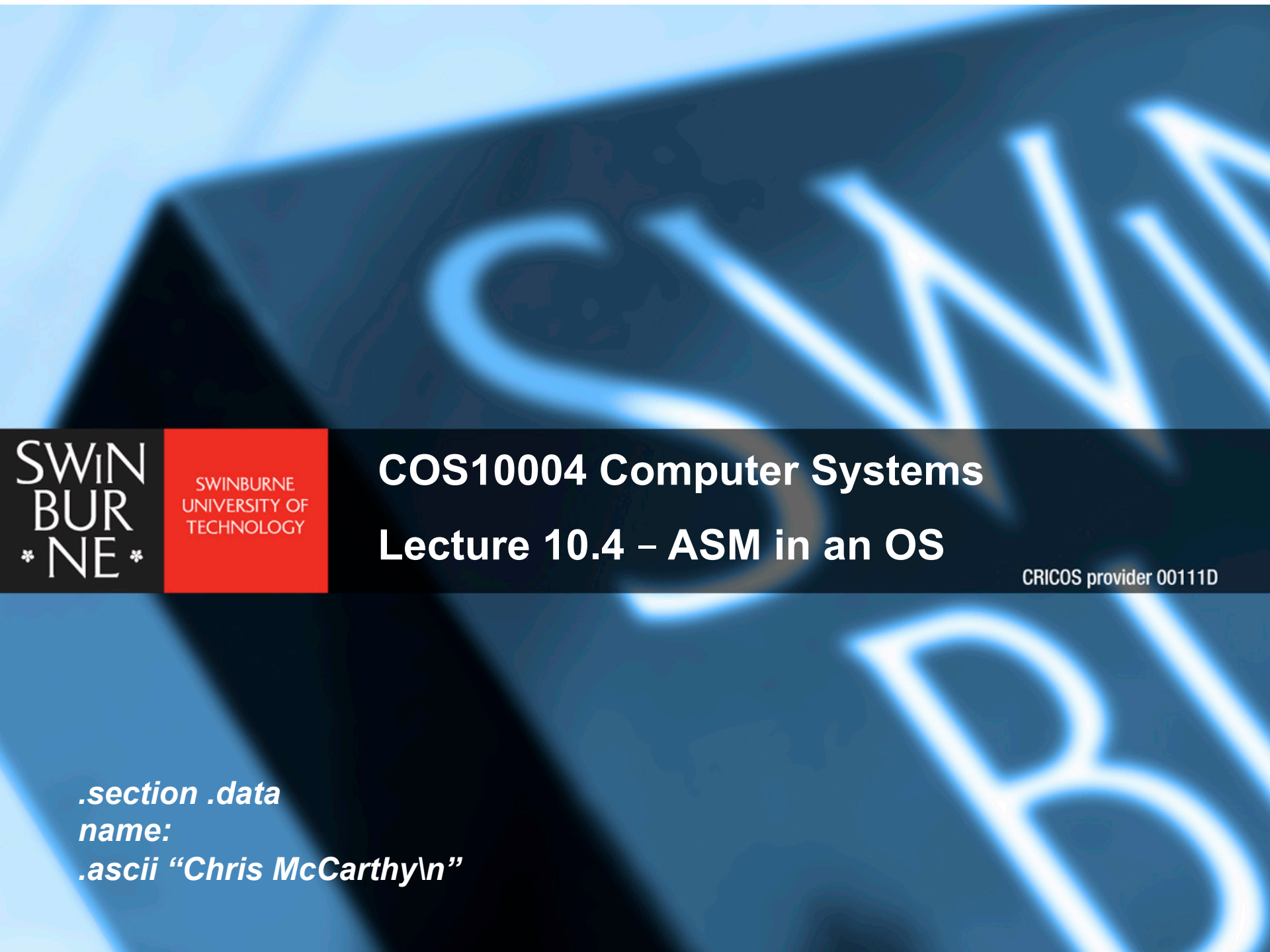


COS10004 Computer Systems

Lecture 10.4 – ASM in an OS


```
.section .data
name:
.ascii “Chris McCarthy\n”
```

Why load an OS?

Drivers

Hardware support

Abstraction

Tools

These things make program development easier, because we don’t have to worry about “minor” hardware changes (2B, 3B, 3B+, 4B).

So...

- We can use a “mature” Linux distribution (Raspberry Pi OS – previously Raspbian) optimised for the R Pi to manage the HID hardware layer that handles USB devices.
- **`Works with model 2B, 3B, 3B+, 4B`**

Do this at home.

- So...
- Back up all of your ASM code, and SD card contents
- Wipe your SD card (use SDFormatter4 from https://www.sdcard.org/downloads/formatter_4/)
- Download and install NOOBS Lite or Rpi OS (http://www.raspberrypi.org/downloads/). If you can connect to the internet, get NOOBS Lite (faster install); if not, get RPi OS.
- Unzip NOOBS or Rpi OS (either) to your SD card
- Put it in your Pi, connect an HDMI screen, Ethernet, keyboard and a mouse (simple cheap devices are more compatible).

Do this at home.

- Boot NOOBS (it will re-partition your card) and or install Rpi OS by following the instructions here: http://www.raspberrypi.org/documentation/installation/installing-images/README.md)
  - Let it install
  - Boot into RPi OS.

The user name is **`pi`**

The password is **`raspberry`**

`sudo raspi-config`

Select 5 (internationalisation)

Select Change Locale

Select en-US.UTF8

Set en-US.UTF8 as the default

Select 5

Select Change Keyboard Layout

Accept the default

`sudo reboot`

IMPORTANT:
select the US keyboard, English (US) language, so that you can get the # symbol instead of the pound symbol

IMPORTANT:
If you forget, you can change the settings after logging in:
`sudo raspi-config`
and select en-US.UTF8

Editing ARM code

- *Leafpad* is the local name for the Linux Notepad editor.
- To assemble, you will need to use LXTerminal, and be in the correct folder: /home/pi is a good place to be.
  - **`cd /home/pi`** or **`cd`**
  - Edit your source code
  - assemble and link:

