1 A Computational Model

1.1 “A Computational Model”

“A Computational Model” Computing, like any other area of knowledge, consists of a huge amount of detail. The science in Computer Science comes from the way we organise and partition the detail so that it can be used. In particular, we often use layers, so we can concentrate on one layer at a time. You can think of this semester’s course-units as a series of layers:

<table>
<thead>
<tr>
<th>Course Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP101</td>
<td>how to use existing computer programs</td>
</tr>
<tr>
<td>COMP161/9</td>
<td>how to create your own computer programs</td>
</tr>
<tr>
<td>COMP151</td>
<td>how programs are made to run on computers</td>
</tr>
<tr>
<td>COMP121</td>
<td>how computers are created</td>
</tr>
</tbody>
</table>

Within each course-unit, we concentrate on the problems and concepts relevant to that layer, and ignore most of the details used in other layers. Within each layer, we often look at how the basic concepts from the layer above can be transformed into concepts that are one step simpler, and at how much of the transformation can be automated. (You may even find that the information within a single course-unit is further divided into a series of sub-layers.)

“Computational…” What happens when a program runs? e.g. a Java program on an ARM computer.

In this course-unit, we assume that Java programs and computers exist, and explore the bridge between them – how one is made to run on the other. In particular, we will look at the transformation of the basic concepts of Java programs (e.g. statements and expressions) into simpler concepts (e.g. computer instructions) and then briefly look at how this can automated (e.g. by compilers).

“A…” There are many computational models. We are going to look at the simplest/earliest/commonest/most important, the Von Neumann Imperative model.

1.2 The Von-Neumann Computer Model

The essence of a computer can be captured in some very simple properties:

- it can store information
- it can manipulate the stored information
- it can make decisions depending on the stored information

A very simple view of a computer considers it as having two main components and a link between them:

The **memory** is a set of **locations** which can hold information, such as numbers. Each location has a unique **address**, usually given as a number. For example, we might say things like: “memory location 100 contains the number 12345”, or: “the word at address 100 is 12345”. (Sometimes, to make things easier for humans to understand, we use names for memory locations rather than numbers – “memory location x contains 12345” – but those names have to somehow be replaced by numbers to make sense to the computer.) Nowadays, typical memories consist of many hundreds of millions of different locations.

A **bus** is a bidirectional communications path (i.e. information can flow in either direction, like a telephone) transmitting addresses and numbers.

The **central processing unit** (CPU) obeys a sequence of **instructions**, known as a **program** (or sometimes **code**, as in “this code calculates the total of a list of numbers”). Just about every kind of CPU has a different style of instructions it recognises, so to start with we will use a very simple style, known as **three-address** format, where each instruction tells the computer to:

- send copies of the values in any two memory locations to the CPU,
- perform some operation, such as adding the copied numbers together,
- copy the result back from the CPU into a third memory location.

For example, suppose we wanted to find the sum of two values, “a” and “b”. In Java, we might write this as:

```
sum = a + b;
```
In Java, the names “a”, “b” and “sum” represent variables, that each can have a value, which can be changed by the actions of the computer. Each variable usually corresponds to a particular memory location which actually contains the value as the program is obeyed.

The corresponding three-address instruction would simply be:

\[
\text{sum} = a + b
\]

To simplify things, we have given the memory locations the same names as the variables (“sum”, “a” and “b”) rather than picking particular numbers. The single instruction tells the computer to copy the values of the memory locations “a” and “b”, add them, and copy the result to memory location “sum”, overwriting whatever was there before.

(For a real computer, we are more likely to write this in a slightly different way, such as \textit{add} \texttt{sum}, \texttt{a}, \texttt{b} but the underlying meaning is the same – we will come back to this in the next lecture.)

Each instruction consists of an \textbf{operation}, such as addition, and (implicit and explicit) \textbf{operands}. For three-address instructions, there are two source operands, such as “a” and “b”, which provide the values to be operated on, and a destination operand, such as “sum”, where the result is to be placed.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example}
\caption{Example of three-address instruction}
\end{figure}

Another example

Suppose we now want to find the sum of three values, “a”, “b” and “c”. In Java, we might write this as:

\[
\text{sum} = a + b + c;
\]

We need to copy the values in three different memory locations to the CPU to add together, but we can only copy two such values in a single three-address instruction. Therefore, we have to use more than one instruction, which the computer will obey in sequence:

\[
\text{sum} = a + b
\]
\[
\text{sum} = \text{sum} + c
\]

As before, the first instruction tells the computer to copy the values of the memory locations “a” and “b”, add them, and copy the result to memory location “sum”, overwriting whatever was there before. The computer now obeys the second instruction, which tells it to copy the values of the memory locations “sum” (which, because of the first instruction, currently contains the total of “a” and “b” and “c”), add them, and copy the result (the total of “a”, “b” and “c”) to memory location “sum”, overwriting the value that was put there before.

When writing in Java, we can make calculations as complicated as we need, but each computer instruction is limited as to how much it can do. Therefore, a single Java statement may be equivalent to several computer instructions, whatever the make of computer.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{example2}
\caption{Three-address instruction example}
\end{figure}

\textbf{Efficiency}

Let’s think about the potential speed of this instruction sequence. Looking at the original Java, we should use memory 4 times, once to obtain the value in each of “a”, “b”, and “c”, and one last time to copy the result back to “sum”. However, the instruction sequence consists of 2 three-address instructions, so we have actually used memory 6 times. To put it another way, “sum” is used two extra times, to remember and then retrieve the intermediate value \texttt{a + b}. Roughly speaking, the speed of a computer is limited by the speed of its memory, and increasing the memory usage from the theoretical value of 4 to an actual value of 6 is equivalent to slowing the computer down by a third!

Although three-address instructions are very simple, they tend to be slow. Most real computers do not use them, but instead use different instruction styles, intended to be faster. Many different instruction styles have been tried since the first true computers were built in the late 1940s. The search has been made more complicated because the optimum solution changes as the available implementation technologies change.
However, it is possible to identify two main themes:
- to minimise use of memory, which has been getting progressively slower compared to CPU technology
- to simplify instructions, so that each one has less to do
  pro: no wasted effort (e.g. saving and reloading “a+b” in “sum”)
  pro: easier to build fast CPUs
  con: need more instructions for any given piece of e.g. Java

Part of the problem has been that these two themes clash – it has been necessary to make instructions more complex to let them use memory more efficiently – by including an extra, very small, very fast memory within the CPU. This is usually in the form of registers, each of which can hold a single number. Nowadays, typical CPUs contain up to a few dozen registers (e.g. an ARM computer has 16 registers known as R0 to R15 or r0 to r15).

The solution has been to make instructions as simple as possible while allowing for the best use of registers:
- **one-address** style, as used in the Intel Pentium family: use registers instead of the other two addresses
- **load-store** style, as used in ARM computers: use registers instead of all three addresses, and use extra instructions to copy information from memory to registers (load) and from registers to memory (store).

For example, load-store style for \(\text{sum} = a + b + c;\)

\[
\begin{align*}
\text{register1} &= a \quad (\text{i.e. load}) \\
\text{register2} &= b \quad (\text{i.e. load}) \\
\text{register2} &= \text{register1} + \text{register2} \quad (\text{i.e. a+b}) \\
\text{register1} &= c \quad (\text{i.e. load}) \\
\text{register2} &= \text{register2} + \text{register1} \quad (a+b+c) \\
\text{sum} &= \text{register2} \quad (\text{i.e. store})
\end{align*}
\]

Comparing this with the original Java, we are now using one load or store instruction per variable (such as “a” or “sum”) and one instruction per operation (such as “+”), so lots of very simple, very fast instructions, but most important – no wasted effort!

