Introduction aux Systèmes d’Exploitation

Unit 6: Basics of Assembly Language

François Taïani

Outline

- Assembly instructions
  - ➔ Encoding
  - ➔ Types of operations
  - ➔ Operands & Registers
- Memory
  - ➔ Little and Big Endian encoding
  - ➔ Signed integers
- Overview of the x86 ISA
- Anatomy of a nasm assembly file

Reminder

- Executable machine code = sequence of instructions
  - → (+ other very important metadata: symbols, libs, ...)
- An instruction = sequence of bits
  - → E.g. '01001000 10000011 11101100 00001000'
  - → Or '48 83 ec 08' in hexadecimal
  - → Means "subtract 8 from 64bits stack pointer register" (on x86-64 architecture)
- Hard to manipulate directly → textual representation
  - → sub rsp,0x8
  - → "assembly code"

Example: helloworld.asm

```
SECTION     .data
message:    db  "Hello, World!", 10      ; note the newline at the end
msgLen:     equ $-message

SECTION    .text
  GLOBAL    _start

_start:
  mov       rax, 1                  ; system call for write
  mov       rdi, 1                  ; file handle 1 is stdout
  mov       rsi, message            ; address of string to output
  mov       rdx, msgLen             ; number of bytes
  syscall                           ; invoke operating system to do the write
  mov       rax, 60                 ; system call for exit
  mov       rdi, 0                  ; exit code 0, equiv to xor rdi, rdi
  syscall                           ; invoke operating system to exit
```

```
$ nasm -felf64 helloworld.asm
$ ld helloworld.o
$ ./a.out
Hello, World!
$
```

Compiling with nasm

- Compiling to machine code
  - ➔ `nasm -felf64 <filename>.asm`
  - ➔ Produces a new file `<filename>.o` (object file)
  - ➔ `-felf64` needed to produce x86-64 code for Linux
- Linking (creating executable from object file)
  - ➔ `ld <filename>.o -o <executable>`
  - ➔ `-o` optional, executable called `a.out` by default
  - ➔ Complex step when shared libs involved
- Executing
  - ➔ `<executable>`

Anatomy of an assembly file

- Organized in sections
  - ➔ "zones" of executable files, and then of memory
  - ➔ Typically two: ".data" for data, and ".text" for code

Data section

```
SECTION       .data
```

- Reserves space in memory for initialized data
- And (optionally) defines labels pointing to this space
- General format

```
[label:]opt <directive> value[,value,..]
```

- Directive: typically `db` (define byte) or `dw` (define word)

```
db    0x55                  ; just the byte 0x55
db    0x55,0x56,0x57        ; three bytes in succession
db    'a',0x55              ; character constants are OK
db    'hello',13,10,'$'     ; so are string constants
dw    0x1234                ; 0x34 0x12
```

Data section (cont.)

- When a label is defined, can be used as an address
- Example : reserving & initializing 14 bytes

```
message: db  "Hello, World!", 10
```

- In the text section, using the label to refer to this zone

```
mov BYTE [message],0x20 ; overwriting 1st byte
mov rsi, message ; memory address of 1st byte to rsi
```

- Constant can also be declared with equu
  - ➔ Can use expression, but must evaluate to a constant

```
my_const:   equ 10 ; my_const replaced by 10 in code
msgLen:     equ $-message ; $ = current address
```

Text Section

```
SECTION      .text
```

- Must contain `GLOBAL _start` somewhere
  - ➔ Sets the label `_start` as external
  - ➔ Needed by the linker (ld) and Linux to launch executable
- Entry point of the program must be labelled by `_start`
  - ➔ Does not need to be at beginning of code

Assembly Instructions

- General format

  [label:]$_{opt}$ <operation> <operand1>,<operand2> [; com]$_{opt}$

- JVM executable
  - ➔ `sub` : instruction
  - ➔ `rsp` : operand1 (64 bit version of stack pointer "sp")
  - ➔ 0x8 : operand2 (number 8 in hexadecimal)

