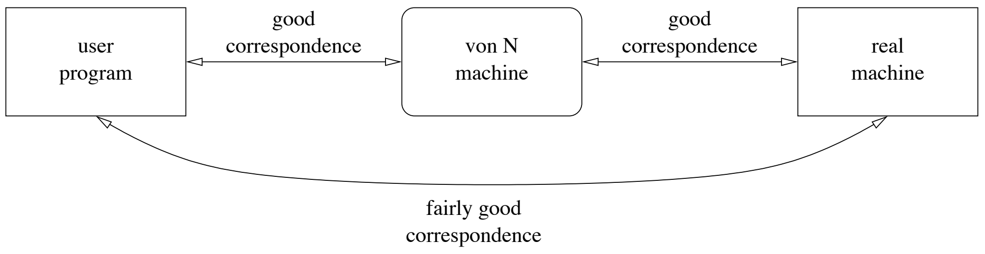


Figure 3: The von Neumann Map

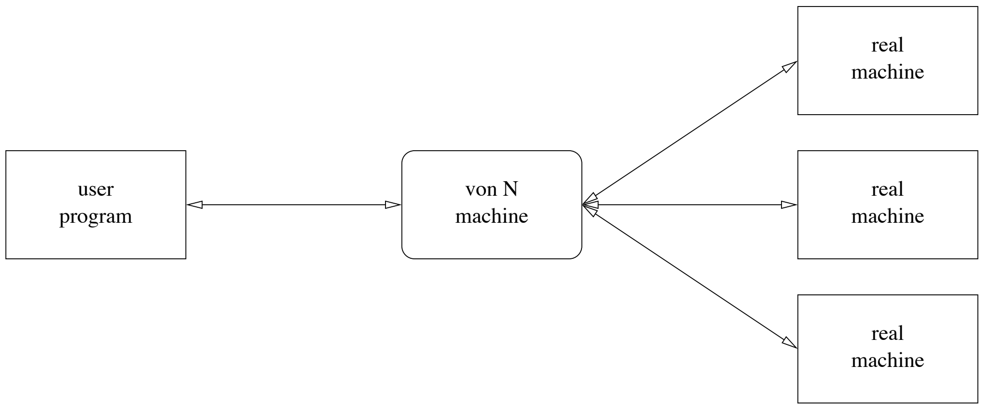


Figure 4: Portable Programs

and so on. This meant, of course, extra work for the programmer and moving programs to other machines was almost impossible.

John von Neumann proposed this: suppose we had a standard way of building computers. Then programmers would not have to rely on arcane knowledge and porting programs should be easy (or easier). Programs would be *machine independent*. Now manufacturers were never going to agree to make all their hardware identical, so the idea was to *pretend* they were identical.

Suppose we had an idealised machine, the *von Neumann architecture*. If (a) we wrote our programs in languages that map well on to this architecture and (b) manufacturers built their machines so they mapped well on to the architecture, then our programs will map pretty well on to the real machines.

This seems a bit of an effort until we realise the benefits. Because we didn't use any of the quirks of any particular machine, our program will map well on to *any* machine that keeps to the von Neumann principle! Manufacturers can play with their designs to their hearts' content, but our program will run (reasonably) well on all of them.

The benefits to everybody are immense.

## 2.1 Towards the von Neumann Architecture

So what should this idealised architecture be? We shall approach this in small steps.

To carry out any operation we need

- an operation specification (e.g. ADD or AND),

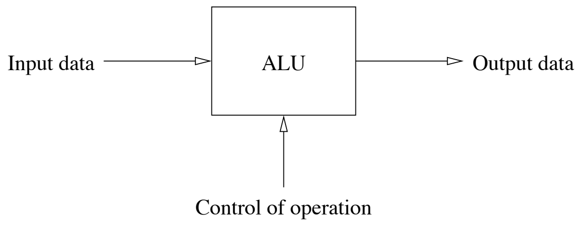


Figure 5: Arithmetic and Logical Unit

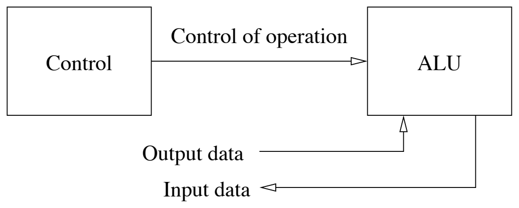


Figure 6: Control Unit

- data or operands to operate upon.

We can now think of an arithmetic and logical unit (ALU) as that where these operations are carried out (Figure 5).

The ALU alone is not automatic, it is pretty much equivalent to an abacus. Operations and operands must be supplied in the correct sequence and at the appropriate times. A *control* unit is added to do this (Figure 6).

Where does the data come from? It comes from a data store unit in response to a stimulus from the control unit (Figure 7).

If data store and the control unit work fast enough the ALU can be kept in continuous operation.

The activities of the control unit are as follows:

- To determine the operation to be performed;
- To select the correct operands and make them available at the right time;
- To perform the operation on the operands by supplying the correct control data to the ALU; and
- To determine the next operation to be performed!

It is thus the control unit that renders the digital data-processor *automatic*.

So where does the operation information come from? The *instruction store* (Figure 8).

The two stores are almost always combined into a single unit (Figure 9). Some embeded machines keep program and data separate: e.g., program in ROM and data in RAM.

We still need communication between the machine and the outside world, using *input and output devices* (Figure 10).

This is the *von Neumann Five Box Model*.

The characteristics of the machine we have constructed are:

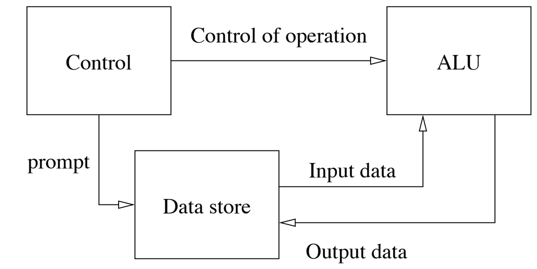


Figure 7: Data Store

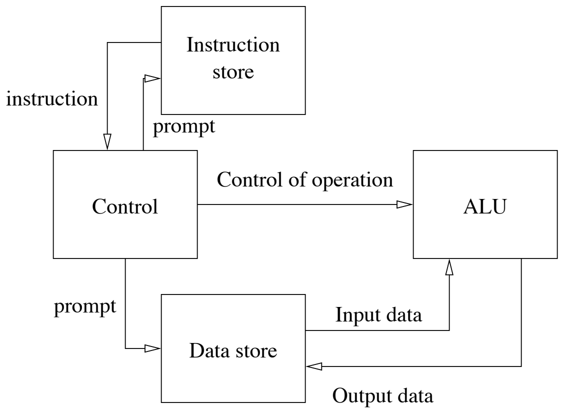


Figure 8: Instruction Store

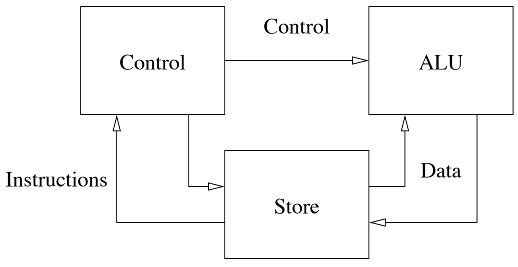


Figure 9: Unified Store

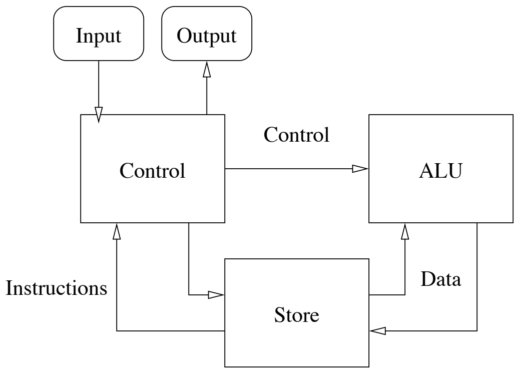


Figure 10: Input and Output

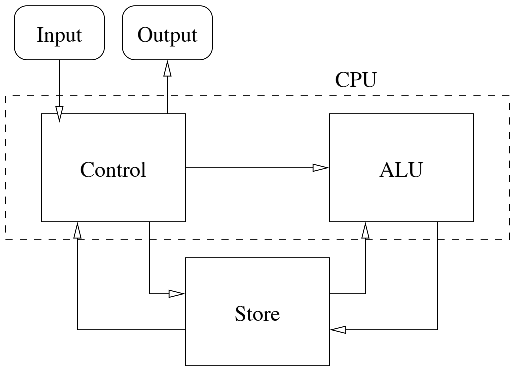


Figure 11: Central Processing Unit

- Discrete (i.e. 'digital') data;
- Storage of data and instructions in exact form, for a possibly indefinite time;
- Processing capabilities, e.g. arithmetic, in ALU;
- Control to enable automation (selection of data and operations, their execution and sequencing); and
- Communications with the outside world via I/O devices.

Babbage's machine had one further feature, that of conditional control.

Operation sequencing can be controlled by the state of the ALU, which is fed back to the control unit (Figure 11).

The control and ALU units are normally grouped together and called the *Central Processing Unit*, or *CPU*.

The von Neumann architecture has been extremely successful and every uniprocessor machine is built according to it. This means that programmers do not need to know or understand the intricacies of how a machine is put together as long as it has memory, I/O and a CPU.

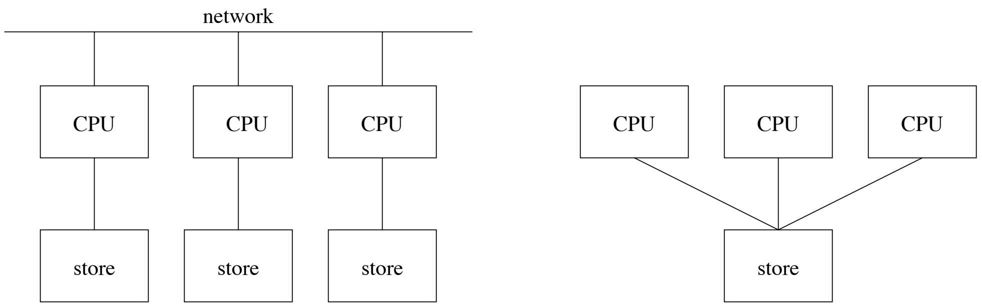


Figure 12: Memory Architectures

## 2.2 Parallel Architectures

The von Neumann architecture has just one processor. In order to increase computation power it would be good to have more than one ALU. So why are multiprocessor machines not universally popular?

A big reason is that there is currently no good model like von Neumann on uniprocessor machines. Multiprocessor machines are somewhat still in the 1960s: to write a parallel program you still need to know what kind of machine it is going to run on.

There is currently a variety of architectures under consideration, both on how to connect memory to CPUs and how to control the CPUs.

- Distributed memory. Each CPU has its own chunk of memory. Contents of memory of other processors needs to be fetched explicitly over, say, a network. Cheap to build using existing components, but accessing remote memory is very slow. This kind of machine is not suitable for programs where multiple processors need fast access to all the dataset.
- Shared memory. Each CPU is connected to the same chunk of memory. Good to program as each processor sees the same data, but is expensive to build as the shared connection to memory soon becomes a bottleneck.

Variants of these are

- Virtual shared memory. Distributed memory where the operating system hides the fact that the memory is distributed and tries to make it appear shared.
- Non Uniform Memory Access (NUMA). A half-way house between shared and distributed, where memory is access as in shared, but the programmer is aware that some chunks of memory are slower than others.
- The alternative to NUMA is *symmetric* shared memory, where every processor gets equal access to all memory in a symmetric way. This does not scale well and is very expensive for large machines.

As regards control

- Multiple Instruction Multiple Data (MIMD). Each ALU has its own control unit. Each processor is doing its own processing on different chunks of data.

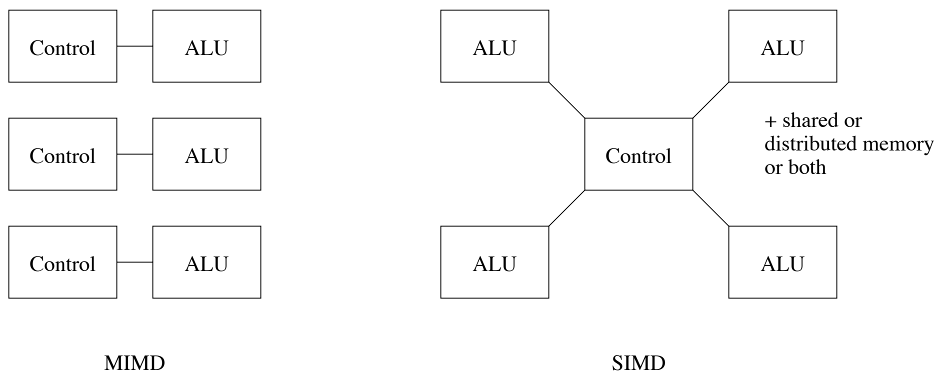


Figure 13: Control Architectures

- Shared Instruction Multiple Data (SIMD). All the ALUs are controlled by a single unit. At any moment in time each ALU is executing the same instruction, but each is working on a different chunk of data. This is good for a computation (such as weather prediction) that is repeated many times of different pieces of data.

Uniprocessors are SISD in this classification.

The two popular architectures are shared memory MIMD and distributed memory MIMD. SIMD machines exist, but are highly specialised: machines with 16 thousand (small) processors have been built. Shared memory machines with two or more processors are reasonably common. Multicore processors, with two or more processors on the same chip are currently popular, with larger counts in processors arriving.

Distributed memory machines can get very large as the architecture scales well. Clusters of commodity workstations connected by a network is a common implementation. Machines of thousands of processors exist. Not every *program* scales well on such machines, though. Some programs are good on certain architectures, while being poor on others:

| | |
|---|---|
| shared | program plus remote interface |
| distributed | graphic rendering, weather |
| SIMD | weather |
| none at all | most programs ever written |

Writing parallel programs is *hard*. The interactions between the parts of the program all of which are running at varying speeds is difficult to get right, and this is on top of the normal problems of debugging serial programs. It's not just a matter of taking a serial program and tweaking it: there will be substantial changes needed.

In terms of popularity we have (most to least popular)

1. clusters (distributed MIMD)
2. shared memory (shared MIMD)
3. SIMD

which mirrors the relative cost of building these things: clusters are very cheap.

The issue is that there are several parallel architectures, none of which are suitable for all kinds of problem. It may be that there is no, or no single generalisation of von Neumann to multiprocessor machines: we shall have to wait and see.

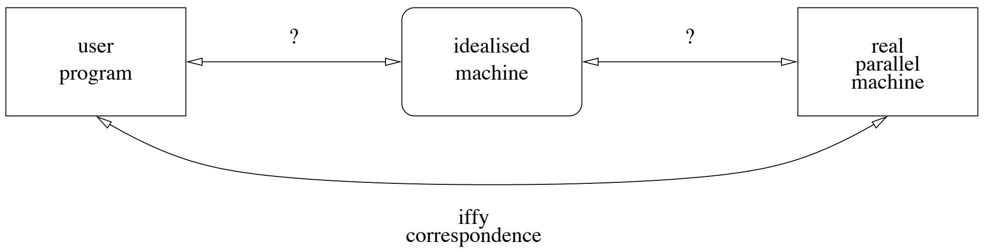


Figure 14: Lack of Parallel von Neumann Map

## 2.3 Limitations on Parallelism

A typical computational task will contain elements or subtasks which can be processed in parallel, in a suitable architecture, but some elements which are inherently sequential - for example, initializing data or collating it at the end of computation. The maximum speedup which can be obtained by performing such a task on a multiprocessor architecture rather than a uniprocessor, is expressed by *Amdahl's Law*:
Suppose the proportion which can be computed in parallel is given as $0 \leq p \leq 1$. Then if the time to perform the computation on a uniprocessor is $t$, the time to perform it on in parallel on $n$ processors is at least $(1 - p).t$ (the necessarily serial part) plus $\frac{p.t}{n}$ (the parallelizable part).

The maximum speedup obtained by computing the task in parallel is the ratio of the time to compute it serially ($t$) to the (minimum) time to compute it in parallel ($(1 - p).t + \frac{p.t}{n}$). Thus it is equal to

$$\frac{1}{(1-p)+\frac{p}{n}} = \frac{n}{(1-p).n+p}$$

Hence, as we would expect, the maximum possible speedup from computing the task with any number of parallel processors is $\frac{1}{1-p}$.

# 3 Numbers I: Integers

Real computers are the product of the electronics industry and are based on two-state devices. Electronically, we need to distinguish only between, for example:

- voltage and no voltage, or
- current and no current, or
- laser light and no light (optical fibre), or
- switch on and switch off, or
- electrical pulse and no electrical pulse.

We shall see that this can be related to a particular numeration system, the binary system.

In the *Hindu-Arabic* system, digits are symbols representing a cardinal number of *standards*, each of which we can denote by 'I'. E.g:

'2' → II and '6' → IIIII I and '7' → IIIII II

In particular, the symbol '0' (zero) represents a *null set* of standards.

In the *decimal* numeration system that we use every day, we do not introduce a new symbol for IIIII IIIII standards, but regard it as a *basic set* of standards, that is, as '1' basic set of standards and '0' unit standards, i.e. '10' (the digit symbols are *ordered*). The base of such a system is the *cardinal number* of standards in the basic set.

In the Hindu-Arabic system, the ordering of the digits is vital and digits have a *place* value as well as a *digit* value, as we have seen. Thus: IIIII IIIII III standards can be written as '13', that is, '1' basic set of standards and '3' unit standards.

We can also have a basic set of basic sets of standards, which we can write as '100' and we can then write numbers like '127', etc.

'9' of anything plus '1' more becomes a new, regrouped set.

What we call tens, hundreds and thousands are simply names for successive new groupings. This is called a *positional* system.

The elements of the numeration system are that:

- as many digit symbols as the base are needed;
- the decimal system's *base* or *radix* of ten, the cardinal number of standards in the basic set, is denoted by '10';
- place values increase from right to left in successive powers of the base;
- *addition* is used to make up a number consisting of combinations of digits and *multiplication* is used to make up the number represented by a digit in a specific place;
- there is an agreed starting point (the 'unit' place); and
- a *point* is used to denote this place.

Example: The number 82047 is computed as

$$\begin{gathered}(8 \times 10 \times 10 \times 10 \times 10) + (2 \times 10 \times 10 \times 10) \\ +(0 \times 10 \times 10) + (4 \times 10) + (7 \times 1)\end{gathered}$$

or

$$\begin{gathered}(8 \times 10^4) + (2 \times 10^3) + (0 \times 10^2) + (4 \times 10^1) \\ +(7 \times 10^0)\end{gathered}$$

or

$$\begin{gathered}(8 \times 10,000) + (2 \times 1,000) + (0 \times 100) + (4 \times 10) \\ +(7 \times 1)\end{gathered}$$

## 3.1 Other Representations

We can extend symmetrically to the right of the point. E.g. the number 179.32 can be computed as:

$$(1 \times 10^2) + (7 \times 10^1) + (9 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$$

All numeration systems are representational, i.e. a form of coding. Coding should be devised so that the system:

- has few symbols that are easy to remember;
- is unambiguous;
- is economical in use (or presentation);
- is a useful quantitative measure;
- can be easily manipulated, e.g. to perform arithmetic. The Roman system was a classic counter-example: XXIV + XCVI = ?

Here is the crux; we can choose any base or radix that we like for a numeration system; it does not have to be ten!

Suppose we choose IIII standards as the basic set and use the digit symbols '0', '1', '2' and '3' for the null set and I, II and III standards, respectively.

Then, doing the calculations in decimal, for convenience, the number 3202 in this 'base-4' system (written as $3202_4$ to make this explicit) would be, in decimal: $(3\times4^3)+(2\times4^2)+(0\times4^1)+(2\times4^0) = 192+32+0+2 = 226_{10}$

## 3.2 A generalised numeration system

Let us assume we have a system with base or radix $B$ and, therefore, with $B$ distinct digits.

Let a $k$-digit number in this system be written as

$$a_{k-1}a_{k-2}\dots a_2a_1a_0$$

with $0 \leq a_i < B$ for all $i$.

The place value of the digit in the $i$th place to the left of the digit $a_0$ will be $B^i$. The value of the entire $k$-digit number will be

$$a_{k-1}B^{k-1} + a_{k-2}B^{k-2} + \cdots + a_2B^2 + a_1B^1 + a_0B^0$$

so the number's value $V$ is

$$V = \sum_{0}^{i-1} a_iB^i.$$

We are most used to a decimal (base ten) radix system.

In general:

- The larger the base, the more symbols that are needed for the digits;
- The smaller the base, the fewer the digit symbols that are needed and the easier they are to remember. They are also less easy to confuse or mis-interpret.

However, the smaller the base, the lengthier the representation (in general) of a number of any given magnitude. For example, $14_{10} = 112_3 = 1110_2$

We are also used to the base $B$ being a positive integer. But this need not be the case. $B$ can be negative:

Suppose $B = -2$. Then

$$\begin{array}{rcl} B^0 & = & 1 \\ B^1 & = & -2 \\ B^2 & = & 4 \\ B^3 & = & -8 \end{array}$$

and so on. We can still represent a number in the form $\sum_0^{i-1} a_i B^i$, now with $0 \leq a_i < |B|$. Some examples

$$\begin{array}{rcl} -4_{10} & = & 1100_{-2} \\ -3_{10} & = & 1101_{-2} \\ -2_{10} & = & 10_{-2} \\ -1_{10} & = & 11_{-2} \\ 1_{10} & = & 1_{-2} \\ 2_{10} & = & 110_{-2} \\ 3_{10} & = & 111_{-2} \\ 4_{10} & = & 100_{-2} \\ 6_{10} & = & 101_{-2} \end{array}$$

Note this is another *binary system.*

This system has the advantage of being able to represent both positive and negative numbers uniformly without using a sign. However, counting and arithmetic is somewhat harder in this system.

$B$ can be complex. Suppose $B = -1 + i$. Then

$$\begin{array}{rcl} B^0 & = & 1 \\ B^1 & = & -1+i \\ B^2 & = & -2i \\ B^3 & = & 2+2i \\ B^4 & = & 4 \end{array}$$

and so on. We now have

$$\begin{array}{rcl} -3_{10} & = & 10001_{-1+i} \\ -2_{10} & = & 11100_{-1+i} \\ -1_{10} & = & 11101_{-1+i} \\ 1_{10} & = & 1_{-1+i} \\ 2_{10} & = & 1100_{-1+i} \\ 3_{10} & = & 1101_{-1+i} \end{array}$$

But also

$$\begin{array}{rcl} -2i_{10} & = & 100_{-1+i} \\ -i_{10} & = & 111_{-1+i} \\ i_{10} & = & 11_{-1+i} \\ 2i_{10} & = & 1110100_{-1+i} \\ -1+i_{10} & = & 10_{-1+i} \\ 1+i_{10} & = & 1110_{-1+i} \end{array}$$

So we can represent positive and negative real and complex numbers in a single uniform way. This is also a binary system.

