6. Architektura procesoru – předávání řízení, instrukce bitových operací, ostatní instrukce

BT, BTS, BTR, BTC, BSF, BSR

instrukce pro práci s bity

BT, BTS, BTR, BTC
instrukce pro práci s bity

```
1. BT/BTS/BTR/BTC r16/32,r16/32
2. BT/BTS/BTR/BTC r16/32,imm16/32
3. BT/BTS/BTR/BTC m16/32,r16/32
4. BT/BTS/BTR/BTC m16/32,imm16/32
```

```
BT AX,BX

BTS EBX,EDI

BTC dword [ptr1], 5

BTR dword [ptr2], EAX
```

```
BT/BTS/BTR/BTC dest,src
(Bit Test and Set/Reset/Complement)
CF = dest[src];
dest[src] = 1;
dest[src] = 0;
dest[src] = ~dest[bit src];
```

Uloží hodnotu bitu ležícího v cílovém operandu na pozici dané zdrojovým operandem do příznaku CF (ostatní příznaky jsou nedefinovány).
BTS – zároveň nastaví tento bit na 1
BTR – zároveň nastaví tento bit na 0
BTC – zároveň invertuje tento bit

r16 ∈ {AX,BX,CX,DX,SI,DI,BP,SP}, r32 ∈ {EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP}
mem = paměť, imm = přímý operand (konstanta)

BSF, BSR

instrukce pro práci s bity

1. BSF/BSR r16/32,r16/32
2. BSF/BSR r16/32,mem16/32

Nalezení prvního nenulového bitu zleva/zprava ve zdrojovém operandu.

```
BSF/BSR dest,src (Bit Scan Forward/Reverse)
ZF=1;
for(int i=0; i < sizeof(src); i++) {
for(int i=sizeof(src); i >= 0; --i) {
  if (src[bit i] == 1) {
    ZF = 0; dest = i; break;
  }
}
```

```
BSF AX,BX
BSR EBX,EDI
BSF ECX,[ptr1]
BSR BX,0x0100 nelze!!!
```

r16 ∈ {AX,BX,CX,DX,SI,DI,BP,SP}, r32 ∈ {EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP}, mem = paměť

LEA, XLATB, NOP

ostatní instrukce

LEA
aritmetická instrukce

1. LEA r16/32,mem16/32

LEA dest,src (*Load Effective Address*)
dest = effective address of src;

```
LEA AX,[BP + 6]              … AX = BP + 6
LEA EBX,[ESI + 1000]         … EBX = ESI + 1000
LEA ECX,[EAX + 4*EBX + 10]… ECX = EAX + 4*EBX + 10
```

Vyčíslí efektivní adresu danou druhým operandem a uloží vypočítanou hodnotu adresy do cílového registru.

r16 ∈ {AX,BX,CX,DX,SI,DI,BP,SP}, r32 ∈ {EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP}, mem = paměť

XLATB

přenosová instrukce

1. XLATB

Do AL vloží hodnotu z paměti uloženou na adrese EBX zvýšené o původní obsah registru AL.

```
XLATB (Table Look-up Translation)
AL = DS:[EBX + ZeroExtend(AL)];
```

NOP

prázdná operace

1. NOP

NOP ~ XCHG EAX,EAX.

```
NOP (No Operation)
tmp = EAX;
EAX = tmp;
```

r16 ∈ {AX,BX,CX,DX,SI,DI,BP,SP}, r32 ∈ {EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP}, mem = paměť

PUSH/POP, PUSHA/POPA, PUSHAD/POPAD, PUSHF/POPF, PUSHFD/POPFD

přenosové instrukce

Zásobník

- zásobník je část paměti, na vrchol zásobníku ukazuje SS:ESP
  - budeme pracovat pouze s ESP
- se zásobníkem pracují instrukce PUSH a POP
  - PUSH *hodnota* uloží hodnotu na vrchol zásobníku
    - sníží hodnotu ESP o 4 byty (`SUB ESP,4`)
    - na adresu ESP uloží hodnotu (`MOV dword [ESP],hodnota`)
  - POP *dest* odstraní hodnotu z vrcholu zásobníku a uloží ji do cílového operandu
    - přečte hodnotu z adresy ESP a uloží ji do cíle (`MOV dest,[ESP]`)
    - zvýší hodnotu ESP o 4 byty (`ADD ESP,4`)
- ze zásobníku lze číst jako z jakékoliv jiné paměti (tedy ne jen z vrcholu)

`MOV EAX,[ESP]` … přečte hodnotu z vrcholu zásobníku (a nechá ji tam)
`POP EAX` … přečte hodnotu z vrcholu zásobníku (a odstraní ji = zvýší ESP o 4)
`MOV EBX,[ESP+4]` … přečte hodnotu, které předchází hodnotě na vrcholu

PUSH src

(*Push word/doubleword onto the stack*) – implicitně ukládá 32bitů

1. PUSH (s)reg16/reg32
2. PUSH mem16/32
3. PUSH imm8/16/32

POP src

(*Pop word/doubleword from the stack*)

1. POP (s)reg16/reg32
2. POP mem16/32

r8 ∈ {AL,BL,CL,DL,AH,BH,CH,DH},
r16 ∈ {AX,BX,CX,DX,SI,DI,BP,SP},
r32 ∈ {EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP},
sreg ∈ {CS,DS,ES,FS,GS,SS},
(s)reg16 = reg16 ∪ sreg,
mem = paměť, imm = přímý operand (konstanta)

- Implicitně ukládá/čte 32 bitů, pokud nevynutíme 16 bitů prefixem – snažíme se toho vyvarovat = ve 32bitovém režimu nepoužíváme **PUSH/POP** reg16:
  - je to nesystémové, znepřehlední to program, vnáší to chyby do programu
- Dvojice PUSH/POP může sloužit k uložení a obnovení obsahu registru.

