# Introduction to Assembler

## Table of Contents

# Introduction

These notes are an introduction to assembly language programming, on Linux-based 64bit systems.

This is just an outline. AMD's manual on their 64bit processors comes in 5 volumes.  The first volume is 350 pages long, and volume 3 is 672 pages. But there are good reasons for not looking at assembler in full detail.

## Why use assembler?

There are many different programming languages. They work at different levels. Some are very close to the hardware, while others are more abstract - we say they are 'high level languages'. Java is high level. C is less high level. Assembler is almost as low as you can get - it is very close to the actual hardware. That means it is good if you want to know how the devices actually work.

Assembler code (if correctly written) is also very fast.

## Why not assembler?

When writing assembler you must think about the hardware. But that stops you thinking about the programming problem - the task the program is supposed to carry out. For this reason, practical software development is usually not written in assembler.

Assembler is also hardware specific. So assembler code needs to be altered to run on a different processor. This is not true of C (which just has to be re-compiled for another system). Java code runs on many different platforms (hardware and operating systems) without any change at all.

The only area where assembler is widely used is in very low-level situations where interaction with the hardware is important - such as in device drivers.

We refer in places to C programming. You might miss those out. Or learn C ;-)

## Write code

You need to work through this at a keyboard, try things out, and write your own code - *small* variations on that given.

## Linux and x86-64

We talk about general ideas, but we also have concrete examples or actual programming to make it more understandable. For these we use the Linux OS (because it is open source and so more accessible than Windows, and has suitable applications such as a compiler as standard), and the x86-64 architecture, since it is common on desktops and laptops.

## Computer architecture

This involves:

- Processor design and operation
- Assembly language programming
- Operating system (OS) concepts

Current versions of these ideas are very sophisticated. Most of the text uses a simplified version of how all this operates. The final chapter outlines some more realistic concepts.

## Further Reference

AMD processor manuals : http://developer.amd.com/resources/developer-guides-manuals/

The GCC compiler: https://gcc.gnu.org/onlinedocs/

GAS, the GNU assembler: https://sourceware.org/binutils/docs/as/

ld, the linker: https://sourceware.org/binutils/docs/ld/

Linux kernel links : https://www.kernel.org/doc/

comp.lang.c FAQ : http://c-faq.com/

# Binary and hex

This section is about binary, decimal and hexadecimal. Skip it if you know about this already.

A bit pattern is something like 0011 1010 0011 0000 - a pattern of 1s and 0s. All software in a digital system is made of bit patterns. That means all program code and all data. The data is might be numbers - these are bit patterns. Or the data might be an image, made of pixels, made of bit patterns. Or audio. Or text. All bit patterns.

Bit patterns are often grouped into sections of 8 bits, known as a *byte*. So 0011 1010 0011 0000 is 2 bytes long.

Program code includes source code, which is a sequence of characters, and this is a sequence of bit patterns. So is native machine code.

To be precise, everything is a set of states of two-state devices. RAM consists of electronic switches which are either open or closed, giving two different voltage levels. A standard hard disc stored data as magnetic patterns, magnetised in one direction or the other. A CD has a spiral of short or long pits on a shiny layer. For each of these, we interpret the two states as being a 1 or a 0 ( or a yes or no, or true or false). When the CD is read, the pits are converted voltage levels - but the bit pattern is the same.

If we have a pattern like 0011 1010 0011 0000 - is that a machine code instruction, or a number, or a pixel, or text? Could be any. They are all represented by bit patterns, so they all look the same. Only the context tells us if we should treat these as pixels or text or whatever.

We talk about the 'most significant bits' or 'leading bits' or 'upper bits' in a bit pattern as being the left-most ones, and the least significant bits as the right-hand ones.

## Number Bases

A *bit* is a binary digit - a 0 or a 1. These are the digits in 'base 2'. So we need to look at number bases first.

Get the idea of the difference between a *number* and the way it is *represented*. The number 3 can be represented as III in Roman numerals, or 11 in base 2, or 3 in base 10. III and 11 and 3 are three different representations of the same number.

A *digit* is one symbol - like 3 or 7 or 9. A *number* is made of one or more digits - like 12.34.

### Base 10

'Ordinary' numbers are written in *base 10* (or decimal or denary). That means we use a set of digits, with different places having different *place values*, like this:

| Place value | Thousands | Hundreds | Tens | Units |
|---|---|---|---|---|
| | $10^3$ | $10^2$ | $10^1$ | $10^0$ |
| Digits | 3 | 2 | 4 | 6 |

So the number 3246 means 3 thousands, 2 hundreds, 4 tens and 6 units.

Notice the place values go up in powers of 10, and we have 10 digits to use - 0 1 2 3 4 5 6 7 8 and 9.

The pattern continues to the right of the 'decimal point', like this:

| Place value | Tens | Units | Tenths | Hundredths |
|---|---|---|---|---|
| | $10^1$ | $10^0$ | $10^{-1}$ | $10^{-2}$ |
| Digits | 7 | 2 | . 1 | 6 |

So 72.16 is 7 tens, 2 units, 1 tenth and 6 hundredths.

## Base 2

We can use the same idea in other bases. In base 2 the place values are powers of 2 - units, 2, 4 8 16 and so on. We only have the digits 0 and 1 to use:

| Place value | Eights | Fours | Twos | Units |
|---|---|---|---|---|
| | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| Digits | 1 | 1 | 0 | 1 |

So 1101 is 1 eight, 1 four, 0 twos and 1 unit = 8 + 4 + 1 = 13.

When we are writing in different number bases, we use a subscript to show which base it is in. So $1101_2 = 13_{10}$

Here is counting in base 2:

| Base 2 | Base 10 |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 10 | 2 |
| 11 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | 10 |

To change a number from base 2 to decimal, just add up the place values where there is a 1 - for example:
$10010_2 = 16 + 2 = 20_{10}$

To change decimal to binary, for small numbers, pick out powers of 2 to total the number. For example

$19_{10} = 16 + 3 = 16 + 2 + 1 = 10011_2$

For larger numbers, repeatedly divide by 2 and track the remainders. For example to

change 328 into binary:

2 into 328 = 164 remainder 0
2 into 164 = 82 remainder 0
2 into 82 = 41 remainder 0
2 into 41 = 20 remainder 1
2 into 20 = 10 remainder 0
2 into 10 = 5 remainder 0
2 into 5 = 2 remainder 1
2 into 2 = 1 remainder 0
so reading the remainders back:

$328_{10} = 101001000_2$

In effect we are finding powers of 2 as before but more methodically. In practice it is faster and safer to do this using a calculator in appropriate mode.

| Exercise |
| --- |
| 1. Change binary 11001 into decimal<br>2. Change 12 into binary<br>3. Change 421 into binary<br>4. How can you tell if a binary number is even, by looking at it? |

*To convert a decimal fraction to binary:*
1. Multiply by 2
2. The whole number part is the next bit. Ignore it when you..
3. Repeat step 1, until you get to zero, or a repeating pattern.

Example: 0.75 - multiply by 2:

1.50 First bit is 1. Now use 0.50

1.00 Second bit is 1, Got to .00, so end

So $0.75_{10}=0.11_2$ ( since 0.75 = 1/2 + 1/4 )

Example 0.625

1.250 first bit 1, use .250

0.50 second bit 0

1.00 third bit 1, ended

So $0.625_{10}=0.101_2$
Example 0.1

0.2 first bit 0
0.4
0.8
1.6

1.2 (but we've had .2 before..)
0.4
0.8
1.6
1.2
0.4
0.8
1.6
1.2 ..
so 0.1 decimal = binary .0001100110011001100110011..
So 0.1 is an example to show that some values have infinite binary fraction expansions - just like 1/3 is 0.3333.. in base 10. We will see later that computer arithmetic with numbers other than integers has only limited accuracy.

| Exercise |
| --- |
| 1. Change decimal 0.125 into binary |

## Hexadecimal

We often use numbers in base 16, known as hexadecimal or hex. Here the place values are powers of 16. For example:

| Place value | 4096's | 256's | Sixteens | Units |
| --- | --- | --- | --- | --- |
| | $16^3$ | $16^2$ | $16^1$ | $16^0$ |
| Digits | 2 | 0 | 7 | 1 |

So $2071_{16}$ = 2 X 4096 + 7 X 16 + 1 X 1 = $8305_{10}$

The big difference is that we need 16 different digits, and we only have 10, as 0 to 9. The problem is solved by using A to F as 10 to 15. So counting in hex (and binary) is like this:

| Hex | Decimal | Binary |
| --- | --- | --- |
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |

| | | |
|---|---|---|
| E | 14 | 1110 |
| F | 15 | 1111 |
| 10 | 16 | 00010000 |
| 11 | 17 | 00010001 |

The reason that hex is used so often is that it is very easy to convert from binary to hex, but hex notation is much shorter than binary. You change hex to binary by replacing each hex digit by the corresponding 4 bits - like

$A2B3_{16}$ = 1010 0010 1011 $0011_2$

and in reverse for binary to hex. Most current processors are 64 bit, meaning they handle 64 bits or 8 bytes at  a time. So a value a processor might process might be

1111 1010 1000 0011 0000 1110 1111 1010 1000 0101 1100 0011 1111 1010 1000 0011

which is almost impossible to write down without making a mistake. The corresponding hex version is not quite so bad:

F A 8 3 0 E F A 8 5 C 3 F A 8 3

One byte is 2 hex digits or 8 bits.

Exercise

1. Change binary 1001 into hex
2. Change $AF_{16}$ into binary
3. Change $12_{16}$ into decimal
4. Change 1100 0011 into hex

## Octal

This is base 8 - using digits 0 to 7:

| Octal | Decimal | Binary |
|---|---|---|
| 0 | 0 | 000 |
| 1 | 1 | 001 |
| 2 | 2 | 010 |
| 3 | 3 | 011 |
| 4 | 4 | 100 |
| 5 | 5 | 101 |
| 6 | 6 | 110 |
| 7 | 7 | 111 |

Each octal digit is 3 bits - so they do not fit into the 8 bits of a byte, so octal is not often used.

# Basic Concepts

## Digital devices

A digital device is a desktop PC, a laptop, a notebook, or a tablet. This also includes things which are not 'computers' - such as smart phones and smart TVs. All these devices have a processor and memory, and represent code and data in binary patterns - patterns of 1 and 0, on or off, yes or no. To keep it brief, we will call all these things 'computers'.

## Binary patterns

Computers hold programs and data. Both of these are represented as patterns which are made of binary, with two states. We can call these two states 0 and 1, true and false, on or off, yes or no. These two states take different forms:

In memory and in processors, they are electronic switches which are open or closed. They need power to 'remember' the state (they are volatile) and lose everything when switched off.

In hard drives, the two states are patterns of magnetisation north-south or south-north

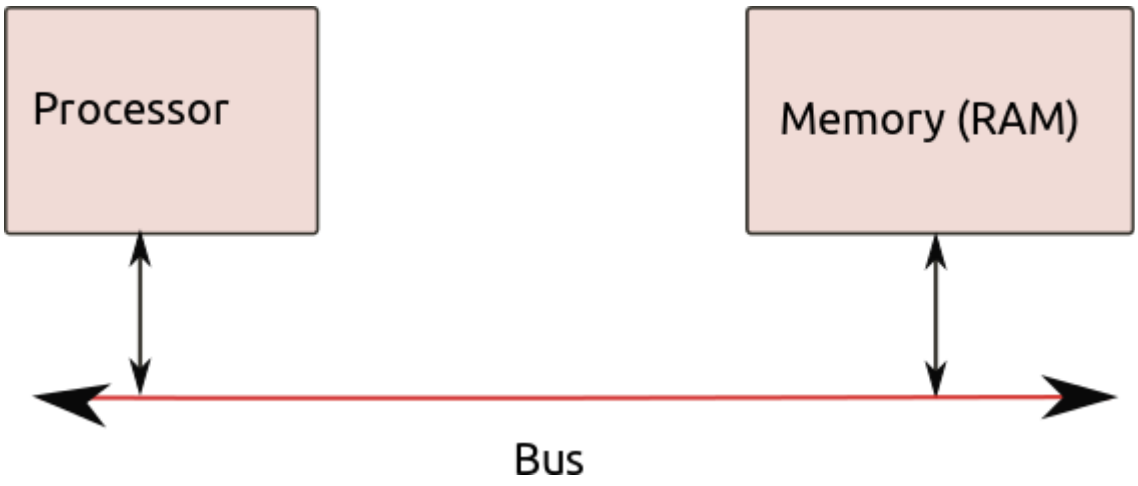
In flash memory, they are non-volatile switches - they keep the state without power.

