Sécurité logicielle

Assembly language, part 1

Samuel Thibault <samuel.thibault@u-bordeaux.fr>
Pieces from Emmanuel Fleury <emmanuel.fleury@u-bordeaux.fr>

Different programming levels

- High-level languages (No Code, Python, Java, C++, Go, rust, …)
  - Abstraction (lambda functions, objects, lists, …)
- Low-level language (C)
  - Pointers!
- Assembly
  - Close to the processor
  - Still readable by a human
- Machine language
  - The bytes as read by the processor

Compiling C

```
gcc test.c -o test
```

Actually 4 steps

- Preprocessing : `cpp test.c -o test2.c`
  - Still C language, but includes, macros etc. expanded
- C compilation : `cc1 test2.c -o test.s`
  - Now assembly langage
- Assembly compilation : `as test.s -o test.o -c`
  - Now machine langage, that the processor would be able to run
- Link : `ld […] Scrt1.o crti.o crtbeginS.o test.o -lc crtendS.o crtn.o -o test`
  - Now a real program that one can run from the shell

And then actual execution

- Dynamic link : `/lib64/ld-linux-x86-64.so.2 ./test`

The gory details

- `cpp test.c -o test2.c`
- `/usr/lib/gcc/x86_64-linux-gnu/10/cc1 test2.c -o test.s`
- `as test.s -o test.o`
- `ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/Scrt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/10/crtbeginS.o test.o -lc /usr/lib/gcc/x86_64-linux-gnu/10/crtendS.o /usr/lib/x86_64-linux-gnu/crtn.o -o test`

Why we care about assembly

5 steps for bugs

- Preprocessing
  - Usually fine
  - Though there are sometimes surprises with macro parameter names
- C Compilation
  - **That's where all the semantic translation lies**
  - If you miswrote your program, that's when the compiler will misinterpret everything
- Assembly compilation
  - Essentially litteral translation
- Link
  - Some horrible things can hide here, e.g. symbol conflict
- Dynamic link
  - We have seen LD_LIBRARY_PATH, LD_PRELOAD, symbol interposition

Why we care about assembly

Assembly is what the processor will understand in the end

And it is still (mostly) human-readable, even if

- Elementary
- Very verbose
- Tedious

Not that seldomly the only way to understand what the hell is happening

Why we care about assembly

What does this do?

```
unsigned x = 0;
while (x < -1) {
    printf("ok\n");
}
```

An infinite loop… Yes!
Because -1 implicitly casted to unsigned, thus 4294967295
Always use **`-Wall -Wextra`**

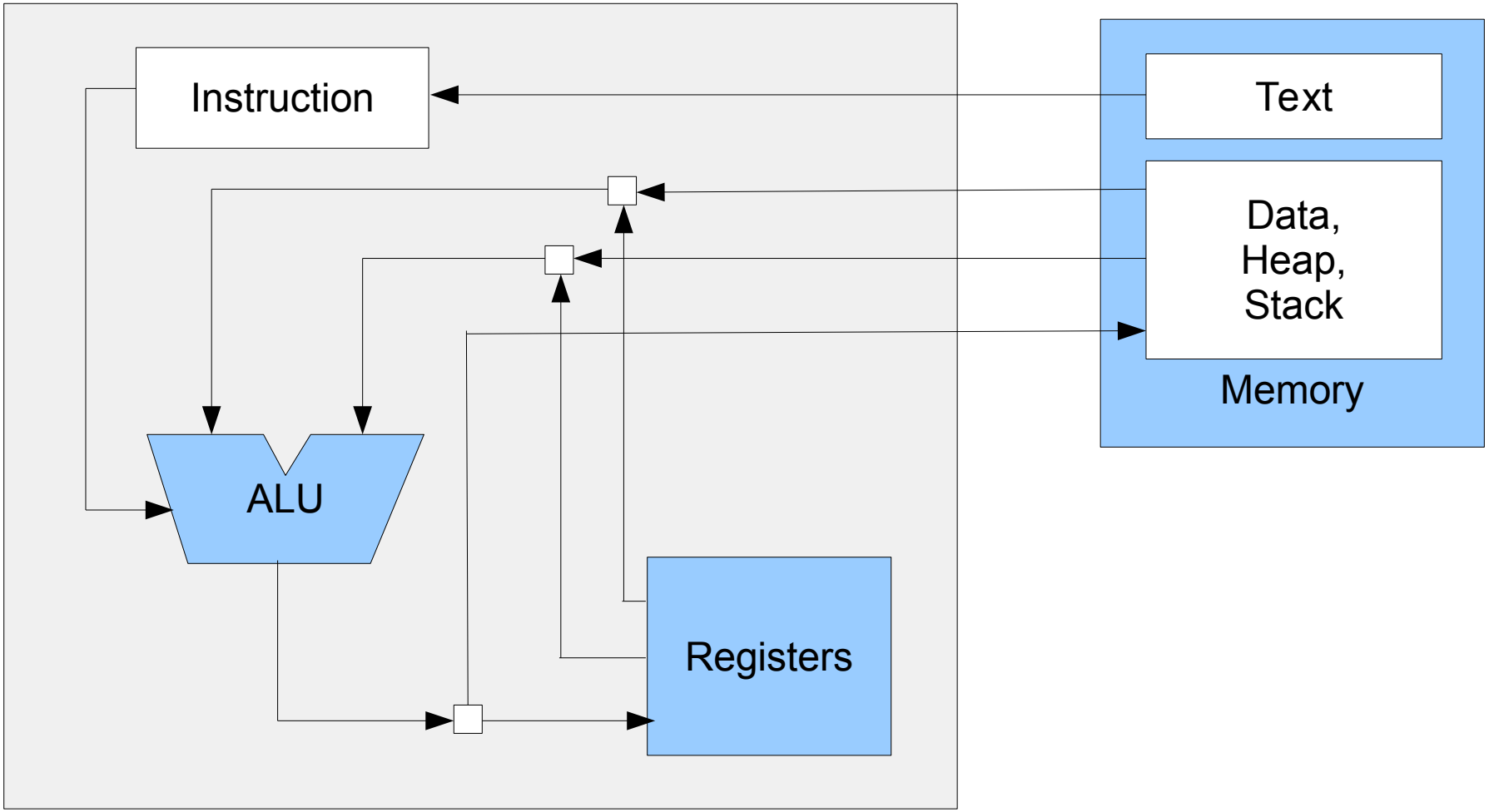
https://www.destroyallsoftware.com/talks/wat

