

Tartalomjegyzék

Bevezetés.....	2
Felhasználói dokumentáció.....	3
A program célja.....	3
A felhasznált módszerek.....	3
Futtatási követelmények.....	3
Üzembe helyezés.....	3
A program kezelése.....	4
Felhasználói felület.....	4
A bemeneti fájlok szintaxisa.....	6
A program futása során fellépő hibaüzenetek.....	14
Fejlesztői dokumentáció.....	20
A program feladata.....	20
Megkötések.....	20
A felhasznált módszerek.....	21
Osztálydiagram.....	21
App.....	22
Az egyes osztályok diagramjai, leírásai.....	22
namespace Utils.....	22
Allapot.....	23
elsoparseParser.....	27
interpretParser.....	30
flagDisplay.....	32
regDisplay.....	33
varDisplay.....	35
mainDisplay.....	36
Tesztelés.....	39
namespace Utils.....	39
Allapot.....	39
elsoparseParser.....	41
interpretParser.....	41
A grafikus felület rész-osztályai.....	42
A teljes program.....	42
Továbbfejlesztési lehetőségek.....	43
A CD-lemez tartalma.....	44

Bevezetés

Az assembly programnyelvek egyszerű, processzor-közelí utasításokból épülnek fel, amiket a számítógép hatékonyan végre tud hajtani, viszont ennek a hatékonyságnak ára van: a program kódja már rövidebb programok esetén is nehezen olvashatóvá és értelmezhetővé válik.

Ennek a programnak a feladata az, hogy egy assembly programkód futását szimulálja: az utasításokat egyenként hajtja végre és minden lépés után a felhasználó számára kijelzi az aktuálisan tárolt adatokat: a regiszterek tartalmát, a változók tömbjét és a futásidejű vermet. Elsődlegesen azok számára készült, akik szeretnék jobban átlátni egy ilyen programnak a menetét, és megérteni, hogy mi is zajlik a háttérben.

Felhasználói dokumentáció

A program célja

A program beolvas egy egy NASM assembly utasításokból álló fájlt, annak a programnak a futását szimulálja, azaz a kódot lépésenként hajtja végre és minden lépés után kijelzi a regiszterekben, a változók tömbjében és a veremben eltárolt értékeket.

Elsősorban olyan felhasználók számára készült, akik rendelkeznek alapvető programozási ismeretekkel, és nagyjából tisztában vannak az assembly nyelvek felépítésével és működésével.

A felhasznált módszerek

A program C++ nyelven íródott, a grafikus felület wxWidgets-szel készült, az assembly kód kezdeti elemzése, valamint menet közbeni értelmezése flex által generált lexikális elemzőt, illetve bisonc++ által generált szintaktikus és szemantikus elemzőt használ.

Futtatási követelmények

64-bites Microsoft Windows7 operációs rendszer.

Az operációs rendszer és az egyéb programokon felül legalább 200 MB RAM, nagyobb programok (1000-nél több utasítás) szimulálása esetén ennél többre is szükség lehet.

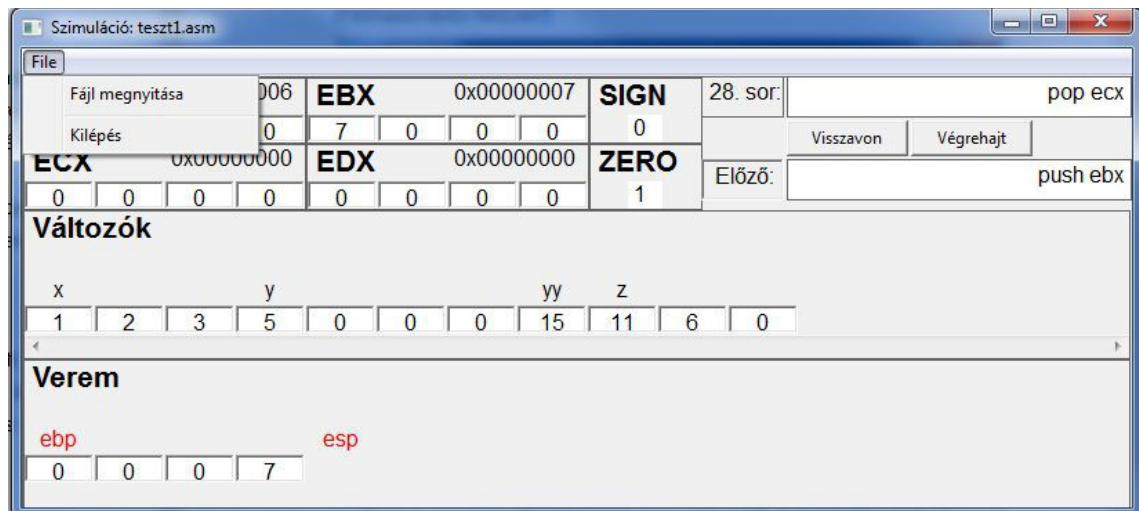
Üzembe helyezés

A CD-lemezen levő „futtathato” nevű mappában található a „szimulacio.exe” állomány, annak az elindításával lehet működésbe hozni a programot. Fontos, hogy a mappájában mellette benne legyenek a libgcc_s_seh-1.dll, a libgib-2.dll, libjpeg-8.dll, liblzma-5.dll, libpng16-16.dll, libtiff-5.dll, wxbase30u_gcc_custom.dll, wxmsw30u_core_gcc_custom.dll és zlib1.dll állományok, ugyanis ezek nélkül a program nem indul el.

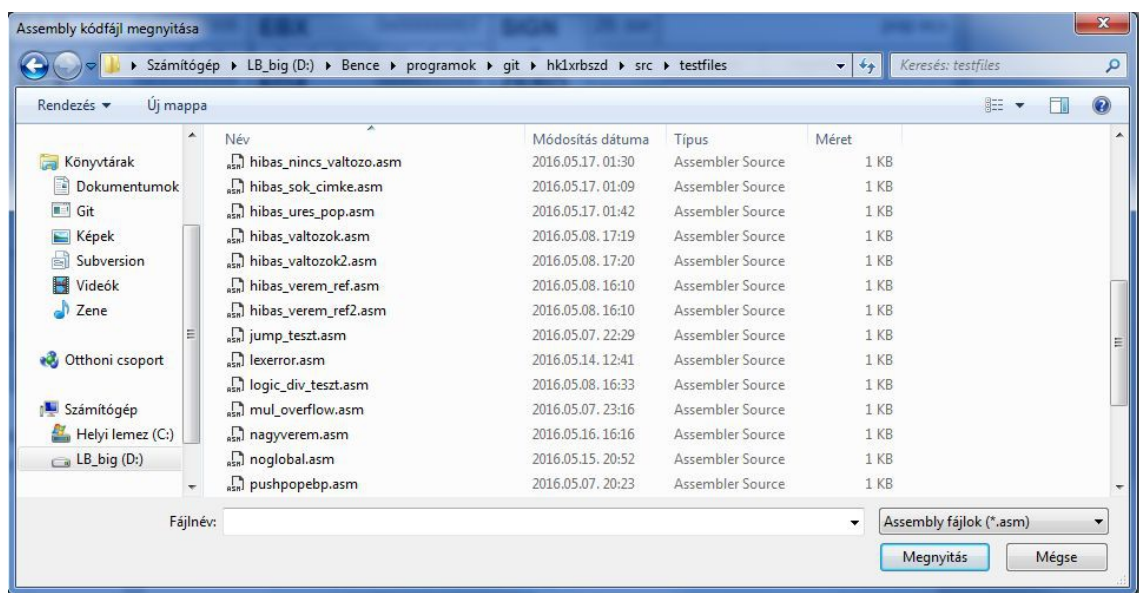
A program kezelése

Felhasználói felület

A program elindítása a felhasználói felület mellett elindít még egy parancssori ablakot. A program futása során nem ajánlott ezt az ablakot bezárni, ugyanis ez a szimulációból is kilép.



A program elindítása után a bal felső sarokban lévő „File” feliratú menüben a „Fájl megnyitása” opcióval lehet betölteni egy assembly kódfájlt.



A fájlböngésző a szokásos módon navigálható, dupla kattintással lehet kiválasztani egy fájlt vagy belépni egy mappába, a felső sávban a mappák nevére kattintva lehet feljebb lépni. Ez a böngésző csak a .asm kiterjesztésű

fájlokat mutatja, más formátumú fájl betöltése ellenjavallt, de alul, középen meg lehet adni ilyen fájlok nevét is. Ebből a menüből a jobb alsó sarokban található „Mégse” gombbal, illetve a jobb felső sarokban levő X-szel lehet kilépni, ha mégsem akarunk új fájlt betölteni.

The screenshot shows a window titled "Szimuláció: teszt2.asm". It contains a "File" menu and a table of CPU registers. The registers are arranged in two columns: EAX, ECX, and EDI on the left; EBX, EDX, and EBP on the right. Each register has a 4-digit hexadecimal value. Below the registers is a section for "Változók" (Variables) with a table for x, y, yy, and z. Below that is a section for "Verem" (Stack) with a table for ebp and esp. The window also has buttons for "Visszavon" (Undo) and "Végrehajt" (Execute), and a "31. sor:" (Line 31) label.

