HLA zsebkönyv

1 Áttekintés

Ez a zsebkönyv azoknak készült, akik már ismerősek az x86 assembly nyelvű programozásával és a lehető leggyorsabban szeretnének elkezdeni dolgozni a HLA fordítóprogrammal. A HLA olyan egyetemi hallgatók számára készült, akik már rendelkeznek előzetes magas szintű nyelvű (C/C++, Pascal, Java, stb.) programozási tapasztalatokkal, de még nem dolgoztak assembly nyelvvel. A HLA-val kapcsolatos dokumentáció kétféle forrásból származik: a HLA kézikönyvéből és az “Az assembly nyelvű programozás művészete/32 bites kiadás” könyvből. “Az assembly művészete” könyv szövege diákoknak és kezdőknek szól; nem tételez fel előismereteket és a HLA használatával tanítja az assembly nyelvű programozást. Sajnos, ez a szöveg nem igazán azoknak a programozóknak szól, akik már tudnak assembly nyelven programozni. A HLA kézikönyvei nagyszerűek akkor, ha a nyelv valamely sajátságára vagyunk kíváncsiak. Teljes részletességgel leírják a HLA nyelvet, de a HLA nyelv meglehetősen terjedelmes, így a HLA nyelvvel először találkozó assembly programozónak hatalmas mennyiségű anyagot kell elsajátítania ahhoz, hogy egyáltalán el tudjon kezdeni dolgozni a HLA-val. A legtöbben meg sem próbálják. Ennek a zsebkönyvnek az a célja, hogy a haladó x86 assembly programozónak minél kisebb terjedelemben bemutassa a HLA egy egészen kis részét. Nem is akarja megtanítani a HLA egyetlen különlegességét sem; feltételezi, hogy az olvasó már ismeri a MASM, TASM, NASM, Gas, stb. assemblerek valamelyikét, és csak azt akarja megtanulni, hogyan lehet a HLA-val az említett assemblereknél megszokott módon assembly kódot írni. A HLA tanulásának célja az, hogy annak fejlett tulajdonságait is használni tudjuk. Természetesen, mielőtt futni tudnánk, meg kell tanulnunk járni; ez a kézi könyv járni tanít. Amint az olvasó „hagyományos assembly” értelemben megismerkedett a HLA-val, át kell térnie a HLA kézikönyveire, hogy a nyelv haladóknak szánt tulajdonságait is megismerje.

2 A HLA üzembe helyezése

Ez a zsebkönyv feltételezi, hogy a HLA a számítógünkön már már üzemkész és helyesen működik. Az üzembe állítás részletes leírását a HLA Kézikönyv tartalmazza.

3 A HLA futtatása

A HLA olyan, parancssorból futtatható segédprogram, amelyet a Win32 parancssorából lehet futtatni. Ez a dokumentum feltételezi, hogy az olvasó már jól ismeri az alapvető parancsok kiadásának módját és az olyan parancsokat, mint "DIR" vagy "RENAME". A HLA-t parancssorból egy ilyen paranccsal lehet futtatni:

```
hla esetleges_parancssori_paraméterek fájlnevek_listája
```

A fájlnevek listája egy vagy több egyértelmű fájlnévből áll, amely fájlnevek kiterjesztése HLA, ASM, vagy OBJ lehet. A HLA először a HLAPARSE programot futtatja le valamennyi HLA kiterjesztésű fájlon (egy ugyanolyan nevű, de .ASM vagy .INC kiterjesztésű fájlt hozva létre). Ezután a HLA a MASM programot futtatja az összes ASM kiterjesztésű fájlon (beleértve a HLAPARSE által létrehozott fájlokat is). Végül, a HLA a linker (kapcsolatszerkesztő) programot futtatja az OBJ fájlok (beleértve a MASM által létrehozott .OBJ fájlokat is) összekapcsolására. A végeredmény, feltéve, hogy nem fordult elő hiba, egy .EXE kiterjesztésű végrehajtható fájl.

A HLA a következő parancssori paramétereket értelmezi:

```
-@          Ne készíts linker válasz fájlt.
-aXXXXX     Add át az assemblernek XXXXX-et parancssori paraméterként.
-c          Csak .OBJ fájlokat készíts.
-dXXXXX     Definiáld a XXXXX szibólumot a fordítás idejére 'igaz' értékkel.
-e name     A végrehajtható fájl neve.
-lXXXXX     Add át a linkernek XXXXX-et parancssori paraméterként.
-m          Készíts memóriatérkép fájlt a linkelés során.
-masm       Használd az MASM-et a kimenő fájl készítésére.
-o:omf      A MASM használata során OMF outputot készíts COFF helyett.
-o:win32    Készíts Win32 COFF fájlokat.
-s          Csak .ASM fájlokat készíts.
-sm         MASM-stílusú .ASM fájlokat készíts (alapértelmezett).
-st         TASM-stílusú .ASM fájlokat készíts.
-sym        Készíts szimbólum táblázatot a fordítás után.
-tasm       Használd a TASM32.EXE assemblert.
-test       A diagnosztikai üzeneteket a stdout-ra küldd a stderr helyett
            (Ez a parancs a HLA tesztelésére/hibakeresésére való).
-v          Részletes fordítási üzenetek.
-w          Windows alkalmazásként fordíts (alapértelmezett a konzol).
-?          Írd ki ezt a súgó üzenetet
```

A parancssori paraméterek leírását a HLA Kézikönyvben találja meg. A legtöbb esetben nem lesz szüksége ilyen paraméterek megadására, ha egy tipikus HLA programot fordít. Felhasználóként legtöbbször a "-c" és "-s" paramétereket fogja használni (és ez a dokumentum feltételezi, hogy a felhasználó tudja, hogy azok mire valók).

4 A HLA nyelv elemei

Ezzel a szakasszal megkezdjük a HLA forrásnyelv tárgyalását. A HLA forrásnyelvi fájljai csak 7 bites ASCII jeleket tartalmazhatnak. Ezek Windows szövegfájlok, amelyek kocsi vissza/soremelés párossal végződnek. Nem nyomtatódó jelek a betűköz, a tabulátor és az új sor. Általában a HLA nem fogad el vezérlő karaktereket és hibaüzenetet adhat, ha ilyen jelenik meg a forrás fájlban.

4.1 Megjegyzések

A HLA a "//" jelölést használja egysoros megjegyzések bevezetésére. Határozatlan hosszúságú megjegyzés bevezetésre pedig a "/*", lezárására a "*/" jeleket. A C/C++, Java, és Delphi nyelvek használói megszokottnak érezhetik e jelölés használatát.

4.2 Különleges jelek

A következő jelek a HLA lexikális elemei és a HLA számára különleges jelentéssel bírnak:

```
* / + - ( ) [ ] { } < > : ; , . = ? & | ^ ! @
&&  ||  <=  >=  <>  !=  ==  :=  ..  <<  >>
##  #(  )#  #{  }#
```

Ez a dokumentum nem magyarázza meg valamennyi jel jelentését, csak annyit, amennyi egyszerű HLA programok írásához szükséges. További részletekért lásd a HLA Kézikönyvet.

4.3 Kulcsszavak

A HLA nagyszámú kulcsszót használ (nagyobbrészt gépi utasításokat). Terjedelmi okokból a listát itt nem mutatjuk be; a teljes és időszerű listát a HLA Kézikönyv tartalmazza. Megjegyezzük, hogy a HLA nem engedi meg ezen kulcsszavak használatát azonosítóként, ezért legalább egyszer nézze át ezt a listát, hogy valamennyire megismerje, milyen azonosítókat használhat assembly nyelvű programjaiban. A HLA nem különbözteti meg a kulcsszavakban a kis- és nagybetűket. Azaz, a HLA a „MOV” és „Mov” kulcsszót (továbbá a kis és nagybetűk bármilyen előfordulását) a „mov” kulcsszóként értelmezi.

4.4 Külső szimbólumok és assembler kulcsszavak

A HLA a fordítás során egy assembly nyelvű fájlt készít és meghívja a MASM, TASM és Gas assemblerek valamelyikét a fordítás befejezésére. A HLA a programban talált közönséges azonosítókat azonosítóként adja át az assembly programnak. A HLA azonban nem fordítja le az EXTERNAL szimbólumokat, csak megőrzi azokat a készített assembly nyelvű programban. Emiatt vigyáznunk kell arra, hogy ezek a külső nevek ne ütközzenek a használni kívánt assembler kulcsszavaival, mert olyan esetben az assembler hibaüzenetet fog adni a HLA által készített fájl feldolgozásakor.

Az assembler kulcsszavak listáját a HLA által készített file továbbfordítására használni kívánt assembler (MASM, TASM, vagy Gas) leírásában találjuk meg.

4.5 A HLA azonosítói

A HLA azonosítóinak alfabetikus jellel vagy aláhúzás jellel kell kezdődni. Az első jel után, az azonosító alfanumerikus jeleket és aláhúzást tartalmazhat. A HLA-ban nincs technikai határa az azonosító hosszának, de nem érdemes 32 jelnél hosszabb azonosítókat használni, mivel az azonosítókat feldolgozó assembler és linker esetleg nem tud kezelni ilyen szimbólumokat.

A HLA nem követeli meg, hogy az azonosítókban kis- vagy nagybetűket használjon. Ez azt jelenti, hogy a HLA megkülönbözteti a kis és nagybetűket, tehát az azonosítókat mindig azonos módon kell leírnia. Nem megengedett olyan azonosítók használata, amelyek között kizárólag az a különbség, hogy a szóban kis- vagy nagybetűk szerepelnek.

