Metody Realizacji Języków Programowania

Bardzo krótki kurs asemblera x86

Marcin Benke

MIM UW

23 października 2013

Uwagi wstępne

Ten, z konieczności bardzo krótki kurs, nie jest w żadnym wypadku systematycznym wykładem. Wiele aspektów jest pominiętych lub bardzo uproszczonych.

Notacja

Dla asemblera x86 istnieją dwa istotnie różne standardy notacyjne: Intel oraz AT&T.
Pierwszy z nich uzywany jest w oficjalnej dokumentacji Intela oraz przez asemblery takie jak MASM i NASM.
Drugi zaś używany jest przez narzędzia GNU: as oraz gcc.

Rejestry

8 32-bitowych rejestrów:
EAX,EDX, EBX,ECX, ESI, EDI, ESP (wskaźnik stosu), EBP (wskaźnik ramki)

Flagi

Rejestr EFLAGS składa się z pól bitowych zwanych flagami, ustawianych przez niektóre instrukcje i używanych głównie przy skokach warunkowych

- ZF — zero
- SF — znak (sign)
- CF — przeniesienie (carry)
- OF — nadmiar/niedomiar (overflow)

Do flag wrócimy przy omówieniu testów i skoków warunkowych.

Architektura x86_64

16 64-bitowych rejestrow: RAX,. . . ,RBP,R8,. . . ,R15.
Nadal można używać rejestrów 32-bitowych np. EAX, R8D oznaczają połówki odpowiednio RAX, R8.

Operandy (czyli argumenty instrukcji)

Instrukcja składa się z tzw. mnemonika (kodu operacji) oraz 0–2 operandów (argumentów), którymi mogą być:

- rejestr (r32/r64)
- stała (immediate operand, i8/i16/i32/i64),
- pamięć (m8/m16/m32/m64)

Najwyżej jeden z operandów może odwoływać się do pamięci

AT&T: rejestry prefiksowane %, stałe prefiksowane znakiem $

Rozmiary operandów

x86 może operować na wartościach 8, 16, 32 lub 64-bitowych.

Przeważnie z kontekstu wynika jaki rozmiar mamy na myśli, czasem jednak trzeba to explicite wskazać.

W składni Intela wskazujemy to poprzedzając operand prefiksem `byte, word, dword` lub `qword`, np (NASM)

```
MOV [ESP], DWORD hello
```

W składni AT&T przez sufiks b (8), w (16), l (32), lub q (64) instrukcji, np.

```
movl    $hello, (%esp)
```

NB kod generowany przez gcc zawsze dodaje takie sufiksy.

Tutaj pomijamy te sufiksy tam, gdzie nie są niezbędne.

Tryby adresowania pamięci

W ogólności adres może być postaci

$$baza + mnożnik * indeks + przesunięcie$$

gdzie baza i indeks są rejestrami, na przykład
`EAX+4*EDI+7`

Dodatkowe ograniczenia:

- ESP nie może być indeksem (pozostałe 7 rejestrów może)
- dopuszczalne mnożniki: 1,2,4,8

Składnia adresów
Intel: `[baza+mnożnik*indeks+przesunięcie]`

AT&T: `przesunięcie(baza,indeks,mnożnik)`

Najczęściej używamy trybu *baza* + *przesunięcie*, np.

```
mov 8(%ebp), %eax
```

Instrukcje przesyłania

Przypisanie

Intel: `MOV dest, src`
na przykład:

```
MOV EAX, [EBP-20h]
```

AT&T: `mov src, dest`
na przykład

```
mov -0x20(%ebp), %eax
```

Instrukcja MOV nie może przesłać między dwoma komórkami pamięci.

Zamiana

`XCHG x, y` zamienia zawartość swoich argumentów

Instrukcje przesyłania nie zmieniają flag.

Operacje na stosie

`PUSH src` np.

```
PUSH [EBP+4]
PUSH DWORD 0
push %ebp
pushl 0
```

`POP dest` np.

```
pop 4(%ebp)
POP [EBP+4]
```

`PUSHA/POPA` — połóż/odtwórz wszystkie 8 rejestrów.

Uwaga:

- operacje na stosie używają i automatycznie modyfikują ESP,
- stos rośnie w dół — PUSH zmniejsza ESP,
- ESP wskazuje na ostatni zajęty element stosu.

