Mikroprocesory I.

Zimní semestr

Doc. Ing. Miroslav Čech, CSc.

míst. 234 – Trojanova, 2.NP

tel.: 778 532 036

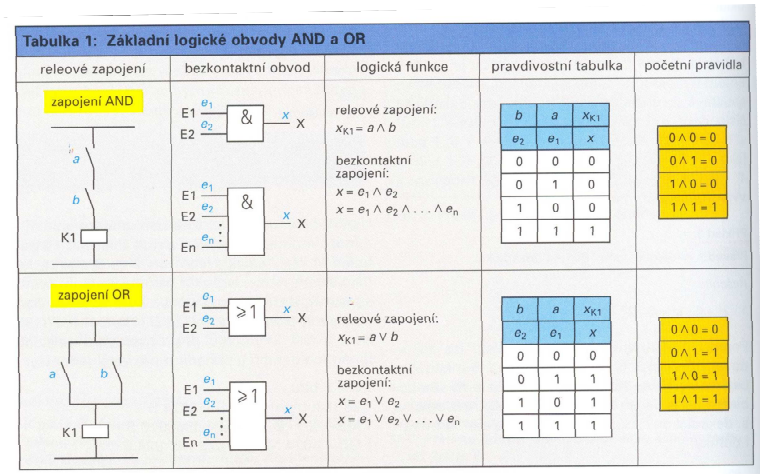
E-mail: miroslav.cech@fjfi.cvut.cz

http://people.fjfi.cvut.cz/cechmiro/vyuka.html

Literatura

- V.Jirovský: Principy počítačů, Matfyz Press, Praha 2000
- J.Blatný, K.Kristoufek, Z.Pokorný, J.Kolenička: Číslicové počítače, SNTL, Praha 1982
- J.Pinker: Mikroprocesory a mikropočítače, Ben, Praha 2004
- J.Strelec, M.Liška: Architektury procesorů RISC, Grada, Praha 1992
- M.Brandejs: Mikroprocesory INTEL Pentium a spol., Grada, Praha, 1994
- H. Häberle: Průmyslová elektronika a informační technologie, Europa-Sobotáles cz., Praha 2003
- J.Pinker,M.Poupa:Číslicové systémy a jazyk VHDL, Ben, Praha 2006
- P. Tvrdík: Architektura superskalárních procesorů, PC World č.11/1994
- Firemní literatura Microchip
- Firemní literatura Intel
- www.wikipedia.com

Základní logické obvody

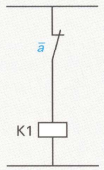
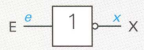


Tabulka 1: Základní logické obvody AND a OR

releové zapojení	bezkontaktní obvod	logická funkce	pravdivostní tabulka	početní pravidla
zapojení AND	E1, E2, &, x, X; E1, E2, En, &, x, X	releové zapojení: $x_{K1} = a \wedge b$; bezkontaktní zapojení: $x = e_1 \wedge e_2$, $x = e_1 \wedge e_2 \wedge \ldots \wedge e_n$	b a x_{K1}; e_2 e_1 x; 0 0 0; 0 1 0; 1 0 0; 1 1 1	$0 \wedge 0 = 0$; $0 \wedge 1 = 0$; $1 \wedge 0 = 0$; $1 \wedge 1 = 1$
zapojení OR	E1, E2, ≥1, x, X; E1, E2, En, ≥1, x, X	releové zapojení: $x_{K1} = a \vee b$; bezkontaktní zapojení: $x = e_1 \vee e_2$, $x = e_1 \vee e_2 \vee \ldots \vee e_n$	b a x_{K1}; e_2 e_1 x; 0 0 0; 0 1 1; 1 0 1; 1 1 1	$0 \wedge 0 = 0$; $0 \wedge 1 = 1$; $1 \wedge 0 = 1$; $1 \wedge 1 = 1$

Základní logické obvody

Tabulka 1: Obvod logické negace (logický invertor)

releové zapojení	bezkontaktní obvod	logická funkce	pravdivostní tabulka	početní pravidla
		releová zapojení $x_{K1} = \bar{a}$ bezkontaktní zapojení $x = \bar{e}$	a | x_{K1} e | x 0 | 1 1 | 0	$\bar{0} = 1$ $\bar{1} = 0$

Výpočetní pravidla - Booleova algebra

Tabulka 1: Výpočetní pravidla

jméno pravidla	příklad
Komutativní zákon o záměně pořadí	$a \wedge b = b \wedge a$ $a \vee b = b \vee a$
Asociativní zákony o sdružování do skupin	$a \wedge b \wedge c = (a \wedge b) \wedge c = a \wedge (b \wedge c) = (a \wedge c) \wedge b$ $a \vee b \vee c = (a \vee b) \vee c = a \vee (b \vee c) = (a \vee c) \vee b$
1. distributivní zákon 2. distributivní zákon	$(a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)$ $(a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$
de Morganovy zákony	$\overline{a \wedge b} = \bar{a} \vee \bar{b}$ $\overline{a \vee b} = \bar{a} \wedge \bar{b}$

Booleova algebra

Tabulka 2: Početní pravidla pro tři základní logické operace s jednou logickou proměnnou

$0 \wedge a = 0$	$0 \vee a = a$
$1 \wedge a = a$	$1 \vee a = 1$
$a \wedge a = a$	$a \vee a = a$
$a \wedge a \wedge \ldots \wedge a = a$	$a \vee a \vee \ldots \vee a = a$
$a \wedge \bar{a} = 0$	$a \vee \bar{a} = 1$
$\bar{\bar{a}} = a$	

Číselné soustavy

- Čísla v číselných soustavách zaznamenáváme pomocí abecedy znaků – **číslic,** z nichž sestavujeme **číselné obrazy**
- Číselné soustavy dělíme na
  - **Polyadické (poziční)** – desítková, dvojková, šestnáctková
  - **Nepolyadické (nepoziční)**– římské číslice (MCXXX), zbytkové třídy

Polyadické soustavy

- V polyadické soustavě o jednom základu **z** (obecně může mít i více základů) definujeme číslice a_i , $0 \le a_i < z$
- Obrazem čísla $A = \sum_{i=-m}^{n-1} a_i z^i$

  je uspořádaná k-tice $a_{n-1}, a_{n-2}, \ldots a_1, a_0, a_{-1}, \ldots a_{-m}$
- Desítková soustava

  číslo $123 = 1.10^2 + 2.10^1 + 3.10^0$
- Dvojková soustava

  číslo $1011 = 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0$
- Šestnáctková soustava (číslice A=10, B=11, … F=15)

  číslo $1AB3 = 1.16^3 + 10.16^2 + 11.16^1 + 3.16^0$

Vyjádření čísel v různých číselných soustavách

Tabulka 1: Zápisy čísel 0 až 15 v různých číselných soustavách

dekadický zápis	hexadeci-mální zápis	oktálový zápis	binární zápis
00	0	00	0000
01	1	01	0001
02	2	02	0010
03	3	03	0011
04	4	04	0100
05	5	05	0101
06	6	06	0110
07	7	07	0111
08	8	10	1000
09	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Mocniny 2

$2^0 = 1$	$2^9 = 512$	$2^{20} = 1\,048\,576$
$2^2 = 4$	$2^{10} = 1\,024$	$2^{32} = 4\,294\,967\,296$
$2^3 = 8$	$2^{11} = 2\,048$	$2^{36} = 68\,719\,476\,736$
$2^4 = 16$	$2^{12} = 4\,096$	$2^{40} = 1\,099\,511\,627\,776$
$2^5 = 32$	$2^{13} = 8\,192$	$2^{46} = 70\,368\,744\,177\,664$
$2^6 = 64$	$2^{14} = 16\,384$	
$2^7 = 128$	$2^{15} = 32\,768$	
$2^8 = 256$	$2^{16} = 65\,536$	

$2^{64} = 18\,446\,744\,073\,709\,551\,616$ (exa)

Konverze mezi polyadickými soustavami o různém základu

Vyjádříme X ve dvou číselných soustavách

$X = a_{n-1}r^{n-1} + a_{n-2}r^{n-2} + \ldots + a_0r^0 =$
$a_0 + r.(a_1 + r.(a_2 + \ldots + r.(a_{n-2} + r.a_{n-1})))\ldots)$
$X = b_0 + t.(b_1 + t.(b_2 + \ldots + t.(b_{n-2} + t.b_{n-1})))\ldots)$

Při dělení rovnice základem t dostaneme polynom ve tvaru $X/t = Q + R$
kde

$Q = b_1 + t.(b_2 + \ldots + t.(b_{n-2} + t.b_{n-1})))\ldots)$
$R = b_0$

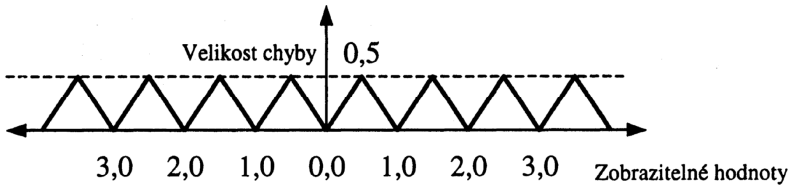
Dalším dělením získáme koeficienty $b_1, b_2, \ldots$

Výsledná chyba v zobrazení čísla

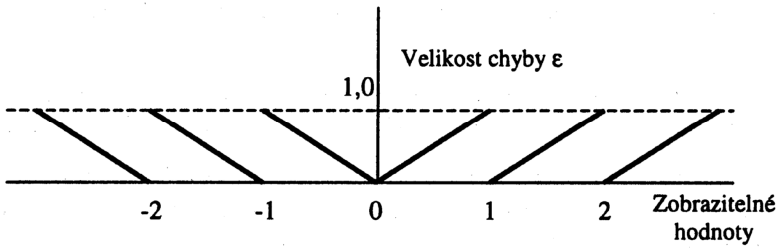
Při pořizování záznamu čísel v počítači může výsledná chyba v zobrazení čísla obecně vznikat ze čtyř příčin:

- 1. chyba vznikající při pořizování čísla (např. omezený rozsah přesnosti při převodu analogových veličin)
- 2. chyba způsobená zavedením stupnice (angl. scaling error) - tato chyba vzniká tak, že na rozdíl od analogové stupnice je digitální stupnice omezena konečným počtem zobrazitelných hodnot,
- 3. chyba zanedbáním části čísla (angl. truncation error), kdy pro nedostatek míst pro zobrazení čísla se bity nejnižších vah odseknou
- 4. chyba vznikající zaokrouhlením (angl. rounding error)

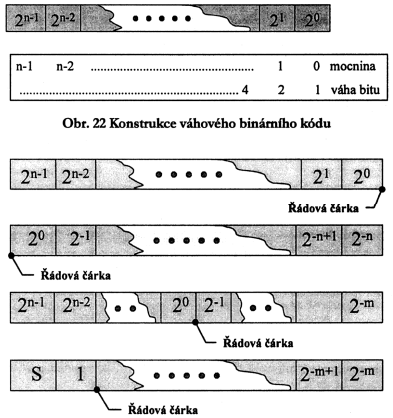
Průběh chyby vzniklé zavedením stupnice nebo zaokrouhlením



Průběh chyby vzniklé odseknutím



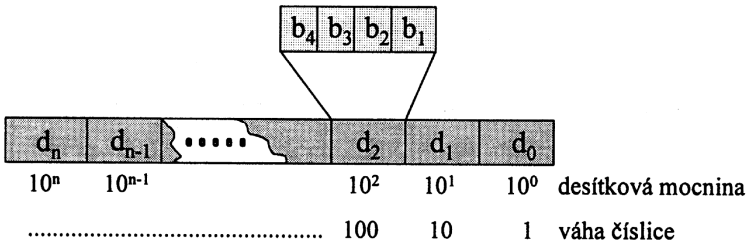
Čísla v pevné řádové čárce



Obr. 22 Konstrukce váhového binárního kódu

Nejčastěji používané formáty záznamu v pevné řádové čárce

Čísla v kódu BCD
(**B**inary **C**oded **D**ecimal)



Každá desítková číslice 0-9 je vyjádřena v binárním kódu

Číslo 123 je vyjádřeno jako 0001 0010 0011

Binární čísla se znaménkem
Reprezentace

- Zavedení znaménka
- Zobrazení s posunutím nuly
- Dvojkový doplněk
- Jedničkový doplněk

Zavedení znaménka

- Např. v 8-bitovém čísle vyhradíme jeden bit (zpravidla nejvyšší) pro znaménko + (0) či – (1), ostatních 7 bitů vyjadřuje kladné číslo

  3 = 0 0000011

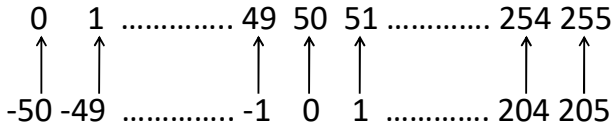
  - 3 = 1 0000011

  Znaménkový bit

- Nevýhody:
  - dvě nuly +0 (00000000), -0 (10000000)
  - složité sčítání a odčítání

Zobrazení s posunutím nuly

- 8 bitové číslo zobrazuje čísla v rozsahu 0-255
- Zadefinujeme-li, že číslo 50 znamená 0 – můžeme zobrazit čísla v rozsahu -50 až 205 – tzv. posuneme nulu

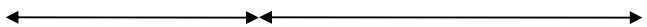


- Používá se pro vyjádření exponentu v číslech s plovoucí řádovou čárkou nebo v elektronice u A/D převodníků

Dvojkový doplněk

- Záporné číslo odpovídající kladnému číslu **x** s opačným znaménkem definujeme jako doplněk do čísla z^n (pro 8 bitové binární číslo doplněk do čísla $2^8 = 256$)

0 1 …….. 49 50 51 …….. 255 256



49
00110001

-49 = 207
11001111

Dvojkový doplněk

- Nejčastěji používaná reprezentace
- Jednoduchá změna znaménka – provede se negace bit po bitu a přičte 1
- Pouze jedna nula
- Nesymetrický interval zobrazení – 8 bitové číslo zobrazí interval <-128, 127>
- Nejvyšší bit = znaménkový bit + = 0, - = 1
- Stačí sčítačka - odečítání 5-3 realizujeme jako sčítání 5 + (-3)

Dvojkový doplněk

Číslo 3 = 00000011

Číslo -3:

3 =	00000011
Negace	11111100
+1	00000001
0	11111101

Případný přenos Zanedbává se

Znaménkový bit

Výpočet 5-3 = 5+(-3)

5 =	00000101
-3 =	11111101
1	00000010 = 2

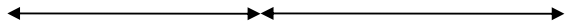
Přenos Zanedbává se

Znaménkový bit

Jedničkový doplněk (inverzní kód)

- Záporné číslo odpovídající kladnému číslu **x** s opačným znaménkem definujeme jako doplněk do čísla z^n-1 (pro 8 bitové binární číslo doplněk do čísla $2^8-1 = 255$)

0 1 …….. 49 50 51 …….. 255



49
00110001

-49 = 206
11001110

Jedničkový doplněk

- Nevýhody:
  - dvě nuly +0 (00000000), -0 (11111111)
  - složité sčítání a odčítání

Číslo 3 = 00000011

Číslo -3:

3 = 00000011

Negace 11111100 = -3

Desítkový doplněk v BCD kódu

- Můžeme definovat záporné číslo jako doplněk do maximálního zobrazeného čísla+1 (obdoba dvojkového doplňku)
- Příklad dvouciferné desítkové číslo 0-99, záporné číslo je desítkový doplněk do čísla 100

  -1 = 99 , -10 = 90 , atd

11 – 10 = 11+(90) = 101 = 1 (řád 100 zanedbáváme)

10 – 11 = 10 + 89 = 99 = -1

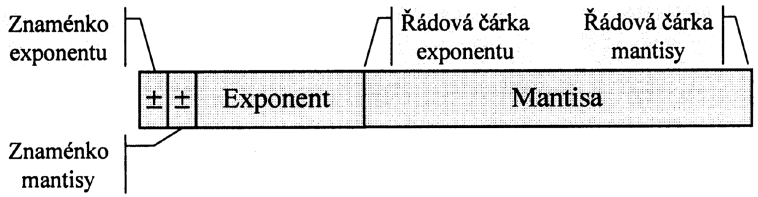
Devítkový doplněk v BCD kódu (Inverzní kód)

- Můžeme definovat záporné číslo jako doplněk do maximálního zobrazeného čísla (obdoba jedničkového doplňku)
- Příklad: dvouciferné desítkové číslo 0-99, záporné číslo je desítkový doplněk do čísla 99

  -1 = -01 = 98 , -10 = 89 , atd.

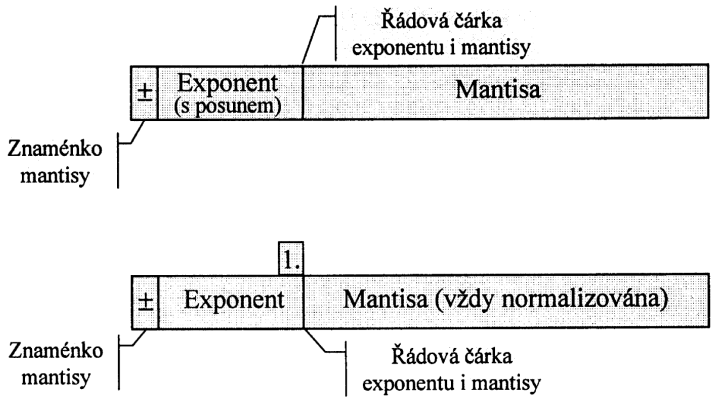
záměna 0 ↔ 9, 1 ↔ 8, 2 ↔ 7, …. (inverze)

Čísla v plovoucí řádové čárce



Přímý záznam čísla v plovoucí řádové čárce

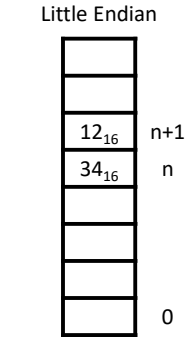
Čísla v plovoucí řádové čárce



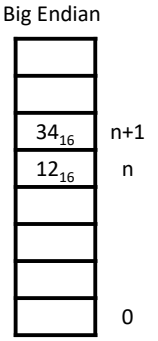
Nejčastěji používané záznamy v plovoucí řádové čárce

Reprezentace víceslabikových čísel v paměti

Pokud např. 16tibitové číslo 1234_{16} je uloženo v paměti s organizací 8 bit, jsou dvě možnosti:



Procesory Intel
LSB na nižší adrese
MSB na vyšší adrese



Procesory Motorola
LSB na vyšší adrese
MSB na nižší adrese

Vyjádření znaků

- ASCII
- EBCDIC
- Latin 2
- Unicode
- .....
- .....

Kódovací tabulka ASCII

(American Standard Code for Information Interchange)

Tabulka 1: ASCII-kód

bit číslo (zpra-va)	7	0	0	0	0	1	1	1	1
	6	0	0	1	1	0	0	1	1
	5	0	1	0	1	0	1	0	1
4 3 2 1		řídicí znaky		cifry, čísla zvláštní znaky					
0 0 0 0		NUL	DLE	SP	0	@	P	‘	p
0 0 0 1		SOH	DC1	1	1	A	Q	a	q
0 0 1 0		STX	DC2	“	2	B	R	b	r
0 0 1 1		ETX	DC3	#	3	C	S	c	s
0 1 0 0		EOT	DC4	$	4	D	T	d	t
0 1 0 1		ENQ	NAK	%	5	E	U	e	u
0 1 1 0		ACK	SYN	&	6	F	V	f	v
0 1 1 1		BEL	ETB	‘	7	G	W	g	w
1 0 0 0		BS	CAN	(	8	H	X	h	x
1 0 0 1		HT	EM	)	9	I	Y	i	y
1 0 1 0		LF	SUB	*	:	J	Z	j	z
1 0 1 1		VT	ESC	+	;	K	[	k	{
1 1 0 0		FF	FS	,	<	L	\	l	|
1 1 0 1		CR	GS	–	=	M	]	m	}
1 1 1 0		SO	RS	.	>	N	^	n	~
1 1 1 1		SI	US	/	?	O	_	o	DEL

Tabulka 2: Význam řídicích znaků (výběr)	
NUL	Null, vynulování přijímače, prázdná značka
SOH	Start of heading, začátek záhlaví
STX	Start of text, začátek textu
ETX	End of text, konec textu
EOT	End of transmission, konec přenosu
ENQ	Enquiry, dotaz
BEL	Bell, zvonek, akustický signál
BS	Backspace, návrat o znak
LF	Linefeed, posun na další řádek
CR	Carriage return, návrat vozíku (na začátek řádku)
ESC	Escape, změna, autoregistr 2
ACK	Acknowledge, potvrzení příjmu
DEL	Delete, výmaz, zrušení, odstranění
DC	Device control (1...4), ovládání zařízení
DC1	start snímače
DC2	zapnutí děrování

Kódovací tabulka EBCDIC

(Extended Binary Coded Decimal Interchange Code)

bity 4 - 7 →

bity 0 - 3 ↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-						(	)	\	0
1	SOH	DC1	SOS				/		a	j	~		A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	TM							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	ETB	UC					f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
8	GE	CAN							h	q	y		H	Q	Y	8
9	RLF	EM						`	i	r	z		I	R	Z	9
A	SMM	CC	SM		¢	!	|	:							
B	VT	CUL	CU2	CU3	.	$	,	#								
C	FF	IFS		DC4	>	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	<	=								
F	SI	IUS	BEL	SUB	|	¬	?	"								

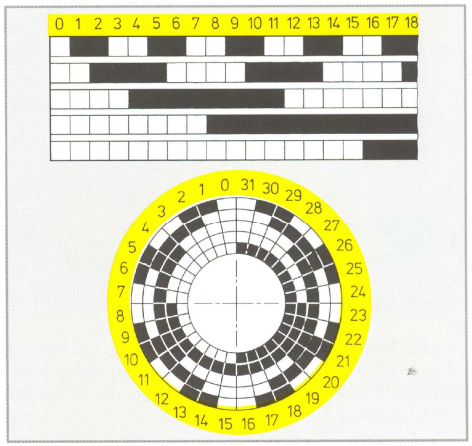
Kódovací tabulka Latin 2

bity 4 - 7 →

bity 0 - 3 ↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	SP	0	@	P	´	p			NB SP	°	Ŕ	Đ	ŕ	đ
1	SOH	DC1	!	1	A	Q	a	q			Ą	ą	Á	Ń	á	ń
2	STX	DC2	"	2	B	R	b	r			˘	˛	Â	Ň	â	ň
3	ETX	DC3	#	3	C	S	c	s			Ł	ł	Ă	Ó	ă	ó
4	EOT	DC4	$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5	ENQ	NAK	%	5	E	U	e	u			Ľ	ľ	Ĺ	Ő	Í	ő
6	ACK	SYN	&	6	F	V	f	v			Ś	ś	Ć	Ö	ć	ö
7	BEL	ETB	'	7	G	W	g	w			§	¨	Ç	×	ç	□
8	BS	CAN	(	8	H	X	h	x				¸	Č	Ř	č	ř
9	HT	EM	)	9	I	Y	i	y			Š	š	É	Ů	é	ů
A	LF	SUB	*	:	J	Z	j	z			Ş	ş	Ę	Ú	ę	ú
B	VT	ESC	+	;	K	[	k	{			Ť	ť	Ë	Ű	ë	ű
C	FF	FS	,	<	L	\	l	|		Ź	ź	Ě	Ü	ě	ü
D	CR	GS	-	=	M	]	m	}			SHY	"	Í	Ý	í	ý
E	SO	RS	.	>	N	↑	n	~			Ž	ž	Î	Ţ	î	ţ
F	SI	US	/	?	O	_	o	DEL			Ż	ż	Ď	ß	ď	˙

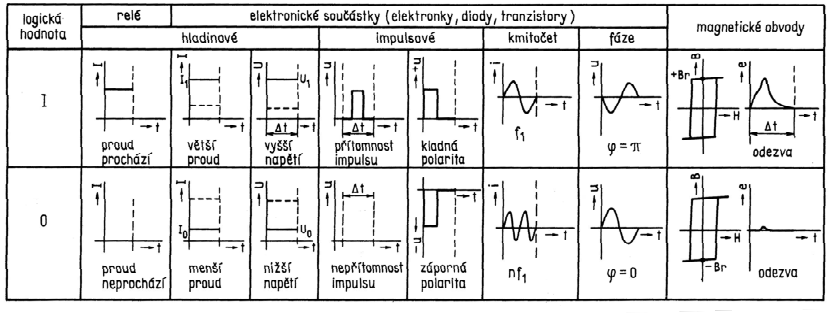
Speciální kódy
Grayův kód

0 0000
1 0001
2 0011
3 0010
4 0110
5 0111
6 0101
7 0100
8 1100
9 1101
10 1111
11 1110
12 1010
13 1011
14 1001
15 1000



Při přechodu do následujícího stavu je změna pouze v jednom bitu – > nenastávají hazardní stavy

Způsoby vyjádření logické veličiny

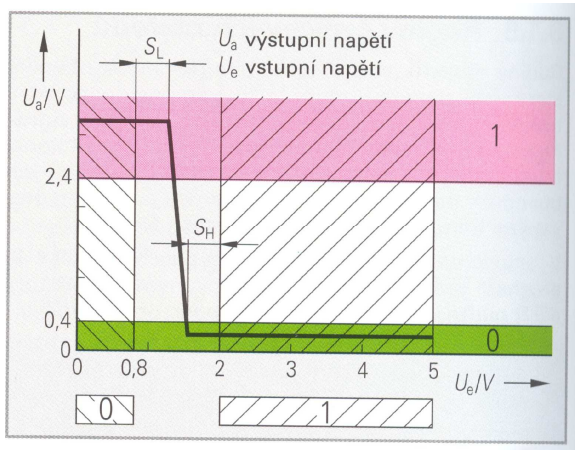


Tabulka 1: Rodiny bipolárních číslicových obvodů	
zkratka	název
TTL	standardní TTL (Tranzistor-Tranzistor-Logik)
H-TTL	rychlé TTL (High-Speed-TTL)
L-TTL	TTL malého výkonu (Low-Power-TTL)
AS-TTL	TTL Schottky (Advanced-Schottky-TTL)
LS-TTL	TTL Schottky malého výkonu (Low-Power-Schottky-TTL)
ALS-TTL	Advanced-Low-Power-Schottky-TTL
ECL	emitorově vázaná logika (Emitter-Coupled-Logic)
I^2L (IIL)	integrované injekční obvody (Integrated-Injection-Logic)

Tabulka 2: Rodiny unipolárních číslicových obvodů	
zkratka	název
NMOS	MOS s N-kanálem (N-Channel-Metal-Oxid-Semiconductor)
PMOS	MOS s P-kanálem (P-Channel-Metal-Oxid-Semiconductor)
CMOS	komplementární symetrický MOS (Complementary-Metal-Oxid--Semiconductor)
HCMOS	vysokorychlostní CMOS (High-Speed-CMOS)
HCTMOS	HCMOS kompatibilní s TTL
CCD	nábojově vázané prvky (Charge-Coupled-Device)

Tabulka 3: Typické parametry důležitých rodin číslicových integrovaných obvodů				
zkratka	napájecí napětí ve V	spotřeba na prvek v mW	zpoždění signálu v ns	spínací kmitočet v MHz
TTL	5	10	10	20
H-TTL	5	22	6	30
L-TTL	5	1	33	3
AS-TTL	5	15	1,7	150
LS-TTL	5	2	9,5	40
ALS-TTL	5	1	4	40
ECL	– 5,5	20 až 60	1	150
I^2L	0 až 8	0,000 001 až 0,1	10 (proměnný)	5
CMOS	3 až 15	0,001 až 0,02	25 až 60	10
HCMOS	2 až 6	< 2 MHz: 0,001 až 1 > 2 MHz: 1 až 10	8 až 10	60
HCTMOS	5	< 2 MHz: 0,001 až 1 > 2 MHz: 1 až 10	8 až 10	60

Hladiny TTL signálů

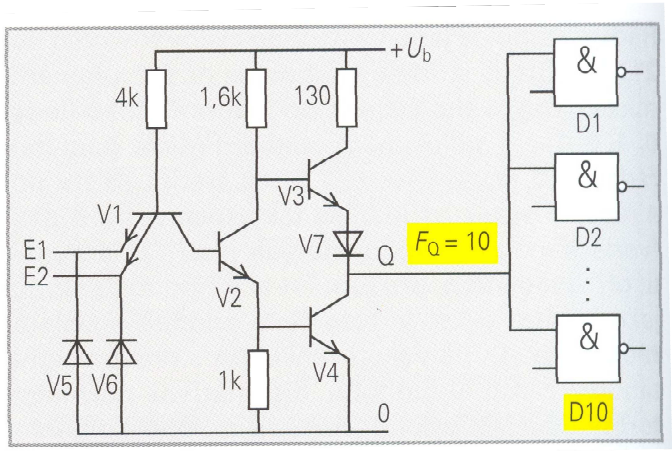


Tabulka 1: Hladiny signálů CMOS obvodů			
hladina signálu pro	U_b = 5 V	U_b = 10 V	U_b = 15 V
U_{eL}	≤ 1,5 V	≤ 3 V	≤ 4,5 V
U_{eH}	≥ 3,5 V	≥ 7 V	≥ 10,5 V
U_{aL}	≤ 0 V	≤ 0 V	≤ 0 V
U_{aH}	≥ 5 V	≥ 10 V	≥ 15 V
U napětí, indexy: a – výstup, b – napájecí, e – vstupní, H – horní úroveň (1), L – dolní úroveň (0)			

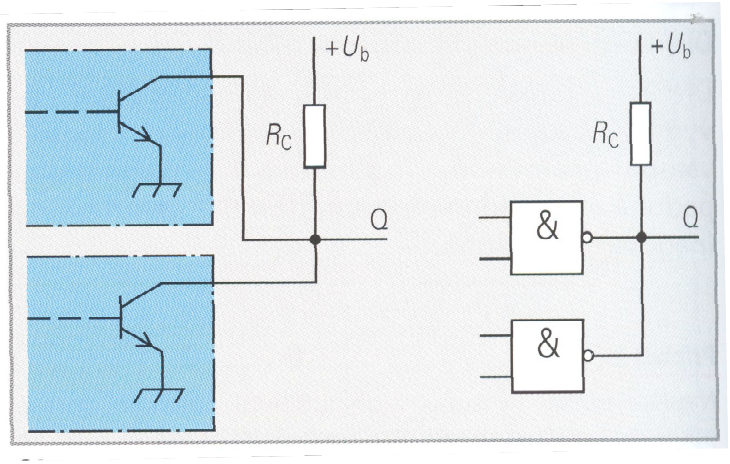
Tabulka 2: Statická odolnost proti rušení				
typ	TTL	ECL	I^2L	CMOS
S_L S_H	0,4 V až 1,2 V 0,4 V až 3,4 V	0,15 V 0,12 V	0,6 V 0,04 V	45 % napájecího napětí
S	dobrá	uspokojivá		velmi dobrá
S – statická odolnost proti rušení, Indexy: *H* – horní úroveň, *L* – dolní úroveň				

Zapojení hradla TTL

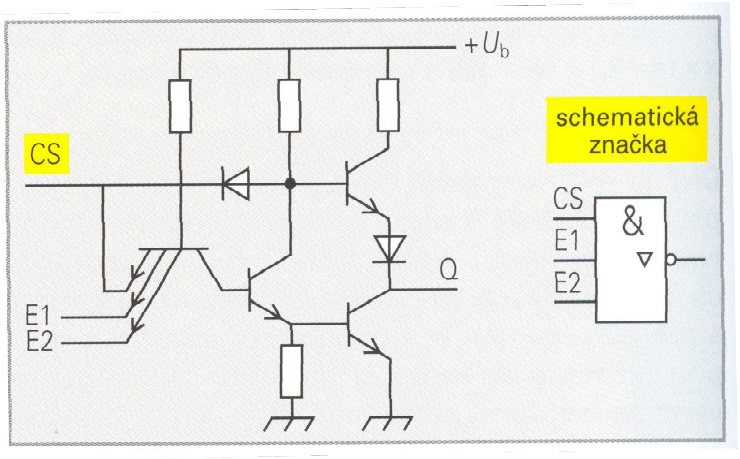
Výstup typu „TOTEM“



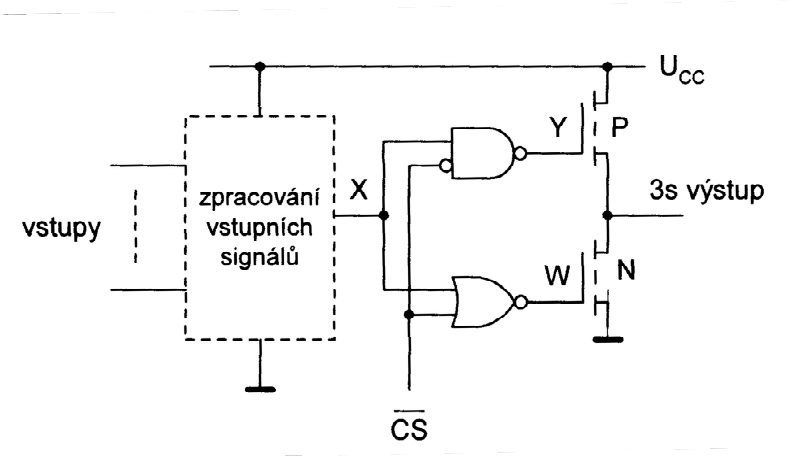
Výstup typu „otevřený kolektor“



Třístavový výstup v bipolárním provedení



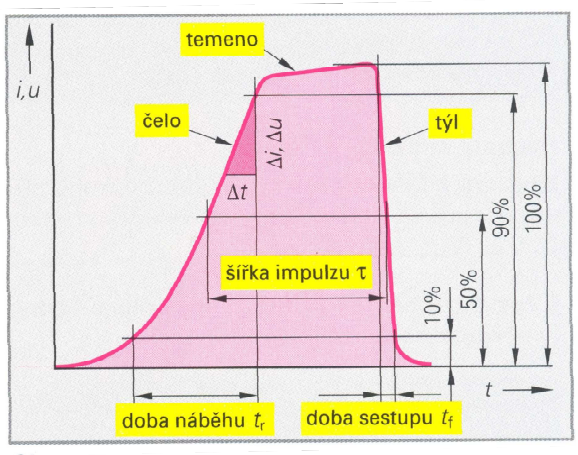
Třístavový výstup v CMOS provedení



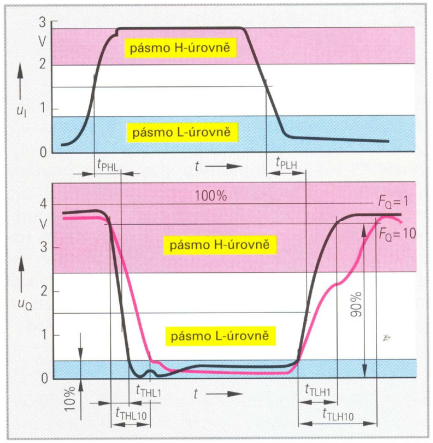
Srovnání jednotlivých typů výstupů

- **Totem**
  - Nelze výstupy spojovat paralelně (nebezpečí zničení)
  - Relativně rychlý
- **Otevřený kolektor**
  - Lze výstupy spojovat paralelně
  - Relativně pomalý (závisí na kolektorovém odporu)
- **Třístavový výstup**
  - Lze výstupy spojovat paralelně (aktivní musí být pouze **jeden** v daném časovém okamžiku
  - Velmi rychlý
  - Nejčastěji používaný pro tzv. **sběrnice**

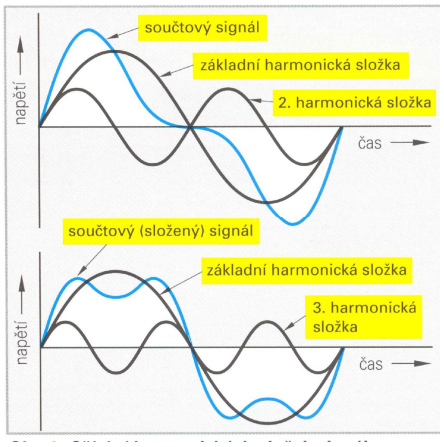
Parametry impulsu



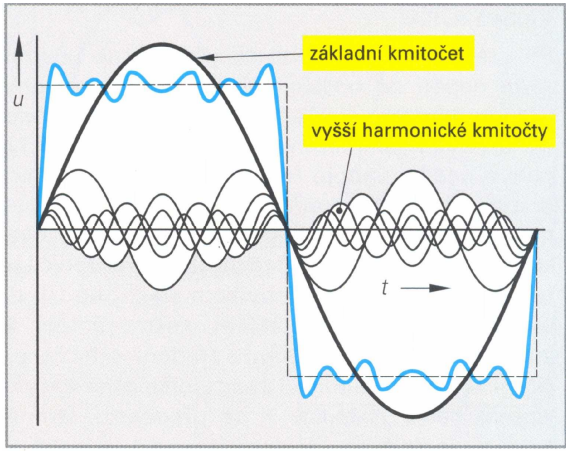
Časové průběhy vstupního a výstupního signálu logického IO



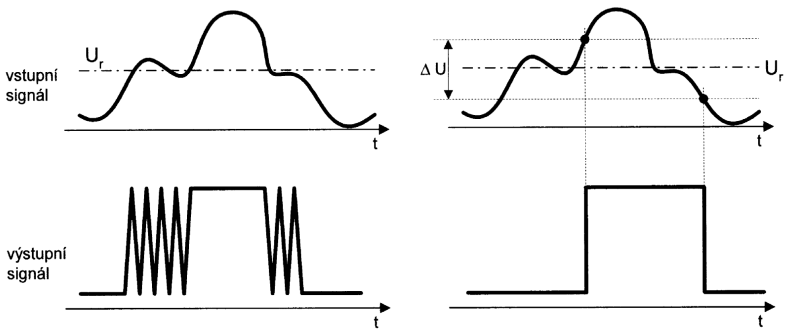
Rozklad signálu na harmonické složky



Rozklad signálu na harmonické složky



Nestabilita obvodu použití hystereze



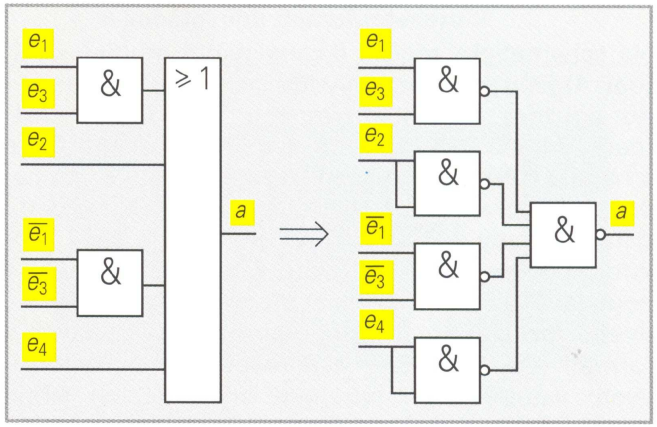
Rozdělení číslicových obvodů

- Kombinační obvody
  - výstup závisí pouze na současném stavu vstupů např. dekodér
- Sekvenční obvody
  - Výstup závisí na okamžitém stavu vstupů a předcházející historii vstupů (nutná paměť pro pamatování historie) – např. čítač

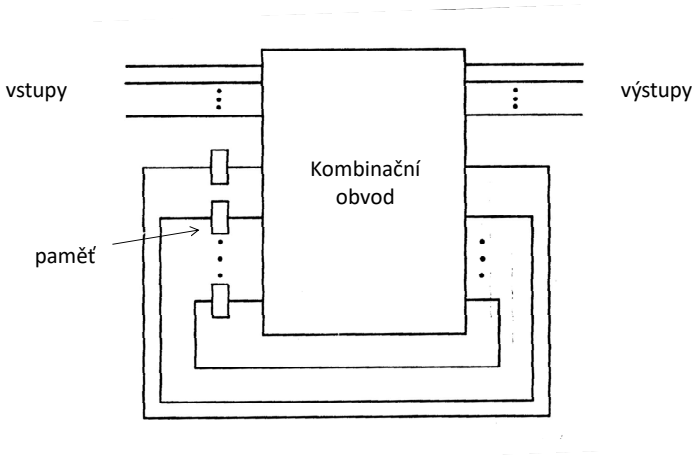
Rozdělení číslicových obvodů

- Asynchronní
  - Vstupy a výstupy nejsou časově svázané
- Synchronní
  - Vstupy a výstupy jsou časově svázané – zavedení signálu hodin – CLK
  - Možnost odstranění tzv. hazardních impulsů

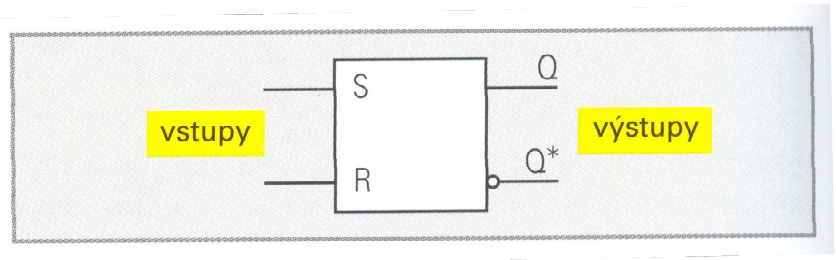
Kombinační obvod



Sekvenční obvod

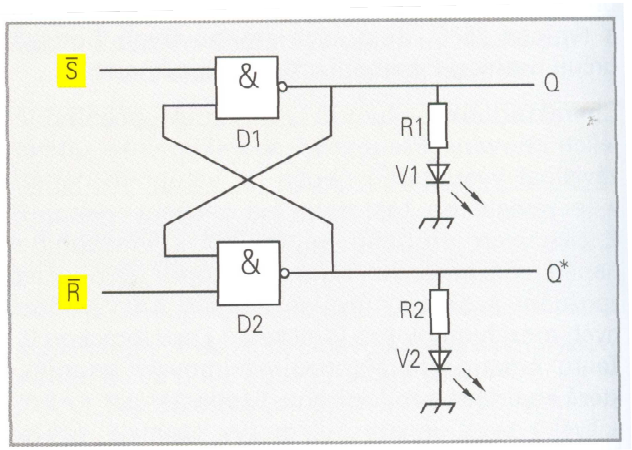


Sekvenční obvod asynchronní



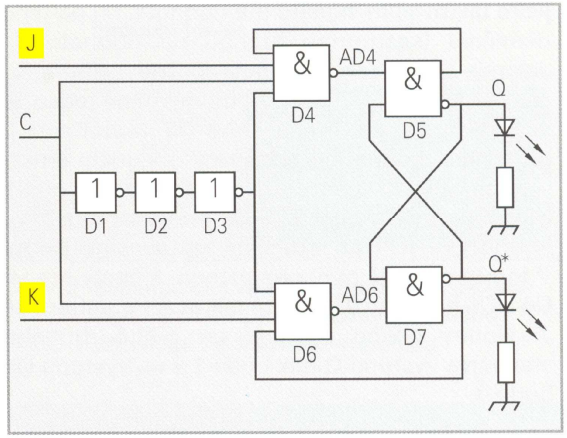
Klopný obvod RS

Sekvenční obvod asynchronní



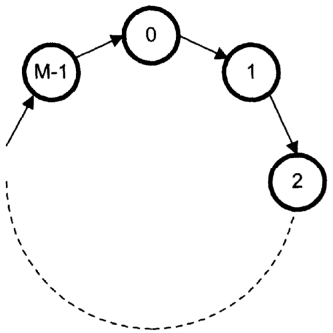
Realizace klopného obvodu RS z hradel NAND

Sekvenční obvod synchronní

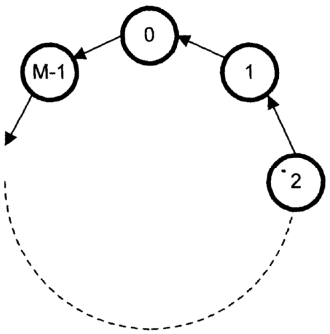


JK klopný obvod řízený hodinami

Čítače – stavový diagram

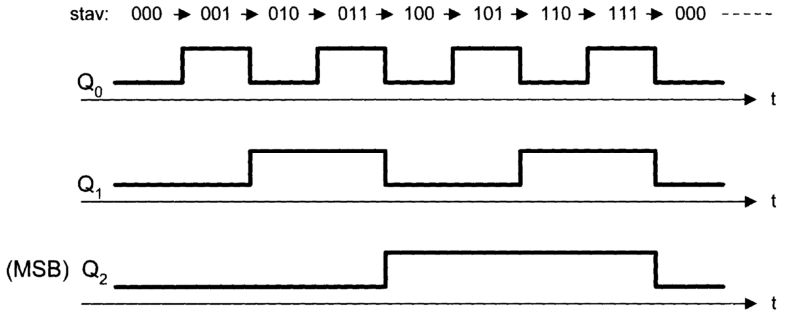


Čítač čítající nahoru



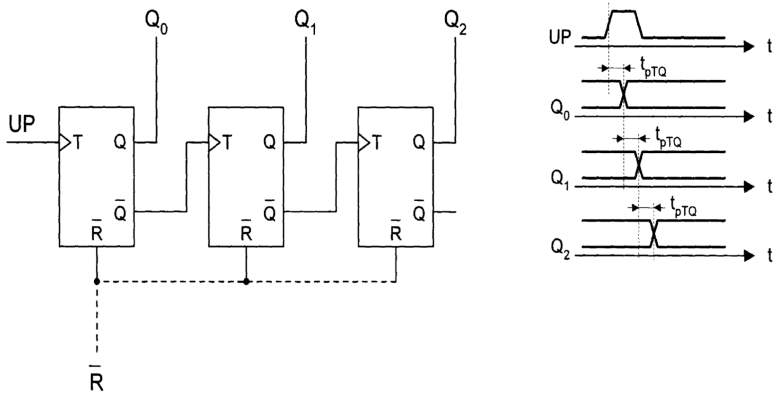
Čítač čítající dolů

Průběh signálů binárního čítače



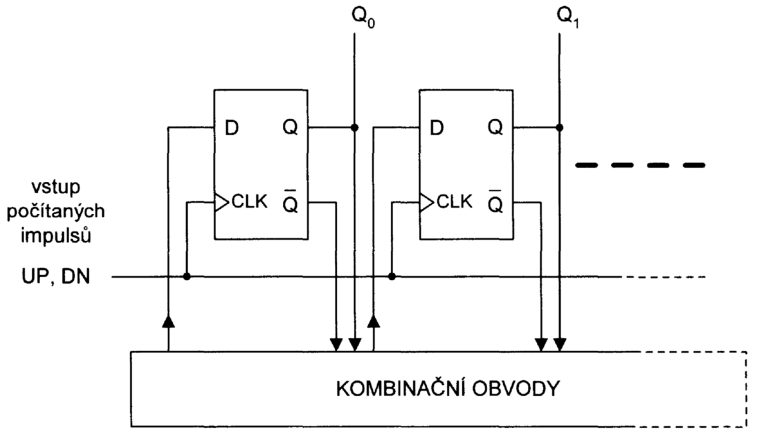
3-bitový čítač čítající nahoru

Zapojení asynchronního binárního čítače



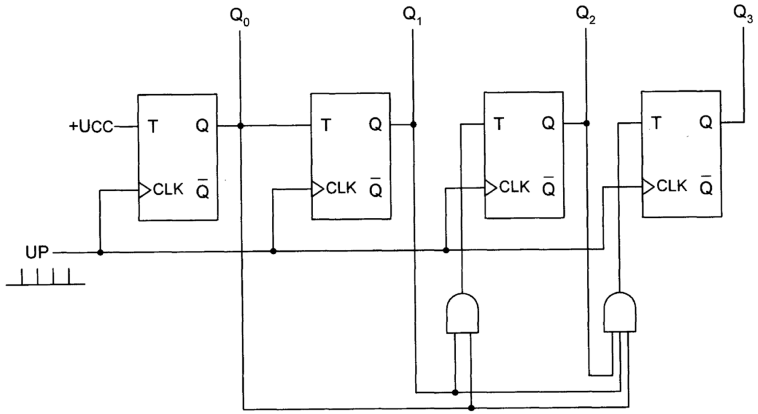
Výstup KO je zapojen do hodinového vstupu následujícího KO
KO se překlápějí postupně – ustálení stavu výstupu čítače může trvat velmi dlouho – nejhorší případ je při překlápění všech KO

