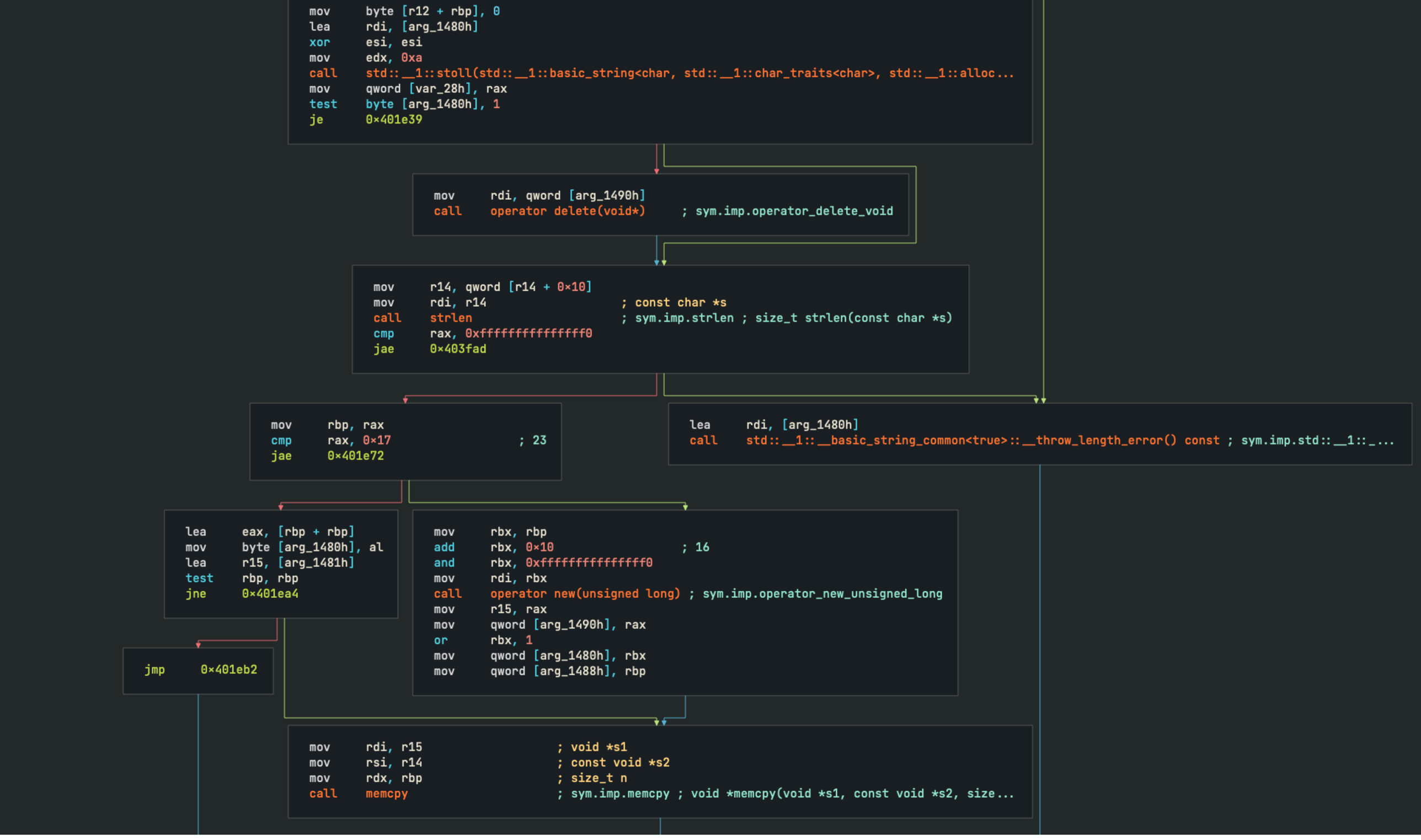


Introduction to the x86_64 architecture

Introduction to executable binaries

Antonin Portelli

Supported by UK EPSRC grants EP/V001329/1, EP/V001191/1, EP/V001477/1, EP/V001248/1, EP/V001140/1

- Course introduction
- Fundamentals on computer programs
- The x86_64 architecture
- Hello, World
- Wrap up

Course introduction

Outcomes

- Understanding x86_64 assembly and binary code
- Understand memory segmentation, heap & stack
- Understand x86_64 execution flow
- Understand executable binaries structure, static & dynamic linking mechanisms
- Apply the above to debug and instrument programs
- Apply the above to profile and plan optimisation of code

What this course will not (really) teach you

- How to write entire programs in assembly from scratch
- An exhaustive description of x86_64
- An exhaustive description of the ELF format
- Magic tricks to optimise code
- How the content of the course can be ported to other systems and architectures, I will purely assume x86_64 Linux which is the one of most common setup in HPC…