Operacje arytmetyczne

```
ADD x, y
SUB x, y
INC x
DEC x
NEG x
```

Podobnie jak dla MOV, w składni Intela wynik w pierwszym argumencie, w AT&T — w drugim

Flagi ustawiane w zależności od wyniku. W przypadku przepełnienia ustawiana jest flaga OF

Operacje arytmetyczne - przykłady

dodaj zawartość rejestru ESI do komórki pod adresem EBP+6:

Intel: `ADD [EBP+6], ESI`

AT&T: `add %esi, 6(%ebp)`

Odejmij 7 od EAX

Intel: `SUB EAX, 7`

AT&T: `sub $7, %eax`

Mnożenie

mnożenie przez 2^n można wykonac przy pomocy przesunięcia o n bitów w lewo (instrukcja SAL), np mnożenie przez 16

Intel: `SAL EAX, 4`
AT&T: `sal $4, %eax`

mnożenie ze znakiem: IMUL;mnożna (i iloczyn) musi być w rejestrze, mnożnik w rejestrze lub pamięci

Przykład

pomnóż ECX przez zawartość komórki pod adresem ESP

Intel: `IMUL ECX, [ESP]`
AT&T: `imul (%esp), %ecx`

Specjalna forma z jednym argumentem (mnożnikiem): `IMUL r/m32` — mnożna w EAX, wynik w EDX:EAX

SAL ustawia flagi, IMUL — tylko OF, CF.

Dzielenie

dzielenie przez 2^n można wykonac przy pomocy przesunięcia o n bitów w prawo z zachowaniem znaku (instrukcja SAR), np dzielenie przez 256

Intel: `SAR EAX, 8`
AT&T: `sar $8, %eax`

`IDIV y`: dzielna w EDX:EAX, dzielnik w rejestrze lub pamięci, iloraz w EAX, reszta w EDX

NB: przy dzieleniu dwóch liczb 32-bitowych przed IDIV należy dzielną załadować do EAX, a jej znak do EDX, czyli jeśli dzielna dodatnia to EDX ma zawierać 0, jeśli ujemna to -1. Mozna ten efekt uzyskać przez przesunięcie o 31 bitów w prawo (albo używając instrukcji CDQ).

SAR ustawia flagi, IDIV — nie.

IDIV zajmuje 43 cykle procesora [ADD r32, r32 — 2 cykle; IMUL 9–38, dokładniej $max(\lceil \log m \rceil, 3) + 6$ dla mnożnika m].

Dzielenie

Przykład (AT&T):

```
mov    28(%esp), %eax
mov    %eax, %edx
sar    $31, %edx
idivl  24(%esp)
```

Przykład: (Intel)

```
MOV EAX, [ESP+28]
MOV EDX, EAX
SAR EDX, 31
IDIV DWORD [ESP+24]
```

Z użyciem CDQ:

```
movl    28(%esp), %eax
cdq
idivl   24(%esp)
```

Instrukcje porównania

`CMP x, y` — ustawia flagi w zależności od różnicy argumentów

- ZF jeśli różnica jest 0
- SF jeśli różnica jest ujemna
- OF jeśli różnica przekracza zakres
- CF jeśli odejmowanie wymagało pożyczki

Skoki

Skok bezwarunkowy: `JMP etykieta`

Skoki warunkowe w zależności od stanu flag; kody jak wynik CMP

Porównania liczb bez znaku:

Mnemoniki	CMP	skok gdy...
`JE/JZ`	$=$	ZF = 1
`JNE/JNZ`	$\neq$	ZF = 0
`JAE/JNB`	$\geq$	CF = 0
`JB/JNAE`	$<$	CF = 1
`JA/JNBE`	$>$	(CF or ZF) = 0
`JBE/JNA`	$\leq$	(CF or ZF) = 1

Porównania liczb ze znakiem:

Mnemoniki	CMP	skok gdy...
`JG/JNLE`	$>$	((SF xor OF) or ZF) = 0
`JGE/JNL`	$\geq$	(SF xor OF) = 0
`JL/JNGE`	$<$	(SF xor OF) = 1
`JLE/JNG`	$\leq$	((SF xor OF) or ZF) = 1

Porównania — przykład

```
int cmp(int a, int b) {
  if(a>b) return 7;
}
```

może zostać skompilowane do

```
cmp:
    pushl %ebp
    movl  %esp, %ebp
    movl  8(%ebp), %eax
    cmpl  12(%ebp), %eax  # cmp a, b
    jng   L4              # skok gdy warunek NIE zachodzi
    movl  $7, %eax
    movl  %eax, %edx
    movl  %edx, %eax
L4:
    popl  %ebp
    ret
```

Funkcje

`CALL adres` — kładzie na stosie adres następnej instrukcji i skacze pod `adres`
`RET` — skok pod adres na szczycie stosu (zdejmuje go)
To jest ten sam stos na którym operują PUSH/POP/ESP.