Zjednodušené zapojení synchronního čítače



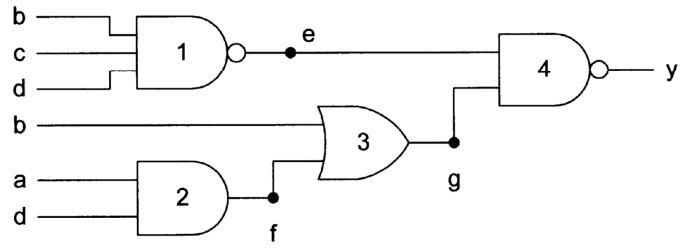
Signál hodin je veden do všech KO paralelně -> všechny KO se překlápějí současně
Doba ustálení výstupů je rovna zpoždění jednoho KO

Skutečné zapojení 4-bitového synchronního čítače

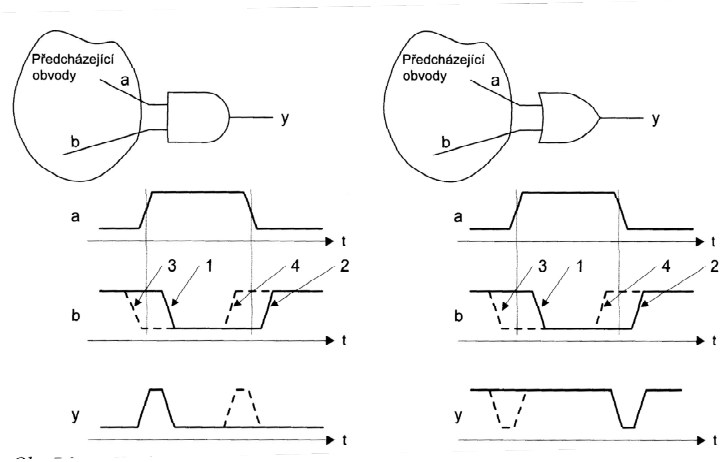


Vznik hazardních impulsů

$$y = \overline{\overline{bcd} \cdot (b + ad)} = bcd + \overline{a}\,\overline{b} + \overline{b}\,\overline{d}$$



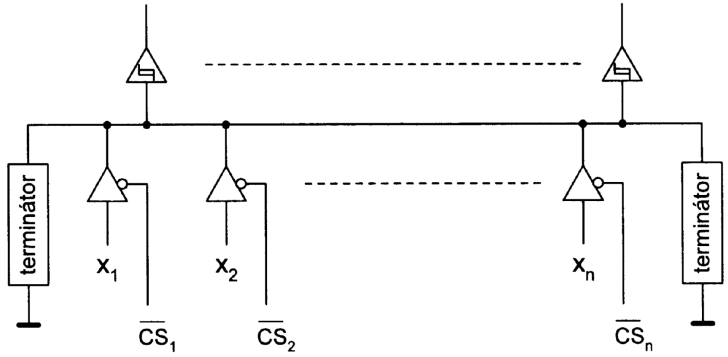
Různé zpoždění v různých větvích



Sběrnice

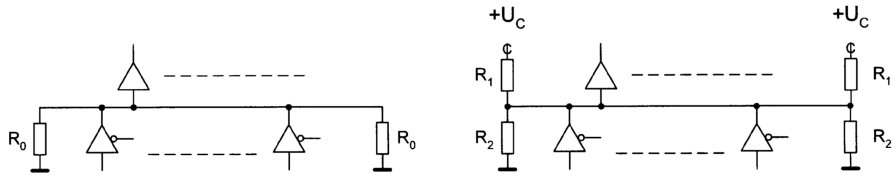
- Sběrnice je soustava jednoho či více vodičů, která slouží k přenosu informace z různých zdrojů k různým cílům
- Rozdělení sběrnic
  - **sériové** (jednovodičové) versus **paralelní** (vícevodičové)
  - **Jednosměrné** versus **obousměrné**

Princip sběrnice s třístavovými výstupy

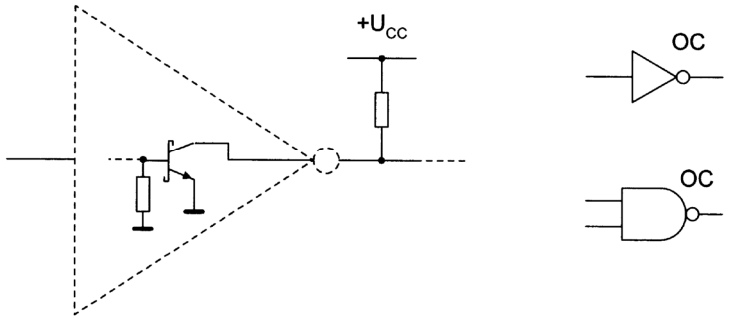


Sběrnice s třístavovými výstupy

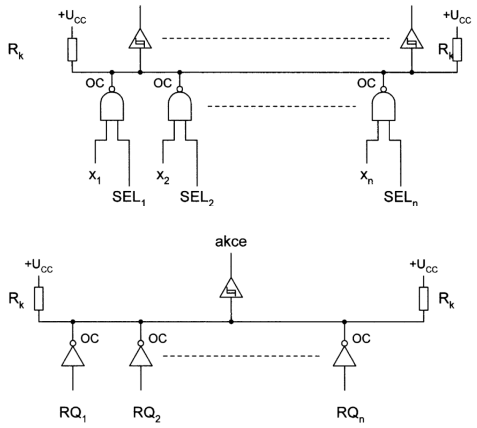
Různé typy provedení terminátorů sběrnice



Otevřený kolektor



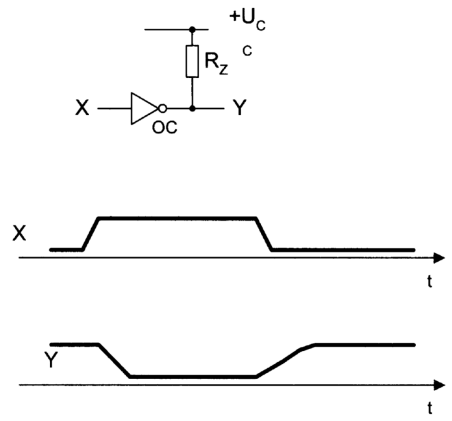
Sběrnice s otevřeným kolektorem



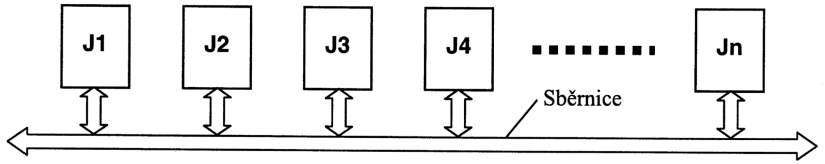
Nahoře – blokování signálem SEL – při aktivaci více budičů => zkreslení signálu
Dole – bez blokování – vnucení akce

Zpomalení hrany impulsu

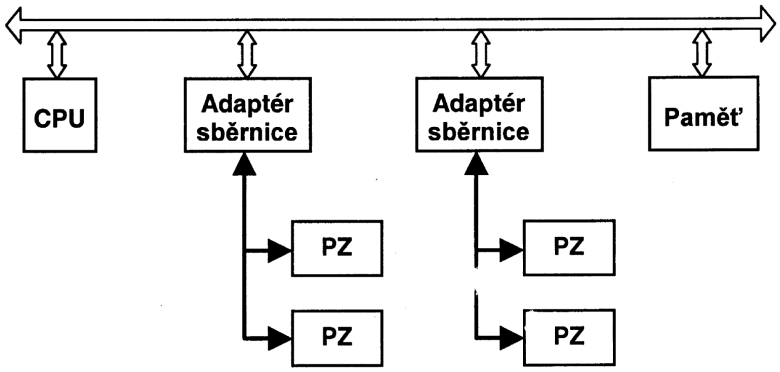
Nestejnoměrné



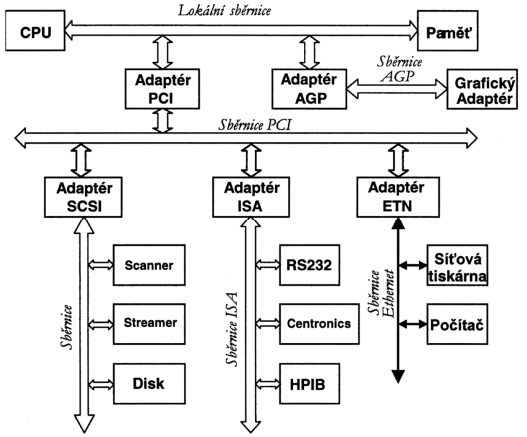
Typické použití sběrnice



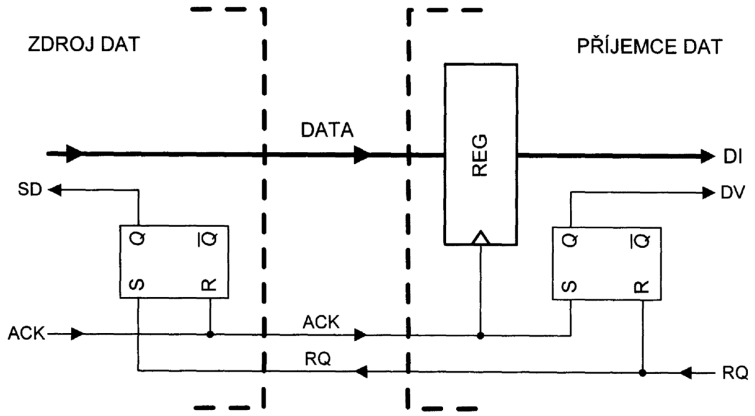
Sběrnice v jednoduchém počítači



Víceúrovňové sběrnice v počítači typu PC

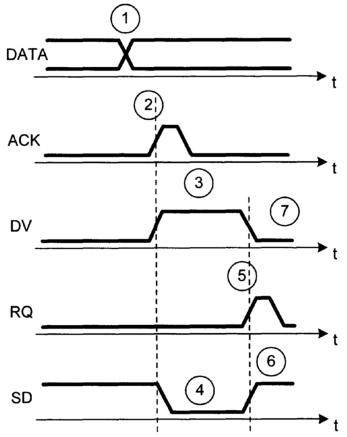


Korespondenční provoz - Handshake



Časové průběhy korespondenčního provozu

- Kromě datových signálů jsou ještě 2 signály pro řízení vzájemné spolupráce
- Zdroj umísti data na datovou sběrnici
- Zdroj potvrdí platnost dat signálem ACK - Acknowledge
- Příjemce dat přebere data
- Příjemce dat vyžaduje nová data signálem RQ - Request



Základní pojmy výpočetní techniky

- **Bit** – **Bi**nary digi**t** – základní jednotka informace nabývající hodnoty 0 nebo 1
- **Slabika (Byte)** – uspořádaná množina 8 bitů
- **Slovo (Word)** – uspořádaná množina bitů – počet bitů není ustálen – 12, 13, 16, 24, 32, 64 bitů
- **Půlslabika (nibble)** – 4 bity
- **Kilobyte, Megabyte, Gigabyte** (kB,MB,GB) obyčejně odvozeny od mocnin 2 (kilo = 2^{10} = 1024, Mega = 2^{20}, atd. Někdy jsou ve významu klasických předpon kilo = 1000, atd
- Pokus zavést jednotky **KiB, MiB, GiB** – mocniny 2 (moc se zatím neujalo)
- **LSB, MSB** - Least Significant Bit (Byte), Most Significant Bit (byte) – nejméně nejvíce významný bit (slabika)

Paměti – rozdělení

Podle způsobu adresace:

- Paměť s adresovým výběrem — data jsou uložena na stabilních adresách a dodání adresy je nutné pro operaci čtení i zápisu.

- Paměti bezadresové
  - zásobníková paměť LIFO (angl. Last In —First Out),
  - paměť fronty FIFO (angl. First In — First Out).

  Adresy jsou generovány automaticky vnitřními obvody v pevně stanoveném pořadí, buď v opačném než zápis (LIFO), nebo ve stejném jako zápis (FIFO). Proto vnější adresa neexistuje.

- Paměti adresované obsahem (CAM – Content Addressable Memory) –umístění je určeno daty (obsahem)

Paměti – rozdělení
Podle typu přístupu

- **RAM** – **R**andom **A**ccess **M**emory – čtení z libovolné adresy v kterýkoli okamžik
- **SAM** – **S**erial **A**ccess **M**emory – vykazují závislost na umístění dat v paměti. Data se čtou postupně až se načte požadovaný obsah
- **CAM** – **C**ontent **A**ddressable **M**emory – paměť adresovaná obsahem (speciální případ)

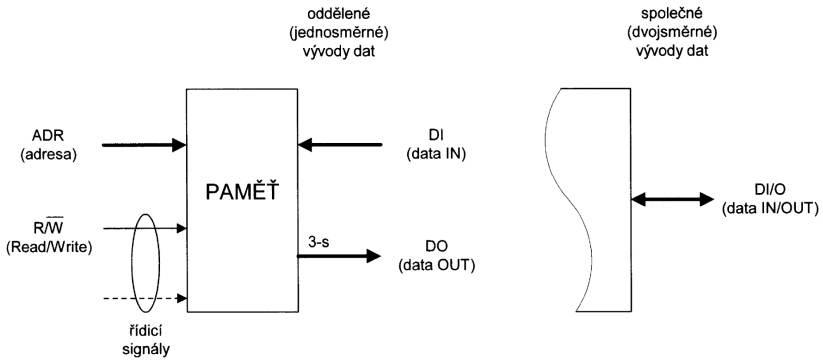
Paměti – rozdělení
Podle trvanlivosti dat po vypnutí napájení

- Paměť‘ energeticky závislá (angl. volatile) — data se ztrácí po vypnutí napájení.
- Paměť energeticky nezávislá (angl. non-volatile) — data se neztrácí po vypnutí napájení. Toto hledisko se vztahuje na samotné paměťové obvody. Celý paměťový systém totiž lze vybavit záložní baterií a i paměti energeticky závislé pak neztrácejí data.

Paměti – rozdělení
Podle snadnosti a rychlosti zápisu

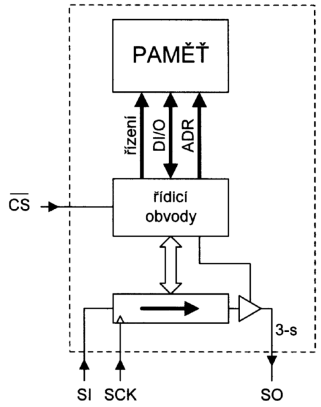
- Paměť s možností čtení i zápisu — RWM (angl. Read-Write Memory) — zápis i čtení jsou stejně snadné a rychlé. Paměť RWM je též (dokonce častěji) označována zkratkou RAM (z angl. Random Access Memory), což doslova znamená „paměť s libovolným přístupem“. Tato zkratka je významově dosti posunutá, avšak všeobecně se ujala.
- Paměti permanentní — ROM (angl. Read Only Memory) — původně jen pro čtení. Postupně s rozvojem technologie byly vyvinuty typy, umožňující více nebo méně snadný zápis, vždy však podstatně pomalejší než čtení. Paměť je energeticky nezávislá. (PROM, EPROM, EEPROM)

Podle datového rozhraní
Paralelní přístup

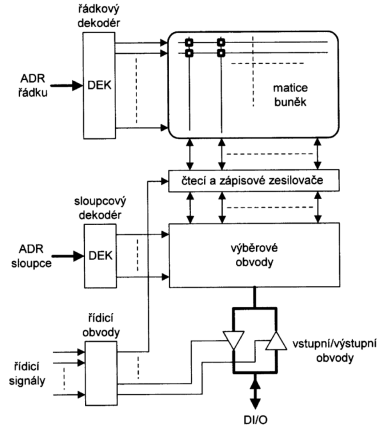


Podle datového rozhraní

Sériový přístup

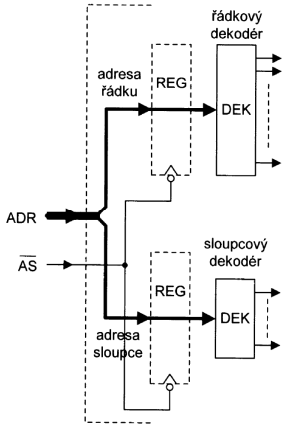


Paměti s adresovým výběrem

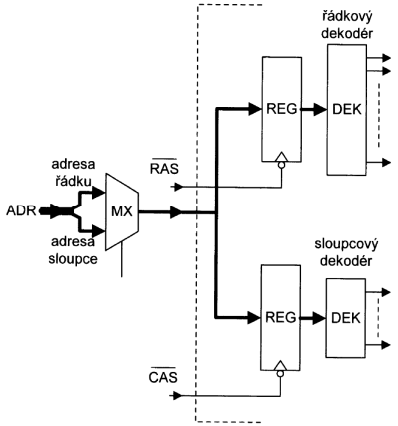


Paměti s adresovým výběrem

Zavedení adresy

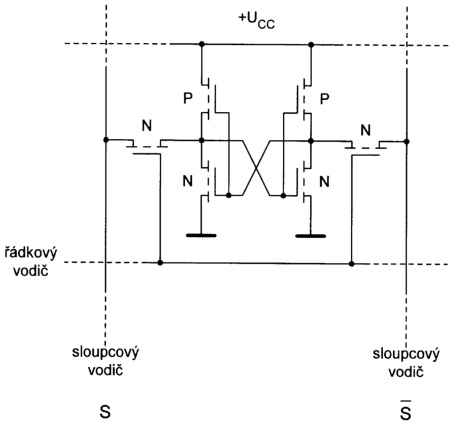


Celá adresa najednou



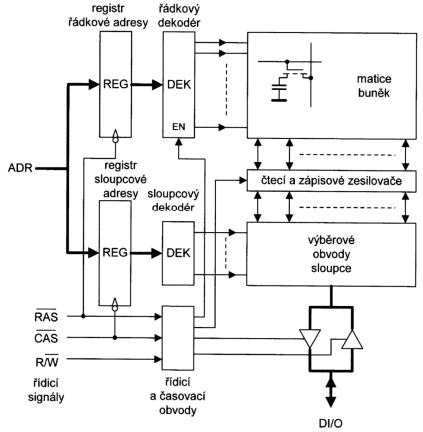
Adresa postupně – tzv. multiplexování

Statická paměť RAM



Není potřeba pravidelně obnovovat informaci

Dynamická paměť RAM

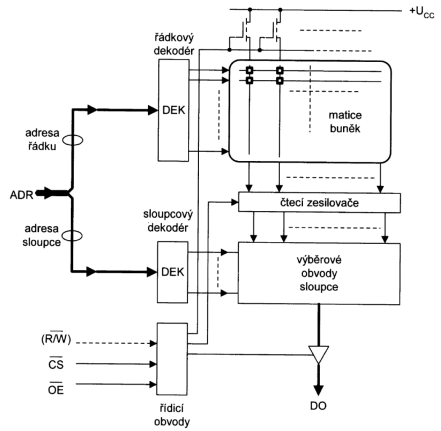


Informaci je nutné pravidelně obnovovat – tzv. refreshing

Permanentní paměti

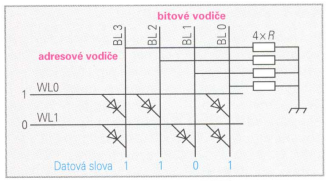
- Paměť **ROM** (ang. Read-Only Memory) — obsah je dán při výrobě a nelze jej již změnit.
- Paměť programovatelná **PROM** (angl. Programmable ROM) — obsah může uživatel zapsat (naprogramovat), ale naprogramování je nevratné.
- Paměť **EPROM** (ang. Erasable PROM) — paměť programovatelná a vymazatelná. Je programována elektricky. Před programováním musí být vymazána ultrafialovým zářením.
- Paměť **EEPROM** (angl. Electrically Erasable PROM) — paměť programovatelná, programována elektricky na libovolné adrese bez předchozího vymazání. Spíše než o programování a mazání zde lze mluvit o zápisu dat na adrese, obdobně jako u paměti RAM. Zápis je však mnohem pomalejší než čtení.
- Paměť **FLASH** – paměť programovatelná a vymazatelná. Je programována elektricky na libovolné adrese, ale elektricky mazána jako celek nebo po velkých částech (blocích). Před programováním musí být vymazána.

Permanentní paměť ROM

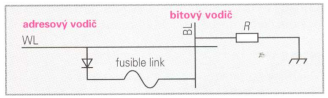


Paměť ROM, PROM, EPROM

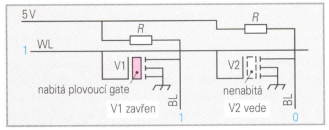
starší provedení



Obr. 1 Princip zapojení paměti ROM

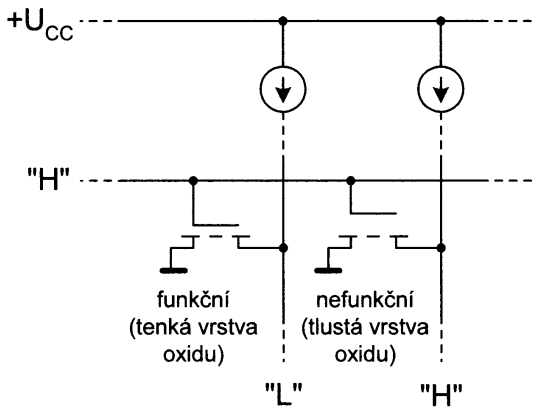


Obr. 2 Paměťový element paměti PROM



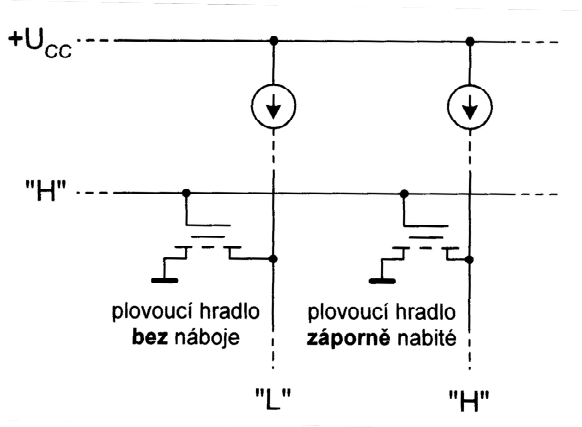
Obr. 3 Pamětový element EPROM

Paměť ROM



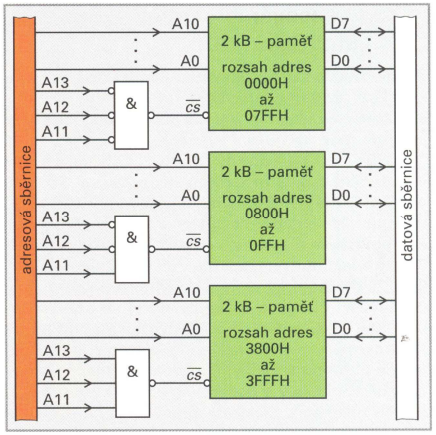
Buňky paměti ROM programovatelné maskou

Paměť PROM



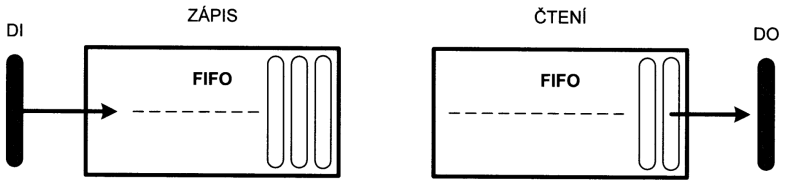
Buňky paměti (EE,E)PROM programovatelné nábojem

Zvýšení kapacity paměti složením více čipů



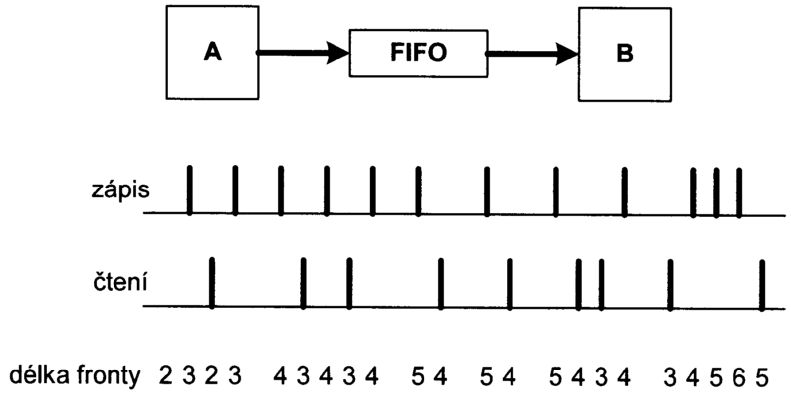
Použití adresového dekodéru

Paměť typu FIFO
First In – First Out



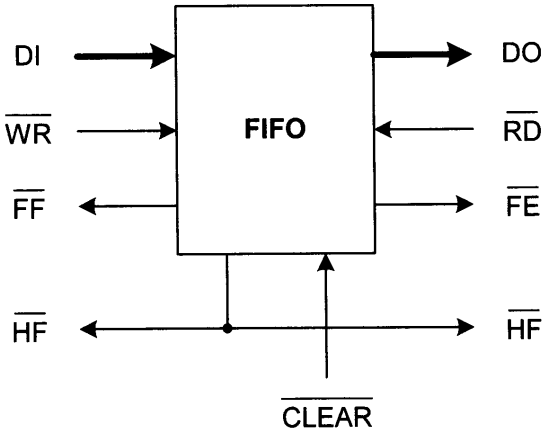
Použití – vyrovnávací fronty v procesorech či periferiích (tiskárny)

Paměť FIFO

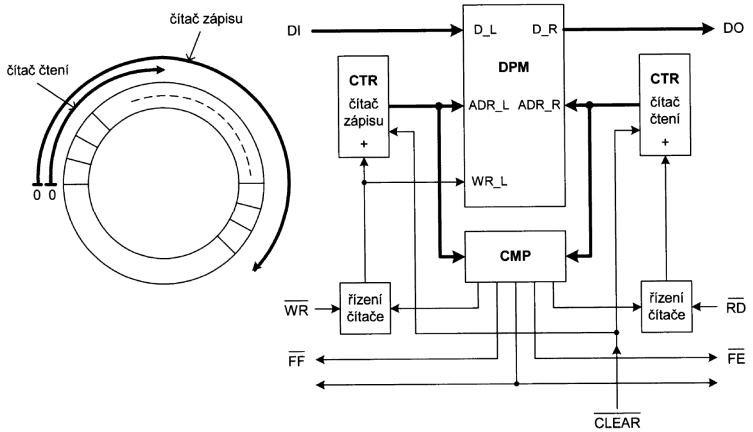


Použití jako vyrovnávací paměť např. v tiskárnách

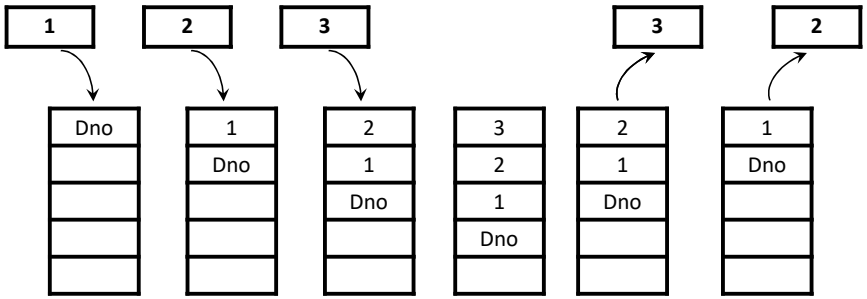
Paměť FIFO - zapojení



Paměť FIFO – emulace pomocí paměti RAM

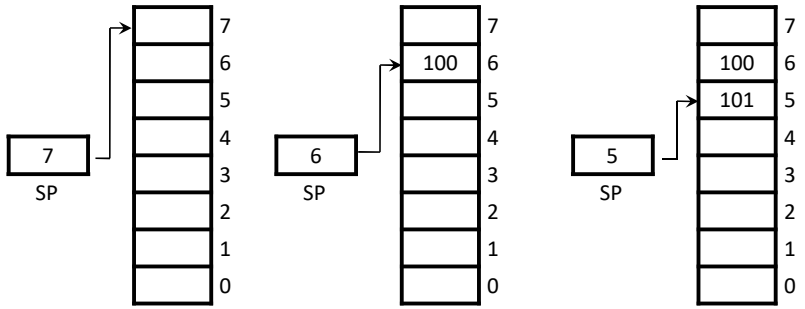


Paměť LIFO
Last In – First Out



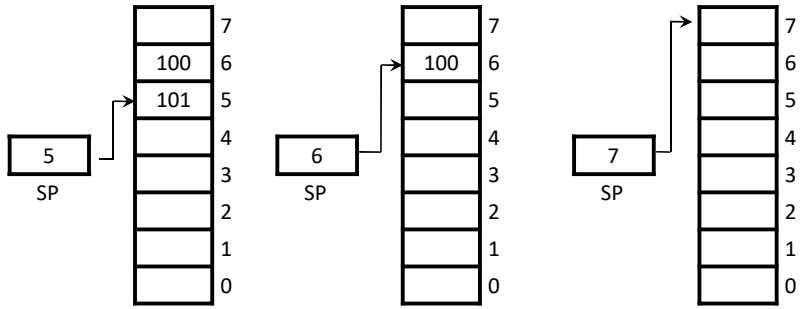
Emulace LIFO v paměti

SP - Ukazatel zásobníku (Stack Pointer)



1. zápis 100 2. zápis 101

Emulace LIFO v paměti



1. čtení 101 2. čtení 100

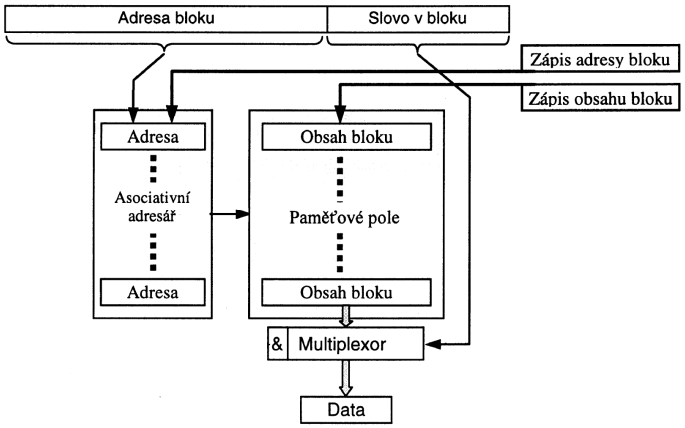
Použití paměti LIFO v počítačích

- Realizace tzv. **zásobníku (Stack)**
- Zásobník se používá pro:
  - uložení návratové adresy při volání podprogramu
  - Předávání parametrů do podprogramu ve vyšších programovacích jazycích
  - Dočasnou úschovu dat – výhoda je bezadresnost paměti LIFO (není si třeba pamatovat adresu, kde jsou data uschována)

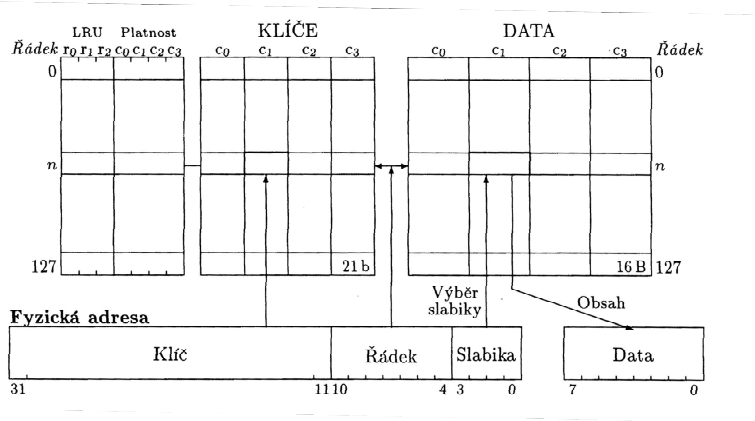
Paměť adresovaná obsahem

- Obsahuje tzv. asociativní paměť – paměť klíčů a datovou paměť – paměť dat
- Dělení na:
  - Plně asociativní paměť – prohledává se celá paměť klíčů na výskyt daného obsahu – technologicky velmi náročné
  - Částečně asociativní paměť – x-cestná asociativní paměť – prohledává se pouze určitá část paměti klíčů na výskyt daného obsahu – lépe realizovatelná (nejčastější použití)
- Použití: vyrovnávací paměti typu cache, převodní tabulky adres (procesory 80x86), tabulky pro předpovídání skoků (procesory typu RISC), …

Plně asociativní paměť

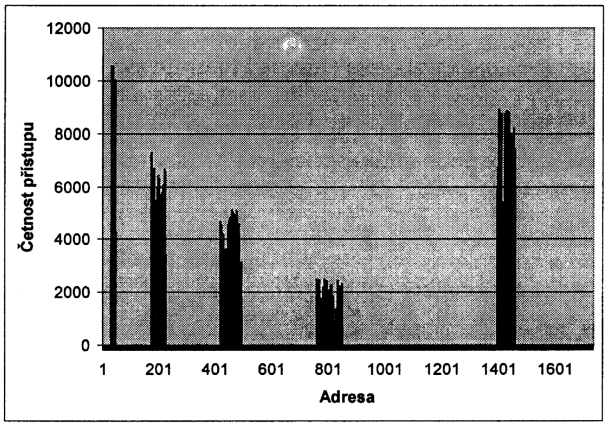


4-cestná asociativní paměť



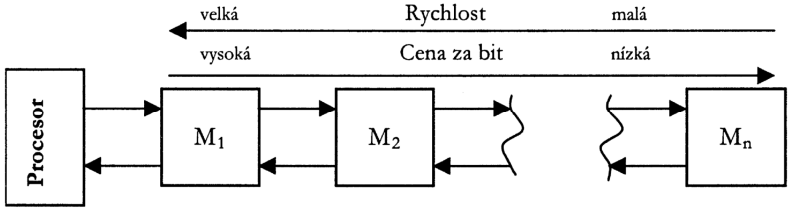
Vyrovnávací paměť cache procesoru Intel 80486

Paměť adresovaná obsahem

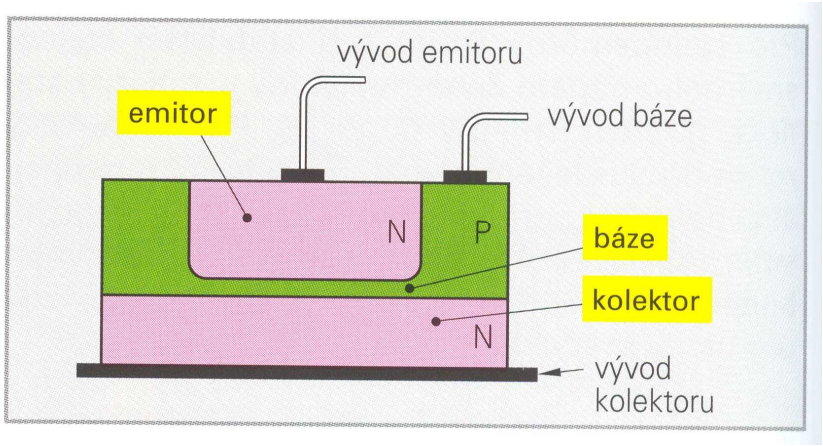


Histogram přístupu do paměti při vykonávání programu

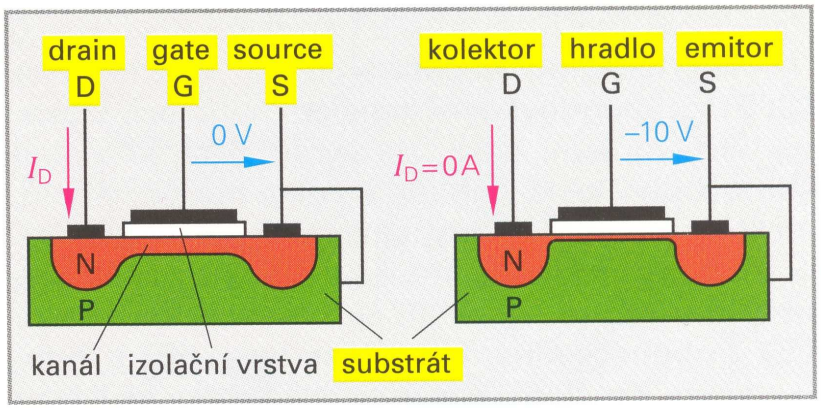
Hierarchické řazení cache pamětí



Princip výroby IO
NPN tranzistor - realizace

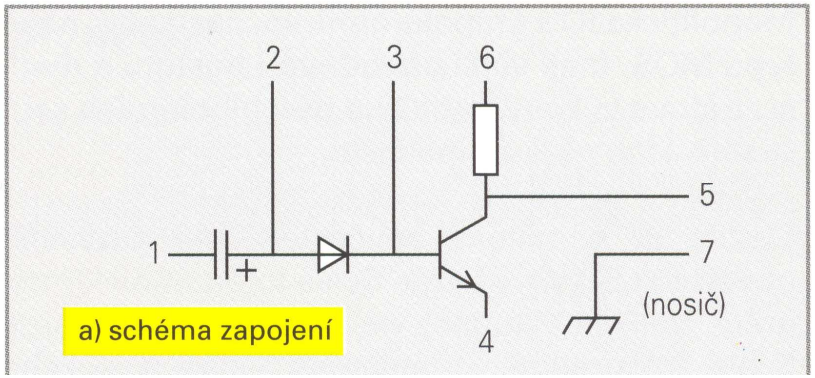


CMOS tranzistor - realizace

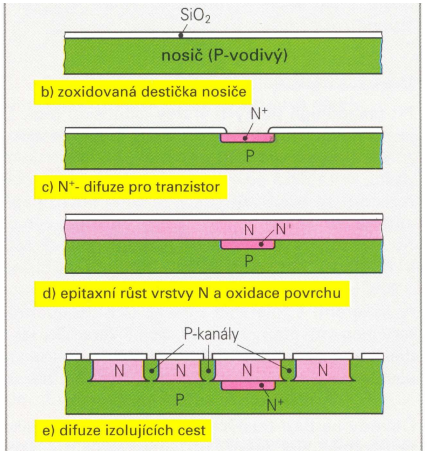


Vliv napětí přivedeného na G na šířku kanálu

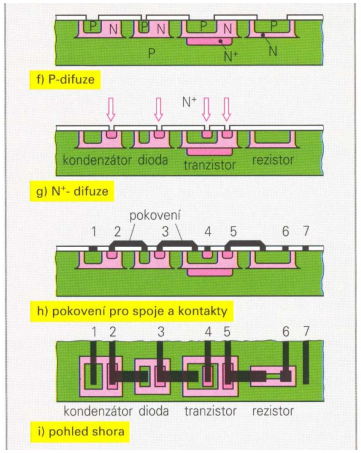
Integrovaný obvod (IO) - zapojení



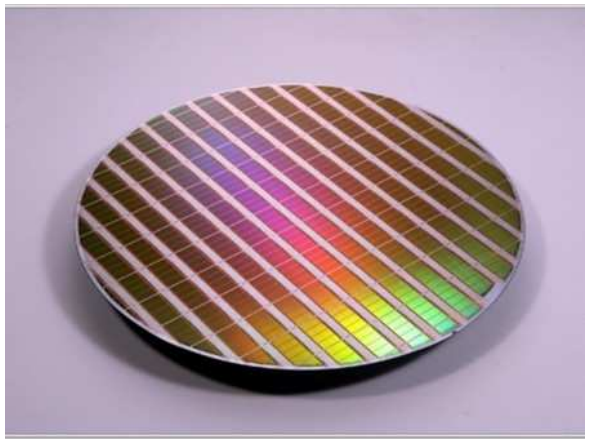
IO – výroba – I.



IO – výroba – II.



Wafer – polovodičový plát

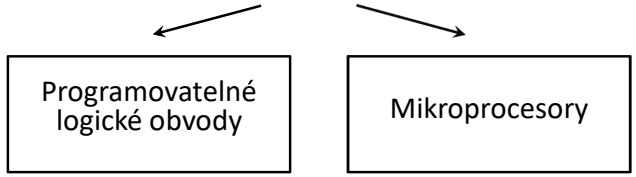


Problémy výroby IO

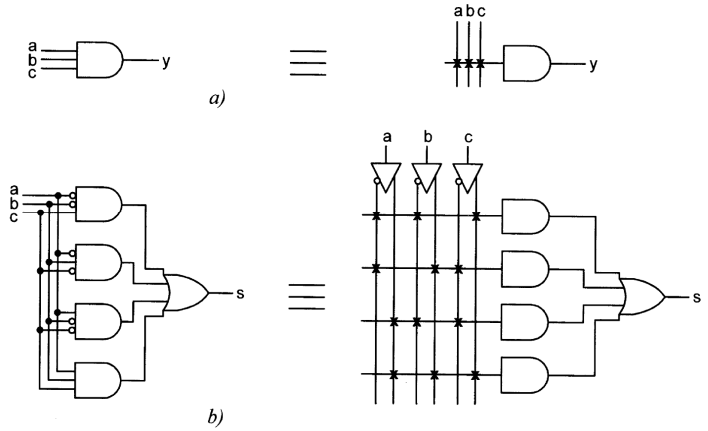
Čím složitější integrovaný obvod, tím speciálnější použití => malé výrobní série => velmi drahé

Řešení:

Vyrobit velmi složitý obvod, ale programovatelný => činnost můžeme změnit **programováním**



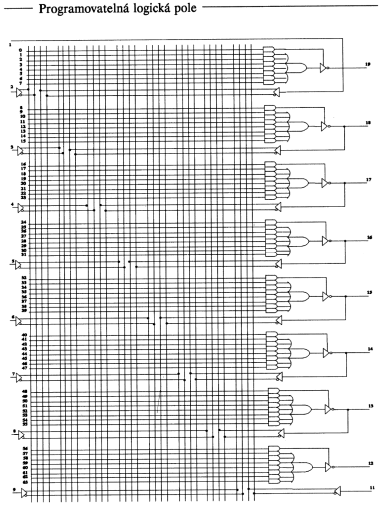
Programovatelné logické obvody PLD – Programmable Logic Devices



Naprogramování různých kombinačních obvodů

Programovatelné logické obvody

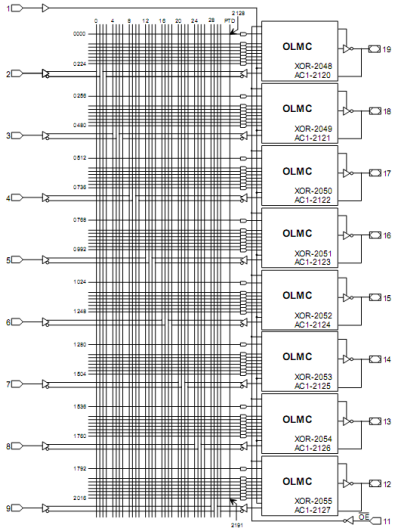
PAL – Program Array Logic



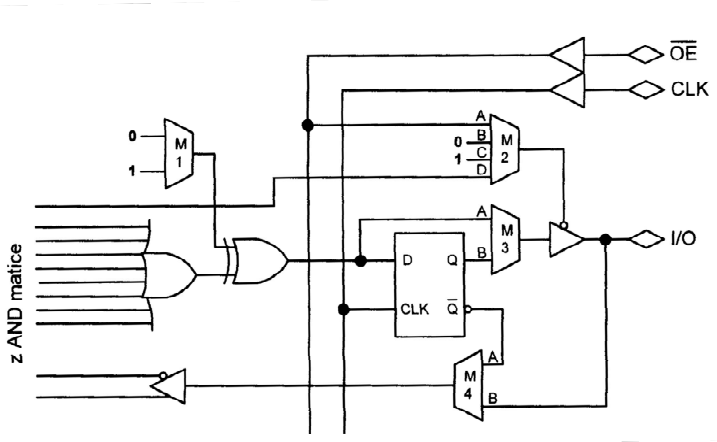
Programovatelné logické obvody

Vnitřní zapojení obvodu GAL16V8

GAL – Generic Array Logic



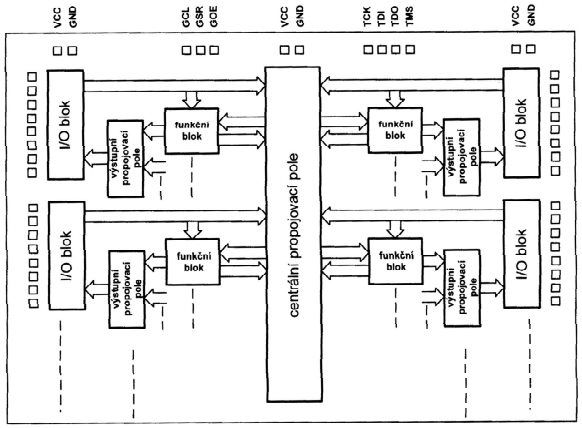
Programovatelné logické obvody



Výstupní část obvodu a makrobuňka GAL fy Lattice (OLMC Output Logic Macrocell)

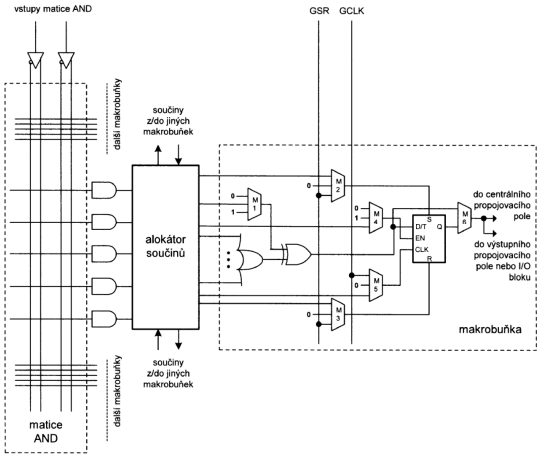
Programovatelné logické obvody

CPLD – Complex Programmable Logic Devices



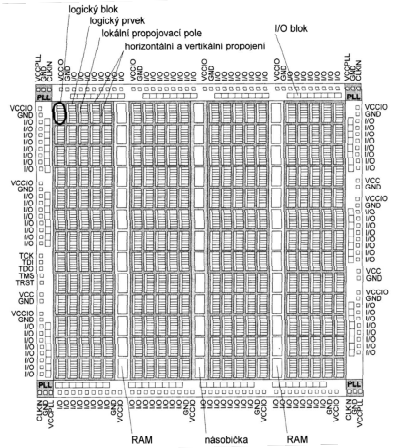
Vnitřní struktura obvodu CPLD fy Lattice

Programovatelné logické obvody



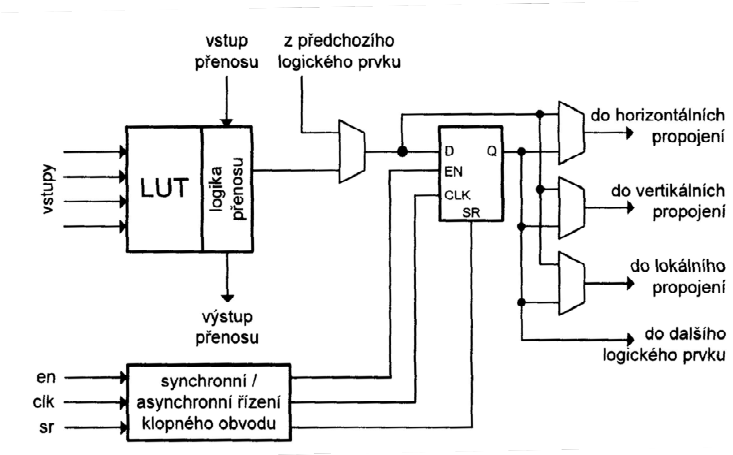
Makrobuňka ve funkčním bloku CPLD

Programovatelné logické obvody
FPGA - Field-Programmable Gate Array



Vnitřní struktura obvodu FPGA fy Lattice

Programovatelné logické obvody



Vnitřní struktura logického prvku FPGA

Programování PLD

- Speciální programovací jazyky HDL (Hardware Description Languege)
  - Jazyk ABEL (Advanced Boolean Expression Language)
  - VHDL (VHSIC */Very High Speed Integrated Circuits/* hardware description language)
  - Verilog – HDL jazyk, také pro verifikaci digitálních či analogových obvodů
  - …..
- Speciální programové systémy pro simulaci chování obvodu – zobrazuje časové průběhy, maximální pracovní frekvenci, atd.