4.6 Külső azonosítók

A HLA lehetővé teszi, hogy a programozó külső azonosító szövegeket adjon meg. A külső azonosítókra nem vonatkoznak a HLA azonosítók korlátozásai. A HLA megengedi bármilyen string használatát külső azonosítóként. A programozó felelősége, hogy csak a HLA közbülső ASM fájlját feldolgozó assembler által megengedett jeleket használjon. Megjegyezzük, hogy ez a tulajdonság lehetővé teszi olyan jelek használatát, amelyek nem engedélyezettek a HLA-ban, de használhatók a külső kódban (pl. a Win32 API használja a '@' jelet azonosítókban). További részletek az @EXTERNAL kulcsszó tárgyalásánál.

4.7 Adattípusok a HLA-ban

4.7.1 A HLA primitív (egyszerű) adattípusai

A HLA a következő egyszerű adattípusok használatát teszi lehetővé:

```
Egybájtos típusok: byte, boolean, enum, uns8, int8, és char.
Kétbájtos típusok: word, uns16, int16.
Négybájtos típusok: dword, uns32, int32, real32, string, pointer.
Nyolcbájtos típusok: uns64, int64, qword, thunk, és real64.
Tízbájtos típusok: tbyte, és real80.
Tizenhatbájtos típusok: uns128, int128, lword, és cset
```

A fenti típusok részletes tárgyalását lásd a HLA kézikönyvben. Ez a dokumentum a következő típusokat használja:

```
byte, word, dword, string, real32, qword, real64, és real80
```

Ezek azok az adattípusok, amelyeket egy tipikus assembler használni képes.

A BYTE változók és objektumok a -128..+255 tartományba eső numerikus értékeket, ASCII karakter konstansokat, valamint a *true* (1) és *false (0) logikai értékeket* képesek tárolni. Bár a HLA általában végez egy minimális típusvizsgálatot, minden olyan értéket lehet benne tárolni, ami nyolc bitben elfér.

A WORD változók és objektumok a -32768..+65535 tartományba eső numerikus értékeket tárolhatnak. A HLA általában nem engedi meg más értékeknek WORD objektumban való tárolását.

A DWORD változók és objektumok a -2147483647..+4294967295 tartományba eső numerikus értékeket, vagy egy objektum címét (a „&" cím-operátorral) tárolhatják.

A STRING változók szintén DWORD objektumok. A STRING objektumok egy nulla vagy több jelből álló álló, nullával végződő ASCII jelsorozat címét tartalmazza. A string mutatót közvetlenül megelőző négy bájtban a string aktuális hossza található. Az aktuális hossz előtti négy bájt pedig a string maximális lehetséges hosszát tartalmazza. Megjegyezzük, hogy a HLA stringek „csak olvasható" értelemben kompatibilisek a Windows és C/C++ által használt ASCIIZ stringekkel (a csak olvasható itt azt jelenti, hogy a HLA át tud adni a Windows vagy C/C++ függvénynek egy stringet, de a függvénynek nem szabad azt módosítania).

A QWORD, UNS64, és INT64 objektumok nyolc bájtot foglalnak el a memóriában. A HLA az 1.37 változattól kezdve támogatja 64-bites előjel nélküli, előjeles és hexadecimális numerikus állandók használatát. A TBYTE objektumok a memóriában 10 bájtot (80 bitet) foglalnak el, de a HLA nem támogatja a 80-bites egész vagy BCD konstansok használatát a programokban. Az LWORD, UNS128 és INT128 értékek szintén használhatók és 128 bites hexadecimális, előjel nélküli vagy előjeles konstansok használatát támogatják.

A HLA REAL32, REAL64, and REAL80 adattípusai az IEEE három különböző lebegőpontos formátumát támogatják.

4.8 Összetett adattípusok

A fenti egyszerű adattípusokon felül a HLA támogatja tömbök, rekordok (struktúrák), uniók, osztályok és a fenti típusokra mutató pointerek használatát (szöveg objektumokat kivéve).

4.8.1 „Tömb” adattípus

A HLA lehetővé teszi hogy tömb adattípust úgy hozzunk létre, hogy a tömb típusjelzése után a tömb elemeinek számát is megadjuk. Tekintsük a következő HLA típus deklarációt amely az intArray tömböt dword objektumokból álló adattömbként definiálja:

```
type intArray : dword[ 16 ];
```

A "[16]" komponens azt mondja meg a HLA-nak, hogy ebben az adattípusban 16 darab 4 bájt hosszú dupla szó van. A HLA nulla alapú indexelést használ, azaz az első elem mindig a nulladik. Az utolsó elem indexe, ebben a példában, 15 (az összes, 0..15 indexű elemek száma 16).

A HLA többdimenziós tömbök használatát is támogatja. Többdimenziós tömböt úgy adhatunk meg, hogy szögletes zárójelek között megadjuk az indexek listáját, pl.

```
type intArray4x4 : dword[ 4, 4 ];
type intArray2x2x4 : dword[ 2,2,4 ];
```

4.8.2 Rekord adattípus[1]

A HLA rekordok lehetővé teszik, hogy a programozó különböző típusú mezőket tartalmazó adattípusokat hozzon létre. A következő HLA statikus változó deklaráció egy négy mezőt tartalmazó egyszerű rekordot definiál:

```
static Planet: record
                    x:          dword;
                    y:          dword;
                    z:          dword;
                    density:    real64;
                endrecord;
```

Egy Planet típusú objektum futás közben 20 bájtnyi tárterületet foglal el.

Egy rekord mezőinek típusa bármely HLA adattípus lehet, beleértve más összetett adattípusokat is. A rekord mezőinek elérésére a pont jelölést használjuk, pl.

```
mov( Planet.x, eax );
```

1. C/C++ programozóknak: Egy HLA rekord egy C struktúrára hasonlít. Számítógép-nyelv tervezők nyelvén a rekordot gyakran „Descartes szorzatnak” is hívják.

4.9 Konstansok (literálok)

A HLA konstansok számos típusát támogatja. A következő szakaszok ezeket a típusokat írják le.

4.9.1 Numerikus konstansok

A HLA különböző típusú numerikus konstansok használatát teszi lehetővé.

Decimális konstans:

A decimális egész konstansok első és utolsó jegye csak decimális számjegy (0..9) lehet. A konstans belsejében decimális számjegyek és aláhúzás jelek egyaránt előfordulhatnak. Az aláhúzás jelek célja a nagy decimális értékek jobb ábrázolhatósága (azaz aláhúzást használhatunk a nagy számértékek tagolására a vessző helyett). Például: 1_234_265.

Hexadecimális konstans:

A hexadecimális konstansok dollár jellel (“$”) kezdődnek, hexadecimális jellel (0..9, A..F, vagy a..f) folytatódnak és azzal is fejeződnek be. A konstansok belsejében hexadecimális jelek vagy aláhúzás jelek állhatnak. A hexadecimális állandókat könnyebb olvasni, ha a számjegyeket (a legkisebb helyiértékű számtól kezdve) négyesével csoportokra osztjuk, a csoportokat egymástól aláhúzás jellel elválasztva. Például: $1A_2F34_5438.

Bináris konstans:

A bináris konstansok százalékjellel (“%”) kezdődnek és legalább egy bináris számjeggyel (0/1) folytatódnak, továbbá bináris számjeggyel végződnek. A belső pozíciók bináris számjegyeket vagy aláhúzás jeleket tartalmazhatnak. A bináris állandókat könnyebb olvasni, ha a számjegyeket (a legkisebb helyiértékű számtól kezdve) négyesével csoportokra osztjuk, a csoportokat egymástól aláhúzás jellel elválasztva. Például: %10_1111_1010.

Valós (Lebegőpontos) konstans:

A lebegőpontos (valós) állandók mindig decimális számjeggyel (és sohasem csak tizedesponttal) kezdődnek. Az egy vagy több decimális számjegy után tizedespont következhet és nulla vagy több decimális számjegy (a törtrész). A törtrész után “e” vagy “E”, előjel (“+” or “-”) és egy vagy több decimális számjegy következhet (a kitevő rész). Aláhúzás jel előfordulhat a lebegőpontos szám szomszédos számjegyei között, ezek a tagolásra használt vessző jeleket helyettesítik.

Logikai konstans:

A logikai konstansoknak *true* and *false* előre definiált értéke lehet. Megjegyezzük, hogy a programozó átdefiniálhatja ezeket az értékeket, de azt rossz programozási stílusnak tekintjük..

Karakter konstans:

A karakter konstansok általában aposztrófok közé zárt egyetlen (grafikus) karakterből állnak. Az aposztrófot négy idézőjellel ábrázolhatjuk, pl. ‘’’’.

A karakter állandók megadásának egy másik módja a “#” jelet követően egy numerikus (decimális, hexadecimális vagy bináris) konstans megadása. Pl.: #13, #$D, #%1 101.

Szöveg konstans:

A szöveg konstansok idézőjelek közé zárt (grafikus) jelek sorozatából állnak. Egy ilyen szövegben aposztrófot úgy ábrázolhatunk, ha aposztróf párokat írunk be a szövegbe, pl. “He said ““This”” to me.”

Ha két szöveg konstans egymás mellett van a forrásfájlban (azaz csak nem nyomtatódó jelek választják el egymástól) akkor a HLA a két szöveget összefűzi és a parsernek egyetlen szövegként adja tovább. Hasonlóképpen, ha egy karakter konstans van egy szöveg mellett, a HLA a karaktert a szöveghez csatolja és egyetlen szöveggé fűzi össze azokat. Ez például akkor hasznos, ha vezérlő karaktereket kell egy stringben ábrázolni, pl.

```
“This is the first line” #$d #$a “This is the second line” #$d #$a
```

A HLA a fentieket egy olyan szövegként kezeli, ahol mindkét sor után új sor jel (CR/LF) áll.

