Jazyk symbolických adres

Proč programovat v JSA

- Pro některé procesory resp. MCU jsou překladače JSA dostupnější.
- Některé překladače vyšších jazyků neumí využít určité speciální vlastnosti procesoru.
- Možnost vytvořit optimalizovaný kód (?).
- „Cvičné důvody“ – programátor se musí důkladně seznámit s daným procesorem.

Program v JSA vs. strojový kód

- **Program v jazyku symbolických adres:**
  - Používá symbolické názvy instrukcí.
  - Používá symbolické adresy operandů.

↓

Překladač

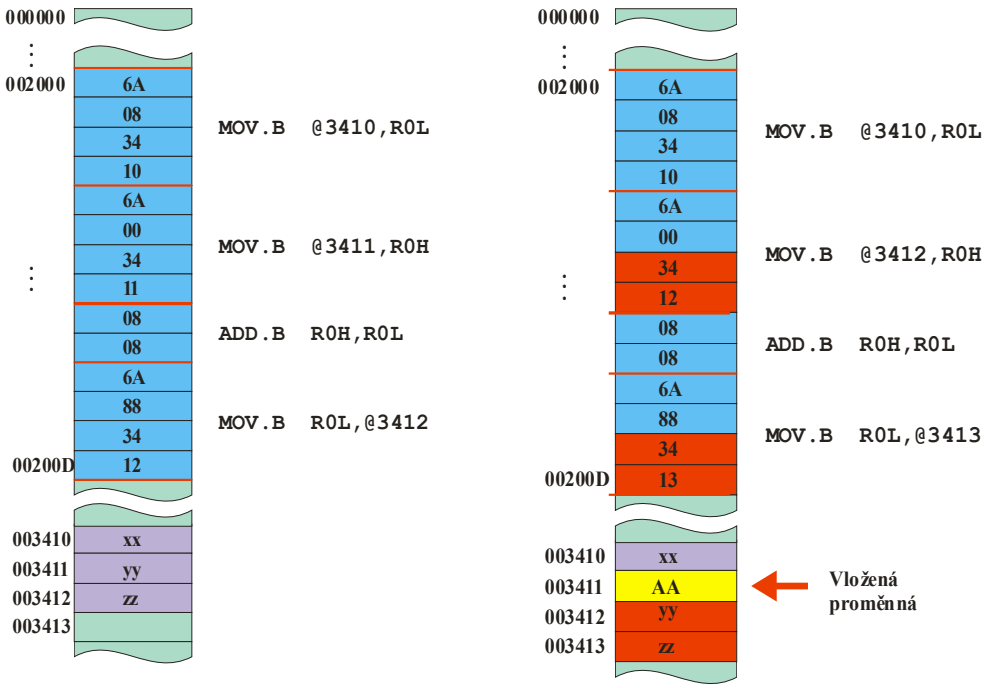
↓

- **Strojový kód = program přeložený do binární podoby:**
  - Obsahuje binární kódy instrukcí,
  - Obsahuje absolutní adresy operandů.

⇓

- Jediná forma programu, kterou umí procesor přímo zpracovat.
- Velmi obtížné úpravy programu.
- Obtížně srozumitelná pro programátora.

Obtížné změny ve strojovém kódu

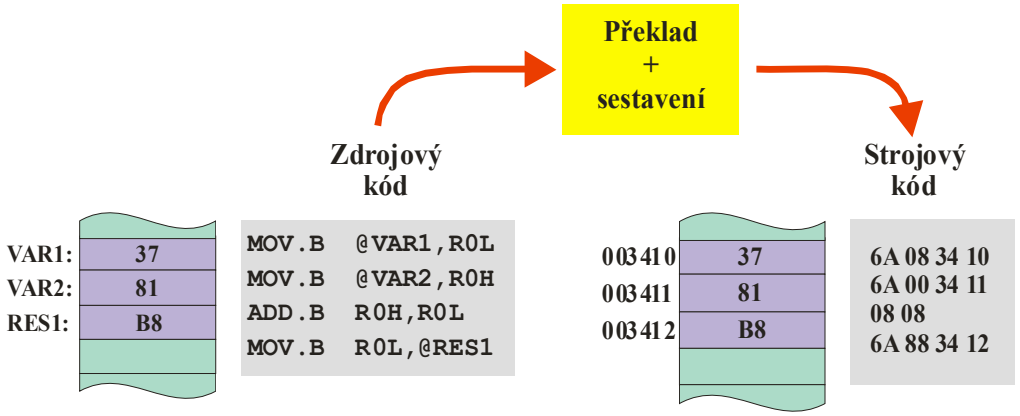


Symbolická adresa (1)

- Symbolická adresa nahrazuje ***ve zdrojovém kódu*** skutečnou (absolutní) adresu.
- Převod symbolická adresa → absolutní adresa provede překladač + sestavovací program.

⇓

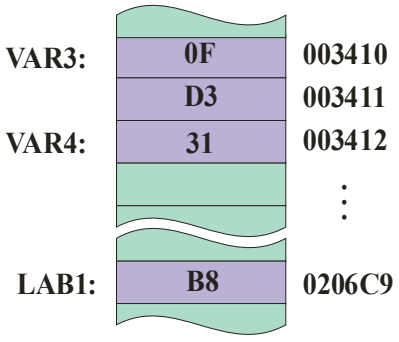
- Programátor nemusí znát skutečné umístění „proměnné“ v paměti.



Symbolická adresa (2)

- Symbolická adresa má
  - **Hodnotu** – odpovídá adrese, kterou reprezentuje.
  - **Obsah** – odpovídá obsahu paměťového místa (bytu, slova, …) na kterou odkazuje.
  - **Typ** – relativní nebo absolutní.

Př:



Symbolická adresa	Hodnota	Obsah
VAR3	003410	0FD3
VAR4	003412	31
LAB1	0206C9	B8

Symbolická adresa (4)

- Použití symbolické adresy
  - **Návěští** – cílová adresa skoku nebo volání procedury.
  - **Proměnná** – adresa pro manipulaci s daty.

```
JMP     @LAB1       ; skok na LAB1
....
MOV.W   @VAR3,R1    ; obsah VAR3 do R1
MOV.L   #VAR4,ER2   ; hodnota VAR4 do ER2
```


Symbolická adresa (3)

- Symbolická adresa může být
  - **Absolutní** – hodnota je známa při překladu, tj. může ji určit přímo asembler (překladač).
  - **Relativní** – hodnotu určí linker (sestavovací program) při sestavování programu.