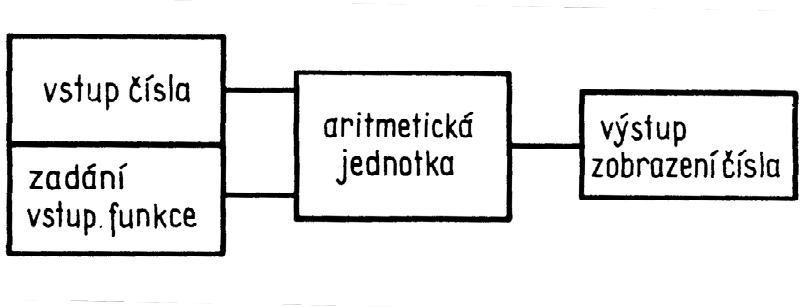
Ukázka v jazyku ABEL

```
{ .OPL FILE FOR UDCOUNT }

begin header
  Navrh popisuje 2bitovy citac nahoru/dolu s resetem s pouzitim CASE prikazu.
  Navrh je pro GAL16V8.
end header

begin definition
  device          G16V8 ;
  inputs          clk=1,up=2,down=3,reset=4;
  statebits       sb1=15,sb0=16;
  state_names     zero=0,one=1,two=2,three=3;
  output ( com )  even=12,odd=13;
  set             cnt=[sb1,sb0];
end definition

begin equation
 even = /sb0;
 odd  = sb0;
 UP   = up * /down ;
 DN   = /up * down ;
 HLD  = up !$ down ;  {Když up a down jsou 0 nebo 1, citac necita}
end equation


begin state_diagram (sb1,sb0)   { Levy bit je nejvyznamnejsi. }

state all : if reset then zero ;

state zero  : case UP   : one ;
                DN   : three ;
                HLD  : zero ;
              endcase ;

state one   : case UP   : two ;
                DN   : zero ;
                HLD  : one ;
              endcase ;

state two   : case UP   : three ;
                DN   : one ;
                HLD  : two ;
              endcase ;

state three : case UP   : zero ;
                DN   : two ;
                HLD  : three ;
              endcase ;

end state_diagram
```

Programovatelné logické obvody

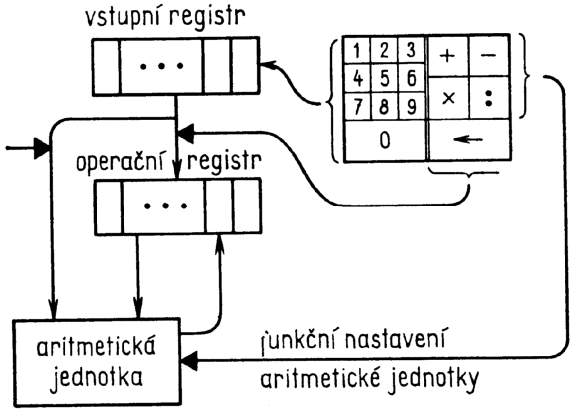
- Výhody:
  - Velmi rychlé (max. frekvence >300 MHz)
  - Programovatelné – jednoduchá změna vnitřního zapojení přes universální programovací rozhraní
  - Relativně levné
  - Všechny výhody pevné logiky s jednoduchou možností změny zapojení (přeprogramováním)

Blokové schéma kalkulátoru



Vnitřní zapojení kalkulátoru

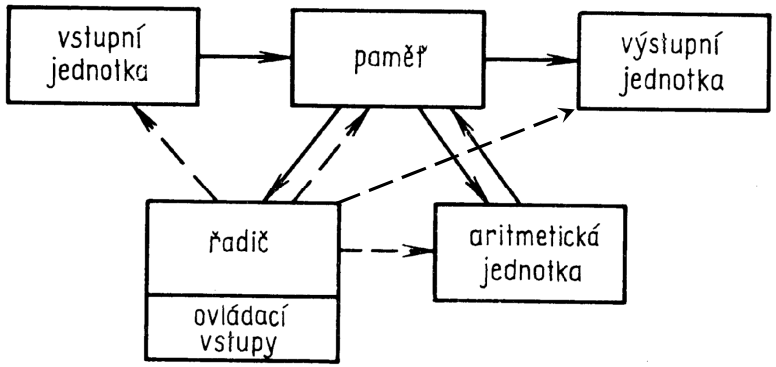
(zjednodušené)



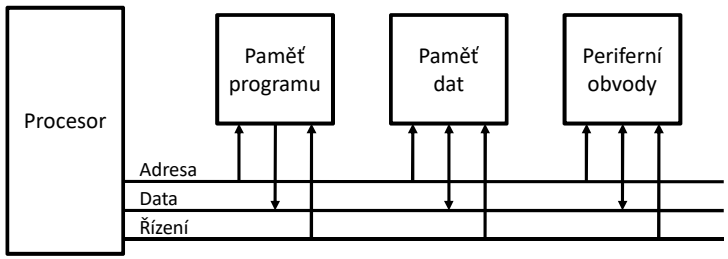
Rozdíl mezi kalkulátorem a počítačem

- Musí se uložit do paměti posloupnost operandů a operací – tzv. **program**
  - Musí se přiřadit k určité operaci binární číslo, které bude operaci identifikovat – tzv. **operační znak**
  - Operace s přiřazeným operačním znakem a údaji o operandech = instrukce
- Musí existovat v počítači speciální obvod – **řadič**, který automaticky řídí činnost počítače dle programu

Blokové schéma počítače

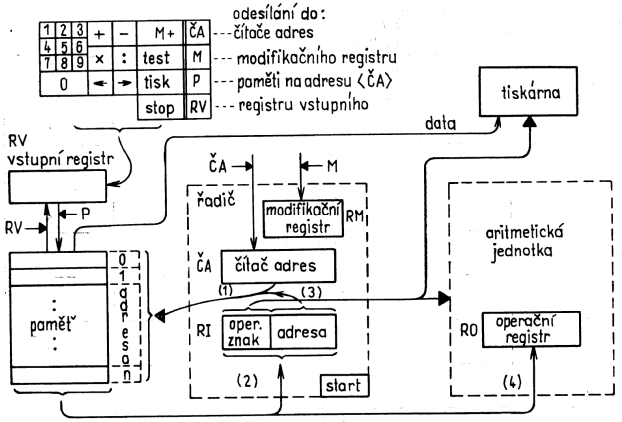


Zjednodušené schéma počítače



Vnitřní zapojení počítače

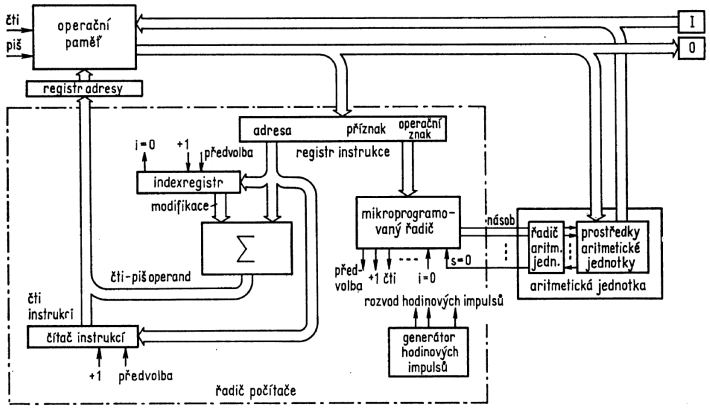
(zjednodušené)



Činnost řadiče při vykonávání příkazů uložených v paměti

1. Vyhledání instrukce na adrese uvedené v čítači adres ČA
2. Přepis této instrukce do registru instrukcí RI
3. Výběr čísla (operandu) z paměťové buňky, jejíž adresa je v adresové části registru instrukcí – je-li operační znak s příznakem M, operand se hledá na adrese, která vznikne přičtením obsahu modifikačního registru k adresové části registru instrukcí
4. Zápis tohoto operandu do registru aritmetické jednotky a provedení operace určené operačním znakem v registru instrukcí
5. Přičtení jedničky k čítači adres (připraví se na zpracování další instrukce)

Vnitřní struktura řadiče počítače



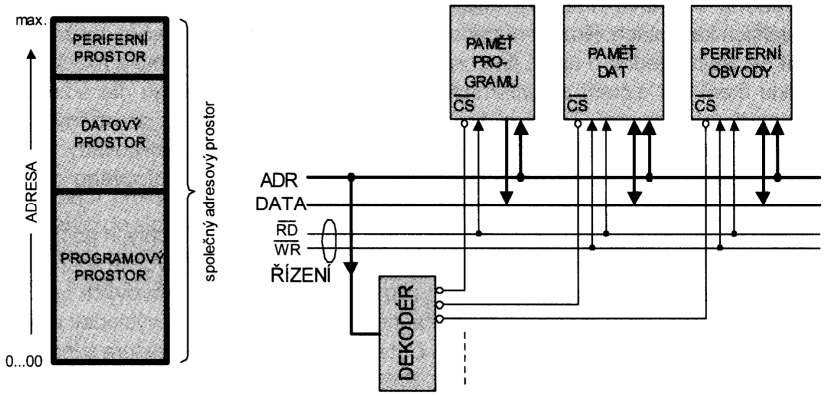
Harvardská koncepce

- Program a data jsou odděleny v samostatných adresových prostorech
- Výhoda – šířka paměti pro instrukce a pro data může být rozdílná

Koncepce John von Neumanna

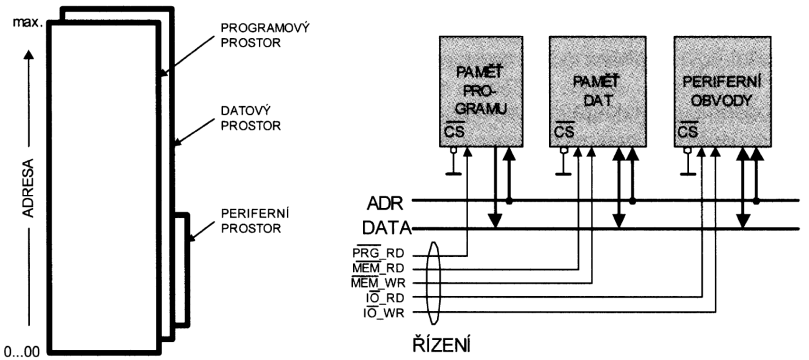
- Počítač je tvořen řadičem, aritmeticko-logickou jednotkou (ALU), pamětí, blokem vstupů a blokem výstupů
- Struktura počítače je neměnná a jeho práce je řízena programem uloženým v paměti.
- V paměti jsou uložena data společně s instrukcemi programu.
- Paměť je tvořena jednotlivými paměťovými místy, do kterých se dá zapisovat a ze kterých je možno uloženou informaci přečíst po zadání adresy tohoto místa. Adresa místa je dána pořadovým číslem tohoto místa.
- Program je tvořen posloupnosti instrukcí, které určují elementární změnu stavu počítače.
- Pořadí provádění instrukcí programu je, až na výjimky, sekvenční. Výjimkou jsou instrukce skoku.
- Instrukce, data a adresy jsou kódovány binárně.

Společný adresový prostor



Nutnost adresového dekodéru

Rozdílné adresové prostory



Odpadá adresový dekodér pro výběr příslušné paměti či periferního adresového prostoru
(Adresový dekodér může být i v této konfiguraci pro jemnější adresování uvnitř adresového prostoru)

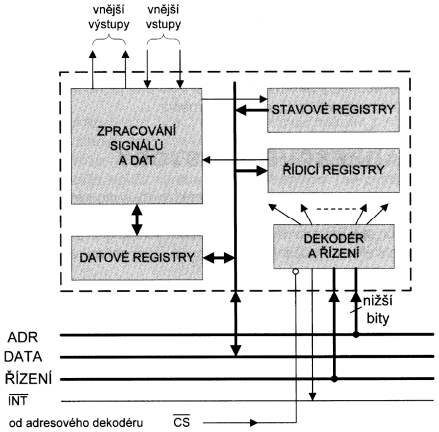
Adresové prostory

- Předcházející dva případy jsou mezní situace, v praxi se používají různé kombinace, např.:
  - 2 adresové prostory – paměťový (společný pro instrukce i data) a vstupně-výstupní adresový prostor (adresa v tomto prostoru se nazývá port) – použití např. u procesorů I8080, Z80, 80x86 (von Neumannova architektura)
  - 2 adresové prostory – paměťový pro instrukce (program) a společný pro data a periferie – např. u procesorů Microchip PICxxxx – harvardská architektura

Periferní obvody

- Každá periferie se jeví z programátorského hlediska jako soustava speciálních registrů do kterých lze zapisovat a ze kterých lze číst.
- Ovládání periferie pomocí programu je zápis a čtení z těchto registrů.
- Ke každému speciálnímu registru je přidělena adresa z adresového prostoru. Pokud je to adresa z adr. prostoru V/V zařízení, místo slova adresa se používá pojem port.
- V případě umístění V/V zařízení (registrů) do paměťového adresového prostoru, mluvíme o tzv. metodě MMIO (Memory Mapped Input Output) – výhoda: možnost použití všech instrukcí pro práci s pamětí (různé adresovací módy => efektivní kód

Periferní obvody



Definice pojmů

- **Takt** (Time-Period) - Procesory počítačů pracují téměř výhradně synchronně, tj. jejich činnost je řízena tzv. hodinovým signálem, který synchronizuje přenos a zpracování signálů mezi jednotlivými logickými členy procesoru, a tím i změny jejich stavů. Hodinový signál je buď jednofázový (u současných procesorů), nebo je vícefázový (u starších procesorů). Taktem nazýváme časový úsek mezi stejnolehlými body dvou po sobě následujících hodinových impulsů. V případě jednofázových hodin je takt totožný s periodou hodin.
- **Strojový cyklus** (Machine Cycle) - skupina taktů, která se označuje i názvy krok, nebo instrukční subcyklus. Je to každá skupina taktů, která začíná taktem, ve které se odesílá na adresovou sběrnici adresa, a končí před nejbližším dalším taktem. Jeden nebo několik strojových cyklů tvoří instrukční cyklus.

- **Instrukční cyklus** je skupina všech taktů nezbytných k uskutečnění jedné strojové instrukce. Instrukční cyklus se skládá z jednoho nebo několika strojových cyklů. Pro různé instrukce může být instrukční cyklus různě dlouhý.
- **Mikrooperace** je činnost procesoru, nebo častěji jeho logicky ohraničené části, během jednoho taktu. Typické mikrooperace jsou např. přesun dat mezi registry, posuv o 1 bit, inverze čísla, sečtení dvou slov atd. Jsou to stručně řečeno operace tak jednoduché, že pro účely popisu činnosti procesoru není účelné je členit na úkony ještě jednodušší.
- **Mikroinstrukce** je kódovaný příkaz pro uskutečňování mikrooperace procesorem. Činnost logicky vymezených samostatných částí procesoru je řízena dílčími skupinami signálů mikroinstrukce, tzv. poli mikroinstrukce.

- **Mikroprogram** je úplná, logicky skloubená posloupnost mikroinstrukcí. Mikroprogram může sestávat z jediné mikroinstrukce, ale může být i velmi složitý a délkou porovnatelný s obsáhlým podprogramem. Složité řadiče mají jednotlivé instrukce implementovány jako mikroprogram (tzv. mikroprogramovatelné řadiče). Některé starší procesory bylo možné mikroprogramovat, tzn. Vytvářet vlastní instrukce.
- **Strojová instrukce** –dále pouze **instrukce**, je kódovaný příkaz k vykonání strojové operace. Její mikroprogram (u mikroprogramového řadiče) je uložen v řadiči a vyvolává se instrukcí většinou přečtenou z hlavní paměti. Úplný soubor instrukcí je u každého procesoru nejnižší programovací úrovní běžně přístupnou uživateli počítače bez zásahů do řadiče. Instrukce vždy obsahuje vlastní operační znak a často i adresy nebo operandy.
- **Makroinstrukce** je označení předem definované posloupnosti instrukcí. Je to vlastně jen pokyn pro překladač, aby do cílového programu zařadil žádoucí skupinu. Použití makroinstrukcí za často používané skupiny instrukcí šetří čas programátora a činí stručnějšími zdrojové programy, avšak v žádném případě nezkracuje cílové programy.
- **Pseudoinstrukce** je příkaz ve zdrojovém programu, který se nepřekládá do cílového programu, ale ovlivňuje překlad. Na rozdíl od instrukcí nejsou pseudoinstrukce i makroinstrukce vlastní procesorům, ale překladačům.

Instrukce

Údaje v instrukci

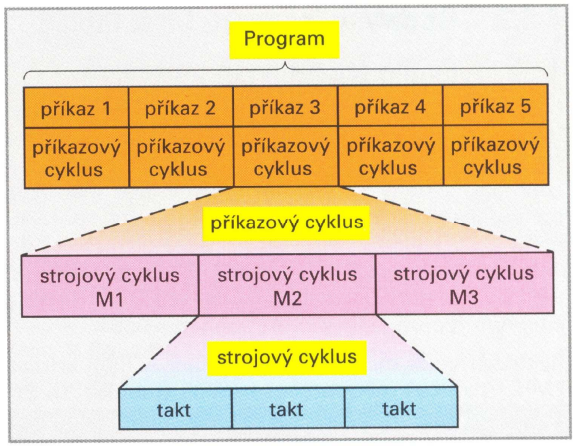
1. druh operace (operační znak)
2. adresa prvního operandu
3. adresa druhého operandu
4. adresa výsledku
5. adresa pokračování
6. adresa alternativního pokračování (např. při podmíněném skoku)

2.,3.,4. - adresová část instrukce

Eliminace bodu 5. - předpokládáme adresu o jedničku větší

Eliminace bodu 6. – předpokládáme umístění v adresové části instrukce

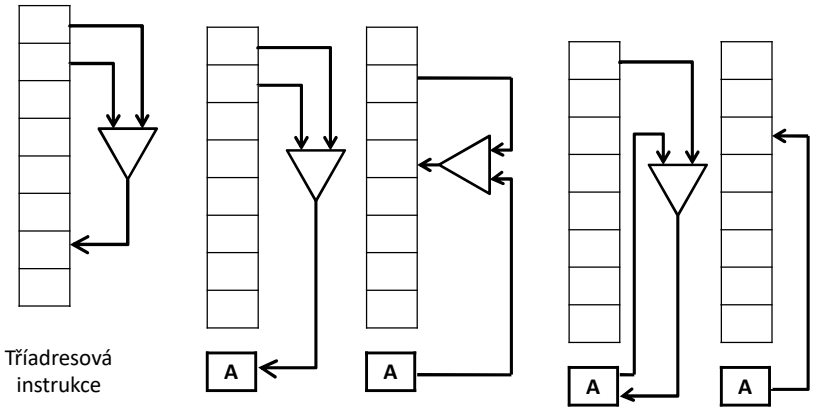
Vnitřní struktura provádění instrukcí (příkazů)



Fáze vykonávání instrukce

1. **Načtení instrukce (Instruction Fetch)** - V tomto kroku dochází k načtení operačního kódu instrukce, která má být zpracována, z paměti. PC (Program counter) vystaví na AB (adresovou sběrnici) adresu další instrukce a její operační kód je načten z paměti do IR (Instruction Register = registr instrukce).
2. **Dekódování instrukce (Instruction Decode)** - Instrukce je dekódována, tzn. dle jejího operačního znaku jsou nastaveny řídící/stavové signály a jsou dekódovány adresy operandů (pokud instrukce nějaké operandy má). Dále je podle výsledku dekódování spočítána adresa další instrukce
3. **Načtení operandů (Operand Fetch)** - Adresa operandů je vystavena na AB a operandy jsou nalezeny v paměti, případně v registrech, a vystaveny na DB (datová sběrnice)
4. **Vykonání instrukce (Execute)** - Provedení vlastní operace v ALU. Operace se liší podle použité instrukce (pozná se z operačního znaku) a provádí se nad operandy načtenými v předchozí krocích.
5. **Uložení výsledku (Result Store)** - Na AB se vystaví adresa cíle (v paměti, případně registr) a následně je do cíle zapsána hodnota spočítaná v ALU.
6. **Kontrola vnějšího přerušení (Asynchronous Interrupt Check)** - Ošetření asynchronních přerušení - pokud se objevila žádost o přerušení, je toto přerušení obslouženo. V opačném případě se pokračuje dále v provádění následující instrukce.

Redukce počtu adres v instrukci



Tříadresová instrukce

Dvouadresová instrukce

Jednoadresová instrukce

A = střadač (Accumulator)

Adresace

- **Implicitní adresování** -Při tomto způsobu není v instrukci explicitně uvedena adresa místa, o které při implicitním adresování jde. Tak např. u mikroprocesoru typu 8080 se u instrukce ADI D8, která k obsahu střadače přičítá číslo D8, předpokládá, že jeden z operandů je ve střadači (1. implicitní adresa), že výsledek zůstane tamtéž (2. implicitní adresa) a že adresa další instrukce je uložena v čítači instrukcí (3. implicitní adresa). Máme tu hned tři implicitní adresy, které však nejsou nikde v instrukci obsaženy ani nemají žádnou číselnou reprezentaci. Jsou dány jen typem instrukce a strukturou procesoru. Všechny ostatní druhy adres uvedené v následujících odstavcích patří k explicitním adresám.
- **Přímý operand — adresa nultého řádu**. Není to adresa operandu v pravém smyslu slova (proto „nultý řád“), nýbrž sám operand bezprostředně obsažený v instrukci. V anglosaské literatuře o mikroprocesorech bývá označována „Immediate address“ a příslušné instrukce „Immediate mode instruction“. Příkladem může být opět instrukce ADI D8, kde D8 je přímý operand obsažený v instrukci.

- **Přímá adresa** — adresa prvního řádu. Přímé adresování je základním adresovacím způsobem. Přímá adresa je v instrukci umístěné pořadové číslo registru nebo vstupu nebo buňky hlavní paměti, s jejímž obsahem se pracuje. příkladem je instrukce LDA D16 (přesuň do střadače obsah buňky hlavní paměti s adresou D16), nebo instrukce OUT D8 (předej obsah střadače na výstup s adresou D8).
- **Nepřímá adresa** — adresa druhého řádu. Při nepřímém adresování není součástí instrukce adresa operandu, nýbrž tzv. nepřímá adresa, na níž teprve najdeme adresu přímou.
- **Nepřímé adresy vyšších řádů** — adresy třetího a vyšších řádů. Jde o rozšíření pojmu nepřímá adresa. V instrukci máme nepřímou adresu, na níž opět najdeme nepřímou adresu, na níž opět najdeme nepřímou adresu atd. až na konci posloupnosti postupně čtených adres najdeme adresu přímou.

- **Adresy v operačním znaku** -Instrukce se běžně skládá ze dvou částí. Z operačního znaku, který určuje, co má procesor vykonat, a z adresové části, která obsahuje adresy nebo operandy nebo oboje. Některé instrukce však sestávají jen z operačního znaku a přesto číselně explicitně určují místa operandů. Tak např. instrukce MOV r1, r2 mikroprocesoru 8080, která uskutečňuje přesun operandu z registru r2 do r1 , má operační znak v binárním vyjádření 01 DDD SSS. Zde dvojici bitů 01 můžeme pokládat za operační znak v užším slova smyslu, neboť určuje, že jde o přesunovou instrukci typu MOV. Dvě trojice bitů DDD a SSS pak nazýváme adresami registrů r1 a r2. V operačních znacích však mohou být obsaženy i adresy vstupu a výstupu, podmínkových klopných obvodů, číselná označeni v hlavní paměti apod. Někteří autoři všechny adresy obsažené v operačním znaku kvalifikují jako implicitní. Toto pojetí však je sporné vždy, kdy jsme schopni v operačním znaku rozpoznat číselné vyjádření adres.

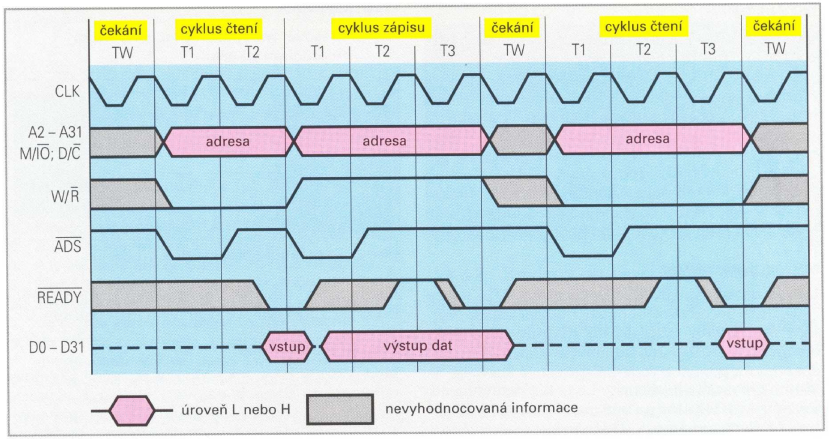
- **Registrové adresování** - Jde o zvláštní případ nepřímého adresování, při kterém operační znak obsahuje adresu registru či skupiny registrů, v nichž je uložena přímá adresa operandu. Tohoto způsobu adresováni lze s výhodou použít zejména tehdy, odkazujeme-li se při výkonu programu na jedno místo několikrát. Pak totiž nemusíme při každém dílčím odkazu používat znova úplnou přímou adresu, ale musíme ji zavést do adresačního registru pouze jednou. Analogicky, odkazujeme-li se postupně na sousední místa v paměti, stačí adresu v adresačním registru upravovat krátkou a rychlou instrukcí inkrementace nebo dekrementace. Mnohé mikroprocesory, a zejména ty, které obsahují zápisníkovou paměť, používají registrové adresování jako hlavní způsob adresování datové části hlavní paměti.

- **Relativní adresování** - Adresová část instrukce neurčuje v tomto případě přímo místo v paměti, ale představuje tzv. posunutí neboli relativní adresu. Úplnou adresu operandu získáme teprve přičtením relativní adresy ke vztažné, nebo též bázové adrese, která je obvykle uložená v bázovém registru. Protože relativní adresa odpovídá jen číselnému rozdílu mezi adresou operandu a obsahem bázového registru, může být adresová část instrukce krátká, což umožňuje jak zrychlení výkonu programů, tak i úsporu paměti pro program. Relativní adresování musí být zajištěno technickými prostředky, především bázovým registrem a adresovací sčítačkou. Z uvedeného popisu plyne, že relativní adresování je vlastně rozšířením registrového adresování.

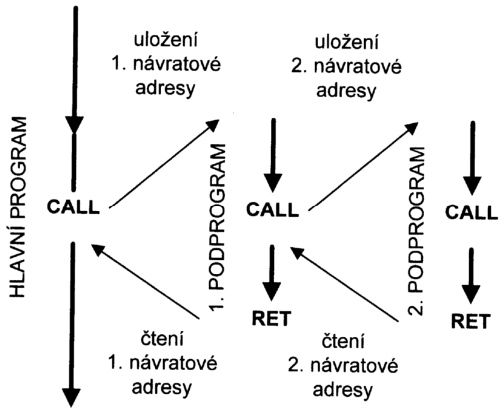
- **Autorelativní adresování** - V tomto případě je bázovou adresou adresa právě vykonávané instrukce a úlohu bázového registru hraje čítač instrukcí.Tento způsob adresování se nejčastěji používá u skokových instrukcí, kde relativní adresa určuje hodnotu přeskoku proti adrese instrukce. Autorelativní adresování lze s výhodou použít pro volně přemístitelné knihovní programy, protože při sestavování uživatelských programů z modulů knihovny podprogramů není nutné přeadresovat skokové instrukce. To však platí pouze tehdy, jestliže hodnoty přeskoků uvnitř knihovního podprogramu nepřesahují možnosti této adresace.

- **Adresování ukazateli** - snad každý mikroprocesor obsahuje speciální registry, které slouží pouze k adresování hlavni paměti. Jejich obsah „ukazuje“ na příslušnou buňku, a proto se jim říká ukazatel (Pointer). Prvním z nich je čítač instrukcí (Program Counter — PC), který automaticky adresuje instrukce. Druhým je ukazatel zásobníkové paměti (Stack Pointer — SP). Obsahují většinou přímé adresy a jejich užívání odlišujeme od předchozího registrového adresování, neboť adresa v ukazatelích se nejčastěji nastavuje automaticky, a tak nepotřebujeme ukazatel identifikovat žádnou explicitní nepřímou adresou.

Časový diagram při práci se sběrnicí



Volání podprogramů



Příznakové bity

- Bit **C** (též CY) — **přenos** (angl. **carry**) z nejvyššího bitu. Mění stav při aritmetických operacích, operacích porovnání a některých typech posunů a rotací.
- Bit **Z** — **nulovost** (angl. **zero**). U některých procesorů je zapamatován jako výsledek aritmeticko-logické operace, u jiných se vztahuje na momentální obsah akumulátoru. Po instrukcích přesunů dat do A tedy v prvém případě není bit Z ovlivněn, ve druhém ano.
- Bit **S** — **znaménko** čísla (angl. **sign**), též N (angl. negative). Je nastaven na I, byl-li znaménkový bit výsledku I (tj. záporné číslo) —jinak je vynulován.
- Bit **OF** — **přetečení** (angl. **overflow**). Mění stav při přetečení čísla se znaménkem (nejvyšší bit slova) po operaci součtu, rozdílu, obrácení znaménka, násobení a dělení. Operandy i výsledek se předpokládají ve dvojkovém doplňku. Kladný výsledek nemůže být větší než 7FF. . . F, vyšší hodnota by způsobila změnu znaménkového bitu. Záporný výsledek nemůže být menší než 800. . .0 a nižší hodnota by opět způsobila změnu znaménkového bitu.
- Bit **AC—pomocný přenos** (angl. **auxiliary carry**). Jedná se o přenos z bitu D3 do D4 při operaci součtu. Je využíván následnou instrukcí dekadické korekce akumulátoru, pokud sčítaná čísla byla v BCD kódu.
- Bit **P** — **parita** (angl. **parity**) vyjadřuje paritu výsledku (počet jedniček). Při liché paritě je P = I při lichém počtu jedniček, u sudé parity je P = I při sudém počtu jedniček. Je třeba zjistit, jakou paritu bit vyjadřuje. Paritní bit mají jen některé procesory.

Příznakové bity většinou využívají instrukce podmíněného skoku pro větvení programu.

Přerušení (Interrupt)

- Přerušení je reakce počítače na tzv. asynchronní, neočekávanou událost, která nemá žádný vztah k právě prováděnému programu a vyvolá její obsluhu
  - Externí přerušení – vyvoláno vnější událostí mimo mikroprocesor, např. vstup znaku z klávesnice, výstup znaku na tiskárnu, když tiskárna je připravena, příchod znaku ze sériové linky, atd.
  - Interní přerušení – vyvoláno vnitřní událostí v mikroprocesoru některou instrukcí, např. dělení nulou, výpadek stránky (u moderních mikroprocesorů), instrukcí INT (programové vyvolání přerušení), atd.
- Externí přerušení
  - Nemaskovatelné přerušení (Non Maskable Interrupt – NMI) – nelze zakázat, mikroprocesor reaguje vždy při výskytu požadavku (např. výpadek napájení)
  - Maskovatelné přerušení (Maskable Interrupt) – lze programově zakázat a povolit speciálními instrukcemi.

Přerušení

- Událost vyvolá přerušení (odtud název Přerušení) dosud běžícího programu a provede odskok na tzv. obslužný (přerušovací) podprogram. V rámci obslužného podprogramu je událost obsloužena a po ukončení obslužného podprogramu se počítač vrací do přerušeného programu.
- Přerušovací podprogram musí dodržovat určitá pravidla, neboť program nesmí poznat, že byl přerušen => přerušovací podprogram nesmí změnit vnitřní registry mikroprocesoru (pokud je mění – což je velmi časté – musí nejprve obsahy registrů uložit – obyčejně do zásobníku - a před ukončením přerušovacího podprogramu je zase obnovit)
- Latence přerušení (interrupt latency), neboli doba od výskytu požadavku na přerušení do začátku obsluhy by měla být co nejmenší.

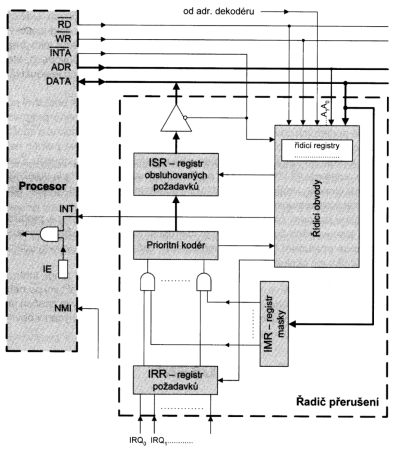
Činnost při vnějším přerušení

1. Vznikne požadavek na obsluhu přerušení
2. Řadič V/V zařízení generuje signál IRQ (Interrupt Request)
3. Řadič přerušení rozhodne o prioritě obsluhy (v případě příchodu více požadavků najednou) a generuje signál INT (Interrupt) pro procesor - někdo žádá o přerušení
4. Procesor rozhodne o obsluze (na základě maskování přerušení) - signál INTA (Interrupt Acknowledged)
5. Identifikace příčiny přerušení (číslo typu přerušení) – na datovou sběrnici řadič přerušení umístí buď instrukci volání podprogramu (starší procesory) nebo tzv. vektor přerušení – obyčejně 8 bitové číslo, které identifikuje příčinu přerušení (nové procesory)
6. Uložení stavové informace na zásobník - PC vždy (návratová adresa z přerušení) někdy i FR (Flag register) např. u 8086,případně i další údaje (tzv. kontext)
7. Nalezení začátku podprogramu pro obsluhu daného typu přerušení pomocí vektoru přerušení (nová hodnota PC) – buď v instrukci volání podprogramu dodané přerušovacím řadičem či určení z vektoru přerušení (obyčejně vyhledáním v tabulce)
8. Zakázaní přerušení – z důvodu zamezení vložených přerušení
9. Provedení podprogramu obsluhy přerušení (v rámci podprogramu je nutné povolit opět přerušení)
10. Návrat do přerušeného programu a obnova kontextu

Pravidla pro psaní přerušovacích podprogramů

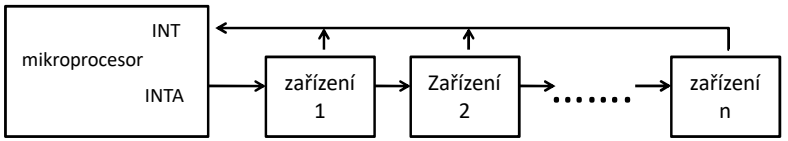
1. Uložit obsahy registrů (kontext) pokud nejsou uloženy automaticky – obyčejně se ukládají do zásobníku
2. Provést požadovanou obsluhu zdroje přerušení
3. Informovat přerušovací řadič o ukončené obsluze (aby mohl reagovat na další požadavek přerušení
4. Obnovit registry mikroprocesoru (pokud se neobnovují automaticky) => srovnat úroveň zásobníku
5. Povolit zakázané přerušení
6. Provést návrat do přerušeného programu instrukcí RET (případně speciální instrukcí pro návrat z přerušení

Blokové schema řadiče přerušení



Přerušovací řadič s paralelním určením priority (nejčastěji používané u externích radičů)

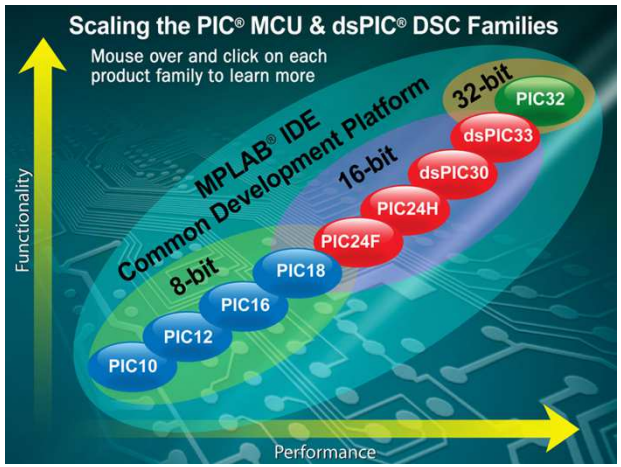
Řadič přerušení se sériovým přidělováním



Použito u stavebnice Zilog Z80

Procesory Microchip PIC

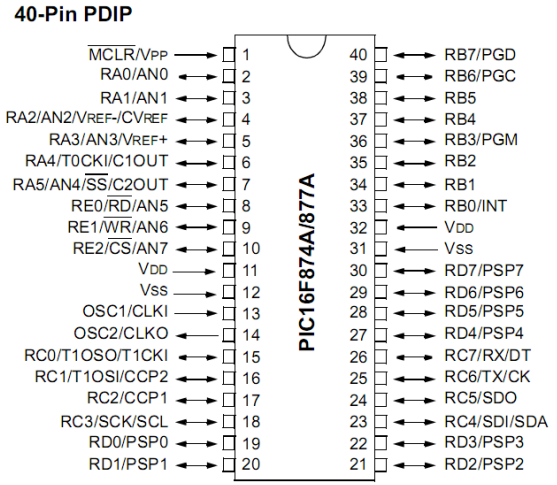
- Výkonné mikrořadiče (mikropočítače – kompletní počítač v jednou pouzdře
- Harvardská architektura
- 8, 16, 32-bitové produkční řady s různě velkou programovou a datovou pamětí a rozmanitými funkcemi a periferiemi (několik desítek typů)
- Hodinové frekvence cca do 80 MHz (podle typu)
- Možnost zvýšení výkonu použitím výkonnějšího typu – vývodově kompatibilní pouzdra
- Reprogramovatelné
- Relativně levné



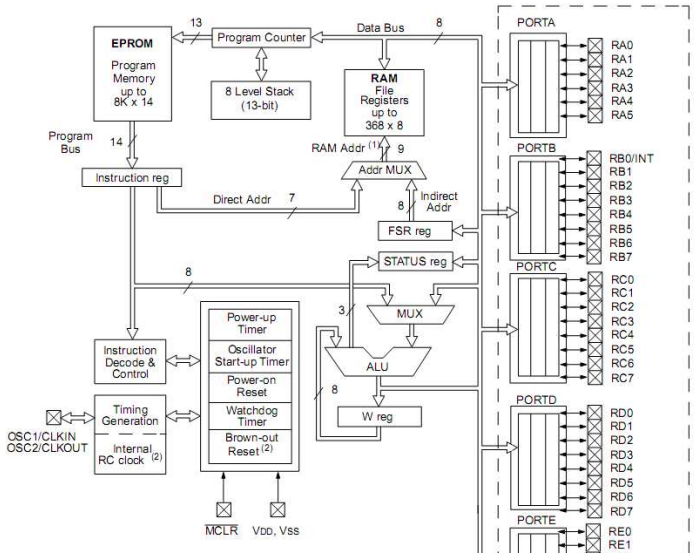
Různé typy PIC16F687xA

Device	Program Memory		Data SRAM (Bytes)	EEPROM (Bytes)	I/O	10-bit A/D (ch)	CCP (PWM)	MSSP		USART	Timers 8/16-bit	Comparators
	Bytes	# Single Word Instructions						SPI	Master I^2C			
PIC16F873A	7.2K	4096	192	128	22	5	2	Yes	Yes	Yes	2/1	2
PIC16F874A	7.2K	4096	192	128	33	8	2	Yes	Yes	Yes	2/1	2
PIC16F876A	14.3K	8192	368	256	22	5	2	Yes	Yes	Yes	2/1	2
PIC16F877A	14.3K	8192	368	256	33	8	2	Yes	Yes	Yes	2/1	2

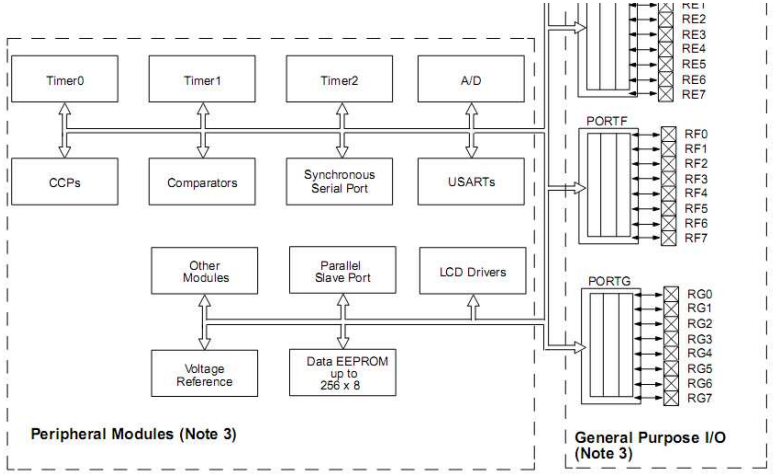
Zapojení pouzdra PIC16F877A



Vnitřní struktura obvodu PIC16F877A

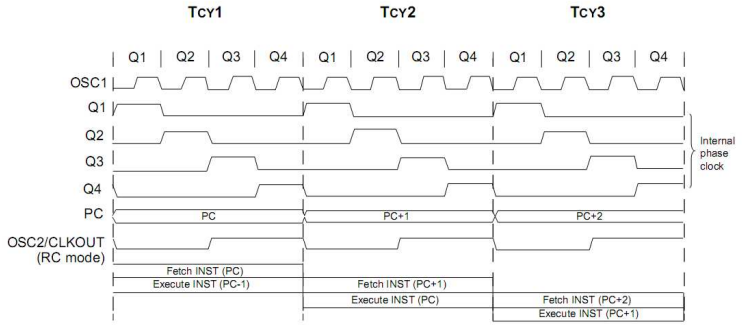


1.část

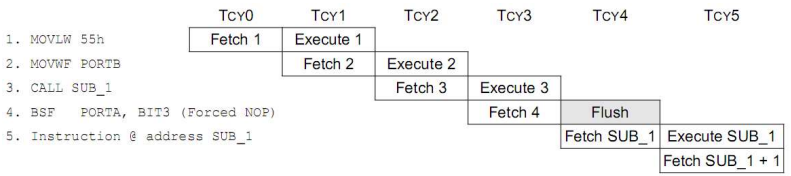


2.část (závislá na typu obvodu)

Časový průběh provádění instrukcí



Průběh instrukčních cyklů při volání podprogramu



Vynechání rozpracované instrukce BSF

Instrukční soubor procesorů PIC

Mnemonic, Operands		Description	Cycles	14-Bit Instruction Word MSb			LSb	Status Bits Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2

1. část

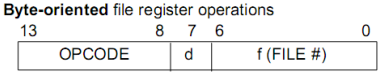
Instrukční soubor procesorů PIC

BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

2. část

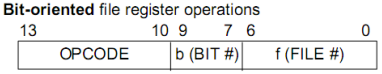
Vytvoření binárního kódu instrukce

Byte-oriented file register operations



d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address

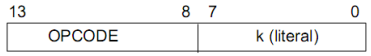
Bit-oriented file register operations



b = 3-bit bit address
f = 7-bit file register address

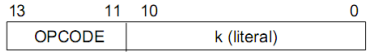
Literal and control operations

General



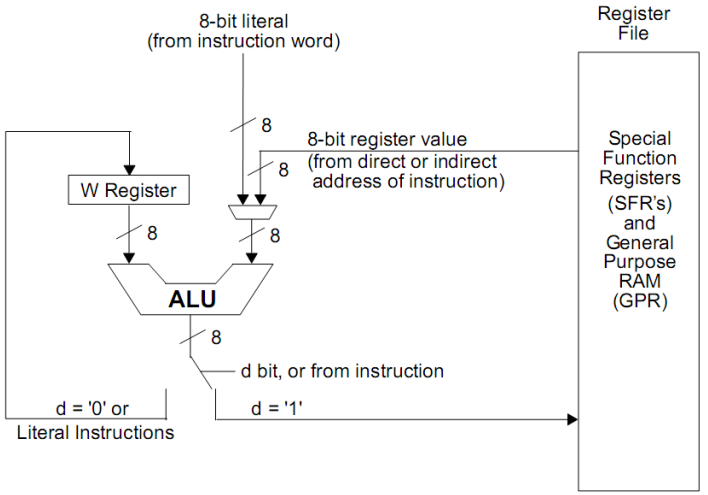
k = 8-bit immediate value

`CALL` and `GOTO` instructions only



k = 11-bit immediate value

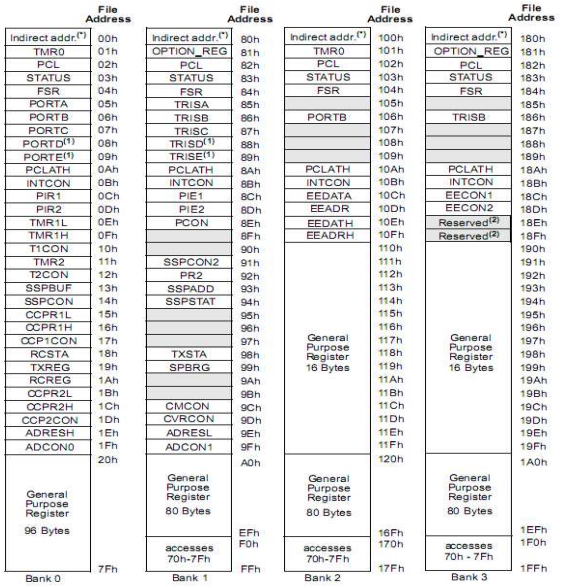
Blokové schema aritmeticko-logické operace s různými operandy



Mapa registrů

- 4 stránky registrů
- Význačné registry – dostupné ve všech stránkách
  - STATUS
  - INDF
  - FSR
  - INTCON
  - PCL
  - PCLATH
- Bity 5 a 6 STATUSu se používají pro určení stránky registrů

Mapa registrů



Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 1											
80h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
82h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
83h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
84h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	43, 150
86h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
87h	TRISC	PORTC Data Direction Register								1111 1111	47, 150
88h(4)	TRISD	PORTD Data Direction Register								1111 1111	48, 151
89h(4)	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	50, 151
8Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
8Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
8Ch	PIE1	PSPIE(2)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	25, 151
8Dh	PIE2	—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE	-0-0 0--0	27, 151
8Eh	PCON	—	—	—	—	—	—	$\overline{POR}$	$\overline{BOR}$	---- --qq	29, 151
8Fh	—	Unimplemented								—	—
90h	—	Unimplemented								—	—
91h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000	83, 151
92h	PR2	Timer2 Period Register								1111 1111	62, 151
93h	SSPADD	Synchronous Serial Port (I^2C mode) Address Register								0000 0000	79, 151
94h	SSPSTAT	SMP	CKE	$D/\overline{A}$	P	S	$R/\overline{W}$	UA	BF	0000 0000	79, 151
95h	—	Unimplemented								—	—
96h	—	Unimplemented								—	—
97h	—	Unimplemented								—	—
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	111, 151
99h	SPBRG	Baud Rate Generator Register								0000 0000	113, 151
9Ah	—	Unimplemented								—	—
9Bh	—	Unimplemented								—	—
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111	135, 151
9Dh	CVRCON	CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0	000- 0000	141, 151
9Eh	ADRESL	A/D Result Register Low Byte								xxxx xxxx	133, 151
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	128, 151