Funkcje zewnętrzne (tylko Intel) i globalne trzeba deklarować, np Intel:

```
extern puts
global main
```

AT&T:

```
.extern puts
.globl main
```

Protokół wywołania funkcji x86

Istnieje wiele wariantów, tu zajmiemy sie protokołem uzywanym przez GCC+libc (aka “cdecl”).

- przy wywołaniu na stosie argumenty od końca, ślad powrotu
- wołający zdejmuje argumenty
- przy powrocie wynik typu int/wskaźnik w EAX
- rejestry EBP,ESI,EDI,EBX muszą być zachowane

Standardowy prolog:

```
pushl   %ebp
movl    %esp, %ebp
subl    $x, %esp    /* zmienne lokalne */
```

Standardowy epilog:

```
movl    %ebp, %esp /* pomijane jesli nop */
popl    %ebp
ret
```

Więcej o protokołach wywołania funkcji — na kolejnych wykładach.

Przykład — stałe napisowe

```
hello:
    .string "Hello\n"
    .globl  main
main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   $hello
    call    puts
    movl    $0, %eax
#   add $4, %esp     # patrz nast. instrukcja
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Protokół wywołania x86_64

- Liczby całkowite przekazywane w EDI,ESI,EDX,ECX,R8D,R9D
- wskaźniki przekazywane w RDI,RSI,RDX,RCX,R8,R9
- jesli więcej argumentów, lub większe niż 128-bitowe, to na stosie
- przy powrocie wynik typu int w EAX; wskaźnik w RAX
- rejestry RBP, RBX i R12 do R15 muszą być zachowane
- $RSP \equiv 0 \mod 16$ (przed CALL, czyli 8 po CALL)

Standardowy prolog:

```
pushl   %rbp
movl    %rsp, %rbp
subl    $x, %rsp    /* zmienne lokalne */
```

Standardowy epilog:

```
movl    %rbp, %rsp /* pomijane jesli nop */
popl    %rbp
ret
```

Protokół wywołania funkcji x86_64 — przykład

Liczby całkowite przekazywane w EDI,ESI,EDX,ECX,R8D,R9D
wskaźniki przekazywane w RDI,RSI,RDX,RCX,R8,R9

```
hello:
    .asciz "Hello\n"
.globl main
main:
    pushq   %rbp
    mov     %rsp, %rbp
    movq    $hello, %rdi
    call    puts
    mov     $0, %eax
    popq    %rbp
    ret
```

Sztuczki

Alternatywny epilog:

```
leave
ret
```

instrukcja LEAVE przywraca ESP i EBP
(istnieje też ENTER, ale jest za wolna)

`TEST x, y` - wykonuje bitowy AND argumentów, ustawia SF i ZF zależnie od wyniku, zeruje OF i CF

Najczęstsze użycie: ustalenie czy EAX jest dodatnie/ujemne/zero

Intel: `TEST EAX,EAX`
AT&T: `test %eax, %eax`

Sztuczki

LEA — ładuje do rejestru wyliczony adres (baza+n*idx+d).
Podstawowe użycie: pobranie adresu l-wartości
`LEA EAX, [EBP+8]`
Inne użycia: operacje arytmetyczne

Przykład

EAX := EBX+2*ECX+1
Intel: `LEA EAX, [EBX+2*ECX+1]`
AT&T: `lea 1(%ebx,%ecx,2), %eax`

Skoki pośrednie

CALL r/m32 — wywołanie funkcji o adresie w rejestrze/pamięci
— może być użyte do realizacji metod wirtualnych
JMP r/m32 — skok jak wyżej, może być użyty do implementacji instrukcji `switch`.

Mac OS X

```
gcc -o hello hello64.s
clang -cc1as: fatal error: error in backend:
32-bit absolute addressing is not supported
 in 64-bit mode
```

Rozwiązanie:

```
hello:
.asciz "Hello\n"
.globl _main
_main:
  pushq %rbp
  mov   %rsp, %rbp
  lea   hello(%rip), %rdi
  call _puts
  mov $0, %rax
  leave
  ret
```

`sumto` w asemblerze 80x86 (gas)

```
        .globl sumto
sumto:
        movl    4(%esp), %ecx ; ecx = n
        xorl    %eax, %eax    ; eax = 0
        testl   %ecx, %ecx    ; ecx <=0?
        jle     .L4           ; skok do L4
        xorl    %edx, %edx    ; edx = 0
.L5:
        addl    $1,%edx       ; edx += 1
        addl    %edx, %eax    ; eax += edx
        cmpl    %edx, %ecx    ; edx != n?
        jne     .L5           ; skok do L5
.L4:
        ret                   ; powrót
```