13. Programátorský model procesoru, instrukce, instrukční soubor, symbolická adresa, operace v registrech, s pamětí, IO operace. Sekvence instrukcí, algoritmizace základních úloh v jazyku symbolických adres. Časování programu, podprogramy, přerušení.

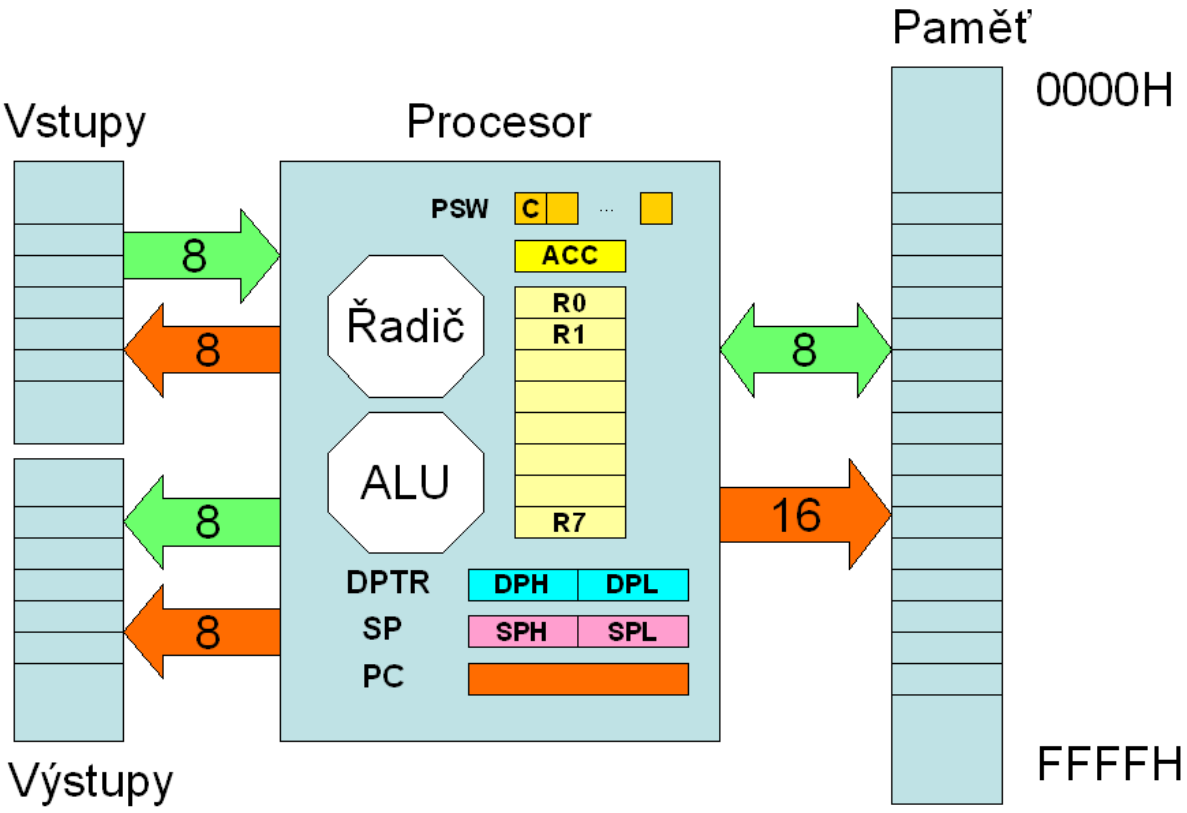
13.1. Programátorský model procesoru

V rámci tohoto předmětu budeme programovat procesor, který je velmi jednoduchý a je do velké míry inspirován procesory řady x51. Abychom ho mohli začít programovat, musíme si představit jeho model.

Tento náš virtuální procesor bude používat harvardskou architekturu, tzn. bude mít oddělenou paměť pro program a paměť pro data. Programovou paměť v našem modelu zanedbáme. Budeme předpokládat, že v ní program který se vykonává. Pro naše účely o ní nic dalšího vědět nepotřebujeme.

Pro zjednodušení budeme předpokládat, že je to procesor typu RISC a tak vykonání každé instrukce zabere právě jeden hodinový cyklus. Tento cyklus bude mít periodu 1 µs, tzn. že tento náš procesor vykoná milión instrukcí za vteřinu.