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 2											
100h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
101h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
102h(3)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	30, 150
103h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
104h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
105h	—	Unimplemented								—	—
106h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
107h	—	Unimplemented								—	—
108h	—	Unimplemented								—	—
109h	—	Unimplemented								—	—
10Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
10Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
10Ch	EEDATA	EEPROM Data Register Low Byte								xxxx xxxx	39, 151
10Dh	EEADR	EEPROM Address Register Low Byte								xxxx xxxx	39, 151
10Eh	EEDATH	—	—	EEPROM Data Register High Byte						--xx xxxx	39, 151
10Fh	EEADRH	—	—	—	—(5)	EEPROM Address Register High Byte				---- xxxx	39, 151
Bank 3											
180h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
181h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
182h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
183h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
184h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
185h	—	Unimplemented								—	—
186h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
187h	—	Unimplemented								—	—
188h	—	Unimplemented								—	—
189h	—	Unimplemented								—	—
18Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
18Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
18Ch	EECON1	EEPGD	—	—	—	WRERR	WREN	WR	RD	x--- x000	34, 151
18Dh	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	39, 151
18Eh	—	Reserved; maintain clear								0000 0000	—
18Fh	—	Reserved; maintain clear								0000 0000	—

STATUS registr 0x03,0x83,0x103,0x183

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7 **IRP**: Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h - 1FFh)
0 = Bank 0, 1 (00h - FFh)

For devices with only Bank0 and Bank1 the IRP bit is reserved, always maintain this bit clear.

bit 6:5 **RP1:RP0**: Register Bank Select bits (used for direct addressing)
11 = Bank 3 (180h - 1FFh)
10 = Bank 2 (100h - 17Fh)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

Each bank is 128 bytes. For devices with only Bank0 and Bank1 the IRP bit is reserved, always maintain this bit clear.

bit 4 **$\overline{TO}$**: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

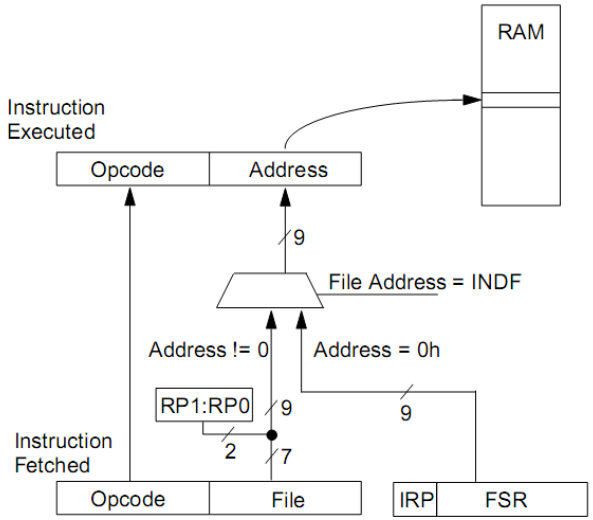
bit 3 **$\overline{PD}$**: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{borrow}$ the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{borrow}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred

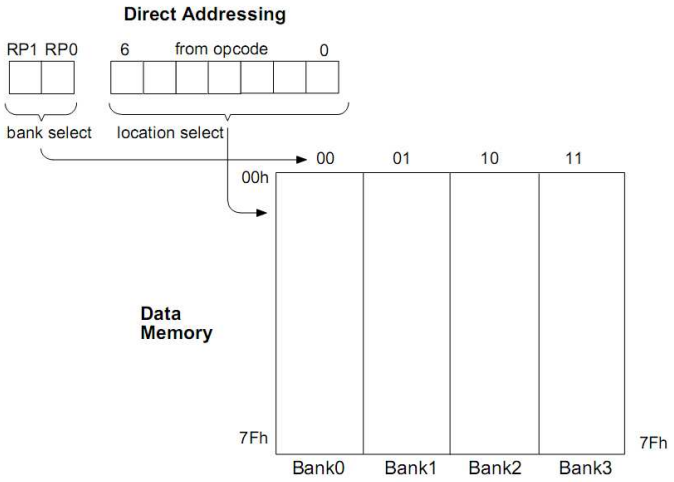
Schéma adresování paměti – přímé a nepřímé



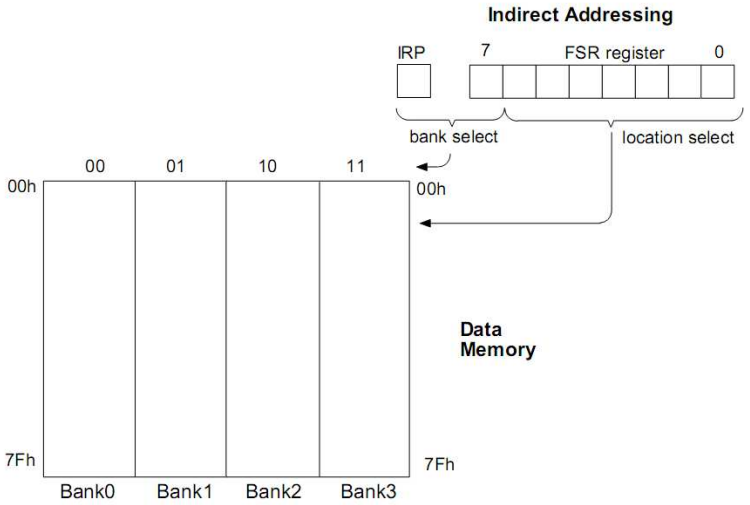
Definice stránek datové paměti

Accessed Bank	Direct (RP1:RP0)	Indirect (IRP)
0	0 0	0
1	0 1	
2	1 0	1
3	1 1	

Přímé adresování



Nepřímé adresování

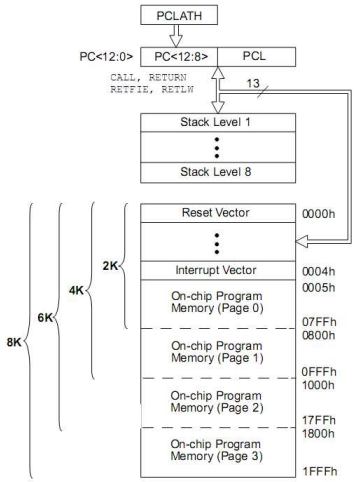


Nepřímé adresování

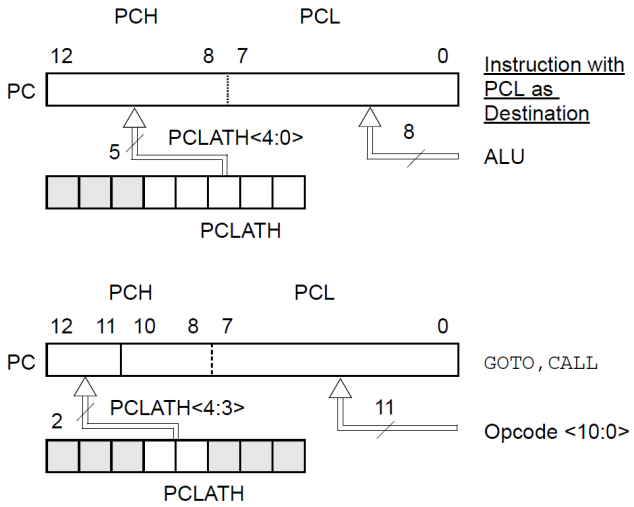
- Speciální registr FSR (0x04) je používán jako ukazatel do paměti dat
- Použije se instrukce s registrem INDF (0x00)
- Užitečné pro manipulaci tabulkových dat v paměti
- Příklad: vymažte blok paměti od adresy 0x20 do 0x2F

```
        movlw 0x20  ; inicializace FSR registru
        movwf FSR
next    clrf  INDF  ; vymaže místo v paměti na
                    které ukazuje FSR
        incf  FSR,1 ; inkrement FSR
        btfss FSR,4 ; kontrola, zda je hotovo
        goto next
continue
```

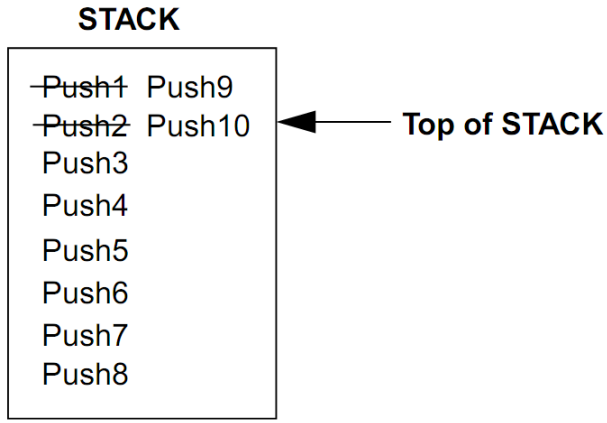
Organizace paměti programu



Práce s PC



Přetečení zásobníku



Důležité adresy

- 0x00 – Reset vector – Po zapnutí napájení či aktivaci Reset signálu se začíná provádět program od této adresy – obyčejně zde bývá instrukce Goto Start_adr
- 0x04 – Interrupt vector – při aktivaci přerušení se začíná provádět přerušovací podprogram od této adresy – obyčejně zde bývá instrukce GOTO INT_sub
- 0x05 a dále – pro programy a podprogramy

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register (0 to 7)
k	Literal field, constant data or label (may be either an 8-bit or an 11-bit value)
x	Don't care (0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f.
dest	Destination either the W register or the specified register file location
label	Label name
TOS	Top of Stack
PC	Program Counter
PCLATH	Program Counter High Latch
GIE	Global Interrupt Enable bit
WDT	Watchdog Timer
$\overline{TO}$	Time-out bit
$\overline{PD}$	Power-down bit
[ ]	Optional
( )	Contents
→	Assigned to
< >	Register bit field
∈	In the set of
italics	User defined term (font is courier)

ADDLW — Add Literal and W

Syntax: [*label*] ADDLW k

Operands: $0 \le k \le 255$

Operation: $(W) + k \rightarrow W$

Status Affected: C, DC, Z

Encoding:

11	111x	kkkk	kkkk

Description: The contents of the W register are added to the eight bit literal 'k' and the result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example1

```
ADDLW   0x15
```

Before Instruction
W = 0x10
After Instruction
W = 0x25

Example 2 `ADDLW MYREG`

Before Instruction
W = 0x10
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x47

Example 3 `ADDLW HIGH (LU_TABLE)`

Before Instruction
W = 0x10
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0xA3

Example 4 `ADDLW PCL`

Before Instruction
W = 0x10
Address of `PCL` † = 0x02
† `PCL` is the symbol for the Program Counter low byte location
After Instruction
W = 0x12

ADDWF — Add W and f

Syntax: [*label*] ADDWF f,d

Operands: $0 \le f \le 127$, $d \in [0,1]$

Operation: (W) + (f) → destination

Status Affected: C, DC, Z

Encoding:

00	0111	dfff	ffff

Description: Add the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1 `ADDWF FSR, 0`

Before Instruction
W = 0x17
FSR = 0xC2
After Instruction
W = 0xD9
FSR = 0xC2

Example 2 `ADDWF INDF, 1`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x37

Example 3 `ADDWF PCL ,1`

Case 1:
Before Instruction
W = 0x10
PCL = 0x37
C = x
After Instruction
PCL = 0x47
C = 0

Case 2:
Before Instruction
W = 0x10
PCL = 0xF7
PCH = 0x08
C = x
After Instruction
PCL = 0x07
PCH = 0x08
C = 1

ANDLW — And Literal with W

Syntax: [*label*] ANDLW k

Operands: $0 \le k \le 255$

Operation: (W).AND. (k) → W

Status Affected: Z

Encoding:

11	1001	kkkk	kkkk

Description: The contents of W register are AND'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example 1 `ANDLW 0x5F`

```
Before Instruction                ; 0101 1111  (0x5F)
    W = 0xA3                      ; 1010 0011  (0xA3)
After Instruction                 ;----------  ------
    W = 0x03                      ; 0000 0011  (0x03)
```

Example 2

```
ANDLW   MYREG
```

Before Instruction
W = 0xA3
Address of MYREG † = 0x37
† MYREG is a symbol for a data memory location
After Instruction
W = 0x23

Example 3

```
ANDLW   HIGH (LU_TABLE)
```

Before Instruction
W = 0xA3
Address of LU_TABLE † = 0x9375
† LU_TABLE is a label for an address in program memory
After Instruction
W = 0x83

ANDWF — AND W with f

Syntax: [*label*] ANDWF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (W).AND. (f) → destination

Status Affected: Z

Encoding:

00	0101	dfff	ffff

Description: AND the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
ANDWF   FSR, 1
```

```
Before Instruction                  ; 0001 0111  (0x17)
        W   = 0x17                  ; 1100 0010  (0xC2)
        FSR = 0xC2                  ;----------  ------
After Instruction                   ; 0000 0010  (0x02)
        W   = 0x17
        FSR = 0x02
```

Example 2

```
ANDWF   FSR, 0
```

```
Before Instruction                  ; 0001 0111  (0x17)
        W   = 0x17                  ; 1100 0010  (0xC2)
        FSR = 0xC2                  ;----------  ------
After Instruction                   ; 0000 0010  (0x02)
        W   = 0x02
        FSR = 0xC2
```

Example 3

```
ANDWF   INDF, 1
```

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x5A
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x15

BCF — Bit Clear f

Syntax: [*label*] BCF f,b

Operands: $0 \le f \le 127$
$0 \le b \le 7$

Operation: 0 → f<b>

Status Affected: None

Encoding:

01	00bb	bfff	ffff

Description: Bit 'b' in register 'f' is cleared.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example 1

```
BCF     FLAG_REG, 7
```

```
Before Instruction
        FLAG_REG = 0xC7         ; 1100 0111
After Instruction
        FLAG_REG = 0x47         ; 0100 0111
```

Example 2 `BCF INDF, 3`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x2F

After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x27

BSF — Bit Set f

Syntax: [*label*] BSF f,b

Operands: $0 \le f \le 127$
$0 \le b \le 7$

Operation: $1 \rightarrow f<b>$

Status Affected: None

Encoding:

01	01bb	bfff	ffff

Description: Bit 'b' in register 'f' is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example 1 `BSF FLAG_REG, 7`

Before Instruction
FLAG_REG =0x0A ; 0000 1010

After Instruction
FLAG_REG =0x8A ; 1000 1010

Example 2 `BSF INDF, 3`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20

After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x28

BTFSC — Bit Test, Skip if Clear

Syntax: [*label*] BTFSC f,b

Operands: $0 \le f \le 127$
$0 \le b \le 7$

Operation: skip if (f<b>) = 0

Status Affected: None

Encoding:

01	10bb	bfff	ffff

Description: If bit 'b' in register 'f' is '0' then the next instruction is skipped.
If bit 'b' is '0' then the next instruction (fetched during the current instruction execution) is discarded, and a NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	No operation

If skip (2nd cycle):

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example 1

```
HERE    BTFSC   FLAG, 4
FALSE   GOTO    PROCESS_CODE
TRUE    •
        •
        •
```

Case 1: Before Instruction

PC = address HERE
FLAG= xxx0 xxxx

After Instruction

Since FLAG<4>= 0,
PC = address TRUE

Case 2: Before Instruction

PC = address HERE
FLAG= xxx1 xxxx

After Instruction

Since FLAG<4>=1,
PC = address FALSE

BTFSS

Bit Test f, Skip if Set

Syntax: [*label*] BTFSS f,b

Operands: $0 \le f \le 127$
$0 \le b < 7$

Operation: skip if (f<b>) = 1

Status Affected: None

Encoding:

01	11bb	bfff	ffff

Description: If bit 'b' in register 'f' is '1' then the next instruction is skipped.
If bit 'b' is '1', then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	No operation

If skip (2nd cycle):

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example 1

```
HERE    BTFSS   FLAG, 4
FALSE   GOTO    PROCESS_CODE
TRUE    •
        •
        •
```

Case 1: Before Instruction

PC = address HERE
FLAG= xxx0 xxxx

After Instruction

Since FLAG<4>= 0,
PC = address FALSE

Case 2: Before Instruction

PC = address HERE
FLAG= xxx1 xxxx

After Instruction

Since FLAG<4>=1,
PC = address TRUE

CALL

Call Subroutine

Syntax: [*label*] CALL k

Operands: $0 \le k \le 2047$

Operation: (PC)+ 1→ TOS,
k → PC<10:0>,
(PCLATH<4:3>) → PC<12:11>

Status Affected: None

Encoding:

10	0kkk	kkkk	kkkk

Description: Call Subroutine. First, the 13-bit return address (PC+1) is pushed onto the stack. The eleven bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH<4:3>. CALL is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

1st cycle:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	No operation

2nd cycle:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example 1 `HERE CALL THERE`

Before Instruction
PC = Address`HERE`
After Instruction
TOS = Address `HERE+1`
PC = Address `THERE`

CLRF Clear f

Syntax: [*label*] CLRF f

Operands: $0 \le f \le 127$

Operation: $00h \rightarrow f$
$1 \rightarrow Z$

Status Affected: Z

Encoding:

00	0001	lfff	ffff

Description: The contents of register 'f' are cleared and the Z bit is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example 1 `CLRF FLAG_REG`

Before Instruction
FLAG_REG=0x5A
After Instruction
FLAG_REG=0x00
Z = 1

Example 2 `CLRF INDF`

Before Instruction
FSR = 0xC2
Contents of Address (FSR)=0xAA
After Instruction
FSR = 0xC2
Contents of Address (FSR)=0x00
Z = 1

CLRW Clear W

Syntax: [*label*] CLRW

Operands: None

Operation: $00h \rightarrow W$
$1 \rightarrow Z$

Status Affected: Z

Encoding:

00	0001	0xxx	xxxx

Description: W register is cleared. Zero bit (Z) is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'W'

Example 1 `CLRW`

Before Instruction
W = 0x5A
After Instruction
W = 0x00
Z = 1

CLRWDT — Clear Watchdog Timer

Syntax: [*label*] CLRWDT

Operands: None

Operation: 00h → WDT
0 → WDT prescaler count,
1 → $\overline{TO}$
1 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

00	0000	0110	0100

Description: CLRWDT instruction clears the Watchdog Timer. It also clears the prescaler count of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process data	Clear WDT Counter

Example 1

```
CLRWDT
```

Before Instruction
WDT counter= x
WDT prescaler =1:128
After Instruction
WDT counter=0x00
WDT prescaler count=0
$\overline{TO}$ = 1
$\overline{PD}$ = 1
WDT prescaler =1:128

Note: The CLRWDT instruction does not affect the assignment of the WDT prescaler.

COMF — Complement f

Syntax: [*label*] COMF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: ($\bar{f}$) → destination

Status Affected: Z

Encoding:

00	1001	dfff	ffff

Description: The contents of register 'f' are 1's complemented. If 'd' is 0 the result is stored in W. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
COMF    REG1, 0
```

Before Instruction
REG1= 0x13
After Instruction
REG1= 0x13
W = 0xEC

Example 2

```
COMF    INDF, 1
```

Before Instruction
FSR = 0xC2
Contents of Address (FSR)=0xAA
After Instruction
FSR = 0xC2
Contents of Address (FSR)=0x55

Example 3

```
COMF    REG1, 1
```

Before Instruction
REG1= 0xFF
After Instruction
REG1= 0x00
Z = 1

DECF

Decrement f

Syntax: [*label*] DECF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) - 1 → destination

Status Affected: Z

Encoding:

00	0011	dfff	ffff

Description: Decrement register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
DECF    CNT, 1
```

Before Instruction
CNT = 0x01
Z = 0
After Instruction
CNT = 0x00
Z = 1

Example 2

```
DECF    INDF, 1
```

Before Instruction
FSR = 0xC2
Contents of Address (FSR) = 0x01
Z = 0
After Instruction
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z = 1

Example 3

```
DECF    CNT, 0
```

Before Instruction
CNT = 0x10
W = x
Z = 0
After Instruction
CNT = 0x10
W = 0x0F
Z = 0

DECFSZ

Decrement f, Skip if 0

Syntax: [*label*] DECFSZ f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) - 1 → destination; skip if result = 0

Status Affected: None

Encoding:

00	1011	dfff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.
If the result is 0, then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

If skip (2nd cycle):

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example

```
HERE    DECFSZ  CNT, 1
        GOTO    LOOP
CONTINUE •
         •
         •
```

Case 1:
Before Instruction
PC = address HERE
CNT = 0x01
After Instruction
CNT = 0x00
PC = address CONTINUE

Case 2:
Before Instruction
PC = address HERE
CNT = 0x02
After Instruction
CNT = 0x01
PC = address HERE + 1

GOTO — Unconditional Branch

Syntax: [*label*] GOTO k

Operands: 0 ≤ k ≤ 2047

Operation: k → PC<10:0>
PCLATH<4:3> → PC<12:11>

Status Affected: None

Encoding:

10	1kkk	kkkk	kkkk

Description: GOTO is an unconditional branch. The eleven bit immediate value is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH<4:3>. GOTO is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

1st cycle:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>	Process data	No operation

2nd cycle:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

INCF — Increment f

Syntax: [*label*] INCF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) + 1 → destination

Status Affected: Z

Encoding:

00	1010	dfff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
INCF    CNT, 1
```

Before Instruction
CNT = 0xFF
Z = 0
After Instruction
CNT = 0x00
Z = 1

Example 2

```
INCF    INDF, 1
```

Before Instruction
FSR = 0xC2
Contents of Address (FSR) = 0xFF
Z = 0
After Instruction
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z = 1

Example 3

```
INCF    CNT, 0
```

Before Instruction
CNT = 0x10
W = x
Z = 0
After Instruction
CNT = 0x10
W = 0x11
Z = 0

INCFSZ — Increment f, Skip if 0

Syntax: [*label*] INCFSZ f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) + 1 → destination, skip if result = 0

Status Affected: None

Encoding:

00	1111	dfff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.
If the result is 0, then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

If skip (2nd cycle):

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example

```
HERE      INCFSZ   CNT, 1
          GOTO     LOOP
CONTINUE  •
          •
          •
```

Case 1: Before Instruction

PC = address HERE
CNT = 0xFF

After Instruction

CNT = 0x00
PC = address CONTINUE

Case 2: Before Instruction

PC = address HERE
CNT = 0x00

After Instruction

CNT = 0x01
PC = address HERE + 1

IORLW

Inclusive OR Literal with W

Syntax: [*label*] IORLW k

Operands: 0 ≤ k ≤ 255

Operation: (W).OR. k → W

Status Affected: Z

Encoding:

11	1000	kkkk	kkkk

Description: The contents of the W register is OR'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example 1

```
IORLW   0x35
```

Before Instruction
W = 0x9A
After Instruction
W = 0xBF
Z = 0

Example 2

```
IORLW   MYREG
```

Before Instruction
W = 0x9A
Address of MYREG † = 0x37
† MYREG is a symbol for a data memory location
After Instruction
W = 0x9F
Z = 0

Example 3

```
IORLW   HIGH (LU_TABLE)
```

Before Instruction
W = 0x9A
Address of LU_TABLE † = 0x9375
† LU_TABLE is a label for an address in program memory
After Instruction
W = 0x9B
Z = 0

Example 4

```
IORLW   0x00
```

Before Instruction
W = 0x00
After Instruction
W = 0x00
Z = 1

IORWF

Inclusive OR W with f

Syntax: [*label*] IORWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (W).OR. (f) → destination

Status Affected: $\overline{Z}$

Encoding:

00	0100	dfff	ffff

Description: Inclusive OR the W register with register 'f'. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
IORWF   RESULT, 0
```

Before Instruction
RESULT= 0x13
W = 0x91
After Instruction
RESULT= 0x13
W = 0x93
Z = 0

Example 2 `IORWF INDF, 1`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x30
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x37
Z = 0

Example 3 `IORWF RESULT, 1`

Case 1: Before Instruction
RESULT= 0x13
W = 0x91
After Instruction
RESULT= 0x93
W = 0x91
Z = 0

Case 2: Before Instruction
RESULT= 0x00
W = 0x00
After Instruction
RESULT= 0x00
W = 0x00
Z = 1

MOVLW Move Literal to W

Syntax: [*label*] MOVLW k

Operands: $0 \le k \le 255$

Operation: $k \rightarrow W$

Status Affected: None

Encoding:

11	00xx	kkkk	kkkk

Description: The eight bit literal 'k' is loaded into W register. The don't cares will assemble as 0's.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example 1 `MOVLW 0x5A`

After Instruction
W = 0x5A

Example 2 `MOVLW MYREG`

Before Instruction
W = 0x10
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x37

Example 3 `MOVLW HIGH (LU_TABLE)`

Before Instruction
W = 0x10
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0x93

MOVF Move f

Syntax: [*label*] MOVF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) → destination

Status Affected: Z

Encoding:

00	1000	dfff	ffff

Description: The contents of register 'f' is moved to a destination dependent upon the status of 'd'. If 'd' = 0, destination is W register. If 'd' = 1, the destination is file register 'f' itself. 'd' = 1 is useful to test a file register since status flag Z is affected.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1 `MOVF FSR, 0`

Before Instruction
W = 0x00
FSR = 0xC2
After Instruction
W = 0xC2
Z = 0

Example 2 `MOVF INDF, 0`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z = 1

Example 3 `MOVF FSR, 1`

Case 1: Before Instruction
FSR = 0x43
After Instruction
FSR = 0x43
Z = 0

Case 2: Before Instruction
FSR = 0x00
After Instruction
FSR = 0x00
Z = 1

MOVWF — Move W to f

Syntax: [*label*] MOVWF f
Operands: $0 \le f \le 127$
Operation: $(W) \rightarrow f$
Status Affected: None
Encoding:

00	0000	1fff	ffff

Description: Move data from W register to register 'f'.
Words: 1
Cycles: 1
Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example 1 `MOVWF OPTION_REG`

Before Instruction
OPTION_REG=0xFF
W = 0x4F
After Instruction
OPTION_REG=0x4F
W = 0x4F

Example 2 `MOVWF INDF`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x17

NOP — No Operation

Syntax: [*label*] NOP
Operands: None
Operation: No operation
Status Affected: None
Encoding:

00	0000	0xx0	0000

Description: No operation.
Words: 1
Cycles: 1
Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation

Example : `HERE NOP`

Before Instruction
PC = address `HERE`
After Instruction
PC = address `HERE + 1`

OPTION — Load Option Register

Syntax:	[*label*] OPTION
Operands:	None
Operation:	(W) → OPTION
Status Affected:	None
Encoding:	00 | 0000 | 0110 | 0010
Description:	The contents of the W register are loaded in the OPTION register. This instruction is supported for code compatibility with PIC16C5X products. Since OPTION is a readable/writable register, the user can directly address it.
Words:	1
Cycles:	1

To maintain upward compatibility with future PIC16CXX products, do not use this instruction.

RETFIE — Return from Interrupt

Syntax:	[*label*] RETFIE
Operands:	None
Operation:	TOS → PC, 1 → GIE
Status Affected:	None
Encoding:	00 | 0000 | 0000 | 1001
Description:	Return from Interrupt. The 13-bit address at the Top of Stack (TOS) is loaded in the PC. The Global Interrupt Enable bit, GIE (INTCON<7>), is automatically set, enabling Interrupts. This is a two cycle instruction.
Words:	1
Cycles:	2

Q Cycle Activity:

1st cycle:

Q1	Q2	Q3	Q4
Decode	No operation	Process data	No operation

2nd cycle:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example

```
RETFIE
```

After Instruction

PC = TOS

GIE = 1

RETLW — Return with Literal in W

Syntax:	[*label*] RETLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → W; TOS → PC
Status Affected:	None
Encoding:	11 | 01xx | kkkk | kkkk
Description:	The W register is loaded with the eight bit literal 'k'. The program counter is loaded 13-bit address at the Top of Stack (the return address). This is a two cycle instruction.
Words:	1
Cycles:	2

Q Cycle Activity:

1st cycle:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

2nd cycle:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example

```
HERE    CALL TABLE   ; W contains table
                     ; offset value
        •            ; W now has table value
        •
        •
TABLE   ADDWF PC     ;W = offset
        RETLW k1     ;Begin table
        RETLW k2     ;
        •
        •
        •
        RETLW kn     ; End of table
```

Before Instruction

W = 0x07

After Instruction

W = value of k8

PC = TOS = Address `Here + 1`

RETURN

Return from Subroutine

Syntax: [*label*] RETURN

Operands: None

Operation: TOS → PC

Status Affected: None

Encoding:

00	0000	0000	1000

Description: Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

1st cycle:

Q1	Q2	Q3	Q4
Decode	No operation	Process data	No operation

2nd cycle:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

Example

```
HERE    RETURN
```

After Instruction

PC = TOS

RLF

Rotate Left f through Carry

Syntax: [*label*] RLF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

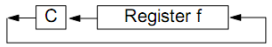
Operation: See description below

Status Affected: C

Encoding:

00	1101	dfff	ffff

Description: The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is stored back in register 'f'.



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1 `RLF REG1,0`

Before Instruction
REG1= 1110 0110
C = 0
After Instruction
REG1=1110 0110
W =1100 1100
C =1

Example 2 `RLF INDF, 1`

Case 1: Before Instruction
W = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 0011 1010
C = 1
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0111 0101
C = 0

Case 2: Before Instruction
W = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 1011 1001
C = 0
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0111 0010
C = 1

RRF

Rotate Right f through Carry

Syntax: [*label*] RRF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

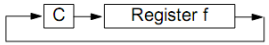
Operation: See description below

Status Affected: C

Encoding:

00	1100	dfff	ffff

Description: The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1 `RRF REG1,0`

Before Instruction
REG1= 1110 0110
W = xxxx xxxx
C = 0
After Instruction
REG1= 1110 0110
W = 0111 0011
C = 0

Example 2 `RRF INDF, 1`

Case 1: Before Instruction
W = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 0011 1010
C = 1
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 1001 1101
C = 0

Case 2: Before Instruction
W = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 0011 1001
C = 0
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0001 1100
C = 1

SLEEP

Syntax: [*label*] SLEEP

Operands: None

Operation: 00h → WDT,
0 → WDT prescaler count,
1 → $\overline{TO}$,
0 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

00	0000	0110	0011

Description: The power-down status bit, $\overline{PD}$ is cleared. Time-out status bit, $\overline{TO}$ is set. Watchdog Timer and its prescaler count are cleared.
The processor is put into SLEEP mode with the oscillator stopped.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	Go to sleep

Example: `SLEEP`

Note: The `SLEEP` instruction does not affect the assignment of the WDT prescaler

SUBLW — Subtract W from Literal

Syntax: [*label*] SUBLW k

Operands: 0 ≤ k ≤ 255

Operation: k - (W) → W

Status Affected: C, DC, Z

Encoding:

11	110x	kkkk	kkkk

Description: The W register is subtracted (2's complement method) from the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example 1: `SUBLW 0x02`

Case 1: Before Instruction

```
W = 0x01
C = x
Z = x
```

After Instruction

```
W = 0x01
C = 1      ; result is positive
Z = 0
```

Case 2: Before Instruction

```
W = 0x02
C = x
Z = x
```

After Instruction

```
W = 0x00
C = 1      ; result is zero
Z = 1
```

Case 3: Before Instruction

```
W = 0x03
C = x
Z = x
```

After Instruction

```
W = 0xFF
C = 0      ; result is negative
Z = 0
```

Example 2 `SUBLW MYREG`

Before Instruction

```
W = 0x10
```

Address of `MYREG` † = 0x37

† `MYREG` is a symbol for a data memory location

After Instruction

```
W = 0x27
C = 1      ; result is positive
```

SUBWF — Subtract W from f

Syntax: [*label*] SUBWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) - (W) → destination

Status Affected: C, DC, Z

Encoding:

00	0010	dfff	ffff

Description: Subtract (2's complement method) W register from register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1: `SUBWF REG1,1`

Case 1: Before Instruction

```
REG1= 3
W   = 2
C   = x
Z   = x
```

After Instruction

```
REG1= 1
W   = 2
C   = 1        ; result is positive
Z   = 0
```

Case 2: Before Instruction

```
REG1= 2
W   = 2
C   = x
Z   = x
```

After Instruction

```
REG1= 0
W   = 2
C   = 1        ; result is zero
Z   = 1
```

SWAPF — Swap Nibbles in f

Syntax: [*label*] SWAPF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f<3:0>) → destination<7:4>,
(f<7:4>) → destination<3:0>

Status Affected: None

Encoding:

00	1110	dfff	ffff

Description: The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0 the result is placed in W register. If 'd' is 1 the result is placed in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1 `SWAPF REG, 0`

Before Instruction

```
REG1= 0xA5
```

After Instruction

```
REG1= 0xA5
W   = 0x5A
```

Example 2 `SWAPF INDF, 1`

Before Instruction

```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20
```

After Instruction

```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x02
```

Example 3 `SWAPF REG, 1`

Before Instruction

```
REG1= 0xA5
```

After Instruction

```
REG1= 0x5A
```

TRIS — Load TRIS Register

Syntax: [*label*] TRIS f

Operands: $5 \le f \le 7$

Operation: (W) → TRIS register f;

Status Affected: None

Encoding:

00	0000	0110	0fff

Description: The instruction is supported for code compatibility with the PIC16C5X products. Since TRIS registers are readable and writable, the user can directly address them.

Words: 1

Cycles: 1

Example

To maintain upward compatibility with future PIC16CXX products, do not use this instruction.

XORLW — Exclusive OR Literal with W

Syntax: [*label*] XORLW k

Operands: 0 ≤ k ≤ 255

Operation: (W).XOR. k → W

Status Affected: Z

Encoding:

11	1010	kkkk	kkkk

Description: The contents of the W register are XOR'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W register

Example 1

```
XORLW   0xAF            ; 1010 1111   (0xAF)
Before Instruction      ; 1011 0101   (0xB5)
    W  =  0xB5          ; ---------   ------
After Instruction       ; 0001 1010   (0x1A)
    W  =  0x1A
    Z  =  0
```

Example 2

```
XORLW   MYREG
```

Before Instruction
W = 0xAF
Address of MYREG † = 0x37
† MYREG is a symbol for a data memory location
After Instruction
W = 0x18
Z = 0

Example 3

```
XORLW   HIGH (LU_TABLE)
```

Before Instruction
W = 0xAF
Address of LU_TABLE † = 0x9375
† LU_TABLE is a label for an address in program memory
After Instruction
W = 0x3C
Z = 0

XORWF — Exclusive OR W with f

Syntax: [*label*] XORWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (W).XOR. (f) → destination

Status Affected: Z

Encoding:

00	0110	dfff	ffff

Description: Exclusive OR the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1

```
XORWF   REG, 1          ; 1010 1111   (0xAF)
Before Instruction      ; 1011 0101   (0xB5)
    REG=  0xAF          ; ---------   ------
    W  =  0xB5          ; 0001 1010   (0x1A)
After Instruction
    REG=  0x1A
    W  =  0xB5
```

Example 2

```
XORWF   REG, 0          ; 1010 1111   (0xAF)
Before Instruction      ; 1011 0101   (0xB5)
    REG=  0xAF          ; ---------   ------
    W  =  0xB5          ; 0001 1010   (0x1A)
After Instruction
    REG=  0xAF
    W  =  0x1A
```

Example 3

```
XORWF   INDF, 1
```

Before Instruction
W = 0xB5
FSR = 0xC2
Contents of Address (FSR) = 0xAF
After Instruction
W = 0xB5
FSR = 0xC2
Contents of Address (FSR) = 0x1A

Jazyk symbolických adres

Obyčejně dvouprůchodový překladač

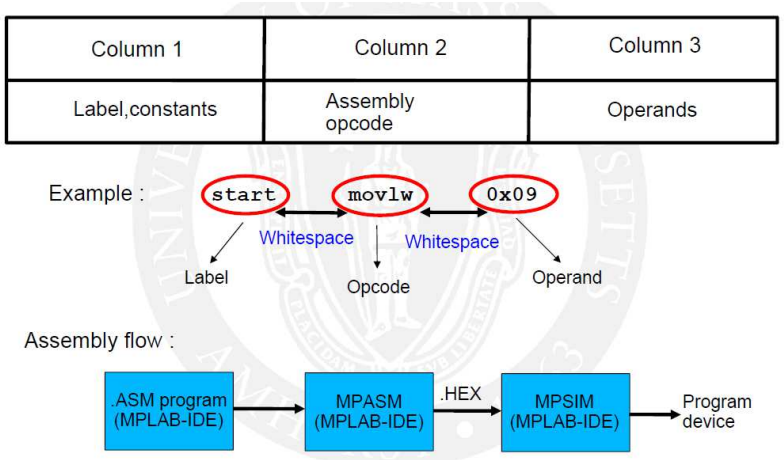
1. průchod – generace binárního kódu instrukce (s vyhrazením místa pro konkrétní adresy) a generace tabulky symbolických adres
2. průchod – doplnění příslušné adresy za symbolickou adresu a vygenerování konečného binárního kódu

Pravidla zápisu zdrojového souboru.

Řádek je rozdělen na 4 části

1. Pole návěští
2. Pole instrukčního kódu (mnemokód)
3. Pole operandů
4. Pole komentářů

Zdrojový text v Assembleru



Pole návěští

- Symbolické označení adresy na kterou je možno se odkazovat ve zdrojovém programu.
- Je nepovinné, musí začínat v prvním sloupci, délka závisí na konkrétním překladači (obyčejně min. 6 znaků). Nesmí začínat číslicí.
- Libovolná kombinace písmen a číslic anglické abecedy + některé speciální znaky
- Ukončeno dvojtečkou (obyčejně) – některé asemblery nevyžadují dvojtečku

Pole instrukčního kódu

- Pouze dovolené mnemokódy instrukcí a pseudoinstrukcí. Nesmí začínat v 1. sloupci.

Pole operandů

- Pouze u instrukcí, pseudoinstrukcí a maker, které připouštějí operandy.
- Od pole instrukcí odděleno min. 1 mezerou
- Operandy odděleny čárkou

Pseudoinstrukce Assembleru

- **ORG** k
  - umístí příští instrukci na adresu k v programové paměti – používá se k definici umístění programu v paměti
  - Příklad: ORG 0x10
movlw 0x09
- Konstanta **EQU** hodnota
  - Hodnota je přiřazena konstantě – V programu potom používáme konstanty a assembler automaticky doplní konkrétní hodnotu – zpřehlední se program
  - Příklad: STATUS EQU 0x03
- **END**
  - Indikuje konec programu
- **MACRO**
  - Definuje tzv. makroinstrukci – může být složena z několika instrukcí
  - Překladač makroinstrukci rozvine do definovaných instrukcí
- **ENDM**
  - Konec definice makroinstrukce

IF-THEN-ELSE kód

```
        btfsc STATUS,Z
        goto Zset
Zclear  bsf ....          → if (z==0)
        ; instructions to execute if Z=0
        goto Zdone
Zset
        ; instructions to execute if Z=1

Zdone   movlw....    ; Carry on with program
        end
```

Šablona pro psaní programů

```
count equ 0x30
```
Assign symbolic names to registers

```
        org 0x00
        goto start
        org 0x04
        call IS_ROUTINE
        org 0x10
```
Initialize reset and interrupt vectors

```
    ; PROGRAM BODY
```
Main program body

```
IS_ROUTINE
        ; ISR BODY
        retfie
        end
```
Interrupt Service Routine body

Práce s registrovými stránkami

- Pro přepínání stránek je možno použít např. tento kód:

```
bsf  STATUS,5   ; přechod ze stránky 0
                  na stránku 1
movlw  0x00     ; naplnění W reg. hodnotou 0
movwf  TRISA    ; adr. 0x85 (0x05 na stránce 1)
bcf  STATUS,5   ; přechod na stránku 0
```

Nastaveni portu A jako výstupy

Přepínání stránek registrů

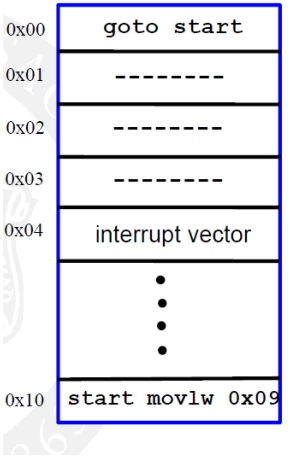
```
CLRF    STATUS          ; Clear STATUS register (Bank0)
:                       ;
BSF     STATUS, RP0     ; Bank1
:                       ;
BCF     STATUS, RP0     ; Bank0
:                       ;
MOVLW   0x60            ; Set RP0 and RP1 in STATUS register, other
XORWF   STATUS, F       ;   bits unchanged (Bank3)
:                       ;
BCF     STATUS, RP0     ; Bank2
:                       ;
BCF     STATUS, RP1     ; Bank0
```

Makro pro přepínání stránek registrů

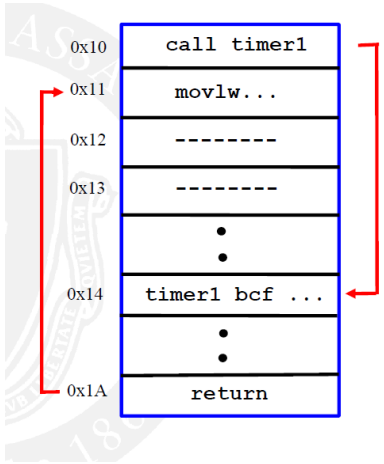
```
ifndef __banks
        define __banks 4
endif

macro banksel reg
        if reg & 0x80
                bsf 3, 5
        else
                bcf 3, 5
        endif
        if __banks > 2
                if reg & 0x100
                        bsf 3, 6
                else
                        bcf 3, 6
                endif
        endif
endm

macro bankisel reg
        if reg & 0x100
                bsf 3, 7
        else
                bcf 3, 7
        endif
endm
```

Resetovací sekvence

```
        ORG 0x00
        GOTO Start
        ORG 0x10
Start:  movlw 0x09
```



Volání podprogramů



Volání podprogramu v jiné stránce

```
            ORG 0x500
            BCF PCLATH,4
            BSF PCLATH,3    ;Select page 1
                            ;(800h-FFFh)
            CALL SUB1_P1    ;Call subroutine in
            :               ;page 1 (800h-FFFh)
            :
            ORG 0x900       ;page 1 (800h-FFFh)
SUB1_P1
            :               ;called subroutine
                            ;page 1 (800h-FFFh)
            :
            RETURN          ;return to
                            ;Call subroutine
                            ;in page 0
                            ;(000h-7FFh)
```

Přerušení

- Mnoho zdrojů přerušení
- INTCON registr
  - Bit 7 (GIE) povoluje a zakazuje přerušení
  - GIE je vynulován, když začne obsluha přerušení
  - GIE je nastaven, když je použita instrukce **retfie**
- Adresa 0x04 obsahuje přerušovací vektor
  - Přerušovací podprogram (ISR – Interrupt Service Routine) začíná na adrese 0x04
  - Uloží se hodnota PC do zásobníku
  - ISR musí uložit a poté obnovit hodnoty registrů STATUS a W
  - ISR si musí zjistit typ přerušení (kdo vyvolal přerušení

INTCON registr 0x0b,0x8b,0x10b,0x18b

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GIE	PEIE (3)	T0IE	INTE (2)	RBIE (1, 2)	T0IF	INTF (2)	RBIF (1, 2)
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **RBIE** (1): RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

INTCON registr (pokračování)

bit 2 **T0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred (must be cleared in software)
0 = The INT external interrupt did not occur

bit 0 **RBIF (1):** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Legend		
R = Readable bit	W = Writable bit	
U = Unimplemented bit, read as '0'		- n = Value at POR reset

PIE1 registr 0x8C

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit(1)
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt
Note 1: PSPIE is reserved on PIC16F873A/876A devices; always maintain this bit clear.

bit 6 **ADIE**: A/D Converter Interrupt Enable bit
1 = Enables the A/D converter interrupt
0 = Disables the A/D converter interrupt

bit 5 **RCIE**: USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit 4 **TXIE**: USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

PIE1 registr pokračování

bit 3 **SSPIE**: Synchronous Serial Port Interrupt Enable bit
1 = Enables the SSP interrupt
0 = Disables the SSP interrupt

bit 2 **CCP1IE**: CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE**: TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE**: TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

PIR1 registr 0x0C

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **PSPIF:** Parallel Slave Port Read/Write Interrupt Flag bit(1)
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred
Note 1: PSPIF is reserved on PIC16F873A/876A devices; always maintain this bit clear.

bit 6 **ADIF**: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed
0 = The A/D conversion is not complete

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer is full
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer is empty
0 = The USART transmit buffer is full

PIR1 registr pokračování

bit 3 **SSPIF**: Synchronous Serial Port (SSP) Interrupt Flag bit
1 = The SSP interrupt condition has occurred and must be cleared in software before returning from the Interrupt Service Routine. The conditions that will set this bit are:
- SPI – A transmission/reception has taken place.
- I^2C Slave – A transmission/reception has taken place.
- I^2C Master
  - A transmission/reception has taken place.
  - The initiated Start condition was completed by the SSP module.
  - The initiated Stop condition was completed by the SSP module.
  - The initiated Restart condition was completed by the SSP module.
  - The initiated Acknowledge condition was completed by the SSP module.
  - A Start condition occurred while the SSP module was Idle (multi-master system).
  - A Stop condition occurred while the SSP module was Idle (multi-master system).

0 = No SSP interrupt condition has occurred

bit 2 **CCP1IF**: CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode.

bit 1 **TMR2IF**: TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF**: TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

PIE2 registr 0x8D

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

bit 6 **CMIE**: Comparator Interrupt Enable bit
1 = Enables the comparator interrupt
0 = Disable the comparator interrupt

bit 5 **Unimplemented:** Read as '0'

bit 4 **EEIE**: EEPROM Write Operation Interrupt Enable bit
1 = Enable EEPROM write interrupt
0 = Disable EEPROM write interrupt

bit 3 **BCLIE**: Bus Collision Interrupt Enable bit
1 = Enable bus collision interrupt
0 = Disable bus collision interrupt

bit 2-1 **Unimplemented:** Read as '0'

bit 0 **CCP2IE**: CCP2 Interrupt Enable bit
1 = Enables the CCP2 interrupt
0 = Disables the CCP2 interrupt

PIR2 registr 0x0D

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

bit 6 **CMIF**: Comparator Interrupt Flag bit
1 = The comparator input has changed (must be cleared in software)
0 = The comparator input has not changed

bit 5 **Unimplemented:** Read as '0'

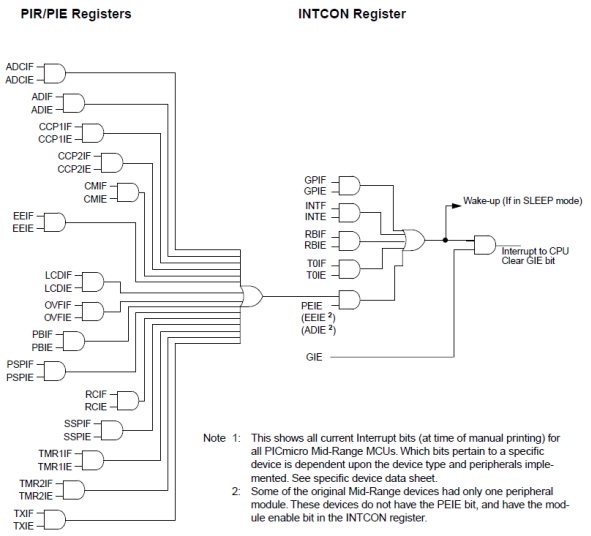
bit 4 **EEIF**: EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **BCLIF**: Bus Collision Interrupt Flag bit
1 = A bus collision has occurred in the SSP when configured for I^2C Master mode
0 = No bus collision has occurred

bit 2-1 **Unimplemented:** Read as '0'

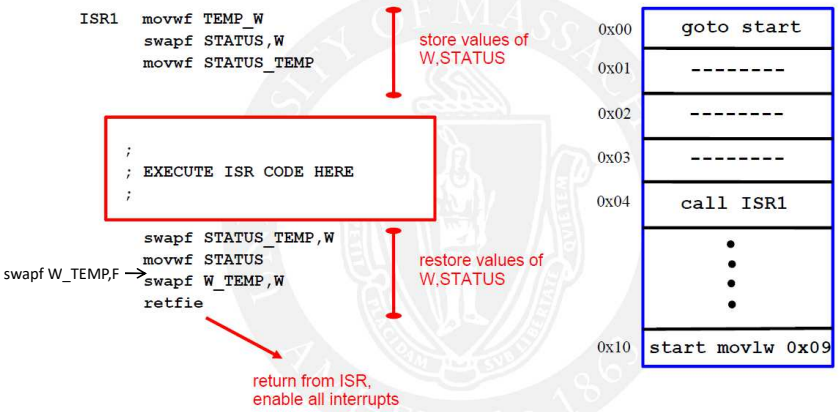
bit 0 **CCP2IF**: CCP2 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused.

Přerušovací logika



Note 1: This shows all current Interrupt bits (at time of manual printing) for all PICmicro Mid-Range MCUs. Which bits pertain to a specific device is dependent upon the device type and peripherals implemented. See specific device data sheet.
2: Some of the original Mid-Range devices had only one peripheral module. These devices do not have the PEIE bit, and have the module enable bit in the INTCON register.

Přerušovací podprogram (ISR)



Makra pro uložení kontextu