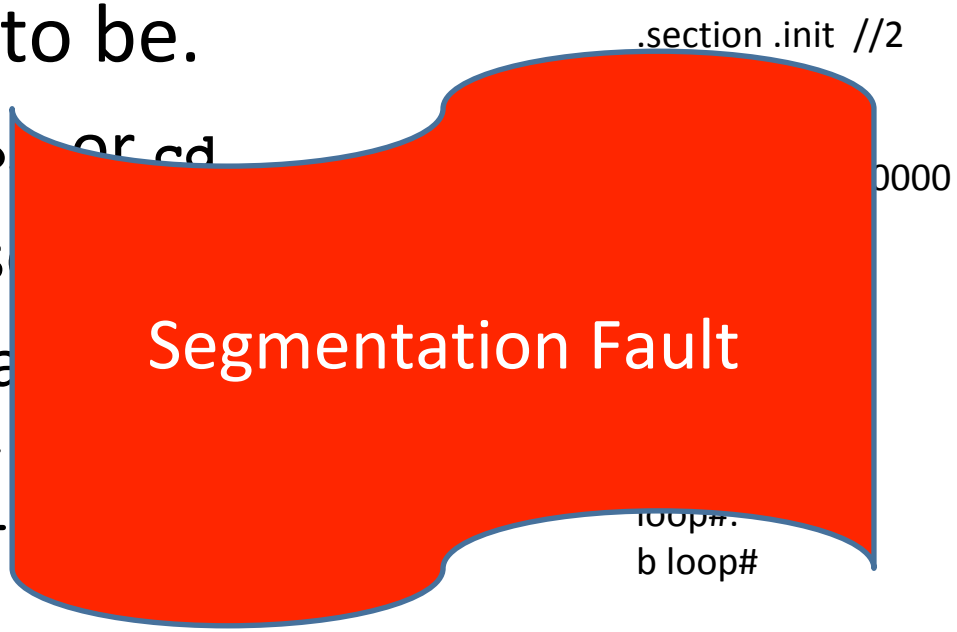
```
as -o ok01.o ok01.s
ld -o ok01 ok01.o
```

```
.section .init  //2
.globl _start
_start:
ldr r0,=0x3F200000
mov r1,#1
lsl r1,#24
 str r1,[r0,#4]
mov r1,#1
lsl r1,#18
str r1,[r0,#28]
loop#:
b loop#
```

Editing ARM code

- *Leafpad* is the local name for the Linux Notepad editor.
- To assemble, you will need to use LXTerminal, and be in the correct folder. /home/pi is a good place to be.
  - `cd /home/p` or `cd`
  - Edit your s
  - assemble a

```
as -o ok01
ld -o ok01
```



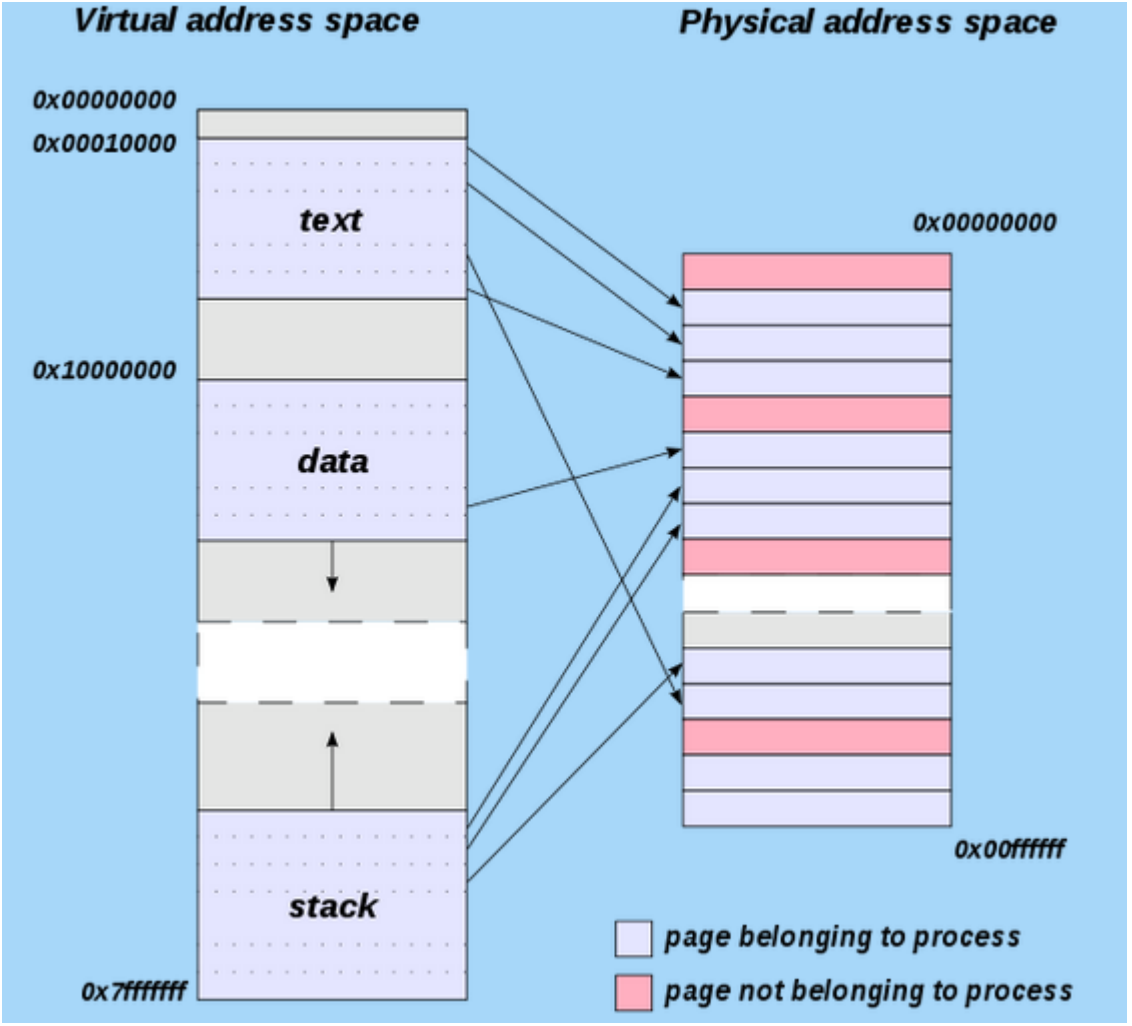
Doesn’t work ? Here’s why..

- The down side of using an operating system is that we lose direct access to hardware addresses.
- The OS maps process (running program) memory addresses to virtual memory (2-3GB) to protect programs from each other.
- No program (except the kernel) can access the “real” addresses.

Linux (i.e. RPi OS) maps memory for each process (running program) to different pages of physical memory, but not all addresses are mapped.

We get a segmentation fault when our code tries to access a memory location that it is not mapped to.

Picture from http://en.wikipedia.org/wiki/Address_space



What about the hardware?

- We can still access the GPIO pins, but we must ask the OS to do it for us.
- In Linux this means setting text files buried deep in the file system to contain 1 or 0.
- Here is a bash shell script example:

```
#!/bin/bash

echo "18" > /sys/class/gpio/export
echo "out" > /sys/class/gpio/direction
i=1
While [ $i –le 5 ]
do
  echo "1" > /sys/class/gpio/gpio18/value
  sleep 1s
  echo "0" > /sys/class/gpio/gpio18/value
  i=`expr $i + 1`
sleep 1s
done
```

Using libraries

- There are C, C++, Java and Python libraries which will allow us to change GPIO pins, access the screen, NIC and USB. http://makezine.com/projects/tutorial-raspberry-pi-gpio-pins-and-python/, http://hertaville.com/2014/07/07/rpimmapgpio/
- These allow us to use high level languages to perform complex operations such as controlling plug-in boards, LED matrix displays and reading digital information from the header pins.
- Assembly support for hardware access in ASM is rare, probably because if you're going to use an OS, you might as well use a high-level language.
  - You've already lost the advantages (speed) of using ASM to access hardware.