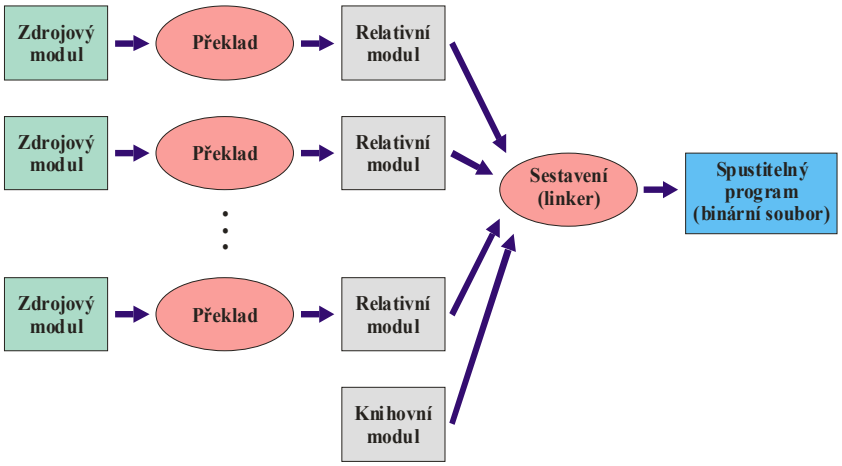
- Výrazy se symbolickými adresami

1. operand	2. operand	Operace	Výsledek
Absolutní	Absolutní	±	Absolutní
Relativní	Absolutní	±	Relativní
Relativní	Relativní	+	Relativní
Relativní	Relativní	–	Absolutní*

* uvnitř jednoho segmentu

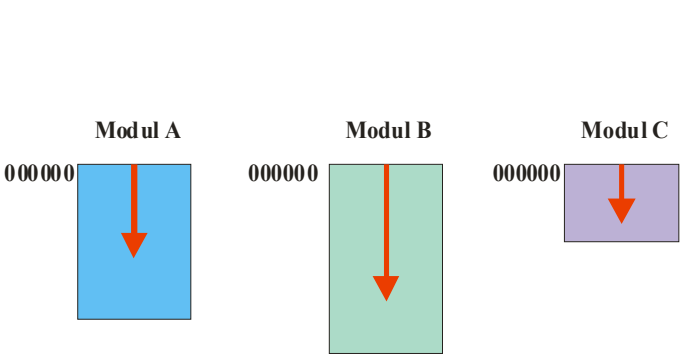
Překlad a sestavení programu

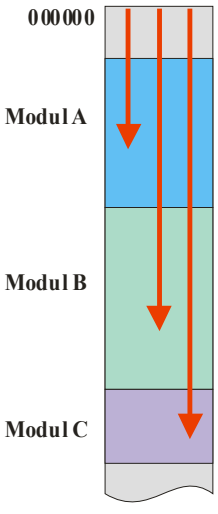
- Program je sestaven z jednoho nebo více modulů.
- Moduly se překládají samostatně.
- Přeložené (relativní) moduly se spojí sestavovacím programem do výsledného souboru.



Relativní a absolutní adresy

- V relativních modulech jsou adresy počítány od začátku modulu.
- V sestaveném programu jsou adresy počítány od začátku paměti.

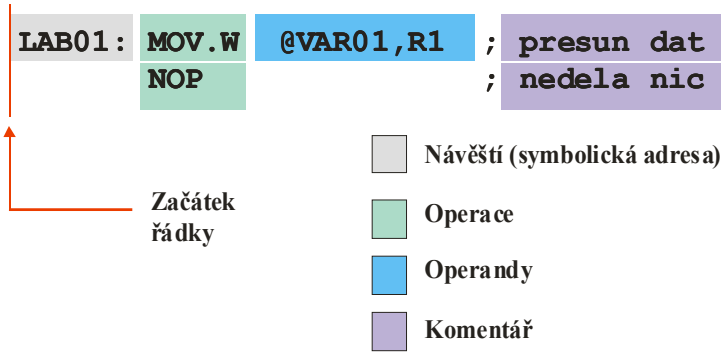




Zápis programu

- Program se zapisuje do 4 sloupců.
- Některá pole se mohou podle situace vynechat.
- Každá řádka obsahuje jednu instrukci, direktivu nebo rozvinutí makra.

  – Návěští – definuje symbolickou adresu.
  – Operace – symbolický název **instrukce** nebo **direktivy**.
  – Operandy – operandy instrukce nebo parametry direktivy.
  – Komentář – je oddělen středníkem.



Instrukční soubor

- Instrukce lze podle funkce rozdělit do několika skupin:

  – přesuny dat,
  – aritmetické operace,
  – logické operace,
  – posuvy a rotace,
  – bitové operace,
  – nepodmíněné a podmíněné skoky,
  – řídicí instrukce.

Instrukční soubor H8S (1)

Function	Instruction	Addressing Modes													
		#xx	Rn	@ERn	@(d:16,ERn)	@(d:32,ERn)	@–ERn/@ERn+	@aa:8	@aa:16	@aa:24	@aa:32	@(d:8,PC)	@(d:16,PC)	@@aa:8	–
Data transfer	MOV	BWL	BWL	BWL	BWL	BWL	BWL	B	BWL	—	BWL	—	—	—	—
	POP, PUSH	—	—	—	—	—	—	—	—	—	—	—	—	—	WL
	LDM, STM	—	—	—	—	—	—	—	—	—	—	—	—	—	L
	MOVEPE, MOVTPE	—	—	—	—	—	—	—	B	—	—	—	—	—	—
Arithmetic operations	ADD, CMP	BWL	BWL	—	—	—	—	—	—	—	—	—	—	—	—
	SUB	WL	BWL	—	—	—	—	—	—	—	—	—	—	—	—
	ADDX, SUBX	B	B	—	—	—	—	—	—	—	—	—	—	—	—
	ADDS, SUBS	—	L	—	—	—	—	—	—	—	—	—	—	—	—
	INC, DEC	—	BWL	—	—	—	—	—	—	—	—	—	—	—	—
	DAA, DAS	—	B	—	—	—	—	—	—	—	—	—	—	—	—
	MULXU, DIVXU	—	BW	—	—	—	—	—	—	—	—	—	—	—	—
	MULXS, DIVXS	—	BW	—	—	—	—	—	—	—	—	—	—	—	—
	NEG	—	BWL	—	—	—	—	—	—	—	—	—	—	—	—
	EXTU, EXTS	—	WL	—	—	—	—	—	—	—	—	—	—	—	—
	TAS*2	—	—	B	—	—	—	—	—	—	—	—	—	—	—
	MAC*1	—	—	—	—	—	○	—	—	—	—	—	—	—	—
	CLRMAC*1	—	—	—	—	—	—	—	—	—	—	—	—	—	○
	LDMAC*1, STMAC*1	—	L	—	—	—	—	—	—	—	—	—	—	—	—

Instrukční soubor H8S (2)

Function	Instruction	Addressing Modes													
		#xx	Rn	@ERn	@(d:16,ERn)	@(d:32,ERn)	@–ERn/@ERn+	@aa:8	@aa:16	@aa:24	@aa:32	@(d:8,PC)	@(d:16,PC)	@@aa:8	–
Logic operations	AND, OR, XOR	BWL	BWL	—	—	—	—	—	—	—	—	—	—	—	—
	NOT	—	BWL	—	—	—	—	—	—	—	—	—	—	—	—
Shift		—	BWL	—	—	—	—	—	—	—	—	—	—	—	—
Bit manipulation		—	B	B	—	—	—	B	B	—	B	—	—	—	—
Branch	Bcc, BSR	—	—	—	—	—	—	—	—	—	—	○	○	—	—
	JMP, JSR	—	—	—	—	—	—	—	—	○	—	—	—	○	—
	RTS	—	—	—	—	—	—	—	—	—	—	—	—	—	○
System control	TRAPA	—	—	—	—	—	—	—	—	—	—	—	—	—	○
	RTE	—	—	—	—	—	—	—	—	—	—	—	—	—	○
	SLEEP	—	—	—	—	—	—	—	—	—	—	—	—	—	○
	LDC	B	B	W	W	W	W	—	W	—	W	—	—	—	—
	STC	—	B	W	W	W	W	—	W	—	W	—	—	—	—
	ANDC, ORC, XORC	B	—	—	—	—	—	—	—	—	—	—	—	—	—
	NOP	—	—	—	—	—	—	—	—	—	—	—	—	—	○
Block data transfer		—	—	—	—	—	—	—	—	—	—	—	—	—	BW

Příklad: Instrukce MOV.W (1)

2.2.39 (5) MOV (W)

MOV (MOVe data) **Move**

Operation

(EAs) → Rd

Assembly-Language Format

`MOV.W` <EAs>, Rd

Operand Size

Word

Condition Code

I	UI	H	U	N	Z	V	C
—	—	—	—	↕	↕	0	—

H: Previous value remains unchanged.
N: Set to 1 if the transferred data is negative; otherwise cleared to 0.
Z: Set to 1 if the transferred data is zero; otherwise cleared to 0.
V: Always cleared to 0.
C: Previous value remains unchanged.