Register	Value
EAX	0x00000006
EBX	0x00000000
ECX	0x00000000
EDX	0x00000000
EDI	0x00000000
EBP	0x00000006

x	y	yy	z
1	2	3	5
0	0	0	15

ebp	esp
0	0
0	6

Miután egy fájlt betöltöttünk, a jobb felső téglalapban található „Végrehajt” gombbal lehet a gombok fölött látható dobozban írt műveletet végrehajtani (tőle balra olvasható, hogy az eredeti assembly programban hanyadik sorban van), a „Visszavon” gombbal pedig az utolsó végrehajtott műveletet visszavonni. A gombok alatti szövegdobozban olvasható az utolsónak végrehajtott utasítás. Az utolsónak használt gomb ki van emelve vékony kerettel, a kiemelt gombot a szóköz billentyű lenyomásával is lehet működtetni.

A műveletek végrehajtásakor a megváltozó értékek pirossal vannak írva fekete helyett a következő művelete végrehajtásáig.

Az éppen szimulált fájl neve az ablak fejlécében olvasható. A bal felső sarokban található a négy regiszter tartalma. A nagybetűs szöveg a regiszter neve, alatta látható a regiszterekben tárolt byteok, balról jobbra helyiérték szerint növekvően, a szövegtől jobbra pedig látható a regiszterben tárolt teljes érték hexadecimálisan (tizenhatos számrendszerben) ábrázolva – ebben a szokásos jelölésmóddal vannak a helyiértékek, két egymást követő karakter értéke felel meg a másik kijelzőn egy bytenak.

A regiszterektől jobbra láthatóak a flagek, ezek egyes vagy nullás értéket vehetnek fel.

Ezek alatt láthatók a változók tömbje, illetve a verem tömbje. Amennyiben ezek a tömbök elég hosszúak, megjelenik alattuk egy gördítősáv, amivel végig lehet haladni rajtuk. A tárolt számok fölött kiegészítő szövegek lehetnek, a változók esetén ezek jelölik az adott nevű változó első byteját, a verem esetén pedig az *esp* és az *ebp* mutatósi helyét (attól egy mezővel jobbra helyezkednek el). Ha ez a két pointer egybe esik, akkor csak az *esp* van kiírva.

Ha a szimuláció a végére ér vagy hibára fut, felugró ablak tájékoztat az eseményről, ekkor a „Végrehajt” gomb szürke lesz és visszalépésig vagy másik fájl betöltéséig nem kattintható.

Lehet szimuláció közben másik fájlt betölteni, viszont ekkor a félbehagyott szimuláció eddigi eredménye elveszik.

A szimulációból kilépni a jobb felső sarokban található X gombbal, illetve a „File” menüben található „Kilépés” opcióval lehet.

A bemeneti fájlok szintaxisa

A program az x86 NASM assembly szintaxisát használja fel alapvetően, viszont az utasításkészlet korlátozott, és bizonyos pontokon vannak engedmények. Hogyha a bemenő fájl szintaxisa nem felel meg a leírtaknak, a program a kezdeti elemzés során hibát jelez.

Az alábbi leírásban használt jelölések:

- **kulcsszó:** a kulcsszó ugyanezzel az írásmóddal szerepelhet (általában kis-nagybetű nem számít)
- **< ... >:** a pontok helyén szereplő kifejezés értelemszerű behelyettesítése
 - pl. <kettőspont> egy darab : karaktert jelöl
 - pl. <azonosító> a program által azonosítóként elfogadott karaktersorozatok jelöli
 - egy azonosító betűvel, aláhúzással vagy ponttal kezdődik, ezt tetszőleges számú (akár 0) betű, szám, aláhúzás vagy pont követi
 - az azonosító nem egyezhet meg a regiszterek nevével, kis-nagy-betű eltérés nem elég

- < ... | ... | ... >: a függőleges vonalakkal elválasztott szavak közül pontosan egy szerepel
 - pl. <resb|resw|resd> a **resb**, **resw** és **resd** szavak közül pontosan egyre illeszkedik

A regiszterek a következők:

- eax, ax, al, ah; ebx, bx, bl, bh; ecx, cx, cl, ch; edx, dx, dl, dh
 - az a, b, c vagy d betűkben megegyező regiszterek egyben vannak tárolva, egymással tartalmazásos kapcsolatban vannak
 - pl. az „eax” regiszter 4 byteos, az „ax” regiszter a két alacsony helyiértékű byteja
 - pl. az „bx” regiszter alacsony helyiértékű byteja „bl”, a nagyobb helyiértékű byteja „bh”
- esp, sp; ebp, bp
 - a fentiekhez hasonlóan itt is tartalmazásos kapcsolat van esp-sp, illetve ebp-bp között
 - „esp” regiszter tartalma határozza meg, hogy a futásidejű veremben hány byte található
 - ha az „esp” regiszterben tárolt érték negatív lenne, a program hibát jelez
 - ezek a regiszterek hivatkozások esetén kitüntetett szerepben vannak
 - ha közülük egy szerepel, akkor a hivatkozás veremhivatkozás, különben a változók tömbjére hivatkozás
 - pl. „word [esp]” a verem tetejéről kezdve 2 byte

Lehet a bemeneti fájlban pontosvessző (;), ekkor annak a sornak a maradékát a program megjegyzésként kezeli és a szimuláció szempontjából figyelmen kívül hagyja.

A szintaxis tehát a következő:

- <szekciók> -ból épül fel a program, egy szekció a következők valamelyike lehet:
 - **section .data**
 - < <azonosító><kettőspont> <db|dw|dd> <kezdő értékek> kifejezések sorozata, szóközzel vagy sorvéggel elválasztva>
 - változók deklarálása, megadott kezdeti értékkel, a későbbiekben <azonosító> névvel lehet rájuk hivatkozni
 - a <kezdő értékek> számok (legalább egy darab) sorozata, vesszővel elválasztva
 - minden egyes számhoz ami a <kezdő értékek>-ben van a méretjelölőtől függően le lesz foglalva 1(**db**), 2(**dw**) vagy 4(**dd**) byte és az adott számra a felsorolásból lesz inicializálva a változó értéke
 - amennyiben a szám túl nagy (pl. 1000 nem fér el 1 byteon), akkor a magas helyiértékek túlcsordulás miatt elvesznek, ekkor az alacsony helyiértékek nem maximumra, hanem a nekik megfelelő értékre inicializálódnak (fentebbi példa esetén: 1000 modulo 256 = 232-ra)
 - amennyiben a szám kicsi és nem tölti ki az összes megadott byteot, akkor kerülhetnek a magasabb helyiértékekre nullák (és a következő szám illetve változó nem csúszik előrébb)
 - pl: „x: **dw** 11, 66000” összesen 4 byteot foglal le x-nek, értékük a lefoglalás sorrendjében 11, 0, 208, 1
 - **section .bss**
 - < <azonosító><kettőspont> <resb|resw|resd> <szám> kifejezések sorozata, szóközzel vagy sorvéggel elválasztva>
 - változó deklarálása, minden byte 0-ra inicializálva
 - a legfoglalt byteok száma <szám> * a méretjelölő értéke (**resb** = 1, **resw** = 2, **resd** = 4)
 - pl. „x: **resw** 3” kifejezés x-nek 3 * 2 = 6 byteot foglal le
 - **section .text**

- <utasítások sorozata, szóközökkel vagy sortörésekkel elválasztva>
- a szekció elején lehet „global <azonosító>” kifejezés, ez jelöli majd a program belépési pontját, ahonnan a végrehajtás elkezdődik
 - pontosan egy helyen kell belépési pontot megjelölni, különben a program hibát jelez
- **global** <azonosító>
 - megjelöli a program belépési pontját
 - lehet önállóan, szekcióként, nem csak text szekció elején
- <utasítás>
 - <azonosító><kettőspont>
 - megjelöl egy címkét az ugró utasításokhoz: amennyiben egy ugró utasítás <azonosító>-t jelöli meg céljául, akkor (amennyiben történik ugrás) a program végrehajtása a címke definícióját követő első utasítástól folytatódik
 - ha egy azonosító meg van adva ugrás célpontjaként, akkor a programnak pontosan egy címkét kell azzal az azonosítóval tartalmaznia
 - egy azonosító legfeljebb egy címkében szerepelhet
 - szerepelhet ugyanaz az azonosító egyszerre változóként és címkeként is (a hivatkozás kontextusából következik, hogy egy azonosító melyiként lesz értelmezve)
 - a címkéket a szimuláció nem kezeli önálló utasításként, a végrehajtás során átugrik rajtuk
 - minden címkét követnie kell utasításnak – nem követhet címkét egy új szekció vagy fájl vége
 - ugró utasítások:
 - **jmp** <azonosító>
 - az <azonosító>-val megjelölt címkét követő utasítással folytatja a program végrehajtását (a továbbiakban ezt csak „ugrásnak” fogom nevezni)

