3

3. ZÁKLADNÍ INSTRUKCE JAZYKA PLC836

Jazyk PLC836 je určen pro efektivní programování interfejsu pro systém CNC836. Oproti programování v assembleru přináší tyto výhody: zkrátí dobu kódování programu, sníží pravděpodobnost chyby při kódování, zajistí co nejoptimálnější využití času mikroprocesoru a napomáhá k zajištění doporučené struktury programu. Jazyk používá výhradně symbolických adres a to i při práci s jednotlivými bity paměti. Zdrojový tvar programu má podobný tvar jako jazyk programovatelných automatů, používaný u automatů řady NS900 TESLY Kolín.

3.1 Zápis zdrojového programu

Zdrojové programy interfejsu lze napsat v libovolném textovém ASCII editoru. Tento návod není učebnicí programování programů interfejsu, proto jsou v této kapitole uvedeny pouze nezbytné informace pro zápis zdrojového programu. Programování je prováděno v mnemonickém kódu. Pro zápis programu platí podobná pravidla jako pro zápis zdrojového programu assembleru.

Programový řádek obsahuje:

a) Návěští
Je nepovinné a určuje symbolicky adresu místa v paměti pro proměnnou nebo na kterou má být např. proveden skok v programu. Návěští může obsahovat max. 31 znaků, t.j. písmen, číslic a tří speciálních znaků:

_ podtrhovátko
? otazník
@ "zavináč"

Prvním znakem nesmí být číslice. Návěští musí být ukončeno dvojtečkou. Návěští může být uvedeno na samostatném řádku, přičemž se vztahuje k nejbližšímu následně uvedenému řádku s operačním kódem, konstantou nebo alokací paměti.

b) Operační kód
Je mnemonický zápis kódu příslušné instrukce. Jako operační kód jsou povoleny instrukce, uvedené v další části návodu. Kromě toho je umožněno používat i instrukce assembleru 8086 i když pro zápis programu interfejsu to není nutné.

c) Operand
Jednotlivé instrukce mohou být bez operandu (např. instrukce BCD) nebo většinou s jedním operandem (např. LDR ALFA), případně některé instrukce mohou mít dva operandy (např. INP port,adr) oddělené čárkami.

d) Komentář
Text, uvedený za středníkem se považuje za komentář a ignoruje se. Komentář může být uveden na samostatném řádku nebo na řádku společně s kódem.

Příklad:

```
                              ;toto je komentar
PAM12:      LDR   ALFA        ;toto je komentar
```

3.2 Pracovní registry jazyka PLC836

Instrukce jazyka PLC836 využívají tyto registry:

a) Jednobitový takzvaný **registr logických operací**, který se bude v dalším textu označovat zkratkou **RLO**. Fyzicky se jedná o bit s váhou 40h - registru AH mikroprocesoru.

b) Standartně šestnácti-bitový takzvaný **datový registr** , který se bude v dalším textu označovat zkratkou **DR**. Některé instrukce mohou pracovat s rozšířeným 32 bitovým DR registrem.
Fyzicky se jedná o registr CX mikroprocesoru. V případě rozšířeného 32 bitového DR registru jsou to registry CX a BP.

c) Zásobník - jedná se o 8 buňek typu WORD.

Pokud se používají pouze instrukce jazyka PLC836, není fyzická reprezentace registrů pro programátora důležitá. Registry se musí respektovat pouze při eventuelním programování částí programu v assembleru 8086.

Změna hodnoty v registru RLO neovlivní datový registr DR a obráceně, změna v datovém registru DR neovlivní registr RLO. Jazyk PLC836 automaticky rozpoznává, kdy se má použít byteový nebo wordový přístup jak k registru DR, tak k paměti.

3.3 Deklarace paměti

Jazyk PLC836 možňuje pracovat s libovolnými adresami v symbolické formě. Jako symbolické adresy lze definovat adresy v programu i adresy bitových, osmibitových (typu byte) a šestnáctibitových (typu word) buněk paměti RAM. Symbolická adresa je definována jako slovo o maximálně 31 znacích (délka není omezená, ale významných je prvních 31 znaků), které může obsahovat jak písmena, tak čísla, přičemž jako první musí být vždy uvedeno písmeno. Toto slovo však nesmí být uvedeno v seznamu klíčových slov, která mají již předem daný význam. Seznam klíčových slov je uveden v příloze.
Přiřazení symbolické adresy určité fyzické adrese lze provést dvěma způsoby. Jedná-li se o deklaraci adresy (místa) v programu (EPROM) nebo deklaraci osmibitového či šestnáctibitového slova v paměti RAM, postačí napsat za příslušný symbol dvojtečku a adresa je pak dána polohou tohoto symbolu nebo-li adresa se vztahuje na instrukci následující za dvojtečkou.

Příklad:
V programu, který je vyjádřen instrukcemi 1 až 5, chceme definovat adresu instrukce 2 a 3 symboly ALFA a BETA.

```
            INSTRUKCE         1
ALFA:       INSTRUKCE         2
BETA:       INSTRUKCE         3
            INSTRUKCE         4
            INSTRUKCE         5
```

3.4 Definice bitu v paměti, byte, word a konstant

instrukce	**DFM**

funkce **definice bitu v paměti**

syntax **DFM [bit0],[bit1],,,,,,[bit7]**

Chceme-li symbolicky definovat jednotlivé bity paměti RAM, použijeme speciální instrukce DFM. Jedna instrukce DFM definuje vždy v místě svého zápisu jednu slabiku paměti (8 bitů). Celou slabiku můžeme symbolicky pojmenovat pomocí symbolu, zapsaného před tuto instrukci a zakončeného dvojtečkou. Takto definovaná slabika umožňuje kromě bitového i bajtový přístup. Symboly, které jsou uvedeny za příkazem DFM, deklarují pak jednotlivé bity této slabiky paměti. První symbol náleží bitu d0, druhý bitu d1 atd., až osmý symbol náleží bitu d7. Symboly musí být odděleny čárkami. V případě, že některý bit definovat nechceme, můžeme příslušný symbol vynechat. Čárky však uvedeny být musí!

Jednotlivé vstupy a výstupy jednotek periferních obvodů jsou snímány a plněny po osmicích. Abychom si tvorbu programu PLC usnadnili, budeme pracovat s těmito vstupy a výstupy výhradně tímto způsobem. Na začátku programu přečteme pomocí speciální instrukce všechny vstupy do paměti RAM a na konci programu PLC naopak vymezené paměti RAM zapíšeme do výstupů. Jednotlivé vstupy či výstupy pak můžeme deklarovat a obsluhovat stejně jako bity paměti RAM.

instrukce	**DS**	**1**
	DS	**2**
	DS	**n**

```
funkce      DS    1     vymezení paměti o délce 1 byte
            DS    2     vymezení paměti o délce 1 word
            DS    n     vymezení paměti o délce n byte

syntax      DS    1
            DS    2
            DS    n
```

Instrukce DS se používá pro definování proměnných a inicializaci paměti. Operandem se deklaruje délka vymezené paměti typu BYTE. Instrukce DS 1 s návěštím proměnné deklaruje tuto proměnnou jako BYTE a instrukce DS 2 deklaruje proměnnou jako WORD.

instrukce	EQUI

funkce **definice konstanty**

syntax **EQUI konst,hodn**

Instrukce EQUI definuje konstanty použité v programu. První parametr je symbolický název konstanty a druhý parametr je jeho hodnota. Hodnota konstanty může být zapsána :

1. dekadicky např. 123, 54213
2. hexadecimálně např. 0F8H, 15H
3. znakově např. 'A', '8'

Příklad:
V paměti RAM chceme definovat slabiku GAMA a její jednotlivé bity d0 až d7 jako symboly GAMA0 až GAMA7. Na adrese DELTA chceme symbolicky definovat bit d0 = DELTAM, bit d2 = DELTAG a bit d7 = DELTAT.

```
GAMA:       DFM         G0,G1,G2,G3,G4,G5,G6,G7
DELTA:      DFM         DELTAM,,DELTAG,,,,,DELTAT
```

Příklad:
Pro proměnnou ALFA vymezte délku 1 byte.
Pro proměnnou BETA vymezte délku 2 byte.
Pro proměnnou GAMA vymezte pole o délce 30 byte.
Definujte konstantu DELTA s hodnotou 1000.

```
ALFA:       DS          1
BETA:       DS          2
GAMA:       DS          30
            EQUI        DELTA, 1000
```

3.5 Logické operace s bity paměti a RLO

instrukce	LDR

funkce **čtení bitu paměti do RLO /LOAD RLO/**

syntax
```
LDR         [-]bit
LDR         [-][adr.]bit      složitější způsoby adresace
LDR         [-][$+xx.]bit
```

Instrukce **LDR** provádí plnění registru logických operací RLO bitem zadané paměti.

Instrukce LDR má povinný operand. Tímto operandem je symbolický název bitu paměti RAM, definovaný pomocí instrukce DFM. Je-li tento symbolický název bitu paměti uveden bez znaménka nebo se znaménkem plus, je příslušný bit čten přímo. Je-li před symbolickým názvem bitu znaménko -, čte se příslušný bit paměti negovaně.

Složitější způsoby adresace bitu: (platí od verze překladače 5.039)
V případě, že je potřeba načíst bit z jiného místa paměti, než je daný bit definovaný, uvede se adresa paměti před názvem bitu a oddělí se tečkou. Místo adresy místa je možno použít i název bitu, který je tam definovaný, způsoby indexace pomocí [BX] a prvky struktury. Tímto je například umožněna práce s rozsáhlejším bitovým polem. Když potřebujeme načíst bit z místa o několik bajtů dál něž je bit definován (například o 12), můžeme místo vlastního názvu použít znak $+ ($ + 12). Složitější způsoby adresace bitu je možno použít u instrukcí LDR, LA, LO, LX a pro instrukce WR, FL1 a FL, když je nastavena VERINSTRU alespoň na V1.
Složitější způsoby adresace bitu se používají i v případě, když je potřeba načíst bit s příslušnou váhou s libovolného místa paměti. V PLC programu se formálně nadeklarují univerzální názvy bitů, například B0,B1,...,B7, které se pak používají pro přístup k libovolnému místu paměti.

