

Predmet Asembler – organizácia

1. Motivácia, základné pojmy, prerekvizity, rodina mikroprocesorov Intel, Intel 8080.
2. Architektúra procesora Pentium, organizácia prístupu k pamäti, jazyk rodiny procesorov x86.
3. Tvorba programu v asembleri. Podprogramy a zásobník.
4. Adresovanie. Celočíselná aritmetika.
5. Skoky, iterácie. Implementácia riadiacich štruktúr jazykov vyššej úrovne.
6. Logické a bitové operácie. Polia, reťazce.
7. Prepojenie asembleru s jazykmi vyššej úrovne.
8. Prerušenia, služby operačného systému.
9. FPU a aritmetika v pohyblivej radovej čiarke.
+ prípadný pohľad na aktuálny vývoj v oblasti (podľa časových možností)

Organizácia cvičení

- podklady dostupné v rámci LMS Moodle

Bodové hodnotenie

Zadanie č.1 – max. 15b (i80, 5.týždeň)

Zadanie č.2 – max. 25b (i86, 8.týždeň)

Zápočet spolu – max. 40b

Skúška – max. 60b

Spolu – max. 100b

Termíny orientačné, možnosť priebežného upresnenia. Poznámky k štúdiu (prednášky, cvičenia, samoštúdium). Literatúra.

Motivácia pre štúdium v oblasti strojovo-orientovaných jazykov

- cieľ predmetu – priblížiť programovanie počítača na nižšej úrovni (ako pri použití vyšších (HL) jazykov)
- programátor – následne často produktívnejší aj pri práci vo vyšších jazykoch

Súčasná situácia

- nie je bežné vytvárať rozsiahle programy (pre PC) v samotnom asembleri, ten je možné využiť pri tvorbe kritických častí
- bežnejšie je využitie v prípade jednočipových systémov, vnorených systémov (zdroje, výkon)
- stále sa však nájde dostatok dôvodov pre učenie sa a používanie assembleru [5]

Prečo assembler?

- edukačné dôvody – cenné poznatky o činnosti počítačov, prekladačov, predikcia efektívnosti konštrukcií vyšších jazykov
- ladenie programov, dekompilácia
- tvorba prekladačov HL jazykov (napr. C), ladiacich nástrojov a pod.
- ovládače zariadení, vývoj systémového softvéru, prístup k hardvéru počítača
- efektívnosť (časová, priestorová) – program v asembleri často menší a/alebo rýchlejší ako kód generovaný kompilátorom
- inštrukcie nedostupné v rámci vyššieho jazyka

Prečo nie assembler?

- rýchlejší vývoj – výsledný kód zvyčajne kratší, prehľadnejší s využitím jazyka vyššej úrovne
- spoľahlivosť a bezpečnosť
- jednoduchšia údržba – oprava chýb, tvorba nových verzií
- prenositeľnosť – kód ľahšie prenositeľný na iné platformy
- vývoj v oblasti prekladačov v posledných rokoch, možnosti optimalizácie, často efektívny kód

Porovnanie výkonnosti HL jazyka a assembleru

- násobenie dvoch 16-bit celých čísel, zdrojové texty procedúr dostupné v [1]
- kompilátory prešli vývojom, výsledok môže byť aj lepší, ako kód vytvorený priemerným programátorom v asembleri

Základné pojmy

Strojový jazyk (strojový kód, machine language)

- každý procesor (typ) – vlastný strojový jazyk
- inštrukcie tohto jazyka – čísla (bajty v pamäti)
- každá inštrukcia – reprezentovaná jedinečným číselným kódom (operačný kód)
- dĺžka inštrukcií – môže byť premenlivá, začína operačným kódom, nasledujú prípadné argumenty
- programovanie značne komplikované

Asembler (jazyk, assembly language)

- program uložený ako text
- priama korešpondencia medzi inštrukciami assembleru a strojového jazyka

Príklad: `add eax, ebx` vs. `03 C3` (add – mnemonika inštrukcie sčítania)

Asembler (program)

- preklad z jazyka Asembler do strojového jazyka
- podobne kompilátor realizuje preklad z HL jazyka (assembler jednoduchší ako kompilátor HL jazyka)
- assembler pre každý konkrétny typ CPU (vlastný jazyk), ťažšie prenositeľný
- niektoré známe assembly: TASM, MASM, NASM

Prerekvizity, poznatky nadobudnuté v rámci doterajšieho štúdia na FEI TU

- v rámci predmetu Asembler predpokladáme zvládnutie vybraných okruhov z predmetov
 - Architektúry počítačových systémov (APS)
 - Princípy počítačového inžinierstva (PPI)
- odporúčame – zopakovať vybrané poznatky získané v rámci predmetov APS a PPI

Číselné sústavy, číselné kódy, aritmetika

- pozičné číselné sústavy, prevody medzi sústavami
- priamy, inverzný, doplnkový kód
- platné pravidlá, rozmer a povaha výsledku operácií
- aritmetika, ukladanie viac-bajtových údajov (malý/veľký endian)

Ukladanie viac-bajtových údajov (byte-ordering scheme [1])

- malý endian (little endian)
 - najnižší bajt (LSB) uložený najskôr (Intel Pentium)
- veľký endian (big endian)
 - najvyšší bajt (MSB) uložený najskôr (MIPS, Power PC)
- problém – prenos údajov medzi rôznymi systémami
- uloženie 4B celočíselnej hodnoty

Organizácia počítačov

- základné stavebné prvky počítačov
- hlavné časti počítačového systému a princípy ich činnosti
 - CPU
 - pamäť
 - vstup a výstup

(MSB)		32-bit data		(LSB)	
11110100	10011000	10110111	00001111		
Address	Content	Address	Content		
...		
103	11110100	103	00001111		
102	10011000	102	10110111		
101	10110111	101	10011000		
100	00001111	100	11110100		
...		
(a) Little-endian ordering		(b) Big-endian ordering			

Mikroprocesory

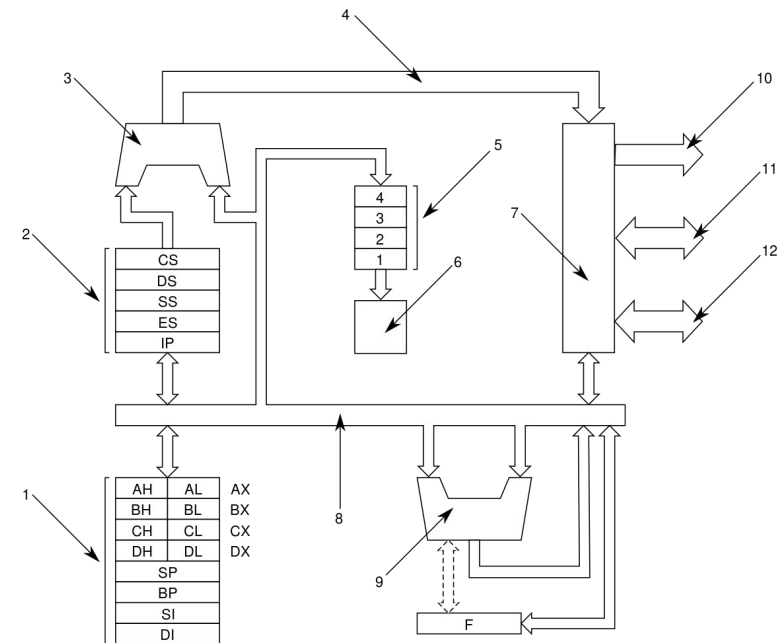
- procesor ako celok integrovaný do jediného integrovaného obvodu
- prvé dostupné mikroprocesory: [Intel 4004](#), Texas Instruments TMS 1000 (1971)
- významné 8-bitové mikroprocesory
 - [Intel 8008](#) (1972, prvý 8-bitový mikroprocesor), 8080 (1974, [MITS Altair 8800](#), [PMD 85](#))
 - [Zilog Z80](#) (1976, SEGA, Nintendo, [ZX Spectrum](#))
 - Motorola 6800 (1974, MITS Altair 680, Tektronix 4051)
 - MOS Technology [6502/6510](#) (1975, [Atari](#) 400/800, [Apple II](#), [Commodore](#), Nintendo NES)
- významné 16-bitové mikroprocesory
 - Intel 8086 (1978), 80286 (1982, IBM PC)
 - [Motorola 68000](#) (interne 32-bit, 1979, [Amiga](#), [Atari ST](#), Apple)
 - Zilog Z8000 (1979, Olivetti M20 - M60)

Rodina mikroprocesorov Intel

vybrané mikroprocesory 70-tych rokov

Processor	Clock Speed(s)	Intro Date(s)	Mfg. Process/ Transistors	Transistors	Addressable Memory	Cache	Bus Speed	Typical Use
4004	108 KHz	Nov-71	10-micron	2,300	640 Bytes	None	108 KHz	Busicom calculator, arithmetic manipulation
8008	200 KHz	Apr-72	10-micron	3,500	16 KB	None	200 KHz	Dumb terminals, general calculators, bottling machines, data/character manipulation
8080	2 MHz	Apr-74	6-micron	6,000	64 KB	None	2 MHz	Traffic light controller, Altair computer (first PC)
8086	10 MHz 8 MHz 4.77 MHz	Jun-78	3-micron	29,000	1 MB	None	10 MHz 8 MHz 4.77 MHz	Portable computing
8088	8 MHz 4.77 MHz	Jun-79	3-micron	29,000	64 kB (?)	None	8 MHz 4.77 MHz	Desktops (standard CPU for all IBM PCs and PC clones at the time)

- i8086 – významný zástupca rodiny z konca 70-tych rokov
 - zavedenie segmentácie pamäti
 - významne obohatený súbor inštrukcií
 - významná zmena architektúry (x86) [6], [7]:
 - 1 – GPR, 2 – SR a IP, 3 – adresová sčítačka,
 - 4 – int. adresná zbernica, 5 – front inštrukcií,
 - 6 – riadiaca jednotka, 7 – rozhranie zbernice,
 - 8 – int. údajová zbernica, 9 – ALU
 - 10/11/12 – ext. adresná/údajová/riadiaca zbernica



vybrané mikroprocesory 80-tych rokov

([Wikimedia](#), Harkonnen2)

Processor	Clock Speed(s)	Intro Date(s)	Mfg. Process/ Transistors	Transistors	Addressable Memory	Cache	Bus Speed	Typical Use
80286	12 MHz 10 MHz 6 MHz	Feb-82	1.5-micron	134,000	16 MB	None	12 MHz 10 MHz 6 MHz	Desktops (standard CPU for all IBM PCs clones at the time)
Intel386™ DX Processor	33 MHz 25 MHz 20 MHz 16 MHz	Oct-85	1.5 micron 1-micron	275,000	4 GB	None	33 MHz 25 MHz 20 MHz 16 MHz	Desktops
Intel486™ DX Processor	50 MHz 33 MHz 25 MHz	Apr-89	1-micron 0.8-micron	1.2 million	4 GB	8 kB	50 MHz 33 MHz 25 MHz	Desktops and servers

- i80286 prináša – ochrana pamäti (16-bit protected mode), kompatibilita s 8086 (real mode)
- i80386 prináša (významný pokrok) – 32 bitov (registre, zbernica)
 - ochrana pamäti (32-bit protected mode, segmenty môžu mať veľkosť až 4GB – flat memory model)
 - stránkovanie pamäti (virtuálna pamäť), kompatibilita (real mode)

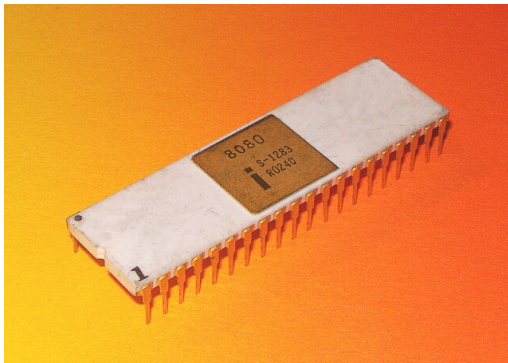
vybrané mikroprocesory 90-tych rokov

Processor	Clock Speed(s)	Intro Date(s)	Mfg. Process/ Transistors	Transistors	Addressable Memory	Cache	Bus Speed	Typical Use
Intel386™ SL Processor	25 MHz 20 MHz	Oct-90	1-micron	855,000	4 GB	None	25 MHz 20 MHz	First CPU designed specifically for portables
Intel486™ SX Processor	33 MHz 25 MHz 20 MHz 16 MHz	Sept-91	1 micron 0.8-micron	1.2 million 900,000	4 GB	8 kB	33 MHz 25 MHz 20 MHz 16 MHz	Low-cost, entry-level desktops
Intel486™ SL Processor	33 MHz 25 MHz 20 MHz	Nov-92	0.8-micron	1.4 million	4 MB	8 kB	33 MHz 25 MHz 20 MHz	First CPU specifically designed for Notebook PCs
Intel® Pentium® Processor	66 MHz 60 MHz	Mar-93	0.8 micron	3.1 million	4 GB	8 kB L1 Cache	66 MHz 60 MHz	Desktops
Intel® Pentium® Processor with MMX™ Technology	233 - 166 MHz	Oct-96	0.35 micron	4.5 million	4 GB	16 kB L1 Cache	66 MHz	High performance desktops and servers
Intel® Pentium® II Processor	300 - 233 MHz	May-97	0.35 micron	7.5 million	64 GB	512 kB L2 Cache	66 MHz	High-end business desktops, workstations and servers
Intel® Pentium® III Processor	600 - 450 MHz	Feb-99	0.25 micron	9.5 million	4 GB	512 kB L2 Cache	133 100 MHz	Business, consumer PCs; 1- and 2-way servers and workstations

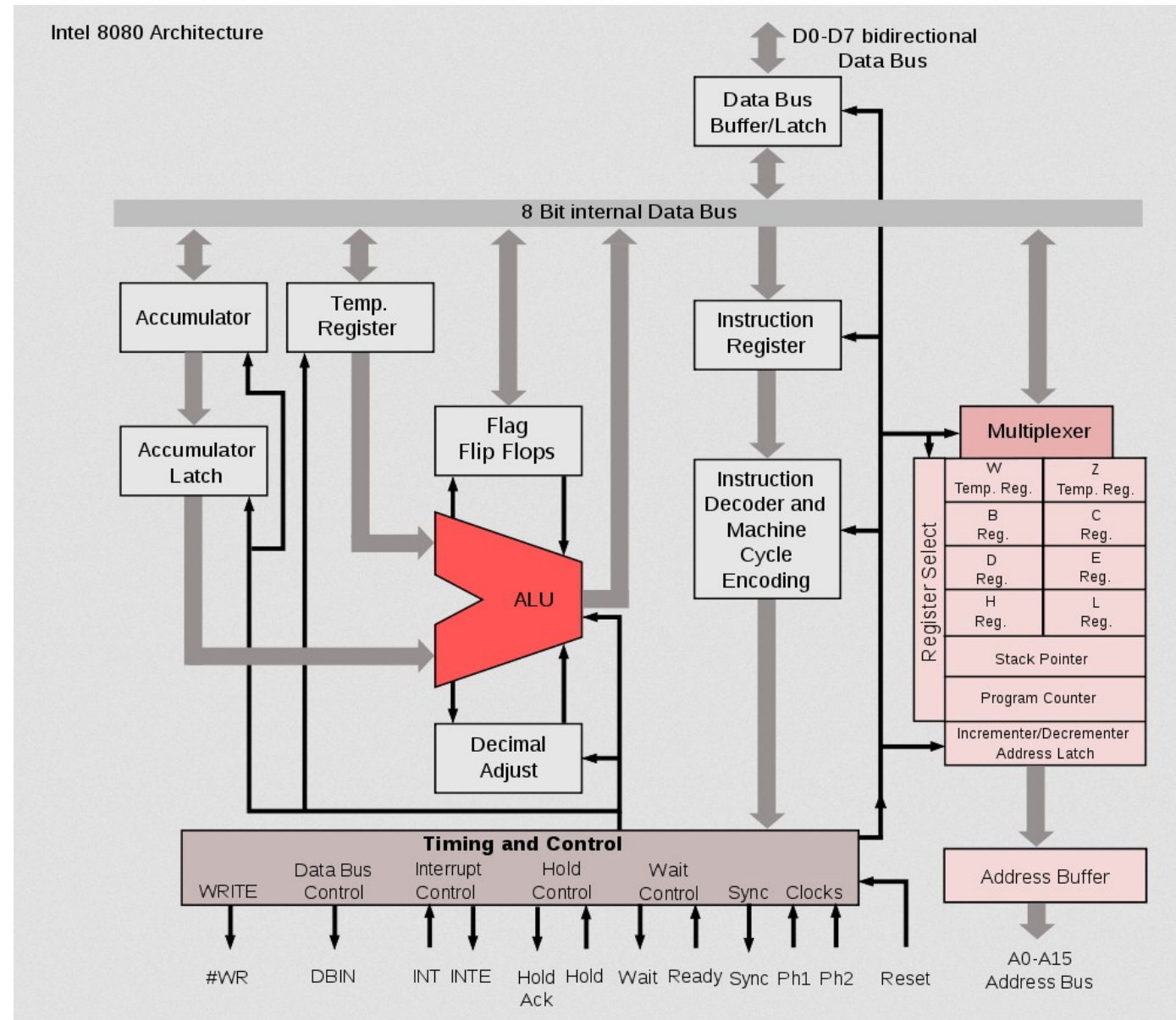
- ďalší vývoj zameraný hlavne na zvýšenie výkonu; zmeny v rámci architektúry ([x86-64](#))
- rozšírenie jazyka o nové inštrukcie (napr. [SIMD](#) rozšírenia – [MMX](#), [SSE](#), [AVX](#))
- spracované podľa [Microprocessor Quick Reference Guide](#)

Mikroprocesor Intel 8080

- významný predstaviteľ 8-bitových CPU
- detailnejší pohľad
- prvé osobné počítače (napr. MITS [Altair 8800](#), 1975)



([Wikimedia](#), Christian Bassow)



([Wikipedia](#), Appaloosa)

Štruktúra i8080

- registre (8-bit): B, C, D, E, H, L, pomocné W, Z (nedostupné programovo), A, F (ALU)
- registre (16-bit)
 - SP (adresa poslednej vloženej položky)
 - PC (adresa nasledujúcej inštrukcie)
 - registrové páry (BC, DE, HL, PSW [3])
- ALU – realizácia operácií aritmetických, logických a rotácií
 - A, F, pomocné registre
- riadiace obvody, register a dekódér inštrukcií – riadenie činnosti procesora
 - výber 1.bajtu inštrukcie (register inštrukcií – RI)
 - dekódovanie inštrukcie dekódérom inštrukcií (DI)
 - signály z dekódera spolu s časovacím signálmi – riadia činnosť ostatných blokov CPU
- zbernica (údajová – 8 bit., adresná – 16 bit.)

Príznaky (register F)

- 5 príznakov (S, Z, Ac, P, Cy), programovo dostupné 4 z nich
- možnosť zmeny vykonávania programu podľa výsledku predošlej operácie
 - podmienené skoky, volania podprogramov, návraty
- potrebná informácia o vplyve realizácie jednotlivých inštrukcií na príznaky
- Sign (S) – najvyšší bit výsledku, znamienko (1 – záporný výsledok)
- Zero (Z) – nulový výsledok operácie (1 – nula)
- Auxiliary Carry (Ac) – prenos z 3. bitu [3] akumulátora, desiatková korekcia (inštrukcia DAA [3])
- Parity (P) – počet jednotiek vo výsledku (1 – párný)
- Carry (Cy) – prenos z najvyššieho bitu výsledku operácie (1 – prenos nastal)

Kódovanie registrov a registrových párov i8080

Register (DDD/SSS)	B	C	D	E	H	L	M	A
Kód	000	001	010	011	100	101	110	111

M – pamäť adresovaná RP HL

Registrový pár (RP)	B	D	H	SP, PSW
Kód	00	01	10	11

Formát údajov a inštrukcií [4]

- údaje – 8-bitové binárne čísla

8-bit binary								
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	DATA WORD

- inštrukcie – dĺžka inštrukcie 1, 2 alebo 3 bajty, podľa typu

1-byte instructions								
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	OP CODE
2-byte instructions								
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	OP CODE
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	OPERAND
3-byte instructions								
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	OP CODE
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	Low address or OPERAND1
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	High address or OPERAND2

Jazyk procesora Intel 8080

Adresovanie operandov (metóda sprístupnenia operandov)

- *implicitné* (implied) – dané funkciou inštrukcie (STC, DAA)
- *registrové* (register) – operand v registri, druhý implicitný operand – akumulátor (CMP, ADD), RP ako operand (INX, DCX)
- *bezprostredné* (immediate) – údaj časťou inštrukcie (za operačným kódom, ADI, CPI, MVI, LXI)
- *priame* (direct) – priama adresa (16-bitov) súčasťou inštrukcie (JMP, LDA, STA)
- *nepriame* (register indirect) – pamäťová referencia v RP (LDAX, STAX)

Kódovanie a činnosť (vybraných) inštrukcií

- D8 – 8-bitový údaj, D16 – 16-bitový údaj (DL/DH), P8 – 8-bitové číslo portu
- ovplyvnené príznaky – S Z Ac P Cy („–“ – bez zmeny)
- registrové páry (RP – B, D, H, SP, PSW)

Inštrukcie presunov

Inštrukcia	1.B (Op.kód)	2.B	3.B	Operácia	Príznaky	RP	RTL
MOV R1, R2	01DDDSSS	–	–	move	– – – – –		R1←R2
MVI R, D8	00DDD110	D8	–	move immediate	– – – – –		R←D8
LXI RP, D16	00RP0001	DL	DH	load immediate RP	– – – – –	B, D, H	RP←D16
LDA D16	00111010	DL	DH	load A direct	– – – – –		A←[D16]
STA D16	00110010	DL	DH	store A direct	– – – – –		[D16]←A
LDAX RP	00RP1010	–	–	load A indirect	– – – – –	B, D	A←[RP]
STAX RP	00RP0010	–	–	store A indirect	– – – – –	B, D	[RP]←A
LHLD D16	00101010	DL	DH	load HL direct	– – – – –		HL←[D16]
SHLD D16	00100010	DL	DH	store HL direct	– – – – –		[D16]←HL
XCHG	11101011	–	–	exchange HL, DE	– – – – –		HL↔DE

Aritmetické operácie

Inštrukcia	1.B (Op.kód)	2.B	3.B	Operácia	Príznaky	RP	RTL
ADD R	10000SSS	-	-	add register to A	S Z AcP Cy		$A \leftarrow A+R$
ADC R	10001SSS	-	-	add reg. w.carry	S Z AcP Cy		$A \leftarrow A+R+Cy$
ADI D8	11000110	D8	-	add imm. to A	S Z AcP Cy		$A \leftarrow A+D8$
ACI D8	11001110	D8	-	add imm. to A w.carry	S Z AcP Cy		$A \leftarrow A+D8+Cy$
DAD RP	00RP1001	-	-	add RP to HL	- - - - Cy	B, D, H, SP	$HL \leftarrow HL+RP$
SUB R	10010SSS	-	-	subtract reg. from A	S Z AcP Cy		$A \leftarrow A-R$
SBB R	10011SSS	-	-	subtract reg. w bor.	S Z AcP Cy		$A \leftarrow A-R-Cy$
SUI D8	11010110	D8	-	subtract immed.	S Z AcP Cy		$A \leftarrow A-D8$
SBI D8	11011110	D8	-	sub.imm.w.borrow	S Z AcP Cy		$A \leftarrow A-D8-Cy$
INR R	00DDD100	-	-	increment reg.	S Z AcP -		$R \leftarrow R+1$
DCR R	00DDD101	-	-	decrement reg.	S Z AcP -		$R \leftarrow R-1$
INX RP	00RP0011	-	-	increment r.pair	- - - - -	B, D, H, SP	$RP \leftarrow RP+1$
DCX RP	00RP1011	-	-	decrement r.pair	- - - - -	B, D, H, SP	$RP \leftarrow RP-1$
DAA	00100111	-	-	decimal adjust accum.	S Z AcP Cy		

Poznámky k realizácii aritmetických operácií:

- aritmetické operácie i8080 predpokladajú doplnkovú reprezentáciu (pri tvorbe doplnku pretečenie ignorované)
- realizácia odčítania s využitím operácie sčítania – zjednodušenie obvodového riešenia
- odčítanie vs. pripočítanie čísla s opačným znamienkom
 - rovnaký výsledok, ale Cy-bit odlišný
 - pri operácii odčítania (SUB, SBB, CMP, ...) Cy-bit je invertovaný (signalizácia potreby pôžičky z vyššieho radu (borrow) pri odčítaní viacbajtových hodnôt)
- DAA – úprava 8-bitového hexadecimálneho čísla v registri A na dve 4-bitové BCD číslice (detaily v napr. [3])

Príklad:

a) 35-12

$$-12 = (1000\ 1100) = (1111\ 0011)_i = (1111\ 0100)_c$$

$$\begin{array}{r} \text{-----} \\ 1\ 0001\ 0111 \end{array} = (23)_{10}$$

b) 12-35

$$-35 = (1010\ 0011) = (1101\ 1100)_i = (1101\ 1101)_c$$

$$\begin{array}{r} \text{-----} \\ (1110\ 1001)_c \end{array} = (1001\ 0111) = (-23)_{10}$$

MVI A, 35 ADI -12	MVI A, 35 SUI 12	MVI A, 12 ADI -35	MVI A, 12 SUI 35
A = 23	A = 23	A = -23	A = -23
Cy = 1	Cy = 0	Cy = 0	Cy = 1

mnemonics opcode

mvi A, 23h	3E 23
adi F4h	C6 F4
sui 0Ch	D6 0C
nop	00
nop	00
nop	00
nop	00

Status

B 00 C 00 BC 0000

D 00 E 00 DE 0000

H 00 L 00 HL 0000

A 17 F 07 PC 0004

SP 0000

Flags (F):

S	Z	A	P	C
0	0	0	1	1

mnemonics opcode

mvi A, 23h	3E 23
adi F4h	C6 F4
mvi A, 23h	3E 23
sui 0Ch	D6 0C
nop	00
nop	00
nop	00
nop	00

Status

B 00 C 00 BC 0000

D 00 E 00 DE 0000

H 00 L 00 HL 0000

A 17 F 06 PC 0008

SP 0000

Flags (F):

S	Z	A	P	C
0	0	0	1	0

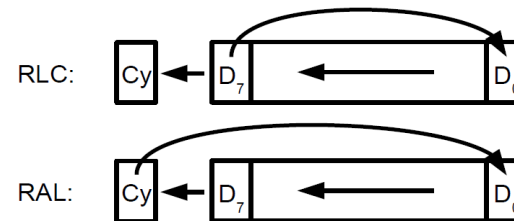
Logické operácie a rotácie

Inštrukcia	1.B (Op.kód)	2.B	3.B	Operácia	Príznačky	RP	RTL
ANA R	10100SSS	-	-	AND reg. with A	S Z 0 P 0		$A \leftarrow A \text{ and } R$
XRA R	10101SSS	-	-	XOR with A	S Z 0 P 0		$A \leftarrow A \text{ xor } R$
ORA R	10110SSS	-	-	OR reg. with A	S Z 0 P 0		$A \leftarrow A \text{ or } R$
CMP R	10111SSS	-	-	compare with A	S Z AcP Cy		$F \leftarrow A ? R$
ANI D8	11100110	D8	-	AND immediate w.A	S Z 0 P 0		$A \leftarrow A \text{ and } D8$
XRI D8	11101110	D8	-	XOR immediate w.A	S Z 0 P 0		$A \leftarrow A \text{ xor } D8$
ORI D8	11110110	D8	-	OR immediate w.A	S Z 0 P 0		$A \leftarrow A \text{ or } D8$
CPI D8	11111110	D8	-	cmp.immediate w.A	S Z AcP Cy		$F \leftarrow A ? D8$
RLC	00000111	-	-	rotate A left	- - - - Cy		
RRC	00001111	-	-	rotate A right	- - - - Cy		
RAL	00010111	-	-	r.A left t.carry	- - - - Cy		
RAR	00011111	-	-	r.A right t.carry	- - - - Cy		
CMA	00101111	-	-	complement A	- - - - -		$A \leftarrow \bar{A}$
STC	00110111	-	-	set carry	- - - - 1		$Cy \leftarrow 1$
CMC	00111111	-	-	complement carry	- - - - x		$Cy \leftarrow \bar{Cy}$

Príznačky (Z, Cy) ovplyvnené inštrukciami CMP, CPI:

$A = X: Z = 1$ $A < X: Cy = 1$
 $A \neq X: Z = 0$ $A \geq X: Cy = 0$

MVI A, 35 CPI -12 (F4h)	MVI A, 35 CPI 12	MVI A, 12 CPI -35 (DDh)	MVI A, 12 CPI 35
Cy = 1	Cy = 0	Cy = 1	Cy = 1

Rotácie akumulátora:RLC $D_i \rightarrow D_{i+1}, D_7 \rightarrow D_0, D_7 \rightarrow Cy$ RRC $D_i \rightarrow D_{i-1}, D_0 \rightarrow D_7, D_0 \rightarrow Cy$ RAL $D_i \rightarrow D_{i+1}, D_7 \rightarrow Cy, Cy \rightarrow D_0$ RAR $D_i \rightarrow D_{i-1}, D_0 \rightarrow Cy, Cy \rightarrow D_7$ **Skoky**

Inštrukcia	1.B (Op.kód)	2.B	3.B	Operácia	Príznaky	RP	RTL
JMP D16	11000011	DL	DH	jump unconditional	- - - - -		$PC \leftarrow D16$
JC D16	11011010	DL	DH	jump on carry	- - - - -		IF $Cy=1$ $PC \leftarrow D16$
JNC D16	11010010	DL	DH	jump on no carry	- - - - -		IF $Cy=0$ $PC \leftarrow D16$
JZ D16	11001010	DL	DH	jump on zero	- - - - -		IF $Z=1$ $PC \leftarrow D16$
JNZ D16	11000010	DL	DH	jump on no zero	- - - - -		IF $Z=0$ $PC \leftarrow D16$
JP D16	11110010	DL	DH	jump on positive	- - - - -		IF $S=0$ $PC \leftarrow D16$
JM D16	11111010	DL	DH	jump on minus	- - - - -		IF $S=1$ $PC \leftarrow D16$
JPE D16	11101010	DL	DH	jump on par.even	- - - - -		IF $P=1$ $PC \leftarrow D16$
JPO D16	11100010	DL	DH	jump on par.odd	- - - - -		IF $P=0$ $PC \leftarrow D16$
PCHL	11101001	-	-	HL to PC	- - - - -		$PC \leftarrow HL$

JMP D16

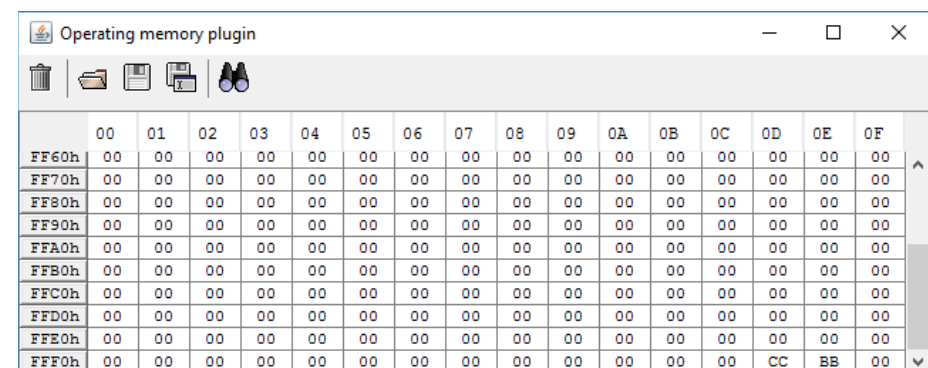
 $PC \leftarrow D16$

JC D16

IF $Cy = 1$ THEN $PC \leftarrow D16$

JNC D16

IF $Cy = 0$ THEN $PC \leftarrow D16$



kódovanie podmienok (skokov, volaní a návratu)

YYY	000	001	010	011	100	101	110	111
Jxx	JNZ	JZ	JNC	JC	JPO	JPE	JP	JM
Cxx	CNZ	CZ	CNC	CC	CPO	CPE	CP	CM
Rxx	RNZ	RZ	RNC	RC	RPO	RPE	RP	RM

Ostatné inštrukcie (I/O, riadiace)

Inštrukcia	1.B (Op.kód)	2.B	3.B	Operácia	Príznaky	RP	RTL
IN P8	11011011	P8	-	input	- - - - -		$A \leftarrow [P8]$
OUT P8	11010011	P8	-	output	- - - - -		$[P8] \leftarrow A$
EI	11111011	-	-	enable interrupt	- - - - -		
DI	11110011	-	-	disable interrupt	- - - - -		
NOP	00000000	-	-	no-operation	- - - - -		
HLT	01110110	-	-	halt	- - - - -		

Vstup a výstup:

- komunikácia s perifériami zabezpečená cez 256 I/O portov
- IN port – do akumulátora prenesený bajt zo špecifikovaného portu
- OUT port – obsah akumulátora prenesený na daný port
- IN, OUT – iba iniciujú prenos (zariadenie je povinné detegovať, že je adresované)

Prerušenia, zastavenie a obnovenie činnosti procesora:

- HLT – zastavenie činnosti procesora (z tohto stavu možno činnosť procesora obnoviť len externou udalosťou, napr. prerušenie)
- periférne zariadenie môže iniciovať prerušenie pomocou signálu INT (spracované, ak prerušenia sú povolené)
- pre obsluhu prerušení bežne využívaná inštrukcia RST [3]
- príslušnú inštrukciu poskytne zariadenie, ktoré iniciuje prerušenie (umiestnením na údajovú zbernicu) [4]

Príklad: násobenie 2 bezznamienkových 8-bitových čísel

- opakované sčítanie ($HL \leftarrow C * D$)

```

MVI B,0
LXI H,0
MOV A,D
CPI 0
JZ done
next: DAD B
DCR D
JNZ next
done: HLT

```

- násobenie s využitím posunov (rýchlejšie, posun vľavo/vpravo) [3]
 - D – násobenec, C – násobiteľ (na začiatku), rozmer výsledku: $2n$ radov (16 bitov, BC)
 - algoritmus:
 1. test najnižšieho bitu násobiteľa: 0 – pokračuj krokom 2, 1 – pripočítaj násobenec k vyššiemu bajtu výsledku (VBV)
 2. posuň celý (2B) výsledok o 1 bit vpravo
 3. opakuj kroky 1 a 2, pokiaľ neboli otestované všetky bity násobiteľa

```

MULT:  MVI B,0      ;inicializacia VBV
        MVI E,9      ;pocitadlo
MULT0:  MOV A,C
        RAR          ;rotacia nizsieho bajtu vysledku (NBV)
        MOV C,A      ;a posun LSb nasobitela do Cy
        DCR E
        JZ DONE      ;koniec
        MOV A,B
        JNC MULT1
        ADD D        ;nasobenec k VBV ak Cy bolo 1
MULT1:  RAR          ;posun VBV, v Cy hodnota pre posun do NBV
        MOV B,A
        JMP MULT0
DONE:   HLT

```

MULTPLICAND
00001010

MULTIPLIER
00000101

$$\begin{aligned}
 &0.0AH.2^7 + 0.0AH.2^6 + 0.0AH.2^5 + 0.0AH.2^4 + \\
 &0.0AH.2^3 + 1.0AH.2^2 + 0.0AH.2^1 + 1.0AH.2^0 = \\
 &00101000 + 00001010 = 00110010 = 50_{10}
 \end{aligned}$$

Platforma emuStudio – emulácia počítača s procesorom Intel 8080

The screenshot displays the emuStudio - Altair8800 application window, which is divided into several functional areas:

- Source code editor:** Contains assembly code for an Altair 8800 program. The code includes instructions like `org 1000`, `dcx sp`, `lxi h, text1`, `call putstr`, `mvi d, 0`, and a loop structure `char_loop` that checks for a character '0' and increments a counter 'd'.
- Debugger:** A table showing the execution flow with columns for breakpoint, address, mnemonics, and opcode. The current instruction being executed is `dcx SP` at address `03E8h`.
- Status:** Displays the current state of the processor registers (B, C, BC, D, E, DE, H, L, HL, A, F, PC, SP) and the flags (S, Z, A, P, C).
- Run control:** Includes a **breakpoint** section and a **Run control** section with fields for CPU frequency (set to 2000 kHz) and Test period (set to 50 ms). The runtime frequency is shown as 8,25 kHz.
- Peripheral devices:** A list of virtual hardware components including Terminal ADM-3A, MITS-88-SIO serial card, and MITS-88 DISK (floppy drive).
- Output window:** At the bottom, it shows the compilation status: "Intel 8080 Compiler, version 2.9b1. Compile was successful. Output: C:\Users\UVT-06\Desktop\emu8\examples\8080\pocet_cislic_getchar.hex. Compiled file was loaded into operating memory."