  PUSH EAX …… POP EAX
- Pozor na zachování pořadí:

  PUSH EBX
  PUSH ECX
  …
  POP ECX
  POP EBX

PUSH src

```
if (sizeof(src) == 16) {
    ESP = ESP – 2;
    mem[ESP] = SignExtend(src);
} else {
    ESP = ESP – 4;
    mem[ESP] = SignExtend(src);
}
```

```
PUSH ECX
PUSH BX
PUSH dword 0x0100
PUSH dword [0x0100]
```

Pokud chceme posunout vrchol zásobníku (~ rezervovat místo na zásobníku), můžeme přímo modifikovat ukazatel na vrchol zásobníku ESP:

SUB ESP,4

		adr	mem[*adr*]	
		FFFFFFFF	67	
		FFFFFFFE	66	
		FFFFFFFD	65	
PUSH EAX	ESP_1	FFFFFFFC	64	
	-4	FFFFFFFB	18	11
		FFFFFFFA	56	22
		FFFFFFF9	98	33
PUSH AX	ESP_2	FFFFFFF8	55	44
	-2	FFFFFFF7	85	77
	ESP_3	FFFFFFF6	78	88

EAX = 11 22 33 44

ESP_1 = FF FF FF FC

byte [ESP_1] = 64

PUSH EAX

ESP_2 = FF FF FF F8

byte [ESP_2] = 44

EAX = 55 66 77 88

ESP_2 = FF FF FF F8

byte [ESP_2] = 44

PUSH AX

ESP_3 = FF FF FF F6

byte [ESP_3] = 88

POP dst

```
if (sizeof(dst) == 16) {
    dst = (word) mem[ESP];
    ESP = ESP + 2;
} else {
    dst = (dword) mem[ESP];
    ESP = ESP + 4;
}
```

```
POP ECX
POP BX
POP dword [0x0100]
POP 0x0100 nelze!!!
```

Pokud chceme odebrat („zahodit“) hodnotu z vrcholu zásobníku, můžeme přímo modifikovat ukazatel na vrchol zásobníku ESP:

ADD ESP,4

Takto lze „zahodit“ i více hodnot:

ADD ESP,n

		adr	mem[*adr*]
		FFFFFFFF	67
		FFFFFFFE	66
		FFFFFFFD	65
	ESP_1	FFFFFFFC	64
	+4	FFFFFFFB	11
		FFFFFFFA	22
		FFFFFFF9	33
POP EAX	ESP_2	FFFFFFF8	44
	+2	FFFFFFF7	77
POP AX	ESP_3	FFFFFFF6	88

EAX = AA BB 33 44
ESP_3 = FF FF FF F6
byte [ESP_3] = 88

POP AX

EAX = AA BB 77 88
ESP_2 = FF FF FF F8

EAX = AA BB 77 88
ESP_2 = FF FF FF F8
byte [ESP_2] = 44

POP EAX

EAX = 11 22 33 44
ESP_1 = FF FF FF FC

Zásobník - příklady

```
PUSH dword 10
POP EAX
PUSH dword -1
PUSH dword 6
PUSH EAX
MOV ECX,[ESP]
MOV EDX,[ESP+4]
POP EBX
PUSH dword 55
MOV [ESP],100
POP EAX
```

	adr	mem[*adr*]
	FFFFFFFF	
	FFFFFFFE	
	FFFFFFFD	
	FFFFFFFC	
	FFFFFFFB	
	FFFFFFFA	
	FFFFFFF9	
	FFFFFFF8	
	FFFFFFF7	
	FFFFFFF6	
	FFFFFFF5	
	FFFFFFF4	
	FFFFFFF3	
	FFFFFFF2	
	FFFFFFF1	
	FFFFFFF0	

PUSHA/POPA/PUSHAD/POPAD PUSHF/POPF/PUSHFD/POPFD

přenosové instrukce, instrukce pro práci s příznaky

Implicitně ukládá/čte 32 bitů, pokud nevynutíme 16 bitů prefixem.

- PUSHAD/POPAD uloží/načte obsah všech registrů pro obecné použití na zásobník/ze zásobníku
- PUSHFD/POPFD uloží/načte obsah všech registru EFLAGS na zásobník/ze zásobníku

PUSHA(D)

```
PUSH (E)AX
PUSH (E)CX
PUSH (E)DX
PUSH (E)BX
PUSH original (E)SP
PUSH (E)SI
PUSH (E)DI
```

POPA(D)

```
POP (E)DI
POP (E)SI
(E)SP += (4) 2
POP (E)BX
POP (E)DX
POP (E)CX
POP (E)AX
```

PUSHF(D)

```
PUSH (E)FLAGS
```

POPF(D)

```
POP (E)FLAGS
```

CALL/RET
INT/IRET
instrukce předávání řízení

CALL (nepodmíněný skok – volání)

