

3.6.2.1 Integers

A binary integer is '0b' or '0B' followed by zero or more of the binary digits '01'.

An octal integer is '0' followed by zero or more of the octal digits ('01234567').

A decimal integer starts with a non-zero digit followed by zero or more digits ('0123456789').

A hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits chosen from '0123456789abcdefABCDEF'.

Integers have the usual values. To denote a negative integer, use the prefix operator '-' discussed under expressions (see [Prefix Operators](#)).

Tomczak analizuje powyższą podstronę Using As.

Całość dostępna pod tym linkiem: <https://sourceware.org/binutils/docs/as/>

Zwraca uwagę na precyzyjność z jaką musi być pisany program, żeby działał poprawnie oraz znaczenie trzymywania się specyfikacji.

Architektura i Organizacja

niedziela, 19 maja 2024 21:52

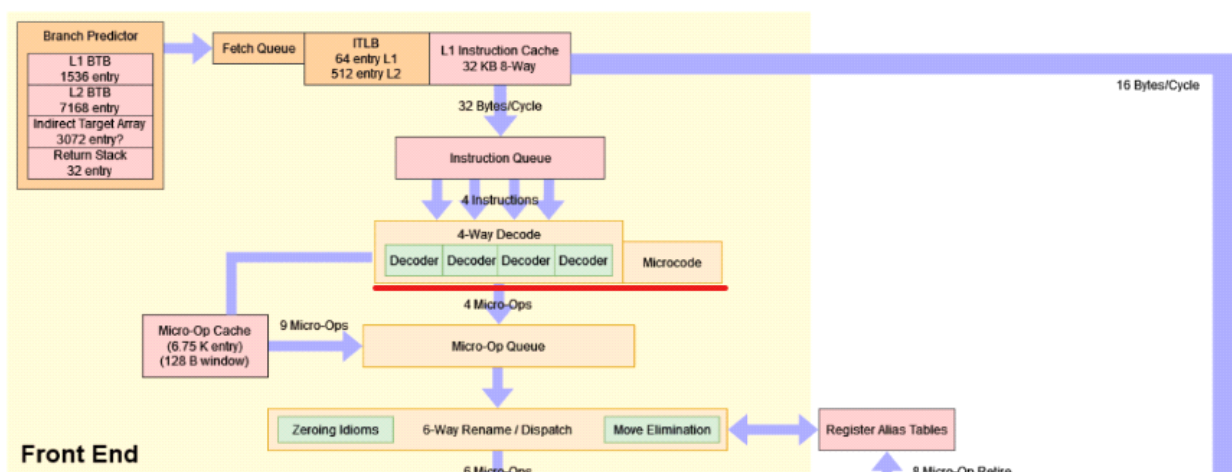
- I. Architektura komputerowa (ISA: Instruction Set Architecture) - komputer z punktu widzenia programisty:
 - lista rozkazów (instrukcje procesora, wykonywanie obliczeń, kopiowanie wartości, itd.)
 - tryby adresowania (sposoby określania operandów) -
 - Rejestry - szybka bo niewielka pamięć w procesorze
 - typy danych - niektóre przypominające to znane z c, inne nie
 - obsługa wyjątków i przerw - dzielenie przez zero, zdarzenia dziejące się "poza kontrolą programu"
 - "jak to mniej więcej działa"
- II. Organizacja komputerowa (Hardware Set Architecture) - układ elementów sprzętowych

Instrukcje procesora:

Podczas asemblacji instrukcje z kodu są rzeczywiście zapisane 1 do 1 "jakoś", ale w dalszych etapach procesor sprzętowo rozbija je sobie na mniejsze.

11:22 Tomczak wskazuje wykres, najpewniej ten ze strony 21, ale nie mam pewności, które miejsce dokładnie

Wydaje się, że chodzi o to miejsce:



W kartach graficznych asembler pisany (programowy) jest tłumaczony przez narzędzia producenta na asembler typowy dla konkretnego procesora graficznego. Żeby ukryć szczegóły implementacji wewnętrznej karty graficznej, ale zaletą jest też uniwersalność kodu pisanego.

Grupowanie instrukcji (w kategorii robiące coraz więcej rzeczy, instrukcje z dolnej części listy najczęściej muszą zrobić coś z górnej części listy):

- przesłanie danych (odczyt, zapis)
- proste operacje logiczne (pojedyncze bramki)
- zmiany formatu (rozszerzanie cyfr w prawo/lewo nie tylko w systemach U2 i NB, ale też zmiennoprzecinkowych)
- konwersje (trochę bardziej skomplikowane)
- arytmetyka (nie tylko binarnej, ale też dziesiętnej, zmiennoprzecinkowej, w zależności od producenta)
- rozgałęzienia (zmiana przepływu sterowania, przydające się do pisania kodów krótszych np. skoki)
- specjalne (NOP, LEA, CPUID...)
- instrukcje koprocessorów

- systemowe (instrukcje zapewniające wygodę i bezpieczeństwo na poziomie systemu operacyjnego)

Tryby adresowania:

Mamy do czynienia z różnymi typami pamięci, ponieważ w ten sposób jest szybciej, na rejestrach wewnętrznych można zrobić operacje 100 razy szybciej niż operując na pamięci zewnętrznej.

*Operacja na pamięci może trwać średnio kilkaset cykli procesora, w tysiącach czy nawet w dziesiątkach, setkach tysięcy (rzadko).

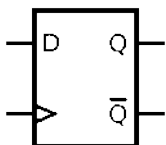
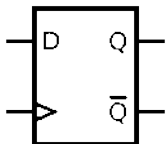
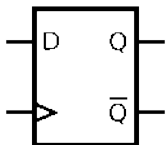
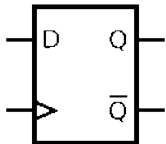
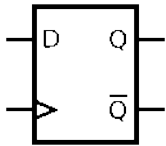
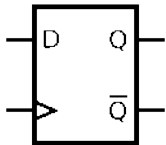
Potrzebna jest metoda skąd wziąć argumenty działania i gdzie zapisać wynik!

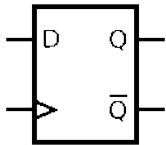
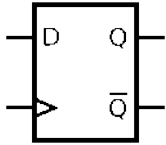
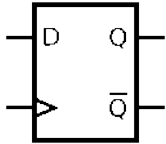
Sposób zapisywania gdzie znajduje się wartość argumentu nazywamy trybami adresowania.

Trzeba odróżnić:

- fizyczną komórkę pamięci lub rejestr
- zawartość tej komórki
- adres komórki lub rejestru

Rejestr to np. 8 przerzutników typu D. Zawartość tego rejestru to stany tych przerzutników. A adres to informacja, o który zestaw przerzutników nam chodzi.