  … yes really, a GPU still needs a CPU to run

What this course will (hopefully) teach you

- How to sort out cryptic crashes and backtraces without writing a panicked email to your supervisor
- How to tell if a code is optimal or not, and how to plan for further optimisation
- How to get a healthy toolbox to investigate binaries comfortably
- How dumb computers really are and why writing very careful code is essential

Hacking challenge

- This is quite a “hacking” course, i.e. break stuff down and explore, even if you don’t get everything that is going on

- 5 teams solving problems with programs provided in binary form, and we’ll discuss the outcome on Friday

- Problems are a collection of password protected programs to hack

  (classic CTF challenge)

- I hope you’ll find it challenging! More info tomorrow

Course repository

- Git repository for the course
  https://github.com/aportelli/exalat-binary-lectures
- Contains slides, code examples, logs of demonstrations
- I will upload the challenge source code at the end

Readings

- Classic, fantastic textbook on executable binaries

  Hacking: the art of exploitation 2nd ed., J. Erickson, 1593271441 (2008)

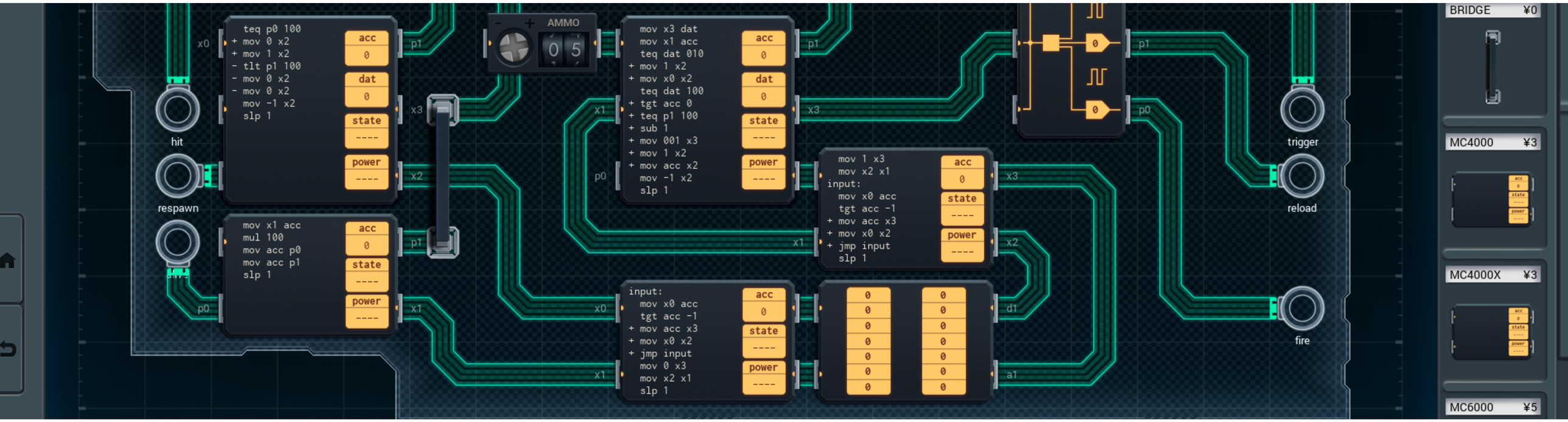
- Bible of computer architecture

  Computer Architecture: A Quantitative Approach 6th ed., J. Hennessy and A. Patterson, 0128119055 (2017)

- Huge amount of online resources, especially in the reverse-engineering and security community (search GitHub, reddit, Medium, …)

Games!

- Some Zachtronics games require to solve puzzles writing assembly (TIS-100, Shenzhen I/O, Exapunks)



- Silicon Zeroes is also great to understand how CPUs are designed from scratch

Disclaimer

I believe that playing around with executable binaries, including manipulating them to change their behaviour is one of the best way to understand how computer works at the lowest level. This is highly relevant for HPC practices.

The cybersecurity and reverse engineering communities are one of the richest on these aspects. This course and some of its references touches on techniques used in offensive & defensive cybersecurity.