- **Intel notation** (used in this module)
  - ➔ Result stored in the **first** operand
  - ➔ (AT&T notation, used in gcc, the reverse)

Encoding Instruction

- Assembly needs to get encoded into machine code
- In x86 size of instruction code ("opcode") may vary.
- Encoding more complex than just concatenation
  - ➔ E.g. 48 in earlier example means "op on 64bit register"
  - ➔ 83 means one of ADD/OR/ADC/SBB/AND/SUB/XOR/CMP
  - ➔ Actual operation + reg selected by further bits in "ec"
- Thankfully, the assembler does this for you ☺
  - ➔ Plus other things (memory layout, constants, labels, ...)

Types of Operations

- 1st Group : **moving/setting data** (e.g. `mov rax,8`)
  - ➔ Setting register to constant values
  - ➔ Moving data in and out into the processor
  - ➔ Mainly in/out of memory, but also HW devices
- 2nd Group : **integer arithmetic** (e.g. `add rax,8`)
  - ➔ Addition, subtraction, multiplication, division
  - ➔ Boolean operations (bit-wise): and, or, not, ...
- 3rd Group **: control flow** (e.g. `jmp` 8, `syscall`)
  - ➔ Jumps (possibly conditional), calls, return, interrupts
- 4th Group **: complex operations (CISC proc)**
  - ➔ Everything else (in particular floating point)

Operands : Registers

- Nb of Registers / role vary between processors

- Main categories
  - → **Data** registers (x86-64 : `rax`, `rbx`, `rcx`, `rdx`, `...`)
  - → **Address** registers (x86-64 : `rsp`, `rbp`, `rsi`, `rdi`)
  - → **Status** register (used for carry, comparison, errors)
  - → (+ Instruction Pointer(*) & current instruction reg)

(*) Also called "Program Counter" (PC)

Operands : Registers (cont.)

- **64 bits** registers denoted by '**R**'
  - → `rax, rbx, ..., rsp`
- **32bit** registers denoted by '**E**'
  - → `eax, ebx, .., esp`
- **16 bit** registers have **no prefix** and contain an '**X**'
  - → `ax, bx, cx, dx`
- Each 16-bit register contains 2 8bit registers
  - → e.g. for `ax` : '**`al`**' (low) + '**`ah`**' (high) (same for `bx`, `cx`, `dx`)
- **Not** independent
  - → A bit like matryoshka dolls
  - → `al` part of `ax`, which is part of `eax`, which is part of `rax`