Back to assembly

- There are still valid reasons for coding in Assembly, even with an operating system.
  - Direct control (speed) over how the code is optimised (or not) because we are not dependent on the compiler.
  - Smaller executables because we choose which libraries are included, and because no debugging information is stored in the executables.
  - And don't forget all those things evil hackers do.

helloworld.c *vs* helloworld.s

```
#include <stdio.h>
int main()
{
  puts("Hello world C");
  return 0;
}
```

```
.section .data
message:
.ascii "Hello world S\n"
.section .text
.globl _start
_start:
mov r7,#4 //sys_write
mov r0,#1 //stdout
mov r2,#14 //length
ldr r1,=message
swi 0          //INTerrupt
mov r7,#1 //exit
swi 0
```

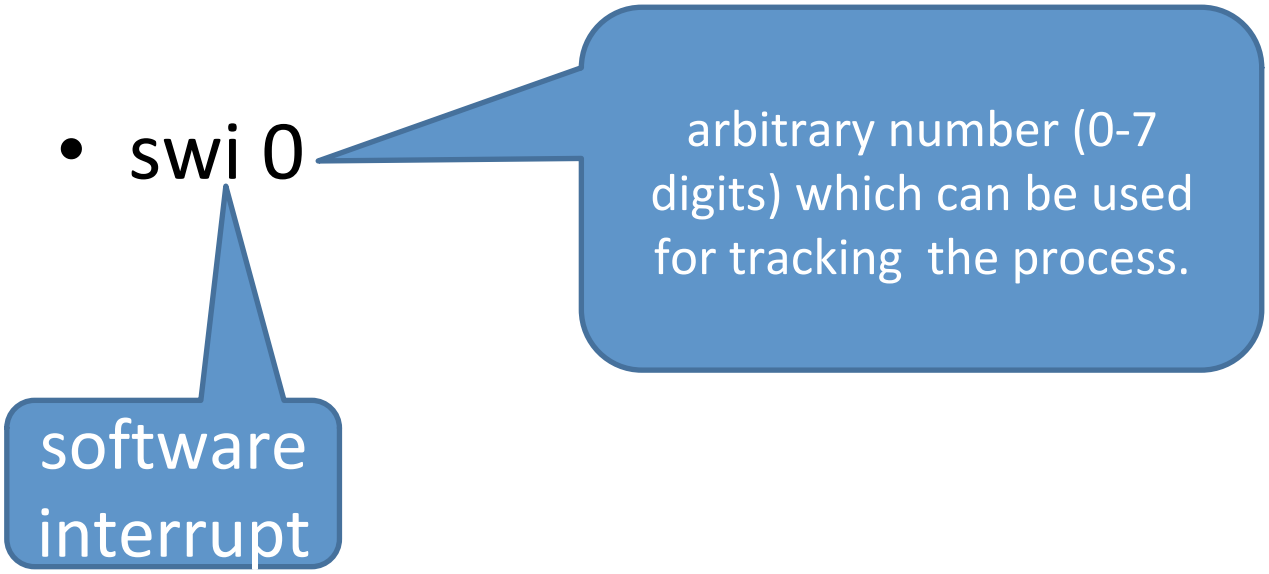
./hello_c *vs* ./hello_s

- Size:
  - C code: 5530 bytes big
  - ASM code: 893 bytes small

- Speed: (time ./hello_x) (real time)
  - C code: 0.013s fast
  - ASM code: 0.009s faster

A new instruction

- swi 0



can also use “svc 0”

more here: http://www.heyrick.co.uk/assembler/swi.html

details

Rather than writing to memory locations, we ask the OS to do things.

The OS has libraries full of system calls

- functions which can be called by:
  - pre-loading registers r0-r3, and then
  - triggered by making an interrupt.
- There are standard conventions for this:
- mov r7,#4 //ARM syscall number for write (the operation)
  mov r0,#1 //1st parameter (output device)
  ldr r1,=message //2nd param (pointer to the array)
  mov r2,#14 //3rd param (length of the array)

ARM Linux sys calls

r7	Name	r0	r1	r2
1	exit	exit code (r0)		
3	read	1 (STDIN)	Pointer to char array (.rodata, .data, .text or .bss)	length
4	write	1 (STDOUT)	Pointer to char array (.bss)	length

Sys calls protect the OS

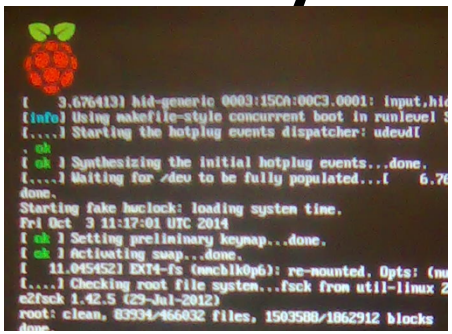
- Here are some for Linux on Intel CPUs:
- http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

- System calls run in the context of the kernel (the OS itself) and are therefore more powerful and faster than ordinary functions.

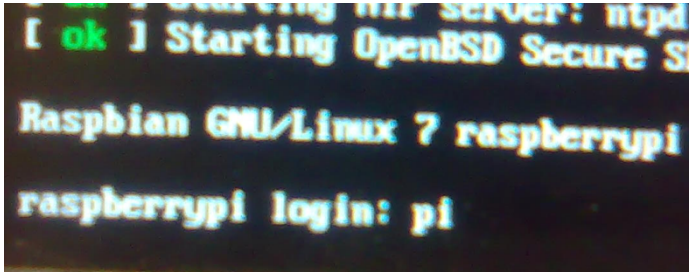
- They separate the (untrusted) user/programmer from the trusted operating system.
  - Each sys call is actually a compiled C function.