On a CD or DVD, they are pits or no pits on a shiny surface.

On a cable, they are two different voltage levels.

We will call the states 0 and 1. So 1100 1010 is a binary pattern.

## How computers work




A computer has:

A processor. This is a chip, which can recognise and execute program instructions. These

instructions are mostly very simple - like to add together two numbers.

Memory - usually volatile RAM. The memory holds binary patterns, representing program instructions and data.

A bus - which is a set of wires over which binary patterns can be transferred in either direction. In effect the bust is a connecting cable.

This model is very simplified. RAM and processor may be on the same chip. There might be more than one bus. There may be a separate graphics processor. There must be some way of doing input/output - like a keyboard or a graphics output.  But the simple picture will do for now.

So long as power is applied, the *fetch-execute cycle* runs, as follows:

The next program instruction is fetched, out of memory, and transferred over the bus to the processor.

The processor decodes it (works out which instruction it is) and executes it (maybe adding two numbers)

This cycle then repeats - get the next instruction and so on.

This is a *simplification*. Several instructions might be fetched at the same time. Multi-core processors can execute several instructions at once. Processors have cache memory. And more.  But this is the basic idea.

## Registers

A register can hold data - a binary pattern, - inside the processor. So a register is like memory - but much faster.

Some registers are general purpose. An AMD 64bit processor has 16 general purpose registers (66 registers in total).

An important special purpose register is the *Instruction pointer*. This holds the address in memory of where the next instruction is. This is how the system tracks which instruction is 'next'.

Another register is the *flags register* . This holds a set of bits (0 or 1) which are items of information about the result of the last instruction, such as if it was positive or not.

## OS and application program

There are two kinds of software:

The *operating system*. This provides facilities like user log ons and security, access to the file system, input through keyboard and so on.

*Application programs* - such as a word-processor.

We will be writing application programs. These will use instructions for processing data in memory. We will also use the operating system, for example for output. In our case the

operating system is the Linux kernel.

## Compilers

A *compiler* is software which inputs a file which is a program, and it outputs another file, which is the same program translated to a different language.

The input is called the *source code* and the output is the *object code*.

Usually (but not always) the source code is in a high level language and the object code is in a low level language.

## Linkers and object files

In a project we often want to use extra program code, maybe from other projects, or from 'libraries' of useful code.

Because of this the output from the compiler is in an intermediate form, called an *object code* format. Some software called a *linker* can take several object code files, join them together and output an executable file.

A *executable file* can be loaded and run by the OS. On Windows systems this is a file with the extension .exe. On Linux there is no standard extension.

The format of an executable depends on the OS. So a Windows .exe will not run on a Linux system.

## Machine code and assembler

*Machine code* is how program instructions are represented in binary. So for example

0100 1000 0000 0001 1100 0011

is the instruction to add the number in register RBX into register RAX

Programming in actual machine is impractical and pointless - if you tried to remember hundreds of patterns of 64 bits, there would be countless errors.

An example of an *assembly language* instruction is

add %rbx, %rax

This means to add register RBX into RAX - same as above, but it makes much more sense.

Assembly language instructions are one-to-one with machine code

In other words, each assembly language instruction corresponds with just one machine code instruction.

An *assembler* is a type of compiler, which translates assembler code into machine code. Because they are one-to-one, this is pretty simple (compared to most compilers).

## Using a terminal

In the next section we start some practical work, and this will be at the command line. Typical computer use is through a GUI, so this may be unfamiliar, so this is a few brief notes.

The word 'terminal' is historical. Old mainframes were not used interactively - programs were input on punched cards and output was on a printer. The early interactive systems provided a keyboard and screen (initially a teletype) for each user. Modern devices normally use a graphical windowing interface - and simulate old terminals in a 'console'. This is convenient for assemply language programming.

CTRL-ALT t will open a terminal.

The terminal use a *shell* - a *command line interpreter*. This is piece of software which

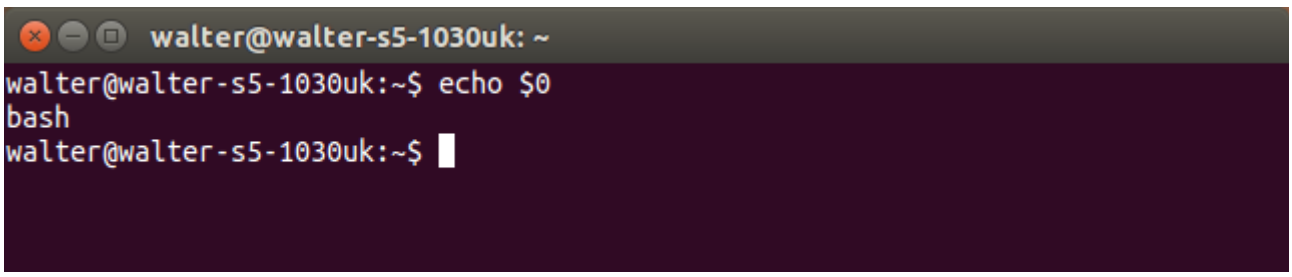
1. Inputs a line from the user

2. Executes it if it is a recognised command

3. If not, it is a program, which it looks for, loads and executes it

4. Loops to step 1 until the shell is closed.

There are different shells. echo $0 will tell you what you are currently using:




bash is the default shell on Ubuntu.

The basic bash commands are

| | |
|---|---|
| ls | lists the names of files and folders in the current folder |
| ls *.c | uses a wildcard - lists all files ending .c |
| cd | change directory (folder). Names are case-sensitive, so to change to the Documents folder, it must be cd Documents. The name is relative - so Documents would need to be a sub-folder in the current folder.<br><br>cd Documents/assembler would change to a folder named assembler, in the |

| | |
|---|---|
| | folder named Documents |
| cd .. | Change to the parent folder (go up a level). The .. means the parent directory. A single dot, like . means the current directory. There must be a space between cd and .. |
| rm | Removes (deletes) a file. Use with care! It does not go to the Trash folder - it is deleted, and not recoverable by normal means. |
| rmdir | Remove a directory (which must be empty) |
| mkdir | Make a directory |
| man x | display the *manual page* documenting command x |
| cat x | display the file x on standard output - normally the screen. This will only make sense if x is a text file |
| ./myProg | Load and run the executable file named myProg in the current directory (which is ./) |

We will often be repeating the same commands. Up and down arrow go through previous commands, to save re-typing them.

## Practical example

We illustrate some ideas with a concrete example.

Use a text editor (gedit or geany or kate or nano) to enter the following program (we explain what this all means later).

```
# A simple assembler example
.text
.global _start

_start:
      mov $4, %rax            # put 4 in register rax
      add $5, %rax            # add 5 into rax
      # and exit
      mov    $60,%rax          # exit code 0
      mov    $0, %edi          # system call number (sys_exit)
      syscall                  # call OS kernel
.data
      .ascii "ABCD"
```

Save it with the file name ass.s ( s for source code)

We assemble it with the command

```
as -o ass.o ass.s
```

This runs the assembly ( as ) on the source code (ass.s) producing object code output ( -o ass.o )

Then we link this with the command

```
ld -o myProg ass.o
```

This runs the linker ( ld ) on the object file ass.o, producing the output file name myProg.

We can run myProg by

```
./myProg
```

- but it should produce no output.

If you get error messages when assembling or linking - check carefully, correct, save and repeat.

We can use objdump to look inside the object code file, ass.o

```
objdump -S -s -d ass.o
```

This produces the following:

```
ass.o:     file format elf64-x86-64

Contents of section .text:
 0000 48c7c004 00000048 83c00548 c7c03c00  H......H...H..<.
 0010 0000bf00 0000000f 05                 .........
Contents of section .data:
 0000 41424344                             ABCD

Disassembly of section .text:

0000000000000000 <_start>:
   0: 48 c7 c0 04 00 00 00    mov    $0x4,%rax
   7: 48 83 c0 05             add    $0x5,%rax
   b: 48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  12: bf 00 00 00 00          mov    $0x0,%edi
  17: 0f 05                   syscall
```

We go through this line by line:

```
ass.o:     file format elf64-x86-64
```

so this file is in *ELF format*, which stands for Executable and Linking Format. This is the standard format on Linux systems. There are different versions for different hardware - this is for 64 bit versions of the x86 architecture.

```
 Contents of section .text:
```

The file has two sections, .text and .data. The .text section is program code, .data is data. When turned into an executable, the data section will be initialised with “ABCD”

```
Contents of section .text:
 0000 48c7c004 00000048 83c00548 c7c03c00  H......H...H..<.
 0010 0000bf00 0000000f 05                 .........
```

This is the contents of the text section (which in fact is *machine code*). The red numbers are addresses in hex. So the second row starts at address 16  (hex 10) (these are addresses relative to the start at 0)

The green numbers are the actual contents, in hex. They are shown in 4 byte blocks - 66b80400 is 4 bytes long.

The blue text is the same, but in ASCII. Since this is not text, there is nothing meaningful here.

```
Contents of section .data:
 0000 41424344                             ABCD
```

Similarly this is the data section. This actually is text, so we see ABCD. The first byte is 41 hex = 65, the ASCII code of A.

```
Disassembly of section .text:
```

Disassembly is the opposite of assembly - taking machine code and producing assembler.

```
Disassembly of section .text:

0000000000000000 <_start>:
   0: 48 c7 c0 04 00 00 00    mov    $0x4,%rax
   7: 48 83 c0 05             add    $0x5,%rax
   b: 48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  12: bf 00 00 00 00          mov    $0x0,%edi
  17: 0f 05                   syscall
```

Red is the address (actually the distance from start at 0).

Green is the assembler instruction.

Black is the machine code

We take one example:

```
 0:   48 c7 c0 04 00 00 00    mov    $0x4,%rax
```

The instruction is 7 bytes long.

The assembler version has a *mnemonic*, which here is mov. Mnemonic is easy to remember - mov means move data.

The instruction has two *operands* - $0x4 and %rax . The mnemonic says what to do - move - and the operands are what to do it with - here to move 4 into the rax register.

This assembler instruction turns into the machine code:

```
  48 c7 c0 04 00 00 00
```

The 48 c7 c0 is an *op-code* : the code for the operation (moving into rax)

rax is a 64 bit register. The value we are storing there are the 32 bits, in hex, 00 00 00 04. In machine code this is 04 00 00 00 - because the AMD64 is *little-endian*. This is about how multi-byte values ( like 00 00 00 04 ) are stored in memory. little-endian means the least significant byte is stored 'first' - at the lowest memory address and the most significant byte (00) is stored at the higher memory address.

## Directives

Most of the contents of an assembler program are program instructions, which will be translated to machine code form.

But some things are *directives*. These give instructions to the assembler, controlling what it does, rather than being converted into machine code..

For example .ascii tells the assembler to convert the following string into ASCII and store at this location.

So

.ascii  “ABCD”

means the ASCII codes for ABCD will be placed into memory at this point. This allows us to preset some data values in memory for the program to use.

# Hello World

A program to output 'Hello world' is traditionally used to introduce a new language. We use it to introduce some more features.

## Tools needed

We need a text editor, to write the assembler code. Anything will do - on Linux, nano or gedit or geany or kate are fine.