- volání je skok, který ukládá na zásobník návratovou adresu
- volání uvnitř jednoho (kódového) segmentu (mění se pouze EIP):
  - blízký (*near jump*) = skok v rámci aktuálního segmentu
- volání do jiného (kódového) segmentu (mění se CS a EIP):
  - vzdálený (*far jump*)
- volání dále může být
  - přímý (*direct*) x nepřímý (*indirect*)

```
1. CALL rel       … relativní, přímé, blízké
2. CALL reg       … absolutní, nepřímé, blízké
3. CALL [mem]     … absolutní, nepřímé, blízké/vzdálené
4. CALL seg:ofs   … absolutní, přímé, vzdálené
```

```
CALL near rel
PUSH EIP
EIP += rel;
```

```
CALL near reg/[mem]
PUSH EIP
EIP = reg/[mem];
```

```
CALL far seg:ofs/[mem]
PUSH ZeroExtend32(CS)
PUSH EIP
CS = seg/[mem+4];
EIP = ofs/[mem];
```

RET/RETF imm (nepodmíněný skok – návrat - *return*)

- návrat je skok, který předává řízení na adresu uloženou na zásobníku – RET = blízký, RETF = vzdálený
- volitelný operand udává, kolik má být po návratu „uklizeno“ ze zásobníku bytů (== o kolik bytů má být zvýšena hodnota ESP)

```
1. RET/RETF
2. RET/RETF imm16
```

```
RET
POP EIP
```

```
RET n
POP EIP
ADD ESP,n
```

```
RETF
POP EIP
POP CS
```

```
RETF n
POP EIP
POP CS
ADD ESP,n
```

CALL a RET tvoří funkční dvojici:

- CALL
  - uloží návratovou adresu (=adresa instrukce následující za CALL) na zásobník
  - skočí na zadanou adresu
- RET
  - přečte adresu, kam skočit z vrcholu zásobníku
  - skočí na přečtenou adresu

INT *n* + **IRET** (nepodmíněný skok – volání přerušení a návrat)

- přerušení (*Interrupt* – INT) je **vzdálený** skok, který ukládá na zásobník (vzdálenou) návratovou adresu a **příznaky**
- návrat z přerušení (*Interrupt Return* – IRET) je skok, který předává řízení na **vzdálenou** adresu uloženou na zásobníku

1. INT imm8
2. INT3 … 1bytová varianta INT 3
3. INTO

```
INT n/INT3 (… INT 3)/INTO (… INT 4)
PUSHFD
PUSH ZeroExtend32(CS)
PUSH EIP
CS = segment adresy přerušení n
EIP = ofset adresy přerušení n
```

1. IRET

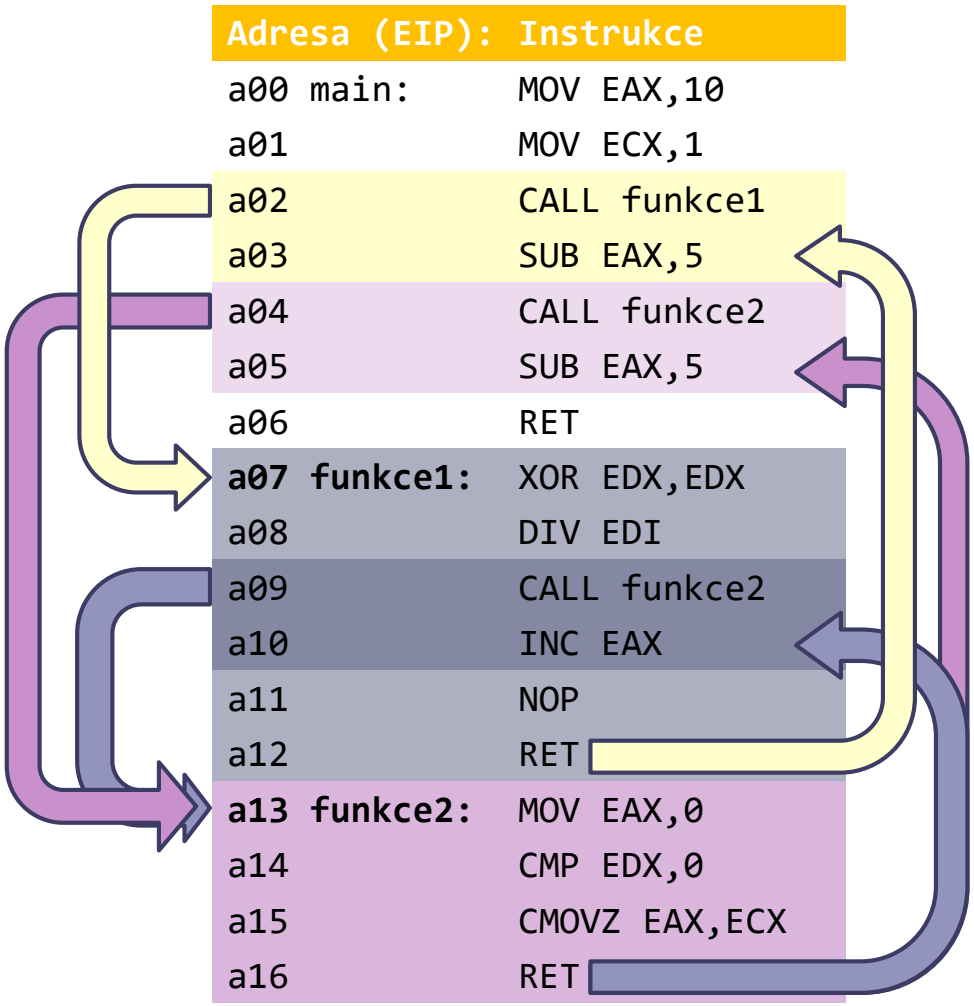
```
IRET
POP EIP
POP CS
POPFD
```

INT a IRET tvoří funkční dvojici, podobně jako CALL a RET, rozdíl = ukládá stav příznakového registru a je pouze vzdálený.