Pointer konstans:

A HLA csak erősen korlátozott formában engedi meg pointer konstansok használatát. Ha a statikus objektum neve (pl. statikus változó neve, csak olvasható változó neve, nem-inicializált változó, szegmens változó, függvény, módszer vagy iterátor) elé ampersand („&”) jelet teszünk, futási idő alatt kiszámítja annak a változónak a cím offset értékét. Pointer változók nem használhatók konstans kifejezésekben. Pointer konstansokat csak olyan kifejezésekben használhatunk, amelyekkel statikus vagy csak olvasható változókat inicializálunk vagy amely kifejezéseket 80x86 utasításokkal kapcsolatban használunk.

Strukturált konstans:

A HLA támogatja bizonyos strukturált konstansok használatát, beleértve szöveges konstans, tömb konstans és rekord konstans használatát. Részletek a HLA Kézikönyvben.

4.10 Konstans kifejezések a HLA-ban

A HLA a fordítási időben kiszámítandó kifejezések feldolgozására gazdag lehetőségeket kínál. A HLA a következő operátorokat támogatja (csökkenő precedencia szerint rendezve):

```
!(egyoperandusú nem) , - (egyoperandusú negálás)
*, div, mod, /, <<, >>
+, -
=, = =, <>, !=, <=, >=, <, >
&, |, &, in
```

`! expr`

A kifejezésnek bool vagy számértéknek kell lennie. Bool változók esetén, *not ("!")* a standard logikai *nem* műveletet számítja ki. Számok esetén *not ("!")* a szám bitjeinek bitenkénti *nem* műveletét számítja ki.

```
-expr               (egyoperandusú negálás operátora)
expr1 * expr2       (szorzó operátor)
expr1 div expr2     (egész osztó operátor)
expr1 mod expr2     (egész maradék képző operátor
expr1 / expr2       (valós osztó operátor)
expr1 << expr2      (egész bit balra toló operátor)
expr1 >> expr2      (egész bit jobbra toló operátor)
expr1 + expr2       (összeadó operátor)
expr1 - expr2       (kivonó operátor)
expr1 = expr2       (egyenlőségvizsgáló operátor)
expr1 <> expr2      (egyenlőtlenségvizsgáló operátor)
expr1 < expr2       (kisebb mint összehasonlító operátor)
expr1 <= expr2      (kisebb vagy egyenlő összehasonlító operátor)
expr1 > expr2       (nagyobb mint összehasonlító operátor)
expr1 >= expr2      (nagyobb vagy egyenlő összehasonlító operátor)
expr1 & expr2       (logikai/bool ÉS operátor)
expr1 | expr2       (logikai/bool VAGY operátor)
expr1 ^ expr2       (logikai/bool XOR operátor)
( expr )            (operátor precedencia megváltoztatás)
```

A HLA számos más konstans operátort is támogat. Emellett a fenti operátorok némelyikét az operandus típusokra átértelmezik. Megjegyezzük, hogy numerikus (integer) operandusokkal a HLA teljes mértékben támogatja a 128-bites aritmetikát. A részleteket lásd a HLA kézikönyvben.

4.11 A program szerkezete

Egy HLA program a következő általános szintaxist követi:

```
program azonosító ;
    deklarációk
begin azonosító ;
    utasítások
end azonosító ;
```

A három „*azonosító*”-nak azonosnak kell lennie. A deklarációs rész (deklarációk) type, const, val, var, static, uninitialized, readonly, segment, procedure és macro definíciókból áll. A deklarációs részben akármilyen számú ilyen szakasz, akármilyen sorrendben megjelenhet, egy-egy szakasz többször is.

Ha egy olyan könyvtári modult akarunk írni, amely csak eljárásokat tartalmaz, főprogramot nem, egy HLA unitot használunk. A unitok szintaxisa majdnem azonos a programokéval, csak itt nincs a unithoz tartozó BEGIN, pl.

```
unit TestPgm;

    procedure LibraryRoutine;
      begin LibraryRoutine;
         << stb. >>
     end LibraryRoutine;

end TestPgm;
```

4.11.1 Eljárás deklarációk

Az eljárások deklarációi csaknem megegyeznek a program deklarációival.

```
procedure identifier; noframe;
begin identifier;
   statements
end identifier;
```

Vegyük észre, hogy a HLA nagyon gazdag szintaktikus lehetőségeket kínál. A fenti minta annak a szintaxisnak felel meg, amely leginkább hasonlít a többi assembler által használt eljárás megvalósítási módhoz. A HLA eljárásai lehetővé tesznek paramétereket, helyi változók deklarálását, és sok olyan eszközt, amit a jelen dokumentum nem ír le. Részletekért lásd a HLA Kézikönyvet.

Jegyezzük meg, *és ez nagyon fontos*, hogy a NOFRAME paraméternek jelen kell lennia a PROCEDURE deklarációjában. Enélkül a HLA további kódot ad az eljáráshoz és az minden bizonnyal nem várakozásunknak megfelelően fog működni (valójában, a beszúrt kód a program elhalásához vezet).

Egy példa eljárás:

```
procedure ProcDemo; @noframe;
 begin ProcDemo;

    add( 5, eax );
    ret();

end ProcDemo;
```

4.12 Deklarációk

A programok, unitok, eljárások, módszerek és iterátorok mind rendelkeznek egy deklarációs szakasszal. Az osztályoknak és a névtereknek is van deklarációs szakasza, de az kicsit korlátozott. A deklarációs szakasz egy vagy több komponenst tartalmaz a következők közül (amelyeket más dolgokhoz hasonlóan ez a dokumentum nem tárgyal):

- 'type' szakasz.
- 'const' szakasz.
- 'static' szakasz.
- Egy eljárás.

A szakaszok sorrendje mellékes; feltéve, hogy a program valamennyi azonosítót annak első használata előtt definiálja. Emellett, mint fentebb megjegyeztük, az egyes szekciók többször is előfordulhatnak egyazon deklarációs részben. Például, a két 'const' szakasz a következő eljárásdeklarációban tökéletesen szabályos:

```
program TwoConsts;
const MaxVal := 5;
type    Limits: dword[ MaxVal ];
const MinVal := 0;
begin TwoConsts;



end TwoConsts;
```

4.12.1 A 'type' szakasz

A 'type' szakaszban deklarálhatunk felhasználói adattípusokat. A type szakasz a deklarációs részben jelenik meg és a ***type*** kulcsszóval kezdődik. Mindaddig tart, amíg egy másik deklarációs kulcsszó (pl. ***const***, ***var***, vagy ***val***) vagy a ***begin*** kulcsszó elő nem fordul. Egy tipikus típusdefiníció egy azonosítóval kezdődik, amelyet kettőspont és a típusdefiníció követ. A következő bekezdés néhány szabályos típusdefiníciót mutat be.

```
id1 : id2; // Definiáld id1-et ugyanolyan típusúnak, mint id2.
id1 : id2 [ dim_list ]; // Definiáld id1-et id2 típusú tömbként.
id1 : record     // Definiáld id1-et rekord típusként.
      mező_deklarációk
endrecord;
```

4.12.2 A 'const' szakasz

A HLA program 'CONST' szakaszában deklarálhatunk konstansokat. Tilos a fordítás során később a konstans értékét megváltoztatni. Természetesen futási idő alatt a konstansok static értékkel rendelkeznek.

A konstansdeklarációs szakasz a ***const*** kulcsszóval kezdődik és konstansdefiníciók sorozatával folytatódik. A konstansdeklarációs szakasz a ***const***, ***var, type***, vagy ***val***, stb. kulcsszavak valamelyikének előfordulásáig tart. A konstansdefiníciók a következő bekezdésekben mutatott alakokat vehetik fel.

```
id := expr; // Az expr értékét és típusát rendeli hozzá id-hez
id1 : id2 := expr; // Létrehoz egy id2 típusú, expr értékű id1 konstanst.
```

Megjegyezzük, hogy a HLA számos olyan konstans típust támogat, amelyeket ez a szakasz nem említ meg (például tömb és rekord állandók valamint fordítási-idő változók). A részleteket lásd a HLA Kézikönyvben.

4.12.3 A 'static' szakasz

A ***static*** szakasz lehetővé teszi, hogy statikus változókat deklaráljon, amelyeket aztán a programkód futási időben használhat. A következő bekezdések néhány olyan formát mutatnak, amelyek megengedettek a STATIC szakaszban. Mint általában, lásd a HLA Kézikönyvét számos olyan tulajdonság leírásáért, amelyeket a HLA támogat a STATIC szakaszban.

```
static
   id1 : id2;          // Egy id2 típusú id1 változót deklarál
   id1 : id2 := expr; // Egy id2 típusú id1 változót deklarál és expr értékűre állítja
   id1 : id2[ expr ]; // Egyid2 típusú, expr elemű id1 változót deklarál
```

4.12.3.1 A @NOSTORAGE kulcsszó

A @nostorage kulcsszó hatására a HLA hozzárendeli a szegmensen belüli offset pillanatnyi értékét a változóhoz, de nem foglal tárterületet az objektum számára. Ez a kulcsszó tehát valójában egy álnevet kreál a static szakasz következő objektuma számára. Tekintsük a következő példát:

```
static
    b:      byte; @nostorage;
    w:      word; @nostorage;
    d:      dword;
```

Mivel a b és w változók mindegyike után a @nostorage kulcsszó áll, a HLA nem foglal tárterületet ezen változók számára. A d változó mellett nem áll a @nostorage kulcsszó, így a HLA négy bájtot foglal le a változó számára. A b és w változók, mivel nincs hozzájuk rendelve tárterület, ugyanazon a címen érhetők el, mint a d változó.