### 1.3 Exercises

1. Have you ever read about/learnt about/used assembly code before? If so, for what computer(s) (real or invented)?

If you can remember enough of the details, use them for part (c) of each of the following questions:

2. Write computer instructions to calculate \(a = b - c;\) using:
   a) three-address style instructions  
   b) load-store style instructions  
   c) any assembly code(s) you already know

3. Write computer instructions to calculate \(a = b \times c \times d;\) using:
   a) three-address style instructions  
   b) load-store style instructions  
   c) any assembly code(s) you already know

4. Write computer instructions to calculate \(a = (b \times c) + (d \times e);\) using:
   a) three-address style instructions  
   b) load-store style instructions  
   c) any assembly code(s) you already know

   (use an extra memory location “temp”)
In the last lecture, we briefly looked at the load-store style of computer instructions:

\[
\text{e.g. for } \text{sum} = a + b + c; \\
\text{register}_1 = a \text{ (i.e. load)} \\
\text{register}_2 = b \text{ (i.e. load)} \\
\text{register}_2 = \text{register}_1 + \text{register}_2 \text{ (i.e. } a+b) \\
\text{register}_1 = c \text{ (i.e. load)} \\
\text{register}_2 = \text{register}_2 + \text{register}_1 \text{ (a+b+c)} \\
\text{sum} = \text{register}_2 \text{ (i.e. store)}
\]

To fill in the details, for most of the rest of this course-unit we will be looking at a typical load-store computer, the ARM. This has load instructions of the form: LDR register, source operand

\[\text{e.g. LDR R1, a } \text{(i.e. Register 1 = a)}\]

and store instructions of the form: STR register, destination operand

\[\text{e.g. STR R2, sum (i.e. sum = Register 2)}\]

The “destination operand” and “source operand” can take various forms, all of which indicate a memory location – for the time being, we will just use a location name or number. We will look at the alternative forms in later lectures.

The ARM also has various instructions for performing arithmetic on registers, such as addition:

\[\text{ADD destination register, source register, source register}\]

\[\text{e.g. ADD R2, R1, R2}\]

Using these, we can rewrite the code above as:

\[
\text{LDR R1, a } \\
\text{LDR R2, b } \\
\text{ADD R2, R1, R2 } \\
\text{LDR R1, c } \\
\text{ADD R2, R2, R1 } \\
\text{STR R2, sum}
\]

We also need to know how to stop the computer after it has obeyed all our instructions. This depends on the system software being used, but you will find that this works in the labs: SWI 2

**ARM Load, Store, and Arithmetic Instructions**

\[
\text{LDR \quad Ra, b \quad \text{register } \text{“a”} = \text{“b”}} \\
\text{STR \quad Ra, b \quad \text{“b”} = \text{register } \text{“a”}} \\
\text{ADD \quad Ra, Rb, Rc \quad \text{register } \text{“a”} = \text{register } \text{“b”} + \text{register } \text{“c”}} \\
\text{SUB \quad Ra, Rb, Rc \quad \text{register } \text{“a”} = \text{register } \text{“b”} - \text{register } \text{“c”}} \\
\text{MUL \quad Ra, Rb, Rc \quad \text{register } \text{“a”} = \text{register } \text{“b”} \times \text{register } \text{“c”}}
\]

**ARM SoftWare Interrupt Instruction**

– access to system software

An Example Program

These instructions are enough to add some numbers together and save the total. For example, it could be used in a cash till, to add the cost of individual items together to give the total cost. We will start by looking at one step of this process – adding the price of a single item, worth £1 9p, to a running total of £15 34p.

First we have to decide on the representation of prices inside the computer. A simple story is that £1 9p is represented by 109 and £15 34p by 1534 (i.e. the number of pence).

Now, suppose we have the running total held in memory location “total” and the item price in location “next”.

We could use this program:

\[
\text{LDR R0, total } \\
\text{LDR R1, next } \\
\text{ADD R2, R1, R0 } \\
\text{STR R2, total } \\
\text{SWI 2}
\]

Then, if we set the computer to go through these instructions in sequence, it will:
• copy 1534 from location “total” and remember it in register 0
• copy 109 from location “next” and remember it in register 1
• copy 1534 from register 0 and 109 from register 1, add them together, and put the result of 1643 into register 2
• copy 1643 from register 2 back into location “total”
• stop the program

and we end up with the answer, 1643 representing £16 43p, in memory location “total” where we keep the running total (and with a copy in register 2) and the computer halted.

2.1 Stored programs and the Program Counter

We said in the last lecture (section 1.2) that part of the essence of a computer is that it can make decisions, and so obey different instructions depending upon the results of those decisions. To see how this is possible, we first need to explore how the sequence of instructions is controlled.

The list of instructions is clearly a vital part of the computational process and following it automatically is an essential part of the operation of a computer.

An important part of the Von-Neumann Model, used by all practical digital computers, is that the memory is used to hold instructions as well as data (i.e. numbers or any other information that the program is working on) – thus "stored program". Computers have a Program Counter (PC) register in the CPU which holds the address of the memory location containing the next instruction to be obeyed (executed). On the ARM, this is register 15.

To be stored in memory locations, instructions have to be represented by numbers. The way in which an operation and its operands are represented by a single number is similar to the way in which pounds and pence are represented by a single number in our cash till example. (We will come back to this in lecture 4.)

At the start of the program described above, the picture is now something like this.

The computation starts with the PC set pointing to the address of the first instruction, in this case 0. The computation proceeds by obeying the instruction pointed to by the PC and then changing the PC to point to the memory location containing the next instruction. (Note that PC actually increases by 4 each time, as each instruction occupies 4 memory locations – we will come back to this in lecture 4.)

• PC (R15) contains 0, so copy the first instruction from memory location 0, and set PC pointing to the memory location containing the next instruction (i.e. 4).
• obey this instruction: copy 1534 from location “total” and remember it in register 0
• copy the next instruction, from memory location 4, and set PC pointing to memory location 8.
• obey this instruction: copy 109 from location “next” and remember it in register 1
• copy the next instruction, from memory location 8, and set PC pointing to memory location 12.
• obey this instruction: add 1534 from register 0 and 109 from register 1 and put the result (1643) into register 2
• copy the next instruction, from memory location 12, and set PC pointing to memory location 16.
• obey this instruction: copy 1643 from register 2 back into location “total”
• copy the next instruction, from memory location 16, and set PC pointing to memory location 20.
• obey this instruction: stop the program

and just as we saw before, we end up with the answer in memory location “total” and the computer halted.

This is known as the fetch-execute cycle: each instruction in turn is fetched from memory and then executed (obeyed).
2.2 Decision making

With linear lists of instructions, we can do limited things. Most of the power of computers comes from the power to change between different sequences of instructions as a result of tests on the stored information.

Suppose, in our cash till example, we want to give customers a £1 discount if they have selected items worth £20 or more. The program must make a decision – compare the total to £20 and see if it is larger – and then, depending on the results of the decision, either perform an action or not – subtract £1 from the total. Because a computer has no intelligence, we have to rework this into a form the computer can use.