$B$ can be real. Suppose $B = \pi$. Then

$$\begin{array}{rcl} 1_{10} & = & 1_\pi \\ 2_{10} & = & 2_\pi \\ 3_{10} & = & 3_\pi \\ \pi_{10} & = & 10_\pi \\ 4_{10} & = & 1.2201\ldots_\pi \end{array}$$

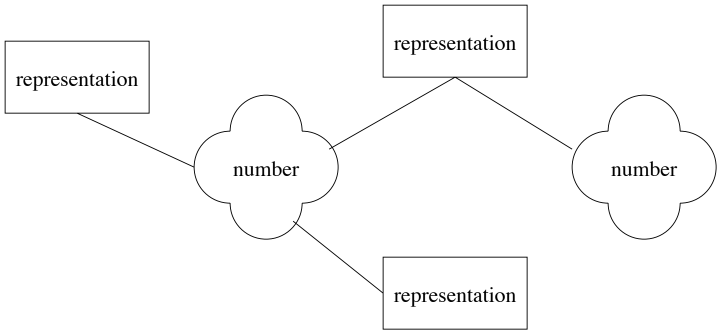


Figure 15: Representations of Numbers

Some integers have an infinite representation base $\pi$.

There are even *mixed radix* representations where we don't use simple powers of a number $B$. For example, using 1, 2, 6, 24, ..., $n!$, $(n+1)!$, ... we have

$$\begin{aligned} 1_{10} &= 1_! \\ 2_{10} &= 10_! \\ 3_{10} &= 11_! \\ 4_{10} &= 20_! \\ 5_{10} &= 21_! \\ 6_{10} &= 100_! \end{aligned}$$

We encode a number to this representation by successively subtracting off the largest multiple of the largest factorial: $119 = 4 \times 4! + 23 = 4 \times 4! + 3 \times 3! + 5 = 4 \times 4! + 3 \times 3! + 2 \times 2! + 1 = 4321_!$.

The thing to remember that a number can have many representations. We can write the number 2 as

- 2
- two
- deux
- II
- $10_2$
- $110_{-2}$

amongst many others. We pick a representation for its convenience. Also, we need to take care in interpreting representations. The pattern 11 can mean

- eleven (base 10)
- 3 (base 2)
- $i$ (base $-1+i$)

and so on. It might not mean a number at all, maybe a particular shade of green or a certain kind of noise.

Remember that numbers are abstract things. One number can have multiple representations, and one representation can mean many numbers. Always make sure you understand which correspondence you are using!

## 3.3 Converting Between Bases

We have already seen one example of a number conversion from a representation in a non-decimal radix system to decimal representation.

We can also convert from decimal to non-decimal radix systems by recalling that the value of a $k$-digit number in a radix B system is:

$$\begin{aligned} a_{k-1}B^{k-1} + a_{k-2}B^{k-2} + \quad \cdots & \quad + a_2B^2 + a_1B^1 + a_0B^0 \\ = & \quad ((\cdots((a_{k-1}B + a_{k-2})B + a_{k-3})B + \cdots a_2)B + a_1)B + a_0 \end{aligned}$$

We can use this factorisation to evaluate numbers in bases other than 10. Let us take, for example, the number $3373_8$.

$$\begin{aligned} 3373_8 &= ((3.8+3).8+7).8+3 \\ &= ((24+3).8+7).8+3 \\ &= (27.8+7).8+3 \\ &= (216+7).8+3 \\ &= 223.8+3 \\ &= 1784+3 \\ &= 1787_{10} \end{aligned}$$

This way of evaluation is called *Horner evaluation.*

We could use the same technique to convert from base 10 to, for example, base 8, but this would require us to have some ease with arithmetic base 8. Instead we tend to use a different technique, one based on division and remainder.

Suppose we take the decimal number 1787. Dividing by 8 each time:

$$\begin{aligned} 1787/8 &= 223 \quad \text{remainder } 3 \\ 223/8 &= 27 \quad \text{remainder } 7 \\ 27/8 &= 3 \quad \text{remainder } 3 \\ 3/8 &= 0 \quad \text{remainder } 3 \end{aligned}$$

If we read off the remainders from bottom to top we get $3373_8$.

This is the base 8 number system representation of the decimal number 1787.

It is left as an exercise to understand how this arises from the factorisation scheme given earlier.

Both Horner and the remainder methods work for any base $B$, not just base 8.

---

Why is Halloween like Christmas? Because Oct 31 = Dec 25.

---

We can easily imagine that if we were operating with a system whose elements were built with two-state devices, we could use the two states of a collection of devices to encode a number in binary representation.

This is exactly what happens in computer systems!

Base 10 machines have been indeed constructed: these store numbers much more compactly, but the electronics is hugely more complex to build. Instead of a simple two-state system, you must built a 10-state system. Moreover, arithmetic base 2 is trivial

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0c1 |

| × | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Arithmetic base 10 requires the machinery to implement 10-by-10 addition and multiplication tables.

### 3.3.1 The Store Unit

We now take a brief look at the store unit as we want to think about our data. Much later we shall come back and look at the other units in the model.

The basic element is a two-state device, usually known as a *bit* (from binary digit). Of course, others have been tried, in particular 10-state devices, though they turned out to be too complicated. Two-state devices are easy and cheap to build.

These elements are grouped together into *cells*. Nowadays, the usual size of a cell or *byte* is 8 bits and it is called a *byte*, but other cell sizes are possible. For example, some early machines had bytes of 9 bits.

The word *octet* is reserved for a byte of exactly 8 bits, though these days "byte" usually means 8 bits.

The *store* or *memory* consists of a collection of cells.

To identify each cell in the store, it is given an *address*. Usually, cell addresses are integer numbers, starting at zero and working up. A cell is typically the smallest addressable unit of a store, though some early machines were able to address each bit individually.

Bytes, these days, are a bit small: 8 bits only describes $2^8 = 256$ different things. So we collect together, say, 4 together in a *word*, and manipulate a word as a single object.

### 3.3.2 Cell contents

The states of the bits of a cell, in a pre-determined order, make up its *contents*. If the states of each two-state device, or bit, were called A and B, we could write the contents of a byte, for example, as: AABABBBAB. It is more normal to represent the states as binary digits and write the cell contents as: 00101101. The big problem with this is that when we see 101 we automatically think of this as a number as it *looks* like a number. It is not. It is three bits.

Note that whether we write the states as A and B, or as 0 and 1, these are *representations* of the actual states, which might electrical voltages or some other physical phenomenon. **They are not numbers, just bits**.

### 3.3.3 Cell addresses

Each cell in the store has a unique address that identifies it. Addresses are often numbered from 0 to $m-1$ for a store with $m$ cells.

Once a cell has been located a a specific address, its contents can be examined or changed.

Note that the contents of cells are simply bit patterns and it is impossible to tell by looking at them what they represent, e.g., data or instructions, and that data could be numbers or letters or colours or pitches and so on.

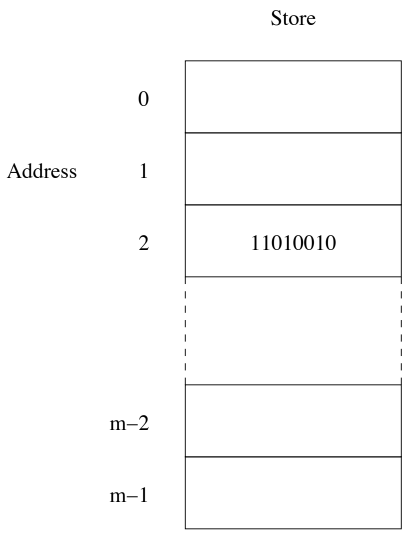


Figure 16: Cells in Store

A point to think about: Since addresses are themselves just positive integers, they can themselves be stored in cells as pieces of data.

A cell of $n$ bits can hold $2^n$ different bit patterns.

If we choose a base or radix of 2, we need only two digits (0 and 1, by convention) to represent any positive integer number, using exactly the same methods as were illustrated before for other radix systems.

Thus, for example, $216_{10} = 11011000_2$ and $11101_2 = 29_{10}$.

---

Converting between bases 2, 8 and 16 is easy.

Base 2 to 8. Group the bits in threes: $11110111_2 = \begin{array}{c|c|c} 11 & 110 & 111 \end{array} = 367_8$. And similarly for base 8 to base 2.

Base 2 to 16. Group the bits in fours: $11110111_2 = \begin{array}{c|c} 1111 & 0111 \end{array} = \mathrm{F}7_{16}$.

---

## 3.4 Unsigned integers

We have already seen that numbers can be coded in binary and stored in cells. In the following, we shall be looking at numbers stored in a byte of 8 bits. There is nothing magic about 8 bits: all of the following applies in a clear way to more (or fewer) bits, particularly 32 and 64 bit representations.

In an 8 bit byte, we can store one of 256 different numbers from 0 to $255_{10}$:

| | |
|---|---|
| 00000000 | 0 |
| 00000001 | 1 |
| 00000010 | 2 |
| 00000011 | 3 |
| ... | ... |
| 11111110 | 254 |
| 11111111 | 255 |

These are called *unsigned integers*, as they have no representations of representing negative values. This is the “best” way of representing unsigned integers: there are many bad ways

- not contiguous;

| | |
|---|---|
| 00000000 | 1 |
| 00000001 | 3 |
| 00000010 | 7 |
| ... | |

- not easy arithmetic;

| | |
|---|---|
| 00000000 | 127 |
| 00000001 | 126 |
| 00000010 | 125 |
| ... | |

And so on.

Arithmetic on this representation is easy: just use the familiar binary operations:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0c1 |

One thing to watch out for is *overflow*. If we add 1 to 255, $00000001 + 11111111 = 00000000$ in 8 bits, i.e., $1 + 255 = 0$. This is, in fact, addition modulo 256. We can't represent numbers larger than 256, so it should be expected that the answer comes out “wrong”. The solution is to use more bits.

So what should we do about negative numbers? There are several ways to represent *signed integers* using binary.

## 3.5 Signed integers: Sign and magnitude

We can adopt a convention that the first bit in a byte represents the sign of the number, the rest its *magnitude* (size).

| | |
|---|---|
| 00000000 | 0 |
| 00000001 | +1 |
| 00000010 | +2 |
| ... | ... |
| 01111110 | +126 |
| 01111111 | +127 |
| 10000000 | −0 |
| 10000001 | −1 |
| 10000010 | −2 |
| ... | ... |
| 11111111 | −127 |

So the representation of $-2$ is a 1 bit for the sign, then 0000010 for the 2.

This representation is easy to read, but has some curiosities:

- There are two representations for 0, namely 00000000 and 10000000. This is quite serious as testing for zero is one of the most common operations in programs (e.g., termination of loops) and having to test for two kinds of zero is extra work.
- This duplication means that there are only 255 distinct values represented, unlike unsigned's 256 distinct values.
- Arithmetic on this representation is fiddly: to add or subtract two numbers we need to strip the sign bit, do the add or subtract, and then put on the correct sign bit for the result. For example, add 2 to -3:

  1. input $00000010 + 10000011$
  2. strip signs $0000010 - 0000011$
  3. add/subtract $-0000001$ (regarding as normal binary values)
  4. restore sign 10000001

  The sign of the result is determined by examining the absolute size of the two arguments.

  Multiplication is marginally easier, apart from the confusion regarding 0.

On the other hand, this is a symmetric representation about 0.

Advantages of this representation:

- easy for humans to read
- symmetric about 0

Disadvantages

- fiddly arithmetic
- $-0$
- only represents $2^n - 1$ distinct numbers in $n$ bits

## 3.6 Signed integers: 1s complement

An improvement over sign and magnitude is *1s complement* representation.

In this representation the negative of a number is computed by flipping all the bits of that number. So, if 1 is 00000001, then -1 is 11111110.

| | |
|---|---|
| 00000000 | 0 |
| 00000001 | $+1$ |
| 00000010 | $+2$ |
| ... | ... |
| 01111110 | $+126$ |
| 01111111 | $+127$ |
| 10000000 | $-127$ |
| 10000001 | $-126$ |
| 10000010 | $-125$ |
| ... | ... |
| 11111110 | $-1$ |
| 11111111 | $-0$ |

---

An alternative view: 1s complement can be regarded as the top bit being $-(2^{n-1}-1)$and the $n-1$ other bits being normal unsigned binary. So $10000101 = -(2^7-1)+2^2+2^0 = -127+4+1 = -122_{10}$, compared to $01111010 = 122_{10}$.

If we have $a_{n-1}a_{n-2}\ldots a_2a_1a_0 \leftrightarrow a_{n-1}(1-2^{n-1}) + a_{n-2}2^{n-2} + \ldots + a_1 2^1 + a_0 2^0$, then flipping bits we get$(1-a_{n-1})(1-2^{n-1}) + (1-a_{n-2})2^{n-2} + (1-a_1)2^1 + (1-a_0)$. So adding these two values $(1-2^{n-1}) + 2^{n-2} + \ldots + 2^1 + 2^0 = (1-2^{n-1}) + (2^{n-1}-1) = 0$. Thus flipping bits is negation.

---

The sign of a number is still indicated by the top bit—1 for negative, 0 for positive or zero—but the bit is now part of the number itself.

Arithmetic is moderately easy. Suppose we have a 3 bit representation

| | |
|---|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | −3 |
| 101 | −2 |
| 110 | −1 |
| 111 | −0 |

Addition is as usual

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0c1 |

For subtraction we use the same table: to compute $n-m$ we actually do the addition $n+(-m)$. This means, given the representation for $m$, we negate it by flipping its bits, and then add this to the representation for $n$.

Some examples

- $1+2 \to 001+010 = 011 \to 3$.
- $3+1 \to 011+001 = 100 \to -3$. We can't represent 4 in three bits.
- $1-2 \to 001-010 = 001+101 = 110 \to -1$.
- $2-2 \to 010-010 = 010+101 = 111 \to -0$

We see the problem with $-0$ is real.

The above examples do not cover all the possibilities, however. There is an issue regarding overflow: *if we have a carry from the top bit, we add it back into the bottom end.* This sounds a bit strange, but is exactly what we need.

- $-0+1 \to 111+001 = 1000 \to 000+1 = 001 \to 1$. We must circulate the carry.
- $-1+-1 \to 110+110 = 1100 \to 100+1 = 101 \to -2$.

- $3 - 1 \to 011 - 001 = 011 + 110 = 1001 \to 001 + 1 = 010 \to 2$.
- $3 - 0 \to 011 - 000 = 011 + 111 = 1010 \to 010 + 1 = 011 \to 3$.

---

This happens (when not overflowing) when a bit is carried from the $(n-2)^{\text{nd}}$ place to the $(n-1)^{\text{st}}$ place and then added in:

$$\begin{array}{rcll}
111 + 001 & \leftrightarrow & ((1-2^2) + 2^1 + 2^0) + (0.(1-2^2) + 0.2^1 + 2^0) & \\
 & = & (1-2^2) + 2^1 + 2.2^0 & \\
 & = & (1-2^2) + 2.2^1 + 0.2^0 & \text{want to carry here} \\
 & = & 0.(1-2^2) + 0.2^1 + 0.2^0 + (1-2^2) + 2^2 & \text{instead it hits the } 1-2^2 \text{ term} \\
 & = & 0.(1-2^2) + 0.2^1 + 0.2^0 + 1 &
\end{array}$$

---

Advantages of this representation:

- fairly easy arithmetic
- add and subtract use the same tables (i.e., use the same hardware)
- symmetric about 0

Disadvantages

- $-0$
- only represents $2^n - 1$ distinct numbers in $n$ bits
- $n - n = -0$
- negative numbers moderately difficult for humans to read

## 3.7 Signed integers: 2s complement

The $-0$ is a problem and a waste: let's get rid of it by taking the 1s complement and shifting the negative numbers down the line a bit:

| | |
|---|---|
| 00000000 | 0 |
| 00000001 | +1 |
| 00000010 | +2 |
| ... | ... |
| 01111110 | +126 |
| 01111111 | +127 |
| 10000000 | −128 |
| 10000001 | −127 |
| ... | ... |
| 11111110 | −2 |
| 11111111 | −1 |

Rearranging to put 0 in the (near) middle:

| | |
|---|---|
| 01111111 | +127 |
| 01111110 | +126 |
| ... | ... |
| 00000010 | +2 |
| 00000001 | +1 |
| 00000000 | 0 |
| 11111111 | −1 |
| 11111110 | −2 |
| ... | ... |
| 10000001 | −127 |
| 10000000 | −128 |

This may seem like an arbitrary fix and we have no reason to believe it might be useful, however, we can see from the table that going up the column is adding 1 in the binary representation and also adding 1 in the integer values. There is a kink at 0 where the carry bit falls off the top, bit otherwise there is a natural correspondence. It turns out that this *2s complement* is a very good representation for signed integers.

---

An alternative view: 2s complement can be regarded as the top bit being $-2^{n-1}$and the $n-1$ other bits being normal unsigned binary. So $10000101 = -2^7 + 2^2 + 2^0 = -128 + 4 + 1 = -123_{10}$.

Now $a_{n-1}a_{n-2}\ldots a_2a_1a_0 \leftrightarrow a_{n-1}(-2^{n-1}) + a_{n-2}2^{n-2} + \ldots + a_1 2^1 + a_0 2^0$, then flipping bits and adding 1 we get$(1-a_{n-1})(-2^{n-1}) + (1-a_{n-2})2^{n-2} + (1-a_1)2^1 + (1-a_0) + 1$. Adding, $-2^{n-1} + 2^{n-2} + \ldots 2^1 + 2^0 + 1 = -2^{n-1} + 2^{n-1} + 1 - 1 = 0$. Thus flipping then adding 1 negates.

---

As with 1s complement, the sign of a number is still indicated by the top bit—1 for negative, 0 for positive or zero. The representation is asymmetric: it goes from $-2^{n-1}$ to $2^{n-1}-1$ in $n$ bits, but it represent one more distinct value than both sign and magnitude and 1s complement. Neither of those can represent $-2^{n-1}$.

To negate any value, positive or negative, we flip the bits, then add 1 (using a normal unsigned add). So as 1 is 00000001, then flipping gives 11111110, and add 1 to get 11111111 as the representation of −1. Given 11111111 for −1, flip to 00000000, then add 1 to get 00000001 as the representation of 1. To negate 0 or 00000000, we flip to 11111111, add 1 to get 00000000 for $-0 = 0$ (ignore the carry). Thus there is no −0 problem. We do find that negating −128 is weird: −128 is 10000000; flip 01111111; add 1 to get 10000000, which is −128 again. Of course, we cannot represent +128 is this system so we are bound to get the wrong answer.

---

Note that while there is only one 0, there are *two* values with $-n = n$, namely 0 and −128. Even though $0 + 1 = 1$, we don't have $-128 + 1 = 1$.

---

Negation here is slightly harder than 1s complement negation, but has a massive advantage: arithmetic is now even easier. Use the normal addition table, and if there is an carry from the top, just drop it.

| | |
|---|---|
| 011 | 3 |
| 010 | 2 |
| 001 | 1 |
| 000 | 0 |
| 111 | -1 |
| 110 | -2 |
| 101 | -3 |
| 100 | -4 |

- $1 + 2 \to 001 + 010 = 011 \to 3$
- $3 + 1 \to 011 + 001 = 100 \to -4$. We can't represent 4 in three bits.
- $1 - 2 \to 001 - 010 = 001 + 110 = 111 \to -1$.
- $2 - 2 \to 010 - 010 = 010 + 110 = 000 \to 0$. The carry just disappears.
- $3 - 1 \to 011 - 001 = 011 + 111 = 010 \to 2$.
- $3 - 0 \to 011 - 000 = 011 + 000 = 011 \to 3$.

Advantages of this representation:

- easy arithmetic
- add and subtract use the same tables (i.e., use the same hardware):

$$\begin{array}{cr} & 001 \\ + & 101 \\ \hline & 110 \end{array} \qquad \begin{array}{cr} & 1 \\ + & 5 \\ \hline & 6 \end{array} \qquad \begin{array}{cr} & 1 \\ - & 3 \\ \hline & -2 \end{array}$$

- add and subtract use the same tables as unsigned arithmetic
- single 0
- represents $2^n$ distinct numbers in $n$ bits
- even numbers end in 0, odd numbers in 1

Disadvantages

- not symmetric about 0
- negative numbers moderately difficult for humans to read

## 3.8 Summary

Some computer languages, in particular C, give us the choice of number representation.

- `unsigned int n;`
- `signed int n;`
- `int n;` this is the same as `signed`.

Also, C has some flexibility on the number of bits we use

- `char` is often 8 bits
- `short` is often 16 bits
- `int` is often 32 bits
- `long int` is often 64 bits

So, typically, a `unsigned char` is 8 bits unsigned, while `long int` is 64 bits signed. C always uses 2s complement.

For Java sizes are fixed and definite:

- `byte` is 8 bits
- `short` is 16 bits
- `int` is 32 bits
- `long int` is 64 bits

All are 2s complement, no unsigned.

(a) Gosling: unsigned are tricky for programmers, so not in Java
(b) Just not implemented in early Java, and never got put in.
C# has unsigned.

In $n$ bits we have ranges

- unsigned: $0$ to $2^n - 1$
- S&M: $-2^{n-1} + 1$ to $2^{n-1} - 1$
- 1s: $-2^{n-1} + 1$ to $2^{n-1} - 1$
- 2s: $-2^{n-1}$ to $2^{n-1} - 1$

Note that $2^{32} = 4,294,967,296$ (about 4 billion) for a 32-bit unsigned, and $2^{31} = 2,147,483,648$ (about 2 billion) for a 32-bit signed representation.

Notice what we are saying here: depending on what we want at the time, the bit pattern 11111111 can either mean 255 if we are thinking of unsigned numbers, or -1 if we are thinking of 2s complement numbers.

**A bit pattern has no intrinsic meaning. We must have some convention to ascribe a meaning to it.**

For example, given 3 bits:

| bits | Un | S&M | 1s | 2s |
|---|---|---|---|---|
| 000 | 0 | 0 | 0 | 0 |
| 001 | 1 | 1 | 1 | 1 |
| 010 | 2 | 2 | 2 | 2 |
| 011 | 3 | 3 | 3 | 3 |
| 100 | 4 | -0 | -3 | -4 |
| 101 | 5 | -1 | -2 | -3 |
| 110 | 6 | -2 | -1 | -2 |
| 111 | 7 | -3 | -0 | -1 |

The bit pattern 101 can mean 5, $-1$, $-2$, $-3$ or anything else! For example, one scheme has 0110001 to represent the character "1" (ASCII). Another has 11110001 for "1" (EBCDIC). Arithmetic on these representations are hard, but nobody cares as we don't generally want to do arithmetic with characters.

There are other number representations than we have covered here, and the bit pattern may not even be representing a number. For example, we can use bit patterns to represent characters. The *American Standard Code for Information Interchange* (ASCII) is a convention on how 7-bit patterns should represent characters. Some examples:

| | |
|---|---|
| 0100001 | ! |
| 0100110 | & |
| 0110001 | 1 |
| 0110010 | 2 |
| 0111001 | 9 |
| 1000001 | A |
| 1000010 | B |
| 1000011 | C |
| 1100001 | a |
| 1100010 | b |

ASCII can represent up to 128 different characters.

Other conventions exist, for example EBCDIC, which uses 8-bit strings. The EBCDIC representation for "A" is 11000000. Another is UNICODE, which uses up to 31 bits, meaning it can represent about 2 billion different characters (in particular, it is not limited to the Western alphabet).