Description

This instruction transfers the source operand contents to a 16-bit register Rd, tests the transferred data, and sets condition-code flags according to the result.

Příklad: Instrukce MOV.W (2)

Operand Format and Number of States Required for Execution

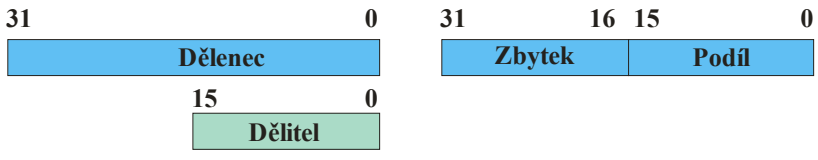
Addressing Mode	Mnemonic	Operands	Instruction Format								No. of States
			1st byte	2nd byte	3rd byte	4th byte	5th byte	6th byte	7th byte	8th byte	
Immediate	MOV.W	#xx:16, Rd	7 9	0 rd	IMM						2
Register indirect	MOV.W	@ERs, Rd	6 9	0 ers rd							2
Register indirect with displace-ment	MOV.W	@(d:16, ERs), Rd	6 F	0 ers rd	disp						3
	MOV.W	@(d:32, ERs), Rd	7 8	0 ers 0	6 B	2 rd	disp				5
Register indirect with post-increment	MOV.W	@ERs+, Rd	6 D	0 ers rd							3
Absolute address	MOV.W	@aa:16, Rd	6 B	0 rd	abs						3
	MOV.W	@aa:32, Rd	6 B	2 rd	abs						4

Instrukční soubor – přesuny dat

- Operandy typu B, W, L.
- Přesuny **paměť ↔ registr**, **registr ↔ registr**,
  přímý operand → registr.
- Lze použít různé adresní mody (bázová/indexová adresa, adresování registrem s autoinkrementem, ...).
- Přesuny nastavují příznakové bity.

Instrukční soubor – aritmetické operace

- Operandy typu B, W, L.
- Operace **registr * registr → registr**,
  přímý operand * registr → registr.
- Sečítání, odčítání, inkrement, dekrement.
- Dekadická korekce.
- Násobení (8 × 8 → 16), (16 × 16 → 32), signed/unsigned.
- Dělení (16 : 8 → 8 + 8), (32 : 16 → 16 + 16).

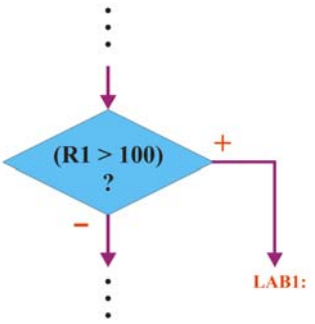


Dělení (32 : 16 → 16 + 16)

Instrukční soubor – porovnání

- Operandy typu B, W, L.
- Operace **registr * registr**, **přímý operand * registr**.
- Formálně provede odečtení operandů
  - nastaví příznaky podle výsledku,
  - výsledek se **neuloží** do cílového registru.
- Použití obvykle s následnou instrukcí typu **Bcc**
  - pro následující **Bcc** se uvažuje **cmp y,x .**

```
...
cmp    #10,R1 ;příznaky podle (R1-10)
bgt    LAB1   ;skok při x > y
...
```



Instrukční soubor – logické a bitové operace

Logické operace:

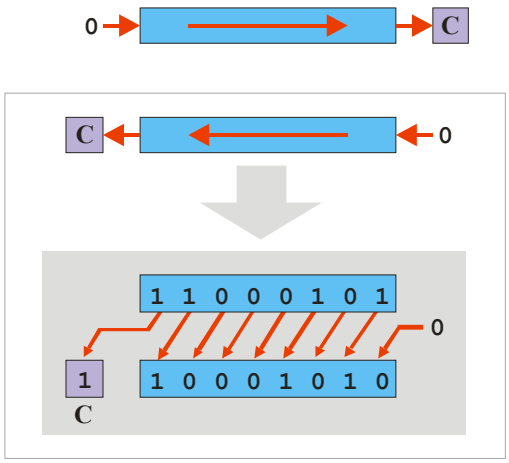
- Operandy typu B, W, L.
- Operace **registr * registr → registr**,
  přímý operand * registr → registr.
- Logický součet (OR), součin (AND), nonekvivalence (XOR), negace (NOT).

Bitové operace:

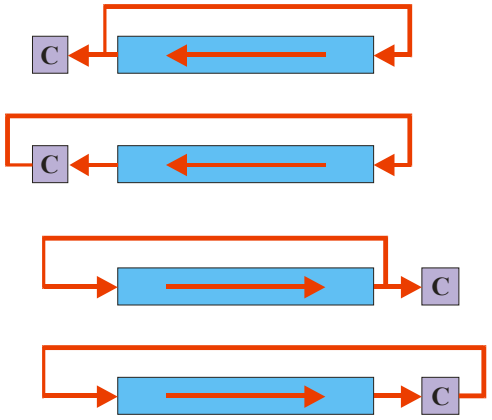
- Nastavení, nulování, negace bitu.
- Operace typu **Carry * bit → Carry** a **Carry * bit → bit** .
- Operandy typu byte – v registrech nebo v paměti.
- Bitové operace v paměti jsou typu **Read – Modify – Write**.

Instrukční soubor – posuvy a rotace

- Operandy typu B, W, L v registrech.
- Rotace s **Carry** nebo bez **Carry**.



Posuvy



Rotace

Instrukční soubor – skoky

Skoky JMP, JSR:

- Přímá nebo nepřímá adresa.

Podmíněné skoky Bcc:

- Relativní adresa 16 nebo 8 bitů.
  - Relativní adresa je signed, tj skok může být v rozsahu < PC-32768; PC+32766 > resp. < PC-128; PC+126 > .

Instrukční soubor – řídicí instrukce

- Uložení registrů CCR a EXR do paměti, resp. přečtení z paměti.
- Instrukce pro ladicí přerušení TRAPA.
- Návrat z přerušení RTE.
- Přechod do režimu sníženého odběru (SLEEP).

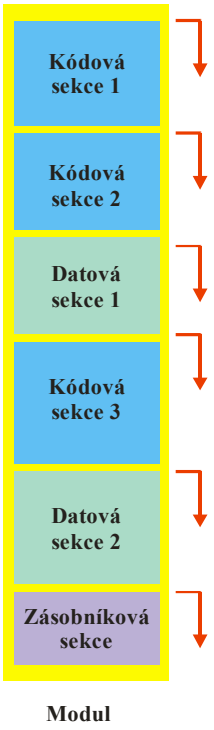
Důležité direktivy

Direktivy (povely pro překladač):

- definice sekcí (segmentů),
- definice dat a symbolů,
- makra,
- podmíněný překlad,
- další: strukturovaný překlad, listing, … .

Struktura modulu

- Modul obsahuje jednu nebo více sekcí (segmentů).
- Každá sekce má nezávislé adresování od svého začátku.
- Pořadí sekcí ve zdrojovém souboru není podstatné – upraví se při sestavení.