4.12.3.2 Az @EXTERNAL kulcsszó

Az external kulcsszó arra ad lehetőséget, hogy más fájlokban deklarált statikus változókra hivatkozzunk. Mint az eljárások external kulcsszava esetén is, két különböző szintaxis fordulhat elő a változó deklarációja után:

```
   varName: varType; @external;
   varName: varType; @external ( "external_Name" );
```

A fenti első forma a változó nevét használja internal és external névként is. A második forma esetén a HLA belső névként használja varName–t a külső modulokban pedig az external_Name nevet rendeli ehhez a változóhoz. Az external kulcsszó mindig a utolsó az adott változóhoz társított kulcsszavak között.

Ha egy külső objektum változó definíciója a forrásfájlban egy external deklaráció után jelenik meg, ezt a HLA úgy értelmezi, hogy ez egy olyan nyilvános változó, amelyet más modulok is használhatnak (az alapértelmezés szerint a változók az adott forrás fájlban lokálisak). Ez az egyetlen módja annak, hogy egy változót publikussá tegyünk, azaz hogy azt más modulok is használhassák. Általában az external deklarációkat egy fejléc (header) fájlba tesszük, amelyet valamennyi (a változót használni kívánó) modul becsatol, és hasonlóan becsatol a változó deklarációját használó forrásfájl is.

4.12.4 Makrók

A HLA makrókifejtési képessége az egyik legfeljlettebb a hasonló fordítóprogramok között. A HLA makrói kulcsszerepet töltenek be a HLA nyelv kiterjesztésekor. Ha gyakran használ makrókat, érdemes megtanulnia a HLA kézikönyből a HLA makró képességeit. Ez a fejezet a HLA korlátozott „Standard Macro" lehetőségeit ismerteti, ami a hasonló assemblerek lehetőségeivel egyenértékű.

Egy program deklarációs részében a következő szintaxissal adhat meg makrókat:

```
#macro azonosító ( opcionális_paraméterlista );
   utasítások
#endmacro;
```

Példa:

```
#macro MyMacro;
   ?i = i + 1;
#endmacro;
```

Az esetleges paraméterlista egy vagy több azonosítóból áll, vesszővel elválasztva. A HLA automatikusan „text" típust rendel a makró paraméterekhez (kivéve egy alább bemutatott speciális esetet). Példa:

```
#macro MacroWParms ( a, b, c );
   ?a = b + c;
#endmacro;
```

Ha egy makró nem használ paramétereket, az azonosítót pontosvessző követi (azaz nincs zárójel vagy paraméterazonosító). Lásd az első példát ebben e fejezetben paraméter nélküli makróként.

Szükségünk lehet olyan szimbólumok definiálására, amelyek lokálisak a makróhívásra nézve (azaz a makró minden egyes hívása egy egyedi szimbólumot eredményez az adott azonosító esetén). A helyi azonosítók listája ezt lehetővé teszi. Helyi azonosítók deklarálásához egyszerűen írjunk a paraméter lista után (a zárójel után, de a pontosvessző előtt) egy kettőspontot és változók vesszővel elválasztott listáját, pl.

```
#macro ThisMacro(parm1) :id1,id2;
...
```

A HLA automatikusan átnevezi az ebben a listában megjelenő szimbólumokat úgy, hogy az új név biztosan egyedi legyen a programban. A HLA az egyedi szimbólumokat "_XXXX_" alakban képzi, ahol XXXX egy hexadecimális érték. Hogy a HLA biztosan egyedi szimbólumokat állíthasson elő, kerüljük az ilyen szimbólumok használatát saját programunkban (általában, az aláhúzással kezdődő és végződő szimbólumok a fordítóprogram és a HLA standard könyvtár számára vannak fenntartva). Például:

```
#macro LocalSym : i,j;

j: cmp(ax, 0)
   jne( i )
   dec( ax )
   jmp( j )
i:
#endmacro;
```

Egy makró hívásához egyszerűen megadjuk annak nevét és a megfelelő paramétereket. Ha csak nem változó számú paramétert írunk elő (tömb szintaxis használatával) a makróban, az aktuális paraméterek számának pontosan meg kell egyeznia a formális paraméterek számával. Ha változó számú paramétert írunk elő a makróban, az aktuális paraméterek száma nem lehet kisebb a formális paraméterek számánál (a tömb paramétert nem számítva).

Az aktuális paraméterek egy karakterfüzérből állnak, amely a határoló vesszőig vagy hátsó zárójelig terjed és az utóbbit már nem tartalmazza, pl.

```
example( v1, x+2*y )
```

"`v1`" az első paraméternek megfelelő szöveg, "`x+2*y`" a második paraméternek megfelelő szöveg. Megjegyezzük, hogy a HLA a kód kifejtésekor lehámozza a bevezető és befejező nem-nyomtatódó és vezérlő karaktereket. Az előbbi példát a HLA a következővé fejti ki:

```
?v1 := x+2*y;
```

Ha (kiegyensúlyozott) zárójelek szerepelnek az aktuális paraméterek listáján, a HLA nem tekinti a belső záró zárójelet a makró paraméter lezárásának. Azaz, teljesen szabályos a következő:

```
example( v1, ((x+2)*y) )
```

Ennek kifejtése:

```
?v1 := ((x+2)*y);
```

4.12.5 Az #Include direktíva

Mint a legtöbb hasonló nyelvnek, a HLA-nak is van forrásnyelvi fájl becsatolására vontakozó direktívája, amely a fordítás során a forrásfájl belsejébe beszúr egy másik fájlt. A HLA #INCLUDE direktívája nagyon hasonló a C/C++ azonos nevű pragmájához, és mindkettőt elsősorban ugyanarra a célra használjuk: könyvtári fejléc fájlokat szúrunk be saját programunkba.

A HLA include direktívájának szintaxisa:

```
#include( szöveg_kifejezés );
```

4.12.6 Az #IncludeOnce direktíva

Ha összetett fejléc fájlokat készítünk, különösen könyvtári fejléc fájlok esetén, szükséges lehet #INCLUDE("file") direktívát beírni más fejléc fájlokba is. Ez általában nem okoz gondot, a HLA (legalább 256 szintig) megengedi fájlok egymásba ágyazását. Azonban, ha csak nem vagyunk rendkívül előrelátók fájljaink szervezésében, könnyen létrehozhatunk egy olyan „beszúrási hurkot", amely abban nyilvánul meg, hogy az egyik fájl beszúr egy másikat, amelyik viszont az elsőt szeretné beszúrni. Ha egy ilyen programot megpróbálunk lefordítani, az a fordításkor egyfajta „végtelen ciklust" eredményez. Ez nyilván nem kívánatos.

E probléma elkerülésére használható az #INCLUDEONCE direktíva. Az előzőhöz képest az a különbség, hogy a HLA nyilvántartja, hogy mely fájlokat dolgozott fel az #INCLUDE és az #INCLUDEONCE direktívák használatával és nem dolgozza fel újból a fejléc fájlt ha azt az #INCLUDEONCE direktívával szúrtuk be.

Amikor a HLA végrehajtja az #INCLUDEONCE direktívát, először összehasonlítja annak szöveg paraméterét az előzőleg feldolgozott #INCLUDE és #INCLUDEONCE direktívák szövegeiből összeállított lista elemeivel. Ha az azonos a lista valamely elemével, a HLA figyelmen kívül hagyja az #INCLUDEONCE direktívát; ha a fejléc file neve nem szerepel a belső listán, akkor a HLA hozzáadja a fájlnevet a listához és beszúrja a fájlt.

4.12.7 Az #asm..#endasm és #emit direktívák

A HLA távolról sem tökéletes. Vannak hiányzó utasításai (néhány szándékosan, néhány lustaságból, néhány tudatlanságból maradt ki). Például, a HLA jelenleg nem támogatja az SSE utasításkészletet. Viszont, két olyan kilépési mechanizmust is biztosít, amelyek lehetővé teszik, hogy bármi olyat megtehessünk, amit a HLA fordítási eredményét feldolgozó assembler program (pl. MASM) megenged.

Az első ilyen mechanizmus az #ASM..#ENDASM szakasz. Az assembly blokk szintaxisa a következő:

```
#asm
   << szöveg amit             >>
   << közvetlenül átadunk     >>
   << a MASM kimenő           >>
   << adatfájlnak.            >>
#endasm
```

Az #ASM és #ENDASM direktívák között megjelenő szöveg teljes egészében és változatlanul beíródik a HLA által készített .ASM fájlba. A MASM (vagy más assembler, amit használ) lesz a felelős ennek a kódnak a fordításáért. Természetesen, az assembly blokkban olyan kódot kell megadnia, amit az assembler képes lefordítani, vagy az assembler fordítási hibát fog jelezni, amikor a kóddal találkozik.

A másik ilyen kilépési mechanizmus az #EMIT direktíva. Ennek formája:

```
#emit( szöveges_kifejezés )
```

A HLA értelmezi a szöveges kifejezést és beírja a szöveges kifejezést a keletkező .ASM fájlba. Az #asm..#endasm blokk és az #emit direktíva részletes leírását és az azokkal kapcsolatos korlátozásokat lásd a HLA kézikönyvben.

4.12.8 A feltételes fordítás utasításai (#if)

A feltételes fordítás utasításai a HLA-ban a következő szintaxissal rendelkeznek:

```
#if( állandó_logikai_kifejezés )

       << Fordítandó utasítások >>
       <<  ha a fenti kifejezés igaz>>

#elseif (állandó_logikai_kifejezés )

    <<  Fordítandó  utasítások  ha  >>
    << a közvetlenül ez előtt álló >>
    <<  kifejezés  igaz  és  az  első  >>
    << kifejezés fent hamis. >>

#else

       << Fordítandó utasítások ha  >>
       << mindkét fenti kifejezés hamis. >>

#endif
```

Az #ELSEIF és #ELSE utasítások opcionálisak. Mint várható, egynél több #ELSEIF is előfordulhat ugyanabban a feltételes direktívában.