We need an assembler. Some common options are

GAS - Windows and Linux

MASM - for Windows, from Microsoft

NASM - Windows or Linux

TASM - Windows and DOS, very old

We will use GAS, the GNU Assembler, which is a Linux standard, and is invoked with 'as'.

The choice of assembler makes a difference - different assemblers use different syntax. The source code here uses AT&T syntax, which is what gas expects.

We also need a linker. We use ld, the GNU linker

## Hello world

Copy and paste this into your text editor:

```
.text
    .global _start
_start:
        mov    $len, %rdx
        mov    $msg, %rsi
        mov    $1,%rdi
        mov    $1,%rax          # system call number (sys_write)
        syscall
        # and exit
      mov    $60,%rax         # exit code
      mov    $0, %edi         # return 0
      syscall                # call OS kernel
.data
msg:
      .ascii    "Hello world\n"
      len = . - msg
```

Save it, calling it hello.s

Assemble it by

as -o hello.o hello.s

(This invokes the assembler, as, on the source code hello.s, producing the object file hello.o)

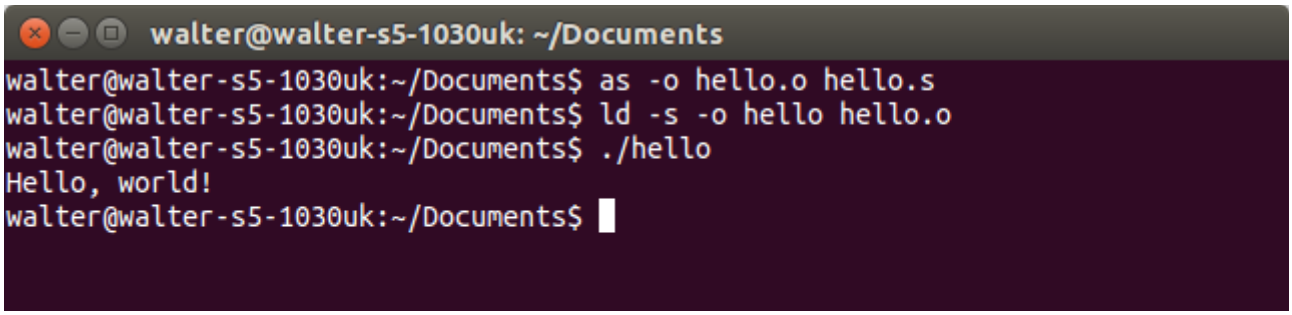
Then use the linker

ld -s -o hello hello.o

Then load and run the executable file, named hello, by

./hello

where the ./ means in this directory. So:




## Hello world line by line

The final executable file has several sections. These are as follows

.text is executable code

.data is data initialised at compile-time, with read-write access

.rodata is initialised with read-only access

.bss uninitialised data with read write access

So the first line,.text, tells the assembler this is code, and the later .data tells it this is initialized read/write data.

_start: is a *label*. As the assembler converts the code to machine code, it can track what address it will be at (relative to the start). A label is a symbolic address - a word, which the assembler knows the actual numerical address.

But _start is a special symbol - the linker will treat this as where to start execution.

.global _start exposes this value to the linker.

Then the code moves towards outputting the string. This output is actually done by the

OS, and we make a call to the Linux kernel:

```
syscall
```

However we need to tell Linux what we want it to do. This is done by passing the code for which service in the rax register. The value 1 means write a string to the output, so we have

```
mov    $1,%rax
```

We also need to tell it where the string to output is, which is expected to be in the rsi register, so

```
 mov    $msg, %rsi
```

(more on this in a moment)

and also how long the string is, in the rdx register:

```
  mov    $len, %rdx
```

The value in ebx is which file handle to write it to. 1 is stdout : standard out, the console:

```
        mov    $1,%rdi
```

The next three instructions are about exiting this program and returning to the OS. So

```
      mov    $60,%rax          # exit code
      mov    $0, %edi          # return 0
      syscall                 # call OS kernel
```

This does the same as System.exit(0) in Java. The value in register edi is the exit code - 0 means no error.

Then we have the data section:

```
.data

msg:
      .ascii    "Hello, world!\n"
      len = . - msg
```

msg: is a label. This is a symbolic address. The assembler works out the actual address, and it uses it in

```
       mov    $msg, %rsi
```

The .ascii directive tells the assembler to convert the following string to ASCII codes, when it places this data in memory.

We also need the length of this string. We could try to count the character (maybe 14) but it is easier to say

```
      len = . - msg
```

Here . means the current address, and $msg is the start address of the string, so .--msg is the length of the string, which is used in

```
  mov    $len, %rdx
```

## x86-64 architecture

Early processors such as the Z80 and 6502 were 8 bit - which meant they had registers holding 8 bits, and used 8 bit units of data.

The Intel 8086 processor was used in the IBM PC, which became a standard for desktop computers. The 8086 had 4 16 bit general purpose registers, called ax, bx, cx and dx. Each of these could be used as 2 8bit halves. So for example ah and al were the high and low 8 bits of ax. This was the start of the x86 architecture.

This was developed into the 80186 and 80286, then the 80386, which was 32bit processor. The 4 registers were extended to 32 bits, giving the eax, ebx, ecx and edx registers. At each stage of this development it was commercially important that existing software would continue to execute on it - so for example even with the 32 bit registers it was possible to access the al and ah 8 bit sections.

This was followed by the 486 and then the Pentium.

In 2000 AMD released their specification of a 64 bit development of the x86 processors, and this architecture has been produced by AMD, Intel and VIA in numerous families of processors.

The original general purpose registers are further extended to 64 bits, and called rax rbx rcx and rdx. The other registers re also extended to 64 bits, as rbp, rsp, rsi and rdi. There are an additional 8 64 bit registers, called r8 to r15. These allow more local variables to be held in registers instead of memory, making execution faster.

There are also 128 bit XMM registers, and floating point instruction - see the chapter on floating point arithmetic.

These processors can operate in different 'modes', related to compatibility with older software - these are 64 bit mode, compatibility mode, protected mode virtual 8086 mode and real mode. We only use 64 bit mode, which has 64 bit addresses and defaults to 32 bit operands.

## Data length terms

The AMD manuals take a 'word' to be 16 bits:

1 byte = 8 bits

1 word = 2 bytes = 16 bits

1 double word = 32 bits

1 quad word = 64 bits

## AT&T assembler syntax

The machine code instructions are fixed for the x86-64 architecture.

But the syntax for assembly language depends on which assembler we use.

We are using gas, the GNU assembler, and it uses the same syntax as the AT&T assembler.

Immediate addressing uses $, and registers start %
So for example

mov $33, %rax

moves the value 33 into register rax (64 bits).

We can write numbers in decimal, hex or octal:

$12 (decimal) is binary 0000 1100

$0x12 (hex) is binary 0001 0010

$012 (octal) is binary 000 001 010 (we only use octal for file permissions). Check the difference between 12 and 012

We are often using labels, like

```
msg:
      .ascii    "Hello, world!\n"
```

The label msg is a symbolic adddress, and is assembled to an actual number - an address.

What is the difference between

```
mov msg, %rax
```

and

```
mov $msg, %rax
```

?

$msg moves the actual *address* into rax (immediate addressing)

msg moves the *data in* address msg into rax  (direct addressing)

so

```
mov msg, %rax
```

is slower. Having decoded the instruction, we must read memory a second time, fetching the data in address msg and placing it in rax.

## syscall and int 80

To make use of OS services, such as input and output ( I/O ), 64bit code uses the syscall

instruction. A value is placed in rax to fix which service we want, and other parameters are placed in rdi, rsi and other registers, then the syscall instruction is issued. Any returned values are in rax. We have used a syscall to output Hello world.

32 bit code used a software interrupt, int 0x80, to do the equaivalent. 64 bit code should use a syscall.

# Basic techniques

In these example programs we use a 'call' instruction to call a subroutine, and push and pop the stack to preserve registers. This is explained in the chapter on subroutines.

## Arithmetic

There are instructions to do arithmetic - add subtract multiply and divide - on operands in memory and registers. These work on integer values (see the Floating point Arithmetic chapter for non-integers).

In this section we use some code which outputs register rax in decimal. The code is in file decOut.s, which we can use like this

```
 #    arith1.s
#     print out ax in decimal format

.text
.global _start
_start:
      # execution starts here
      mov $123, %rax          # test value
      call intOut

      # and exit
      mov    $60,%rax          # exit code
      mov    $0, %edi          # return 0
      syscall                  # call OS kernel
```

We assemble this with

```
as -o t.o arith1.s
```

link with

```
ld -o myProg t.o decOut.o
```

and run by

```
./myProg
```

The file decOut.s is listed and explained in the Appendix.

So we can do simple addition:

```
      mov $2, %rax      # test values
      add $3, %rax      # result is in rax
      call intOut       # 5
```

subtraction

```
      mov $2, %rbx
      mov $9, %rax
```

```
    sub %rbx, %rax
    call intOut       # 7
```

multiplication is different:

```
    mov $9, %al       # 8 bits
    mov $2, %bl
    mul %bl
    call intOut       # 18
```

The mul instruction takes just one operand. If that' s a byte, as here, it multiplies al and the result is in ax. Otherwise it multiplies rax with the result in rdx:rax (when you multiply say 1000 X 1000, the result is 1000 000 - twice as many digits).

Division is also different. Because it is integer arithmetic, there are two results - the quotient and the remainder:

```
    mov $13, %ax
    mov $3, %bl
    div %bl           # div 13 by 3
    push %rax         # preserve ax
    and $0x00ff, %ax  # zero top byte
    call intOut       # quotient : 4
    pop %rax          # get ax back
    shr $8, %ax       # shift high byte to low byte
    call intOut       # remainder : 1
```

(push pop and shift are discussed elsewhere). The 16 bit ax register is made of the top 8 bits in ah and the bottom 8 bits in al.

The div puts the quotient in al and the remainder in ah

so after the div, ax is, in hex

| ah | | al | |
|---|---|---|---|
| 0 | 1 | 0 | 4 |

We save this on the stack. Then do a logical AND with 00ff, which forces the upper byte to zero (see the section on shifts and masks)

| ah | | al | |
|---|---|---|---|
| 0 | 0 | 0 | 4 |

This is the quotient only, and we print it out.

Then get ax back off the stack, and shift it right 8 places. This moves the remainder into al

| ah | | al | |
|---|---|---|---|
| 0 | 0 | 0 | 1 |

and we print this out.

## Branches

There is a simple *jump* or unconditional branch to a new location:

```
jmp newLoc
```

which is like a GOTO in BASIC. Most modern high level languages do not use GOTOs - but we need them in assembler.

A conditional branch is like an if in C. For example how do we do

```
if (x>5)
     y=2;
else
     y=3;
```

The processor's *flags register* holds characteristics of the previous instruction - such as if it was greater than zero. A conditional branch uses this information. A compare instruction sets flags as if a subtraction has taken place.

So:

```
      mov $4, %bx       # x=4
      cmp $5, %bx       # compare 5 and x
      jg loc1           # jump on greater - if 5>x goto loc1
      mov $3, %ax       # make y 3
      jmp loc2          # jump over
loc1: mov $2, %ax       # make y 2
loc2: call intOut       # 3
```

## Loops

We can similarly use cmp and a branch to do loops:

```
      mov $5, %ax
loc1: call intOut       # 5 4 3 2 1
      sub $1, %ax
      cmp $0, %ax
      jne loc1          # jump if not equal
```

## Rotates and shifts

Processors typically have *shift* and *rotate* instructions. Shifts move bits along a register, left or right. Bits that move out the end go into the carry flag (in the flags register).