- minden ugró utasításnál van lehetőség a címke azonosítója előtt a „**near**” kulcsszó használatára, ennek a szimuláció szempontjából nincs hatása (viszont assembly kódban szerepelhet „hosszú” ugrások jelzésére)
- **ja** <azonosító>
 - amennyiben a sign flag és a zero flag értéke is 0, ugrik
- **jb** <azonosító>
 - amennyiben a sign flag értéke 1, és a zero flag értéke 0, ugrik
- **je** <azonosító>; **jz** <azonosító>
 - amennyiben a zero flag értéke 1, ugrik
- **jna** <azonosító>
 - amennyiben a sign flag vagy a zero flag értéke 1, ugrik
- **jnb** <azonosító>
 - amennyiben a sign flag értéke 0, vagy a zero flag értéke 1, ugrik
- **jne** <azonosító>; **jnz** <azonosító>
 - amennyiben a zero flag értéke 0, ugrik
- **call** <azonosító>
 - elmenti 4 byteon a következő utasítás sorszámát a verembe, majd ugrik
- **ret**
 - kivesz a veremből 4 byteot, majd a következő utasítás sorszámát a kivett értékre állítja
 - amennyiben a verem teteje ugyanoda mutat, ahová az utolsó **call** utasítás végrehajtása után mutatott (és az az által eltárolt 4 byte nem változott), akkor azt a **call**-t követő utasítástól folytatódik a végrehajtás – a **call** és a **ret** együtt körülbelül egy paraméterek nélküli függvényhívást valósítanak meg
 - a program nem végez ellenőrzést arra, hogy a verem tetejéről megfelelő értéket kapott a művelet – bár nem hibásodik meg a szimuláció, könnyen az utolsó utasítás utánra hivatkozhat, így befejezheti a szimuláció futását

A továbbiakban szereplő utasításoknál a következő jelöléseket alkalmazom:

- **<hely>**
 - egy olyan helyet jelöl, aminek az értéke módosítható, azaz lehet regiszter, változó- vagy veremhivatkozás
 - hivatkozás esetén a szintaxis: [<érték>]
 - amennyiben az <érték> kifejezése tartalmaz „esp”, „ebp”, „sp” vagy „bp” regiszterre hivatkozást, akkor a veremből veszi ki az értéket a program, hogyha nem tartalmaz ilyet, akkor a változók tömbjéből
 - amennyiben a hivatkozás érvénytelen helyre mutat (olyan változókezdetre, ami nincs a változók tömbjének határain belül, vagy a veremnek egy olyan pontjára, ami nincs a verem teteje és alja között), akkor a program hibát jelez
- **<érték>**
 - lehet regiszter, <azonosító> vagy konstans, illetve ezekből felépülhet zárójelezésekkel, összeadás (+), kivonás (-), szorzás (*) és osztás (/) műveletekkel
 - <azonosító> esetén az értéke az adott változónak a változók tömbjében elfoglalt első bytejának sorszáma
 - lehet hivatkozás is, ekkor a hivatkozott helyen található byteokból a program előállít egy számot – de hivatkozáson belül nem lehet még egy hivatkozás

A további utasításoknál fontos, hogy egyértelműnek kell lennie az argumentumok méretének, a méretet meghatározhatja regiszter vagy az egyik argumentum előtt szereplő **byte**, **word** vagy **dword** szavak (rendre 1, 2 és 4 byteos argumentumméretre utalnak). Amennyiben nincs méret megadva, vagy két eltérő méret van megadva, a program a kezdeti beolvasásnál hibát jelez.

- **mov** <hely>, <érték>
 - a <hely> által megjelölt helyre bemásolja az <érték>-ben szereplő értéket
- **add** <hely>, <érték>

- a <hely>-en levő értékhez hozzáadja <érték>-et
- amennyiben az összeadás eredménye 0 (nullák összeadásából, vagy túlcsordulás miatt), akkor a zero flag értékét 1-re állítja, különben 0-ra
- **sub** <hely>, <érték>
 - a <hely>-en levő értékből kivonja <érték>-et
 - amennyiben <érték> nagyobb mint a <hely>-en levő szám, akkor a sign flaget 1-re állítja, különben 0-ra
 - amennyiben a kivonás eredménye 0, a zero flaget 1-re állítja, különben 0-ra
- **and** <hely>, <érték>
 - „bitenkénti és” műveletet hajt végre a két argumentumon, az eredményt <hely>-en tárolja el
 - ha a művelet eredménye 0, a zero flaget 1-re állítja, különben 0-ra
- **or** <hely>, <érték>
 - „bitenkénti vagy” műveletet hajt végre a két argumentumon, az eredményt <hely>-en tárolja el
 - ha a művelet eredménye 0 (azaz mindkét argumentum értéke 0), a zero flaget 1-re állítja, különben 0-ra
- **xor** <hely>, <érték>
 - „bitenkénti kizáró vagy” műveletet hajt végre a két argumentumon, az eredményt <hely>-en tárolja el
 - ha a művelet eredménye 0 (a két argumentum egymás bitenkénti komplementere), akkor a zero flaget 1-re állítja, különben 0-ra
- **inc** <hely>
 - a <hely>-en található szám értékét megnöveli eggyel
 - túlcsordulás esetén a zero flaget 1-re állítja, különben 0-ra
- **dec** <hely>
 - a <hely>-en található szám értékét csökkenti eggyel
 - ha az érték 0-ra csökken, akkor a zero flaget 1-re állítja, különben 0-ra
- **not** <hely>

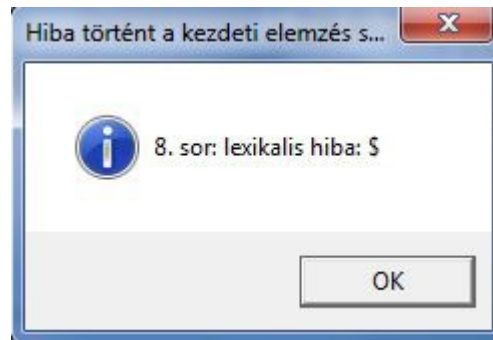
- „bitenkénti nem” műveletet hajt végre a <hely>-en, azaz minden bitet lecserél a komplementerére
- ha a művelet eredménye 0 (csupa egyes volt a <hely>-en a művelet előtt), akkor a zero flaget 1-re állítja, különben 0-ra
- **cmp <érték>, <érték>**
 - összehasonlítja a két értéket
 - ha megegyeznek, a zero flaget 1-re állítja, különben 0-ra
 - ha az első szám kisebb a másodikonál, a sign flaget 1-re állítja, különben 0-ra
- **mul <érték>**
 - <érték>-nek megadott argumentummérettől függően szorzást végez el
 - ha 1 byteos, akkor „al” regiszterrel szorozza meg, és „ax”-ben tárolja el a szorzás eredményét
 - ha 2 byteos, akkor „ax” regiszterrel szorozza meg, „ax”-ben tárolja el az eredmény kisebb helyiértékű 2 byteját és „dx”-ben a nagyobb helyiértékű 2 byteját
 - ha 4 byteos, akkor „eax” regiszterrel szorozza meg, „eax” regiszterben tárolja el az eredmény 4 alsó helyiértékű byteját és az „edx”-ben a 4 felső helyiértékű byteját
 - ha az eredmény 0, akkor a zero flag értékét 1-re állítja, különben 0-ra
- **div <érték>**
 - <érték>-nek megadott argumentummérettől függően maradékos osztást végez el
 - ha 1 byteos az argumentumméret, akkor az „ax” regiszter tartalmát elosztja <érték>-kel, az eredmény „al”-be, a maradék „ah”-ba kerül
 - ha 2 byteos, akkor összefűzi a „dx” és az „ax” regisztereket 4 byteon (ennek nincs nyoma a regiszterekben), ezt elosztja <érték>-kel, az eredmény „ax”-be, a maradék „dx”-be kerül
 - ha 4 byteos, akkor összefűzi az „edx” és az „eax” regisztereket 8 byteon, ezt elosztja <érték>-kel, az eredmény „eax”-be, a maradék „edx”-be kerül

- **push <érték>**
 - a futásidejű verem tetejére (ami az „esp” regiszter tartalmaz határoz meg) eltárolja <érték>-et az argumentum méret által meghatározott számú byteon, és az „esp” regiszter értékét ennek megfelelően módosítja
 - ha az eltárolás után 268.435.455-nél több byteot tartalmaz a verem, a program hibát jelez
- **pop <hely>**
 - a futásidejű verem tetejéről elvesz az argumentumméretnek megfelelő számú byteot, és <hely>-re eltárolja
 - amennyiben a veremben nincs annyi byte, amennyit a művelet ki akar venni, a program hibát jelez
 - az „esp” regiszter értékét megfelelően módosítja

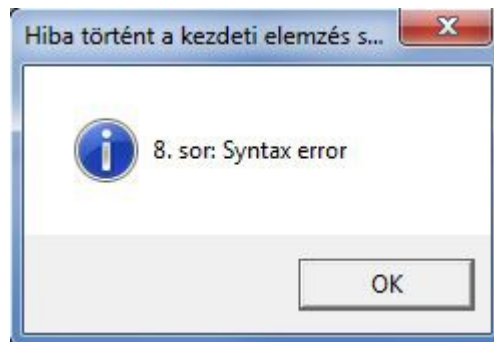
A program futása során fellépő hibaüzenetek

A program a futása során felugró ablakokkal jelzi, ha hibát talál. Ekkor a szimuláció nem folytatható tovább, ha a fájlt kijavítják közben, akkor újra meg kell nyitni. Fontos megjegyezni, hogy ha a program hibát jelez egy ponton, attól még lehetnek a bemeneti fájl későbbi részeiben is hibák, a szimuláció csak a legelső megtalált hibáról tájékoztat.