Příklady adresace bitu:

```
DFM   B0,B1,B2,B3,B4,B5,B6,B7 ;formální definice bitů

LDR   ALFA              ;načtení bitu ALFA z paměti, kde je definován
LDR   BUN5.ALFA         ;z buňky BUN5 se načte bit ze stejné pozice,
                        ;jak je původne definován bit ALFA.
LDR   BUN5.B4                 ;načtení 4.bitu (váha 10h) z buňky BUN5
LDR   -(BUN5+3).ALFA          ;načte se negace bitu z buňky BUN5+3
LDR   BUN5[BX].ALFA           ;použití indexace (viz kapitola.18)
LDR   -(BUN5[BX]-6).ALFA      ;další kombinace
LDR   (BUN5[BX]+PRVNI+2).ALFA ;PRVNI je prvek struktury
LDR   $+3.ALFA          ;načtení bitu ALFA z pozice o 3 bajty dál
```

3.6 Princip zásobníku a koncových instrukcí

Jsou-li dvě nebo více instrukcí LDR zapsány v programu za sebou, aniž byl mezi nimi registr log. operací nějak využit (pro zápis do paměti či podmíněný skok), ukládá se před další instrukcí LDR předchozí stav RLO do zásobníku. S takto uloženými hodnotami lze potom pracovat pomocí instrukcí LA, LO nebo LX bez udání operandu. Pomocí zásobníku je takto umožněno programovat závorkové operace.

Vyrovnanost zásobníku se kontroluje na konci každého modulu (viz dále) a na konci každého sekvenčného logického celku. Nevyrovnanost zásobníku ohlásí chybu v úvodní fázi kompilace překladače TECHNOL. Kontrolu vyrovnanosti zásobníku ve fázi odlaďování programu možno vnutit pomocnou instrukcí **CHECK** (viz popis pomocných instrukcí).

Každá logická rovnice naprogramovaná pomocí instrukcí **LDR, LA, LO** a **LX** musí být ukončena tzv. **koncovou instrukcí**, která ukončuje logickou rovnici. Koncová instrukce nuluje vnitřní příznak vnořování do zásobníku, takže u první následující instrukce **LDR** nedojde k uložení RLO do zásobníku.

Koncové instrukce v jazyku PLC836 jsou:

- WR
- JL0, JL1
- STO1
- FL1
- CONRD
- EX, EX0, EX1, BEX
- TEX0, TEX1
- TIM

instrukce	LA LO LX

```
funkce      LA          logický součin s RLO /AND/
            LO          logický součin s RLO /OR/
            LX          nonekvivalence s RLO /XOR/

syntax      LA
            LO
            LX

syntax2     LA          [-bit]
            LO          [-bit]
            LX          [-bit]
            LA,LO,LX    [-][adr.]bit      složitější způsoby adresace
            LA,LO,LX    [-][$+xx.]bit
```

Instrukce **LA**, **LO** a **LX** mají operand volitelný, t.j. může být zadán, ale nemusí. Tímto operandem je symbolický název bitu paměti RAM, definovaný pomocí instrukce DFM. Je-li tento symbolický název bitu paměti uveden bez znaménka nebo se znaménkem plus, je příslušný bit čten přímo. Je-li před symbolickým názvem bitu znaménko -, čte se příslušný bit paměti negovaně.

Jsou-li dvě nebo více instrukcí LDR zapsány v programu za sebou, aniž byl mezi nimi registr log. operací nějak využit (pro zápis do paměti či podmíněný skok), ukládá se před další instrukcí LDR předchozí stav RLO do zásobníku. S takto uloženými hodnotami lze potom pracovat pomocí instrukcí LA, LO nebo LX bez udání operandu. Instrukce LA provede logický součin naposledy uložené hodnoty zásobníku s RLO a výsledek uloží do RLO. Instrukce LO provede obdobně logický součet a instrukce LX nonekvivalenci. Vyrovnanost zásobníku se kontroluje na konci každého modulu (viz dále) a na konci každého sekvenčně logického celku. Nevyrovnanost zásobníku ohlásí chybu v úvodní fázi kompilace překladače TECHNOL.

Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce	CA

```
funkce      negace RLO

syntax      CA
```

Instrukce **CA** je bez operandu a provádí negaci obsahu registru logických operací RLO. Má-li před touto instrukcí registr log. operací hodnotu 1, po této instrukci bude mít hodnotu 0 a naopak.

Příklad:
Mějme symbolicky definovat bity paměti A1, A2. Naplňte registr logických operací v závislosti na stavu těchto pamětí dle logického výrazu:

```
RLO = A1 + A2

řešení a):  LDR         A1
            LDR         A2
            LO
```

Je-li za instrukcí LA, LO nebo LX uveden operand (symbolicky označený bit paměti), má instrukce následující

význam:

Instrukce LA s operandem provede logický součin zadaného bitu paměti s registrem logických operací a výsledek opět uloží do RLO. Instrukce LO provede obdobně logický součet instrukce LX nonekvivalenci. Podíváme-li se zpět na příklad 3, zjistíme,že jeho další možné řešení je následující:

```
řešení b):      LDR         A1
                LO          A2
```

Řešení 3b je co do funkce naprosto rovnocenné řešení 3a, délka programu je však v tomto případě kratší.

Pomocí výše uvedených instrukcí (LDR, LA, LO, LX, CA) lze tedy naprogramovat libovolnou logickou podmínku (logický výraz). Postup programování názorně ukazují následující příklady 4, 5, 6.

Příklad:
Naprogramujte logický výraz (pozn.: <> je NONEKVIVALENCE):

```
RLO = (ALFA <> BETA) + (GAMA <> DELTA)

      LDR         ALFA
      LX          BETA
      LDR         GAMA        ;ULOŽÍ RLO DO ZÁSOBNÍKU A NAČTE BIT GAMA
      LX          DELTA
      LO                      ;LOG.SOUČET RLO SE ZÁSOBNÍKEM
```

Příklad:
Naprogramujte logický výraz:

```
         ----     ----     ----                ----
RLO = [(A1.A2.A3) + (B1.B2)] . (C2 + C3)

      LDR         -A1
      LA          A2
      LA          -A3
      LDR         -B1   ;ULOŽÍ RLO DO ZÁSOBNÍKU A NAČTE NEGACI BITU B1
      LA          B2
      LO                ;LOG.SOUČET RLO SE ZÁSOBNÍKEM
      LDR         C2    ;ULOŽÍ MEZIVÝSLEDEK DO ZÁSOBNÍKU A NAČTE BIT C2
      LO          -C3
      LA                ;LOG.SOUČIN RLO SE ZÁSOBNÍKEM
```

Příklad:
Naprogramujte logický výraz:

```
         _______      ______
RLO = A1.A2.A3 + B1.B2.B3

      LDR         A1
      LA          A2
      LA          A3
      CA
      LDR         B1
      LA          B2
      LA          B3
      CA
      LO
```

3.7 Zápis bitů do paměti

instrukce WR

```
funkce          zápis obsahu RLO do paměti

syntax1   WR        bit

syntax2   WR        <bit1 AND bit2 [AND bit3 ...]>  starší způsob

syntax3   WR        bit1 [,bit2 ...]  pro VERINSTRU WR_V1
          WR        [adr.]bit   složitější způsoby adresace
          WR        [adr.]bit [,[adr2.]bit2 ...]
          WR        [$+xx.]bit[,[$+yy.]bit2 ...]
```

Instrukce WR provádí zápis obsahu RLO do vyjmenovaných bitů jedné slabiky paměti. Obsah RLO a DR se po vykonání této instrukce nemění. Instrukce WR je ***koncová instrukce*** pro logické rovnice.

Původní syntaxe instrukce **WR** může mít 1 nebo více (maximálně 8) operandů (ponechána pro kompatibilitu). Tyto operandy jsou symbolické názvy bitů paměti RAM, definované pomocí instrukce DFM. Jsou-li uvedeny dva nebo více operandů, musí se jednat o bity jedné slabiky paměti.
Jednotlivé názvy bitů jsou spojovány slůvkem AND a jejich seznam je uzavřen ve špičatých závorkách " < >". V případě jednoho symbolu není nutné závorky uvádět.

Instrukce **WR** existuje i v modifikované novější verzi. Modifikace se provádí instrukcí **VERINSTRU**. (viz popis instrukce VERINSTRU v podkapitole 3.17 Pomocné příkazy). Modifikace umožní v parametrech instrukce použít více bitových operandů, které na rozdíl od původních provedení instrukcí nemusí být umístěny v jednom bajtu. Operandy se neuzavírají do špičatých závorek, ale oddělí se čárkou.

Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce FL

```
funkce          nastavování bitův paměti

syntax1   FL    0,bit
          FL    1,bit

syntax2   FL    0,<bit1 AND bit2 [AND bit3 ...]> starší způsob
          FL    1,<bit1 AND bit2 [AND bit3 ...]>

syntax3   FL    0,bit1 [,bit2 ...]      pro VERINSTRU FL_V1
          FL    1,bit1 [,bit2 ...]      pro VERINSTRU FL_V1
          FL    0,[adr.]bit             složitější způsoby adresace
          FL    1,[adr.]bit [,[adr2.]bit2 ...]
          FL    0,[$+xx.]bit[,[$+yy.]bit2 ...]
```

Instrukce FL zajistí, nezávisle na obsahu RLO, plnění bitů jedné slabiky paměti nulou nebo jedničkou. Za příkazem FL, jako první operand, následuje hodnota 0 nebo 1. Jako druhý operand je nutné uvést seznam bitů, do kterých se bude uvedená hodnota plnit. Tyto operandy jsou povinné. Volitelně lze navíc zadat, formou třetího a čtvrtého operandu, konstantu opačnou a bity, do kterých se má plnit. Obsah RLO a DR se po vykonání této instrukce nemění.

Instrukce **FL** existuje i v modifikované novější verzi. Modifikace se provádí instrukcí **VERINSTRU** (viz popis instrukce VERINSTRU v podkapitole 3.17 Pomocné příkazy). Modifikace umožní v parametrech instrukce

použít více bitových operandů, které na rozdíl od původních provedení instrukcí nemusí být umístěny v jednom bajtu. Operandy se neuzavírají do špičatých závorek, ale oddělí se čárkou.
Složitější způsoby adresace jsou popsány u instrukce LDR.

instrukce	**FL1**

funkce	**podmíněné nastavování bitův paměti**		
syntax1	**FL1**	**0,bit**	
	FL1	**1,bit**	
syntax2	**FL1**	**0,<bit1 AND bit2 [AND bit3 ...]>**	
	FL1	**1,<bit1 AND bit2 [AND bit3 ...]>**	
syntax3	**FL1**	**0,bit1 [,bit2 ...]**	**pro VERINSTRU FL1_V1**
	FL1	**1,bit1 [,bit2 ...]**	**pro VERINSTRU FL1_V1**
	FL1	**0,[adr.]bit**	**složitější způsoby adresace**
	FL1	**1,[adr.]bit [,[adr2.]bit2 ...]**	
	FL1	**0,[$+xx.]bit[,[$+yy.]bit2 ...]**	

Instrukce **FL1** zajistí plnění bitů jedné slabiky paměti nulou nebo jedničkou jenom tehdy, když je v registru RLO hodnota **1**. Za příkazem FL1, jako první operand, následuje hodnota 0 nebo 1. Jako druhý operand je nutné uvést seznam bitů, do kterých se bude uvedená hodnota plnit. Tyto operandy jsou povinné. Obsah RLO a DR se po vykonání této instrukce nemění. Instrukce FL1 je ***koncová instrukce*** pro logické rovnice.