For example, initially

| CF | | Register | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ? | | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

after a shift left

| CF | | Register | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |

and again:

| CF | | Register | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

This has the same effect as multiplying the contents by 2 - but it is faster than a multiplication

A rotation is similar, but the bit that goes out one end goes back to the other. For example a rotate right:

Initially:

| Register | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

after a rotate right:

| Register | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

We will use a shift to output the al register in binary:

```
binOut:
      ## output register al in binary
      push %r8          # preserve register
      push %rax
      mov $8, %r8       # count bits
nextBit:
      shl %al           # shift left - leftmost bit into carry flag
      jc one            # jump on carry to print 1
                        # here we want to print 0
      push %rax         # keep rax
      mov $0, %al       # 0 in al
      call digitOut     # print it
      pop %rax          # get rax back
      jmp more          # unconditional jump over printing 1
one:  ## output 1
      push %rax         # just like printing a 0
      mov $1, %al
      call digitOut
      pop %rax
more: dec %r8           # counting bits
      jne nextBit       # if not down to 0, do next bit
```

```
      call newline
      pop %rax
      pop %r8           # get r8 back
      ret
```

This uses the routines digitOut and newline:

```
digitOut:   ## output the digit in al
      push %rcx         # preserve registers
      push %rbx
      push %rdx
      push %r8
      add $48, %al      # convert digit to ascii
      mov $buff2, %ecx  # put pointer to memory space in ecx
      mov %al, (%ecx)   # store in memory (indirect addressing)

      ## output
      mov $buff2, %rsi  # start of data to be output
      mov $1, %rdx      # character count : just 1
      mov $1,%rdi       # to stdout
      mov $1,%rax       # system call number (sys_write)
      syscall           # call kernel

      pop %r8
      pop %rdx          # restore registers
      pop %rbx
      pop %rcx
      ret
newline:    ## output a new line
      mov $buff2, %ecx  # start of data to be output
      movb $10, (%ecx)  # ASCII line feed
      mov $1, %rdx      # character count : just 1
      mov $1,%rdi       # to stdout
      mov $1,%rax       # system call number (sys_write)
      syscall           # call kernel
      ret


.data
buffer:    .zero 100          # 100 bytes reserved, filled with 0
buff2:     .zero 100
```

We test this with a main program:

```
      mov $0x17, %al
      call binOut
```

We use hex format to put the value in al - then it is easier to see the result should be 00010111

## Masks

Processors also have bitwise logical operations. For example

```
and $0x0f, %al
```

The two operands are the al register and 00001111 in binary. Bitwise means each pair of bits have a logical AND applied to them, with the result written into the second operand.

So what will be the effect? Suppose al initially contains 01110111:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Initial | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| AND | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| result | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

If you AND a bit with 0, you force it to a 0 { because 0 AND 0 is 0, and 1 AND 0 is 0 }. If you AND it with 1, it remains unchanged. So you can use an AND mask to force given bits to 0 - in this example, we are forcing the top 4 bits to 0, and the bottom 4 remain unchanged.

We can check with

```
      mov $0x77, %al
      call binOut       # get 01110111
      and $0x0f, %al
      call binOut       # get 00000111
```

In a similar way, a logical OR will force bits to a 1:

```
      mov $0x77, %al
      call binOut       # get 01110111
      or $0x0f, %al
      call binOut       # get 01111111
```

The AMD64 also has xor, not and andn (and not) instructions.

If you XOR a bit with itself, you get 0:
0 xor 0 is 0

1 xor 1 is 0

so

```
      mov $0x77, %al
      call binOut       # get 01110111
      xor %al, %al
      call binOut       # get 00000000
```

but this is faster than mov $0, %al, and is often used as a fast way to zero a register.

# Addressing Modes

This means how we specify where the operand is

## Immediate addressing

Here the operand is *in the instruction*.

For example

```
      mov $0x43, %al          # 8 bit transfer
      mov $0x1234, %ax        # 16 bits
      mov $0x12341234, %eax   # 32 bits in eax
      mov $0x12341234, %rax   # 32 bits into rax
```

These move data into the al register (8 bit), ax (16 bit), eax (32 bit) and rax (64bit) registers.

The corresponding code is

```
   0: b0 43                   mov    $0x43,%al
   2: 66 b8 34 12             mov    $0x1234,%ax
   6: b8 34 12 34 12          mov    $0x12341234,%eax
   b: 48 c7 c0 34 12 34 12    mov    $0x12341234,%rax
```

In each case, the data to move is in the instruction, in little-endian format.

## Absolute addressing

The operand is the address in memory to use:

```
# absadd.s
# Absolute addressing

.data                    # section declaration
      num1: .quad       3     # quad = quad word = 8 bytes
      num2: .quad       4

.text
.global assFunc
assFunc:
      mov   num1, %rax  # rax contains 3
      add   num2, %rax  # rax contains 7
```

## Relative addressing

The operand is an address relative to the current location. For example:

```
      jmp loc1
      nop
      nop
      nop
loc1: jmp loc2
      nop
      nop
```

```
loc2: nop
```

jmp is jump. nop is no operation - it does nothing. When assembled the machine code is

```
 0000000000000000 <_start>:
   0: eb 03                    jmp    5 <loc1>
   2: 90                       nop
   3: 90                       nop
   4: 90                       nop
0000000000000005 <loc1>:
   5: eb 02                    jmp    9 <loc2>
   7: 90                       nop
   8: 90                       nop
0000000000000009 <loc2>:
   9: 90                       nop
   a: bb 00 00 00 00           mov    $0x0,%ebx
```

so the first jump is disassembled as jmp 5 = jump to address 5. But the coding is eb 03 - the 3 because we are jumping 3 bytes ahead.

The second jump is jmp 9 - loc2 is at address 9. But the code is eb 02 =jump 2 bytes forward.

The address of the current instruction is held in the instruction pointer, RIP, in the processor. The displacement is simply added to RIP.

If the code is *relocated* - moved to a different location in memory - it works the same.

## Indirect addressing

The operand contains the address of the data.

For example

```
.data                           # section declaration
      num1: .quad       7       # quad word = 8 bytes
      num2: .quad       1

.text
.global assFunc
assFunc:
      mov   $num1, %rbx         # move the address of num1 into rbx
                                # not the contents of num1
      mov   (%rbx), %rax        # rax contains 7
      add   $8, %rbx            # rbx points to num2
      add     (%rbx), %rax      # rax is 8
```

This uses rbx as a pointer to the data. The data is in address num1 (then num2).

```
      mov   $num1, %rbx
```

puts that address in register rbx

( mov num1, %rbx would have used absolute addressing and put 7 in rbx)

Then in

```
      mov   (%rbx), %rax
```

the brackets around %rbx mean indirect addressing - so we use rbx as a pointer to the data. mov %rbx, %rax would have simply moved what is in rbx (the address) into rax.

Then

```
      add   $8, %rbx
```

changes rbx - it moves the pointer on by 8 bytes, so i now points to num2. Then

```
      add     (%rbx), %rax
```

adds num2 into rax.

This corresponds to the following in C:

```
int i=7;
int * p = &i;
// use *p
p++;
// use next int
```

p++ looks like it just adds 1 to p. But we have told it p points to an int, so it actually adds on the number of bytes occupied by an int.

Indirect addressing is slower than direct - more clock cycles are need. To complete mov (%rbx), %rax, once it has been fetched, it must be

1. decoded

2. The contents of rbx are placed on the address bus

3. A memory read takes place

4. The value read is placed in rax

but mov %rbx, %rax just requires

1. decoding

2. transferring rbx to rax, inside the processor only

## Stack operations

The push and pop instructions, and call and ret, implicitly use the stack pointer register (in other words, they use the stack pointer without actually refer to it).

For example

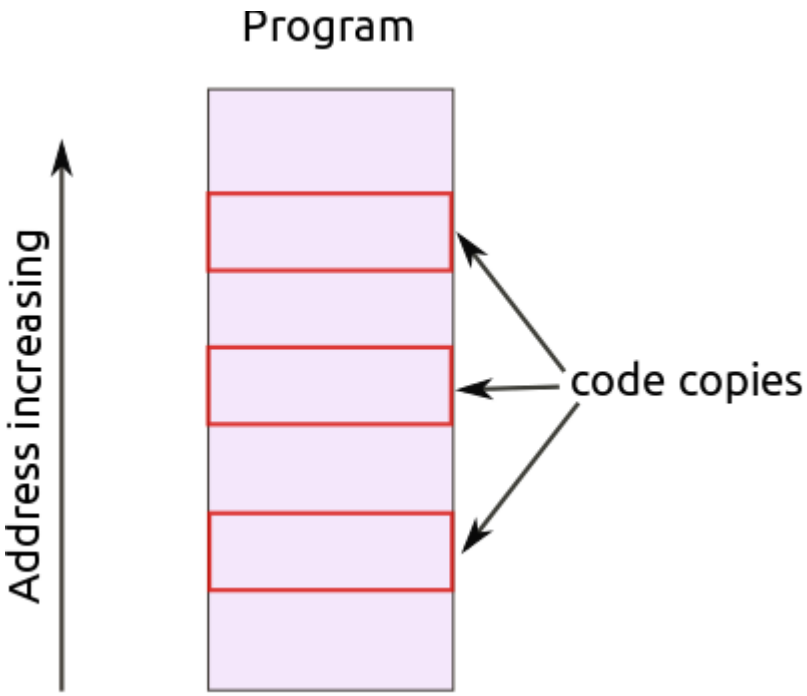
push %rax

puts a copy of rax onto the top of the stack, as pointed to by the stack pointer, and then adjusts the stack pointer.

See the section on sub-routines.

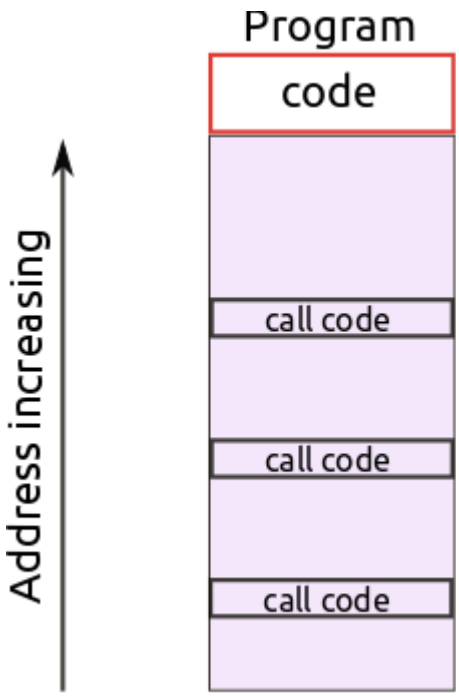
## Subroutines

Suppose we have a useful piece of code - for example something to output the contents of a register in decimal. This is not trivial - the register actually contains binary, so we must convert it to decimal digits, then convert them to ASCII characters. We will develop it shortly. But then - how to use it? More than once?




Suppose we want to use to use it three times. One way would be copy the instructions three times, as show. This is *in-lining* the function. This is fast but obviously expensive of memory. And if we modify the code to a better version, we must alter every copy.

The alternative is just to have one copy, but to be able to run it three times. In effect we treat the code block as a small program itself, and 'run' the small program whenever we need it.