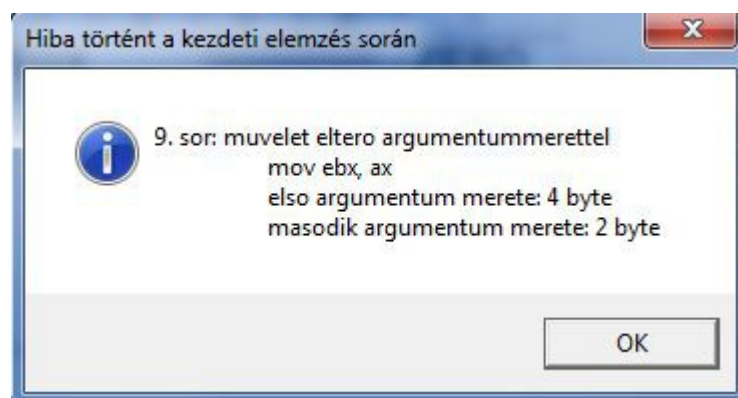
A továbbiakban következnek a lehetséges hibaüzenetek és javaslatok a megoldásukra. A javaslatokat megvalósítása során érdemes a szimulált program céljait is figyelembe venni, hogy a hiba kijavításával a kívánt működést is elérjük.



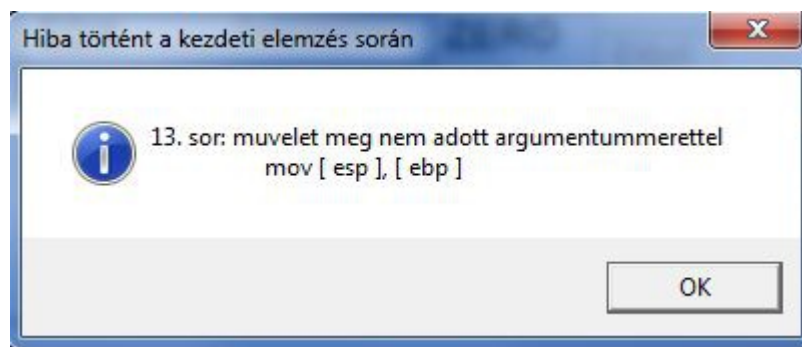
Lexikális hiba: a megnyitott assembly fájlban egy olyan karaktert talált, ami nem ismerhető fel a lexikális elemzés során. A hibaüzenet megadja, hogy hanyadik sorban van a hiba, és milyen karakterrel. Érdeemes megnézni, hogy nincs-e abban a sorban elírás.



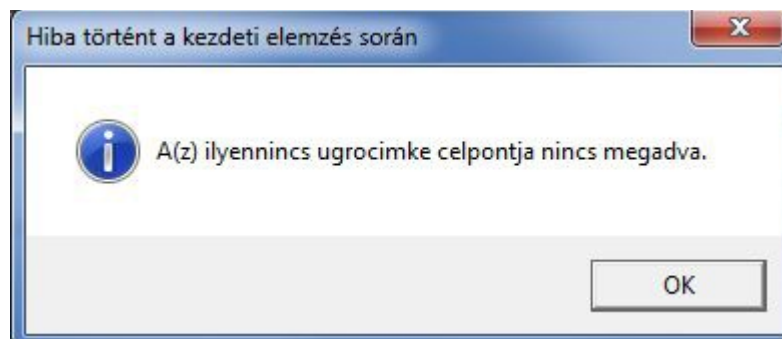
Szintaktikus hiba: a felismert szavakból nem áll elő az előző fejezetben leírt szintaxis. A hibaüzenet tájékoztat a hiba sorszámról, attól a sortól kezdve javasolt a hiba keresése.



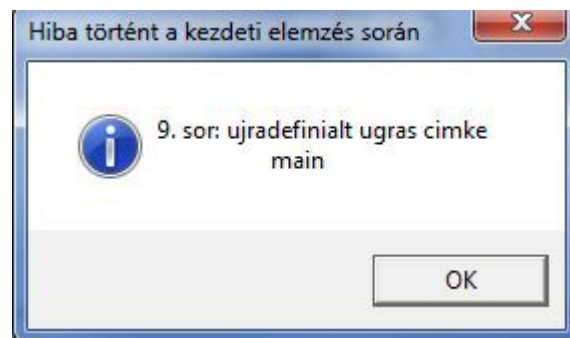
Egy műveletnek a két argumentumának a mérete (byteok száma) eltérő. Az egyik argumentum módosítása (hogy a másikkal megegyező méretű legyen) megoldja a problémát.



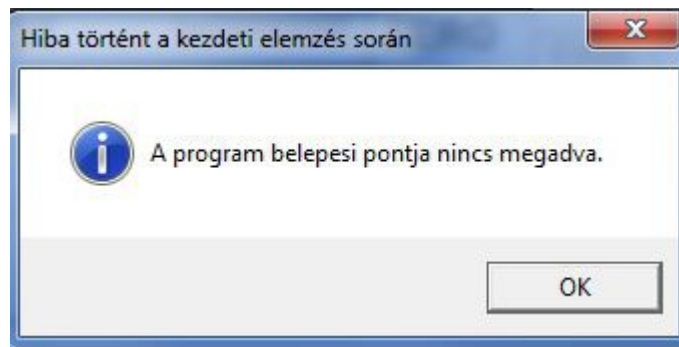
Egy műveletnek nincsen megadva az argumentummérete, nem egyértelmű, hogy hány byteosak az argumentumok. A **byte**, **word** vagy **dword** kulcsszavak valamelyikének beillesztése megszünteti a hibát.



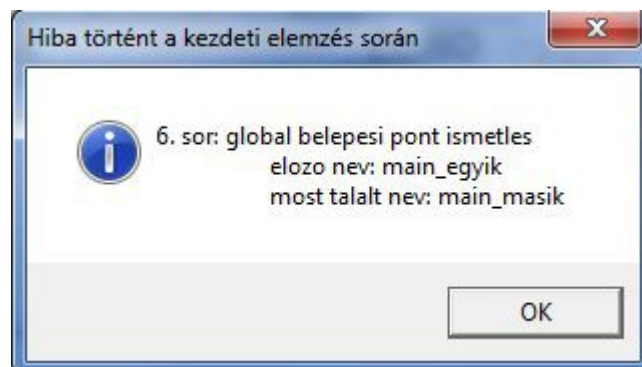
Van hivatkozás egy olyan ugrási címkére, ami nincs definiálva a programban. A címke létrehozásával megszüntethető a probléma.



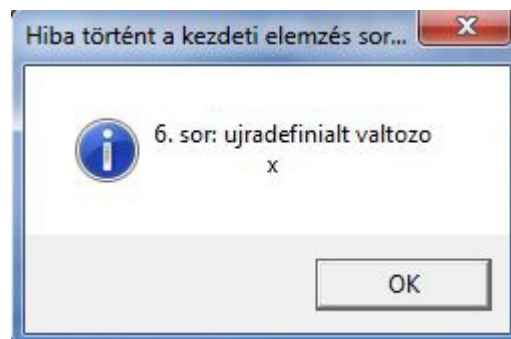
Ugyanazzal az azonosítóval legalább kettő ugrási címke van megadva, a hibaüzenetben írt sorszám a második előfordulást jelöli. Az egyik törlése megoldhatja a problémát, ha nincs belőle még legalább egy harmadik.



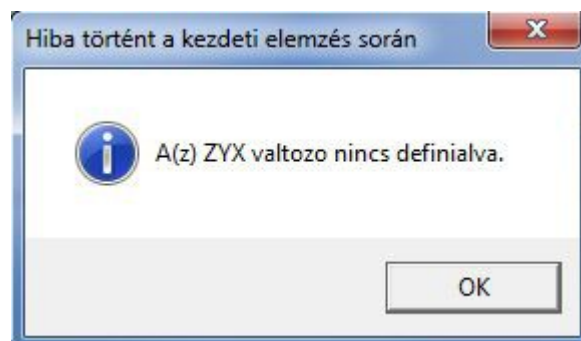
Nincs **global** kulcsszóval megadott belépési címke a fájlban.



A fájlban egynél több **global** kulcsszó van megadva, a hibaüzenet írja a másodiknak a sorszámát és mindkettőnek az azonosítóját.



Egy változó legalább kétszer van létrehozva. A megadott sorszám a második definíció helyét jelöli.

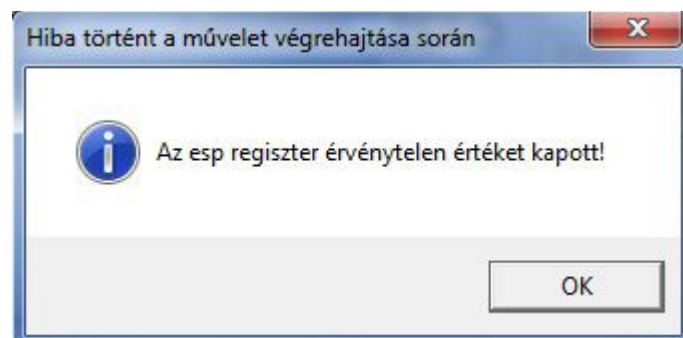


A hibaüzenetben jelzett nevű változóra van hivatkozás, de nincs létrehozva.