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] [Intel 8080 Assembly Language Programming Manual](#), Intel Corporation, 1975.
- [4] [Intel 8080 Microcomputer Systems User's Manual](#), Intel Corporation, 1975.
- [5] Fog, A.: [Optimizing subroutines in assembly language](#), Technical University of Denmark, 1996 – 2021.
- [6] Hudák, Š: Strojovo-orientované jazyky, FEI TU v Košiciach, 2003.
- [7] Wikipedia, [Intel 8086](#), 2023.

Architektúra procesora Pentium

- architektúra rodiny procesorov x86 (32-bit)
- snaha o zachovanie spätnej kompatibility novších procesorov
- organizácia pamäti, segmentácia

Registre

a) pre všeobecné použitie [1]

EAX (32-bit)					
			AX (16-bit)		
				AH (8-bit)	AL (8-bit)
31	16	15	8	7	0

- *hlavné údajové registre* (EAX, EBX, ECX, EDX)
- možnosť použitia ako 32/16/8-bitových registrov
- väčšina operácií presunov, aritmetických a logických operácií
- špeciálne určenie pri vykonávaní špecifických operácií (násobenie, iterácie)
- *ukazovatele* (pointer registers) – ESP, EBP (práca so zásobníkom)
- *indexové* (index registers) – ESI, EDI (aj ich 16-bitové subregistre – SI, DI; reťazcové operácie)

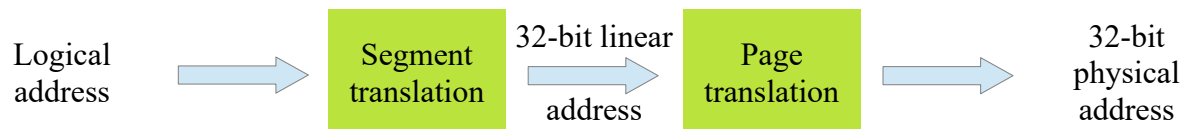
- šesť 16-bitových registrov, podpora segmentácie pamäti (umiestnenie segmentov)
- CS (code), DS (data), SS (stack), ES, FS, GS (pomocné)

Organizácia prístupu k pamäti v systémoch s procesorom Pentium

- chránený režim (protected mode) – natívny režim, podpora segmentácie i stránkovania (virtuálna pamäť, transparentná pre aplikácie)
- reálny režim (real mode) – spätná kompatibilita s programami pre i8086, segmentácia

Organizácia pamäti v chránenom režime

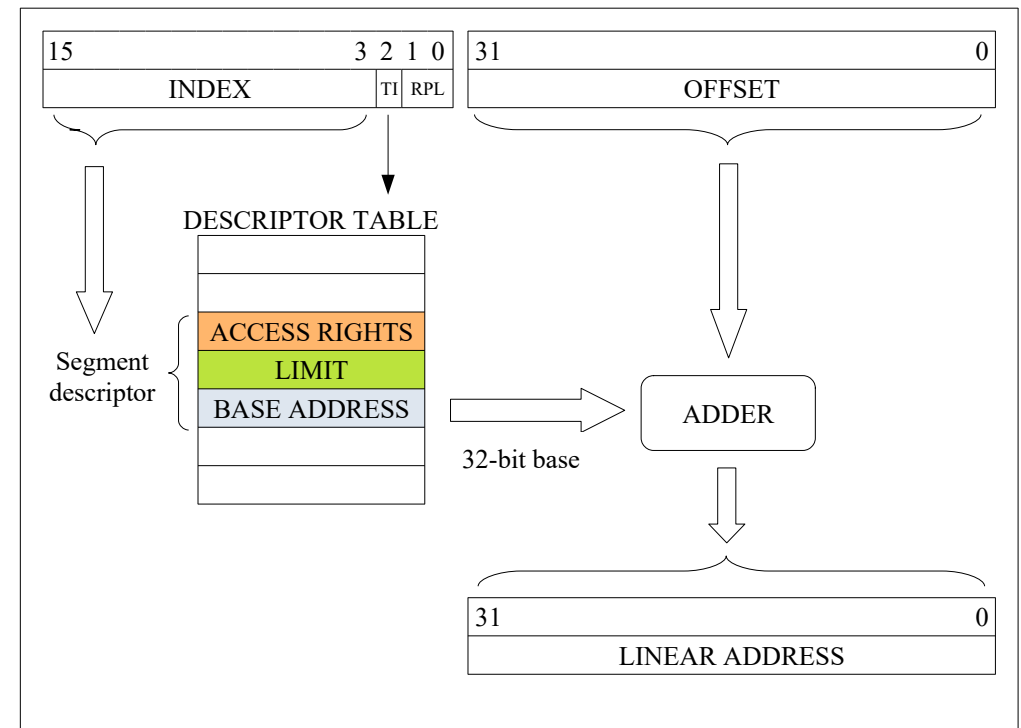
- segmentácia – prevod logickej adresy na 32-bit. lineárnu adresu (odlišná od reálneho (16-bit.) režimu)
- stránkovanie [8] – prevod lineárnej adresy na fyzickú (ak sa stránkovanie nevyužíva, lineárna adresa = fyzická) [1]



Segmentové registre (SR)

SR obsahuje segmentový selektor (segment selector) [1]

- index* (13-bit) do tabuľky deskriptorov (DT)
 - deskriptor segmentu (8-bajtov)
 - báza segmentu (32-bit), veľkosť, prístupové práva
 - adresa deskriptora: $\text{index} * 8 + \text{bázová adresa DT}$
 - výpočet lineárnej adresy
 - bázová adresa segmentu + posun (offset, 16/32-bit)
- TI* (table indicator, 1-bit)
 - indikuje použitie lokálnej(1)/globálnej(0) DT
- RPL* (requester privilege level, 2-bit)
 - úroveň oprávnenia prístupu k údajom
 - nižšia hodnota, vyššie oprávnenie (0 – jadro OS)



Segmentový deskriptor [1]

- poskytuje atribúty segmentu:
- bázoá adresa (32-bit) – začiatok segmentu vo fyzickom adresnom priestore
- veľkosť segmentu (20-bit), dve interpretácie
 - 1B – 1MB (2^{20} B), krok 1B ($G = 0$)
 - 4KB – 4GB, krok 4KB ($G = 1$)
- stavové a riadiace informácie
 - G (granularity) – interpretácia veľkosti segmentu (jednotka 1B/4KB)
 - D/B – dvojaká interpretácia, podľa typu segmentu
 - kódový (D) – (default operand size) predpokladaná veľkosť operandov a posunutia (offset): 0 – 16-bit, 1 – 32-bit.
 - dátový (B) – veľkosť zásobníka: $B = 0$ – použije sa SP register a max. veľkosť je FFFFH, ak $B = 1$ – ESP a FFFFFFFFH
 - AVL (available) – pre softvérové použitie, nevyužívané hardvérom
 - P (present) – (1) prítomnosť segmentu v pamäti, (0) – generovanie výnimky (prípadné zavedenie do pamäti) pri prístupe k seg.
 - DPL (descriptor privilege level) – úroveň oprávnenia potrebná pre prístup k segmentu, riadenie prístupu k segmentu
 - $DPL \geq RPL$ potrebné pre povolenie prístupu
 - (v skutočnosti sa overuje aj $DPL \geq CPL$, kde CPL (current privilege level) je získaný z reg. CS, viac informácií napr. v [5])
 - S – systémový segment (0) využívaný pre špeciálne účely, segment (kódový alebo dátový) aplikácie (1)
 - TYPE – typ segmentu (kódový/dátový, čítanie/zápis, ...)

[illegible]

Tabuľky deskriptorov

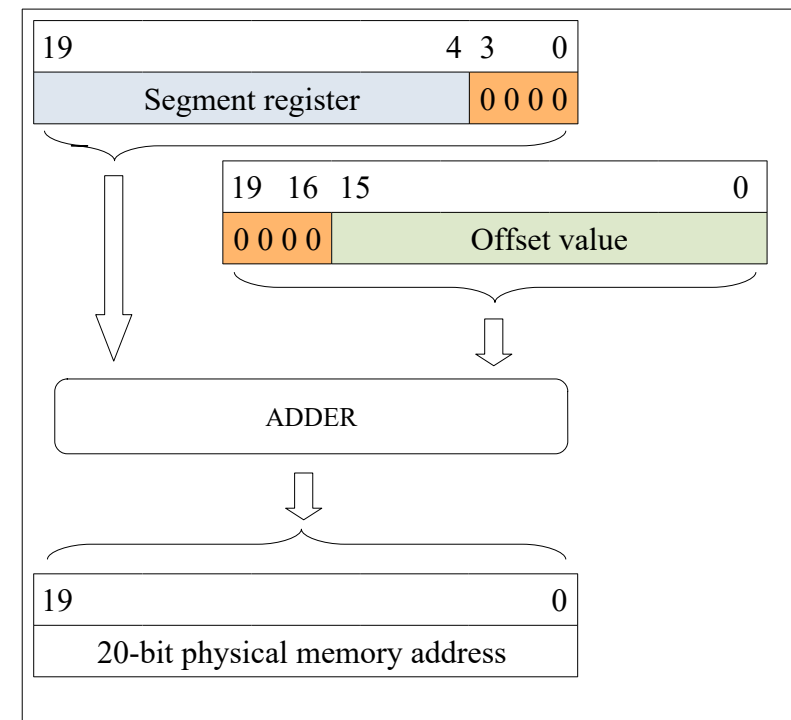
- pole segmentových deskriptorov, 3 typy tabuliek:
 - GDT (global descriptor table) – deskriptory dostupné všetkým úlohám, jediná v systéme, OS
 - LDT (local descriptor table) – deskriptory jednotlivých úloh (kód, dáta, zásobník)
 - IDT (interrupt descriptor table) – spracovanie prerušení
- tabuľky LDT a GDT (8B – 64KB, max. 8192 (2^{13}) 8B deskriptorov)
- s tabuľkou asociovaný register (32-bit bazová adresa, 16-bit veľkosť tabuľky)
 - LDTR (inštrukcie LLDT/SLDT používané OS), GDTR (inštrukcie LGDT/SGDT používané OS)

Modely segmentácie

- jednotný (flat model) – ukrytie mechanizmu segmentácie (bázy segmentov = 0, veľkosť 4GB)
- viac-segmentový (multisegment model) – súčasne aktívnych max. 6 segmentov (ich selektory v segmentových registroch)
 - generovanie výnimky (general protection) pri pokuse o prístup k pamäti nad rámec veľkosti segmentu (limit)

Organizácia pamäti v reálnom režime

- Pentium ako rýchly 8086
 - 1MB pamäti, registre veľkosti 16-bit v 8086 (dopad na adresovanie)
 - veľkosť segmentu – 64KB
 - logická adresa (v programe) – 2 časti: segment:offset (SEG:OFF)
 - SEG – báza segmentu v pamäti
 - OFF – relatívna adresa v rámci segmentu
- výpočet fyzickej adresy z logickej [1]
 - predpokladajú sa 0000 v najnižších 4-bitoch bázy segmentu
 - umiestnenie začiatku segmentu v pamäti (napr. 00010H, FFFE0H, ...)



Příklad: SEG = 1100H, OFF = 450H

```
11000 (segment * 24)
  450 (offset)
-----
11450 (fyzická adresa)
```

- logická vs. fyzická adresa
 - pre každú logickú adresu existuje jedinečná fyzická
 - opačne to neplatí – viac ako jedna logická adresa môže referovať na rovnakú fyzickú adresu

Příklad: SEG1 = 1000, OFF1 = 20A9, SEG2 = 1200, OFF2 = A9.

```
10000      12000
  20A9      00A9
-----
120A9      120A9 (fyzická adresa)
```

- možno použiť 6 segmentových registrov (CS, DS, SS, ES, FS, GS), aj keď i8086 mal len prvé 4 [1]
- segmentové registre sú nezávislé – segmenty môžu byť oddelené, môžu sa čiastočne/úplne prekrývať

Činnosť procesora v zmiešanom režime (mixed-mode operation)

- operandy a adresy 16/32-bit
- bit D/B deskriptora určuje predpokladanú veľkosť
- prefixy určenia veľkosti 2 typov, umožnenie programovania v zmiešanom režime (signalizovaná iná ako predpokladaná veľkosť):
 - operandu
 - adresy

Formát inštrukcie

- detailný opis jednotlivých častí napr v. [6], [7]
- voliteľné prefixy i ďalšie časti inštrukcie
- ModR/M – špecifikácia typu adresovania
 - niektoré hodnoty indikujú ďalší bajt (SIB) nasledujúci za ModR/M bajtom
- SIB (Scale Index Base)
- max. dĺžka inštrukcie – 15B [7], [9]

Instruction prefix	Address-size prefix	Operand-size prefix	Segment override
0 or 1B	0/1B	0/1B	0/1B

Opcode	ModR/M	SIB	Displacement	Immediate
1/2B	0/1B	0/1B	0,1,2 or 4B	0,1,2 or 4B

Použitie segmentových registrov

v závislosti od typu pamäťovej referencie:

- výber inštrukcie – CS (báza segmentu), IP/EIP (offset)
- zásobníkové operácie – SS (báza), SP/ESP (offset) – push a pop, pre ďalšie operácie často BP/EBP
- prístup k údajom – obyčajne DS (báza), posun v závislosti od režimu adresovania

Inicializácia

- BIST (built-in self-test), výsledok testu v EAX (0 = OK), EDX – identifikácia procesora
- štartovacia adresa: 0FFFFFFF0H (16B od konca 32-bit adresného priestoru)
- ROM pamäť (BIOS) mapovaná v tejto oblasti – inicializácia systému
- Pentium štartuje v reálnom režime, prepnutie do chráneného režimu (tabuľky GDT, IDT, register CR0)

Jazyk rodiny procesorov x86

Asembler – príkazy:

- *inštrukcie* (procesor, inštrukcie strojového jazyka)
- *direktívy* (prekladač – riadenie prekladu, nie generovanie strojových inštrukcií pre procesor)
- *makrá* (skupina príkazov, nahradenie počas prekladu – expanzia)

Formát príkazov:

[návestie] mnemonika [operandy] [; komentár]

- návestie (oddelené „:“ od mnemoniky inštrukcie, nie však v prípade direktív)

Príklad:

```
opakuj: inc      pocet    ;zvyš 'pocet' o 1
CR      EQU      0DH      ;carriage return
```

Alokácia údajových objektov

- HL jazyky – nepriamo, špecifikáciou typu premennej (`int suma`)
- okrem alokácie miesta, daný aj spôsob interpretácie (`unsigned s1` vs. `int s2`; 4B obidve premenné)
- assembler – alokácia, príp. aj inicializácia (nie interpretácia), formát:

[premenná] direktíva poč.hodnota [, poč.hodnota]

- direktívy alokácie (inicializácia):
 - DB (define byte, 1B), DW (define word, 2B), DD (define doubleword, 4B), DQ (define quadword, 8B), DT (def. ten bytes, 10B)

Príklad:

```
sorted  DB  'y'           (rovnako: sorted DB 79H)
value   DW  25159         (6247H, Little endian: 47, 62)
expr1   DW  7*25          (inicializácia vo forme výrazu)
```

- rozsah hodnôt (znamienkové a bezznamienkové čísla)
- direktívy alokácie (bez inicializácie) – RESB (reserve byte, 1B), RESW (2B), RESD (4B), RESQ (8B), REST (10B)

Príklad:

```
buffer  resw    100      (pole 100 slov)
```

Viacnásobné definície

- alokácia súvislého bloku pamäti pre postupnosť definícií
- možnosť definície skrátiť

Príklad:

```
message DB 'BYE', 0DH, 0AH
```

- direktíva TIMES – viacnásobná inicializácia na rovnakú hodnotu (polia)

Príklad:

```
marks   TIMES 8 DW 0
```

Operandy

- väčšina inštrukcií vyžaduje operandy
- *adresovanie* – spôsob špecifikácie umiestnenia operandu
- základné typy operandov (detaily neskôr):
 - *registrový* (register CPU obsahuje údaj, rýchlosť)

```
mov     EAX,EBX
```

- *bezprostredný* (údaj v rámci inštrukcie – kód. segment)

```
mov     AL,65                (zdroj bezprostredný, cieľ - register)
```

- *pamäťový* (efektívna adresa (offset): súčasťou inštrukcie – *priama*, v registri – *nepriama*)
 - adresa vs. hodnota (NASM: `mov EBX,table, resp. mov EBX,[table]`)

```
table    TIMES 20 DD 0
...
mov      [table], 'A'        ; direct memory addressing

mov      EBX,table           ; indirect memory addressing
mov      [EBX],100           ; table[0] = 100                (rozmer údajaja?)
add      EBX,4               ; položka tabuľky má 4B (DD)
mov      [EBX],99            ; table[1] = 99
```

- *implicitný* operand

```
stc, clc, pusha, popa, ...
```

Základné inštrukcie jazyka procesora Pentium

- referenčná príručka (napr. v [2], [3], [4])

Inštrukcie presunov

- inštrukcia MOV (kopírovanie údajov)
 - syntax, pravidlá použitia (src – bez zmeny, rovnaký rozmer operandov)

```
mov      dst,src             (dst ← src)
```

- kombinácie operandov (R – register, M – memory, I – immediate)

mov	R, R			mov	R, M	
mov	R, I	(mov	EAX, 3)	mov	M, R	
mov	M, I			mov	M, M	- neprípustná kombinácia!

- (explicitná) špecifikácia rozmeru údajov (BYTE, WORD, DWORD, QWORD, TBYTE)

mov	[EBX], 100	; nie je jasný rozmer údajov pre zápis
mov	BYTE [EBX], 100	; zapíše sa 1B
mov	WORD [EBX], 100	; zapíše sa 2B

- inštrukcia XCHG (exchange)

- výmena operandov, syntax

xchg dst, src (dst ↔ src)

- pravidlá (obidva operandy v pamäti – neprípustné)
- výmena bez pomocného registra (triedenie, malý/veľký endian)

mov	ECX, EAX	vs.	xchg	EAX, EDX
mov	EAX, EDX			
mov	EDX, ECX			

- inštrukcia XLAT (translate)

- prevod znakov, syntax

xlat (AL ← [EBX + AL])

- použitie

- vstup: EBX – zač. adresa tabuľky, AL – index
- výstup: AL (bajt z výslednej adresy)

Aritmetické operácie

- inštrukcie INC a DEC
 - zvýšenie/zníženie hodnoty operandu (8, 16, 32-bit; register/pamäť) o 1, syntax

```
inc    dst                (inc EBX)
dec    dst                (dec DL)
```

Príklad:

```
count  DW 0
value  DB 25
.code
inc     [count]           ; OK (typ 'count' známy)
mov     EBX, count
inc     [EBX]             ; nejednoznačnosť
inc     WORD[EBX]         ; OK
```

- inštrukcia ADD
 - sčítanie 8, 16, 32-bit operandov, syntax

```
add     dst, src          (dst ← dst + src)
```

- pravidlá (podobne ako MOV)
- INC vs. ADD (preferované INC – menej pamäti)

```
inc     EAX               vs.    add     EAX, 1
```

- inštrukcia SUB
 - odčítanie 8, 16, 32-bit operandov, syntax

```
sub     dst, src          (dst ← dst - src)
```

- inštrukcia CMP
 - podobne ako SUB, ale výsledok sa neukladá ('dst' bez zmeny, nastavenie príznakov)
 - typické použitie – podmienené skoky

Skoky a iterácie

- nepodmienенý skok – JMP
 - odovzdanie riadenia na návestie label

```
jmp    label    (EIP ← label)
```

```
        mov     EAX, 1
inc_again:
        inc     EAX
        jmp     inc_again
        mov     EBX, EAX
        . . .
```

- podmienené skoky – vykonávanie programu sa prenesie na návestie (pri splnení podmienky)

```
j<podm> label
```

Príklad:

```
cmp     AL, 0DH
je      CD_received    ; jump on equal
```

- register FLAGS (ZF = 1 pri rovnosti operandov CMP)
- nie všetky inštrukcie modifikujú príznaky (napr. MOV)
- podmienky
 - jednoduchý test hodnoty príznaku (jz, jnz, jc, jnc, ...)
 - komplexnejšie podmienky (jg, jl, jge, jle, ...)

- iterácie
 - možno realizovať aj pomocou skokov
 - priama podpora iterácií – loop
 - syntax

```
loop    label
```

- činnosť (ECX – počítadlo):

```
ECX = ECX - 1
if ECX ≠ 0 then EIP ← label
```

```
        mov     CL, 50
repeat:
        <loop body>
        dec     CL
        jnz     repeat
        . . .
```

```
        mov     ECX, 50
repeat:
        <loop body>
        loop    repeat
        . . .
```


Logické operácie

- inštrukcie AND, OR, XOR, NOT

- syntax

```

and    dst,src      (or, xor)
not    dst           (unárny operátor)

```

- tabuľky hodnôt

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

xor	0	1
0	0	1
1	1	0

Příklad: operácie AND, OR, XOR, NOT [1]

AL	BL	and AL,BL	or AL,BL	xor AL,BL	not AL
		AL	AL	AL	AL
1010 1110	1111 0000	1010 0000	1111 1110	0101 1110	0101 0001
0110 0011	1001 1100	0000 0000	1111 1111	1111 1111	1001 1100
1100 0110	0000 0011	0000 0010	1100 0111	1100 0101	0011 1001
1111 0000	0000 1111	0000 0000	1111 1111	1111 1111	0000 1111

Příklad: použitie logických operácií

```

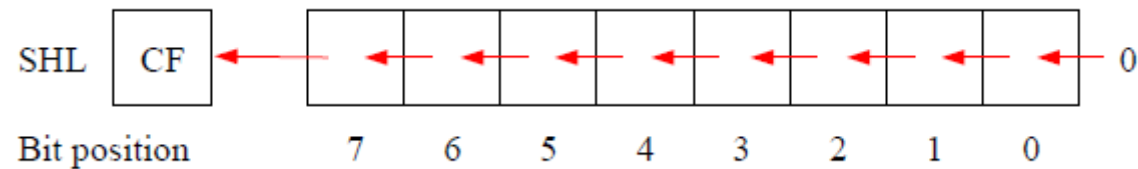
and    AL,01H
je     bit_is_zero
<code to be executed when the bit is 1>
jmp    skip1
bit_is_zero:
<code to be executed when the bit is 0>
skip1:
<rest of the code>

```

- inštrukcia TEST
 - ak je problémom modifikácia operandu
 - vykonáva bitovú operáciu AND (ako inštrukcia and), nastaví príznaky rovnako, bez modifikácie operandu

```
test    AL,01H           ;možno použiť v predošlom príklade namiesto: and    AL,01H
```

- posuny
 - SHL (shift left) – posun o jeden bit vľavo ($CF \leftarrow MSb$, $LSb \leftarrow 0$) [1]



- SHR (shift right) – posun o jeden bit vpravo ($CF \leftarrow LSb$, $MSb \leftarrow 0$)
- syntax (SHL, SHR), dst (8, 16, 32-bit M/R)

```
shl     dst,count        ; špecifikácia posunu priamo (0-31)
shl     dst,CL            ; nepriamo, v registri CL
```

Príklad: využitie operácií posunov (LSb , CF)

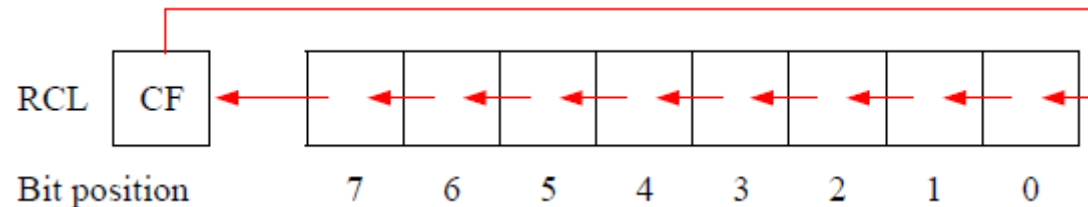
```
shr     AL,1
jnc     bit_is_zero
<code to be executed when the bit is 1>
jmp     skip1
bit_is_zero:
<code to be executed when the bit is 0>
skip1:
<rest of the code>
```

- rotácie
 - strata bitov pri posunoch – niekedy nežiadúca
 - rotácie bez CF – inštrukcie ROL, ROR
 - posledný vystupujúci bit zachytený v CF
 - syntax (ROL, ROR)

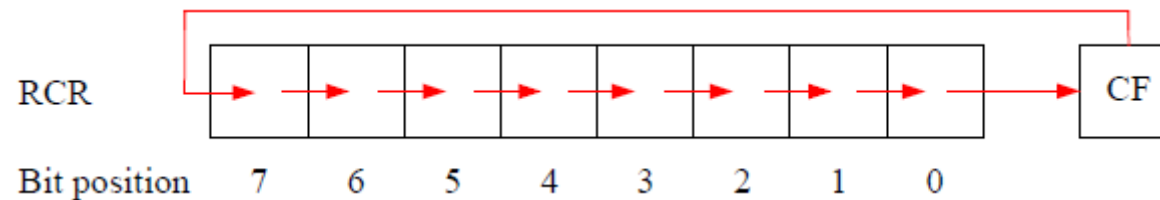
```
rol    dst, count
rol    dst, CL
```

- ROL – rotácia vľavo; bity vystupujúce vľavo vstupujú sprava ($D_{i+1} \leftarrow D_i$, $LSb \leftarrow MSb$, $CF \leftarrow MSb$)
- ROR – rotácia vpravo; bity vystupujúce vpravo vstupujú zľava ($D_i \leftarrow D_{i+1}$, $MSb \leftarrow LSb$, $CF \leftarrow LSb$)

- rotácie cez CF – inštrukcie RCL, RCR
 - CF zahrnutý v procese rotácie, syntax (ako u ROL, ROR)
 - RCL – rotácia vľavo; vystupujúci bit vstupuje do CF, CF vstupuje sprava ($D_{i+1} \leftarrow D_i$, $CF \leftarrow MSb$, $LSb \leftarrow CF$) [1]



- RCR – rotácia vpravo; vystupujúci bit vstupuje do CF, CF vstupuje zľava ($D_i \leftarrow D_{i+1}$, $CF \leftarrow LSb$, $MSb \leftarrow CF$) [1]



Príklad: využitie pri viacsloných posunoch (64-bit, EDX:EAX o 1 bit vpravo)

```
shr    EDX, 1
rcr    EAX, 1
```

Konštanty a Makrá

NASM poskytuje viacero možností, priblížime 3 direktívy:

- direktíva EQU
 - syntax

```
name EQU expr
```
 - použitie – priradenie výsledku výrazu 'expr' menu 'name' (expr – možno vyhodnotiť v čase prekladu)
 - výhody: čitateľnosť programu, viacnásobný výskyt konštanty – zmena na jednom mieste
 - priradené hodnoty nemožno meniť v danom zdrojovom module

Príklad:

```
ROWS EQU 40
COLS EQU 20
ARRAY_SIZE EQU ROWS * COLS
```

- direktíva %assign – podobne ako EQU (definícia numerických konštánt), ale opätovná definícia možná

Príklad:

```
%assign i j+1
<fragment kódu>
%assign i j+2
```

- direktíva %define – numerické i reťazcové konštanty, opätovná definícia možná

Príklad:

```
%define X1 [EBP+4]
<fragment kódu>
%define X1 [EBP+20]
```

Makrá

- reprezentácia bloku textu menom; pri výskyte mena makra v programe – nahradenie príslušným blokom (macro expansion)

```
%macro macro_name para_count
    <macro body>
%endmacro
```

- volanie makra – použije sa meno makra s príslušnými parametrami
- makrá bez parametrov

definícia:

```
%macro multEAX_by_16
    sal    EAX, 4
%endmacro
```

volanie makra:

```
...
mov      EAX, 27
multEAX_by_16
...
```

po rozvinutí makra, assembler prekladá:

```
...
mov      EAX, 27
sal      EAX, 4
...
```

- makrá s parametrami – generalizácia makra; v tele makra – prístup k parametrom podľa poradia (%1)

definícia:

```
%macro mult_by_16 1
    sal    %1, 4
%endmacro
```

volanie makra:

```
...
mult_by_16 DL
...
```

po rozvinutí makra:

```
...
sal      DL, 4
...
```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] [NASM](#) – The Netwide Assembler, The NASM Development Team, 1996-2015.
- [3] Carter, A.P.: [PC Assembly Language](#), 2019.
- [4] Cloutier, F.: [x86 and amd64 instruction reference](#), 2022.
- [5] Stack Overflow, [Difference between DPL and RPL in x86](#), 2014.
- [6] Intel 80386 Reference Programmer's Manual, [Instruction Format](#).
- [7] Kholodov, I.: Bristol Community College, CIS-77 Introduction to Computer Systems, [Encoding Real x86 Instructions](#).
- [8] Intel 80386 Reference Programmer's Manual, [Page Translation](#).
- [9] [Intel® 64 and IA-32 Architectures Software Developer's Manual](#), Intel Corporation, 2022.