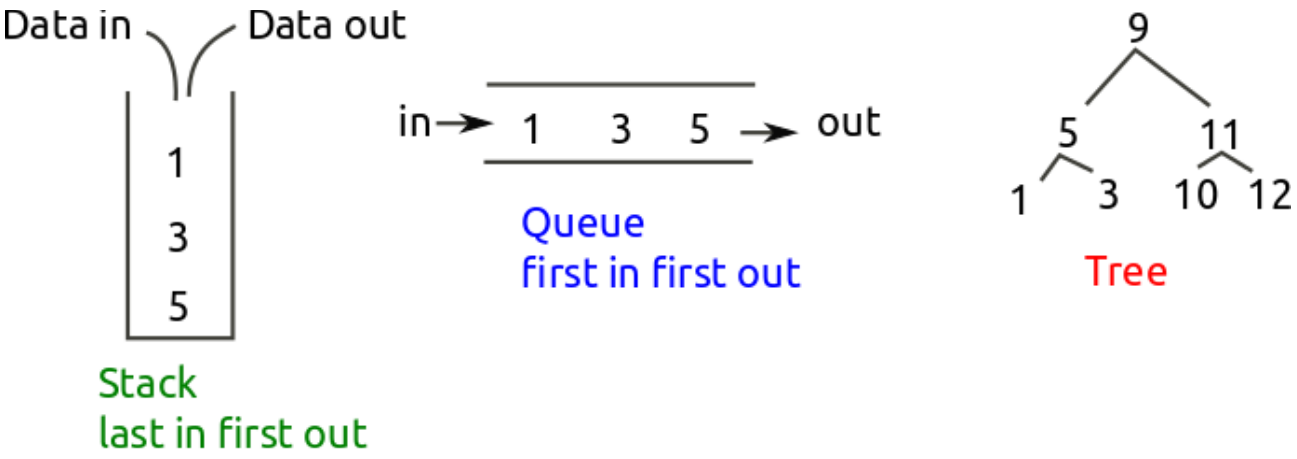



This is the basis of *structured programming*. We arrange code into small sections, called (in different languages) sub-programs, sub-routines, modules, procedures and functions. Sub-routines can be called whenever needed. They can be re-used in different programs. They can be tested separately from each other. They can be written by different members of a team, and so speed up development. Structured programming is effective, in assembler and in high level languages..

But, a return mechanism is needed. That is, when the sub-routine is called from a given location, after it completes, it must return to the next location. How do we remember *where to return to*? By using a stack.

## Stacks

A stack is a *data structure*. Here are three common data structures - stack queue and tree:




Each data structure has a range of processes - *algorithms* - for operations such as adding a data item, finding one, deleting and so on. The study of data structures and algorithms is a major part of classical computer science.

Here we use a stack. It has just two operations - to *push* a new value onto the top of the stack, and to *pop* a value off. We need to keep track of where the top of stack is, in a *stack pointer.*

## Stacks for return address

We go through in detail the process of using a stack to handle return addresses.

The diagram on the right shows memory contents, with addresses and contents, and the stack pointer and instruction register.

The instruction pointer (IP) contains 100. This means the next instruction about to be executed is the one at address 100 - which says call 200. This is the start of a sub-routine, which ends at a return at address 202.

The stack pointer (SP) contains 500. This means that if anything is pushed on the stack, it will go at address 500.

Then the instruction at 100 is executed, this happens:

IP is incremented, to 101

101 (the *return address*) is pushed on the stack, where SP points to = 500.

200 is transferred to the IP.

The result is:

| | |
|---|---|
| 501 | .. |
| 500 | .. |
| | |
| 202 | return |
| 201 | .. |
| 200 | .. |
| | |
| 105 | call 200 |
| 104 | .. |
| 103 | call 200 |
| 102 | .. |
| 101 | .. |
| 100 | call 200 |
| address | contents |
| | |
| stack pointer | 500 |
| instruction pointer | 100 |

Now we are executing the sub-routine, starting at 200. This will run until IP gets to 202, which is a return instruction. When this happens, the stack is popped, and the value placed in IP.

This value is 101 - the return address after the sub-routine call at 100. So after the sub-routine has ended, we return to the instruction after the call.

| | |
|---|---|
| 501 | .. |
| 500 | 101 |
| | |
| 202 | return |
| 201 | .. |
| 200 | .. |
| | |
| 105 | call 200 |
| 104 | .. |
| 103 | call 200 |
| 102 | .. |
| 101 | .. |
| 100 | call 200 |
| address | contents |
| | |
| stack pointer | 501 |
| instruction pointer | 200 |

As we continue, the call at 103 will mean 104 will go on the stack, and we switch to 200 again. At 202 we pop 104 off and return.

At 105 we call it again, and we will again get the correct return address on the stack.

Why not simply keep the return address in a special register - maybe the 'return register'?

Because one sub-routine might call another. So we

call sub1

and in sub1 it say

call sub2.

At the end of sub2 we have to go back to sub1, and at the end of that, to go back to after sub1.

Sub-routine calls can be nested like this indefinitely. With a stack there is no problem the return addresses are placed on the stack and the stack pointer grows. As returns are hit addresses are popped back off the stack, the stack pointer is reduced, and we are back to where we started - or rather, the next instruction.

But, there must be some limit - the stack is in memory, so sooner of later we will run out.

The Linux command ulimit -a shows current limits on various thing:

```
walter@walter-s5-1030uk:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 30940
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
```

| | |
|---|---|
| | |
| 501 | .. |
| 500 | ... |
| | |
| 202 | return |
| 201 | .. |
| 200 | .. |
| | |
| 105 | call 200 |
| 104 | .. |
| 103 | call 200 |
| 102 | .. |
| 101 | .. |
| 100 | call 200 |
| address | contents |
| | |
| stack pointer | 500 |
| instruction pointer | 101 |

```
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 30940
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

This value of 8 MB is the standard default for Linux.

Suppose a sub-routine calls itself? If it will always call itself, we have an *infinite recursion*, which would require an infinite stack - which we do not have. When the stack is exhausted, we will instead get a *stack overflow* exception.

## Sub-routine example

Suppose we want a sub-routine to compare two numbers.

We must pass this sub-routine two parameters (or arguments), the two numbers to compare.

And it must return a value - the larger one.

One way to do this is in registers. We can pass the two numbers in two registers - rdi and rsi - and return the larger in rax - like this

```
 .text
.global _start
_start:
      # execution starts here
      mov $4, %rdi            # test values
      mov $5, %rsi
      call bigger
      ..

      ..
      ################################################################

bigger: # return larger of two parameters - in rdi and rsi
            # return larger in rax
            cmp %rdi, %rsi
            jl rdBig          # jump if less
            mov %rsi, %rax
            ret
rdBig:      mov %rdi, %rax
            ret
```

This uses a *calling convention* - parameters in rdi and rsi, and return value in rax. More on this later.

```
Or we could pass the parameters on the stack, and pass the return value back on
the stack:
      push $6     # test values
      push $5
      call bigger
      pop %rax    # put resullt in rax


bigger: # return larger of two parameters - on stack
      # return larger on stack
      pop %rbp          # get return address off stack
      pop %rdi          # get parameters off stack
      pop %rsi
      cmp %rdi, %rsi
      jl rdBig
      push %rsi         # rsi is bigger - put on stack
      push %rbp         # and put return address there also
      ret
rdBig:
      push %rdi         # rdi bigger
      push %rbp
      ret
```

We must also take into account the use of the stack for the return address.

We push the parameters on the stack, then call the routine - which pushes the return address on top of stack. So when we enter the routine there are 3 things on the stack - 2 parameters, and on top, the return address. We take that off and keep it in rbp. We work out which is bigger, and push it, then push rbp back. So when w ehit ret, the top of stack is the return address.

Back in the calling code, we pop a value off the stack - and get the returned value.

Using the stack is slower, but allows for a reasonably unlimited number of parameters - and several return values.

## Subroutines and software interrupts

We are using syscall to get OS services - why not sub-routine calls?

A main purpose of the operating system is to allow access to system resources to applications - such as outputting text. It might be possible to write code to do this and copy it in each application - but that would not be a sensible approach. Better to have it in the operating system and use it wherever required.

However the operating system is allowed to do things applications cannot - like reading and writing all memory, starting and stopping processes and so on. x86 processors have 4 *privilege levels*. The Linux runs at level 0, and applications are at level 3.

So the application needs access to code at a higher privilege level, which a simple sub-routine call would not allow.

But the syscall does.

# Assembler and C

This is in two parts:

1. Mixing C and assembler code

2. Looking at compiled code from C as assembler, to see how C turns to machine code.

## Mixing assembler and C

gcc can handle in-line assembler in C functions. However the syntax is ugly, and we want more than in-line code - we want to be able to call assembler sub-routines from C.

We will write some C source code and save it in a .c file. Then we write some assembler source in a .s file.

Then we call gcc, and give it the .s and .c files. gcc is smart enough to know what it needs to do - compile the .c as C, assemble the .s, then link them to produce a single executable file.

The C source code is

```
#include <stdio.h>

int main(void)
{
printf("Hello from C\n");
ass1();
}
```

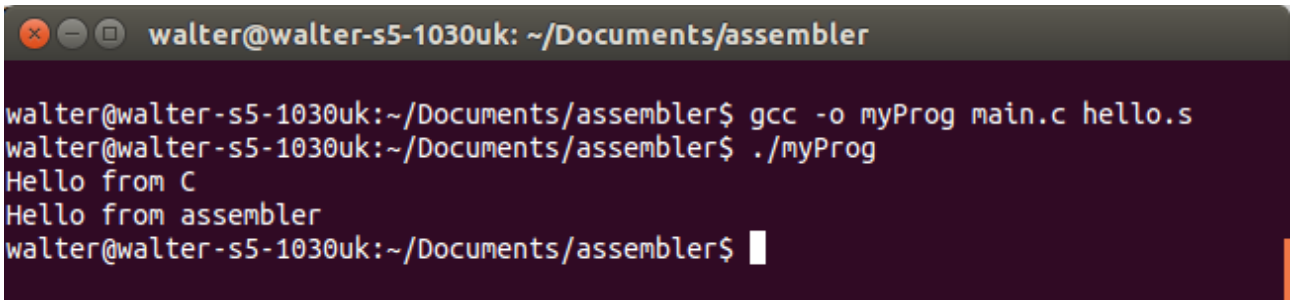
This is in file main.c

The assembler in file hello.s is

```
.text
    .global ass1
ass1:
        mov    $len, %rdx
        mov    $msg, %rsi
        mov    $1,%rdi
        mov    $1,%rax          # system call number (sys_write)
        syscall
        # and exit
      mov    $60,%rax         # exit code
      mov    $0, %edi         # return 0
      syscall                # call OS kernel
.data
msg:
      .ascii    "Hello from assembler\n"
      len = . - msg
```

This is just like 'Hello world', but without _start, and exposing the label ass1 instead. This is because we will start execution at the main of C, not in the assembler. But we must expose ass1, so the linker can find it and link it to the C code.

then:




## Return values

But suppose the assembler function is not a void function? If it returns a value, how is that transferred back to the calling C code? It has to be in rax:

```
#include <stdio.h>


int main(void)
{
      int x = ass1();
      printf("x = %d\n",x); // x = 67

      return 0;
}
```

and

```
.text
    .global ass1
ass1:
      mov $67, %rax
      ret
```

But how do we know it is in rax? Because that is the *calling convention*.

## Pass parameters into assembler

Another example - suppose the assembler function has parameters. We write a subroutine which take 2 parameters, subtracts them and returns the difference:

```
#include <stdio.h>

int main(void)
{
printf("Hello from C\n");
int x =ass2(5,2);
printf("x = %d\n",x);
}
```

and

```
#
# receive 2 parameters, subtract them and return the result
#
.text
    .global ass2
ass2:
        sub %rsi, %rdi
        mov %rdi, %rax
        ret
```

We get x=3. So the two parameters are passed in registers rsi and rdi, because the calling convention says so.

## System V ABI

System V is Unix System V, a very successful commercial version of Unix. ABI is the 'Application Binary Interface'. It is used in Linux, BSD and other systems. It is written in a flexible way so that it can deal with different processors (not just AMD64). It has two parts:

### Executable and Linkable Format

ELF is a standard for the format of executable (native code program) files, and linkable file( with a .o ) name.

### Calling Convention

This is about standards for parameter passing, return values, preserving registers and stack use. It has to be processor-specific, since it names registers. The x86-64 version is outlined here.

Up to 6 parameters are passed in in registers `rdi`, `rsi`, `rdx`, `rcx`, `r8 and r9. Any` more than 6 are passed in on the stack.

Registers rbx, rsp, rbp, r12, r13, r14, and r15 must be preserved (if used, pushed on the stack and popped back at the end). Registers rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11 can be changed.

Any return value is in rax, or rdx:rax if 128 bits.

Full details are here http://www.sco.com/developers/devspecs/gabi41.pdf

## The assembler of C

C has been described as 'high level assembly language'. It is interesting to look at the code generated in compiled C, as assembler. We start with a very simple piece of C:

```
int main(void)
{
```

```
      int x=0x1234;
      x++;
      int y=0x5678;
      y+=x;

      return 0;
}
```

We compile this with

```
gcc -o m.o -g main.c
```

This compiles main.c, producing the output file m.o (so just to the object file stage. The -g option includes debugging information. Then

```
objdump -d -S m.o
```

lets us look at the disassembled file m.o, with _s again retaining debug info. The section for main is

```
int main(void)
{
  4004ed:   55                      push   %rbp
  4004ee:   48 89 e5                mov    %rsp,%rbp
      int x=0x1234;
  4004f1:   c7 45 f8 34 12 00 00    movl   $0x1234,-0x8(%rbp)
      x++;
  4004f8:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
      int y=0x5678;
  4004fc:   c7 45 fc 78 56 00 00    movl   $0x5678,-0x4(%rbp)
      y+=x;
  400503:   8b 45 f8                mov    -0x8(%rbp),%eax
  400506:   01 45 fc                add    %eax,-0x4(%rbp)

      return 0;
  400509:   b8 00 00 00 00          mov    $0x0,%eax
}
  40050e:   5d                      pop    %rbp
  40050f:   c3                      retq
```

The main uses two variables local to main - x and y. These will be stored on the stack. The code uses rbp to track this. But first it preserves the current value of rbp:

```
push   %rbp
```

then puts the stack pointer into it

```
mov    %rsp,%rbp
```

so now rbp points to the top of stack.

x is held 8 bytes from the stack top, so

```
      int x=0x1234;
```

turns into

```
movl   $0x1234,-0x8(%rbp)
```

and

```
x++;
```

just adds 1 to this location:

```
addl   $0x1,-0x8(%rbp)
```

y is 4 bytes from  stack top, so

```
int y=0x5678;
```

becomes

```
movl   $0x5678,-0x4(%rbp)
```

Then how does it do

```
y+=x?
```

It puts x in eax:

```
mov    -0x8(%rbp),%eax
```

then adds that to y

```
add    %eax,-0x4(%rbp)
```

main returns 0, and this will be returned in eax - so

```
mov    $0x0,%eax
```

then we can get back rbp, pushed as the first step

```
pop    %rbp
```

Most C instructions are turning into just 1 or 2 machine code instructions - which is why C is fast.

Another example:

```
int main(void)
{
      int a=1;
      int b=2;
      int c=3;
      char d='A';

      return 0;
}
```

becomes

```
int main(void)
{
  4004ed:   55                      push   %rbp
  4004ee:   48 89 e5                mov    %rsp,%rbp
      int a=1;
  4004f1:   c7 45 f4 01 00 00 00    movl   $0x1,-0xc(%rbp)
      int b=2;
  4004f8:   c7 45 f8 02 00 00 00    movl   $0x2,-0x8(%rbp)
      int c=3;
  4004ff:   c7 45 fc 03 00 00 00    movl   $0x3,-0x4(%rbp)
      char d='A';
```

```
  400506:   c6 45 f3 41             movb   $0x41,-0xd(%rbp)

      return 0;
  40050a:   b8 00 00 00 00          mov    $0x0,%eax
}
  40050f:   5d                      pop    %rbp
  400510:   c3                      retq
```

So the layout of memory from the stack down is

| Stack top rsp | Contents | rbp |
|---|---|---|
| c | 3 | 4 bytes down |
| b | 2 | 8 byte sdown |
| a | 1 | 12 bytes down |
| 'A' | ^% | 13 bytes down |

The System V ABI allows a *red zone* of 128 bytes for functions to hold local variables - which is what we are using here.

a=1 has become

```
c7 45 f4 01 00 00 00
```

because it is *little-endian.*

d='A'

has become

```
c6 45 f3 41
```

because ASCII A is hex 41.

The C declarations and assignments are each just 1 instruction.

How do if statements work?

```
int main(void)
{
      int a=5;
      int b;
      if (a>7)
            b=1;
      else b=2;

      return 0;
}
```

becomes

```
int main(void)
{
  4004ed:   55                      push   %rbp
  4004ee:   48 89 e5                mov    %rsp,%rbp
```

```
      int a=5;
  4004f1:   c7 45 f8 05 00 00 00    movl   $0x5,-0x8(%rbp)
      int b;
      if (a>7)
  4004f8:   83 7d f8 07             cmpl   $0x7,-0x8(%rbp)
  4004fc:   7e 09                   jle    400507 <main+0x1a>
            b=1;
  4004fe:   c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%rbp)
  400505:   eb 07                   jmp    40050e <main+0x21>
      else b=2;
  400507:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)

      return 0;
  40050e:   b8 00 00 00 00          mov    $0x0,%eax
}
  400513:   5d                      pop    %rbp
  400514:   c3                      retq
```

Note the declaration

int b;

does not generate any code. Local variables have no default initialisation. We've just told the compiler that b is an int.

Then the if is a compare

```
cmpl   $0x7,-0x8(%rbp)
```

followed by a branch if less than

```
jle    400507 <main+0x1a>
```

which jumps over b=1;

```
movl   $0x1,-0x4(%rbp)
```

which itself is followed by a jump over the b=2

jmp 40050e <main+0x21>

```
      else b=2;
  400507:   c7 45 fc 02 00 00 00    movl   $0x2,-0x4(%rbp)
```

How does a C function call work? For example

```
int add(int x, int y)
{
      return x+y;
}
int main(void)
{
      int i=add(2,3);

      return 0;
}
```

Function add becomes:

```
int add(int x, int y)
{
  4004ed:   55                      push   %rbp
  4004ee:   48 89 e5                mov    %rsp,%rbp
  4004f1:   89 7d fc                mov    %edi,-0x4(%rbp)
  4004f4:   89 75 f8                mov    %esi,-0x8(%rbp)
      return x+y;
  4004f7:   8b 45 f8                mov    -0x8(%rbp),%eax
  4004fa:   8b 55 fc                mov    -0x4(%rbp),%edx
  4004fd:   01 d0                   add    %edx,%eax
}
  4004ff:   5d                      pop    %rbp
  400500:   c3                      retq
```

This starts by preserving rbp and putting the stack pointer in rbp:

```
push   %rbp
mov    %rsp,%rbp
```

The two parameters, x and y, will come in in registers edi and esi (due to the calling convention). These are placed on the stack red zone:

```
 mov    %edi,-0x4(%rbp)
 mov    %esi,-0x8(%rbp)
```

To do return x+y, it loads one into eax, then adds the other in. eax will be the return value:

```
  mov    -0x8(%rbp),%eax
  mov    -0x4(%rbp),%edx
  add    %edx,%eax
```

In main, calling add is just:

    int i=add(2,3);

```
  400509:   be 03 00 00 00          mov    $0x3,%esi
  40050e:   bf 02 00 00 00          mov    $0x2,%edi
  400513:   e8 d5 ff ff ff          callq  4004ed <add>
  400518:   89 45 fc                mov    %eax,-0x4(%rbp)
```

This puts the parameters in esi and edi, calls the sub-routine, then puts the returned value, in eax, onto the stack, where local variable 'i' is held.

# I/O

How do devices do input/output? Do processors have a write instruction, to write data to a file somehow?

No - there are several reasons why that cannot work.

- The processor does not do the I/O. In the case of a write to a disk file, it is the drive which does the writing - so it can't be a CPU instruction
- We might be doing output to a wide range of devices. The file might be on a magnetic disc, or a writeable CD, or a USB flash drive, or on a drive on another machine shared on a network, or whatever. How to deal with all those different mechanisms.
- I/O is very slow, compared with CPU clock speeds. It would be good if we could enable the CPU to do something else while I/O is happening

## Abstraction

Abstraction is a key idea in computer science. We use a generalized idea of something, and focus on what it does, rather than how it does it. Then we can use that abstraction, independently of how it is actually implemented.

Data structures are often dealt with as abstractions (abstract data types of ADTs). A stack, for example, is an ADT which can do just two things - push and pop. Stacks can be coded in many different ways - but they all just do a push and pop.

Abstraction is often used in high level languages such as Java. But it is also a useful idea in OS theory and assembly language programming.

Here the abstraction is a *file*. This is something which can (in general) be read or written. The OS deals in terms of abstract files, and so does not need to be concerned about whether it a flash drive or network drive or whatever.

The use of specific drive types is done through device drivers (software) device controllers (hardware) and some king of I/O bus arrangement. But to the application programmer, and OS syscalls, they are just files.

## I/O concepts

*Polling* means the CPU asking the peripheral if it needs attention, in a loop. Not a good idea.

*Interrupts* - the peripheral signals the CPU it needs attention by means of a hardware interrupt - a lag on an interrupt bus. The OS might handle the interrupt by calling an interrupt service routine, as appropriate for which interrupt it is.

*Direct memory access* (DMA) means the peripheral transfers data to RAM, without going through the CPU. This needs to be physical RAM not virtual memory.

A *buffer* is a block of memory used to store data before of after I/O. Running an I/O operation on a single byte is very inefficient - better to transfer a block of several kilo-bytes or more in a single burst. For output, a buffer will normally only be written when the buffer is full. When a file is closed, its buffer will be flushed.

*Standard streams* - since Unix, I/O has happened through abstracted streams, which are treated as files which can be read or written. To do file output, the file is opened, written to and closed. But there are 3 streams already open. stdin, stdout and stderr. stdin is normally connected to the keyboard for input, and stdout is the 'console' - probably a GUI simulation of a character output device. Error messages go to stderr - which usually go to the console as well.

## The I/O sandwich

The OS stores information on each open file - such as its name including the path, the owner, permissions (read or write) and so on.

It tracks which file is which info block using a file identifier FD (file handle on Windows). This is an integer which matches some file info block.

All file use should take the form of a sandwich

open file and get a FD

use the file (read/write) through its FD

close the file through its  FD

Failure to close a file is a *resource leak*. If software repeatedly opens files with closing them, eventually the file descriptor limit will be reached. On Linux

ulimit -n

shows you the maximum number of open files allowed per process.

## Example - write to file

We show a very simple example - create a new text file, write to it and close it:

```
.text
.global _start
_start:
      # open file
      mov $85, %rax           # create
      mov $filename, %rdi     # filename
      mov $0644, %esi         # flags - octal
      syscall                 ## call kernel to open file
      mov %rax, buffer        ## file id returned in rax
```

```
      # write file

      mov $1, %rax            # write call number
      mov buffer, %rdi        # get fd into ebx
      mov $textdata, %rsi     # pointer to data
      mov $len, %rdx          # bytes to wrte
      syscall                 # call kernel to write to file

      # close file
      mov $6, %rax            # sys call close file
      mov buffer, %rdi        # get fd
      syscall                 # close file


      # exit
      mov    $60,%rax          # exit code 0
      mov    $0, %edi          # system call number (sys_exit)
      syscall                 # call OS kernel
      ###################################################################