Instrukce **FL1** existuje i v modifikované novější verzi. Modifikace se provádí instrukcí **VERINSTRU**(viz popis instrukce VERINSTRU v podkapitole 3.17 Pomocné příkazy). Od verze překladače TECHNOL 2.3 je možno použít i obdobnou instrukci **FL0**. Modifikace umožní v parametrech instrukce použít více bitových operandů, které na rozdíl od původních provedení instrukcí nemusí být umístěny v jednom bajtu. Operandy se neuzavírají do špičatých závorek, ale oddělí se čárkou.

Složitější způsoby adresace jsou popsány u instrukce LDR.

Instrukce **FL1** se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
      JL0   OBSKOK
      FL    1,BIT       ekvivalent: FL1   1,BIT
OBSKOK:
```

Příklad:
Symbolicky označené bity paměti PETR, IVAN, JANA naplňte v závislosti na pamětech PAVEL a EVA takto:

```
PETR = IVAN = PAVEL + EVA
JANA = PAVEL . EVA

JMENA1:     DFM         PAVEL,EVA, , , , , ,
JMENA2:     DFM         PETR,IVANA,JANA, , , , , ,
            ...
            LDR         PAVEL
            LO          EVA
            WR          <PETR AND IVAN>
            LDR         PAVEL
            LA          EVA
            WR          JANA
```

Příklad:
Mějme definované bity paměti těmito symboly:

A1, A2
B1, B2, B3, B4 ve slabice B

Naplňte tyto paměti následovně:

0 do A1, B3, B4
1 do B1, B2, A2

```
FL          0,A1
FL          1,A2
FL          1,<B1 AND B2>,0,<B3 AND B4>
```

3.8 Větvení programu

instrukce	**JUM** **JL0** **JL1**

funkce	**JUM**	**nepodmíněný skok**
	JL0	**skok, je-li RLO = 0**
	JL1	**skok, je-li RLO = 1**
syntax	**JUM**	**adr**
	JL0	**adr**
	JL1	**adr**

Všechny tyto instrukce, které umožňují větvení programu, nebo-li programový skok, musí mít jako operand uvedenu adresu instrukce, která má být vykonávána dál v případě splnění příslušné podmínky skoku. Není-li tato podmínka splněna, pokračuje procesor ve zpracování instrukce následující (skok se neprovede). Tato adresa se zadává symbolickou formou, přičemž příslušný symbol lze definovat pomocí dvojtečky v kterémkoli místě programu.

Instrukce **JUM** nevyžaduje splnění žádné podmínky, skok se tedy provede vždy. Instrukce **JL0** zajistí skok na zadanou adresu pouze v případě, že RLO = 0. Instrukce **JL1** zajistí skok na zadanou adresu pouze v případě, že RLO = 1. Instrukce JL0 a JL1 jsou ***koncové instrukce*** pro logické rovnice.

Příklad:
Zapište pomocí instrukcí jazyka PLC836 následující logický algoritmus:

```
Je-li A1 = A2, naplň 0 do B1 a B2 definovaných v jedné slabice paměti.
Je-li A1 . A2 = 1, naplň 1 do B3.

            LDR         A1
            LX          A2
            JL1         NAV1
            FL          0,<B1 AND B2>

 NAV1:      LDR         A1
            LA          A2
            JL0         NAV2
            FL          1,B3
NAV2:       ...
            ...
            ...
```

3.9 Způsoby předefinování typu u datových proměnných

Jazyk PLC836 v datových operacích přistupuje k operandům automaticky podle toho, jaká byla jejich definice. Například instrukce LOD načte hodnotu typu BYTE, WORD nebo konstantu podle způsobu definice operandu:

DEFINICE:				**OPERACE:**		
BUNKA1:	DS	1	.....	LOD	BUNKA1	načte BYTE do DR registru horní část DR registru je nulová
BUNKA2:	DS	2	.....	LOD	BUNKA2	načte WORD do DR registru
EQUI	KONST,124		.....	LOD	KONST	načte konstantu 124 do DR registru

Některé instrukce, které pracují s DR registrem, mají možnost při svém vykonávání předefinovat typ datové proměnné. Následující popis se bude týkat instrukcí LOD, STO, STO1, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, TM, TIM, TEX0, TEX1, MULB a DIVB.

Instrukce můžou mít nepovinný prefix před názvem proměnné (TYPE.), který může předefinovat nebo dodefinovat typ proměnné na :

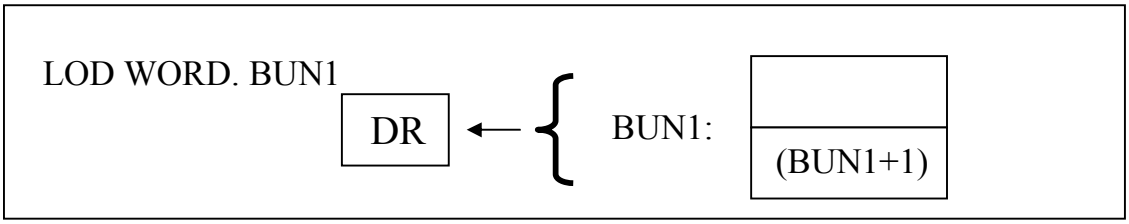
- **CNST**
- **BYTE**
- **WORD**
- **HIGH**
- **DWRD**

Prefix se připojí k operandu instrukce pomocí tečky.

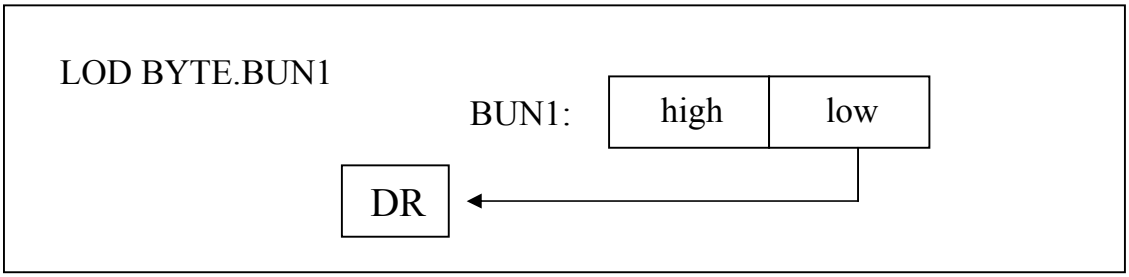
Instrukce LOD, AD, SU, EQ, EQ1, LT, GT, LE, GE, RR, RL, MULB, DIVB mohou mít jako operand konstantu, která není definovaná pomocí instrukce **EQUI**. V tomto případě se uvede před hodnotou konstanty typ: **"CNST."**. Předpokládá se, že konstanta je maximálně 16 bitová. Jedinou výjimkou je použití prefixu CNST. u instrukcí LOD, kdy z praktických důvodů dojde k vynulování horních 16 bitů rozšířeného 32 bitového DR registru.

Dále uvedeme příklady použití prefixů u instrukcí LOD. U ostatních instrukcí působí prefixy obdobně, ale nemusí se jednat o čtení ale například o zápis DR registru do paměti.

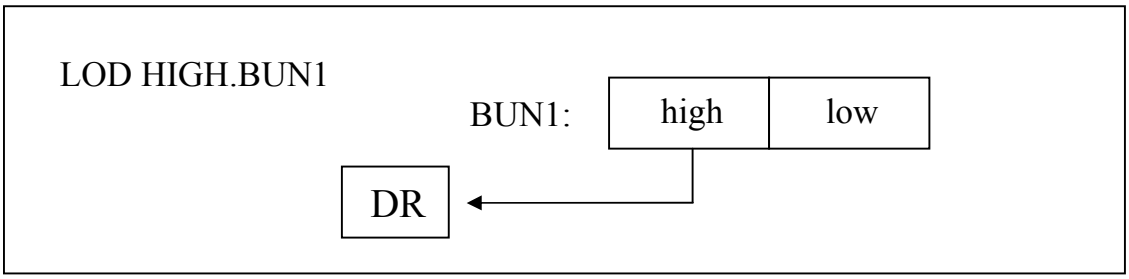
Prefix **"WORD."** způsobí načtení 2 byte za sebou do DR registru bez ohledu na to, jak jsou tyto definovány:



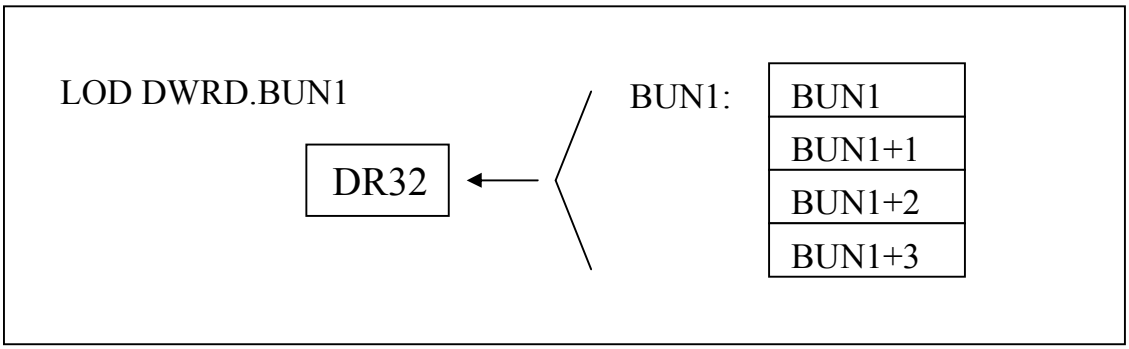
Prefix **"BYTE."** způsobí načtení 1 byte do DR registru bez ohledu na to, jak je definován. V případě, že je buňka definována typu WORD, načte se do DR registru spodní byte:



Prefix **"HIGH."** způsobí načtení 1 byte do DR registru bez ohledu na to, jak je definován z následujícího paměťového místa: V případě, že je buňka definována typu WORD, načte se do DR registru vrchní byte:



Prefix **"DWRD."** způsobí načtení 4 byte za sebou do rozšířeného 32 bitového DR registru bez ohledu na to, jak jsou tyto definovány. U instrukcí, které mohou pracovat se čtyřbyteovými (double-word) operandy je prefix DWRD povinný:



Příklad:
Zapište do buňky odměřování systému v ose X:

```
BUNKA:      DS    4

            CLI                     ;zákaz přerušení
            LOD   DWRD.B_POL        ;načte do rozšířeného DR registru B_POL
            AD    DWRD.B_INK        ;připočte dvě slova B_INK
            STI                     ;povolení přerušení
            STO   DWRD.BUNKA        ;zapíše do BUNKA 32 bitů
```

Příklad:
Použití prefixů:

```
            LOD   CNST.123          ;přímé naplnění konstantou
            AD    CNST.50h          ;připočte 50 hexadecimálně
            AD    BYTE.ALFA         ;připočte spodní byte ALFA
            STO   WORD.BETA         ;zapíše do BETA jako WORD
```

Možnosti deklarace a předefinování typu u datových proměnných:

	DEKLARACE			***MOŽNOST PŘEDEFINOVÁNÍ TYPU***				
	BYTE	**WORD**	**CNST**	**BYTE.**	**WORD.**	**HIGH.**	**CNST.**	**DWRD.**
LOD	✓	✓	✓	✓	✓	✓	✓	✓
STO, STO1	✓	✓	.	✓	✓	✓	.	✓
TM	✓	✓	.	✓	✓	✓	.	.
CU, CD	✓	✓	.	.	.	.	.	.
CUBCD	✓	✓	✓	.	.	.	.	.
AD, SU	✓	✓	✓	✓	✓	✓	✓	✓
MULB	✓	.	✓	✓	✓	✓	✓	✓
DIVB	✓	.	✓	✓	✓	✓	✓	✓
RR, RL	✓	.	✓	✓	.	✓	✓	.
EQ, EQ1	✓	✓	✓	✓	✓	✓	✓	✓
LT, GT	✓	✓	✓	✓	✓	✓	✓	✓
LE, GE	✓	✓	✓	✓	✓	✓	✓	✓
TIM	✓	✓	.	✓	✓	✓	.	.
TEX0, TEX1	✓	✓	.	✓	✓	✓	.	.

3.10 Zápis a čtení do paměti z datového registru DR

instrukce	LOD

```
funkce      LOD         čtení byte, word, konstanty (nebo dword) do DR

syntax      LOD         [-]adr
            LOD         [-][TYPE.]adr
            LOD         [-]TYPE.(adr+n)
            LOD         [-]CNTS.konst
                              TYPE = BYTE. WORD. HIGH. DWRD.
```

Instrukce **LOD** zajistí načtení obsahu paměti o délce byte nebo word (dword) nebo konstanty do datového registru DR. Adresa paměti se zadává formou operandu instrukce v symbolickém tvaru. Konstanta musí být definována pomocí instrukce EQUI, nebo pomocí prefixu "**CNST.**". Použití instrukce s prefixem "CNST." způsobí naplnění i rozšířené části DR registru.

Zadání znaménka - způsobí, že hodnota bude vzata do DR registru záporně. Ve skutečnosti se jedná o dvojkový doplněk z načtené hodnoty.

Předefinování typu je posáno v kapitole "Způsoby předefinování typu u datových proměnných".

instrukce	STO

```
funkce      STO         zápis DR do paměti byte, word (nebo dword)

syntax      STO         adr
            STO         [TYPE.]adr
            STO         TYPE.(adr+n)
                              TYPE = BYTE. WORD. HIGH. DWRD.
```

Instrukce **STO** umožňuje zapsat obsah datového registru DR do zadané paměti o délce byte, word (nebo dword s požitým prefixu). Adresa paměti se opět zadává formou operandu instrukce v symbolickém tvaru.

Předefinování typu je posáno v kapitole "Způsoby předefinování typu u datových proměnných".

instrukce	STO1

```
funkce      STO1        podmíněný zápis DR do paměti byte, word (nebo dword)

syntax      STO1        adr
            STO1        [TYPE.]adr
            STO1        TYPE.(adr+n)
                              TYPE = BYTE. WORD. HIGH. DWRD.
```

Instrukce **STO1** je podobná instrukci **STO**, ale zápis DR do paměti se provede jenom tehdy, když je v registru RLO hodnota 1. Instrukce STO1 je zavedena pro kompatibilitu s automatem NS915. Důležitý rozdíl vzhledem k instrukci STO je, že instrukce **STO1** je ***koncová instrukce*** vzhledem k logickým operacím.

Od verze překladače TECHNOL 2.3 je možno použít i obdobnou instrukci **STO0**, která provede podmíněný zápis DR do paměti tehdy, když je v registru RLO hodnota 0.

Instrukce **STO1** se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
            JL0   OBSKOK
            STO   BUNKA      ekvivalent:  STO1  BUNKA
OBSKOK:
```

Příklad:
Mějme symbolicky definované slabiky paměti RAM těmito symboly:

ALFA, BETA, GAMA

Obsah slabiky ALFA zapište do slabiky BETA, konstantu 123H do GAMA.

```
            EQUI        K123, 123H

            LOD         ALFA
            STO         BETA
            LOD         K123
            STO         GAMA
```

Způsob zápisu je stejný i pokud se jedná o délky typu word.

Příklad:
Mějme symbolicky definované slabiky paměti RAM těmito symboly: ALFA typu WORD a BETA typu BYTE. Záporný obsah slabiky BETA zapište do spodního byte proměnné ALFA.

```
            LOD         -BETA
            STO         BYTE.ALFA
```

Příklad:
Přepište hodnotu z buňky NASTAVENI do buňky BUNKA, když výraz ALFA*BETA je roven jedné:

```
            LDR         ALFA
            LA          BETA
            LOD         NASTAVENI
            STO1        BUNKA
```

3.11 Realizace časově závislých funkcí

instrukce	DFTM01 DFTM1 DFTM10 DFTM100

```
funkce      DFTM01      úsek programu aktivován po 0,1 s
            DFTM1       úsek programu aktivován po 1 s
            DFTM10      úsek programu aktivován po 10 s
            DFTM100     úsek programu aktivován po 100 s

syntax      DFTM01      konec
            DFTM1       konec
            DFTM10      konec
            DFTM100     konec
```

Aby bylo dosaženo co nejoptimálnějšího využití strojového času procesoru a pro definici nastavované doby časovačů **TM** a **TIM,** jsou časově závislé funkce realizovány ve zvláštních blocích. Takovým programovým blokem je úsek programu, který je na začátku ohraničen instrukcí DFTM01, DFTM1, DFTM10 či DFTM100 a na konci návěštím "konec".

Zvláštností těchto programových bloků je to, že jsou aktivovány v delších časových intervalech než ostatní části řídicího programu PLC, které jsou aktivovány po 20ms. Programový blok, definovaný instrukcí DFTM01, je aktivován ("procházen procesorem") po intervalech 0,1 sec. Programový blok, definovaný instrukcí DFTM1, je aktivován po intervalech 1 sec. Programový blok, definovaný instrukcí DFTM10, je aktivován po intervalech 10 sec a programový blok definovaný instrukcí DFTM100 je aktivován po 100 sec.

V standardní verzi systému může být použita instrukce DFTM01, DFTM1, DFTM10 a DFTM100 pouze jednou. V systémech řady CNC8x9 – DUAL mohou být použity instrukce DFTM ve všech modulech, a to i vícekrát.

Po zapnutí systému se provede automaticky nulování pracovních čítačů, které slouží k vytváření intervalů 0.1, 1, 10 a 100 sec.

instrukce	TM

```
funkce      časovač závislý na DR, RLO a bloku DFTM

syntax      TM          citac
            TM          [TYPE.]citac
            TM          TYPE.(citac+n)
            TM
                              TYPE = BYTE. WORD.
```

Instrukce **TM** pracuje jak s registrem DR, tak s registrem RLO. Jako operand této instrukce je nutno zadat symbolickou adresu slabiky paměti, která bude sloužit pro čítání příslušného času (typ BYTE nebo WORD).
Pro systémy řady CNC8x9 –DUAL je parametr nepovinný. V tomto případě si překladač definuje "automatickou" proměnnou pro tento čítač.
Čítání času v instrukci TM je závislé od toho, v jakém časovém bloku DFTM se instrukce TM nachází. Když se například instrukce TM nachází v úseku programu definovaném instrukcí DFTM10 (je aktivován po 10 sec.), nastavená doba je v desítkách vteřin.

Instrukce TM pracuje takto:

1) Je-li RLO = 0, obsah zadaného čítače se nuluje a instrukce tím končí.

2) Je-li RLO = 1, provede se porovnání obsahu zadaného čítače s obsahem datového registru DR.
   a) Je-li /ČÍTAČ/ >= DR, nastaví se RLO do stavu 1 a instrukce tím končí.
   b) Je-li /ČÍTAČ/ < DR, nastaví se RLO do stavu 0 a provede se zvětšení zadaného čítače o jedničku (V bloku DFTM01 to znamená čas 0,1 sec. a v bloku DFTM1 čas 1 sec.).

Kromě vlastních instrukcí TM musí být v bloku časovačů ještě instrukce pro nastavení registru DR a RLO počátečními podmínkami a samozřejmě také instrukce pro nastavení jedné nebo více paměťových buněk dle výsledku této instrukce (RLO).

Příklad:
Realizujte tuto časově závislou funkci:
Je-li bit ALFA po dobu delší než 0,4 sec. ve stavu 1, nastavte do stavu 1 též bit GAMA. K čítání času použijte slabiku paměti CITACA.

```
            EQUI        DOBA,4

            DFTM01      NAV30
            ...
            LDR         ALFA
            LOD         DOBA
            TM          CITACA
            FL1         1,GAMA
            ...
NAV30:
```

Příklad:
Je-li logický součin A1 a A2 po dobu delší než 5 sec. roven nule, vynulujte bit GAMA a vynulujte buňku BYTE. K čítání času použijte slabiku CITACB.

```
            DFTM1       NAV50
            ...
            LDR         A1
            LA          A2
            CA
            LOD         CNST.5
            TM          CITACB
            FL1         0,GAMA
            LOD         CNTS.0
            STO1        BYTE
            ...
NAV50:
```

Příklad:
Je-li signál TLAK po rekonfigurovatelnou dobu (REKONFIG+20) roven jedné, nastavte bit HAVAR a buňku POCET na 67h:

```
            LDR         TLAK
            LOD         WORD.(REKONFIG+20)
            TM          WORD.(CITAC+10)
            FL1         1,HAVAR
            LOD         CNTS.67H
            STO1        POCET
```

instrukce	**CU** **CD** **CUBCD**

funkce	**CU**	**čítač nahoru závislý na DR, RLO a bloku DFTM**
	CD	**čítač dolů závislý na DR, RLO a bloku DFTM**
	CUBCD	**BCD čítač nahoru závislý na DR, RLO a bloku DFTM**
syntax	**CU**	**citac**
	CD	**citac**
	CUBCD	**citac**

Čítačové instrukce jsou určeny k realizaci čítacích funkcí. Instrukcí čítač nahoru **CU** se provádí v případě, že je splněna podmínka pro čítání (nulovací vstup čítače je sepnut), inkrementování buňky, určené adresovou částí instrukce CU. Ke změně stavu čítače dojde pouze při změně stavu bitu, určeného jako vstup čítače ze stavu log.0 do stavu log.1. Zároveň je porovnáván stav datového registru DR, ve kterém je uložena předvolba čítače se stavem adresovaného čítače. V případě rovnosti čítače a DR je nastaven RLO do log. 1, v opačném případě do log.0. Po skončení instrukce CU je obsah adresovaného čítače uložen v datovém registru DR.