Ogólny podział adresowania:

A) Adresowanie Bezpośrednie (w kodzie operacji):

- Zwarte (instrukcje wykonują działania same z siebie tylko na konkretnych argumentach) np. CPUID
- Natychmiastowe (na fragmencie ciągu bitów np. przez programistę jest zapisana dokładna kombinacja bitów, która ma być użyta jako argument, odwołujemy się więc do pamięci z kodem)
Np. push \$1, "gdzieś" - wprowadź wartość 1 do jakiejś komórki
- Bezwzględne (odwołujemy się do konkretnej komórki pamięci, z której weźmiemy wartość operandu)
Np. inc 100 - zwiększ wartość w komórce o adresie 100 o 1.
- Bezpośrednie rejestrowe (podobnie jak w bezwzględnym z tym, że zamiast odwoływać się do komórki pamięci zewnętrznej szukamy wartości wewnątrz określonego rejestru)
Np. inc %eax - zwiększ wartość w akumulatorze o 1

B) Adresowanie Pośrednie (w kodzie rozkazu jest informacja jak wyliczyć adres operandu, na przykład weź wartość z komórki o adresie zapisanym w akumulatorze i zwiększ ją o jeden:
inc (%rax))



W niektórych procesorach są takie śmieszne myki:

Adres z komórki prowadzi do kolejnej komórki, w której jest adres kolejnej komórki, w której jest wartość.

Tryby adresowania mogą być wielopoziomowe.

To producent określa jednak dostępne tryby adresowania i nie wszystko co przyjdzie nam do głowy może być poprawne.

Adres to informatyczna nazwa na numer komórki, jej indeks, numer początkowy!

JEST JESZCZE DRUGI ETAP!

Adresy wytwarzane w programie to **adresy logiczne**, które są następnie tłumaczone w maszynach rzeczywistych na adresy fizyczne pamięci.

Dla przykładu:

Jeśli program odpalamy w 3 osobnych instancjach, które działają jednocześnie komórka o adresie 100 musi fizycznie być w różnych miejscach dla każdego procesu.

Ten drugi etap nazywamy translacją adresu. Pojawi się na wykładzie pod koniec semestru.

W uproszczeniu tym zajmuje się system operacyjny. Te mechanizmy mają duży wpływ na wydajność!

Rejestracja

czwartek, 30 maja 2024 16:45

Rejestry ogólnego przeznaczenia (BARDZO WAŻNE, WYKUĆ NA BLACHĘ OBRAZEK):

General-Purpose Registers					
31	16	15	8	7	0
			AH	AL	
			BH	BL	
			CH	CL	
			DH	DL	
			BP		
			SI		
			DI		
			SP		
			16-bit		32-bit
			AX		EAX
			BX		EBX
			CX		ECX
			DX		EDX
					EBP
					ESI
					EDI
					ESP

x86-64 (1999)

- rax, rbx, ...
- dodatkowe r8-r15
- sil, dil, bpl, spl
- r8b, r8w, r8d, ...
- ograniczenia dla AH, BH, ...

APX (2023)

- r16-r31
- ...

Mamy 8 rejestrów A, B, C, D, BP, SI, DI, SP.

Z rejestrów możemy wybierać fragmenty. Ale tylko wybrane.

Można się odwoływać

- Do młodszych szesnastu bitów rejestru (AX, BX, CX, DX, BP, SI, DI, SP)
- Do najmłodszych 8 bitów (AL, BL, CL, DL)
- Do starszych bitów części młodszych szesnastu (AH, BH, CH, DH)
- Do wersji 32-bitowych (mają E z przodu)

L - low

H- high

W architekturach 64 bitowych:

- "r" z przodu oznacza, że rejestr jest 64-bitowy.
- dodano dodatkowe rejestry r8 - r15

CO ZAPAMIĘTAĆ:

Rejestry 32 bitowe i to, że w architekturze 64- bitowej jest więcej rejestrów

Co jeszcze warto zauważyć:

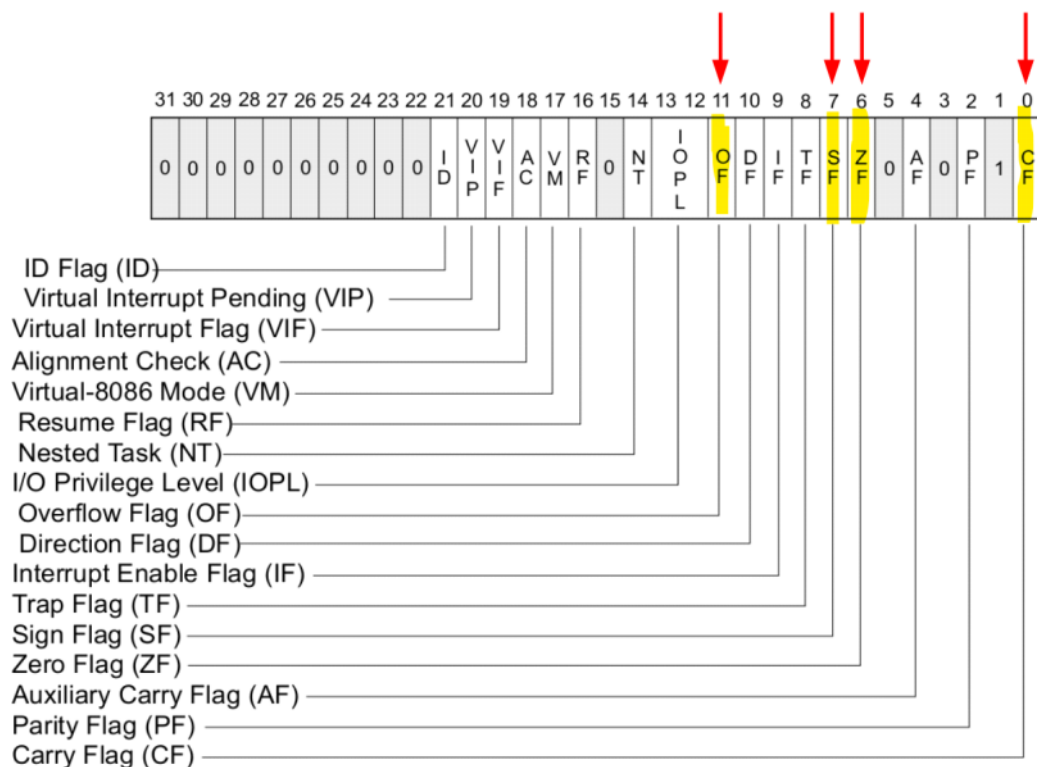
- Rejestr flag przechowujący flagi (EFLAGS - wersja 32 bitowa)
- Rejestr przechowujący adres aktualnie wykonywanej instrukcji (EIP - Instruction Pointer) Tomczak nazywa go też "licznikiem rozkazów"

To samo tylko dla architektury 64-bitowej:

64-bits	RFLAGS Register
64-bits	RIP (Instruction Pointer Register)

Rejestr flag:

Można po tym rejestrze pisać, ale nie każdy fragment można zmienić.