.data
filename:
            .asciz "data.txt"
textdata:
            .asciz "This is a test"
            len = . - textdata
buffer:     .zero 100
```

The filename is in ASCIZ form. This is how strings are normally held in C - as a sequence of ASCII character codes, with a zero byte at the end. As a result we only need to tell the syscall where the filename starts, not its length - it ends at the zero.

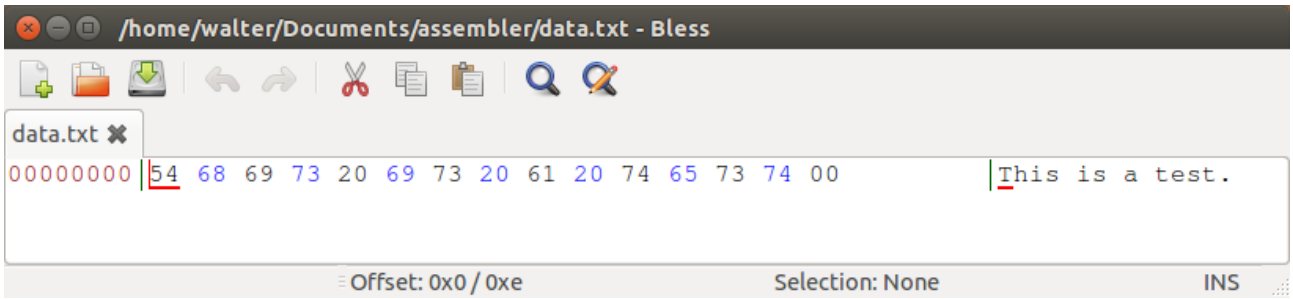
Hoever the dat to be written to the file might be binary, and contain zeros to be written - so we have to tell that syscall what the data length is.

When the file is created, we set permissions with

```
      mov $0644, %esi         # flags - octal
```

The permissions are read, write and execute, for the file owner and others. We are using octal, so that is 644 = 110 100 100. So we get read and write for the owner, and read only for the group and others.

When this executes, we get a file data.txt on disc (next to the executable). We can look at its contents with a hex editor like Bless:



# Floating point arithmetic

Integers are usually represented as straight binary - or two's complement for signed integers. Floating point values must be stored some other way. How?

## Mantissa and exponent

For example, the number 283.89. In 'scientific notation' this is written as 2.8389 X $10^2$. This is the idea - to store the number in two parts:

| | |
|---|---|
| 28389 | The mantissa |
| 2 | The exponent |

That's it. This is the basic idea.

## Mantissa and exponent in binary

Since we are using digital systems, everything must be in binary. So the *mantissa* and *exponent* are in binary, and the exponent is the power of two, not ten.

They are both fixed width. Suppose we have an 8 bit mantissa and 4 bit exponent ( the real thing is much longer).

What would 13.75 look like? First write it in binary

$13.75_{10} = 1101.11_2$ ( the binary fraction values are 1/2 and 1/4, so 0.75 = .11 )

Then adjust the point to before the first digit

$1101.11 = .110111 \text{ X } 2^4$

So the exponent is $4_{10}=100_2$, and the mantissa is 0.110111. In our format this would be

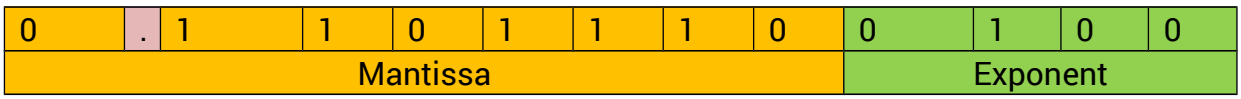
| 0 | . | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mantissa | | | | | | | | | Exponent | | | |

What is the biggest number we can have in this format? The biggest exponent is 0111 (1000 would be negative, since this is two's complement). The biggest mantissa is 0111 1111. So the biggest value is

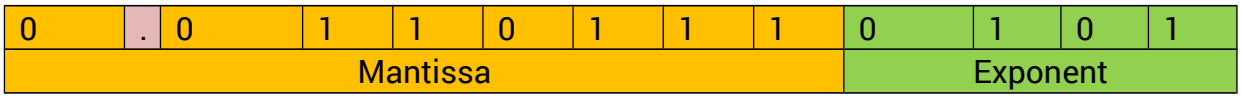
$0.111\ 1111 \text{ X } 2^{0111} = 0.111\ 1111 \text{ X } 2^7 = 1111111.0_2 = 127$

## Normalisation

This number:




and this:




are equal - we have shifted the mantissa 1 to the right, and increased the exponent.

This is like in base 10

$0.123 \times 10^4 = 0.0123 \times 10^5$

Does it matter which way we store it? Yes. The mantissa has fixed width, and as we shift right, eventually we will start to lose bits at the end - which looses precision. It is best to have the mantissa as far left as possible, then we have room to hold the maximum number of bits - which equals best precision. A number in this form is called *normalised*.

Often in floating point calculations we get an intermediate result which is not normalised (and uses more bits than usual). We follow this by normalising it, to maintain precision.

## Cautions

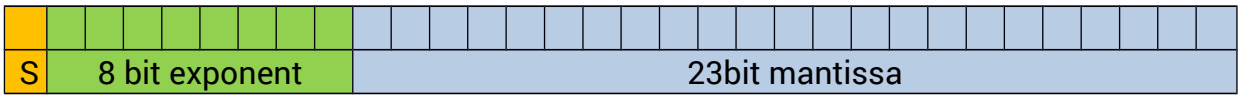
Most values cannot be represented in floating point with complete accuracy.

Most floating point arithmetic does not produce exact results

## IEEE754

Floating point standards set out how many bits are used for exponent and mantissa, in what order and so on. gcc uses the IEEE754 standard for representing 32bit floats and 64bit doubles. We will look at 32bit floats - 64bit is the same idea, but with added bits.

The 32 bits are used as three fields - a sign bit, an 8bit exponent and a 23bit mantissa:




Sign - 0 for positive, 1 for a negative number

Exponent - the 8 bits hold the 'actual' exponent plus a 'bias' of 127

Mantissa - the fraction part, with the number adjusted so that the leading 1 bit is to the left of the point - for example 1.0101. Since we know the leading bit is 1, we do not need to store it. So if the stored mantissa is 0101000000.. this represents 1.0101.

## Floating point in software

This means doing floating point arithmetic by using combinations of integer arithmetic and logical instructions.

To illustrate (and working in base 10 for simplicity) suppose we want to subtract:

$0.1234 \text{ X } 10^2$ subtract

$0.11 \text{ X } 10^1$

The first step is to make the exponents equal. We make the smaller equal the larger, by shifting the mantissa right (this would be some logical shift instructions):

$0.1234 \text{ X } 10^2$ subtract

$0.011 \text{ X } 10^2$

Then we subtract the mantissas. This would be some integer subtracts, like 'long subtraction'

$0.1124 \text{ X } 10^2$

The result is already normalised, but we might have needed to adjust it.

So we can do floating point using integer instructions - and before the 8087 (and AMDs Am9511 of 1977) this was the only option - but it was slow.

The Intel 8087 numeric co-processor was introduced in 1980 to accompany the 8086 16 bit processor. The 8087 had floating point arithmetic instructions - to be done 'in hardware', which meant a significant speed increase.

## x86 IA64 floating point instructions

There are 3 sets of floating point instructions

1. The 64 bit media programming instructions, which use the mmx registers, mmx0 to mmx7

2. The SSE instructions, which use the 64bit xmm registers, and the 128bit ymm registers.

3. The x87 legacy instructions

The first and second of these use SIMD (single instruction multiple data) instructions. The idea of these is shown in this diagram, from the AMD manual:

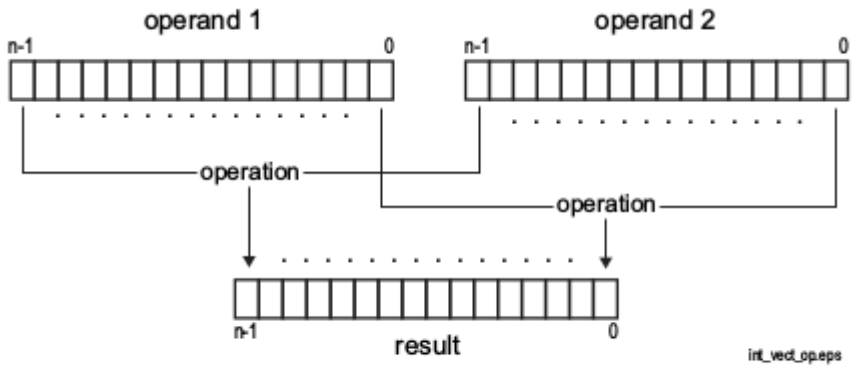



**Figure 4-9. Mathematical Operations on Integer Vectors**

This shows a single operation (maybe an add) being applied to each byte (or word or quadword) of two operands (so multiple data).

So then we have Streaming SIMD Extensions - or SSE.

Since we just want to look at the idea, we just cover the SSE versions

## SSE floating point

We do not want to have to output floating point in assembler - so we use C printf to do this:

```
#include <stdio.h>

double assfp(double, double);

int main(void)
{
      double f=assfp(0.3, 0.4);
      printf("%f\n",f); // 0.7
      return 0;
}
```

The assembler is just

```
# assfp.s
# Floating point
#
.text
    .global assfp
assfp:
      addsd %xmm1, %xmm0
      ret
```

We compile and link this by

gcc -o myProg main.c assfp.s

and run it by

./myProg

The SSE instructions use 8 64 bit registers, xmm0 to xmm7. The calling convention says parameters are passed in through xmm0 and the rest in order, and a double value returned in xmm0. We have to tell C

```
double assfp(double, double);
```

so it knows how to handle parameters and the returned value

The assembler simply adds the parameters, leaving the result in xmm0, to be returned.

```
addsd %xmm1, %xmm0
```

The addsd means add scalar (not vector) type double.

As another example, we write a function to calculate the distance between two points, x1,y1 and x2,y2. So we need to calculate

$$\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$$

The C is

```
double assfp(double, double, double, double);

int main(void)
{
      double f=assfp(0.0, 0.0, 1.0, 1.0);
      printf("%f\n",f); // 1.414214

      return 0;
}
```

The four parameters will be passed in xmm0, xmm1, xmm2 and xmm3:

```
.text
    .global assfp
assfp:
      subsd %xmm2, %xmm0      # x1-x2 in xmm0
      mulsd %xmm0, %xmm0      #(x1-x2)^2 in xmmo
      subsd %xmm3, %xmm1      # y1-y2 in xmm1
      mulsd %xmm1, %xmm1      # (y1-y2)^2 in xmm1
      addsd %xmm1, %xmm0      # (x1-x2)^2 + (y1-y2)^2 in xmm0
      sqrtsd %xmm0, %xmm0     # square root
      ret
```

# Some more realistic aspects

## Cache memory

We have thought of storage in registers or memory. In fact memory access is relatively slow, and we want to minimise memory reads or writes. Cache memory does this.

Cache memory is a small amount of fast memory placed between processor and main memory. The following happens:

1. On a memory read, there is first a check whether there is already a copy of that location in the cache. If there is, it is used and there is no actual memory read.

If it is not in the cache, memory is read and used, with a copy placed in the cache for possible use next time.

2. On a memory write, the value is written to the cache. It is only written to memory if the process ends, or if the cache is full, when it is written and cache space re-used.

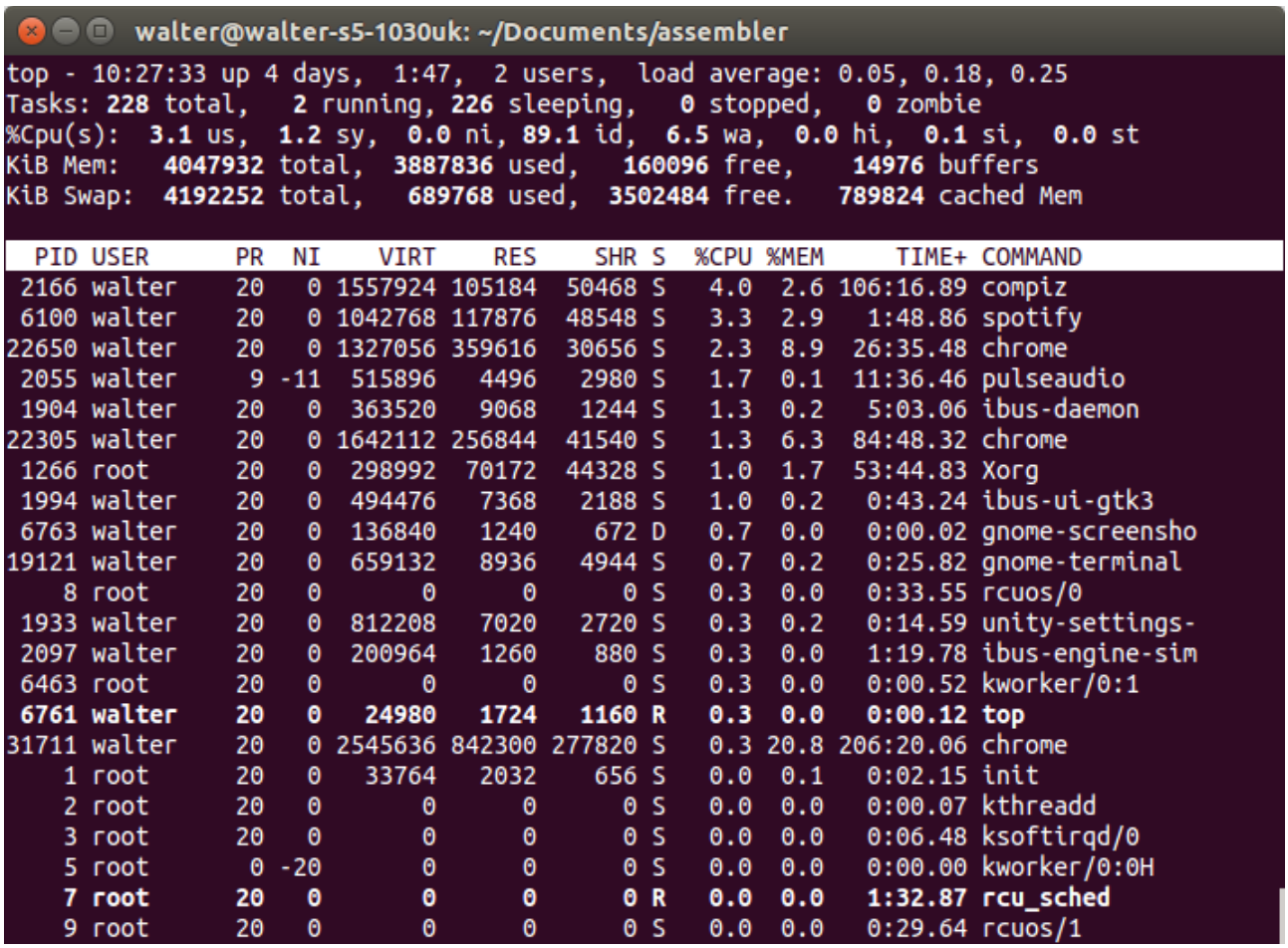
The idea is that programs often use a fairly small number of memory locations, repeatedly in loops. Then most of the access is fast read and writes to the cache.

In fact processors structure cache into several levels, of slower and larger amounts. Currently they use 3 levels of up to 18 MB.

## Programs and processes

We have thought of our programs as being loaded, run and ending - with nothing else happening at the same time. The operating system sees things differently.

Try the command *top*:




top displays a view of the system, updated in real time. Line by line:

Linux has been running here for 4 days, with 2 users (me and root), system load average over the last 1, 5 and 15 minutes.

Then 228 tasks (processes). 2 are running, 226 sleeping, 0 stopped. zombies are dead processes.

Then % cpu time on different task types. us = user, sy = kernel, id is idle. ni is nice - user process with raised priority (see below).

Then memory usage

Then a table giving information about each process. In the command column we can see compiz - (fancy window effects), spotify, chrome browser, screenshot (which got the image) - plus lots of system processes.

So at any time, the operating system has a set of processes. Most of these (226) are sleeping - not running, while 2 are running (because it is a multi-core processor).

Since it can only run a very small number of processes at the same time, it must schedule when to run them. Because we want to, for example, listen to Spotify while using the browser, it cannot simply queue them and run each one in turn to completion.

Instead it uses a time-slice (of maybe a few milliseconds). A running process runs until

its time-slice ends, or it wants to do some input or output operation. These are slow - so rather than waste time waiting for it, the process is suspended (sleeps) and the next one gets a time-slice.

There are various scheduling algorithms in use to determine in which order processes will get a time-slice. Processes have a 'priority' which can be taken into account. A 'nice' user process has raised priority.

## Virtual memory

We have taken the addresses in machine code to be locations in RAM. They are not.

The operating system must

- somehow organise the use of physical RAM among all the processes needing to use it
- ensuring one process cannot write into the memory space of another process without permission
- cope with total RAM requirement being larger than total RAM present.

This is normally done using the idea of virtual memory. This is a mapping between addresses in the program and actual RAM addresses, plus disc storage:

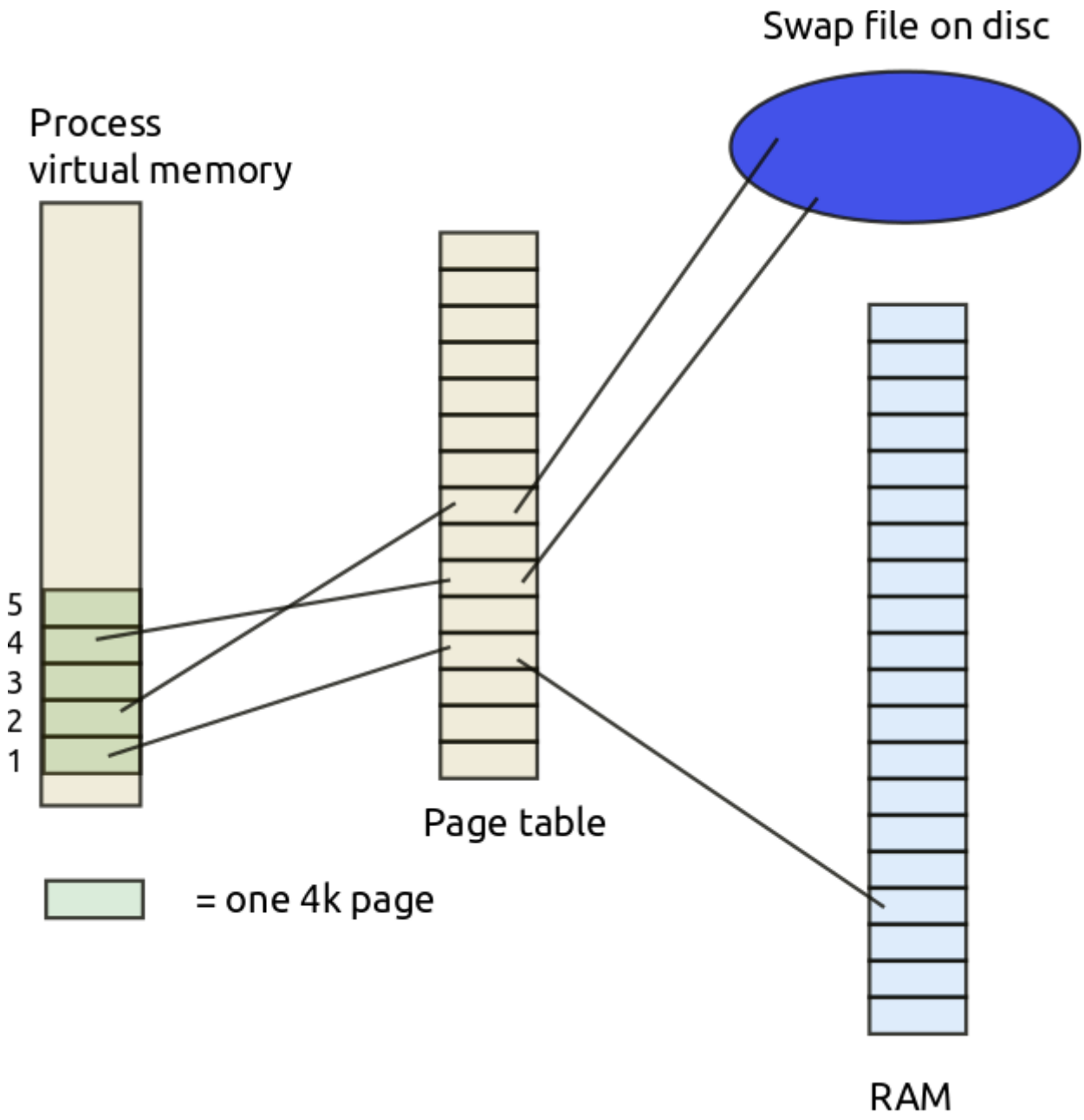
1. Memory is handled in blocks called *pages*. On Linux a page is normally 4k.

2. The application program is written treating the whole of the 64bit address space as being usable. In other words it just sees all addresses available, with no other processes present. This is the *virtual address space*.

3. A disc file (a swap file) is also used for storage. Pages stored on disc must be loaded into RAM before being used.

4. The operating system (or hardware) maintains a table, one entry per page, which connects a page in virtual memory to a page in RAM, or in the swap file.

Like this:




Page 1 in the application is in RAM. Pages 2 and 4 are swapped out.

Suppose a JMP instruction in page 1 jumps to a location in page 2. That must be swapped in from disc to RAM and the table updated. It might be that another page in memory is swapped out to disc. All this is transparent to the application, which just thinks it actually runs by itself in memory.

This shows just one process. In reality there are many processes, all mapping through the page table to RAM or the swap file.

Access to the swap file is very slow, so swapping is avoided where possible.

The system may use *lazy allocation* - pages are not allocated an entry in the page table until used. In the diagram, page 3 has no allocation - because it is not actually used. This saves time and memory.

# Appendix

This is the decOut listing used in the section about arithmetic.

The code outputs the contents of register ax, in decimal format.

In a loop it divides ax by 10, to obtain the separate decimal digits, which are stored in memory (and counted)

It then loops through that memory, loads each digit into al, and calls digitOut to display it. This adds 48 to convert it to ASCII, then calls the Linux software interrupt to output it:

```
#     decOut.s
#     print out ax in decimal format

.text
.global intOut

      ###################### sub-routines start here

intOut:     ## output integer in ax, in decimal
      push %cx
      push %rax
      mov $10, %bx            # we divide by 10 in loop
      mov $0, %rdx            # used by div instruction

      mov $0, %cx             # counting digits found
      mov $buffer+10, %r8     # pointer to where digits stored
again:
      div %bx                 # divide by 10
      mov %dl, (%r8)          # store remainder in memory
      mov $0, %rdx            # clear dx
      dec %r8                 # adjust memory pointer
      inc %cx                 # increment digitCount
      cmp $0, %ax             # is ax zero?
      jne again               # jump on not equal to do next digit


      ## now to print out digitCount digits.
      ## They start one back from %r8
      inc %r8                 # point to first digit
next: mov (%r8), %al          # fetch it into al
      call digitOut           # print it
      inc %r8                 # point to next digit
      dec %cx                 # decrease digitCount
      cmp $0, %cx             # if not zero..
      jne next                # do it again
      call newline
      pop %rax
```

```
      pop %cx
      ret

newline:    ## output a new line
      mov $buff2, %ecx  # start of data to be output
      movb $10, (%ecx)  # ASCII line feed
      mov $1, %edx            # character count : just 1
      movl $1,%ebx            # to stdout
      movl $4,%eax            # system call number (sys_write)
      int $0x80               # call kernel
      ret

digitOut:   ## output the digit in al
      push %rcx         # preserve registers
      push %rbx
      push %rdx
      push %r8
      add $48, %al            # convert digit to ascii
      mov $buff2, %ecx  # put pointer to memory space in ecx
      mov %al, (%ecx)   # store in memory (indirect addressing)
      ## output
      mov $buff2, %ecx  # start of data to be output
      mov $1, %edx            # character count : just 1
      movl $1,%ebx            # to stdout
      movl $4,%eax            # system call number (sys_write)
      int $0x80               # call kernel
      pop %r8
      pop %rdx          # restore registers
      pop %rbx
      pop %rcx
      ret

.data                           # section declaration

buffer:    .zero 100          # 100 bytes reserved, filled with 0
buff2:     .zero 100
```