Tvorba programu v asembleri

- preklad (assembling) – preklad programu v asembleri do strojového kódu
- spájanie (linking) – spojenie strojového kódu a údajov v objektových súboroch, prípadne knižníc → vykonateľný súbor
- ladenie (debugging)

Vstup a výstup

- neexistencia štandardného mechanizmu pri programovaní v asembleri (systémovo závislé)
- riešenia
 - priamy prístup k hardvéru
 - služby operačného systému
 - HL jazyky – štandardné knižnice
- v tomto kurze použité riešenie [2]
 - podprogramy pre zjednodušenie základných I/O operácií (využívajú knižnicu jazyka C)
 - `print_int` (celočíselná hodnota v EAX)
 - `print_char` (ASCII v AL)
 - `print_string` (adresa v EAX, nulový terminátor)
 - `print_nl` (nový riadok)
 - `read_int` (uloží hodnotu do EAX)
 - `read_char` (ASCII do EAX)
 - zachovávajú hodnoty registrov (`read*` – modifikujú EAX)
 - použitie
 - `%include "asm_io.inc"`
 - volanie jednotlivých operácií (`call`)

Ladenie programu

- zvolený prístup použitý v [2]
- zobrazenie stavu počítača (bez jeho modifikácie)
- makrá definované v `asm_io.inc`
- operácie:
 - `dump_regs` (HEX výpis obsahu registrov, FLAGS, argument – číslo výpisu)
 - `dump_mem` (HEX/ASCII výpis obsahu pamäti, arg1 – číslo výpisu, arg2 – adresa, arg3 – počet 16B blokov)
 - `dump_stack` (výpis obsahu zásobníka, arg1 – číslo výpisu, arg2 – # dvoj-slov pod EBP, arg3 – # dvoj-slov nad EBP)
 - `dump_math` (hodnoty v registroch FPU, argument – číslo výpisu)

Využitie jazyka C pri tvorbe programov v asembleri

- programy (pre PC) v súčasnosti zriedka vytvárané v samotnom asembleri (HLL – jednoduchosť, prenositeľnosť, ...)
- častejšie – kombinácia s HL jazykom (napr. C) – kritické časti v asembleri (rýchlosť, prístup k hardvéru, špeciálne inštrukcie ...)
- štart (asm) programu z C ovládača (C-driver), motivácia:
 - prepnutie do chráneného režimu (inicializácia SR, tabuľky v pamäti)
 - dostupnosť štandardnej knižnice jazyka C (I/O, ...)

Vzorový program

- ovládač v jazyku C (driver.c) [2]

```
int main()
{
    int ret_status;
    ret_status = asm_main();
    return ret_status;
}
```

- samotný program v jazyku assembler
 - segment .data (inicializované údaje, reťazce ukončené 0)
 - segment .bss (neinicializované údaje, ak sú potrebné)
 - segment .text (kód programu)
 - _asm_main (C-volacia konvencia; všetky symboly C (funkcie, globálne premenné) – prefix '_' (Win))
 - global _asm_main (vytvorenie globálneho návestia, viditeľné aj v iných moduloch)
- preklad (MinGW 32-bit)
 - assembler nasm -f win32 -d COFF_TYPE asm_io.asm (asm_io.obj)
 - assembler nasm -f win32 prvy.asm (prvy.obj)
 - driver.c gcc -c driver.c (driver.obj)
 - spájanie gcc -o prvy driver.obj prvy.obj asm_io.obj (gcc volá linker s vhodnými parametrami, prvy.exe)
- preklad (MinGW 64-bit, „multilib“ verzia)
 - assembler – bez zmeny
 - spájanie gcc -o prvy -m32 prvy.obj driver.c asm_io.obj (prvy.exe)
- ďalšie výstupné formáty NASM napr. v [3]

Príklad: prevod a výpis hodnoty zadaného znaku v HEX formáte (adaptované podľa [1]).

```
; Objective:   HEX hodnota zadaného znaku.  
; Input:      Znak z klavesnice.  
; Output:     Vytlačí ASCII kód znaku v HEX.
```

```
%include "asm_io.inc"
```

```
segment .data
```

```
char_prompt    db  "Zadaj znak: ",0  
out_msg1       db  "ASCII kód znaku '",0  
out_msg2       db  "' v HEX je ",0
```

```
segment .text
```

```
    global _asm_main
```

```
_asm_main:
```

```
    enter    0,0  
    pusha
```

```
    mov     EAX,char_prompt  
    call    print_string  
    call    read_char  
    mov     EBX,EAX  
    call    print_nl
```

```
    mov     EAX,out_msg1  
    call    print_string  
    mov     EAX,EBX  
    call    print_char  
    mov     EAX,out_msg2  
    call    print_string  
    mov     EAX,EBX  
    mov     AH,AL  
    shr     AL,4  
    mov     ECX,2
```

```
C:\Dev-Cpp\bin\asm>nasm -f win32 prvvy.asm  
C:\Dev-Cpp\bin\asm>nasm -f win32 -d COFF_TYPE asm_io.asm  
C:\Dev-Cpp\bin\asm>gcc -c driver.c  
C:\Dev-Cpp\bin\asm>gcc -o prvvy driver.o prvvy.obj asm_io.obj  
C:\Dev-Cpp\bin\asm>prvvy  
Zadaj znak: A  
  
ASCII kód znaku 'A' v HEX je 41  
C:\Dev-Cpp\bin\asm>
```

```
print_digit:
```

```
    cmp     AL,9  
    jg      A_to_F      ; prevod na A - F  
    add     AL,'0'      ; prevod na 0 - 9  
    jmp     skip
```

```
A_to_F:
```

```
    add     AL,'A'-10
```

```
skip:
```

```
    call    print_char  
    mov     AL,AH  
    and     AL,0FH  
    loop    print_digit
```

```
    popa  
    mov     EAX,0  
    leave  
    ret
```

Zásobník, podprogramy

- procedúry a modulárne programovanie
- úloha zásobníka pri používaní procedúr
- implementácia zásobníka (Pentium) [1]

Zásobník

- LIFO údajová štruktúra, operácie PUSH a POP
- priamo prístupný len prvok na vrchole zásobníka (TOS)
- poradie vkladáných/vyberaných prvkov

Implementácia zásobníka (Pentium)

- oblasť pamäti rezervovaná v zásobníkovom segmente
- vrchol zásobníka (TOS) daný SS:ESP
 - SS – začiatok segmentu zásobníka
 - ESP – relatívna adresa (offset)
- možno vložiť slovo (16-bit), dvoj-slovo (32-bit)
- zásobník narastá smerom k nižším adresám
- TOS – posledná vložená položka (nižší byte)

TOS → ↑ ESP (256) ↓ SS →		TOS → ↑ ESP (254) ↓ SS →		TOS → ↑ ESP (250) ↓ SS →	
	??		21		21
	??		AB		AB
	??		??		7F
	??		??		BD
	??		??		32
	??		??		9A
	:		:		:
	??		??		??
	??		??		??
(a) Empty stack (256B)		(b) After pushing 21ABH		(c) After pushing 7FBD329AH	

Príklad: zásobník o veľkosti 256B

- prázdny – pri pokuse o výber – chyba (stack underflow)
 - vložené slovo (2B) – zníženie ESP, potom uloženie (little endian)
 - vložené dvoj-slovo (4B)
- plný zásobník (ESP = 0), pokus o uloženie – chyba (stack overflow)
 - výber zo zásobníka
 - zvýšenie hodnoty ESP (pamäť bez zmeny)
 - uvoľnené lokality použiteľné pre uloženie nových hodnôt

Operácie so zásobníkom

- inštrukcie push a pop
 - uloženie slova/dvoj-slova na zásobník
 - syntax

```
push    src      (src - 16/32-bit GPR, SR, pamäť, konštanta)
pop     dst      (dst - 16/32-bit GPR, SR, pamäť)
```

- sémantika

```
push    src16    ESP ← ESP - 2, [SS:ESP] ← src16
push    src32    ESP ← ESP - 4, [SS:ESP] ← src32
pop     dst16    dst16 ← [SS:ESP], ESP ← ESP+2
pop     dst32    dst32 ← [SS:ESP], ESP ← ESP+4
```

- inštrukcie pushf a popf
 - uloženie a obnovenie registra príznakov (FLAGS)
 - syntax

```
pushfd          (32-bit EFLAGS)
popfd
pushfw          (16-bit FLAGS)
popfw
```

- inštrukcie pusha a popa
 - uloženie GPR registrov (8)
 - syntax

```
pushad          (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI)
popad           (obnoví, okrem ESP)

pushaw          (AX, CX, DX, BX, SP, BP, SI, DI)
popaw           (okrem SP)
```

Typické využitie zásobníka

- dočasné ukladanie údajov
 - limitovaný počet GPR registrov
 - uloženie a obnova obsahu registrov

Příklad: výmena obsahu dvoch 32-bit premenných (value1, value2)

a)

```
push    EAX      ; uloženie registrov
push    EBX
mov     EAX,value1
mov     EBX,value2
mov     value1,EBX
mov     value2,EAX
pop     EBX
pop     EAX      ; obnovenie obsahu
```

b)

```
push    value1
push    value2
pop     value1
pop     value2
```

- riadenie vykonávania programu
 - volanie procedúr (návratová adresa)
- odovzdávanie parametrov procedúram (parameter passing)
 - HL jazyky

Procedúry

- logicky samostatná časť kódu pre vykonávanie určitej úlohy
- mechanizmy odovzdávania parametrov
 - hodnotou (call by value) – iba aktuálna hodnota parametrov (bez možnosti zmeny procedúrou)
 - referenciou (call by reference) – adresy parametrov (možnosť zmeny hodnôt parametrov)
- HL jazyky (2 typy podprogramov)
 - funkcie (jediná návratová hodnota)
 - procedúry (žiadna/viac návratových hodnôt)

Procedúry v jazyku x86 (Pentium)

- volanie procedúry (proc-name – relatívne voči inštrukcii nasledujúcej za call (offset))

```
call    proc-name                (ESP ← ESP-4, [SS:ESP] ← EIP, EIP ← EIP+rel.posun)
```

Príklad [1]:

offset (hex)	machine code (hex)	
		main:
		...
00000002	E816000000	call sum
00000007	89C3	mov EBX,EAX
		...
		; end of main
; *****		
		sum:
0000001D	55	push EBP
		...
		; end of sum procedure
; *****		
		avg:
		...
00000028	E8F0FFFFFF	call sum
0000002D	89D8	mov EAX,EBX
		...
		; end of avg procedure

- po výbere call, EIP = 00000007H
- uloženie EIP do zásobníka
- prenos riadenia na začiatok procedúry 'sum'
 - pripočítanie posunu k EIP
 - 00000007H + 00000016H = 0000001DH
- dopredné volanie – kladný posun (signed, 32-bit)
- procedúra 'avg' – volanie späť (záporný posun)

- návrat z procedúry
 - odovzdanie riadenia späť z volanej do volajúcej procedúry (inštrukcia nasledujúca za call)
 - návratová adresa (zásobník)

```
ret    (EIP ← [SS:ESP], ESP ← ESP+4)
```

- parameter inštrukcie `ret` (počet bajtov pre uvoľnenie zo zásobníka, voliteľné)

```
ret    n
```

Odovzdávanie parametrov

- uloženie parametrov na dohodnuté miesto (registre, zásobník)
- volanie procedúry

Registrová metóda

- GPR registre

Priklad: odovzdávanie hodnotou, registrová metóda (adaptované [1]).

```
%include "asm_io.inc"
segment .data
prompt_msg1 db "Zadaj prve cislo: ",0
prompt_msg2 db "Zadaj druhe cislo: ",0
sum_msg      db "Suma cisel je: ",0

segment .text
global _asm_main
_asm_main:
    enter    0,0
    pusha

    mov     EAX,prompt_msg1
    call    print_string
    call    read_int
    mov     ECX,EAX
    call    print_nl
    mov     EAX,prompt_msg2
```

```
    call    print_string
    call    read_int
    mov     EDX,EAX
    call    print_nl
    mov     EAX,sum_msg
    call    print_string
    call    sum
    call    print_int
    call    print_nl

    popa
    mov     EAX,0
    leave
    ret

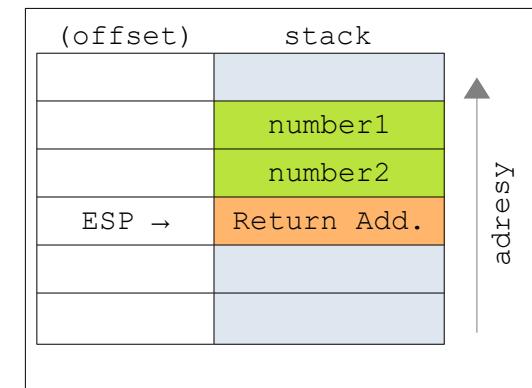
sum:
    mov     EAX,ECX
    add     EAX,EDX
    ret
```

- registrová metóda – vlastnosti
 - výhody
 - jednoduchšia pre malý počet parametrov
 - rýchla (parametre v registroch)
 - nevýhody
 - obmedzený počet GPR registrov (parametrov)
 - často potrebná dočasná úschova používaných GPR (zásobník) – strata výhody rýchlosti

Zásobníková metóda

- všetky parametre v zásobníku pred volaním procedúry
- problém – prístup k parametrom (call/EIP)

```
push    number1
push    number2
call    sum
```



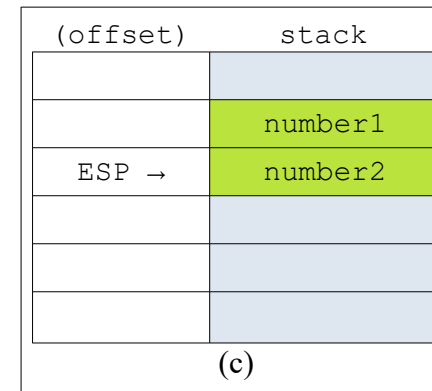
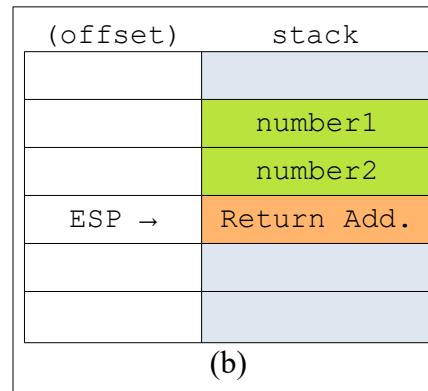
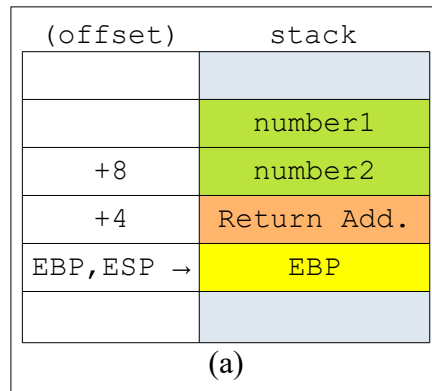
- riešenie – prístup s využitím EBP (obsah ESP sa mení – push/pop)
- štandardný spôsob prístupu k parametrom

```
mov     EBP, ESP
mov     EAX, [EBP+4]
```

- uschovanie obsahu EBP (možná obnova pôvodnej hodnoty pred návratom k volajúcej procedúre)

```
push    EBP
mov     EBP, ESP
```

- informácia v zásobníku (parametre, návratová adresa, starý EBP, príp. lokálne premenné) – *zásobníkový rámeček* (stack frame) [1]
 - zásobník po uložení EBP (a)
 - pred ukončením procedúry – obnova EBP (b)
 - po vykonaní inštrukcie `ret` (c)



- uvoľnenie parametrov zo zásobníka
 - úprava ESP *volajúcou* procedúrou (napr. jazyk C)

```

push    number1
push    number2
call    sum
add     ESP, 4           (ak máme 2 premenne po 2B)

```

- úprava *volanou* procedúrou

```

ret     hodnota          (EIP ← SS:ESP, ESP ← ESP + 4 + hodnota)

```

- uvoľnenie zásobníka – porovnanie prístupov
 - procedúry s pevným počtom parametrov (preferovaná druhá metóda)
 - procedúry s premenlivým počtom parametrov (potrebné použiť prvú)

Uchovanie stavu procedúry

- napr. volaná procedúra manipuluje s registrom využívaným volajúcou
- obsahy registrov spravidla uchováva volaná procedúra
 - dôvody (zmeny volanej procedúry, viacnásobné volania – dlhší kód)
- ktoré registre uchovať?
 - všeobecne: používané volajúcou, modifikované volanou procedúrou
 - push vs. pushad
 - rýchlosť (5x, efektívne pri úschove > 5 registrov)
 - návratová hodnota v registri (gcc – EAX), popa následne znehodnotí
 - modifikácia prístupu k parametrom (offset)

```
pusha
mov     EBP, ESP
```

Inštrukcie ENTER a LEAVE

- enter – alokácia zásobníkového rámca pri vstupe do procedúry

```
enter   bytes, level
```

- bytes – počet bajtov pre lokálne premenné
- level – úroveň vnorenia procedúry (bežne 0)
- leave – uvoľnenie zásobníkového rámca (bez parametrov)

```
enter   XX, 0                                leave
push     EBP                                mov     ESP, EBP
mov      EBP, ESP                            pop      EBP
sub      ESP, XX
```

```
proc-name:
    enter    XX, 0
    ...
    procedure body
    ...
    leave
    ret      YY
```

Príklad: odovzdávanie parametrov, zásobníková metóda (adaptované [1]).

```
%include "asm_io.inc"
segment .data
prompt_msg1 db "Zadaj prve cislo: ",0
prompt_msg2 db "Zadaj druhe cislo: ",0
sum_msg      db "Suma cisel je: ",0

segment .text
global _asm_main
_asm_main:
    enter    0,0
    pusha

    mov     EAX,prompt_msg1
    call    print_string
    call    read_int
    push    EAX
    call    print_nl
    mov     EAX,prompt_msg2
    call    print_string
    call    read_int
    push    EAX
```

```
call    print_nl
mov     EAX,sum_msg
call    print_string
call    sum
call    print_int
call    print_nl
```

```
popa
mov     EAX,0
leave
ret
```

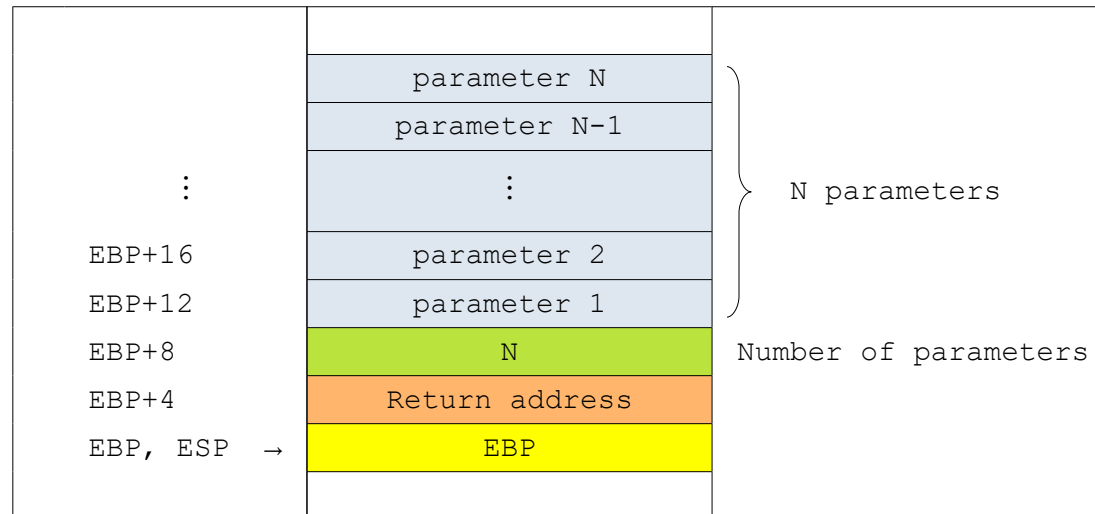
sum:

```
enter    0,0                ; save EBP
mov     EAX,[EBP+12]
add     EAX,[EBP+8]
leave   ; restore EBP
ret     8                    ; params
```

(offset)	stack
+12	num1
+8	num2
+4	R.A.
EBP →	old EBP

Procedúry s premenlivým počtom parametrov

- bežne prítomné napr. v jazyku C (`scanf`, `printf`)
- volaná procedúra potrebuje informáciu o počte parametrov [1]
 - posledný parameter vložený do zásobníka – počet parametrov
- parametre zo zásobníka odstránené volajúcim

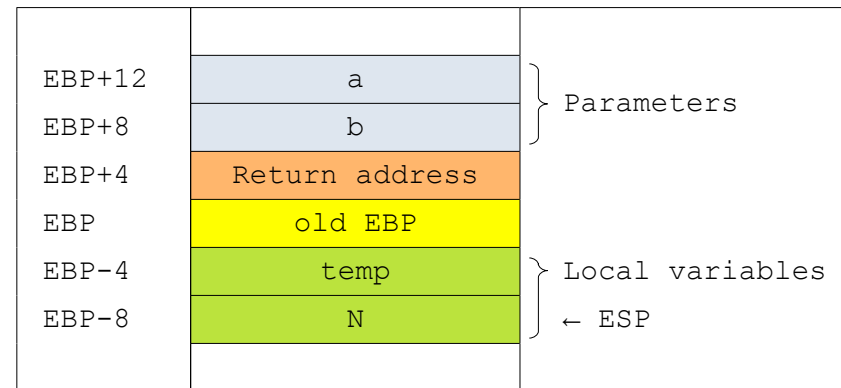


Lokálne premenné

- vznikajú pri aktivácii procedúry, zanikajú pri jej terminácii (sú dynamické)
- alokácia v dátovom segmente nežiadúca:
 - v DS alokácia statická (dostupné aj keď procedúra je neaktívna)
 - nepracuje korektne s rekurzívnymi procedúrami (volajú sami seba priamo/nepriamo)
- preto – alokácia v zásobníku [1]

Príklad:

```
int compute(int a, int b)
{
    int temp, N;
    ...
    ...
}
```



- prístup k obsahu zásobníkového rámca – EBP (frame pointer)
 - parametre (a, b – EBP+12, EBP+8) – poradie parametrov v zásobníku závislé od HL jazyka
 - lokálne premenné (temp, N – EBP-4, EBP-8)
- možnosť sprehľadnenia programu:

```
%define      a      dword[EBP+12]
%define      temp    dword[EBP-4]
```

Použitie:

mov	EBX, a	namiesto:	mov	EBX, [EBP+12]
mov	temp, EAX		mov	[EBP-4], EAX

Viacmodulové programy

- reálne aplikácie (často množstvo procedúr)
- rozdelenie zdrojového textu na menšie časti (moduly)
- výhody
 - zmeny v module, preklad (iba) konkrétneho modulu
 - jednoduchšia, bezpečnejšia modifikácia (menšie súbory)
- oddelený preklad modulov – potrebná špecifikácia rozhrania
 - NASM – direktívy GLOBAL a EXTERN

Direktíva GLOBAL

- sprístupní návestie iným modulom programu (mená procedúr, premenných, ...)

```
global label1, label2, ...
```

Direktíva EXTERN

- informuje assembler, že návestie nie je definované v aktuálnom module (definované v inom)

```
extern label1, label2, ...      (label1, label2 – sprístupnené v inom module pomocou GLOBAL)
```

```
global  error_txt, sum, mylabel

.DATA
error_txt  db 'Out of range!',0
sum        dw 0
          ...
.CODE
          ...
mylabel:
          ...
          ret
```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] [NASM](#) – The Netwide Assembler, The NASM Development Team, 1996-2015.

Adresovanie

- špecifikácia prístupu k údajom s ktorými inštrukcie pracujú
- polia – špecifikácia a manipulácia
- vplyv adresovania na výkon

Architektúra procesorov

- CISC (vyšší počet režimov adresovania)
- RISC (podpora obmedzeného množstva režimov)
 - inštrukcie pracujú s operandmi v registroch CPU
 - load/store inštrukcie – presuny medzi registrami a pamäťou

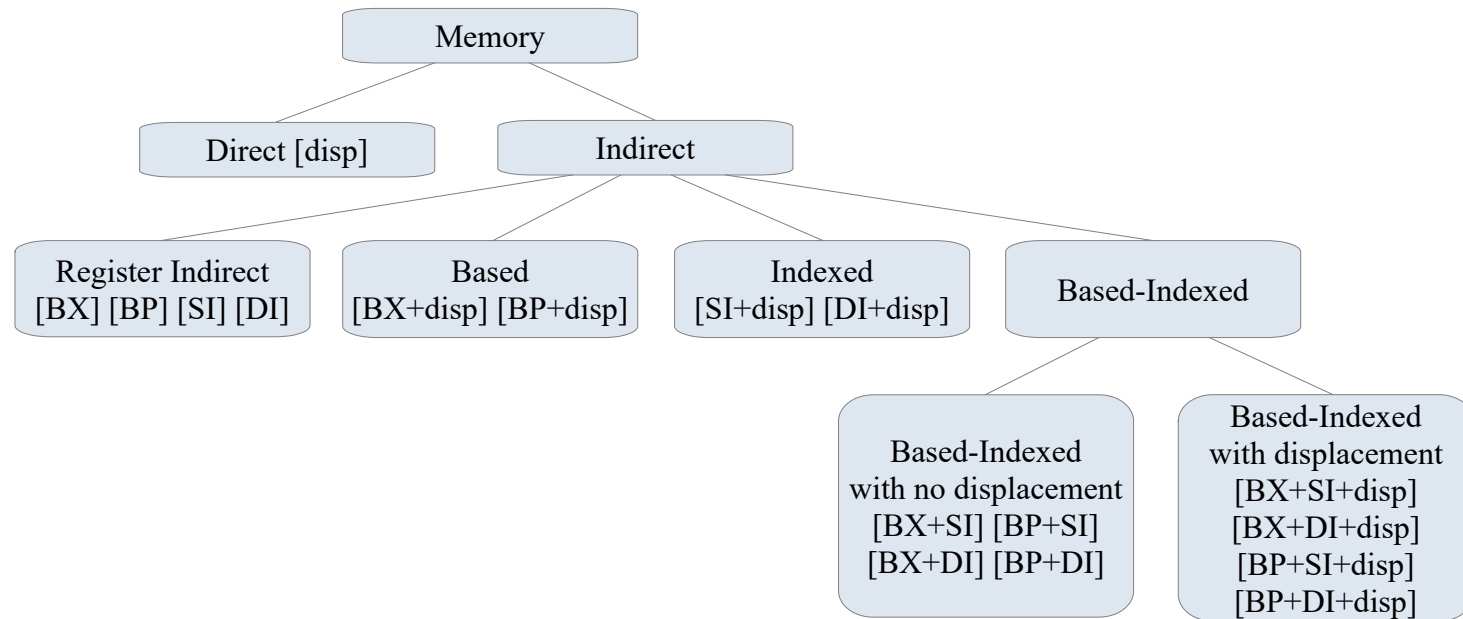
Adresovanie procesorov Pentium (x86, CISC)

- registrový (Register Addressing Mode) – registre CPU, rýchlosť
- bezprostredný (Immediate Addressing Mode) – najviac jeden, operand časťou inštrukcie
- pamäťový (Memory Addressing Mode) – množstvo režimov (spôsob špecifikácie efektívnej adresy (offset))

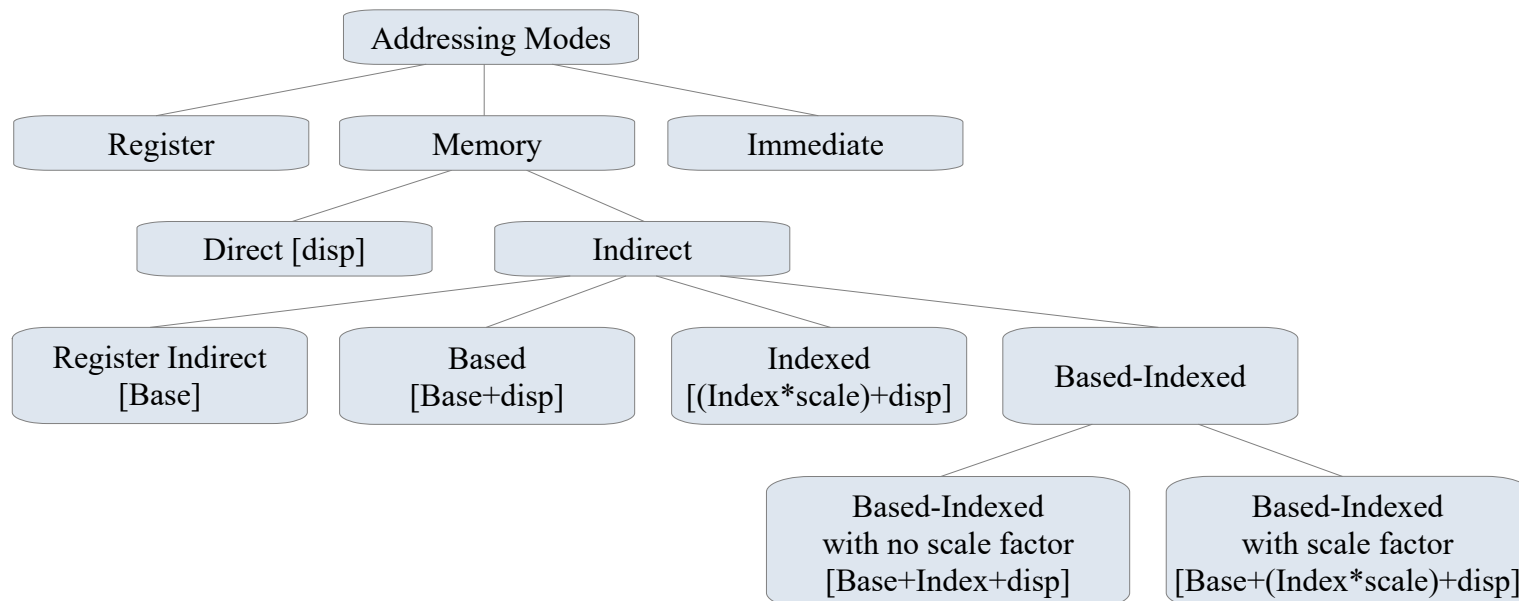
Pamäťový režim adresovania

- motivácia – efektívna podpora konštrukcií HL jazykov a údajových štruktúr
- dostupné režimy adresovania (podľa použitej veľkosti adresy)
 - 16-bit adresy (podobne ako i8086)
 - 32-bit adresy (vyššia flexibilita)

16-bit adresy [1]



32-bit adresy [1]



Porovnanie 16-bit a 32-bit režimov

- 32-bit režim prináša väčšiu flexibilitu v použití registrov [1]
- možnosť zohľadniť rozmer operandov (scale factor)

	16-bit addressing	32-bit addressing
Base register	BX BP	EAX, EBX, ECX, EDX ESI, EDI, EBP, ESP
Index register	SI DI	EAX, EBX, ECX, EDX ESI, EDI, EBP
Scale factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

- ktorý adresový režim má CPU použiť?
 - bit D segmentového deskriptora CS (D = 1: 32-bit default)
 - možnosť implicitnú voľbu explicitne zmeniť (size override prefix)
 - 66H (operand size override prefix)
 - 67H (address size override prefix)
- použitím prefixov zmeny veľkosti – možné zmiešané použitie 16/32-bit údajov a adres
- štandardne (predmet ASM) využívame 32-bit režim

Priklad: asembler

generovaný kód

mov	EAX, 123	B8 0000007B	
mov	AX, 123	66 B8 007B	(prefix automaticky vložený)
mov	AL, 123	B0 7B	(odlišný op.kod)
mov	EAX, [BX]	67 8B 07	
mov	AX, [BX]	66 67 8B 07	(obidva prefixy)

Bázové adresovanie

- jeden z registrov v úlohe bázy pri výpočte adresy operandu
- efektívna adresa – súčet obsahu registra a posunutia (so znamienkom)

Base + disp ; báza a posunutie so znamienkom

- prístup k prvkom štruktúry

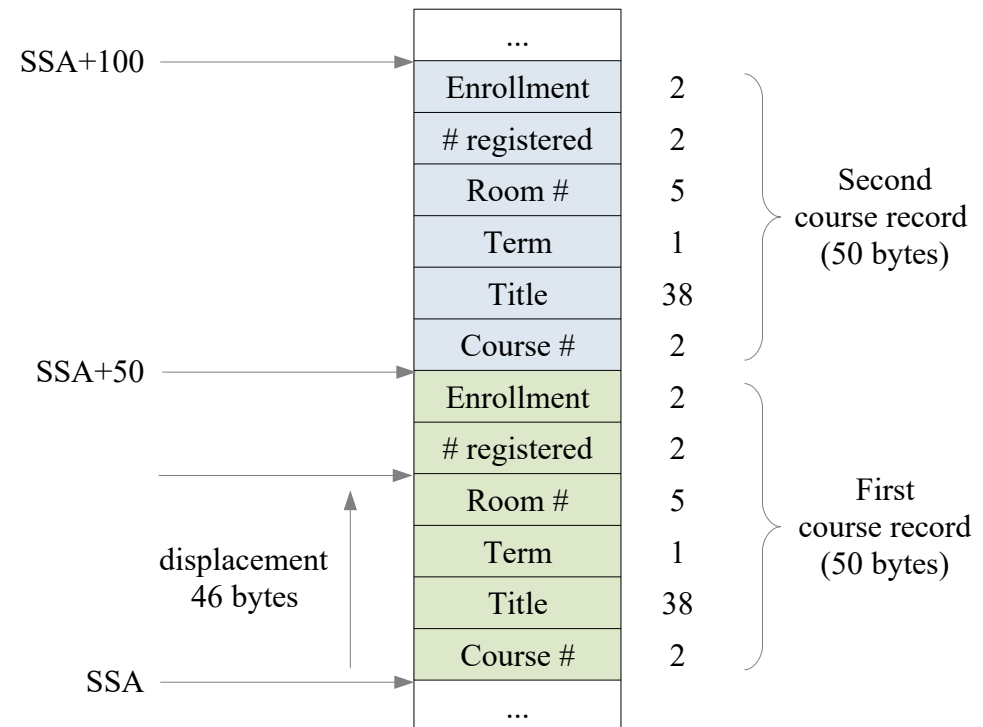
Příklad: počet voľných miest v danom kurze? [1]

Course number	Integer	2 bytes
Course title	Character string	38 bytes
Term offered	Single character	1 byte
Room number	Character string	5 bytes
Number registered	Integer	2 bytes
Enrollment limit	Integer	2 bytes
Total storage per record		50 bytes

nech EBX obsahuje SSA (Structure Starting Address):

```

...
mov    AX, [EBX + 48]
sub    AX, [EBX + 46]
...
```



Indexové adresovanie

- výpočet efektívnej adresy

$(\text{Index} * \text{scale}) + \text{disp}$; posunutie so znamienkom

- prístup k prvkom poľa
 - začiatok poľa (disp), prvok v poli (index register)
 - veľkosť prvkov ($\text{scale} - 2, 4, 8$ – iba 32-bit režim)

Příklad: `mov EAX, [marks_table + ESI*4]` ; prvky marks_table a table1 - 4B (scale)
`add EAX, [table1 + ESI]` ; ESI - offset v bytoch (napr. 36 pre 10.prvok)

Bázovo – indexové adresovanie

- dve verzie
 - bez zohľadnenia veľkosti operandu (B-I with No Scale Factor)

$\text{Base} + \text{Index} + \text{disp}$; posunutie so znamienkom (8/16 v 16-bit, 8/32 v 32-bit)

- dvojrozmerné polia (disp – začiatok poľa)
- polia záznamov (disp – offset položky záznamu)

- so zohľadnením veľkosti operandu (B-I with Scale Factor)
 - efektívny spôsob prístupu k prvkom dvojrozmerných polí (prvky veľkosti 2, 4, 8 B)

$\text{Base} + (\text{Index} * \text{scale}) + \text{disp}$

Příklad: porovnanie dvoch nasledujúcich prvkov (4B) poľa (adresa v EBX).

```
mov EAX, [EBX+ESI]
cmp EAX, [EBX+ESI+4]
```

Príklad: triedenie vkladáním – program získa postupnosť celých čísel, vypíše ich v utriedenom poradí (adaptované podľa [1]).

