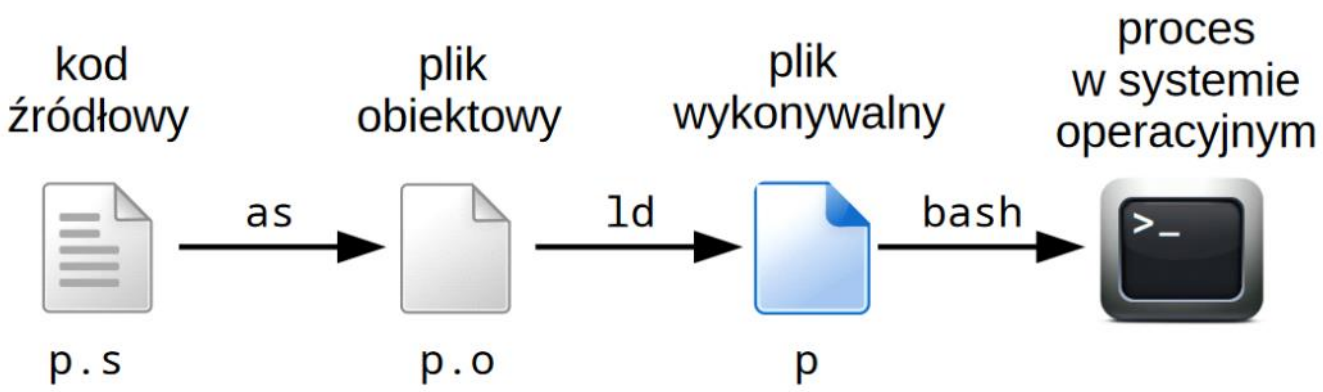
POWTARZAM

piątek, 31 maja 2024 14:02



Tomczak powtarza to co wyżej, wykład 1, strona: Programowanie.
Mówi o przekształcaniu z pliku źrodłowego do pliku wykonywalnego.
Zwraca uwagę, że nie jest to to samo.
Podkreśla znaczenie rozumienia związania napisanego kodu z pamięcią.
Mówi o tym, że wszystko co napiszemy jest przekształcane do reprezentacji binarnej, bajtów w pamięci.
Ważne jest to, że to co napiszemy nie jest bezpośrednio w pliku wykonywalnym tylko jest przekształcane.
Jest to robione przez "narzędzie tłumaczące".
W kodzie maszynowym są po prostu ciągi liczby i tyle.

```
$ 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
```

Powtarza o nieustaleniu adresów na poziomie pliku obiektowego. (etykieta msg)

```
$ 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
```

Po linkowaniu adresy są ustalane.

Czytanie i Pisanie

piątek, 31 maja 2024 14:23

Tu niżej jest błąd, powinien być "write".

$ man 2 read

```
WRITE(2)                          Linux Programmer's Manual                          WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the file referred to by the file
       descriptor fd.

       The number of bytes written may be less than count if, for example, there is insufficient  space  on
       the  underlying  physical  medium,  or  the  RLIMIT_FSIZE  resource  limit is encountered (see setr-
```

/usr/include/asm/unistd_32.h

```
1 #ifndef _ASM_X86_UNISTD_32_H
2 #define _ASM_X86_UNISTD_32_H 1
3
4 #define __NR_restart_syscall 0
5 #define __NR_exit 1
6 #define __NR_fork 2
7 #define __NR_read 3
8 #define __NR_write 4
9 #define __NR_open 5
10 #define __NR_close 6
11 #define __NR_waitpid 7
12 #define __NR_creat 8
13 #define __NR_link 9
14 #define __NR_unlink 10
15 #define __NR_execve 11
```

```
24 mov $WRITE_NR, %eax
25 mov $STDOUT  , %ebx
26 mov $msg     , %ecx
27 mov $msgLen  , %edx
28 int $0x80
```

37

mov $STDOUT , %ebx - deskryptor pliku
mov $msg , %ecx - wskaźnik (adres obszaru pamięci, z którego będzie przepisywana kolejna sekwencja bajtów do strumienia)
mov $msgLen, %edx - liczba bajtów, które chcemy przesłać

Do akumulatora trafia też numer funkcji, nas interesują trzy:

a) EXIT - 1,
b) READ - 3,
c) WRITE - 4,

Po załadowaniu wszystkich rejestrów wykonujemy przerwanie programowe za pomocą
int $0x80
Tomczak wspomina też o tym, że nagłówek w c jest

ASCII

piątek, 31 maja 2024 15:13

Kod ASCII (część międzynarodowa) = 0 | | ISO-7 (CCITT No 5)

H \ L	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0010	SP	!	″	#	$	%	&	‘	(	)	*	+	,	–	.	/
0011	**0**	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	—
0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

NUL *nullify*	SOH – *start of header*	STX – *start of text*	ETX – *end of text*
EOT – *end of transfer*	ENQ – *enquire*	ACK – *acknowledge*	BEL – *bell*
BS – *backspace*	HT – *horizontal tab*	LF – *line feed*	VT – *vertical tab*
FF – *form feed*	CR – *carriage return*	SO / SI – *shift out i in*	DLE – *data link ESC*
DC1,...4 – *data control*	NAK – *negative ACK*	SYN – *synchronize*	ETB – *end of text block*
CAN – *cancel*	EM – *end of medium*	SUB – *substitute*	ESC – *escape*
FS – *file separator*	GS – *group separator*	RS – *record separator*	US – *unit separator*

UNICODE – kod 16/32-bitowy, obejmujący znaki diaktryczne większości języków

H - bardziej znacząca
L - mniej znacząca

Regularności:

a) Duże i małe litery różnią się jednym bitem, 3 od początku
(dla małych jest jedynką, a dla dużych zerem, a więc A+ 32 = a)
b) Cyfry mają wyższą część równą 0011 czyli 3 szesnastkowo i dziesiętnie.

Tomczak mówi o sztuczkach związanych z ASCII, przykład:
Jeśli do kodu ASCII A dodamy jedynkę wyjdzie nam kod ASCII litery B.
Na kodach ASCII można w zasadzie wykonywać działania arytmetyczne.

Literały znakowe:

A single character may be written as a single quote immediately followed by that character.

Some backslash escapes apply to characters, \b, \f, \n, \r, \ t, and \" with the same meaning as for strings, plus \' for a single quote. So if you want to write the character backslash, you must write '\\ where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement.

The value of a character constant in a numeric expression is the machine's byte-wide code for that character. as assumes your character code is ASCII: 'A means 65, 'B 44 means 66, and so on.

As przekształca literały znakowe na kody ASCII na liczby, A to 0100 0001 czyli
64 + 1 = 65 .

Align

piątek, 31 maja 2024 15:43

Są instrukcje, które nakładają pewne ograniczenia na adresy, ale póki co te, których używamy mogą przyjmować adresy dowolne.
JEDNAKŻE… NIE KAŻDY ADRES JEDNAKOWO DOBRZE DZIAŁA

Systemy komputerowe nie przesyłają danych pomiędzy pamięcią a procesorem po jednym bajcie tylko przesyłają magistralami o trochę większej szerokości. Pamięci typu DIMM (ang. *Dual In-Line Memory Module*) mają 64-bitowe lub 32- bitowe magistrale. DDR 1-4 mają 64 bitowe, a DDR5 dla lepszej wydajności każdy kanał ma 32 bity. W jendym cyklu zegara przesyłane jest takie słowo.

Uproszczenie:
Jeśli nasze słowo mieści się w szerokości magistrali może zostać przesłane w jednym cyklu.
Podobnie dla słowa większego niż 4 bajty dla magistrali 4-bajtowej musimy wykonać dwa transfery, jeden kawałek danych w jednym słowie, a drugi w drugim słowie.
Jeśli adres początkowy bloku bajtów będzie podzielny przez 3 będzie można takie 4 bajty
przesłać jednym ruchem.
Te ograniczenia występują w systemach komputerowych na wielu różnych poziomach.

Mechanizmy translacji adresu są robione teraz w taki sposób, że przekształcana jest tylko starsza część adresu, najmniej znaczące bity adresu (zazwyczaj 12 bitów)

Załóżmy, że mamy 32 bitowy adres. Dzielimy go na 12 najmniej znaczących bitów i resztę.