```
PUSH_MACRO    MACRO           ; This Macro Saves register contents
      MOVWF   W_TEMP          ; Copy W to a Temporary Register
                              ;    regardless of current bank
      SWAPF   STATUS,W        ; Swap STATUS nibbles and place
                              ;    into W register
      MOVWF   STATUS_TEMP     ; Save STATUS to a Temporary register
                              ;    in Bank0
      ENDM                    ; End this Macro
;
POP_MACRO     MACRO           ; This Macro Restores register contents
      SWAPF   STATUS_TEMP,W   ; Swap original STATUS register value
                              ;    into W (restores original bank)
      MOVWF   STATUS          ; Restore STATUS register from
                              ;    W register
      SWAPF   W_TEMP,F        ; Swap W_Temp nibbles and return
                              ;    value to W_Temp
      SWAPF   W_TEMP,W        ; Swap W_Temp to W to restore original
                              ;    W value without affecting STATUS
      ENDM                    ; End this Macro
```

Inicializace a povolení přerušení

```
PIE1_MASK1    EQU B'01101010'   ; This is the Interrupt Enable
      :                         ;    Register mask value
      :
      CLRF    STATUS          ; Bank0
      CLRF    INTCON          ; Disable interrupts and clear some flags
      CLRF    PIR1            ; Clear all flag bits
      BSF     STATUS, RP0     ; Bank1
      MOVLW   PIE1_MASK1      ; This is the initial masking for PIE1
      MOVWF   PIE1            ;
      BCF     STATUS, RP0     ; Bank0
      BSF     INTCON, GIE     ; Enable Interrupts
```

Zjištění zdroje přerušení v ISR (vzor)

```
      org  ISR_ADDR           ;
        PUSH_MACRO            ; MACRO that saves required context registers,
                              ;   or in-line code
        CLRF   STATUS         ; Bank0
        BTFSC  PIR1, TMR1IF   ; Timer1 overflow interrupt?
        GOTO   T1_INT         ; YES
        BTFSC  PIR1, ADIF     ; NO, A/D interrupt?
        GOTO   AD_INT         ; YES, do A/D thing
        :                     ; NO, do this for all sources
        :                     ;
        BTFSC  PIR1, LCDIF    ; NO, LCD interrupt
        GOTO   LCD_INT        ; YES, do LCD thing
        BTFSC  INTCON, RBIF   ; NO, Change on PORTB interrupt?
        GOTO   PORTB_INT      ; YES, Do PortB Change thing
INT_ERROR_LP1                 ; NO, do error recovery
        GOTO   INT_ERROR_LP1  ; This is the trap if you enter the ISR
                              ;    but there were no expected
                              ;    interrupts
T1_INT                        ; Routine when the Timer1 overflows
        :                     ;
        BCF     PIR1, TMR1IF  ; Clear the Timer1 overflow interrupt flag
        GOTO    END_ISR       ; Ready to leave ISR (for this request)
AD_INT                        ; Routine when the A/D completes
        :                     ;
        BCF     PIR1, ADIF    ; Clear the A/D interrupt flag
        GOTO    END_ISR       ; Ready to leave ISR (for this request)
LCD_INT                       ; Routine when the LCD Frame begins
        :                     ;
        BCF     PIR1, LCDIF   ; Clear the LCD interrupt flag
        GOTO    END_ISR       ; Ready to leave ISR (for this request)
PORTB_INT                     ; Routine when PortB has a change
        :                     ;
END_ISR                       ;
          POP_MACRO           ; MACRO that restores required registers,
                              ;   or in-line code
          RETFIE              ; Return and enable interrupts
```

Mapa registrů

- 4 stránky registrů
- Význačné registry – dostupné ve všech stránkách
  - STATUS
  - INDF
  - FSR
  - INTCON
  - PCL
  - PCLATH
- Bity 5 a 6 STATUSu se používají pro určení stránky registrů

Speciální funkční registry

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 0											
00h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
01h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
02h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
03h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
04h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
05h	PORTA	—	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	43, 150
06h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
07h	PORTC	PORTC Data Latch when written: PORTC pins when read								xxxx xxxx	47, 150
08h(4)	PORTD	PORTD Data Latch when written: PORTD pins when read								xxxx xxxx	48, 150
09h(4)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	49, 150
0Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
0Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
0Ch	PIR1	PSPIF(3)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	26, 150
0Dh	PIR2	—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF	-0-0 0--0	28, 150
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	60, 150
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{T1SYNC}$	TMR1CS	TMR1ON	--00 0000	57, 150
11h	TMR2	Timer2 Module Register								0000 0000	62, 150
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	61, 150
13h	SSPBUF	Synchronous Serial Port Receive Buffer/Transmit Register								xxxx xxxx	79, 150
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000	82, 82, 150
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	63, 150
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	63, 150
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	64, 150
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	112, 150
19h	TXREG	USART Transmit Data Register								0000 0000	118, 150
1Ah	RCREG	USART Receive Data Register								0000 0000	118, 150
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	63, 150
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	63, 150
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	64, 150
1Eh	ADRESH	A/D Result Register High Byte								xxxx xxxx	133, 150
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{DONE}$	—	ADON	0000 00-0	127, 150

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 1											
80h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
82h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
83h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
84h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	43, 150
86h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
87h	TRISC	PORTC Data Direction Register								1111 1111	47, 150
88h(4)	TRISD	PORTD Data Direction Register								1111 1111	48, 151
89h(4)	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	50, 151
8Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
8Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
8Ch	PIE1	PSPIE(2)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	25, 151
8Dh	PIE2	—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE	-0-0 0--0	27, 151
8Eh	PCON	—	—	—	—	—	—	$\overline{POR}$	$\overline{BOR}$	---- --qq	29, 151
8Fh	—	Unimplemented								—	—
90h	—	Unimplemented								—	—
91h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000	83, 151
92h	PR2	Timer2 Period Register								1111 1111	62, 151
93h	SSPADD	Synchronous Serial Port (I^2C mode) Address Register								0000 0000	79, 151
94h	SSPSTAT	SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF	0000 0000	79, 151
95h	—	Unimplemented								—	—
96h	—	Unimplemented								—	—
97h	—	Unimplemented								—	—
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	111, 151
99h	SPBRG	Baud Rate Generator Register								0000 0000	113, 151
9Ah	—	Unimplemented								—	—
9Bh	—	Unimplemented								—	—
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111	135, 151
9Dh	CVRCON	CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0	000- 0000	141, 151
9Eh	ADRESL	A/D Result Register Low Byte								xxxx xxxx	133, 151
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	128, 151

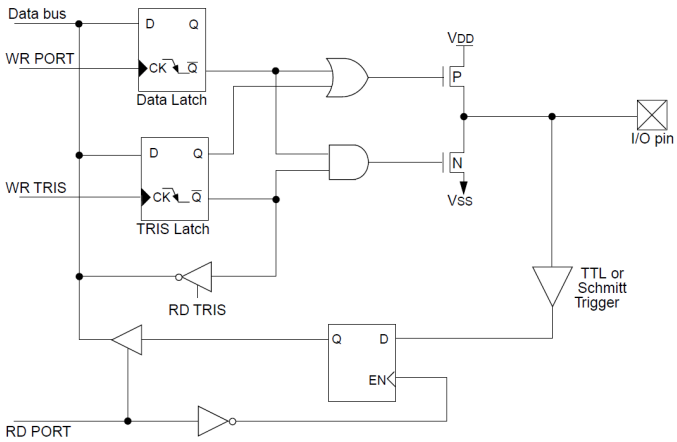
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Details on page:
Bank 2											
100h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
101h	TMR0	Timer0 Module Register								xxxx xxxx	55, 150
102h(3)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	30, 150
103h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
104h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
105h	—	Unimplemented								—	—
106h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	45, 150
107h	—	Unimplemented								—	—
108h	—	Unimplemented								—	—
109h	—	Unimplemented								—	—
10Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
10Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
10Ch	EEDATA	EEPROM Data Register Low Byte								xxxx xxxx	39, 151
10Dh	EEADR	EEPROM Address Register Low Byte								xxxx xxxx	39, 151
10Eh	EEDATH	—	—	EEPROM Data Register High Byte						--xx xxxx	39, 151
10Fh	EEADRH	—	—	—	—(5)	EEPROM Address Register High Byte				---- xxxx	39, 151
Bank 3											
180h(3)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	31, 150
181h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	23, 150
182h(3)	PCL	Program Counter (PC) Least Significant Byte								0000 0000	30, 150
183h(3)	STATUS	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	22, 150
184h(3)	FSR	Indirect Data Memory Address Pointer								xxxx xxxx	31, 150
185h	—	Unimplemented								—	—
186h	TRISB	PORTB Data Direction Register								1111 1111	45, 150
187h	—	Unimplemented								—	—
188h	—	Unimplemented								—	—
189h	—	Unimplemented								—	—
18Ah(1,3)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	30, 150
18Bh(3)	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	24, 150
18Ch	EECON1	EEPGD	—	—	—	WRERR	WREN	WR	RD	x--- x000	34, 151
18Dh	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	39, 151
18Eh	—	Reserved; maintain clear								0000 0000	—
18Fh	—	Reserved; maintain clear								0000 0000	—

Paralelní porty

- Pokud nejsou vývody (některé) použity pro jiné účely, mohou se použít jako vstupní či výstupní bity
- Podle typu mikropočítače mohou být k dispozici porty A, B, C, ….
- Orientace vývodu se určuje nastavením příslušného bitu v registru TRISx (x=A,B,C,…)

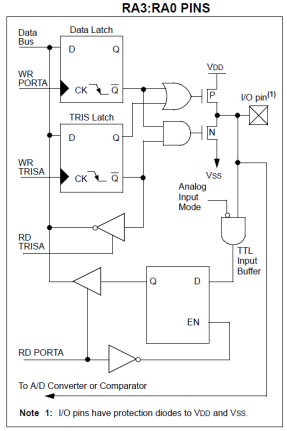
  1=vstup, 0= výstup
- Data pro výstup (vstup) se zapisují (čtou) do registru PORTx (x= A,B,C,…)

Zapojení portů - princip

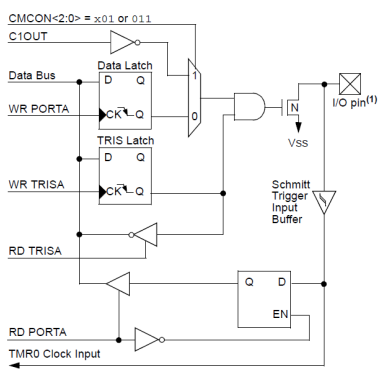


Zapojení portu A

FIGURE 4-1: BLOCK DIAGRAM OF RA3:RA0 PINS



BLOCK DIAGRAM OF RA4/T0CKI PIN



Port A

TABLE 4-1: PORTA FUNCTIONS

Name	Bit#	Buffer	Function
RA0/AN0	bit 0	TTL	Input/output or analog input.
RA1/AN1	bit 1	TTL	Input/output or analog input.
RA2/AN2/VREF-/CVREF	bit 2	TTL	Input/output or analog input or VREF- or CVREF.
RA3/AN3/VREF+	bit 3	TTL	Input/output or analog input or VREF+.
RA4/T0CKI/C1OUT	bit 4	ST	Input/output or external clock input for Timer0 or comparator output. Output is open-drain type.
RA5/AN4/SS/C2OUT	bit 5	TTL	Input/output or analog input or slave select input for synchronous serial port or comparator output.

Legend: TTL = TTL input, ST = Schmitt Trigger input

TABLE 4-2: SUMMARY OF REGISTERS ASSOCIATED WITH PORTA

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
05h	PORTA	—	—	RA5	RA4	RA3	RA2	RA1	RA0	--0x 0000	--0u 0000
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	--11 1111
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111	0000 0111
9Dh	CVRCON	CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0	000- 0000	000- 0000
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	00-- 0000

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by PORTA.

ADCON0 registr 0x1F

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	—	ADON
bit 7							bit 0

bit 7-6 **ADCS1:ADCS0:** A/D Conversion Clock Select bits (ADCON0 bits in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	**00**	Fosc/2
0	**01**	Fosc/8
0	**10**	Fosc/32
0	**11**	FRC (clock derived from the internal A/D RC oscillator)
1	**00**	Fosc/4
1	**01**	Fosc/16
1	**10**	Fosc/64
1	**11**	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits
000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

Note: The PIC16F873A/876A devices only implement A/D channels 0 through 4; the unimplemented selections are reserved. Do not select any unimplemented channels with these devices.

bit 2 **GO/DONE:** A/D Conversion Status bit
When ADON = 1:
1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)
0 = A/D conversion not in progress

bit 1 **Unimplemented:** Read as '0'

bit 0 **ADON:** A/D On bit
1 = A/D converter module is powered up
0 = A/D converter module is shut-off and consumes no operating current

ADCON1 registr 0x9F

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.
0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-4 **Unimplemented:** Read as '0'

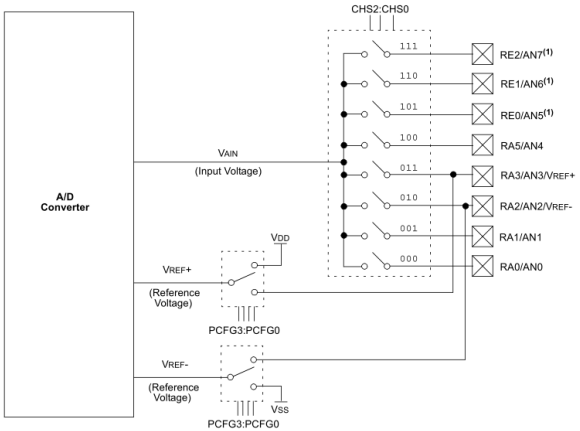
ADCON1 registr pokračování
Určení funkce bitu – analog/digital

bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

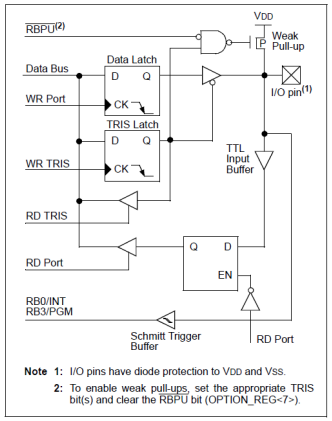
A/D blokové schéma



ADRESH a ADRESL registry (adresy 0x1E a 0x9E) obsahují 10bitové číslo výsledku A/D převodu. Po ukončení převodu se vynuluje bit ADCON0<2> a nastaví bit ADIF

Zapojení portu B

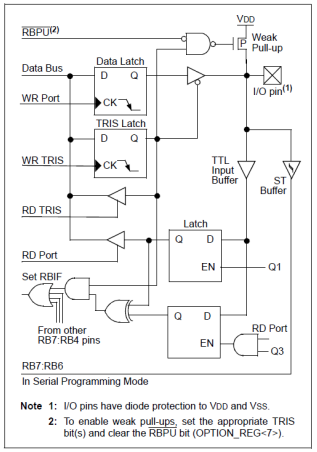
FIGURE 4-4: BLOCK DIAGRAM OF RB3:RB0 PINS



Note 1: I/O pins have diode protection to VDD and VSS.

2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RBPU bit (OPTION_REG<7>).

FIGURE 4-5: BLOCK DIAGRAM OF RB7:RB4 PINS



Note 1: I/O pins have diode protection to VDD and VSS.

2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RBPU bit (OPTION_REG<7>).

Port B

TABLE 4-4: SUMMARY OF REGISTERS ASSOCIATED WITH PORTB

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
06h, 106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuuu uuuu
86h, 186h	TRISB	PORTB Data Direction Register								1111 1111	1111 1111
81h, 181h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged. Shaded cells are not used by PORTB.

Port B

TABLE 4-3: PORTB FUNCTIONS

Name	Bit#	Buffer	Function
RB0/INT	bit 0	TTL/ST(1)	Input/output pin or external interrupt input. Internal software programmable weak pull-up.
RB1	bit 1	TTL	Input/output pin. Internal software programmable weak pull-up.
RB2	bit 2	TTL	Input/output pin. Internal software programmable weak pull-up.
RB3/PGM(3)	bit 3	TTL	Input/output pin or programming pin in LVP mode. Internal software programmable weak pull-up.
RB4	bit 4	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB5	bit 5	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB6/PGC	bit 6	TTL/ST(2)	Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming clock.
RB7/PGD	bit 7	TTL/ST(2)	Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming data.

Legend: TTL = TTL input, ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.

2: This buffer is a Schmitt Trigger input when used in Serial Programming mode or in-circuit debugger.

3: Low-Voltage ICSP Programming (LVP) is enabled by default which disables the RB3 I/O function. LVP must be disabled to enable RB3 as an I/O pin and allow maximum compatibility to the other 28-pin and 40-pin mid-range devices.

Příklad 1.

wire_16.asm

```
;*******************************************************************************
;
;                          PVK40 Example : Wire
;
; Processor : PIC16F877A
; Clock : XT 3.2768 MHz
;
; On your PVK40 board set DIP switches according to following way:
; 1) Switch off all DIP switches except:
; 2) Switch on OSC 3.276M on S9
; 3) Switch on B3 LED on S11
;*******************************************************************************
; Assembler directives :
        list    p = PIC16F877A  ; processor type
        __config 0x3F71         ; configuration setting
;-------------------------------------------------------------------------------
; Symbol definition :
status  equ     0x03            ;status is on the 0x03 address
portb   equ     0x06
trisb   equ     0x06            ;direct addressing
portd   equ     0x08
trisd   equ     0x08            ;direct addressing
;
#define PB      portd,0         ;pushbutton 0 is on the RD0 pin
#define LED     portb,3         ;LED is on the RB3 pin
#define RP0     status,5        ;RP0 is bit 5 in status register
;-------------------------------------------------------------------------------
        org     0               ;program starts at address 0x000
        bsf     RP0             ;bank 1 in RAM memory
        movlw   B'11110111'
        movwf   trisb           ;pin RB3 is output
        movlw   B'11111111'
        movwf   trisd           ;portd pins are inputs
        bcf     RP0             ;bank 0 in RAM memory
;
Main:   btfss   PB              ;is PB 0 or 1?
        goto    Main_A          ;if PB=0, jump to main_A
        bcf     LED             ;PB=1, LED off
        goto    Main            ;closes the loop
Main_A: bsf     LED             ;LED on
        goto    Main            ;closes the loop
;*******************************************************************************
; end of PVK40 Example : Wire
        end
```

Příklad 2.

blink_16.asm

```
;*****************************************************************************
;
;                           PVK40 Example : Blink
;
; Processor : PIC16F877A
; Clock : XT 3.2768 MHz
;
; On your PVK40 board set DIP switches according to following way:
; 1) Switch off all DIP switches except:
; 2) Switch on OSC 3.276M on S9
; 3) Switch on B3 LED on S11
;*****************************************************************************
; Assembler directives :
        list    p = PIC16F877A  ; processor type
        __config 0x3F71         ; configuration setting
;-----------------------------------------------------------------------------
; Special Function Register definition :
status  equ     0x03            ;status is on the 0x03 address
portb   equ     0x06
trisb   equ     0x06            ;direct addressing
; General Purpose Register definition :
cnt1    equ     0x20            ;used for Wait subroutine
cnt2    equ     0x21            ;used for Wait subroutine
cnt3    equ     0x22            ;used for Wait subroutine
; Destination definition :
w       equ     0x00
f       equ     0x01
; Bits definition :
#define LED     portb,3         ;LED is on the RB3 pin
#define RP0     status,5        ;RP0 is bit 5 in status register
;-----------------------------------------------------------------------------
        org     0               ;program starts at address 0x000
        bsf     RP0             ;bank 1 in RAM memory
        movlw   B'11110111'
        movwf   trisb           ;pin RB3 is output
        bcf     RP0             ;bank 0 in RAM memory
;
Main:   bsf     LED             ;LED on
        call    Wait            ;wait 0.5 second
        bcf     LED             ;LED off
        call    Wait            ;wait 0.5 second
        goto    Main            ;closes the loop
```

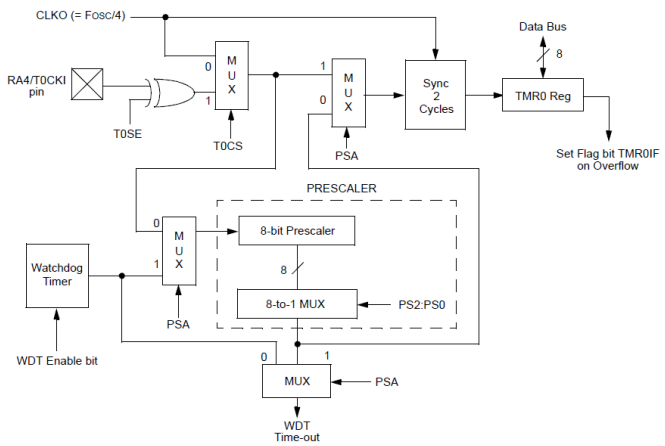
Příklad 2. pokračování

```
;-----------------------------------------------------------------------------
Wait:   movlw   0x05            ;this subroutine wait 0.5 second
        movwf   cnt3            ;1 cycle = 1/(Fosc/4) second =>
Wait_A: movlw   0x6B            ;=> we need 409600 cycles
        movwf   cnt2            ;0x05*0x6B*0xFF*3 = 409275
Wait_B: movlw   0xFF
        movwf   cnt1
Wait_C: decfsz  cnt1,f          ;decrement cnt1
        goto    Wait_C
        decfsz  cnt2,f          ;and if cnt1=0 then decrement cnt2
        goto    Wait_B
        decfsz  cnt3,f          ;and if cnt2=0 then decrement cnt3
        goto    Wait_A
        return                  ;if cnt3=0 then return
;*****************************************************************************
; end of PVK40 Example : Blink
        end
```

Časovač 0

- 8-bitový časovač/čítač
- Umožňuje čtení i psaní
- Má k dispozici 8-bit předdělička
- Volba externích nebo interních hodin
- Volba aktivní hrany pro externí hodiny
- Generuje přerušení při přetečení z FFh do 00h

Blokový diagram předděličky WDT a časovače 0



Časovač 0

TABLE 5-1: REGISTERS ASSOCIATED WITH TIMER0

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
01h,101h	TMR0	Timer0 Module Register								xxxx xxxx	uuuu uuuu
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
81h,181h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by Timer0.

OPTION registr 0x81,0x181

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **RBPU:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG:** Interrupt Edge Select bit
1 = Interrupt on rising edge of INT pin
0 = Interrupt on falling edge of INT pin

bit 5 **T0CS:** TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE:** TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA:** Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

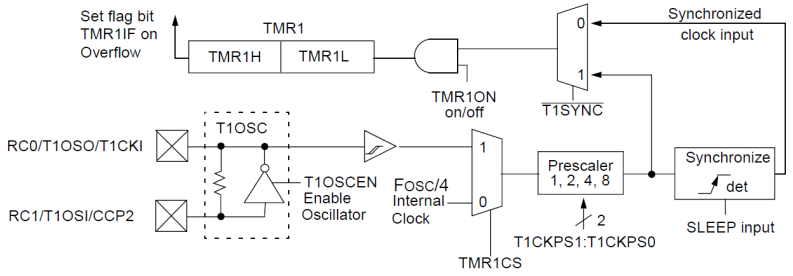
bit 2-0 **PS2:PS0:** Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Časovač 1

- Časovač 1 je 16-bitový časovač/čítač skládající se ze dvou 8-bitových registrů TMR1H a TMR1L, které mohou být čteny či zapisovány
- Čítač čítá od 0h do FFFFh a poté přeteče do 0h
- Přerušení se generuje (pokud je povoleno) při přetečení do 0h
- V módu časovače je inkrementován každým instrukčním cyklem
- V módu čítače je inkrementován náběžnou hranou externího hodinového impulsu

Blokové schéma časovače 1



Časovač 1

TABLE 6-2: REGISTERS ASSOCIATED WITH TIMER1 AS A TIMER/COUNTER

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	--00 0000	--uu uuuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer1 module.
Note 1: Bits PSPIE and PSPIF are reserved on the 28-pin devices; always maintain these bits clear.

Časovač 1

REGISTER 6-1: T1CON: TIMER1 CONTROL REGISTER (ADDRESS 10h)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **T1CKPS1:T1CKPS0:** Timer1 Input Clock Prescale Select bits
11 = 1:8 prescale value
10 = 1:4 prescale value
01 = 1:2 prescale value
00 = 1:1 prescale value

bit 3 **T1OSCEN:** Timer1 Oscillator Enable Control bit
1 = Oscillator is enabled
0 = Oscillator is shut-off (the oscillator inverter is turned off to eliminate power drain)

bit 2 **$\overline{\text{T1SYNC}}$:** Timer1 External Clock Input Synchronization Control bit
When TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
When TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS:** Timer1 Clock Source Select bit
1 = External clock from pin RC0/T1OSO/T1CKI (on the rising edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR1ON:** Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

Nastavení předděličky časovače 1

	Frequency (kHz)		
Prescale	**32.768**	**100**	**200**
1	2	0.655	0.327
2	4	1.31	0.655
4	8	2.62	1.31
8	16	5.24	2.62

TMR1H Load Value	Overflow Time (@ 32.768 kHz)
80h	1 Second
C0h	0.5 Second
E0h	0.25 Second
F0h	0.125 Second

Ukázka použití časovače 1 pro generování přerušení po 1 sekundě

```
      org   0x000
Reset_V     GOTO    START
;
      org   0x004
PER_INT_V
            BCF     STATUS, RP0     ; Bank0
            BTFSC   PIR1, TMR1IF    ; Timer1 overflowed?
            GOTO    T1_OVRFL        ; YES, Service the Timer1 Overflow Interrupt
;
; Should NEVER get here
;
ERROR1                              ; NO, Unknown Interrupt Source
            BSF     PORTD, 1        ; Toggle a port pin to indicate error
            BCF     PORTD, 1
            GOTO    ERROR1
;
T1_OVRFL
            BCF     PIR1, TMR1IF    ; Clear Timer1 Interrupt Flag
            MOVLW   0x80            ; Since doing key inputs, clear TMR1
            MOVWF   TMR1H           ;   for 1 sec overflow.
            :
            :                       ; Do Interrupt stuff here
            :
            RETFIE                  ; Return / Enable Global Interrupts
;
;
```

```
START                           ; POWER_ON Reset (Beginning of program)
            CLRF    STATUS      ; Do initialization (Bank0)
            BCF     T1CON, TMR1ON ; Timer1 is NOT incrementing
            :
            :                   ; Do Initialization stuff here
            :
            MOVLW   0x80        ; TIM1H:TMR1L = 0x8000 gives 1 second
            MOVWF   TMR1H       ;   overflow, at 32 KHz.
            CLRF    TMR1L       ;
;
            CLRF    INTCON
            CLRF    PIR1
            BSF     STATUS, RP0 ; Bank1
            CLRF    PIE1        ; Disable all peripheral interrupts
;
    if ( C74_REV_A )            ; See PIC16C74 Errata
            BSF     TRISC, T1OSO ; RC0 needs to be input for the oscillator to function
    endif
            BSF     PIE1, TMR1IE ; Enable TMR1 Interrupt
;
; Initialize the Special Function Registers (SFR) interrupts
;
            BCF     STATUS, RP0 ; Bank0
            CLRF    PIR1        ;
            BSF     INTCON, PEIE ; Enable Peripheral Interrupts
            BSF     INTCON, GIE ; Enable all Interrupts
;
            MOVLW   0x0E
            MOVWF   T1CON       ; Enable T1 Oscillator, Ext Clock, Async, prescaler = 1
            BSF     T1CON, TMR1ON ; Turn Timer1 ON
;
zzz         SLEEP
            GOTO    zzz         ; Sleep, wait for TMR1 interrupt
```

Konfigurační slovo CONFIG

- Slouží pro nastavení základních parametrů procesoru PIC.
- Není součástí programu
- Adresa konfiguračního slova u procesoru PIC16F877A je 2007H
- Nastavení typu oscilátoru, ladění, chránění paměti programu nebo dat v EEPROM proti zápisu, nastavení nulování procesoru a watchdog časovače, atd.

Konfigurační slovo 0x2007

R/P-1	U-0	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1
CP	—	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	—	—	$\overline{\text{PWRTEN}}$	WDTEN	Fosc1	Fosc0
bit 13													bit0

bit 13 **CP:** Flash Program Memory Code Protection bit
1 = Code protection off
0 = All program memory code-protected

bit 12 **Unimplemented:** Read as '1'

bit 11 **DEBUG:** In-Circuit Debugger Mode bit
1 = In-Circuit Debugger disabled, RB6 and RB7 are general purpose I/O pins
0 = In-Circuit Debugger enabled, RB6 and RB7 are dedicated to the debugger

bit 10-9 **WRT1:WRT0** Flash Program Memory Write Enable bits
For PIC16F876A/877A:
11 = Write protection off; all program memory may be written to by EECON control
10 = 0000h to 00FFh write-protected; 0100h to 1FFFh may be written to by EECON control
01 = 0000h to 07FFh write-protected; 0800h to 1FFFh may be written to by EECON control
00 = 0000h to 0FFFh write-protected; 1000h to 1FFFh may be written to by EECON control
For PIC16F873A/874A:
11 = Write protection off; all program memory may be written to by EECON control
10 = 0000h to 00FFh write-protected; 0100h to 0FFFh may be written to by EECON control
01 = 0000h to 03FFh write-protected; 0400h to 0FFFh may be written to by EECON control
00 = 0000h to 07FFh write-protected; 0800h to 0FFFh may be written to by EECON control

bit 8 **CPD:** Data EEPROM Memory Code Protection bit
1 = Data EEPROM code protection off
0 = Data EEPROM code-protected

bit 7 **LVP**: Low-Voltage (Single-Supply) In-Circuit Serial Programming Enable bit
1 = RB3/PGM pin has PGM function; low-voltage programming enabled
0 = RB3 is digital I/O, HV on $\overline{\text{MCLR}}$ must be used for programming

bit 6 **BOREN**: Brown-out Reset Enable bit
1 = BOR enabled
0 = BOR disabled

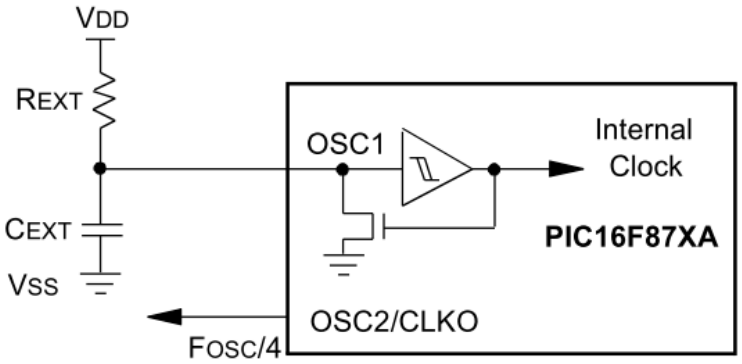
bit 5-4 **Unimplemented:** Read as '1'

bit 3 **$\overline{\text{PWRTEN}}$**: Power-up Timer Enable bit
1 = PWRT disabled
0 = PWRT enabled

bit 2 **WDTEN**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

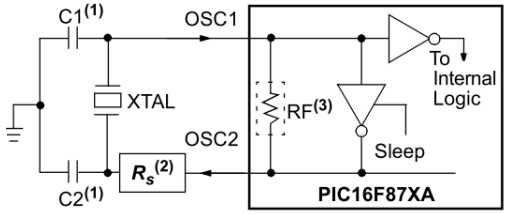
bit 1-0 **Fosc1:Fosc0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

RC oscilátor



Recommended values: 3 kΩ ≤ REXT ≤ 100 kΩ
CEXT > 20 pF

HS, XT a LT typy oscilátoru



Osc Type	Crystal Freq.	Cap. Range C1	Cap. Range C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF

Obsah souboru p16f877a.inc

```
;       Register Definitions

W                       EQU     H'0000'
F                       EQU     H'0001'