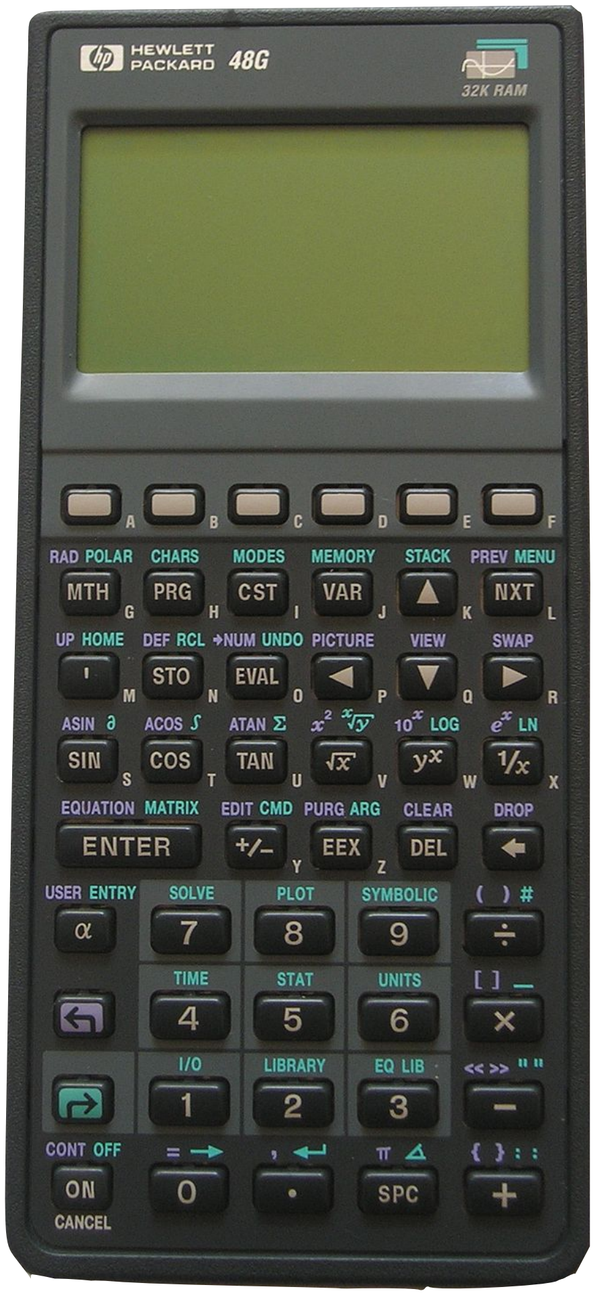
However everything discussed in this course is purely for pedagogical purposes, and I do condemn strongly any unethical and/or illegal use of computing knowledge.

Fundamentals on computer programs

“The inside of a computer is as dumb as hell but it goes like mad!”

– Richard P. Feynman

“as dumb as hell”



HP48G calculator

- The CPU is essentially **a calculator**
- It can read/write numbers from memory and perform arithmetic operations
- Such operations are internally represented as **instructions**
- A specification for a set of instruction is an **Instruction Set Architecture (ISA)**
- A CPU is a micro-electronic implementation of an ISA

“it goes like mad!”

```
VRTSTART        TS      WCHVERT
# Page 801
                CAF     TWO             # WCHPHASE = 2 ---> VERTICAL: P65,P66,P67
                TS      WCHPHOLD
                TS      WCHPHASE
                TC      BANKCALL        # TEMPORARY, I HOPE HOPE HOPE
                CADR    STOPRATE        # TEMPORARY, I HOPE HOPE HOPE
                TC      DOWNFLAG        # PERMIT X-AXIS OVERRIDE
                ADRES   XOVINFLG
```

Apollo 11 lunar landing routine

- Computers are **programmable**
- A sequence of instructions, **a program**, can be stored in memory and executed by the CPU
- Programs can be executed **extremely fast,** nowadays billions of instructions per second
- Programs can run and create other programs, and **complexity emerges through automation**

Executable binaries

- Instructions can be encoded in binary, as specified by the ISA, and stored in a file, **an executable binary**.

- When the file is loaded in memory to be executed, the processor loop over the **instruction cycle**

1. **Fetch**: the next instruction is moved to an internal memory in the processor (instruction register)

2. **Decode**: the instruction binary form is decoded into a more relevant internal representation

3. **Execute**: signals and relevant data are sent to dedicated processing units in the chip

Programming from scratch

- Most minimal form of programming

  1. Write on “paper” your program using the ISA

  2. Code the program in a binary file following the ISA encoding specification

- It used to really be like that!