- činnosť algoritmu (vložiť nový prvok do utriedeného poľa na správnu pozíciu)
 - na začiatku prázdne pole
 - po vložení prvého prvku – utriedené
 - vložiť nový prvok na správnu pozíciu
 - opakovat' proces, kým nie sú vložené všetky prvky
- pseudokód
 - index i – vkladáný prvok
 - prvky vľavo od i – utriedené
 - prvky pre vloženie – vpravo od i (vrátane)

```
insertion_sort(array, size)
  for(i=1 to size-1)
    temp:=array[i]
    j:=i-1
    while((temp<array[j]) AND (j>=0))
      array[j+1]:=array[j]
      j:=j-1
    end while
    array[j+1]:=temp
  end for
end insertion_sort
```

```
Zadaj vstupne pole: (zaporne cislo ukonci vstup)
66
12
89
123
6
11
-7

Utriedene pole:
6
11
12
66
89
123
```

Hlavný program:

01 %include "asm_io.inc"	29 exit_loop:
02 MAX_SIZE EQU 100	30 mov EDX,EBX
03 segment .data	31 sub EDX,array
04 input_prompt db "Zadaj vstupne pole: "	32 shr EDX,2
05 db "(zaporne cislo ukonci vstup)",0	33 push EDX
06 out_msg db "Utriedene pole:",0	34 push array
07	35 call insertion_sort
08 segment .bss	36 mov EAX,out_msg
09 array resd MAX_SIZE	37 call print_string
10	38 call print_nl
11 segment .text	39 mov ECX,EDX
12 global _asm_main	40 mov EBX,array
13 _asm_main:	41 display_loop:
14 enter 0,0	42 mov EAX,[EBX]
15 pusha	43 call print_int
16	44 call print_nl
17 mov EAX,input_prompt	45 add EBX,4
18 call print_string	46 loop display_loop
19 mov EBX,array	47 done:
20 mov ECX,MAX_SIZE	48 popa
21 array_loop:	49 mov EAX,0
22 call read_int	50 leave
23 call print_nl	51 ret
24 cmp EAX,0	
25 jl exit_loop	
26 mov [EBX],EAX	
27 add EBX,4	
28 loop array_loop	

Procedúra insertion_sort:

<pre> 52 %define SORT_ARRAY EBX 53 insertion_sort: 54 pushad 55 mov EBP,ESP 56 57 mov EBX,[EBP+36] 58 mov ECX,[EBP+40] 59 mov ESI,4 60 for_loop: 61 ; mapovanie premennych: 62 ; EDX = temp, ESI = i, EDI = j 63 mov EDX,[SORT_ARRAY+ESI] 64 mov EDI,ESI ; j = i-1 65 sub EDI,4 </pre>	<pre> 66 while_loop: 67 cmp EDX,[SORT_ARRAY+EDI] 68 ; temp < array[j] 69 jge exit_while_loop 70 ; array[j+1] = array[j] 71 mov EAX,[SORT_ARRAY+EDI] 72 mov [SORT_ARRAY+EDI+4],EAX 73 sub EDI,4 ; j = j-1 74 cmp EDI,0 ; j >= 0 75 jge while_loop 76 exit_while_loop: 77 ; array[j+1] = temp 78 mov [SORT_ARRAY+EDI+4],EDX 79 add ESI,4 ; i = i+1 80 dec ECX 81 cmp ECX,1 82 jne for_loop 83 sort_done: 84 popad 85 ret 8 </pre>
---	--

- procedúra nevracia hodnotu (pushad/popad)
- prístup k parametrom (pushad – 32B)
- slučka while (r.66 – 76)
- slučka for (r.60 – 82)
- bázoové adresovanie (r.57, 58)
- bázoovo-indexové adresovanie (r.63, 67, 71, 72, 78)

```

insertion_sort(array,size)
    for(i=1 to size-1)
        temp:=array[i]
        j:=i-1
        while((temp<array[j])AND(j≥0))
            array[j+1]:=array[j]
            j:=j-1
        end while
        array[j+1]:=temp
    end for
end insertion_sort

```

Polia

Jednorozmerné polia

- jednorozmerné pole v C (index začína 0)

```
int    test_marks[10];
```

- HL deklarácia (veľkosť 40B)
 - meno poľa
 - počet prvkov (10)
 - veľkosť prvku (4)
 - typ prvku (int)
 - indexy (0 – 9)

- pole v asembleri – alokácia požadovaného miesta [1]

```
test_marks    resd    10
```

- korektný prístup k prvkom – úloha programátora (indexy, veľkosť prvkov)
- prvky usporiadané lineárne
- potrebné zistiť posun od začiatku poľa ($\text{posun} = \text{index} * \text{veľkosť prvku}$)

High memory

test_marks[9]
test_marks[8]
test_marks[7]
test_marks[6]
test_marks[5]
test_marks[4]
test_marks[3]
test_marks[2]
test_marks[1]
test_marks[0]

Low memory

← test_marks

Viacrozmerné polia

- dvojrozmerné pole v C (5r x 3s)

```
int    class_marks[5][3];
```

- reprezentácia v pamäti (lineárne pole bajtov, transformácia)
 - po riadkoch (row-major ordering, napr. C)
 - po stĺpcoch (column-major ordering, napr. Fortran)

- dvojrozmerné pole v asembleri [1]

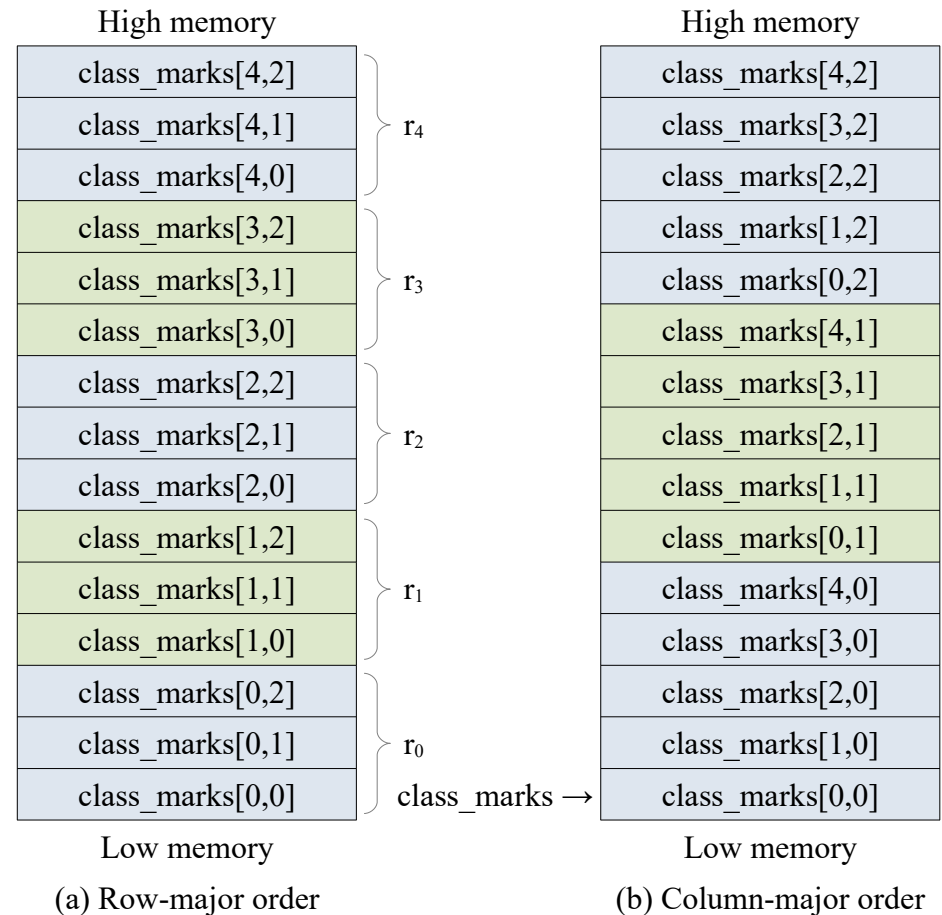
- reprezentácia poľa v pamäti podstatná
- alokácia (60B)

```
class_marks    resd    5*3
```

- preklad indexov na hodnotu posunu (po riadkoch)

```
posun = (i * COLS + j) * ELM_SIZE
```

- COLS – počet stĺpcov, i – riadok, j – stĺpec



Celočíselná aritmetika

- vplyv realizácie aritmetických a logických inštrukcií na stavové príznaky (FLAGS)
- násobenie a delenie
- aritmetické operácie nad viacslovnými údajmi (multi-word arithmetic)

Stavové príznaky

- 6 príznakov – monitorovanie výsledkov operácií
- ZF, CF, OF, SF, AF, PF
- ak je príznak aktualizovaný – ostáva nezmenený, pokiaľ jeho stav nezmení ďalšia inštrukcia
 - nie všetky inštrukcie ovplyvňujú stavové príznaky (add, sub – 6; inc, dec – okrem CF; mov, push – žiadne)
- príznaky možno testovať (individuálne, kombinácie) za účelom riadenia vykonávania programu

Príznak nuly (zero flag)

- výsledkom poslednej operácie (ovplyvňujúcej ZF) bola 0 – ZF = 1, ináč ZF = 0
- sub – intuitívne, iné inštrukcie možno menej

Príklad:

```
mov     AL, 0FH
add     AL, 0F1H          (nastaví ZF = 1, všetkých 8 bitov AL - 0)

mov     AX, 0FFFFH
inc     AX                (nastaví ZF = 1)

mov     EAX, 1
dec     EAX               (nastaví ZF = 1)
```

- inštrukcie podmienených skokov: jz (ak ZF = 1), jnz (ak ZF = 0)

- využitie príznaku ZF
 - test rovnosti (často inštrukcia `cmp`)

```
cmp    char, '$'
```

```
cmp    EAX, EBX
```

- počítanie do zadanej hodnoty [1]
 - $M, N \geq 1$, vnútorná slučka (`ECX/loop` – neovplyvňuje príznaky)
 - vonkajšia slučka (`EDX/dec/jnz`)

```
sum = 0
for (i=1 to M)
    for (j=1 to N)
        sum=sum+1
    end for
end for
```

```
sub    EAX, EAX    ; EAX = 0
mov    EDX, M
outer_loop:
    mov    ECX, N
inner_loop:
    inc    EAX
    loop   inner_loop
    dec    EDX
    jnz    outer_loop
exit_loops:
    mov    sum, EAX
```

Príznak prenosu (carry flag)

- výsledok aritmetickej operácie na bezznamienkových číslach prekročil rozsah cieľa (R/M)

Príklad:

```
mov    AL, 0FH          00001111
add    AL, 0F1H          11110001
                        -----
                        100000000
```

- v prípade 8-bit registra potreba 9.bitu (AL – 8-bitov)
- rozsah hodnôt bezznamienkových čísel [1]

Size (bits)	Range
8	0 to 255
16	0 to 65 535
32	0 to 4 294 967 295

- operácia produkujúca výsledok mimo rozsahu nastaví CF
- záporný výsledok teda mimo rozsahu

Príklad:

```
mov     EAX, 12AEH
sub     EAX, 12AFH
(4782 - 4783 = -1, CF = 1)
```

```
mov     EAX, 0
dec     EAX
(CF = 0, inc, dec - neovplyvňujú CF)
```

- inštrukcie podmienených skokov: `jc` (ak $CF = 1$), `jnc` (ak $CF = 0$)
- využitie CF
 - šírenie prenosu/výpožičky (carry/borrow) pri viacslovných operáciách sčítania/odčítania
 - inštrukcie – operandy rozmeru 8, 16, 32b, pri rozmernejších – po krokoch, zohľadnenie prenosov
 - detekcia pretečenia/podtečenia (overflow/underflow)
 - indikácia výsledku mimo rozsahu (ošetrenie situácie programátorom)
 - test bitu s využitím posunov/rotácií
 - bit (MSb, LSb) zachytený v CF – možno využiť, podmienené skoky (podmienečné vykonanie kódu)
- inštrukcie `inc`, `dec` neovplyvňujú CF
 - často počet iterácií slučiek (32b hodnota postačuje pre väčšinu aplikácií)
 - podmienka detegovaná CF detegovateľná aj pomocou ZF (nastavenie CF redundantné)
 - ak `ECX = FFFFFFFFH` a vykoná sa


```
inc     ECX
```
 - očakávame $CF = 1$, ale rovnako možno detegovať túto podmienku pomocou ZF ($ECX = 0$)

Preplnenie (overflow flag)

- obdoba CF pre operácie s číslami so znamienkom (signed)
- indikácia výsledku operácie mimo platného rozsahu
- rozsahy čísel so znamienkom [1]

Size (bits)	Range
8	-128 to +127
16	-32 768 to +32 767
32	-2 147 483 648 to +2 147 483 647

Príklad:

```

mov     AL, 72H
add     AL, 0EH           (114 + 14 = 128, OF = 1)

```

- 128 (80H) je korektný výsledok súčtu bezznamienkových čísel
- pri znamienkovej interpretácii nekorektný: 80H predstavuje -128

Znamienková/bezznamienková interpretácia

- ako systém rozpozná spôsob interpretácie reťazca bitov programom? (nijako)
- procesor zohľadňuje obidve interpretácie – podľa toho nastavuje CF a OF

```

mov     AL, 72H
add     AL, 0EH           (114 + 14 = 128: CF = 0, OF = 1)

```

- zohľadnenie príslušného bitu je úlohou programátora
- inštrukcie podmienených skokov: `jo` (ak OF = 1), `jno` (ak OF = 0)
- inštrukcia SW prerušenia: `into` (interrupt on overflow, generuje INT 4)

Znamienko (sign flag)

- znamienko výsledku operácie
- užitočný len pri znamienkovej interpretácii
- kópia najvyššieho bitu výsledku

Príklad:

```
mov    EAX, 15
add    EAX, 97    (15 + 97 = 112, SF = 0)

mov    EAX, 15
sub    EAX, 97    (15 - 97 = -82, SF = 1)
```

```
15 + (-97):    00001111 (15)
                10011111 (-97, c-repr.)
                -----
                10101110 (-82, c-repr.)
```

- inštrukcie podmienených skokov: `js` (ak $SF = 1$), `jns` (ak $SF = 0$)
- použitie
 - znamienko výsledku
 - slučky s hodnotou riadiacej premennej klesajúcou k nule (vrátane) [1]

```
for (i=M downto 0)
    <loop body>
end for
```

```
    mov    ECX, M
for_loop:
    ...
    <loop body>
    ...
    dec    ECX
    jns    for_loop
```

Pomocný prenos (auxiliary carry flag)

- prenos z (výpožička do) nižších 4 bitov (nibble) operandu

```

mov     AL, 43           00101011 (43)
add     AL, 94 (AF = 1)  01011110 (94)
                        -----
                        10001001 (137)

```

```

mov     AL, 43           00101011 (43)
add     AL, 84 (AF = 0)  01010100 (84)
                        -----
                        01111111 (127)

```

- príbuzné inštrukcie
 - neexistencia podmienených skokov s testom AF
 - aritmetické operácie s číslami v BCD formáte využívajú AF
 - aaa, aas, aam, aad (ASCII adjust for addition, subtraction, ...) [3]
 - daa, das (decimal adjust for addition, ...)

Parita (parity flag)

- parita 8-bit výsledku operácie (iba spodných 8 bitov má vplyv na PF)
- párny počet 1 (PF = 1), nepárny (PF = 0)

```

mov     AL, 53           00110101 (53)
add     AL, 89 (PF = 1)  01011001 (89)
                        -----
                        10001110 (142)

```

- príbuzné inštrukcie – skoky: j_p (ak PF = 1), j_{np} (ak PF = 0)
- použitie (napr. kódovanie údajov)

Príklad: prenos cez modem využívajúci 7-bit ASCII kód [1]

- detekcia jednoduchých chýb prenosu – prídanie paritného bitu (k 7-bit údaju)
- predpokladáme kódovanie párnej parity (doplnenie 8.bitu podľa potreby)
- prijímač spočíta počet jednotiek v prijatom bajte (chyba, ak obsahuje nepárny počet)
 - A – 41H (kód: 01000001, MSb – 0)
 - C – 43H (kód: 11000011, MSb – 1, nastavený)

```
parity_encode PROC
    shl     AL
    jp      parity_zero
    stc          ; CF=1
    jmp     move_parity_bit
parity_zero:
    clc          ; CF=0
move_parity_bit:
    rcr     AL
parity_encode ENDP
```

Príklad: vplyv realizácie aritmetických operácií na príznaky [1]

	Code	AL	CF	ZF	SF	OF	PF
Example1	mov AL, -5 sub AL, 123	80H	0	0	1	0	0
Example2	mov AL, -5 sub AL, 124	7FH	0	0	0	1	0
Example3	mov AL, -5 add AL, 132 add AL, 1	7FH 80H	1 0	0 0	0 1	1 1	0 0
Example4	sub AL, AL	00H	0	1	0	0	1
Example5	mov AL, 127 add AL, 129	00H	1	1	0	0	1

Aritmetické inštrukcie

- sčítanie (add, adc, inc) a odčítanie (sub, sbb, dec, neg, cmp)
- násobenie a delenie (mul, imul, div, idiv)
- príbuzné inštrukcie (cbw, cwd, cdq, cwde, movsx, movzx)
- inštrukcia neg – zmena znamienka operandu (IF $dst = 0$ THEN $CF \leftarrow 0$; ELSE $CF \leftarrow 1$; $dst = -dst$)

neg dst (dst – 8, 16, 32-bit GPR, pamäť)

Inštrukcie pre násobenie

- vlastnosti operácie násobenia
 - rozmer výsledku ($2n$ bitov pri násobení dvoch n -bit. čísel)
 - násobenie čísel so znamienkom realizované odlišne od násobenia čísel bez znamienka (dôsledok – odlišné inštrukcie mul/imul)
- násobenie čísel bez znamienka (mul)
 - syntax

mul src (src – 8, 16, 32-bit GPR, pamäť)

- sémantika (podľa rozmeru src)
 - 8 bitov: $AX \leftarrow src * AL$
 - 16 bitov: $DX:AX \leftarrow src * AX$
 - 32 bitov: $EDX:EAX \leftarrow src * EAX$
- inštrukcia ovplyvňuje všetky stavové (6) príznaky, nastavuje však len CF a OF, zvyšné nedefinované
 - CF a OF nastavené, ak vrchná (upper half) časť výsledku je nenulová (AH, DX, EDX)

Príklad:

mov	AL, 10		mov	AL, 10	
mov	DL, 25		mov	DL, 26	
mul	DL	; CF = OF = 0	mul	DL	; CF = OF = 1

- násobenie čísel so znamienkom (`imul`)
 - syntax (podobne ako `mul`, podpora ďalších formátov, napr. bezprostredný údaj ako parameter)
 - CF, OF – nastavené, ak vrchná časť výsledku nie je znamienkovým rozšírením spodnej

Príklad: znamienkové rozšírenie hodnoty -66

$$\begin{aligned} (-66)_{10} &= (10111110)_2 && 8\text{-bit} \\ (-66)_{10} &= (1111111110111110)_2 && 16\text{-bit} \end{aligned}$$

Príklad:

```

mov     DL, 0FFH      ; DL = -1
mov     AL, 42H       ; AL = 66
imul    DL            ; AX = -66      (1111111110111110)2, CF = OF = 0
  
```

Inštrukcie pre delenie

- vlastnosti operácie delenia
 - výsledkom sú dve hodnoty – podiel a zvyšok
 - pri násobení (výsledok s dvojnásobnou dĺžkou) pretečenie sa nevyskytuje, pri delení sa môže vyskytnúť (divide overflow)
- syntax

```

div     src           (bezznamienkové, src – 8, 16, 32-bit GPR, pamäť)
idiv    src           (znamienkové)
  
```

- sémantika inštrukcie `div` (podľa rozmeru deliteľa, `src`)
 - 8 bitov: $AL \leftarrow \text{quot}(AX/src), AH \leftarrow \text{rem}(AX/src)$
 - 16 bitov: $AX \leftarrow \text{quot}(DX:AX/src), DX \leftarrow \text{rem}(DX:AX/src)$
 - 32 bitov: $EAX \leftarrow \text{quot}(EDX:EAX/src), EDX \leftarrow \text{rem}(EDX:EAX/src)$
- príznaky – ovplyvnené inštrukciami – nedefinované
- sémantika inštrukcie `idiv` – rovnaký formát a správanie ako `div`
 - komplikácia – ak delenec je záporný – potrebné znamienkové rozšírenie

Príklad: delenie –251/12 (16-bit)

- (–251) = FF14H, preto DX inicializovaný na FFFFH
- ak DX inicializovaný na 0000H (ako pri `div`), DX:AX reprezentuje kladné číslo!
- ak delenec kladný – DX má byť 0000H
- inštrukcie pre znamienkové rozšírenie
 - `cbw` (convert byte to word) – rozšírenie AL do AH (8-bit `idiv`)
 - `cwd` (convert word to doubleword) – rozšírenie AX do DX (16-bit `idiv`)
 - `cdq` (convert doubleword to quadword) – rozšírenie EAX do EDX (32-bit `idiv`)
- ďalšie príbuzné inštrukcie
 - `cwde` – znamienkové rozšírenie AX do EAX
 - `movsx dst, src` (move sign-extended src to dst)
 - `dst` – R, `src` – R/M, ak `src` 8-bit → `dst` 16-bit alebo 32-bit, ak `src` 16-bit → `dst` 32-bit
 - `movzx dst, src` (move zero-extended src to dst)
 - ako `movsx`

Príklad: 16-bit. znamienkové delenie

```
mov     AX, -5147
cwd                      ; DX = FFFFH
mov     CX, 300
idiv    CX               ; AX = FFEFH (-17) podiel, DX = FFD1H (-47) zvyšok
```

Použitie posunov pre násobenie a delenie

- efektívna alternatíva pre realizáciu týchto operácií; ak je to vhodné, možno využiť (násobenie/delenie mocninou 2)

Príklad: $AX * 32$, 2 alternatívy (b – rýchlosť, miesto)

a) `mov CX, 32`
 `mul CX`

b) `sal AX, 5`

Aritmetické operácie nad viacslovnými údajmi (multi-word arithmetic)

- aritmetické inštrukcie pracujú s údajmi o veľkosti 8, 16, 32-bit (rozmernejšie údaje – problém)
- základy viacslovnej aritmetiky (sčítanie, odčítanie, násobenie, delenie)

Sčítanie a odčítanie (64-bit, unsigned)

- relatívne nenáročné operácie, sčítanie – sčítame pravých 32 bitov, v ďalšom kroku ľavých (s prenosom z prvého kroku)

Příklad: sčítanie dvoch 64-bit čísel v EBX:EAX a EDX:ECX, výsledok v EBX:EAX. Pretečenie indikované pomocou CF.

```
add64:  add    EAX,ECX      ; odčítanie - podobne (add→sub, adc→sbb)
        adc    EBX,EDX
        ret
```

Násobenie (64-bit, unsigned)

- viacero známych prístupov, uvádzaný algoritmus – tzv. longhand multiplication [1]
- testujú sa bity násobiteľa (A) sprava a podľa ich hodnoty sa (ne)pripočítava násobenec (B)
- činnosť algoritmu (4-bit čísla A = 13, B = 5)

```
P:=0
A:=multiplier
B:=multiplicand
count:=64
while (count>0)
  if (LSB of A = 1)
    then
      P:=P+B
      CF:=carry generated by P+B
    else
      CF:=0
    end if
    shift right CF:P:A by one bit position
    {LSB of multiplier is not used in the rest of the algorithm}
    count:=count-1
  end while
```

	After P+B			After the shift		
	CF	P	A	CF	P	A
init.state	?	0000	1101	-	-	-
iterat. 1	0	0101	1101	?	0010	1110
iterat. 2	0	0010	1110	?	0001	0111
iterat. 3	0	0110	0111	?	0011	0011
iterat. 4	0	1000	0011	?	0100	0001

- implementácia (dve 64-bit čísla v EBX:EAX (A) a EDX:ECX (B), 128-bit výsledok: EDX:ECX:EBX:EAX)

```
%define COUNT    word[EBP-2] ; lokalna premenna
mult64:
    enter    2,0                ; 2-byte
    push     ESI
    push     EDI
    mov      ESI,EDX             ; ESI:EDI = B
    mov      EDI,ECX
    sub      EDX,EDX             ; P = 0
    sub      ECX,ECX
    mov      COUNT,64           ; 64-bit cislo
step:
    test     EAX,1               ; LSB A je 1?
    jz       shift1             ; ak nie, nepricitaj
    add      ECX,EDI             ; P = P + B
    adc      EDX,ESI
```

```
shift1:
    rcr      EDX,1
    rcr      ECX,1
    rcr      EBX,1
    rcr      EAX,1

    dec      COUNT
    jnz      step
    pop      EDI
    pop      ESI
    leave
    ret
```

- mapovanie premenných:
 - ESI:EDI – B, EBX:EAX – A, EDX:ECX – P
- lokálna premenná COUNT
- 64-bit sčítanie (P + B)
- posun vpravo CF:P:A (rcr)
 - inštrukcia test nuluje CF

```
P:=0
A:=multiplier
B:=multiplicand
count:=64
while (count>0)
    if (LSB of A = 1)
        then
            P:=P+B
            CF:=carry generated by P+B
        else
            CF:=0
        end if
        shift right CF:P:A by one bit position
        {LSB of multiplier is not used in the rest of the algorithm}
        count:=count-1
    end while
```

Delenie (n -bit unsigned)

- niekoľko známych algoritmov ('nonrestoring' division algorithm) [1]
- A – delenec, B – deliteľ, $A \leftarrow \text{quot}(A/B)$, $P \leftarrow \text{rem}(A/B)$
- uvádzame iba pseudokód (bez asm-implementácie)
- pomocný register (P) dĺžky $n+1$ bitov
- metóda
 - testuje sa znamienko P , podľa neho $P \leftarrow P + B$, alebo $P \leftarrow P - B$
 - $P:A$ posunutý vľavo s manipuláciou LSb A
 - po n -opakovaniach, podiel v A , zvyšok v P

Příklad: činnosť algoritmu (4-bit. čísla $A = 0101$ (5), $B = 0010$ (2); pomocný register $P = 00000$ (0))

	P:A	P:A (shift left)	$P \pm B:A$
count = 4	00000:0101	00000:1010	11110:1010
count = 3	11110:1010	11101:0100	11111:0100
count = 2	11111:0100	11110:1000	00000:1000
count = 1	00000:1001	00001:0010	11111:0010
			00001:0010

$-1 = (11111)_c$ výsledok: $A = 2$ (podiel)
 $-2 = (11110)_c$ $P = 1$ (zvyšok)
 $-3 = (11101)_c$

```

P:=0
A:=dividend
B:=divisor
count:=64
while(count>0)
  if(P is negative)
    then
      shift left P:A by one bit position
      P:=P+B
    else
      shift left P:A by one bit position
      P:=P-B
    end if
  if(P is negative)
    then
      set low-order bit of A to 0
    else
      set low-order bit of A to 1
    end if
  count:=count-1
end while
if(P is negative)
  P:=P+B
end if

```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] Cloutier, F.: [x86 and amd64 instruction reference](#), 2022.