---

Words for counting.

| power of 2 | prefix | | binary prefix | | power of 10 | name | |
|---|---|---|---|---|---|---|---|
| $2^{10}$ | kilo | K/k | kibi | Ki | $10^3$ | thousand | |
| $2^{20}$ | mega | M | mebi | Mi | $10^6$ | million | |
| $2^{30}$ | giga | G | gibi | Gi | $10^9$ | billion | SI billion |
| $2^{40}$ | tera | T | tebi | Ti | $10^{12}$ | trillion | filestores this large are common |
| $2^{50}$ | peta | P | pebi | Pi | $10^{15}$ | quadrillion | |
| $2^{60}$ | exe | E | exbi | Ei | $10^{18}$ | quintillion | all storage on Internet? |
| $2^{70}$ | zetta | Z | zebi(?) | Zi | $10^{21}$ | sextillion | |
| $2^{80}$ | yotta | Y | yobi(?) | Yi | $10^{24}$ | septillion | |
| $2^{90}$ | nona | N | nobi(?) | Ni | $10^{27}$ | octillion | |
| $2^{100}$ | dogga | D | dogbi(?) | Di | $10^{30}$ | nonillion | |

The words "kilo" and "mega" and so on mean different things to different people. "Kilo" can mean 1000 or 1024 depending on context. The International Electrotechnical Commission (IEC) defines a standard that says "kibi" is definitely 1024, and "mebi" is definitely $2^{20}$. So "20MB" means $20 \times 10^6 = 20,000,000$ bytes, while "20MiB" means $20 \times 2^{20} = 20,971,520$ bytes. This standard is only slowly catching on.

The old British (actually, European) "billion" used to be $10^{12}$, or a million million. This usage has fallen out of use and the standard (SI) billion is now $10^9$, or a thousand million.

# 4 Numbers II: Floating Point

We have seen a variety of representations of integers on computers. We now look at non-integers, namely numbers with digits after the (decimal) point. We shall concentrate on em floating point representations as these are the most common these days, but be aware there are alternatives, in particular *fixed point*.

The representations are called "floating" point as the point can move about, allowing us to represent things like 10.2, 1.0001, 10000.1234. This in contrast to fixed point when we always have the same number of places after the point.

## 4.1 Scientific Notation

Non-integer numbers are often presented in scientific notation (the right-hand side version below):

$$\begin{aligned} 16.37 &= 1.637 \times 10^1 \\ -234.0 &= -2.34 \times 10^2 \\ 0.0000367 &= 3.67 \times 10^{-5} \\ 10360000000 &= 1.036 \times 10^{10} \end{aligned}$$

You may also see the variant $16.37 = 0.1637 \times 10^2$ with a zero before the decimal point.

This notation has many virtues:

- It is simple to generate, understand and manipulate;
- It can represent numbers of widely varying magnitudes in a compact and essentially *fixed size* representation;

- It can, if properly constructed and used, preserve the number of significant digits in the representation; and
- It is *accurate* to the precision of the representation.

## 4.2 Definition of Normalized Scientific Representation

Any number can be written in terms of two quantities, namely: a signed *mantissa*, $m$, and a signed *exponent*, $e$:

$$\text{Number} = \pm m \times R^{\pm e}$$

where $R$ is the *radix* of the system (10 in decimal).

The notation is *normalised* if *either*

- $1 \leq m < R$, *or*
- $m = 0$ and $e = 0$

(we assume, for the moment, that $R > 0$, actually $R > 1$)

This simply means that

- our number is written $d$.something $\times R^e$ where the digit $d$ is not 0, or
- our number is 0

Note that $1 \leq d$.something $< R$.

In the normalized decimal system the normalisation criterion becomes

$$1 \leq m < 10$$

Thus 12.3 can be written in the normalised form $1.23 \times 10^1$ (radix 10); while $123.0 \times 10^{-1}$ and $0.0123 \times 10^3$ are not normalised by our definition.

---

This goes back to the idea of many different representations of a single number. Here we are picking a single representation and calling it "normalised".

---

Note that normalization preserves the *maximum number of significant digits in a fixed length mantissa* and we avoid wasting digits on zeroes as in, e.g:

$$230 = 0.0023 \times 10^5$$

The normalised representation is much more *compact* than the usual decimal representation for very large and very small numbers.

## 4.3 Accuracy of Scientific Notation

Given a fixed mantissa size (let us take, say, 5 digits), then we can only *approximate* numbers which have a greater number of significant digits (as opposed to integers, which are simply correct or not).

For example,

$$10260030476000$$

is, to 5 significant digits:

$$1.0260 \times 10^{13}$$

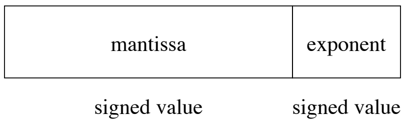
Now, in a computer we will only have a fixed number of bits to represent the mantissa, so it is essential we use the best possible (meaning most accurate) representation we can.

## 4.4 Floating Point in Computers

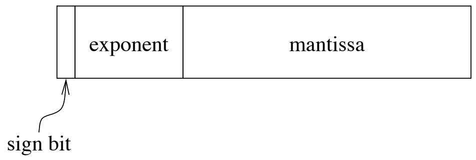
We shall be using a normalised scientific notation, but with radix 2:

$$\text{Number} = \pm m \times 2^{\pm e}$$

so that it can be represented in principle as two signed parts (in practice it is done somewhat differently):



Now we could use any of the signed integer representations have have seen before, but here is how the IEEE standard does it (simplified somewhat) for a 32 bit representation.



- there is a single sign bit to determine the sign of the number, so this is a sign and magnitude representation. The magnitude is given by the mantissa and exponent.
- eight bits represent the exponent in *127 excess* form
- 23 bits represent the mantissa as an unsigned value

The 127 excess form takes the binary value and subtracts 127:

$$\begin{array}{lcr} 00000001 & \rightarrow & -126 \\ 00000010 & \rightarrow & -125 \\ 01111110 & \rightarrow & -1 \\ 01111111 & \rightarrow & 0 \\ 10000000 & \rightarrow & 1 \\ 11111101 & \rightarrow & 126 \\ 11111110 & \rightarrow & 127 \end{array}$$

The special bit patterns

- 0000000 represents a *denormalised* number (see below)
- 1111111 represents *infinity* or *not a number*.

The mantissa is also represented specially. Since we are using a normalised representation, we know that the mantissa must satisfy

$$1 \leq m < 2$$

that is, the mantissa must start 1.something. IEEE is of the opinion that it is wasteful to include bits for the the leading 1 and so simply drops it.

For example, represent 5. Base 2 this is $101 = 1.01 \times 10^{10}$. Thus we use the bits 01 for the mantissa. Reversing, given the bits 01, we prepend a 1. to get the full 1.01.

The main advantage of this is that we can get one extra bit of accuracy packed into the representation. Disadvantages are this include being a format that is not terribly human friendly and arithmetic will be very hard. Fortunately, engineers that design the hardware to do the arithmetic understand the IEEE format very well and generally make a good job of it.

One exception to all this is the number 0: it is represented by all-zeros

| | | |
|---|---|---|
| 0 | 00000000 | 00000000000000000000000 |
| 1 | 8 | 23 |

Negative zero is allowed, too:

1 00000000 00000000000000000000000

The IEEE format comes in various sizes. There are 32 bit (*(single) float*), 64 bit (*double*) and 128 (*long double*) bit standards. The longer varieties are more accurate and can represent larger ranges of values, but of course take up more room and are slower to manipulate. The double format is the most widely used currently.

---

In fact hardware is often optimised so that it is usually faster to use double than single! A float operation is implemented as "convert to double; do the operation; convert result to float".

---

| Name | Length mant | Length exp | exp excess | accuracy decimal figs | largest | smallest | smallest denorm |
|---|---|---|---|---|---|---|---|
| float | 23 | 8 | 127 | 7 | $3 \times 10^{38}$ | $1 \times 10^{-38}$ | $1 \times 10^{-45}$ |
| double | 52 | 11 | 1023 | 16 | $2 \times 10^{308}$ | $2 \times 10^{-308}$ | $5 \times 10^{-324}$ |
| long double | 112 | 15 | 32767 | 34 | $10^{4932}$ | $10^{-4932}$ | $10^{-4966}$ |

These figures are approximate.

In terms of precision, 1 decimal digit takes about 3.3 bits. This really means that 3 decimal digits are equivalent to about 10 bits. This is because $1000 = 10^3 \approx 2^{10} = 1024$.

In single precision $1_{10} = 1.0 \times 10^0$ is

0 01111111 00000000000000000000000

meaning

- 0; sign bit, non-negative
- 01111111; exponent 127 excess $127 - 127 = 0$
- 00000000000000000000000; mantissa, prepend 1. to get $1.0\ldots0$

In single precision $-5_{10} = -101 = -1.01 \times 10^{10}$ is

1 10000001 01000000000000000000000

meaning

- 1; sign bit, negative
- 10000001; exponent 127 excess $129 - 127 = 2$
- 01000000000000000000000; mantissa, prepend 1. to get $1.01\ldots0$

## 4.5 Denormalised Numbers

If we require that all floating point numbers be normalised, then we are excluding many bit patterns from representing numbers. For example, the smallest positive number we can represent is

0 00000001 00000000000000000000000

meaning $1.0 \times 10^{-1111110} = 1.0 \times 2^{-126} \approx 10^{-38}$ written in base 10. This is due to the requirement that a normalised number starts 1. (base 2).

IEEE also allows *denormalised* numbers, that is, numbers that *don't* start with a 1., but only for numbers that can't be represented in a normalised way. So we are allowed $0.1 \times 2^{-126}$ and $0.01 \times 2^{-126}$ and so on. Two things of note

1. this only works for small numbers, not big numbers (think about it!)
2. we have to lose the idea of having an implicit initial 1. in denormalised numbers, as they don't start with a 1.

The exponent 00000000 is reserved for denormalised numbers and is how we recognise we have one (or, indeed 0 in the special case). When we have a denormalised value, we have exponent $-126$ and we don't prepend 1. to the mantissa.

Thus

0 00000000 00100000000000000000000

represents $.001 \times 10^{-1111110} = 0.125 \times 2^{-126} \approx 1.5\times^{-39}$.

The advantages of this are that we can extend the range of representable numbers. For example, for single floating point we extend the range from $10^{-38}$ to $10^{-45}$. The disadvantage is at extra complexity of the representation.

## 4.6 Infinity and Not-a-Number

This still leaves a few bit patterns that don't represent numbers, namely those with exponent 11111111. IEEE use patters with this exponent for two things:

1. Infinities. An exponent of 11111111 and a mantissa of 00000000000000000000000 is taken to be *infinity*, positive or negative according to the sign bit. This value has the properties on might naïvly expect of infinity such as

   - 1.0/0.0 = infinity
   - infinity + 1 = infinity
   - infinity × 2 = infinity

   and so on. This is so, if there is a bug in our program somewhere, the infinity is propagated throughout the calculation and we can see something went wrong when we get infinity out as an answer.

2. Not-a-number. An exponent of 11111111 and a non-zero mantissa is *not-a-number*. Just as infinity, NaN is propagated throughout the calculation so we can tell at the end something went wrong. We can get NaN in a few cases, such as

   - 0.0/0.0
   - infinity − infinity
   - infinity/infinity
   - infinity × 0

---

NaNs come in two flavours: *quiet* and *signalling*.

- QNaN: the most significant bit of the mantissa is set. These propagate through computations as described above.
- SNaN: the most significant bit of the mantissa is clear. These cause an interrupt in the processor if an operation is performed on one.

QNaN's are indicate *indeterminate* operations, while SNaN's indicate *invalid* operations.

SNaN's are good as the initial value of a variable: if the value is used before the variable is set, and error is signalled.

---

Examples (for 32 bit floats):

| | |
|---|---|
| 0 00000000 00000000000000000000000 | 0 |
| 1 00000000 00000000000000000000000 | $-0$ |
| 0 10000001 01100000000000000000000 | $1.011 \times 2^2 = 101.1 = 5.5$ |
| 0 00000000 01100000000000000000000 | $0.011 \times 2^{-126} \approx 4.4 \times 10^{-39}$ |
| 0 11111110 11111111111111111111111 | $\approx 3 \times 10^{38}$ |
| 0 11111111 00000000000000000000000 | $\infty$ |
| 1 11111111 00000000000000000000000 | $-\infty$ |
| 0 11111111 00000000000000000000001 | NaN |
| 1 11111111 11111111111111111111111 | NaN |

## 4.7 Arithmetic in a Normalised Representation

We illustrate addition in decimal:

$$(9.6 \times 10^1) + (6.6 \times 10^0)$$

1. Denormalize the smaller operand by adjusting its exponent to equal that of the other operand:

$$(9.6 \times 10^1) + (0.66 \times 10^1)$$

2. Add the mantissae and retain the common exponent:

$$(10.26 \times 10^1)$$

3. Renormalise the mantissa and adjust the exponent if necessary:

$$(1.0 \times 10^2)$$

In this, at the same time as the renormalisation, we have rounded the answer to two significant figures.

And multiplication

$$(2.7 \times 10^{-5}) \times (5.6 \times 10^6)$$

1. Multiply the mantissae and add the exponents. This will yield:

$$(15.12 \times 10^1)$$

2. Renormalize (and round) if necessary:

$$(1.5 \times 10^2)$$

In real life, the process is more subtle than this, as we need to treat cases of underflow and overflow, infinities and NaNs.

Arithmetic with denormalised numbers is similar, but just more fiddly. Luckily, hardware to do arithmetic on IEEE numbers is common and these days is built into most processors.

# 5 Machine instructions and execution

We saw that, as well as data in various forms, *sequences of instructions* are also held in the store of a von Neumann computer.

We recall that any operation needs a specification of:

- the *operation* to be executed; and
- the *operand* or *operands* upon which the operation is to be performed: typically two operands, as in addition; though it may be one, as in negation; or even, just occasionally, none - for example if we wish to stop the computer!

## 5.1 An Example coding for an instruction

Operands are most naturally coded as an *address* where the actual operand can be found. Taking the example of the addition operation, the essential information to be coded would be, symbolically:

```
ADD     10,12
```

meaning that the piece of data held at the address or location 10 of the store is to be added to that held at the location 12 (the *contents* of those locations can be readily changed).

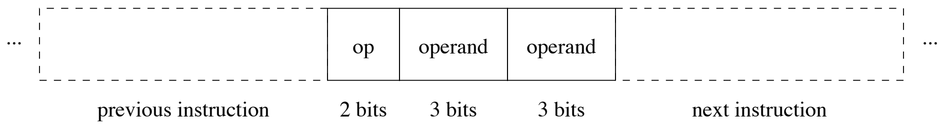
In a typical store in which the cells are 8-bit bytes, each byte can hold 256 different bit patterns.

If a single byte is to hold an instruction, we imagine it must therefore hold at least one coded bit pattern representing an operation and up to two more patterns representing the operands.

It is important to realise that different types of processor allocate bit patterns to operations in different ways. For example, a Sparc does it differently from an Intel and both are different from a ARM. The choices of which bit patterns are usually made after a great deal of research into which are the best ways to proceed. Only when two architectures are identical (such as AMD and Intel) can a program compiled for one architecture run on another.

While pretty much every processor has an ADD instruction, quite which bits and how they are used varies from processor type to processor type.

Let us assume that we divide the byte in memory as follows:



Then, in this example, we can have *only 4 possible operations* and a choice (for both operands) from *8 possible operands*, which is not enough to be useful.

## 5.2 Coding instructions

The design of computers has as one of its main tasks the coding of instructions so that as much power as possible (e.g. in the range of operations) can be as compactly coded as possible.

In reality, instructions occupy more that one byte of storage and can take a variable number of bytes (e.g. in the Motorola 68000 series of machines, a minimum of 2 bytes and as many as 10 bytes of store may be used for one instruction).

We might find that:

- 2 bytes are used to hold an operation specification (we can thus have a maximum of 65,536 different operations, which should be enough!) and
- 2 (or more likely 4) bytes are used to specify the address of each of the operands (with 2 bytes, each operand can be from any one of 65,536 different locations in store).

Question: If 4 bytes are used, how many operand addresses are accessible?

## 5.3 Compacting the space occupied by an instruction

If, in an operation like addition, we allow one of the operands to come from any location in the store, but we restrict the second operand to one particular place, we should not need to code that into the instruction at all. It would be assumed, or *implicit*.

We could therefore reduce very considerably the space we need to store an instruction. For example, adding 1 is such a common operation, many processors have an `INC` instruction that takes just one argument and increments it.

## 5.4 Registers and the ALU

Storage, usually for very specific kinds of information, can be provided in other places than the store unit. It is usually provided by very fast, electronically active storage called *registers* (to distinguish them from ordinary store locations). Like locations of the store, however, they consist of arrays of bits, though they are usually more than one byte in capacity.

The ALU, where data transformations are carried out, contains one or more registers for holding the data where it can be operated upon. They go under a variety of names, e.g. data registers, or *accumulators* (especially in older literature).

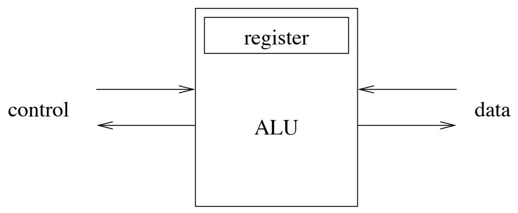
The reason behind this is as follows: processors are very fast these days (GHz), but memory speeds are not as fast (800MHz?). This means a processor would be forever waiting for the memory to catch up, thus wasting the advantage of having a fast processor. One of the ways of fixing this is to have registers on the chip. These can run at the full speed of the processor.

---

When working on paper documents it is convenient to store them in a filing cabinet. But it would be a pain to have to retrieve each page individually from the cabinet, amend it, and then return it to the cabinet every time you wanted to make a change. Instead, we pull a bunch of documents and keep them handily on the desk. The desk is not big enough to hold all the contents of the cabinet, so we occasionally have to put some documents back and fetch some more out. However, it is much faster to keep the documents we are working on on the desk.

---

## 5.5 ALU Register Structure



The path to main memory is slow relative to the speed the register can be accessed (but it still very fast in real terms!).

## 5.6 One Address Codes

If a data register is always used as one operand of a two-operand operation (like addition, for example), then we do not need to specify this operand explicitly in the instruction:

| |
|---|
| operation<br>2 bytes |
| operand<br>2 bytes |

Note that picturing the operand on top of the operand is merely a convenience: for the bytes in memory read left-to-right, top-to-bottom. In this example, we have allowed ourselves 2 bytes for the operator and 2 for the operand: of course, these sizes are processor-dependent.

This of course assumes that there are operations a) to copy an operand from a memory location into a register ($LOAD$) and b) to save them from a register ($STORE$).

```
LOAD    10
ADD     12
STORE   14
```

This adds the value in location 12 to the value in location 10 and stores the result in location 14.

## 5.7 One and a Half Address Codes

If, instead of restricting one of the operands to a single, fixed place, we allow it to come from one of a very small number of places (e.g., one from a set of, say 8, data registers), then we need only 3 bits in the instruction word to code for it. This greatly extends the power of the instruction set without greatly increasing the amount of store we need to hold the instructions. We might even squeeze these bits into the operation code:

| | |
|---|---|
| operation<br>13 bits | register<br>3 bits |
| operand<br>16 bits | |

Notice that the instruction does not nicely fits within the bytes: the operation code takes all of the first byte and 5 bits of the next byte. Instructions are normally described in terms of bits in each part, rather than bytes, for just this reason.

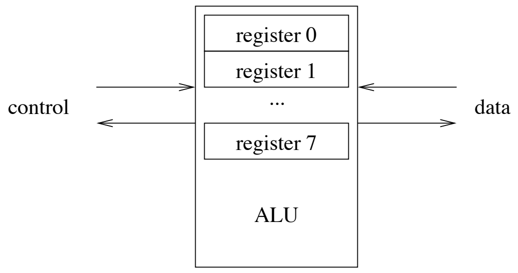
Three bits gives us 8 registers, normally numbered 0 to 7. Modern processors can have large numbers of registers. For example. there are 256 data registers in the new Intel chip.

---

128 integer; 128 float; and the rest are miscellaneous branch and machine registers.

---

Our ALU picture is now more like



## 5.8 Zero Address Codes

Some instructions require no argument. For example `INC` in a single-register machine; or the `HALT` instruction that stops the computer.

## 5.9 Two Address Codes

For example, `ADD D1, D4`.

## 5.10 Three Address Codes

For example, `ADD D1, D4, D2` for adding `D4` and `D2` and putting the result in `D1`. Or possibly, `D1` and `D4` and putting the result in `D4`. There are variants on the direction of action of such codes from architecture to architecture.

## 5.11 Operation Types

There are many combinations of source and destination register and memory.

- `ADD 12, 34` memory-memory
- `ADD 12, D0` memory-register
- `ADD D2, 34` register-memory
- `ADD D2, D4` register-register
- `ADD D1, D3, D2` register-register-register

and so on. A given processor may allow some subset of these.

In the last the value in register `D1` is added to the value in `D3` and stored in `D2`. Or, in some processors, the value in `D3` is added to the value in `D2` and stored in `D1`. You always need to read the manual to check.

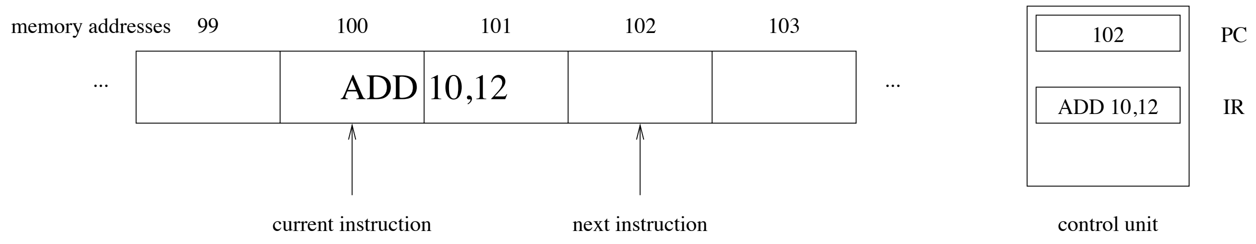


Figure 17: Instruction Register and Program Counter

## 5.12 Addresses and Sequencing Instructions

A program can be regarded as an ordered sequence of instructions. If we store the successive instructions of the sequence in successively higher-addressed memory locations, we can carry them out by stepping through the memory addresses in order.

To control this access, we add some registers to the *control* unit.

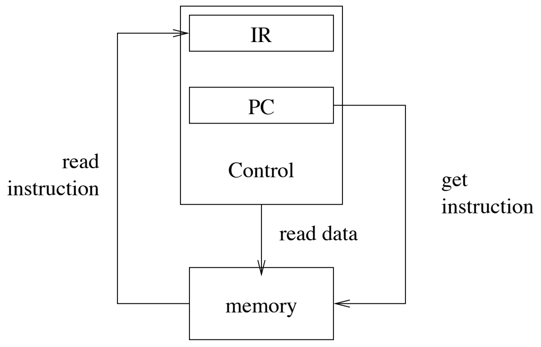
Unlike the data registers in the ALU, these are not for general use by the programmer.

## 5.13 Registers of the Control Unit

To find an instruction we need to know its address in store. A register in the control unit holds the address (in store) of *the next instruction to be executed.* It works like a counter and is called the *program counter*, or PC (the next instruction address register, NIAR in older books).