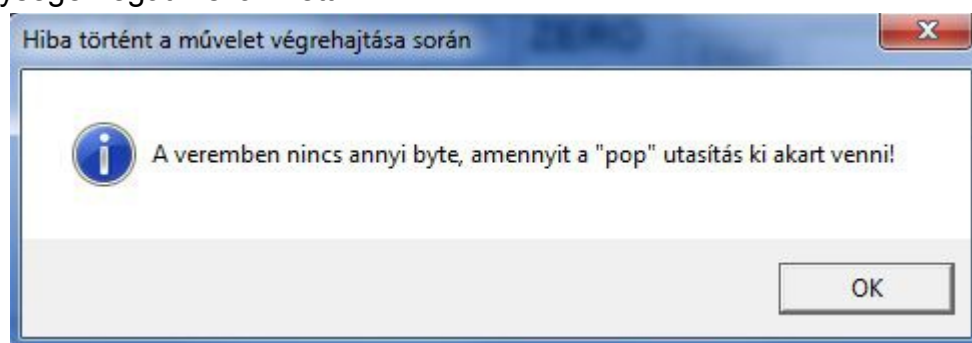
Futásidejű hibák:



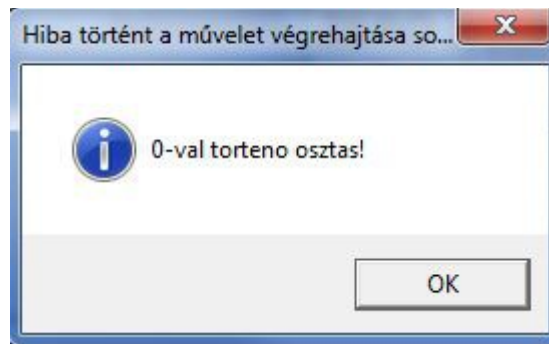
Egy változó vagy verem hivatkozás (szögleteszárójellel) érvénytelen: a változó tömbön vagy a vermen kívülre mutat.



Az *esp* vagy az *sp* regiszter módosítása során az *esp* által jelölt verem mélysége negatív szám lett.



A *pop* művelet argumentummérete nagyobb byteokban, mint a veremben tárolt byteok száma.



A végrehajtani próbált művelet „**div** 0” valamilyen argumentummérettel, a nullával való osztás nem hajtható végre.

"Megtelt a verem! A tartalom mérete túllépte a 268.435.455 byteot."

A futásidejű verem megtelt. Ez hiba a szimulációs program implementációjából származik, nem feltétlenül a a szimulált programban van a hiba.

Fejlesztői dokumentáció

A program feladata

A program célja, hogy egy x86 NASM assembly utasításokat használó programot utasításonként szimuláljon, azaz minden utasítás után megjelenítse, hogy a program milyen adatokat tárol a regiszterekben, a változóknak és a futásidejű veremben.

Megkötések

A program csak bizonyos utasítások felismerésére és végrehajtására képes, ezek a következők:

mov, add, sub, and, or, xor, cmp, inc, dec, not, mul, div; push, pop; jmp, ja, jb, je, jz, jna, jnb, jne, jnz, call, ret;

valamint csak bizonyos regisztereket kezel:

eax, ebx, ecx, edx, ax, bx, cx, dx, ah, al, bh, bl, ch, cl, dh, dli esp, ebp, sp, bp.

az *esp, ebp, sp, bp* regisztereket bizonyos szempontokból (változó hivatkozás) megkülönböztetett módon kezeli

Ezek mellett a szokásos módon lehet változókat deklarálni és inicializálni, belépési módot megadni, sor végéig kommentet írni (amit a szimuláció figyelmen kívül hagy), illetve címkéket elhelyezni (amikhez az ugró utasítás léptetheti a végrehajtást).

A könnyebb használat érdekében bizonyos helyeken nagyobb szabadságot adunk a felhasználónak:

- van lehetőség arra, hogy egy utasításnak mindkét argumentuma memória-hivatkozás legyen (normál esetben legfeljebb az egyik lehet az, a másiknak mindenképp regiszternek vagy konstansnak kell lennie)
- amikor valamilyen konstans szám-értéket akar a programban megadni, lehetőség van arra, hogy aritmetikai műveletek sorozatával adja meg
 - pl: ***mov*** *eax*, $256 * 256 * 4 + 256 * 11 + 37$

A felhasznált módszerek

A program C++ nyelven íródott, a lexikális elemzést flex által generált lexikális elemző csinálja, a szintaktikus és szemantikus elemzést bisonc++ által generált elemző hajtja végre.

A C++ Standard Template Libraryjéből az `std::map`, az `std::stack` és az `std::vector` osztályokat alkalmaztam.

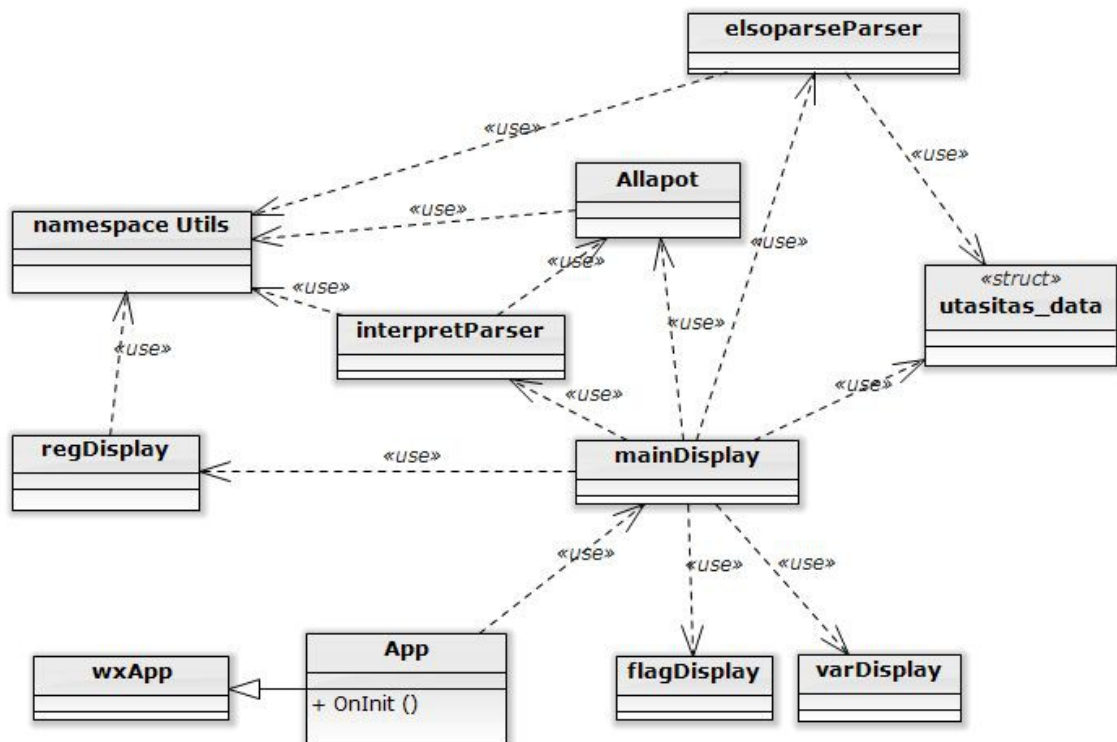
A grafikus felülethez wxWidgets eseményvezérelt könyvtárat használtam.

A programhoz a kódfájlokat Notepad++ alatt írtam, a fordításhoz a 64-bites Cygwin által csomagként telepíthető `x86_64-w64-mingw32-g++` -t használtam.

A tesztelés 64-bites Windows7 operációs rendszeren történt.

Mivel a bisonc++ Windows-on nem elérhető, a lexikális és szintaktikus elemzőket virtuális gépen, Debian operációs rendszeren generáltattam le.

Osztálydiagram



Az egyes osztályok leírásainál láthatók a részletes osztálydiagramok az osztályokról. A „wx” előtaggal kezdődő osztálynevek a wxWidgets library része, azoknak csak a funkciójáról lesz rövid leírás.

App

Az eseményvezérelt környezet elindításáért felelős osztály. Az OnInit() metódusból egy makró segítségével a wxWidgets main()-függvényt generál, ami megadja a program belépési pontját. Ősosztálya, a wxApp teszi ezt lehetővé.