Skoky a iterácie

- podmienené a nepodmienené skoky
- iterácie (LOOP)
- implementácia riadiacich štruktúr HL jazykov

Nepodmienený skok

- nepodmienený prenos riadenia na špecifikovaný cieľ, syntax:

```
jmp    cieľ
```

Špecifikácia cieľa

- cieľová adresa špecifikovaná
 - priamo (časť inštrukcie, dopredu/späť – znamienková reprezentácia)
 - nepriamo (register/pamäť obsahuje adresu)
- priame skoky (direct jumps)
 - adresa špecifikovaná v inštrukcii – relatívny posun medzi cieľom a inštrukciou nasledujúcou za `jmp` (!)
 - po výbere inštrukcie `jmp`, EIP automaticky aktualizovaný
 - posun – číslo so znamienkom (kladné – skok vpred)
 - relatívne adresy – vhodné pre dynamicky relokovateľný kód (position-independent code)
- cieľ skoku
 - v rámci segmentu – cieľ v rovnakom segmente ako inštrukcia `jmp` (intra-segment jump)
 - doposiaľ uvažovaný tento typ
 - $EIP \leftarrow EIP + \text{rel. posun}$

- v inom segmente (intersegment jump, far jump)
 - $CS \leftarrow \text{cieľ. segment}$
 - $EIP \leftarrow \text{cieľ. offset}$
 - segment aj offset špecifikované v rámci inštrukcie (pre 32-bit. segment inštrukcia – 7B)
- väčšina skokov – v rámci segmentu, 2 spôsoby špecifikácie cieľa podľa veľkosti rel. posunu [4]
 - short jump (2B, 1B – op. kód (EBH) + 1B – rel. posun, znamiekové číslo v rozsahu –128/+127)
 - near jump (3/5B, 1B – op. kód (E9H) + 2/4B – rel. posun)
 - 2B rel. posun pre 16-bit. segmenty, 4B pre 32-bit segmenty
- špecifikácia krátkych skokov (SHORT)
 - chceme použiť krátky skok – informácia pre prekladač:

```
jmp      SHORT ECX_init_done
```

- ak cieľ je vzdialenejší – chybové hlásenie
- assembler automaticky doplní SHORT pre skoky vzad (ak je cieľ v platnom rozsahu)
- skoky vpred – assembler nepozná vzdialenosť cieľa, pomoc programátora vítaná

Príklad: kódovanie short/near skokov (adaptované [1])

- r.8 (2B) – špecifikovaný ako SHORT, op. kód EBH, offset 14H
- r.10 (5B) – špecifikovaný ako NEAR, op. kód E9H, offset 0000000AH
- r.18 (2B) – skok späť, assembler dokáže rozhodnúť, že SHORT stačí (FDH = -3)
- r.13 (5B) – NEAR, malý endian, offset 00000017H

```

7                                     ...
8 00000000 EB14                      ; unconditional jumps encoding
9 00000002 B978563412                jmp     SHORT ECX_init_done
10 00000007 E90A000000                mov     ECX,12345678H
11                                     jmp     NEAR ECX_init_done
12                                     init_ECX:
13 0000000C B912EFCDA8                mov     ECX,0ABCDEF12H
14 00000011 E917000000                jmp     NEAR near_jump
15                                     ECX_init_done:
16 00000016 89C8                      mov     EAX,ECX
17                                     repeat1:
18 00000018 49                        dec     ECX
19 00000019 EBFD                      jmp     repeat1
20                                     ...
21                                     jmp     short_jump
22 00000021 EB05                      mov     ECX,0FF00FFFFH
23 00000023 B9FFFF00FF                short_jump:
24                                     mov     EDX,98765432H
25                                     near_jump:
26 00000028 BA32547698                jmp     init_ECX
27 0000002D EBDD
28                                     ...
29                                     jmp     init_ECX
30                                     ...
```

- nasm – možnosť generovať tzv. listing file pomocou voľby -l, napr.

```
nasm -f win32 L5_jumps_encoding.asm -l L5_jumps_encoding.lst
```

Inštrukcia porovnania (*cmp*)

- nastavenie príznakov, nasledujúca inštrukcia podmieneného skoku ich otestuje (bolo prediskutované, prednáška 4)
- implementácia HL konštrukcie IF-THEN-ELSE v asembleri v dvoch krokoch
 - aritmetická/porovnávacia inštrukcia
 - podmienený skok

Podmienené skoky

- možno rozdeliť do troch skupín
 - podľa hodnoty jedného príznaku
 - podľa výsledku bezznamienkových porovnaní
 - podľa výsledku znamienkových porovnaní

Skoky podľa hodnoty jedného príznaku

- dve inštrukcie (0/1) pre každý stavový príznak okrem AF
- výskyt dvojitého pomenovaní (alias) pre ZF, PF
- príznak nuly (ZF)
 - *jz*, *je* (ZF = 1)
 - *jnz*, *jne* (ZF = 0)
 - *jecxz* (jump if ECX = 0, bez testovania príznakov), *jcxz* (if CX = 0)
- prenos (CF)
 - *jc* (CF = 1), *jnc* (CF = 0)
- preplnenie (OF)
 - *jo* (OF = 1), *jno* (OF = 0)
- znamienko (SF)
 - *js* (SF = 1), *jns* (SF = 0)
- parita (PF)
 - *jp*, *jpe* (PF = 1)
 - *jnp*, *jpo* (PF = 0)

Skoky podľa výsledku bezznamienkových porovnaní

- pri porovnaní dvoch čísel (`cmp num1, num2`) – znamienkové alebo bezznamienkové čísla?

Príklad: AL = 10110111 (183/−73)₁₀, DL = 01101110 (110)₁₀

`cmp AL, DL`

AL > DL (bezznamienková interpretácia)

AL < DL (znamienková interpretácia)

- poradie v porovnaní (`cmp num1, num2`) – vždy vzťah num1 k num2, možné relácie (6):
 - num1 = num2, num1 ≠ num2
 - num1 > num2, num1 ≥ num2
 - num1 < num2, num1 ≤ num2
- pre čísla bez znamienka CF a ZF relevantné, výskyt synonym (aliases)

<i>mnemonika</i>	je/jz	jne/jnz	ja/jnbe	jae/jnb	jb/jnae	jbe/jna
<i>význam</i>	equal/ zero	not equal/ not zero	above/ not below or equal	above or equal/ not below	below/ not above or equal	below or equal/ not above
<i>podmienka</i>	ZF = 1	ZF = 0	CF = 0 AND ZF = 0	CF = 0	CF = 1	CF = 1 OR ZF = 1

Skoky podľa výsledku znamienkových porovnaní

- porovnania $=$, \neq pracujú rovnako na číslach so znamienkom i bez znamienka
- pre čísla so znamienkom relevantné SF, OF, ZF

<i>mnemonika</i>	je/jz	jne/jnz	jb/jnle	jge/jnl	jl/jnge	jle/jng
<i>význam</i>	equal/ zero	not equal/ not zero	greater/ not less or equal	greater or equal/ not less	less/ not greater or equal	less or equal/ not greater
<i>podmienka</i>	ZF = 1	ZF = 0	ZF = 0 AND SF = OF	SF = OF	SF \neq OF	ZF = 1 OR SF \neq OF

- predpokladajme inštrukciu `cmp snum1, snum2`; 8-bitové operandy [1]:
 - podmienky pre `snum1 > snum2` (`jb`)

snum1	snum2	ZF	OF	SF
56	55	0	0	0
56	-55	0	0	0
-55	-56	0	0	0
55	-75	0	1	1

- podmienky pre `snum1 < snum2` (`jl`, ZF – redundantný, ZF = 1 \rightarrow SF = OF = 0)

snum1	snum2	ZF	OF	SF
55	56	0	0	1
-55	56	0	0	1
-56	-55	0	0	1
-75	55	0	1	0

Inštrukcia setCC (Set Byte on Condition) [6]

- nastaví `dest` na 1 ak je splnená podmienka CC, ináč nastaví `dest` na 0

`setCC dest (dest - 8-bit register, pamäť)`

- podmienka CC reprezentuje jednotlivé príznaky, ako aj bezznamienkové/znamienkové porovnania (SETZ, SETNZ, SETA, SETG, ...)

Vzdialenosť cieľa podmienených skokov

- podmienené skoky – SHORT/NEAR [5]
- najefektívnejšie, ak sú kódované ako 2B inštrukcie (SHORT – 1B op. kód + 1B offset)
 - rozsah –128/127 B (SHORT)
 - pri prekročení tejto vzdialenosti – možnosť náhrady (negácia podmienky + nepodmienený skok)
- podmienené skoky NEAR – op. kód 2B + offset 2/4B

Priklad: kód vľavo – možnosť náhrady kódom vpravo [1]

```
target1:    ...
            ...
            cmp     AX,BX
            je      target1 ; not a short jump
            mov     CX,20
            ...
```

```
target1:    ...
            ...
            cmp     AX,BX
            jne     skip1 ; skip1 is a short jump
            jmp     target1
skip1:      mov     CX,20
            ...
```

Iterácie

- inštrukcie iterácií využívajú CX/ECX register (počet opakovaní) podľa veľkosti operandu (v ďalšom predpokladáme 32-bit)
- dekrementujú register pred testom na nulu (bez ovplyvnenia príznakov)
- vzdialenosť cieľa v dosahu –128/127 B (1B posun)

Inštrukcie loop, loope/loopz, loopne/loopnz

- synonymá (aliases), syntax

```
loop    cieľ
loope   cieľ
loopne  cieľ
```

- inštrukcie loope/loopz, loopne/loopnz – podpora cyklov s dvoma podmienkami terminácie

<i>mnemonika</i>	loop	loope/loopz	loopne/loopnz
<i>význam</i>	loop	loop while equal/ loop while zero	loop while not equal/ loop while not zero
<i>sémantika</i>	ECX = ECX – 1 IF ECX ≠ 0 skok na <i>cieľ</i>	ECX = ECX – 1 IF (ECX ≠ 0 AND ZF = 1) skok na <i>cieľ</i>	ECX = ECX – 1 IF (ECX ≠ 0 AND ZF = 0) skok na <i>cieľ</i>

Příklad: program číta z klávesnice čísla, končí po zadaní stanoveného počtu čísel (SIZE), alebo nuly

<pre>%include "asm_io.inc" SIZE EQU 10 segment .bss buffer resd SIZE segment .text global _asm_main _asm_main: enter 0,0 pusha mov EBX,buffer mov ECX,SIZE</pre>	<pre>read_more: call read_int mov [EBX],EAX add EBX,4 cmp EAX,0 loopne read_more popa mov EAX,0 leave ret</pre>
---	---

- problém: ak na začiatku ECX = 0 (FFFFFFFFH opakovaní / nula na vstupe), riešenie: inštrukcia `jecxz` pred vstupom do slučky
- rýchlosť vykonávania inštrukcií `loop` a `jcxz` (pre potreby optimalizácie, údaje platné pre procesor Pentium) [3]
 - dvojica inštrukcií (spolu 2 hod. cykly) sa vykoná rýchlejšie ako zodpovedajúca (`loop cieľ`, 5/6 cyklov)

```
dec        ECX
jnz        cieľ
```
 - dvojica inštrukcií (spolu 2 hod. cykly) sa vykoná rýchlejšie ako zodpovedajúca (`jecxz cieľ`, 5/6 cyklov)

```
cmp        ECX,0
jz         cieľ
```
- určenie časovania inštrukcií pre rôzne x86/x86-64 procesory náročnejšie, detailné informácie napr. v [9]

Implementácia riadiacich štruktúr HL jazykov

- s využitím inštrukcií porovnania, skokov a iterácií, príklady inšpirované [1]
- gcc – možnosť generovať asm kód pomocou voľby -S, napr. `gcc -S -masm=intel L5_hl_constr_1.c -m32`

Konštrukcia **if-then-else**

```
if (condition)
then
    true-alternative
else
    false-alternative
end if
```

Príklad: konštrukcia if a relačný operátor (C kód – priradí väčšiu z dvoch hodnôt (typ int) premennej bigger)

a) kód v jazyku C

```
if(value1 > value2)
    bigger = value1;
else
    bigger = value2;
```

b) po preklade (GCC: (GNU) 9.1.0)

```
...
mov     eax,value1      ; value1
cmp     eax,value2      ; value2
jle     L2
mov     eax,value1      ; redundant
mov     bigger,eax      ; then part
jmp     L1
L2:
mov     eax,value2      ; else part
mov     bigger,eax
L1:
...
```

- podmienka testovaná pomocou `cmp/jle`
- generovaný redundantný kód

Príklad: konštrukcia `if` a logický operátor (`&&`) – (test, či sa jedná o malé písmeno a prípadný prevod na veľké) [1]

a) kód v jazyku C

```
if((ch >= 'a') && (ch <= 'z'))  
    ch = ch - 32;
```

b) po preklade (GCC: (GNU) 9.1.0)

```
...  
    cmp     byte_ch,96      ; 'a' = 97  
    jle     L4              ; not lower case  
    cmp     byte_ch,122     ; 'z'  
    jg      L4              ; not lower case  
    movzx   eax,byte_ch     ; lower case  
    sub     eax,32  
    mov     byte_ch,al  
L4:  
    ...
```

- zložená podmienka – dva páry `cmp/jx` inštrukcií
- generovaný redundantný kód

Iteratívne konštrukcie

- konštrukcie ako `while`, `repeat-until`, `for`

Slučka `while`

- test podmienky pred vykonaním tela slučky (entry-test loop)
- telo slučky sa vykonáva opakovane, pokiaľ podmienka je splnená

a) kód v jazyku C

```
while(total < 700)
{
    <loop body>
}
```

b) po preklade (GCC: (GNU) 9.1.0)

```
...
L3:    jmp     L2          ; test condition
      <while loop body>    ; loop body
...
L2:    cmp     total,699    ; while condition
      jle     L3
      ...
```

- nepodmienený prenos riadenia na začiatku (test podmienky)

Slučka repeat-until

- podmienka testovaná po vykonaní tela slučky (exit-test loop)
- príkazy v tele teda vykonané aspoň raz

a) kód v jazyku C

```
do
{
    <loop body>
}
while (number > 0);
```

b) po preklade (GCC: (GNU) 9.1.0)

```
L2:    ...
      <do-while loop body> ; loop body
      ...
      cmp     number,0     ; cond. test
      jg      L2
      ...
```

Slučka for

- pevne daný počet iterácií (counting loop)

a) kód v jazyku C

```
for(i = 0; i < SIZE; i++)  
{  
    <loop body>  
}
```

b) po preklade (GCC: (GNU) 9.1.0)

```
    ...  
    mov     var_i, 0           ; i = 0  
    jmp     L2  
L3:    <for loop body>         ; loop body  
    ...  
    add     var_i, 1           ; increment i  
L2:    cmp     var_i, SIZE-1    ; for condition  
    jle     L3  
    ...
```

- nepodmienený skok na začiatku (test podmienky)
- zvýšenie i (zníženie podobne)

Nepriame skoky

- doposiaľ inštrukcie priamych skokov (adresa cieľa zakódovaná v inštrukcii samotnej)
- uvažujeme skoky v rámci segmentu
- adresa cieľa špecifikovaná R/M [4]
- špecifikovaná absolútna hodnota posunu (offset, u priamych skokov relatívna)
- použitie

```
jmp     [ECX]
```


Viaccestné vetvenie (multiway conditional execution)

- pri väčšom počte vetiev použitie `if` neefektívne, možnosť zanesenia chyby [1]

Konštrukcia `switch`

a) kód v jazyku C

b) po preklade (GCC: (GNU) 9.1.0)

```
switch(ch)
{
    case '0':
        count[0]++;
        break;
    case '1':
        count[1]++;
        break;
    case '2':
        count[2]++;
        break;
    case '3':
        count[3]++;
        break;
    default:
        count[4]++;
}
```

- inštrukcia `movsx` – znamienkové rozšírenie

```
...
movsx    eax,byte_ch
cmp      eax,51                ; '3'
je       L2
cmp      eax,51
jg       L3
cmp      eax,50                ; '2'
je       L4
cmp      eax,50
jg       L3
cmp      eax,48                ; '0'
je       L5
cmp      eax,49                ; '1'
je       L6
jmp      L3
L5:      mov     eax,count[0]
add      eax,1
mov      count[0],eax
jmp      L7
L6:      mov     eax,count[1]
add      eax,1
mov      count[1],eax
jmp      L7
L4:      mov     eax,count[2]
add      eax,1
mov      count[2],eax
jmp      L7
L2:      mov     eax,count[3]
add      eax,1
mov      count[3],eax
jmp      L7
L3:      mov     eax,count[4]
add      eax,1
mov      count[4],eax
L7:      ...
```

Ďalšie užitočné inštrukcie (miscellaneous instructions)

Inštrukcia LEA (Load effective address) [7]

`lea r,m` (r – 16/32-bit register, m – pamäť)

- vypočíta efektívnu adresu druhého operandu (m) a uloží ju do prvého (r)
- druhý (zdrojový) operand je pamäťová referencia (offset) špecifikovaná niektorým z režimov adresovania

Inštrukcia CPUID (Processor identification) [8]

`cuid`

- umožňuje softvéru zistiť detaily o procesore
- vráti informácie v registroch EAX, EBX, ECX, EDX
- parameter v registri EAX (niekedy aj ECX) určuje kategóriu informácií, ktoré inštrukcia vráti

```
AsmCPUInfo
-----
Basic CPU information based on CPUID instruction
Highest Value for Basic Processor Information: b
Highest Value for Extended Processor Information: 80000008
CPU Vendor ID String: GenuineIntel
Processor Brand: Intel(R) Core(TM) i3 CPU          530  @ 2.93GHz
Selection of supported CPU features:
+ Onboard x87 FPU
+ MMX instructions
+ SSE instructions
+ SSE2 instructions
+ Prescott New Instructions-SSE3 (PNI)
+ Supplemental SSE3 instructions
+ SSE4.1 instructions
+ SSE4.2 instructions
+ AES instruction set
+ Advanced Vector Extensions
- Advanced Vector Extensions 2
- AVX-512 Foundation
- AVX-512 Doubleword and Quadword Instructions
```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] Rafiquzzaman, M.: Microprocessor Theory and Applications with 68000/68020 and Pentium, [Appendix F](#), John Wiley & Sons, Inc., 2008.
- [4] Cloutier, F.: [x86 and amd64 instruction reference](#), [JMP — Jump](#), 2022.
- [5] Cloutier, F.: [x86 and amd64 instruction reference](#), [Jcc — Jump if Condition Is Met](#), 2022.
- [6] Cloutier, F.: [x86 and amd64 instruction reference](#), [SETcc — Set Byte on Condition](#), 2022.
- [7] Cloutier, F.: [x86 and amd64 instruction reference](#), [LEA — Load Effective Address](#), 2022.
- [8] Cloutier, F.: [x86 and amd64 instruction reference](#), [CPUID — CPU Identification](#), 2022.
- [9] Fog, A.: [Instruction tables](#), Technical University of Denmark, 1996 – 2022.

Logické a bitové operácie

- inštrukcie pre realizáciu logických operácií, posuny a rotácie [1], [2]
- logické výrazy a bitové manipulácie
- reťazce

Logické operácie

- bit ako jednotka informácie 0/1 (True/False)
- inštrukcie (`and`, `or`, `not`, `xor`, `test`) pre realizáciu logických operácií
- binárne a unárna (`not`) operácie – nad 8,16,32-bit operandmi
- ovplyvňujú stavové príznaky (okrem CF, OF – 0, AF – nedefinovaný)
- inštrukciám sme sa už venovali, poukážeme na ich typické použitie

inštrukcia `and`

- podpora zložených log. výrazov a bitová operácia log. súčinu HL jazykov (neskôr)
- nulovanie bitov
- izolácia bitov

Nulovanie bitov

- bitové masky (ak bit masky = 0 → výstup: 0, ak bit masky = 1 → výstup: kópia druhého vstupného bitu)

AL = 11010110 (operand)

BL = 11111100 (maska)

and AL,BL = 11010100

Príklad: konverzia ASCII znakov na čísla [1]

- relácia medzi ASCII kódom a binárnou reprezentáciou čísel 0 – 9
- maskovanie horných 4 bitov (AL): `and AL, 0FH`

Decimal digit	8-bit binary	ASCII code
0	0000 0000	0011 0000
1	0000 0001	0011 0001
2	0000 0010	0011 0010
3	0000 0011	0011 0011
4	0000 0100	0011 0100
5	0000 0101	0011 0101
6	0000 0110	0011 0110
7	0000 0111	0011 0111
8	0000 1000	0011 1000
9	0000 1001	0011 1001

Izolácia bitov

- odmaskovanie všetkých ostatných bitov

Príklad: párne/nepárne číslo? [1]

- test (LSb = 1 – nepárne)

```
    and AL, 1      ; mask
    jz  even_num
odd_num:
    ...
    <code for odd number>
    ...
even_num:
    ...
    <code for even number>
    ...
```

inštrukcia or

- podpora zložených log. výrazov a bitová operácia log. súčtu HL jazykov (neskôr)
- nastavenie bitov (ak bit masky = 0 → výstup: kópia druhého vstupného bitu, ak bit masky = 1 → výstup: 1)

```
AL = 11010110 (operand)
BL = 00000011 (maska)
-----
or AL,BL = 11010111
```

Priklad: konverzia čísel (8-bit bez znamienka, 0-9) na ASCII znaky

- nastavenie bitov b₄, b₅ bez zmeny ostatných (maska 00110000)

```
or    AL, 30H
```

Vystrihnutie (cut) a vkladanie (paste) bitov

- nový bajt v AL – kombinácia nepárnych bitov z AL a párnych z BL

```
and    AL, 55H           ; nepárne bity z AL
and    BL, 0AAH          ; párne bity z BL
or     AL, BL            ; spojenie
```

inštrukcia xor

- podpora zložených log. výrazov HL jazykov (neskôr)
- preklopenie (toggle) hodnoty bitu/bitov
- inicializácia registrov (0)

Preklopenie hodnoty bitov

- maska má hodnotu 1 na pozícii, kde sa má hodnota preklopiť
- použitie xor druhý krát – pôvodná hodnota

Príklad: jednoduché kódovanie údajov

- kľúč pre kódovanie ako maska pre inštrukciu `xor`

```
xor  AL,26H           ; kľúč 26H
```

- dekódovanie – rovnaký proces nad zakódovaným údajom

kódovanie:	01000010 (znak B, 42H)	dekódovanie:	01100100 (znak d)
	00100110 (maska, 26H)		00100110 (maska)
	-----		-----
	01100100 (znak d, 64H)		01000010 (znak B)

Inicializácia registrov

- `mov` – viac miesta v pamäti

```
mov  EAX,0           (B800000000)
xor  EAX,EAX         (31C0, ovplyvňuje príznaky)
```

inštrukcia `not`

- podpora zložených log. výrazov HL jazykov (neskôr)
- negácia (complement) bitov operandu
- zmena znamienka operandu – inštrukcia `neg`

inštrukcia `test`

- logický ekvivalent inštrukcie `cmp` (log. súčin bez zmeny cieľového operandu – nedeštruktívny `and`) [1]
- za účelom nastavenia príznakov, často nasleduje podmienený skok

```
test AL,1
jz   even_num
odd_num:
...
even_num:
...
```

Posuny

- dva typy posunov
 - logické (čísla bez znamienka) – `shl`, `shr`
 - aritmetické (čísla so znamienkom) – `sal`, `sar`
- stavové príznaky
 - `AF` – nedefinovaný
 - `ZF`, `PF` – podľa výsledku operácie, `CF` – posledný bit vysunutý z operandu
 - `OF` – nedefinovaný pre viac-bitové posuny
 - posuny o 1 bit – `OF` = 1 pri zmene bitu znamienka, ináč `OF` = 0

Inštrukcie logických posunov

- bitové manipulácie

Příklad: iné kódovanie – výmena hornej a dolnej polovice bajtu (obnova údaj – druhá aplikácia)

```
mov  AH,AL      ; AL obsahuje bajt pre kódovanie
shl  AL,4        ; shl/shr - na uvoľnené pozície vstupujú 0
shr  AH,4
or   AL,AH
```

- násobenie a delenie čísel bez znamienka mocninou 2 [1]
 - dvojnásobok/polovica čísla bez znamienka (všeobecne – mocnina 2)
 - delenie – celočíselné (prípadná desatinná časť zahodená)

Příklad: 28, 168

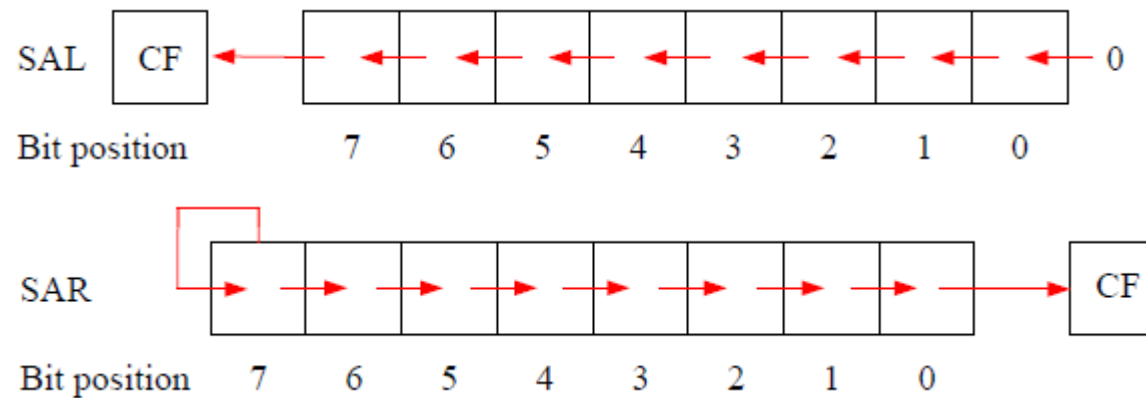
Binary	Decimal
00011100	28
00111000	56
01110000	112
11100000	224
10101000	168
01010100	84
00101010	42
00010101	21

Inštrukcie aritmetických posunov

- syntax

```
sal  dest,count      sar  dest,count
sal  dest,CL         sar  dest,CL
```

- sémantika [1]



Zdvojnásobenie čísel so znamienkom

- posun vľavo o 1 bit (MSb – znamienko – nepredstavuje problém) [1]
 - znamienkové rozšírenie (na väčší počet bitov ako je potrebné na reprezentáciu čísla)
 - žiadny rozdiel (v operácii) v porovnaní s číslom bez znamienka
 - nie je potrebná špeciálna inštrukcia (sal je alias pre shl)

Signed binary	Decimal
00001011	+11
00010110	+22
00101100	+44
01011000	+88
11110101	-11
11101010	-22
11010100	-44
10101000	-88

Delenie čísel so znamienkom na polovicu

- uvoľnený ľavý bit – potrebné nahradiť kópiou znamienkového [1]
- potrebná špeciálna inštrukcia na tento účel – `sar`
- posuny – efektívnejšie ako zodpovedajúce inštrukcie násobenia/delenia

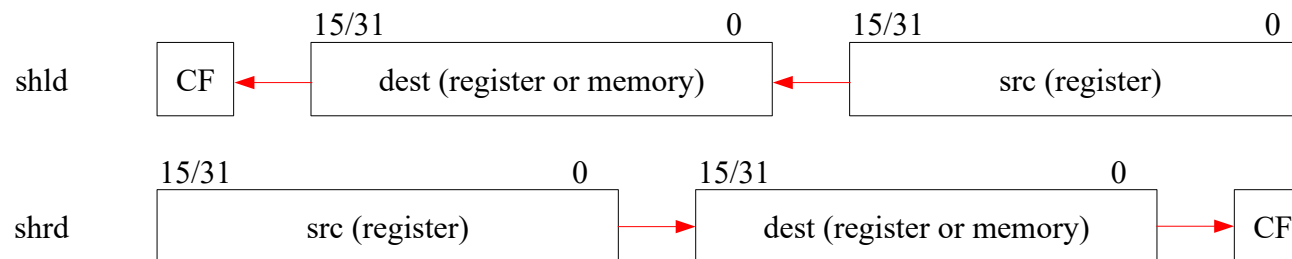
Signed binary	Decimal
01011000	+88
00101100	+44
00010110	+22
00001011	+11
10101000	-88
11010100	-44
11101010	-22
11110101	-11

Dvojité posuny

- dve inštrukcie pre 32 a 64-bit posuny
- syntax (`count` – bezprostredný alebo `CL`)

```
shld dest,src,count      ; dest/src - word, doubleword
shrd dest,src,count      ; dest - R/M, src - R
```

- rozdiel v porovnaní s posunmi – bity vysunuté zo `src` vstupujú do `dst` (`src` – bez modifikácie) [1]



Rotácie

- posuny – vysunuté bity stratené (nie vždy žiadúce)
- inštrukcie už ozrejmene (prednáška 2) – typické použitie

Rotácie bez CF (**rol**, **ror**)

- preusporiadanie bitov v bajte, slove, dvojslove

Příklad: kódovanie – výmena hornej a dolnej polovice bajtu (s využitím `rol/ror` – jednoduchšie)

```
mov  CL, 4
ror  AL, CL      ; podobne rol  AL, CL
```

Rotácie cez CF (**rcl**, **rcr**)

- CF ako vstup

Příklad: posuny 64-bit čísel (násobenie 64-bit čísla (EDX:EAX) bez znamienka 16-timi, delenie analogicky)

a) s využitím rotácií

```
mov  ECX, 4
shift_left:
shl  EAX, 1      ; MSb EAX → CF
rcl  EDX, 1      ; CF → LSb EDX
loop shift_left
```

b) s využitím dvojitého posunu

```
shld EDX, EAX, 4 ; EAX bez zmeny
shl  EAX, 4
```

Logické výrazy v HL jazykoch

- reprezentácia booleovskej hodnoty
 - stačí jediný bit, nevýhoda – potreba jeho izolácie
 - väčšina jazykov – využíva bajt (0 – false, ináč – true)

- logické výrazy
 - napr. jazyk C – logické operátory (&& – AND, || – OR)

Príklad: preklad logického výrazu jazyka C (inšpirované [1])

a) kód v jazyku C

b) po preklade (GCC: (tdm-1) 5.1.0)

```
z = ~(x && y) ^ (y || z);
```

- test premennej x (ak je 0, netestuje sa y, presun na L2)
- výsledok (x && y) v EAX
- návěstie L3 – negácia EAX: ~(x && y)
- test premennej y (ak y ≠ 0, netestuje sa z, presun na L4)
- návěstie L6 – operácia xor
- uloženie výsledku do premennej z

Implementácia operácií HL jazykov

- logických – tokom riadenia
- bitových – ekvivalenty v inštrukciách

```
    cmp    var_x,0
    je     L2
    cmp    var_y,0
    je     L2
    mov    eax,1          ; eax - term (x && y) = 1
    jmp    L3
L2:    mov    eax,0          ; eax - term (x && y) = 0
L3:    not    eax          ; ~(x && y)
    cmp    var_y,0
    jne    L4
    cmp    var_z,0
    je     L5
L4:    mov    edx,1          ; edx - term (y || z) = 1
    jmp    L6
L5:    mov    edx,0          ; edx - term (y || z) = 0
L6:    xor    eax,edx       ; ~(x && y) ^ (y || z)
    mov    var_z,eax
```

Bitové manipulácie

- bitové logické operátory jazyka C: and (&), or (|), xor (^), not (~)
- operátory bitových posunov jazyka C: vľavo (<<), vpravo (>>)

Příklad: preklad bitových operácií (premenná `mask` v registri SI) [1]

C statement	ASM code
<code>mask = mask >> 2</code> (right-shift mask by two bit positions)	<code>shr SI, 2</code>
<code>mask = mask << 4</code> (left-shift mask by four bit positions)	<code>shl SI, 4</code>
<code>mask = ~mask</code> (complement mask)	<code>not SI</code>
<code>mask = mask & 85</code> (bitwise and)	<code>and SI, 85</code>
<code>mask = mask 85</code> (bitwise or)	<code>or SI, 85</code>
<code>mask = mask ^ 85</code> (bitwise xor)	<code>xor SI, 85</code>

Vyhodnocovanie logických výrazov

- úplné vyhodnotenie (full evaluation)
 - celý logický výraz je vyhodnotený pred priradením hodnoty výrazu (Pascal)
- čiastočné vyhodnotenie (partial evaluation)
 - výsledok vyhodnotenia možno získať bez vyhodnotenia celého výrazu (jazyk C)
 - používané pravidlá:
 - `cond1 AND cond2` (výsledok `false`, ak jeden zo vstupov je `false`)
 - `cond1 OR cond2` (ak `cond1` je `true`, netreba vyhodnotiť `cond2`)
 - čiastočné vyhodnotenie – efektívnejší kód

Bitové inštrukcie

- test a modifikácia bitu (bit test and modify)
 - 4 inštrukcie, syntax

`bt operand, bit_pos` (operand: 16/32-bit R/M, bit_pos: I/R, LSb – pozícia 0, `bts`, `btr`, `btc` – podobne)

- sémantika (kopíruje bit do CF) [1]

Inštrukcia	Efekt na vybraný bit
<code>bt</code> (Bit Test)	bez efektu
<code>bts</code> (Bit Test and Set)	vybraný bit $\leftarrow 1$
<code>btr</code> (Bit Test and Reset)	vybraný bit $\leftarrow 0$
<code>btc</code> (Bit Test and Complement)	vybraný bit $\leftarrow \neg(\text{vybraný bit})$

```

mov     EAX, 7
bt      EAX, 1    ; CF = 1
bt      EAX, 3    ; CF = 0