- Jak to widzi CHATGPT:
- **Adres logiczny**: 10101111110011010101010100101010
  - Niższe 12 bitów (offset): 010101001010
  - Wyższe 20 bitów (numer strony): 10101111110011010101
- **Translacja numeru strony**:
  - Załóżmy, że numer strony 10101111110011010101 jest mapowany na numer strony 00111010100101100011 w pamięci fizycznej.
- **Składanie adresu fizycznego**:
  - Numer strony fizycznej: 00111010100101100011
  - Łączymy go z offsetem: 00111010100101100011|010101001010

Ta reszta jest jakoś przekształcana i potem do niej jest doklejane to 12 bitów. 12 bitów się nie zmienia. Cała reszta się zmienia. W niewielkim obszarze pamięci adresy logiczne są takie same jak fizyczne.

I. Wyrównanie adresów - Zachowanie ułożenia

……
……

. align - dyrektywa służąca do tego, żeby na etapie tworzenia obrazu pamięci zagwarantować, że pierwszy adres, który po tej dyrektywie występuje będzie podzielny przez potęgę dwójki

7.3 .align [abs-expr[, abs-expr[, abs-expr]]]

Pad the location counter (in the current subsection) to a particular storage boundary.

The first expression (which must be absolute) is the alignment required, as described below. If this expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the arc, hppa, i386 using ELF, iq2000, m68k, or1k, s390, sparc, tic4x and xtensa, the first expression is the alignment request in bytes. For example '.align 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. For the tic54x, the first expression is the alignment request in words.

For other systems, including ppc, i386 using a.out format, arm and strongarm, it is the number of low-order zero bits the location counter must have after advancement. For example '.align 3' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides .balign and .p2align directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

Przykład:
Mamy w obrazie pamięci umieszczone 3 bajty od początku sekcji.
Używamy .align gwarantującego, że adres będzie podzielny przez 8 to zostanie przeskoczone 5 bajtów tak, żeby następnym adresem była ósemka.

Można określić co wpisać w te przeskakiwane bajty, określić ograniczenia dodatkowe,

Jak chcemy ten adres wyrównywać. Jedne systemy wymagają podania wartości do wyrówania, inne potęgi dwójki.

Wszystkie niuanse wyżej w dokumentacji.

POWTÓRZENIE

piątek, 31 maja 2024 16:27

Tomczak powtarza jak czytać Instruction Set Reference.
Zwraca uwagę na dostępne argumenty, kod binarny po lewej, opis, z którego interesuje nas kilka pierwszych linii oraz modyfikowane **flag**i oraz jak ważne, żeby **ich** pilnować szczególnie dla rzadkich przypadków błędów.

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.

"MNEMOR(E)Y"

piątek, 31 maja 2024 16:34

Mnemonik	Pełna nazwa	Rodzaj operacji	Typ
add	*add*	dodaj	A
sub	*subtract*	odejmij	A
mul, mpy	*multiply*	pomnóż	A
div	*divide*	podziel	A
cmp, cp	*compare*	porównaj (określ relację)	A
test	*test*	porównaj (sprawdź zgodność)	L
and	*and*	iloczyn logiczny	L
or	*or*	suma logiczna	L
xor	*exclusive-or*	suma wyłączająca (modulo 2)	L
inc / dec	*increment/decrement*	zwiększ / zmniejsz	K
shr / shl	*shift right/left*	przesuń w prawo / lewo	K
rr / rl	*rotate right/left*	przesuń cyklicznie w prawo / lewo	K
mov(e)	*move*	kopiuj (przemieść)	T
ld, load	*load*	pobierz (z pamięci) do rejestru	T
st, store	*store*	zapisz do pamięci (z rejestru)	T
bcc, jcc	*branch | conditional jump*	rozgałęziaj (wybierz ścieżkę)	T
call, jsr	*call procedure*	wywołaj procedurę	T

Tomczak opisuje krótko wybrane mnemoniki. Wspomina, że mogą one wyglądać inaczej dla różnych systemów.

Mov - rozkaz uniwersalny używany w architekturze x86 do kopiowania danych w zasadzie między dowolnymi miejscami
W procesorach konstruowanych do bardzo szybkiego wykonywania operacji wyraźnie odróżnia się działania na pamięci od działań na rejestrach. Wtedy większość rozkazów działa wyłącznie na rejestrach natomiast dane między rejestrami a pamięcią przesyła się specjalnymi rozkazami. Rozkazy load ładują rozkazy z pamięci do rejstru, a store z rejstru do pamięci.
W Intelu tego rodzaju rozkazy nie występują, ale w maszynach do obliczeń DSP (ang. digital signal processing)np. kartach graficznych już tak.
*DSP to cyfrowe przetwarzanie sygnałów rozumianych kiedyś jako sygnały dźwiękowe, ale w tej chwili jako sygnały graficzne czy inne.

Skoki:
W intelu "jump" stąd jcc
W innych systemach "branch" stąd bcc.

MOV & XCHG

piątek, 31 maja 2024 16:35

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 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /0 ib	MOV r/m8***, imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /0 iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /0 id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /0 io	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

Tomczak zagląda do funkcji mov.

Szczególna uwaga:

Rejestry segmentowe nas nie interesują

Ciekawa funkcja mov, rozszerza szesnastobitowy rejestr (co prawda segmentowy ale no) zerami do 64-bitowego

REX.W + 8C /r	MOV r/m64,Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r/m64.

Większość rozkazów nie modyfikuje szerokości argumentów, rozkazy przesyłania arguemtnów natychmiastowych do rejestru argument natychmiastowy musi być taki sam jak rejestr.

xchg (ang. *exchange*) – wymień, kopiuj wzajemnie

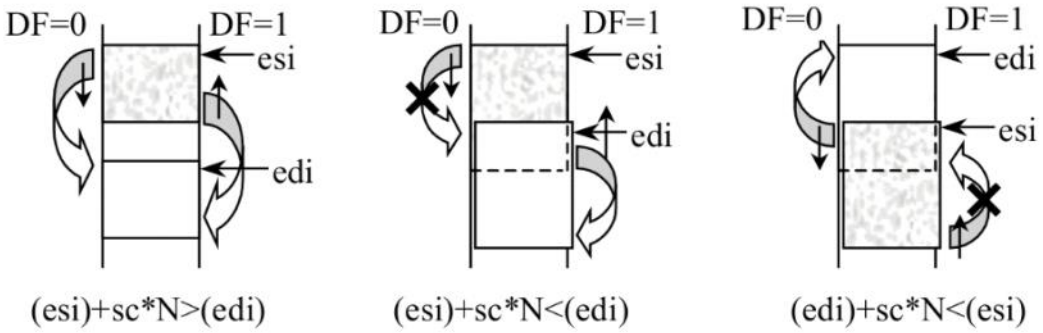
xchgl %eax, %ebx **- zawartość rejestru eax skopiuj do rejestru ebx a zawartość ebx do eax**

XCHG - funkcja jednocześnie odczytuje dwa argumenty, zamienia się je miejscami i jednocześnie zapisuje.

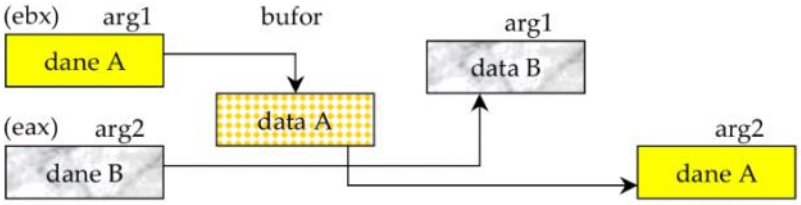
DYGRESJA: SĄ INSTRUKCJE OPERUJĄCE NA DANCYH ŁAŃUCHOWYCH CZYLI CIĄGU BAJTÓW

KTO PIERWSZY TEN BŁĄD

piątek, 31 maja 2024 17:25



Transfer łańcucha słów **movs**



wymiana bez bufora:

```
xor %eax, %ebx
xor %ebx, %eax
xor %eax, %ebx
```

Kopiowanie wzajemne (z wymianą): **xchg** %ebx, %eax

Kolejność kopiowania jest ważna, jeśli kopiujemy jeden obszar pamięci do drugiego obszaru pamięci i te dwa obszary na siebie zachodzą to w zależności od tego czy ten, z którego kopiujemy nachodzi na tego do którego kopiujemy czy odwrotnie to musimy od dobrej strony zacząć kopiować.

Biały zaczyna się w środku szarego. Trochę poniżej, ale w środku jeśli z szarego kopiujemy pierwszy element do pierwszego elementu białego to zamażemy przy okazji element szarego.

Trzeba zacząć od końca szarego do końca białego. To jest oczywiste, ale za pierwszym razem może nie być.