Más assemblerektől és magas szintű nyelvektől eltérően, a HLA feltételes direktívái mindenütt előfordulhatnak, ahol nem-nyomtatódó jelek lehetnek. Még az utasítások közepén is! Bár ilyen direktíváknak az utasításokba való ilyen beágyazása nem ajánlott (mivel nehezen olvashatóvá teszi a kódot), jó tudni, hogy ezeket a direktívákat makrókban is el lehet helyezni és azután az utasítás operandusát makróhívással lehet helyettesíteni.

Fontos megjegyezni e direktíváról, hogy az #IF és #ELSEIF direktívákban szereplő konstans kifejezéseknek logikai típusúaknak kell lenniük, különben a HLA hibajelzést ad. Bármely, logikai értéket eredményező konstans kifejezés elfogadható e helyen.

Ne tévesszük szem elől, hogy a feltételes fordítási direktívák fordítási időben és nem futási idő alatt hajtódnak végre. Ezeket a direktívákat nem használhatja döntések meghozatalára amikor a programja már ténylegesen fut.

5 A 80x86 utasításkészlete a HLA-ban

A HLA és a standard 80x86 assembly nyelv között az egyik legnyilvánvalóbb különbség a gépi utasítások szintaxisában van. A két legfontosabb eltérés, hogy a HLA függvény-szerű jelölést használ és hogy a HLA-ban (forrás,cél) sorrendben szerepelnek az operandusok az Intelnél megszokott (cél,forrás) sorrend helyett.

5.1 Operandus nélküli (Nulla operandusú) utasítások

A következő utasításoknak nem szükséges operandus. Az ilyen utasításoknak kétféle szintaxis is megengedett:

```
instr;
instr();
```

A nulla operandusú utasítások mnemonikjai

aaa, aad, aam, aas, cbw, cdq, clc, cld, cli, cmc, cmpsb, cmpsd, cmpsw, cpuid, cwd, cwde, daa, das, insb, insd, insw, into, iret, iretd, lahf, leave, lodsb, lodsd, lodsw, movsb, movsd, movsw, nop, outsb, outsd, outsw, popad, popa, popf, popfd, pusha, pushad, pushf, pushfd, rdtsc, rep.insb, rep.insd, rep.insw, rep.movsb, rep.movsd, rep.movsw, rep.outsb, rep.outsd, rep.outsw, rep.stosb, rep.stosd, rep.stosw, repe.cmpsb, repe.cmpsd, repe.cmpsw, repe.scasb, repe.scasd, repe.scasw, repne.cmpsb, repne.cmpsd, repne.cmpsw, repne.scasb, repne.scasd, repne.scasw, sahf, scasb, scasd, scasw, stc, std, sti, stosb, stosd, stosw, wait, xlat

5.2 Általános aritmetikai és logikai utasítások

Ezek az utasítások az adc, add, and, mov, or, sbb, sub, test, és xor. Ezek mind ugyanazt az alapformát használják (az alábbi példákban helyettesítse az "adc" utasításnevet a megfelelő utasításnévvel):

Alapforma:

adc(forrás, cél);

Megengedett speciális formák:

```
adc( Reg8, Reg8 );
adc( Reg16, Reg16 );
adc( Reg32, Reg32 );

adc( const, Reg8 );
adc( const, Reg16 );
adc( const, Reg32 );

adc( const, mem );

adc( Reg8, mem );
adc( Reg16, mem );
adc( Reg32, mem );

adc( mem, Reg8 );
adc( mem, Reg16 );
adc( mem, Reg32 );

adc ( Reg8, AnonMem );
adc( Reg16, AnonMem );
adc( Reg32, AnonMem );

adc ( AnonMem, Reg8 );
adc( AnonMem, Reg16 );
adc( AnonMem, Reg32 );
```

Figyelem: az "adc(const, mem)" formájú utasítások esetén, ha a memóriahelyhez nincs hozzárendelve méret vagy típus, a memória operandus méretét explicite meg kell határozni, pl. "adc(5,(type byte [eax]));"

5.3 Az XCHG utasítás

Az xchg utasítás a következő szintatikus változatokban létezhet:

Alap alak:

```
xchg( forrás, cél );
```

Speciális alak:

```
xchg( Reg8, Reg8 );
xchg( Reg8, mem );
xchg( Reg8, AnonMem);
xchg( mem, Reg8 );
xchg( AnonMem, Reg8 );

xchg( Reg16, Reg16 );
xchg( Reg16, mem );
xchg( Reg16, AnonMem);
xchg( mem, Reg16 );
xchg( AnonMem, Reg16 );

xchg( Reg32, Reg32 )
xchg( Reg32, mem )
```

```
xchg( Reg32, AnonMem);
xchg( mem, Reg32 );
xchg( AnonMem, Reg32 );
```

5.4 A CMP utasítás

A "cmp" utasítás a következő általános alakot használja:

Alap forma:

```
cmp ( BalOperandus, JobbOperandus );
```

Speciális alakok:

```
cmp( Reg8, Reg8 );
cmp( Reg8, mem );
cmp( Reg8, AnonMem );
cmp( mem, Reg8 );
cmp( AnonMem, Reg8 );
cmp( Reg8, const );

cmp( Reg16, Reg16 );
cmp( Reg16, mem );
cmp( Reg16, AnonMem );
cmp( mem, Reg16 );
cmp( AnonMem, Reg16 );
cmp( Reg16, const );

cmp( Reg32, Reg32 );
cmp( Reg32, mem );
cmp( Reg32, AnonMem );
cmp( mem, Reg32 );
cmp( AnonMem, Reg32 );
cmp( Reg32, const );

cmp( mem, const );
```

Vegyük észre, hogy a CMP utasítás operandusai „cél,forrás” sorrendben vannak, a szokásos „forrás,cél” helyett (azaz az operandusok abban a sorrendben vannak, ahogyan a MASM várja azokat). Ez azért van így, hogy lehetővé tegye az utasítás mnemonic intuitív használatát (azaz, a CMP-t általában úgy olvasssuk, hogy „hasonlítsd a célt a forráshoz”"). Ez a keveredést azzal akarjuk elkerülni, hogy az operandusokat „bal operandusnak” és „jobb operandusnak” nevezzük. A bal és a jobb egyértelműen jelöli az összehasonlító operátor két oldalán az operandusok elhelyezkedését, pl. "<=" (e.g., "left <= right").

A "cmp(mem, const)" formájú utasítások esetén a memória operandusnak típussal vagy mérettel kell rendelkeznie. Ha névtelen memóriára hivatkozunk, a memóriahelyet típuskényszerítéssel kell megadni, pl. "cmp((type word [ebp-4]), 0);".

5.5 A szorzó (Multiply) utasítások

A HLA a 80x86 „MUL” és „IMUL” utasításainak számos változatát támogatja. Ezeknek a formája a következő lehet:
A szokásos szintaxis:

```
mul ( reg8 );
mul(  reg16);
mul( reg32 );
mul ( mem );

mul( reg8, al );
mul( reg16, ax );
mul( reg32, eax );

mul ( mem, al );
mul( mem, ax );
mul ( mem, eax );

mul ( AnonMem, ax );
mul ( AnonMem, dx: ax );
mul ( AnonMem, edx: eax );

imul( reg8 );
imul(  reg16);
imul( reg32 );
imul( mem );

imul( reg8, al );
imul( reg16, ax );
imul( reg32, eax );

imul( mem, al );
imul( mem, ax );
imul( mem, eax );

imul( AnonMem, ax );
imul( AnonMem, dx:ax );
imul( AnonMem, edx:eax );

intmul( const, Reg16 );
intmul( const, Reg16, Reg16 );
intmul( const, mem, Reg16 );
intmul( const, AnonMem, Reg16 );

intmul( const, Reg32 );
intmul( const, Reg32, Reg32 );
intmul( const, mem, Reg32 );
intmul( const, AnonMem, Reg32 );

intmul( Reg16, Reg16 );
intmul( mem, Reg16 );
intmul( AnonMem, Reg16 );

intmul ( Reg32, Reg32 );
intmul ( mem, Reg32 );
intmul ( AnonMem, Reg32 );
```

Kiterjesztett szintaxis:

```
mul( const, al );
mul( const, ax );
mul( const, eax );
```

```
imul( const, al );
imul( const, ax );
imul( const, eax );
```

Az első és legfontosabb tudnivaló a HLA szorzó utasításáról, hogy a HLA különböző utasítás nevet (mnemonik) használ a kiterjesztett pontosságú és az egyszeres pontosságú szorzás esetén (IMUL és INTMUL). A standard MASM mindkét utasításra ugyanazt az utasításnevet használja. Két oka volt a HLA ilyen szintaxis változtatásának. Először, hogy valahogyan meg kell különböztetni a "mul(const, al)" és "intmul(const, al)" utasításokat (hasonlóan az AX és EAX regiszterekre vonatkozó utasításokhoz). A másik, hogy az INTMUL utasítás viselkedése lényegesen eltér az IMUL utasításétól, így van értelme ezekre az utasításokra eltérő utasításnevet használni.

A kiterjesztett szintaxisú utasítások egy static változót hoznak létre, annak a megadott konstanst adják kezdőértékül és ennek a változónak a címét adják meg a MUL vagy IMUL utasítás forrás operandusaként.