```

- prehľadávanie operandu (bit scan)
 - dve inštrukcie, smer prehľadávania (bit scan forward/reverse)
 - prehľadávanie operandu, ak sa nájde bit nastavený na hodnotu 1, vráti pozíciu v registri

`bsf dst_reg, operand` (operand: 16/32-bit R/M, dst_reg: 16/32-bit R, pozícia bitu)
`bsr dst_reg, operand`

- ak všetky bity operand-u nulové: ZF = 1
- ináč: ZF = 0 a `dst_reg` obsahuje pozíciu prvého nájdeného bitu nastaveného na 1
- ostatné stavové príznaky – nedefinované

Práca s reťazcami

Reprezentácia reťazcov

- reprezentácia s pevnou dĺžkou
 - kratšie reťazce – doplnené na zadanú dĺžku
 - dlhšie reťazce – skrátené
 - nevýhody reprezentácie – neefektívne využívanie pamäti (ak sa chceme vyhnúť skracovaniu)
- reprezentácia s premenlivou dĺžkou
 - odstránenie nedostatkov predošlej reprezentácie
 - atribút reťazca udávajúci jeho dĺžku
 - dĺžka explicitne určená

```
string    DB    'Error message'
str_len   DW    $ - string    ; $ - aktuálna pozícia v kóde (location counter)
```

- použitá značka konca reťazca (špeciálny znak, nevyskytuje sa v reťazci, sentinel character)
 - obvyčajne 00H (ASCIIZ string), jazyk C, ďalej uvažovaná táto reprezentácia

```
string    DB    'Error message',0
```

Reťazcové inštrukcie

- 5 základných reťazcových inštrukcií v jazyku x86 [1]
 - operandy (source, destination) implicitné
 - explicitná špecifikácia veľkosti operandu (NASM)
 - použitie – aj iné účely (kopírovanie údajov pamäť – pamäť)

Mnemonika	Význam	Operandy
LODS	LOaD string	source
STOS	STORe string	destination
MOVS	MOVE string	source, destination
CMPS	CoMPare strings	source, destination
SCAS	SCAn string	destination

Operandy reťazcových inštrukcií

- zdroj (DS:ESI), cieľ (ES:EDI), obidva
- pre 16-bit segmenty (SI, DI)

Variácie

- podpora 8,16,32-bit údajov; automatická aktualizácia (inc/dec) použitých indexových registrov (1,2,4)
- prefix – podpora opakovaného vykonávania operácie (repetition prefix)
- smer spracovania – vpred/vzad (direction flag, DF)

Prefixy reťazcových inštrukcií

- nepodmienené/podmienené opakovanie [1]
- neovplyvňujú stavové príznaky
- prefix **rep** (nepodmienené opakovanie, podľa hodnoty v ECX/CX)
 - najprv test ECX na 0 (rozdiel v porovnaní s loop)

```
while(ECX ≠ 0)
    execute the string instruction;
    ECX:=ECX-1;
end while
```

- prefix **repe/repz** (okrem ECX relevantný aj ZF)

```
while(ECX ≠ 0)
    execute the string instruction;
    ECX:=ECX-1;
    if(ZF = 0)
        then
            exit loop
        end if
    end while
```

	Prefix	Význam
nepodmienené opakovanie	rep	REPeat
podmienené opakovanie	repe/repz	REPeat while Equal REPeat while Zero
	repne/repnz	REPeat while Not Equal REPeat while Not Zero

- prefix **repne/repnz** (ECX a ZF)

```
while (ECX ≠ 0)
    execute the string instruction;
    ECX:=ECX-1;
    if (ZF = 1)
        then
            exit loop
        end if
    end while
```

Smer spracovania reťazcov

- podľa hodnoty DF (DF = 0 – vpred (auto-increment), DF = 1 – vzad)
- manipulácia DF (2 inštrukcie bez operandov, 1B)

```
std  (set DF)
cld  (clear DF)
```

- často smer spracovania nie je podstatný (v niektorých prípadoch však áno)
 - napr. posun reťazca o jednu pozíciu vpravo (začínáme od konca: abc0 → aabc0, od začiatku: abc0 → aaaa0)

Presuny reťazcov (**movs**, **lods**, **stos**)

- 3 formy pre každú z inštrukcií, syntax:

```
movsb, movsw, movsd      (podobne lods, stos)
```

- prípona b, w, d – explicitná špecifikácia veľkosti operandu (aj pre iné reťazcové inštrukcie)
- **movs** – kopírovanie hodnoty (b, w, d) zdrojového reťazca do cieľového
- **lods** – kopírovanie hodnoty zdrojového reťazca (DS:ESI) do AL (**lods b**), AX (**lods w**), alebo EAX (**lods d**)
- **stos** – kopírovanie hodnoty v AL, AX, alebo EAX do cieľového reťazca (ES:EDI)

- sémantika inštrukcií movs, lods, stos [1]:

movsb (move a byte string)	lodsb (load a byte string)	stosb (store a byte string)
<pre> ES:EDI := (DS:ESI) ; byte if (DF = 0) ; forward then ESI := ESI + 1 EDI := EDI + 1 else ; backward ESI := ESI - 1 EDI := EDI - 1 end if </pre>	<pre> AL := (DS:ESI) ; byte if (DF = 0) ; forward then ESI := ESI + 1 else ; backward ESI := ESI - 1 end if </pre>	<pre> (ES:EDI) := AL ; byte if (DF = 0) ; forward then EDI := EDI + 1 else ; backward EDI := EDI - 1 end if </pre>
Ovplyvnené príznaky: žiadne	Ovplyvnené príznaky: žiadne	Ovplyvnené príznaky: žiadne

Porovnanie reťazcov (cmps)

- porovná bajty (slová, dvoj-slová) na DS:ESI a ES:EDI a nastaví príznaky (ako cmp)
- aktualizuje hodnoty ESI, EDI (podľa hodnoty DF a veľkosti operandov)

Príklad: porovnanie reťazcov [1]

- zanechá ESI ukazujúce na 'g' v string1
- EDI ukazujúce na 'f' v string2
- po vykonaní dec ESI, dec EDI, ukazujú tieto na prvý výskyt odlišných znakov
- možnosť použiť podmienené skoky ...

```

C:\Dev-Cpp\bin\asm>L9str
string1 position: g
string2 position: f
C:\Dev-Cpp\bin\asm>

```

```

segment .data
string1 db "abcdefghi", 0
strlen EQU $ - string1
string2 db "abcdefgh", 0

segment .text
...
mov AX, DS
mov ES, AX
mov ECX, strlen
mov ESI, string1
mov EDI, string2
cld
repe cmpsb

```

Prehľadávanie reťazcov (**scas**)

- vyhľadanie určitej hodnoty v reťazci
- hodnota v AL (**scasb**), AX (**scasw**), alebo EAX (**scasd**), ES:EDI – prehľadávaný reťazec
- porovná hodnotu v AL (AX, EAX) s údajom na ES:EDI a nastaví príznaky (ako **cmp**)
- aktualizuje EDI (podľa DF)
- možno použiť prefixy **repe/repz**, **repne/repnz**

Inštrukcie **lds**, **les**

- syntax

```
lds reg,src    (reg - 32b GPR, src - smerník na 48-bit operand v pamäti)
les reg,src
```

- sémantika (32-bit hodnota kopírovaná do **reg**, nasledujúca 16-bit hodnota do segmentového registra, bez ovplyvnenia príznakov)

```
lds: reg ← [src], DS ← [src+4]
les: reg ← [src], ES ← [src+4]
```

- inštrukcie možno využiť pri nastavení registrov (napr. ES:EDI) používaných pri reťazcových operáciach (**les EDI, [string2p]**)

Výhody využitia reťazcových inštrukcií

- automatická aktualizácia indexových registrov
- schopnosť pracovať s dvoma operandmi v pamäti
- v porovnaní s použitím **mov** a pomocného registra - elegantné a efektívne riešenie

```
segment .data
string1    db    "abcdefghi",0
strlen     EQU   $ - string1
string1p    dd    string1
            dw    0
string2     db    "abcdefgh",0
string2p    dd    string2
            dw    0

segment .text
...
mov    [string1p+4],DS
mov    [string2p+4],DS
mov    ECX,strlen
lds    ESI,[string1p]
les    EDI,[string2p]
cld
repe   cmpsb
```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.

Prepojenie assembleru s HL jazykmi

- motivácia, programovanie v zmiešanom režime (mixed-mode)
- volanie asm-procedúr z C, volanie C-funkcií z assembleru
- spôsob výmeny údajov medzi volaným a volajúcim kódom (calling conventions – odovzdávanie parametrov, návratové hodnoty, ...)

Prečo kombinácia assembleru a HL jazyka?

- programovanie v zmiešanom režime – časť programu v C, iná časť v assembleri
- demonštrácia princípov s využitím gcc a NASM (možno rozšíriť na iné prekladače)
- pripomenieme nevýhody asm-programovania (nízka produktivita, náročná údržba, nízka prenositeľnosť)
- a jeho výhody (prístup k hardvéru, k inštrukciám nedostupným vo vyššom jazyku, efektívnosť, ...)
- dôsledok – niektoré programy napísané v zmiešanom režime (napr. systémový softvér)

Základy programovania v zmiešanom režime

- dva spôsoby
 - in-line assembler (program v C obsahuje aj asm-inštrukcie; informácia pre prekladač – `asm`; vhodné pri malom rozsahu asm v C)
 - oddelené asm-moduly (preferované pri väčšom rozsahu asm-kódu)
- oddelené asm-moduly (začneme predstavením tohto spôsobu, in-line neskôr)
 - preklad modulov do objektového tvaru
 - spoločné linkovanie objektových súborov

Príklad: súbory `sample1.c`, `sample2.asm`

```
nasm -f win32 sample2.asm          → sample2.obj
gcc -o sample.exe sample1.c sample2.obj → sample.exe
```

Volanie asm-procedúr z C-programu

- spôsob komunikácie medzi C a asm programom (parametre – zásobník, návratová hodnota – registre)

Odovzdávanie parametrov

- odovzdávanie parametrov (v akom poradí sú parametre ukladané na zásobník)
 - zľava doprava (left-pusher languages, väčšina HL jazykov)
 - sprava doľava (right-pusher, napr. jazyk C – predmet nášho záujmu)

Priklad: preklad volania procedúry (test1) [1]

Details gcc optimalizácie v [4]

volanie procedúry v C:	preklad (gcc hll_test.obj hll_ex1.c.c -m32 -S -masm=intel -Og):
<pre>int main(void) { int x = 25, y = 70; int value; extern int test1(int, int, int); value = test1(x, y, 5); ... return 0; }</pre>	<pre>_main: push ebp mov ebp, esp and esp, -16 sub esp, 16 ... mov DWORD PTR [esp+8], 5 mov DWORD PTR [esp+4], 70 mov DWORD PTR [esp], 25 call _test1 ...</pre>

- pri volaní test1 – argumenty uložené sprava (5, y (70), x (25))
- odstránenie parametrov C programom (detailnejšie neskôr)
 - konvencia jazyka C (cdecl) – volajúca funkcia uvoľní miesto v zásobníku použité na odovzdanie argumentov
 - C povoľuje premenlivý počet argumentov funkcií

Návratové hodnoty

- využívaný register EAX (pre 8, 16, 32-bit celočíselné hodnoty), EDX:EAX (64-bit)
- plávajúca čiarka (float, double) – register ST0 (FPU)

Uchovanie obsahu registrov

- EBX, ESI, EDI, EBP a segmentové registre – mali by ostať zachované (prvé 3 obyčajne používané pre *registrové* premenné)
- v prípade ich zmeny v rámci podprogramu – obnova pôvodných hodnôt (napr. zo zásobníka)

Príklad: ilustrácia odovzdávania parametrov hodnotou/referenciou (adaptované z [1])

- asm-funkcia `min_max` (nájde maximum a minimum zadaných hodnôt)
- 2 súbory: `hll_minmaxc.c` (vľavo), `hll_minmaxa.asm` (vpravo)

```
#include <stdio.h>
int main(void)
{
    int    value1, value2, value3;
    int    minimum, maximum;
    extern void min_max (int, int, int, int*, int*);

    printf("Enter number 1 = ");
    scanf("%d", &value1);
    printf("Enter number 2 = ");
    scanf("%d", &value2);
    printf("Enter number 3 = ");
    scanf("%d", &value3);

    min_max(value1, value2, value3, &minimum, &maximum);
    printf("Minimum = %d, Maximum = %d\n", minimum, maximum);
    return 0;
}
```

```
G:\KPI\ASM>nasm -f win32 hll_minmaxa.asm
```

```
G:\KPI\ASM>gcc -o minmax hll_minmaxc.c hll_minmaxa.obj -m32
```

```
G:\KPI\ASM>minmax.exe
```

```
Enter number 1 = 10
```

```
Enter number 2 = 20
```

```
Enter number 3 = 30
```

```
Minimum = 10, Maximum = 30
```

```
segment .text
global _min_max

_min_max:
    enter    0,0
    ; EAX - minimum, EDX - maximum
    mov     EAX,[EBP+8]      ; value 1
    mov     EDX,[EBP+12]    ; value 2
    cmp     EAX,EDX
    jl      skip1
    xchg    EAX,EDX
skip1:
    mov     ECX,[EBP+16]    ; value 3
    cmp     ECX,EAX
    jl      new_min
    cmp     ECX,EDX
    jl      store_result
    mov     EDX,ECX
    jmp     store_result
new_min:
    mov     EAX,ECX
store_result:
    mov     EBX,[EBP+20]    ; EBX=&minimum
    mov     [EBX],EAX
    mov     EBX,[EBP+24]    ; EBX=&maximum
    mov     [EBX],EDX
    leave
    ret
```

Global/Extern

- samostatné moduly (C a asm)
 - deklarácia funkcií z iného modulu (`extern`)
 - procedúry využívané iným modulom (deklarácia `global`)
- väčšina prekladačov jazyka C – externé návestia (funkcie, globálne premenné) s prefixom (`_`)
 - prefix prekladač pripája automaticky
 - Linux `gcc` – štandardne prefix nevyužíva

Volanie C-funkcií z asm-programu

- niekedy výhodné také volanie (komplikované úlohy, vstup/výstup, ...)
- odovzdávanie parametrov (konvencie používané jazykom C)

Příklad: výpočet súčtu prvkov poľa, využitie funkcií jazyka C `printf()` a `scanf()` [1]

- súbory: `hll_arraysum2c.c`, `hll_arraysum2a.asm`
- pred volaním týchto funkcií – ich argumenty v zásobníku (r. 14 – 16 `printf()`, r. 23 – 25 `scanf()`)
- napr. `printf()` – posledný uložený parameter – adresa formátovacieho reťazca (teda je možné určiť počet parametrov funkcie)
- výpočet súčtu prvkov (r. 36 – 40)

`hll_arraysum2c.c`

```
#include <stdio.h>
#define     SIZE  10

int main(void)
{
    int     value[SIZE];
    extern int array_sum(int*, int);

    printf("sum = %d\n", array_sum(value, SIZE));

    return 0;
}
```

```
G:\KPI\ASM>nasm -f win32 hll_arraysum2a.asm
G:\KPI\ASM>gcc -o arraysum hll_arraysum2c.c hll_arraysum2a.obj -m32
G:\KPI\ASM>arraysum.exe
Input 10 array values:
1
2
3
4
5
6
7
8
9
0
sum = 45
```

h11_arraysum2a.asm

01	segment .data	21	read_loop:
02	scan_format db "%d",0	22	push ECX
03	printf_format db "Input %d array	23	push EDX
04	values:",10,13,0	24	push dword scan_format
05		25	call _scanf ; scanf("%d",&value[i])
06	segment .text	26	add ESP,4 ; clear one argument
07		27	pop EDX
08	global _array_sum	28	pop ECX
09	extern _printf,_scanf	29	add EDX,4 ; update array pointer
10		30	dec ECX
11	_array_sum:	31	jnz read_loop
12	enter 0,0	32	
13	mov ECX,[EBP+12] ; array size	33	mov EDX,[EBP+8]
14	push ECX	34	mov ECX,[EBP+12]
15	push dword printf_format	35	sub EAX,EAX
16	call _printf	36	add_loop:
17	add ESP,8	37	add EAX,[EDX]
18		38	add EDX,4
19	mov EDX,[EBP+8] ; array pointer	39	dec ECX
20	mov ECX,[EBP+12]	40	jnz add_loop
		41	leave
		42	ret

In-line (vložený) assembler

- asm-příkazy vložené v C-kóde, použití konstrukce asm [1,7]
- syntax používaná prekladačem gcc (AT&T) je odlišná od syntaxe v NASM (Intel)
- uvedíme stručný přehľad hlavných syntaktických odlišností

AT&T syntax

- *mená registrov* – prefix % (napr. %eax)

(offset)	stack
+12	SIZE
+8	value ptr.
+4	R.A.
EBP →	old EBP

- *poradie operandov* – opačné poradie (zdroj vľavo, cieľ vpravo)

`mov eax, ebx` je nahradené: `movl %ebx, %eax`

- *rozmer operandov* – inštrukcia špecifikuje rozmer operandu (b, w, l)
 - nie je potrebná explicitná špecifikácia jeho rozmeru (byte, word, dword)

`movb %bl, %al`
`movw %bx, %ax`
`movl %ebx, %eax`

- *bezprostredné operandy, konštanty* – špecifikované prefixom \$

`movb $255, %al`
`movl $total, %eax` (total – globálna C-premenná, ináč rozšírená asm-konštrukcia)

- *adresovanie* – okrúhle zátvorky (namiesto [])

`mov eax, [ebx]` nahradené: `movl (%ebx), %eax`

- úplný 32-bit formát: `imm32(base, index, scale)` výpočet adresy: `imm32 + base + index * scale`

Příklad: `marks[5]` → EAX, kde `marks` – globálne pole (int)

`movl $5, %ebx`
`movl marks(, %ebx, 4), %eax`

Jednoduché in-line príkazy

- asm-príkaz (inkrement EAX): `asm("incl %eax");`

- skupina asm-príkazov:

```
asm("pushl  %eax");  
asm("incl  %eax");  
asm("popl   %eax");
```

alebo: `asm("pushl %eax; incl %eax; popl %eax");`

Rozšírené in-line príkazy

- možnosť prístupu k neglobálnym C-premenným, informácia pre prekladač o nami používaných registroch
- formát príkazu asm: (4 komponenty, posledné 3 – nepovinné)

```
asm(asm-code  
    :outputs  
    :inputs  
    :clobber list);
```

asm-code

- komponent obsahuje asm-príkazy (jednotlivé/skupiny)
- inštrukcie v tejto sekcii môžu používať operandy špecifikované v ďalších dvoch komponentoch (outputs, inputs)
- pokiaľ nemá prekladač vykonávať optimalizáciu – kľúčové slovo `volatile` (za asm)

outputs

- špecifikácia výstupných operandov, formát:

`"=op-constraint" (C-expression)`

- = identifikuje výstup, napr. `"=r" (sum)`, kde premenná (sum) bude mapovaná na register (r)
- ďalšie prípustné voľby: m (memory), i (immediate), rm, ri, g (general) = rim
- špecifikácia registra (gcc) [1, 5]:

Označenie	Register
a	EAX
b	EBX
c	ECX
d	EDX
S	ESI
D	EDI
r	Lubovoľný z 8 GPR (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Lubovoľný zo 4 údajových registrov (EAX, EBX, ECX, EDX)
A	64-bit hodnota v EAX a EDX
f	FPU register
t	Prvý (top) FPU register
u	Druhý FPU register

inputs

- podobne ako výstupy, okrem symbolu =
- operandom (špecifikovaným v inputs a outputs) sú pridelené poradové čísla (0,1,2 ..., 9 – max. 10 operandov)
- v kóde je možné sa na ne odvolávať (%číslo)
- ak je operand vstupom a zároveň výstupom (prijíma výsledok) – uvedený v obidvoch zoznamoch
- registre – prefix % (keďže AT&T syntax tu využíva tiež %, EAX označíme %%eax)

Príklad: `sum = sum + number1` (premenná `sum` – vstup aj výstup súčasne, avšak v inputs uvedená už ako "0") [1]

```
asm("addl %1,%0"
    : "=r" (sum)           /* output */
    : "r" (number1), "0" (sum) /* inputs */
);
```

clobber list

- špecifikuje zoznam registrov modifikovaných v rámci príkazu `asm` (informácia pre gcc – prípadná obnova obsahov)
- "memory" – pri zmene v pamäti, "cc" – pri zmene registra príznakov

Príklad: žiadny výstup, EAX modifikovaný (uvedený v zozname 'clobber list') [1]

```
asm("movl %0,%%eax"
    :/* no output */
    : "r" (number1)      /* inputs */
    : "%eax"             /* clobber list*/
);
```

Príklad: procedúra `test1` v in-line režime (vstupné operandy – x, y, z; modifikované – EAX, cc; výsledok (x+y-z) vrátený v EAX) [1]

```
#include <stdio.h>

int main(void)
{
    int    x = 25, y = 70;
    int    value;
    int    test1(int, int, int);

    value = test1(x, y, 5);
    printf("Result = %d\n", value);

    return 0;
}
```

```
int test1(int x, int y, int z)
{
    asm("movl  %0,%%eax;"
        "addl  %1,%%eax;"
        "subl  %2,%%eax;"
        :/* no outputs */
        : "r"(x), "r"(y), "r"(z) /* inputs */
        : "cc", "%eax");
}
```

```
G:\KPI\ASM>gcc -o ex1_inline ASM10_h11_ex1_inline.c -m32
G:\KPI\ASM>ex1_inline.exe
Result = 90
```

Konvencie volania funkcií [3]

- uvedené sú štandardné C-konvencie volania funkcií
- prekladače často podporujú aj ďalšie konvencie (CL-prepínače, rozšírenie syntaxe jazyka C)

- prekladač GCC
 - podporuje rôzne konvencie volania funkcií [3, 6], uvedieme niektoré z nich
 - možnosť špecifikácie atribútov (`__attribute__`) pri deklarácii funkcie, napríklad:
 - `cdecl` – prekladač bude predpokladať, že *volajúca* funkcia uvoľní miesto v zásobníku použité na odovzdanie argumentov
 - `stdcall` – prekladač bude predpokladať, že *volaná* funkcia uvoľní miesto, pokiaľ nemá premenlivý počet argumentov
 - `regparm` – celočíselné argumenty budú odovzdané v *registroch* (EAX, EDX, ECX), funkcie s premenlivým počtom argumentov budú naďalej používať zásobník

```
int test(int, int, int) __attribute__((cdecl));
int test(a, b, c)
{
    return a;
}

void main(void)
{
    int x = 25, y = 70, value;
    value = test(x, y, 5);
    printf("value: %d\n", value);
    return;
}
```

```
_test:
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR [ebp+8]
    pop     ebp
    ret
```

```
int test(int, int, int) __attribute__((stdcall));
int test(a, b, c)
{
    return a;
}

void main(void)
{
    int x = 25, y = 70, value;
    value = test(x, y, 5);
    printf("value: %d\n", value);
    return;
}
```

```
_test@12:
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR [ebp+8]
    pop     ebp
    ret     12
```

```
int test(int, int, int) __attribute__((regparm(3)));
int test(a, b, c)
{
    return a;
}

void main(void)
{
    int x = 25, y = 70, value;
    value = test(x, y, 5);
    printf("value: %d\n", value);
    return;
}
```

```
_main:
    ...
    mov     edx,DWORD PTR [esp+24]
    mov     eax,DWORD PTR [esp+28]
    mov     ecx,5
    call    _test
    ...
```

```
_test:
    push    ebp
    mov     ebp,esp
    sub     esp,12
    mov     DWORD PTR [ebp-4],eax
    mov     DWORD PTR [ebp-8],edx
    mov     DWORD PTR [ebp-12],ecx
    mov     eax,DWORD PTR [ebp-4]
    leave
    ret
```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] A GNU Manual, [Declaring Attributes of Functions](#), Free Software Foundation, Inc., 1988-2023.
- [4] A GNU Manual, [Options That Control Optimization](#), Free Software Foundation, Inc., 1988-2023.
- [5] A GNU Manual, [Constraints for Particular Machines](#), Free Software Foundation, Inc., 1988-2023.
- [6] A GNU Manual, [x86 Function Attributes](#), Free Software Foundation, Inc., 1988-2023.
- [7] A GNU Manual, [How to Use Inline Assembly Language in C Code](#), Free Software Foundation, Inc., 1988-2023.

Prerušená

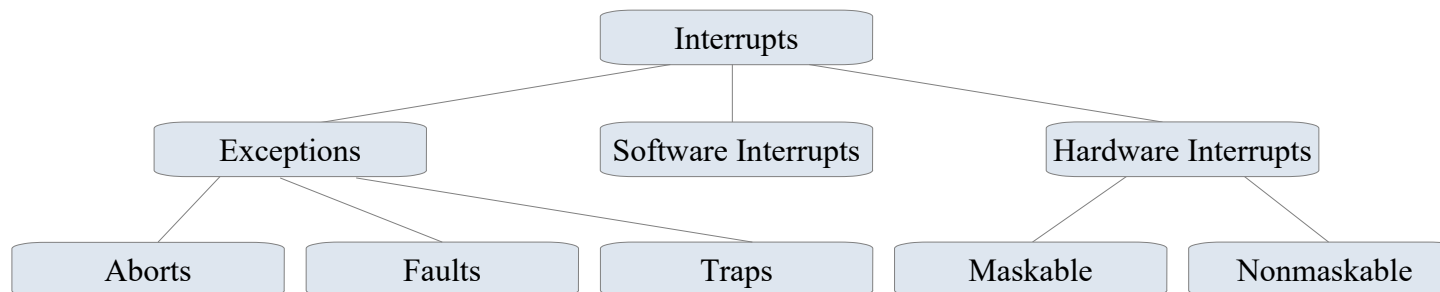
- mechanizmy prerušenia v chránenom režime (Pentium)
- HW a SW prerušenia, výnimky, systémové volania (Linux)
- prerušenia v reálnom režime (volania služieb systémov DOS, BIOS)

Úvod

- prerušenie ako mechanizmus pre zmenu sekvenčného vykonávania programu (aj skoky a procedúry), podobnosť s volaním procedúr
 - prenos riadenia na procedúru obsluhy prerušenia (ISR – Interrupt Service Routine)
 - po ukončení ISR, obnova vykonávania prerušeného programu
- odlišnosti
 - prerušenia možno iniciovať softvérovo aj hardvérovo (externé zariadenia)
 - softvér (software interrupts) – inštrukcia `int` (očakávaná/plánovaná udalosť)
 - hardvér – napr. `ctrl-c/ctrl-break` – návrat riadenia OS (neočakávaná udalosť)
 - ďalšie odlišnosti (ISR – bežne rezidentné v pamäti, identifikácia číslom miesta mena, automaticky uložený FLAGS, ...)

Klasifikácia prerušení

- okrem dvoch spomínaných kategórií (SW a HW – inicializované) aj výnimky (exceptions) [1]
 - ošetrovanie chýb pri spracovaní inštrukcií (napr. delenie 0)
- SW prerušenia (`int`) – prístup k I/O zariadeniam (systémové (system defined) / používateľské (user defined))
 - možno použiť na emuláciu výnimiek programovo (špecifikovaním príslušného vektora ako operandu)
- HW prerušenia (generované hardvérom – získanie času procesora zariadením)
 - maskovateľné (maskable) – spracovanie možno odložiť
 - nemaskovateľné (non-maskable, NMI) – vždy akceptované/spracované procesorom (RAM parity error)

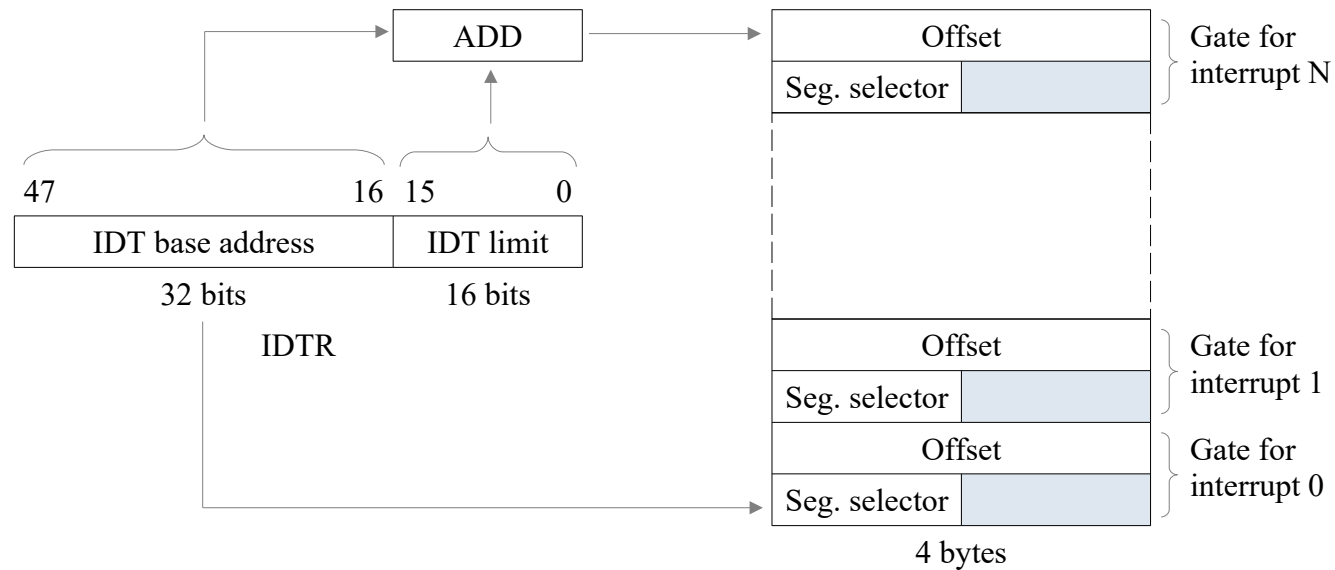


Spracovanie prerušení v chránenom režime (protected mode)

- identifikácia prerušenia číslom (Pentium – 256 typov prerušení)
- číslo (typ) prerušenia (vector, 0-255) – index do tabuľky s adresami ISR
 - tabuľka IDT (Interrupt Descriptor Table), číslo prerušenia * 8 → index do tabuľky (položka – descriptor, 8B) [1]
 - podobne ako (spomínané) tabuľky LDT a GDT
 - IDT – umiestnená v pamäti, pozícia daná obsahom registra IDTR (48b, 32b báza + 16b limit)
 - špeciálne inštrukcie `lidt`, `sidt` (load/store, 6B pamäťový operand)
 - IDT – môže obsahovať 3 typy deskriptorov (interrupt gate, trap gate, task gate)
 - task gate – ďalej nebudeme uvažovať (nesúvisí priamo s mechanizmom prerušení)
 - interrupt gate, trap gate – 16b segment selector, 32b offset, DPL, P (present)
 - segment – voľba selektora z GDT alebo LDT (bit TI)
 - offset – z interrupt gate

3																				1	1	1	1	1					8	7			5	4							0								
1																				6	5	4	3	2																									
Offset 31:16																					P	DPL	0 1 1 1 0					0 0 0			Not used																		
Segment selector																					Offset 15:00																												
3																				1	1																						0						
1																				6	5																												
Interrupt gate																																																	

Organizácia IDT [1]:



Výskyt prerušenia (prípád bez zmeny privilégií)

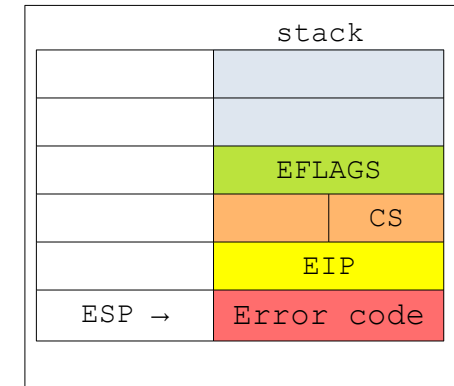
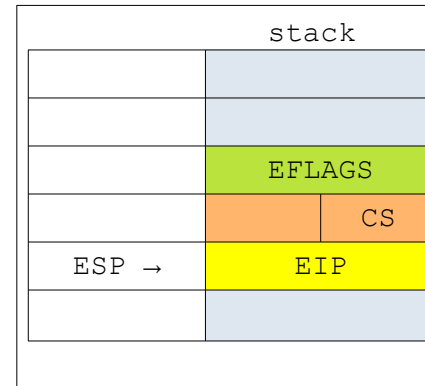
- EFLAGS → zásobník
- IF = 0 a TF = 0 (zakázanie ďalších prerušení; možné opätovné povolenie v ISR (`sti`, `cli`), pokiaľ nie je dôvod ich zakazovať)
- CS, EIP → zásobník
- CS ← 16b segment selector (interrupt gate)
- EIP ← 32b offset (interrupt gate)