Základní typy sekcí (GNU as)

- Datová sekce
  - Obsahuje inicializovaná data („proměnné“) programu.
- Kódová sekce
  - Obsahuje kód programu.
- Další sekce
  - Neinicializovaná data, zásobník, přerušovací vektory, další uživatelem (programátorem) definované sekce.

Definice sekce

- Hlavička sekce (zjednodušeně)
  - Standardní sekce GNU as a ld:

```
.data   [subsekce]      začátek datové sekce
.text   [subsekce]      začátek kódové sekce
```

  - Libovolné další sekce:

```
.section  jméno         začátek sekce jméno
```

```
   ...
.text 0
   kód programu
.data 3
   data (proměnné)
.section MOJE_SEKCE
   ...
```


Počítadlo adres

- Každá sekce má (při překladu) samostatné počítadlo adres (PLC – Programm Location Counter)
- Není-li určeno jinak, inicializuje se PLC na 0 na začátku sekce.
- Možnosti nastavení PLC:

```
.org      výraz
.align    uložení
```

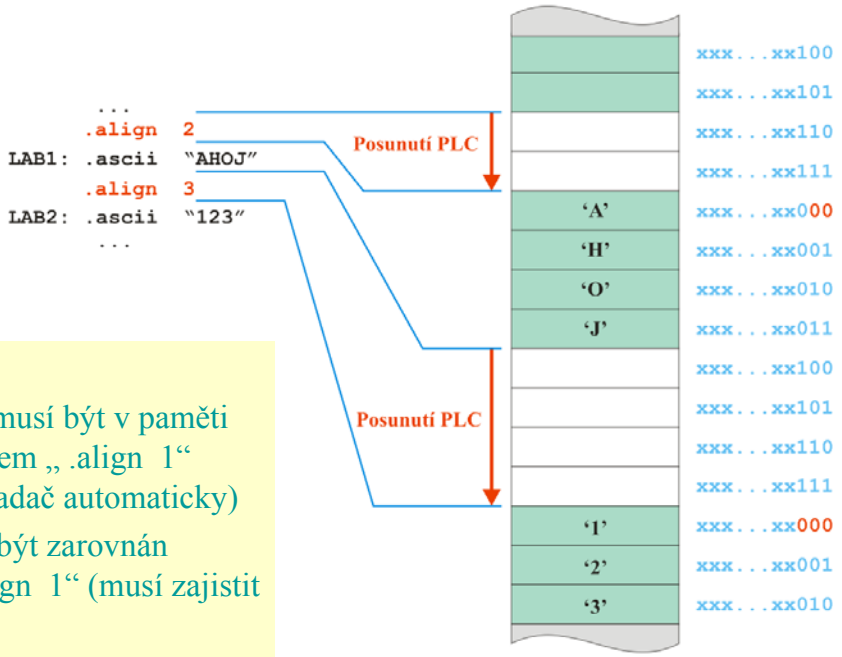
`.org` = nastaví PLC na hodnotu ***výraz***

`.align` = nastaví PLC na hodnotu **MOD(2^*uložení*)**

☝ Všechny adresy jsou vztaženy k začátku sekce. Je-li sekce relativní, musí se jiným způsobem zabezpečit potřebné umístění sekce v paměti.

```
.text
MOV     @VAR1,R1
JMP     @LAB1
.org    0x000400
JMP     @LAB2
.align  4
JMP     LAB3
...
```

Příklad použití `.align`

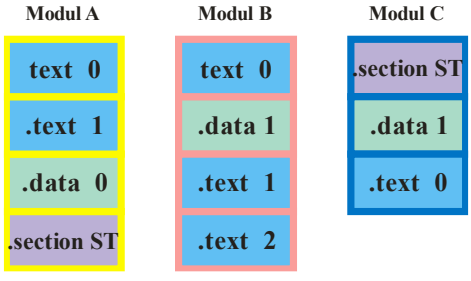


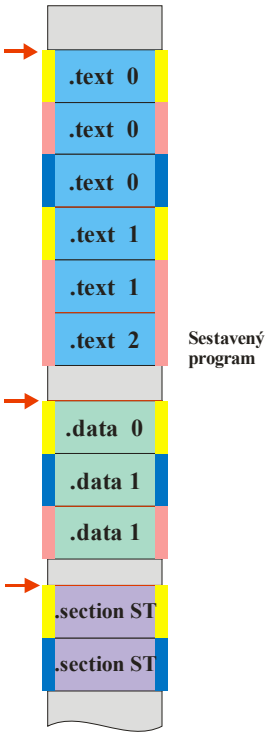
Poznámka:

1. Instrukce H8S musí být v paměti uložené způsobem „ .align 1“ (zajišťuje překladač automaticky)
2. Zásobník musí být zarovnán způsobem „ .align 1“ (musí zajistit programátor).

Sestavení sekcí a modulů

- Sestavovací program spojí stejné sekce dohromady.
- Jsou-li definovány subsekce, spojí dohromady i stejné subsekce





Definice dat a symbolů

- Definice a přiřazení hodnoty symbolu

```
.equ   symbol,výraz
symbol = výraz
```

symbol – symbol, kterému bude přiřazena hodnota výrazu.
výraz – jeho hodnota bude přiřazena symbolu.

- `.equ` a `=` pouze definují symbol a jeho hodnotu. Nevyhrazují místo v paměti.
- Platnost symbolu je omezena na modul ve kterém je symbol definován.
- Hodnotu symbolu nelze změnit.

```
        ...
.equ    BASE,0x008000
.equ    LIMIT,0x100
.equ    NEXT,BASE+LIMIT
        NIC = 0
        ...
```


Definice dat (1)

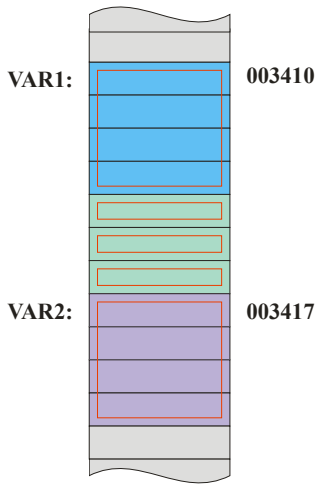
- Definice místa pro „proměnnou“

```
[návěští]   .space   položky
```

návěští – symbolická adresa,
položky - výrazy definující počet položek

- Vyhradí v paměti místo určené délky (počtu bytů).
- Je-li uvedeno návěští, odpovídá jeho hodnota adrese 1. bytu dat.

```
        ...
VAR1:   .space  4
        .space  1
        .space  1
        .space  1
VAR2:   .space  4
        ...
```



Definice dat (2)

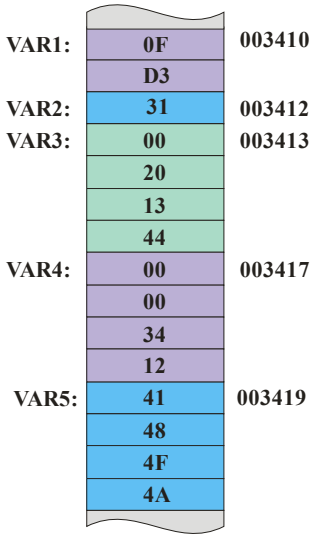
- Definice „proměnné“ s počáteční hodnotou

```
[návěští]   .byte    výrazy
[návěští]   .word    výrazy
[návěští]   .long    výrazy

[návěští]   .ascii   řetězec
[návěští]   .asciz   řetězec
```

návěští – symbolická adresa,
výrazy – výrazy definující obsah jednotlivých položek, oddělené čárkou.