To start with, we need to formalise the decision and action:

\[
\text{if total } \geq \text{ £20 then } \text{ deduct £1 from total}
\]

Then, we need to rewrite it into a form that is harder for a human to understand, but more suited for use by a computer:

\[
\text{if total } < \text{ £20 then don't deduct £1 from total (i.e. otherwise do this)}
\]

Finally, we need to convert this into ARM instructions, and because each instruction can only do a limited amount, the decision actually takes two steps:

- Firstly, compare the two values (“total” and “£20”), using a CMP instruction.
- Secondly, determine the exact result of the comparison using a conditional instruction – in this case a BLT branch.

CMP compares its two operands, by subtracting the second from the first. Rather than saving the numerical result anywhere, it just causes the CPU to remember whether it is e.g. positive, negative or zero.

Branch instructions, such as B or BLT, allow us to change from one sequence of instructions to another. The second part of each branch instruction is the address of the start of the next sequence of instructions, normally indicated by some invented name e.g. nodiscount. This name is known as a label.

The B instruction is an unconditional branch, as it always causes a change to the value of PC, and thus a “branch” to a different instruction sequence. (It can also be written as BAL, for Branch ALways.)

BLT is known as a conditional branch, as it may or may not cause a change, depending on the comparison made just before. It causes a change only if the previous result was Less Than (in this case, total less than £20).

```assembly
LDR R0, total as before, add total and next, copying result back into total
LDR R1, next
ADD R2, R1, R0
STR R2, total the new value of total is also still in R2

LDR R3, minimum
CMP R2, R3 compare total and £20
BLT nodiscount if total < £20 then skip over next part

LDR R4, discount
SUB R5, R2, R4 subtract £1 from copy of total still in R2
STR R5, total and copy result back to total

nodiscount SWI 2 either way, stop, as before
```

We need to set aside memory locations for “total” and “next”. We can use DEFW (DEFine Word) to do this. DEFW is a pseudo operation i.e. it looks like an ARM instruction, but in fact it isn’t. DEFW just tells the system you will be using in the labs to put the number after it into the next memory location in the program. We can use this to also put the right starting values into “total” and “next”.

For now, the simplest way of using constants like “100” and “2000” is to also put them into memory locations. (We will look at a better solution in lecture 4.)
CMP Ra, Rb  
calculate register “a” – register “b” but don’t save the result  
instead remember if it was positive or negative, zero or non-zero, etc.

ARM Comparison Instruction

<table>
<thead>
<tr>
<th>B</th>
<th>somewhere</th>
<th>set PC (R15) so next instruction is taken from “somewhere”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bxx</td>
<td>somewhere</td>
<td>if condition “xx” (see table of conditions below) is true then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set PC (R15) so next instruction is taken from “somewhere”</td>
</tr>
</tbody>
</table>

ARM Branch Instructions

<table>
<thead>
<tr>
<th>AL or blank</th>
<th>ALways</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>EQual</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
</tr>
<tr>
<td>GE</td>
<td>signed Greater than or Equal</td>
</tr>
<tr>
<td>LT</td>
<td>signed Less Than</td>
</tr>
<tr>
<td>GT</td>
<td>signed Greater Than</td>
</tr>
<tr>
<td>LE</td>
<td>signed Less than or Equal</td>
</tr>
<tr>
<td>VS</td>
<td>overflow Set</td>
</tr>
<tr>
<td>VC</td>
<td>overflow Clear</td>
</tr>
<tr>
<td>CS or HS</td>
<td>Carry Set, unsigned Higher or Same</td>
</tr>
<tr>
<td>CC or LO</td>
<td>Carry Clear, unsigned LOwer</td>
</tr>
<tr>
<td>HI</td>
<td>unsigned Higher</td>
</tr>
<tr>
<td>LS</td>
<td>unsigned Lower or Same</td>
</tr>
<tr>
<td>MI</td>
<td>minus (negative)</td>
</tr>
<tr>
<td>PL</td>
<td>plus (positive or zero)</td>
</tr>
</tbody>
</table>

ARM Conditions

2.3 Exercises

1: What are the contents of the registers (R0, R1, R2) and variables (one, two, three) after each of the following instructions is obeyed:

```
LDR R0, one
STR R0, three
LDR R1, two
ADD R2, R1, R0
STR R2, one
SWI 2
```

one DEFW 23

two DEFW 45

three DEFW 10

2: Write ARM instructions to calculate \( a = (b \times c) - (b + c) \);
(hint: try to avoid loading \( b \) or \( c \) more than once)

3: Assuming the first instruction in question 1 is at address 0, what will be the contents of the PC register after each instruction is obeyed?

4: Write ARM instructions to do the following:
If \( a \) is greater than \( b \), then subtract \( b \) from \( a \).
i.e. if \( a > b \) then \( a = a - b \);

5: Write ARM instructions to put zero into a variable e.g. \( a = 0 \);
(hint: you can make your code more efficient if you can think of an arithmetic instruction that will always give you an answer of zero)
3 How are programs stored?

In the last lectures, we looked at how the PC register can be used to step through a program held in memory, and modified to branch to a different sequence of instructions. However, we left a whole series of questions unanswered, including:

- How are numbers and instructions held in a computer memory?
- Why does PC change in steps of 4?
- How big are registers like PC?

3.1 Binary

If we look at the electronics of how a computer works, we need to talk about such things as voltages and currents. If we look deeper, at the physics, we need to talk about such things as atoms and electrons. However, we are only concerned with a computational model, and so don’t want to get bogged down in this sort of detail. Instead, we want to concentrate on a higher level, on meanings. As far as these lectures are concerned, everything in a computer is composed of \textbf{bits}, each of which can hold one of two values – either a 0 or a 1 (a bit is a \textit{binary} digit).

Suppose we want to represent the answer to a simple question on a computer, such as “Is today a Monday?” The answer we, as human beings, expect is either a “Yes” or a “No”. We can represent these two possible answers using a single bit, with “Yes” being represented by a 1 and “No” by a 0. Alternatively, we might decide to represent “Yes” by a 0 and “No” by a 1. At first glance, either representation would be equally useful – we just have to remember which representation we have chosen, and stick with it. However, by convention, we normally use the representation that “Yes” or “True” is 1 and “No” or “False” is 0 – unless there is a good reason not to!

What if we want to deal with a question with a slightly bigger set of possible answers, such as “What colour is this traffic light?” There are now four possible answers: “Red”, “Amber”, “Green”, and “Red and amber”. We have to use two bits, which together can take four different patterns: 00, 01, 10, and 11. There is no normal convention here, so how do we match up the answers to their representations? Here is one possibility:

\begin{itemize}
  \item 00 is “Green”
  \item 01 is “Red”
  \item 10 is “Amber”
  \item 11 is “Red and amber”
\end{itemize}

(Can you see any reason why I might prefer this representation?)

Suppose now we want to represent the day of the week – this has seven different possible answers: “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday” and “Sunday”. We certainly can’t represent these just using two bits, but three bits gives us eight patterns: 000, 001, 010, 011, 100, 101, 110, and 111. We first have to decide which seven patterns we are going to use, and then which pattern corresponds to each day. For example, we might use:

\begin{itemize}
  \item 001 is “Monday”
  \item 010 is “Tuesday”
  \item 011 is “Wednesday”
  \item 100 is “Thursday”
  \item 101 is “Friday”
  \item 110 is “Saturday”
  \item 111 is “Sunday”
\end{itemize}

000 is unused, so if we ever saw it, it would indicate some sort of error.