Prerušenie cez Trap gate

- podobne ako Interrupt gate (doposiaľ), ale bez modifikácie IF
- niektoré typy výnimiek (8, 10-14, 17) ukladajú na zásobník aj chybový kód (identifikácia príčiny výnimky v rámci obsluhy)

Návrat z obsluhy prerušenia

- podobne ako procedúry (inštrukcia `iret`), činnosť:
 - $EIP \leftarrow$ zásobník (32b)
 - $CS \leftarrow$ zásobník (16b)
 - $EFLAGS \leftarrow$ zásobník (32b)

**Vektory prerušenia a výnimiek v chránenom režime**

- vektory 0 – 31 rezervované; architektúrou definované výnimky a prerušenia (nie všetky nutne obsadené) [6]
- vektory 32 – 255 používateľsky definované; všeobecne dostupné pre externé zariadenia

Vector no.	Mnemonic	Description	Type	Error code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions
1	#DB	Debug Exception	Fault/Trap	No	Breakpoints, single-step, ...
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt
3	#BP	Breakpoint	Trap	No	INT 3 instruction
4	#OF	Overflow	Trap	No	INTO instruction
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction
...
22-31	—	Intel reserved. Do not use.	Interrupt		External interrupt or INT <i>n</i> instruction
32-255	—	User defined (non-reserved) interrupts			

Výnimky

- klasifikované do 3 skupín (*faults*, *traps*, *aborts*) podľa spôsobu hlásenia ich výskytu a možnosti prípadného reštartu inštrukcie
 - **fault** (porucha) – výnimočný stav, ktorý všeobecne možno napraviť v rámci obsluhy
 - v prípade úspešnej nápravy program reštartovaný bez straty kontinuity
 - obnova stavu do bodu pred vykonaním (výnimku generujúcej) inštrukcie (uchované CS:EIP ukazujúce na danú inštrukciu)
 - napr. výpadok segmentu (ISR zabezpečí zavedenie z disku, vráti riadenie programu – reštart inštrukcie)
 - **trap** (pasca) – výnimka signalizovaná po vykonaní inštrukcie (napr. overflow)
 - umožňuje pokračovanie v programe bez straty kontinuity; bez reštartu inštrukcie
 - návratová adresa ukazuje bezprostredne za (výnimku generujúcu) inštrukciu
 - napr. volanie služieb systému
 - **abort** (zlyhanie) – výnimka nie vždy signalizovaná presne na mieste generujúcej inštrukcie
 - neumožňuje pokračovanie programu, ktorý zlyhanie spôsobil
 - závažné chyby (chyby HW, nekorektné hodnoty v systémových tabuľkách)
 - úlohou obsluhy je zber diagnostických informácií a ukončenie aplikácie čo najelegantnejšie
- existencia vyhradených prerušení (dedicated interrupts), napr. prvých 5 prerušení:
 - Divide error (0, #DE) – `div/idiv` (podiel väčší ako špecifikovaný cieľ)
 - Single-step (1, #DB) – krokovanie programu (ladiace nástroje), ak TF = 1, CPU generuje toto prerušenie po vykonaní inštrukcie
 - Nonmaskable interrupt (NMI, 2) – neskôr
 - Breakpoint (3, #BP) – spracovanie bodov prerušenia, inštrukcia `int 3` (CCh, 1B)
 - Overflow (4, #OF) – dva spôsoby generovania (`int 4, into`), bežne pretečenie ošetrené podmieneným skokom (`jo, jno`)

Softvérové prerušenia

- iniciované vykonaním inštrukcie prerušenia

`int int-type (int-type: 0-255)`

- jedná sa o typy prerušení
- parametrizácia (napr. systémové volania Linuxu `int 0x80`, cca.180 volaní (v závislosti od verzie), číslo volania v EAX)

Systémové volania – Linux

- Linux využíva prvých 32 vektorov (0-31) pre výnimky a NMI
- ďalších 16 (32-47) – HW prerušenia (IRQ)
- jeden vektor (128/0x80) – SW prerušenia (služby systému)

Súborové I/O operácie (file I/O)

- Linux (Unix) – intenzívne využívanie súborov (napr. klávesnica, displej)
- vstup a výstup z pohľadu systému – prúd bajtov (stream)
- štandardne definované 3 súbory – `stdin` (klávesnica), `stdout` a `stderr` (displej)

Deskriptor súboru (file descriptor)

- každý otvorený súbor – 16b celé číslo (file id), prístup k súboru
- návratová hodnota volaní *file open*, *file create*
- identifikátory s najnižšími hodnotami: `stdin` (0), `stdout` (1), `stderr` (2)

Ukazovateľ pozície v súbore (file pointer)

- asociovaný s každým otvoreným súborom
- posun (offset) v bajtoch od začiatku súboru (pozícia v súbore pre operácie read/write), pri otvorení súboru nulový
- sekvenčný (aktualizácia ukazovateľa vzhľadom na prečítané/zapísané bajty) vs. priamy prístup (manipulácia ukazovateľa) k súboru

Systémové volania pre prácu so súbormi (file system calls)

- začiatok práce so súborom – vytvorenie (call 8)/otvorenie (call 5) súboru
- po otvorení (vytvorení) dostupné ďalšie operácie – čítanie (call 3), zápis (call 4)
- priamy prístup (direct access) k údajom – manipulácia ukazovateľa (call 19)
- po ukončení práce – uzatvorenie súboru (call 6)

System call 8 (Create and open a file)

Vstup: EAX = 8, EBX = file name (smerník), ECX = file permissions [1]

Výstup: EAX = file descriptor (kladný)

Chyba: EAX = error code (záporný)

8	7	6	5	4	3	2	1	0
R	W	X	R	W	X	R	W	X
User			Group			Other		

System call 5 (Open a file)

Vstup: EAX = 5, EBX = file name (smerník), ECX = file access mode (R-only (0), W-only (1), R-W (2), ...), EDX = file permissions

Výstup: EAX = file descriptor

Chyba: EAX = error code

System call 3 (Read from a file)

Vstup: EAX = 3, EBX = file descriptor, ECX = input buffer (smerník), EDX = buffer size (bytes)

Výstup: EAX = number of bytes read

Chyba: EAX = error code

- po ukončení čítania, ukazovateľ aktualizovaný (bajt za posledným prečítaným bajtom), ďalšie volanie – sekvenčný prístup
- ak $EAX < EDX$ – dosiahnutý koniec súboru

System call 4 (Write to a file)

Vstup: EAX = 4, EBX = file descriptor, ECX = output buffer (smerník), EDX = buffer size (bytes)

Výstup: EAX = number of bytes written

Chyba: EAX = error code

- ak $EAX < EDX$ – chyba (napr. disk full)

System call 6 (Close a file)

Vstup: EAX = 6, EBX = file descriptor

Výstup: EAX = –

Chyba: EAX = error code

System call 19 (lseek – update file pointer)

Vstup: EAX = 19, EBX = file descriptor, ECX = offset, EDX = whence

Výstup: EAX = offset from the beginning of file

Chyba: EAX = error code

- nesequenčný (priamy, náhodný) prístup k údajom v súbore
- offset (ECX) sa pripočíta k pozícii udanej EDX (0 – začiatok súboru, 1 – aktuálna pozícia, 2 – koniec súboru)
- ďalšie volania a ich opis napr. v [3, 4]

Příklad: načítanie reťazca z klávesnice (zachovanie príznakov pushf/popf), pripojenie NULL na konci (ASCIIZ) [1]

; getstr receives input buffer pointer in EDI, buffer size in ESI

```
getstr:
    pusha
    pushf
    mov     EAX, 3           ; read file
    mov     EBX, 0           ; 0 = stdin
    mov     ECX, EDI
    mov     EDX, ESI
    int     0x80
    dec     EAX
    mov     byte[EDI+EAX], 0 ; NULL character appended
    popf
    popa
    ret
```

Hardvérové prerušenia

- SW prerušenia – synchrónne udalosti (`int` v programe)
- HW prerušenia – asynchrónne (pôvodcom je hardvér – I/O zariadenia)
 - nemaskovateľné (NMI) – pripojené priamo na NMI vývod CPU (vždy na ne reaguje, prerušenie typu 2)
 - maskovateľné – väčšina HW prerušení (signál na vývod (pin) `INTR` – `INTerrupt Request`) – obslúžené, ak `IF` = 1
 - `IF` (`Interrupt Enable`), inštrukcie `sti`, `cli`

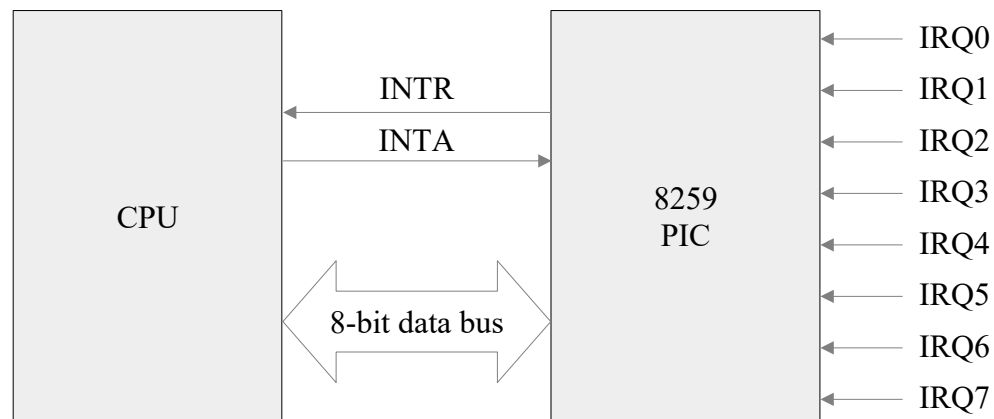
Typ prerušenia

- aký je typ (číslo) prerušenia iniciovaného hardvérom?
 - ako odpoveď na signál `INTR` – procesor vyšle signál `INTA` (`interrupt acknowledge`)
 - zariadenie umiestni vektor prerušenia na údajovú zbernicu

Spracovanie prerušení od viacerých zariadení

- vývod `INTR` – ako pripojiť viac zariadení?
- mechanizmus určovania priorít (pri súčasnom príchode)
 - jednu požiadavku prepustiť ďalej, ostatné podržať
 - mechanizmus implementovaný špeciálnym čipom – Intel 8259 (`Programmable Interrupt Controller`, `PIC`) [1], [8]
 - možnosť priamo obslúžiť až 8 zariadení (cez linky `IRQ0` – `IRQ7`)
 - dvojica (8-bit) registrov pre konfiguráciu: `ICR` (`interrupt command register`) a `IMR` (`interrupt mask register`)
 - `IMR` – povolenie/zakázanie jednotlivých požiadaviek
 - `ICR` – pridelovanie priorít požiadavkám (bežná inicializácia BIOS-om: `IRQ0` – najvyššia, `IRQ7` – najnižšia priorita)
 - v rámci inicializácie priradené tiež typy (vektory) prerušení (špecifikovaný len najnižší, ostatné priradené automaticky)
 - ak 8259 prijme signál `INTA`, vyšle tento údaj (vektor) na údajovú zbernicu
 - komunikácia CPU a 8259 cez údajovú zbernicu; registre `ICR` a `IMR` mapované v I/O adresnom priestore (`20H` a `21H`)
 - prerušenia od 8259 akceptované ak `IF` = 1; selektívne povolenie/zakázanie prerušení – pomocou `IMR` (ak bit = 1 – zakázať)
 - viacero požiadaviek súčasne – serializácia podľa priority; ukončenie obsluhy signalizované zápisom hodnoty `20H` do `ICR`
 - v moderných PC nahrádzaný novším systémom `APIC` (`Advanced Programmable Interrupt Controller`) [9]

<code>mov</code>	<code>AL, 20H</code>
<code>out</code>	<code>20H, AL</code>



Prerušená v reálnom režime

- MS DOS, BIOS poskytujú niekoľko služieb prostredníctvom prerušení
- Pentium používa v tomto prípade mechanizmy prerušení procesora 8086
 - IDT začína na adrese 0
 - každý vektor má veľkosť 4B (typ prerušená * 4 pre získanie správnej položky)
 - vektor obsahuje CS:IP smerník na ISR

Pri výskyte prerušená

FLAGS → zásobník

IF = 0, TF = 0

CS, IP → zásobník

CS ← 16b (typ prerušená * 4 + 2)

IP ← 16b (typ prerušená * 4)

Návrat z prerušená (iret)

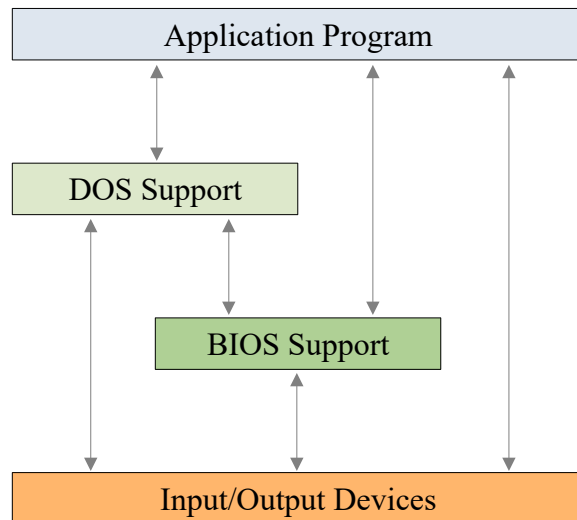
IP ← zásobník

CS ← zásobník

FLAGS ← zásobník

Softvérové prerušenia

- služby systému DOS (int 21h), vyše 80 funkcií
- DOS a BIOS obsahujú aj služby pre prístup k I/O zariadeniam
- spôsoby interakcie s I/O zariadeniami (DOS, BIOS, priame riadenie) [1]:



address			
003FF	CS high byte	CS	int type 255
003FE	CS low byte		
003FD	IP high byte	IP	
003FC	IP low byte		
...
0000B	CS high byte	CS	int type 2
0000A	CS low byte		
00009	IP high byte	IP	
00008	IP low byte		
00007	CS high byte	CS	int type 1
00006	CS low byte		
00005	IP high byte	IP	
00004	IP low byte		
00003	CS high byte	CS	int type 0
00002	CS low byte		
00001	IP high byte	IP	
00000	IP low byte		

Práca s klávesnicou

- všeobecne – radič zariadenia (I/O controller, HW rozhranie) a ovládač zariadenia (device driver, SW rozhranie)
 - klávesnica – špeciálny obvod, pri stlačení/uvoľnení klávesu generované prerušenie (int 9)
 - prerušenie obslužené BIOS-om (prijíma tzv. scan-kód klávesy, generuje ASCII kód, uloží do frontu (FIFO buffer) klávesnice)

*Základné služby systému DOS pre prácu s klávesnicou:**Funkcia 01H (Keyboard input with echo)*

Vstup: AH = 01H

Výstup: AL = ASCII code

- ak je zadáný ctrl-break, vyvolané je prerušenie 23H

Funkcia 08H (Keyboard input without echo)

Vstup: AH = 08H

Výstup: AL = ASCII code

Funkcia 0AH (Buffered keyboard input)

Vstup: AH = 0AH

DS:DX = pointer to the input buffer (1.byte should have the buffer size)

Výstup: string in the buffer

- zadaný reťazec umiestnený od 3. bajtu, číta sa po Enter, alebo zaplnenie buffra
- po ukončení čítania – počet prečítaných znakov v 2.bajte buffra
- špeciálne klávesy (extended keyboard keys)
 - nie sú súčasťou ASCII (F1, F2, ... Kurzorové šípky, Home, End, ...)
 - pri výskyte takého klávesu – 2B vo fronte (00H a príslušný scan-kód) – potrebné dve volania funkcie ...

Základné služby systému BIOS pre prácu s klávesnicou:

- dostupné pomocou int 16H

Funkcia 00H (Read a character from the keyboard)

Vstup: AH = 00H

Výstup: if AL \neq 0 then AL = ASCII code, AH = scan-code
if AL = 0 then AH = scan-code

Funkcia 01H (Check keyboard buffer)

Vstup: AH = 01H

Výstup: ZF = 1 if keyboard buffer is empty
ZF = 0 if nonempty (returns ASCII in AL and scan-code in AH, does not remove them from buffer)

Funkcia 02H (Check keyboard status)

Vstup: AH = 02H

Výstup: AL = status of shift and toggle keys (0 – RShift, 1 – LShift, 2 – Control, ...)

- ďalšie služby systémov DOS, BIOS – napr. [5]

Výstup textu na obrazovku

- DOS aj BIOS poskytujú niekoľko funkcií pre zobrazovanie znakov (uvádzame služby DOS-u)

Funkcia 02H (Display a character on the screen)

Vstup: AH = 02H

DL = ASCII code to be displayed

Výstup: nothing

Funkcia 09H (Display a string of characters)

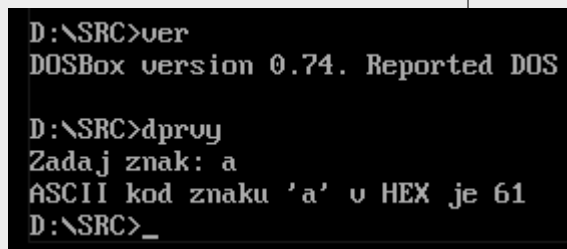
Vstup: AH = 09H

DS:DX = pointer to a character string (\$ - terminated)

Výstup: nothing

Príklad: verzia programu pre výpis HEX kódu zadaného znaku s využitím volaní systému DOS

<pre>segment .data prompt db "Zadaj znak: ", '\$', 0 msg1 db 10, 13, "ASCII kod znaku '", '\$', 0 msg2 db "' v HEX je ", '\$', 0 segment .stack stack resb 64 segment .text ..start: mov ax, data mov ds, ax mov ax, stack mov ss, ax mov ah, 9 ; vypis retazca mov dx, prompt int 21h mov ah, 1 ; nacitaj znak int 21h mov cl, al mov ah, 9 ; vypis retazca mov dx, msg1 int 21h</pre>	<pre>mov ah, 2 ; vypis nacitaného znaku mov dl, cl int 21h mov ah, 9 mov dx, msg2 int 21h mov dl, cl ; prevod do HEX shr dl, 4 call HEX_znak mov dl, cl call HEX_znak mov ax, 4C00h ; ukoncenie programu int 21h HEX_znak: and dl, 0fh ; spodne 4 bity DL cmp dl, 9 jle addzero add dl, 'A'-10-'0' addzero: add dl, '0' mov ah, 2 int 21h ret</pre>
---	---



```
D:\SRC>ver
DOSBox version 0.74. Reported DOS
D:\SRC>dprvy
Zadaj znak: a
ASCII kod znaku 'a' v HEX je 61
D:\SRC>_
```

- využívané služby systému DOS (01h – vstup znaku, 02h – výpis znaku, 09h – výpis reťazca, 4Ch – ukončenie programu)
- preklad programu: `nasm -f obj dprvy.asm`, linkovanie: `alink -oEXE dprvy.obj`
- spustenie: na 64-bit systémoch v prostredí DOSBox [7]

Priame riadenie I/O zariadení

- neexistujúca podpora cez DOS/BIOS
- neštandardný prístup k zariadeniu
- Pentium – špeciálny I/O adresný priestor (64K 8-bit portov, 32K 16-bit portov, alebo 16K 32-bit portov, resp. ich kombinácie)
- prístup k I/O portom
 - registrové I/O inštrukcie (prenosy register-port, 8/16/32b – podľa voľby `acc`)

<code>in</code>	<code>acc, port8</code>	(<code>acc</code> – AL, AX, EAX, <code>port8</code> – priamo adresovaný port 0 – FFH)
<code>in</code>	<code>acc, DX</code>	(DX – adresa portu, 0 – FFFFH)
<code>out</code>	<code>port8, acc</code>	
<code>out</code>	<code>DX, acc</code>	

- blokové I/O inštrukcie (pamäť-porty, podobnosť s reťazcovými inštrukciami)
 - bez operandov, možno použiť prefix `rep` (nie však `repe`, `repne`), DF – ako u reťazcových inštrukcií
 - DX a indexové registre automaticky aktualizované

<code>ins</code>	(<code>insb</code> , <code>insw</code> , <code>insd</code> – adresa portu v DX, pamäť: ES:EDI)
<code>outs</code>	(<code>outsb</code> , <code>outsw</code> , <code>outsd</code> – adresa portu v DX, pamäť: DS:ESI)

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] Boldyshev, K.: [List of Linux/i386 system calls](#), 1999-2000.
- [4] Linux Syscall Reference (32 bit), <https://syscalls32.paolostivanin.com/>
- [5] Jurgens, D.: HelpPC Reference Library, [Interrupt Services DOS/BIOS/EMS/Mouse](#)
- [6] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Chapter 6, 2023.
- [7] DOSBox, <https://www.dosbox.com/>, 2021.
- [8] OSDev.org, [8259 PIC](#), 2023.
- [9] Wikipedia, [Advanced Programmable Interrupt Controller](#), 2022.

Aritmetika s pohyblivou rádovou čiarkou

- reprezentácia čísel s pohyblivou rádovou čiarkou
- organizácia jednotky pre prácu s pohyblivou rádovou čiarkou (FPU)
- inštrukcie pre prácu s číslami s pohyblivou rádovou čiarkou

Reprezentácia čísel

- doposiaľ na reprezentáciu hodnôt využívané celé čísla
- čísla s pohyblivou rádovou čiarkou pre reprezentáciu reálnych čísel (*float*, *double* v jazyku C)
- problém zaokrúhľovania (neprítomný v aritmetike s celými číslami)
- reprezentácia s pevnou rádovou čiarkou (fixed-point representation)
 - jednoduchá ($N = I + F$, I – celá časť, F – desatinná časť, N – celkový počet bitov)

$$a_{I-1} \dots a_1 a_0 . a_{-1} a_{-2} \dots a_{-F}$$

- presnosť (F) vs. rozsah (I)
- vážne nedostatky (obmedzený rozsah, resp. vysoké priestorové požiadavky)
- reprezentácia s využitím exponenciálnej notácie (znamienko, mantisa, exponent)
 - lepšie využitie daného počtu bitov pre reprezentáciu reálneho čísla

$$\text{veľkosť} = \text{mantisa} * 2^{\text{exponent}}$$

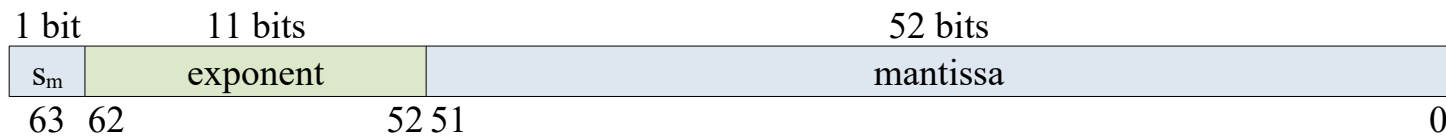
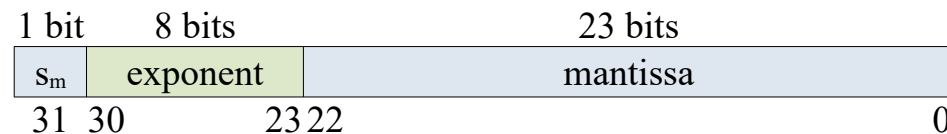
- normálna forma (binárna reprezentácia), X_i , Y_j ($1 \leq i \leq M$, $0 \leq j < N$) – bity mantisy resp. exponentu

$$\pm 1.X_1 X_2 \dots X_{M-1} X_M \times 2^{\pm Y_{N-1} Y_{N-2} \dots Y_1 Y_0}$$

Príklad: normálna forma čísla $+1101.101 \times 2^{+11010}$ je:

$$+1.101101 \times 2^{+11101} \quad \text{resp.} \quad +1.101101\text{E}11101$$

- implementácia na procesore Pentium v súlade so štandardom IEEE 754
 - výmena údajov medzi systémami, numerické knižnice
- FPU procesora Pentium podporuje 3 formáty čísel [1]:
 - čísla s jednoduchou presnosťou (32-bit, float v C)
 - čísla s dvojitou presnosťou (64-bit, double v C)
 - rozšírený formát (80-bit, interný)



- poznámky k formátom čísel s jednoduchou a dvojitou presnosťou:
 - *mantisa* – uchováva iba desatinnú časť normalizovaného čísla (1 naľavo od čiarky sa neukladá (predpokladá sa stále 1))
 - *exponent* – bez znamienkového bitu, kód s posunutou nulou (pripočítanie 127 (7FH) resp. 1023 (3FFH))

Prevod reálneho čísla do formátu s pohyblivou rádovou čiarkou

- ilustrácia na prevode čísla 78.8125
- krok 1 – prevod čísla do binárneho tvaru

$$(78)_{10} = (1001110)_2, (0.8125)_{10} = (0.1101)_2, (78.8125)_{10} = (1001110.1101)_2$$

- krok 2 – normalizácia

$$(1001110.1101E0)_2 = (1.0011101101E110)_2$$

- krok 3 – prevod exponentu do kódu s posunutou nulou

$$(110)_2 + (1111111)_2 = (10000101)_2 \text{ takže: } (78.8125)_{10} = (1.0011101101E10000101)_2$$

- krok 4 – oddelenie jednotlivých častí

znamienko: 0, mantisa: 0011101101 (1. - implicitná), exponent: 10000101

Špeciálne hodnoty

- reprezentácia špeciálnych hodnôt – dohodnutou kombináciou hodnôt Z, M, E [1]

Hodnota	Znamienko	Exponent	Mantisa
+0	0	0	0
-0	1	0	0
$+\infty$	0	FFH	0
$-\infty$	1	FFH	0
NaN	0/1	FFH	$\neq 0$
Denormals	0/1	0	$\neq 0$

- NaN (not-a-number) – reprezentácia hodnôt, ktoré sú nedefinované, resp. nereprezentovateľné, napr. 0/0 alebo $\sqrt{-1}$
- Denormals – hodnoty menšie ako je možné zobrazit' v normálnej forme (implicitná 1 nahradená 0)

Operácie s číslami s pohyblivou rádovou čiarkou

- sčítanie (odčítanie)
 - vyrovnanie exponentov (na vyšší)
 - sčítanie (odčítanie) mantís
 - normalizácia výsledku
 - test na pretečenie/podtečenie
- násobenie
 - sčítanie exponentov (pozor – kód s posunutou nulou)
 - násobenie mantís
 - určenie znamienka
 - normalizácia súčinu
 - test na pretečenie/podtečenie

Príklad: $13.25 + 4.75$ ($1.10101 \times 2^3 + 1.0011 \times 2^2$)
 $1.0011 \times 2^2 = 0.10011 \times 2^3$
 $1.10101 + 0.10011 = 10.01$
 1.001×2^4 ($=18_{10}$)
žiadne

Príklad: $15 * 5$ ($1.111 \times 2^3 * 1.01 \times 2^2$)
 $3 + 2 = 5$
 $1.111 \times 1.01 = 10.01011$
kladné
 $10.01011 \times 2^5 = 1.001011 \times 2^6$ ($=75_{10}$)
žiadne

Jednotka FPU

- zvýšenie výkonu procesora v oblasti výpočtov s pohyblivou rádovou čiarkou – pridaním špeciálneho hardvéru
 - procesor bez FPU – emulácia činnosti FPU (strata rýchlosti) – procedúry obsahujúce bežné (non-floating point) inštrukcie pre vykonávanie operácií s pohyblivou čiarkou
- v minulosti vo forme samostatných koprocesorov (8087 pre 8086, podobne 80287 a 80387) [5]
- počnúc procesorom 80486 DX – FPU integrovaná do procesora (1989)
- v ďalšom texte predmetom nášho záujmu – jednotka FPU integrovaná v procesore Pentium

Organizácia jednotky FPU

- registre jednotky FPU rozdelené do troch skupín [1]
 - údajové (data registers, 8x80 bitov)
 - riadiace a stavové (control and status registers, 3x16 bitov)
 - ukazovatele (pointer registers) – podpora spracovania výnimiek



Údajové registre

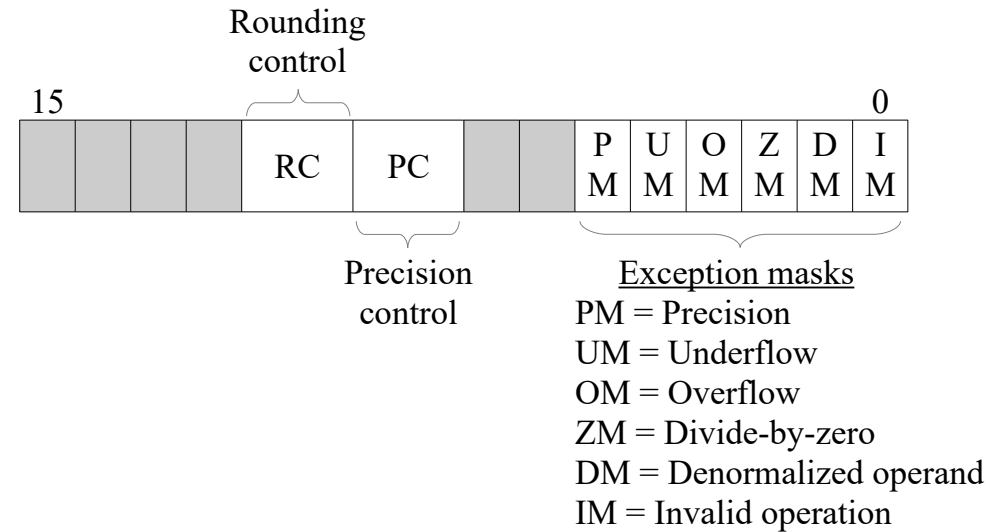
- osem registrov pre uloženie operandov s pohyblivou rádovou čiarkou (80 bitov)
- organizované vo forme zásobníka
- dostupné individuálne pod menami ST0, ST1, ..., ST7
 - mená nie sú priradené staticky (ST0 – register predstavujúci vrchol zásobníka, ST1 – ďalší v poradí, ...)
 - v stavovom registri FPU – ukazovateľ vrcholu zásobníka (TOS, 3 bity)
- stav a obsah registra indikovaný značkou (2 bity, register značiek (Tag register))

Riadiace a stavové registre

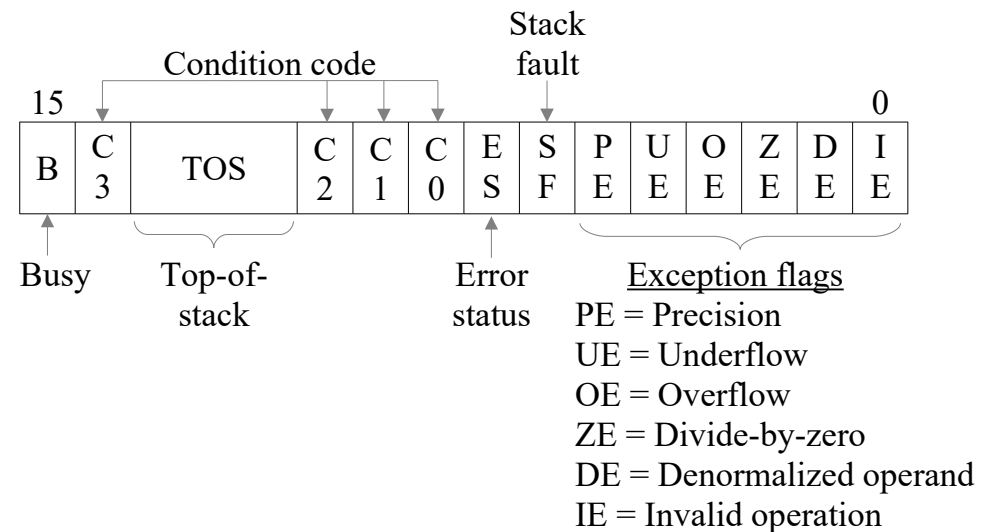
- tri registre rozmeru 16 bitov
 - Riadiaci register (FPU Control Register)
 - Stavový register (FPU Status Register)
 - Register značiek (Tag Register)

Riadiaci register [1]