- Vyhradí v paměti místo, jehož obsah je dán jednotlivými výrazy.
- Je-li uvedeno návěští, odpovídá jeho hodnota adresa 1. bytu dat.

```
VAR1:    .word    0x0FD3
VAR2:    .byte    '1'      ; '1' = 0x31
VAR3:    .byte    0,32,19,68
VAR4:    .long    VAR2     ; hodnota VAR2
VAR5:    .ascii   „AHOJ"
```



Sdílení dat mezi moduly (1)

- Symboly mají platnost jen v modulu, ve kterém jsou definovány.
- Rozšíření platnosti (export) symbolů:

```
.global   symboly
```

symboly – seznam exportovaných symbolů oddělených čárkami.

- Použití symbolů definovaných v jiném modulu (import):

```
.extern symboly
```

symboly – seznam importovaných symbolů oddělených čárkami.

- Lze použít pouze pro symboly, definované jako návěští (ne EQU).

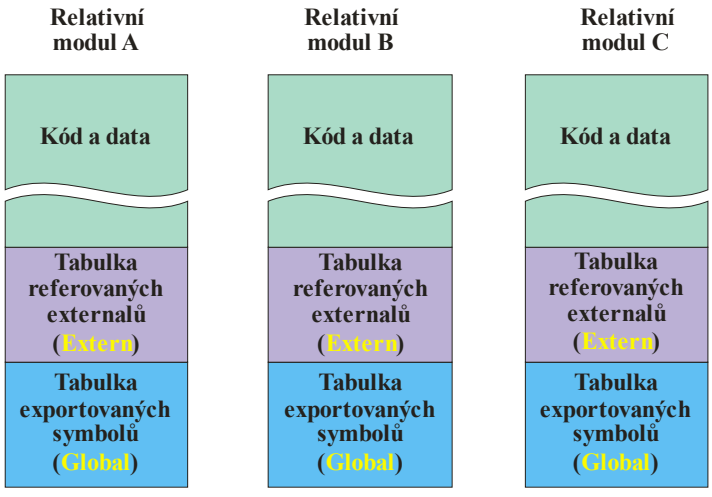
Sdílení dat mezi moduly (2)

```
;       A_MODUL
        .global VAR_A1,LAB_A1
        .extern VAR_B1,LAB_B1
VAR_A1: .word   0x0FD3
VAR_A2: .space  2
        ...
LAB_A1: MOV.W   @VAR_A2,R1
        ...
        MOV.W   R1,@VAR_B1
        JMP     @LAB_B1
        ...
```

```
;       B_MODUL
        .global VAR_B1,LAB_B1
        .extern VAR_A1,LAB_A1
VAR_B1: .word   0x0FD3
VAR_B2: .space  2
        ...
LAB_B1: MOV.W   @VAR_B2,R1
        ...
        MOV.W   R1,@VAR_A1
        JMP     @LAB_A1
        ...
```


Sdílení dat mezi moduly (3)

- Relativní moduly obsahují tabulku referovaných externích symbolů (Extern) a tabulku exportovaných symbolů (Global).
- Sestavovací program hledá externí symboly v tabulkách Global ostatních modulů.



Makra (definice)

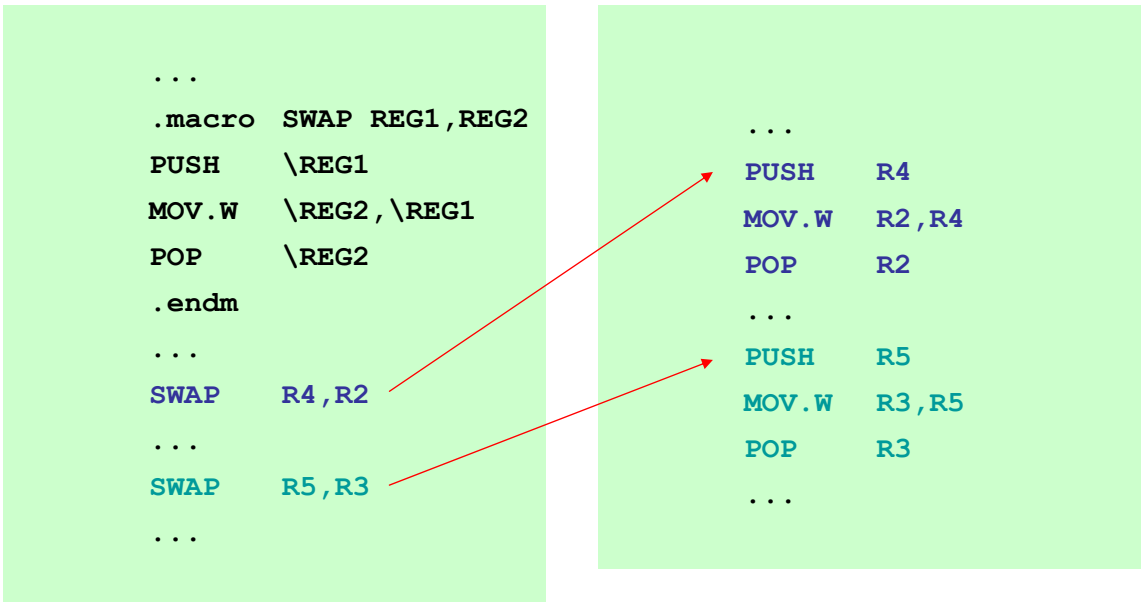
- Umožňuje definovat část programu, která bude použita na více místech.

```
.macro jméno argumenty
   tělo makra
.endm
```

jméno – jméno makra.
argumenty – seznam symbolických argumentů. V těle se referují s \ na začátku.
tělo makra – jednotlivé instrukce.
.endm – ukončuje rozvoj makra.

- Rozvinutí makra vloží tělo makra, tj. **kopie jednotlivých instrukcí** do daného místa programu.

Makra (rozvinutí)



Zápis v programu

Rozvinutí makra

Makra (lokální symboly)

- Je-li v makru definován symbol (např. návěští), vznikají při vícenásobném rozvinutí problémy.
- Symbol se musí definovat jako lokální v makru.

```
LOCAL symboly
```

symboly – seznam lokálních symbolů, oddělených čárkami.

- Překladač vytvoří v každém rozvinutí unikátní jméno symbolu .
- Před použitím LOCAL se musí použít direktiva `.altmacro`

Makra (bez lokálních symbolů)

```
        ...
        .altmacro
        .macro ONES VAR,RESULT
        MOV.W   @\VAR,R1
        MOV.B   #16,R0L
        XOR.B   R0H,R0H
LAB1:   ROTR    R1
        BCC     LAB2
        INC     R0H
LAB2:   DEC     R0L
        BNE     LAB1
        MOV.B   R0H,@\RESULT
        .endm
        ...
```

Definice makra

```
        ...
        ONES    VAR1,VAR2
        MOV.W   @VAR1,R1
        MOV.B   #16,R0L
        XOR.B   R0H,R0H
LAB1:   ROTR    R1
        BCC     LAB2
        INC     R0H
LAB2:   DEC     R0L
        BNE     LAB1
        MOV.B   R0H,@VAR2
        ...
        ONES    VARX,VARY
        MOV.W   @VARX,R1
        MOV.B   #16,R0L
        XOR.B   R0H,R0H
LAB1:   ROTR    R1
        BCC     LAB2
        INC     R0H
LAB2:   DEC     R0L
        BNE     LAB1
        MOV.B   R0H,@VARY
        ...
```