Volání funkcí

Volání funkce/návrat z funkce

- volání funkce = nepodmíněný skok – volání – CALL
- návrat z funkce = nepodmíněný skok – návrat – RET



Adresa (EIP):	Instrukce
a00 main:	MOV EAX,10
a01	MOV ECX,1
a02	CALL funkce1
a03	SUB EAX,5
a04	CALL funkce2
a05	SUB EAX,5
a06	RET
a07 funkce1:	XOR EDX,EDX
a08	DIV EDI
a09	CALL funkce2
a10	INC EAX
a11	NOP
a12	RET
a13 funkce2:	MOV EAX,0
a14	CMP EDX,0
a15	CMOVZ EAX,ECX
a16	RET

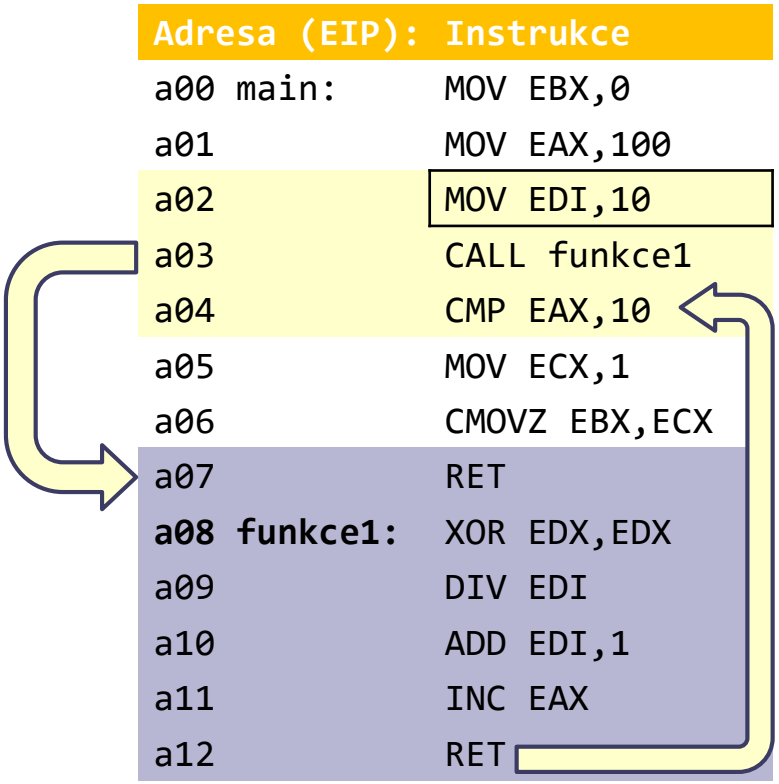
ADRESA	mem[ADRESA]
FFFFFFFF	
FFFFFFFE	
FFFFFFFD	
FFFFFFFC	
FFFFFFFB	
FFFFFFFA	
FFFFFFF9	
FFFFFFF8	
FFFFFFF7	
FFFFFFF6	
FFFFFFF5	
FFFFFFF4	
FFFFFFF3	
FFFFFFF2	
FFFFFFF1	
FFFFFFF0	
…	…

Předávání parametrů funkcím

- Funkce ke své činnosti potřebují parametry => voláme-li funkci, musíme jí předat parametry. Jak?
- Parametry předáváme funkci:
  - **v globálních proměnných** (paměť)
    - globální proměnné = jakákoliv změna, kterou na nich funkce provede, se projeví globálně => musíme dávat pozor
  - **v registrech**
    - registry ~ globální proměnné (problémy viz výše)
    - málo registrů => musím uchovat jejich obsah (bere paměť a čas procesoru)
  - **na zásobníku** (= paměť, ale lokální)
    - řeší problém globálnosti = jejich změna se neprojeví – po ukončení funkce se parametry ze zásobníku „uklidí“
- Předávání parametrů přes zásobník je nejčastěji používaná metoda předávání parametrů.

Předávání parametrů a výsledků přes registry

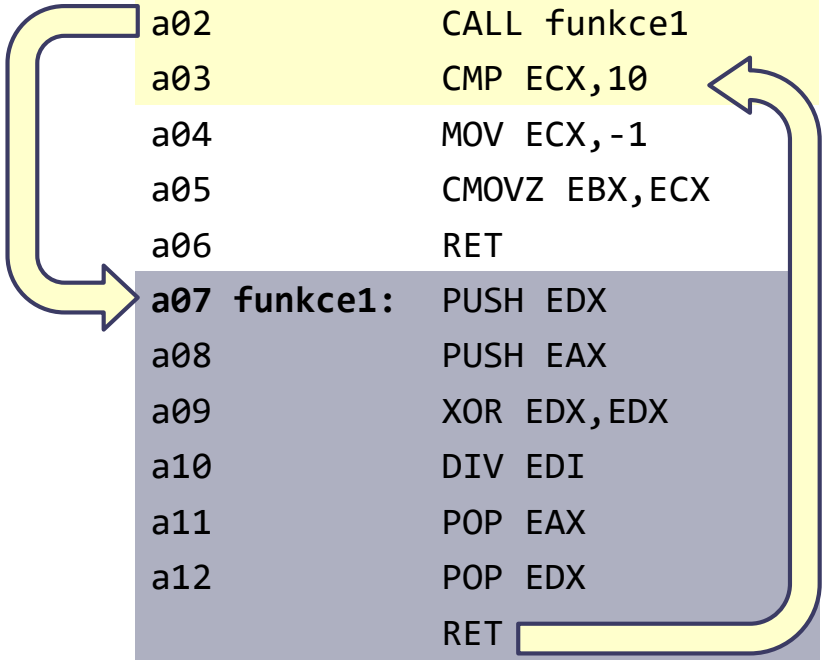
- registr ~ globální proměnná => platí všechna rizika spojená s použitím registrů
  - změna registru ve volané funkci = zničení obsahu pro volajícího
- musíme jednoznačně definovat, ve kterých registrech jsou vstupy a výstupy funkce