Suppose we want to represent the day of the month – this is any number from 1 to 28 or 1 to 29 for a February or from 1 to 30 or 1 to 31 for other months. Let’s just decide how to represent any number from 1 to 31. We have seen above that:

- 1 bit can take 2 different patterns,
- 2 bits allow up to 4 different patterns,
- 3 bits allow up to 8 different patterns.
- In general, n bits allow up to \(2^n\) different patterns.

As \(2^4\) is 16, and \(2^5\) is 32, 4 bits would not be enough, but 5 bits would allow us to represent all 31 different day numbers, with one combination unused. The simplest way to pick a bit-pattern to represent each day-number would just be to use the same representation as for any other numbers (see below).
3.2 Characters

Suppose we want to represent characters – how many different characters are there?

- A to Z is 26
- a to z is another 26
- 0 to 9 is 10
- + - * / ( ) < = > is 9
- . , : ; ? ! ‘ ’ is 8

space, tab and newline make 3 more, and so on – there are over 80 characters we might want to use, so how many bits do we need? 6 bits gives us $2^6$ which is 64 combinations, and 7 bits gives us $2^7$ which is 128, so it looks like 7 bits would be about right.

An early standard was ASCII, which defined 128 different English and American characters. e.g.

- A is represented by 1000001
- B is represented by 1000010
- Z is represented by 1011010
- a is represented by 1100001
- b is represented by 1100010
- z is represented by 1111010
- 0 is represented by 0110000
- 9 is represented by 0111001
- + is represented by 0101011
- . is represented by 0101110

(An alternative encoding, now essentially abandoned, was EBCDIC.)

However, this list of characters is hopelessly parochial – what about the various kinds of accents found in European languages (e.g. ó ò ô ö ò ò ô ø œ ɔ ̈ o ç ́ o) and Greek letters (e.g. α β γ δ) and Japanese, and Chinese, and Arabic, and so on – there would seem to be no limit to how many characters we need!

The ASCII standard was extended to include 128 extra European characters by the ISO 8859 standard. (This standard actually defines at least 15 different possible sets of 128 extra characters, known as 8859-1 to 8859-15, but only one such set can be used at once. Linux uses the 8859-1 representation for characters.) ISO 8859 characters are represented by 8 bits, which has become a fundamental unit of storage, known as a byte – larger pieces of memory are nowadays usually defined as multiples of bytes.

This in turn was extended to support characters sets used by the rest of the world in the Unicode (ISO 10646) standard, as used by Java – however this requires 16-bit and longer encodings. The first 256 characters of ISO 8859-1 and ISO 10646 are the same.

For more information, type ‘man ascii’ or ‘man iso8859-1’ or ‘man -k 8859’ or ‘man unicode’ or ‘man utf-8’ on a Linux computer.

3.3 Integers

Suppose we want to represent integers. How many bits should we use? Mathematically speaking, there is no maximum size for an integer, but we need to pick some standard size or sizes, that allow us to use a reasonable range of numbers without wasting too much memory space.

- 8 bits would allow us to represent $2^8$ i.e. 256 different numbers – say 0 to 255
- 16 bits would allow us to represent 65,536 different numbers
- 24 bits would allow us to represent 16,777,216 different numbers
- 32 bits would allow us to represent 4,294,967,296 different numbers
- 64 bits would allow us to represent 18,446,744,073,709,551,616 different numbers

In the past, computers have used 8, 16, and 24 bits, and many other sizes, to hold numbers. Nowadays, most use 32 bits, but some even use 64 bits. Java defines integers to be exactly 32 bits long (or at least to behave as if they are).

We now need to assign a bit pattern to each integer value. We will start by considering just positive integers i.e. 0, 1, 2, 3 etc. Although in theory we could use any arbitrary way of assigning numerical values to bit patterns, in practice there is really only one sensible representation, that makes it easy to encode arbitrary numbers and build simple hardware to perform arithmetic: base 2 representation.

To convert a positive decimal number to base 2, all we need to do is repeatedly divide by 2, making a note of the remainder at each step. e.g. to find the binary representation for 100:
100 / 2 = 50 remainder 0
50 / 2 = 25 remainder 0
25 / 2 = 12 remainder 1
12 / 2 = 6 remainder 0
6 / 2 = 3 remainder 0
3 / 2 = 1 remainder 1
1 / 2 = 0 remainder 1

The remainders (in reverse order) are: 1100100, and this is the binary encoding for 100 i.e. $100_{10} = 1100100_2$.

What we have created is a 7-bit pattern. If we wanted e.g. the 8-bit or 16-bit pattern we would have to keep dividing by two, but at each extra stage we would just be dividing 0 by 2 to get 0 remainder 0, so the 7-bit pattern 1100100 represents the same number as the 8-bit pattern 01100100 or the 16-bit pattern 0000000001100100 and so on.

To convert a positive base 2 number to decimal, we need to multiply each bit value of 0 or 1 by the corresponding power of two and add the results:

\[
\begin{align*}
0 \times 2^0 &= 0 \\
0 \times 2^1 &= 0 \\
1 \times 2^2 &= 1 \times 4 = 4 \\
0 \times 2^3 &= 0 \times 8 = 0 \\
0 \times 2^4 &= 0 \times 16 = 0 \\
1 \times 2^5 &= 1 \times 32 = 32 \\
1 \times 2^6 &= 1 \times 64 = 64 \\
\text{total} &= 100
\end{align*}
\]

To give some more examples, using a 16 bit representation, one is $0000000000000001$, two is $0000000000000010$, three is $0000000000000011$, four is $0000000000000100$, ten is $000000000001010$, one hundred is $000000001100100$, one thousand is $0000011111101000$, and ten thousand is $0010011100100000$.

As you can see from the examples, binary numbers are not easy for humans to use, as we are so used to base 10. There are two tricks we use to make life a little easier:

- The first is just to group the bits into threes ($1 100 100$ or $001 100 100$) or fours ($110 0100$ or $0110 0100$).
- The second is to replace the groups of bits by the corresponding decimal value for each group, so $001 100 100$ can be written as $144_8$ (octal notation) and $0110 0100$ can be written as $64$ or $64_{16}$ (hexadecimal or hex notation).

In hexadecimal notation, we have to deal with the 4-bit patterns $1010$ to $1111$ (ten to fifteen). As there are no more decimal digits available, we start using the letters of the alphabet, replacing:

- $1010$ (ten) by A or a,
- $1011$ (eleven) by B or b,
- $1100$ (twelve) by C or c,
- $1101$ (thirteen) by D or d,
- $1110$ (fourteen) by E or e,
- $1111$ (fifteen) by F or f.

So, the hexadecimal number $1A5E_{16}$ is equivalent to the binary number $0001 1010 0101 1110$, which is the same as $0 001 101 001 011 110$, which is equivalent to the octal number $015136_8$ or $15136_8$.

When you are in the lab, the system assumes numbers are decimal unless told otherwise. You can use hexadecimal numbers by writing e.g. $0x1A5E$ and binary by writing e.g. $0b0001101001011110$ but there is nothing similar for octal. Java has similar conventions – find out what they are!