5.6 Az osztó (Divide) utasítások

A HLA a 80x86 DIV és IDIV utasításainak számos változatát támogatja. Ezek formái:

Alapforma:

```
div( source );
div( source, dest );

mod( source );
mod( source, dest );

idiv( source );
idiv( source, dest );

imod( source );
imod( source, dest );
```

Speciális formák:

```
div( reg8 );
div( reg16);
div( reg32 );
div( mem );

div( reg8, ax );
div( reg16, dx:ax);
div( reg32, edx:eax );

div( mem, ax );
div( mem, dx:ax);
div( mem, edx:eax );

div ( AnonMem, ax );
div ( AnonMem, dx: ax );
div ( AnonMem, edx:eax );


mod( reg8 );
mod( reg16);
mod( reg32 );
mod( mem );
```

```
mod( reg8, ax );
mod( reg16, dx:ax);
mod( reg32, edx:eax );

mod( mem, ax );
mod( mem, dx:ax);
mod( mem, edx:eax );

mod ( AnonMem, ax );
mod ( AnonMem, dx: ax );
mod( AnonMem, edx:eax );

idiv( reg8 );
idiv(  reg16);
idiv( reg32 );
idiv( mem );

idiv( reg8, ax );
idiv( reg16, dx:ax);
idiv( reg32, edx:eax );

idiv( mem, ax );
idiv( mem, dx:ax);
idiv( mem, edx:eax );

idiv( AnonMem, ax );
idiv( AnonMem, dx:ax );
idiv( AnonMem, edx: eax );

imod( reg8 );
imod(  reg16);
imod( reg32 );
imod ( mem );

imod( reg8, ax );
imod( reg16, dx:ax);
imod( reg32, edx:eax );

imod( mem, ax );
imod( mem, dx:ax);
imod( mem, edx:eax );

imod ( AnonMem, ax );
imod ( AnonMem, dx: ax );
imod( AnonMem, edx:eax );
```

Kiterjesztett szintaxis:

```
div( const, ax );
div( const, dx:ax );
div( const, edx:eax );

mod( const, ax );
mod( const, dx:ax );
mod( const, edx:eax );
```

```
idiv( const, ax );
idiv( const, dx:ax );
idiv( const, edx:eax );

imod( const, ax );
imod( const, dx:ax );
imod( const, edx:eax );
```

A cél operandus a 80x86 "div" és "idiv" utasításaiban mindig alapértelmezett (AX, DX:AX, vagy EDX:EAX). A HLA a program könnyebb olvashatósága érdekében megengedi a céloperandus megadását (bár a cél operandus megadása opcionális).

A HLA osztási utasítás támogat egy olyan kiterjesztett szintaxist, amely megengedi, hogy osztóként (forrás operandusként) konstansot használjunk. A HLA a static adat szegmensben foglal le egy tárterületet, a megadott konstanst használja annak kezdőértékéül, majd az akkumlátort ezzel az újonnan meghatározott memóriacím tartalmával osztja.

5.7 Egyoperandusú aritmetikai és logikai utasítások

Ezek az utasítások a dec, inc, neg, és not. Ezek általános formája (helyettesítsük a megadott mnemonikot a megfelelővel):

Alapforma:

```
dec ( dest );
```

A megengedett speciális formák:

```
dec ( Reg8 );
dec ( Reg16 );
dec ( Reg32 );
dec ( mem );
```

Megjegyzés: ha `mem` típus és méret nélküli memóriahely (azaz névtelen memóriahely), explicit módon meg kell adni annak típusát, pl. "dec((type word [edi]));"

5.8 Biteltoló és -forgató utasítások

Ezek az utasítások a következők: RCL, RCR, ROL, ROR, SAL, SAR, SHL, és SHR. Ezek az utasítások a következő alap szintaxissal rendelkeznek, a megfelelő mnemonik helyettesítése után.

Alap forma:

```
shl ( szám, cél );
```

Speciális forma:

```
shl ( const, Reg8 );
shl ( const, Reg16 );
shl ( const, Reg32 );
```

```
shl( const, mem );

shl(  cl,  Reg8  );
shl(  cl,  Reg16  );
shl( cl, Reg32 );

shl( cl, mem );
```

A "const" operandus egy előjel nélküli egész állandó zérus és a cél operandus bitszáma által meghatározott maximum között. A memória operandust használó operandusoknak típussal vagy mérettel kell rendelkezniük, azaz ha névtelen memóriahelyet használunk, típuskényszerítést kell alkalmaznunk:

shl(2, (type dword [esi]));

5.9 A kétszeres pontosságú biteltoló utasítások

Ezen utasítások általános formája a következő (az alábbiakban SHRD-t is használhat SHLD helyett):

Általános forma:

```
shld( szám, forrás, cél );
```

Speciális forma:

```
shld( const, Reg16, Reg16 );
shld( const, Reg16, mem );
shld( const, Reg16, AnonMem );

shld( cl, Reg16, Reg16 );
shld( cl, Reg16, mem );
shld( cl, Reg16, AnonMem );

shld( const, Reg32, Reg32 );
shld( const, Reg32, mem );
shld( const, Reg32, AnonMem );

shld( cl, Reg32, Reg32 );
shld( cl, Reg32, mem );
shld( cl, Reg32, AnonMem );
```

5.10 A lea utasítás

Ez az utasítás a következő szintaxist használja:

```
lea( Reg32, memory );
lea( Reg32, AnonMem );
lea( Reg32, ProcID );
lea( Reg32, LabelID );
```

Kiterjesztett szintaxis:

```
lea( memory, Reg32 );
lea( AnonMem, Reg32 );
lea( ProcID, Reg32 );
lea( LabelID, Reg32 );
lea( StringConstant, Reg32 );
lea( const ConstExpr, Reg32 );
```

A „lea” utasítás a megadott 32-bites regiszterbe tölti a megadott memória operandus, eljárás vagy utasítás címét. Megjegyezzük, hogy a kiterjesztett szintaxist használva az operandusok formáját megfordíthatja. Mivel pontosan egy operandusnak regiszter típusúnak kell lennie, a kétféle forma között semmi kétség nem merül fel (ez utóbbi szintaxis azok kedvéért szerepel, akik panaszkodtak a (reg,memory) szintaxis miatt). Természetesen az a jó programozói stílus, ha csak az egyik formát használja programjában (vagy reg,memory vagy memory,reg).

Megjegyzés: a HLA nem támogatja azt a LEA utasítást, amelyik 16 bites címet tölt be egy 16 bites regiszterbe. Ez a forma nem igazán használható, amikor egy 32 bites programot futtatunk egy 32 bites operációs rendszeren.

5.11 Az előjeles és nulla módú hosszkiterjesztő utasítások

A HLA MOVSX és MOVZX utasítások szintaxisa a következő:

Általános alak:

movsx(source, dest); movzx(source, dest);

Speciális alakok:

```
movsx ( Reg8, Reg16 );
movsx ( Reg8, Reg32 );
movsx( Reg16, Reg32 );
movsx ( mem8, Reg16 );
movsx ( mem8, Reg32 );
movsx( mem16, Reg32 );

movzx ( Reg8, Reg16 );
movzx ( Reg8, Reg32 );
movzx( Reg16, Reg32 );
movzx ( mem8, Reg16 );
movzx ( mem8, Reg32 );
movzx( mem16, Reg32 );
```

Ezek az utasítások előjellel (MOVSX) vagy nullával (MOVZX) kiterjesztve másolják a forrás operandust a cél operandusba.

5.12 A Push és Pop (veremkezelő) utasítások

Ezen utasítások általános formája a következő:

```
pop( reg16 );
pop( reg32 );
pop( mem );
push( Reg16 );
push( Reg32);
```

```
push( memory );

pushw( Reg16 );
pushw( memory );
pushw( AnonMem );
pushw( Const );

pushd( Reg32 );
pushd( memory );
pushd( AnonMem );
pushd( Const );
```

Ezek az utasítások a veremtárolóba teszik vagy onnét előveszik a meghatározott operandust.

5.13 Eljárás hívások

Ha adott egy "MyProc" nevű eljárás vagy az eljárás címét tartalmazó DWORD változó, az eljárást a következő paranccsal lehet hívni:

```
call( MyProc );
```

A HLA ténylegesen számos más szintaxisú eljáráshívást is támogat, beleértve olyanokat is, amelyek automatikusan paramétereket löknek a veremtárolóba. A részleteket lásd a HLA Kézikönyvében.

5.14 A ret utasítás

A RET( ) utasítás kétféle szintaktikus formát enged meg:

```
ret( );
ret ( egész_konstans_kifejezés );
```

Az első forma egyszerűen a 80x86 RET utasítást adja ki, a második pedig a 80x86 RET utasítását a (veremtárolóban őrzött paraméterek eltávolítására használt) numerikus kostans kifejezés értékével.

5.15 Az ugró (jmp) utasítások

A HLA "jmp" utasítások a következő szintaxist követik:

```
jmp    Label;
jmp    ProcedureName;
jmp( dwordMemPtr );
jmp( anonMemPtr );
jmp( reg32 );
```

A `"Label"` a folyó eljárás egy utasítás címkéjét jelenti. (A mostani verzióban nem megengedett egy másik eljárásban levő címkére ugrani. A következő verziókban ez a korlátozás enyhülhet.) Az utasításcímke egyedi azonosító (az adott eljáráson belül), amelyet a kettőspont követ, pl.

```
VégtelenCiklus:
   << A végtelen cikluson belüli kód>>
   jmp VégtelenCiklus;
```

Egy eljárásra ugrani annyit jelent, mint a meghatározott eljárás első utasítására adni át a vezérlést. A programozó felelőssége, hogy a szükséges paramétereket és a visszatérési címet a veremtárolóba tegye.

Ezek az utasítások egy üres sttringet adnak meg „returns” értékként.