We also need to hold a copy of the *current instruction* (i.e., the contents of the instruction word) while it is being executed. For the moment we can think of an instruction as a single cell that can be held in a single *instruction register*, or IR.

So our control unit looks like this:



Memory in fact also has (inaccessible) registers to store the address that is being accessed (the *memory address register*, or MA) and the contents of that location while they are being written into or read from the store (the *memory data*, MD, or *memory buffer register*, MB):

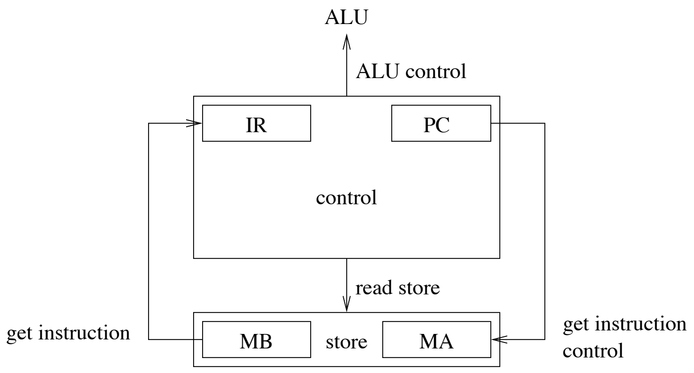
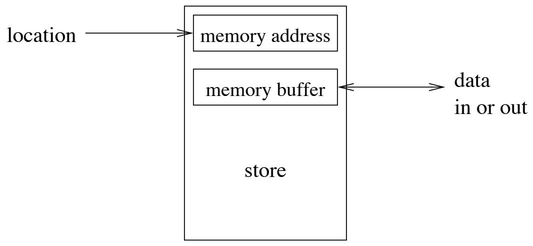


Figure 18: Instruction Access



Note that accessing memory is not instant: there is a gap between asking for the data at a memory location and getting it.

We now have a complete picture for instruction access (figure 18).

## 5.14 Instruction Execution

To execute an instruction it must first be fetched from store to the control unit.

Once there, it must be interpreted to discover what operation is to be performed and how the operands are to be accessed. The instruction is held in the instruction register while this (possibly complex) *decoding* process occurs.

The operands (if any) must actually be brought from store or registers to the ALU and the operation upon them must be carried out, using control signals supplied by the control unit.

Finally, the result of the operation (if any) must be stored in memory or a register.

Thus we have the process

- Fetch instruction
- Decode instruction
- Fetch argument(s)
- Execute operation
- Store result

## 5.15 The Fetch Phase

We can now see how this might be carried out:

1. Contents of the PC are placed in the MA: this sets the store location of the next instruction;
2. Contents of the cell whose address is in MA are placed in the MB: this gets the contents of the cell;
3. Contents of the MB are placed in the IR: the instruction is now ready for decoding; and
4. Contents of PC are incremented: so that the next instruction in sequence will be accessed next.

## 5.16 The Decoding Phase

This phase need to determine (a) what the instruction is actually supposed to do, (b) what arguments it should use (if any) and (c) where to put the result (if any).

This can be a very complicated operation. Some processors have a very haphazard way of encoding the information into the bits of the instruction. The Intel 386 architecture is a good example. The location of the bits that tell you where the arguments are depends on the operation, so you must decode the operation before you can start thinking about arguments.

Other processors, such as the MIPS, have a very simple layout. This means the decoding phase is very fast. For example the bits for the operands are always in the same place in the instruction. So we can start decoding the operands *at the same time* as decoding the operation. This save time, and makes processing faster.

The Intel is an example of a *complex instruction set*, or CISC design, while the MIPS is a *reduced instruction set.* or RISC design.

## 5.17 The Fetch Arguments Phase

If the instruction refers to registers, getting the values to (say) the add unit in the ALU is easy. If the instruction refers to memory locations, then a process much as the instruction fetch occurs

1. The required address is placed in the MA;
2. Contents of the cell whose address is in MA are placed in the MB: this gets the contents of the cell;
3. Contents of the MB are placed in the ALU: the value is now ready for the operation.

Indirect addressing (see later) is correspondingly more complicated.

This phase may repeat to fetch zero or one or two (rarely more) arguments.

## 5.18 The Execution Phase

This is the actual processing of the data. It may produce a result to be stored in the next phase, or it may have some other effect. One useful effect is to put a value into the *program counter.* Now the program counter is the address of the next instruction to be loaded and executed. If we put a different address into the PC, then a different instruction will be loaded and executed. This is how the processor can *jump* to different parts of the program and not simply execute each instruction in strict sequence. More on this shortly.

## 5.19 The Store Phase

This is nearly identical to the fetch argument phase, but in reverse. If the destination is a register, this is easy. Otherwise, to store in memory:

1. The required address is placed in the MA;
2. The value to be stored is put in the MB;
3. Contents of the MB are placed in the store.

## 5.20 Unconditional Branches

We now return to the idea of storing to the PC.

So far, the in model we have described, we can only execute a list of instructions taken from sequential addresses. But if we have instructions that can alter he contents of the PC, we can branch to instructions out of the normal sequence.

These are called *unconditional branch* or *jump* operations.

The execute phase of these operations takes the operand address part of the word in the IR and transfers it to the PC.

The next operation then comes from the location addressed by the PC, in the normal way, but the important difference is that this new instruction is (generally) *not* the next in sequence.

## 5.21 Conditional Branches

Even more powerful is being able to make a choice of whether to branch.

These can be implemented if the PC can be set according to the state of the ALU so that the next operation comes from one location if the ALU is in a particular state (e.g., state $s$ is TRUE), or from another location if the ALU is not in that state (e.g., state $S$ is FALSE).

This appears to involve *two* operands, the two addresses.

However, the usual arrangement is to execute the next instruction from the next location in the normal sequence if the condition is FALSE, and branch (by changing the PC contents) only if the condition is TRUE. Then we need only one explicit operand.

Example. In a single register architecture, jump to a location 24 for the next instruction if the contents of the data register in the ALU are zero.

If TRUE (i.e. contents of the data register are zero): we set PC contents to the address 24

If FALSE: we take the next instruction from the location following that of this instruction. In this case, the PC contents are unaltered, which is fast and simple!

We might write this, for example, as

```
JMZ 24
ADD ...
```

This would jump to address 24 if the value of the register is zero, otherwise we proceed normally to the `ADD` instruction.

Memory location 24 can be before of after the `JMZ` instruction, it does not matter.

---

> It can also be the *same* as the `JMZ` instruction. In this case the processor executes the `JMZ`, jumps to location 24, executes the `JMZ`, jumps to location 24, ... The program is stuck in a loop.

---

In a multiple register architecture we might have

```
JMZ D3, 24
ADD ...
```

which would jump only if the value in register `D3` was zero.

Conditional branches can come in a variety of forms. One-bit *flags* or *condition codes* are usually used to denote the state of various aspects of the ALU, for example, whether the data register is zero or not, whether it holds a negative number (in 2's complement) or not, and so on. For example, and `ADD` instruction might set the condition code for zero if the result of the add is zero and set the condition code for negative if the result was negative. In this way we can test the results of operations.

Conditional branches allow the programmer to construct the programming structures that make computers so powerful:

- Decisions: if a condition is true do X, else do Y;
- Loops: do Z a fixed number of times; or do Z until a condition is true; or do Z while a condition is true.

## 5.22 Example of Execution: the MC68000

We now give a more detailed example of the execution of an instruction. We shall illustrate this using the Motorola MC68000 processor. This processor was popular many years ago. but is now regarded as an antique. But it has several features that make it good to teach ideas of machine operation.

The 68000 had eight 32-bit data registers (plus some other registers). The PC was initially 24 bits, then 32 bits in later versions. The IR was 16 bits. Instructions were (at least) two bytes long, and memory addresses were either two bytes or four bytes.

Let us take the example of an addition operation that adds the 16-bit contents of a specified location (we shall choose 1202 hexadecimal) to one of the eight data registers of this machine (we shall choose register 5).

For a 68000, a 16 bit value is called a *word*, while a *byte* is 8 bits and a *longword* is 32 bits.

The word "word" means different things to different people. In the context of the 68000 is it 16 bits (2 bytes). In more modern processors a word is generally 32 bits (4 bytes). It often means "the natural size of a chunk of data for this processor."

Here is the instruction as it might be written in assembler:

```
ADD $1202, D5
```

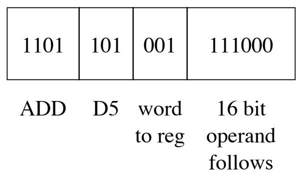


Figure 19: `ADD` instruction

The `$` indicates a hex value.

The bits of the instruction word are 110101001111000.

- The first four bits, 1101, indicate this is an `ADD` instruction.
- The next three bits, 101, indicate the register. Here we want register D5.
- The next three bits tell us whether the add is *to* or *from* the register, and how long the data should be, namely a byte, a word (2 bytes) or a longword (4 bytes).

  | | Byte | Word | Lword |
  |---|---|---|---|
  | store to reg | 000 | 001 | 010 |
  | reg to store | 100 | 101 | 110 |

  The bit patterns 011 and 111 are not used here. So `ADD D5, $1202` would require 101.
- The last six bits are the *mode* of the instruction and indicate that a 16 bit value follows immediately after the instruction and it is the address of the store location.

So the value 1202 would follow: 0001001000000010. Thus the complete instruction plus data is

11010100111100000001001000000010

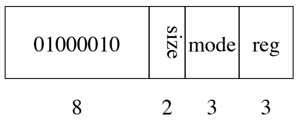
As this is hard to read, we normally write this as

| | |
|---|---|
| DA | 78 |
| 12 | 02 |

To execute this instruction:

- the 16 bit value in memory location 1202 is fetched (i.e., the word starting at location 1202, meaning the byte at 1202 and the byte at 1203).
- it is added to the word in register 5
- the result is left in register 5

Not only do the first four bits indicate the operation, but they also indicate the function of the following bits, i.e., the next three are a register and so on. Other instructions have the register (and other) fields in different places. For example, the `CLR` instruction, which clears a word to zero looks like



Here it takes eight bits to indicate the `CLR`, then the register is the last three bits. ("Size" is byte, word or long, "mode" is direct, indirect, indexed and so on.)

## 5.23 The Mode Field

One of the the operands in the `ADD` instruction, a register in this case, is given directly. The mode field tells us where to find the other. In the above example a 16 bit address of a memory location is indicated.

If the last six bits were 11100, this would mean the 16-bit extension word that followed would have been treated not as the *address* of the operand, but as the *value* of the operand. So, if the extension word contained 1202 (hex.) as before, we would add the *value* 1202 (hex) to data register 5.

This could be written

```
ADD #$1202, D5
```

The `#` indicates an *immediate* value, not an address.

Clearly there are potentially many different ways of using the bits of this field and this is important in determining the power of a computer's instructions.

We can summarise what we have discovered by observing that even the simplest, one-address, instruction in general has not just two, but three parts:

1. Operation specification;
2. Operand specification;
3. Mode specification (to determine how the operand is interpreted).

Quite how these are laid out in bits in the instruction is up to the designer of the processor.

## 5.24 Modes of Addressing

We have seen

- *direct* memory address `ADD $1202, D5`
- *register* `ADD D6, D5`
- *immediate* `ADD #$1202, D5`

There are also

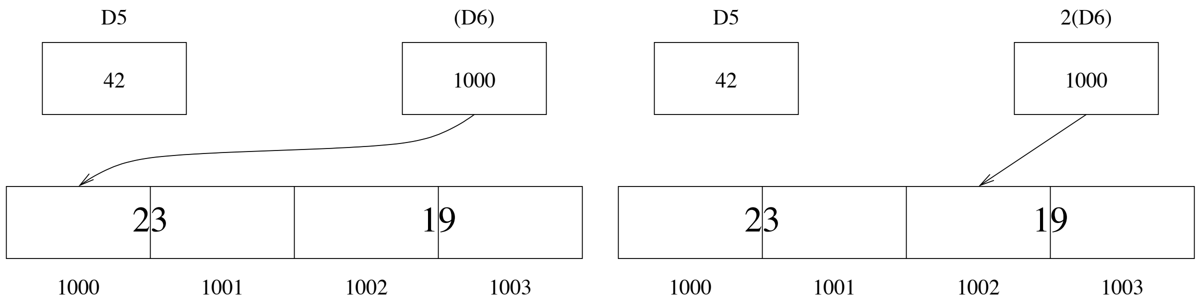


Figure 20: Indirect Addressing

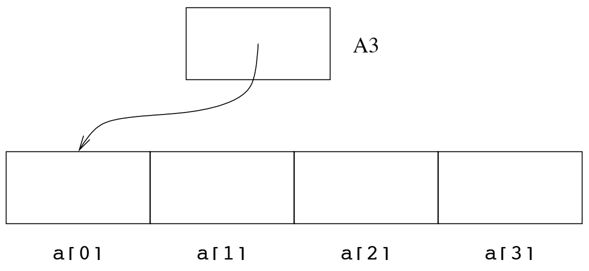


Figure 21: Array Access

- *indirect* `ADD (D6), D5` the value in register 6 is used as the address of the memory location for the value. Suppose `D6` contains 1000, while `D5` contains 42 and the word at memory location 1000 contains 23. Then the above add would result in 65. In contrast `ADD D6, D5` would add $1000 + 23 = 1023$.
- *indirect with displacement* `ADD 2(D6), D5` the value in register 6 is used as the address of the memory location for the value and then a constant *displacement* value is added, 2 in this example. This gives the location of the required value. Suppose `D5`, `D6` and memory location 1000 are as above, and memory location 1002 contains 19. Then this `ADD` would be $42 + 19 = 61$.

In fact, the 68000 has eight other *address* registers that are usually used to do indirect, indexed etc., addressing.

```
ADD (A6), D5
```

This was a way around the limits in processor design at the time: data registers could be used in arithmetic instructions but not in many indirect modes. On the other hand, address registers could be used in several kinds of indirect modes, but only a few arithmetic operations. Address registers and data registers are not distinguished in modern processors.

These kinds of memory addressing are **very** important and appear everywhere in code. Consider, for example, the C or Java array access

```
a[2]
```

This corresponds directly to the indexed mode: if the array `a` is stored starting at address 1000, then `1000(D2)` refers to array value “`a[D2]`”. Varying the value of register varies which element of the array we get.

## 5.25 More Addressing Modes

Other modes included

- *indirect with postincrement* `ADD (A3)+, D5` As indirect, but the register has 1 or 2 or 4 according to the data size is added to it after fetching the requisite value from memory. This is a way of stepping along an array of values: next time we execute this instruction, `A3` is already indicating the next value in the array.
- *indirect with predecrement* `ADD -(A3), D5` As postincrement, but the register is decremented *before* fetching the value from memory. This is good for stepping *down* an array.
- *indirect with index* `ADD 2(A0,D3), D5` This takes the value in `A0`, adds the value in `D3`, and adds the displacement 2 to get the address. This allows us to vary the displacement by changing the value in a register.

## 5.26 Yet More Addressing Modes

And there are more.

- *program counter with displacement* `ADD 4(PC), D5` This is like indirect with displacement, but the register used is the program counter. This allows us to access values relative to *where the program is currently executing*
- *program counter with index* `ADD 4(PC, D3), D5` The obvious extension of indirect with index.

And that's not all. There are several more specialised modes, too.

Summary: there are many ways of specifying the arguments to an operation, some simple, some complex.

## 5.27 A RISC Architecture

For contrast, we briefly describe a RISC architecure, the MIPS. It is a three-register instruction set, with operations like

```
add $1, $2, $3
```

to add the contents of registers 2 and 3 and place the result in register 1.

There are just four formats of instruction:

- register `$1`
- direct address `lw $1, 100` to load a word (32 bits) from memory location 100 to register 1
- indirect with offset `sw $1, 100($2)` to store a word (32 bits) from register 1 to memory location 100 off register 2
- immediate `addi $1,$2,100` to add 100 to register 2 and put the result in register 1

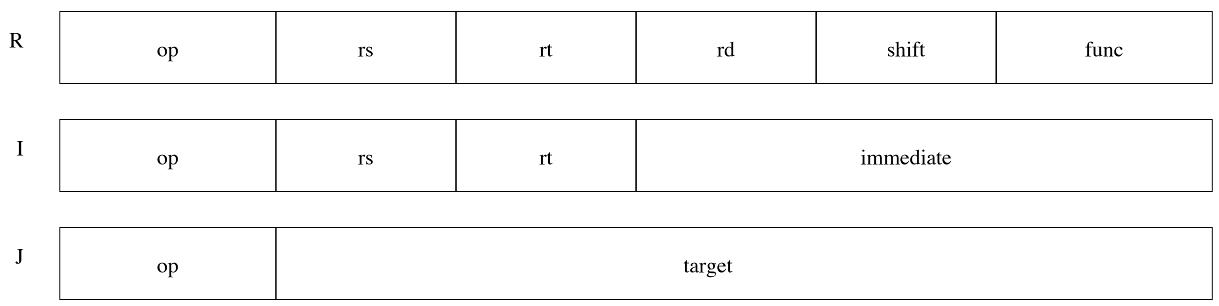


Figure 22: MIPS architecture

Only load and store can use direct and indirect addressing; all other instructions are register only.

There are just three formats of instruction:

- R format: operator and three registers
- I format: operator, two registers and 16 bits of immediate value
- J format: bits of operator and 26 bits of address

The important points to note are that

- the operand is always specified by the top 6 bits
- the register fields (5 bits each, 32 registers) are always in the same place

This means that MIPS instructions are very easy to decode and the various fields can be decoded *in parallel*, i.e., faster. With the 68000 we had to partially decode an instruction before we could figure out what registers it acted upon. Here we can prepare three registers in parallel with determining the operation: in the case of an I and a R we don't need some of those registers but that doesn't matter, we are faster overall.

## 6 Assemblers

Let us suppose that we wish to write a very simple program that will add one 16-bit (2-byte) signed integer stored at location 1200 (hexadecimal) to another 16-bit signed integer stored at location 1202 (hexadecimal). The result is to be stored as a 16-bit signed integer at location 1204 (hexadecimal).

We wish to run this program on a machine with a Motorola MC68000 CPU.

It is roughly equivalent to writing a single high-level instruction like

```
z := x + y;
```

At low level on a 68000 CPU we need to execute the following sequence of three instructions:

1. Move the first operand from its location in store into a data register (we shall choose register 5 of the eight available).
2. Add the contents of the second operand location to the contents of the data register 5.

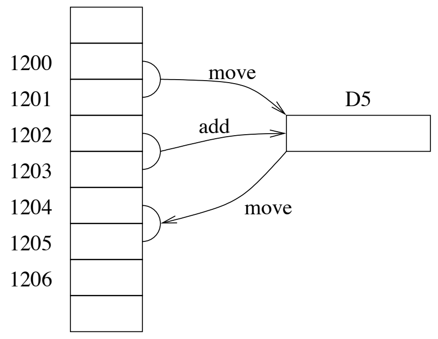


Figure 23: Addition on the 68000

3. Move the contents of the data register 5 (the sum) into the store location reserved for the result.

In the 68000 CPU, instructions are coded using multiple bytes.

For a 'move' operation, the first two bytes hold a control word specifying the operation and the way in which the operands are to be accessed. The bit pattern format of the control word is:

| | destination | | source | | |
|---|---|---|---|---|---|
| Op | reg | mode | mode | reg | |
| 4 | 3 | 3 | 3 | 3 | number of bits |

The MOVE operand is

| | |
|---|---|
| 0001 | move byte |
| 0011 | move word |
| 0010 | move long word |

So we require 0011. The Destination register is number 5, binary 101. The destination mode is register direct, which is indicated by 000.

The source location is memory direct, which is indicated by 000 for register and 111 for mode.

We get

| | destination | | source | |
|---|---|---|---|---|
| 0011 | 101 | 000 | 111 | 000 |

The memory location address is given by the next two bytes. This is (hex) 1200.

| | | | | | |
|---|---|---|---|---|---|
| 0011 | 101 | 000 | 111 | 000 | 0001001000000000 |

The other move operation, to store the result after the add is similar.

- 0011: move word
- destination memory direct: 000 111
- source register 5 direct: 101 000

Followed by the memory location, this time 1204.

| | | | | | |
|---|---|---|---|---|---|
| 0011 | 000 | 111 | 000 | 101 | 0001001000000100 |

The ADD we have seen before.

| | destination | | source | | |
|---|---|---|---|---|---|
| op | reg | mode | mode | reg | address |
| 1101 | 101 | 001 | 111 | 000 | 0001001000000010 |

We can now show what the bit patterns of the program in store will be. Let us suppose that the program begins at location 1000 (hexadecimal). We will use hexadecimal notation:

| | |
|---|---|
| 0FFE | ... |
| 1000 | 3A38 |
| 1002 | 1200 |
| 1004 | DA78 |
| 1006 | 1202 |
| 1008 | 31C5 |
| 100A | 1204 |
| 100C | ... |

The data area is also in store, at location 1200. This will contain the two operands and a 2-byte space for the result. Let us suppose that the operands are ABCD and 702C:

| | |
|---|---|
| 11FE | ... |
| 1200 | ABCD |
| 1202 | 702C |
| 1204 | *any* |
| 1206 | ... |

## 6.1 Pipelining

When we execute a sequence of instructions we go

```
F D F E S F D F E S F D F E S
        |         |         |
```

We get one completed (in the parlance, *retired*) instruction every 5 cycles. Modern processors can *overlap* instructions

```
F D F E S
  F D F E S
    F D F E S
      F D F E S
```

That is, as we decode the first we fetch the second; as we fetch arguments for the first we decode the second and fetch the third. And so on.

After a while we get one completed instruction *every* cycle. This is 5 times as fast as previously.

This is called a *pipeline*, made popular by the Ford Motor Company (there called the *assembly line*).

"5" is not a magic number, if we subdivide the execution stages more finely (e.g., break up the fetch stages into smaller parts) we get a *deeper* (sometimes called *longer*) pipeline that has more overlap and so greater speedup. The speedup is maximised if we keep the pipeline full at all times: the speedup is not so good while we are filling the pipeline at the start.

There are problems, though.

- Branches. Sometimes the next instruction to be executed is not the next partially decoded instruction in the pipeline. When we branch we want the next instruction from somewhere else entirely. If this happens we *break the pipeline*, which means we throw away the part-done work, empty the pipeline and re-fill it starting at the jump target.

  This means that a branch will cause a temporary drop in speedup as we re-fill the pipeline. Unfortunately, branches happen lot in real code and if we have a deep pipeline (which is good for a large speedup) we are more likely the pipeline will cover a jump instruction. Fixes include

  1. writing code to avoid branches: this sometimes forces unintuitive code that does apparently silly things just to avoid that jump
  2. predictive decoding, where the processor tries to guess the target of a conditional branch (probably using some statistics of previous branches) and decoding its guess in the hope that is it more often correct than not
  3. multiple pipelines, where there is a pipeline for *each* possible branch and both are decoded. This gets worse when you realise a really deep pipeline may cover several branches and so require a pipeline for every eventuality. Eventualities can multiply exponentially.

