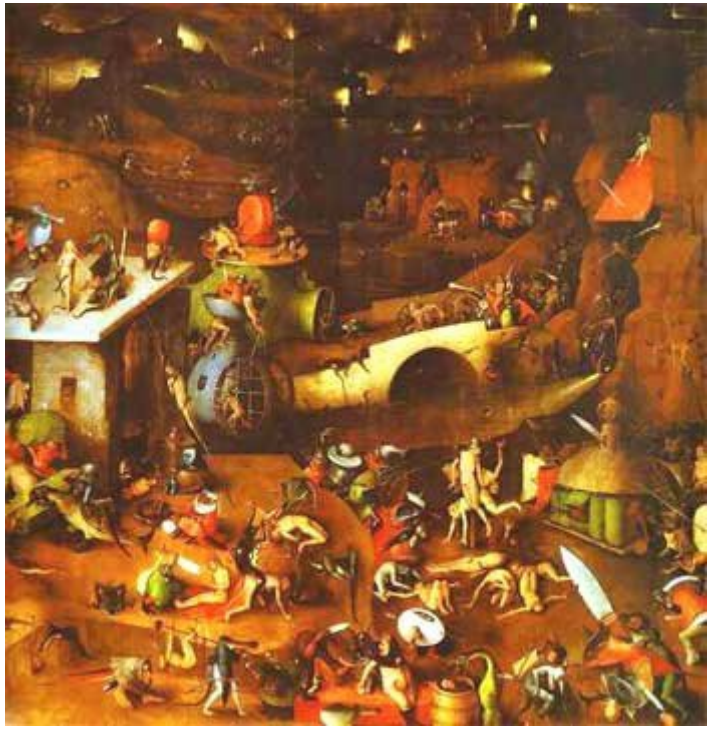
Architektura komputerów

Wykład 12

Jan Kazimirski

Assembler a języki wysokiego poziomu

Linux asm i x86-64
Lasciate ogni speranza, voi ch'intrate



Architektura x86-64 wprowadziła daleko idące modyfikacje w konwencje obsługi wywołań funkcji (przekazywanie parametrów, wartości zwracanej), oraz sposobu wywołań funkcji systemowych (syscalls).
Zmieniła się też numeracja funkcji systemowych.

Assembler – składnia Intel vs. AT&T

- Nazwy rejestrów
  - Intel: eax
  - AT&T: **%**eax
- Kolejność operandów
  - Intel: mov **dest, source**
  - AT&T: movl **source,dest**

Assembler – składnia Intel vs. AT&T c.d.

- Dane natychmiastowe
  - Intel: mov ebx, d00dh
  - AT&T: movl **$**0xd00d, %ebx
- Określenie rozmiaru operandu
  - Intel: mov bx, ax
  - AT&T: mov**w** %ax, %bx

Assembler – składnia Intel vs. AT&T c.d.

- Dostęp do zmiennej globalnej
  - Intel: [_foo]
  - AT&T: _foo
- Adresowanie poprzez rejestr
  - Intel: [eax]
  - AT&T: (%eax)

gcc i assembler – metody „rozpracowywania”

1. Deasemblacja

   objdump -d

2. Kompilacja z zatrzymaniem na etapie assemblera:

   gcc -S

[P1] Program „Hello”

```
#include <stdio.h>

int main() {

  printf("HELLO\n");

  return 0;
};
```

Kompilacja pod kątem deasemblacji

- Pełna kompilacja (z konsolidacją) powoduje że plik wynikowy (a.out) zawiera również kod bibliotek wykorzystywanych w programie, deasemblacja jest więc trudniej czytelna
- Lepszym rozwiązaniem jest kompilacja do pliku obj (bez konsolidacji). Plik wynikowy zawiera tylko kod interesującej funkcji.