5.16 A feltételes ugró utasítások

Ezek az utasítások: JA, JAE, JB, JBE, JC, JE, JG, JGE, JL, JLE, JO, JP, JPE, JPO, JS, JZ, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ. Valamenyi utasítás a következő általános formát követi (a „JA”-t a megfelelő paranccsal helyettesítve).

```
ja      HelyiCímke;
```

"*HelyiCímke*" a folyó eljárásban definált utasításcímke

Megjegyzés: a HLA fordítási folyamat sajátságai miatt kerülje a JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, és LOOPNZ utasítások használatát. A többi feltételes ugró utasítástól eltérően, ezen utasítások relatív címtartománya a nagyon korlátozott +/- 128 tartomány. Sajnálatos módon, a HLA nem veszi észre, ha a relatív cím ezen a tartományon kívül esik (ezt a feladatot a MASM látja el), azaz ha egy ilyen címtartomány-hiba lép fel, a HLA nem tud erre figyelmeztetni. A MASM fordítás nem fog sikerülni, és nehéz kitalálni, mi is a hiba oka. Szerencsére ezek az utasítások könnyen és általában hatékonyabban megvalósíthatók más 80x86 utasításokkal, ezért ez nem igazán probléma.

5.17 A feltételes Set utasítások

Ilyen utasítások a következők: SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETO, SETP, SETPE, SETPO, SETS, SETZ, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, and SETNZ. Ezek ilyen alapformát vehetnek fel (a megfelelő mnemonikot helyettesítve seta helyére):

```
seta( Reg8 );
seta( mem );
seta( AnonMem );
```

5.18 A feltételes move utasítások

Ilyen utasítások a következők: CMOVA, CMOVAE, CMOVB, CMOVBE, CMOVC, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVO, CMOVP, CMOVPE, CMOVPO, CMOVS, CMOVZ, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, and CMOVNZ. Ezek a következő általános szintaxist követik:

```
CMOVcc ( src, dest );
```

A lehetséges operátorok:

```
CMOVcc( reg16, reg16 );
CMOVcc( reg32, reg32 );
CMOVcc( mem16, reg16 );
CMOVcc( mem32, reg32 );
```

Ezek az utasítások csak akkor mozgatják az adatokat, ha (a *cc* feltétellel) meghatározott feltétel igaz. Ha a feltétel hamis, ezen utasítások üres műveletként (no-operation) viselkednek.

5.19 Adat be- és kiviteli utasítások

Az "in" és "out" utasítások szintaxisa a következő:

```
in( port, al );
in( port, ax );
in( port, eax );

in( dx, al );
in( dx, ax );
in( dx, eax );

out( al, port );
out( ax, port );
out( eax, port );

out( al, dx );
out( ax, dx );
out( eax, dx );
```

A "port" paraméternek a 0..255 tartományba eső előjel nélküli egésznek kell lennie Megjegyezzük, hogy ezek az utasítások az operációs rendszernek vannak fenntartva ha a Win32 operációs rendszer alatt dolgozunk. Emiatt bizonyos portok használata vagy a portok bizonyos körülmények között való használata jogosultsági hibát eredményezhet.

5.20 A megszakítás (int) utasítás

Eme utasítás szintaxisa "int(konstans)" ahol a konstans operandus 0..255 tartományba eső előjel nélküli egész.

Ez az utasítás üres sztringet ad vissza „returns” értékként.

Lásd „Az assembly művészete” (DOS verzió) hatodik fejezetét az utasítás további tárgyalásáért. Megjegyezzük azonban, hogy Win32 alatt általában nem csinálunk OS vagy BIOS hívásokat. Az "int $80" paranccsal lehet Linux alatt nagyon alacsony szintű hívásokat csinálni.

5.21 A bound utasítás

Az utasításnak a következő formái lehetnek:

```
bound ( Reg16, mem );
bound ( Reg16, AnonMem );

bound ( Reg32, mem );
bound ( Reg32, AnonMem );
```

A kiterjesztett szintaxisú forma:

```
bound( Reg16, constL, constH );
bound( Reg32, ConstL, ConstH );
```

A kiterjesztett szintaxisú forma két állandót tesz a statikus adatszegmensbe és az első konstans címével (`ConstL`) helyettesíti a memória operandust.

5.22 Az enter utasítás

Az ENTER utasítás szintaxisa: "enter(const, const);". Az első konstans operandus az eljárás lokális változói összes bájtjának száma, a második operandus pedig az eljárás lex szintje. Általános szabály, hogy ha lehet ne használja ezt az utasítást (meg az ennek megfelelő LEAVE utasítást). A HLA eljárásai automatikusan létrehozzák a display és aktivációs rekordot (hatékonyabban, mintha az ENTER-t használná). Lásd a HLA kézikönyvet további részletekért az eljárás aktivációs rekordjainak felépítéséről.

5.23 A CMPXCHG utasítás

Ez az utasítás a következő szintaxist használja:

Alap forma:

```
cmpxchg( reg/mem, reg );
```

Speciális forma:

```
cmpxchg(    Reg8, Reg8 );
cmpxchg(    Reg8, Memory );
cmpxchg(    Reg8, AnonMem );

cmpxchg(    Reg16,   Reg16 );
cmpxchg(    Reg16,   Memory );
cmpxchg(    Reg16,   AnonMem);

cmpxchg(    Reg32,   Reg32 );
cmpxchg(    Reg32,   Memory );
cmpxchg(    Reg32,   AnonMem);
```

Megjegyzés: a HLA jelenleg nem támogatja a Pentium cmpxchg8b utasítását, bár egyszerűen létre lehet hozni egy ezt az utasítást implementáló makrót. Az #ASM..#ENDASM vagy #EMIT utasítást is használhatja a CMPXCHG8B utasítás implementálására.

5.24 Az XADD utasítás

Az XADD utasítás a következő szintaxist használja:

Alap forma:

```
xadd( forrás, cél );
```

Speciális forma:

```
xadd( Reg8, Reg8 );
xadd( mem, Reg8 );
xadd( AnonMem, Reg8 );

xadd( Reg16, Reg16 );
xadd( mem, Reg16 );
xadd( AnonMem, Reg16 );

xadd( Reg32, Reg32 );
xadd( mem, Reg32 );
xadd( AnonMem, Reg32 );
```

5.25 A BSF és BSR utasítások

Ezek a bitpásztázó utasítások a következő szintaxist használják (helyettesítse BSR-t BSF-fel, ha szükséges):

Alap forma:

```
bsr( forrás, cél );
```

A megengedett speciális formák:

```
Bsf ( Reg16, Reg16 );
bsf ( mem, Reg16 );
bsf ( AnonMem, Reg16 );

bsf ( Reg32, Reg32 );
bsf ( mem, Reg32 );
bsf ( AnonMem, Reg32 );
```

5.26 A BSWAP utasítás

Ez az utasítás "bswap(reg32)" formájú. A megadott 32-bites regiszter tartalmát alakítja át a „little endian” és a „big endian” adatformátumok között.

5.27 „Bit Test”utasítások

Ez az utasításcsoport a BT, BTC, BTR, és BTS utasításokból áll. Ezek a következő alap formában fordulhatnak elő:

Alap forma:

```
bt ( Bitszám, cél );
```

Speciális forma:

```
bt( const, Reg16 );
bt( const, Reg32 );

bt( const, mem );

bt( Reg16, Reg16 );
bt( Reg16, mem );
bt ( Reg16, AnonMem );

bt( Reg32, Reg32 );
bt( Reg32, mem );
bt ( Reg32, AnonMem );
```

Helyettesítsük a BTC, BTR, vagy BTS mnemonikok valamelyikét a BT helyére a fenti példákban.

5.28 Lebegőpontos utasítások

A HLA a következő FPU utasításokat támogatja.

```
fld( Fpreg );
fst ( Fpreg );

fld( Fpmem );    // Returns operand.
fst( Fpmem );    // 32 and 64-bits only! Returns operand.
fstp( Fpmem );   // Returns operand.

fxch( Fpreg );

fild( Fpmem ) ;         // Returns operand.
fist( Fpmem );          // 32 and 64-bits only! Returns operand.
fistp( Fpmem );         // Returns operand.

fbld( Fpmem );          // Returns operand.
fbstp ( Fpmem );        // Returns operand.

fadd( );
fadd(Fpreg, st0);
fadd(st0, Fpreg);
fadd(Fpmem);            // Returns operand.
fadd(Fpconst);          // Returns operand.

faddp( );
faddp( st0, Fpreg );
```

```
fmul( );
fmul ( Fpreg, st0 );
fmul( st0, Fpreg );
fmul ( Fpmem );       // Returns operand.
fmul ( Fpconst );     // Returns operand.

fmulp( );
fmulp( st0, Fpreg );

fsub( );
fsub( Fpreg, st0 );
fsub( st0, Fpreg );
fsub( Fpmem );      // Returns operand.
Fsub( Fpconst );    // Returns operand.

Fsubp( );
fsubp( st0, Fpreg );


fsubr( );
fsubr( Fpreg, st0 );
fsubr( st0, Fpreg );
fsubr( Fpmem );       // Returns operand.
fsubr( Fpconst );     // Returns operand.

fsubrp( );
fsubrp( st0, Fpreg );

fdiv( );
fdiv( Fpreg, st0 );
fdiv( st0, Fpreg );
fdiv( Fpmem );        // Returns operand.
fdiv( Fpconst );      // Returns operand.

fdivp( );
fdivp( st0, Fpreg );


Fdivr( );
fdivr( Fpreg, st0 );
fdivr( st0, Fpreg );
fdivr ( Fpmem );       // Returns operand.
Fdivr( Fpconst );      // Returns operand.

Fdivrp( );
fdivrp( st0, Fpreg );

fiadd( mem16 );        // Returns operand.
Fiadd( mem32 );        // Returns operand.
Fiadd( const );        // Returns operand.

Fimul ( mem16 );       // Returns operand.
Fimul ( mem32 );       // Returns operand.
Fimul ( const );       // Returns operand.

Fidiv( mem16 );        // Returns operand.
Fidiv( mem32 );        // Returns operand.
Fidiv( mem32 );        // Returns operand.
Fidiv( const );        // Returns operand.

Fidivr( mem16 );       // Returns operand.
Fidivr( mem32 );       // Returns operand.
Fidivr( const );       // Returns operand.
```

```
Fcom( );
fcom( Fpreg );
fccom( Fpmem );         // Returns operand.

Fcomp( );
fcomp ( Fpreg );
fcomp ( Fpmem );        // Returns operand.

Fucom( );
fucom( Fpreg );

fucomp( );
fucomp ( Fpreg );

fcompp();
fucompp();


ficom(  mem16  );       //  Returns  operand.
ficom(  mem32  );       //  Returns  operand.
ficom( const );         // Returns operand.


ficomp(  mem16  );      //  Returns  operand.
ficomp(  mem32  );      //  Returns  operand.
ficomp( const );        // Returns operand.

fsqrt();            // The following all return "st0"
fscale();
fprem();
fprem1();
frndint();
fxtract();
fabs();
fchs();
ftst();
fxam();
fldz();
fld1();
fldpi();
fldl2t();
fldl2e();
fldlg2();
fldln2();
f2xm1();
fsin();
fcos();
fsincos();
fptan();
fpatan();
fyl2x();
fyl2xp1();



finit();            // Returns ""
fwait();
fclex();
fincstp();
fdecstp();
fnop();
ffree( Fpreg );
```

```
fldcw(  mem  );
fstcw(  mem  );
fstsw( mem );
```