Ważne:

CF(Carry Flag) - flaga przeniesienia

OF (Overflow Flag) - flaga nadmiaru w U2

SF (Sign Flag) - flaga znaku, nazwana przez Tomczaka flagą "najbardziej znaczącego bitu reprezentacji U2"

ZF (Zero Flag) - flaga zera, czy wynikiem ostatniego działania było zero

Dodatkowo wspomniane:

*IF (Interrupt Enable Flag) - flaga aktywowania przerwań (przerwania są aktywowane jeśli flaga = 1)

*NT (Nested Task) - informacja o tym czy jesteśmy w zagłębiony zadaniu

*PF (Parity Flag)- flaga parzystości

I jakieś inne rzeczy

Typy danych (wspieranych przez procesor)

czwartek, 30 maja 2024 17:20

Podstawowe:

- Byte - 8 bitów (1B)
- Word - 16 bitów (2B)
- Doubleword - 32 bity (4B)

Liczbowe (używane np. w operacji dodawania):

- Kod naturalny binarny (w C nazywa się to unsigned (bez znaku))
- Reprezentacja w kodzie U2 (w C nazywa się to signed (ze znakiem))
- *BCD (rzadko się używa), każda cyfra reprezentacji dziesiętnej na 4 bitach reprezentacji binarnej, zostaje nam 6 dodatkowych 10, 11, 12, 13, 14, 15 i to już na różne sposoby wykorzystywane
- Zmiennoprzecinkowe (reprezentacje zgodne ze standardem IEEE 754) - procesor wykonuje sprzętowo działania na tych reprezentacjach

Wektorowe (Packed) - "będziemy mówili o nich pod koniec":

- Tryb wskaźnikowy - można w nim zapisać adres, dla architektury 32 bitowej - 32, dla 64-bitowej - 64

W adresowaniu pośrednim wartość w rejestrze, którą traktujemy jako adres jest typu wskaźnikowego!

*Typy wskaźnikowe występują w dwóch wariantach:

- bliskim - zajmujące się numerem komórki
- dalekim - składające się z numeru komórki i segmentu

- Dodatkowo ode mnie - sufiksy mnemoników
- b - bajt (8 bitów)
- w - słowo (16 bitów)
- l - podwójne słowo (32 bity)
- q - czterokrotne słowo (64 bity)

Np. `movl %eax, %ebx`

...

Są instrukcje, które potrafią wykonywać operacje na bitach (niedużo), stąd

Pola bitowe (ang. Bit Field)

Ciągi (ang. String) - dla instrukcji traktujących fragment pamięci jako ciąg bajtów, przeliterować po iluś bajtach i coś z nimi zrobić

Typy danych zaprezentowane na obrazku (dla architektury 64-bitowej):

a) Podstawowe:

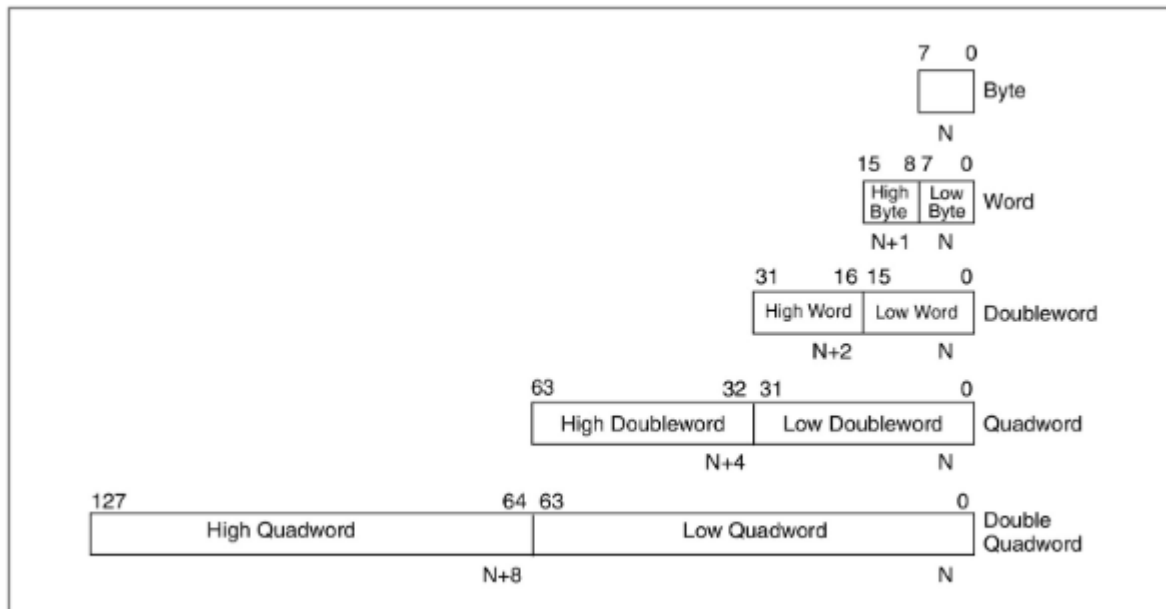


Figure 4-1. Fundamental Data Types

b) Liczbowe:

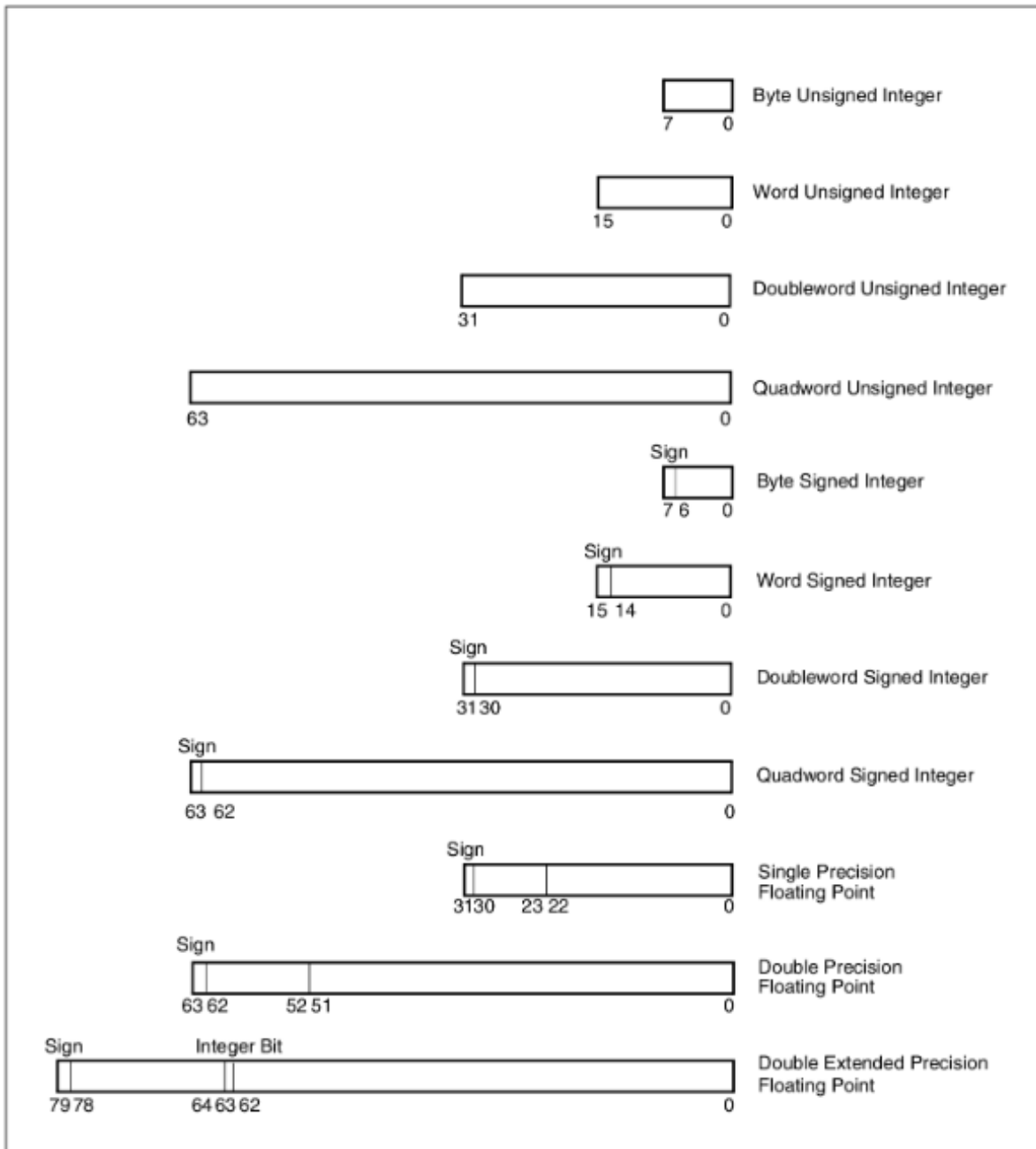


Figure 4-3. Numeric Data Types

c) Wskaźnikowe:

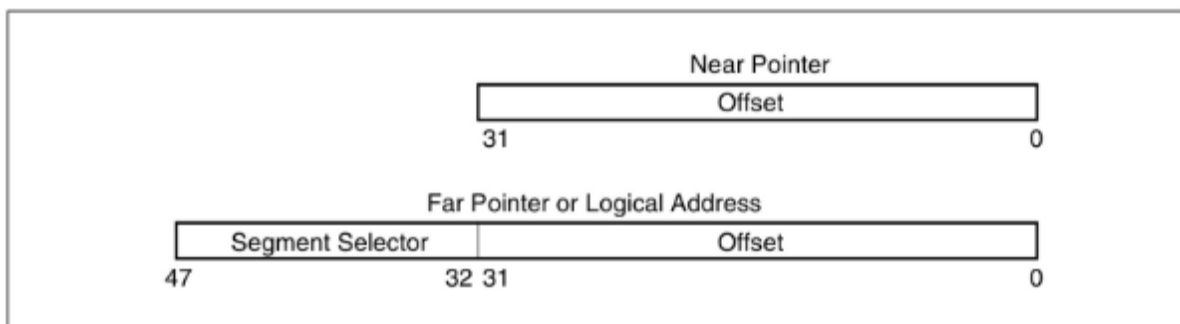


Figure 4-4. Pointer Data Types

"INDIANIN"

czwartek, 30 maja 2024 18:51



Pamięć to tablica jednobajtowych komórek. Nikt nie robi maszyn o innej szerokości podstawowego słowa i szerokości adresowania. Głównie ze względów wydajnościowych.

Jak ułożyć bajty w kilkubajtowej reprezentacji (np. word lub doubleword)?

Są dwa sposoby:

- a) Ważniejszy niższy (Little endian (tak to się pisze w ten sposób)) - najmniej znaczący bajt umieszczony jest jako pierwszy

Procesor zapisujący 32-bitowe wartości w pamięci, przykładowo 0x4A3B2C1D pod adresem 100, umieszcza dane zajmując adresy od 100 do 103 w następującej kolejności:

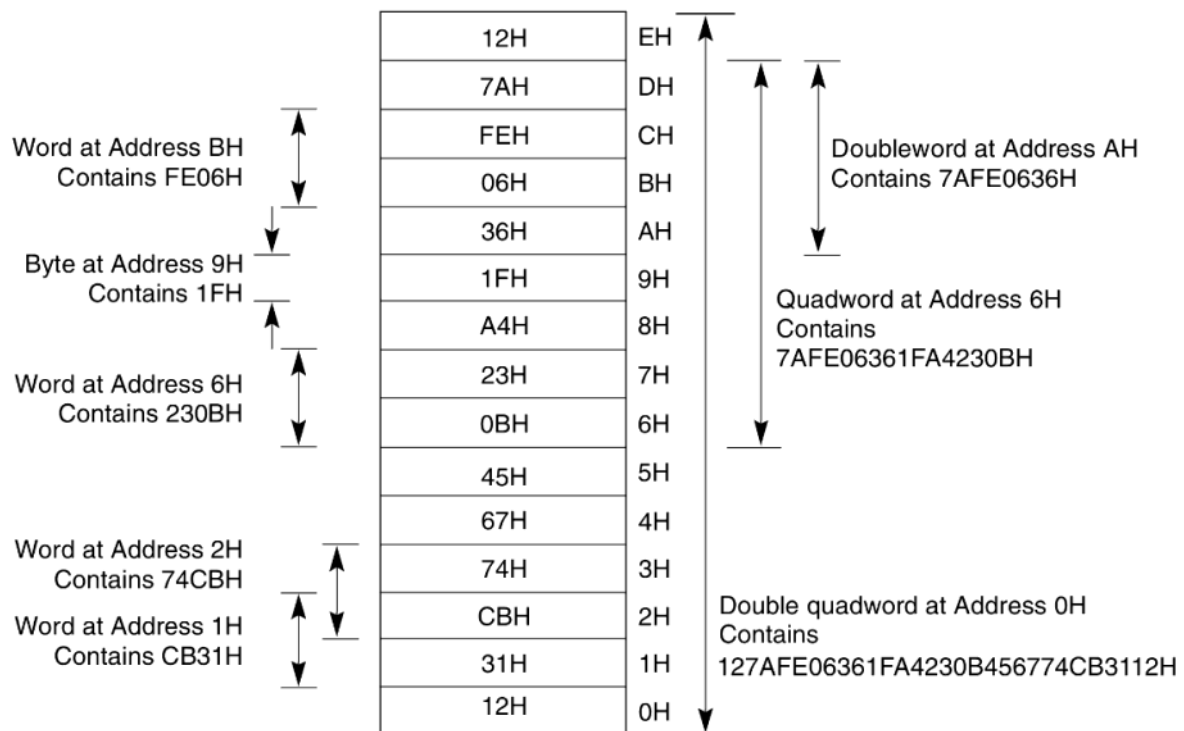
	100	101	102	103	
...	1D	2C	3B	4A	...