Makra (lokální symboly)

```
          ...
          .altmacro
          .macro ONES VAR,RESULT
          LOCAL    LAB1,LAB2
          MOV.W    @\VAR,R1
          MOV.B    #16,R0L
          XOR.B    R0H,R0H
LAB1:     ROTR     R1
          BCC      LAB2
          INC      R0H
LAB2:     DEC      R0L
          BNE      LAB1
          MOV.B    R0H,@\RESULT
          .endm
          .noaltmacro
          ...
```

Definice makra

```
          ...
VAR1:     .space   2
VAR2:     .space   1
VARX:     .space   2
VARY:     .space   1
          ...
          ONES     VAR1,VAR2
          ...
          ONES     VARX,VARY
          ...
```

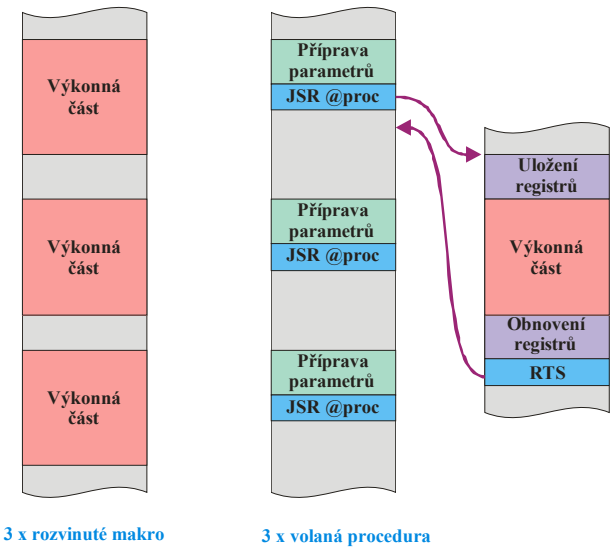
Použití makra

Makra a procedury - porovnání

- Použití makra:
  - Kód výkonné části je v paměti pro každé rozvinutí.
  - Bez dodatečného zpoždění pro JSR, RTS, přenos parametrů, ...
- Použití procedury:
  - Výkonná část je v paměti jen 1× (úspora paměti).
  - Dodatečné zpoždění pro JSR, RTS, ... (pomalejší než makro).

⇓

- Makro: obvykle rychlejší.
- Procedura: obvykle úspornější.



Spojování programů v JSA a vyšších jazycích

- Typický případ: procedura v JSA volaná z programu ve vyšším jazyku (např. C).
- Je nutné znát:
  - pravidla pro vytváření jmen (proměnných, segmentů, ...),
  - pravidla pro předávání argumentů do procedury,
  - pravidla pro předávání hodnoty funkce do volajícího programu,
  - pravidla pro zacházení se zásobníkem,
  - pravidla pro sestavení programu ve vyšším jazyku (inicializační modul, knihovny, ...).
- Výše uvedené bývá popsáno v příslušném manuálu.
  - Záleží na konkrétním procesoru (počet registrů, ...) a implementaci překladače.

Jednoduchý příklad

```
void proc1(void);
short aaa;

void main(void) {
   ...
    proc1();
   ...
}
```

```
        .extern  _aaa
        .text
_proc1: push.w R1
        mov.w   #0x1234,R1
        mov.w   R1,@_aaa
        pop.w    R1
        rts
```

- Identifikátory mají po překladu prefix _ (podtržítko).

Předávání návratové hodnoty funkce

- Návratová hodnota se obvykle předává v registrech.
  - GCC pro H8S předává hodnotu v R0L, R0 nebo ER0 resp. v C (carry).

```
short funkce1(void);
/* vrací součet aaa + bbb */

short aaa;
short bbb;
short xxx;

void main(void) {
    ...
     xxx = funkce1();
    ...
}
```

```
          .extern  _aaa
          .extern  _bbb

          .text
_funkce1: mov.w    @_aaa,R0
          mov.w    @_bbb,R1
          add.w    R1,R0
          rts
; návratová hodnota je v R0
```


Předávání argumentů v registrech

- Použitelné pro omezený počet argumentů.
  - GCC pro H8S předává první 3 argumenty v ER0, ER1, ER2 nebo v jejich části.

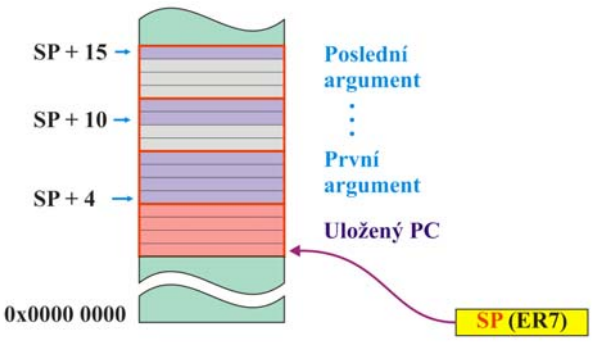
```
short funkce2(short a, short b);
/* vrací součet a + b */

short aaa;
short bbb;
short xxx;

void main(void) {
    ...
     xxx = funkce2(aaa,bbb);
    ...
}
```

```
          .text
; aaa je v R0, bbb je v R1
_funkce2: add.w    R1,R0
          rts
; návratová hodnota je v R0
```

Předávání argumentů v zásobníku (1)

- Obvykle v pořadí od posledního argumentu.
  - GCC pro H8S ukládá vždy 4 byty bez ohledu na typ argumentu.
  - Lze vynutit přenos všech argumentů v zásobníku parametrem `-mno-quickcall` při překladu (jinak by byly první 3 argumenty v registrech).

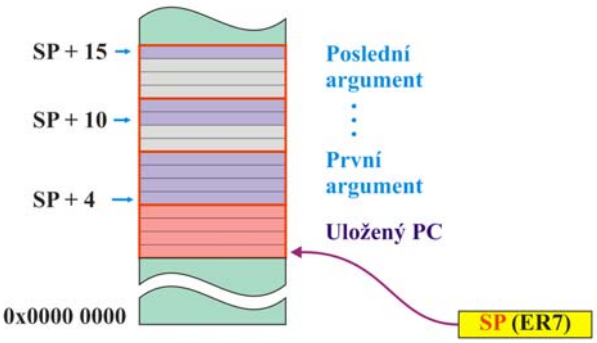
```
void proc2(char *a, short b, char c);
/* nahradí v a znak na pozici b znakem c */

char aaa[] = "UPRAVOVANY TEXT";
short bbb;
char  ccc;

void main(void) {
   ...
    proc2(aaa,bbb,ccc);
   ...
}
```



Předávání argumentů v zásobníku (2)

- Obvykle v pořadí od posledního argumentu.
  - GCC pro H8S ukládá vždy 4 byty bez ohledu na typ argumentu.
  - Bylo by vhodné uložit ER1 a ER0 do zásobníku → změní se offset argumentů k SP.