Lásd „Az assembly programozás művészete” megfelelő fejezetét ezen utasítások leírásáért. Megjegyezzük, hogy a HLA nem támogatja a teljes FPU utasításkészletet. Ha valóban szüksége van a kimaradt néhány utasításra, használja az #ASM..#ENDASM vagy #EMIT direktívákat az utasítások kiadására.

5.29 További lebegőpontos utasítások a Pentium Pro és későbbi processzorokban

Az FCMOVcc utasítások (cc= a, ae, b, be, na, nae, nb, nbe, e, ne, u, nu) a következő alap szintaxist használják:

```
FCMOVcc( stn, st0); // n=0..7
```

Ezek a meghatározott lebegőpontos regisztert ST0-ba mozgatják, ha a megadott feltétel

teljesül. Az FCOMI és FCOMIP utasítások a következő szintaxist használják:

```
fcomi( st0, stn );
fcomip( st0, stn );
```

Ezek az utasítások úgy viselkednek, mint (a szintaktikusan egyenértékű) FCOM és FCOMP; kivéve, hogy az állapotot közvetlenül az EFLAG regiszterbe mentik el, nem pedig a lebegőpontos állapotregiszterbe.

5.30 Az MMX utasítások

A HLA támogatja a következő MMX utasításokat, amelyek a Pentiumban és a későbbi processzorokban elérhetők (megjegyezzük, hogy bizonyos utasítások csak a Pentium III és későbbi processzorokon elérhetők; lásd az Intel kézikönyveit a részletekért):

A HLA az mm0, mm1, ..., mm7 szimbólumokkal jelöli az MMX regiszterkészletét.

A következő MMX utasítások ugyanazt a szintaxist követik, amely

```
mmxInstr( mmxReg, mmxReg );
mmxInstr( mem64, mmxReg );
```

mmxInstrs:

```
paddb;
paddw;
paddd;
paddsb;
paddsw;
paddusb;
paddusw;

psubb;
psubw;
psubd;
psubsb;
psubsw;
psubusb;
psubusw;
```

```
pmulhuw;
pmulhw;
pmullw;
pmaddwd;


pavgb;
pavgw;


pcmpeqb;
pcmpeqw;
pcmpeqd;
pcmpgtb;
pcmpgtw;
pcmpgtd;


packsswb;
packuswb;
packssdw;


punpcklbw;
punpcklwd;
punpckldq;
punpckhbw;
punpckhwd;
punpckhdq;


pand;
pandn;
por;
pxor;


psllw;
pslld;
psllq;
psrlw;
psrld;
psrlq;
psraw;
psrad;


pmaxsw;
pmaxub;

pminsw;
pminub;


psadbw;
```

A következő MMX utasítások különleges szintaxist követelnek meg, ami az egyes utasításokra a következő:

```
pextrw(  constant,  mmxReg,  Reg32  );
pinsrw(  constant,  Reg32,  mmxReg  );
pmovmskb ( mmxReg, Reg32 );
pshufw( constant, mmxReg, mmxReg );
pshufw( constant, mem64, mmxReg );


movd (  mem32,  mmxReg  );
movd (  mmxReg,  mem32  );
movq (  mem64,  mmxReg  );
movq ( mmxReg, mem64 );
```

```
emms ();
```

Lásd a megfelelő Intel dokumentációt vagy „Az assembly programozás művészete” könyvet ezen utasítások viselkedésének leírásáért.

6 A HLA memóricímzési módjai

A HLA az Intel 80x86 utasításkészletének valamennyi 32-bites címzési módját támogatja[2]. A 80x86 memóriacíme három lehetséges összetevőből állhat: egy eltolásból (amit offsetnek is neveznek), egy alap mutatóból és egy skálázott indexből. A fentiekből a következő kombinációk lehetségesek:

```
Displacement
basePointer
displacement + basePointer
displacement + scaledIndex
basePointer + scaledIndex
displacement + basePointer + scaledIndex
```

Megjegyezzük, hogy a skálázott index érték nem létezhet önmagában.

A memória címzési módok szintaxisa a HLA-ban a következő formák valamelyike lehet:

```
staticVarName


staticVarName [ constant ]

staticVarName[     breg32     ]
staticVarName[     ireg32     ]
staticVarName[ ireg32*index ]


staticVarName[   breg32   +   ireg32   ]
staticVarName[ breg32 + ireg32*index ]

staticVarName[ breg32 + constant ]
staticVarName[ ireg32 + constant ]

staticVarName[ ireg32*index + constant ]

staticVarName[  breg32 + ireg32 + constant ] breg32
staticVarName[  + ireg32*index + constant ]

staticVarName[  breg32 - constant ] ireg32
staticVarName[  - constant ] ireg32*index
staticVarName[  - constant ]
staticVarName[  breg32 + ireg32 - constant ] breg32
staticVarName[  + ireg32*index - constant ]


[ breg32 ]
```

2. Nem támogatja a 16-bites címzési módokat, mivel azok nem igazán használhatók Win32 alatt.

```
[ breg32 + ireg32 ]
[ breg32 + ireg32*index ]

[ breg32 + constant ]

[ breg32 + ireg32 + constant ]
[ breg32 + ireg32*index + constant ]

[ breg32 - constant ]

[ breg32 + ireg32 - constant ]
[ breg32 + ireg32*index - constant ]
```

"staticVarName" pillanatnyilag elérhető (lokális vagy globális) statikus változót jelöl

"basereg" egy általános 32-bites regisztert jelöl.

"$breg_{32}$" egy alap regisztert jelöl ami bármelyik általános 32-bites regiszter lehet.

"$ireg_{32}$" egy index regisztert jelöl és lehet bármelyik általános célú regiszter, akár a címkifejezésben szereplő alap regiszter is.

"index" az "1", "2", "4", vagy "8" lehetséges konstansok valamelyikét jelöli. Azokban a kifejezésekben, amelyekben az index regiszter index konstans nélkül szerepel, "*1" az alapértelmezett index.

Azokat a memóriacímzési módokat, amelyekben nem szerepel egy azt megelőző változónév, „névtelen memóriahelynek” nevezzük. A névtelen memóriahelyeknek nincs típusa és sok esetben a HLA típuskényszerítő operátorát kell alkalmaznunk, hogy a HLA elfogadja a kifejezést.

Azok a memóriacímzési módok, amelyekhez tartozik egy változónév, öröklik a változó típusát. Olvassa el a következő szakaszt a HLA adattípusairól.

A HLA egy másik módot is lehetővé tesz arra, hogy a különféle címzési mód komponenseket egy címkifejezésben használjuk – a komponeseket külön négyszög zárójelekbe tehetjük és ezeket egymás mellé írhatjuk. A következő példák a standard és a másik lehetséges szintaxist mutatják:

```
[ebx+2]         [ebx] [2]
[ebx+ecx*4+8]   [ebx] [ecx] [8]
lbl[ebp-2]      lbl[ebp] [-2]
```

A kitejesztett szintaxis használatának az oka, hogy ezen a címzési módok darabkáit makrón belül is lehet használni és sokkal könnyebb a már zárójelek közé zárt két operandust egymással összekombinálni mint ezeket a standard címkifejezésben használni.

7 Típuskényszerítés a HLA-ban

Bár egy assembly nyelv soha nem lehet egy valóban erősen típusellenőrző nyelv, a HLA a többi assembly nyelvnél sokkal szigorúbban ellenőrzi a típusokat.

A szigorú típusellenőrzés egy assembly nyelvben igen zavaró lehet. Ezért a HLA bizonyos feltevéseket használ, hogy megelőzze a típusellenőrzési rendszer és a tipikus assembly nyelvű programozó konfliktusát. Egy 80x86 gépi utasításban az egyetlen tényleges típusellenőrzés annak a megkövetelése, hogy az operandusok mérete megfelelő legyen.

Annak ellenére, hogy a HLA gyorsan és lazán bánik a gépi utasításokkal, sok esetben típuskényszerítést kell alkalmazni az operátorokra. A HLA a következő szintaxist használja a memóriahely vagy regiszter operandus típusának kényszerítésére:

```
(type typeID memOrRegOperand)
```

Két olyan eset van, amikor a típuskényszerítés különösen fontos: (1) amikor egy olyan típust akarunk hozzárendelni egy regiszterhez, amely nem bájt, szó vagy dupla szó[3]; (2) amikor egy névtelen memóriához kell típust rendelni.

3. Valószínűleg a legtipikusabb egy regisztert előjeles számként használni a HLA magas szintű nyelvű utasításaiban. Lásd a HLA magas szintű utasításait a részletekért.