- **Prohibitively tedious** for complex programs…

- That’s where **programming languages** come in

Programming languages

- Human-made (and hopefully friendly) text language with a **logical grammar**
- The language can be **processed logically** in a sequence of instructions for a given ISA, then **coded in binary**
- This can be automated through another computing program…
  … so most programs are written by other programs
- **Languages abstract the primitive nature of the processor**

```
++++++++
[
    >++++
    [
        >++
        >+++
        >+++
        >+
        <<<<-
    ]
    >+
    >+
    >─
    >>+
    [<]

    ←
]

>>.
>---.
+++++++..+++.
>>.
←.
<.
+++.------.--------.
>>+.
>++.
```

“Hello World” in Brainfuck

Compilation and interpretation

- All languages are written in text form, the **source code**
- Compiled languages: the source code is transformed into a machine language binary file: the **executable**
  e.g.: Fortran, C/C++, Go, Haskell, Rust, …
- Interpreted languages: the sourced is transformed in machine code by an **interpreter** during the execution
  e.g.: Javascript, Python, MATLAB, Perl, …
- Compiled languages are typically **faster** because of code optimisation and **no interpreter overhead**
- Interpreted languages are typically **more portable**
- It is not a clear line (think about JIT compilation techniques)

Executable binary creation

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Hello world\n");

  return EXIT_SUCCESS;
}
```

C language

Compiler

Assembly (ASM):
human-readable machine code

```asm
main: # @main
  push rbp
  mov rbp, rsp
  sub rsp, 16
  mov dword ptr [rbp - 4], 0
  movabs rdi, offset .L.str
  mov al, 0
  call printf
  xor ecx, ecx
  mov dword ptr [rbp - 8], eax # 4-byte Spill
  mov eax, ecx
  add rsp, 16
  pop rbp
  ret
.L.str:
  .asciz "Hello world\n"
```

x86_64 assembly

Assembler + linker

file offset

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0002 003e 0001 0000 1040 0040 0000 0000
0000020 0040 0000 0000 0000 3938 0000 0000 0000
0000030 0000 0000 0040 0038 000b 0040 001d 001c
0000040 0006 0000 0004 0000 0040 0000 0000 0000
0000050 0040 0040 0000 0000 0040 0040 0000 0000
0000060 0268 0000 0000 0000 0268 0000 0000 0000
0000070 0008 0000 0000 0000 0003 0000 0004 0000
0000080 02a8 0000 0000 0000 02a8 0040 0000 0000
0000090 02a8 0040 0000 0000 001c 0000 0000 0000
...
```

x86_64 ELF binary

Bits & bytes

- 1 bit (b): 0 or 1

- 1 byte (B): 8 bits
  can be represented by 2 hexadecimal digits

1 byte: integer between 0x00 (0) and 0xff (255)

- word: 2B (16b)
  double word: 4B (32b)
  quad word: 8B (64b)

What now?

- We will focus on the very common x86_64 architecture

- I’ll make a comment about other architectures and GPUs at the end of the course, most concepts developed in the course apply directly

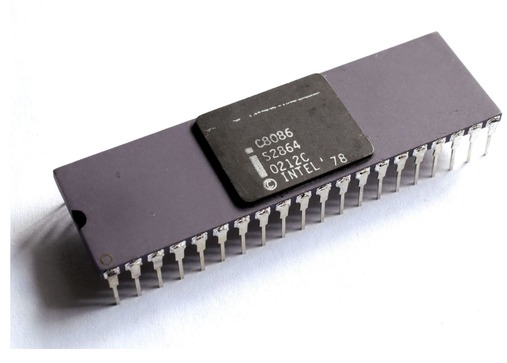
- This morning: overview of x86_64 assembly language

- This afternoon: practical view of the memory structure and the execution flow

The x86_64 architecture

A bit of history: x86

- x86 was originally a 16 bit ISA developed **in the 70s by Intel** and first released through the 8086 processor

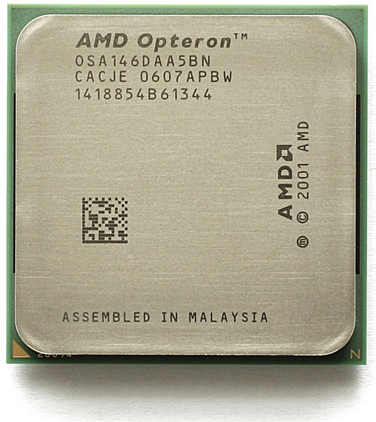


- Quickly it will become a 32 bit ISA
- Gained a huge popularity during the 90s technology boom, especially with the **Pentium processors**
- Now manufactured by other vendors than Intel (e.g. AMD)
- Omnipresent in desktops/workstations but also supercomputers. Marginal in mobile devices and embedded systems (mainly dominated by the ARM architecture)

A bit of history: x86_64

- x86_64 is a 64 bit extension of x86

- It was actually **first developed by AMD** for the opteron CPU in 1999
  That's why x86_64 is also often called amd64

