Metody Realizacji Języków Programowania

Bardzo krótki kurs asemblera x86

Marcin Benke

MIM UW

10 stycznia 2011

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.

Operandy (czyli argumenty instrukcji)

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

- rejestr (r32)
- stała (i8/i16/i32)
- pamięć (m8/m16/m32)

Najwyżej jeden z operandów może odwoływać się do pamięci

AT&T: rejestry prefiksowane %, literały całkowite prefiksowane znakiem $

Rozmiary operandów

Ze względów (głównie historycznych), i386 może operować na wartościach 8, 16 lub 32-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 albo dword, np (NASM)

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

W składni AT&T przez sufiks b w lub l instrukcji, np.

```
movl    $C0, (%esp)
```

NB kod generowany przez gcc dodaje takie sufiksy do wszystkich instrukcji.
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: `[base+index*scale+disp]`
AT&T: `disp(base,index,scale)`

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
```

`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

Przykład

dodaj zawartość rejestru ESI do komórki pod adresem EBP+6:
Intel: `ADD [EBP+6], ESI`
AT&T: `add %esi, 6(%ebp)`

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.

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
IDIVL [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`	$>$	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      # skocz jeśli warunek NIE zachodzi
movl $7, %eax
movl %eax, %edx
movl %edx, %eax
L4:
popl %ebp
ret
```

Protokół wywołania funkcji

`CALL adres` skok, śladem powrotu na stosie
`RET` — skok pod adres na szczycie stosu (zdejmuje go)
Protokół używany przez `gcc` oraz `libc`

- przy wywołaniu na stosie argumenty od końca, ślad powrotu
- przy powrocie wynik typu int w EAX, typu double w ST

Standardowy prolog:

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

Standardowy epilog:

```
movl    %ebp, %esp
popl    %ebp
ret
```

Przykład — stałe napisowe

```
.LC0:
.string "Hello\n"
.globl main
main:
pushl %ebp
movl %esp, %ebp
pushl $.LC0
call puts
movl $0, %eax
leave
ret
```

Operacje zmiennoprzecinkowe

Dość skomplikowane, przyjmiemy bardzo uproszczony (wręcz nieprawdziwy) model:

jeden rejestr ST typu double i operacje:

FLD m64 — załaduj spod adresu do ST
FST m64 — zapisz z ST do pamięci
FSTP m64 — zapisz z ST do pamięci i zwolnij rejestr ST
FADD x — ST += x (analogicznie FSUB, FMUL, FDIV)
FSUBR x — ST = x - ST (analogicznie FDIVR)

Operacje zmiennoprzecinkowe — przykład

```
double  dsub(double a, double b) {
  return a-b;
}
```

może zostać skompilowane do

```
dsub:
        pushl   %ebp
        movl    %esp, %ebp
        fldl    16(%ebp) # b
        fsubrl  8(%ebp)  # a
        popl    %ebp
        ret
```

Stałe zmiennoprzecinkowe — przykład

```
double  dcon(double a, double b) {
  return a / 1.6180339887 ;
}
```

może zostać skompilowane do

```
dcon:
        pushl   %ebp
        movl    %esp, %ebp
        fldl    .LC0
        fdivrl  8(%ebp)
        popl    %ebp
        ret

        .align 8
.LC0:
        .float 1.6180339887
```

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 zawartość rejestru EAX jest dodatnie/ujemne/zero
Intel: `TEST EAX,EAX`
AT&T: `test %eax, %eax`

Sztuczki

LEA — ładuje do rejestru wyliczony adres (który może być postaci może być postaci baza+mnożnik*indeks+przesunięcie). Może być wykorzystane do wykonania jedną instrukcją ciekawych operacji arytmetycznych

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 zawartym w rejestrze/komórce pamięci — może być użyte do realizacji metod wirtualnych
JMP r/m32 — skok jak wyżej, moze być użyty do implementacji instrukcji `switch`.