Operands: Memory

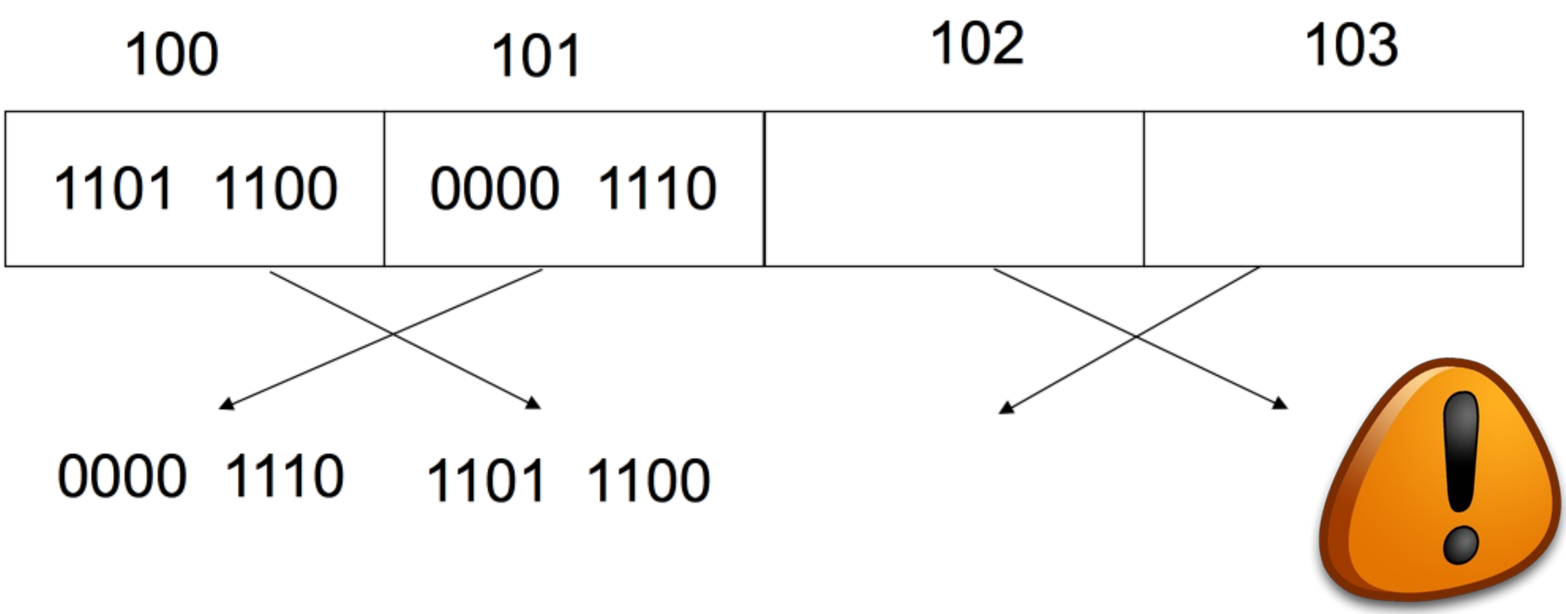
- At machine code level : memory = 1D array of bytes
  - ➔ address = position of data in this array
  - ➔ first position = zero
  - ➔ In Intel notation: "[200]" content of address 200

- Before virtual memory (8086, etc.)
  - ➔ Direct access to physical memory

- Recent processors: virtualisation (more in OS course)
  - ➔ Each executing program = illusion to be alone
  - ➔ Transparent translation (in HW+OS) between virtual/phys

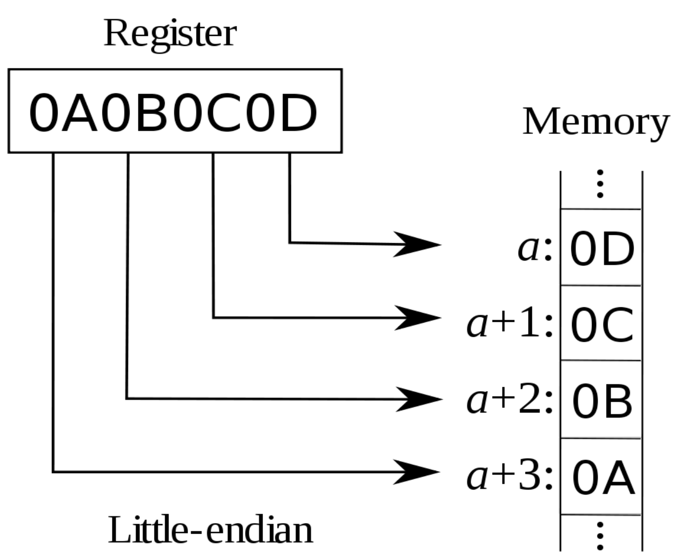
Operands: Memory (cont)

- Most instructions manipulate several bytes
  - ➔ e.g. `mov [100],ax` modifies [100] **and** [101]
- Beware : **Intel processor use "little endian"** convention
  - ➔ Least significant byte (**`al`**, "little end") starts encoding
  - ➔ Least significant byte at smallest address