- Funkce „*funkce1*“ má jeden parametr a ten je uložen v registru EDI
- `EDI = ?, EAX = ?, EBX = ?`
- Změna obsahu registru EDI ve funkci znamená změnu obsahu registru EDI všude

Přechodné uložení obsahu registrů

- obsahy registrů stále jsou (a vždy budou) globální – řešení = přechodné „uchování“ obsahu registrů na zásobníku
  - schováme obsah registrů, které měníme (pozor na pořadí PUSH/POP)
  - nebo schováme obsah všech registrů (pro líné – PUSHAD/POPAD)
  - lze schovat i obsah příznakového registru (PUSHFD/POPFD)



```
Adresa (EIP): Instrukce
a00 main:     MOV EAX,10
a01           MOV EDI,10
a02           CALL funkce1
a03           CMP ECX,10
a04           MOV ECX,-1
a05           CMOVZ EBX,ECX
a06           RET
a07 funkce1:  PUSH EDX
a08           PUSH EAX
a09           XOR EDX,EDX
a10           DIV EDI
a11           POP EAX
a12           POP EDX
              RET
```

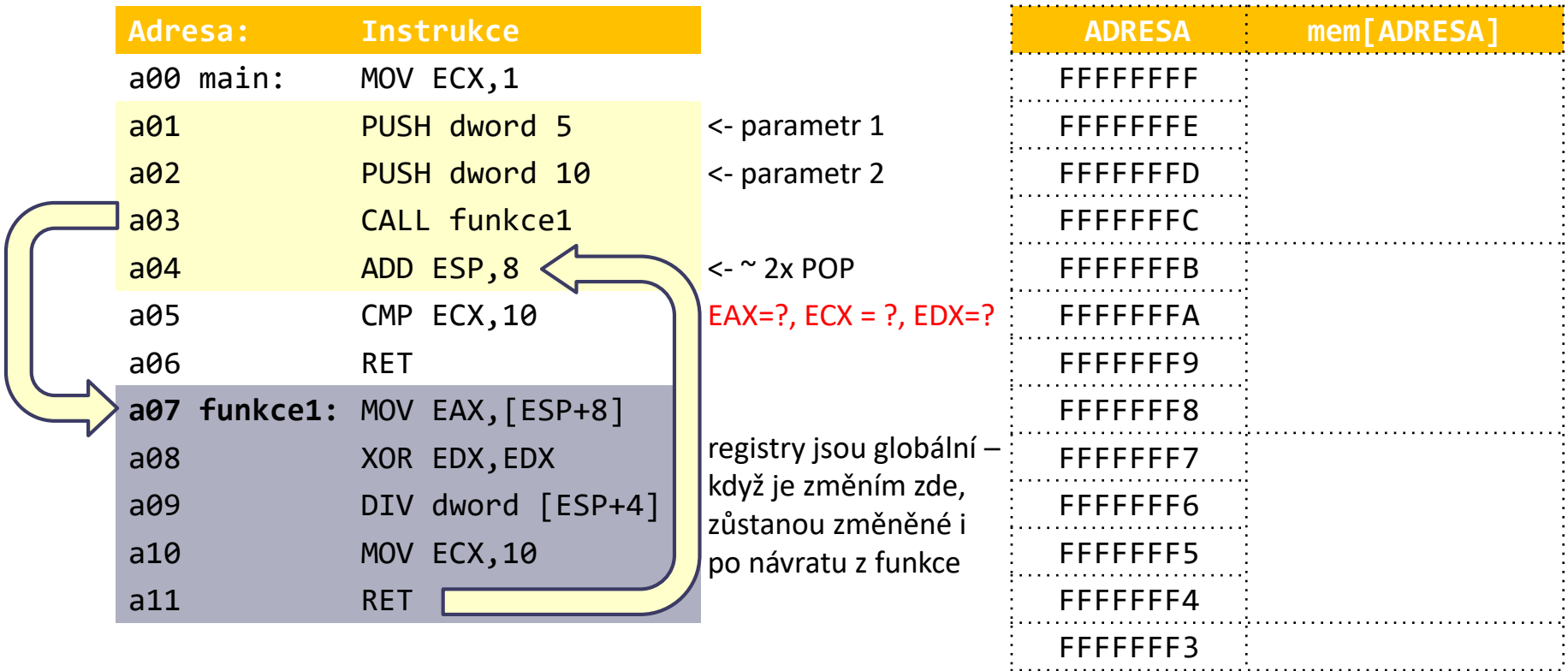
Pozor na pořadí PUSH/POP:

- Co se stane, když pořadí PUSH/POP změním?