- Instruction conflicts. Often one instruction needs the result of the previous one. This means that the fetch of one instruction is dependent on the completion of the execution stage of the previous. This can cause complications in data flow through the processor.

  ```
  ADD D1, D5
  STORE D5, 1000
  ```

  An internal pass-through mechanism is sometimes used to get the value in D5 to the right place in the pipeline in time.

- Load delays. Fetching is relatively slow: certainly slower than a decode or an execute stage. Sometimes the pipeline *stalls* when one stage takes unexpectedly long. This is very bad as we are halting the progress of 5 (or more) instructions, not just the one. Stalls must be avoided, often by writing code carefully so that data is loaded a long time before it is needed.

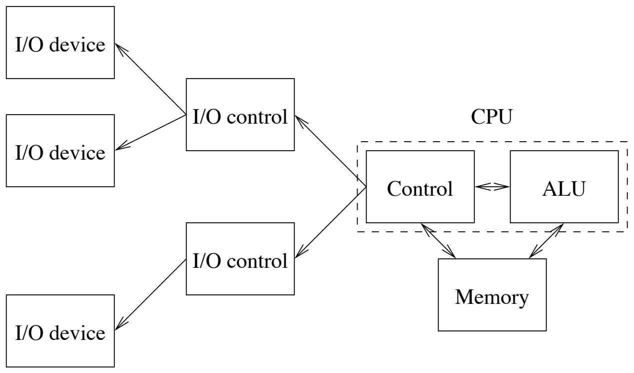


Figure 24: I/O Control

```
LOAD D1, 1000  --- load from memory
ADD D4, D4  --- do something else for a while
SUB D4, #1
ADD D1, D6  --- eventually can use result D1
```

If you try to use D1 before it is ready, you might find (a) the processor stalls, or (b) you get the old value that was in D1, or even a junk value.

- Unequal stage times. For a pipeline to flow smoothly each stage must take (roughly) the same amount of time. Otherwise the pipeline moves at the rate of the slowest stage. This means that the subdivisions of execution must be chosen carefully to make this so. The crude 5 stages above are certainly not good in this regard. For example, MUL and DIV take a long time. We might

  1. break MUL (and DIV etc.) into smaller parts: early SPARC had a "multiply step" instruction that you had to repeatedly use
  2. disallow the use of the result of MUL for a while (c.f. load).

     ```
     MUL D1, D2
     ADD D4, D4  --- do something else for a while
     SUB D4, #1
     ADD D2, D6  --- eventually can use result D2
     ```

     If you attempt to use D2 too soon you get the same possiblities as for load.
  3. make mul faster!

---

Think of the construction of a car in an assembly line and how this relates to each of the problems above.

---

## 7 I/O architectures

### 7.1 Control of I/O Devices

There are many kinds of I/O device that we might want to connect to the computer, including disk, video card, sound card, CD, keyboard, mouse and so on. In fact, the designer of the computer cannot know in advance which devices might one day be connected. So the machine must be built to be flexible: this is achieved by the use of *I/O control units*. These units hide the complexity of individual devices from the

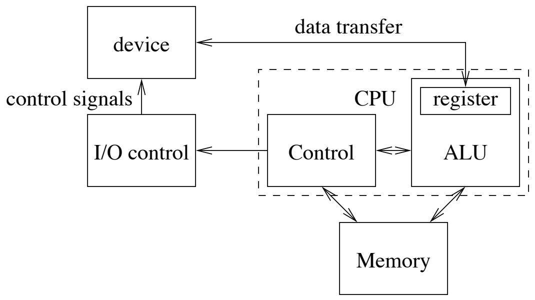


Figure 25: I/O Control

main processor. An I/O control unit connects to the processor in a standard way that the designer can rely on, but connect to a variety of I/O devices in whatever way the makers of the devices see fit.

If a new kind of device is invented, we need only build a suitable I/O control unit to interface it to the processor.

How does the control system work?

- An instruction from the control unit to the I/O control unit causes the I/O control unit to send signals to the device to set it into action.
- This will happen if a stored instruction is fetched to the IR in the CPU and turns out, upon decoding, to be an I/O operation.
- Data will need to be transferred to the device (a write) or from the device (a read). An obvious place to put this data is in a data register.

In general, there may be several devices attached to the computer. In this case, the I/O instructions must be structured and coded into a bit pattern with three parts that describe:

1. An I/O operation code; e.g., data transfer, state change, state report
2. The direction of transfer (read/write); and
3. The *address* of the device, so that data can be placed on a common data highway, or bus and be picked up only by the target device. The address is the name of the device we want: this might well be just a number.

So: "get me 10 bytes from disk" or "write these bytes to video".

The use of a *data bus* simplifies the design of the processor. Instead of multiple connections into the processor we have just one. This simplification is at the cost of a more complex I/O subsystem:

- a shared data bus is expensive to build
- is is *shared*, so that there might be *contention* between the various devices that might want to send data to the processor
- this means that programs must be written carefully so that they avoid contention
- or that some hardware (i.e., at a cost) be designed to share the limited resource of the bus appropriately

However, the benefit of simplifying the processor I/O far outweighs the costs of the shared bus.

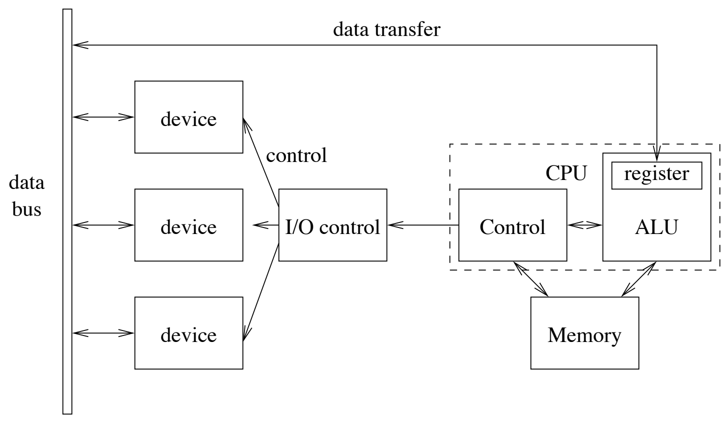


Figure 26: Data Bus

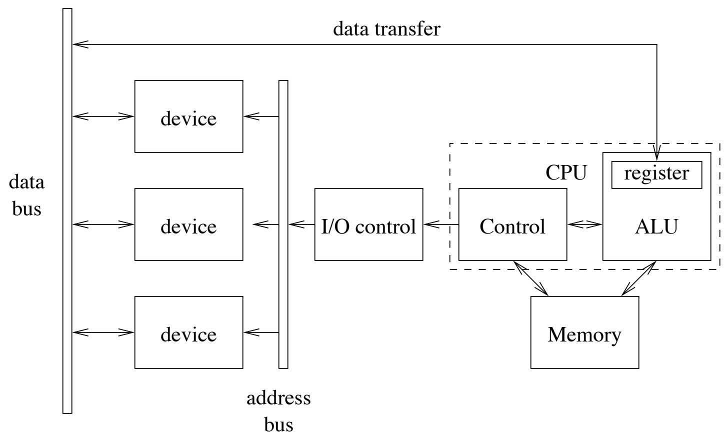


Figure 27: Address Bus

## 7.2 Multiple Device Control

The idea of multiple devices sharing a common bus can be extended: the address information can also be placed on a common I/O address bus.

We can easily imagine that within each device's electronics there is (not unlike the store unit) a data register that holds data passing into or out of the device, also a means of identifying the device's own address.

This is called an *address bus*, as it carries (along with other control information) the address of the appropriate device.

The same trades-off of flexibility and cost apply to the address bus.

## 7.3 Memory-mapped I/O

Many architectures connect the CPU to the store unit by a single electronic highway, or bus.

The number of store locations that can be accessed are determined by the number of bits in the address lines of the bus. So if the bus has, say 24 address lines, we can address any of $2^{24}$ different locations. This is the *address space* of such a machine.

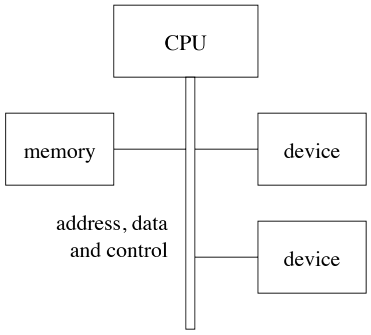


Figure 28: Memory Mapped I/O

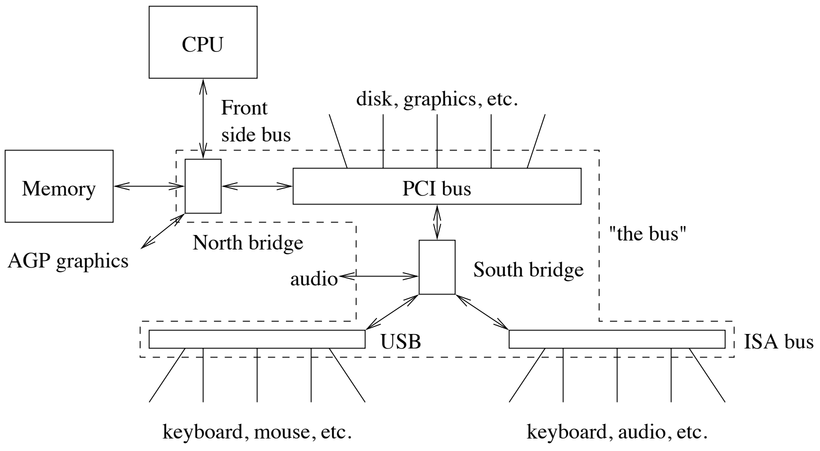


Figure 29: North and South Bridge

But the address space need not consist only of store locations. Some may be *device addresses*.

This is *memory-mapped* I/O.

We now treat store and devices in just the same way. An address could refer to a location in memory, or a register in a device. For example, address region 0000-0FFF could be memory, 1000-1FFF could be assigned to device 1, 2000-2FFF could be assigned to device 2, and so on. Real assignments are much more complicated, of course.

The big benefit of this arrangement is that no special I/O instructions are needed in the machine's instruction set!

We can operate I/O devices by loading and storing I/O data as if we were loading and storing data in memory. In the real world, this picture of a single bus is somewhat misleading. The bus is usually split up into smaller, easier to control parts.

## 7.4 Bus Architectures

Here we outline two bus architectures as used in modern machines.

The *North and South Bridge* architecture splits the bus up into two parts. The first, connected to the CPU by an I/O controller called the *North Bridge*, is primarily for devices that must have very fast access. This

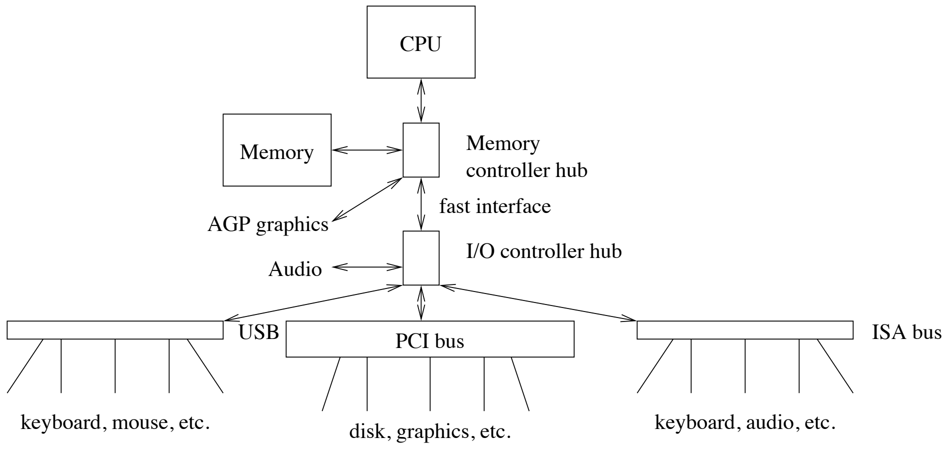


Figure 30: Intel Hub Architecture

includes main memory and items like graphics cards which need to move large amounts of data in very short times.

Connected to the North Bridge is the main bus, a *PCI* bus. This interfaces to most high speed devices like disk and networks.

Slower devices, which do not need the cost of a high speed connection, are connected to the PCI bus via another controller, the *South Bridge*. This can interface to slower devices such as mice and keyboards, often via further buses, such as *USB* or the ancient *ISA* bus.

The latest architecture in common use is the *Intel Hub architecture*. As its name suggests, there are controllers, the *hubs*, to which other buses or devices connect. Very fast memory or graphics connect to a *memory hub*, which itself interfaces directly to the CPU via a very fast connection.

Other devices and buses connect to a general I/O hub.

This architecture is somewhat simpler than the North/South bridge architecture, and so can be made to run at higher speeds.

## 7.5 Direct Memory Access

Some devices are so fast (disks, network cards, graphics cards) that even the time that is required to carry out the operations that prepare for the transfer of data between registers of the ALU and an I/O device is too long. Furthermore, while the CPU is doing rote copying of data is is not doing the work it was designed for: computation.

Instead of using the CPU, a more complex and expensive form of I/O hardware is employed, called *direct memory access*, or DMA. In older books this was called a *data channel*.

Using DMA, an I/O device can be connected directly to the memory of a computer, under the control of the I/O control unit.

Previously, to move a large block of data from (say) a disk to memory, the CPU would have to

- load a word from the disk
- save the word to memory

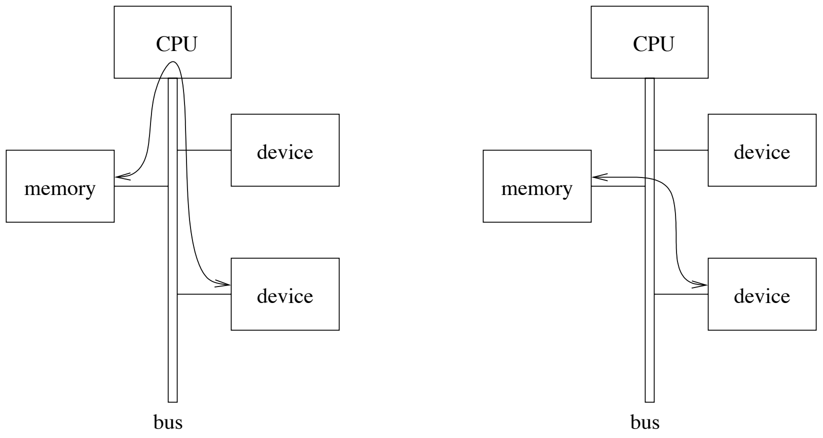


Figure 31: Block copy and Direct Memory Access

- repeat until the whole block is done

This is (a) slow and (b) ties up the CPU for the duration of the transfer. With DMA this becomes

- send “start transfer” instruction to the disk (this would probably be writing a value to the disk’s memory mapped I/O
- go and do something useful

The device can transfer data directly to memory without the intervention of the CPU.

When the transfer is done the CPU can return to do whatever it wants do with the data now safely in memory. The CPU could be sent an interrupt by the device to indicate completion.

Now, again, this frees up the CPU to some useful work instead of waiting on the device.

The “start transfer” must include information on

- which device
- the location of the block of memory in the device
- the length of the block
- the address of the source/destination in memory
- the direction of transfer, i.e., to or from the device.

If a device is capable of DMA, it is usually much faster to use it than to have the CPU do a word-by-word copy. Of course, the cost is extra complexity in the device and bus to support DMA transfers.

## 7.6 Interrupts

The kind of I/O we have been talking about, both direct and memory mapped, is called *programmed I/O*, because it is the programmer who writes code to access data and transfer it to or from I/O devices.

The major problem with I/O programming is that I/O device operation usually involves sort sort of electro-mechanical process (e.g., depressing keys on a keyboard, or waiting for a magnetic disc to spin into the correct position for data to be transferred to or from a specific place on the disc surface).

This is inherently much, much slower than the electronic transfer of data within a CPU or between the CPU and the memory. This, if we are not careful, means the processor can spend a lot of time waiting on these slow devices. Therefore we need to have some mechanism for the device to indicate its readiness, so that the processor need not spend all its time waiting.

## 7.7 Device Flags

Each device has a register that contains (minimally) a one-bit register which acts as a *device status flag.* A "flag" is just the name for a single bit, and is usually 1 for "up" or "set", and 0 for "down" or "unset" or "clear".

When an I/O transfer is started, the device status flag is unset (usually to the '0' state) to indicate that the device is *busy* doing a transfer. When the transfer is complete the device itself sets the device status flag (i.e. sets it into the '1' state), to indicate that the transfer is complete and that the device is *idle*, that is, ready to perform another I/O transfer.

Device operation can therefore be controlled by testing the device status flag and not starting an I/O transfer until the device is ready (device status flag is 1).

We can see that, in general terms, we could synchronise the CPU and a device by carrying out the following kind of program (it is exemplified here for output):

- get data ready to send
- test device status
- if device not ready, go to previous step
- write data to device

This is called *busy waiting.* The processor does nothing but wait for the device to be ready for the transfer.

Often better is to use *polling*:

- get data ready to send
- test device status
- if device not ready, do something else for a while, then go to previous step
- write data to device

The idea here is that we continue processing on some other problem while occasionally checking the device to see if it ready. When it *is* ready, then we can do the transfer.

Busy wait is

- easy to program
- has good responsiveness: when the device is ready, the processor responds immediately

- expensive on processor time: the processor does no other useful work while waiting.

Polling

- is harder to program: we need to able to write our programs to check devices regularly and this can be quite complex if there are lots of devices with varying requirements
- has worse responsiveness than busy wait: if the device just misses a poll it may be a long time (relatively) before the next poll. This can cause problems if the device absolutely must be processed as soon as possible (e.g., safety critical devices)
- is slightly wasteful of computer time: each failed poll is a slice of time when the processor could have been doing something useful. The slices are much smaller than busy wait, of course.

Busy wait is sometimes used when we know a device is going to be ready very soon, and it is judged that the cost of going away and doing something else for that short period of time is not worth it. It can be better overall just to busy wait a short period.

We can estimate the time that busy waiting uses: Suppose a CPU can perform $m$ cycles per second and an I/O device can operate at $n$ transfers per second. If each I/O transfer takes $c$ cycles to carry out, then the percentage of available CPU cycles wasted by waiting for the device is:

$$w = \frac{100(m - n.c)}{m}$$

For example, if $m = 100,000,000$ and $n = 1500$ and $c = 200$ (a VDU might be roughly in this range), then

$$w = 99.7\%$$

## 7.8 Interrupts

We wish to avoid having to wait until a device status flag indicates that a device is ready to use. This time could be spent on useful processing within the CPU.

To achieve this, an *interrupt* system can be employed.

The interrupt system is arranged so that a device can request that CPU processing be interrupted only when the device is ready to perform another transfer. No time is wasted waiting for the device.

This reverses who initiates the I/O action: when a device is ready it tells the processor. This means the processor can continues processing useful work until it is interrupted by a device. Whatever background process the CPU was busy computing is then suspended while the I/O device is *serviced*. Once the I/O device has been serviced, the CPU's original activity is resumed at the point it left off.

The device already has several control lines connecting it (via a bus) to the processor. One more control line is the interrupt. This is normally set at (say) 0. When the device wants to interrupt the processor it sets the line to 1. This electrical signal sets off hardware in the processor to start servicing the interrupt.

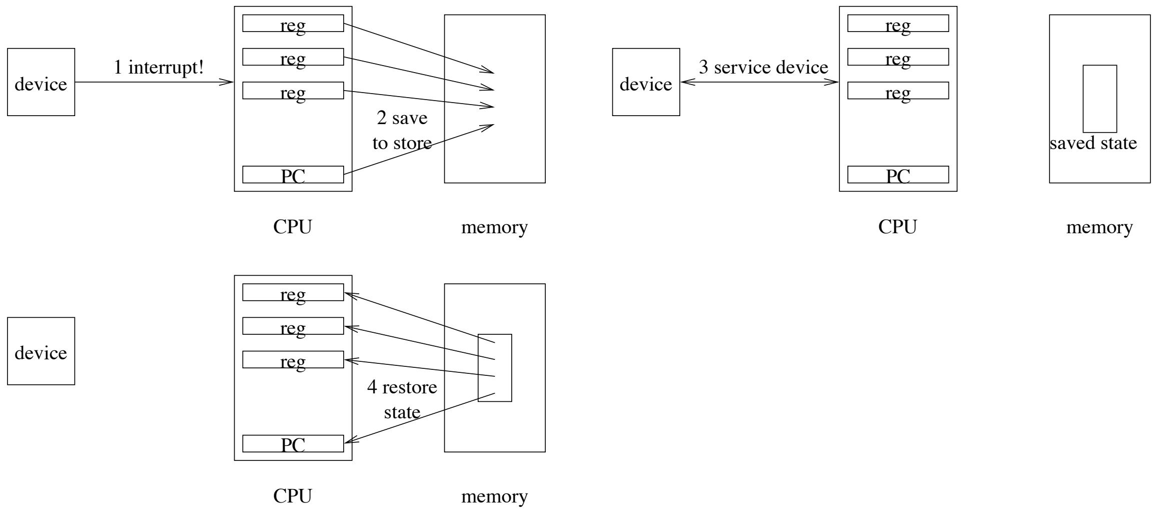


Figure 32: Servicing an Interrupt

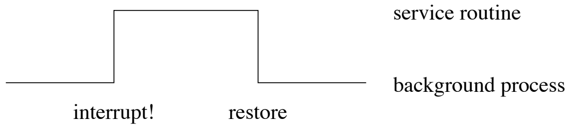
## 7.9 Processing an Interrupt

The (simplified outline) mechanism for processing an interrupt is as follows:

1. A device that is ready to handle an I/O transfer and needs attending to, i.e. has its device status flag raised, sends an *interrupt request* to the CPU.
2. Processing and execution of the current instruction is completed and if the interrupt request is accepted by the CPU (it may not in practice always be), the device sending the interrupt request is identified.
3. The address of the next instruction from the background process about to be executed (the contents of the PC) is saved (call it location X), plus the current state of the computation, which usually means saving all the processor's registers. All this data are saved in some convenient area of memory.
4. The contents of the PC are replaced so that the next instruction is taken from the first location, call it Y, of a sequence of locations that contains a procedure for servicing the interrupting I/O device. The replacement addresses (like Y, for example) are usually obtained from specific store locations that are associated with each device attached to the computer (an interrupt vector).
5. The last instruction of the device handling routine restores the saved address X to the PC and similarly the rest of the state of the processor (i.e., the registers).
6. The next instruction after that is therefore taken from the next instruction that would have been executed from the background program had it not been interrupted.

As we have restores the processor back to the identical state it was in before the interrupt it can proceed in its computation as if nothing had happened. In this case we call the interrupt *transparent*.

Graphically we might see this:



The *background process* is the main task the processor is working on. At the interrupt, the processor pauses the background process and start the interrupt routine. When done, the processor returns to the background process.

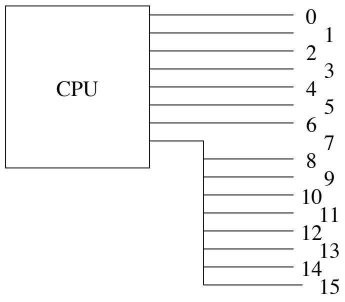
When an interrupt happens, how do we know which device needs servicing? There are two main ways:

1. when interrupted, the processor goes through all the attached devices and checks their status flags. This is easy to design for, but has the disadvantage that it takes a little time to check all the device flags and this time lag might be important. Also, if two devices cause interrupts in close succession, device B then device A, say, , the processor might service them in order A then B simply because it found A first. This too, might be important. Further complication arise when we introduce *priorities*, see later.
2. each device has it own dedicated control line connecting it to the processor. Thus the processor knows immediately which device requires servicing.