Instrukce čítače nahoru CU pracuje s daty v binárním kódu v rozsahu byte nebo word. Po dosažení nastavené hodnoty se čítače nenulují a není žádné předdeklarování čítačů.

Instrukce čítač dolů **CD** pracuje podobně s tím rozdílem, že provádí dekrementaci čítače. Instrukce **CUBCD** pracuje podobně jako instrukce CU, ale v BCD kódu.

Příklad:
Změny vstupního signálu ALFA z nuly do jedničky jsou čítány, pokud blokovací vstup BETA je v jedničce.

```
            LDR         ALFA        ;VSTUP
            LA          BETA        ;BLOKOVÁNÍ
            LOD         GAMA        ;NAČTENÍ PŘEDVOLBY
            CU          DELTA       ;ČÍTAČ (BYTE, WORD)
            JL0         NAV1
            ...
NAV1:
```

Příklad:
Vynulování čítače po dočítání na zadanou předvolbu GAMA

```
            LDR         ALFA        ;VSTUP
            LOD         GAMA        ;NAČTENÍ PŘEDVOLBY
            CU          DELTA       ;ČÍTAČ (BYTE, WORD)
            LOD         CNTS.0
            STO1        DELTA       ;VYNULOVÁNÍ
```

3.12 Aritmetické a logické instrukce s operandy a DR registrem

Výše uvedené instrukce jsou určeny k provádění aritmetických nebo logických operací, přičemž většinou platí pravidlo, že operace se provádí mezi DR a obsahem paměti nebo konstantou. Adresa paměti se uvádí jako parametr instrukce, vyjma instrukcí INR, DCR, BIN, BCD, ABS, INV, RR a RL které jsou bez operandu.

instrukce	**AD** **SU**

```
funkce      AD          sčítání byte, word nebo konstanty do DR
            SU          odčítání byte, word nebo konstanty od DR

syntax      AD (SU)     adr
            AD (SU)     [TYPE.]adr
            AD (SU)     TYPE.(adr+a )
            AD (SU)     CNTS.konst
                              TYPE = BYTE. WORD. HIGH. DWRD.
```

Instrukce AD sečte obsah DR registru s obsahem paměti, na kterou ukazuje operand nebo konstantou. Výsledek operace zůstane v DR registru. Instrukce může pracovat s operandy typu BYTE, WORD nebo DWORD .

Instrukce SU odečte od obsahu DR registru obsah paměti, na kterou ukazuje operand nebo konstantu. Výsledek operace zůstane v DR registru. Instrukce může pracovat s osmibitovými (byte) i šestnáctibitovými (word) operandy.

Předefinování typu je posáno v kapitole "Způsoby předefinování typu u datových proměnných".

instrukce	**MULB** **DIVB**

```
funkce      MULB        násobení DR registru a operandu
            DIVB        celočíselné dělení DR registru s operandem

syntax      MULB(DIVB)  adr
            MULB(DIVB)  TYPE.adr
                              TYPE = BYTE. CNTS. HIGH. WORD. DWRD.
```

Instrukce **MULB** vynásobí registr DR s obsahem paměti, na kterou ukazuje operand. Výsledek operace zůstane v DR registru. Instrukce může pracovat s osmibitovými (byte) operandy a výsledek je typu word. Při použití předeklarování typu operandu pomocí prefixu **"WORD."** pracuje instrukce se šestnáctibitovými (word) operandy a výsledek je 32 bitů (dword) v rozšířeném 32 bitovém DR registru.

Instrukce **DIVB** vydělí registr DR (obecně typu word) s obsahem paměti, na kterou ukazuje osmibitový (byte) operand. Výsledek operace zůstane v DR registru. Dělení je celočíselné, eventuální zbytek se ignoruje. Výsledek je maximálně velikosti 8 bitů.

Při použití předeklarování typu operandu pomocí prefixu **"WORD."** instrukce vydělí obsah rozšířeného 32 bitového DR registru se šestnáctibitovým (word) operandem a výsledek zůstane v DR registru maximálně o velikosti 16 bitů. V tomto případě je nutno před použitím instrukce zabezpečit naplnění 32 bitového DR registru.

Při použití procesoru CPU04 s příkazem P386 (viz dále) je povoleno také násobení a dělení čtyřbytevých (double-wordových) operandů. Při násobení se použije předeklarování typu na **"DWRD"**. V tomto případě pracuje instrukce s 32 bitovými (double-word) operandy a výsledek je 64 bitů (8 byte = QWORD). Nižších 32 bitů je uloženo do rozšířeného 32-bitového DR registru. Předpokládá se, že po násobení typu DWRD bude

bezprostředně následovat dělení s předeklarováním typu **"DWRD".** Instrukce vydělí obsah 64 bitů po násobení s 32-bitovým (double-word) operandem a výsledek zůstane v rozšířeném 32 bitovém DR registru.

Příklad:
Hodnotu z buňky ALFA (BYTE) zmenšenou o 23h vynásobte hodnotou z buňky BETA (BYTE) a výsledek zapište do buňky GAMA (WORD).

```
LOD         ALFA            ;načte ALFA do DR
SU          CNST.23H        ;odečte 23h
MULB        BETA            ;vynásobení s BETA (výsledek 16 bitů)
STO         GAMA            ;zapíše do GAMA
```

Příklad:
Součet buňky ALFA (WORD) a BETA (BYTE) zapište do VYSLEDEK1, odečtěte konstantu 123h a vynásobte buňkou GAMA (WORD). Výsledek 32 bitů zapište do buňky VYSLEDEK2 (DWORD).

```
LOD         ALFA            ;načte ALFA do DR
AD          BETA            ;připočte k DR buňku BETA
STO         VYSLEDEK1       ;zapíše do VYSLEDEK1
SU          CNST.123H       ;odečte 123h
MULB        WORD.GAMA       ;vynásobení s  GAMA (výsledek 32 bitů)
STO         DWRD.VYSLEDEK2  ;zapíše výsledek 32 bitů do VYSLEDEK2
```

Příklad:
Podíl 32 bitové buňky DELENEC (DWORD) a 16 bitové buňky DELITEL (WORD) zapište do buňky PODIL (WORD).

```
LOD         DWRD.DELENEC    ;načte dělenec 32 bitů do DR
DIVB        WORD.DELITEL    ;dělení 32 bitového DR s dělitelem 16 bitů
STO         PODIL           ;výsledek v PODIL 16 bitů
```

instrukce	**ORB** **ANDB** **XORB**

```
funkce      ORB   logický OR po bitech mezi DR registrem a pamětí,konst.
            ANDB  logický AND po bitech mezi DR registrem a pamětí, konst.
            XORB  logický XOR po bitech mezi DR registrem a pamětí, konst.

syntax      ORB (ANDB,XORB)   adr
            ORB (ANDB,XORB)   [TYPE.]adr
            ORB (ANDB,XORB)   TYPE.(adr+n)
            ORB (ANDB,XORB)   CNTS.konst
```

Výše uvedené instrukce jsou určeny k provádění logických operací, přičemž platí pravidlo, že operace se provádí mezi DR a obsahem paměti nebo konstantou. Adresa paměti se uvádí jako parametr instrukce.

Instrukce provedou logický OR, AND nebo XOR po jednotlivých bitech mezi DR registrem a pamětí nebo konstantou.

3.13 Bezoperandové instrukce pro práci s DR registrem

Některé bezoperandové instrukce pro operace s datovým DR registrem mohou pracovat s rozšířeným 32 bitovým DR registrem (dword). Jedná se o instrukce **INR, DCR, INV, ABS, RR**, **RL, CONDR, CONRD**. V tomto případě je nutné modifikovat bezoperandové instrukce pomocí parametru **DWRD**.

instrukce	**INR** **DCR** **INRBCD**

funkce	**INR**	**inkrement DR registru**
	DCR	**dekrement DR registru**
	INRBCD	**inkrement DR registru v BCD tvaru**
syntax	**INR (DCR)**	
	INR (DCR)	**[DWRD]**
	INRBCD	

Instrukce **INR** bez operandu zvětší registr DR o jedničku. Při překročení rozsahu začíná od nuly.
Instrukce **DCR** bez operandu zmenší registr DR o jedničku.
Instrukce **INRBCD** zvětší registr DR o jedničku v BCD kódu. Pracuje v rozsahu 0.. 9999.
Instrukce INR a DCR s parametrem **DWRD** zvětšují nebo zmenšují rozšířený 32 bitový DR register.

instrukce	**BIN** **BCD**

funkce	**BIN**	**převod BCD → BIN kódu**
	BCD	**převod BIN → BCD kódu**
syntax	**BIN**	
	BCD	
	BIN	**[DWRD]**
	BCD	**[DWRD]**

Instrukce **BIN** bez operandu převede číslo ve tvaru BCD, uložené v registru DR (maximálně 16 bitů) na binární číslo. Výsledek zůstane v DR registru.
Instrukce **BIN** s parametrem **DWRD** převede číslo v rozšířeném 32 bitovém DR registru. Záporné BCD číslo před převodem má nastavenou na jedničku bit s váhou 31.

Instrukce **BCD** bez operandu převede číslo v binárním tvaru, uložené v registru DR na BCD číslo. Výsledek zůstane v DR registru. Hodnota v DR registru před převodem nesmí být větší než 9999d = 270Fh.
Instrukce **BCD** s parametrem **DWRD** převede číslo v rozšířeném 32 bitovém DR registru. Hodnota v DR registru před převodem nesmí být větší než 99999999d = 5F5E0FFh.

instrukce	RL RR

```
funkce      RL          logický posuv DR registru vlevo
            RR          logický posuv registru vpravo

syntax      RL (RR)     n
            RL (RR)     [TYPE.]n
            RL (RR)     [TYPE.]n[,DWRD]
                              TYPE = BYTE. HIGH. CNST.
```

Instrukce **RL n** provede logický posuv DR vlevo o "n" bitů. Operand je konstanta nebo hodnota v buňce (maximálně o velikosti 8 bitů) udávající počet rotací.
Instrukce **RR n** provede logický posuv DR vpravo o "n" bitů. Operand je konstanta nebo hodnota v buňce (maximálněo velikosti 8 bitů) udávající počet rotací.
Instrukce mohou mít druhý parametr **DWRD**, který udává, že se provede posuv rozšířeného 32 bitového DR registru.

instrukce	INV ABS

```
funkce      INV         negace DR registru (dvojkový doplněk)
            ABS         absolutní hodnota DR registru

syntax      INV (ABS)
            INV (ABS)   [DWRD]
```

Instrukce **INV** provede negaci DR registru, to je dvojkový doplněk.
Instrukce **ABS** provede absolutní hodnotu DR registru.
Instrukce mohou mít nepovinný parametr **DWRD**, který modifikuje instrukce tak, že negace nebo absolutní hodnota se provede nad rozšířeným 32 bitovým DR registrem.