Celý procesor je osmibitový a proto všechny datové sběrnice a registry budou mít osm bitů. Adresová sběrnice k datové paměti bude šestnáctibitová. To znamená, že do paměti se vejde 2 16 = 65536 bytů dat.



Aby procesor mohl komunikovat s okolním světem, potřebuje i nějaké vstupy a výstupy. Ty budou také osmibitové a pracovat se s nimi bude obdobně jako s datovou pamětí. Adresová sběrnice pro vstupy i výstupy bude osmibitová. To znamená, že vstupů a výstupů může být až 2 8 = 256 bytů.

Uvnitř procesoru se nachází:

- Řadič instrukcí – dekóduje instrukci na vstupu a provede potřebné operace
- ALU – Aritmeticko-logická jednotka, ta provádí vlastní výpočty
- Univerzálně použitelné registry R0..R7 – slouží jako rychlá paměť pro výpočty
- Registr PSW (Program State Word) – tento registr obsahuje stav procesoru, v našem případě budeme používat z tohoto registru pouze dva bity, a to bit C (Carry) který je v jedničce pokud poslední matematická operace skončila přetečením a bit Z (Zero) který je v jedničce pokud výsledek poslední matematické operace byl 0
- Registr DPTR (Data Pointer) – slouží pro nepřímé adresování v datové paměti, je šestnáctibitový ale přistupujeme k němu osmibitově pomocí registrů DPL a DPH
- Registr SP (Stack Pointer) – slouží jako ukazatel na vrchol zásobníku, viz kapitola o zásobníku, podobně jako DPTR je šestnáctibitový ale pracujeme s ním osmibitově pomocí registrů SPL a SPH
- Registr PC (Program Counter) – tento registr obsahuje adresu aktuálně vykonávané instrukce v programové paměti. Je šestnáctibitový, takže program může mít maximálně 65536 instrukcí

13.2. Instrukce

Jednotlivé instrukce jsou uloženy za sebou v programové paměti. Každá instrukce se v paměti skládá z operačního kódu a operandů.

V reálném mikroprocesoru může zápis instrukce v paměti zabírat různé množství paměťových buněk. Instrukce která nemá žádné operandy zabere pouze jednu buňku, kdežto např. instrukce která ukládá konstantu do nějakého registru zabere tři buňky. Registr PC (Program Counter) obsahuje adresu aktuálně vykonávané instrukce a po startu procesoru je v něm hodnota 0. Po každém vykonání instrukce přičte řadič instrukcí do registru PC velikost instrukce v paměti a tak PC ukazuje na další instrukci, která se má vykonat. Jedinou vyjímkou jsou instrukce skoku, po kterých se může vykonávat i jiná instrukce, než která následuje za touto instrukcí.

Operační kód instrukce je v paměti samozřejmě uložen jako číslo. Instrukcí sice procesor nemá mnoho

(jednočipové mikroprocesory mají řádově okolo 100 různých instrukcí), ale stejně je pro člověka nepředstavitelné, že by znal všechny jejich operační kódy zpaměti a pamatoval si jejich význam. Navíc by si musel pamatovat i kódy jednotlivých operandů a v případě skoků by musel při každé změně v programu přepočítat adresy instrukcí v paměti.

Takový program by se nejen obtížně psal, ale i četl a ladil. Proto se procesory neprogramují přímo ve strojovém kódu ale v takzvaném jazyce symbolických adres (JSA), který se někdy označuje jako assembler. V tomto jazyce je každé instrukci přiřazena tzv. mnemonická zkratka. Stejně tak jsou zavedeny jednoduchá jména pro registry procesoru místo jejich čísel. Také je možné používat v takovém programu symbolická návěstí pro označení místa, kam ukazuje instrukce skoku.

Program napsaný v assembleru je pak nutné přeložit pomocí překladače assembleru do strojového kódu. Překladač nahradí jména instrukcí, registrů a návěstí konkrétními čísly a poskládá instrukce za

sebe do paměti procesoru. Toto je triviální úloha, která nám ale přináší obrovský komfort při psaní programů na úrovni jednotlivých instrukcí.