Endianness

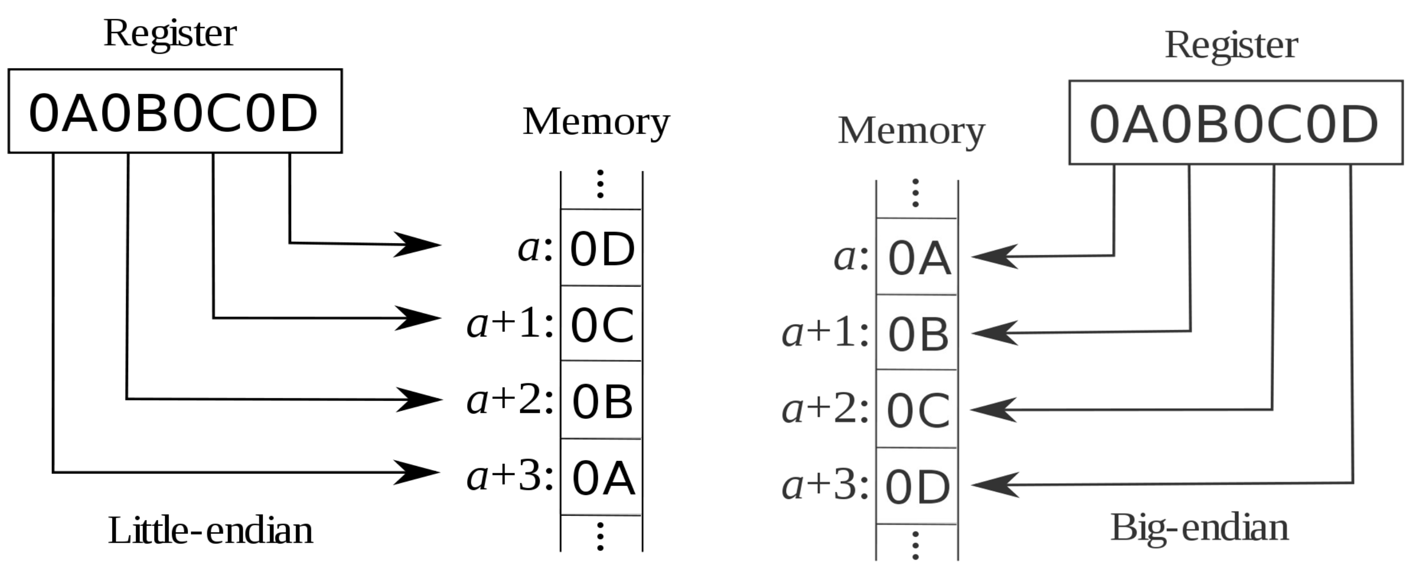
- Exercise: consider x86 program
  - ➔ `mov eax, 0x0a0b0c0d`
    `mov [a], eax ; a is some constant in assembly program`
- Content of memory ?



Source : http://en.wikipedia.org/wiki/Endianness

Endianness

- Exercise: consider x86 program
  - ➔ `mov eax, 0x0a0b0c0d`
    `mov [a], eax ; a is some constant in assembly program`
- Content of memory ?
  - ➔ What if proc were using Big-endian convention ?



Source : http://en.wikipedia.org/wiki/Endianness

Signed Numbers

- Sometimes necessary to work with numbers < 0
- Solution on n bits:
  - ➔ use raw values $\geq 2^{n-1}$ for negative numbers
  - ➔ $[-2^{n-1}, 2^{n-1}-1]$
  - ➔ A form of encoding!
- Benefit
  - ➔ `add` & `sub` operations work the same
- Danger
  - ➔ Multiplication & division are ≠: `mul` & `imul`, `div` & `idiv`
  - ➔ Misinterpreting raw values $\geq 2^{n-1}-1$
  - ➔ Important to remember which type has been used where

Signed Numbers (cont)

- Example: on 8 bits
  - ➔ Encoding of -2 ?
  - ➔ Encoding of 2 ?
  - ➔ What happens if added as unsigned integers ?