The PC combines both the above: it has 16 control lines, but some of the lines are also shared between devices.

---

The reality is that they have 8 control lines, but the last line is connected to a secondary controller with 8 more lines. This is for historical reasons.



---

## 7.10 Multiple Interrupts

When there are several devices, any one can in principle request an interrupt, regardless of the activity of any of the others. However:

- we should treat very fast devices (e.g., discs) as being more important than slower devices (e.g., keyboards), which can be delayed longer before servicing without losing data.
- Devices whose data is critical (e.g. clocks and ALU event detectors like arithmetic failure checks) may also need to be treated as more important, since their data can in no circumstances be lost.

So it is sensible to identify and service the fastest or most critical devices first and pend interrupt requests from slower or less critical devices.

This is inadequate, for example, if a device is already being serviced and no other device can be dealt with while that is happening.

A faster or more critical device which then needed attention could be delayed until the current service routine had been completed and this delay might be just too long, so that the data of the faster device would be lost.

What is needed is a system that will

- allow a more important device to interrupt the servicing of a less important device, and
- prevent a less important device from interrupting the servicing of a more important device.

## 7.11 Priorities

Each I/O device is given a *priority* and when the device sends an interrupt request to the CPU, its priority is compared with the current operating priority of the CPU.

A priority is usually a small integer, 0, 1, 2, say. Curiously, often it is arranged that smaller values represent higher priorities. Thus 0 would be the most important event possible.

An interrupt request will not be accepted by the CPU unless the priority of the device issuing the interrupt request is higher than the current operating priority of the CPU. The background process also has a priority, often a low value.

In this case, the CPU operating priority will normally be set to that of the interrupting device upon entry to the service routine. This is so we can tell whether to interrupt the current service routine or not.

## 7.12 Servicing Multiple Interrupts

Suppose that the CPU has background priority a, device 1 has a priority b and device 2 has priority c, where

$$a < b < c$$

Here "<" means "less important".

If device 1 interrupts, then device 2 interrupts while device 1 is being serviced, the sequence of events will be, in order:

- Background program suspended;
- Service routine, device 1 entered, then suspended;
- Service routine, device 2 entered and completed;
- Service routine, device 1 resumed and completed;
- Background program resumed.

Now suppose that device 2 interrupts first, then device 1 interrupts while device 2 is being serviced, the sequence of events will be,

- Background program suspended;
- Service routine, device 2 entered

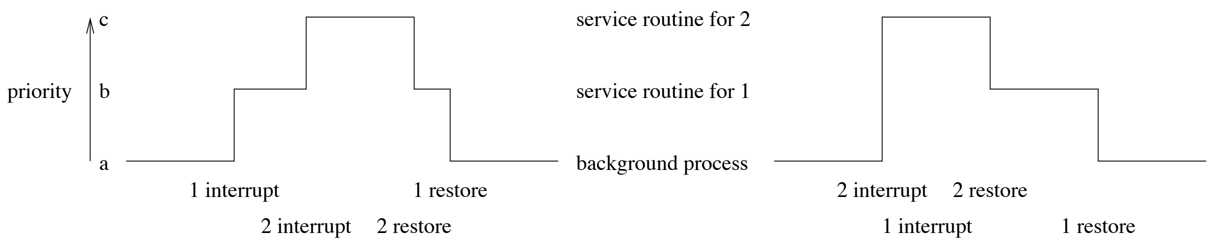


Figure 33: Multiple Interrupts

- device 1 interrupts, interrupt is recognised but *pended*
- Service routine, device 2 completed;
- Service routine, device 1 entered and completed;
- Background program resumed.

Another possibility, occasionally used, is that in the latter case, the interrupt for device 1 would be simply *ignored.*

## 7.13 Saving State

If we have *nested* interrupts, as above, we must have a means of saving and restoring the correct state each time we enter and leave a service routine.

We cannot simply save the processor state to a single well-known area of memory, as nested interrupt would cause the saved state of the previous interrupt to be overwritten with the new current state. Thus each successive interrupt needs to save its state in a *different* area of memory.

This is normally achieved using a *stack.*

A stack is a normal area of memory which we use in a particular way. Suppose, say, we need to save 10 bytes of state on each interrupt (very small, but this is an example). We have a register, the *stack pointer*, that contains the address of the start of our area of memory, location 1000, say.

When an interrupt happens we

1. We save the state of the processor in the area indicated by the SP, and then add 10 the SP.
2. Process the interrupt.
3. We subtract 10 from the SP. and then restore state from the area indicated by the SP.

The beauty of this is that this handles nested interrupts:

1. An interrupt. We save the state of the processor in the area indicated by the SP (Xs in the figure), and then add 10 the SP.
2. Start processing the interrupt.
3. Another interrupt. We save the state of the processor in the area indicated by the SP (Ys in the figure), and then add 10 the SP.

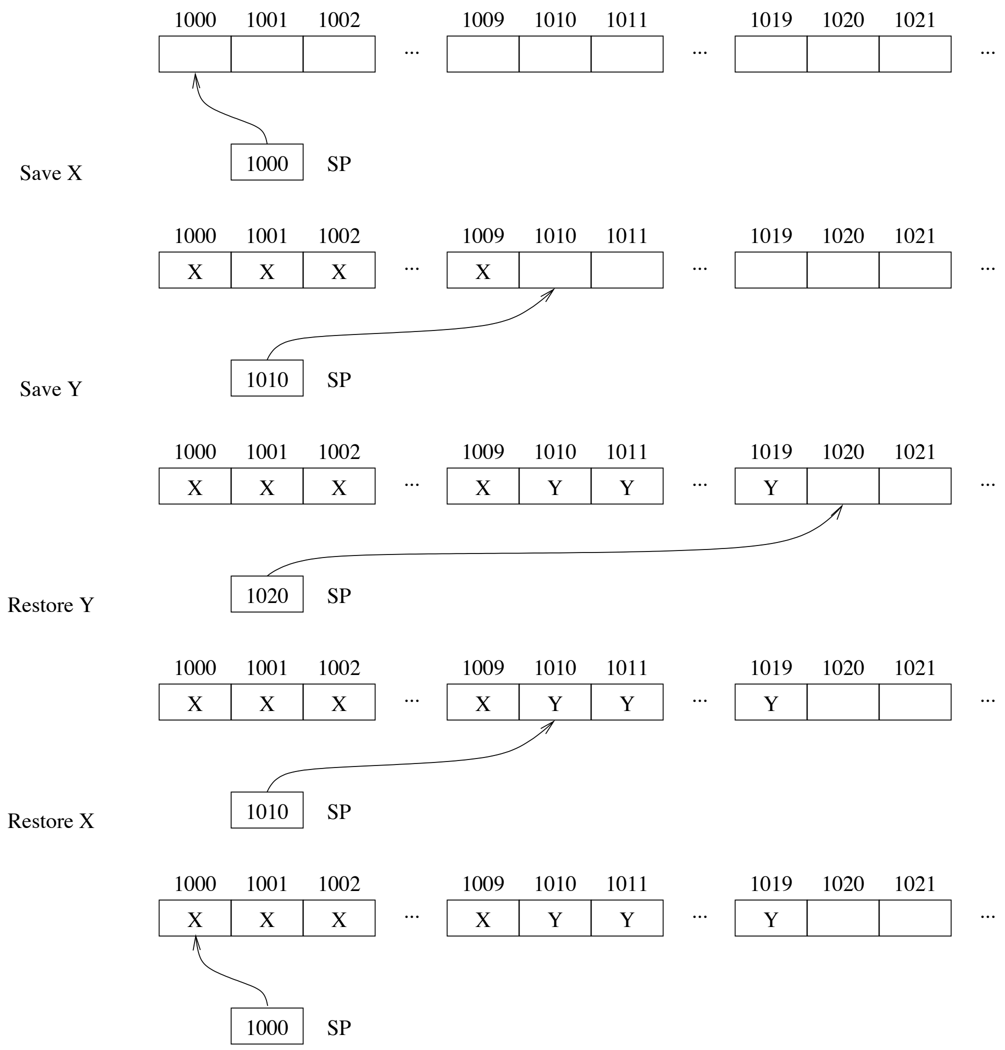


Figure 34: Saving State on a Stack

4. Process the interrupt.
5. We subtract 10 from the SP, and then restore state from the area indicated by the SP (Ys).
6. Finish processing the first interrupt.
7. We subtract 10 from the SP. and then restore state from the area indicated by the SP (Xs).

The saved state remains in memory, but we don't care as we don't need it anymore and can overwrite it with the state from a later interrupt.

This technique, the stack, nests our saved states exactly in the way we need, and can handle any number (up to memory constraints) of nested interrupts happening in any order. Every time we have an interrupt, the SP points to where in memory the state should be saved. Every time we finish an interrupt, subtract 10 from the SP and this now tells us where in memory the old state can be found.

In the 68000 an address register, `A7`, is used as a stack pointer.

## 7.14 Summary of Interrupts

Interrupts are a good way to ensure the computer spends most of its time doing useful computation rather than waiting for relatively slow devices.

The cost of an interrupt is fairly high in terms of

- the hardware needed to support a fast interrupt mechanism, and
- the overhead in time servicing an interrupt takes, as it must save state, determine which device needs servicing, and restore state at the end.

On the other hand, interrupts happen relatively infrequently, so that the overhead cost is easily much smaller than polling.

An additional of an interrupt-driven system is that programming is much easier than when we have polling: making sure that multiple devices are serviced in a timely fashion and making sure each device gets serviced according to its priority is *extremely* difficult if you have to poll more than a couple of devices. With interrupts, multiple devices are easy.

## 7.15 Summary of I/O

- Programmed I/O: simple and cheap, but works best for slow devices.
- Interrupt driven I/O: more complex hardware, but better in terms of programming complexity and program throughput
- Direct memory access/data channel: more complex and expensive, but good for fast devices.

Note these are *not* mutually exclusive and a single machine can have combinations of any of them.

# CM100194: Systems Architecture I
# Lecture Notes, part II

Jim Laird

Semester I: 2012/13

## Contents

## 1 Boolean Logic and Circuits

We may use circuits constructed from logic gates to implement logic (e.g. conditionals), arithmetic, data storage and control cycles. Our main tool for analysing the behaviour of these circuits is Boolean logic, which allow us to represent their input/output behaviour as Boolean formulas (representing the output state) of Boolean variables (representing the input state). The logical operations on which Boolean algebra is based correspond to logic gates. They have operands and produce results whose values can be only two-fold, namely: TRUE and FALSE. These are called *Boolean* values and the operations on them are *Boolean* operations.

There are three fundamental logical operations, though they are not actually independent. A fourth operation is also commonly used.

- Logical AND:

  $$A.B$$

  Here $A$ and $B$ are Booleans, namely TRUE and FALSE, *not* numbers.

  We write AND using the multiplication operator, but it is important to realise this is *not* arithmetic multiplication of A and B. We might also see

  $$A \wedge B \quad A\&B \quad A \cap B \quad AB$$

  for AND.

  The result is TRUE if both A and B are TRUE; otherwise the result is FALSE.

- Logical (inclusive) OR:
  $$A + B$$
  This is *not* addition, we are just using the plus sign in a different way. We might also see
  $$A \vee B \quad A|B \quad A \cup B$$
  for OR.

  The result is TRUE if either A or B are TRUE; otherwise the result is FALSE.

- Logical NOT:
  $$A'$$
  We also find
  $$\sim A \quad \neg A \quad -A \quad \overline{A} \quad !A$$
  The result is TRUE if A is FALSE; the result is FALSE if A is TRUE.

- Logical Exclusive OR (EOR or XOR):
  $$A \text{ XOR } B$$
  We might also see
  $$A \oplus B \quad A \wedge B$$
  for XOR.

  The result is TRUE if A is TRUE and B is FALSE; the result is TRUE if A is false and B is TRUE; otherwise the result is FALSE.

  $A$ XOR $B$ is TRUE when $A$ and $B$ are *different*.

It is conventional and convenient to represent TRUE by the bit 1 and FALSE by the bit 0. **These are not numbers.** We can now write

| $A$ | 0 | 0 | 1 | 1 |
|---|---|---|---|---|
| $B$ | 0 | 1 | 0 | 1 |
| $A.B$ | 0 | 0 | 0 | 1 |
| $A + B$ | 0 | 1 | 1 | 1 |
| $A$ XOR $B$ | 0 | 1 | 1 | 0 |

| $A$ | 0 | 1 |
|---|---|---|
| $A'$ | 1 | 0 |

These are called *Truth Tables.*

These are also commonly written as

| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| OR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

So we have, for example, $1.1 = 1$, $1.0 = 0$, $1 + 0 = 1$, $1 + 1 = 1$. **This is not arithmetic.**

## 1.1 Boolean Algebra

Combinatorial logic circuits (i.e. without feedback) correspond directly to formulas of *Boolean algebra*. An *algebra* is nothing complicated, just some class of data plus some operations on them. Thus arithmetic is an algebra on numbers. We are interested in an algebra on Boolean values, namely TRUE and FALSE. Our operations are AND, NOT, OR and so on.

What are the properties of this algebra?

We have already seen the truth tables

| Operands | | AND | OR |
|---|---|---|---|
| $A$ | $B$ | $A.B$ | $A+B$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| Operand | NOT |
|---|---|
| $A$ | $A'$ |
| 0 | 1 |
| 1 | 0 |

Thus we can see that $A + B = B + A$, $A + 0 = A$ and so on.

| | AND rule | OR rule | NOT rule |
|---|---|---|---|
| zero | $A.0 = 0$ | $A + 0 = A$ | $0' = 1$ |
| unit | $A.1 = A$ | $A + 1 = 1$ | $1' = 0$ |
| idempotent | $A.A = A$ | $A + A = A$ | $(A')' = A$ |
| inverse | $A.A' = 0$ | $A + A' = 1$ | |
| commutative | $A.B = B.A$ | $A + B = B + A$ | |
| associative | $A.(B.C) = (A.B).C$ | $A + (B + C) = (A + B) + C$ | |
| distributive | $A.(B + C) = A.B + A.C$ | $A + (B.C) = (A + B).(A + C)$ | |
| DeMorgan | $(A.B)' = A' + B'$ | $(A + B)' = A'.B'$ | |

The way we prove these assertions is to look at the truth tables. For example, $A + 1 = 1$.

| $A$ | $B$ | $A+B$ | |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | 1 | 1 | $\leftarrow$ |
| 1 | 0 | 1 | |
| 1 | 1 | 1 | $\leftarrow$ |

in the indicated rows we have $A + 1$ and the result column is 1 for all values of $A$. Thus $A + 1 = 1$.

Another example, $A + A = A$.

| $A$ | $B$ | $A+B$ | |
|---|---|---|---|
| 0 | 0 | 0 | $\leftarrow$ |
| 0 | 1 | 1 | |
| 1 | 0 | 1 | |
| 1 | 1 | 1 | $\leftarrow$ |

in the indicated rows we have $A = B$ and the result column is the same as $A$ for all values of $A$. Thus $A + A = A$.

---

This is a proof by enumeration of cases, which is quite common for this kind of Boolean algebra.

---

Notice that $(A'.B')' = (A')' + (B')' = A + B$ by application of DeMorgan and idempotent. Thus we can define OR in terms of AND and NOT (or vice versa!).

It is very important to realise that $A$, $B$ and $C$ in the above rules stand for general formulae, so, for example, the unit rule says that $1 + A.B.C' = 1$ and the zero rule says that $0.A.B.(B + C + 1) = 0$ amongst an infinite number of other things.

## 1.2 Proving Boolean Identities by Algebra

Now we know these basic identities, we can use them to prove things. For example, to show that $A.(A+B) = A$ we can apply the Boolean equalities:

$$\begin{array}{rcll} A.(A+B) & = & A.A + A.B & \text{distributive} \\ & = & A + A.B & \text{idempotent} \\ & = & A.1 + A.B & \text{unit} \\ & = & A.(1+B) & \text{distributive} \\ & = & A.1 & \text{unit} \\ & = & A & \text{unit} \end{array}$$

Notice that we **do not have cancellation rules**. So $A + 1 = B + 1$ **does not imply** $A = B$. For example, $0 + 1 = 1 + 1$ but $0 \neq 1$.

## 1.3 Proving Boolean Identities by Truth Tables

The alternative to algebra is to use truth tables:

| $A$ | $B$ | $A + B$ | $A.(A + B)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |

We see that the column for $A.(A + B)$ is identical to the column for $A$. Thus, $A.(A + B) = A$.

Another algebra proof:

$$\begin{array}{rcll} A + A.B & = & A.1 + A.B & \text{unit} \\ & = & A.(1+B) & \text{distributive} \\ & = & A.1 & \text{unit} \\ & = & A & \text{unit} \end{array}$$

So we have proved $A + A.B = A$. Proof by truth table is also easy.

A bigger proof:

$$\begin{array}{rcl} (A + B' + C).(A + B) & = & A.(A+B) + B'.(A+B) + C.(A+B) \\ & = & A.A + A.B + B'.A + B'.B + C.A + C.B \\ & = & A + A.B + A.B' + A.C + B.B' + B.C \\ & = & A.(1 + B + B' + C) + 0 + B.C \\ & = & A.1 + B.C \\ & = & A + B.C \end{array}$$

We are now led to ask whether there is a way to show that this is indeed the *simplest* form of the original Boolean function $(A + B' + C).(A + B)$.

To do this we need to develop the notion of a *standard form* of expression of a Boolean function.

## 1.4 Standard Forms

Given a function of $n$ Boolean variables, it is always possible to reduce the function to a form in which there is series of terms joined by OR operations and in which each of the terms contains a combination of *all* $n$ variables (or their complements) joined by AND operations.

Let us take an example and put it into this form:

$$\begin{aligned}
A + B.C &= A.(B + B') + B.C \\
&= A.B + A.B' + B.C \\
&= A.B.(C + C') + A.B' + B.C \\
&= A.B.C + A.B.C' + A.B' + B.C \\
&= A.B.C + A.B.C' + A.B'.(C + C') + B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C.(A + A') \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + A.B.C + A'.B.C \\
&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C
\end{aligned}$$

This is a standard form. It is known somewhat misleadingly as the *standard sum of products* form, even though we know they are *not* sums and products. It is also called *disjunctive normal form*, or DNF.

There is also a *standard product of sums* form, where we have a series of terms joined by AND operations and in which each of the terms contains a combination of *all* $n$ variables (or their complements) joined by OR operations. This is also called *conjunctive normal form*, or CNF.

The above example becomes

$$A + B.C = (A + B + C).(A + B + C').(A + B' + C)$$

as a standard product of sums.

Exercise: show this.

The SofP form is generally used in preference to the PofS forms purely because they are psychologically easier to compute given our background in the rules of arithmetic. There is nothing to separate them mathematically.

## 1.5 Standard Forms and Truth Tables

There is an easy way to compute standard forms using truth tables (for small numbers of variables). For example

| $A$ | $B$ | $C$ | $B.C$ | $A+B.C$ | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | | $A+B+C$ |
| 0 | 0 | 1 | 0 | 0 | | $A+B+C'$ |
| 0 | 1 | 0 | 0 | 0 | | $A+B'+C$ |
| 0 | 1 | 1 | 1 | 1 | $A'.B.C$ | |
| 1 | 0 | 0 | 0 | 1 | $A.B'.C'$ | |
| 1 | 0 | 1 | 0 | 1 | $A.B'.C$ | |
| 1 | 1 | 0 | 0 | 1 | $A.B.C'$ | |
| 1 | 1 | 1 | 1 | 1 | $A.B.C$ | |

In the truth table for $A+B.C$ we look at each row that has a 1 for $A+B.C$. Write down $A.B.C$ and put a prime against each variable that has a 0 in its column. Thus 011 becomes $A'.B.C$. Join all these terms with OR. Therefore we get

$$A'.B.C + A.B'.C' + A.B'.C + A.B.C' + A.B.C,$$

as before.

It is just as easy to get the PofS form: we look at each row that has a 0 for $A+B.C$. Write down $A+B+C$ and put a prime against each variable that has a 1 in its column. Thus 001 becomes $A+B+C'$. Join all these terms with AND. This time we get

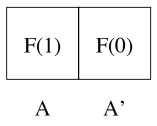
$$(A+B+C).(A+B+C').(A+B'+C)$$

for the PofS form.

## 1.6 Karnaugh Maps

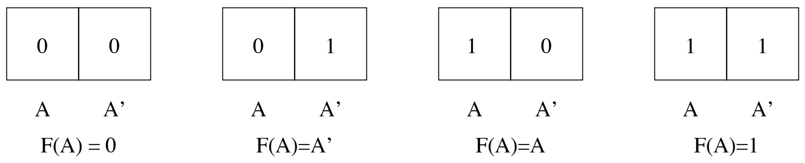
So far we have only made things more complicated, but this is just the first step on the path to the simplest form.

Given a function, $F$, of one Boolean variable, $A$, we can plot the function $F(A)$, whatever it is, on a map with two regions:



We put the value of $F(1)$ (be it 0 or 1) in the box marked $A$, and the value of $F(0)$ in the box marked $A'$.

Here are the four possible function of one variable:



These are the four possible functions in truth table format:

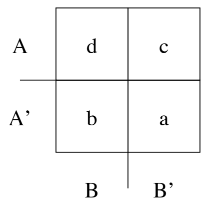
| $A$ | $F(A)$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

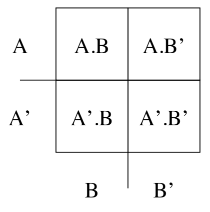
| $A$ | $F(A)$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $A$ | $F(A)$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $A$ | $F(A)$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

We can extend this to two variables:





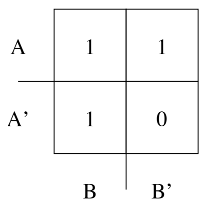
corresponding to the function

| $A$ | $B$ | $F(A,B)$ |
|---|---|---|
| 0 | 0 | $a$ |
| 0 | 1 | $b$ |
| 1 | 0 | $c$ |
| 1 | 1 | $d$ |

So, for example, the function

| $A$ | $B$ | $F(A,B)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

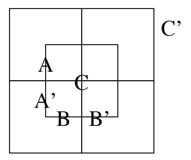
is mapped as



These are called *Karnaugh Maps*, after Maurice Karnaugh who popularised them in the early 1950s.