ADC & CLC

piątek, 31 maja 2024 17:57

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	I	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	I	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	I	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	I	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	MI	Valid	Valid	Add with carry *imm8* to *r/m8*.
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	MI	Valid	N.E.	Add with carry *imm8* to *r/m8*.
81 /2 *iw*	ADC *r/m16, imm16*	MI	Valid	Valid	Add with carry *imm16* to *r/m16*.
81 /2 *id*	ADC *r/m32, imm32*	MI	Valid	Valid	Add with CF *imm32* to *r/m32*.
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	MI	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64*.
83 /2 *ib*	ADC *r/m16, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16*.
83 /2 *ib*	ADC *r/m32, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32*.
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	MI	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64*.
10 /*r*	ADC *r/m8, r8*	MR	Valid	Valid	Add with carry byte register to *r/m8*.
REX + 10 /*r*	ADC *r/m8**, *r8**	MR	Valid	N.E.	Add with carry byte register to *r/m64*.
11 /*r*	ADC *r/m16, r16*	MR	Valid	Valid	Add with carry *r16* to *r/m16*.
11 /*r*	ADC *r/m32, r32*	MR	Valid	Valid	Add with CF *r32* to *r/m32*.
REX.W + 11 /*r*	ADC *r/m64, r64*	MR	Valid	N.E.	Add with CF *r64* to *r/m64*.
12 /*r*	ADC *r8, r/m8*	RM	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /*r*	ADC *r8**, *r/m8**	RM	Valid	N.E.	Add with carry *r/m64* to byte register.
13 /*r*	ADC *r16, r/m16*	RM	Valid	Valid	Add with carry *r/m16* to *r16*.
13 /*r*	ADC *r32, r/m32*	RM	Valid	Valid	Add with CF *r/m32* to *r32*.
REX.W + 13 /*r*	ADC *r64, r/m64*	RM	Valid	N.E.	Add with CF *r/m64* to *r64*.

Ten rozkaz to dodawanie trójargumentowe, do argumentu drugiego dodajemy pierwszy
i dodajemy do nich dodatkowo flagę CARRY(zero lub jeden) first + second + CF, dodatkowo flaga carry jest przez ten rozkaz ustawiana, jeśli realizujemy dodawanie tych argumentów to w argumencie docelowym będziemy mieć bity sumy i zmienamiy flagę carry na zero lub jeden zawsze. Ustawia flagi: OF, SF, ZF, AF, CF, i PF.

Szczególna uwaga:

Specjalna wersja, która do pamięci różnego rodzaju potrafi dodawać argument natychmiastowy ośmiobitowy

83 /2 *ib*	ADC *r/m16, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16*.
83 /2 *ib*	ADC *r/m32, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32*.
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	MI	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64*.

Przykład:
Jeśli dodajemy reprezentacje o rozmiarze większym niż słowo maszynowe to musimy zagwarantować, że przy dodawaniu kolejnych cyfr reprezentacji np. 32 bitowych przeniesienie będzie propagowało.

Jeśli chcemy dodać bez przeniesienia użyjemy :
CLC - rozkaz zerujący flagę Carry + ADC

lub
ADD - dodawanie bez przeniesienia

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /r	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /r	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /r	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /r	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /r	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /r	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /r	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /r	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /r	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /r	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

UWAGA! DODAWANIE W NB i U2 jest takie samo, inne są tylko warunki wykrywania nadmiaru, dlatego są dwie flagi. OF do U2 i CF do NB.

Rozszerzenia znakowe

piątek, 31 maja 2024 18:11

Instrukcje do generowania cyfr rozszerzeń znakowanych:

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD	NP	Valid	Valid	DX:AX ← sign-extend of AX.
99	CDQ	NP	Valid	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO	NP	Valid	N.E.	RDX:RAX← sign-extend of RAX.

CWD - Do dwóch rejestrów DX:AX wpisane zostaje rozszerzenie znakowane AX, domyślnym argumentem jest akumulator. EDX jest wypełniany zerami lub jedynkami w zależności od tego czy bit znaku był ustawiony w AX.

Do generowania cyfr rozszerzeń nieoznakowanych nie potrzebujemy osobnej instrukcji bo będą to po prostu zera.

ZERNŻNIĘTE OD PROF. B

piątek, 31 maja 2024 18:23

Tomczak omawia ten slajd:
UWAGA! KOD Z BŁĘDEM

Przykład: wytworzenie liczby przeciwnej do danej dużej liczby (N słów)

```
inverse:  push %ebp                          #
          movl %esp, %ebp
          movl 8(%ebp), %ebx                 # adres liczby (tablicy)
          movl 12(%ebp), %ecx                # rozmiar liczby
          movl $-1, %esi                     # –X = not(X)+ulp
          stc                                # ustawienie CF=1      (clc)
pocz:     inc %esi                           #
          not (%ebx, %esi,4)
          adc $0, (%ebx, %esi,4)
          jnc koniec
          loop pocz                          # licznik pętli w %ecx
koniec:   movl %ebp, %esp
          pop %ebp
          ret
```

rozwiązanie alternatywne to (tylko fragment zacieniony)

```
          movl $-1, %esi                     # –X = 0–X
          clc                                # ustawienie CF=0
pocz:     inc %esi                           #
          movl $0, %eax
          sbbl (%ebx,%esi,4), %eax           # (0–(%ebx, %esi,4) do eax
          movl %eax, (%ebx,%esi,4)
          loop pocz                          # licznik pętli w %ecx
```


W tym kodzie wyznaczamy reprezentację dużej N-słowowej reprezentacji w U2 przy czym jedno słowo to jest jeden rejestr.

not (%ebx, %esi, 4) - negacja wszystkich bitów w komórce pamięci (brakuje przyrostka mówiącego ile tych bajtów chcemy zanegować)
adc $0, (%ebx, %esi, 4) - dodanie przeniesienia do zanegowanych bitów
jnc koniec - skok do zakończenia jeśli przeniesienie jest zerem.
Loop - rozkaz pętli, skoku

W pierwszym kroku dodajemy 1, a w następnych obiegach pętli dokonujemy propagacji przeniesienia.

SZCZEGÓŁY PRZEANALIZOWAĆ NA SPOKOJNIE.
NAJLEPIEJ TEN KOD ZASEMBLOWAĆ I OBEJRZEĆ W DEBUGERZE.
UWAGA! KOD Z BŁĘDEM

Przykład: dekrementacja dużej liczby (N×32 b)

(etykieta)	Rozkaz	Komentarz
ldecr:	push %ebp	# przez dodanie (1) (czyli –1)
	movl %esp, %ebp	
	movl 8(%ebp), %ebx	# adres zmiennej
	movl 12(%ebp), %ecx	# rozmiar zmiennej
	clc	# ustawienie CF=0
pocz:	inc %esi	
	adcl $–1, (%ebx,%esi,4)	
	jnc end	
end:	movl %ebp, %esp	
	pop %ebp	
	ret	

rozwiązanie alternatywne to (tylko fragment zacieniony):

	stc	# ustawienie CF=1
pocz:	inc %esi	#
	movl $0, %eax	
	sbbl (%ebx,%esi,4), %eax	# (adc $-1, (%ebx, %esi,4)
	movl %eax, (%ebx,%esi,4)	
	jnc end	

W TYM KODZIE JEST BŁĄD, BRAKUJE FUNKCJI LOOP po skoku: jnc end !
TOMCZAK WSPOMINA, ŻE "DOBRA LITERATURA DYDAKTYCZNA MA "subtelne" BŁĘDY W PRZYKŁADACH KODU"
OBA TE KODY MAJĄ BŁĘDY!

INC & LEA

piątek, 31 maja 2024 18:59

inc (ang. *increment*) – zwiększ (indeks) o 1
dec (ang. *decrement*) – zmniejsz (indeks) o 1
! ustawiane kody w rejestrze flag (F): OF, ZF, SF, bez zmiany CF

incl %eax **- zwiększ o 1 zawartość rejestru eax, nie zmienia CF**
add $1, %eax **- zwiększ o 1 zawartość rejestru eax, zmienia CF**
sub $-1, %eax **- zmniejsz o 1 zawartość rejestru eax, zmienia CF**

decb %bl **- zmniejsz o 1 zawartość rejestru bl, nie zmienia CF**
add $-1, %eax **- zmniejsz o 1 zawartość rejestru eax, zmienia CF**
sub $1, %eax **- zmniejsz o 1 zawartość rejestru eax, zmienia CF**

lea (ang. *load effective address*) – wpisz obliczony adres do wskazanego rejestru

lea (%esi,%eax,4), %esi - zwiększ wartość rejestru esi o 4*(eax), nie zmienia F
lea (%edi,%ecx,1) - edi:= (edi) + (ecx), indeksacja edi skokiem ecx
lea skok(%edi) - edi:= (edi) +skok, indeksacja edi skokiem ecx

Rozkaz inkrementacji INC zwiększa wartość w rejestrze o 1 i nie modyfikuje flagi carry, jest to ważne w przypadku pętli.

pocz:	inc %esi	#
	not (%ebx, %esi,4)	
	adc $0, (%ebx, %esi,4)	
	jnc koniec	
	loop pocz	# licznik pętli w %ecx
koniec:	movl %ebp, %esp	
	pop %ebp	
	ret	

W tym przypadku warunkiem jest ustawienie flagi carry. Jeśli chcielibyśmy zwiększyć licznik pętli między dodawaniem a skokiem wykonanie zwiększenia rozkazem dodawania zepsułoby flagę Carry. Z tego powodu INC wykorzystuje się często w pętlach.