V následujícím textu si budeme postupně představovat jednotlivé instrukce našeho virtuálního procesoru. Nebudeme se ale již zabývat jejich číselnou interpretací a ukládáním v paměti, ale budeme pracovat na úrovni JSA.

Zápis začíná vždy mnemotechnickou zkratkou instrukce a následuje seznam operandů, které jsou odděleny čárkou. Operandem může být:

- registr procesoru
- konstanta
- místo v paměti určené přímou adresou
- místo v paměti určené nepřímou adresou

13.2.1. Přesun dat

MOV X, Y

Instrukce pro přesun dat má mnemotechnickou zkratku MOV, což je zkratka z anglického slova „move“. Tato instrukce provede překopírování osmibitových dat ze zdroje Y do cíle X.

instrukce kopíruje data osmibitově, existují i vyjímky. Do registrů SP a DPTR je možné vložit přímo šestnáctibitovou konstantu.

Př.: Vyměnit obsahy registrů R0 a R1:

- mov A, R0
- mov R2, A
- mov A, R1
- mov R0, A
- mov A, R2
- mov R1, A

Jako vedlejší efekt program také změní obsahy registrů R2 a Acc.

13.2.2. Vstup a výstup

in A, Adresa8

Instrukce IN zkopíruje data z vybraného vstupního portu se zadanou adresou do akumulátoru – registru A. Vstupní porty nemají funkci registru. Jejich obsah je v každém okamžiku dán děním mimo počítač. Instrukce IN zkopíruje obsah portu do registru A jen jednorázově, v okamžiku vykonání instrukce.

```
out Adresa8, A
```

Instrukce OUT zkopíruje data z akumulátoru na adresou vybraný výstupní port. Výstupní porty mají funkci registru, jejich obsah je dán posledním provedením instrukce OUT.

13.2.3. Logické operace

```
cpl A
```

Instrukce CPL (z anglického slova complement=doplněk) provede **negaci** všech osmi bitů v akumulátoru. To znamená, že každý bit, který byl v hodnotě 0 změní na 1 a naopak.

```
anl A, X
```

Instrukce ANL provede po bitech **logický součin** obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A. Operace součin se provede na dvojicích stejnolehlých bitů obou operandů.

```
orl A, X
```

Instrukce ORL provede po bitech **logický součet** obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.

```
xrl A, X
```

Instrukce ORL provede po bitech **exkluzivní logický součet** obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.

13.2.4. Rotace

Následující instrukce rotují jednotlivé bity doprava (RR a RRC) nebo doleva (RL a RLC). Pracují vždy s akumulátorem nebo s akumulátorem a jednobitovým registrem C (ten je součástí registru PSW).

```
rr A
rrc A
rl A
rlc A
```

13.2.5. Aritmetické operace

```
inc X
```

Instrukce INC (incrementation) provede **zvýšení** hodnoty operandu X o jedničku.

```
dec X
```

Instrukce DEC (decrementation) provede snížení hodnoty operandu X o jedničku.

```
add A, X
```

Instrukce ADD provede aritmetický součet obsahu akumulátoru a druhého operandu X. Výsledek uloží do akumulátoru A.

```
addc A, X
```

Instrukce ADD provede aritmetický součet obsahu akumulátoru, druhého operandu X **a jednobitového registru** C. Výsledek uloží do akumulátoru A.

```
subb A, X
```

Instrukce SUBB provede aritmetický rozdíl obsahu akumulátoru, druhého operandu X a **jednobitového registru C**. Výsledek uloží do akumulátoru A.

13.2.6. Skoky

```
jmp Adresa16
```

Instrukce JMP (jump) je instrukce nepodmíněného skoku. To znamená, že vykonání této instrukce způsobí nahrání nové adresy do registru PC a následně se vykonávají instrukce od této adresy. Adresu lze zadat přímo jako číselnou konstantu, ale to je velmi nepraktické. Místo toho se používají pro skoky symbolická návěstí. Takové návěstí si definujeme pomocí identifikátoru zakončeného dvojtečkou a danou instrukci můžeme odkazovat tímto jménem. Každé návěstí může být v programu definováno pouze jednou. Pokud chceme provést skok na aktuální instrukci, můžeme místo návěstí použít znak $. V tomto případě provede instrukce JMP skok na sebe a vytvoří tím nekonečnou smyčku, tzv. dynamický stop.