Příklad:
Různé operace:

```
LOD         ALFA              ;načtení ALFA (typ podle deklarace)
INR                           ;inkrementace DR (16 bitů)
ABS                           ;absolutní hodnota DR (16 bitů)
RL          POSUN             ;posun DR doleva o hodnotu v POSUN
STO         VYSLEDEK1         ;zápis do VYSLEDEK1
INV                           ;dvojkový doplněk DR (16 bitů)
RR          CNST.3                  ;posun DR doprava o 3 bity
STO         VYSLEDEK2         ;zápis do VYSLEDEK2

LOD         DWRD.BETA         ;načtení 32 bitů z BETA do DR
ABS         DWRD              ;absolutní hodnota DR 32 bitů
INR         DWRD              ;inkrementace DR 32 bitů
RR          CNST.18,DWRD      ;posun o 18 bitů vpravo registru DR32
STO         DWRD.VYSLEDEK3    ;zápis do VYSLEDEK3 32 bitů
```

3.14 Logické instrukce s operandy a DR registrem

Skupina logických instrukcí provede porovnání DR registru s obsahem paměti nebo konstantou a na základě výsledku porovnání se nastaví bitový RLO registr.

instrukce	**EQ** **EQ1** **LT** **GT** **LE** **GE**

funkce

EQ	**porovnávání DR registru s operandem**
EQ1	**podmíněné porovnávání DR s operandem, když RLO = 1**
LT	**DR je menší než operand**
GT	**DR je větší než operand**
LE	**DR je menší nebo rovno než operand**
GE	**DR je větší nebo rovno než operand**

syntax

EQ (EQ1,LT,GT,LE,GE)	**adr**
EQ (EQ1,LT,GT,LE,GE)	**[TYPE.]adr**
EQ (EQ1,LT,GT,LE,GE)	**TYPE.(adr+n)**
EQ (EQ1,LT,GT,LE,GE)	**CNTS.konst**
	TYPE = BYTE. HIGH. WORD. DWRD.

Instrukce **EQ** je logická a provádí porovnání DR registru s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR shodný s obsahem paměti nebo s konstantou, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován. Operandy mohou být typu byte, word, konstanta nebo dword .

Instrukce **LT** resp. **LE** jsou logické a provádí porovnání DR registru s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR menší resp. menší nebo roven než obsah paměti nebo konstanty, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován.

Instrukce **GT** resp. **GE** jsou logické a provádí porovnání DR registru s obsahem paměti, na kterou ukazuje operand nebo konstantou. Je-li DR větší resp. větší nebo roven než obsah paměti nebo konstanty, je nastaven RLO registr do jedničky, v opačném případě je RLO vynulován.
Instrukce **EQ1** je zavedena pro přiblížení se ke kompatibilitě s automatem NS915. Porovnání se provede jenom tehdy, když je v registru RLO hodnota **1**, jinak zůstane v RLO hodnota 0. Instrukci EQ1 možno použít na porovnání větších paměťových oblastí, protože instrukce EQ1 je možné zřetězit. Instrukce EQ1 se dá nahradit pomocí dvou instrukcí a jednoho návěští:

```
            JL0   OBSKOK
            EQ    BUNKA ekvivalent:       EQ1   BUNKA
OBSKOK:
```

Při použití prefixu "**DWRD.**" před operandem způsobí porovnání rozšířeného 32 bitového DR registru s 32 bitovým operandem (DWORD).

Předefinování typu je posáno v kapitole "Způsoby předefinování typu u datových proměnných".

Příklad:
Skok na návěští MENSI, když buňka BUNKA1 je menší něž buňka BUNKA2

```
      LOD         BUNKA1            ;načte BUNKA1 do DR
      LT          BUNKA2            ;když je DR < BUNKA2 tak RLO=1, jinak 0
      JL1         MENSI             ;skok na menší, když je RLO=1
```

Příklad:
Porovnejte hodnotu z buňky NASTAVENI s buňkou BUNKA, když výraz ALFA*NOT(BETA) je roven 1. Výsledek zapište do GAMA:

```
LDR         ALFA            ;načte bit ALFA do RLO
LA          -BETA           ;logický součin RLO s negací BETA
LOD         NASTAVENI       ;načte buňku NASTAVENI do DR
EQ1         BUNKA           ;podmíněné porovnání když RLO=1
WR          GAMA            ;zápis RLO do GAMA
```

Příklad:
Porovnejte mezi sebou oblasti paměti PAM1 a PAM2 o velikosti 8 BYTE.

```
LOD         DWRD.PAM1       ;načte 4 BYTE z PAM1 do DR 32 bitů
EQ          DWRD.PAM2       ;porovnání DR 32 bitů z 4 BYTE PAM2
LOD         DWRD.(PAM1+4)   ;načte další 4 BYTE s PAM1 do DR 32 bitů
EQ1         DWRD.(PAM2+4)   ;další porovnání se provede jen když
                            ;při prvním byla rovnost RLO=1
JL1         ROVNO           ;odskok při rovnosti
```

Příklad:
Nastavte bit AKCE do hodnoty 1, když buňka MATTL je rovna kódu ´W´

```
LOD         MATTL           ;načte buňku MATTL do DR
EQ          CNST.'W'        ;porovnání DR s konst. a nastavení RLO
FL1         1,AKCE          ;když RLO=1 nastaví bit AKCE na 1
```

Příklad:
Při přetečení diferenčního čítače DIFCIT_X (DWORD) přes hodnotu LIMIT skok na ERROR.

```
CLI                         ;zákaz přerušení (ASM86)
LOD         DWRD.DIFCIT_X   ;načte diferenční čítač 32 bitů do DR
STI                         ;povolení přerušení (ASM86)
ABS         DWRD            ;absolutní hodnota 32 bitového DR reg.
GE          DWRD.LIMIT      ;když DR(32 bitů) >= LIMIT (DWORD), tak
                            ;RLO=1, jinak RLO=0
JL1         ERROR           ;skok na chybu
```

3.15 Konverze a přesuny registrů a paměti

instrukce	CONDR CONRD

funkce	**CONDR**	**konverze DR → RLO**
	CONRD	**konverze RLO → DR**
syntax	**CONDR (CONRD)**	
	CONDR	**bit**
	CONDR (CONRD)	**[DWRD]**

Instrukce **CONDR** provede konverzi registru DR do bitového registru RLO. Pokud je registr DR nulový, naplní se RLO registr na hodnotu 0. Pokud je registr DR nenulový, naplní se RLO registr na hodnotu 1. Instrukce CONDR neovlivní obsah DR registru.
Pokud instrukce CONDR obsahuje operand, kterým je číslo 0 - 31, instrukce přesune z datového registru DR do bitového registru RLO jen odpovídající bit.

Instrukce **CONRD** provede konverzi bitového registru RLO do datového DR registru. Pokud je registr RLO nulový, naplní se DR registr na hodnotu 0h. Pokud je registr RLO roven hodnotě 1, naplní se DR registr na hodnotu FFFFh. Instrukce CONRD neovlivní obsah RLO registru. Instrukce CONRD je ***koncová instrukce*** pro logické výrazy.
Při použití parametru **DWRD** se provede konverze s rozšířeným 32 bitovým DR registrem.

instrukce	MV

funkce	**MV**	**přesun paměti**
syntax	**MV**	**zdroj,cil,pocet**

Instrukce **MV** slouží pro přesun oblasti paměti. Parametr "zdroj" je adresa zdroje - odkud se bude přesouvat, parametr "cil" je adresa cíle - kam se bude paměťová oblast přesouvat a parametr "pocet" je počet přesouvaných bajtů paměti.

instrukce	CLEAR

```
funkce      CLEAR       nulování paměti

syntax      CLEAR       zacatek, cil
            CLEAR       GLOBAL, ALL
            CLEAR       INTERNAL, ALL
```

Instrukce **CLEAR** slouží pro nulování paměti. Parametr "zacatek" je adresa začátku nulované oblasti a parametr "cil" je adresa konce nulované oblasti. Instrukce v systémech řady CNC8x9 vynuluje jak paměť pro globální proměnné, tak paměť pro lokální proměnné. Příslušná oblast je určena podle výskytu adres proměnných v parametru instrukce.

Instrukce "**CLEAR GLOBAL, ALL**" vynuluje všechna globální data včetně inicializačních proměnných mechanizmů a časovačů. Proto po této instrukci musí proběhnout nová inicializace všech mechanizmů. Instrukce se používá například v modulu PIS_CLEAR (MODULE_CLEAR), aby nulování (start a stop) PLC programu bylo ekvivalentní s průchodem modulu PIS_INIT (MODULE_INIT), který proběhne jen při zapnutí stroje.

Instrukce "**CLEAR INTERNAL, ALL**" vynuluje všechna lokální data definovaná návrhářem PLC programu včetně "automatických" proměnných definovaných v rozvoji instrukcí jazyka PLC836. Instrukce se používá například v modulu PIS_CLEAR (MODULE_CLEAR).

3.16 Procedury

Definování a volání procedur platí jen pro řadu systémů CNC8x9 – DUAL. Ve všech souborech PLC programu mohou být definovány procedury (podprogramy)a také ve všech souborech mohou být libovolná volání procedur definovaných i v jiných souborech.

instrukce	PROC_BEGIN PROC_END PROC_CALL

```
funkce          PROC_BEGIN  začátek definice procedury
                PROC_END    konec definice procedury
                PROC_CALL   volání procedury

Syntax          PROC_BEGIN  nazev
                PROC_END    nazev
                PROC_CALL   nazev
```

Definice procedury se provede pomocí příkazů **PROC_BEGIN** a **PROC_END**, které mají jako parametr název procedury. Umístění procedury při její definici může být na libovolném místě v modulu PROVOZ_VYSTUP (MODULE_MAIN) . Sled vykonávání instrukcí přeskočí definiční oblast procedury. Procedura se vykoná jen pomocí instrukce **PROC_CALL** s příslušným parametrem, který je názvem procedury. Po vykonání procedury se sled vykonávání instrukcí vrátí za místo volání procedury.