Do all your work in RPi OS

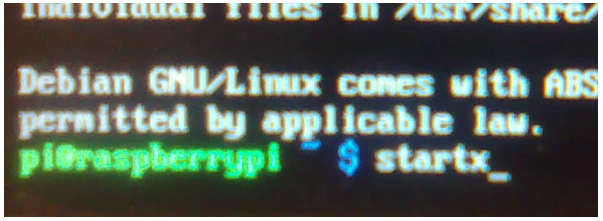
- Boot



- Log in (username: **`pi`**
  password: **`raspberry`**)



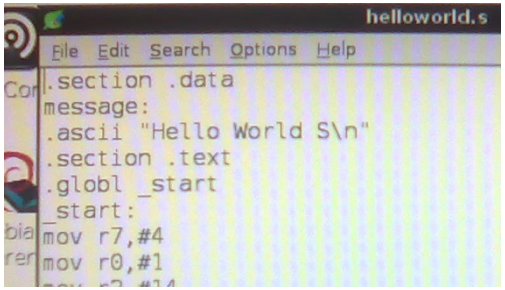
- Launch the GUI (**`startx`**)



- Open LXTerminal

Edit with Leafpad

- Edit with Leafpad

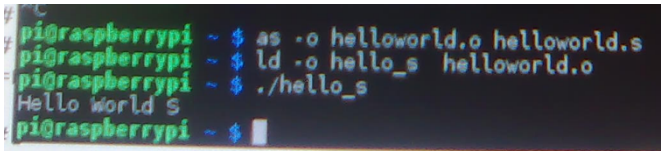


- Compile in the LXTerminal console:

```
as -o helloworld.o helloworld.s
ld -o hello_s helloworld.o
```

- Run:

```
./hello_s
```



Display Text: helloworld.s

```
.section .data
message:
.ascii "Hello world S\n"
.section .text
.globl _start
_start:
mov r7,#4 //sys_write
mov r0,#1 //stdout
mov r2,#14 //length
ldr r1,=message
swi 0      //interrupt
mov r7,#1 //exit
swi 0
```

Get a text from the kbd

```
/*getchar   returns r0 – ascii code */
.section .bss //statically allocated and pre-zeroed memory
.comm buffer, 48  //up to 48 bytes
.section .data
msg:
    .ascii "Enter a number: "
   msgLen = . – msg  //.- returns the length of a string that you have
just declared
.section .text
.globl _start
_start:
  mov r0, #1 // stdout -  print program's opening message
  ldr r1, =msg
  ldr r2, =msgLen
  mov r7, #4
swi #0
  mov r0, #1 // read from stdin
  ldr r1, =buffer
  mov r2, #0x30 // number of bytes to read
  mov r7, #3 // 3 is the "read" syscall
swi 0  //can also use svc 0
```

read and display text: greet.s

```
// greet.s - a little asm greeter.
.section .bss
.comm buffer, 48
.section .data
msg: .ascii "** Greeter **\nPlease enter your name: "
msgLen = . - msg
msg2: .ascii "Hello "
msg2Len = . – msg2
.section .text
.globl _start
_start:
mov r0, #1 // print program's opening message
ldr r1, =msg
ldr r2, =msgLen
mov r7, #4
svc #0
mov r7, #3 // read syscall
mov r0, #1
ldr r1, =buffer
mov r2, #0x30
svc #0
mov r0, #1 // print msg2
ldr r1, =msg2
ldr r2, =msg2Len
mov r7, #4
svc #0
mov r0, #1 // now print the user input
ldr r1, =buffer
mov r2, #0x30
mov r7, #4
svc #0
mov r7, #1 //exit syscall
svc 0 // wake kernel
.end

//http://raspberrypiassembly.wordpress.com/
```

count chars

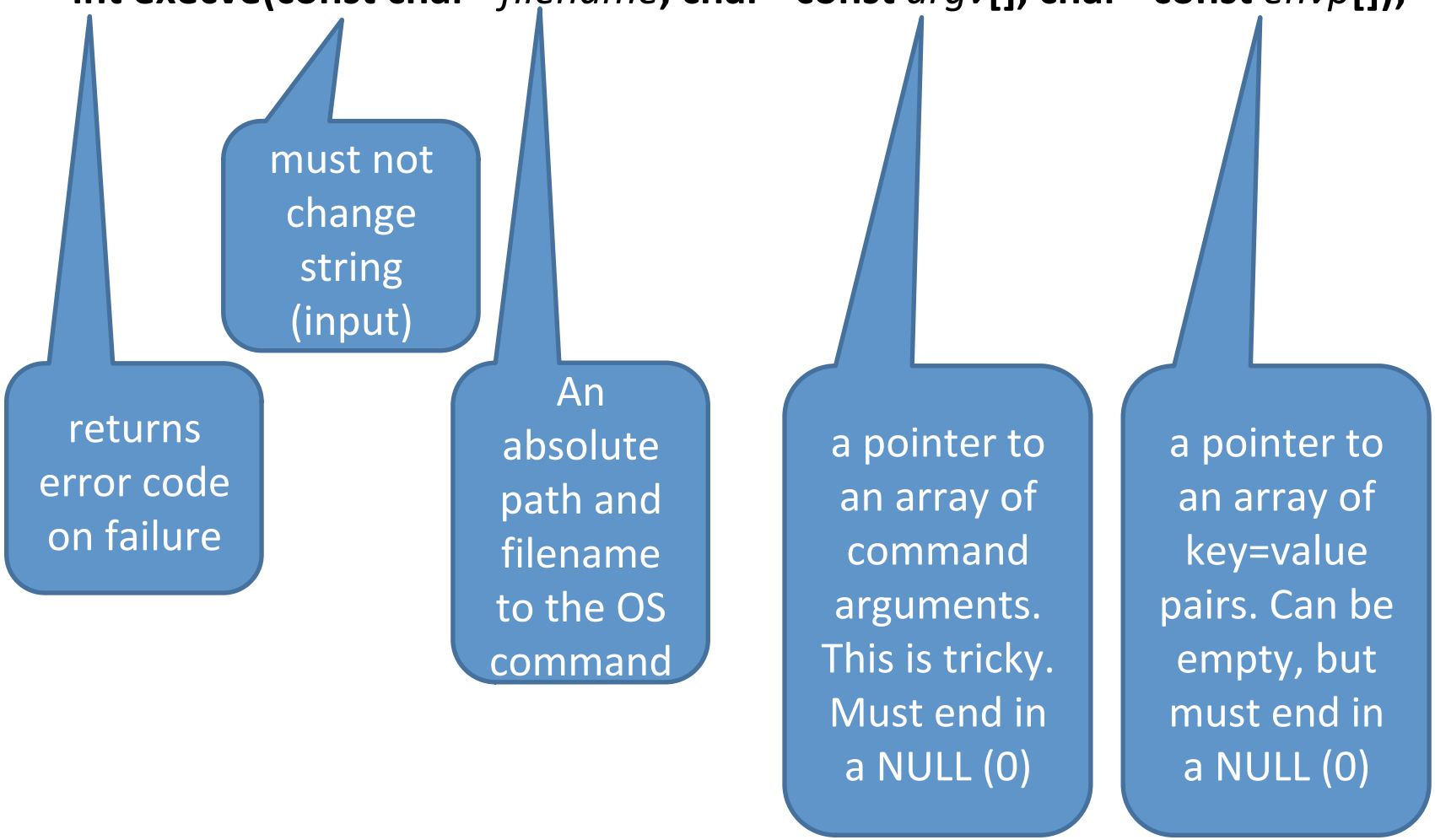
```
/*getchar4.s
* count chars in string
* display with echo #?*/
.section .bss //
.comm buffer, 128
.section .data
message:
.ascii "Enter a character:\n"
msgLen =.-message //or count characters
.section .text
.globl _start
_start:
mov r7,#4 //write
mov r0,#1 //device
ldr r2,=msgLen  //length
ldr r1,=message  //ptr
swi 0 //interrupt
mov r7,#3  //read
mov r0,#1  //device
ldr r1,=buffer
mov r2,#127  // max no. of chars read //
remaining chars are left in kbd
swi 0 //get length
ldr r4,=buffer //copy the pointer
mov r5,#0  //index
mov r6,#0  //sentinel value (NULL)
loop:
add r5,#1
ldr r7,[r4,r5] //r7 = *(r4+r5)
cmp r7,r6 //is this char == 0
bne loop
//r5 is now the location of the null
//print
mov r7,#4
mov r0,#1
mov r2,#127
ldr r1,=buffer
swi 0//exit
sub r5,#1
//now = number of chars before null
mov r0,r5  //return count to OS
mov r7,#1    //exit(count)
swi 0
```