- a) Ważniejszy wyższy (Big endian) - najbardziej znaczący bajt umieszczony jest jako pierwszy

Procesor zapisujący 32-bitowe wartości w pamięci, przykładowo 0x4A3B2C1D pod adresem 100, umieszcza dane, zajmując adresy od 100 do 103 w następującej kolejności:

	100	101	102	103	
...	4A	3B	2C	1D	...

RYSUNEK TOMCZAKA:



Opis: Adresy rosną od dołu w górę, 0H, 1H, 2H, 3H itd.

Notacje zapisane są w "Little endian".

To jest też kolejność bajtów obowiązująca u nas na laboratoriach.

Przykład: Dla Doubleword at Address AH jest odczytywany z góry na dół, w wyższym adresie są umieszczone bardziej znaczące bajty więc odczytujemy od góry w dół.

Kiedy wyświetla się wartości poziomo widać 31CB, a nie CB31, a więc trzeba czytać od tyłu bajtami.

Pierwszy bajt jest ostatni, a ostatni pierwszy w pamięci.

H oznacza system szesnastkowy.

TRYBY ADRESOWANIA

czwartek, 30 maja 2024 20:55



Natychmiastowe (immediate) - kod argumentu jest zapisany w kodzie instrukcji
(nie mylić ze zwartymi, chodzi o to, operandy, a nie o to, że argumenty są domyślnie razem z mnemonikiem)

Rejestrowe (register) - argumenty umieszczone w rejestrach

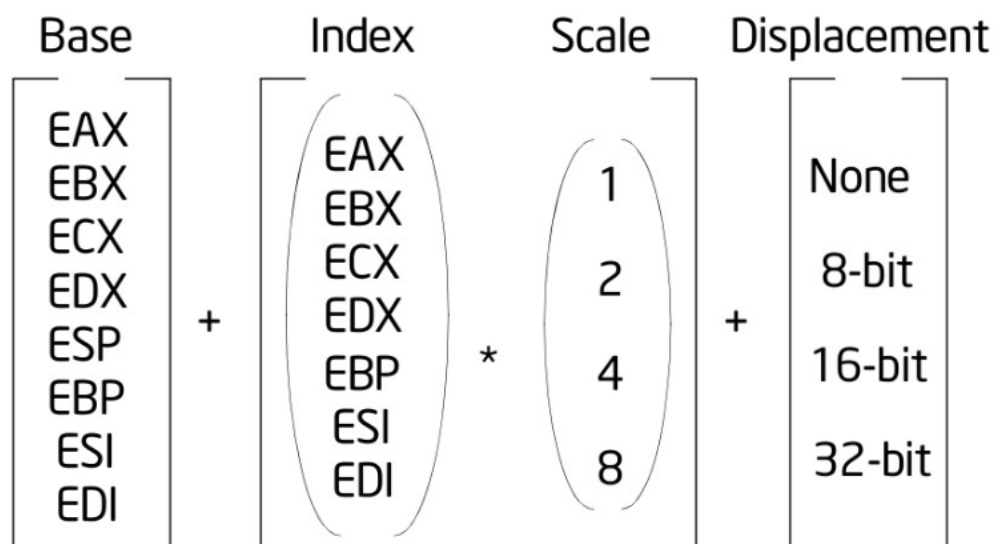
*We/Wyj (IO ports)- to nas nie interesuje

Lokacje w pamięci (memory) - umieszczone w pamięci (w komórkach), nie interesuje adres komórki tylko jej zawartość! Podobnie zresztą z rejestrami

Ale ile jest tych komórek pamięci?

10^5 (mikrokontroler 8051), 10^9 do 10^{11} (desktop), 10^{13} do 10^{16} (serwer)

Jak oblicza się adres komórki w pamięci?



$$\text{Offset} = \text{Base} + (\text{Index} * \text{Scale}) + \text{Displacement}$$

WYKUĆ NA BLACHĘ!

Przesunięcie = Rejestr bazowy + rejestr indeksowy * skala + przemieszczenie

Wynikowy kod jest traktowany jako reprezentacja w kodzie naturalnym binarnym, jest to adres komórki.

Przykład:

tablica(%edx, %esi, 4)