Limitation on Operands

- **Fundamental limitation** of processor instructions
  - ➔ Impossible to work on 2 memory locations in same op
  - ➔ `mov [200],[100]` is **impossible**
  - ➔ On some proc (RISC), even ADD AX, [100] impossible
  - ➔ Registers (almost) always involved

- Using **registers smartly** is essential for performance
  - ➔ Very fast, but small number
  - ➔ Work of compilers (and gurus for HPC libraries)
  - ➔ Increasingly complex heuristics (-O flag in gcc)

x86-64 Instruction Overview

- Syntax used
  - R: register (`rax`, `rbx`, etc.)
  - M: memory (typically denoted by a label: `message`)
  - V: constant (often computed by nasm: `msgLen`)
  - R/M: register or memory
- Note
  - nasm usually able to guess the size of operands

    ```
    mov [message],al ; AL=8bits=1Byte
    ```
  - but not always possible, in particular with constant

    ```
    mov  BYTE [message],0x20
    mov  [message],BYTE 0x20 ; equivalent
    ```

Transferring Data

- Instructions de transfert de données :
  - ➔ `mov` R, R/M
  - ➔ `mov` R/M, R
  - ➔ `mov` R/M, V
- Remarque :
  - ➔ `mov` ~~M, M~~ → impossible
- Example :
  - ➔ `mov ax,[100]`
  - ➔ `mov [message],ax`

Arithmetic

- Sign change
  - ➔ `neg` R/M
- Adding
  - ➔ `add` R, R/M
  - ➔ `add` R/M, R/V
- Subtracting
  - ➔ `sub` R, R/M
  - ➔ `sub` R/M, R/V
- Increment, decrement
  - ➔ `inc` R
  - ➔ `dec` R

Logic

- Beware: bitwise operations !
- 1-complement
  - `not` R/M16
- Bit-wise AND
  - `and` R16,R/M16
  - `and` R/M16,R16
- And similarly: bitwise `or`, `xor`
- Note 
  - `xor` used by compilers to set to 0 as faster than `mov`
  - E.g. `xor rdi,rdi` faster than `mov rdi,0`

Jumps

- Two types of jumps
  - ➔ Unconditional jumps (always jumps)
  - ➔ Conditional jumps (jumps based on status register)

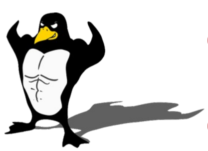
Unconditional jump

- `jmp` <address>
- Notes
  - ➔ Address typically encoded as a label

    ```
    jmp loop ; loop defined somewhere else
    ```
  - ➔ Actual machine code (opcode) uses an offset

Conditional Jumps

- Based on **Status Register**
  - ➔ Never manipulated directly
  - ➔ Contains “flags” (status bit)
  - ➔ Flags set by last “computed” result
    - S : 1 if negative
    - Z : 1 if zero
    - C : 1 if carry occurend (relevant for unsigned arithmetic)
    - O : 1 if overflow occured (relevant for signed arithmetic)



- Special instruction to perform “virtual” substraction
  - ➔ `cmp` R,R/M
  - ➔ `cmp` R/M, R
  - ➔ `cmp` R/M, V

Result lost,
but sets the status register

Conditional Jumps

- `je` : jumps if equal
  - ➔ Synonym of `jz` : jumps if zero
- `jne` : jump if ≠
  - ➔ Synonym of `jnz` : jumps if non-zero
- `jg` : jump if >
- `jge` : jump if ≥
- `jl` : jump if <
- `jle` : jump if ≤

Summary

At the end of this session you should be able to

- manipulate the most common x86 assembly instructions;
- do simple signed integer arithmetic;
- be able to manipulate numbers encoded in big and little endian;
- describe and explain the structure of a simple nasm assembly file.