[P1] objdump -d (część „main” pliku a.out)

```
0000000000400534 <main>:
  400534: 55                    push   %rbp
  400535: 48 89 e5              mov    %rsp,%rbp
  400538: bf 3c 06 40 00        mov    $0x40063c,%edi
  40053d: e8 ee fe ff ff        callq  400430 <puts@plt>
  400542: b8 00 00 00 00        mov    $0x0,%eax
  400547: c9                    leaveq
  400548: c3                    retq
  400549: 90                    nop
  40054a: 90                    nop
  40054b: 90                    nop
  40054c: 90                    nop
  40054d: 90                    nop
  40054e: 90                    nop
  40054f: 90                    nop
```

[P1] objdump -d (pliku obj)

```
p1.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:  55                          push   %rbp
   1:  48 89 e5                    mov    %rsp,%rbp
   4:  bf 00 00 00 00              mov    $0x0,%edi
   9:  e8 00 00 00 00              callq  e <main+0xe>
   e:  b8 00 00 00 00              mov    $0x0,%eax
  13:  c9                          leaveq
  14:  c3                          retq
```

[P1] gcc -S

```
    .file   "p1.c"
    .section .rodata
.LC0:
    .string "HELLO"
    .text
.globl main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    movq%rsp, %rbp
    .cfi_offset 6, -16
    .cfi_def_cfa_register 6
    movl$.LC0, %edi
    callputs
    movl$0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

[P2] Prosta funkcja numeryczna

```
int f() {

  int a = 5;
  int b = 4;

  int c = a*b;

  return c;
};
```

```
0000000000000000 <f>:
   0:   55              push   %rbp
   1:   48 89 e5        mov    %rsp,%rbp
   4:   c7 45 fc 05 00 00 00    movl   $0x5,-0x4(%rbp)
   b:   c7 45 f8 04 00 00 00    movl   $0x4,-0x8(%rbp)
  12:   8b 45 fc        mov    -0x4(%rbp),%eax
  15:   0f af 45 f8     imul   -0x8(%rbp),%eax
  19:   89 45 f4        mov    %eax,-0xc(%rbp)
  1c:   8b 45 f4        mov    -0xc(%rbp),%eax
  1f:   5d              leaveq
  20:   c3              retq
```

[P2] UWAGA! PUŁAPKA
Funkcja zoptymalizowana (-O3)

```
int f() {

  int a = 5;
  int b = 4;

  int c = a*b;

  return c;
};
```

```
0000000000000000 <f>:
   0:    mov    $0x14,%eax
   5:    retq
```

Agresywna optymalizacja znacząco wpływa na generowany kod i może utrudnić analizę.

[P3] Instrukcja warunkowa

```
int f() {

  int a = 0;

  if(a) {
    return 1;
  } else {
    return 2;
  }
};
```

```
0000000000000000 <f>:
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   movl   $0x0,-0x4(%rbp)
   b:   cmpl   $0x0,-0x4(%rbp)
   f:   je     18 <f+0x18>
  11:   mov    $0x1,%eax
  16:   jmp    1d <f+0x1d>
  18:   mov    $0x2,%eax
  1d:   leaveq
  1e:   retq
```

[P4] Pętla for

```
#include<stdio.h>

int f() {

  int a=3;

  for(int i=0;i<a;i++)
   printf("\n");

  return 0;
};
```

```
0000000000000000 <f>:
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   sub    $0x10,%rsp
   8:   movl   $0x3,-0x8(%rbp)
   f:   movl   $0x0,-0x4(%rbp)
  16:   jmp    26 <f+0x26>
  18:   mov    $0xa,%edi
  1d:   callq  22 <f+0x22>
  22:   addl   $0x1,-0x4(%rbp)
  26:   mov    -0x4(%rbp),%eax
  29:   cmp    -0x8(%rbp),%eax
  2c:   jl     18 <f+0x18>
  2e:   mov    $0x0,%eax
  33:   leaveq
  34:   retq
```

[P4] Pętla for (z optymalizacją -O3)