;----- Register Files------------------------------------------------------

INDF                    EQU     H'0000'
TMR0                    EQU     H'0001'
PCL                     EQU     H'0002'
STATUS                  EQU     H'0003'
FSR                     EQU     H'0004'
PORTA                   EQU     H'0005'
PORTB                   EQU     H'0006'
PORTC                   EQU     H'0007'
PORTD                   EQU     H'0008'
PORTE                   EQU     H'0009'
PCLATH                  EQU     H'000A'
INTCON                  EQU     H'000B'
PIR1                    EQU     H'000C'
PIR2                    EQU     H'000D'
atd .....
```

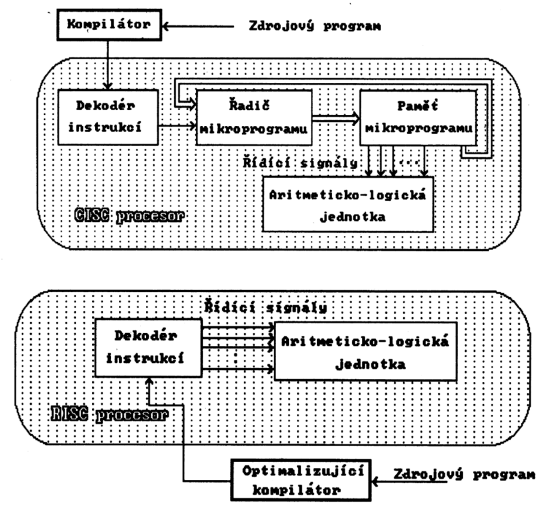
RISC procesory

- Procesory dělíme podle architektury na:
  - CISC – Complex Instruction Set Computers
  - RISC – Reduced Instruction Set Computers
- V současné době se tato ostrá hranice stírá a vznikají procesory, které si z každé architektury berou pozitivní věci

Vznik procesorů RISC

- Odstranění nevýhod CISC procesorů
  - Velmi složité řadiče (pro vykonávaní velmi složitých instrukcí) – obyčejně řešeny jako mikroprogramovatelné automaty
  - Z důvodů komplikovaného řadiče i pomalé vykonávání jednoduchých instrukcí
  - Analýzou vytvořených programů bylo zjištěno, že se většina složitějších instrukcí velmi málo používá (překladače vyšších programovacích jazyků obyčejně generují kód pomocí jednodušších instrukcí) – cca 10-15% instrukcí z instrukčního souboru se používá z více než 90% (většinou jednodušší) => zbytečné prodražení návrhu i implementace při nevyužívání technických prostředků

Architektury procesorů CISC a RISC



CISC

RISC

Zvýšení výkonu RISC procesorů

- Celkový čas T_c realizace určitého programu

$$T_c = T.\sum_{i=1}^{N} C_i$$

kde: N je počet vykonaných instrukcí procesoru ($N_{CISC} < N_{RISC}$)

C_i je počet hodinových taktů pro vykonání i-té instrukce (C_{CISC}=3- cca 350, C_{RISC}=1 nebo <1)

T je doba jednoho hodinového taktu ($T_{CISC} > T_{RISC}$)

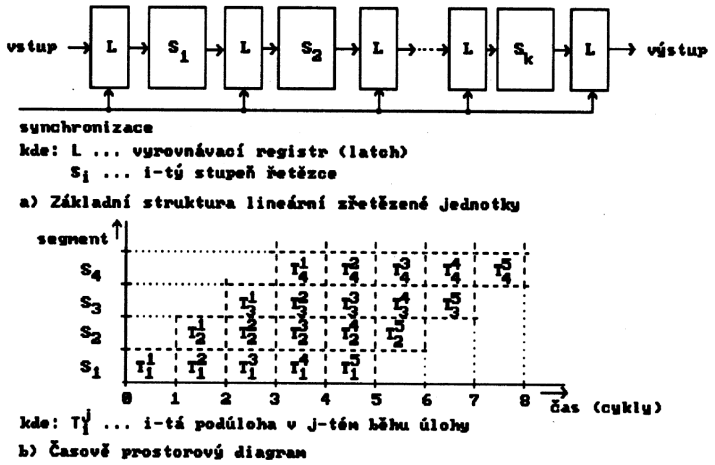
Charakteristické rysy architektury RISC

- Minimální instrukční soubor (cca 50 až 450 instrukcí)
- Jednoduché způsoby adresování
- Jeden nebo málo formátů instrukcí (rychlé dekódování)
- Řízení jednoduchou pevnou logikou
- Jednotaktové strojové instrukce (v každém taktu končí jedna instrukce)
- Styk s pamětí výlučně prostřednictvím Load/Store instrukcí (instrukce pracují přednostně nad lokálními registry mikroprocesoru – load/store architektura)
- Datové operace pouze nad registry
- Velký počet vnitřních registrů mikroprocesoru
- Paměť „cache“ mezi procesorem a hlavní pamětí
- Zřetězená realizace instrukcí
- Optimalizující kompilátor (společný vývoj HW a kompilátoru)

Zřetězené zpracování informace

- Analogie výrobní linky
  - Rozdělení úlohy na posloupnost dílčích úloh, z nichž každá může být vykonána samostatnými technickými prostředky (moduly)
  - Jednotlivé moduly pracují souběžně
  - Algoritmus se vykonává ve zřetězené jednotce složené z lineárního řetězce modulů
  - Každý modul vykonává určitou dílčí úlohu, přičemž funkčně závisí na předcházejícím modulu
  - V ideálním případě jsou jednotlivé dílčí úlohy vykonávany za stejný časový interval (v opačném případě se nejpomalejší modul stává „úzkým místem“)

Zřetězený lineární (skalární) procesor

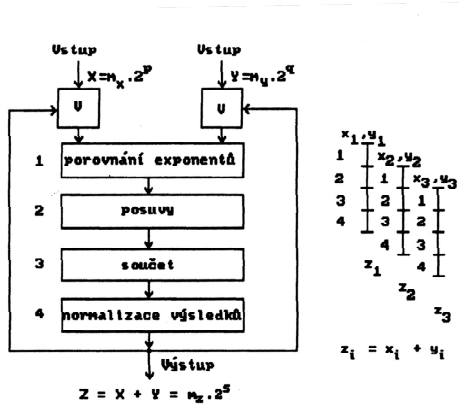


Zřetězené zpracování informace

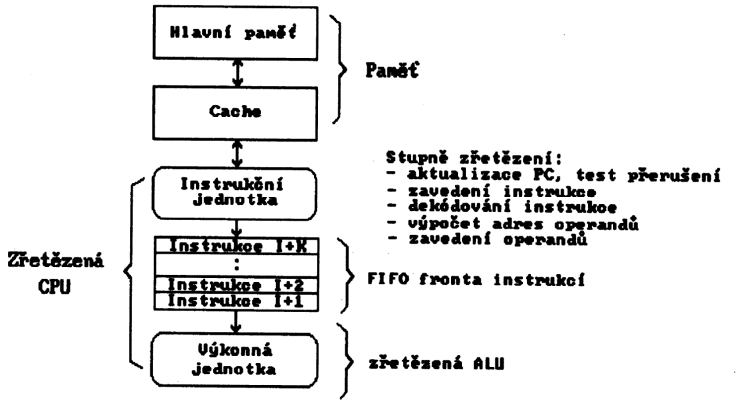
- Pro dosažení vysoké efektivnosti musí vstupní, resp. výstupní údaje algoritmu soustavně proudit z paměti do zřetězené jednotky, resp. ze zřetězené jednotky do paměti (analogie – *po vpuštění proudu vody do trubky začne voda na druhém konci vytékat až po naplnění trubky, každé přerušení proudu se projeví až z určitým zpožděním na výstupu*)
- V každém časovém bodě je na různé úrovni rozpracováno **x** instrukcí (mluvíme o tzv. **x**-úrovňovém zřetězení**)**
- V každém taktu je dokončena jedna instrukce

Typy zřetězení

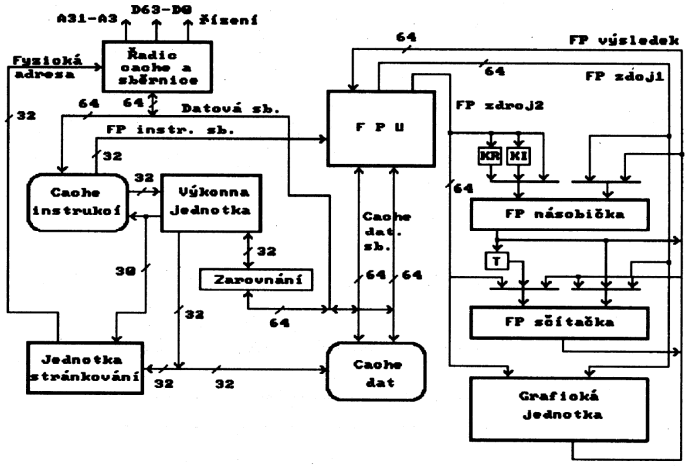
- **Aritmetické** – je dosaženo v ALU (např. pro výpočty v pohyblivé desetinné čárce - sčítání)



- **Zřetězení instrukcí** – vykonání instrukce je rozděleno na načtení z paměti,dekódování instrukce, výběr operandů, provedení operace, zápis výsledku



Blokové schéma procesoru Intel I860



Problémy zřetězení instrukcí

- Nejvyšší efektivity zřetězení instrukcí je dosaženo, když instrukce spojitě proudí do procesoru a jejich tok není narušován
- V reálných programech dochází k narušení spojitosti toku instrukcí a z toho vyplývajícího narušení zřetězeného zpracování
- Narušení zřetězeného zpracování znamená, že se poruší plynulá návaznost činnosti jednotlivých modulů řetězce a to jak z hlediska zpracování jedné instrukce, tak z hlediska zpracování celkového toku instrukcí.
- Při narušení průběhu zřetězeného zpracování je nutné některé operace zpozdit nebo je dokonce potřeba rozbíhat zřetězené zpracování opět od jeho začátku
- Každé narušení plynulosti vede ke ztrátě výkonu procesoru
- Existují dva druhy konfliktů – **datový konflikt a skokový konflikt**

Datový konflikt

- Datový konflikt vzniká v případě, kdy několik instrukcí v řetězci za sebou pracuje se stejným operandem – např. následující instrukce už vybírá data z registru, aniž byla dokončena jejich modifikace instrukcí předcházející.
- Datové konflikty jsou podstatně častější u tzv. superskalárních procesorů, u nichž dochází ke zpracování instrukcí tzv. **mimo pořadí – out of order** (vykonávání instrukcí není sekvenční dle programu, ale vykonání některých instrukcí se opožďuje)
- Existují tři druhy datových konfliktů:
  - Zápis po čtení (Write after Read –WAR)
  - Čtení po zápisu (Read after Write – RAW)
  - Zápis po zápisu (Write after Write – WAW)

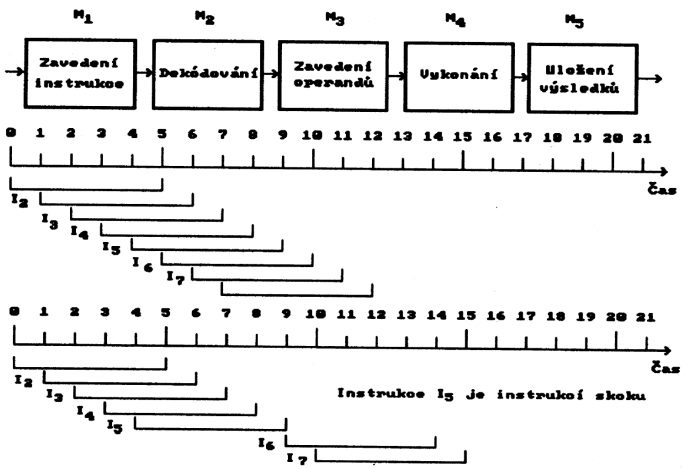
Řešení datových konfliktů

- Na úrovni optimalizujícího překladače – překladač zpřehází pořadí instrukcí (mezi instrukce pracující nad stejným registrem vloží instrukci , která s registrem nepracuje – oddálí krizové instrukce od sebe)
- Některé procesory nespoléhají na překladač a pokud detekují datový konflikt, řeší problém změnou pořadí zpracování instrukcí za běhu programu

Skokový konflikt

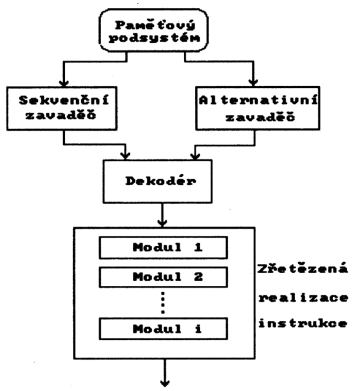
- Skokový konflikt může vzniknout na základě použití instrukce, která mění obsah čítače instrukcí (příští zpracovávaná instrukce není následující instrukce v programovém kódu)
- K těmto instrukcím patří instrukce nepodmíněného skoku, instrukce podmíněného skoku při splnění podmínky, instrukce volání podprogramu a také všechny druhy přerušení (volání přerušovacího podprogramu)

Vliv instrukce větvení na spojitost zřetězení instrukcí



Řešení skokového konfliktů

- Pomocí tzv. alternativního zavaděče – je-li dekódována instrukce větvení, je zahájena souběžná činnost alternativního zavaděče od nové adresy. Podle výsledku podmínky je vybrána odpovídající fronta instrukcí, přičemž druhá fronta je potlačena



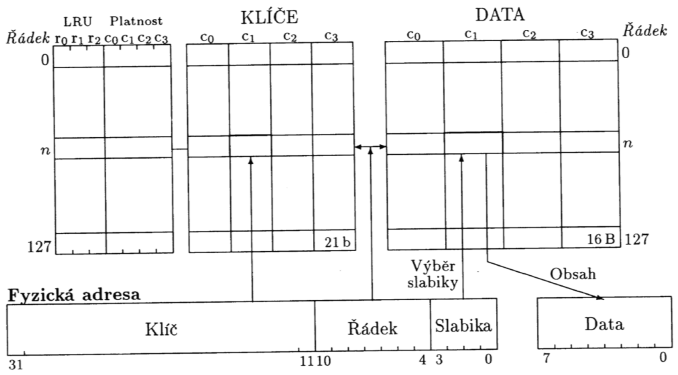
Metoda opožděného skoku

- Je-li ve zřetězeném procesoru detekována instrukce větvení, nedojde k její okamžité realizaci. Její vykonání je „zpožděno“ o dobu, která je nutná k dokončení instrukcí rozpracovaných **za** touto instrukcí
- Tyto instrukce nesmí stav procesoru oproti stavu, který by nastal bez použití techniky opožděného skoku
- Potom se pokračuje instrukcí na cílové adrese skoku nebo se skok potlačí a následuje další instrukce v sekvenčním pořadí
- Umístění „vhodných“ instrukcí za instrukci větvení (které nemění stav procesoru) je záležitostí optimalizujícího kompilátoru.
- Kompilátor se „podívá“ na instrukce před instrukci větvení a vhodné instrukce přehodí až za instrukci větvení (tyto se vždy provedou a skok nastane až po nich)
- Nejlepší metoda – nehodí se ale pro řešení konfliktů vznikajících z důvodů přerušení – zde se často rozpracované instrukce dokončí a teprve poté se provede skok do přerušovacího podprogramu => delší odezva na přerušení
- Nevhodné také pro tzv. superskalární procesory, neboť nenajdeme dostatek vhodných instrukcí

Vyrovnávací paměť (cache)

- Umístěna mezi procesor a hlavní paměť – velmi rychlá asociativní paměť
- Nutná z důvodu dodávání instrukce z hlavní paměti v každém hodinovém taktu (neboť v každém hodinovém taktu je dokončena jedna instrukce)
- Realizována velmi často jako x-cestná asociativní paměť (neboť plně asociativní paměť je velmi náročná na realizaci)
- V systému může být několik úrovní cache pamětí – úroveň (level) 1 v procesoru (tzv. L1), úroveň 2 na desce (tzv. L2), …
- Problémy s tzv. konzistencí dat

Blokové schéma cache paměti



Cache paměť procesoru I80486

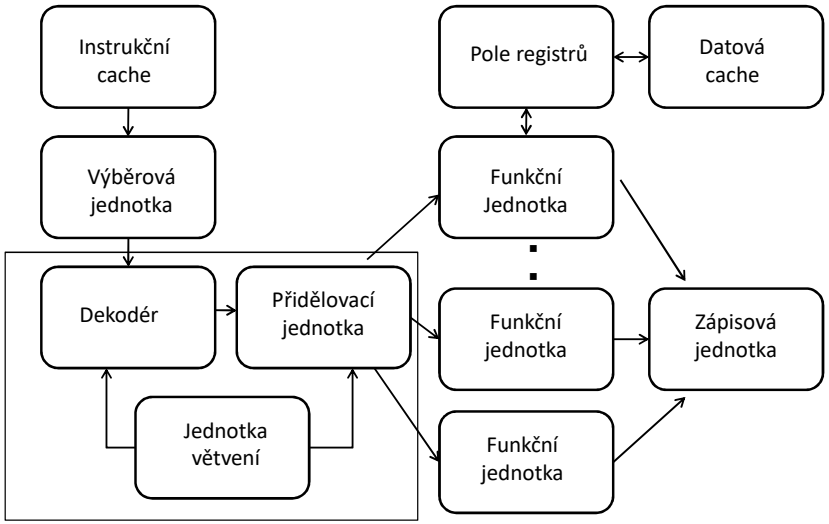
Konzistence dat

- Pokud jsou data na konkrétní adrese umístěna jak v hlavní paměti, tak v cache paměti, při modifikaci dat může vzniknout problém konzistence dat
- Data jsou zmodifikována (zapsána) buď jen v hlavní paměti nebo jen v cache paměti => rozdílnost (nekonzistence) dat na stejné adrese
- Řešení:
  - Současný zápis do hlavní i cache paměti – tzv. **Write-Through**
    - Výhoda – data vždy identická (konzistentní)
    - Nevýhoda – pomalé (musí se čekat na dokončení zápisu v hlavní paměti)
  - Zápis pouze do cache paměti, ale je nutný „úklid“ (přepis) do hlavní paměti při uvolňování položky v cache paměti – tzv. **Write-Back**
    - Výhoda – velmi rychlé
    - Nevýhoda – nekonzistence dat, nevhodné pro multiprocesorové systémy se sdílenou hlavní paměti

Superskalární procesory

- Procesory RISC s několika zřetězenými jednotkami – dokončí v jednom hodinovém cyklu více než jednu instrukci => podstatné zvýšení výkonu
- Umožnila to dokonalejší výrobní technologie, kdy bylo možné naintegrovat do procesoru více modulů pro paralelní činnost
- Přestože procesor najednou zpracovává několik instrukcí, musí být výsledek stejný, jakoby se instrukce prováděly postupně za sebou.

Blokové schéma superskalárního procesoru



Výběrová jednotka (Fetch Unit)

- Načítá z instrukční cache instrukce po n-ticích, kde n >1 se označuje jako šířka výběru (Fetch Width). U současných superskalárních procesorů je n=2-8
- Jelikož je většinou jednotná délka instrukcí, je paralelní výběr zjednodušen tím, že se vybírá z adresy dělitelné celkovým počtem vybíraných bajtů
- Načtené instrukce se předávají dekodéru

Dekodér (Decoder)

- Počet instrukcí, který je dekodér schopen dekódovat za 1 hodinový takt, se označuje jako šířka dekodéru (decoding width).
- Zpravidla je rovna šířce výběrové jednotky
- Některé procesory provádějí částečné dekódování již při výběru instrukce
- U superskalárních procesorů nelze u skokových instrukcí realizovat metodu opožděného skoku. Používá se technika spekulativního výpočtu založeného na predikci skoku

- Dekodér je složitější, vnitřně zřetězený, a na jeho výstupu se instrukce jednotlivých větví slévají do jediného sekvenčního toku, ve kterém se instrukce větvení nevyskytují.
- Instrukce větvení se provádějí již v samotném dekodéru, resp. Jeho speciální části, v tzv. jednotce větvení (branch unit).
- Jakmile jednotka dekodéru dekóduje instrukci větvení, předá ji jednotce větvení. Ta vyhodnotí adresní operandy a zajistí načtení z nové větve programu, aby do přidělovací jednotky plynule přicházely instrukce. Která větev se bude provádět se rozhodne podle predikce skoku (statické či dynamické)
- Instrukce z dekodéru jsou předávány přidělovací jednotce

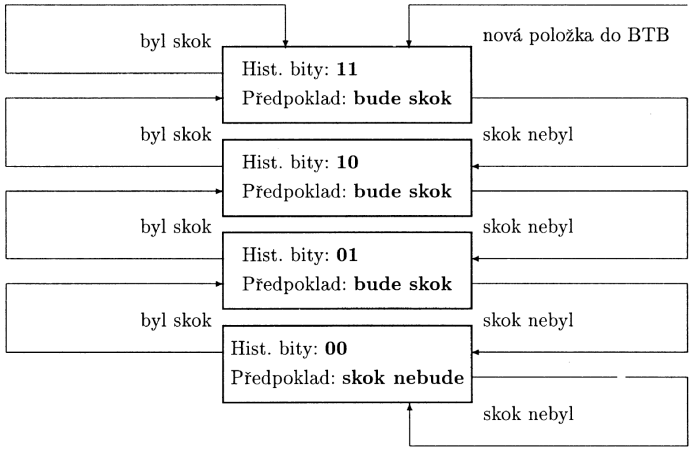
Predikce skoku

- Odhadne se, zda skok bude či nebude. Pokud odhad nebude správný a načte se nesprávná větev programu, musí se anulovat několik rozpracovaných instrukcí, tzv. skoková penalizace. Musí se provést tzv. vypláchnutí zřetězení (pipeline flushing).
- Proto je nutné odhadovat zda skok bude či nebude pokud možno s co největší pravděpodobností
- **Statická predikce** – rozhodování podle statické informace, např. podle směru skoku. Pokud je to skok zpět, je velká pravděpodobnost, že je to instrukce na konci cyklu a tedy že skok se uskuteční
- **Dynamická predikce** – vychází se nejen ze současného stavu, ale i z historie.
  - Tabulka posledních skoků – procesor vytváří tabulku cílových adres několika posledních skoků (obyčejně asociativní paměť) s historií skoků, zda byl či nebyl v minulosti na tuto adresu, případně také s několika instrukcemi dané větve.
  - Heuristická analýza několika předcházejících instrukcí může dát také informaci, zda skok bude či nikoli – expertní systém v procesoru

Tabulka skoků BTB u procesoru Pentium

- Mechanismus předvídání výsledku podmíněných skokových instrukcí.
- Algoritmus: vždy po provedení těla cyklu se podmíněnou skokovou instrukcí rozhoduje o jeho opětném provedení. S významně větší pravděpodobností nastává situace, že při zopakování téže podmíněné skokové instrukce bude její výsledek stejný (tj. tělo cyklu se bude opakovat vícekrát a jenom jednou se cyklus opustí).
- Implementace paměti adres skoků **BTB (Branch Target Buffer).**
- Do této paměti procesor ukládá cílové adresy ze skokových instrukcí (získaných fází Dl), které provedly skok, spolu s 2bitovým záznamem historie skoků.
- Při výběru instrukce se testuje obsah BTB na shodu s adresou vybírané instrukce. Pokud se adresa v BTB najde, zkoumá se obsah bitů historie. Předpokládá-li se skok, pokračuje se výběrem následujících instrukcí počínaje instrukcí, na kterou ukazuje operand skokové instrukce. Pokud se ve fázi provedení skokové instrukce zjistí, že předpověď byla špatná, musí se fronta předvybraných instrukcí vyprázdnit a výběr se zahajuje znovu ze správné adresy.
- Historické bity představují automat. Vždy po provedení skoku se bity aktualizují.
- Pouze v kombinaci 00 se předpokládá, že skok nenastane.

Graf přechodu historických bitů v BTB



Přidělovací jednotka, funkční jednotka (Dispatch Unit, Function Unit)

- Z dekodéru přichází data do přidělovací jednotky, která předává dekódované instrukce jednotlivým funkčním jednotkám
- Tyto jednotky jsou specializované na určitou třídu operací (mohou být opět zřetězené). Pro stejnou operaci muže být i více funkčních jednotek (logické operace, celočíselná ALU, FPU sčítání, násobení dělení, atd.)
- Může vzniknout nový typ konfliktu, tzv. **strukturální konflikt**. Nastává tehdy, když není požadovaná funkční jednotka volná. Instrukce musí počkat a tzv. se opozdit.
- Tedy vykonávání instrukcí v superskalárním procesoru nemusí být dle pořadí v programu, některé instrukce se mohou předbíhat či naopak zpožďovat (zpracování instrukcí **mimo pořadí – out of order**). Datové konflikty mohou být častější a nejdou řešit optimalizujícím kompilátorem (neboť vzniknou až za běhu programu).

Ošetření výjimek u superskalárních procesorů

- Při paralelním rozpracování více instrukcí najednou je těžké poznat, která výjimka nastala, protože chybná instrukce mohla být prováděna mimo pořadí a instrukce před ní a za ní v programu už mohly - ale ještě nemusely - být provedeny.
- Ošetření výjimek představuje podobný problém, jako zotavení z nesprávné spekulace při zpracování podmíněného skoku v jednotce větvení.
- Nejprve je třeba zjistit, která instrukce chybu způsobila. Potom je třeba dokončit všechny instrukce, které v sekvenčním programu chybnou instrukci logicky předcházejí. Tím se procesor dostane do tzv. **skalárního stavu**, který odpovídá sémantice sekvenčního programu. Pak je teprve možné výjimku/přerušeni ošetřit a pokračovat ve výpočtu. Takové řešeni se označuje termíny **přesné přerušeni (precise interrupt)** nebo **přesné ošetření výjimek (precise exception handling)**.
- Jsou cca tři možné přístupy:

1. Přesné přerušení není na úrovni hardwaru podporováno. Když některá instrukce způsobí přerušení, stav provádění instrukcí v jejím okolí (záleží na šířce dekódování) není zaručen. Přerušení pouze nastaví stavové příznaky, které se programově otestují, aby bylo možné zjistit stav a z chyby se zotavit. Tomu se říká **nepřesné přerušení**
2. Přesné přerušení je podporováno pouze ve skalárním modu procesoru, který některé procesory podporují. Ve skalárním módu vysílá přidělovací jednotka pouze 1 instrukci za 1 takt v pořadí daném sémantikou sekvenčního programu.
3. Přesné přerušení je implementováno hardwarově. To předpokládá implementaci mechanismu dokončení všech instrukcí, které chybné instrukci logicky předcházejí a zrušení účinku všech instrukcí, které v programu následují za chybnou instrukcí a které se již nacházejí v procesoru v různém stavu rozpracovanosti. Procesor tedy musí udržovat zpětně stavové informace do určité doby, odpovídající počtu taktů nutných pro úplné zpracování nejdelší instrukce, (tedy řádově 10 taktů) aby mohl anulovat změny způsobené instrukcemi, které by ve skalárním procesoru teprve čekaly na provedeni. Po uvedení procesoru do původního stavu může začít obsluha přerušení.

Přehled mikroprocesorů Intel

- 1968 založena firma Intel (Robert Noyce a Gordon Moore) - **Int**egrated **El**ectronics
- 1970 – mikroprocesor 4004 - 4bitová struktura, 2 250 tranzistorů MOS , rychlost 60 000 operací za sekundu. Adresoval 1 280 půlslabik dat a 4 KB instrukcí
- 1972 – první 8bitový mikroprocesor 8008 sestavený z 3 300 tranzistorů MOS a pracující rychlostí 30 000 operací za sekundu, paměť max. 16 KB.
- 1974 - mikroprocesor 8080 - 8bitový procesor, na čipu je 4 500 NMOS tranzistorů, rychlost 200 000 operací za sekundu. Instrukční repertoár je kompatibilní s typem 8008 a je rozšířen o 30 nových instrukcí. Adresovací kapacita je 64 KB. Později byly na trh uvedeny mírně vylepšené modifikace 8080: 8080A a 8085A.
- 1978 - první 16bitový mikroprocesor Intel 8086. Jde o první člen řady iAPX 86. Adresovací kapacita procesoru je 1 MB paměti. K procesoru 8086 byl vyprojektován pomocný specializovaný procesor pro určitý typ výpočtů, nazývaný koprocesor. Nejznámějším je koprocesor pro matematické operace v pohyblivé řádové čárce Intel 8087.
- 1979 - mikroprocesor Intel 8088, což je modifikace 8086. Liší se tím, že má vnější 8bitovou strukturu (datová sběrnice), a tím jej lze zapojovat do 8bitového prostředí. Firma IBM ho převážně používala v počítačích IBM PC a IBM PC/XT. Procesory Intel 80186 a 80188 (dodnes vyráběny a používány především v řídících aplikacích) jsou rozšířením procesorů 8086 a 8088 o dva kanály rychlého přístupu k paměti (DMA), tři programovatelné časovače, programovatelný řadič přerušení, generátor časovacích impulsů a o několik málo instrukcí.

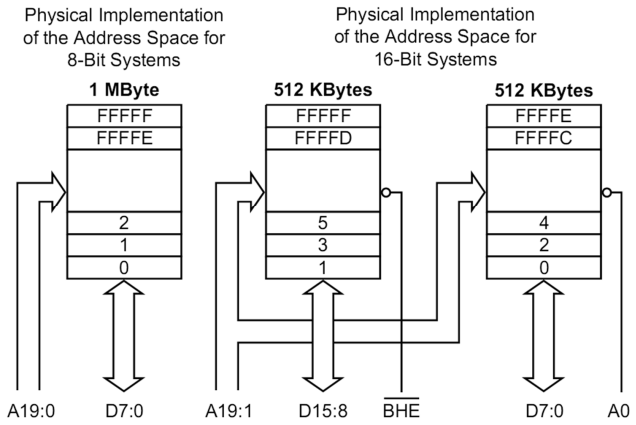
Srovnání procesorů řady 80x86

Intel Processor	Date of Product Intro-duction	Perform -ance in MIPs[1]	Max. CPU Frequency at Intro-duction	No. of Transis -tors on the Die	Main CPU Register Size[2]	Extern. Data Bus Size[2]	Max. Extern. Addr. Space	Caches in CPU Pack-age[3]
8086	1978	0.8	8 MHz	29 K	16	16	1 MB	None
Intel 286	1982	2.7	12.5 MHz	134 K	16	16	16 MB	Note 3
Intel386™ DX	1985	6.0	20 MHz	275 K	32	32	4 GB	Note 3
Intel486™ DX	1989	20	25 MHz	1.2 M	32	32	4 GB	8KB L1
Pentium®	1993	100	60 MHz	3.1 M	32	64	4 GB	16KB L1
Pentium® Pro	1995	440	200 MHz	5.5 M	32	64	64 GB	16KB L1; 256KB or 512KB L2
Pentium II®	1997	466	266	7 M	32	64	64 GB	32KB L1; 256KB or 512KB L2
Pentium® III	1999	1000	500	8.2 M	32 GP 128 SIMD-FP	64	64 GB	32KB L1; 512KB L2

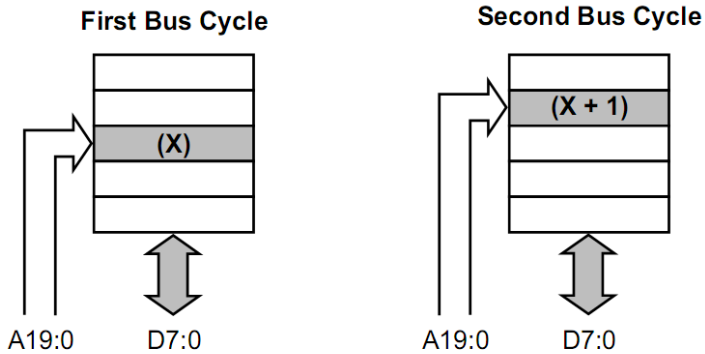
Procesor 8086

- Typická von Neumannova architektura – adresový prostor paměťový (společný pro program i data – velikost 1 MB) a adresový prostor vstupně-výstupních zařízení (pro adresaci portů – max. velikost 64k portů)
- Pracuje s daty šířky 16 nebo 8 bitů a vytváří 20bitovou adresu (tzv. fyzickou adresu) pro adresování 1 MB paměti. Virtuální paměť nepodporuje. Paměť je rozdělena na slabiky. Může být šíře 8 nebo 16 bitů.
- Pro uložení dat v paměti používá reprezentaci Little Endian (na nižší adrese méně významná slabika).
- Znaménková čísla jsou vyjádřena ve dvojkovém doplňku.
- Umí pracovat i s BCD čísly.

Organizace paměti šíře 8 nebo 16 bitů

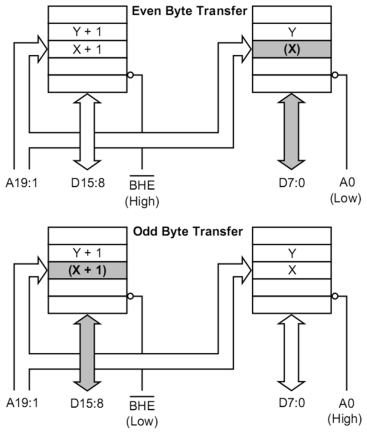


Čtení 16-bitového slova z paměti o šíři 8 bitů

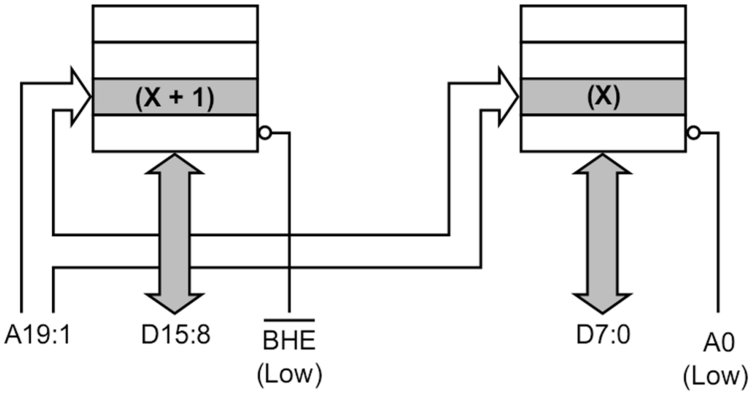


Jsou potřeba dva sběrnicové cykly

Čtení bytu z paměti o šíři 16bitů

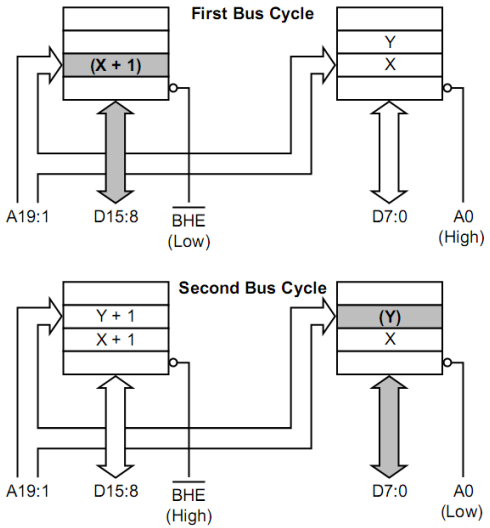


Čtení slova se sudé adresy z paměti o šíři 16 bitů



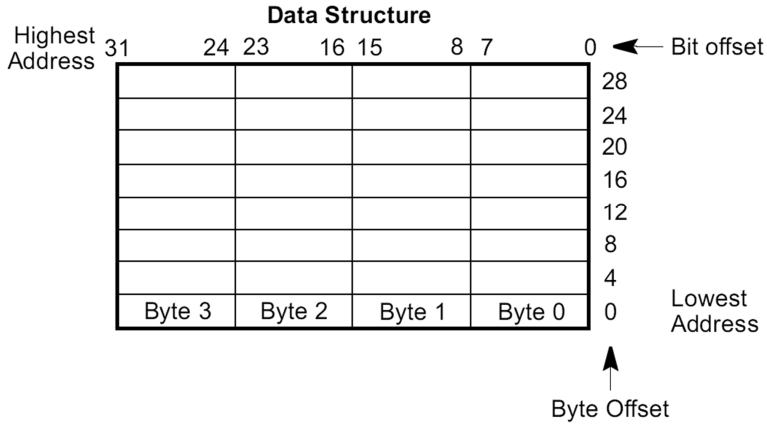
Je potřeba pouze jeden sběrnicový cyklus

Čtení slova z liché adresy - 16bit šířka paměti

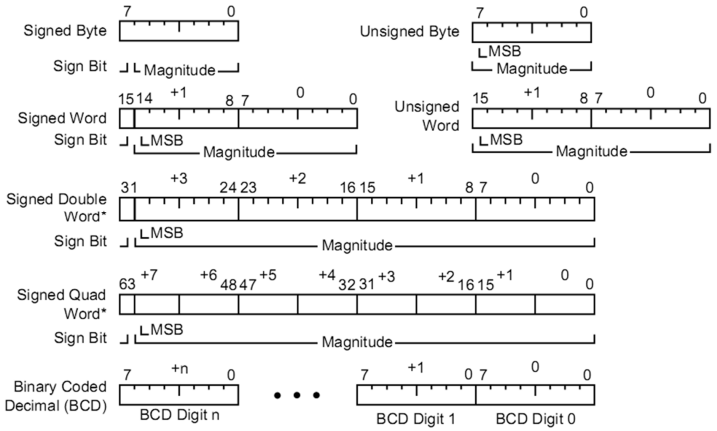


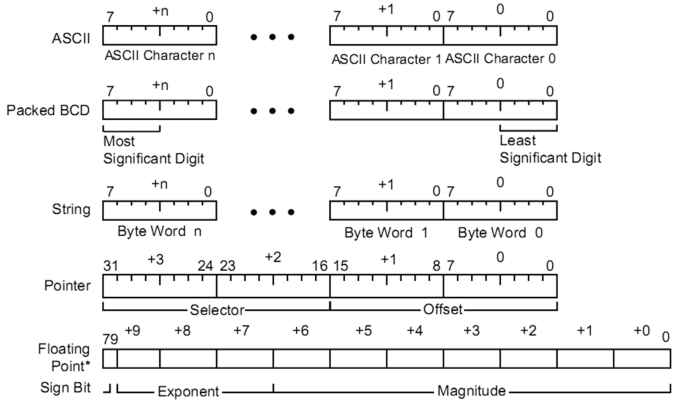
Jsou potřeba dva sběrnicové cykly

Adresace paměti při šíři 32 bitů



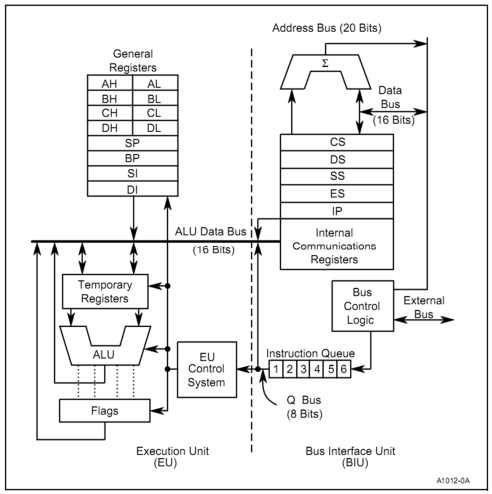
Formát dat v paměti





NOTE: *Directly supported if the system contains an 80C187.

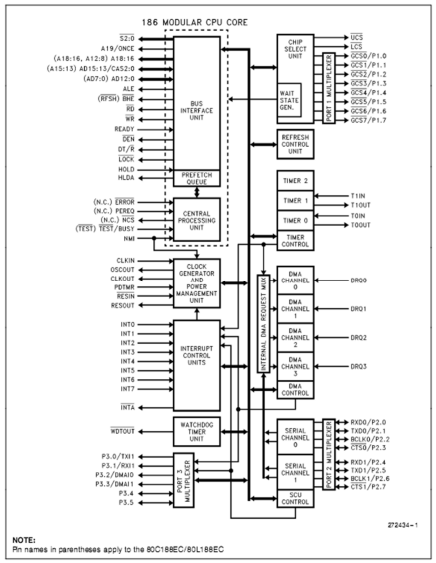
Struktura procesoru 8086



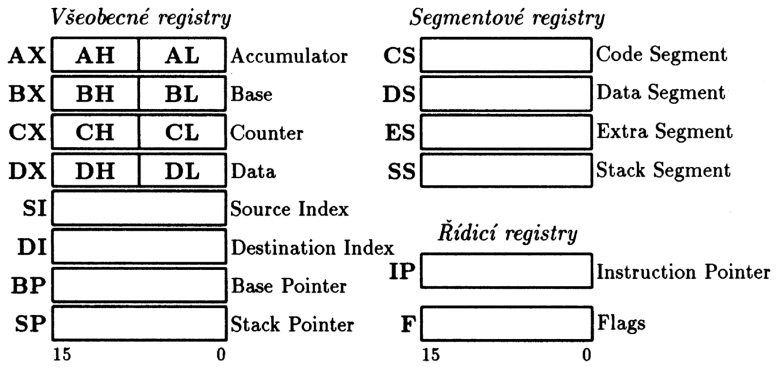
Dvě nezávislé jednotky EU a BIU

- Jednotka **Bus Interface Unit (BIU)** – **Sběrnicová jednotka** –vybírá instrukční kód z paměti a přenáší data po sběrnicích. Ukládá vybrané instrukční kódy do instrukční fronty.
- Jednotka **Execution Unit (EU)** – **Prováděcí jednotka** – vybírá instrukční kódy z instrukční fronty a postupně je zpracovává (náznak zřetězeného zpracování instrukcí v RISC procesorech). Problémy nastávají při instrukcích skoku. Instrukční fronta se musí vyprázdnit a BIU nahrát nové instrukční kódy.

Vnitřní struktura 80186



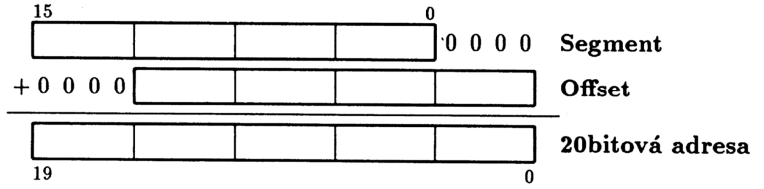
Vnitřní registry 8086



Logická a fyzická adresa

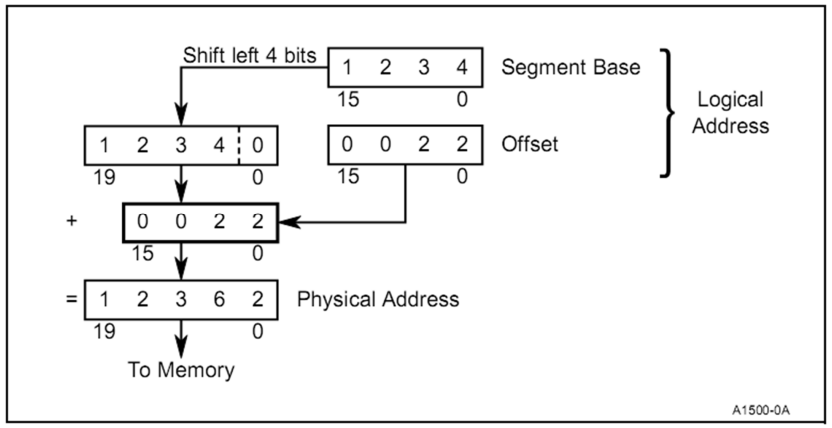
- **Logická adresa** – 32 bitové číslo skládající se ze dvou 16bitových čísel – segmentu a offsetu
- Zapisujeme **segment:offset** (např. 21AB:0030)
- Používá programátor
- **Fyzická adresa** – pro přístup do paměti – BIU jednotka převádí logickou adresu na fyzickou.
- Segmentová část se obyčejně vezme z tzv. segmentového registru (CS, SS, DS, ES)
- Převod z logické adresy na fyzickou - 16bitová segmentová část se vynásobí 16 (posune o 4 bity doleva) a přičte se offsetová část => 20bitové číslo

Převod logické adresy na fyzickou

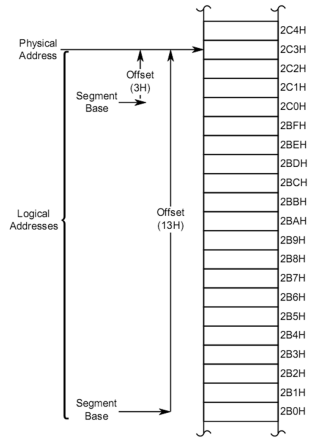


- Převod je jednoznačný.
- Převod z fyzické adresy na logickou je nejednoznačný – existuje 4096 možných vyjádření

Ukázka převodu logické adresy 1234:0022 na fyzickou



Převod fyzické adresy na logickou

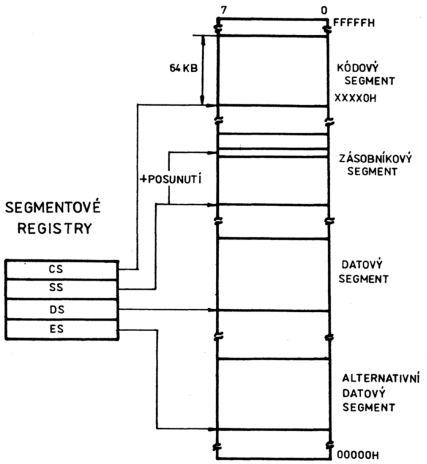


Ukázány 2 možnosti převody fyzické adresy 002C3H na logickou: 002B:0013H a 002C:0003H

Segmentové registry

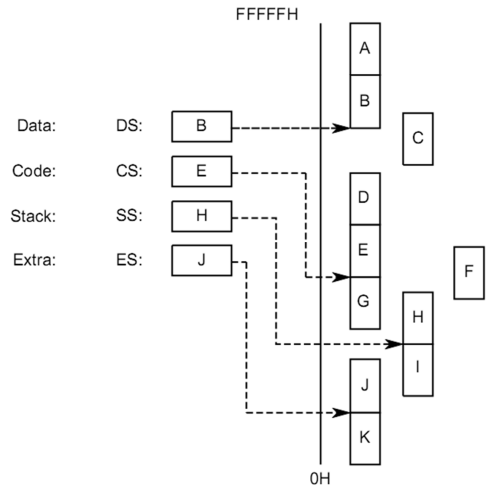
- CS (Code Segment) – je určen pro výpočet adresy instrukce (definuje instrukční segment). Logická adresa právě zpracovávané instrukce je CS:IP
- SS (Stack Segment) – definuje zásobníkový segment. Vrchol zásobníku je určen logickou adresou SS:SP
- DS (Data Segment) – definuje datový segment. Použije se při výpočtu adresy dat
- ES (Extra Segment) – obdoba DS registru, definuje pomocný datový segment nebo se používá při tzv. řetězcových instrukcích

Způsob adresace paměti 8086

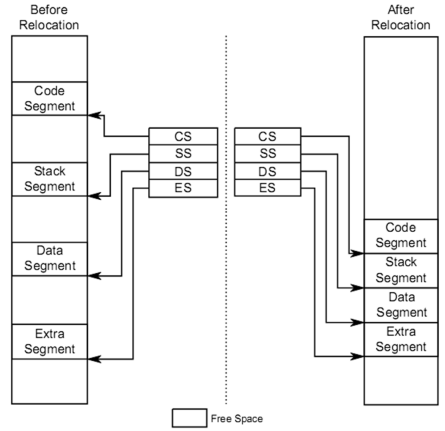


tzv. segmentování paměti pomocí segmentových registrů – okna do adresového prostoru

Rozmístění a zpřístupnění segmentů v paměti



Výhoda použití segmentování na tzv. relokaci paměti



Používá se často v operačních systémech na tzv. defragmentaci paměti

Adresovací techniky

- Příkazy určené procesoru zadává programátor v asembleru ve formě instrukcí. Instrukce je zpravidla složena ze dvou částí: z operačního kódu a operandů. Operační kód je vlastním příkazem pro procesor a operandy obsahují vlastní data (příp. doplňující informace). Operandů může být nula, jeden nebo dva a mohou být zpřístupněny (adresovány) osmi různými technikami.
- První dvě adresovací techniky zpřístupňují operand uložený v jednom z všeobecných registrů procesoru nebo operand přímo uložený v instrukci

Příklady adresovacích technik si budeme uvádět na instrukci **MOV AH,operand**, která naplní registr AH zadanou hodnotou

- Přímý operand
  - Operand je uložen přímo v instrukci (za operačním kódem) jako konstanta.
  - Příklad: Instrukce MOV AH,50 naplní registr AH číslem 50.
- Registr
  - Operand je uložen v některém 16bitovém (AX, BX, CX, DX, SI, DI, SP, BP) nebo 8bitovém (AH, AL, BH, BL, CH, CL, DH, DL) registru. Některé instrukce také připouštějí uložení operandu v registrech CS, DS, ES, SS nebo F.
  - Např. instrukce MOV AH,BL přepíše obsah registru BL do AH.

Dalších šest technik zpřístupňuje operandy uložené ve fyzické paměti. Základní operace výpočtu 20bitové fyzické adresy je součástí každé z technik. Část adresy nazývaná segment je uložena v některém ze čtyř segmentových registrů. V instrukci se potom určí, ze kterého z registru CS, DS, ES nebo SS se segment vybere a použije se spolu s offsetem pro výpočet fyzické adresy.

Typ přístupu do paměti jednoznačně určuje, který ze segmentových registrů má být použit. Přiřazení segmentových registrů typům přístupu do paměti popisuje následující obrázek:

Při přístupu k	*se použije registr*	*Operace*
instrukcím	**CS** (Code Segment)	Výběr operačního kódu nebo přímého operandu.
zásobníku	**SS** (Stack Segment)	Při všech přístupech k zásobníku nebo ve spojitosti s registrem BP.
datům	**DS** (Data Segment)	Při všech přístupech k datům v paměti vyjma zásobníku a přímých operandů. V řetězcových operacích segmentuje zdrojový operand.
alternativním datům	**ES** (Extra Segment)	V řetězcových operacích pro segmentování cílového operandu.

Každý registr, který lze použít pro uložení offsetové části adresy, má přiřazen jeden ze segmentových registrů. Není-li v instrukci specifikováno jinak, použije se toto tzv. implicitní přiřazení.

Registr s offsetem	Implicitně použitý segmentový registr
SP	SS
BP	SS
BX	DS
SI	DS
DI	DS (ES v řetězcových operacích)
BP v kombinaci s SI nebo DI	SS
BX v kombinaci s SI nebo DI	DS

Šest adresovacích technik, o kterých bude diskutováno dále, se už týká pouze výpočtu offsetové části adresy (neboli tzv. efektivní adresy EA). Offset se počítá z těchto tří složek:

1. přímé adresy (Displacement) uložné v instrukci,
2. báze (obsah registru BX nebo BP),
3. indexu (obsah registru SI nebo DI).

Každá z níže uvedených technik kombinuje tyto složky jiným způsobem.

Přímá adresa

Offsetové části adresy operandu umístěného ve fyzické paměti se přiřadí 16bitová přímá adresa uložená v instrukci. Přímá adresa v instrukci adresující data (MOV) se segmentuje přes DS a v instrukci adresující jinou instrukci (JMP) se segmentuje přes CS.

Příklad: MOV AH,Adresa, kde Adresa je symbolicky zapsaná přímá adresa slabiky, jejíž obsah se uloží do AH.

Pozn.

Potřebujeme-li přímou adresu zadat absolutní hodnotou, musíme zapsat MOV AH ,DS:[101] . V takto specifikované adrese musí být uveden segmentový registr a číslo v hranatých závorkách se nechápe jako konstanta, jíž se má naplnit registr AH, ale jako adresa požadovaného obsahu. Instrukce MOV AH, [Adresa] je ekvivalentní instrukci MOV AH,Adresa.

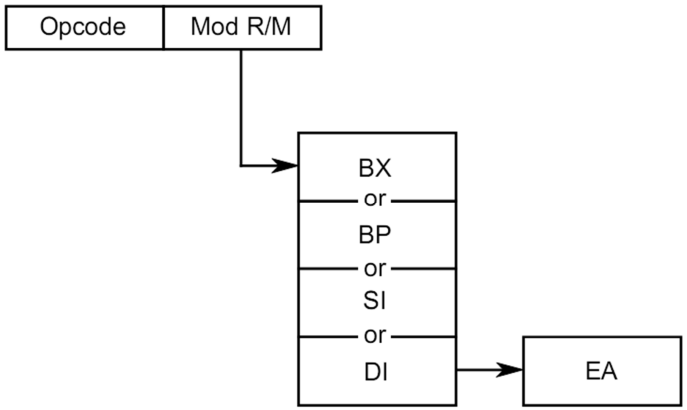
Přímá adresa

Nepřímá adresa

Offsetové části adresy se přiřadí obsah registru SI, DI, SP, BP nebo BX.

Příklad: MOV AH,[BX]. Je-li jméno registru BX (nebo SI, DI, SP, BP) uvedeno hranatých závorkách, znamená to, že se do AH neuloží jeho obsah, ale obsah ležící na adrese uvedené v zadaném registru.

Nepřímá adresa



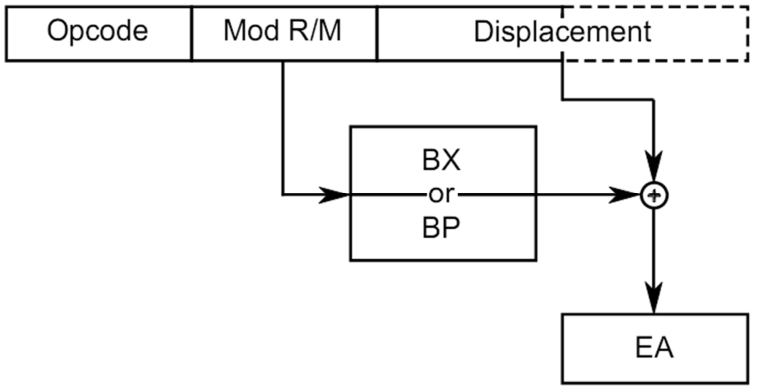
Bázovaná adresa

Offsetová část adresy se získá sečtením přímé adresy umístěné v instrukci a jednoho z bázových registrů (BX nebo BP).

Příklad: MOV AH,[BP+Adresa]

Bázovaná adresa je určena pro přístupy k datovým strukturám typu záznam. Do bázového registru se průběžně ukládají adresy začátků záznamů a přímá adresa obsahuje vzdálenost žádaného prvku od začátku záznamu ve slabikách.

Bázovaná adresa



Indexovaná adresa

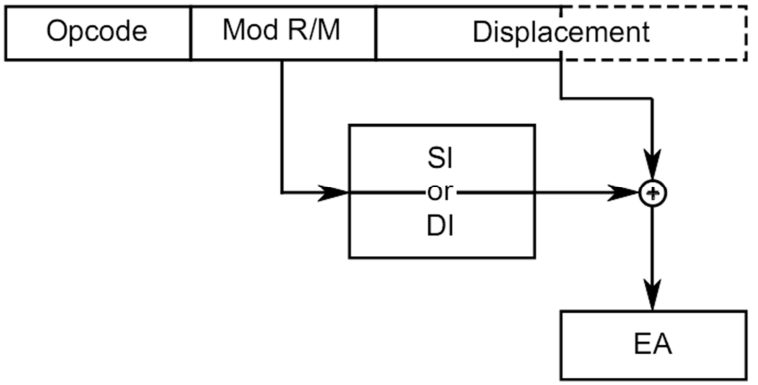
Offsetová část adresy se získá sečtením přímé adresy umístěné v instrukci a jednoho z indexových registrů (SI nebo DI).

Příklad: MOV AH,[Adresa+SI] a MOV AH,Adresa[SI] jsou ekvivalentní instrukce.

Indexování se používá v případě přístupu k datové struktuře s pevnou začáteční adresou (např. pole). Přímá adresa ukazuje pak na začátek pole a hodnota v registru SI nebo DI je indexem (ve slabikách) vybírajícím danou slabiku pole.

Indexování a bázování se v procesoru 8086 od sebe liší jenom použitím buď indexového nebo bázového registru. V konečném efektu jsou to techniky shodné.

Indexovaná adresa



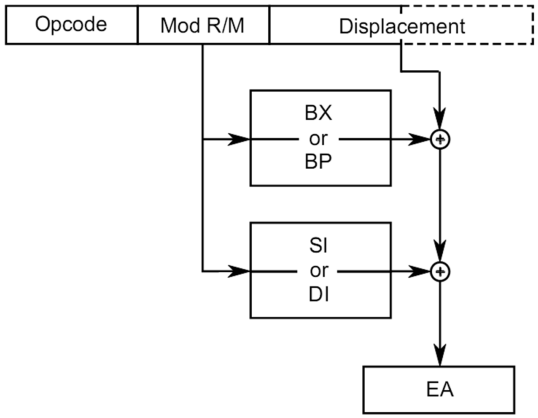
Kombinovaná adresa: báze+index

Tato adresní technika je kombinací dvou předchozích. Offset operandu je vypočítán součtem hodnoty uložené v jednom z bázových registrů (BX, BP) a hodnoty uložené v jednom z indexových registrů (SI, DI)

Příklad: MOV AH,[BX][DI] nebo MOV AH,[BP+DI]

S výhodou se dá tato adresovací technika použít v případech, kdy potřebujeme pracovat s dynamickou datovou strukturou. To znamená, že například počáteční adresa pole se bude během výpočtu měnit. Pak bázový registr udržuje počáteční adresu pole a indexový registr ukazuje na jednotlivé prvky tohoto pole.

Kombinovaná adresa



Kombinovaná adresa: přímá+báze+index

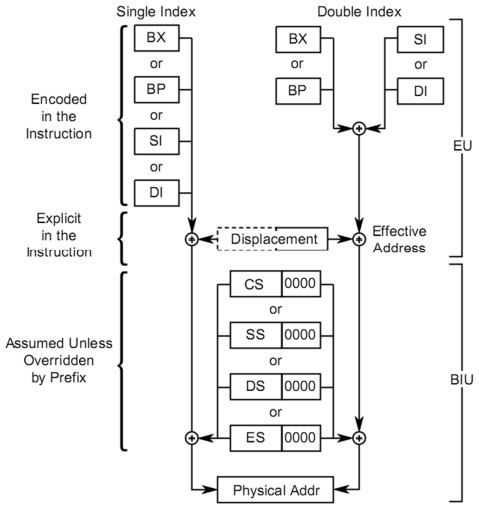
V tomto případě je offset vypočítán jako součet obsahu bázového registru (BX nebo BP), indexového registru (SI nebo DI) a přímé adresy uvedené v instrukci.

Příklad: MOV AH,Adresa[BX][DI] nebo MOV AH,[Adresa+BX+DI]

Poznamenejme, že instrukce např. MOV AH, [Adresa+BX+DI+1] je stále stejného typu, protože překladač hodnotu (Adresa+1) vyčíslí a uloží jako přímou adresu.

Nejčastěji je tento způsob adresace používán při přístupu k složitým datovým strukturám, jako je pole v záznamu. Bázový registr ukazuje na začátek záznamu, přímá adresa určuje vzdálenost začátku pole uvnitř záznamu a hodnota indexového registru vybírá jednotlivé prvky daného pole.

Přehled všech adresovacích technik



Adresovací techniky a změna implicitního segmentového registru

Adresovací technika	Vypočtená adresa =
přímá adresa	*přímá adresa*
nepřímá adresa	*báze*
bázovaná adresa	*přímá adresa + báze*
indexovaná adresa	*přímá adresa + index*
kombinovaná: báze+index	*báze + index*
kombinovaná: přímá+báze+index	*přímá adresa + báze + index*

Obr. 2.9. Adresovací techniky 8086

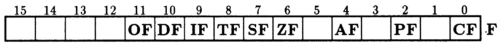
Segmentový registr	*Instrukční prefix*
DS (Data Segment)	3Eh
CS (Code Segment)	2Eh
SS (Stack Segment)	36h
ES (Extra Segment)	26h

Obr. 2.10. Prefixy měnící přiřazení segmentových registrů

Příklady změn implicitního segmentového registru

- MOV AH,CS:[BX] - Nepřímá adresa CS:BX (nikoli implicitně DS:BX)
- ADD AH,DS: [BP] [SI] - Kombinovaná adresa
- ADC AH,ES:Adresa2 - Přímá adresa segmentovaná přes ES
- MOV AH,SS:Adresa - Přímá adresa segmentovaná přes SS

Příznakový registr 8086



Obr. 2.4. Příznakový registr procesoru 8086

CF (Carry Flag) se nastaví na jedničku, jestliže při právě provedené aritmetické operaci došlo k přenosu z nejvyššího bitu, a to jak při práci s 8, tak 16bitovým operandem. Tohoto příznaku je také využíváno při operacích logického posuvu a rotace. Příznak může měnit i programátor instrukcemi STC, CLC a CMC.

PF (Parity Flag) se nastaví na jedničku, pokud dolní osmice bitů výsledku právě provedené operace obsahuje sudý počet "1" (sudá parita výsledku). Při lichém počtu jedniček (lichá parita) se příznak nuluje. PF se vypočítává pouze z dolní osmice bitů bez ohledu na šířku operandu.

AF (Auxiliary Carry Flag) je rozšířením příznaku CF pro přenos přes hranici nejnižší půlslabiky operandu (vždy z bitu 3 do 4 bez ohledu na šířku operandu). Má význam v BCD aritmetice (viz str. 67).

ZF (Zero Flag) je nastaven na jedničku při nulovém výsledku právě dokončené operace.

SF (Sign Flag) indikuje kladný (SF=0) nebo záporný (SF=1) výsledek právě provedené operace. Tento bit je kopií znaménkového bitu výsledku operace.

OF (Overflow Flag) se nastaví na jedničku, pokud při právě dokončené operaci došlo k aritmetickému přeplnění (výsledek spadá mimo rozsah zobrazení). Procesor oznámí přeplnění, pokud se přenos do znaménkového bitu nerovnal přenosu ze znaménkového bitu.

TF (Trap Flag) uvádí procesor do krokovacího režimu, ve kterém je po provedení první instrukce generováno přerušení (INT 1). Příznak nastavují různé ladicí systémy, které tímto režimem po jednotlivých instrukcích krokují laděný program. Ladicí systém se aktivuje generovaným přerušením a potom zjišťuje, co se během provádění instrukce změnilo. Příznak lze nastavit pouze přes zásobník instrukcí IRET.

IF (Interrupt Enable Flag) vynulovaný instrukcí CLI zabrání uplatnění vnějších maskovatelných přerušení (generovaných signálem INTR[1]). Nastavení příznaku na jedničku (instrukcí STI) přerušení povoluje. Maskovat lze přerušení od vnějších zařízení (klávesnice, tiskárna atd.), nikoli programově simulovaná přerušení (instrukce INT) a NMI (přerušení ze skupiny nemaskovatelných přerušení).

DF (Direction Flag) řídí směr zpracovávání řetězcových operací. Při DF nastaveném instrukcí STD se obsah registrů SI a DI zvyšuje (řetězce se zpracovávají odpředu) a při DF vynulovaném instrukcí CLD se obsah SI a DI snižuje (řetězce se zpracovávají odzadu).

Přerušení - Interrupt

- Přerušení dělíme do skupin podle toho, čím jsou generovány:
  - Technickými prostředky (vnější):
    - Nemaskovatelná (NMI)
    - Maskovatelná (INTR)
  - Programově (vnitřní):
    - Instrukcí INT x
    - Chybou při běhu programu
- Procesor rozlišuje 256 různých přerušení. Přerušovací řadič jako odpověď na signál INTA umístí na datovou sběrnici 8-bitové číslo 0-255
- Pro každé přerušení je v paměti uloženo 32 bitů adresy (tzv. přerušovací vektor) začátku obslužného přerušovacího podprogramu
- Adresy jsou zapsány v tabulce přerušovacích vektorů, která začíná na adrese 0000:0000 a má velikost 1kB (256x4 byty). Index do této tabulky je číslo 0-255 vrácené přerušovacím řadičem na signál INTA.

Tabulka přerušovacích vektorů

31	0	Adresa	Číslo přer. vektoru
segment	*offset*	0:03FC	INT 0FFh
	⋮	⋮	
segment	*offset*	0:000C	INT 3
segment	*offset*	0:0008	INT 2
segment	*offset*	0:0004	INT 1
segment	*offset*	0:0000	INT 0

- Přerušení způsobená vnějšími V/V zařízeními lze maskovat vynulováním příznaku **IF** v příznakovém registru instrukcí **CLI**. Tento zákaz přerušení se nevztahuje na vnitřní přerušení a NMI
- Zakázané přerušení se povoluje instrukcí **STI**
- Předpokládejme, že nastalo přerušení číslo **n** nebo procesor dekódoval instrukci **INT n**. Potom se provedou činnosti v tomto pořadí:

Vyvolání přerušovacího podprogramu:

1. do zásobníku se uloží registr příznaků (F),
2. vynulují se příznaky IF a TF (zakáže se přerušení a krokování),
3. do zásobníku se uloží registr CS,
4. registr CS se naplní 16bitovým obsahem adresy n x 4 + 2,
5. do zásobníku se uloží registr IP ukazující na neprovedenou instrukci,
6. registr IP se naplní 16bitovým obsahem adresy n x 4.

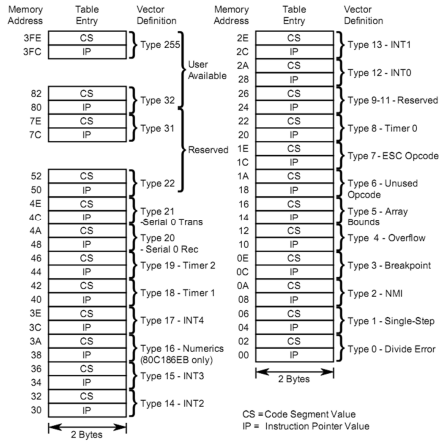
Návrat z přerušovacího podprogramu instrukcí IRET

1 . ze zásobníku obnoví registr IP,
2. ze zásobníku obnoví registr CS,
3. ze zásobníku obnoví příznakový registr (F).

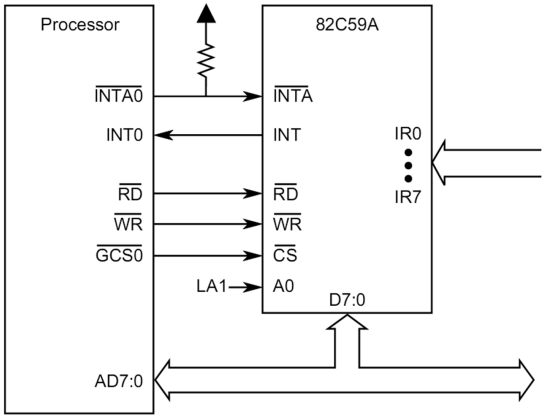
Vyhrazená přerušení procesoru 8086

INT *n*	*Význam*
0	Celočíselné dělení nulou (Divide by Zero)
1	Krokovací režim (Single-Step)
2	Nemaskovatelná přerušení (NMI)
3	Ladicí bod (Breakpoint Trap)
4	Přeplnění (Overflow Trap)

Vyhrazená přerušení pro procesor 80186



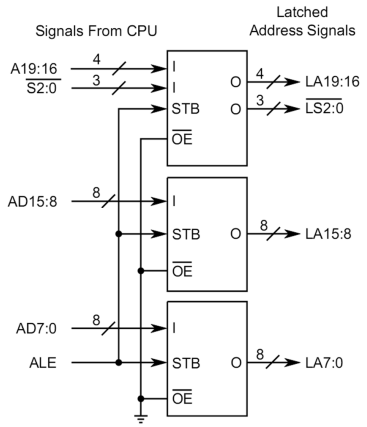
Připojení přerušovacího řadiče k mikroprocesoru



Počáteční nastavení procesoru

- Procesor je iniciován aktivní úrovní signálu RESET. V tom okamžiku:
  - vynuluje příznakový registr (tedy zakáže přerušení a krokování)
  - vynuluje registr IP
  - vynuluje segmentové registry SS, DS a ES a registr CS naplní hodnotou 0FFFFH
  - Zruší obsah instrukční fronty
- První instrukce, která bude provedena po signálu RESET nebo zapnutí napájení je CS:IP = FFFF:0000H = 0FFFF0H (16 bytů před koncem 1MB adresového prostoru)

Multiplexování sběrnic a stavové slovo

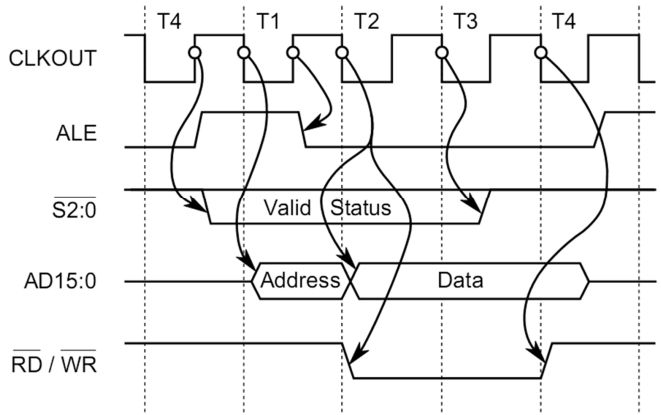


Signál ALE navzorkuje adresu a stavové slovo, které se objeví na sběrnici AD0-15, do vyrovnávací paměti. Pak se sběrnice AD0-15 použije pro přenos dat

Význam kombinace stavových bitů

Status Bit			Operation
S2	S1	S0	
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Instruction Prefetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Idle (passive)

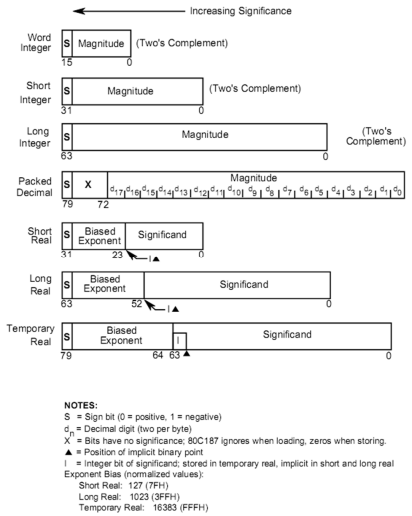
Časový diagram multiplexované sběrnice



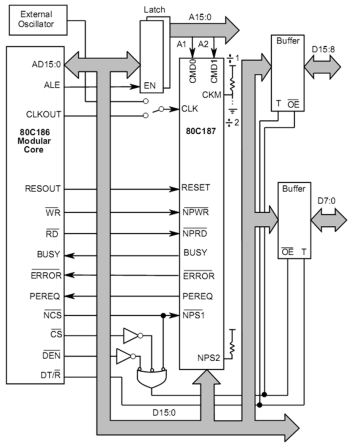
Matematický koprocesor 8087

- Specializovaný obvod, který umožňuje velmi rychlé výpočty s čísly v pevné i pohyblivé desetinné čárce
- Zrychluje výpočty 50 až 100x

Formáty dat matematického koprocesoru 8087



Zapojení koprocesoru do systému



Instrukční soubor 8086
Skupiny instrukcí

1. Instrukce MOV
2. Aritmetické instrukce
3. Logické instrukce
4. Rotace a posuvy
5. Větvení programu
6. Zásobník a příznakový registr
7. Přerušovací systém
8. Cykly
9. Ovládání V/V
10. Přesuny dat
11. Řetězcové instrukce
12. Instrukce BCD aritmetiky
13. Řídicí instrukce

Význam parametrů v popisu instrukcí

rel8	Relativní 8bitová adresa v intervalu −128 až 127.
rel16	Relativní 16bitová adresa v intervalu 0 až 65 535.
ptr16:16	32 bitů absolutní přímé adresy složené ze segmentu a offsetu. Viz též poznámka u *m16:16*.
r8	8bitový všeobecný registr.
r16	16bitový všeobecný registr.
imm8	8bitová přímá hodnota.
imm16	16bitová přímá hodnota.
r/m8	Buď 8bitový registr nebo offset slabiky v paměti.
r/m16	Buď 16bitový registr nebo offset 16bitového slova v paměti.
m16:16	32 bitů nepřímé adresy umístěné v paměti složené ze segmentu a offsetu (složky adresy jsou v paměti uloženy v obráceném pořadí – např. adresa `1234:5678h` bude mít v paměti, vypsané se vzrůstající adresou, tvar: `78 56 34 12h`).
Sreg	Segmentový registr (ES, CS, SS, DS).

Význam příznakových bitů v popisu instrukcí

O	OF (Overflow Flag)
D	DF (Direction Flag)
I	IF (Interrupt Flag)
T	TF (Trap Flag)
S	SF (Sign Flag)
Z	ZF (Zero Flag)
A	AF (Auxiliary CF)
P	PF (Parity Flag)
C	CF (Carry Flag)

?	hodnota příznaku nedefinována
*	nastavena podle výsledku
0	hodnota příznaku vždy nula
1	hodnota příznaku vždy jedna
	hodnota příznaku nezměněna

Instrukce MOV

MOV — *Move Data*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce MOV přenáší obsah zdrojového operandu (tj. operandu stojícího více vpravo) do cílového operandu. Přitom obsah zdrojového operandu zůstává beze změny.

SYNTAX: ⌜ MOV *cílový_operand,zdrojový_operand*

Podle typu zdrojového a cílového operandu je k dispozici několik variant instrukce MOV:

Instrukce	*Příklad*	*Komentář*
MOV *r/m8,r8*	`MOV AL,BL`	`; AL je naplněn obsahem BL`
	`MOV Slab,CH`	`; Slab je naplněn obsahem CH`
MOV *r/m16,r16*	`MOV BX,CX`	`; BX je naplněn obsahem CX`
	`MOV Slovo,DX`	`; Slovo je naplněn obsahem DX`
MOV *r8,r/m8*	`MOV CL,Slab`	`; CL je naplněn obsahem Slab`
MOV *r16,r/m16*	`MOV CX,Slovo`	`; CX je naplněn obsahem Slovo`
MOV *r/m16,Sreg*	`MOV AX,CS`	`; AX je naplněn obsahem CS`
	`MOV Slovo,SS`	`; Slovo je naplněn obsahem SS`
MOV *Sreg,r/m16*	`MOV DS,AX`	`; DS je naplněn obsahem AX`
	`MOV ES,Slovo`	`; ES je naplněn obsahem Slovo`
MOV *r/m8,imm8*	`MOV DL,32`	`; DL je naplněn konstantou 32`
	`MOV Slab,-128`	`; Slab je naplněn konstantou -128`
MOV *r/m16,imm16*	`MOV DI,32767`	`; DI je naplněn konstantou 32767`
	`MOV Slovo,-1`	`; Slovo je naplněn konstantou -1`

Aritmetické instrukce

ADD — *Integer Addition*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce ADD celočíselně přičítá obsah zdrojového operandu k obsahu cílového operandu a výsledek této operace umístí do cílového operandu. Obsah zdrojového operandu se nezmění. Pokud dojde při sčítání k přeplnění, nastaví se příznak OF na 1.

SYNTAX: ⌜ ADD *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
ADD *r/m8,imm8*	`ADD AL,80`	`; AL := AL + 80`
	`ADD Slab,-10`	`; Slab := Slab + (-10)`
ADD *r/m16,imm16*	`ADD CX,10000`	`; CX := CX + 10000`
	`ADD Slovo,30000`	`; Slovo := Slovo + 30000`
ADD *r/m16,imm8*	`ADD DI,2`	`; DI := DI + 2`
	`ADD Slovo,57`	`; Slovo := Slovo + 57`
ADD *r/m8,r8*	`ADD CH,CL`	`; CH := CH + CL`
	`ADD Slab,BL`	`; Slab := Slab + BL`
ADD *r/m16,r16*	`ADD AX,BX`	`; AX := AX + BX`
	`ADD Slovo,DX`	`; Slovo := Slovo + DX`
ADD *r8,r/m8*	`ADD CL,Slab`	`; CL := CL + Slab`
ADD *r16,r/m16*	`ADD BP,Slovo`	`; BP := BP + Slovo`

ADC — *Integer Addition with Carry*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce ADC přičte k cílovému operandu obsah zdrojového operandu a hodnotu příznaku přenosu (CF). Obsah zdrojového operandu zůstane nezměněn a příznak přenosu (CF) se nastaví podle konečného součtu. Operace je určena k postupnému sčítání operandů, jejichž šířka je větší než možná zpracovávaná.

SYNTAX: ⌜ ADC *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
ADC *r/m8,r8*	`ADC AL,CL`	`; AL := AL + CL + CF`

Další varianty viz ADD

INC — *Increment by 1*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

POPIS:

Instrukce INC zvyšuje hodnotu operandu o jedničku. Tato instrukce neovlivňuje příznak CF.

SYNTAX: ⌜ INC *operand*

Instrukce	*Příklad*	*Komentář*
INC *r/m8*	`INC CL`	`; CL := CL + 1`
	`INC Slab`	`; Slab := Slab + 1`
INC *r/m16*	`INC SP`	`; SP := SP + 1`
	`INC Slovo`	`; Slovo := Slovo + 1`

SUB — *Integer Subtraction*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce SUB odčítá od obsahu cílového operandu obsah operandu zdrojového a výsledek této operace ukládá do cílového operandu. Obsah zdrojového operandu zůstane nezměněn.

SYNTAX: ⌜ SUB *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
SUB *r/m8,r8*	`SUB AL,CL`	`; AL := AL - CL`
	Další varianty viz ADD	

SBB — *Integer Subtraction with Borrow*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce SBB odčítá od obsahu cílového operandu obsah operandu zdrojového a obsah příznaku přenosu (CF). Výsledek této operace ukládá do cílového operandu. Obsah zdrojového operandu zůstane nezměněn a příznak přenosu se nastaví podle výsledku operace. Instrukce se používá pro odčítání operandů, jejichž šířka je větší než možná zpracovávaná.

SYNTAX: ⌜ SBB *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
SBB *r/m8,r8*	`SBB AL,CL`	`; AL := AL - CL - CF`
	Další varianty viz ADD	

DEC — *Decrement by 1*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	

POPIS:

Instrukce DEC snižuje hodnotu operandu o 1. Nemění nastavení příznaku CF.

SYNTAX: ⌜ DEC *operand*

Instrukce	*Příklad*	*Komentář*
DEC *r/m8*	`DEC BL`	`; BL := BL - 1`
	`DEC Slab`	`; Slab := Slab - 1`
DEC *r/m16*	`DEC AX`	`; AX := AX - 1`
	`DEC Slovo`	`; Slovo := Slovo - 1`

IMUL — *Signed Multiplication*

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

POPIS:

Instrukce **IMUL** provádí znaménkové násobení (tzn. respektuje znaménka činitelů) obsahu registru AL nebo AX operandem. Je-li operand typu slabika, násobí se obsah registru AL a výsledek se umístí do registru AX (AH obsahuje vyšší řády a AL nižší). Je-li operand 16bitový, pak se násobí obsah registru AX a výsledek se uloží do registrového páru DX&AX tak, že registr DX obsahuje vyšších 16 bitů a registr AX nižších 16 bitů součinu. Je-li vyšší polovina výsledku násobení znaménkovým rozšířením nižší poloviny, jsou příznaky CF a OF nulovány.

SYNTAX: ⌜ **IMUL** *operand*

Instrukce	*Příklad*	*Komentář*
IMUL *r/m8*	`IMUL BL`	`; AX := AL * BL`
	`IMUL Slab`	`; AX := AL * Slab`
IMUL *r/m16*	`IMUL CX`	`; DX&AX := AX * CX`
	`IMUL Slovo`	`; DX&AX := AX * Slovo`

MUL — *Unsigned Multiplication*

O	D	I	T	S	Z	A	P	C
*				?	?	?	?	*

POPIS:

Instrukce **MUL** má stejnou syntax a funkci jako instrukce **IMUL** s tím rozdílem, že provádí neznaménkové násobení (bit nejvyššího řádu se neuvažuje jako znaménkový).

IDIV — *Signed Divide*

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

POPIS:

Instrukce **IDIV** provádí celočíselné znaménkové dělení obsahu registru AX nebo DX&AX (dělenec) operandem instrukce (dělitel). Je-li operand typu slabika, pak je dělencem obsah registru AX. Potom se výsledek (podíl) uloží do registru AL a zbytek po dělení do registru AH. Je-li operand 16bitový, dělí se obsah registrového páru DX&AX (DX obsahuje vyšších 16 bitů a AX nižších 16 bitů dělence). Podíl je potom uložen do registru AX a zbytek po dělení do registru DX. Pokud je podíl větší než rozsah zobrazení registru AL (AX), tj. při dělení nulou, nastane přerušení **INT 0**.
Zbytek má stejné znaménko jako dělenec.

SYNTAX: ⌜ **IDIV** *operand*

Instrukce	*Příklad*	*Komentář*
IDIV *r/m8*	`IDIV BL`	`; AL := AX ÷ BL a AH :=` **AX** *mod* **BL**
	`IDIV Slab`	`; AL := AX ÷ Slab a AH :=` **AX** *mod* **Slab**
IDIV *r/m16*	`IDIV CX`	`; AX := DX&AX ÷ CX a DX :=` **DX&AX** *mod* **CX**
	`IDIV Slovo`	`; AX := DX&AX ÷ Slovo a DX :=` **DX&AX** *mod* **Slovo**

DIV — *Unsigned Divide*

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

POPIS:

Instrukce **DIV** má stejnou syntax a funkci jako instrukce **IDIV** s tím rozdílem, že provádí neznaménkové dělení.

NEG — *Two's Complement Negation*

POPIS:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Instrukce **NEG** nahradí obsah operandu jeho dvojkovým doplňkem (změní znaménko, tj. vynásobí operand hodnotou -1). Dvojkový doplněk je inverze všech bitů a přičtení jedničky.

SYNTAX: ⌜ **NEG** *operand*

Instrukce	*Příklad*	*Komentář*
NEG *r/m8*	`NEG AH`	`; AH := -AH`
	`NEG Slab`	`; Slab := -Slab`
NEG *r/m16*	`NEG SI`	`; SI := -SI`
	`NEG Slovo`	`; Slovo := -Slovo`

CBW — *Convert Byte to Word*

POPIS:

O	D	I	T	S	Z	A	P	C

Instrukce **CBW** aritmeticky převádí obsah slabiky do 16bitového slova se zachováním znaménka. **CBW** konkrétně obsah registru AL rozšíří do AX tak, že zkopíruje znaménkový bit AL do všech bitů registru AH.

SYNTAX: ⌜ **CBW**

Instrukce	*Příklad*	*Komentář*
CBW	`CBW`	`; Převede obsah AL do AX se zachováním znaménka`

CWD — *Convert Word to Doubleword*

POPIS:

O	D	I	T	S	Z	A	P	C

Instrukce **CWD** aritmeticky převádí obsah 16bitového slova do 32bitového dvojslova se zachováním znaménka. **CWD** konkrétně obsah registru AX rozšíří do DX&AX tak, že zkopíruje znaménkový bit AX do všech bitů registru DX (tj. vyšší bity výsledku jsou v DX a nižší bity v AX).

SYNTAX: ⌜ **CWD**

Instrukce	*Příklad*	*Komentář*
CWD	`CWD`	`; Převede obsah AX do DX&AX se zachováním znaménka`

CMP — *Compare Two Operands*

POPIS:

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

Instrukce **CMP** nastavuje příznaky, podle kterých lze porovnat obsah zdrojového operandu s obsahem cílového operandu. Příznaky nastaví tak, že odečte zdrojový operand od cílového operandu a podle výsledku nastaví bity příznakového registru. Instrukce neukládá rozdíl do cílového operandu (oba operandy zůstávají po provedení operace nezměněny). Nastavených příznaků lze využít některou instrukcí ze skupiny *Jcond*.

Ve skupině instrukcí podmíněných skoků (*Jcond*) jsou instrukce pro znaménkové a neznaménkové porovnávání. Díky promyšlené konstrukci příznakového registru nastavuje instrukce **CMP** příznaky pro oba typy porovnávání. Znaménkové a neznaménkové porovnávání se liší až použitím instrukce *Jcond*.

SYNTAX: ⌜ **CMP** *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
CMP *r/m8,r8*	`CMP AL,CL`	`; F nastav podle AL - CL`
	Další varianty viz ADD	

AND — *Logical AND*

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

POPIS:

Instrukce **AND** provádí logický součin obsahu zdrojového operandu s obsahem cílového operandu a výsledek uloží do cílového operandu. Operace se provádí po bitech podle následující tabulky:

op_1	op_2	*AND*
0	0	0
0	1	0
1	0	0
1	1	1

SYNTAX: ⌜ **AND** *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
AND *r/m8,imm8*	`AND AL,7`	`; AL := AL ∧ 7`
	`AND Slab,15`	`; Slab := Slab ∧ 15`
AND *r/m16,imm16*	`AND CX,0FFFh`	`; CX := CX ∧ 0FFFh`
	`AND Slovo,1FFFh`	`; Slovo := Slovo ∧ 1FFFh`
AND *r/m16,imm8*	`AND Slovo,0Fh`	`; Slovo := Slovo ∧ 000Fh`
	`AND Slovo,0FFh`	`; Slovo := Slovo ∧ 0FFFFh`
AND *r/m8,r8*	`AND CH,CL`	`; CH := CH ∧ CL`
	`AND Slab,BL`	`; Slab := Slab ∧ BL`
AND *r/m16,r16*	`AND AX,BX`	`; AX := AX ∧ BX`
	`AND Slovo,DX`	`; Slovo := Slovo ∧ DX`
AND *r8,r/m8*	`AND CL,Slab`	`; CL := CL ∧ Slab`
AND *r16,r/m16*	`AND BP,Slovo`	`; BP := BP ∧ Slovo`

Logické instrukce

OR — *Logical Inclusive OR*

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

POPIS:

Instrukce **OR** provádí logický součet obsahu zdrojového operandu s obsahem cílového operandu a výsledek uloží do cílového operandu. Operace se provádí po bitech podle následující tabulky:

op_1	op_2	*OR*
0	0	0
0	1	1
1	0	1
1	1	1

SYNTAX: ⌜ **OR** *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
OR *r/m8,imm8*	`OR AL,7`	`; AL := AL ∨ 7`
	Další varianty viz AND	

XOR — *Logical Exclusive OR*

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

POPIS:

Instrukce **XOR** provádí logickou funkci nonekvivalence (součet modulo 2) obsahu zdrojového operandu s obsahem cílového operandu. Výsledek se uloží do cílového operandu. Operace se provádí po bitech podle následující tabulky:

op_1	op_2	*XOR*
0	0	0
0	1	1
1	0	1
1	1	0

SYNTAX: ⌜ **XOR** *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
XOR *r/m8,imm8*	`XOR AL,7`	`; AL := AL ≠ 7`
	Další varianty viz AND	

NOT — *One's Complement Negation*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce **NOT** nahradí obsah operandu jeho jedničkovým doplňkem. Jedničkový doplněk je inverze všech bitů (všechny jedničky se nahradí nulami a obráceně). Instrukce **NOT** neovlivňuje registr příznaků.

SYNTAX: ⌜ **NOT** *operand*

Instrukce	*Příklad*	*Komentář*
NOT *r/m8*	`NOT AH`	`; AH := ĀH`
	`NOT Slab`	`; Slab := S̄l̄ā̄b̄`
NOT *r/m16*	`NOT SI`	`; SI := S̄Ī`
	`NOT Slovo`	`; Slovo := S̄l̄ō v̄ō`

TEST — *Logical Compare*

O	D	I	T	S	Z	A	P	C
0				*	*	?	*	0

POPIS:

Instrukce TEST provádí logický součin (viz instrukce AND) obsahu zdrojového operandu s obsahem cílového operandu. Výsledek neuloží do cílového operandu, ale podle výsledku nastaví příznaky v příznakovém registru. Oba operandy jsou po provedení instrukce nezměněny.

SYNTAX: ⌜ TEST *cílový_operand,zdrojový_operand*

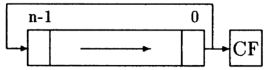
Instrukce	*Příklad*	*Komentář*
TEST *r/m8,imm8*	`TEST AL,7`	`; F := AL ∧ 7`
	`TEST Slab,15`	`; F := Slab ∧ 15`
TEST *r/m16,imm16*	`TEST CX,0FFFh`	`; F := CX ∧ 0FFFh`
	`TEST Slovo,1FFFh`	`; F := Slovo ∧ 1FFFh`
TEST *r/m8,r8*	`TEST CH,CL`	`; F := CH ∧ CL`
	`TEST Slab,BL`	`; F := Slab ∧ BL`
TEST *r/m16,r16*	`TEST AX,BX`	`; F := AX ∧ BX`
	`TEST Slovo,DX`	`; F := Slovo ∧ DX`

Rotace a posuny

ROR — *Rotate Right*

O	D	I	T	S	Z	A	P	C
*								*

POPIS:



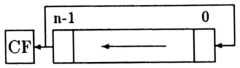
Instrukce ROR provádí rotaci bitů cílového operandu o jeden nebo více bitů vpravo.

Každý bit cílového operandu se posune vpravo, bit nejnižšího řádu se přenese na pozici nejvyššího řádu a současně se zkopíruje do příznakového bitu CF. Zdrojový operand udává počet rotací cílového operandu. Další podrobnosti jsou shodné s instrukcí ROL.

ROL — *Rotate Left*

O	D	I	T	S	Z	A	P	C
*								*

POPIS:



Instrukce ROL provádí rotaci bitů cílového operandu o jeden nebo více bitů vlevo.

Každý bit cílového operandu se posune, bit nejvyššího řádu se přenese na pozici nejnižšího řádu a současně se zkopíruje do příznakového bitu CF. Hodnota zdrojového operandu udává, o kolik bitů se cílový operand rotuje. Na místě zdrojového operandu smí být buď přímý operand s hodnotou 1, nebo odkaz na registr CL, který obsahuje počet bitů. Při rotaci se žádný bit neztrácí.

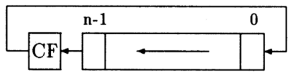
SYNTAX: ⌜ ROL *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
ROL *r/m8,1*	`ROL AL,1`	`; Rotace obsahu AL o 1 bit vlevo`
	`ROL Slab,1`	`; Rotace obsahu Slab o 1 bit vlevo`
ROL *r/m8,CL*	`ROL BH,CL`	`; Rotace BH o CL bitů vlevo`
	`ROL Slab,CL`	`; Rotace obsahu Slab o CL bitů vlevo`
ROL *r/m16,1*	`ROL BP,1`	`; Rotace obsahu BP o 1 bit vlevo`
	`ROL Slovo,1`	`; Rotace obsahu Slovo o 1 bit vlevo`
ROL *r/m16,CL*	`ROL DX,CL`	`; Rotace obsahu DX o CL bitů vlevo`
	`ROL Slovo,CL`	`; Rotace obsahu Slovo o CL bitů vlevo`

RCL — *Rotate Left through Carry*

O	D	I	T	S	Z	A	P	C
*								*

POPIS:

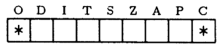


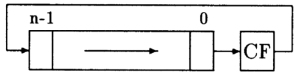
Instrukce RCL provádí rotaci bitů cílového operandu o jeden nebo více bitů vlevo.

Každý bit cílového operandu se posune vlevo, bit nejnižšího řádu se naplní hodnotou příznaku CF a bit z pozice nejvyššího řádu se přepíše do příznaku CF. Další podrobnosti jsou shodné s instrukcí ROL.

RCR — *Rotate Right through Carry*

POPIS:



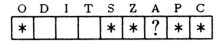


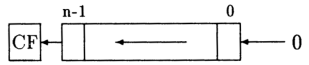
Instrukce RCR provádí rotaci bitů cílového operandu o jeden nebo více bit vpravo.

Každý bit cílového operandu se posune vpravo, bit nejvyššího řádu se naplní hodnotou příznaku CF a bit z pozice nejnižšího řádu se přepíše do příznaku CF. Další podrobnosti jsou shodné s instrukcí ROL.

SAL / SHL — *Shift Arithmetic Left* / *Shift Logical Left*

POPIS:





Obě instrukce SAL a SHL (synonymní označení jedné akce) posouvají všechny bity cílového operandu vlevo.

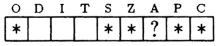
Přitom je původní hodnota nejvyššího bitu cílového operandu přenesena do příznaku CF a nejnižší uvolněný bit je naplněn nulou. Jde o operaci znaménkového aritmetického násobení 2. Hodnota zdrojového operandu udává, o kolik bitů se cílový operand posouvá. Na místě zdrojového operandu smí být buď přímý operand s hodnotou 1, nebo odkaz na registr CL obsahující počet bitů. Při posuvech se „ztrácí“ hodnota alespoň jednoho bitu.

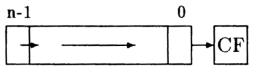
SYNTAX: ⌜ SAL *cílový_operand,zdrojový_operand*
⌜ SHL *cílový_operand,zdrojový_operand*

Varianty instrukce viz ROL.

SAR — *Shift Arithmetic Right*

POPIS:

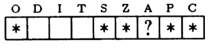


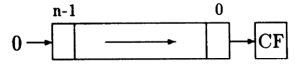


Instrukce SAR posouvá všechny bity cílového operandu vpravo.

SHR — *Shift Logical Right*

POPIS:





Instrukce SHR posouvá všechny bity cílového operandu vpravo.

Při posunutí je nejnižší bit přepsán do příznaku CF a nejvyšší bit je naplněn 0. Při posuvech o jeden bit je do OF nastavena hodnota nejvyššího bitu původního posouvaného operandu. Instrukce SHR nezachovává hodnotu znaménkového bitu, a proto patří do skupiny logických posuvů. Další podrobnosti viz SAL/SHL.

Větvení programu

JMP — *Unconditional Jump*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce nepodmíněného skoku JMP předá řízení instrukci, jejíž adresa je zadána operandem. Rozlišujeme tyto varianty instrukce JMP:

- Přímý skok. V operandu instrukce JMP je některým z následujících způsobů přímo specifikována cílová adresa.
  - Instrukce měnící obsah segmentového registru CS. Touto instrukcí lze předávat řízení na libovolnou adresu v adresovém prostoru. Instrukce JMP obsahuje přímou absolutní adresu (segmentovou i offsetovou část) cílového místa v paměti, kterou se naplní registry CS:IP. Tuto operaci nazýváme **vzdálený skok** (Far Jump).
  - Instrukce neměnící obsah segmentového registru CS. Instrukce obsahuje pouze „vzdálenost" místa, kam se má provést skok, od místa výskytu instrukce. Tato relativní adresa se přičte ke stávajícímu obsahu registru IP. Potom podle délky skoku (tj. rozdílu adresy právě prováděné instrukce JMP a cílové adresy) rozlišujeme další dva typy:
    * Relativní adresa v instrukci JMP je 16bitová. Taková instrukce dovoluje předávat řízení uvnitř celého 64 KB segmentu tím, že k současnému obsahu IP přičte relativní adresu, která je číslo v intervalu 0 až 65 535 (adresa je číslo bez znaménka a ve skocích „vzad" se při přičítání k IP ignoruje přeplnění). Tuto operaci nazýváme **blízký skok** (Near Jump).
    * Relativní adresa v instrukci JMP je 8bitová a je chápána jako číslo se znaménkem. Potom lze řízení předat pouze v intervalu −128 až 127 slabik od místa uložení právě prováděné instrukce JMP. Tuto operaci nazýváme **krátký skok** (Short Jump).
- Nepřímý skok. V tomto případě je cílová adresa popsána buď odkazem na registr, který může obsahovat offsetovou část absolutní cílové adresy (SI, DI, SP, BP nebo BX), nebo přímou adresou místa v paměti, které obsahuje absolutní cílovou adresu (buď jenom offset, nebo segment:offset), nebo kombinací obou způsobů tak, jak to bylo popsáno v odstavci 2.8 od strany 24.

SYNTAX:

```
JMP operand            ; Blízký a nepřímý skok
JMP FAR PTR operand    ; Vzdálený skok
JMP SHORT operand      ; Krátký skok
```

Instrukce	*Příklad*	*Komentář*
JMP *rel8*	`JMP SHORT Navesti`	`; Krátký skok na Navesti`
		`;  IP := IP + vzdálenost Navesti`
JMP *rel16*	`JMP Navesti`	`; Blízký skok na Navesti`
		`;  IP := IP + vzdálenost Navesti`
JMP *ptr16:16*	`JMP FAR PTR Navesti`	`; Vzdálený skok na Navesti`
		`;  CS:IP := segment:offset Navesti`
JMP *r/m16*	`JMP [BX]`	`; Nepřímý skok na adresu v BX`
		`;  IP := BX`
	`JMP [Slovo]`	`; Nepřímý skok podle obsahu Slovo`
		`;  IP := Slovo`
JMP *m16:16*	`JMP [DvojSlovo]`	`; Nepřímý skok podle obsahu DvojSlovo`
		`;  CS:IP := DvojSlovo`

J*cond* — *Conditional Jumps*

O	D	I	T	S	Z	A	P	C

POPIS:

Zkratkou J*cond* jsou zde výjimečně myšleny všechny instrukce podmíněného skoku. Instrukce J*cond* (s výjimkou instrukce JCXZ) testují hodnotu jednoho nebo více jednobitových příznaků příznakového registru.

Odpovídá-li hodnota příznaků kombinaci, která je testována konkrétní instrukcí podmíněného skoku, provede se krátký skok zadaný 8bitovou relativní adresou. Neodpovídá-li, pokračuje se další instrukcí.

Instrukcí podmíněného skoku lze předávat řízení pouze v intervalu −127 až 128 slabik v okolí instrukce J*cond*. Další podrobnosti o krátkém skoku viz instrukce JMP. V instrukci J*cond* se neuvádí SHORT.

Instrukce podmíněného skoku nejčastěji následuje po instrukci CMP, která nastaví příznaky odpovídající rozdílu testovaných operandů. Příznaky jsou konstruovány tak, že operandy vstupující do instrukce CMP mohou být jak čísla bez znaménka, tak i se znaménkem. Byly-li operandy čísla se znaménkem, musí se vybrat instrukce, které jsou dále označeny zkratkou „**z.**". Byly-li operandy čísla bez znaménka, musí se vybrat instrukce označené „**nz.**"

Instrukce JCXZ se liší od ostatních instrukcí podmíněného skoku jen tím, že testuje místo jednobitových příznaků obsah registru CX. Pokud je CX=0, provede se krátký skok na specifikovanou adresu. Instrukce se používá pro řízení cyklů.

SYNTAX: ⌜ *Jcond operand*

Instrukce	*Příklad*	*Komentář*
J*cond rel8*	`JZ Navesti`	`; Krátký podmíněný skok při ZF=1`
JCXZ *rel8*	`JCXZ Navesti`	`; Krátký podmíněný skok při CX=0`

Zkratka instrukce	*Název instrukce (Jump short if ...)*	*Výsledek poslední operace ...*	*Testovaná podmínka*
JE/JZ	equal zero	roven nulový	ZF=1
JNE/JNZ	not equal not zero	různý nenulový	ZF=0
JP/JPE	parity parity even	sudé parity	PF=1
JNP/JPO	not parity parity odd	liché parity	PF=0
JS	sign	záporný	SF=1
JNS	not sign	kladný nebo nulový	SF=0
JC	carry	nastal přenos	CF=1
JNC	not carry	nenastal přenos	CF=0
JO	overflow	nastalo přeplnění	OF=1
JNO	not overflow	nenastalo přeplnění	OF=0

JB/JNAE	below not above nor equal	nz. menší	CF=1
JAE/JNB	above or equal not below	nz. větší nebo roven	CF=0
JBE/JNA	below or equal not above	nz. menší nebo roven	(CF=1)∨ ∨(ZF=1)
JA/JNBE	above not below nor equal	nz. větší	(CF=0)∧ ∧(ZF=0)
JL/JNGE	less not greater nor equal	z. menší	SF≠OF
JGE/JNL	greater or equal not less	z. větší nebo roven	SF=OF
JLE/JNG	less or equal not greater	z. menší nebo roven	(ZF=1)∨ ∨(SF≠OF)
JG/JNLE	greater not less nor equal	z. větší	(ZF=0)∨ ∨(SF=OF)

Porovnávání typu menší a větší je rozděleno na neznaménkové a znaménkové. Neznaménkové porovnávání (v přehledu uvedeno zkratkou „**nz.**“ a v originále slovy „above“ a „below“) je určeno pro čísla bez znamének (bit nejvyššího řádu se neuvažuje jako znaménkový). Znaménkové porovnávání (zkratka „**z.**“ a v originále „less“ a „greater“) probíhá podle pravidel dvojkového doplňkového kódu s respektováním znamének.

CALL — *Call Procedure*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce CALL předá řízení podprogramu, jehož počáteční adresa je specifikována operandem instrukce a současně do zásobníku uloží návratovou adresu (tj. adresu ukazující bezprostředně za tuto instrukci CALL). Návrat z podprogramu provede instrukce RET.

Různé varianty instrukce CALL vyplývají z různých mechanismů zadávání cílové adresy a jsou shodné s instrukcí JMP kromě varianty krátkého skoku. Oproti instrukci JMP musí CALL ukládat do zásobníku návratovou adresu. V případě blízkého volání podprogramu se ukládá pouze offsetová část návratové adresy. U vzdáleného volání se do zásobníku vloží nejprve současný obsah registru CS, a teprve potom offsetová část návratové adresy.

SYNTAX: ⌜ CALL *operand* ; Blízké a nepřímé volání
CALL FAR PTR *operand* ; Vzdálené volání

Instrukce	*Příklad*	*Komentář*
CALL *rel16*	`CALL Navesti`	`; Blízké volání Navesti` `; IP := IP + vzdálenost Navesti`
CALL *ptr16:16*	`CALL FAR PTR Navesti`	`; Vzdálené volání Navesti`