ADRESA	mem[ADRESA]
FFFFFFFF	
FFFFFFFE	
FFFFFFFD	
FFFFFFFC	
FFFFFFFB	
FFFFFFFA	
FFFFFFF9	
FFFFFFF8	
FFFFFFF7	
FFFFFFF6	
FFFFFFF5	
FFFFFFF4	
FFFFFFF3	

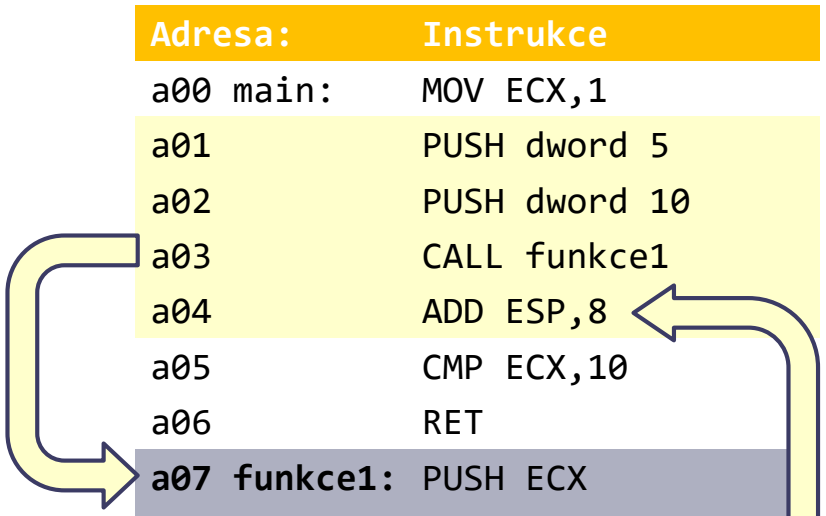
Předávání parametrů přes zásobník 1.

- Registrů je málo:
  - Kam uložit více parametrů než je počet registrů? Na zásobník.
- Na zásobník uložíme parametry, které chceme předat funkci:
  - Kde jsou parametry? Jak se k nim dostat? Pomocí ukazatele ESP?
- Po návratu z funkce musíme ze zásobníku „uklidit“ parametry
- Obsahy registrů stále jsou (a vždy budou) globální – myslete na to!



Předávání parametrů přes zásobník 2.

- PUSH/POP umožní zachovat původní obsah registrů, ale jejich použití uvnitř těla funkce způsobí problémy s adresováním parametrů prostřednictvím registru ESP, které jsou na zásobníku



```
Adresa:      Instrukce
a00 main:    MOV ECX,1
a01          PUSH dword 5
a02          PUSH dword 10
a03          CALL funkce1
a04          ADD ESP,8
a05          CMP ECX,10
a06          RET
a07 funkce1: PUSH ECX
a08          MOV EAX,[ESP+?]
a09          PUSH EDX
a10          XOR EDX,EDX
a11          DIV dword [ESP+?]
a12          MOV ECX,10
a13          POP EDX
a14          POP ECX
a15          RET
```

<- parametr 1

<- parametr 2

<- ~ 2x POP

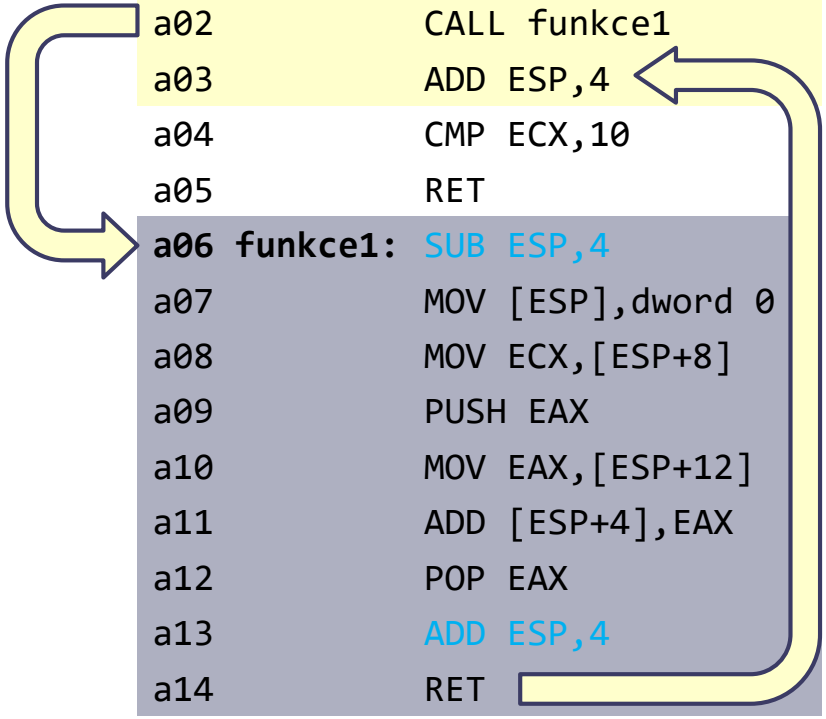
EAX=?, ECX = ?, EDX=?

- Změna registru uvnitř funkce – řešíme přechodným uložením registru na zásobník pomocí PUSH/POP, ale:
- PUSH/POP změní ESP => Kde leží parametry funkce???

ADRESA	mem[ADRESA]
FFFFFFFF	
FFFFFFFE	
FFFFFFFD	
FFFFFFFC	
FFFFFFFB	
FFFFFFFA	
FFFFFFF9	
FFFFFFF8	
FFFFFFF7	
FFFFFFF6	
FFFFFFF5	
FFFFFFF4	
FFFFFFF3	
FFFFFFF2	
FFFFFFF1	
FFFFFFF0	
FFFFFFEF	

Lokální proměnné

- vytvoření za běhu programu na zásobníku = rezervování místa na zásobníku podle jejich počtu, např.: 3 proměnné ~ 3*4 = 12 bytů:
  SUB ESP,12 = posune vrchol zásobníku o 12 bytů dále



Adresa:	Instrukce
a00 main:	MOV ECX,1
a01	PUSH dword 5
a02	CALL funkce1
a03	ADD ESP,4
a04	CMP ECX,10
a05	RET
a06 funkce1:	SUB ESP,4
a07	MOV [ESP],dword 0
a08	MOV ECX,[ESP+8]
a09	PUSH EAX
a10	MOV EAX,[ESP+12]
a11	ADD [ESP+4],EAX
a12	POP EAX
a13	ADD ESP,4
a14	RET

<- parametr 1

<- ~ 1x POP

EAX=?, ECX = ?, EDX=?