Následující instrukce jsou tzv. podmíněné skoky. Tyto instrukce provedou skok na jiné místo programu pouze tehdy, je-li splněna nějaká podmínka. Používají se pro větvení programu.

```
jz Adresa16
```

Jump if Zero: Pokud je jednobitový registr Z = 1 (tzn. výsledek poslední matematické nebo logické operace byl nula) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

```
jnz Adresa16
```

Jump if Not Zero: Pokud je jednobitový registr Z = 0 (tzn. výsledek poslední matematické nebo logické operace byl nenulový) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

```
jc Adresa16
```

Jump if Carry: Pokud je jednobitový registr C = 1 (tzn. při poslední matematické operaci došlo k přetečení) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

jnc Adresa16

Jump if Not Carry: Pokud je jednobitový registr C = 0 (tzn. při poslední matematické operaci nedošlo k přetečení) pak se tato instrukce chová jako instrukce JMP. Jinak se chová jako instrukce NOP (No Operation).

cjne A, #data8, Adresa16

Compare and Jump if Not Equal: Tato instrukce provede porovnání obsahu akumulátoru s daty. Pokud A<>data8, pak se provede skok na zadanou adresu. Jinak se pokračuje další instrukcí. Navíc pokud A<data8, pak se nastaví jednobitový registr C na 1, jinak na 0.

cjne Rn, #data8, Adresa16

Tato instrukce je naprosto stejná, jen místo akumulátoru pracuje s registry R0..R7.

djnz Rn, Adresa16

Decrement and Jump if Not Zero: Instrukce nejprve sníží o jedničku obsah registru a pokud je výsledek různý od nuly, pak se provede skok na zadanou adresu. Jinak se pokračuje následující instrukcí.

Instrukce JZ, JNZ, JC a JNC se používají zejména pro větvení programu na základě nějaké podmínky nebo pro tvorbu podmíněných cyklů.

Instrukce CJNE a DJNZ jsou vhodné pro vytváření nepodmíněných cyklů.

Příklady použití instrukcí skoků:

13.3. Instrukční soubor

Instrukční sada (tzv. ISA - Instruction Set Architecture), někdy nepřesně architektura procesoru, je obecný popis organizačních, funkčních a provozních principů procesoru, z pohledu programátora je to seznam dostupných mechanismů pro programování. Programový model procesoru se může skládat například z následujících prvků:

- seznamu instrukcí procesoru
- datových typů
- dostupných režimů, jež jsou k dispozici
- seznamu registrů
- pravidel pro manipulaci s výjimkami a přerušeními

Instrukce přesunu				

mov X,Y	mov A, Rn
	mov A, DPH
	mov A, DPL
	mov A, SPH
	mov A, SPL
		mov A, @Rn
		mov DPH, A
		mov DPL, A
		mov SPH, A
		mov SPL, A
			mov A, #Data8
			mov Rn, #Data8
			mov DPTR, #Data16
			mov SP, #Data16
			mov A, Adresa8
				mov A, @Rn
				mov @Rn, A
				mov A, @DPTR
				mov @DPTR, A
				mov Adresa8, A

Instrukce vstupu a výstupu				
in A, Adresa8			out Adresa8, A	
Logické operace				
cpl A				
anl A, X	anl A, Rn	anl A, #Data8	anl A, @Rn	anl A, Adresa8
orl A, X	orl A, Rn	orl A, #Data8	orl A, @Rn	orl A, Adresa8
xrl A, X	xrl A, Rn	xrl A, #Data8	xrl A, @Rn	xrl A, Adresa8
Instrukce rotací				
rr A	rrc A	rl A	rlc A	
Aritmetické instrukce				
inc X	inc A	inc Rn	inc DPTR	
dec X	dec A	dec Rn		
add A, X	add A, Rn	add A, #Data8	add A, @Rn	add A, Adresa16
addc A, X	addc A, Rn	addc A, #Data8	addc A, @Rn	addc A, Adresa16
subb A, X	subb A, Rn	subb A, #Data8	subb A, @Rn	subb A, Adresa16
Instrukce skoků				
jmp Adresa16	cjne A, #data8, Adresa16	cjne Rn, #data8, Adresa16	djnz Rn, Adresa16	
jz Adresa16	jnz Adresa16	jc Adresa16	jnc Adresa16	
Práce se zásobníkem				
push X	push A	push Rn	push PSW	
pop X	pop A	pop Rn	pop PSW	
Podprogramy				
call Adresa16	ret			
Prázdná instrukce				
nop				