What is assembly?

The core of computation

- Not as hardcore as a Turing machine
  but close (RAM machine from Cook & Reckhow)
- Von Neumann architecture: processor keeps repeating this:
  - Read instruction from memory at address given by Program Counter
  - Read operands (from memory or register)
  - Compute
  - Write result (to memory or register)
  - Update Program Counter

What is assembly?



What is assembly?

Unstructured programming language:

- Trivial expressions (arithmetic, bitwise, logic)
- Read/write over memory
- Jump operators
- Tests

No

- Function calls (argument passing is done manually)
- Structured loop (do this manually)
- Structured scope (all variables and functions are global)

What is assembly?

Put another way, can be thought as C with

- Only global functions and variables
- Only `if (x > 0) goto foo;` (or `>=, <, <=, ==, !=`)
- Assignments with only one operation

```
int abs(int j) {
    if (j >= 0) goto end;
    j = -j;
end:
    return j;
}
```

What is assembly?

Put another way, can be thought as C with

- Only global functions and variables
- Only `if (x > 0) goto foo;` (or `>=, <, <=, ==, !=`)
- Assignments with only one operation

```
int r, i, t;
int factorial(int j) {
    r = 1;                  // result
    i = 2;                  // counter
loop:
    t = i - j;
    if (t > 0) goto end;    // if i > j
    r = r * i;
    i += 1;
    goto loop
end:
    return r;
}
```

Assembly languages

Basically as many as processors, most common:

- Intel IA-32 (aka x86, aka i386, aka x32 (but not only :/))
- AMD/Intel x86-64 (aka amd64, aka IA-32e, aka x64)
- Accorn ARM
- RISC-V
- Motorola PowerPC

Intel processors

History... Yes, it's important for understanding the mess of x86 assembly language

- Intel 4004 (1971): First commercial complete microprocessor!
  - 4bit registers, 640B addressable memory, 740kHz
- Intel 8008 (1972):
  - 8bit registers, 16kB addressable memory, 800kHz
- Intel 8086 (1978)
  - 16bit registers, 1MB addressable memory, 10MHz
- Intel 80386 (1985): Virtual memory, floating point
  - 32bit registers, 4GB addressable memory, 16MHz
- Intel Pentium MMX (1997): parallel floating point
  - 32bit registers, 4GB addressable memory, 166MHz
- AMD Opteron (2003): 64bit
  - 64bit registers, 1TB addressable memory, 1.4GHz

Vocabulary

C: `t[i]++;`

Assembly: `addl $1, t(%eax,4)`

`addl` is the mnemonic, here "add long"
`$1` and `t(%eax,4)` are the operands
`$1` is an immediate
`%eax` is a register
`t` is a reference
`4` is a multiplicator
`t(%eax,4)` is a memory location, at address t + %eax * 4

Operand order: `source, destination` (AT&T syntax)
(Intel uses the converse :/)

Vocabulary

- Mnemonic
  - Instruction name
- Operand
  - Argument of the instruction
- Immediate:
  - Constant value encoded along the instruction
- Register:
  - Storage inside the processor
- Memory location:
  - Storage outside the processor (RAM)
- Reference:
  - Refers to a **label/symbol** that designates an address in memory
- Instruction set:
  - Set of the instructions that the processor understands
- Opcode (Operation Code):
  - Instruction as encoded in binary in machine language