**Negative numbers**

There is still the problem of dealing with negative numbers, like $-1$. Most computers use twos-complement representation to handle signed (i.e. both positive and negative) numbers. You will be given lots of details about this in CS1211 but for the time being here is a very brief summary. If we use e.g. a 8-bit representation allowing for $2^8$ (256) different values, instead of these values being 0 to 255 they are $-128$ to $+127$. We use modulo 256 arithmetic so $-x$ is represented by the value $256-x$. A useful side-effect of this is that you just have to look at the most significant bit of a twos-complement number to tell if it is positive (0) or negative (1). However, this also means that we need to make sure that every positive number starts with at least one 0 bit, or else it will be misunderstood to be negative.


3.4 Powers of 10 and of 2

Kilo  
\[ 1k = 1000 = 10^3 = 1 \text{ thousand} \approx 2^{10} = 1024 \]

Mega  
\[ 1M = 10^6 = 1 \text{ million} \approx 2^{20} = 1048567 \]

Giga  
\[ 1G = 10^9 = 1 \text{ US billion} \approx 2^{30} = 1073741824 \]

Tera  
\[ 1T = 10^{12} = 1 \text{ UK billion} \approx 2^{40} = 109951162776 \]

In computing, powers of 10 are often approximated by powers of 2, so we talk about e.g. a Megabyte of memory meaning \(2^{20}\) bytes. However, we also talk about a MegaFLOPS meaning \(10^6\) FLOPS. Originally, typical computer numbers were Ks, and the errors introduced were only about 2%. Nowadays we usually talk about Ms and Gs, and are starting to talk about Ts, where the differences are much bigger, so it is probably better to avoid the powers of 2 when there is any ambiguity.

milli:  \(1m = .001 = 10^{-3} = 1 \text{ thousandth}\)
micro:  \(1\mu = 10^{-6} = 1 \text{ millionth}\)
nano:  \(1n = 10^{-9} = 1 \text{ US billionth}\)

3.5 Exercises

1: Without using a calculator, convert the decimal number 47 to binary, and then to octal and hexadecimal.

2: Without using a calculator, convert the hexadecimal number 47 to octal and binary, and then to decimal.

3: University of Manchester student registration numbers used to consist of a year number (e.g. 5 for 2005) followed by 5 more digits.
What was the maximum number of students that the University could register in any one year without changing the format of registration numbers?

How many bits would be needed to number this many students?

4: In Britain, new car registration numbers are of the form “LLDD LLL” where L indicates any capital letter except “I”, and D any decimal digit. The “DD” part indicates the date, and changes every 6 months.

What is the maximum number of cars that can be registered in Britain in any 6-month period?

How many bits would be needed to number this many cars?

5: FAT-based filesystems, like early versions of MS-Windows, used the last 3 characters of the filename to indicate its type (e.g. .txt, .htm, .mp3). Assume that each of the 3 characters can be a decimal digit or a letter, but upper-case and lower-case are treated as being the same.

How many different file types can be encoded?

How many bits would be needed to record this information?

Assuming that the filesystem actually uses a byte to hold each character, how much memory is being wasted?
4 How are ARM programs stored?

The design of any real computer is always a trade-off between including everything that the programmers would like to be able to use, and only building the minimum that the hardware designers can get away with. One of these trade-offs is deciding what sizes of bit-patterns will be easy to access – it might be useful if we could use any size (e.g. 1-bit, 2-bit, 3-bit, and so on), but it would be much simpler to design hardware that can only access a few different fixed sizes.

The most important access sizes are those used for values (characters and numbers), for instructions, and for memory addresses. At the time the ARM was being designed, the most reasonable size for characters was 8 bits and for integers and addresses was 32 bits. A fundamental aim of the designers was to simplify the hardware as much as possible, so the original design only allowed these two different access sizes: 8-bit (byte), and 32-bit (word). (A word can be different sizes, depending which computer you are talking about, but it usually means the largest size of integer that fits in a register. This used to be 16 bits, and nowadays is usually 32 bits. There are examples at http://homepages.cwi.nl/~robertl/mash/128vs32.) To keep things simple, they decided to also always use 32 bits for instructions.

One further simplification was to say that a word cannot consist of any 4 adjacent bytes, but must have an address that is a multiple of 4 i.e. we can access words at locations 0, 4, 8, 12 etc., but not words at locations 1, 2, 3, 5, 6, 7 etc.. (As on most computers, ARM addresses always give the memory location in byte units.)

How are the four bytes that make up a word arranged?

If we put the hexadecimal number 0x12345678 into the word at location 0, and then look at the bytes in locations 0, 1, 2 and 3, which byte will the 0x78 be in? There are two possible conventions:

– byte 0 is the least significant byte of word zero, and holds the 0x78 (known as little-endian)
– byte 3 is the least significant byte of word zero, and holds the 0x78 (known as big-endian)

(The names little-endian and big-endian come from Swift’s Gulliver’s Travels).

ARM computers can be configured to use either convention, but in the labs we will be using little-endian:

<table>
<thead>
<tr>
<th>byte 3</th>
<th>byte 2</th>
<th>byte 1</th>
<th>byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12</td>
<td>0x34</td>
<td>0x56</td>
<td>0x78</td>
</tr>
</tbody>
</table>

How many different bytes and words can an ARM computer access?

If an ARM address is 32 bits, as all addresses are in byte units, then it can access 2^{32} (about 4 billion) different bytes, which is 2^{32}/4 (about 1 billion) different words.

4.1 How are ARM instructions represented in binary?

The rest of this lecture includes lots of details about ARM instructions that you won’t need to remember – however, you ought to become familiar with the kinds of tricks used to encode computer instructions as binary.

SWI

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>1111</td>
<td>action</td>
</tr>
</tbody>
</table>

The first 4 bits have a special purpose, that will be described in a later lecture (section 6.1). The next 4 bits specify the particular operation to be a SWI i.e. ask the system software to do something. The remaining 24 bits are available to specify exactly what we want the system software to do – 24 bits gives us 2^{24} (about 16 million) different possible actions! (The lab simulator only uses five: 0 = output a character, 1 = input a character, 2 = stop, 3 = output a string, 4 = output an integer.)

Branch

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>1010</td>
<td>target location</td>
</tr>
</tbody>
</table>

The first 4 bits specify which condition (e.g. BLT, or BEQ etc.) we are using. For example, a pattern of 1011 indicates a BLT, and 1110 indicates an unconditional branch. 4 bits gives us 2^{4} i.e. 16 different possible conditions, although only 15 of these are actually used for ARM (refer back to the list of branch conditions in section 2.2).

As with the SWI instruction, the next 4 bits specify the particular operation i.e. Branch. The remaining 24 bits allow us to specify where to branch to i.e. any one of 2^{24} different locations – about 16 million locations out of 2^{32} (about 4 billion) different addresses.

What if we want to use one of the other 3 billion, 984 million addresses? We obviously need to try to pick the 2^{24} different locations to be as useful as possible.
We can start by remembering that all instructions occupy a word, and the addresses of words are always a multiple of 4, so we can multiply the 24-bit number from the branch instruction by 4 to get an instruction address – now we can refer to $2^{24}$ different locations out of $2^{30}$ possible different instruction locations – better, but still not very good.