13.4. Symbolická adresa

Symbolické adresy / instrukce je symbolická reprezentace jednotlivých strojových instrukcí a konstant potřebných pro vytvoření strojového kódu programu pro daný procesor. Symbolickou reprezentaci tvoří zpravidla výrobce procesoru a je založena na mnemotechnických zkratkách, které vyjadřují, co daná strojová instrukce dělá, označují symbolicky registr, slovní zkratku podmínky a podobně. JSA je proto závislý na konkrétním procesoru a zapsaný program je obtížně přenositelný na jinou platformu (na rozdíl od vyšších programovacích jazyků).

Pro překlad JSA do strojového kódu se používá program, který nazýváme assembler (překladač). Oba názvy jsou často nesprávně zaměňovány.

Symbolicky zapsané strojové instrukce jsou při překladu nahrazeny odpovídajícím strojovým kódem

13.5. Operace v registrech s pamětí

Instrukce přesunu				

mov X,Y	mov A, Rn
	mov A, DPH
	mov A, DPL
	mov A, SPH
	mov A, SPL
		mov A, @Rn
		mov DPH, A
		mov DPL, A
		mov SPH, A
		mov SPL, A
			mov A, #Data8
			mov Rn, #Data8
			mov DPTR, #Data16
			mov SP, #Data16
			mov A, Adresa8
				mov A, @Rn
				mov @Rn, A
				mov A, @DPTR
				mov @DPTR, A
				mov Adresa8, A

Instrukce vstupu a výstupu				
in A, Adresa8			out Adresa8, A	
Logické operace				
cpl A				
anl A, X	anl A, Rn	anl A, #Data8	anl A, @Rn	anl A, Adresa8
orl A, X	orl A, Rn	orl A, #Data8	orl A, @Rn	orl A, Adresa8
xrl A, X	xrl A, Rn	xrl A, #Data8	xrl A, @Rn	xrl A, Adresa8
Instrukce rotací				
rr A	rrc A	rl A	rlc A	
Aritmetické instrukce				
inc X	inc A	inc Rn	inc DPTR	
dec X	dec A		dec Rn	
add A, X	add A, Rn	add A, #Data8	add A, @Rn	add A, Adresa16
addc A, X	addc A, Rn	addc A, #Data8	addc A, @Rn	addc A, Adresa16
subb A, X	subb A, Rn	subb A, #Data8	subb A, @Rn	subb A, Adresa16

13.6. I/O operace

in A, Adresa8
out Adresa8, A

13.7. Sekvence instrukcí

Příklad blikání diody:

```
LOOP:  mov    A, #0
       out    12, A
       mov    R1, #0
W1:    mov    R0, #0
       djnz   R0, $
       djnz   R1, W1
       mov    A, #1
       out    12, A
       mov    R1, #0
W2:    mov    R0, #0
       djnz   R0, $
       djnz   R1, W2
       jmp    LOOP
```

13.8. Algoritmizace základních úloh v jazyku symbolických adres

/

13.9. Časování programu

```
call Adresa16
nop
```

13.10. Podprogramy

```
call Adresa16
ret
```

Podobně jako při skoku se při volání podprogramu předává řízení instrukci na jiném místě programu. Na rozdíl od skoku se ale předpokládá návrat do místa, odkud byl podprogram volán. Proto je nutné nejprve uložit adresu aktuální instrukce (registr PC – Program Counter) a pak teprve provést skok. Na konci podprogramu se tato adresa opět vyzvedne a program pokračuje na původním místě, odkud byl podprogram volán. K tomuto slouží instrukce CALL a RET.

Instrukce **CALL** uloží do zásobníku (stejným způsobem jako PUSH) nejprve nižší a poté vyšší byte adresy té instrukce, která následuje za instrukcí CALL. Potom se provede skok na zadanou adresu.

Instrukce **RET** odebere ze zásobníku nejprve vyšší a poté nižší byte adresy instrukce, které poté předá řízení.