wskaźnik bazowy

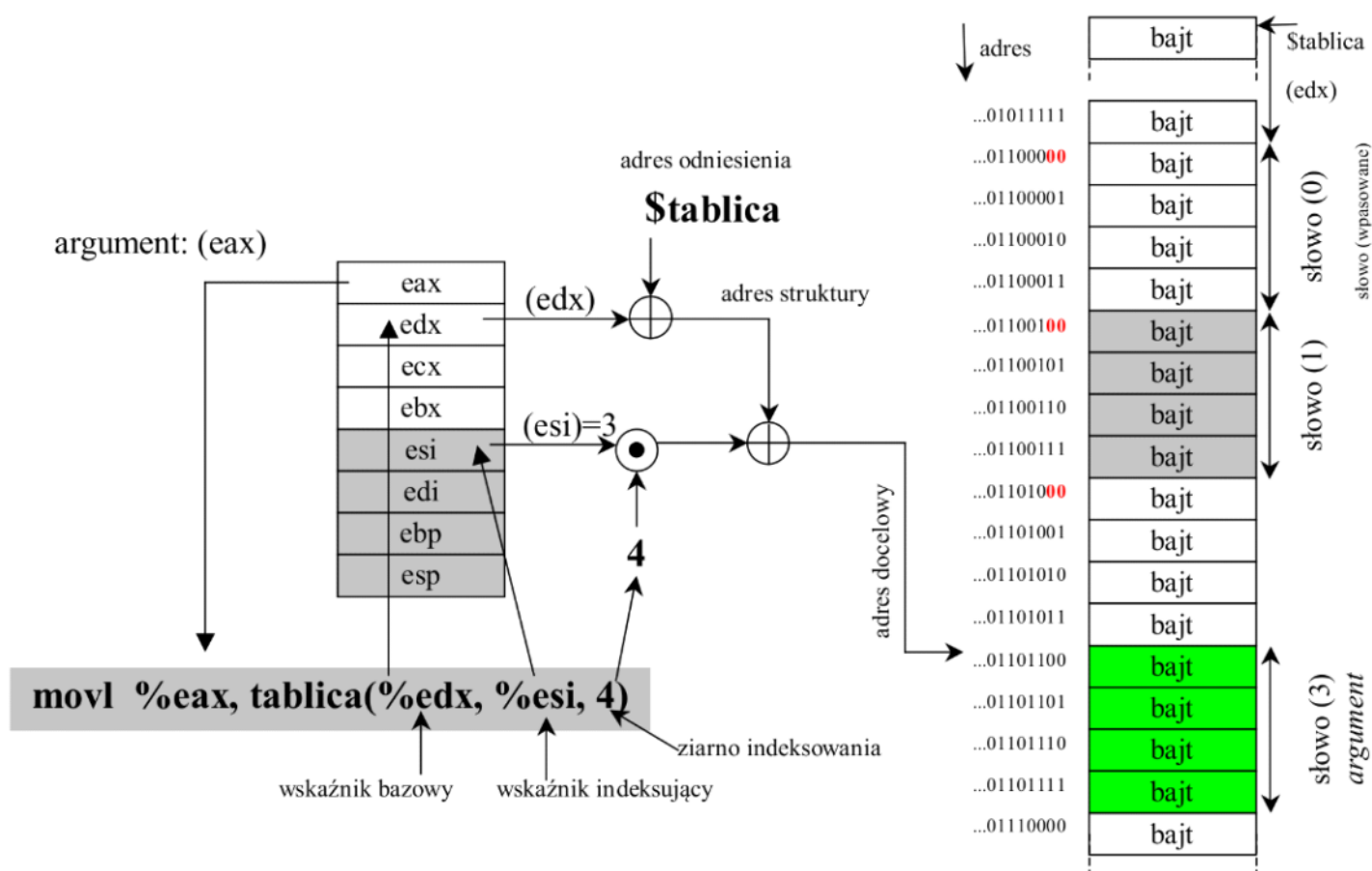
wskaźnik indeksujący

ziarno indeksowania

W podanym przykładzie:

- tablica - displacement (Przemieszczenie)
- edx - rejestr bazowy
- Esi - rejestr indeksowy
- 4 - skala (mnoży się ją razy wartość w rejestrze indeksowym)

Nie da się podać samej skali, resztę można.



Opis: W podanym przykładzie kopiujemy wartość z rejestru eax do 4 komórek w pamięci (rejestr eax ma 32 bity więc 4 bajty). Interesują nas adresy od adresu określonego jako tablica. Dodawana jest do niego wartość z rejestru edx. Przemnożona zostaje również wartość z rejestru esi z 4 i dodana do poprzedniej sumy. Mamy już nasz wynik, przesunięcie będące adresem pierwszej komórki. Wpisujemy do niego nasz quadword. Choć tutaj w intelu to ma być z pamięci do akumulatora. Ale u nich jest odwrotnie.

Zdarza się, że będzie potrzeba określenia na ilu bajtach chcemy wykonać dzielenie. Użyjemy do tego odpowiednich sufiksów:

Movd - move doubleword
Movq - move quadword
i inne

Symbol

czwartek, 30 maja 2024

21:46

- I. Symbol - twór w programie, który jest złożonym obiektem, ma **nazwę** oraz **wartość** i **inne atrybuty dla nas nie istotne**.

```
WRITE_NR = 4
```

W tym przypadku nadaliśmy WRITE_NR wartość 4 jawnie.

Ale możemy też zlecić kompilatorowi nadawanie wartości symbolom.

Etykieta też jest symbolem! Podczas tworzenia etykiety jej wartością jest adres bieżącego miejsca.

Assembler montuje obraz pamięci, czyta napisany kod po kolei i jeżeli zapiszemy w nim elementy, które powodują umieszczenie czegoś w pamięci assembler domontuje to do obrazu pamięci.

```
EXIT_NR = 1
```

```
READ_NR = 3
```


```
WRITE_NR = 4
```

```
STDOUT = 1
```

```
EXIT_CODE_SUCCESS = 0
```

Półprawda:

Symbole natomiast nie powodują umieszczenia czegoś w pamięci, bo symbole są tylko ona poziomie kodu.

EXIT_NR = 1		\$ objdump -d rw
READ_NR = 3		
WRITE_NR = 4		rw: file format elf64-x86-64
STDOUT = 1		
EXIT_CODE_SUCCESS = 0		Disassembly of section .text:
.text		0000000000400078 <.text>:
msg: .ascii "Hello, world!\n"		400078: 48 rex.W
msglen = . - msg		400079: 65 gs
		40007a: 6c insb (%dx),%es:(%rdi)
.global _start		40007b: 6c insb (%dx),%es:(%rdi)
_start:		40007c: 6f outsl %ds:(%rsi),(%dx)
		40007d: 2c 20 sub \$0x20,%al
mov \$WRITE_NR, %eax		40007f: 77 6f ja 0x4000f0
mov \$STDOUT, %ebx		400081: 72 6c jnb 0x4000ef
mov \$msg, %ecx		400083: 64 21 0a and %ecx,%fs:(%rdx)
mov \$msglen, %edx		400086: b8 04 00 00 00 mov \$0x4,%eax
int \$0x80		40008b: bb 01 00 00 00 mov \$0x1,%ebx
		400090: b9 78 00 40 00 mov \$0x400078,%ecx
mov \$EXIT_NR, %eax		400095: ba 0e 00 00 00 mov \$0xe,%edx
mov \$EXIT_CODE_SUCCESS, %ebx		40009a: cd 80 int \$0x80
int \$0x80		40009c: b8 01 00 00 00 mov \$0x1,%eax
		4000a1: bb 00 00 00 00 mov \$0x0,%ebx
		4000a6: cd 80 int \$0x80

Tomczak wyjaśnia, że samo zdefiniowanie symbolu nie umieszcza jego wartości w kodzie, dzieje się to dopiero przy zastosowaniu go np. jako argumentu, zwraca uwagę na zapis binarny.

Symbol nie jest makrem! Makro jest definiowane i używane na poziomie samego kodu. Stosuje się je, żeby nie pisać kilka razy tych samych fragmentów kodu.

Jak widać na obrazku kod binarny jest zapisany w postaci little endian, najpierw mamy wartość ostatniego bajtu czyli 4, a potem 3 zerowe bajty. Od najmniej znaczącego do najbardziej znaczącego.

Wartością etykiety _start będzie adres miejsca, w którym ona się znajduje w obrazie pamięci. Assembler dokleja do obrazu pamięci to co wynika z kodu.

Na kod powinniśmy więc patrzeć jak na pamięć.

Żeby podejrzeć wartości symboli korzystamy z polecenia "nm".

```

$ nm p.o
0000000000000000 a EXIT_CODE_SUCCESS
0000000000000001 a EXIT_NR
0000000000000003 a READ_NR
0000000000000001 a STDOUT
0000000000000004 a WRITE_NR
000000000000000e T _start
0000000000000000 t msg
000000000000000e a msgLen

$ nm p
0000000000000000 a EXIT_CODE_SUCCESS
0000000000000001 a EXIT_NR
0000000000000003 a READ_NR
0000000000000001 a STDOUT
0000000000000004 a WRITE_NR
0000000000601000 A __bss_start
0000000000601000 A _edata
0000000000601000 A _end
0000000000400086 T _start
0000000000400078 t msg
000000000000000e a msgLen

```

W pliku obiektowym niektóre symbole mają już określone wartości (są to wartości bezwzględne)

Adresy etykiet są ustalane dopiero na poziomie linkowania.