ARM Linux sys calls

r7	Name	r0	r1	r2
1	exit	exit code (r0)		
3	read	1 (STDIN)	Pointer to char array (.rodata, .data, .text or .bss)	length
4	write	1 (STDOUT)	Pointer to char array (.bss)	length
11	**execve** (executes OS command)	Path or command	Pointer to array of args (null-terminated)	Pointer to array of args (null-terminated)

execve() – standard OS syscall

int execve(const char ****filename*, char *const** *argv*[], **char *const** *envp*[]);**



ASM... pointers

- Dereferencing pointers in ASM is easy – put the memory location inside [ ] and you get the value inside.
- Need to get the 5th element of an array?

```
//assume an array of single bytes
mov r0,[address,#4]

//dereference address + 4 bytes
```

But how do we get the address?

- ASM has no syntax like '&' in C/C++).
  - It does have **`adr r0,x`** – copies the address of x into r0 – but that would be too easy!
- Can't get an address of a register-r0,r1... they are hardware.
- so...Have to copy a value into memory...
  - but where?

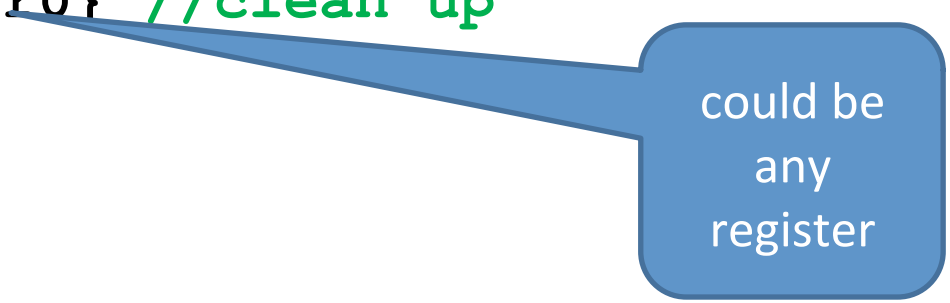
sp

- The stack pointer holds the location of the software stack pointer
- It points to the next free space in RAM
- Every time we push, it changes (up or down depending on the hardware)
- We can use the value in sp like a variable.

But how do we get the address?

- `push {register}` copies the value onto the stack
- `sp` (a register) holds the address of the top of the stack.
- `sp` is incremented by 1 word size each time we do a push.
- We can read the value in sp.

```
ldr r0,=100
push {r0}
mov r0,sp  //r0 now contains the location of the
100
//r0 is now a pointer to the value we put in r0
...
pop {r0} //clean up
```



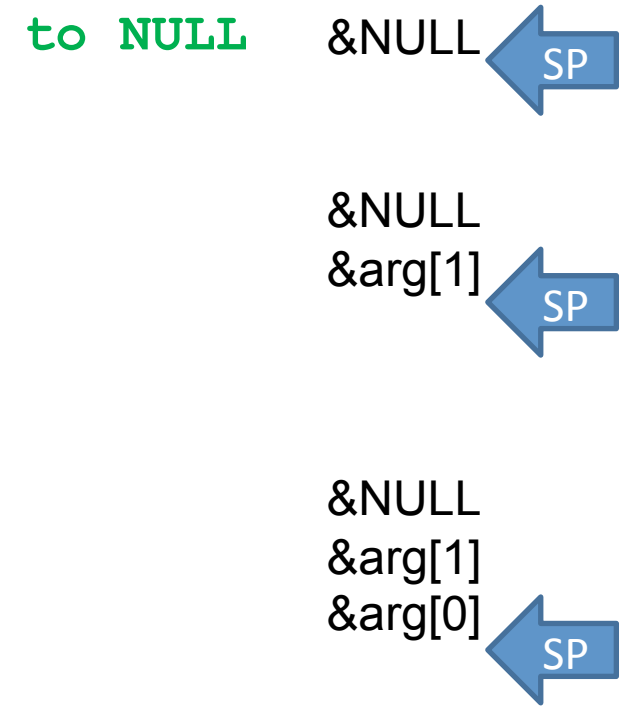
Works with ascii too!

```
.section .rodata  //read-only
path:  //label for our string
.string "/bin/ls"
//get pointer to text
ldr r0,=path //r0 now points to first char (char*
r0)
push {r0}
mov r0,sp  //r0 now points to a pointer to the first
char (char **r0 or char *r0[ ])
```

Loading up a pointer to an array of strings (char***)

```
ldr r2,=0  //NULL = 0
push {r2}
mov r2,sp  //r2 now points to NULL
//args
ldr r1,=arg1
push {r1} //add arg[0]
ldr r0,=arg0
push {r0}
mov r1,sp  //now points to
{arg[0],arg[1], null}
```

Stack:



See if it works?

- Assemble
- Use ObjDump to decompile and see where the variables were put
- **`objdump -d -marm -S <file>.o`**
  - puts “decompiled” asm on the screen
  - should find labels containing values,
  - and pointers to these labels

Having Trouble?

- gdb is the Linux debugger.
- gdb program – launch program in debugger
- (gdb) – the command prompt in gdb
- (gdb) run – run the program
- (gdb) step – run one line at a time
- (gdb) frame – show stack pointer
- (gdb) list 1,10 – list lines 1-10
- (gdb) info locals – show local variables
- (gdb) quit – exit the debugger
  - heaps more commands in gdbcomm.txt

Having trouble? Run home to C

- Write the program in C
- Compile to exe, test, refine
- Compile to ASM

  `gcc -Wall -Wextra -S -o <file>.s <file>.c`
- Compare the generated ASM (.s) file with yours.