```
#include<stdio.h>

int f() {

  int a=3;

  for(int i=0;i<a;i++)
   printf("\n");

  return 0;
};
```

```
0000000000000000 <f>:
   0:  sub    $0x8,%rsp
   4:  mov    $0xa,%edi
   9:  callq  e <f+0xe>
   e:  mov    $0xa,%edi
  13:  callq  18 <f+0x18>
  18:  mov    $0xa,%edi
  1d:  callq  22 <f+0x22>
  22:  xor    %eax,%eax
  24:  add    $0x8,%rsp
  28:  retq
```

[P5] Wywołanie funkcji

```c
#include<stdio.h>

int f() {
  int x;
  scanf("%d",&x);
  return x;
};

int main() {
  int x = f();
  return x;
};
```

```
0000000000400564 <f>:
  400564:    sub    $0x18,%rsp
  400568:    lea    0xc(%rsp),%rsi
  40056d:    mov    $0x40068c,%edi
  400572:    mov    $0x0,%eax
  400577:    callq  400468 <__isoc99_scanf@plt>
  40057c:    mov    0xc(%rsp),%eax
  400580:    add    $0x18,%rsp
  400584:    retq

0000000000400585 <main>:
  400585:    sub    $0x8,%rsp
  400589:    mov    $0x0,%eax
  40058e:    callq  400564 <f>
  400593:    add    $0x8,%rsp
  400597:    retq
```

[P5a] Funkcja typu inline (optymalizacja -O1)

```
#include<stdio.h>

inline int f() {
  int x;
  scanf("%d",&x);
  return x;
};

int main() {
  int x = f();
  return x;
};
```

```
0000000000400564 <f>:
  400564:    sub    $0x18,%rsp
  400568:    lea    0xc(%rsp),%rsi
  40056d:    mov    $0x40069c,%edi
  400572:    mov    $0x0,%eax
  400577:    callq  400468 <__isoc99_scanf@plt>
  40057c:    mov    0xc(%rsp),%eax
  400580:    add    $0x18,%rsp
  400584:    retq

0000000000400585 <main>:
  400585:    sub    $0x18,%rsp
  400589:    lea    0xc(%rsp),%rsi
  40058e:    mov    $0x40069c,%edi
  400593:    mov    $0x0,%eax
  400598:    callq  400468 <__isoc99_scanf@plt>
  40059d:    mov    0xc(%rsp),%eax
  4005a1:    add    $0x18,%rsp
  4005a5:    retq
  4005a6:    nop
```

[P6] Parametry funkcji

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

int g() {
  int x = f(1,2,3,4);
  return x;
};
```

```
0000000000000000 <f>:
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   mov    %edi,-0x4(%rbp)
   7:   mov    %esi,-0x8(%rbp)
   a:   mov    %edx,-0xc(%rbp)
   d:   mov    %ecx,-0x10(%rbp)
  10:   mov    -0x8(%rbp),%eax
  13:   mov    -0x4(%rbp),%edx
  16:   lea    (%rdx,%rax,1),%eax
  19:   add    -0xc(%rbp),%eax
  1c:   add    -0x10(%rbp),%eax
  1f:   leaveq
  20:   retq

0000000000000021 <g>:
  21:   push   %rbp
  22:   mov    %rsp,%rbp
  25:   sub    $0x10,%rsp
  29:   mov    $0x4,%ecx
  2e:   mov    $0x3,%edx
  33:   mov    $0x2,%esi
  38:   mov    $0x1,%edi
  3d:   callq  42 <g+0x21>
  42:   mov    %eax,-0x4(%rbp)
  45:   mov    -0x4(%rbp),%eax
  48:   leaveq
  49:   retq
```