Rozkaz LEA (Load Effective Address) obliczony adres efektywny pierwszy wpisuje do docelowego rejestru.

Przykłady:

leal 4(%ebx), %eax
leal (%ebx,%ecx,2), %eax
leal 8(%ebx), %eax

MUL & IMUL

sobota, 1 czerwca 2024 12:57

Na zdjęciu "muł"

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL *r/m8*	M	Valid	Valid	Unsigned multiply (AX ← AL * *r/m8*).
REX + F6 /4	MUL *r/m8**	M	Valid	N.E.	Unsigned multiply (AX ← AL * *r/m8*).
F7 /4	MUL *r/m16*	M	Valid	Valid	Unsigned multiply (DX:AX ← AX * *r/m16*).
F7 /4	MUL *r/m32*	M	Valid	Valid	Unsigned multiply (EDX:EAX ← EAX * *r/m32*).
REX.W + F7 /4	MUL *r/m64*	M	Valid	N.E.	Unsigned multiply (RDX:RAX ← RAX * *r/m64*).

Rozkaz służy do wykonywania mnożenia dla reprezentacji naturalnej binarnej. Pierwszym argumentem zawsze jest akumulator, a drugim wybrany operand z rejestru lub pamięci. W zależności od długości wybranego operandu wyliczona wartość trafi do akumulatora lub rejestru rozszerzonego D:A gdzie bardziej znacząca część trafia do rejestru D.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL *r/m8**	M	Valid	Valid	AX← AL * *r/m* byte.
F7 /5	IMUL *r/m16*	M	Valid	Valid	DX:AX ← AX * *r/m* word.
F7 /5	IMUL *r/m32*	M	Valid	Valid	EDX:EAX ← EAX * *r/m32*.
REX.W + F7 /5	IMUL *r/m64*	M	Valid	N.E.	RDX:RAX ← RAX * *r/m64*.
0F AF /*r*	IMUL *r16, r/m16*	RM	Valid	Valid	word register ← word register * *r/m16*.
0F AF /*r*	IMUL *r32, r/m32*	RM	Valid	Valid	doubleword register ← doubleword register * *r/m32*.
REX.W + 0F AF /*r*	IMUL *r64, r/m64*	RM	Valid	N.E.	Quadword register ← Quadword register * *r/m64*.
6B /*r ib*	IMUL *r16, r/m16, imm8*	RMI	Valid	Valid	word register ← *r/m16* * sign-extended immediate byte.
6B /*r ib*	IMUL *r32, r/m32, imm8*	RMI	Valid	Valid	doubleword register ← *r/m32* * sign-extended immediate byte.
REX.W + 6B /*r ib*	IMUL *r64, r/m64, imm8*	RMI	Valid	N.E.	Quadword register ← *r/m64* * sign-extended immediate byte.
69 /*r iw*	IMUL *r16, r/m16, imm16*	RMI	Valid	Valid	word register ← *r/m16* * immediate word.
69 /*r id*	IMUL *r32, r/m32, imm32*	RMI	Valid	Valid	doubleword register ← *r/m32* * immediate doubleword.
REX.W + 69 /*r id*	IMUL *r64, r/m64, imm32*	RMI	Valid	N.E.	Quadword register ← *r/m64* * immediate doubleword.

Rozkaz robi to samo, ale dla reprezentacji w U2
Dodatkowo potrafi obsługiwać wersję z dwoma argumentami oraz z trzema argumentami oraz trzema argumentami, z których jeden jest argumentem natychmiastowym.

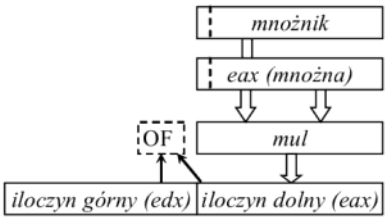
Ciekawostka:
Ponieważ wynik może być dwa razy szerszy od argumentów, ale nie musi flaga

carry i overflow zawiera informacje o tym czy wynik się mieści w mniej znaczącym akumulatorze czy nie.

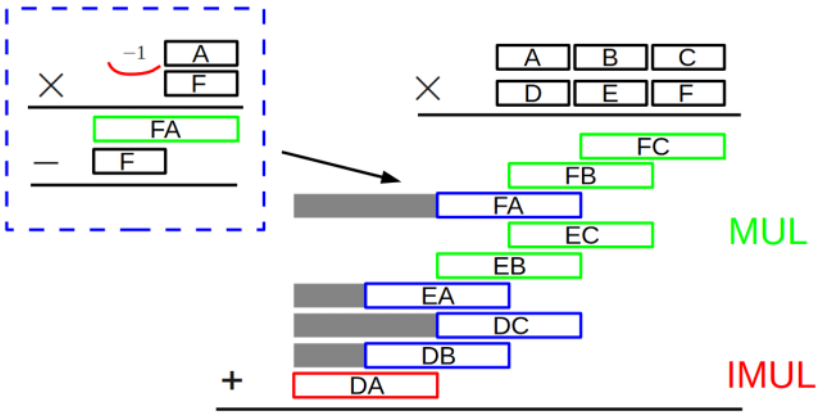
BARDZO WAŻNE!
Po wykonaniu tych rozkazów flagi SF, ZF, AF i PF są niezdefiniowane!!! Mogą być dowolne.

mul (ang. *multiply*) – wymnóż jak liczby naturalne
imul (ang. *integer multiply*) – wymnóż jak liczby całkowite (ze znakiem)
! ustawiane kody w rejestrze flag (F): CF, OF
iloczyn liczb 1-cyfrowych jest 2-cyfrowy

[i]mul arg ; **(edx:eax):=(eax)*arg**



[i]mull %ecx ; **(edx:eax):=(eax)*(ecx)**
imull %ecx, $const ; **(edx:eax):=(eax)*(ecx)**



W mnożeniu wielocyfrowych reprezentacji w systemie uzupełnieniowym pełnym iloczyny częściowe B,C,E,F są jak w kodzie naturalnym binarnym, mnożenie A z D jak w uzupełnieniowym pełnym, a reszta dziwnie bo jeden argument jest naturalny a drugi uzupełnieniowy pełny.

Trzeba stosować sztuczki, które pozwalają obejść problem braku odpowiedniego rozkazu.

DIV & IDIV

sobota, 1 czerwca 2024 13:31

DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV *r/m8*	M	Valid	Valid	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV *r/m8**	M	Valid	N.E.	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV *r/m16*	M	Valid	Valid	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV *r/m32*	M	Valid	Valid	Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV *r/m64*	M	Valid	N.E.	Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ←

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-24.