The C program

```
#include <unistd.h>
int main()
{
char *program = "/bin/uname";
char *args[3]={"/bin/uname","-a",NULL};
execve(program, args, NULL);
return 0;
}
```

The generated ASM

```
.arch armv6
.eabi_attribute 27, 3
.eabi_attribute 28, 1
.fpu vfp        .eabi_attribute 20, 1
.eabi_attribute 21, 1
.eabi_attribute 23, 3
.eabi_attribute 24, 1
.eabi_attribute 25, 1
.eabi_attribute 26, 2
.eabi_attribute 30, 6
.eabi_attribute 18, 4
.file           "name.c"
.section        .rodata
.align          2
.LC0:
.ascii          "/bin/uname\000“
.align          2
.LC1:
.ascii          "-a\000"
.align          2
.LC2:           .word
.LC0            .word
.LC1            .word           0
.text
.align          2
.global         main
.type           main, %function
main:           @ args = 0, pretend = 0, frame = 16        @
frame_needed = 1, uses_anonymous_args = 0
stmfd           sp!, {fp, lr}
add             fp, sp, #4
sub             sp, sp, #16
ldr     r3, .L2
str     r3, [fp, #-8]
ldr     r2, .L2+4
sub     r3, fp, #20
ldmia   r2, {r0, r1, r2}
stmia   r3, {r0, r1, r2}
sub     r3, fp, #20
ldr     r0, [fp, #-8]
mov     r1, r3
mov     r2, #0
bl      execve
mov     r3, #0
mov     r0, r3
sub     sp, fp, #4
ldmfd   sp!, {fp, pc}
.L3:
.align  2
.L2:    .word
.LC0    .word
.LC2    .size
main, .-main
.ident
"GCC: (Debian 4.6.3-14+rpi1) 4.6.3“
.section .note
.GNU-stack,"",%progbits
```

The generated ASM

```
.arch armv6
.eabi_attribute 27, 3
.eabi_attribute 28, 1
.fpu vfp        .eabi_attribute 20, 1
.eabi_attribute 21, 1
.eabi_attribute 23, 3
.eabi_attribute 24, 1
.eabi_attribute 25, 1
.eabi_attribute 26, 2
.eabi_attribute 30, 6
.eabi_attribute 18, 4
.file           "name.c"
.section        .rodata
.align          2
.LC0:
.ascii          "/bin/uname\000“
.align          2
.LC1:
.ascii          "-a\000"
.align          2
.LC2:           .word
.LC0            .word
.LC1            .word           0
.text
.align          2
.global         main
.type           main, %function
main:           @ args = 0, pretend = 0, frame = 16      @
frame_needed = 1, uses_anonymous_args = 0
stmfd           sp!, {fp, lr}
add             fp, sp, #4
sub             sp, sp, #16
ldr     r3, .L2
str     r3, [fp, #-8]
ldr     r2, .L2+4
sub     r3, fp, #20
ldmia   r2, {r0, r1, r2}
stmia   r3, {r0, r1, r2}
sub     r3, fp, #20
ldr     r0, [fp, #-8]
mov     r1, r3
mov     r2, #0
bl      execve
mov     r3, #0
mov     r0, r3
sub     sp, fp, #4
ldmfd   sp!, {fp, pc}
.L3:
.align  2
.L2:    .word
.LC0    .word
.LC2    .size
main, .-main
.ident
"GCC: (Debian 4.6.3-14+rpi1) 4.6.3“
.section .note
.GNU-stack,"",%progbits
```

don’t need this red stuff

The generated ASM

```
.section .rodata
.align   2
.LC0:
.ascii   "/bin/uname\000"
.align   2
.LC1:
.ascii   "-a\000"
.align   2
.LC2:    .word
.LC0     .word
.LC1     .word   0
.text
.align   2
.global  main
.type    main, %function
```

NULL-terminated arg[0]

NULL-terminated arg[1]

NULL-arg[2]

```
stmfd       sp!, {fp, lr}
add  fp, sp, #4
sub  sp, sp, #16
ldr  r3, .L2
str  r3, [fp, #-8]
ldr  r2, .L2+4
sub  r3, fp, #20
ldmia       r2, {r0, r1, r3}
stmia       r3, {r0, r1,
sub  r3, fp, #20
ldr  r0, [fp, #-8]
mov r1, r3
mov r2, #0
bl   execve
mov r3, #0
mov r0, r3
sub  sp, fp, #4
ldmfd       sp!, {fp, pc}
```

relative location of LC1

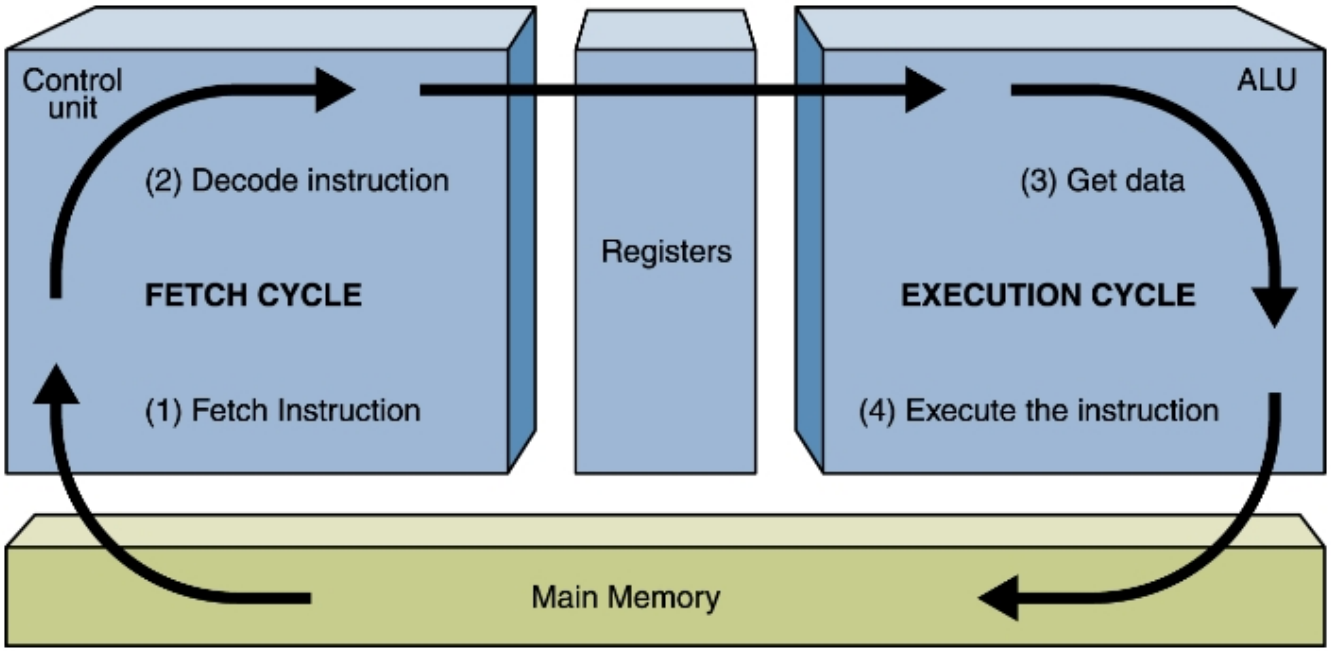
relative location of LC0

relative location of LC0

calling the right function

returns 0?