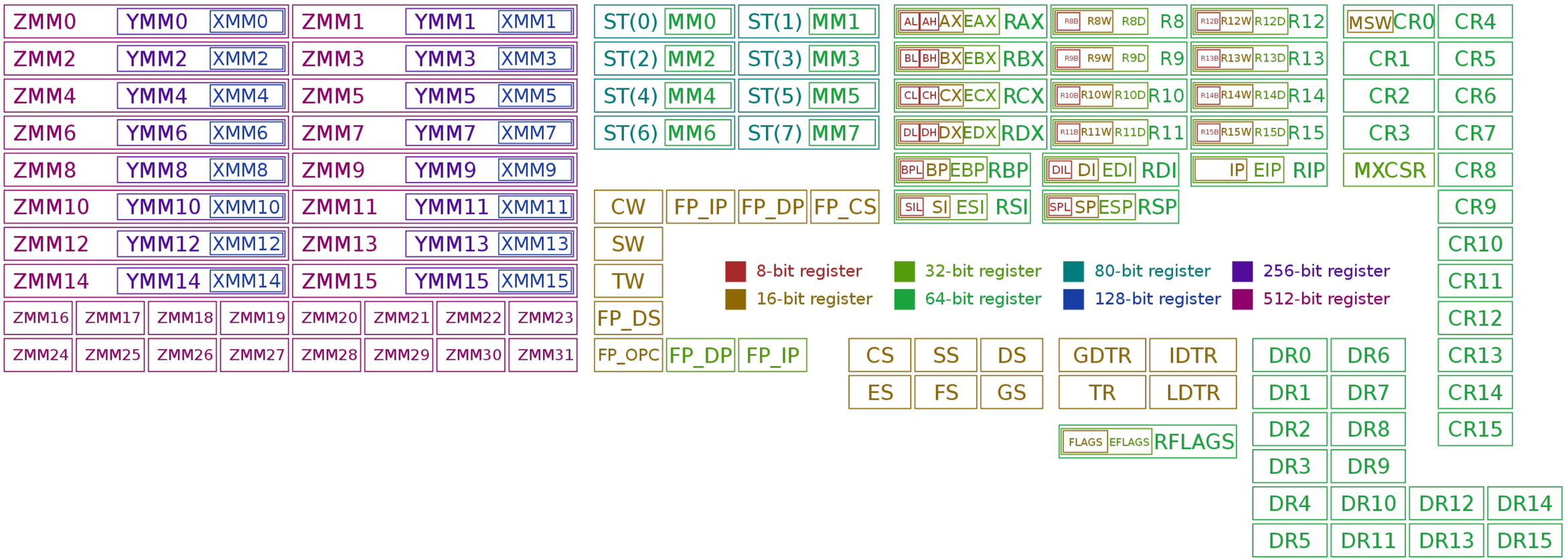


- Intel developed at the time an entirely new 64 bit architecture (IA-64), but **eventually fell back on x86_64**

- Nowadays Intel & AMD have several **cross-licensing agreements** to develop the architecture further

- It received **many extensions through the years**, especially targeted at more performant floating-point arithmetic

x86_64 registers



- On-chip memory
- Very limited, **almost instantaneous access** (1-2 cycles)
- Supported registers depend on the CPU model

Classic general purpose registers

Register	Typical purpose
ax	accumulator
cx	counter
dx	data
bx	base pointer
sp	stack pointer
bp	stack base pointer
si	source index
di	destination index

- These are all **16 bits**,
  they all exists in 32 bits (e*) and 64 bits (r*) versions

  e.g.: ax (16b), eax (32b) & rax (64b)

Floating point registers

- **xmm0-15: 128b vector registers**
  Can store 2 double prec./4 single prec. FP numbers.
  Requires SSE extension of the ISA (Pentium III 1999)

- **ymm0-15: 256b vector registers**
  Can store 4 double prec./8 single prec. FP numbers.
  Requires AVX extension of the ISA (Sandy Bridge 2011)

- **zmm0-31: 512b vector registers**
  Can store 4 double prec./8 single prec. FP numbers.
  Requires AVX512 extension of the ISA (KNL 2013)

- **All very crucial for HPC applications!**

More registers…

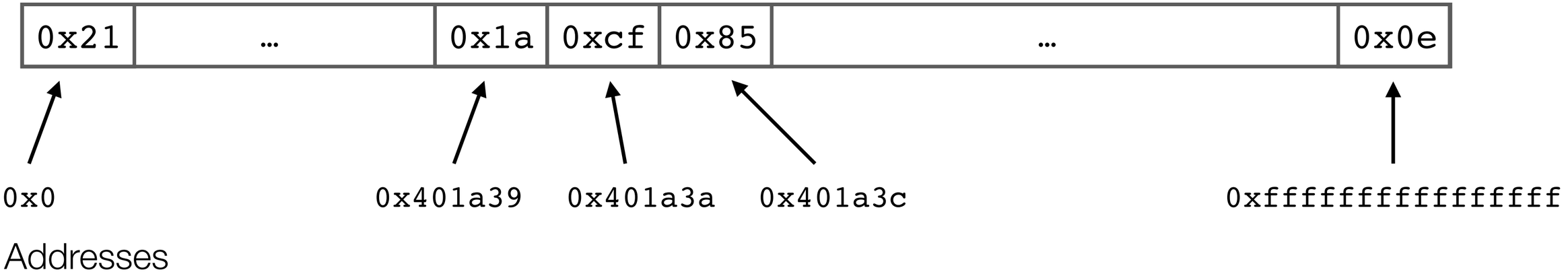
- x86_64 got more 64b general purpose registers **r8-15**.