---

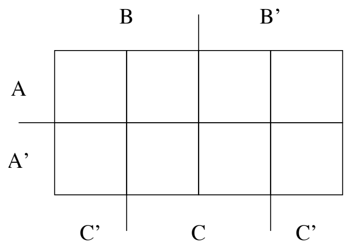
Also known as *Carroll Diagrams*, after Lewis Carroll, who was interested in using these diagrams to solve problems in logic. His form of drawing the diagrams was slightly different, but equivalent.



inner square C
outer square C'

---

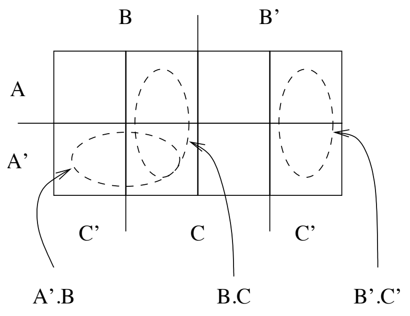
For three variables we get:

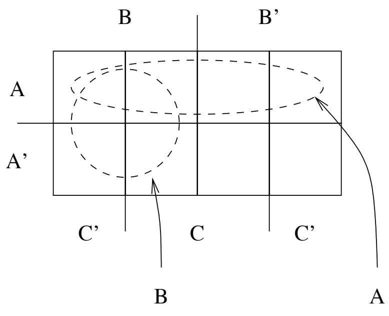


We should view this as a $2 \times 2 \times 2$ diagram unrolled flat.

The point of note is that *each cell in the diagram corresponds to one term in the standard sum of products form*. (And to the PofS form.) The cell marked by $A$, $B$, $C'$ corresponds to the term $A.B.C'$. (And $(A' + B' + C)$.)

Moreover, pairs and quads of cells correspond to other terms: the OR of the cells.



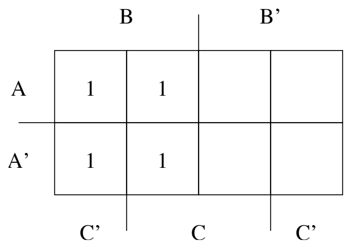


The whole of the diagram corresponds to 1, while the empty diagram corresponds to 0. The larger the are, the smaller the term.

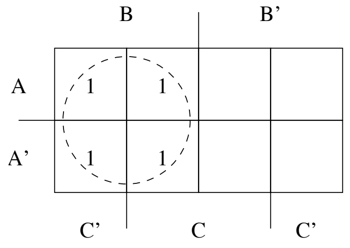
It is easy to find the term associated with an area. If the area lies completely within the region defined by a label (such as $A$ or $A'$), write down that label. If the area covers a label and its complement (such as both $A$ and $A'$) do not include it.

So an area covering $A'.B.C'$ and $A'.B.C$ corresponds to the term $A'.B$. This, of course, is just using pictures to compute $A'.B.C' + A'.B.C = A'.B.(C' + C) = A'.B.1 = A'.B$.

So suppose we have a function $F = A.B.C + A'.B.C + A.B.C' + A'.B.C'$ in standard form. We plot on a Karnaugh map:



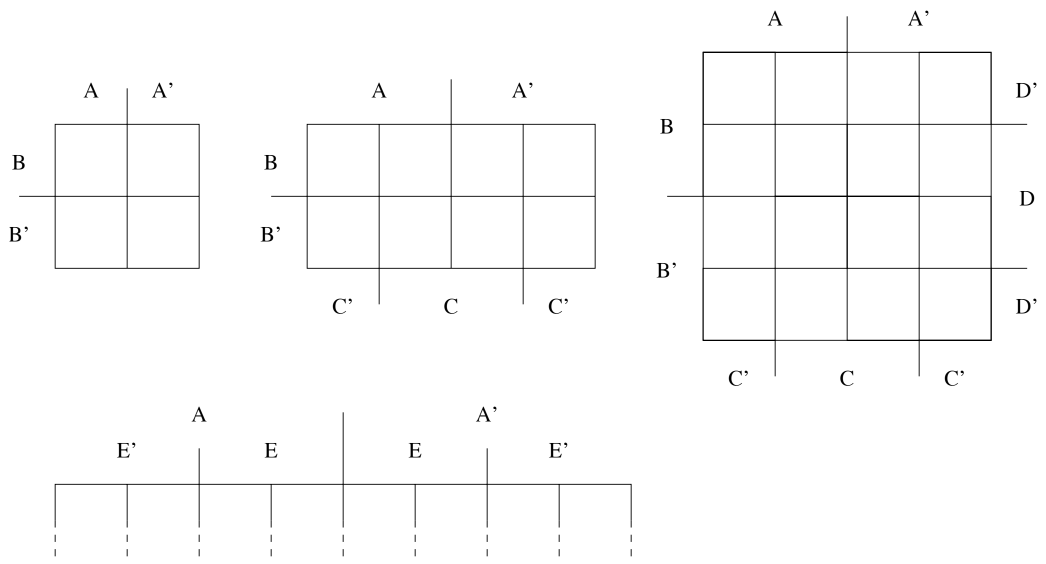
(the convention is not to bother writing in the 0s). Next, draw a region about the 1s:



and then determine what term this corresponds to. In this case it is just $B$. Therefore we have shown that $F = A.B.C + A'.B.C + A.B.C' + A'.B.C' = B$.

Karnaugh maps are our way of finding simplest forms.

The maps extend to more variables and more:



## 1.7 Simplifying a Boolean Function

Thus we can

1. Write it in the form of a standard sum of products;
2. Plot the product terms on a Karnaugh map;
3. Find the *smallest* number of the *biggest* valid groupings that will encompass all of the marked regions of the Karnaugh map; and
4. Read off the function that results.

To clarify: a *grouping* is a rectangle containing a power of 2 cells. Overlapping rectangles is fine, and we need to make them as big as possible (remember the bigger the area the smaller the term it represents).

Also, the rectangles might wrap around the edge of the diagram in the case of three or more variables.

lsls

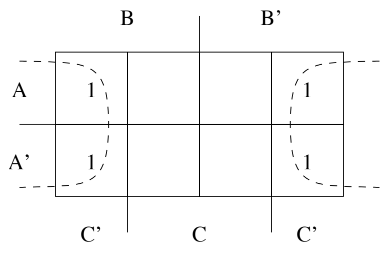
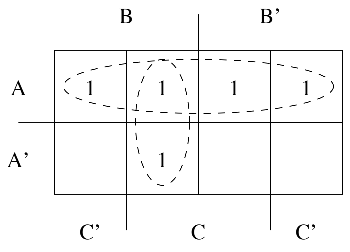


Figure 1: Wrapping in Karnaugh Maps

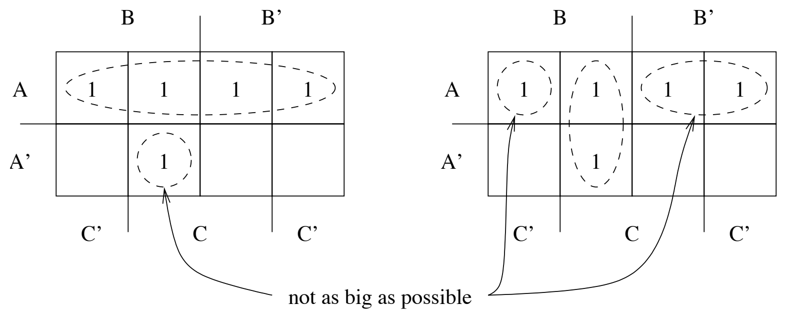
We can try this out on our function $F = A + B.C = A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C$. The map is



The two areas correspond to $A$ (upper area) and $B.C$ (other area). Thus, the simplest form for $F$ is

$$F = A + B.C$$

Note that other putative groupings



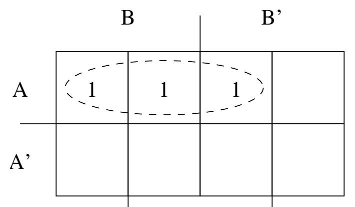
are not valid as they are not as large as possible.

---

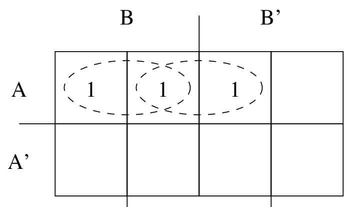
Of course, they *do* correspond to valid formulae for $F$, but not *simplest* formulae. The first is $A + A'.B.C$, the second $A.B.C' + B.C + A.B'$. Both simplify to $A + B.C$

---

In this example:



Incorrect



Correct

the grouping is not a power of 2 elements. It must be a power of 2 to be able to determine the corresponding SofP term.

**Exercise:** Go through all 16 functions of two variables $A$ and $B$ and plot their Karnaugh maps. Then find their simplest forms.

---

| $A$ | $B$ | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | 0 | $A.B$ | $A.B'$ | $A$ | $A'.B$ | $B$ | $A.B'+$ $A'.B$ | $A+B$ |

| $A$ | $B$ | $F_8$ | $F_9$ | $F_{10}$ | $F_{11}$ | $F_{12}$ | $F_{13}$ | $F_{14}$ | $F_{15}$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | $A'.B'$ | $A.B+$ $A'.B'$ | $B'$ | $A+B'$ | $A'$ | $A'+B$ | $A'+B'$ | 1 |

---

We should, however, note that the simplest form of a function yielded by this method may not in fact be the most efficient in terms of the total number of logic gates needed to implement it. For example, the form $M = A.B + A.C$ which requires two AND gates and one OR gate, can be factorised into $M = A.(B + C)$ which needs only one AND gate and one OR gate.

The Karnaugh maps approach will find the simplest form *as a sum of products*.

Everything follows through for the product of sums form and similarly allows us to determine the simplest PofS form for a given formula. Sometimes it will be simpler (in count of logic gates) than the SofP form, sometimes not. In the case of $M = A.B+A.C$, the SofP simplest form is $A.B+A.C$, while the PofS simplest form is $A.(B+C)$.

## 1.8 Can't Happen and Don't Care Terms

*Can't happen* (CH) terms: some combinations of values of the input variables in Boolean functions may not occur in certain problems.

*Don't care* (DC) terms: the output of a Boolean function may be immaterial for certain combinations of the values of the input variables.

By marking these cases with an "X" in Karnaugh maps, we can choose to *use* or *ignore* them when simplifying a Boolean function.

They act like blank tiles in Scrabble: we can choose to think of them as either 1 or 0 as we wish. If they can't happen, we don't care what outputs they might give.

## 1.9 Simplifying Functions with "Can't Happen" Terms

Let us take a problem in which Boolean variables $A$, $B$, $C$ and $D$ are used to represent the digits of a decimal counter as shown below:

$$A'B'C'D' = 0000; \quad A'B'C'D = 0001; \quad A'B'CD' = 0010$$

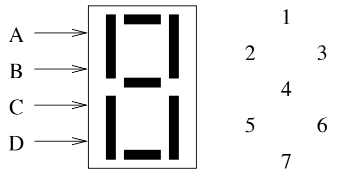
and so on, up to:

$$A'BCD = 0111; \quad AB'C'D' = 1000; \quad AB'C'D = 1001$$

The patterns 1010, 1011, 1100, 1101, 1110 and 1111 will not occur, as they are not patterns for decimal digits; they are the *can't happen* inputs in this function.

This encoding for the numbers 0-9 is called *Binary Coded Decimal*, or BCD. The number 42 would be represented by the eight bits 01000010. This encoding is used occasionally for simplicity, for example the track numbers on a CD are encoded in 8-bit BCD. This means the maximum number of tracks on a CD is 99.

We could use these four inputs to control a decimal LCD display:



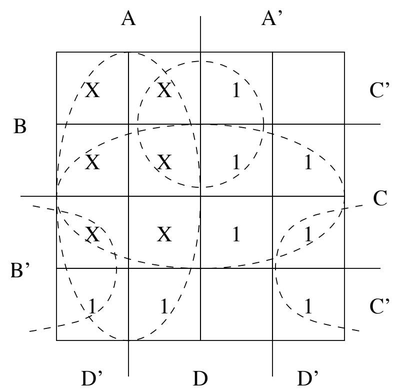
Each combination of inputs would light up a selection of the LED segments. We can regard this as seven different functions of the four input bits. So segment number 1 need to be on for inputs corresponding to decimal values 0, 2, 3, 5, 6, 7, 8 and 9. The truth table for segment 1 is

|  | $A$ | $B$ | $C$ | $D$ | on? |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 |

|  | $A$ | $B$ | $C$ | $D$ | on? |
|---|---|---|---|---|---|
| 8 | 1 | 0 | 0 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | CH |
| 11 | 1 | 0 | 1 | 1 | CH |
| 12 | 1 | 1 | 0 | 0 | CH |
| 13 | 1 | 1 | 0 | 1 | CH |
| 14 | 1 | 1 | 1 | 0 | CH |
| 15 | 1 | 1 | 1 | 1 | CH |

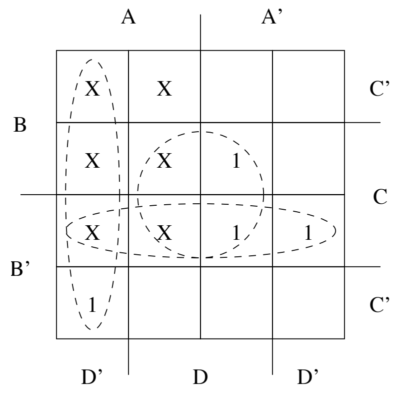
Similarly for the other six segment functions. So what is the simplest function that lights segment 1 for just the correct combination of inputs?

Here is the Karnaugh map



This corresponds to $A + C + B.D + B'.D'$.

In this next example we don't use all the Xs: construct the simplest Boolean function that will give an output if a decimal digit 2, 3, 7 or 8 occurs (but will give no output for any other decimal digit).



This corresponds to $A.D' + B'.C + C.D$.

## 1.10 Simplifying Functions with "Don't Care" Terms

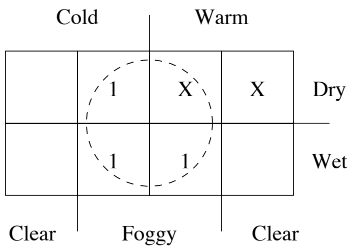
The nationalised electricity generation industry in the small and unimportant nation of Dementia has been ordered by the government to have its generation plant ready to run if the weather is

- both cold and foggy; or
- both foggy and wet.

The government has, however, told the industry that, as a concession, the state of readiness of the generation plant is immaterial if the weather is both warm and dry.

What is the simplest way to determine the conditions under which the plant should the plant be ready?

The map:



We see that, taking the "Don't Care" terms into account, the conditions under which the generation plant must be ready to run is, quite simply, when the weather is foggy!

## 1.11 Implementing Arithmetic

We shall now consider how we may design combinatorial logic circuits for implementing arithmetical operations on (binary representations of) signed and unsigned integers. First, we consider the technicalities of binary arithmetic in a little more detail. We observed that the results of arithmetic, like addition, with signed integers could lead to overflow. For example, in 6 bits, 2's complement:

$$\begin{array}{cc} 010000 & 16 \\ \underline{011010} & \underline{26} \\ 101010 & -22 \end{array}$$

Adding two positive integers, as in this example, leads to an apparently negative result.

We need to be able to detect this kind of *overflow*.

Overflow can be detected if, within the ALU we add an additional, hidden bit, the *guard bit*, to the most significant end of the integer representations. This lengthens the internal representation by extending their most significant bits to the left. Arithmetic is now performed internally, not on $n$-bit operands, but on $n+1$ bit operands. Using the same example:

$$\begin{array}{c|c} 0 & 010000 \\ 0 & 011010 \\ \hline 0 & 101010 \end{array}$$

Note that the hidden bit (0) in the result is different from the most significant bit (1) of the 6-bit signed integer representation. This indicates *arithmetic overflow.*

There is often a condition code flag, the *overflow flag*, that is set when the guard bit disagrees with the top bit. So by testing the this flag we can tell if there was an overflow.

Some examples of addition with the guard bit, using 6-bit 2's complement representations:

$$\begin{array}{cc} 0 & 010111 \\ 0 & 001000 \\ \hline 0 & 011111 \end{array} \quad \begin{array}{c} 23 \\ 8 \\ \hline 31 \end{array} \qquad\qquad \begin{array}{cc} 0 & 011011 \\ 0 & 010000 \\ \hline 0 & 101011 \end{array} \quad \begin{array}{c} 27 \\ 16 \\ \hline \textit{ov} \end{array}$$

$$\begin{array}{cc} 0 & 011011 \\ 1 & 100001 \\ \hline 1 & 111100 \end{array} \quad \begin{array}{c} 27 \\ -31 \\ \hline -4 \end{array} \qquad\qquad \begin{array}{cc} 0 & 010001 \\ 1 & 100000 \\ \hline 1 & 110001 \end{array} \quad \begin{array}{c} 17 \\ -32 \\ \hline -15 \end{array}$$

$$\begin{array}{cc} 1 & 101111 \\ 1 & 101111 \\ \hline 1 & 011110 \end{array} \quad \begin{array}{c} -17 \\ -17 \\ \hline \textit{ov} \end{array} \qquad\qquad \begin{array}{cc} 1 & 101111 \\ 1 & 111101 \\ \hline 1 & 101100 \end{array} \quad \begin{array}{c} -17 \\ -3 \\ \hline -20 \end{array}$$

## 1.12 Multiplication of Positive Integers

We can multiply by adding repeatedly. Thus binary 000110 $\times$ 000011 (decimal $6 \times 3$) is just:

$$\begin{array}{c} 000110 \\ 000110 \\ 000110 \\ \hline 010010 \end{array} \quad \begin{array}{c} 6 \\ 6 \\ 6 \\ \hline 18 \end{array}$$

However, multiplication can also be performed by the familiar tabular method. Consider the decimal example $324 \times 216$:

$$\begin{array}{r} 324 \\ 216 \\ \hline 1944 \\ 3240 \\ 64800 \\ \hline 69984 \end{array}$$

In effect we can are doing this:

$$(324 \times 200) + (324 \times 10) + (324 \times 6)$$

namely

$$(324 \times 2 \times 10^2) + (324 \times 1 \times 10^1) + (324 \times 6 \times 10^0)$$

So we are doing multiplications by a single digit, then multiplying by powers of 10.

And we can do the same in binary, by noticing that

- multiplying by a single digit is easy as the only digits are 0 and 1
- multiplying by integral powers of 2 is simply a case of *shifting a bit pattern to the left.*

Example.

| | |
|---|---|
| $14 \times 1$ | $1110 \times 1 = 1110$ |
| $14 \times 2$ | $1110 \times 10 = 11100$ |
| $14 \times 4$ | $1110 \times 100 = 111000$ |
| $14 \times 8$ | $1110 \times 1000 = 1110000$ |
| $14 \times 16$ | $1110 \times 10000 = 11100000$ |
| $14 \times 32$ | $1110 \times 100000 = 111000000$ |

To multiply by $2^n$ we shift left by $n$ bits.

This means the tabular method is extra-easy: $010111 \times 001011$ (decimal $23 \times 11$) becomes

$$\begin{array}{rcr r}
010111 & \times & 1000 & 10111000 \\
010111 & \times & 10 & 101110 \\
010111 & \times & 1 & 10111 \\
\cline{4-4}
 & & & 11111101
\end{array}$$

For each 0 we write down nothing; for each 1 we write down the number shifted left by a suitable number of places; then we add. Binary multiplication is no more than a few shifts and a few adds!

Notice that, in general, if we multiply 2 $n$-bit integers, we could produce a result of up to $2n$ bits in length. In this case, 11111101 is equivalent to decimal 253, which is the correct result.

## 1.13 Summary of Shift Operations

Shifts can move bit patterns both left and right, by one or more places. They also come in various forms, including:

- Logical shifts: bits moved out are lost and zeroes fill the vacated positions.

  A left shift

  $$11100101 \xrightarrow{\text{left 3}} 00101000$$

  A right shift

  $$11100101 \xrightarrow{\text{right 3}} 00011100$$

- Arithmetic right shifts: are like logical right shifts, except that *copies of the sign bit are copied into the vacated bit positions on the left.* A right shift

  $$01100101 \xrightarrow{\text{right 3}} 00001100$$

  The 0 top bit is copied. Another

  $$11100101 \xrightarrow{\text{right 3}} 11111100$$

  The 1 top bit is copied.

An arithmetic left shift is the same as a logical left shift.

More interesting is the effect these shifts have when we regard the bits as integers:

- a logical left shift is a multiplication of an *unsigned* value by a power of 2

- a logical right shift is a division of an *unsigned* value by a power of 2
- an arithmetic left shift is the same as a logical left shift
- an arithmetic right shift is a division of an *signed 2s complement* value by a power of 2

Divisions are *integer divisions*, namely the quotient discarding the remainder. Multiplications are subject to overflow very often.

Examples.

$$\begin{array}{lclcl}
00000101 \xrightarrow{\text{left } 3} 00101000 & & 5 \times 2^3 & = & 40 \\
11100101 \xrightarrow{\text{right } 3} 00011100 & & 229 \div 2^3 & = & 28 \text{ (rem 5)} \\
11100101 \xrightarrow{\text{right } 3} 11111100 & & -27 \div 2^3 & = & -4 \text{ (rem 5)}
\end{array}$$

There is also

- *Circular shifts* or *rotates*: bits moved out of one end of a register are moved in at the opposite end of the register.

but these are not related to arithmetical operations.

A left rotate

$$11100101 \xrightarrow{\text{left } 3} 00101111$$

A right rotate

$$11100101 \xrightarrow{\text{right } 3} 10111100$$

**Exercise:** Show how to construct a rotate from a pair of logical shifts and an OR.

In summary: to do arithmetic we need (at least) shifts and (unsigned) add operations. Let us turn to the add operations.

## 1.14 Addition in Decimal and Binary

Decimal addition: sum digits from pairs of addends are obtained as follows ($c$ is a carry):

| Digits | 0 | 1 | 2 | ... | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | ... | 7 | 8 | 9 |
| 1 | 1 | 2 | 3 | ... | 8 | 9 | 0$c$ |
| 2 | 2 | 3 | 4 | ... | 9 | 0$c$ | 1$c$ |
| 3 | 3 | 4 | 5 | ... | 0$c$ | 1$c$ | 2$c$ |
| ⋮ | | | | ⋮ | | | |
| 7 | 7 | 8 | 9 | ... | 4$c$ | 5$c$ | 6$c$ |
| 8 | 8 | 9 | 0$c$ | ... | 5$c$ | 6$c$ | 7$c$ |
| 9 | 9 | 0$c$ | 1$c$ | ... | 6$c$ | 7$c$ | 8$c$ |

lsls

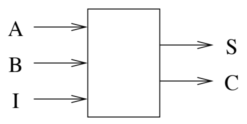


Figure 2: Half Adder

## 1.15 Binary Addition

The sum digit table for binary addition is very simple indeed, as we would expect from a system with only two digits (again, $c$ is a carry):

| Digits | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0$c$ |

In fact, both the decimal and binary tables of the previous slides are really twice as big, because when addition is performed, there are not simply two addend digits. There may also be a *carry-in* digit. For example, the binary table is thus:

| Carry-in | Digits | 0 | 1 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| | 1 | 1 | 0$c$ |
| 1 | 0 | 1 | 0$c$ |
| | 1 | 0$c$ | 1$c$ |

Our addition is actually a *three* input operation: two summands and a carry from the previous column. It also has *two* outputs: the sum and a carry to the next column.

## 1.16 Addition and Logic

We are now in a position to see the connection between arithmetic and logic.

At any general stage in the process of adding two binary numbers, there will be:

- Three inputs: the two addend digits, $A$ and $B$ and the carry-in, $I$; and
- Two outputs: the sum digit, $S$, and the carry out, $C$.

When the two least significant digits of the two binary numbers are added, the initial carry-in $I = 0$.