<- 1 lokální proměnná

- Změna registru uvnitř funkce – řešíme přechodným uložením registru na zásobník pomocí PUSH/POP, ale:
- PUSH/POP změní ESP => Kde leží parametry funkce a lokální proměnné???

ADRESA	mem[ADRESA]
FFFFFFFF	
FFFFFFFE	
FFFFFFFD	
FFFFFFFC	
FFFFFFFB	
FFFFFFFA	
FFFFFFF9	
FFFFFFF8	
FFFFFFF7	
FFFFFFF6	
FFFFFFF5	
FFFFFFF4	
FFFFFFF3	
FFFFFFF2	
FFFFFFF1	
FFFFFFF0	

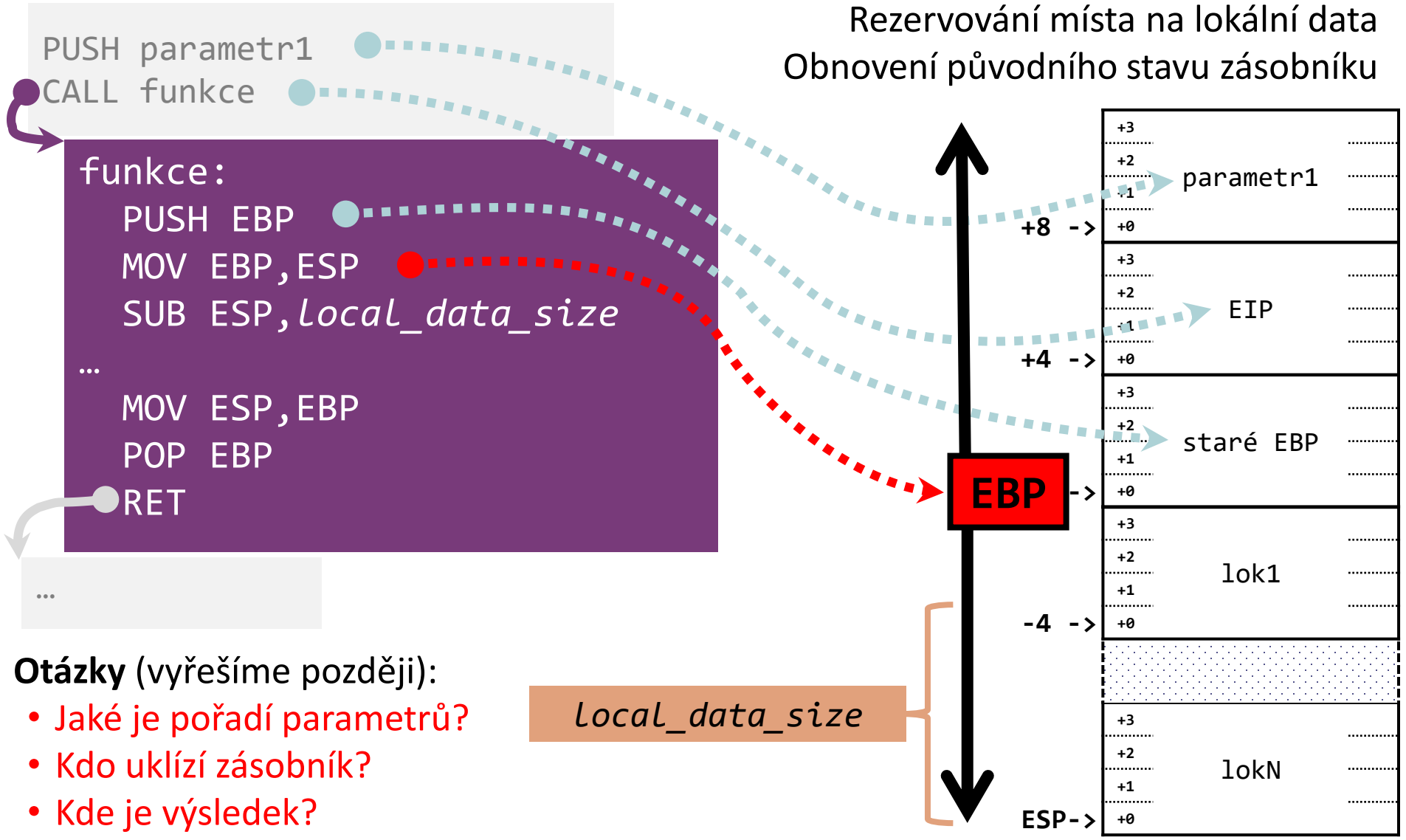
Zásobníkový rámec

Stanovení záchytného bodu („záložka“ v registru **EBP**)
Rezervování místa na lokální data
Obnovení původního stavu zásobníku

```
PUSH parametr1
CALL funkce
```

```
funkce:
   PUSH EBP
   MOV EBP,ESP
   SUB ESP,local_data_size
…
   MOV ESP,EBP
   POP EBP
   RET
```

```
…
```



Otázky (vyřešíme později):

- Jaké je pořadí parametrů?
- Kdo uklízí zásobník?
- Kde je výsledek?