Událostní procedury:

V jazyku PLC836 jsou rezervovány některé názvy procedur, které slouží pro konkrétní účel a jsou automaticky spuštěny při dané události. Jedná se o tyto názvy procedur:

_ON_ESET	Procedura se zavolá automaticky při vzniku PLC chyby. Umístění procedury může být v libovolném souboru s PLC programem. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky z rozvoje instrukce ESET. (viz kapitolu „Chybová hlášení, varování a informační hlášení z PLC programu“.
_ON_MSET	Procedura se zavolá automaticky při vzniku informačního hlášení z PLC programu. Umístění procedury může být v libovolném souboru s PLC programem. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky z rozvoje instrukce MSET. (viz kapitolu „Chybová hlášení, varování a informační hlášení z PLC programu“.
_ON_REK	Procedura se zavolá automaticky při změně strojních konstant. Umístění procedury může být v libovolném souboru s PLC programem. Překladač TECHNOL zkoumá existenci takovéto procedury a v případě, že existuje, spustí ji automaticky po nové rekonfiguraci systému. Na tomto místě mohou být nové převody z BCD tvaru do binárního pro strojní konstanty využité v PLC programu.

3.17 Práce s textovými řetězci

Od verze PLC překladače 6.321 je možnost vysílat textové zprávy z PLC programu do záznamu událostí. Textové zprávy do záznamu událostí mohou být vysílány trvale nebo jich může PLC program využít jen jako záznam ladících informací. (přesný popis je v Návodu k programování PLC, Chybová hlášení, varování a informační hlášení z PLC programu.) Když se při prohlížení událostí zvolí příslušný filtr, vznikne tak vlastně jednoduchý PLC terminál, který umožní časově analyzovat sledovaný děj.
Kromě událostí se práce s textem uplatní při obsluze PLC obrazovek a při točítku s alfanumerickým displejem.

Pro práci s textovými řetězci je potřeba mít možnost definovat řetězec, skládat a přesouvat řetěze a konvertovat čísla na řetězce.

instrukce	**STR**

Funkce **STR** **definice textového řetězce**

Syntax

```
TX1:  STR   n
TX1:  STR   n [,'ABCDabcd..']
TX1:  STR   n [,LABEL_MEM + xy]
TX1:  STR   n [,TAB_TECHNOL.TCH_DATA + xy]
TX1:  STR   n [,STCH_IN_FIELD + xy]
TX1:  STR   n [,STCH_OUT_FIELD + xy]
TX1:  STR   n [,BUN + xy]
```

Instrukce **STR** definuje textový řetězec.
Instrukce musí mít povinně návěští, které budě dále sloužit pro identifikaci definovaného řetězce. Návěští bude známo a přístupno ve všech souborech PLC programu. Instrukce STR se může umístit v libovolném modulu PLC programu. (Překladač ve skutečnosti definuje také obecný pointer, který je nasměrovaný na příslušný řetězec.)

První parametr instrukce je povinný a vyjadřuje maximální počet znaků v řetězci. (Překladač ve skutečnosti vymezí o jeden bajt víc pro koncový znak řetězce.)

Příklad:

```
TX_POKUS:   STR   50
```

Druhý parametr instrukce je nepovinný a může obsahovat:

Druhý parametr je příme zadání textového řetězce. Text musí být ohraničen apostrofy a měl by mít maximálně tolik znaků, kolik vymezuje 1. parametr instrukce. (Překladač zabezpečí předplnění zadaného řetězce při každém přenosu nového PLC programu)

Příklad:

```
TX_ZPR:     STR   10,'Zapnuto'
```

Druhý parametr je odkaz do některé z pamětí LABEL_MEM, TAB_TECHNOL, STCH_IN_FIELD a STCH_OUT_FIELD. Paměťové oblasti se používají například jako zálohovaná paměť, nebo pro tvorbu PLC obrazovek.

Příklady:

```
TX_SCR1:    STR   15,STCH_OUT_FIELD+124

EQUI        TEXT_ZAP,124                ;symbolický offset
TX_SCR2:    STR   15,STCH_OUT_FIELD+TEXT_ZAP
```

Druhý parametr je odkaz na libovolnou paměťovou proměnou, kromě definice textu, například definovaných pomocí instrukcí DFM a DS. Potom instrukce STR musí být umístěna v tom samém modulu.

Příklad:

```
PAM25:      DS    20          ;pole v datové části PLC programu
TX_BUN25:   STR   15,PAM25
```

instrukce	**STRCPY** **STRADD**

Funkce **STRCPY** **kopírování textových řetězců**
STRADD **spojení textových řetězců**

```
Syntax   STRCPY (STRADD)   text1, text2
         STRCPY (STRADD)   text1, 'ABCDabcd..'
         STRCPY            text1, text2 [,pocet]
         STRCPY            text1+xx, text2+yy
         STRCPY            text1+xx, text2+yy, pocet
         STRCPY            text1+BX, text2+yy, pocet
```

Instrukce **STRCPY** překopíruje textový řetězec na který ukazuje druhý operand, do textového řetězce na který ukazuje první operand. Když je druhý operand přímé zadání textového řetězce, překopíruje jej do řetězce podle prvního operandu. Oba řetězce jsou definovány pomocí instrukce STR. Řetězec podle prvního operandu musí mít rezervován dostatek prostoru.

Počet kopírovaných znaků může být určen:

Pokud v instrukci není zadán 3.parametr, který určuje počet znaků pro překopírování:		
	1.	Kopírování se ukončí výskytem koncového znaku ve zdrojovém řetězci (text2). Koncový znak je binární 0 a překopíruje se jako poslední. (cílový řetězec může být zkrácen)
	2.	Kopírování se ukončí, když je dosažena velkost cílového řetězce (text1). Zdrojový řetězec by se do cílového řetězce nevešel.
Pokud je v instrukci zadán 3. parametr, který určuje počet znaků:		
	1.	Kopírování se ukončí výskytem koncového znaku ve zdrojovém řetězci (text2). Koncový znak je binární 0 a překopíruje se jako poslední. (cílový řetězec může být zkrácen)
	2	Kopírování se ukončí, když je dosažena velkost cílového řetězce (text1). Zdrojový řetězec by se do cílového řetězce nevešel. (cílový řetězec zůstane v původní délce)
	3.	Kopírování se ukončí dosažením zadaného počtu znaků. (cílový řetězec nebude zkrácen)

Pokud potřebujeme kopírovat řetězec do jiného řetězce na konkrétní pozici, můžeme použít instrukci s offsetem (text1+10) (platí od verze 6.388).

Když řetězce obsahují libovolná binární čísla (včetně binární 0), tak musíme místo instrukce STRCPY použít instrukci MEMCPY (viz dále).

Instrukce **STRADD** připojí textový řetězec na který ukazuje druhý operand, do textového řetězce na který ukazuje první operand. Když je druhý operand přímé zadání textového řetězce, připojí jej do řetězce podle prvního operandu.

Příklady:

```
TEXT1:      STR   50, 'prvni text '
TEXT2:      STR   20, 'druhy text '
TEXT3:      STR   50, STCH_OUT_FIELD + 100
TEXT4:      STR   3 , 'ABC'

      STRADD      TEXT1, TEST2      ;spojení textů TEXT1 <- TEXT1+TEXT2

      STRCPY      TEXT3,'OBR. '     ;naplní TEXT3 v STCH_OUT_FIELD+100
      STRADD      TEXT3,TEXT2       ;připojí k TEXT3 ještě TEXT2

      STRCPY      TEXT2+6, TEXT4    ;v řetězci TEXT2 přepíše ‚text' na ‚ABC'
                                    ;řetězec bude zkrácen

      STRCPY      TEXT2+6, TEXT4,2  ;v řetězci TEXT2 přepíše ‚te' na ‚AB'
                                    ;řetězec nebude zkrácen

      STRCPY      TEXT3+BX, TEXT4   ;v řetězci TEXT3 na offsetu BX se
                                    ;se přepíše 'ABC'
```

instrukce	**BINSTR** **BCDSTR**

Funkce **BINSTR** **převod binárního čísla na řetězec**
BCDSTR **převod BCD hodnoty na řetězec**

Syntax **BINSTR (BCDSTR) text1**
BINSTR (BCDSTR) text1[,DWRD]

Instrukce **BINSTR** převede binárně hodnotu z datového registra DR na řetězec na který ukazuje první parametr instrukce. První parametr instrukce ukazuje na řetězec, který musí mít velkost definovanou instrukcí STR minimálně 1 a maximálně 10 znaků (bajtů). Instrukce podle velkosti řetězce přizpůsobí převod. Pokud je číslo v DR registru menší, doplní se řetězec na prvních místech (první zleva) znaky nul.

Instrukce **BCDSTR** je podobná jako instrukce BINSTR, ale hodnotu v DR registru převádí na řetězec jako hodnotu v BCD kódu..

Instrukce mohou mít druhý parametr **DWRD**, který udává, že se provede převod z rozšířeného 32 bitového DR registru.

Příklad:

```
TEXT1: STR        4                ;pro převod 4 cifry
TEXT2: STR        30

      LOD         BUN1             ;wordová buňka
      BINSTR      TEXT1            ;převede word na řetězec TEXT1
      STRCPY      TEXT2,'POCET-'
      STRADD      TEXT2,TEXT1      ;připojí převedený řetězec
```

instrukce STRCMP

Funkce STRCMP porovnání textových řetězců

Syntax

```
STRCMP          text1, text2
STRCMP          text1, 'ABCDabcd..', pocet
STRCMP          text1, text2 [,pocet]
STRCMP          text1+xx, text2+yy
STRCMP          text1+xx, text2+yy, pocet
```

Instrukce **STRCMP** porovnává textové řetězce na které ukazuje první a druhý operand. Druhý operand může být přímé zadání textového řetězce. Řetězce jsou definovány pomocí instrukce STR. Pokud jsou textové řetězce stejné, instrukce nastaví registr RLO na hodnotu 1, jinak bude RLO mít hodnotu 0. Pokud je zadán 3. parametr (pocet), tak se z obou řetězců porovná jen zadaný počet znaků (pokud porovnávání nebude dřív ukončeno nerovností nebo výskytem koncového znaku 0).

Pokud instrukce neobsahuje 3. parametr o počtu znaků, bude se porovnávat počet znaků podle velkosti řetězce, na který ukazuje druhý parametr instrukce včetně koncového znaku.