Table 3-24. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} - 1$

96

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV *r/m8*	M	Valid	Valid	Signed divide AX by *r/m8*, with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV *r/m8**	M	Valid	N.E.	Signed divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /7	IDIV *r/m16*	M	Valid	Valid	Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /7	IDIV *r/m32*	M	Valid	Valid	Signed divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /7	IDIV *r/m64*	M	Valid	N.E.	Signed divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

Table 3-59. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31}-1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{63}-1$

Z dzieleniem problemy są podobne jak z mnożeniem.
Też mamy osobne instrukcje dla NB i U2.
DIV i IDIV mają podobnie mało zestawów argumentów.
Dzielna i wynik są przechowywane w akumulatorze.
Dzielnik wybieramy jako argument własny.
NIE MA DZIELENIA PRZEZ ARGUMENT NATYCHMIASTOWY!

div (ang. *divide*) – podziel naturalne tworząc iloraz i resztę
idiv (ang. *integer divide*) – podziel całkowite tworząc iloraz i resztę

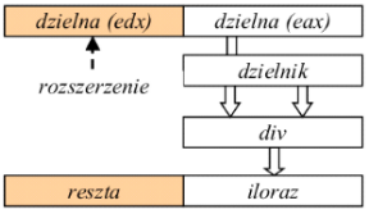
– jeśli iloraz zbyt duży do zapisu w rejestrze **a** – błąd *Divide Overflow*

[i]div arg ; (**eax**):=(**edx:eax**)/**arg** – iloraz, ; (**edx**):=(**edx:eax**) **mod arg** – reszta

Trzeba zwracać uwagę na to czy wynik będzie można zapisać w danej określonej części rejestru, nie możemy podzielić 1 000 000 000 przez 1, bo nie zapiszemy go do 32 bitowego rejestru. Wtedy mielibyśmy wyjątek: błąd dzielenia. Dlatego mamy podane maksymalne zakresy ilorazu jakie mogą powstać!:

Quotient Range
−128 to +127
−32,768 to +32,767
-2^{31} to $2^{31} - 1$
-2^{63} to $2^{63} - 1$

O wyjątkach później na wykładzie!
DZIELENIE SPRAWIA, ŻE CF, OF, SF, ZF, AF, i PF STAJĄ SIĘ NIEZDEFINIOWANE!



dzielenie krótkie: – konieczne rozszerzenie dzielnej

```
movl $0, %edx     ; (edx):= 0 (rozszerzenie zerowe)
div  %ecx         ; (eax):=(edx:eax) /(ecx)

cwde              ; rozszerzenie znakowe eax na edx
idiv %ecx         ; (eax):=(edx:eax) /(ecx)
```

MOŻLIWE, ŻE TUTAJ JEST BŁĄD!

Jeśli chcemy podzielić przez 32 bitową reprezentację nasza dzielna musi być reprezentacją 64-bitową, musimy więc rozszerzyć akumulator o 32 bity. W powyższym przykładzie robimy to dla wersji bez znaku wypełniając edx zerami oraz w wersji ze znakiem za pomocą instrukcji cwde (Convert word to doubleword extended).

Ciekawostka:
Rozkaz dzielenia jest tak skonstruowany, żeby kolejne rozkazy dzielenia mogły występować bezpośrednio po sobie.

Instrukcje pomocnicze

sobota, 1 czerwca 2024 14:22

Używane do pomiarów czasu, o tym później

cpuid (ang. ***cpu id****entification*) – identyfikacja procesora i specyfikacja wersji

rdtsc (ang. ***r****ead* ***t****ime* ***s****tamp* ***c****ounter*) – odczyt licznika cykli procesora

Rozkazy do zmiany flagi carry (dla pozostałych flag wyglądają podobnie):

stc (ang. ***s****e****t*** ***c****arry*) – ustawienie CF=1

clc (ang. ***c****l****ear*** ***c****arry*) – ustawienie CF=0

Rozkaz zmiany kolejności bajtów w słowie:

bswap (ang. ***b****yte* ***swap***) – odwrócenie kolejności bajtów w słowie

Rozkaz rozszerzenia znakowego:

cwde / cdq – rozszerzenie znakowe akumulatora **a** (eax na edx:eax/rax na rdx:rax)

Rozkazy rotacji:

rotacje (przesunięcia cykliczne) i przesunięcia

rol (ang. *rotate left through* ***c****arry*) – przesunięcie cykliczne bitów w lewo

ror (ang. *rotate right through* ***c****arry*) – przesunięcie cykliczne bitów w prawo

rlc (ang. *rotate left through* ***c****arry*) – przesunięcie bitów w lewo z dołączonym CF

rrc (ang. *rotate right through* ***c****arry*) – przesunięcie bitów w prawo z dołączonym CF

shl (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w lewo

shr (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w prawo

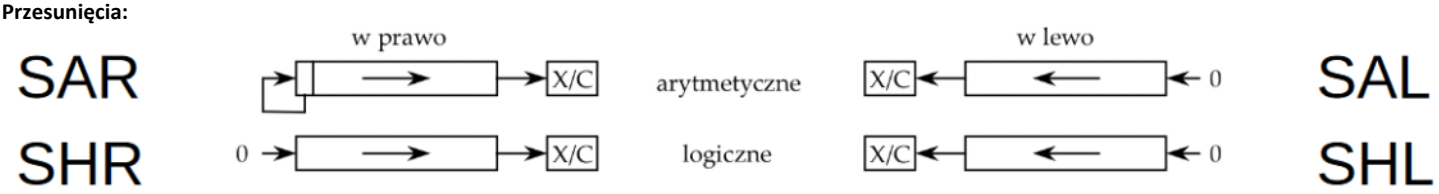
sar (ang. *rotate left through* ***c****arry*) –przesunięcie bitów w prawo z powieleniem

*Tomczak wspomina jeszcze o rozkazach w BCD, których jednak nie używa, bo nie są mu potrzebne

Przesunięcia i Rotacje

sobota, 1 czerwca 2024 14:30

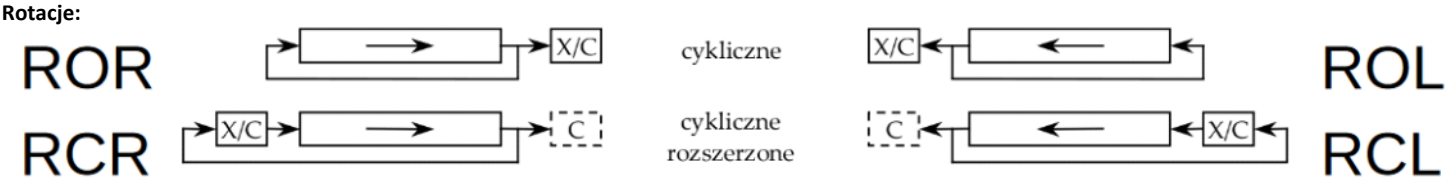
Przesunięcia:



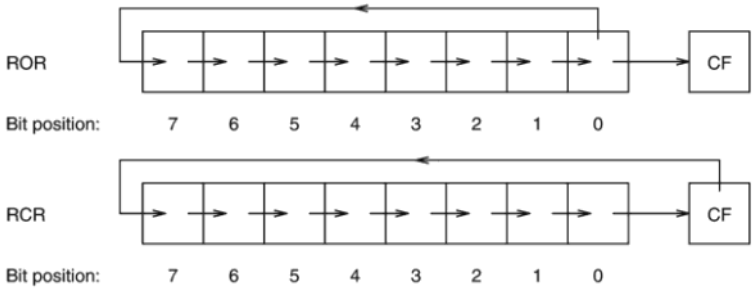
Z literą A przesuwa kopiując najbardziej znaczącą cyfrę z lewej,
Z literą H wypełnia zerami.

WSZYSTKIE ROZKAZY TO CO ZOSTAJE WYSUNIĘTE TRAFIA DO FLAGI CARRY.
PRZESUWAJĄC O JEDEN BIT WYDOBYWAMY POJEDYŃCZE BITY Z FLAGI CARRY.

Rotacje:



W rotacjach wszystko co się wysuwa wsuwa się z drugiej strony
Są rotacje, które to co się wysuwa to się wsuwają i przy okazji dodają flagi carry.
RCR działa tak, że najpierw się wsuwa do flagi carry, a dopiero potem do argumentu.
Dobrze obrazuje ten obrazek, który znalazłem:

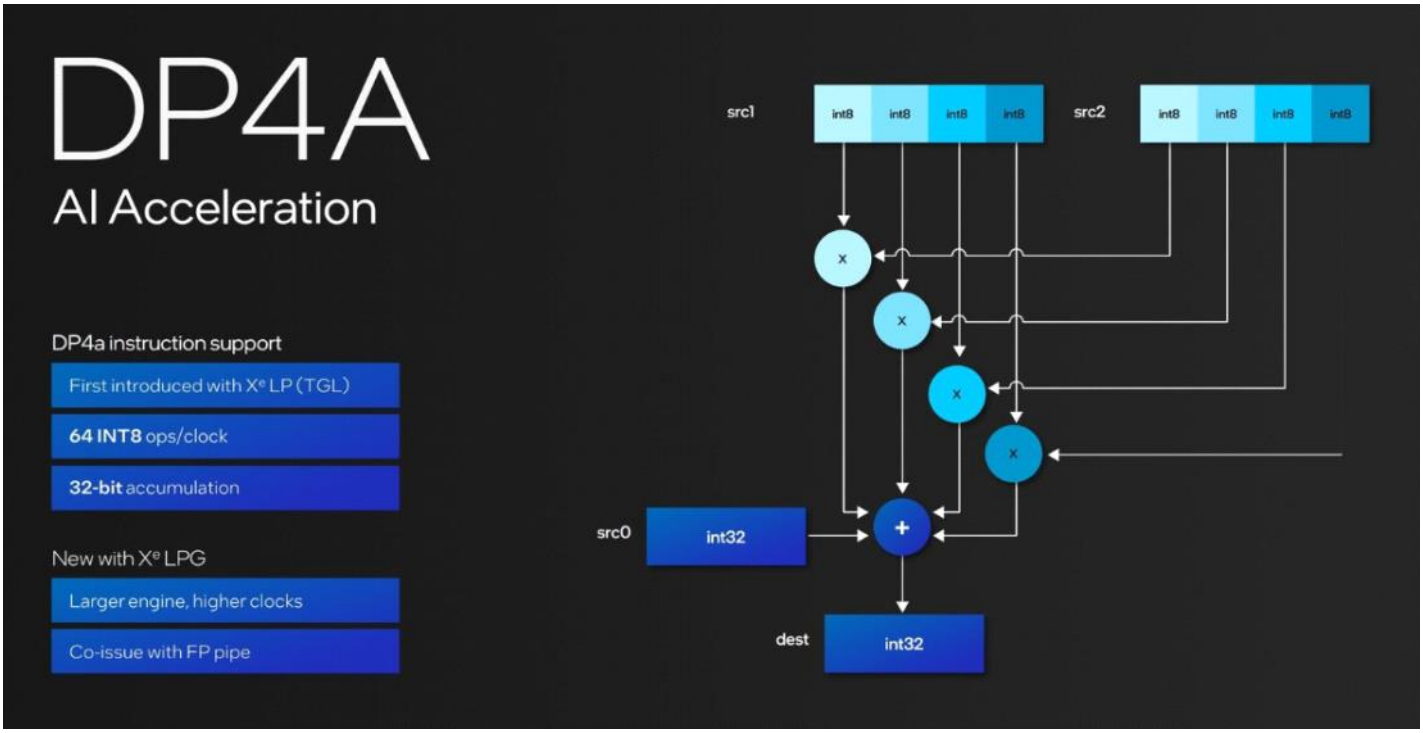


DP4A

sobota, 1 czerwca 2024 20:43

DP4A (Dot Product of 4-element vectors with Accumulation) - instrukcja w procesorach graficznych intela, ma 32-bitowe argumenty, dzieli je na kawałki 8-bitowe, traktuje je całkiem niezależnie, wykonuje naraz 4 mnożenia 9-bitowe, pierwszy z pierwszego z pierwszym z drugiego, drugi z pierwszego z drugim z drugiego itd. jak na obrazku. Wyniki są 16-bitowe, dodawane wszystkie do słowa 32 bitowego i wynik dodawania tych 5 operandów jest zapisywany do argumentu docelowego 32-bitowego. "W najnowszych standardach pojawia się chyba wersja z 128 argumentami. Tamto słowo jest wyraźnie większe".

CHATGPT:
Jest używana głównie w kontekście obliczeń związanych z głębokim uczeniem (deep learning) i sztuczną inteligencją (AI), a także w innych zastosowaniach, które wymagają intensywnych obliczeń wektorowych.



NA STOS!

sobota, 1 czerwca 2024 20:52

Ten procesor i wiele innych obsługują sprzętowo strukturę stosu.

I. Stos - fragment pamięci, do którego nie odwołujemy się w dowolnym miejscu tylko za pomocą "organizacji stosowej". Dostęp naturalny jest tylko do elementów na szczycie stosu.

Na szczyt stosu można dodawać elementy lub z niego zdejmować.
Co nas interesuje najbardziej:
Instrukcje:
- PUSH
- POP
Rejestr:
ESP - rejestr wskaźnika stosu, wskazuje na szczyt stosu.

6.2. STACK

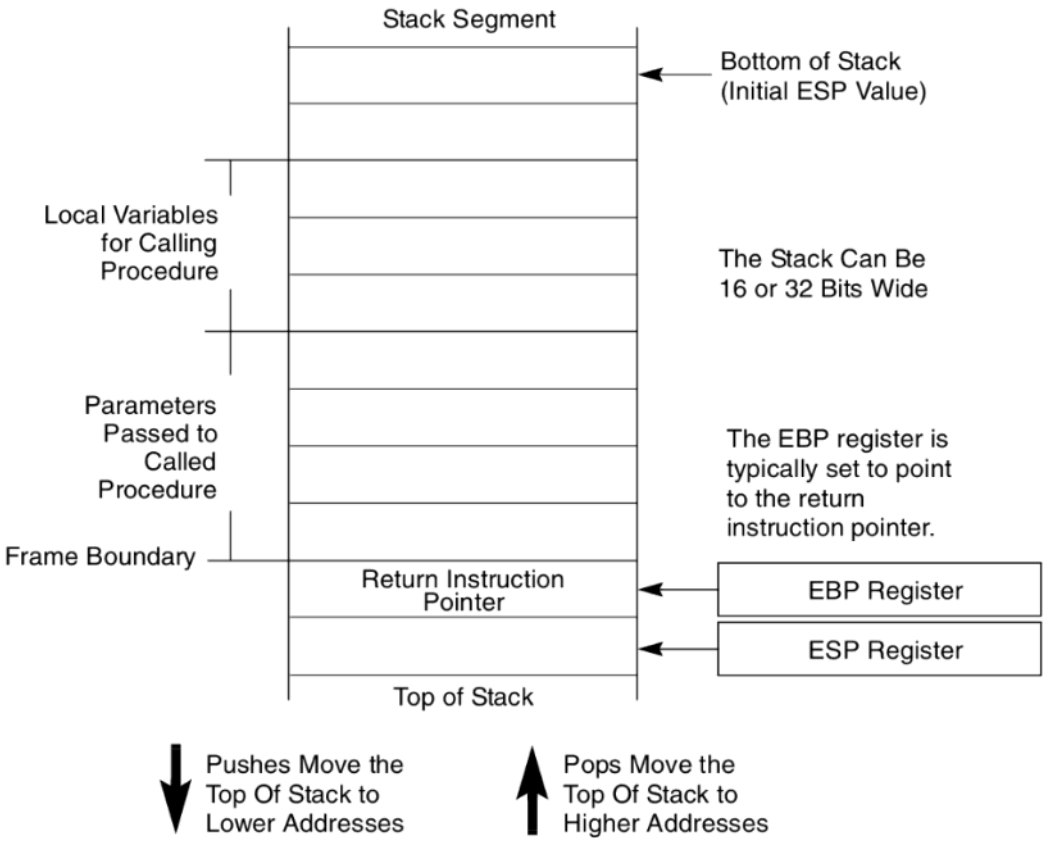
The stack (see Figure 6-1) is a contiguous array of memory locations. It is contained in a segment and identified by the segment selector in the SS register. (When using the flat memory model, the stack can be located anywhere in the linear address space for the program.) A stack can be up to 4 gigabytes long, the maximum size of a segment.

Items are placed on the stack using the PUSH instruction and removed from the stack using the POP instruction. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new top of stack. When an item is popped off the stack, the processor reads the item from the top of stack, then increments the ESP register. In this manner, the stack grows **down** in memory (towards lesser addresses) when items are pushed on the stack and shrinks **up** (towards greater addresses) when the items are popped from the stack.

A program or operating system/executive can set up many stacks. For example, in multitasking systems, each task can be given its own stack. The number of stacks in a system is limited by the maximum number of segments and the available physical memory.

When a system sets up many stacks, only one stack—the **current stack**—is available at a time. The current stack is the one contained in the segment referenced by the SS register.

Obrazek przedstawia strukturę stosu:



Co jest ważne:
Rejestr wskaźnika stosu.
Stos rośnie w stronę mniejszych adresów!
Kiedy stos jest pusty jesteśmy pod najwyższym adresem i w miarę dokładania elementów na stos ----> adres szczytu stosu się zmniejsza! Tak zrobili w intelu.

ROZKAZ PUSH:

PUSH—Push Word, Doubleword, or Quadword Onto the Stack

Opcode[1]	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH r/m16	M	Valid	Valid	Push r/m16.
FF /6	PUSH r/m32	M	N.E.	Valid	Push r/m32.
FF /6	PUSH r/m64	M	Valid	N.E.	Push r/m64.
50+rw	PUSH r16	O	Valid	Valid	Push r16.
50+rd	PUSH r32	O	N.E.	Valid	Push r32.
50+rd	PUSH r64	O	Valid	N.E.	Push r64.
6A ib	PUSH imm8	I	Valid	Valid	Push imm8.
68 iw	PUSH imm16	I	Valid	Valid	Push imm16.
68 id	PUSH imm32	I	Valid	Valid	Push imm32.
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Intel Core and Intel Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

106

Co można odłożyć na szczyt stosu:

- 16 bitowy argument
- 32 bitowy argument
- 64 bitowy argument

32 bitowy argument można odłożyć na stos tylko w trybie 32-bitowym, w kodzie 64-bitowym nie można odłożyć na stos 32-bitowego argumentu. Natomiast w trybie 32-bitowym nie można odłożyć na stos 64-bitowego argumentu!