The next thing we can do is realise that the destinations of most branches will be to locations relatively near to the branch instruction itself – if we treat the 24-bit number as an offset from the current value of PC, we should normally be able to reach any instruction we need. Note that the offset is therefore a signed number (encoded using two's-complement notation) so we can branch both forwards and backwards from the current instruction location.

Looking at (part of) the simple program from section 2.2, if the condition is true, the branch instruction needs to move PC on by 4 instructions, so you might expect an offset of 4. However, if you remember what was said in section 2.1 about the fetch-execute cycle, PC will already have been incremented before the branch instruction is executed, so we must reduce the branch offset by 1, to 3. Another way of thinking about this is that we want to skip over 3 instructions. (Because of a quirk of the ARM design, we actually have to reduce the branch offset once more, to 2, but you really don’t need to remember this! Most computer designs use an offset of 3, as described above.)

Load and Store

<table>
<thead>
<tr>
<th>LDR</th>
<th>?</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>R</th>
<th>RD</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>?</td>
<td>0</td>
<td>R</td>
<td>RD</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Just like the instructions above, the first 4 bits have a special purpose, and then 4 more bits specify the operation (in this case, split into two fields, one of 3 bits, and one of 1 bit). The LDR and STR instructions have to also specify a register: to identify a particular one from 16 different possibilities, we need a 4-bit field (RD).

So far, we have used up 12 bits, leaving 20 bits to specify the memory location. As with the branch instruction, we want to be able to access many more locations than these 20 bits would allow by themselves, and we have to use similar tricks to create all the different addresses we might need.

As before, the main trick is to use another register to help create the address. However, as we are specifying data rather than instructions, it is not sufficient to always use PC, so we need another 4-bit field (R) to specify the register we will be using. Also, we want to be able to access values from individual bytes as well as words, so we cannot multiply offsets by 4.

Once we have decided to use a register to help create the address, we will see in future lectures that there are other ways to use it rather than simply adding an offset (and even ways that don’t involve an offset) and these are encoded in some more of the bits of the instruction, leaving 12 bits for the signed byte offset.

For example, if we try to encode the LDR R4,discount instruction in the example code above, the simplest way to encode the address is as an offset from PC (register 15). As we are using PC, we can calculate the offset in a similar way to that for the branch offset, as given above, except that the offset is in byte units (e.g. 20) rather than word units (e.g. 5).
Data-Processing Instructions

We have already met some ARM instructions that perform operations on data, such as ADD and SUB. These use three registers, as with `SUB R5, R2, R4` in the example above, so we need three 4-bit fields in the instruction to specify them. The ARM actually has 16 different data-processing instructions, so it needs a 4-bit “opcode” field to specify the particular operation (e.g. an ADD instruction has an opcode of 0100):

```
4 2 1 4 1 4 4 8 4
? 0 0 0 opcode 1 R RD ? R
```

As you can see, this leaves about 8 bits that we haven’t managed to find a use for. The ARM designers decided that the best way to use these would be to allow one of the operands to be a small number, instead of a register. For example, in the code above, we can try to replace:

```
LDR R4, discount
SUB R5, R2, R4
```

by:

```
LDR R4, discount
SUB R5, R2, #100
```

The “#” means that we want to put the number to subtract, “100”, directly into the instruction doing the subtraction – a literal. (The 7th bit along changes from a “0” to a “1” to indicate that we are using an instruction format that includes a literal, rather than a register.)

Can we do the same thing with the other fixed number, “2000”, and replace:

```
LDR R3, minimum
CMP R2, R3
```

by:

```
LDR R3, minimum
CMP R2,#2000
```

(CMP instructions don’t have a destination register, so the RD field is unused, but otherwise the instruction format is the same.)

The binary for 2000₁₀ is 0111 1101 0000 i.e. 7D₀₁₆ and this seems too big to fit into 8 bits. However, the ARM designers have used clever encodings to squeeze numbers into the available space. The simplest thing to do is just to try it – if the lab system accepts it then ok, otherwise you will have to use something more complicated (you will see more tricks for this in later lectures). In this example #2000 will indeed fit into an instruction.

4.2 Exercises

1. If the word at address 20 contains the hexadecimal value 0x12345678, what is the hexadecimal value in the byte at address 23? (Assume we are using the same convention as in the labs.)
2. What is the ARM encoding for a SWI 2 instruction?
3. Supposing the ARM encoding for a branch instruction had an offset of 0, where would it actually branch to, counting from the branch instruction itself?
4. In the cash-till program, suppose that by mistake we typed BLT discount instead of BLT nodiscount
   Even though this doesn’t make any sense, what would the offset for the instruction be?
   What would happen when we ran the program?
5. In the cash-till program, what is the offset for LDR R3, minimum?
5 Integer Arithmetic

We have seen most of the instructions needed to perform calculations with integers, but we haven’t seen many examples of their use yet. In this lecture, we are going to look at various snippets of Java, and see what ARM code would be equivalent.

It does not really matter which registers we use, as long as we don’t use a register for more than one thing at once. So, we shouldn’t try to use R15 (PC) for arithmetic! We will come across a few more registers with special uses in the lectures, but we don’t need to worry about them for now. I am just going to start with R0 and then use R1 and so on.

However, as we need to use registers for everything, we often try to reuse registers as much as possible, and avoid using them until we absolutely have to. An important part of a modern compiler is an analysis of the program being compiled, to discover the best way of using the registers.

(Why doesn’t ARM provide more registers? Registers add significantly to the complexity and cost of the CPU, and we would also have to find some way of using any extra registers in the instructions – most instructions have no spare bits that could be used to select from more than 16 registers.)

For example (assuming \(a, b, c, d, e\) etc. are integer variables):

\[
a = b + c + d + e;
\]

Variable “\(a\)” will hold the result, so we need a STR instruction for it, and there are 4 variables in the expression (“\(b\)”, “\(c\)”, “\(d\)” and “\(e\)”) so we need 4 LDR instructions for them. The expression consists of 3 “+”s, so we need 3 ADD instructions.

However, by putting these 8 instructions into slightly different orders we can use different numbers of registers.

It could be coded using 7 registers:

```
LDR R0, b
LDR R1, c
LDR R2, d
LDR R3, e
ADD R4, R0, R1 ; b + c
ADD R5, R4, R2 ; + d
ADD R6, R5, R3 ; + e
STR R6, a
```

or using just 2 registers:

```
LDR R0, b
LDR R1, c
ADD R0, R0, R1 ; b + c
ADD R0, R0, R1 ; + d
ADD R0, R0, R1 ; + e
STR R0, a
```

The instruction sequences are very similar, and essentially require the same space and time, so there is no obvious problem caused by using less registers.

However, if we are trying to create the fastest possible code, and so really minimise the use of memory, we have to try and keep variables in registers for as long as possible e.g.: \(a = b + c + d + e; \quad e = a \ast b \ast c \ast d;\)

could be coded using 5 registers:

```
LDR R0, b
LDR R1, c
LDR R2, d
LDR R3, e
ADD R4, R0, R1 ; b + c
ADD R4, R4, R2 ; + d
ADD R4, R4, R3 ; + e
STR R4, a
MUL R3, R4, R0 ; a * b
MUL R3, R3, R1 ; * c
MUL R3, R3, R2 ; * d
STR R3, e
```

or using 2 registers:

```
LDR R0, b
LDR R1, c
ADD R0, R0, R1 ; b + c
ADD R0, R0, R1 ; + d
ADD R0, R0, R1 ; + e
STR R0, a
```

The extra LDR instructions needed to load “\(b\)”, “\(c\)” and “\(d\)” the second time increase the space used (more instructions) and time taken (more instructions to obey, more memory accesses for the variables, more memory accesses for the extra instructions) as we try to reduce the number of registers in use, and this will probably get worse as we look at longer pieces of Java.