- voľba parametrov činnosti FPU (control word)
- masky pre spracovanie výnimiek
 - Exception masks, 6 najnižších bitov
 - nastavený bit blokuje generovanie výnimky
- presnosť (PC, 2 bity)
 - zníženie internej presnosti
 - kompatibilita so staršími FPU
 - urýchlenie výpočtu
 - 00 – 24-bit, 01 – N/A, 10 – 53-bit, 11 – 64-bit
- zaokrúhľovanie (RC, 2 bity)
 - 00 – najbližšie
 - 01 – nadol, 10 – nahor
 - 11 – skrátenie (orezanie nadbytočných bitov)

*Stavový register* [1]

- stav jednotky FPU
- signalizácia povahy výsledku aritm. operácií (C0 – C3)
 - podobnosť s príznakmi registra FLAGS (C0~CF, C2~PF, C3~ZF)
 - C1 – podtečenie/pretečenie zásobníka
 - využitie pri realizácii podmienených skokov – kopírovanie do FLAGS registra CPU
 - uloženie stavu (napr. do registra AX – `fstsw`)
 - naplnenie FLAGS registra (`sahf`)
- identifikácia registra na vrchole zásobníka (TOS, 3 bity)
 - 8 údajových registrov FPU organizovaných cyklicky
 - TOS identifikuje register na vrchole
 - TOS aktualizovaný pri vkladaní/výbere údajov



- príznaky výnimiek (nastavené, ak sa vyskytne výnimka počas spracovania)
 - 6 najnižších bitov
 - IE – výnimku môžu spôsobiť: aritmetická operácia, operácia so zásobníkom (SF bit nastavený)
 - podtečenie ($C1 = 0$) / pretečenie ($C1 = 1$) zásobníka ďalej indikované bitom C1
 - OE, UE – číslo príliš veľké / malé
 - PE – číslo nie je možné reprezentovať presne
 - ZE – delenie nulou
 - DE – aritmetická inštrukcia sa pokúša spracovať denormalizovaný operand

Register značiek [1]

- informácia o stave a obsahu údajových registrov
- značka indikuje, či príslušný register je prázdny, ak nie je, identifikuje jeho obsah
 - 00 – platný (valid)
 - 01 – nula (zero)
 - 10 – špeciálny (invalid, infinity, denormal)
 - 11 – prázdny (empty)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST7	ST6	ST5	ST4	ST3	ST2	ST1	ST0								
Tag	Tag	Tag	Tag	Tag	Tag	Tag	Tag								

Inštrukcie jednotky FPU

- inštrukcie pre presuny údajov, aritmetické operácie, porovnanie, transcendentné funkcie a ďalšie
- uvádzame výber z množiny podporovaných inštrukcií, detaily napr. v [2, 3, 4]
- všetky mnemoniky FPU inštrukcií začínajú písmenom f
- operácie so zásobníkom POP/TOP

Presuny údajov

- vloženie (load) operandu do zásobníka FPU (FPU stack)

`fld src (src - FPU R/M; M:32/64/80-bit číslo v pohyblivej čiarke)`

- dekrementuje TOS a uloží `src` do (nového) ST0, čísla nižšej presnosti konvertované na internú presnosť (80-bit)
- ak je operandom FPU R, STi – pred zmenou TOS

- vloženie celého čísla

`fild src (src - 16/32-bit celé číslo v pamäti)`

- konvertuje číslo na internú presnosť a uloží do ST0

- inštrukcie pre vloženie konštánt (bez operandov) [1]

<code>fldz</code>	+0.0 do zásobníka FPU
<code>fldl</code>	+1.0 do zásobníka FPU
<code>fldpi</code>	π do zásobníka FPU
<code>fldl2t</code>	$\log_2 10$ do zásobníka FPU
<code>fldl2e</code>	$\log_2 e$ do zásobníka FPU
<code>fldlg2</code>	$\log_{10} 2$ do zásobníka FPU
<code>fldln2</code>	$\log_e 2$ do zásobníka FPU

- presun (store) hodnoty z vrcholu FPU zásobníka (bez odstránenia, TOP)

```
fst      dest          (dest - FPU R/M)
```

- pamäťový operand 32/64/80-bit (v prípade 32 a 64-bit – konverzia)

- výber hodnoty z vrcholu FPU zásobníka (s odstránením – inkrement TOS po výbere z ST0, POP)

```
fstp     dest          (dest - FPU R/M)
```

- ak je operandom FPU R, STi – pred zmenou TOS

Příklad:

```
fstp     ST1           ; odstráni ST1, ST0 ponechá na vrchole
fstp     ST0           ; odstráni ST0 (POP)
```

- celočíselná verzia – presun (konvertuje hodnotu v ST0 (znamienkové celé číslo) a uloží do pamäti)

```
fist     dest          (dest - M)
```

- celočíselná verzia – s výberom (POP)

```
fistp    dest
```

Aritmetické operácie

- operácia sčítania

```
fadd                                     (výber ST0 a ST1, sčítanie a uloženie sumy späť na FPU zásobník)
```

```
fadd     src          (src - FPU R/M, M:32/64-bit; ST0 ← ST0 + src)
```

```
fadd     dest,src      (src, dest - FPU R, jeden z nich ST0; dest ← dest + src)
```

```
faddp    dest,src      (src, dest - FPU R, src = ST0; dest ← dest + src; POP)
```

Příklad:

```
fldz
sum_loop:
dec     ECX
fadd    qword[EDX+ECX*8]
jnz     sum_loop
```

faddp dest (dest - FPU R, dest \leftarrow dest + ST0; POP)

fiadd src (src - M, 16/32-bit celé číslo)

- operácia odčítania

fsub (výber ST0 a ST1, výpočet ST1 - ST0, uloženie výsledku späť)

fsub src (src - M, 32/64-bit; ST0 \leftarrow ST0 - src)

fsub dest,src (src, dest - FPU R, jeden z nich ST0; dest \leftarrow dest - src)

fsubp dest,src (verzia s výberom ST0 (POP); src = ST0)

fsubp dest (dest - FPU R; dest \leftarrow dest - ST0, POP)

fsubr src (ST0 \leftarrow src - ST0, reverse subtract)

fisub src (src - M, 16/32-bit celé číslo; dostupné aj fisubr src)

- operácia násobenia

fmul (výber ST0 a ST1, výpočet ST0 * ST1, uloženie späť)

fmul src (src - M, 32/64-bit; ST0 \leftarrow ST0 * src)

fmul dest,src (src, dest - FPU R, jeden z nich ST0; dest \leftarrow dest * src)

fmulp dest,src (verzia s výberom ST0 (POP); src = ST0)

fmulp dest (dest - FPU R, dest \leftarrow dest * ST0; POP)

fimul src (src - M, 16/32-bit celé číslo; ST0 \leftarrow ST0 * src)

- operácia delenia

<code>fdiv</code>		(výber ST0 a ST1, výpočet ST1 / ST0, uloženie, dostupné aj <code>fdivr</code>)
<code>fdiv</code>	<code>src</code>	(<code>src</code> – M, 32/64-bit; $ST0 \leftarrow ST0 / src$)
<code>fdiv</code>	<code>dest,src</code>	(<code>src</code> , <code>dest</code> – FPU R, jeden z nich ST0; $dest \leftarrow dest / src$)
<code>fdivp</code>	<code>dest,src</code>	(verzia s výberom ST0 (POP); $src = ST0$)
<code>fdivr</code>	<code>src</code>	(<code>src</code> – M, 32/64-bit; $ST0 \leftarrow src / ST0$)
<code>fdivrp</code>	<code>dest</code>	(<code>dest</code> – FPU R; $dest \leftarrow ST0 / dest$, POP)
<code>fidiv</code>	<code>src</code>	(<code>src</code> – M, 16/32-bit celé číslo; dostupné aj <code>fidivr src</code>)

Operácie porovnania

- porovnanie dvoch čísel s pohyblivou rádovou čiarkou

`fcom` `src` (`src` – FPU R/M; M:32/64-bit)

- porovnanie ST0 a `src`, nastavenie príznakov stavového registra FPU
- príznaky C0, C2, C3 – indikácia relácie:

`fcom` (bez operandov)

- porovnanie ST0 s ST1, nastavenie príznakov
- porovnanie s výberom ST0 (`fcomp`), resp. s výberom ST0 a ST1 (`fcompp`)

- porovnanie ST0 s celým číslom v pamäti

`ficom` `src` (`src` – M, 16/32-bit)

- porovnanie s výberom (`ficomp src`)

Relácia	C3	C2	C0
$ST0 > src$	0	0	0
$ST0 = src$	1	0	0
$ST0 < src$	0	0	1
Not comparable	1	1	1

- porovnanie s nulou (ST0 s 0.0)

ftst

- preskúmanie povahy operandu v ST0 (examine)

fxam

- znamienko v C1 (1 – záporné), ďalšia informácia v príznakoch C0, C2 a C3 [1]

Typ	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal	1	1	0

Ďalšie operácie

- funkcie \sin (\cos) – výber argumentu (radiány) z FPU zásobníka, výpočet funkcie \sin (\cos), uloženie hodnoty späť

f sin (f cos)

- výpočet odmocniny z hodnoty na vrchole zásobníka (ST0) a nahradenie hodnoty výsledkom (záporná hodnota – výnimka)

fsqrt

- zmena (inverzia) znamienka čísla v ST0

fchs

- absolútna hodnota čísla v ST0 (výsledok v ST0)

fabs

- načítanie riadiaceho slova (control word, 16-bit) do riadiaceho registra FPU

fldcw src (src – M, 16-bit)

- uloženie riadiaceho slova (po vykonaní inštrukcie sú bity C0 – C3 nedefinované)

```
fstcw    dest    (dest - M, 16-bit)
```

- uloženie stavového slova (status word, 16-bit), C0 – C3 nedef.

```
fstsw    dest    (dest - M/register AX)
```

- kopírovanie AH do FLAGS registra CPU

```
sahf
```

Příklad:

```
...
fld      qword[EDX+ECX*8]    ; nový prvok v ST0
fcom     ST1                 ; stare maximum v ST1
fstsw    AX
sahf
jbe      cmp_min             ; bezznamienkove
fst      ST1                 ; nove maximum v ST1
cmp_min:
...
```

Příklad: Riešenie kvadratickej rovnice (C časť – interakcia s používateľom, ASM časť – výpočet koreňov) [1].

C časť:

```
#include    <stdio.h>

int main(void)
{
    double    a, b, c, root1, root2;
    extern int quad_roots(double, double, double, double*, double*);

    printf("Zadaj konstanty a, b, c: ");
    scanf("%lf %lf %lf",&a, &b, &c);

    if (quad_roots(a, b, c, &root1, &root2))
        printf("koren1 = %lf a koren2 = %lf\n", root1, root2);
    else
        printf("Neexistuju realne korene.\n");

    return 0;
}
```

Kvadratická rovnica:

$$ax^2+bx+c=0$$

Výpočet koreňov:

$$root\ 1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$root\ 2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

ASM časť:

<pre> %define a qword[EBP+8] %define b qword[EBP+16] %define c qword[EBP+24] %define root1 dword[EBP+32] %define root2 dword[EBP+36] segment .text global _quad_roots _quad_roots: enter 0,0 fld a ; a fadd ST0 ; 2a fld a ; a, 2a fld c ; c, a, 2a fmulp ST1 ; ac, 2a fadd ST0 ; 2ac, 2a fadd ST0 ; 4ac, 2a fchs ; -4ac, 2a fld b ; b, -4ac, 2a fld b ; b, b, -4ac, 2a fmulp ST1 ; b*b, -4ac, 2a faddp ST1 ; b*b-4ac, 2a ftst ; porovnaj b*b-4ac s 0 fstsw AX ; stavove slovo do AX sahf </pre>	<pre> jb no_real_roots fsqrt ; sqrt(b*b-4ac), 2a fld b ; b, sqrt(b*b-4ac), 2a fchs ; -b, sqrt(b*b-4ac), 2a fadd ST1 ; -b+sqrt(b*b-4ac), sqrt(b*b-4ac), 2a fddiv ST2 ; -b+sqrt(b*b-4ac)/2a, sqrt(b*b-4ac), 2a mov EAX, root1 fstp qword[EAX] ; ulozi koren1 fchs ; -sqrt(b*b-4ac), 2a fld b ; b, -sqrt(b*b-4ac), 2a fsubp ST1 ; -b-sqrt(b*b-4ac), 2a fddivrp ST1 ; -b-sqrt(b*b-4ac)/2a mov EAX, root2 fstp qword[EAX] ; ulozi koren2 mov EAX, 1 ; realne korene existuju jmp short done no_real_roots: sub EAX, EAX ; EAX=0 (ziadne realne korene) done: leave ret </pre>
--	--

```

C:\Dev-Cpp\bin\asm>quad
Zadaj konstanty a, b, c: 1 4 3
koren1 = -1.000000 a koren2 = -3.000000

C:\Dev-Cpp\bin\asm>quad
Zadaj konstanty a, b, c: 1 2 3
Neexistuju realne korene.

```

Študijná literatúra:

- [1] Dandamudi, S.P.: Introduction to Assembly Language Programming, Springer Science+Business Media, Inc., 2005.
- [2] Carter, A.P.: [PC Assembly Language](#), 2019.
- [3] [Intel® 64 and IA-32 Architectures Software Developer's Manual](#), Volume 2: Instruction Set Reference, Chapter 3, 2023.
- [4] [NASM](#) – The Netwide Assembler, The NASM Development Team, 1996-2015.
- [5] Wikipedia, [Floating-point unit](#), 2023.

Stručne o x86-64

- pôvodná špecifikácia vytvorená spoločnosťou AMD, 64 bitová verzia súboru inštrukcií x86 (1999, plná špecifikácia 08/2000) [1]
- prvé významné rozšírenie architektúry x86 navrhnuté inou spoločnosťou ako Intel (ten ju nakoniec prijal a implementoval)
- AMD64 ako evolúcia existujúcej x86 architektúry s podporou 64-bit výpočtov (Intel vytvoril vlastnú, nekompatibilnú IA-64)
- prvý procesor s podporou AMD64, Opteron, uvedený v apríli 2003 (prvý Intel x64 procesor Xeon, 2004)
- dva nové operačné (pod)režimy (*Long mode: 64-bit mode* + *Compatibility mode*) a nový 4-úrovňový systém stránkovania

64-bit mode

- podpora väčšieho množstva pamäte, rozšírenie rozmeru (64 bit) a počtu (16) GPR registrov, ďalšie rozšírenia
- operácie s pohyblivou čiarkou – preferované SSE(2) inštrukcie, avšak x87/MMX stále dostupné
- vektorové registre (16 x 128 bit, neskôr rozšírené aj tie), rôzne formáty údajov
- podpora 64 bitových operandov a 64 bitového režimu adresovania

Compatibility mode

- podpora behu 16 a 32 bitových aplikácií (16/32 bitové inštrukcie stále dostupné)
- x86-64 procesor stále štartuje v reálnom režime (real mode) za účelom spätnej kompatibility

Základné vlastnosti architektúry

- GPR rozšírené na 64 bitov, ako aj príslušné (celočíselné) operácie (aritmetické, logické, presuny údajov)
- ďalšie registre (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8, R9, ... R15)
- ďalšie XMM (SSE) registre (128 bitov) – z pôvodných 8 na 16 (XMM8 – XMM15) [2]
- väčší virtuálny adresný priestor – 64-bitový formát virtuálnej adresy (aktuálne využívaných spodných 48 bitov)
- väčší fyzický adresný priestor – pôvodná špecifikácia 40-bitové fyzické adresy, aktuálne implementácie 48-bitové
- väčší fyzický adresný priestor aj v Legacy mode
- RIP relatívne adresovanie (position independent code, x86 – iba inštrukcie skokov, v x86-64 aj prístup k údajom) [6]
- SSE inštrukcie (adoptovanie SSE a SSE2 od spoločnosti Intel)
- No-execute bit (NX bit) – OS môže určiť, ktoré stránky môžu obsahovať vykonateľný kód a ktoré nie (podobná možnosť bola dostupná aj prípade využitia segmentových deskriptorov (avšak len pre celý segment); moderné OS segmentáciu obchádzajú)
- odstránenie/redukcia niektorých vlastností (segmentované adresovanie, TSS mechanizmus, virtual 8086 mode – ostávajú prítomné v Legacy mode); niektoré inštrukcie (uloženie/obnova SR (CS, SS, DS, ES) na zásobníku, PUSH/POPA, BOUND, INTO ...) [1,3]

Operačné režimy architektúry (ďalšie detaily napr. v [1])

Operačný		Požadovaný OS	Kód
režim	pod-režim		
Long mode	64-bit mode	64-bit OS, 64-bit UEFI	64-bit
	Compatibility mode	Bootloader alebo 64-bit OS	32-bit
			16-bit protected mode
Legacy mode	Protected mode	Bootloader, 32-bit OS, 32-bit UEFI	32-bit
		16-bit protected mode OS	16-bit protected mode
	Virtual 8086 mode	16-bit protected mode OS alebo 32-bit OS	podmnožina real mode
	Unreal mode	Bootloader alebo real mode OS	real mode
	Real mode	Bootloader, real mode OS, OS využívajúci BIOS rozhranie	real mode

Registre [6]

RAX		
zero-extended	EAX	
not modified	AX	
not modified	AH	AL

R8		
zero-extended	R8D	
not modified	R8W	
not modified	R8B/L	

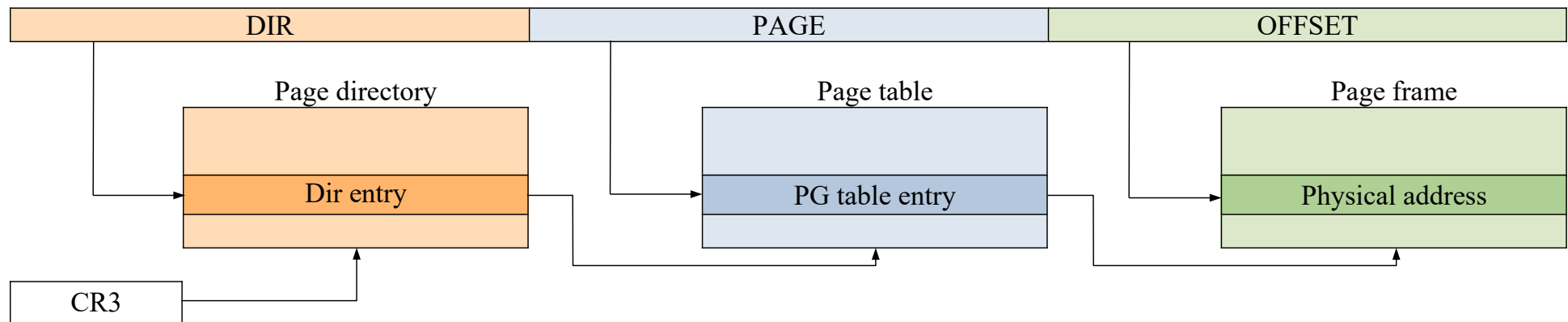
- NASM štandardne podporuje názvy R8B – R15B [7]
 - v niektorých dokumentoch (Intel) použité názvy R8L – R15L
 - možnosť použiť makrá v NASM pre názvy R8L – R15L
- SIMD registre neskôr ďalej rozširované (rozmer, počet – XMM (128 bitov), YMM (256 bitov), ZMM (512 bitov)) [8]

Stránkovanie (x86)

- transformácia lineárnej adresy na fyzickú (voliteľná, bit PG v registri CR0) [4]
- stránka (*page frame*) – 4KB súvislý blok fyzickej pamäti
- lineárna adresa – odkazuje nepriamo na fyzickú adresu (špecifikuje tabuľku stránok, stránku v rámci tejto tabuľky a offset)

31	22	21	12	11	0
DIR		PAGE			OFFSET

- spôsob prevodu polí DIR, PAGE a OFFSET na fyzickú adresu (2 úrovne tabuliek)
 - DIR – index (10 bitov) do adresára stránok (*Page directory*)
 - PAGE – index (10 bitov) do tabuľky stránok (*Page table*) určenej adresárom stránok
 - OFFSET – 12 bitov, adresovanie bajtu v rámci stránky (určenej z tabuľky stránok)

**Tabuľka stránok (page table)**

- pole 32-bitových špecifikátorov stránok (tiež stránka, 4KB, max. 1024 položiek)
- dve úrovne tabuliek
 - prvá úroveň – adresár stránok (max. 1K tabuliek stránok druhej úrovne)
 - druhá úroveň – tabuľka stránok, adresovanie max. 1K stránok

- všetky tabuľky adresované 1 adresárom: 1M stránok (2^{20})
- každá stránka má rozmer 4KB (2^{12} B), tabuľky jedného adresára môžu obsiahnuť celý 32-bitový adresný priestor (2^{32} B)
- zmena adresára stránok – zmenou registra CR3

Položky tabuľky stránok

- položky majú jednotný formát pre každú z úrovní
- *page frame address* – fyzická začiatková adresa stránky (vyšších 20 bitov)
- stránky začínajú na hraniciach 4K, t.j. nižších 12 bitov nevyužitých (pre adresovanie)
 - tieto bity využité na uloženie ďalších informácií (napr. P – present, ak P = 0 položka nie je platná pre prevod adresy) [4]

Ďalší vývoj stránkovania

- PAE (*Physical Address Extension*) – zavádza 3-úrovňovú hierarchiu tabuliek (namiesto 2-úrovňovej), Intel Pentium Pro, 1995 [5]
 - položka tabuľky stránok (*Page directory* aj *Page table*) má veľkosť 64 bitov
 - pribudla ďalšia tabuľka (*Page directory pointer table*) so 4 položkami, [pôvodná implementácia](#):
 - pole *page frame address* – rozšírené z 20 na 24 bitov (veľkosť offset-u zostáva 12 bitov)
 - zvýšenie veľkosti fyzických adries z 32 na 36 bitov (fyzická pamäť adresovaná CPU 4GB → 64GB)
 - aplikácie – naďalej 32-bitové adresy (limit 4GB vo flat režime)
 - OS mohol mapovať tento virtuálny priestor na fyzickú pamäť (max. 64GB)
- x86-64 CPU (*Long mode*) – ďalej rozširuje hierarchiu stránkovacích tabuliek na 4 úrovne
- v procesoroch, ktoré podporujú “no-execute“, NX bit je najvyšší bit (bit 63) položky tabuľky stránok

Formát inštrukcií

V 64 bitovom režime využívané aj ďalšie prefixy [11,12]:

- REX prefix (špecifikácia GPR a SSE registrov, 64 bitových operandov, ...)
 - nie všetky inštrukcie vyžadujú prefix v 64-bit režime
- VEX prefix (kódovanie AVX inštrukcií) / EVEX (AVX-512 inštrukcií)

Programovanie

- ABI (Application Binary Interface) – parametre, návratová hodnota, zásobník ... [6,10]
- dôležité pri volaní C funkcií, služieb OS ...
- volacie konvencie – Microsoft x64 ABI (Windows) [9], System V x64 ABI (Linux, BSD, Mac)

Microsoft x64 ABI

- prvé 4 parametre v registroch, ďalšie v zásobníku
 - RCX, RDX, R8, R9 (celočíselné)
 - XMM0, XMM1, XMM2, XMM3 (s plávajúcou čiarkou)
- pre funkcie s premenlivým počtom parametrov (*vararg*) – hodnoty s plávajúcou čiarkou duplikované v zodpovedajúcich GPR
- návratová hodnota v RAX alebo XMM0 (*float, double, ...*)
- hodnoty v niektorých registroch môžu byť funkciou zmenené (*volatile* – RAX, RCX, RDX, R8, R9, R10, R11)
- hodnoty niektorých registrov musí funkcia zachovať (push/pop) (*non-volatile* – RBX, RBP, RDI, RSI, R12, R13, R14, R15)
- zásobník
 - alokovanie miesta pre uložené registre, lokálne premenné, parametre
 - zarovnaný na 16B (*non-leaf* funkcia)
 - volajúci (caller) alokuje 32B (*shadow space*) pred volaním funkcie (pre registre RCX, RDX, R8, R9)
 - volajúci vyčistí zásobník (po návrate z volanej funkcie)

Príklad: ilustrácia vlastností operácií na registroch v 64 bitovom režime.

```
G:\x86-64_ex>nasm -f win64 ex1asm64_regs.asm
G:\x86-64_ex>gcc ex1asm64_regs.obj -o ex1asm64_regs
G:\x86-64_ex>ex1asm64_regs.exe
11111111111111110
```

Po zmene (dec r8d):

```
G:\x86-64_ex>nasm -f win64 ex1asm64_regs.asm
G:\x86-64_ex>gcc ex1asm64_regs.obj -o ex1asm64_regs
G:\x86-64_ex>ex1asm64_regs.exe
0000000011111110
```

```
global main
extern printf
section .text
pformat: db "%.16llx",13,10,0

main:
    sub     rsp,28h ; vyhradenie a zarovnanie
    mov     r8,1111111111111111h
    dec     r8b     ;dec r8d
    mov     rdx,r8
    lea     rcx,pformat
    call    printf
    add     rsp,28h
    ret
```

Príklad: ilustrácia využitia vektorových registrov (XMM) a SIMD inštrukcií pri realizácii operácií s plávajúcou čiarkou.

```
G:\x86-64_ex>nasm -f win64 fpcalc64_vsum4.asm
G:\x86-64_ex>gcc -o fpcalc64_vsum4 fpcalc64_vsum4.obj
G:\x86-64_ex>fpcalc64_vsum4.exe
The sum of 2.500000 and 3.100000 is 5.600000
The sum of 3.600000 and 4.200000 is 7.800000
```

- SSE2 inštrukcia `movupd` pre presun zabalených (packed) čísel s plávajúcou čiarkou s dvojitou presnosťou
- SSE2 inštrukcia `addpd` pre realizáciu sčítania zabalených čísel s plávajúcou čiarkou s dvojitou presnosťou [3]

```
global main
extern printf

section .data
nums1 dq 2.5,3.6
nums2 dq 3.1,4.2
nums3 dq 0.0,0.0
f_sum db "The sum of %f and %f is %f",10,0

section .text
main:
    sub    rsp,28h
    movupd xmm0,[nums1]
    addpd  xmm0,[nums2]
    movupd [nums3],xmm0
    mov    rcx,f_sum
    mov    rdx,[nums1]
    mov    r8,[nums2]
    mov    r9,[nums3]
    call   printf
    mov    rcx,f_sum
    mov    rdx,[nums1+8]
    mov    r8,[nums2+8]
    mov    r9,[nums3+8]
    call   printf
    add    rsp,28h
    ret
```


Príklad: ilustrácia vyhradenia miesta a zarovnania údajov v zásobníku.

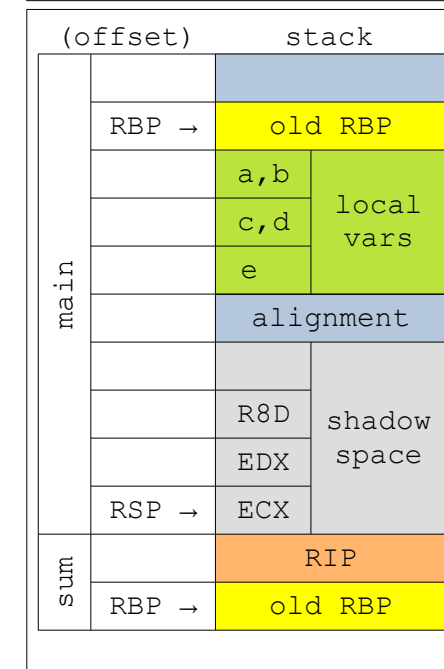
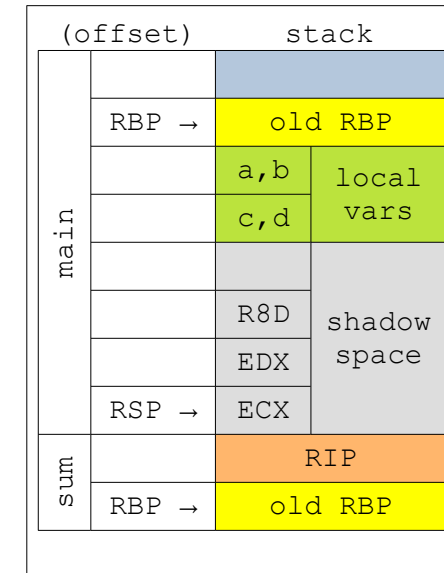
```
#include <stdio.h>
long sum(long a, long b, long c);

void main()
{
    long a = 1;
    long b = 2;
    long c = 3;
    long d = 4;
    c = sum(a,b,c);
    asm("movl %%esp,%0" : "=r" (d));
    printf("Sum: %lx Stack: %lx", c, d);
}

long sum(long a, long b, long c)
{
    return a + b + c;
}
```

```
sum:
    push rbp
    mov rbp, rsp
    mov DWORD PTR 16[rbp], ecx
    mov DWORD PTR 24[rbp], edx
    mov DWORD PTR 32[rbp], r8d
    mov edx, DWORD PTR 16[rbp]
    mov eax, DWORD PTR 24[rbp]
    add edx, eax
    mov eax, DWORD PTR 32[rbp]
    add eax, edx
    pop rbp
    ret
    .ident "GCC: (GNU) 9.1.0"
```

```
.LC0:
    .ascii "Sum: %lx Stack: %lx\0"
    ...
main:
    push rbp
    mov rbp, rsp
    sub rsp, 48
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov ecx, DWORD PTR -12[rbp]
    mov edx, DWORD PTR -8[rbp]
    mov eax, DWORD PTR -4[rbp]
    mov r8d, ecx
    mov ecx, eax
    call sum
    mov DWORD PTR -12[rbp], eax
/APP
# 11 "ex2_stack_rsp.c" 1
    movl %esp,eax
# 0 "" 2
/NO_APP
    mov DWORD PTR -16[rbp], eax
    mov edx, DWORD PTR -16[rbp]
    mov eax, DWORD PTR -12[rbp]
    mov r8d, edx
    mov edx, eax
    lea rcx, .LC0[rip]
    call printf
    add rsp, 48
    pop rbp
    ret
```



Ako sa zmení situácia v zásobníku, ak pridáme lokálnu premennú (long e = 5;)?

Sum: 6 Stack: 61fdf0

Sum: 6 Stack: 61fde0

System V x64 ABI

- 6 celočíselných parametrov v RDI, RSI, RDX, RCX, R8, R9
- 8 registrov pre parametre s plávajúcou čiarkou XMM0 – XMM7 (*variadic* funkcie – počet použitých vektorových registrov v RAX)
- ďalšie parametre v zásobníku; návratové hodnoty – celočíselné RAX (RDX), s plávajúcou čiarkou XMM0 (XMM1)
- bez *home/shadow space* ako je to v prípade Microsoft x64 ABI
- využívané zarovnanie (hranica 16B)
- funkcie, ktoré nevolajú ďalšie funkcie (*leaf-node functions*) – môžu využiť tzv. *red zone* (128B oblasť pod RSP) [10]

Příklad: ilustrácia programovania v prostredí OS Linux.

```
simonak@hron:~/ASM$ nasm -f elf64 hello64c.asm
simonak@hron:~/ASM$ gcc hello64c.o -no-pie
simonak@hron:~/ASM$ ./a.out
Hello world from Linux!
simonak@hron:~/ASM$
```

```
global  main
extern  printf

section .data
msg     db      "Hello world from Linux!",10,0

section .text
main:
        push    rbp
        mov     rbp, rsp
        mov     rdi, msg
        mov     rax, 0
        call    printf
        mov     rsp, rbp
        pop     rbp
        ret
```

Zdroje:

- [1] Wikipedia, [x86-64](#), 2023.
- [2] Wikipedia, [Streaming SIMD Extensions](#), 2023.
- [3] Cloutier, F.: [x86 and amd64 instruction reference](#), 2022.
- [4] Intel 80386 Reference Programmer's Manual, [Page Translation](#), 1986.
- [5] Wikipedia, [Physical Address Extension](#), 2023.
- [6] Larimer, J.: [Intro to x64 Reversing](#), SummerCon 2011.
- [7] NASM – The Netwide Assembler, Documentation, [Chapter 12: Writing 64-bit Code \(Unix, Win64\)](#), 2020.
- [8] Wikipedia, [Registers available in the x86-64 instruction set](#), 2014.
- [9] Microsoft technical documentation, [x64 calling convention](#), 2022.
- [10] Wikipedia, [x86 calling conventions](#), 2022.
- [11] [Intel® 64 and IA-32 Architectures Software Developer's Manual](#), Intel Corporation, 2023.
- [12] OSDev Wiki, [X86-64 Instruction Encoding](#), 2021.