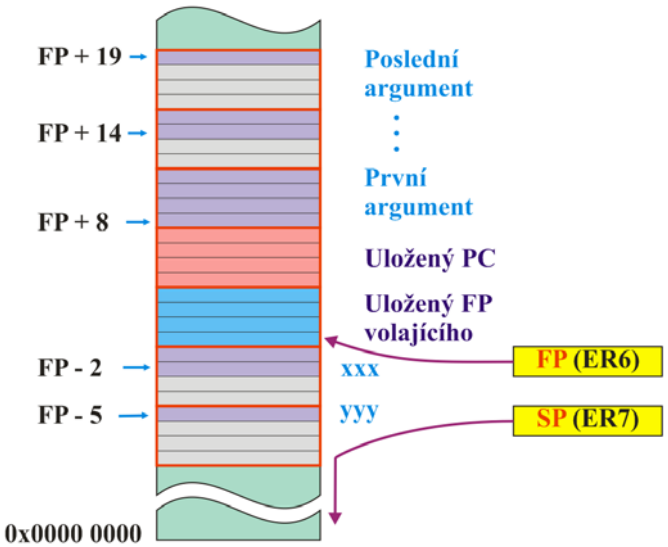
```
        .text
_proc2: mov.l   @(SP+4),ER0
        mov.w   @(SP+10),R1
        xor.w   E1,E1
        add.l   ER1,ER0
        mov.b   @(SP+15),R1L
        mov.b   R1L,@ER0
        rts
```



Předávání argumentů v zásobníku – frame (1)

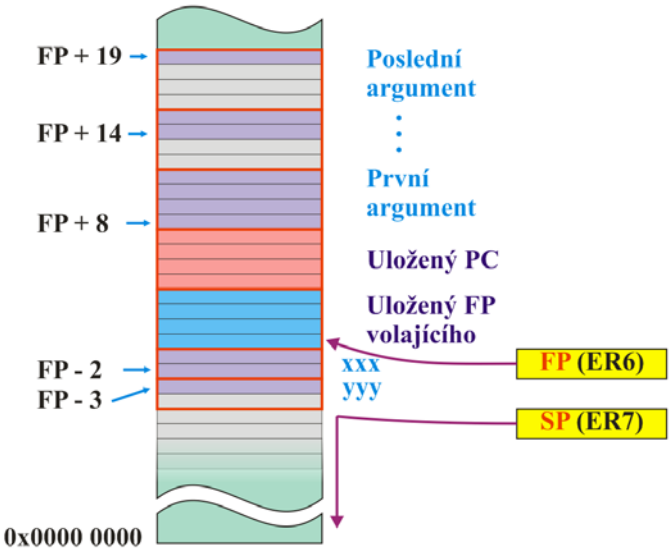
- Dokonalejší technika pro práci s argumenty a automatickými proměnnými.
- Frame obsahuje argumenty volání, uložené registry a automatické proměnné.
  - FP se během výpočtu procedury nemění.

```
void proc2(char *a,
           short b,
           char c) {
    short xxx;
    char yyy;
    ...
    ...
    ...
}
```



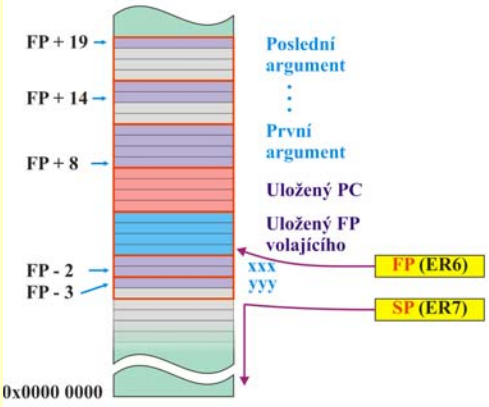
Předávání argumentů v zásobníku – frame (2)

```
char    aaa[] = "UPRAVOVANY TEXT";
short   bbb;
char    ccc;

int main(void) {
    bbb = 20;
    ccc = 'X';

    proc2(aaa,bbb,ccc);
    while (1) {
    }
    return 0;
}

void proc2(char *a, short b, char c){

    short     xxx; /* jen na ukazku */
    char      yyy; /* jen na ukazku */

    a[b] = c;
    xxx = 100;      /* jen na ukazku */
    yyy = 100;      /* jen na ukazku */
}
```



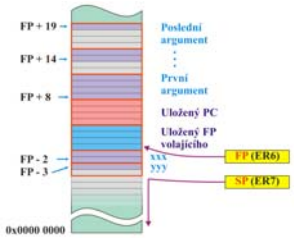
Předávání argumentů v zásobníku – frame (3)

```
        _main:
79020014      mov.w   #20,r2
6BA200000000  mov.w   r2,@_bbb:32
FA58          mov.b   #88,r2l
6AAA00000000  mov.b   r2l,@_ccc:32
6A2B00000000  mov.b   @_ccc:32,r3l
6B2200000000  mov.w   @_bbb:32,r2
01006DF3      mov.l   er3,@-er7    ; 3. argument
01006DF2      mov.l   er2,@-er7    ; 2. argument
7A0200000000  mov.l   #_aaa,er2
01006DF2      mov.l   er2,@-er7    ; 1. argument
5E000000      jsr     @_proc2
7A170000000C  add.l   #12,er7      ; vyprázdnění
                                     zásobníku
          .L2:
4000                  bra     .L2
```

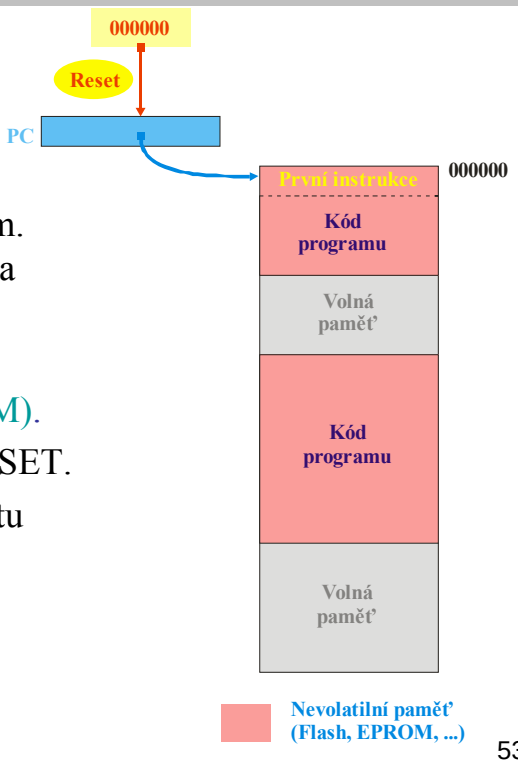


Předávání argumentů v zásobníku – frame (4)

```
        _proc2:
01006DF6      mov.l   er6,@-er7     ; uložení FP volajícího
0FF6          mov.l   er7,er6       ; FP <- SP
1B97          subs    #4,er7        ; místo pro aut. proměnné
6F62000E      mov.w   @(14,er6),r2  ; 2. argument
17F2          exts.l  er2
01006F630008  mov.l   @(8,er6),er3  ; 1. argument
0AB2          add.l   er3,er2
6E6B0013      mov.b   @(19,er6),r3l ; 3. argument
68AB          mov.b   r3l,@er2
79020064      mov.w   #100,r2
6FE2FFFE      mov.w   r2,@(-2,er6)  ; 1. automatická proměnná
FA64          mov.b   #100,r2l
6EEAFFFD      mov.b   r2l,@(-3,er6) ; 2. automatická proměnná
0B97          adds    #4,er7        ; odstranění aut.
                                    ; proměnných
01006D76      mov.l   @er7+,er6     ; obnovení FP volajícího
5470          rts
```