- **rip** is very important, it is the instruction pointer register, **storing the address of the next instruction during the program execution**.

- There a more of them…
  but that’s probably enough for now.

The memory

- In x86_64 the memory space is a linear sequence of bytes addressed by 64 bit addresses



- It can take many physical forms (RAM, VRAM, disk, …)

- One is generally never looking at physical memory addresses, the memory space is abstracted through **virtual memory** (more on that later)

Assembly (ASM) generalities

- 1:1 human-readable representation of machine code
- **2 conventions** AT&T and Intel ASM, unfortunately quite different and widely used.
- I prefer Intel but I will have to use a bit of both.

```asm
main: # @main
  push rbp
  mov rbp, rsp
  sub rsp, 16
  mov dword ptr [rbp - 4], 0
  movabs rdi, offset .L.str
  mov al, 0
  call printf
  xor ecx, ecx
  mov dword ptr [rbp - 8], eax # 4-byte Spill
  mov eax, ecx
  add rsp, 16
  pop rbp
  ret
.L.str:
  .asciz "Hello world\n"
```

Intel ASM

```asm
main: # @main
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  movl $0, -4(%rbp)
  movabsq $.L.str, %rdi
  movb $0, %al
  callq printf
  xorl %ecx, %ecx
  movl %eax, -8(%rbp) # 4-byte Spill
  movl %ecx, %eax
  addq $16, %rsp
  popq %rbp
  retq
.L.str:
  .asciz "Hello world\n"
```

AT&T ASM

x86_64 ASM generalities

- **More than 1500 instructions**,
  not something you learn by heart!

- Many of them exotic or a variant on simple operations
  (copying memory, arithmetic, jumps)

- **References**:
  Intel official: https://intel.ly/3pYRdOQ (Nov 2020 version)
  Unofficial Felix Cloutier reference: https://bit.ly/3b64K33

ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.

Move instructions

- Copy data between registers or memory and a register (not memory to memory!)

```
mov rbx, rax   ; copy rax into rbx
mov rax, [rsp] ; copy 8B at the address in rsp to rax
```

- Variants for other types

```
movsxd  rbx, eax   ; copy a signed 32 bits integer
movsd   xmm0, xmm1 ; copy a double-precision FP number
vmovupd ymm0, ymm1 ; copy a double-precision FP vector of 4 numbers
```

- There are many others!
(precision change, zero extended, …)

Intel ASM operands syntax rules

- Registers: by name, order is **destination-origin**
- **Indirection** (accessing pointed value) using

```
[rsp]        ; memory at address contained in rsp
[rbp - 0×8] ; memory 8B before address in rbp
```

- Indirection can have a **size** (think pointer type in C)

```
qword ptr [rbp - 0×8] ; 8B at address rbp - 8
dword ptr [esp]       ; 4B at address esp
```

- Sizes: `byte` (1B), `word` (2B), `dword` (4B), `qword` (8B)

AT&T vs. Intel syntax

- Source & destination swapped
- Register names prefixed by `%`
- Indirection with `( )` instead of `[ ]`
- Size goes in the instruction suffix
- Shift indicated as prefix to `( )`
- Example:

```
movq %rax, %rbx         ; Intel: mov rax, rbx
movq (%rsp), %rax       ; Intel: mov rax, [rsp]
movl -0×8(%esp), %eax ; Intel: mov eax, dword ptr [esp - 0×8]
```

Integer arithmetic instructions

- 2 operands form, does not work memory-to-memory.
- Otherwise fairly straightforward

```
add  rax, rbx ; add rbx into rax (think rax += rbx)
sub  rax, rbx ; subtract
mul  rax, rbx ; multiply (unsigned)
imul rax, rbx ; multiply (signed)
div  rax, rbx ; divide (unsigned)
idiv rax, rbx ; divide (signed)

inc rax ; increment the content of rax (think rax++)
dec rax ; decrement
```