Etykiet nie zmienia się raczej w trakcie działania programu. Tomczak tak nigdy nie robił.

```

1          # Numbers of kernel functions.
2          EXIT_NR  = 1
3          READ_NR  = 3
4          WRITE_NR = 4
5
6          STDOUT = 1
7          EXIT_CODE_SUCCESS = 0
8
9
10         .text
11 0000 48656C6C      msg: .ascii "Hello, world!\n"
11      6F2C2077
11      6F726C64
11      210A
12          msgLen = . - msg
13
14
15         .global _start
16
17         _start:
18
19 000e B8040000      mov $WRITE_NR, %eax
19      00
20 0013 B8010000      mov $STDOUT , %ebx
20      00
21 0018 B9000000      mov $msg      , %ecx
21      00
22 001d BA0E0000      mov $msgLen  , %edx
22      00
23 0022 CD80          int $0x80
24
25
26 0024 B8010000      mov $EXIT_NR      , %eax
26      00
27 0029 BB000000      mov $EXIT_CODE_SUCCESS, %ebx
27      00
28 002e CD80          int $0x80
29

```

Tomczak opisuje powyższy kod, mówi o dyrektywach, o tym, że jeśli coś jest poprzedzone kropką to na prawie na pewno jest to dyrektywa. Opisuje dyrektywę `.text`, która określa, że wszystko od niej ma być dołączone do sekcji kodu.

- II. Sekcje to ciągłe obszary pamięci, można je potem umieszczać w różnych miejscach. Jest ich wiele. Z naszej perspektywy są w zasadzie 3:
 - a) Sekcja kodu **.text**:
Miejsce na kody operacji i trochę innych rzeczy
 - b) Sekcja danych zainicjowanych **.data**:
Pozwala umieścić wiele różnych rzeczy, które nie będą traktowane domyślnie jako kody operacji
 - c) Sekcja danych niezainicjowanych **.bss**:
Miejsce, dla rzeczy, które na początku mogą być wyzerowane i dzięki temu nie trzeba zapisywać ich wartości tylko zapisywać jak dużą sekcję `bss` chcemy. Dzięki temu plik jest mniejszy. (Ta sekcja zajmuje miejsce w

pamięci po załadowaniu, ale nie zajmuje miejsca w pliku, tylko informacja, że potrzebujemy miliard bajtów)

Procesor Tomczaka obsługuje trzy prawa dostępu:

- a) Zapis
- b) Odczyt
- c) Wykonywanie

W sekcji kodu można czytać dane i je wykonywać. Nie można do niej pisać.

W sekcji danych można czytać i pisać. Nie można wykonywać.

W sekcji bss tak samo jak w sekcji danych.

Etykieta nie została jeszcze określona, w pamięci 0000

.ascii dyrektywa umieszczająca w pamięci kody ascii liter przekazanych jako argument. Widać w pamięci 48(H)65(e)6C(l)6C(l) itd. I znak nowej linii.

Uwaga! Umieścić tę wartość w pamięci moglibyśmy też za pomocą mnemoników.

msgLen - określenie długości wiadomości odejmując od bieżącego adresu adres etykiety czyli początku ciągu znaków.

Stąd właśnie nazwa Assembler (montować), program domontowuje kolejne rozkazy w pamięci. W tej architekturze różne instrukcje zajmują różną ilość miejsca w pamięci.

Tomczak pokazuje jak to dokładnie wygląda w Intelu, mówi o tym, że jest to skomplikowane, ale jednoznaczne dla komputera.

Symbole są zastępowane wartościami podczas asemblacji, linkowania, ładowania programu, a nawet jego wykonywania (ang. lazy binding/linking)

Stosuje się różnie, żeby szybciej inicjalizować program. Dla "lazy" dopiero kiedy symbol jest potrzebny zostaje określony.

A jest jeszcze wersja gdzie programista może sobie jawnie załadować bibliotekę i pobrać adres funkcji z tej biblioteki.

"Objdumb" ;)

czwartek, 30 maja 2024 22:34

Wracamy do deasemblacji.

```
EXIT_NR = 1
READ_NR = 3
WRITE_NR = 4
STDOUT = 1
EXIT_CODE_SUCCESS = 0

.text
msg: .ascii "Hello, world!\n"
msglen = . - msg

.global _start
_start:

mov $WRITE_NR, %eax
mov $STDOUT, %ebx
mov $msg, %ecx
mov $msglen, %edx
int $0x80

mov $EXIT_NR, %eax
mov $EXIT_CODE_SUCCESS, %ebx
int $0x80
```



```
$ objdump -d rw

rw:      file format elf64-x86-64

Disassembly of section .text:

0000000000400078 <.text>:
400078: 48                rex.W
400079: 65                gs
40007a: 6c                insb  (%dx),%es:(%rdi)
40007b: 6c                insb  (%dx),%es:(%rdi)
40007c: 6f                outsl  %ds:(%rsi),(%dx)
40007d: 2c 20            sub   $0x20,%al
40007f: 77 6f            ja    0x4000f0
400081: 72 6c            jnb   0x4000ef
400083: 64 21 0a         and   %ecx,%fs:(%rdx)
400086: b8 04 00 00 00   mov   $0x4,%eax
40008b: bb 01 00 00 00   mov   $0x1,%ebx
400090: b9 78 00 40 00   mov   $0x400078,%ecx
400095: ba 0e 00 00 00   mov   $0xe,%edx
40009a: cd 80            int   $0x80
40009c: b8 01 00 00 00   mov   $0x1,%eax
4000a1: bb 00 00 00 00   mov   $0x0,%ebx
4000a6: cd 80            int   $0x80
```

Jak widać objdump (narzędzie deasemblujące obraz programu), przypisał zlepkom bajtów (kodom ascii) instrukcje. A nasz kod w rzeczywistości zaczyna się dopiero od `mov $WRITE_NR, %eax`

WIĘCEJ TRYBÓW ADRESOWANIA!

czwartek, 30 maja 2024 23:12

AT&T: `'-4(%ebp)'`, Intel: `'[ebp - 4]'`

base is `'%ebp'`; disp is `'-4'`. section is missing, and the default section is used (`'%ss'` for addressing with `'%ebp'` as the base register). index, scale are both missing.

AT&T: `'foo(,%eax,4)'`, Intel: `'[foo + eax*4]'`

index is `'%eax'` (scaled by a scale 4); disp is `'foo'`. All other fields are missing. The section register here defaults to `'%ds'`.