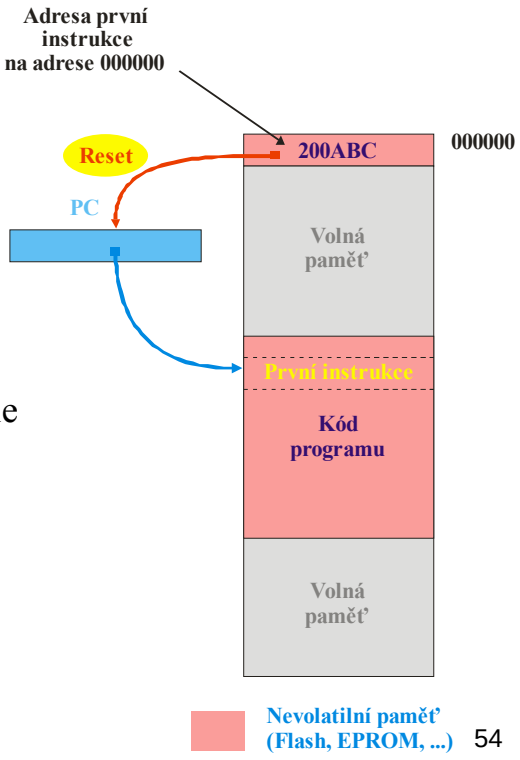


Start procesoru (1)



1. V paměti musí být připraven program. První instrukce programu musí být na určité pevně dané adrese (obvykle 0x0...00).
   - Paměť musí být nevolatilní (ROM).
2. Po připojení napájení se provede RESET.
3. Při resetu nastaví CPU do PC hodnotu 0x0...00 (nebo jinou pevně danou hodnotu).

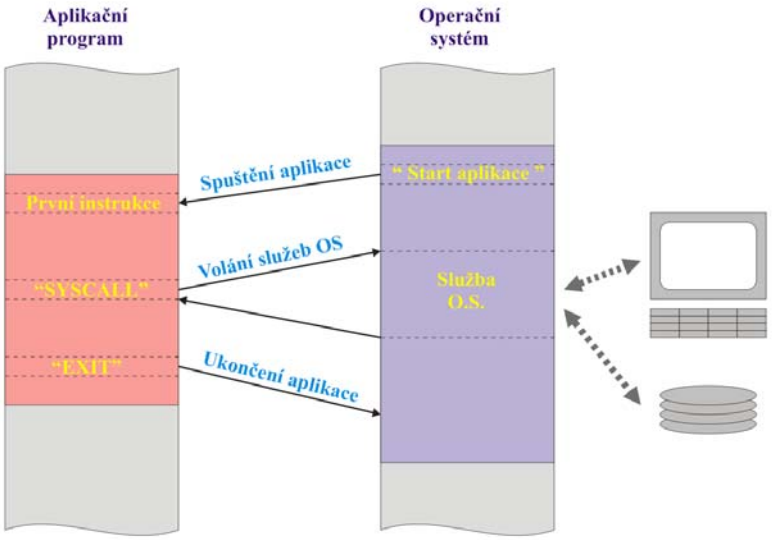
Start procesoru (2)



- **Jiná možnost (použitá i u H8S):**

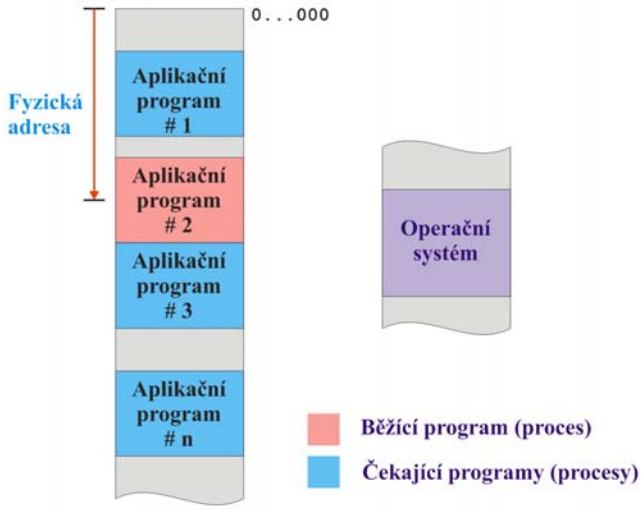
1. Po připojení napájení se provede RESET.
2. V paměti musí být připraven program. Adresa první instrukce programu musí být na určité pevně dané adrese (obvykle 0x0...00).
   - Paměť musí být nevolatilní (ROM).
3. Při resetu nastaví CPU do PC obsah adresy 0x0...00.

Operační systém



Multitasking

- Zjednodušený pohled:
  - V paměti je zavedeno několik programů.
  - OS střídavě spouští jednotlivé programy podle různých pravidel.

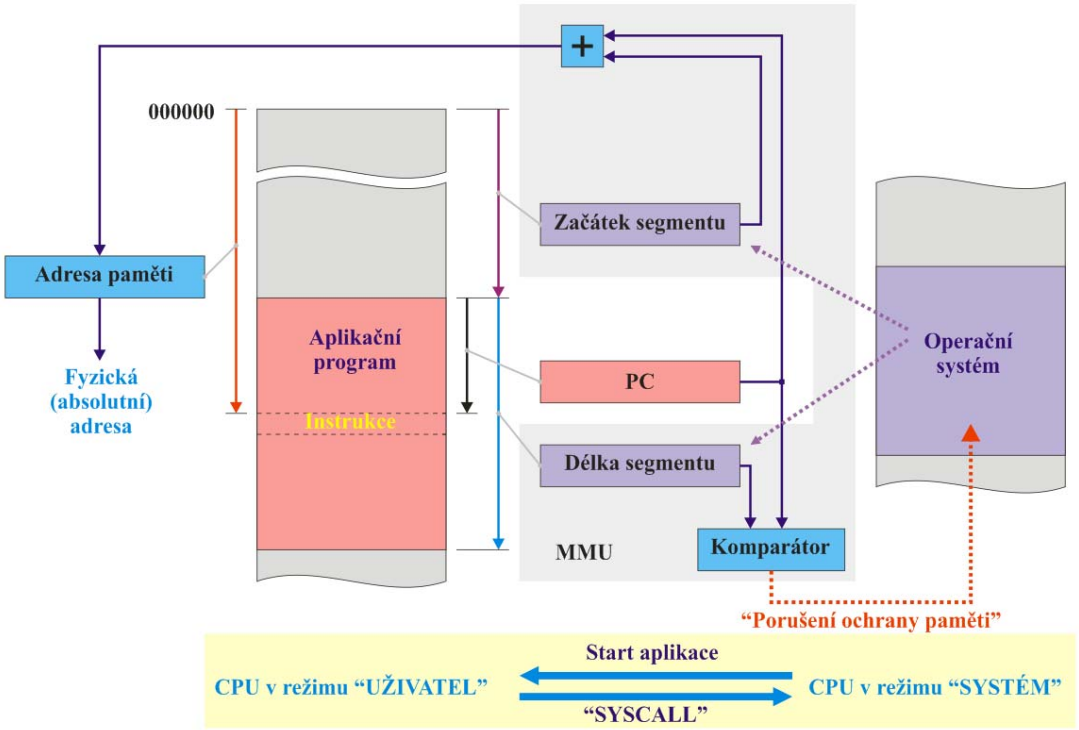


Terminologie:

V OS se používá termín **proces** (task).

- OS plánuje spouštění jednotlivých procesů.
- Aplikační program může pro OS představovat jeden nebo více procesů.

Správa paměti (1)



Správa paměti (2)

- Pro spolupráci s OS může mít procesor prostředky pro správu paměti (MMU – Memory Management Unit).
- MMU obsahuje registry určující polohu začátku segmentu s programem (báze) a jeho délku (limit).
- Registry báze a limit jsou programově přístupné pro OS, ale nejsou přístupné pro aplikaci (CPU rozlišuje režim „Uživatel“ a „Systém“).
- MMU má obvykle několik sad registrů báze + limit (pro kódový, datový, zásobníkový segment atd.).