With more complex expressions, we may be forced to use an extra register to hold the value of a sub-expression before we can combine it into the final result e.g.: \(a = b \ast c + d \ast e; \quad (We \ need \ to \ keep \ “b*c” \ in \ R0 \ while \ we \ evaluate \ “d*e” \ using \ R1 \ and \ R2.)\)
LDR R0, b
LDR R1, c
MUL R0, R0, R1 ; b * c
LDR R1, d
LDR R2, e
MUL R1, R1, R2 ; d * e
ADD R0, R0, R1 ; +
STR R0, a

We can rearrange an expression, as long as we don’t change its value. However, this is not quite as straightforward as you might imagine. Because of the limited size of a number, it is possible that rearranging an expression in a way that should not cause a difference might do so after all.

For example, if we have three very large numbers, there might be an important difference between \(a - b + c\) and \(a + c - b\). Doing the subtraction first would leave some small difference to be added to \(c\), but doing the addition first could result in a number that is too large to fit in a register. This sort of problem is especially common in calculations involving numbers with fractional parts i.e. floats.

For example:

\[
a - b + c
\]

and

\[
a + c - b
\]

Doing the subtraction first would leave some small difference to be added to \(c\), but doing the addition first could result in a number that is too large to fit in a register. This sort of problem is especially common in calculations involving numbers with fractional parts i.e. floats.

We can rearrange an expression, as long as we don’t change its value. However, this is not quite as straightforward as you might imagine. Because of the limited size of a number, it is possible that rearranging an expression in a way that should not cause a difference might do so after all.

For example, if we have three very large numbers, there might be an important difference between \(a - b + c\) and \(a + c - b\). Doing the subtraction first would leave some small difference to be added to \(c\), but doing the addition first could result in a number that is too large to fit in a register. This sort of problem is especially common in calculations involving numbers with fractional parts i.e. floats.

e.g.: \(a = b - e + c * d\);
Using the “natural” order of evaluation: Assuming we can safely reorder the expression:

LDR R0, b
LDR R1, e
SUB R0, R0, R1 ; b - e
LDR R1, c
LDR R2, d
MUL R1, R1, R2 ; c * d
ADD R0, R0, R1 ; +
STR R0, a

uses 3 registers (R0, R1, R2)

LDR R0, c
LDR R1, d
MUL R0, R0, R1 ; c * d
LDR R1, b
ADD R0, R0, R1 ; b +
LDR R1, e
SUB R0, R0, R1 ; - e
STR R0, a

uses 2 registers (R0, R1)

There is no significant difference in space or time needed, but the reordered sequence uses one less register (R2), which may allow a saving elsewhere.

One final note – ARM does not have a built-in division instruction, so we have to obey many instructions to perform the equivalent calculation. Normally, someone else writes this code for us, and we just make use of it, just like we make use of e.g. `System.out.println` in Java programs. You will see how to invoke a pre-defined method using the BL instruction in a later lecture.

### 5.1 Less important ARM arithmetic instructions

The ARM has a variant of SUB known as **RSB** (Reverse Subtract). So, for example, we can write a SUB instruction:

\[
\text{SUB R0, R1, R2 meaning R0} = \text{R1} - \text{R2}
\]

and this has an exactly corresponding RSB instruction:

\[
\text{RSB R0, R2, R1 meaning R0} = -\text{R2} + \text{R1}
\]

This does not seem to be very useful, until we remember that the final register in most arithmetic instructions, including SUB and RSB, can be replaced by a short literal. (I am using an extra register to make the action of the RSB instructions a little clearer – the next few sequences could all be rewritten to use just one register.)

e.g. \(a = b - 1\); \(\text{e.g. } a = 1 - b\);

LDR R0, b
LDR R0, b

SUB R1, R0, #1 ; R1 = R0 - 1
RSB R1, R0, #1 ; R1 = -R0 + 1

STR R1, a
STR R1, a
We can also use RSB to negate a value e.g. \( a = -b; \)

\[
\begin{align*}
\text{LDR R0, b} \\
\text{RSB R1, R0, #0 ; R1 = R0 + 0} \\
\text{STR R1, a}
\end{align*}
\]

Similarly, the ARM has a variant of CMP known as \text{CMN} \text{ (CoMpare Negative)}.

\[
\begin{align*}
e.g. \quad \text{if} \ (a < 4) \ldots & \quad \text{e.g. if} \ (a < -4) \ldots \nonumber \\
\text{LDR R0, a} & \quad \text{LDR R0, a} \\
\text{CMP R0, #4 ; R0 - 4} & \quad \text{CMN R0, #4 ; R0 + 4 i.e. if} \ (a + 4 < 0) \ldots \\
\text{BGE skip} & \quad \text{BGE skip}
\end{align*}
\]

Of the different arithmetic instructions we have used so far, we can use short literals with ADD, SUB, RSB, CMP and CMN but not with MUL, so how do we deal with e.g. \( a = b \times 6; \)

One answer is to use the \text{MOV} \text{ (MOVe) instruction.} This allows us to move a value:

\[
\begin{align*}
\text{from one register to another} & \quad \text{or from a short literal to a register} \\
\text{e.g. (silly code for} \ a = b;) & \quad \text{e.g.} \ a = b \times 6; \\
\text{LDR R0, b} & \quad \text{LDR R0, b} \\
\text{MOV R1, R0} & \quad \text{MOV R1, #6} \\
\text{STR R1, a} & \quad \text{MUL R0, R0, R1} \\
& \quad \text{STR R0, a}
\end{align*}
\]

The ARM also has a variant of MOV known as \text{MVN} – unfortunately, not move negative, but MoVe Not. This still allows us to use small negative numbers, but they are encoded in a slightly strange way (don’t worry about the details for now).

\[
\begin{align*}
e.g. \ a = -1; & \quad \text{e.g.} \ a = b \times -2; \\
\text{MVN R0, #0} & \quad \text{LDR R0, b} \\
\text{STR R0, a} & \quad \text{MVN R1, #1} \\
& \quad \text{MUL R0, R0, R1} \\
& \quad \text{STR R0, a}
\end{align*}
\]

There is one more ARM instruction that may be used in integer arithmetic: \text{MLA} \text{ (MuLtiply and Add)}. This performs a multiply and an add in a single instruction but, like MUL, can only use registers as operands – in the case of MLA, 4 of them!

\[
\text{e.g.} \quad \text{MLA R0, R1, R2, R3 means} \ R0 = (R1 \times R2) + R3
\]

Looking back at a previous example from this lecture: \( a = b \times c + d \times e; \)

\[
\begin{align*}
\text{LDR R0, b} & \quad \text{LDR R0, b} \\
\text{LDR R1, c} & \quad \text{LDR R1, c} \\
\text{MUL R0, R0, R1 ; b \times c} & \quad \text{MUL R0, R0, R1 ; b \times c} \\
\text{LDR R1, d} & \quad \text{LDR R1, d} \\
\text{LDR R2, e} & \quad \text{LDR R2, e} \\
\text{MUL R1, R1, R2 ; d \times e} & \quad \text{MLA R0, R1, R2, R0 ; } + d \times e \\
\text{ADD R0, R0, R1 ; } + & \quad \text{STR R0, a} \\
\text{STR R0, a} & \quad \text{STR R0, a}
\end{align*}
\]

### 5.2 Exercises

For all questions, assume \( a, b, c, d, e \) etc. are integer variables, and try to your make code as efficient as you can.