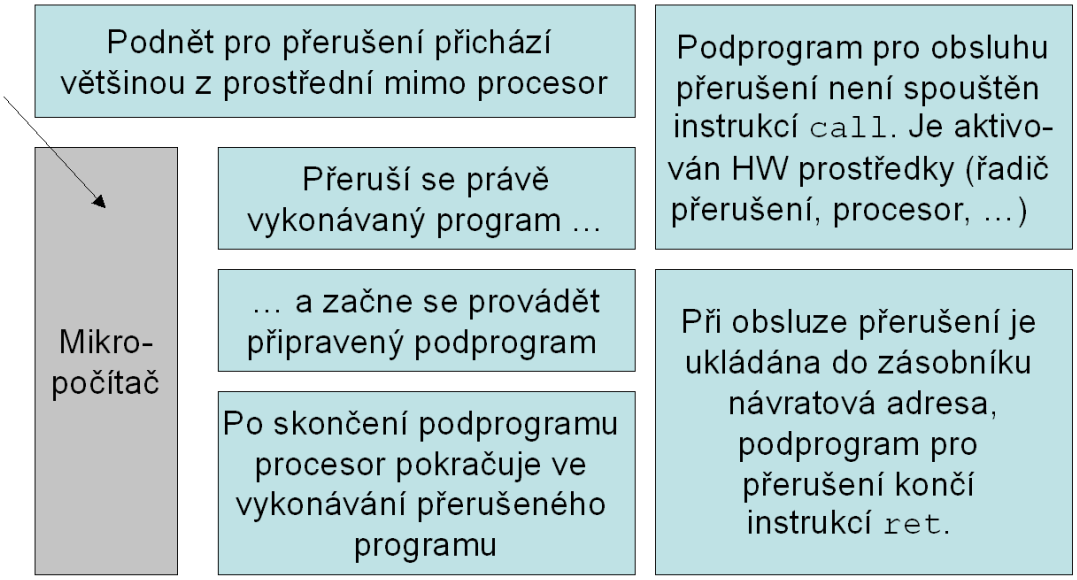
Každým voláním podprogramu se nejprve velikost zásobníku zvětší o dva byte (uloží se návratová adresa) a po návratu z podprogramu se opět zmenší o dva byte (odebere se návratová adresa). Proto opět platí, že počet provedených instrukcí CALL musí být stejný jako instrukcí RET.

Dále by se v rámci podprogramu neměli používat instrukce skoku, které míří mimo aktuální podprogram. Stejně tak mimo podprogram by se neměli používat instrukce skoku, které míří do podprogramu.

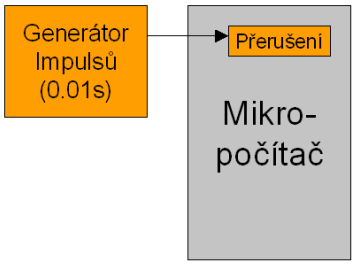
Z podprogramu je možné provádět volání dalších podprogramů a to do téměř libovolného množství úrovní. Omezením je pouze velikost volné paměti pro zásobník (pro ukládání návratových adres).

13.11. Přerušení

Systém přerušení procesoru je velmi důležitý zejména pro obsluhu zařízení, připojených k procesoru. Pokud je přerušení správně nastaveno, může se procesor věnovat vykonávání hlavního programu a nemusí se zdržovat testováním, jestli není potřeba vykonat obslužnou rutinu. Pokud tato potřeba nastane, pak se aktivuje žádost o přerušení procesoru a procesor provede automaticky skok na předem nastavenou obslužnou rutinu. Po jejím vykonání se procesor opět vrátí k vykonávání hlavního programu. Zdrojem přerušení může být podnět vně procesoru (např. stisk tlačítka) nebo uvnitř procesoru (např. vnitřní časovač).



Příklad na přerušení



Př.: Přerušení 0.01s

```
IntT1: push A
       mov A, 0AAh
       jz IT1
       dec A
       mov 0AAh, A
IT1_1: pop A
       ret
```

Dokud je obsah proměnné na adrese 0AAh větší než 0, je periodicky v taktu 0.01s snižován o jedničku.

Hlavní program a ostatní podprogramy mohou (ale nemusí) tuto funkci použít pro řešení úloh Časování.

Ve složitějších aplikacích je možno podobných ČASOVAČŮ zřídit mnoho (každý na jiné adrese).