To /6 w opcodzie oznacza, że może być tam coś jeszcze, jest tam zapisana wielkość rejestru i argumentu, który chcemy odłożyć na stos. I możemy to rozróżnić na jeden z dwóch 16 albo 32/64.

Na stos można odłożyć:

- komórkę pamięci! Jest to jeden z niewielu rozkazów, który pozwala na przesyłanie danych między pamięcią a pamięcią, bo stos jest w pamięci.
- Zawartość rejestru
- Argument natychmiastowy

BARDZO WAŻNE!
Jak działa rozkaz odkładania na stos?
1.Zmniejsza wskaźnik stosu
2.Odkłada argument źródłowy na szczyt stosu

WSKAŹNIK STOSU WSKAZUJE ZAWSZE NA ELEMENT NA SZCZYCIE STOSU!
"Adresy ze wskaźnika stosu są związane z tzw. segmentem stosu,

Segment stosu jest przesunięty względem segmentu kodu czy segmentu danych, gdybyśmy tego adresu użyli tak ogólnie jako adresu to nie koniecznie wskazuje szczyt stosu. Często jest jednak tak ,że położenie segmentu stosu jest w tym samym miejscu co wszystkich innych segmentów. Wtedy wskaźnik stosu można traktować bezpośrednio jako wskaźnik dowolnego miejsca w pamięci w dowolnym segmencie. No trzeba uważać."

"Oni jednak nic z tego nie zrozumieli. Rzecz ta była zakryta przed nimi i nie pojmowali tego, o czym była mowa"

Łk, 18, 34

ROZKAZ POP:

POP—Pop a Value From the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/m16	M	Valid	Valid	Pop top of stack into m16; increment stack pointer.
8F /0	POP r/m32	M	N.E.	Valid	Pop top of stack into m32; increment stack pointer.
8F /0	POP r/m64	M	Valid	N.E.	Pop top of stack into m64; increment stack pointer. Cannot encode 32-bit operand size.
58+ rw	POP r16	O	Valid	Valid	Pop top of stack into r16; increment stack pointer.
58+ rd	POP r32	O	N.E.	Valid	Pop top of stack into r32; increment stack pointer.
58+ rd	POP r64	O	Valid	N.E.	Pop top of stack into r64; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when writing to a destination operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).
- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

107

BARDZO WAŻNE!

Jak działa rozkaz zdejmowania ze stosu?(odwrotnie do odkładania):

1.Odczytuje argument

2.Zwiększa adres wskaźnika stosu

Argument instrukcji nie może być tutaj natychmiastowy, ponieważ argumentem jest miejsce docelowe dla odczytanego wcześniej argumentu.

TO CO JEST NA STOSIE WYNIKA TYLKO ZE WSKAŹNIKA STOSU! UŻYWAJĄC POPA NIE USUWAMY WARTOŚCI, SPRAWIAMY TYLKO, ŻE WSKAŹNIK JEST POWYŻEJ NIEJ!

PO CH*J MI TO?

sobota, 1 czerwca 2024 21:27

Instrukcje mało użyteczne

Arytmetyka dziesiętna

aaa (ang. *ascii adjust after addition*) –
aad (ang. *adjust before division*) –
aam (ang. *adjust after multiplication*) –
aas (ang. *ascii adjust after subtraction*) –
daa (ang. *decimal adjust for addition*) –
das (ang. *decimal adjust for subtraction*) –

xlat (ang. …) – tablicowa translacja kodu
neg (ang. *negate*) – wytworzenie liczby przeciwnej
cmc (ang. *complement carry*) –
cmpxchg (ang. *compare and exchange*) –
xadd (ang. *exchange and add*) –
lods (ang. *load string*) –
stos (ang. *store string*) –
movs (ang. *move string*) –

TEGO SIĘ W NORMALNYCH PROGRAMACH NIE UŻYWA! TOMCZAK ODSYŁA DO KSIĄŻEK PROF. BOLESŁAWA POCHOPIENIA! TAM JEST CAŁE MNÓSTWO PRZYKŁADÓW UŻYCIA TEGO.

SKOKI

sobota, 1 czerwca 2024 21:43

DWA SKRAJNEPRZYPADKI:

a) Instrukcja wykonują się całkowicie po kolei sekwencyjnie.
   +szybkie odczytywanie kolejnych instrukcji
   -Mała elastyczność, nie da się kodu spakować do znacznie mniejszego rozmiaru (jeśli chcielibyśmy przetworzyć miliard elementów potrzebujemy miliard operacji)
b) Razem z kodem każdej instrukcji zapisujemy gdzie jest adres następnej instrukcji
   + dowolność całkowita
   - duże kody instrukcji
   - kosztowne czasowo (nie wiemy jaka będzie następna instrukcja bez odczytania poprzedniej)

KOMPROMIS: SKOKI - ZMIENIAJĄ KOLEJNOŚĆ WYKONYWANIA INSTRUKCJI WARUNKOWO LUB BEZWARUNKOWO

Warunki mogą być proste lub złożone.
FUNKCJE - Skoki z powrotem, pamiętają skąd skok został wykonany. Tego rodzaju skoki nazywamy "skokami ze śladem".
W intelu nazywamy je CALL.
W motoroli BRANCH.

Często warunek określa się przeciwny do tego, ,który określa nam wykonanie fragmentu kodu i na tym przeciwnym warunku omijamy fragment kodu, natomiast jeśli nie jest spełniony to rozkaz skoku nie skacze i następuje fragment kodu, który chcemy wykonać dla warunku właściwego.

Przykład:

```
cmp $1, $eax
jne end;
//kod, który ma się wykonać jeśli rejestr a zawiera wartość 1
end:
```

UWAGA! SKOKI SĄ KOSZTOWNE Z POWODU CZASU ICH WYKONANIA! JEŻELI CHCEMY NAPISAĆ KOD SZYBKI MOŻE SIĘ OKAZAĆ, ŻE CHCEMY UNIKNĄĆ SKOKU..(część niezrozumiała).

Alternatywa: wytwarzamy dwa wyniki. Mnożymy jeden przez zero, a drugi przez jedynkę.
Jeżeli obliczenia są proste może to być bardzo efektywne.

Są też instrukcje potrafiące warunkowo wykonać operację:

Warunkowe kopiowanie danych:

CMOV*cc*—Conditional Move

Opcode	Instruction	Description
0F 47 /r	CMOVA *r16, r/m16*	Move if above (CF=0 and ZF=0)
0F 47 /r	CMOVA *r32, r/m32*	Move if above (CF=0 and ZF=0)
0F 43 /r	CMOVAE *r16, r/m16*	Move if above or equal (CF=0)
0F 43 /r	CMOVAE *r32, r/m32*	Move if above or equal (CF=0)
0F 42 /r	CMOVB *r16, r/m16*	Move if below (CF=1)
0F 42 /r	CMOVB *r32, r/m32*	Move if below (CF=1)
0F 46 /r	CMOVBE *r16, r/m16*	Move if below or equal (CF=1 or ZF=1)
0F 46 /r	CMOVBE *r32, r/m32*	Move if below or equal (CF=1 or ZF=1)
0F 42 /r	CMOVC *r16, r/m16*	Move if carry (CF=1)
0F 42 /r	CMOVC *r32, r/m32*	Move if carry (CF=1)
0F 44 /r	CMOVE *r16, r/m16*	Move if equal (ZF=1)
0F 44 /r	CMOVE *r32, r/m32*	Move if equal (ZF=1)
0F 4F /r	CMOVG *r16, r/m16*	Move if greater (ZF=0 and SF=OF)
0F 4F /r	CMOVG *r32, r/m32*	Move if greater (ZF=0 and SF=OF)
0F 4D /r	CMOVGE *r16, r/m16*	Move if greater or equal (SF=OF)
0F 4D /r	CMOVGE *r32, r/m32*	Move if greater or equal (SF=OF)
0F 4C /r	CMOVL *r16, r/m16*	Move if less (SF<>OF)
0F 4C /r	CMOVL *r32, r/m32*	Move if less (SF<>OF)
0F 4E /r	CMOVLE *r16, r/m16*	Move if less or equal (ZF=1 or SF<>OF)
0F 4E /r	CMOVLE *r32, r/m32*	Move if less or equal (ZF=1 or SF<>OF)
0F 46 /r	CMOVNA *r16, r/m16*	Move if not above (CF=1 or ZF=1)
0F 46 /r	CMOVNA *r32, r/m32*	Move if not above (CF=1 or ZF=1)
0F 42 /r	CMOVNAE *r16, r/m16*	Move if not above or equal (CF=1)
0F 42 /r	CMOVNAE *r32, r/m32*	Move if not above or equal (CF=1)
0F 43 /r	CMOVNB *r16, r/m16*	Move if not below (CF=0)
0F 43 /r	CMOVNB *r32, r/m32*	Move if not below (CF=0)
0F 47 /r	CMOVNBE *r16, r/m16*	Move if not below or equal (CF=0 and ZF=0)
0F 47 /r	CMOVNBE *r32, r/m32*	Move if not below or equal (CF=0 and ZF=0)
0F 43 /r	CMOVNC *r16, r/m16*	Move if not carry (CF=0)
0F 43 /r	CMOVNC *r32, r/m32*	Move if not carry (CF=0)
0F 45 /r	CMOVNE *r16, r/m16*	Move if not equal (ZF=0)
0F 45 /r	CMOVNE *r32, r/m32*	Move if not equal (ZF=0)
0F 4E /r	CMOVNG *r16, r/m16*	Move if not greater (ZF=1 or SF<>OF)
0F 4E /r	CMOVNG *r32, r/m32*	Move if not greater (ZF=1 or SF<>OF)
0F 4C /r	CMOVNGE *r16, r/m16*	Move if not greater or equal (SF<>OF)
0F 4C /r	CMOVNGE *r32, r/m32*	Move if not greater or equal (SF<>OF)
0F 4D /r	CMOVNL *r16, r/m16*	Move if not less (SF=OF)
0F 4D /r	CMOVNL *r32, r/m32*	Move if not less (SF=OF)
0F 4F /r	CMOVNLE *r16, r/m16*	Move if not less or equal (ZF=0 and SF=OF)
0F 4F /r	CMOVNLE *r32, r/m32*	Move if not less or equal (ZF=0 and SF=OF)

Warunkowa wymiana danych:

CMPXCHG—Compare and Exchange

Opcode	Instruction	Description
0F B0/*r*	CMPXCHG *r/m8,r8*	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
0F B1/*r*	CMPXCHG *r/m16,r16*	Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AX
0F B1/*r*	CMPXCHG *r/m32,r32*	Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into EAX

Description

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* accumulator = AL, AX, or EAX, depending on whether *)
(* a byte, word, or doubleword comparison is being performed*)
IF accumulator = DEST
    THEN
        ZF ← 1
        DEST ← SRC
    ELSE
        ZF ← 0
        accumulator ← DEST
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Wiemy jaka będzie następna instrukcja więc możemy ją na zapas zacząć wykonywać.

SKOKI:
Skok bezwarunkowy:

JMP—Jump

Opcode	Instruction	Description

JMP—Jump

Opcode	Instruction	Description
EB *cb*	JMP *rel8*	Jump short, relative, displacement relative to next instruction
E9 *cw*	JMP *rel16*	Jump near, relative, displacement relative to next instruction
E9 *cd*	JMP *rel32*	Jump near, relative, displacement relative to next instruction
FF /4	JMP *r/m16*	Jump near, absolute indirect, address given in *r/m16*
FF /4	JMP *r/m32*	Jump near, absolute indirect, address given in *r/m32*
EA *cd*	JMP *ptr16:16*	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	Jump far, absolute indirect, address given in *m16:32*

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

113

Skoki warunkowe (zależne od stanów flag):

Jcc—Jump if Condition Is Met

Opcode	Instruction	Description
77 *cb*	JA *rel8*	Jump short if above (CF=0 and ZF=0)
73 *cb*	JAE *rel8*	Jump short if above or equal (CF=0)
72 *cb*	JB *rel8*	Jump short if below (CF=1)
76 *cb*	JBE *rel8*	Jump short if below or equal (CF=1 or ZF=1)
72 *cb*	JC *rel8*	Jump short if carry (CF=1)
E3 *cb*	JCXZ *rel8*	Jump short if CX register is 0
E3 *cb*	JECXZ *rel8*	Jump short if ECX register is 0
74 *cb*	JE *rel8*	Jump short if equal (ZF=1)
7F *cb*	JG *rel8*	Jump short if greater (ZF=0 and SF=OF)
7D *cb*	JGE *rel8*	Jump short if greater or equal (SF=OF)
7C *cb*	JL *rel8*	Jump short if less (SF<>OF)
7E *cb*	JLE *rel8*	Jump short if less or equal (ZF=1 or SF<>OF)
76 *cb*	JNA *rel8*	Jump short if not above (CF=1 or ZF=1)
72 *cb*	JNAE *rel8*	Jump short if not above or equal (CF=1)
73 *cb*	JNB *rel8*	Jump short if not below (CF=0)
77 *cb*	JNBE *rel8*	Jump short if not below or equal (CF=0 and ZF=0)
73 *cb*	JNC *rel8*	Jump short if not carry (CF=0)
75 *cb*	JNE *rel8*	Jump short if not equal (ZF=0)
7E *cb*	JNG *rel8*	Jump short if not greater (ZF=1 or SF<>OF)
7C *cb*	JNGE *rel8*	Jump short if not greater or equal (SF<>OF)
7D *cb*	JNL *rel8*	Jump short if not less (SF=OF)
7F *cb*	JNLE *rel8*	Jump short if not less or equal (ZF=0 and SF=OF)
71 *cb*	JNO *rel8*	Jump short if not overflow (OF=0)
7B *cb*	JNP *rel8*	Jump short if not parity (PF=0)
79 *cb*	JNS *rel8*	Jump short if not sign (SF=0)
75 *cb*	JNZ *rel8*	Jump short if not zero (ZF=0)
70 *cb*	JO *rel8*	Jump short if overflow (OF=1)
7A *cb*	JP *rel8*	Jump short if parity (PF=1)
7A *cb*	JPE *rel8*	Jump short if parity even (PF=1)
7B *cb*	JPO *rel8*	Jump short if parity odd (PF=0)
78 *cb*	JS *rel8*	Jump short if sign (SF=1)
74 *cb*	JZ *rel8*	Jump short if zero (ZF = 1)
0F 87 *cw/cd*	JA *rel16/32*	Jump near if above (CF=0 and ZF=0)
0F 83 *cw/cd*	JAE *rel16/32*	Jump near if above or equal (CF=0)
0F 82 *cw/cd*	JB *rel16/32*	Jump near if below (CF=1)
0F 86 *cw/cd*	JBE *rel16/32*	Jump near if below or equal (CF=1 or ZF=1)
0F 82 *cw/cd*	JC *rel16/32*	Jump near if carry (CF=1)
0F 84 *cw/cd*	JE *rel16/32*	Jump near if equal (ZF=1)

I tutaj wchodzą różne wariacje warunków, za pomocą CMP możemy sprawdzać czy operand jest większy czy mniejszy od drugiego (CF) Jeśli od większej odejmiemy mniejszą flaga carry nie będzie ustawiona, a jeśli od mniejszej większą to flaga carry będzie ustawiona. Czasami dochodzi flaga zero. UWAGA! W U2 nie da się określić czy był nadmiar czy nie wyłącznie na podstawie flagi CARRY. W TYM KODZIE MUSIMY BADAĆ FLAGĘ OF! To jest jeszcze dodatkowo porównywalne z flagą znaku.

ZAWSZE DWÓCH ICH JEST

1. Rozkaz ustawiający flagi

2. Rozkaz skoku warunkowego na podstawie flag.

WAŻNE JEST, ŻEBY DOSŁOWNIE NIE BRAĆ NAZW INSTRUKCJI SKOKU, PATRZEĆ ZAWSZE NA WARUNEK.

Jak ustawić flagi?
a)Wykonać rozkaz odejmowania
lub
b)Wykonać rozkaz CMP

Jaka jest w takim razie różnica?
Rozkaz CMP wykonuje odejmowanie, ustawia flagi, ale wynik wyrzuca! NIE ZMIENIA ARGUMENTU DOCELOWEGO!

CO OBEJRZEĆ? PDF z KATALOGU PROF. BIERNATA O NAZWIE SKOKI I FUNKCJE, ZNAJDZIEMY TAM PORÓWNANIA WARUNKÓW JĘZYKA WYSOKIEGO POZIOMU I ASEMBLERA

DZIĘKUJĘ BARDZO!