1. Give ARM code for: \( a = (b + c) \times (d + e) \times (b + d); \)
2. Give ARM code for \( a = (b \times 2) + (c \times -2); \)
3. Give ARM code for \( a = (2 - b) \times (c - 2); \)
4. Give ARM code for \( a = -2 \times (b + 2); \)
6 If and While

We have already seen a simple example of an if statement in the Cash-till program. We started with:

```java
if total ≥ £20 then
deduct £1 from total
which, in Java, would be:
if (total >= 2000) {
total = total - 100;
}
```

and then rewrote it until it was in a simple enough form for the computer to understand. We ended up with something like this:

Thus, we can translate any Java if-statement:

```java
if (condition) {
 action;
}
```

into a pattern of ARM instructions:

```assembly
LDR R0, a
LDR R1, b
ADD R0, R0, R1 ; R0 = a + b
LDR R1, c
LDR R2, d
MUL R1, R1, R2 ; R1 = c * d
CMP R0, R1
BLT skip ; not >=
LDR R0, f
STR R0, e
skip . . .
```

To evaluate the condition, we need to evaluate any arithmetic expressions, and then use a compare instruction followed by a conditional branch instruction, remembering to logically invert the condition in the branch. For example,

```java
if (a == b) would become BNE skip,
if (a < b) would become BGE skip etc.
```

Thus:

```java
if (a + b >= c * d) {
  e = f;
}
```

would become:

We can construct a similar pattern for a Java if-else-statement:

```java
if (condition) {
  action;
} else {
  alternative;
}
```
Thus:
if (a == b) {
    c = d;
} else {
    e = f;
}
would become:

We can do something very similar for a Java
while-statement:

while (condition) {
    action;
}

Thus:
while (a < b) {
    a = a * 2;
}
would become:

As it is a loop that we are translating into ARM code, we
can reasonably guess that the code will be obeyed many
times, and therefore it is worth putting extra effort into
improving the code.
For example, we could move the test. We have the same
number of lines of code, but we are obeying 1 less in-
struction each time around the loop (we have got rid of
one of the two branch instructions) at the cost of obey-
ing 1 extra branch instruction right at the start. This
is a common trick, but we won’t pursue this any further
here. However, in the next section, we will find another
way of saving one of the branch instructions each time
around the loop.

A much more effective technique is to load all needed
values into registers at the start of the loop, and save any
results back into memory at the end. In this example,
we can reduce the number of instructions obeyed each
time around the loop from 8 to 4.

(We can avoid using the extra register, R2, by noticing that a = a * 2; is the same as a = a + a; and replacing the
multiply instruction by ADD R0, R0, R0)
6.1 Conditional Instructions

There are other optimisations possible using ARM instructions, using facilities that we have not looked at before. The first of these is that every ARM instruction can be made conditional. This is a generalisation of the way that the unconditional branch instruction, B, can be made conditional using BEQ, BNE, BLT etc. In fact, every ARM instruction can use the same set of conditions. (The condition is encoded using the first 4 bits of every instruction, as hinted at in lecture 4.)

For example, in the loop above, we can reduce the number of instruction obeyed each time around the loop by one more by using a conditional multiplication, at the cost of 1 extra instruction obeyed when the loop finishes.

Here is another example – a loop that finds the greatest common divisor (GCD) of two numbers a and b:

```assembly
while (a != b) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
```

A simple translation that only uses conditional branches but that keeps the variables in registers obeys 6 or 7 instructions each time around the loop:

```assembly
LDR R0, a ; keep a in R0
LDR R1, b ; and b in R1
start CMP R0, R1 ; test loop condition
    BEQ skip ; test if condition
BLE else ; test if condition
    SUB R0, R0, R1 ; then a = a - b
    B both
else SUB R1, R1, R0 ; else b = b - a
    both B start ; and repeat loop
skip STR R0, a ; store a
STR R1, b ; and b
```

An improved translation merges some of the code the while statement with the code for the if-else statement: it reuses the result of the comparison, and it avoids a branch to another branch instruction. However, it still only uses conditional branches. It obeys 5 instructions each time around the loop:

```assembly
LDR R0, a ; keep a in R0
LDR R1, b ; and b in R1
start CMP R0, R1 ;
    BEQ skip ; test loop condition
BLE else ; test if condition
    SUB R0, R0, R1 ; then a = a - b
    B start
else SUB R1, R1, R0
    B start
skip STR R0, a
STR R1, b
```

A translation that uses conditional arithmetic instructions as well as conditional branches obeys one less instruction each time around. It uses the fact that, because of the test at the start of the loop, a can not be equal to b within the if-else statement.

```assembly
LDR R0, a ; keep a in R0
LDR R1, b ; and b in R1
start CMP R0, R1
    SUBGT R0, R0, R1 ; if a>b a= a-b
    SUBLT R1, R1, R0 ; if a<b b= b-a
    BNE start ; if a!=b then repeat
STR R0, a
STR R1, b
```

Note: on most computers the only conditional instructions are conditional branches – other instructions are always unconditionally executed.
6.2 Avoiding CMP instructions

So far, whenever we have wanted to test a condition, we have used a CMP instruction. For example, a simple translation of this loop (that finds the largest power of 2 that a Java integer can hold):

```java
a = 1;
while (a * 2 > 0) {
    a = a * 2;
}
```

In fact, there is another optimisation possible using ARM instructions – we can sometimes avoid using a CMP instruction, and instead use the result of an arithmetic instruction, by appending an S to the operation:

```assembly
MOV R0, #1 ; keep a in R0
MOV R1, #2 ; keep 2 in R1
start MUL R2, R0, R1
CMP R2, #0 ; while (a * 2 > 0)
BLE skip
MOV R0, R2 ; a = a * 2
B start ; and repeat loop
```

We can further speed up the code by using conditional instructions:

```assembly
MOV R0, #1 ; keep a in R0
MOV R1, #2 ; keep 2 in R1
start MULS R2, R0, R1 ; while (a * 2 > 0)
BLE skip
MOV R0, R2 ; a = a * 2
B start ; and repeat loop
```

(A clue that this trick might be useful, in this example, is that we were comparing a register with zero, when we had just calculated a value in the register.)

Note: on most computers, every arithmetic instruction or load instruction automatically tests its result, and it is not possible to turn this off (i.e. the instructions always behave as if they have a “S”).

6.3 Exercises

For all questions, assume a, b, c, etc. are integer variables.

1: Give ARM code for:

```assembly
if (a == b * c)
    a = b;
else
    b = b - 1;
```

2: Assuming that 300, 500, 100, 150 all fit into 8-bit literals, give ARM code for:

```assembly
if (people > 300)
    watts = 500;
else if (people > 100)
    watts = 150;
else
    watts = normal;
```

3: Without using conditional arithmetic instructions, give ARM code for:

```assembly
while (side * side * side < vol)
    side = side + 1;
```

4: Give the best ARM code that you can, using conditional arithmetic etc., for the loop in the previous question.