First example

- Compiler Explorer is fantastic to play with compiled code
- **Updates ASM in real-time** as you type C/C++ code
- Extremely **wide range of compilers**

```c
int func(int a, int b)
{
    return a + b;
}
```

```asm
func:
  ; we'll deal with that part later...
  push rbp
  mov rbp, rsp
  mov dword ptr [rbp - 4], edi
  mov dword ptr [rbp - 8], esi

  ; return a+b;
  mov eax, dword ptr [rbp - 4] ; copy a to eax
  add eax, dword ptr [rbp - 8] ; add b to eax
  pop rbp                      ; we'll see that later...
  ret                          ; return
```

https://godbolt.org/z/95MK3M

Tests and jumps

- All types of loops are just “ifs” and “gotos”!
- The **`cmp`** instructions compare 2 operands, the result is stored in a special register called EFLAGS
- The **`jmp`** instruction **makes the program jump** to a given address in the code (like a good old “goto”)
- **`jmp`** exists in **many conditional versions**, using EFLAGS to get the result of a previous **`cmp`** instruction

```
1 cmp rbx, rcx ; compare rbx & rcx
2 jne r8       ; jump to r8 if rbx ≠ rcx
3 jle r9       ; jump to r9 if rbx ⩽ rcx
4              ; etc... there are a lot of them
```

Loop example

- For loop in ASM

```asm
; ASM loop, C equivalent for (int i = 0; i < 5; ++i)
  mov ecx, 0      ; ecx = 0
.loop:
  cmp ecx, 5      ; compare ecx & 5
  jge .end        ; jump to .end if ecx ⩾ 5
  ; loop body
  inc ecx         ; ecx++
  jmp .loop       ; next iteration
.end:
  ; rest of the program
```

- Local labels (written as `.label:`) can be used to mark fixed positions in the code

Floating point arithmetics

- With AVX instructions operations on vectors of FP numbers can be done in one execution.
  Single Instruction, Multiple Data (SIMD)

- **Performance-critical** in numerical software

- Implemented through vector `v*` versions of FP instructions, and vector registers `ymm` and `zmm`

- **Fused** multiplied & add possible

```
1 vmovupd      ymm1, ymmword ptr [rsi] ; move 32B at rsi to ymm0
2 vaddpd       ymm0, ymm1, ymm2        ; ymm0 = ymm1 + ymm2 element-wise
3 vfmadd132pd ymm0, ymm1, ymm2         ; ymm0 = ymm0*ymm2 + ymm1 element-wise
```

More examples, compiler optimisations

- Loop optimisation https://godbolt.org/z/f6K1Gr
- Loop vectorisation https://godbolt.org/z/8Y8ffP

Hello, World

Linux system calls

- Call directly procedures from the kernel. **OS-dependent**
- Available in x86_64 through the `syscall` instruction.
- For Linux defined in https://bit.ly/3bSE6cS
  Friendlier reference https://bit.ly/3bQrSSg
- Call number must be set in `rax`, arguments in other registers depending on the call.

1	write	sys_write	fs/read_write.c
%rdi **unsigned int** fd		%rsi **const char __user *** buf	%rdx **size_t** count

- Rarely needed unless you code directly in ASM, generally wrapped-up in C functions.

Mandatory ASM “Hello world”

```asm
section .data
  msg db "Hello world",0xa ; 0xa is '\n'

section .text
  global _start

  _start:
    ; syscall 1 (write)
    ; rdi: unsigned int fd
    ; rsi: const char *msg
    ; rdx: size_t count
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, 13
    syscall
    ; syscall 60 (exit)
    ; rdi: int error_code
    mov rax, 60
    xor edi, edi
    syscall
```

“Hello world” in C

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Hello world\n");

  return EXIT_SUCCESS;
}
```

You can write ASM directly in C

```c
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    const char   msg[] = "Hello world\n";
    const size_t size  = sizeof(msg);
    ssize_t      ret;

    __asm__ volatile
    (
        "movq $1, %%rax;"
        "movq $1, %%rdi;"
        "movq %1, %%rsi;"
        "movq %2, %%rdx;"
        "syscall"
        : "=a"(ret)
        : "g"(msg), "g"(size)
        : "%rdi", "%rsi", "%rdx", "%rcx", "%r11"
    );

    return EXIT_SUCCESS;
}
```

GNU inline ASM syntax https://bit.ly/3bHeI9S

AT&T syntax, Intel syntax support really bad (especially clang)

You can write machine code directly in C!

“shellcode”