AT&T: `'foo(,1)'`; Intel `'[foo]'`

This uses the value pointed to by `'foo'` as a memory operand. Note that base and index are both missing, but there is only one `'.'`. This is a syntactic exception.

Tomczak pokazuje przykłady:

1. Przesunięcie i rejestr bazowy
2. Przesunięcie rejestr indeksowy i skala , - dla pokazania, że nie ma bazowego
3. Przesunięcie i rejestr indeksowy

Jak czytać dokumentację?

czwartek, 30 maja 2024 23:16

The screenshot shows the Intel 64 and IA-32 Architectures Software Developer's Manual. The left pane displays the 'Index' with a tree view under 'Chapter 4 Instruction Set Reference, N-Z'. The 'NOP—No Operation' entry is selected, showing its page number 1080. The main pane displays the 'INSTRUCTION SET REFERENCE, N-Z' section for the 'NOP—No Operation' instruction. It includes a table of opcodes, a table of instruction operand encoding, a description, an operation section, and a table of recommended multi-byte sequences. Red circles highlight the 'NOP' instruction in the opcode table and the 'Flags Affected' section.

Index

- Chapter 4 Instruction Set Reference, N-Z 1073
 - 4.1 Imm8 Control Byte Operation for PCPESTR / PCPESTRM / ... 1073
 - 4.2 Instructions (N-Z) 1077
 - NEG—Two's Complement Negation 1078
 - NOP—No Operation 1080**
 - NOT—One's Complement Negation 1081
 - OR—Logical Inclusive OR 1083
 - ORPD—Bitwise Logical OR of Double-Precision Floating-Point ... 1085
 - ORPS—Bitwise Logical OR of Single-Precision Floating-Point V... 1087
 - OUT—Output to Port 1089
 - OUTS/OUTSB/OUTSW/OUTSD—Output String to Port 1091
 - PABSB/PABSW/PABSD—Packed Absolute Value 1095
 - PACKSSWB/PACKSSDW—Pack with Signed Saturation 1099
 - PACKUSDW—Pack with Unsigned Saturation 1104
 - PACKUSWB—Pack with Unsigned Saturation 1107
 - PADDB/PADWD/PADDQ—Add Packed Integers 1110
 - PADDQ—Add Packed Quadword Integers 1114
 - PADDSB/PADDSW—Add Packed Signed Integers with Signed ... 1116
 - PADDUSB/PADDUSW—Add Packed Unsigned Integers with Un... 1119
 - PALIGNR—Packed Align Right 1122
 - PAND—Logical AND 1125
 - PANDN—Logical AND NOT 1127
 - PAUSE—Spin Loop Hint 1129
 - PAB/GB/PAB/GW—Average Packed Integers 1130
 - PBLENDB—Variable Blend Packed Bytes 1133
 - PBLENDD—Blend Packed Words 1137
 - PCLMULQDQ—Carry-Less Multiplication Quadword 1140
 - PCMPQB/PCMPQW/PCMPQD—Compare Packed Data for E... 1143
 - PCMPQEQ—Compare Packed Qword Data for Equal 1147
 - PCPESTR—Packed Compare Explicit Length Strings, Retur... 1149
 - PCPESTRM—Packed Compare Explicit Length Strings, Ret... 1151
 - PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Int... 1153
 - PCMPGTQ—Compare Packed Data for Greater Than 1157
 - PCMPISTR—Packed Compare Implicit Length Strings, Retur... 1159
 - PCMPISTRM—Packed Compare Implicit Length Strings, Retu... 1161
 - PDEP—Parallel Bits Deposit 1163
 - PEXT—Parallel Bits Extract 1165
 - PEXTRB/PEXTRD/PEXTRQ—Extract Byte/Dword/Qword 1167
 - PEXTRW—Extract Word 1170
 - PHADDB/PHADDQ—Packed Horizontal Add 1173
 - PHADDSW—Packed Horizontal Add and Saturate 1177
 - PHMINPOSUW—Packed Horizontal Word Minimum 1179
 - PHSUBW/PHSUBD—Packed Horizontal Subtract 1181
 - PHSUBSW—Packed Horizontal Subtract and Saturate 1184
 - PINSRB/PINSRD/PINSRQ—Insert Byte/Dword/Qword 1186
 - PINSRW—Insert Word 1188
 - PMADDUSBW—Multiply and Add Packed Signed and Unsign... 1190
 - PMADDWD—Multiply and Add Packed Integers 1192

INSTRUCTION SET REFERENCE, N-Z

NOP—No Operation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
90	NOP	NP	Valid	Valid	One byte no-operation instruction.
0F 1F 0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
0F 1F 0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
M	ModRm/r/m(r)	NA	NA	NA

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-9. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None.

Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

4-B Vol. 2B NOP—No Operation

Instrukcja NOP - nic nie robi ale zajmuje miejsce w pamięci i trochę czasu.

Pierwsze zaznaczenie - określenie możliwych argumentów instrukcji

Description (opis) - z naszego punktu widzenia najważniejsze pierwsze zdania

Drugie zaznaczenie - zmieniane flagi

Na podstawie flag program może zachowywać się w inny sposób

Instrukcje

- Przesłania danych (MOV, XCHG, PUSH, POP)
- Operacje logiczne (NOT, NEG, OR, AND, XOR, BT, CLC, STC)
- Zmiana formatu (CWD, CDQ, CQO, przesunięcia SAR, SHR, SAL, SHL, rotacje ROR, RCR, ROL, RCL)
- Operacje arytmetyczne (ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC)
- Rozgałęzienia (JMP, CALL, RET, skoki warunkowe CMP/TEST + Jcc)

Tomczak po prostu wymienia szybko rodzaje instrukcji.

Instrukcja MOV:

MOV—Move

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8 ^{***} ,r8 ^{***}	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8 ^{***} ,r/m8 ^{***}	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg ^{**}	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r/m64,Sreg ^{**}	MR	Valid	Valid	Move zero extended 16-bit segment register to r/m64.
8E /r	MOV Sreg,r/m16 ^{**}	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64 ^{**}	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8 [*]	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8 [*]	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16 [*]	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32 [*]	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64 [*]	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8 ^{***} ,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16 [*] ,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32 [*] ,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64 [*] ,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8,imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8 ^{***} ,imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16,imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32,imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64,imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /O ib	MOV r/m8,imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /O ib	MOV r/m8 ^{***} ,imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /O iw	MOV r/m16,imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /O id	MOV r/m32,imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /O io	MOV r/m64,imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

Jak widać tą instrukcją możemy przesyłać dane z rejestru do pamięci lub rejestru o tym samym rozmiarze.

Potem z pamięci do rejestru o tym samym rozmiarze.

Osobne rozkazy dla przesyłania z pamięci do akumulatora.

Nie ma rozkazów przesyłania z pamięci do pamięci.

DZIĘKUJĘ BARDZO ZA UWAGĘ!