[P6a] Parametry funkcji

```
#include<stdio.h>

double f(double x) {
  return x*x;
};

int g() {
  double x=2.0;
  double y=f(x);
};
```

```
0000000000000000 <f>:
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   movsd  %xmm0,-0x8(%rbp)
   9:   movsd  -0x8(%rbp),%xmm0
   e:   mulsd  -0x8(%rbp),%xmm0
  13:   leaveq
  14:   retq

0000000000000015 <g>:
  15:   push   %rbp
  16:   mov    %rsp,%rbp
  19:   sub    $0x10,%rsp
  1d:   mov    $0x4000000000000000,%rax
  27:   mov    %rax,-0x8(%rbp)
  2b:   movsd  -0x8(%rbp),%xmm0
  30:   callq  35 <g+0x20>
  35:   movsd  %xmm0,-0x10(%rbp)
  3a:   leaveq
  3b:   retq
```

Assembler w programach C/C++

- Kompilatory gcc umożliwiają wstawianie fragmentów kodu assemblera wewnątrz programu C.
- Służy do tego słowo kluczowe asm.

[P7] Assembler w programie C

```c
#include<stdio.h>

int main() {

  asm("movq $60,%rax");
  asm("syscall");

  printf("TEST\n");
  return 0;
};
```

```
0000000000000000 <main>:
   0: push   %rbp
   1: mov    %rsp,%rbp
   4: mov    $0x3c,%rax
   b: syscall
   d: mov    $0x0,%edi
  12: callq  17 <main+0x17>
  17: mov    $0x0,%eax
  1c: leaveq
  1d: retq
```

W efekcie wykonania programu NIE wyświetli się komunikat TEST. Funkcja systemowa nr 60 oznacza zakończenie programu.

DYGRESJA: syscall

- Architektura 64-bitowa zmienia sposób wywołania funkcji systemowych.
- Zamiast wywołania przerwania stosuje się specjalną instrukcję maszynową ***syscall***
- Zmianie uległy również numery funkcji systemowych – można je znaleźć w źródłach jądra linuxa (arch/x86/include/asm/unistd_64.h)

asm – pełna składnia

- Pełna składnia polecenia asm:

  asm(**kod:wyjście:wejście:rejestry**)

  - **kod** – instrukcje assemblera
  - **wyjście** – parametry wyjściowe
  - **wejście** – parametry wejściowe
  - **rejestry** – lista użytych rejestrów

asm – pełna składnia c.d.

- Parametry wejściowe/wyjściowe – składnia i ograniczenia.
- Składnia: „*ograniczenie*” (*identyfikator*)
  - identyfikator – symbol C (np, nazwa zmiennej)
  - ograniczenia – określają rodzaj i sposób użycia np. „r” - poprzez rejestr, „m” - poprzez pamięć, „g” - brak ograniczeń. Modyfikator „=” - tylko parametr wyjściowy.

[P8] Dostęp do zmiennych C

```c
#include <stdio.h>

int main() {

  int x = 5;
  int y = 4;

  int z;

  asm volatile ("movl %1,%%eax\n" \
                "addl %2,%%eax\n" \
                "movl %%eax,%0" \
                : "=r"(z) \
                : "r"(x), "r"(y) \
                : "%eax");


  printf("%d\n",z);

  return 0;
};
```

```
0000000000400534 <main>:
  400534:   push   %rbp
  400535:   mov    %rsp,%rbp
  400538:   push   %rbx
  400539:   sub    $0x18,%rsp
  40053d:   movl   $0x5,-0x14(%rbp)
  400544:   movl   $0x4,-0x18(%rbp)
  40054b:   mov    -0x14(%rbp),%edx
  40054e:   mov    -0x18(%rbp),%ecx
  400551:   mov    %edx,%eax
  400553:   add    %ecx,%eax
  400555:   mov    %eax,%ebx
  400557:   mov    %ebx,-0x1c(%rbp)
  40055a:   mov    $0x40066c,%eax
  40055f:   mov    -0x1c(%rbp),%edx
  400562:   mov    %edx,%esi
  400564:   mov    %rax,%rdi
  400567:   mov    $0x0,%eax
  40056c:   callq  400430 <printf@plt>
  400571:   mov    $0x0,%eax
  400576:   add    $0x18,%rsp
  40057a:   pop    %rbx
  40057b:   leaveq
  40057c:   retq
```