```c
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>

const unsigned char code [] =
  "\xeb\x1b\xb8\x01\x00\x00\x00\xbf\x01\x00\x00\x00\x5e\xba\x0c"
  "\x00\x00\x00\x0f\x05\xb8\x3c\x00\x00\x00\x31\xff\x0f\x05\xe8"
  "\xe0\xff\xff\xff\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";

int main(int argc, char **argv)
{
  // It used to be easier to break computers...
  // the commented code will likely end up in SIGSEGV on modern OSs
  // for security data is tagged non-executable
  ////////////////////////////////////////////////////////////////
  // void (*hello)(void);
  //
  // hello = (void(*)(void))code;
  // hello();
  ////////////////////////////////////////////////////////////////

  char *buf;
  int  prot  = PROT_READ | PROT_WRITE | PROT_EXEC;
  int  flags = MAP_PRIVATE | MAP_ANONYMOUS;

  buf = mmap(0, sizeof(code), prot, flags, -1, 0);
  memcpy(buf, code, sizeof(code));

  ((void (*)(void))buf)();

  return EXIT_SUCCESS;
}

```

Make the “Hello world” shell code

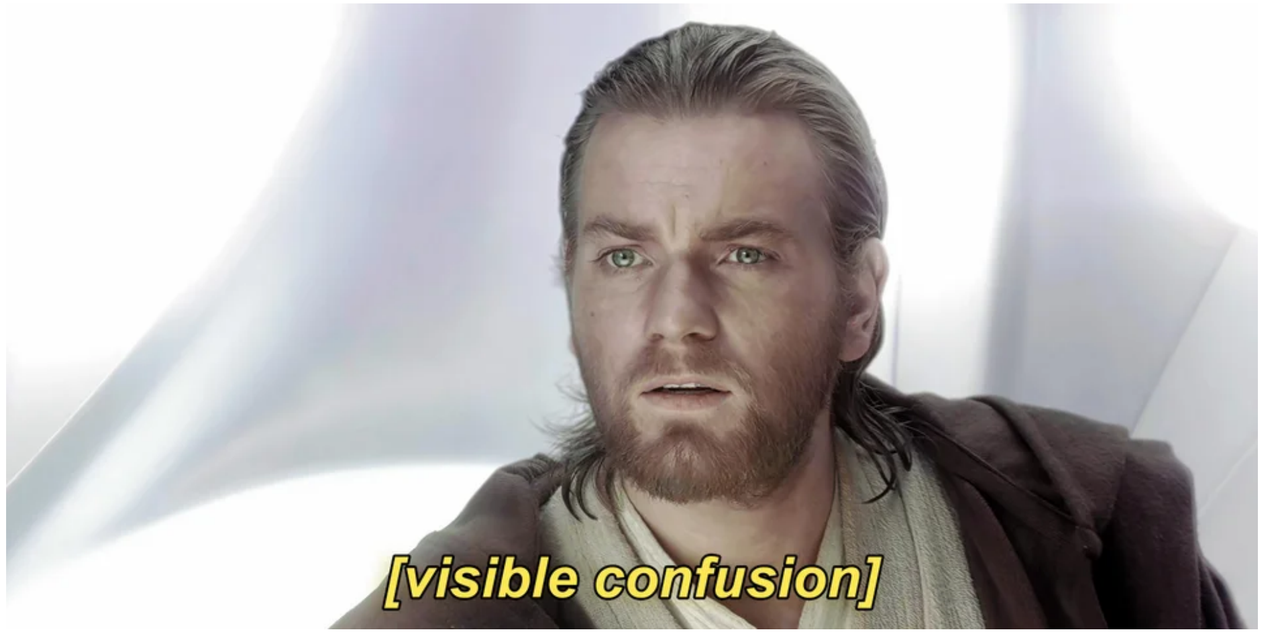
- Position-independent ASM

```
1  section .text
2    global _start
3
4    _start:
5      jmp msg ; jump & call trick
6    print:
7      mov rax, 1
8      mov rdi, 1
9      pop rsi
10     mov rdx, 12
11     syscall
12     mov rax, 60
13     xor edi, edi
14     syscall
15   msg:
16     call print
17     db "Hello world",0xa
```

Making the “Hello world” shell code

- Assemble and transform into a binary string

```
nasm -f elf64 hello-pi.asm -o hello-pi.o                # assemble
ld hello-pi.o -o hello-pi                               # link
objcopy -O binary hello-pi hello-pi.bin                 # extract binary section
hexdump -v -e '"\\""x" 1/1 "%02x" ""' hello-pi.bin # convert to C string
```



- Probably better after next lecture

Wrap up

Summary

- Whatever language you used, it gets eventually converted into a **binary machine language** defined by the processor's architecture (ISA)

- The x86_64 ISA is **one of the most common architecture**, mainly implemented through Intel and AMD CPUs

- Assembly is a **human-readable** form of machine language. Compilers translate other languages into assembly

- Assembly and machine language can also be used directly in C/C++

Next lecture

- We will look at programs execution flow
- We will discuss the memory layout of a program
- We will see the structure of Linux executable binaries
- We will do all that from a practical point of view rather than a theoretical one