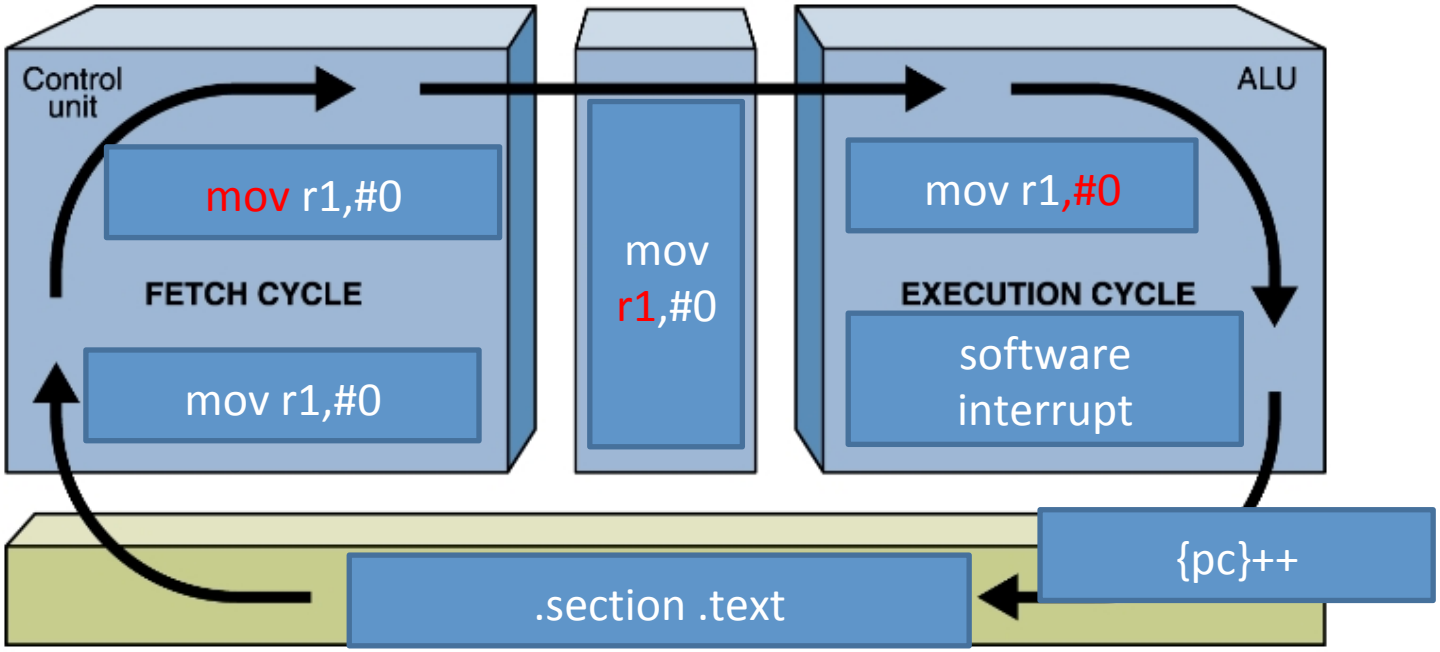
sp, lr, pc, my brain hurts



http://www.cs.hofstra.edu/~cscvjc/Fall06/Slides/Sess09/img22.html

sp, lr, pc, my brain hurts

Figure 5.3 The Fetch-Execute Cycle



http://www.cs.hofstra.edu/~cscvjc/Fall06/Slides/Sess09/img22.html

pc

- pc = program counter (instruction pointer)
- It's where we are up to in the code – the next operation to perform
- We can jump around by writing to it
- we can control it directly! DON'T

lr

- Link register
- Holds the address of the instruction to return to when a function is called
- If we do a **bl**, **lr** is set to **pc**+1 just before the branch
- inside the function we can return any time by setting **pc** (the next instruction) to **lr**
- We use the stack to do this.

Example code

- Lets walk through some example code.
  - Your task: work out what it is doing

```
.section .bss
.comm buffer, 48 // reserve 48 byte buffer
.section .data
msg: .ascii "Enter a number: "
msgLen = . - msg
.section .text
.globl _start
_start:
mov r0, #1
ldr r1, =msg
ldr r2, =msgLen
mov r7, #4
svc #0 // get input
mov r7, #3 // 3 is the "read" syscall
mov r0, #1
ldr r1, =buffer
mov r2, #0x30
svc #0
ldrb r2, [r1]
mov r3, #0
pushDigits:
stmfd sp!, {r2}
add r3, r3, #1
ldrb r2, [r1, #1]!
cmp r2, #0xA  //if r2==10 convert digits else push digits
beq convDigits
bne pushDigit
convDigits:
mov r4, #1
mov r0, #0
mov r6, #0
jumpBack:
ldmfd sp!, {r2}
cmp r2, #0x30 //if r2 < 48 goto error
blt error
cmp r2, #0x39 //if r2 > 57 goto error
bgt error
sub r2, r2, #0x30 // take away 48 ... the digit value
mul r2, r4, r2 // multiply it ...
add r0, r0, r2
add r6, r6, #1
cmp r6, r3 // check to see if done // if r6==r3 exit
beq exit
mov r5, r4, lsl #3
add r4, r5, r4, lsl #1 // x * 8 + x * 2 = x * 10
bal jumpBack // Least significant byte available via "echo $?"
error:
mov r0, #-1
bal exit
exit:
mov r7, #1 // exit syscall
svc #0 // wake kernel
.end
```

print instructions

read in a char

SDMFD sp!, {r2}
Push onto a Full
Descending Stack.

10 is the ASCII code for
line-feed (return)

LDMFD sp!, {r2}
Pop from a Full
Descending Stack.

Read one numeric digit at a time and convert to a number

48 is the ASCII code for 0,
59 is the ASCII code for 9

What we have so far...

- It reads in numbers from the keyboard
- pushes each character onto the stack

```
.section .bss
.comm buffer, 48 // reserve 48 byte buffer
.section .data
msg: .ascii "Enter a number: "
msgLen = . - msg
.section .text
.globl _start
_start:
mov r0, #1
ldr r1, =msg
ldr r2, =msgLen
mov r7, #4
svc #0 // get input
mov r7, #3 // 3 is the "read" syscall
mov r0, #1
ldr r1, =buffer
mov r2, #0x30
svc #0
ldrb r2, [r1]
mov r3, #0
pushDigits:
stmfd sp!, {r2}
add r3, r3, #1
ldrb r2, [r1, #1]!
cmp r2, #0xA //if r2==10 convert digits else push digits
beq convDigits
bne pushDigit
convDigits:
mov r4, #1
mov r0, #0
mov r6, #0
jumpBack:
ldmfd sp!, {r2}
cmp r2, #0x30 //if r2 < 48 goto error
blt error
cmp r2, #0x39 //if r2 > 57 goto error
bgt error
sub r2, r2, #0x30 // take away 48, to get the digit value
mul r2, r4, r2 // multiply it by r4
add r0, r0, r2
add r6, r6, #1
cmp r6, r3 // check to see if done // if r6==r3 exit
beq exit
mov r5, r4, lsl #3
add r4, r5, r4, lsl #1 // x * 8 + x * 2 = x * 10
bal jumpBack // Least significant byte available via "echo $?"
error:
mov r0, #-1
bal exit
exit:
mov r7, #1 // exit syscall
svc #0 // wake kernel
.end
```

print instructions

read in a char

LDMFD sp!, {r2}
Push onto a Full
Descending Stack.

convert on Enter

10 is the ASCII code for line-feed (return)

LDMFD sp!, {r2}
Pop from a Full
Descending Stack.