The full table is:

| I | A | B | S | C |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Compare the addition table and the XOR table (ignore the carry for now)

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| XOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Thus, $s$, the sum ignoring carry is

$$s = A \text{ XOR } B$$

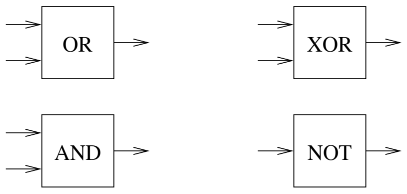
Similarly, look at the carry and the AND tables:

| $c$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

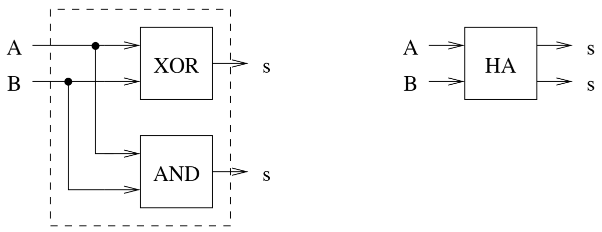
| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Therefore, $c$, the carry ignoring the previous carry is

$$c = A.B$$



We can therefore draw the logic gate diagram for the *partial sum* and *carry* digits (we have yet to deal with the carry-in, $I$) as follows:



lsls

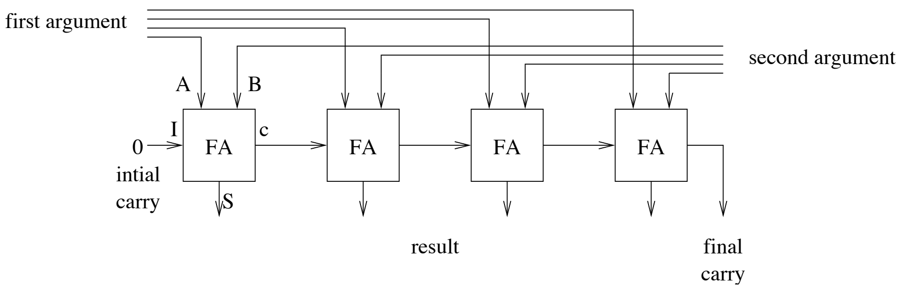


Figure 3: Addition using Full Adders

This is a *half adder*, as it is about half of what we need to make a full adder.

We now need to deal with the carry-in, $I$.

If the partial sum digit, $s$ (i.e. the half-adder sum) and the carry-in, $I$, differ, then the final sum digit $S$ will be 1. Otherwise, it will be 0. That is, it is just the sum of $I$ and $s$, or $S = I$ XOR $s$.

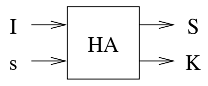
The carry-out, $C$, will be 1 if either:

1. the partial carry digit, $c$ (i.e. the half-adder carry) is 1, or
2. both the partial sum digit, $s$ and the carry-in digit, $I$, are 1:

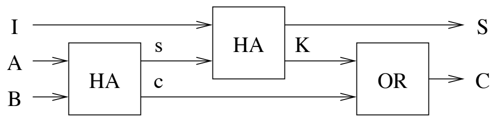
$$K = I.s$$

These conditions (1 and 2) cannot in reality occur together.

Notice that to compute $S$ and $K$, we can use a half adder whose inputs are $s$ (the partial sum digit) and $I$:



So the whole single stage addition process can be computed using two half adders and an OR gate:



These comprise a *full adder*. Full adders can be strung together to add multi-bit numbers. See Figure 3.

The full adders are used *serially*, as each must wait for the previous to provide its result before it can compute its own result. The the bits of the answer are produced one at a time.

A larger example. Add $011100 + 010110$ $(28 + 22)$. Read this table right to left, down the columns.

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| $A$ | 0 | 1 | 1 | 1 | 0 | 1 |  |
| $B$ | 0 | 1 | 0 | 1 | 1 | 0 |  |
| $I$ | 1 | 1 | 1 | 0 | 0 | 0 | previous carry $C$ |
| $s$ | 0 | 0 | 1 | 0 | 1 | 0 | half sum: $A$ XOR $B$ |
| $c$ | 0 | 1 | 0 | 1 | 0 | 0 | half carry: $A.B$ |
| $K$ | 0 | 0 | 1 | 0 | 0 | 0 | half carry: $I.s$ |
| $S$ | 1 | 1 | 0 | 0 | 1 | 0 | sum: $I$ XOR $S$ |
| $C$ | 0 | 1 | 1 | 1 | 0 | 1 | carry: K+C |

The result is 110010 (50).

## 1.17 A Parallel Scheme

If the bit patterns for $A$ and $B$ are available in parallel, addition can be accomplished by logic as follows ($A$ and $B$ now stand for sequences of bits, e.g., a byte or word):

1. while $B$ not all zeros do
    - $C \leftarrow A.B$ (carries)
    - $A \leftarrow A$ XOR $B$ (sums)
    - $B \leftarrow C << 1$ (shift carries up one place)
2. return $A$.

In this "$A.B$" means the *bitwise* AND of the bits from $A$ and $B$, namely ANDing each pair of bits in parallel.

In this, $C$ contains the carries of the individual bit sums. We add the bits of $b$ into $A$; shifting $C$ up by one place puts the carries into the right place to be added in the next loop. When $B$ is all zeros, that is there are no more carries, we are finished.

Example.

|  |  |  |
|---|---|---|
| $A$ | 011100 |  |
| $B$ | 010110 |  |
| $C$ | 010100 | bitwise AND |
| $A$ | 001010 | $A$ XOR $B$ |
| $B$ | 101000 | $C$ shifted |
| $C$ | 001000 |  |
| $A$ | 100010 |  |
| $B$ | 010000 |  |
| $C$ | 000000 |  |
| $A$ | 110010 |  |
| $B$ | 000000 |  |

Result is 110010, as before.

So, given hardware to do *bitwise parallel* operations on binary words, we can add together much faster than the serial full adder solution.

**Exercise:** Given $n$ bits, what is the *maximum* number of times we might go around the loop?

## 1.18 Bitwise Logical Instructions

It will by now come as no surprise to learn that there is a functional class of operations on most computers that performs the logical operations between bit patterns.

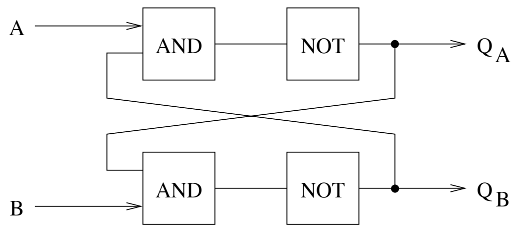
Thus, if $A = 01101010$ and $B = 11010000$

| | |
|---|---|
| AND will generate the pattern | 01000000 |
| INCLUSIVE OR will generate | 11111010 |
| EXCLUSIVE OR will generate | 10111010 |
| NOT on $A$ will generate | 10010101 |

# 2 Sequential Logic

The output of a logic gate in a *combinatorial* logic system may form the input to a gate "further down" the system. However, in a *sequential* system, the output of a logic gate can also be fed back so as to become the input to a gate "earlier" in the system.

The simple (inverted SR, or $\overline{SR}$) latch is an example:



Here we have the simultaneous equations

$$\begin{aligned} Q_A &= (A.Q_B)' \\ Q_B &= (B.Q_A)' \end{aligned}$$

This has solutions

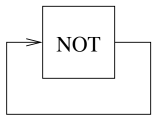
| $A$ | $B$ | $Q_A$ | $Q_B$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

But these solutions are not particularly useful in a dynamic system.

It is quite difficult to work out what happens in a system like this one. Let us make some helpful assumptions (which are valid for real implementations of such a system), namely:

- That there is a delay before a logic gate responds to an input;
- Separate logic gates cannot switch output simultaneously;
- When the power to the gates is switched on, we have no knowledge about the states of the outputs from the logic gates.
- We can therefore think in terms of the output one moment after the input, for all possible starting states of the system.

The requirement for a delay between input and output is essential if circuits are to make physical sense. Otherwise what does



do? The output would have to be simultaneously 0 and 1. With a small delay, the circuit could oscillate between the two values, which is at least physically possible.

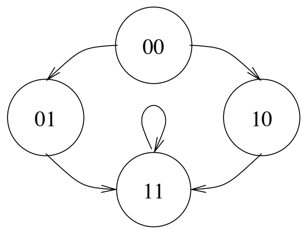
Let us assume that A and B are both set to zero. By chasing the values around the circuit we find:

| Old Outputs | | Inputs | | New Outputs | |
|---|---|---|---|---|---|
| $Q_A$ | $Q_B$ | $A$ | $B$ | $Q_A^{\text{new}}$ | $Q_B^{\text{new}}$ |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |

Thus, if $A = B = 0$ and we start with $Q_A = Q_B = 0$, we have, a moment later, $Q_A = Q_B = 1$.

## 2.1 State Transition Diagrams

We can also represent this table as a *state transition diagram*. The possible outputs of the system, $Q_A$ and $Q_B$, are represented in each circle and the arrowed arcs show the changes from an old output to a new one. Here is the state transition diagram when $A$ and $B$ are both 0:



Point of note:

- state 00 does not go directly to state 11 as the table might suggest. Slight asymmetries in the system means the state will either go through 01 or 10 first.
- we can't tell in advance which way it will jump: this is a *non-deterministic* change of state
- the transitions from 01 and 10 are *deterministic*, as we *do* know what will happen.

| Old Outputs | | Inputs | | New Outputs | |
|---|---|---|---|---|---|
| $Q_A$ | $Q_B$ | $A$ | $B$ | $Q_A^{\text{new}}$ | $Q_B^{\text{new}}$ |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

lsls

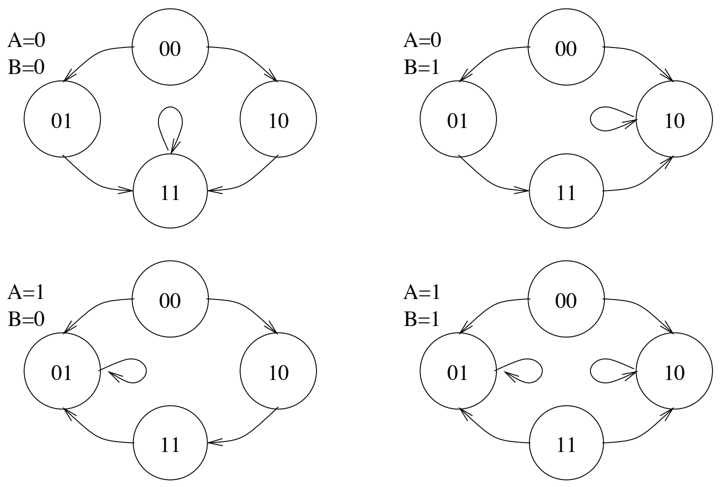


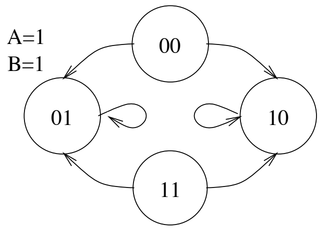
Figure 4: State Transitions of the Latch

The full set of state transition diagrams is in figure 4.

The state transition diagrams for all possible input combinations of A and B can be summarised by specifying the new state of $Q_A$ (that is, $Q_A^{\text{new}}$).

| $A$ | $B$ | $Q_A^{\text{new}}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | $Q_A$ |

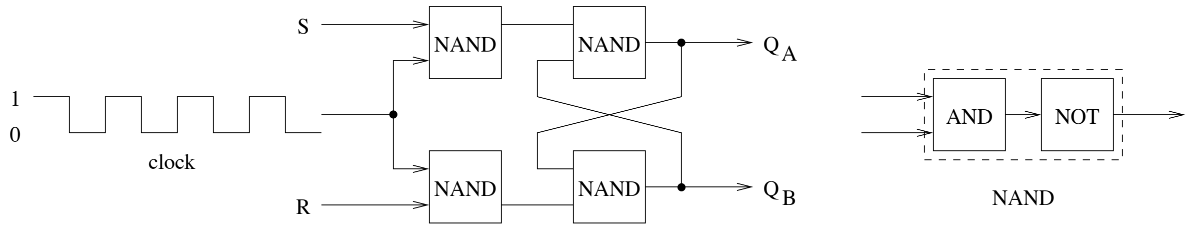
The last line needs some explanation.



We saw from the state transition diagrams that $Q_A$ and $Q_B$ will settle in different states, unless $A = B = 0$. If $A$ and $B$ both started at 0, they would not both become 1 simultaneously, but would pass through $A = 1$ and $B = 0$ or $A = 0$ and $B = 1$ on the way. Even in this case, therefore, $Q_A$ and $Q_B$ would settle into different states. Thus, when $A = B = 1$, $Q_A$ and $Q_B$ will always stay as they were.

## 2.2 The SR Flip-Flop

The simple latch needs extending to become a useful logic system. It needs a *clock* signal so that the latch responds to inputs only when the clock signal is present, rather than at any time (to simplify the diagram, a NAND gate is shown. It is simply an AND gate followed by a NOT gate):



Truth table for NAND:

| $A$ | $B$ | AND | NAND |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

The inputs are traditionally called $R$ and $S$, and the while thing an *SR flip-flop*, or and *SP gated latch.*

The purpose of the clock signal is this: when the signals $R$ and $S$ are changing the circuit can act in unpredictable ways (witness the non-deterministic transitions). Much better is to put the system into a *known* state, change the inputs, and only then look at the outputs.

This is what the clock does. When the clock is 0, the latch is held in a stable state. We can play with $R$ and $S$ as much as we like, as long as they have settled when the clock changes to 1. When the clock is 1, we then read off the new state. So

- clock 0 change inputs
- clock 1: read outputs

This minimises the problems of unstable transitions.

We can do exactly the same kind of analysis on this system as we did on the simple latch. In fact only four *distinguishable* state transition diagrams result:

| Inputs | | | Diagram |
|---|---|---|---|
| S | R | C | type |
| 0 | 0 | 0 | a |
| 0 | 0 | 1 | a |
| 0 | 1 | 0 | a |
| 0 | 1 | 1 | b |
| 1 | 0 | 0 | a |
| 1 | 0 | 1 | c |
| 1 | 1 | 0 | a |
| 1 | 1 | 1 | d |

When there is a clock signal only, or no clock signal, the system settles into one of the two stable states 01 and 10 (it is a *flip-flop* or *bistable* system).

Diagram (d) is important. It shows that if $S$ and $R$ are both set and the clock pulse is then removed, the system will transition to one of two possible steady states (01 or 10) and we cannot know which: it will be a non-deterministic transition.

lsls

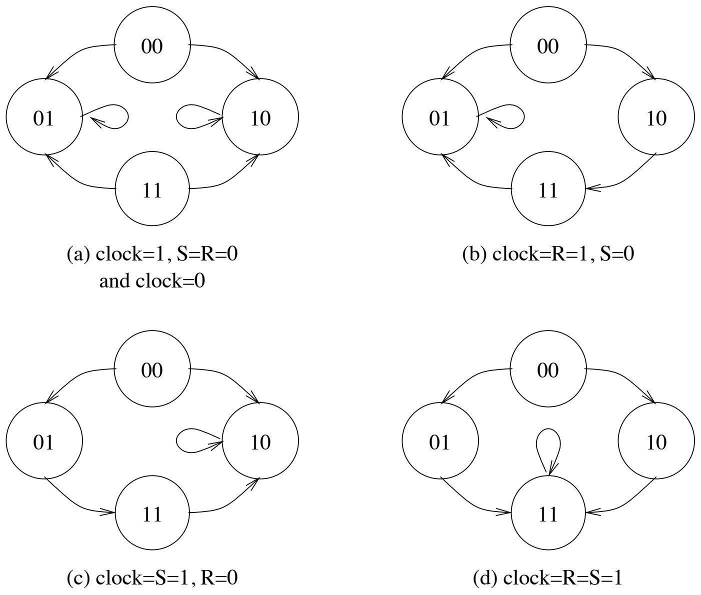


Figure 5: State Diagrams for SR Latch

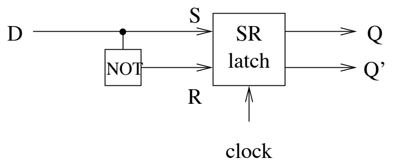
## 2.3 Summary of the Behaviour Of the SR Flip-Flop

The table below summarises our findings by showing the new output state ($Q_A^{\text{new}}$) of the flip-flop, given the $S$ and R inputs ($C$ is not included, as the flip-flop can only change while $C$ is set):

| $S$ | $R$ | Output ($Q_A^{\text{new}}$) |
|---|---|---|
| 0 | 0 | $Q_A$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | ? |

## 2.4 D-type Flip-Flop

One way to overcome the defect in the SR flip-flop (that is, the uncertainty of the state resulting from dropping the clock signal when $R$ and $S$ are both 1) is to ensure that the two inputs are always different. This is ensured in the *D-type flip-flop*:



The table showing the behaviour of the D-type flip-flop is thus very simple:

| D | $Q^{\text{new}}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

This is equivalent, in effect, to an SR flip-flop which is allowed only the two input states:

$$\begin{array}{lcl} S=0 & \text{and} & R=1 \\ S=1 & \text{and} & R=0 \end{array}$$

What is the point of the D-type flip-flop? It is a *one clock cycle delay.* That is, the $Q^{\text{new}}$, which appears as output one cycle after $D$ was input is equal to that $D$.

## 2.5 JK Flip-Flop

This type of flip-flop makes positive use of the fourth input combination, the one that is deficient in the SR flip-flop. It is used to *toggle*, or change the output state, of the flip-flop. The table describing the behaviour of the JK flip-flop is thus as follows:

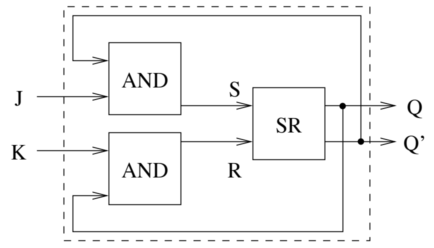
| $J$ | $K$ | $Q^{\text{new}}$ |
|---|---|---|
| 0 | 0 | $Q$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $Q'$ |

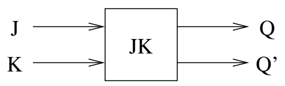
From the table showing JK flip-flop behaviour we can see, for example, that if the old JK output, $Q$, was 0 and the new output is to be 1, then we can achieve this by setting $J=1$ and $K=0$, or $J=1$ and $K=1$. The table for all output changes is:

| $Q^{\text{old}}$ | $Q^{\text{new}}$ | J | K |
|---|---|---|---|
| 0 | 0 | 0 | X |
| 0 | 1 | 1 | X |
| 1 | 0 | X | 1 |
| 1 | 1 | X | 0 |

X's are "don't care". So $J,K=0,\text{X}$ stands for both $J,K=0,0$ and $J,K=0,1$.

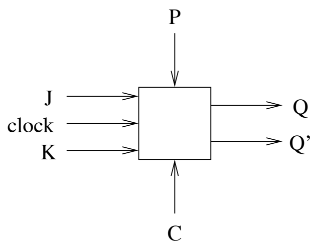
The *unclocked JK flip-flop* (or latch) can be constructed with an SR flip-flop plus two AND gates (the analysis is not given here):





The more common type of J-K flip-flop is the *clocked J-K flip-flop* which changes in response to the *lowering* of the clock signal.

As well as having a clock signal input, the flip-flop may also be shown with two additional inputs, namely, *unclocked set* ($P$) and *clear* ($C$) inputs:



## 2.6 Traffic Light Controller

A traffic light cycles through four states repeatedly, namely: red; red and amber; green; and amber. Let is represent these four states by two Boolean variables, $X$ and $Y$:

| State | $X$ | $Y$ |
|---|---|---|
| Red | 0 | 0 |
| Red & amber | 0 | 1 |
| Green | 1 | 0 |
| Amber | 1 | 1 |

We can now show the successor states for each state in this cycle:

| Starting | | Successor | |
|---|---|---|---|
| $X$ | $Y$ | $X$ | $Y$ |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

We shall use one clocked JK flip-flop to generate $X$ and another to generate $Y$.

Here is the control table we need:

| Old | | New | | JK controls | | | |
|---|---|---|---|---|---|---|---|
| $X$ | $Y$ | $X$ | $Y$ | $J_X$ | $K_X$ | $J_Y$ | $K_Y$ |
| 0 | 0 | 0 | 1 | 0 | X | 1 | X |
| 0 | 1 | 1 | 0 | 1 | X | X | 1 |
| 1 | 0 | 1 | 1 | X | 0 | 1 | X |
| 1 | 1 | 0 | 0 | X | 1 | X | 1 |

As before, X denotes a "don't care" term.

We can now draw Karnaugh maps of the JK controls and simplify them in the usual way.

The J-K controls are thus:

$$J_X = K_X = Y$$
$$J_Y = K_Y = 1$$

The control system is therefore:

lsls

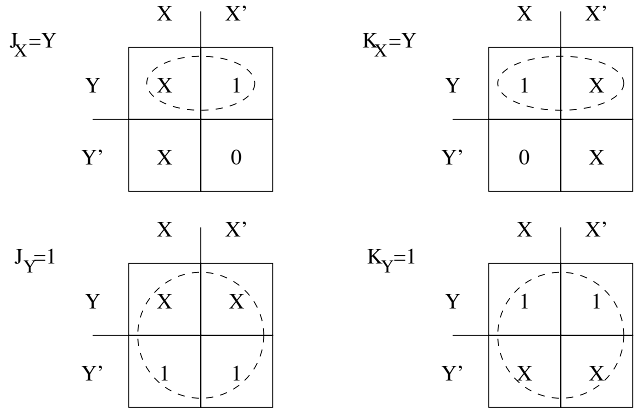
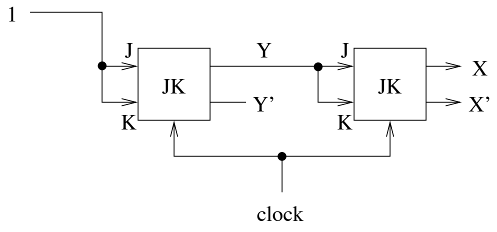


Figure 6: Karnaugh Maps for Traffic Lights



We now need to connect the outputs $X$ and $Y$ to the lights.

| $X$ | $Y$ | Red |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

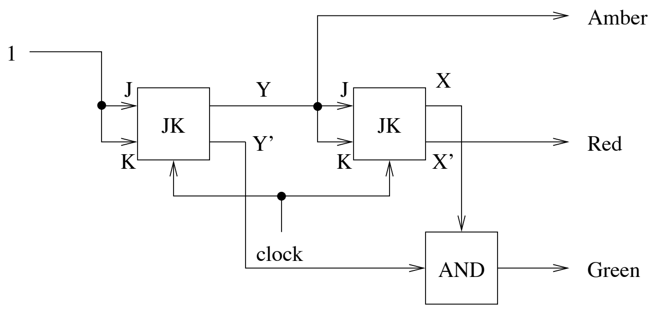
| $X$ | $Y$ | Amber |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $X$ | $Y$ | Green |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Thus

$$\begin{aligned} R &= X'.Y' + X'.Y = X' \\ A &= X'.Y + X.Y = Y \\ G &= X.Y' \end{aligned}$$

So the final control system is



As the clock ticks, so the lights cycle though the expected sequence.