Příklad:

```
TEXT1: STR        8,'abcdefgh'
TEXT2: STR        8,'abcde'
TEXT4: STR        3,'cde'

      STRCMP      TEXT1+2, TEXT4,3  ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1

      STRCMP      TEXT1+2, TEXT4    ;porovnání řetězců - neshoda
      JL1         OK                ;je nastaveno RLO=0

      STRCMP      TEXT2+2, TEXT4    ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1

      STRCMP      TEXT1+2, 'cde',3  ;porovnání řetězců - shoda
      JL1         OK                ;je nastaveno RLO=1
```

instrukce MEMCPY

```
Funkce   MEMCPY      kopírování binárních řetězců

Syntax   MEMCPY            text1, text2 [,pocet]
         MEMCPY            text1, text2
         MEMCPY            text1+xx, text2+yy
         MEMCPY            text1+xx, text2+yy, pocet
```

Instrukce **MEMCPY** překopíruje binární řetězec na který ukazuje druhý operand, do řetězce na který ukazuje první operand (od verze 6.388). Oba řetězce jsou definovány pomocí instrukce STR. Řetězec podle prvního operandu musí mít rezervován dostatek prostoru.

Instrukce je podobná instrukci STRCPY s rozdílem, že řetězce nemají definovaný koncový znak 0. Kopírování se proto neukončí výskytem koncového znaku ve zdrojovém řetězci (text2). V instrukci MEMCPY se proto vždy doporučuje definovat počet kopírovaných znaků pomocí 3. parametru. Pokud v instrukci není 3.parametr zadán, kopírování se ukončí, když je dosažena velkost cílového řetězce (text1).

instrukce MEMCMP

```
Funkce   MEMCMP      porovnání binárních řetězců

Syntax   MEMCMP            text1, text2 [,pocet]
         MEMCMP            text1, text2
         MEMCMP            text1+xx, text2+yy
         MEMCMP            text1+xx, text2+yy, pocet
```

Instrukce **MEMCMP** porovnává binární řetězce na které ukazuje první a druhý operand (od verze 6.388). Řetězce jsou definovány pomocí instrukce STR. Pokud jsou řetězce stejné, instrukce nastaví registr RLO na hodnotu 1, jinak bude RLO mít hodnotu 0.

Instrukce je podobná instrukci STRCMP s rozdílem, že řetězce nemají definovaný koncový znak 0. Porovnávání se proto neukončí výskytem koncového znaku v řetězci. V instrukci MEMCMP se vždy doporučuje definovat počet porovnávaných znaků pomocí 3. parametru.

Pokud je zadán 3. parametr (pocet), tak se z obou řetězců porovná jen zadaný počet znaků. Pokud instrukce neobsahuje 3. parametr o počtu znaků, bude se porovnávat počet znaků podle velkosti řetězce, na který ukazuje druhý parametr instrukce.

instrukce STRLOAD

Funkce STRLOAD načtení textového řetězce z externího souboru

```
Syntax   STRLOAD          text1, index
         STRLOAD          text1, index [,pocet]
         STRLOAD          text1+xx, index
         STRLOAD          text1+xx, index, pocet
```

Instrukce **STRLOAD** slouží na lokalizaci textů pro PLC program (platí od verze panelu 40.53 a sekundárního procesoru 6.401).
Instrukce STRLOAD načte textový řetězec podle indexu který je určen druhým operandem (viz dále), do textového řetězce na který ukazuje první operand.. První řetězec je definován pomocí instrukce STR a musí mít rezervován dostatek prostoru v paměti.

Pokud potřebujeme načíst řetězec na konkrétní pozici cílového řetězce, můžeme použít instrukci s offsetem (text1+10).

Pokud je zadán 3. parametr „**pocet**“, tak se načitávání řetězce ukončí podle zadaného počtu znaků nebo výskytem koncového znaku (binární 0) nebo vyčerpáním místa v cílovém řetězci.

2. parametr „**index“** udává identifikační kód textu. Jedná se o číselný kód v rozsahu 1 – 2000, takže může být načítáno maximálně 2000 externích textů.

Instrukce STRLOAD má návratovou hodnotu v DR registru:

návratová hodnota v DR registru	**popis**
0	funkce se provedla bez chyb
1	funkce se neprovedla - nízká verze software
2	funkce se neprovedla - zadaný index je mimo rozsah
3	funkce se neprovedla - text se nenašel

Příklad:
Načtení textu č.20 s externího souboru do řetězce TEXT1:

```
TEXT1:      STR   50    ;definice textového řetězce s vymezením prostoru

STRLOAD     TEXT1, 20   ;načtení lokalizovaného textu podle indexu 20
```

3.17.1 Soubory s texty pro instrukci STRLOAD

Systém při zapnutí automaticky prozkoumá existenci souboru s PLC texty podle nastaveného jazyka a v případě že daný soubor existuje načte jej do paměti. Instrukce STRLOAD za běhu PLC programu proto ve skutečnosti nezpůsobí přímý přístup na disk, ale načítá konvertovaný text z paměti systému.

Soubor s PLC texty může být napsán v kódu **Windows** nebo v **Unicode**.

Jazyk systému se určí v konfiguračním souboru CNC836.KNF v parametru **$100**. Soubor s PLC texty musí být umístěn v adresáři **SYSFILES** (cesta je určena parametrem $62) a musí mít název v závislosti na jazykové verzi. Podobně systém řeší i problematiku PLC chyb (musí být nastaven $114 viz. lokalizace textů v novinkách na www.mefi.cz)

jazyk	**parametr $100**	**název souboru s PLC texty**	název souboru s PLC errory
čeština	Czech	**PLCS_CZE.TXT**	PLCERROR.CZE
polština	Polish	**PLCS_POL.TXT**	PLCERROR.POL
maďarština	Hungarian	**PLCS_HUN.TXT**	PLCERROR.HUN
angličtina	English	**PLCS_ENG.TXT**	PLCERROR.ENG
francouzština	French	**PLCS_FRE.TXT**	PLCERROR.FRE
němčina	German	**PLCS_GER.TXT**	PLCERROR.GER
ruština	Russian	**PLCS_RUS.TXT**	PLCERROR.RUS
slovinština	Slovenian	**PLCS_SLO.TXT**	PLCERROR.SLO

Syntaktická pravidla pro zápis textů:

- Soubor může být napsán v kódu WINDOWS nebo v UNICODE
- Soubor musí obsahovat klíč s verzí**: `$TXT 01 .`**
- Identifikace textu **INDEX** je maximálně 4 místní číselný kód v rozsahu 1-2000.
- Za indexem je uveden text ohraničený v uvozovkách **“** .
- Když uvozovky textovou část ukončí a potom nové uvozovky otevřou novou textovou část bez výskytu indexu, bude text spojen (tím je umožněno formátovat text pro lepší přehlednost)
- Komentář začíná středníkem a končí koncem řádku „**;“**
- V textu se mohou vyskytovat Escape sekvence:

sekvence	**význam**	**Hexadecimální kód**
\n	nový řádek	0Ah
\r	return	0Dh
\\	znak obrácené lomítko	5Ch
\“	znak uvozovky	22h
\t	znak tabulátor	09h
\x..	libovolný dvoumístní hexadecimální kód	00h - FFh

Příklad:

```
$TXT 1
10      "Неисправность смазки"      ; český komentář
110     "Čeština: ščřžýáíéě"
1546    "Obecná chyba\n"            ; dvouřádkový text
        "mechanizmu CW"
1547    "Неисправность"             ;  ruský text s indexem 1547
```

3.18 Pomocné příkazy

V jazyku PLC836 je několik pomocných instrukcí, které se uplatní jen v době překladu PLC programu. Všechny dále popsané instrukce jsou bezoperandové.

instrukce	**SYMBOLTAB**
	INTERSTACK
	CHECK
	P386
	CPU04
	VERINSTRU

Instrukce **SYMBOLTAB** způsobí zkrácení tabulky symbolů při překladu PLC programu. Instrukci je vhodné použít až když překladač zahlásí přeplnění tabulky symbolů. Při použití překladu PLC programu prostředky fy. BORLAND instrukce SYMBOLTAB je neúčinná. **U systémech řady DUAL se nemusí zadávat.**

Instrukce **INTERSTACK** způsobí používaní vlastního (interního) zásobníku procesoru ve výpočtech logických rovnic. Použití instrukce INTERSTACK urychlí průchod PLC programu, ale při chybném návrhu PLC programu mohou nastat nebezpečné situace s nevyrovnaným zásobníkem, přestože překladač TECHNOL hlídá vyrovnanost zásobníku. Například je znemožněno nastavit BREAK-POINTER v oblasti programu, která má posunutý zásobník (například vícero instrukcí LDR). **U systémech řady DUAL se nemusí zadávat.**

Instrukce **CHECK** kontroluje vyrovnanost zásobníku. Vyrovnanost zásobníku je kontrolována vždy na konci příslušného modulu. V době ladění programu je možno použít instrukci CHECK, která provede kontrolu vyrovnanosti zásobníku na daném místě a tak pomůže lokalizovat chybné místo v programu.

Instrukce **P386** modifikuje instrukce jazyka PLC836 pro rozvoj do "assembleru 386" pro procesor 80486DX použitý v kartě procesoru CPU04. Datové operace s dvojitou přesností "double-word" pro vnitřní reprezentaci používají místo registrů BP, CX registr ECX. Instrukce vykoná všechny změny, nutné na přechod na procesor CPU04. Instrukci P386 je nutno zadat před instrukci data (viz kapitola Struktura PLC programu). **U systémech řady DUAL se nemusí zadávat.**

Instrukce **CPU04** vykoná všechny změny nutné pro přechod na kartu procesoru CPU04 (s procesorem 80486DX), ale nemodifikuje instrukce jazyka PLC836 pro rozvoj do "assembleru 386". Instrukci CPU04 je možno použít pro případ, kdy PLC program obsahuje mnoho instrukcí přímo v "assembleru 86" (např. počítá s double-wordovými operacemi v registrech BP, CX). Instrukci P386 je nutno zadat před instrukci data (viz kapitola Struktura PLC programu). **U systémech řady DUAL se nemusí zadávat.**

Instrukce **VERINSTRU** modifikuje instrukce podle zadané verze. V parametru instrukce se zadává jméno instrukce, která má být modifikována. K názvu instrukce se pomocí znaku "dolní podtrženo" ('_') připojí údaj o verzi. Pro překladač TECHNOL od verze 2.3 jsou k dispozici tyto modifikace instrukcí podle verze.

FL_V1 FL1_V1 WR_V1

Všechny uvedené modifikace umožní v parametrech instrukcí FL, FL1 a WR použít více bitových operandů, které na rozdíl od základních provedení instrukcí, nemusí být umístěny v jednom bajtu. Operandy se neuzavírají do špičatých závorek, ale se oddělí čárkou. **U systémech řady DUAL se nezadává, protože je modifikace provedena automaticky.**

Od softwerové verze kazety 4.036 je k dispozici modifikace všech instrukcí souvisejících s analogovými vstupy a s odměřováním v závislosti na typu jednotek souřadnic. **U systémech řady DUAL se nemusí zadávat.**