Volání funkce s parametry – postup

- pojmy
  - volající (*calling*): ten, kdo volá funkci instrukcí CALL
  - volaný (*called*, *calee*): ten, kdo je volán = volaná funkce
- volající **uloží na zásobník parametry** například instrukcí PUSH
- volající použije instrukci CALL a volaný přebírá řízení:
  - funkce s parametry vytvoří **zásobníkový rámec**
    1. uloží obsah registru EBP na zásobník
    2. do registru EBP zkopíruje obsah registru ESP
    3. lokální data = snížíme obsah registru ESP o počet bytů, které tato data zabírají (= rezervujeme si prostor na zásobníku = v paměti)
    4. přístup k parametrům: [EBP + posunutí]
    5. přístup k lokálním datům: [EBP – posunutí]
    6. volitelně uložíme obsah registrů, které měníme
    7. jakmile volaný ukončí činnost, obnoví uložené registry a hodnotu ESP a EBP a provede návrat instrukcí RET nebo RET *n* (dle **konvence volání**)
- po návratu z funkce je nutné uklidit po volání zásobník = odstranit parametry, které jsme tam vložili – buď uklízí volaný při návratu z funkce (instrukcí RET *n*) nebo volající (např. zvýšením hodnoty ukazatele ESP)

ENTER lok,nl

(*Make stack frame for procedure parameters*)

1. ENTER imm16,imm8

```
ENTER lok,nl
PUSH EBP; tmpESP = ESP;
for (int i = 1; i < nl; i++) {
  PUSH [EBP-4*i];
}
EBP = tmpESP; ESP = ESP – lok;
```

LEAVE

(*High level procedure exit*)

1. LEAVE

```
LEAVE
ESP = EBP;
POP EBP;
```

imm = přímý operand (konstanta)

ENTER 12,0	PUSH EBP
	MOV EBP,ESP
	SUB ESP,12

…	…
LEAVE	MOV ESP,EBP POP EBP

ENTER 8,2	PUSH EBP
	MOV EAX,ESP
	PUSH dword [EBP-4]
	MOV EBP,EAX
	SUB ESP,8

…	…
LEAVE	MOV ESP,EBP POP EBP

Knihovna RW32-2015: výstup

Jméno funkce	Vstup	Mění registr	Popis činnosti
`WriteNewLine`	---	---	vypíše konec řádku (skok na další řádek)
`WriteChar`	AL	---	vypíše znak uložený v registru AL
`WriteInt8/WriteUInt8`	AL	---	vypíše číslo se znaménkem/bez znaménka uložené v registru AL
`WriteInt16/WriteUInt16`	AX	---	vypíše číslo se znaménkem/bez znaménka uložené v registru AX
`WriteInt32/WriteUInt32`	EAX	---	vypíše číslo se znaménkem/bez znaménka uložené v registru EAX
`WriteBin8/WriteBin16/ WriteBin32`	AL/AX/EAX	---	vypíše 8/16/32-bitové číslo v binární formě
`WriteFloat`	EAX	---	vypíše 32bitové číslo v plovoucí řádové čárce
`WriteDouble`	st0	---	vypíše 64bitové číslo v plovoucí řádové čárce
`WriteString`	ESI	---	vypíše řetězec ukončený 0 uložený na adrese ESI
`WriteFlags`	EFLAGS	---	vypíše příznakový registr

Knihovna RW32-2015: vstup

Jméno funkce	Vstup	Mění registr	Popis činnosti
`ReadChar`	`---`	`AL`	načte z klávesnice znak do registru AL
`ReadInt8/ReadUInt8`	`---`	`AL`	načte z klávesnice do registru AL číslo se znaménkem/bez znaménka
`ReadInt16/ReadUInt16`	`---`	`AX`	načte z klávesnice do registru AX číslo se znaménkem/bez znaménka
`ReadInt32/ReadUInt32`	`---`	`EAX`	načte z klávesnice do registru EAX číslo se znaménkem/bez znaménka
`ReadFloat`	`---`	`EAX`	načte z klávesnice do registru EAX 32bitové číslo v plovoucí řádové čárce
`ReadDouble`	`---`	`st0`	načte z klávesnice do registru st0 64bitové číslo v plovoucí řádové čárce
`ReadString`	`EBX, EDI`	`EAX`	načte z klávesnice řetězec znaků a uloží je na adresu danou registrem EDI, maximální počet znaků, které budou čteny je v registru EBX a počet skutečně přečtených znaků vrací v registru EAX

Knihovna RW32-2015: použití

```
%include „rw32-2015.inc“; vložíme soubor s funkcemi knihovny

segment .data           ; datový segment místo pro data (proměnné)
  ptrData DB „ahoj“,0   ; „ptrData“ ukazuje na hodnotu (byte) 6

segment .text           ; kódový segment – místo pro kód
main:                   ; vstupní místo programu
  MOV ESI,ptrData       ; instrukce s immediate operandem
  CALL WriteString
  CALL WriteNewLine     ; výpis nového řádku (odskok na nový řádek)
  MOV EAX,-1            ; do EAX uložíme (-1)10 = (111…1)2
  CALL WriteBin32       ; vypíšeme binárně
  CALL WriteNewLine
  CALL WriteInt8        ; vypíšeme se znaménkem hodnotu AL
  CALL WriteNewLine
  CALL WriteInt16       ; vypíšeme se znaménkem hodnotu AX
  CALL WriteNewLine
  CALL WriteUInt8       ; vypíšeme bez znaménka hodnotu AL
  CALL WriteNewLine
  CALL WriteUInt32      ; vypíšeme bez znaménka hodnotu EAX
  CALL WriteNewLine
  RET                   ; konec programu
```