```
                                ;  CS:IP := segment:offset Navesti
CALL r/m16      CALL [BX]       ; Nepřímé volání podle obsahu BX
                                ;  IP := BX
                CALL [Slovo]    ; Nepřímý skok podle obsahu Slovo
                                ;  IP := Slovo
CALL m16:16     CALL [DvojSlovo]; Nepřímý skok podle DvojSlovo
                                ;  CS:IP := DvojSlovo
```

Další podrobnosti viz JMP.

RET / RETF

Near Return From Procedure
Far Return From Procedure

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce RET vrací řízení z podprogramu volaného instrukcí CALL. Návratovou adresu převezme z vrcholu zásobníku.

Z podprogramu volaného blízkým voláním CALL (v zásobníku je uložena jenom offsetová část návratové adresy) se návrat musí provést pouze instrukcí blízkého návratu RET (ze zásobníku převezme pouze jedno 16bitové slovo jako offsetovou část návratové adresy). Registr CS se provedením instrukce RET nezmění.

Z podprogramu volaného vzdáleným voláním CALL (v zásobníku je uložen jak segment, tak i offset návratové adresy) se návrat musí provést pouze instrukcí vzdáleného návratu RETF (ze zásobníku převezme dvě 16bitová slova: nejprve offsetovou, pak segmentovou část návratové adresy), která návratovou adresou naplní registry CS:IP.

Za instrukcí RET/RETF se může jako přímá hodnota uvést číslo, které udává, kolik slabik ze zásobníku se má po vyzvednutí návratové adresy dodatečně odstranit. Této funkce se většinou využívá k odstranění parametrů předávaných podprogramu pomocí zásobníku. Odstranění se provede přičtením operandu k registru SP.

SYNTAX:
```
RET
RET  operand
RETF
RETF operand
```

Instrukce	*Příklad*	*Komentář*
RET	`RET`	`; Návrat z podprogramu`
RET *imm16*	`RET 2`	`; Návrat a odstranění 2 slabik ze zásobníku`
RETF	`RET`	`; Vzdálený návrat z podprogramu`
RETF *imm16*	`RET 6`	`; Vzdálený návrat a odstranění 6 slabik ze zás.`

Poznamenejme, že chybná kombinace typů instrukcí CALL (blízké nebo vzdálené volání) a RET povede ke zhroucení programu.

Zásobník a příznakový registr

PUSH

Push a Word onto the Stack

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce PUSH uloží hodnotu operandu do zásobníku. Operaci provede ve dvou krocích (viz též str. 19): sníží obsah registru ukazatele vrcholu zásobníku (SP) o 2 a na nový vrchol zásobníku (SS:SP) uloží obsah operandu. Operandem instrukce může být 16bitové slovo v paměti nebo 16bitový registr (všeobecný nebo segmentový).

SYNTAX:
```
PUSH operand
```

Instrukce	*Příklad*	*Komentář*
PUSH *m16*	`PUSH Slovo`	`; Do zásobníku uloží obsah Slovo`
PUSH *r16*	`PUSH AX`	`; Do zásobníku uloží obsah AX`
PUSH *Sreg*	`PUSH CS`	`; Do zásobníku uloží obsah CS`

POP — *Pop a Word from the Stack*

O	D	I	T	S	Z	A	P	C

POPIS:

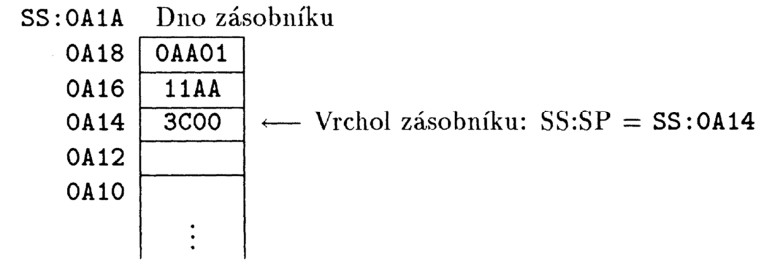
Instrukce POP vybere slovo ze zásobníku a uloží je do operandu instrukce. Činnost provede ve dvou krocích (viz též str. 19): z vrcholu zásobníku (SS:SP) vezme slovo a naplní jím operand, registr ukazatele vrcholu zásobníku zvýší o 2. Operandem instrukce může být 16bitový všeobecný registr, segmentový registr (**kromě CS**) nebo 16bitové slovo v paměti.

SYNTAX: ⌜ POP *operand*

Instrukce	*Příklad*	*Komentář*
POP *m16*	`POP Slovo`	`; Vybere hodnotu a uloží ji do Slovo`
POP *r16*	`POP BX`	`; Vybere hodnotu a uloží ji do BX`
POP DS	`POP DS`	`; Vybere hodnotu a uloží ji do DS`
POP ES	`POP ES`	`; Vybere hodnotu a uloží ji do ES`
POP SS	`POP SS`	`; Vybere hodnotu a uloží ji do SS`

Instrukce POP SS zakazuje všechna přerušení (vč. NMI) do skončení následující instrukce. Předpokládá se, že bude následovat instrukce POP SP. Tím procesor umožní naplnit registry SS:SP, aniž by na krátký okamžik byl k dispozici chybný ukazatel do zásobníku.

Určení vrcholu zásobníku pomocí SS:SP



PUSHF — *Push Flag Register onto the Stack*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce PUSHF ukládá na vrchol zásobníku registr příznaků (viz též instrukci PUSH a tvar registru příznaků na str. 17).

SYNTAX: ⌜ PUSHF

Instrukce	*Příklad*	*Komentář*
PUSHF	`PUSHF`	`; Do zásobníku uloží obsah registru F`

POPF — *Pop from the Stack into the Flag Register*

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

POPIS:

Instrukce POP vybere z vrcholu zásobníku hodnotu a uloží ji do registru příznaků (viz též instrukci POP a tvar registru příznaků na str. 17).

SYNTAX: ⌜ POPF

Instrukce	*Příklad*	*Komentář*
POPF	`POPF`	`; Vybere hodnotu a uloží do registru F`

STx / CLx — *Set x Flag* / *Clear x Flag*

O	D	I	T	S	Z	A	P	C
	*	*						*

POPIS:

Do této skupiny patří instrukce nastavující (STx) a nulující (CLx) příznaky **I**F, **D**F a **C**F z příznakového registru.

SYNTAX: ⌜ STx ; Jedničkování příznaku *x*
CLx ; Nulování příznaku *x*

Instrukce	*Příklad*	*Komentář*
STI	`STI`	`; IF := 1 (povolení přerušení)`
STD	`STD`	`; DF := 1 (zpracovávání řetězců odzadu)`
STC	`STC`	`; CF := 1 (nastavení příznaku přenosu)`
CLI	`CLI`	`; IF := 0 (zákaz přerušení)`
CLD	`CLD`	`; DF := 0 (zpracování řetězců odpředu)`

Instrukce	*Příklad*	*Komentář*
CLC	`CLC`	`; CF := 0 (nulování příznaku přenosu)`

Je-li přerušení zakázáno po delší dobu, je pravděpodobné, že se budou požadavky na přerušení, přicházející od různých zdrojů, hromadit (přijde-li další přerušení od stejného zdroje, předchozí se ztratí). Z toho vyplývá, že po odblokování přerušovacího systému se s poměrně velkou pravděpodobností uplatní čekající přerušení. Aby se v zásobníku nehromadilo příliš mnoho adres přerušených procesů, povolí instrukce STI přerušení **až po dokončení následující instrukce**. Předpokládá se, že za touto instrukcí následuje návrat do vyšší vrstvy (RET), čímž se ze zásobníku průběžně odstraňují adresy přerušených procesů.

Přerušení se povolí po provedení následující instrukce jenom tehdy, pokud jej tato nezakázala. Např. během provádění posloupnosti instrukcí `STI` a `CLI` (ani po provedení posloupnosti) není přerušení povoleno.

CMC — *Complement Carry Flag*

O	D	I	T	S	Z	A	P	C
								*

POPIS:

Instrukce CMC invertuje obsah bitu CF v registru příznaků F. Pokud byla hodnota CF=1, pak nastaví CF=0, a naopak.

SYNTAX: ⌜ CMC

Instrukce	*Příklad*	*Komentář*
CMC	`CMC`	`; CF := CF`

LAHF — *Load Flags into AH Register*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce LAHF kopíruje nižších 8 bitů příznakového registru do registru AH (viz tvar registru příznaků na str. 17).

SYNTAX: ⌜ LAHF

Instrukce	*Příklad*	*Komentář*
LAHF	`LAHF`	`; Uloží do AH nižší slabiku registru F`

SAHF — *Store AH Register into Flags Register*

O	D	I	T	S	Z	A	P	C
				*	*	*	*	*

POPIS:

Instrukce SAHF naplní dolních 8 bitů příznakového registru hodnotou uloženou v registru AH (viz tvar registru příznaků na str. 17).

SYNTAX: ⌜ SAHF

Instrukce	*Příklad*	*Komentář*
SAHF	`SAHF`	`; Zapíše do nižší slabiky F registr AH`

Přerušovací systém

INT — *Call to Interrupt Procedure*
INTO — *Call to Interrupt Procedure if Overflow*

O	D	I	T	S	Z	A	P	C
		0	0					

POPIS:

Instrukce INT generuje programové přerušení. Operand instrukce (0 až 255) je použit jako index do tabulky přerušovacích vektorů (viz obr. 2.6 na str. 20), kde je uložena adresa rutiny obsluhující přerušení odkazované operandem. V rámci této instrukce se do zásobníku uloží registr příznaků, CS:IP **ukazující na instrukci následující** za INT, vynulují se příznaky IF a TF a naplní se registry CS:IP (viz též str. 20). Instrukce se používá zpravidla pro volání služeb operačního systému nebo do počítače vestavěného programového vybavení (jde o zvláštní formu vzdáleného volání podprogramu, z něhož se návrat provádí instrukcí IRET).

Instrukce INTO je jednoslabiková instrukce (operační kód 0CEh) generující přerušení číslo 4, pokud je nastaven příznak OF v příznakovém registru. Instrukce využívá programové vybavení pro jednoduché testování přeplnění po provedení aritmetických operací.

Další variantou je opět jednoslabiková instrukce INT 3 s operačním kódem 0CCh. Tato se používá pro aktivaci ladicího systému (ladicí bod – Breakpoint). Ostatní instrukce INT *n* jsou dvouslabikové.

SYNTAX: ⌜ INT *operand*
INTO

Instrukce	*Příklad*	*Komentář*
INT *imm8*	`INT 10h`	`; Vyvolání přerušení číslo 10h`
INTO	`INTO`	`; Vyvolání přerušení číslo 4 při OF=1`

IRET — *Return from Interrupt*

O	D	I	T	S	Z	A	P	C
*	*	*	*	*	*	*	*	*

POPIS:

Instrukce IRET provádí návrat z rutiny vyvolané přerušením nebo instrukcí INT. Z vrcholu zásobníku jsou instrukcí odstraněna tři slova a uložena do registrů IP, CS a F (viz též str. 21).

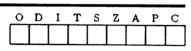
SYNTAX: ⌜ IRET

Instrukce	*Příklad*	*Komentář*
IRET	`IRET`	`; Návrat z přerušení`

Cykly

LOOP — *Unconditional Loop Control*

POPIS:



Instrukce LOOP podporuje konstruování nepodmíněných cyklů. Používá registr CX jako čítač počtu průchodů cyklem. Činnost instrukce si vysvětlíme na příkladu:

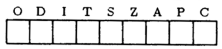
```
        MOV  CX,počet_průchodů
Opakuj:
         :          ; Tělo cyklu.
        LOOP Opakuj ; CX:=CX-1 a je-li CX nenulové
                    ;   provede se SHORT skok na Opakuj.
        ; Zde, je-li po odečtení jedničky CX=0.
```

Při provádění instrukce LOOP je nejdříve snížen obsah registru CX o jedničku a pak je CX testován. V případě, že CX je různý od nuly, provede se krátký skok na návěští uvedené za instrukcí. Návěští reprezentuje 8bitovou hodnotu se znaménkem, která určuje vzdálenost cílového návěští od instrukce LOOP. Proto vzdálenost návěští musí být v rozmezí −128 až 127 slabik. V případě, že CX je roven nule, pokračuje zpracování následující instrukcí.

SYNTAX: ⌜ LOOP *operand*

LOOP*cond* — *Conditional Loop Control*

POPIS:



Instrukce podmíněného provádění cyklu je rozšířením instrukce LOOP. Tyto instrukce provedou krátký skok na návěští tehdy, je-li po odečtení jedničky CX≠0 a zároveň je splněna zadaná podmínka.

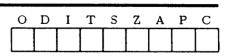
SYNTAX: ⌜ LOOP*cond* *operand*

Instrukce	*Příklad*	*Komentář*
LOOPE *rel8*	`LOOPE Navesti`	`; CX:=CX-1 a skok při (CX≠0)∧(ZF=1)`
LOOPZ *rel8*	`LOOPZ Navesti`	`; CX:=CX-1 a skok při (CX≠0)∧(ZF=1)`
LOOPNE *rel8*	`LOOPNE Navesti`	`; CX:=CX-1 a skok při (CX≠0)∧(ZF=0)`
LOOPNZ *rel8*	`LOOPNZ Navesti`	`; CX:=CX-1 a skok při (CX≠0)∧(ZF=0)`

Ovládání V/V

IN — *Input from Port*

POPIS:



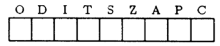
Instrukce IN přenáší data o délce slabiky nebo slova z V/V brány do registru AL nebo AX podle cílového operandu. Zdrojový operand určuje adresu V/V brány. Zde může být zadána buď přímá 8bitová hodnota bez znaménka (0 až 255), nebo registr DX (obsahující hodnotu 0 až 65 535). Pokud je použita přímá 8bitová hodnota, je vyšší slabika adresy brány nulová. Přenáší-li se slovo, uloží se do cílového registru obsahy bran na adresách *zdrojový_operand* a *zdrojový_operand+1*.

SYNTAX: ⌜ IN *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
IN AL,*imm8*	`IN AL,0Fh`	`; Přenos slabiky z brány 0Fh do AL`
IN AX,*imm8*	`IN AX,0Fh`	`; Přenos slova z bran 0Fh a 10h do AX`
IN AL,DX	`IN AL,DX`	`; Přenos slabiky z brány podle DX do AL`
IN AX,DX	`IN AX,DX`	`; Přenos slova z brány podle DX do AX`

OUT — *Output to Port*

POPIS:



Instrukce OUT přenáší data v délce slabiky nebo slova z registru AL nebo AX do V/V brány. Cílový operand určuje adresu V/V brány (viz instrukci IN).

SYNTAX: ⌜ OUT *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
OUT *imm8*,AL	`OUT 0Fh,AL`	`; Přenos slabiky z AL do brány 0Fh`
OUT *imm8*,AX	`OUT 0Fh,AX`	`; Přenos slova z AX do bran 0Fh a 10h`
OUT DX,AL	`OUT DX,AL`	`; Přenos slabiky z AL do brány podle DX`
OUT DX,AX	`OUT DX,AX`	`; Přenos slova z AX do brány podle DX`

Přesuny dat

XCHG — *Exchange Memory/Register with Register*

O D I T S Z A P C

POPIS:

Instrukce XCHG vzájemně zamění obsahy zdrojového a cílového operandu. Výměna je možná mezi dvěma registry nebo mezi registrem a pamětí.

SYNTAX: ⌜ XCHG *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
XCHG *r/m8,r8*	`XCHG AL,AH`	`; Zamění obsahy AH a AL`
	`XCHG Slab,BH`	`; Zamění obsahy BH a Slab`
XCHG *r8,r/m8*	`XCHG CL,Slab`	`; Zamění obsahy Slab a CL`
XCHG *r/m16,r16*	`XCHG AX,BX`	`; Zamění obsahy BX a AX`
	`XCHG Slovo,BP`	`; Zamění obsahy BP a Slovo`
XCHG *r16,r/m16*	`XCHG DX,Slovo`	`; Zamění obsahy Slovo a DX`

Je-li jeden operand v paměti, je po dobu trvání instrukce blokována sběrnice signálem $\overline{\text{LOCK}}$ bez ohledu na uvedení nebo neuvedení instrukčního prefixu LOCK.

XLAT — *Table Lookup Translation*

O D I T S Z A P C

POPIS:

Instrukce XLAT transformuje obsah AL podle tabulky. Před provedením instrukce musí být v DS:BX uložena adresa tabulky a v registru AL index (neznaménkové číslo 0 až 255), který se sečte s obsahem registru BX. Slabikou z vypočtené adresy se naplní opět registr AL.

Instrukce se používá pro transformaci mezi kódy (např. pro převod mezi ASCII a EBCDIC).

SYNTAX: ⌜ XLAT

Instrukce	*Příklad*	*Komentář*
XLAT	`XLAT`	`; AL := DS:[BX+AL]`

Některé asemblery pro tuto instrukci vyžadují zadání operandu, který slouží pouze pro kontrolu typů. Operand musí být typu slabika. Instrukce XLATB bývá zpravidla instrukce neprovádějící kontrolu a nevyžadující operand.

LEA — *Load Effective Address*

O D I T S Z A P C

POPIS:

Instrukce LEA ukládá do cílového operandu, kterým je jeden z všeobecných 16bitových registrů, offsetovou část adresy zdrojového operandu, na jehož místě je uložena adresa objektu v paměti. LEA je rozšířením instrukce MOV *r16,imm16* (v asemblerech se tato instrukce zapisuje např. `MOV BX,OFFSET Slovo`) o možnost zadat adresu ve zdrojovém operandu indexované nebo bázované podle pravidel z části 2.8.

SYNTAX: ⌜ LEA *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
LEA *r16,m*	`LEA BX,Slab[BP][SI]`	`; Naplní BX offsetem` `;  adresy Slab[BP][SI]`

LDS — *Load Doubleword Pointer Using DS*
LES — *Load Doubleword Pointer Using ES*

O D I T S Z A P C

POPIS:

Instrukce LDS a LES naplňují segmentový registr (DS a ES) a současně jeden z 16bitových všeobecných registrů 32bitovou adresou (segment a offset) uloženou v paměti v objektu specifikovaném zdrojovým operandem.

První 16bitové slovo 32bitové adresy v paměti (offset) je uloženo do registru, který je specifikován cílovým operandem, druhé slovo adresy (segment) je uloženo do segmentového registru určeného mnemonikou instrukce (tzn. DS pro LDS a ES pro LES).

SYNTAX: ⌜ LDS *cílový_operand,zdrojový_operand*
LES *cílový_operand,zdrojový_operand*

Instrukce	*Příklad*	*Komentář*
LDS *r16,m16:16*	`LDS BX,DvojSlovo[SI]`	`; DS:BX := DvojSlovo[SI]`
LES *r16,m16:16*	`LES DI,DvojSlovo[4]`	`; ES:DI := DvojSlovo[4]`

Řetězcové instrukce

Instrukce ze skupiny řetězcových instrukcí pracují zpravidla se dvěma místy v paměti. Zdrojové místo v paměti bývá adresováno dvojicí DS:SI a cílové místo dvojicí ES:DI. Poněvadž řetězce jsou určeny adresami uloženými v těchto registrech, instrukce nepotřebují operandy. Řetězcové instrukce pracují buď se slabikami, nebo s 16bitovými slovy. Jsou-li v instrukci operandy, používá je asembler pro kontrolu typů a pro generování instrukce určené pro práci buď se slabikami, nebo se slovy. Instrukce s názvem končícím písmenem „B“ nevyžaduje operandy, protože předpokládá, že obsah dvojice DS:SI a ES:DI ukazuje na slabikovou strukturu. Instrukce, jejíž název končí „W“, pracuje s 16bitovými slovy. Třetí typ instrukcí (nekončící „B“ ani „W“) vyžaduje operandy. Překladač zkontroluje, jsou-li operandy stejného typu (oba slabika nebo slovo), a podle typu vygeneruje buď instrukci pracující se slabikami, nebo se slovy.

Po provedení instrukce se upravuje obsah SI a DI tak, aby ukazoval na další zpracovávaný prvek. Obsah SI a DI se zvyšuje nebo zmenšuje o velikost zpracovávaného prvku (o 1 pro slabiku a o 2 pro slovo). Zda se řetězce zpracovávají zepředu nebo zezadu, určuje momentální hodnota příznaku DF. Hodnota DF=0 způsobí zvětšování obsahu SI a DI po provedení operace a hodnota DF=1 zmenšování obsahu SI a DI o velikost zpracovávaného prvku. Pří

celý řetězec prvků, můžeme použít prefix REP nebo REP*cond* (viz též str. 66).

der

re-

áz

MOVS/MOVSB/MOVSW *Move String*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce MOVS přesouvá obsah slabiky (slova) z adresy DS:SI do paměťového místa s adresou ES:DI. Instrukce může být uvedena buď bez operandů (MOVSB/MOVSW), nebo s operandy (MOVS). Pokud není v instrukci uveden explicitně u zdrojového operandu segmentový registr, je použit DS. Cílový operand musí být vždy adresován segmentovým registrem ES. Při použití instrukce s operandy je překladačem kontrolován typ operandů a podle délky (slabika, slovo) je generována odpovídající instrukce. Jiný význam operandy nemají. Je-li příznak DF=0, pak jsou obsahy obou registrů SI a DI po provedení přenosu zvětšeny, v opačném případě se zmenšují. Hodnota, o kterou se obsah registrů zvětšuje nebo zmenšuje, je 1 pro slabikové operandy a 2 pro slovní operandy.

SYNTAX: ⌜ MOVS *cílový_operand,zdrojový_operand*
MOVSB
MOVSW

Instrukce	*Příklad*	*Komentář*
MOVSB	`MOVSB`	`; Slabiku z adresy DS:SI přepíše na adresu ES:DI`
MOVSW	`MOVSW`	`; Slovo z adresy DS:SI přepíše na adresu ES:DI`

CMPS/CMPSB/CMPSW *Compare String Operands*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce CMPS porovnává slabiku (slovo) cílového operandu (na adrese ES:DI) se zdrojovým operandem (na adrese DS:SI). Porovnání se provede tak, že se odečte cílový operand od zdrojového, tj. (obsah podle DS:SI) – (obsah podle ES:DI). V této instrukci, na rozdíl od konvencí firmy Intel, je zaměněno pořadí zdrojového a cílového operandu.

Podle výsledku rozdílu se nastaví obsah příznakového registru a hodnota cílového operandu se nemění. Po provedení porovnání se aktualizují registry SI a DI.

SYNTAX: ⌜ CMPS *zdrojový_operand,cílový_operand*
CMPSB
CMPSW

Instrukce	*Příklad*	*Komentář*
CMPSB	`CMPSB`	`; F:=(obsah podle DS:SI)-(obsah podle ES:DI)`
CMPSW	`CMPSW`	`; F:=(obsah podle DS:SI)-(obsah podle ES:DI)`

SCAS/SCASB/SCASW — *Compare String Data*

O	D	I	T	S	Z	A	P	C
*				*	*	*	*	*

POPIS:

Instrukce SCAS porovnává prvek řetězce s obsahem registru AL nebo AX. Je-li prvkem řetězce slabika, pracuje se s registrem AL, je-li prvkem slovo, použije se registr AX. Porovnání se provede tak, že se od obsahu registru AL (AX) odečte obsah prvku řetězce na adrese ES:DI. Podle výsledku se nastaví příznakový registr, rozdíl se nikam neukládá. Po provedení porovnání se aktualizuje registr DI.

SYNTAX: ⌜ SCAS *cílový_operand*
SCASB
SCASW

Instrukce	*Příklad*	*Komentář*
SCASB	`SCASB`	`; F := AL - (obsah podle ES:DI)`
SCASW	`SCASW`	`; F := AX - (obsah podle ES:DI)`

STOS/STOSB/STOSW — *Store String Data*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce STOS uloží obsah registru AL nebo AX do prvku řetězce na adrese ES:DI. Po provedení operace se aktualizuje registr DI.

SYNTAX: ⌜ STOS *cílový_operand*
STOSB
STOSW

Instrukce	*Příklad*	*Komentář*
STOSB	`STOSB`	`; Podle ES:DI := AL`
STOSW	`STOSW`	`; Podle ES:DI := AX`

LODS/LODSB/LODSW — *Load String Operand*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce LODS naplní registr AL nebo AX obsahem prvku řetězce uloženého na adrese DS:SI. Po provedení operace se aktualizuje registr SI.

SYNTAX: ⌜ LODS *zdrojový_operand*
LODSB
LODSW

Instrukce	*Příklad*	*Komentář*
LODSB	`LODSB`	`; AL := (obsah podle DS:SI)`
LODSW	`LODSW`	`; AX := (obsah podle DS:SI)`

REP/REP*cond* — *Repeat Following String Operation*

O	D	I	T	S	Z	A	P	C
					*			

POPIS:

Instrukční prefix REP lze použít s takovou instrukcí ze skupiny řetězcových operací, která nenastavuje příznaky. Prefix způsobí opakované provádění instrukce. Při každém opakování se zmenší obsah registru CX o jedničku. Prefix instrukci opakuje tak dlouho, dokud je CX nenulové. Podmíněné varianty prefixu REP*cond* testují navíc ještě příznak ZF a má význam je používat s instrukcemi CMPS nebo SCAS, které nastavují příznaky.

Přesný algoritmus činnosti prefixů REP je následující:

1. Testování CX. Je-li CX=0, provede se skok na další instrukci (opakování se již neprovádí).
2. Test, nežádá-li někdo o přerušení. Pokud ano, přerušení se uplatní.
3. Jedno provedení požadované instrukce.
4. Snížení CX o jedničku. Toto odečtení nemění nastavení registru příznaků a nemá na něj vliv ani hodnota příznaku DF.

5. Je-li prefix nepodmíněný (REP), pokračuje se v algoritmu bodem 1. Je-li prefix podmíněný, testuje se nastavení příznaku ZF. Je-li prefix REPE nebo REPZ a ZF=0 (poslední porovnávání detekovalo nerovnost), provede se skok na další instrukci (opakování se již neprovádí). Je-li prefix REPNE nebo REPNZ a ZF=1 (poslední porovnávání detekovalo rovnost), provede se skok na další instrukci. V opačných případech se v algoritmu pokračuje bodem 1.

Příklad:

```
LDS  SI,adresa_zdrojového_řetězce
LES  DI,adresa_cílového_řetězce
MOV  CX,počet_prvků_řetězce
REP  MOVSW   ; Zkopíruje prvky řetězce typu slovo.
```

Příklad:

```
LDS  SI,adresa_zdrojového_řetězce
LES  DI,adresa_cílového_řetězce
MOV  CX,počet_prvků_řetězce
REPE CMPSB  ; Srovnává obsahy dvou řetězců typu slabika.
JE   Shodne ; Řetězce mají shodný obsah.
; Řetězce se liší.
```

REP a REP*cond* jsou pouze symbolické zápisy skutečných instrukčních prefixů, které bývají uvedeny před operačním kódem instrukce. Operační kódy diskutovaných prefixů jsou: F3h (REP, REPE) a F2h (REPNE).

Instrukce BCD aritmetiky

Instrukce BCD (Binary Coded Decimal – dvojkově kódované desítkové číslo) aritmetiky pracují se dvěma typy dat. V obou typech budeme pracovat s pojmem **BCD číslice**, což je číslice v intervalu 0 až 9 uložená ve čtyřech bitech. Tzn., že se v těchto bitech **nesmí** vyskytnout kombinace v intervalu 10 až 15. Čtyři bity (tj. polovina slabiky) čísel 0 až 3 nebo 4 až 7 budeme nazývat **půlslabika** (Nibble).

Nezhuštěný tvar (Unpacked Decimal) je tvar, ve kterém je v jedné slabice uložena jedna BCD číslice. Číslice je uložena v dolní půlslabice (v dolních 4 bitech), horní půlslabika musí být pro operace násobení a dělení nulová, pro ostatní operace může mít libovolný obsah.

Zhuštěný tvar (Packed Decimal) je tvar, ve kterém jsou v jedné slabice uloženy dvě BCD číslice. Každá číslice v jedné půlslabice. Číslice vyššího řádu je uložena ve vyšší půlslabice. Slabika ve zhuštěném BCD tvaru může tedy obsahovat číslo v intervalu 0 až 99.

Instrukce BCD aritmetiky používají příznak AF (Auxiliary Carry) příznakového registru. Do tohoto příznaku se ukládá přenos z nejnižší do vyšší půlslabiky (tedy vždy z bitu 3 do bitu 4 bez ohledu na šířku operandu).

Procesor přímo nenabízí instrukce pro operace (sčítání, násobení atd.) v BCD kódu, poskytuje pouze sadu instrukcí, které připraví data v BCD kódu do tvaru vhodného pro zpracování klasickými (ne BCD) instrukcemi a které po zpracování výsledek převedou zpět do BCD kódu.

Instrukce této skupiny nemají operand. Proto nebudeme uvádět ani zápis syntaxe ani příklady.

AAA — *ASCII Adjust after Adition*

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

POPIS:

Instrukce **AAA** se používá následně po **ADD**, která sčítala dvě BCD číslice v nezhuštěném tvaru a součet uložila v binárním tvaru do registru AL. V registru AL na vstupu této instrukce může teoreticky být číslo v intervalu 0 až 19.

AAA převádí binární obsah AL na BCD číslici a případný přenos do vyššího BCD řádu následovně:

Je-li nastaven příznak AF nebo číslo v AL je > 9, zvýší se obsah AH o jedničku, AL se zvýší o 6 (konverze zpět na BCD číslici), vynuluje se horní půlslabika AL a nastaví se AF=CF=1.

Není-li nastaven příznak AF a zároveň číslo v AL je < 10, nemění se obsah AX a nastaví se AF=CF=0.

Pro převedení BCD číslice v AL na ASCII znak je možné po instrukci **AAA** použít instrukci např. `OR AL,30H`.

AAD — *ASCII Adjust AX before Division*

O	D	I	T	S	Z	A	P	C
?				*	*	?	*	?

POPIS:

Instrukce **AAD** převádí dvě BCD číslice v nezhuštěném tvaru uložené v registru AX (v AH je číslice vyššího řádu) na binární číslo do registru AX.

Převod se provádí tak, že hodnota registru AH vynásobená číslem 10 se přičte k obsahu registru AL. Registr AH se nuluje. Tato instrukce se většinou používá před instrukcí **DIV**.

AAM — *ASCII Adjust AX after Multiplication*

O	D	I	T	S	Z	A	P	C
?				*	*	?	*	?

POPIS:

Instrukce **AAM** se používá po instrukci násobení dvou BCD číslic v nezhuštěném tvaru, která uložila výsledek do registru AX. **AAM** převádí binární hodnotu AL (protože výsledek násobení nemůže být větší než rozsah zobrazení slabiky) do dvou BCD číslic uložených v registrech AH (vyšší řád) a AL.

Obsah registru AX dělí číslem 10, celá část výsledku se uloží do registru AH a zbytek po dělení do registru AL.

AAS — *ASCII Adjust AL after Subtraction*

O	D	I	T	S	Z	A	P	C
?				?	?	*	?	*

POPIS:

Instrukce **AAS** se používá po instrukci odečítání dvou BCD číslic v nezhuštěném tvaru, která uložila výsledek do registru AL. **AAS** převádí výsledek v AL na dekadickou číslici uloženou ve spodních čtyřech bitech registru AL.

Převod probíhá podobně jako u instrukce **AAA**. Je-li nastaven příznak AF nebo číslo v AL je > 9, sníží se obsah AH o jedničku, AL se sníží o 6 (konverze zpět na BCD číslici), vynuluje se horní půlslabika AL a nastaví se AF=CF=1. Není-li nastaven příznak AF a zároveň číslo v AL je < 10, nemění se obsah AX a nastaví se AF=CF=0.

DAA — *Decimal Adjust AL after Adition*

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

POPIS:

Instrukce DAA se užívá po instrukci sečtení dvou BCD čísel ve zhuštěném tvaru, která uložila výsledek do registru AL. DAA převádí binární obsah registru AL na dvě BCD číslice ve zhuštěném tvaru a ukládá jej zpět do AL.

Pokud je hodnota dolní půlslabiky AL > 9 nebo je nastaven příznak AF, pak je zvýšen obsah AL o 6 a příznak AF je nastaven na 1.

Následně pokud je výsledek předcházející operace >9Fh nebo je nastaven příznak CF, pak je zvýšen obsah AL o 60h a CF se nastaví na 1.

DAS — *Decimal Adjust AL after Subtraction*

O	D	I	T	S	Z	A	P	C
?				*	*	*	*	*

POPIS:

Instrukce DAS se používá po instrukci odečtení dvou BCD čísel ve zhuštěném tvaru, která uložila výsledek do registru AL. Instrukce DAS převádí binární obsah registru AL na dvě BCD číslice ve zhuštěném tvaru a ukládá jej do AL.

Pokud je hodnota dolní půlslabiky AL > 9 nebo je nastaven příznak AF, pak je zmenšen obsah AL o 6 a příznak AF je nastaven na 1.

Následně pokud je výsledek předcházeiící operace >9Fh nebo je nastaven příznak CF, pak je zmenšen obsah AL o 60h a CF se nastaví na 1.

Řídící instrukce

NOP — *No Operation*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce NOP neprovádí žádnou akci. Má délku 1 slabiky nesoucí operační kód 90h (stejný operační kód má instrukce XCHG AX,AX). Hojně se užívá tam, kde je potřeba doplnit slabiku na zarovnání adresy následující instrukce na hodnotu dělitelnou 2.

SYNTAX: ⌜ NOP

HLT — *Halt CPU*

O	D	I	T	S	Z	A	P	C

POPIS:

Instrukce HLT zastaví provádění instrukcí a uvede procesor do stavu, ze kterého jej vyvede pouze NMI (nemaskovatelné přerušení), vnější přerušení (je-li povoleno) nebo signál RESET. Je-li činnost obnovena pomocí přerušení, tak se tímto přerušením do zásobníku uloží adresa instrukce následující za instrukcí HLT. Návratem z přerušovací rutiny (IRET) pokračuje provádění instrukcí za HLT.

SYNTAX: ⌜ HLT

ESC — *Escape*

POPIS:

O	D	I	T	S	Z	A	P	C

Pomocí ESC procesor rozpoznává, že jde o příkaz určený externímu koprocesoru (např. 8087).

Příkazy pro koprocesor jsou součástí instrukcí pro procesor 8086. Liší se pouze tím, že jsou uvedeny vzorkem ESC. Jeho výskyt procesoru 8086 oznamuje, že tuto instrukci nemá zpracovat, ale že ji má předat koprocesoru včetně operandů. Činnost obou procesorů je možno synchronizovat pomocí instrukce WAIT (viz dále).

ESC není součásti asemblerů, protože jde jenom o posloupnost pěti bitů (11011) zařazených před příkaz koprocesoru. Příkazy koprocesoru 8087 zpravidla překladače zpracovat umějí.

WAIT — *Wait until BUSY Pin Is Inactive*

POPIS:

O	D	I	T	S	Z	A	P	C

Instrukce WAIT uvede procesor do čekacího stavu. Instrukce je určena k čekání na dokončení operace koprocesorem (např. 8087), který svoji nečinnost oznamuje signálem $\overline{\text{TEST}}$ (ve vyšších typech procesorů je tento signál nazýván $\overline{\text{BUSY}}$). Činnost procesoru je pozastavena, dokud je signál $\overline{\text{TEST}}$ aktivní. Během čekání se může uplatnit povolené přerušení.

SYNTAX: ⌜ WAIT

LOCK — *Assert LOCK Signal Prefix*

POPIS:

O	D	I	T	S	Z	A	P	C

Instrukční prefix LOCK aktivuje signál $\overline{\text{LOCK}}$ po dobu zpracovávání instrukce bezprostředně následující za prefixem. Pokud je v systému více procesorů připojených na společnou sběrnici, pak tento signál zabezpečuje procesoru výlučný přístup k této sběrnici. Prefix LOCK se smí použít pouze u těchto instrukcí: ADD, OR, ADC, SBB, AND, SUB, XOR, NOT, NEG, INC, DEC, mají-li jeden z operandů uložen v paměti. Automaticky se signál $\overline{\text{LOCK}}$ generuje při provádění instrukce XCHG a INT. Použití tohoto prefixu s jinými instrukcemi způsobí u vyšších typů procesorů přerušení s chybou „nedefinovaný operační kód".

SYNTAX: ⌜ LOCK instrukce

Instrukce	*Příklad*	*Komentář*
LOCK	`LOCK AND Slovo,00FFh`	`; Nedělitelně Slovo := Slovo ∧ 00FFh`

V tomto příkladu LOCK zajistí, aby mezi okamžikem čtení cílového operandu z paměti a okamžikem zápisu cílového operandu do paměti jiný procesor s tímto operandem nepracoval.

Programování v assembleru 8086

- Syntaxe zápisu programu v assembleru 8086 je velmi podobná syntaxi assembleru pro osmibitové mikroprocesory
- Používají se kódy instrukcí, direktivy (pseudoinstrukce) a symbolická jména
- Pravidla psaní platných symbolických jmen se mohou v různých verzích assembleru lišit (většinou v detailech)
- Zápis instrukce v assembleru má tvar

  návěští instrukce cíl, zdroj komentář
- Kteroukoli z těchto částí lze vynechat

- **Návěští** je symbolické jméno, které označuje offset adresy v daném segmentu, např. adresu instrukce. Návěští se dělí na blízká (**NEAR**) a vzdálená (**FAR**). Blízké návěští představuje pouze offsetovou část adresy. Používá se v případě, že se na návěští odkazujeme v rámci segmentu, ve kterém je definováno.
- Identifikátor blízkého návěští se odděluje od instrukce dvojtečkou. Vzdálené návěští představuje kompletní 32-bitovou adresu (segment:offset).

- **Cíl a zdroj** jsou operandy instrukce. Některé instrukce mají pouze jeden nebo nemají žádný operand. Operandem může být registr, konstanta nebo paměťové místo.
- **Komentář** je libovolný text oddělený od instrukce středníkem. Překladač ignoruje vše od středníku do konce řádky.
- **Direktivy** nemají přesnou syntaxi zápisu. Způsob zápisu direktiv a jejich funkci najdete v referenční části dokumentace konkrétního použitého assembleru.
- Pro označení proměnných, návěští, konstant atd. se používají symbolická jména, kterým se říká identifikátory. Identifikátor v assembleru je libovolné slovo, složené ze znaků $, %, , ?, A...Z, a...z, O....9. Identifikátor nesmí začínat číslicí, nelze pro něj také použít rezervovaná slova. Malá a velká písmena se v assembleru standardně nerozeznávají (opět ale záleží na verzi assembleru).

Segmentování

- Paměť počítače je rozdělena na segmenty.
- Instrukce a data jsou umístěny ve stejné paměti (von Neumannova arch.)
- O tom, co je program a co data, rozhoduje programátor. Z důvodu přehlednosti je vhodné umístit program, data a zásobník do zvláštních segmentů.
- Větší programy mají umístěn kód, případně data, ve více segmentech. U malých programů je zase vhodné, aby všechny segmenty začínaly na stejné adrese a jednotlivé části programu byly umístěny v segmentu za sebou.
- Je tedy zřejmé, že jednotlivé segmenty je třeba nějakým způsobem označit a určit jejich vlastnosti.
- Pro definování segmentů v assemblerském programu slouží direktiva SEGMENT. Pomocí této direktivy lze kompletně popsat všechny vlastnosti segmentů.
- Jednodušší způsob definic segmentů je použití direktivy MODEL. Tato direktiva umožňuje automatické definování segmentů a jejich vlastností. Zjednodušená syntaxe této direktivy je následující:

  MODEL *paměťový model*

Pamětový model

- **TINY**

  Data i kód programu mají jeden společný segment, který nesmí přesáhnout velikost 64 KB. Data i kód jsou volány krátce (NEAR). Tento model se používá zejména pro .COM soubor.
- **SMALL**

  Data i kód jsou uloženy do samostatných segmentů. Obě složky jsou volány krátce (NEAR).
- **MEDIUM**

  Kódových segmentů je více, velikost kódu programu tedy může přesáhnout jeden segment, ale velikost dat nikoliv, protože datový segment je pouze jeden. Kód je volán dlouze (FAR) a data krátce (NEAR).
- **COMPACT**

  Kód má pouze jeden segment, data mohou mít segmentů více. Volání: kód NEAR a data FAR
- **LARGE**

  Data i kód mohou být ve více segmentech. Oba se volají dlouze (FAR), ale velikost celistvého pole dat nesmí přesáhnout jeden segment (ukazatel na prvky pole je pouze 16bitové číslo).
- **HUGE**

  Velikosti dat i kódu jsou stejné jako u LARGE, ale velikost celistvého pole dat může přesáhnout velikost jednoho segmentu. Jednotlivé položky stejného pole se tedy také adresují dlouze (tedy ukazatele jsou 32bitová čísla -> pomalá ukazatelová aritmetika).

Šablona pro psaní programů typu .COM

```
IDEAL ; režim IDEAL – direktivy odlišné od direktiv MASM
MODEL TINY

DATASEG ; označení datového segmentu
    ; <zde budou data programu>

CODESEG
STARTUPCODE  ; assembler doplní potřebné instrukce pro start programu
    ; <zde bude kód programu>
EXITCODE  ; assembler doplní potřebné instrukce pro ukončení programu a
            předání řízení do operačního systému

END
```

Větvení programu

- Pro větvení programu ve vyšších programovacích jazycích existují různé příkazy např. IF, CASE, SWITCH, GOTO atd. Assembler nemá pro všechny tyto příkazy přímé ekvivalenty.
- Větvení programu v assembleru se provádí pomocí instrukcí skoku. Instrukce skoku jsou dvojího druhu - nepodmíněné a podmíněné.
- Nepodmíněný skok odpovídá příkazu GOTO ve vyšších programovacích jazycích. Jeho syntaxe je

  JMP [*vzdálenost*] *cíl*

  kde *cíl* je návěští, na které chceme skočit. Pokud skáčeme v rámci jednoho segmentu, bude *vzdálenost* typu NEAR, jinak FAR. Pokud vzdálenost neuvedeme, překladač určí vzdálenost sám (ovšem ne vždy ideálně).

Instrukce podmíněného skoku

instrukce	podmínka	význam	ar.
JA / JNBE	CF = 0 & ZF = 0	skok při větším	+
JAE / JNB	CF = 0	skok při větším nebo rovném	+
JB / JC	CF = 1	skok při menším	+
JBE / JNA	CF = 1 ¦ ZF = 1	skok při menším nebo rovném	+
JE / JZ	ZF = 1	skok při rovnosti	
JG / JNLE	ZF = 0 & SF = OF	skok při větším	±
JGE / JNL	SF = OF	skok při větším nebo rovném	±
JL / JNGE	SF <> OF	skok při menším	±
JLE / JNG	ZF = 1 ¦ SF <> OF	skok při menším nebo rovném	±
JNC	CF = 0	skok při CF = 0	
JNE / JNZ	ZF = 0	skok při nerovnosti	
JNO	OF = 0	skok, pokud nebylo přetečení	
JNP / JPO	PF = 0	skok při liché paritě	
JNS	SF = 0	skok při kladném čísle	
JO	OF = 1	skok při přetečení	
JP /JPE	PF = 1	skok při sudé paritě	
JS	SF = 1	skok při záporném čísle	

- Před provedením instrukce podmíněného skoku je potřeba provést nějakou (většinou aritmetickou) instrukci, která nastaví bity v příznakovém registru.
- Často se používá instrukce CMP cíl, zdroj. Tato instrukce “odečte“ zdroj od cíle a podle výsledku nastaví příslušné příznaky. Cíl ani zdroj se přitom nezmění.
- Použití této instrukce odpovídá i popis ve třetím sloupci tabulky.
- Kromě těchto podmíněných instrukcí skoku existuje ještě instrukce JCXZ, která provede skok v případě, že registr CX je roven nule. Její využití je zejména při programování cyklů.

Program pro převod čísla do jiné číselné soustavy

- Operační systémy obsahují služby systému volané obyčejně instrukcí INT (výhoda – pokud bychom volali službu operačního systému jako podprogram, v programu by musela být konkrétní adresa podprogramu zajišťující službu. Při změně verze operačního systému adresy podprogramů budou pravděpodobně změněny a program by nefungoval. Použití INT instrukce zajistí, že při spouštění programu na jiné verzi daného operačního systému nemusí být program přeložen znovu. Adresa služby je adresa přerušovacího podprogramu a správná adresa je v přerušovací tabulce.
- Operační systém DOS obsahuje službu 09h volanou pomocí INT 21h, která vypíše na standardní výstupní zařízení řetězec. Adresa tohoto řetězce musí být v registrech DS:DX a řetězec musí být ukončen znakem $.
- Operační systém však neobsahuje žádnou službu pro vypsání čísla. Proto musíme nejdříve číslo převést na řetězec ASCII znaků a tento řetězec pak můžeme pomocí služby 09h vypsat.

- Pro převod použijeme jednoduchý algoritmus. Číslo celočíselně vydělíme základem. Zbytek po celočíselném dělení je hodnota poslední cifry. Tuto hodnotu pomocí tabulky převedeme na znak a zapíšeme do pomocného bufferu. Dělení opakujeme tak dlouho, dokud není výsledek dělení roven nule. Tím dostaneme jednotlivé cifry čísla.
- Pro dělení použijeme instrukci DIV operand. V případě, že je operand 16 bitový (například registr BX), procesor vydělí 32 bitové číslo uložené v registrech DX (vyšší část) a AX (nižší číslo). Výsledek dělení je uložen v registru AX, zbytek po celočíselném dělení je v registru DX.
- Před převodem ještě zkontrolujeme, zda je základ v povoleném rozsahu (2 až 16). V programu tedy budou tři podmíněné skoky:
  - kontrola, zda je základ menší než 2 = > chyba
  - kontrola, zda je základ větší než 16 = > chyba
  - kontrola, zda je výsledek dělení různý od nuly = > opakuj dělení

```
; soubor CISLO.ASM

IDEAL
MODEL TINY
DATASEG
Digits      DB '0123456789ABCDEF'
Text        DB 'Číslo = ' , '$'
Buffer      DB 16 DUP(?) , '$' ; v dvojkové soust. může mít číslo až 16 cifer
Message     DB 'Základ musí být v rozsahu 2 až 16!' , '$'

CODESEG
STARTUPCODE
    MOV AX,54321 ; toto číslo se vypíše
    MOV BL,16    ; základ
    CMP BL,2     ; základ musí být 2 až 16
    JB Error     ; základ je menší než 2
    CMP BL,16
    JA Error     ; základ je větší než 16
    MOV DI,16    ; do bufferu budeme zapisovat odzadu
```

```
@Loop:  DEC DI
        XOR DX,DX           ; DX <- 0
        DIV BX              ; DX AX  / BX
        MOV SI,DX           ; do SI zbytek po celočíselném dělení
        MOV DL,[Digits+SI]          ; do DL odpovídající písmeno
        MOV [buffer+DI],DL          ; a zapíšeme do bufferu
        AND AX,AX           ; je výsledek dělení nula ?
        JNZ @Loop           ; ne > další cifra
        MOV DX,OFFSET  Text         ; nejdříve vypíšeme text
        MOV AH,09h
        INT 21h
        MOV DX,OFFSET Buffer        ; a potom číslo
        ADD DX,DI           ; v DI je pozice první cifry v bufferu
        MOV AH,09h
        INT 21h
        JMP NEAR @Ret  ; skok na konec programu
```

```
Error:  MOV DX,OFFSET Message   ; vypíše zprávu
        MOV AH,09h
        INT 21h
@Ret:   EXITCODE                ; ukončení programu
END
```

Realizace příkazu CASE

Pomocí nepodmíněného skoku lze realizovat obdobu příkazu CASE, resp. SWITCH. Nepodmíněný skok má totiž ještě jednu variantu, tzv. nepřímý skok. Za instrukcí JMP se v tomto případě neuvede návěští, ale odkaz na paměťové místo, kde se nachází adresa, na kterou se má skočit. Nepřímý skok lze použít například takto:

```
DATASEG
tabulka   DW Navesti1        ; seznam NEAR návěští
          DW Navesti2        ; (FAR => musí se použít DD )
          ....
CODESEG
          ....               ; zde se naplní registr SI
          JMP [tabulka + SI] ; podle obsahu SI se skočí na příslušné návěští
          . . .

Navesti1:  ……
Navesti2:  ……
           ……
```

Podmíněný skok na větší vzdálenost než -128 až +127 bytů

- Podmíněné skoky dovolují pouze skok SHORT (parametr je znaménkové 8bitové číslo)
- Pro podmíněný skok NEAR nebo FAR použijeme instrukci podmíněného skoku s inverzní podmínkou a instrukci JMP:

```
        ……
        Jinverzni_podminka PomNav
        JMP Navesti  ; Podle zápisu NEAR nebo FAR
PomNav:
        …….
```

Realizace cyklu DO WHILE

- Tělo cyklu se vykoná alespoň jednou

```
CODESEG
        ....
Navesti:
        ; <tělo cyklu>
        Jpodminka Navesti
```

- Kontrola podmínky před cyklem (tělo cyklu se nemusí vykonat ani jednou)

```
CODESEG
        ....
Navesti:    Jinverzni_podminka PomNav
            ; <tělo cyklu>
            JMP Navesti
PomNav:  ....
```

Realizace cyklu FOR

- FOR i = 1 TO 10 (v BASICu)

```
        MOV   CX,1
Loop1:
        ; tělo cyklu
        INC  CX
        CMP  CX,10
        JNZ  Loop1
```

- Vzdálenost mezi JNZ Loop1 a Loop1 nesmí být větší než -128

Modifikace předcházejícího cyklu

```
        ….
        MOV   CX,počet_opakování
Loop1:
        ; tělo cyklu
        LOOP  Loop1
```

- Proměnná CX se dekrementuje a když je CX různé od nuly, skočí na adresu danou návěštím
- Pokud je počet_opakování roven nule, provede se tělo cyklu 65536krát . Nedostatek můžeme odstranit následující konstrukcí:

```
        ….
        MOV  CX,počet_opakování
        JCXZ    PomNav
Loop1:
        ; tělo cyklu
        LOOP   Loop1
PomNav:  …..
```

- Před vstupem do cyklu zkontrolujeme hodnotu CX. Pokud je nula, celý cyklus přeskočíme

Program pro odstranění diakritiky

```
IDEAL
MODEL TINY
BUFFER_LEN EQU 2000h

DATASEG
Cstab     DB ‘CuedaDTceELIllAAEzZooOuUyoUSLYRtaiounNUOsrrR‘  ;kód
   Kamenických
Buffer    DB BUFFER_LEN DUP(?)

CODESEG
ORG 100h
          STARTUPCODE
Load:     CLD
          MOV  DX,0FFSET buffer
          MOV  CX,BUFFER_LEN       ; délka bufferu
          MOV  BX,0                ; standardní vstupní zařízení
          MOV  AH,3Fh      ; přečti BUFFER_LEN znaků
          I NT  21 h
          OR AX,AX                 ; v AX je počet skutečně načtených znaků
          JNZ  PrintBuffer    ; alespoň jeden => vypiš
Stop:     EXITCODE
```

```
PrintBuffer:    MOV CX,AX       ; CX bude počitadlo znaků
                MOV SI,DX       ; SI bude ukazatel v bufferu
                XOR BH,BH       ; BH <- 0
LoopCh:         LODSB           ; AL <- DS:SI, SI = SI + 1
                MOV DL,AL       ; znak do DL
                CMP DL,128      ; znaky z diakritikou jsou v rozsahu
                JB NoCS         ; 128 až 171
                CMP DL,171
                JA NoCS
                SUB DL,128
                MOV BL,DL       ; BX offset v převodní tabulce
                MOV DL, [cstab+BX]      ; přečte
NoCS:           MOV AH,02h      ; zapiš znak do standardního výst. zař.
                INT 21h
                LOOP LoopCh
                JMP Load        ; další znak
END
```

- Program lze spustit FILTCS < název souboru

Podprogramy

- Důležitou součástí strukturovaných programů jsou funkce a procedury. Na úrovni assembleru je mezi nimi rozdíl pouze v tom, že funkce vrací nějakou hodnotu, nejčastěji v některém z registrů.
- Na rozdíl od vyšších jazyků může funkce vracet i více hodnot současně (ve více registrech).
- Funkce i procedury budeme jednotně nazývat procedury.
- Procedura je libovolná část kódu programu, která končí instrukcí RET. Začátek procedury bývá označen návěštím. Volání procedury se uskutečňuje pomocí instrukce CALL návěští.
- Tato instrukce uloží na zásobník návratovou adresu a provede skok na dané návěští. V závislosti na tom, zda je podprogram ve stejném segmentu jako hlavní program nebo v jiném, musí se provést volání typu NEAR nebo FAR.

- Při volání typu NEAR se na zásobník uloží pouze offsetová část návratové adresy. Při volání typu FAR se uloží jak segmentová, tak offsetová část návratové adresy (32b)
- Typu volání musí odpovídat samozřejmě i instrukce RET. Pro "vzdálené" procedury použijeme instrukci RETF a pro "blízké" RETN. Volání procedury lze schematicky znázornit takto:

```
CODESEG
        ....
        CALL NEAR Proc1
        ....
Proc1:
        ; tady budou instrukce procedury
        RETN
```

- Výhodnější způsob psaní procedur je pomocí direktivy PROC. Použitím této direktivy lze snadno předávat parametry pomocí zásobníku, vytvářet lokální proměnné, atd.
- Schematické volání procedury pomocí direktivy PROC:

```
CODESEG
        ....
        CALL Proc2
        ....
PROC Proc2    ; zde se může uvést NEAR nebo FAR
        ; <tady budou instrukce procedury>
        RET
ENDP Proc2
```

- Pokud použijeme direktivu MODEL, nemusíme specifikovat, zda je procedura blízká nebo vzdálená. Překladač vybere správný typ podle zvoleného modelu (u modelu TINY budou všechny procedury blízké). Nemusíme se také starat o instrukci RET - překladač vybere verzi podle typu procedury.

Předávání parametrů do procedur

- Jednodušší a rychlejší způsob je předání parametrů přímo v registrech procesoru. Při použití tohoto způsobu jsme omezeni počtem registrů.
- Druhý způsob, který používají i překladače vyšších jazyků, je předávání parametrů pomocí zásobníku. Na zásobník se pomocí instrukce PUSH umístí předávané hodnoty a potom se zavolá procedura normálním způsobem. Například:

```
MOV  AX,param1
PUSH  AX
MOV  AX,param2
PUSH  AX
CALL Proc3
```

- Stav zásobníku po provedení instrukce CALL Proc3 - procedura je typu NEAR (tedy na začátku procedury):

Adresa	Obsah
SP	Návratová adresa
SP+2	param2
SP+4	param1

- Pro přístup k předaným parametrům uloženým na zásobníku se používá registr BP. Protože lze předpokládat, že ke stejnému účelu používá registr BP i hlavní program, je potřeba původní obsah registru uložit a před návratem jej znovu do registru vrátit.
- Nejčastěji se registr BP ukládá opět na zásobník. S tím musíme počítat při odkazu na předávané hodnoty. Hodnota SP se ještě zmenší o dva byty.
- Po návratu z procedury je potřeba ze zásobníku vložené hodnoty opět vyjmout. To lze uskutečnit pomocí instrukcí POP. Většinou je vhodnější přičíst k hodnotě SP počet bytů, které jsme na zásobník uložili, což má podobný účinek jako instrukce POP.

```
CODESEG
        ....
        MOV   AX,param1          ;na zásobník uložím první parametr
        PUSH  AX
        MOV   AX,param2          ; a druhý parametr
        PUSH  AX
        CALL  Proc4              ; a zavolám proceduru
        ADD   SP,4               ; zruším dva parametry v zásobníku
        ....

PROC  Proc4
        PUSH  BP          ; uschování BP
        MOV  BP,SP
                ; na tomto místě BP+4 ukazuje na param2
                ;                 BP+6 ukazuje na param1
                ; lze například napsat MOV AX,[BP+4]
                ; registr AX potom bude obsahovat param2
        ; <tady bude kód procedury>
        POP  BP ; obnovení BP
        RET
ENDP  Proc4
```

- Protože tento způsob předávání parametrů je velmi častý a používají ho i vyšší programovací jazyky, umožňuje rozšířená syntaxe příkazu CALL a direktivy PROC automatické předávání parametrů pomocí zásobníku. Volání procedury pak bude vypadat například takto:

```
CODESEG
        ....
        CALL Proc4   C,param1,param2  ; zavolám proceduru s
                                      ; parametry
        ....

PROC C Proc4  arg1,arg2
                ; na tomto místě arg1 ukazuje na param1
                ;              arg2 ukazuje na param2
                ; Lze například napsat MOV AX, [arg1]
                ; registr AX potom bude obsahovat param1
        ; <tady bude kód procedury>
        RET
ENDP  Proc4
```

- Písmeno “C“ v instrukci CALL a direktivě PROC označuje jazyk C, jehož konvenci volání chceme použít. Můžeme zde uvést I jiný jazyk, např. PASCAL Každý jazyk používá trošku jiný způsob předávání parametrů (např. pořadí, ve kterém jsou parametry ukládány na zásobník), princip však zůstává stejný. Překladači musíme proto oznámit, kterou konvenci má použít.
- param1 a param2 (případně i další parametry) mohou být konstanty, registry, nebo paměťová místa.
- při použití této rozšířené syntaxe se vůbec nemusíme starat o zásobník ani o registr BP. Vše zařídí překladač.
- Syntaxe direktivy PROC umožňuje i další věci, například vytváření lokálních proměnných na zásobníku.