[P8] Assembler w C
Agresywna optymalizacja (-O3)

```c
#include <stdio.h>

int main() {

  int x = 5;
  int y = 4;

  int z;

  asm volatile ("movl %1,%%eax\n" \
                "addl %2,%%eax\n" \
                "movl %%eax,%0" \
                : "=r"(z) \
                : "r"(x), "r"(y) \
                : "%eax");


  printf("%d\n",z);

  return 0;
};
```

```
0000000000400540 <main>:
  400540:   sub    $0x8,%rsp
  400544:   mov    $0x4,%esi
  400549:   mov    $0x5,%edx
  40054e:   mov    %edx,%eax
  400550:   add    %esi,%eax
  400552:   mov    %eax,%esi
  400554:   mov    $0x40065c,%edi
  400559:   xor    %eax,%eax
  40055b:   callq  400430 <printf@plt>
  400560:   xor    %eax,%eax
  400562:   add    $0x8,%rsp
  400566:   retq
```

[P8a] Assembler w C
Zmienne typu register

```c
#include <stdio.h>

int main() {

  register int x = 5;
  register int y = 4;

  register int z;

  asm volatile ("movl %1,%%eax\n" \
                "addl %2,%%eax\n" \
                "movl %%eax,%0" \
                : "=g"(z) \
                : "g"(x), "g"(y) \
                : "%eax");


  printf("%d\n",z);

  return 0;
};
```

```
0000000000400534 <main>:
  400534:   push   %rbp
  400535:   mov    %rsp,%rbp
  400538:   push   %r12
  40053a:   push   %rbx
  40053b:   mov    $0x5,%ebx
  400540:   mov    $0x4,%r12d
  400546:   mov    %ebx,%eax
  400548:   add    %r12d,%eax
  40054b:   mov    %eax,%ebx
  40054d:   mov    $0x40065c,%eax
  400552:   mov    %ebx,%esi
  400554:   mov    %rax,%rdi
  400557:   mov    $0x0,%eax
  40055c:   callq  400430 <printf@plt>
  400561:   mov    $0x0,%eax
  400566:   pop    %rbx
  400567:   pop    %r12
  400569:   leaveq
  40056a:   retq
```

[P8a] Assembler w C
Zmienne typu register + optymalizacja

```c
#include <stdio.h>

int main() {

  register int x = 5;
  register int y = 4;

  register int z;

  asm volatile ("movl %1,%%eax\n" \
                "addl %2,%%eax\n" \
                "movl %%eax,%0" \
                : "=g"(z) \
                : "g"(x), "g"(y) \
                : "%eax");

  printf("%d\n",z);

  return 0;
};
```

```
0000000000400540 <main>:
  400540:   sub    $0x8,%rsp
  400544:   mov    $0x5,%eax
  400549:   add    $0x4,%eax
  40054c:   mov    %eax,%esi
  40054e:   mov    $0x40065c,%edi
  400553:   xor    %eax,%eax
  400555:   callq  400430 <printf@plt>
  40055a:   xor    %eax,%eax
  40055c:   add    $0x8,%rsp
  400560:   retq
```

CPUID

```
#include <stdio.h>

int main() {

  char cpuName[13];

  asm volatile ("movl $0,%%eax\n" \
                "cpuid\n" \
                "lea %0,%%rax \n" \
                "movl %%ebx,(%%rax)\n" \
                "movl %%edx,0x4(%%rax)\n" \
                "movl %%ecx,0x8(%%rax)\n" \
                : "=m"(cpuName) \
                : \
                : "%rax","%ebx","%ecx","%edx");

  cpuName[12]='\0';
  printf("%s\n",cpuName);

  return 0;
};
```

GenuineIntel