Read one numeric digit at a time and convert to a number

48 is the ASCII code for 0, 59 is the ASCII code for 9

- R3 is a count in a loop
- Counts the number of digits entered before <Enter> key
- r6 counts up from 0 to r3
  - does something to each digit

```
.section .bss
.comm buffer, 48 // reserve 48 byte buffer
.section .data
msg: .ascii "Enter a number: "
msgLen = . - msg
.section .text
.globl _start
_start:
mov r0, #1
ldr r1, =msg
ldr r2, =msgLen
mov r7, #4
svc #0 // get input
mov r7, #3 // 3 is the "read" syscall
mov r0, #1
ldr r1, =buffer
mov r2, #0x30
svc #0
ldrb r2, [r1]
mov r3, #0
pushDigits:
stmfd sp!, {r2}
add r3, r3, #1
ldrb r2, [r1, #1]!
cmp r2, #0xA //if r2==10 convert digits else push digits
beq convDigits
bne pushDigit
convDigits:
mov r4, #1
mov r0, #0
mov r6, #0
jumpBack:
ldmfd sp!, {r2}
cmp r2, #0x30 //if r2 < 48 goto error
blt error
cmp r2, #0x39 //if r2 > 57 goto error
bgt error
sub r2, r2, #0x30 // take away 48, to get the digit value
mul r2, r4, r2 // multiply it by r4
add r0, r0, r2
add r6, r6, #1
cmp r6, r3 // check to see if done // if r6==r3 exit
beq exit
mov r5, r4, lsl #3
add r4, r5, r4, lsl #1 // x * 8 + x * 2 = x * 10
bal jumpBack // Least significant byte available via "echo $?"
error:
mov r0, #-1
bal exit
exit:
mov r7, #1 // exit syscall
svc #0 // wake kernel
.end
```

print instructions

read in a char

convert on Enter

LDMFD sp!, {r2}
Push onto a Full Descending Stack.

LDMFD sp!, {r2}
Pop from a Full Descending Stack.

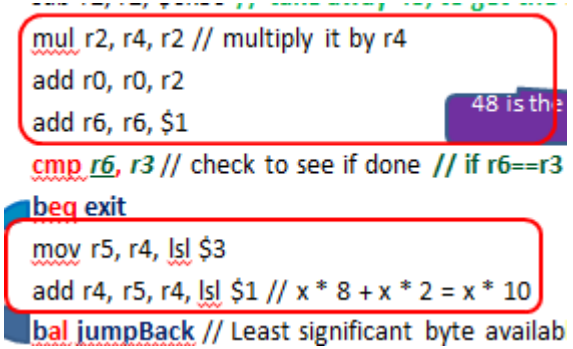
Read one numeric digit at a time and convert to a number

48 is the ASCII code for 0, 59 is the ASCII code for 9

10 is the ASCII code for line-feed (return)

Algorithm

- r0 contains the total
- r2 (digit)
- r6 digit place (1's, 10, 100, etc)
- Each digit is processed individually
  - (r6 counts from 0 to r3)



- r4 is shifted left by 3 (2^3 = 8)
- r4 is shifted left by 1 (2^1=2)
- Each run of the loop multiplies r4 by 10
- r2 (digit) is multiplied by 1, 10, 100... etc

```
.section .bss
.comm buffer, 48 // reserve 48 byte buffer
.section .data
msg: .ascii "Enter a number: "
msgLen = . - msg
.section .text
.globl _start
_start:
mov r0, #1
ldr r1, =msg
ldr r2, =msgLen
mov r7, #4
svc #0 // get input
mov r7, #3 // 3 is the "read" syscall
mov r0, #1
ldr r1, =buffer
mov r2, #0x30
svc #0
ldrb r2, [r1]
mov r3, #0
pushDigits:
stmfd sp!, {r2}
add r3, r3, #1
ldrb r2, [r1, #1]!
cmp r2, #0xA //if r2==10 convert digits else push digits
beq convDigits
bne pushDigit
convDigits:
mov r4, #1
mov r0, #0
mov r6, #0
jumpBack:
ldmfd sp!, {r2}
cmp r2, #0x30 //if r2 < 48 goto error
blt error
cmp r2, #0x39 //if r2 > 57 goto error
bgt error
sub r2, r2, #0x30 // take away 48, to get the digit value
mul r2, r4, r2 // multiply it by r4
add r0, r0, r2
add r6, r6, #1
cmp r6, r3 // check to see if done // if r6==r3 exit
beq exit
mov r5, r4, lsl #3 //r5 = r4*8  //r4=r5+r4*2
add r4, r5, r4, lsl #1 // x * 8 + x * 2 = x * 10
bal jumpBack // Least significant byte available via "echo $?"
error:
mov r0, #-1
bal exit
exit:
mov r7, #1 // exit syscall
svc #0 // wake kernel
.end
```

print instructions

read in a char

LDMFD sp!, {r2}
Push onto a Full
Descending Stack.

convert on Enter

10 is the ASCII code for line-feed (return)

LDMFD sp!, {r2}
Pop from a Full
Descending Stack.

Read one numeric digit at a time and convert to a number

48 is the ASCII code for 0, 59 is the ASCII code for 9

- Reads in a multi-digit number
- On <Enter>, converts each digit, multiplies each digit by 1, 10, 10*10, 10*10*10...
  (x = x*8+x*2)
- Adds each digit to r0
- returns product (r0) to the OS
- Amazingly convoluted, but it does maths on a string!

atoi()

- It is actually implementing the C library function atoi():
  - It asks for a number
  - reads a number as a string (array of characters)
  - Converts the string to its numerical (integer) value and puts it in r0
- return r0

Let's test it.

```
pi@raspberrypi ~ $ as -o mystery.o mystery.s
pi@raspberrypi ~ $ ld -o mystery mystery.o
pi@raspberrypi ~ $   ./mystery
Enter a number: 123
pi@raspberrypi ~ $ echo $?
123
pi@raspberrypi ~ $
```

"echo $?" prints the exit value of the last run command… so it works!