Az egyes osztályok diagramjai, leírásai

namespace Utils

namespace Utils
+ vec_cout (vector<unsigned char>, string) + vecc2uint (vector<unsigned char>): unsigned int + vecc2sint (vector<unsigned char>): int + uint2vecc (unsigned int, vector<unsigned char>) + sint2vecc (int, vector<unsigned char>)

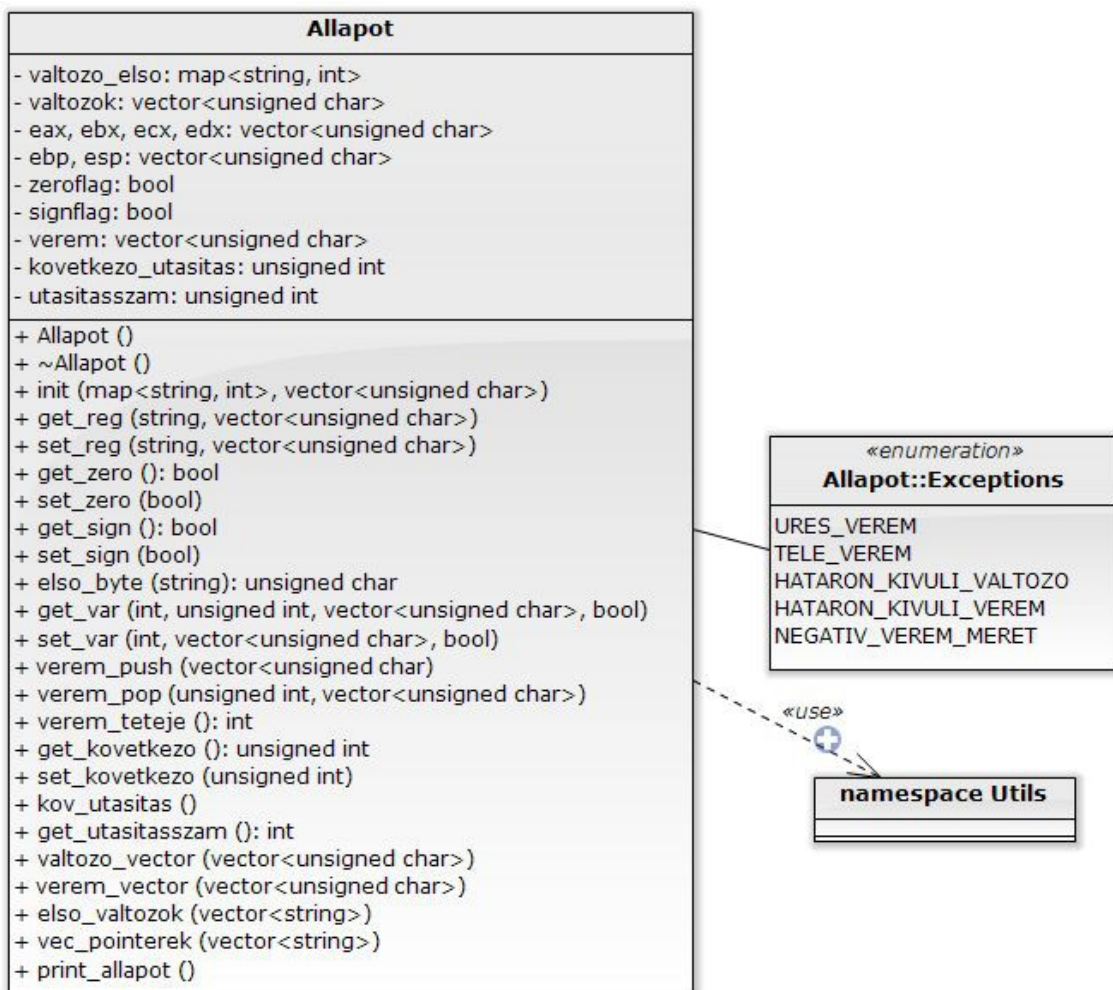
Kisegítő függvényeket tárol. A vektorra konvertáló műveletek esetén az átadott vektor paraméter hosszának akkorának kell lennie, ahány byteon tárolni akarjuk a kapott eredményt

- `vec_cout(vector<unsigned char>, string)`: kiírja a vektorban található értékeket tabulátorokkal elválasztva; ha a string nem üres, akkor az előtte levő sorba kiírja a string tartalmát
 - elsősorban tesztelésnél és debuggolásnál használt
- `vecc2uint(vector<unsigned char>)`: a vektor tartalmát átkonvertálja előjel nélküli egész számmá, mintha 256-os számrendszerbeli szám lenne, a vektor elején a kisebb helyiértéket tárolva
 - a konverzió egyértelmű
 - `uint2vecc(unsigned int, vector<unsigned char>)`: a számot átkonvertálja vektorra, ugyanezen elv alapján
- `vecc2sint(vector<unsigned char>)`: a vektor tartalmát átkonvertálja előjeles egész számmá
 - ha a vektor tartalma [255, 255, 255, 15], akkor ez a függvény nullát ad vissza, ha a vektor értéke (úgy, mint 256-os számrendszerbeli szám,

balról jobbra növekvő helyiértékekkel) nő, akkor a visszaszámított szám csökken

- elsősorban a verem regisztereinek konverziójára szolgál
 - lehetővé teszi, hogy ha pl. az `esp` regiszterhez hozzáadunk 4-et, akkor a verem mérete csökkenjen 4 byte-tal (ahogy azt egy assembly program esetében tenné)
- `sint2vecc(int, vector<unsigned char>)`: számot konvertál vektorrá, ugyanezen elv alapján

Allapot



Az ábrázolásnál a tárolt adatok bytejait minden esetben *unsigned char* típussal tároljuk. Ez az osztály az adott bemeneti fájlra egyértelműen meghatározza, hogy a szimulációnak melyik pontján vagyunk.

Adattagok:

- *valtozok*: a változók bytejainak tömbje
- *valtozo_elso*: azonosítókhoz tárolja a változók tömbben hozzá tartozó első byte sorszámát
- *eax, ebx, ecx, edx, ebp, esp*: regiszterek tömbje
 - az *eax, ebx, ecx, edx* esetén ha a regiszter értéke akkor nulla, ha a tömb összes eleme nulla
 - az *ebp, esp* esetén a regiszter értéke akkor nulla, ha *Utils::vecc2sint()* függvény nullát ad vissza
 - a *Utils::vecc2sint()* függvény ezekre a regiszterekre nempozitív számot ad vissza, annak az abszolútértéke jelöli, hogy hány byte van a pointer és a verem alja között
- *zeroflag, signflag*: a megfelelő flag aktuális értéke
- *verem*: a veremben tárolt byteok tömbje
- *kovetkezo_utasitas*: a következő utasítás sorszáma
- *utasitasszam*: az eddig végrehajtott utasítások száma

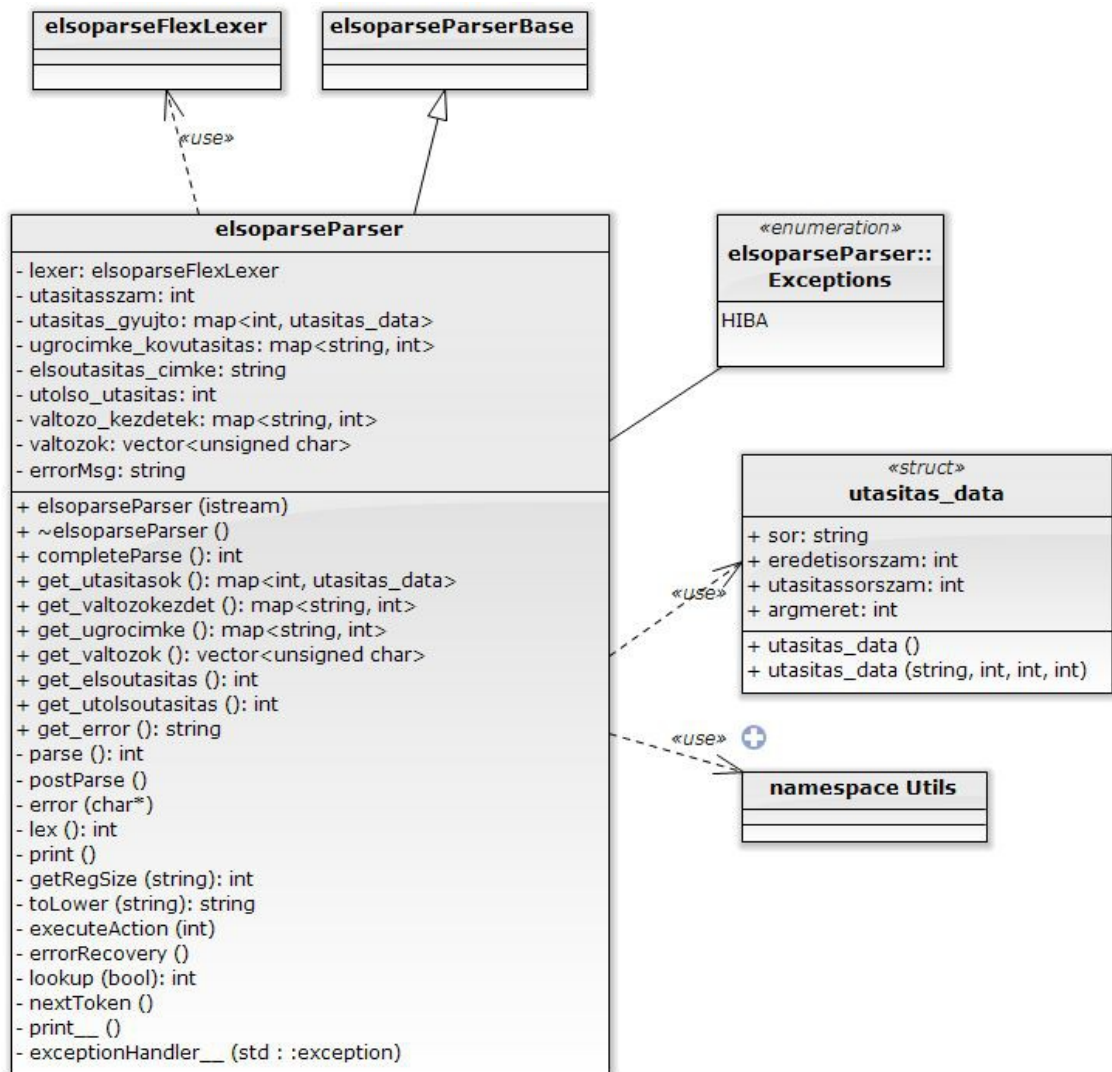
Metódusok:

- *Allapot()*: konstruktor, inicializálja a regisztereket, a vermet és a flageket
- *~Allapot()*: destruktork, jelenleg nem csinál semmit
- *init(map<string, int>, vector<unsigned char>)*: a megadott mappal inicializálja *valtozo_elso*-t, a vektorral pedig *valtozok*-at
- *get_reg(string, vector<unsigned char>)*: a megadott vektorba kimásolja a stringben megadott nevű regiszter értékét
 - ha nem 4 byte méretű regisztert kérdez le a függvény, akkor rövidebb lesz a visszaadott tömb is, és a megfelelő regiszter tartalma kerül bele
 - nem néz teljes egyezést a regiszter nevével, csak bizonyos részleteket – a string hosszát, a regisztert jelölő betűt (a, b, c, d, s)
 - amennyiben a string nem egyezik meg ezen a szinten egy regiszter nevével, akkor a függvény nem csinál semmit (és nem ad hibát)
 - a helyes működés biztosítása a programozó feladata

- `set_reg(string, vector<unsigned char>)`: a stringben megadott nevű regiszterbe bemásolja a vektor tartalmát
 - a `get_reg()`-hez hasonlóan lehet 4 byte-nál rövidebb regiszter névével is meghívni, ekkor a vektor elejéből vesz ki annyi byteot, amennyi a regiszterbe fér
 - feltételezzük, hogy a vektor hossza legalább akkora, mint ahány byteot ki akarunk másolni belőle, ennek biztosítása a programozó feladata
 - mint a `get_reg()`-nél, nincs teljes név ellenőrzés
 - dobhat `NEGATIV_VEREM_MERET` kivételt, amennyiben az `esp` regiszter negatív veremméretet jelölne
- `get_zero()`: visszaadja a *zero*flag értékét
- `set_zero(bool)`: beállítja a *zero*flag értékét
- `get_sign()`, `set_sign(bool)`: ugyanezek, csak a *sign*flagre
- `elso_byte(string)`: visszatér a stringben megnevezett változó első bytejának sorszámát
 - a programozó feladata arról gondoskodni, hogy a lekérdezett változó már tárolva legyen, tehát a kezdeti elemzés részeként érdemes ellenőrizni
- `get_var(int, unsigned int, vector<unsigned char>, bool)`
 - az `int` megadja az első byte sorszámát, az `unsigned int` a lekérdezett byteok számát, a vektor a cél ahova kimásolja a byteokat
 - ha a `bool` hamis (alapértelmezett), akkor a változók tömbjéből vesz ki, ha igaz, akkor a verem tömbjéből
 - dobhat `HATARON_KIVULI_VALTOZO` és `HATARON_KIVULI_VEREM` kivételt, ha az első byte és/vagy a hossz érvénytelen hivatkozást eredményez
- `set_var(int, vector<unsigned char>, bool)`
 - az `int`-ben megadott első byte sorszámától kezdve bemásolja a *valtozok* tömbbe a vektorban megadott értékeket (a darabszámot a vektor hossza határozza meg)
 - ha a `bool` hamis (alapértelmezett), akkor a változók tömbjét módosítja, ha igaz, akkor a verem tömbjét

- dobhat HATARON_KIVULI_VALTOZO és HATARON_KIVULI_VEREM kivételt, ha az első byte és/vagy a vektor hossza érvénytelen hivatkozást eredményez
- verem_push(vector<unsigned char>)
 - eltárolja a verembe a vektor tartalmát, *esp* regisztert megfelelően módosítva
 - dobhat TELE_VEREM kivételt, hogyha a veremben a művelet után 268435455-nál több byte van
- verem_pop(unsigned int, vector<unsigned char>)
 - a verem tetejéről kivesz az *int*-ben megadott számú byteot, és a vektorba másolja, *esp* regisztert megfelelően módosítva
 - dobhat URES_VEREM kivételt, ha a veremben nincs annyi byte, ahányat ki akar venni
- verem_teteje(): az *esp* regiszter alapján megmondja, hány byte van a veremben
- get_kovetkezo(): megadja a következő utasítás sorszámát
- set_kovetkezo(unsigned int): beállítja a következő utasítás sorszámát
- kov_utasitas(): a *utasitasszam* értékét eggyel növeli
- get_utasitasszam(): visszaadja az *utasitasszam* értékét
- valtozo_vector(vector<unsigned char>), verem_vector(vector<unsigned char>): a megadott tömbbe másolja a lekérdezett tömb elemeit
- elso_valtozok(vector<string>): a megadott tömböt feltölti üres stringekkel vagy az adott sorszámon elkezdődő változó nevével (a *valtozo_elso* mapból)
- vec_pointerek(vector<string>): a megadott tömböt feltölti üres stringekkel, illetve ahova az *esp*, *ebp*, regiszter mutat, oda a megfelelő névvel
- print_allapot(): kiírja a standard kimenetre az objektum tartalmát

elsoparseParser



Az **elsoparse.y** fájlból bisonc++ által generált szintaktikus és szemantikus elemző osztály. Ez az osztály hajtja végre az assembly fájl kezdeti beolvasását, és tőle lehet megkapni a szimulációhoz szükséges adatokat.

Az osztály által felismert környezetfüggetlen nyelvtan a következő (nagybetűs szavak jelölik a terminálisokat, kisbetűsek a nemterminálisokat):

- start → szekciok
- szekciok → szekcio szekciok | szekcio
- szekcio → global | SECTION DATA datadeklaraciok | SECTION BSS bssdeklaraciok | SECTION TEXT utasitasok | SECTION TEXT global utasitasok

- global → GLOBAL AZONOSITO
- datadeklaraciok → datadecl datadeklaraciok | datadecl
- bssdeklaraciok → bssdecl bssdeklaraciok | bssdecl
- datadecl → AZONOSITO KETTOSPONT meretdata szamok
- szamok → szamok VESSZO SZAM | SZAM
- bssdecl → AZONOSITO KETTOSPONT meretbss SZAM
- meretdata → DB | DW | DD
- meretbss → RESB | RESW | RESD
- utasitasok → utasitas utasitasok | utasitas
- utasitas → KETARGUMENTUMOS argumentum VESSZO argumentum | EGYARGUMENTUMOS argumentum | UGROUTAS AZONOSITO | UGROUTAS NEAR AZONOSITO | RET | cimke utasitas
- cimke → AZONOSITO KETTOSPONT
- argumentum → kifejezes | BYTE kifejezes | WORD kifejezes | DWORD kifejezes | NYITOSZOGZAROJEL kifejezes CSUKOSZOGZAROJEL | BYTE NYITOSZOGZAROJEL kifejezes CSUKOSZOGZAROJEL | WORD NYITOSZOGZAROJEL kifejezes CSUKOSZOGZAROJEL | DWORD NYITOSZOGZAROJEL kifejezes CSUKOSZOGZAROJEL
- kifejezes → NYITOZAROJEL kifejezes CSUKOZAROJEL | kifejezes MULTIPLY kifejezes | kifejezes DIVIDE kifejezes | kifejezes PLUS kifejezes | kifejezes MINUS kifejezes | REGISZTER | AZONOSITO | SZAM

Kisegítő osztályai:

- elsoparseFlexLexer
 - az **elsoparse.l** fájlból flex által generált lexikális elemző
- elsoparseParserBase
 - bisonc++ által generált ősosztály
- utasitas_data
 - az beolvasott utasításokat ilyen struktúrában tárolva menti el, az eredeti programbeli sorszámmal, hogy hányadik utasításként van számontartva, illetve hogy az argumentumok hány byteosak

Adattagok:

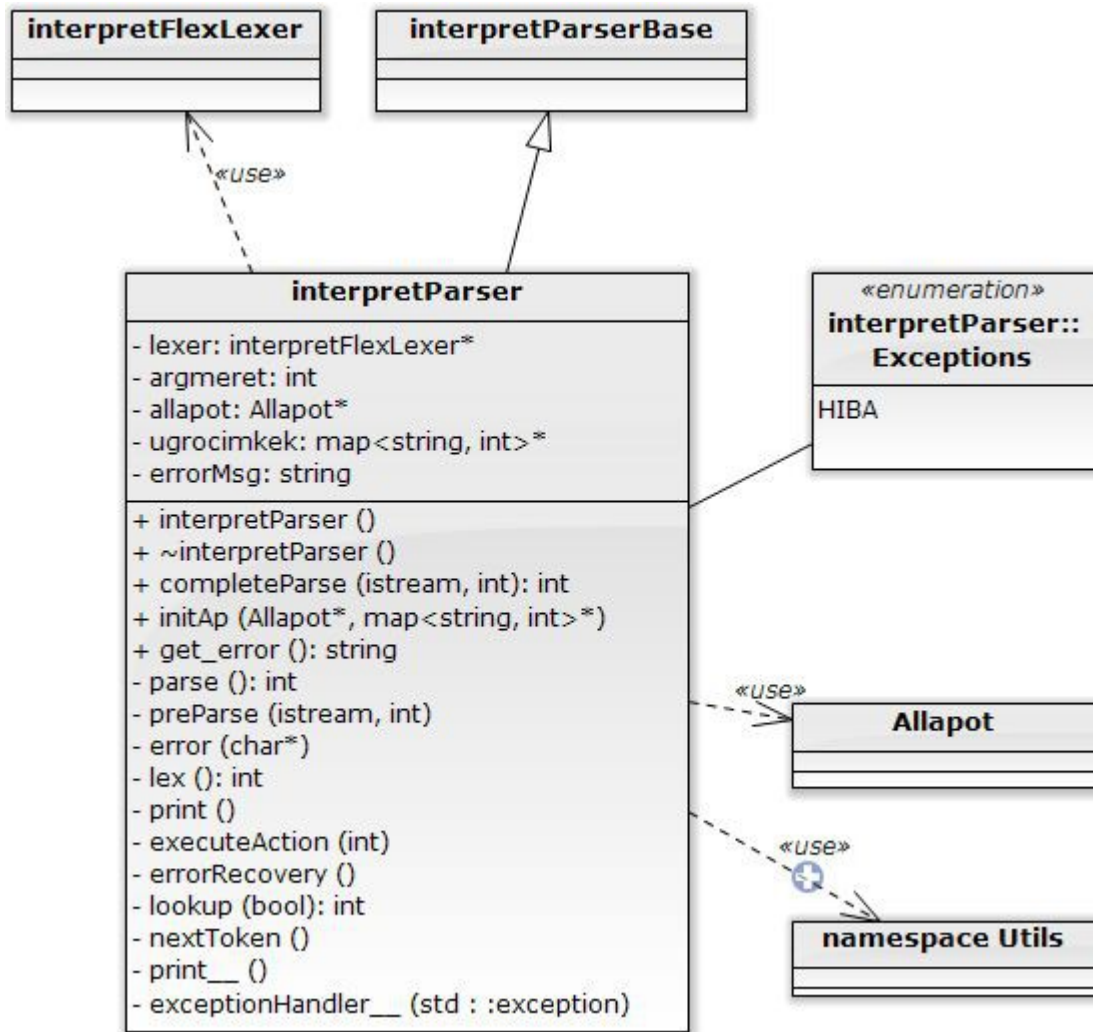
- *lexer*: a flex által generált lexikális elemző
- *utasitasszam*: hány utasítás van eltárolva jelenleg
- *utasitas_gyujto*: számhoz hozzárendelve az, hogy az adott számmal milyen utasítás van eltárolva
- *ugrocimke_kovutasitas*: a címkékhez hozzárendelve, hogy hanyadik utasítás követi őket
- *elsoutasitas_cimke*: a **global** kulcsszóval megjelölt belépési pontja az assembly programnak
- *utolso_utasitas*: az elemzés végén az a szám, amihez már nem tartozik utasítás – amennyiben a szimulációnál a jelenlegi utasítás felveszi ezt a számot, véget ért a szimuláció
- *valtozo_kezdetek*: a változók azonosítóihoz hozzárendeli, hogy hanyadik byte-tól indul az a változó
- *valtozok*: az eltárolt változók kezdeti értékei
- *errorMsg*: hiba esetén HIBA kivételt dob az elemző, és ez tárolja a hiba üzenetét

Metódusok:

- *elsoparseParser(istream)*: inicializálja az adattagokat
- *~elsoparseParser()*: destruktork, jelenleg nem csinál semmit
- *completeParse()*: meghívja a *parse()* metódust, majd a *postParse()* metódust
- *get_utasitasok()*, *get_valtozokezdet()*, *get_ugrocimke()*, *get_valtozok()*, *get_elsoutasitas()*, *get_utolsoutasitas()*, *get_error()*: lekérdező műveletek az adattagokhoz
- *parse()*: a bisonc++ által generált, a teljes elemzést végrehajtó metódus az **elsoparse.y** fájlban leírt nyelvtan alapján
- *postParse()*: az elemzés utáni ellenőrzéseket végrehajtó metódus: minden hivatkozott címke definiálva van-e, minden hivatkozott változó deklarálva van-e, van belépési pontja a programnak
- *getRegSize(string)*: egy regiszterre megmondja, hogy hány byte-os
- *toLower(string)*: visszatér a szöveggel, csupa kisbetűvel

- `error(char*)`, `lex()`, `print()`, `executeAction(int)`, `errorRecovery()`, `lookup(bool)`, `nextToken()`, `print__()`, `exceptionHandler__(std::exception)`: a bisonc++ által generált, a parseolás során felhasznált metódusok

interpretParser



Az **interpret.y** fájlból bisonc++ által generált szintaktikus és szemantikus elemző osztály. Ez az osztály értelmez egy assembly utasítást, és végrehajtja a megfelelő műveleteket a megadott állapoton.

Az osztály által felismert környezetfüggetlen nyelvtan a következő:

- `start` → utasítás
- `utasitas` → `MOV elsoarg VESSZO masodarg` | `ADD elsoarg VESSZO masodarg` | `SUB elsoarg VESSZO masodarg` | `CMP masodarg VESSZO`

- masodarg | AND elsoarg VESSZO masodarg | OR elsoarg VESSZO
 masodarg | XOR elsoarg VESSZO masodarg | MUL masodarg | DIV
 masodarg | INC elsoarg | DEC elsoarg | NOT elsoarg | PUSH masodarg |
 POP elsoarg | ugroutas AZONOSITO | CALL AZONOSITO | RET
- ugroutas → JMP | JA | JB | JE | JZ | JNA | JNB | JNE | JNZ
 - elsoarg → REGISZTER | NYITOSZOGZAROJEL érték
CSUKOSZOGZAROJEL
 - masodarg → érték | NYITOSZOGZAROJEL érték CSUKOSZOGZAROJEL
 - érték → NYITOZAROJEL érték CSUKOZAROJEL | érték MULTIPLY érték |
érték DIVIDE érték | érték PLUS érték | érték MINUS érték | REGISZTER |
AZONOSITO | SZAM

Kisegítő osztályai:

- interpretFlexLexer:
 - az **interpret.l** fájlból flex által generált lexikális elemző
- interpretParserBase:
 - bisonc++ által generált ősosztály

Adattagok:

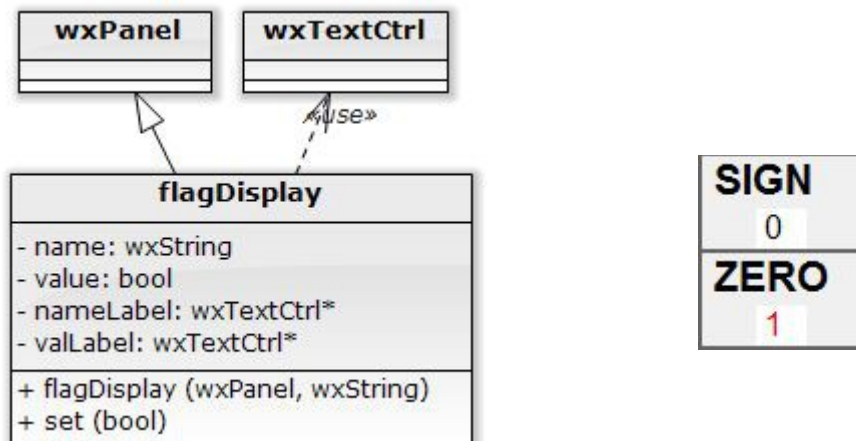
- *lexer*: a flex által generált lexikális elemző
- *argmeret*: az aktuálisan végrehajtott utasításnak az argumentum mérete
byteban
- *allapot**: az az Allapot, amit az utasítások hatására módosít
- *ugrocimkek**: ebben tárolja el, hogy melyik címkét hanyas számú utasítás
követi
- *errorMsg*: az elemzés során esetlegesen tapasztalt hibák hibaüzenetét
tárolja, lehet nullával osztás vagy lexikális hiba

Metódusok:

- interpretParser(): konstruktor, a *lexer* pointert nullpointerre állítja
- ~interpretParser(): destruktork, ha a *lexer* pointer nem nullpointer, akkor törli a
lexert

- az *allapot* és az *ugrocimkek* kívülről átadott változók, az átadó osztály felelőssége felszabadítani a memóriát (ha dinamikusan vannak lefoglalva)
- `completeParse(istream, int)`: meghívja a `preParse()` és a `parse()` függvényeket, illetve az *allapot*-ot a következő utasításra állítja (ami még a `parse()` futása során módosulhat ugró utasítás esetén)
- `initAp(Allapot*, map<string, int>*)`: inicializálja az *allapot* és az *ugrocimkek* változókat
- `get_error()`: lekérdezi az *errorMsg* változót
- `parse()`: a bisonc++ által generált, egy utasítás végrehajtását szimuláló metódus az **interpret.y** fájlban leírt nyelvtan alapján
- `preParse(istream, int)`: létrehozza a *lexert* a megadott adatfolyammal és beállítja az *argmeret* változót
- `error(char*)`, `lex()`, `print()`, `executeAction(int)`, `errorRecovery()`, `lookup(bool)`, `nextToken()`, `print__()`, `exceptionHandler__(std::exception)`: a bisonc++ által generált, a parseolás során felhasznált metódusok

flagDisplay



A flag-értékek kijelzéséért felelős osztály. A jobb oldali ábrán látható a megjelenése a programban, a szöveg a flag nevét jelöli, a szám pedig az aktuális értékét.

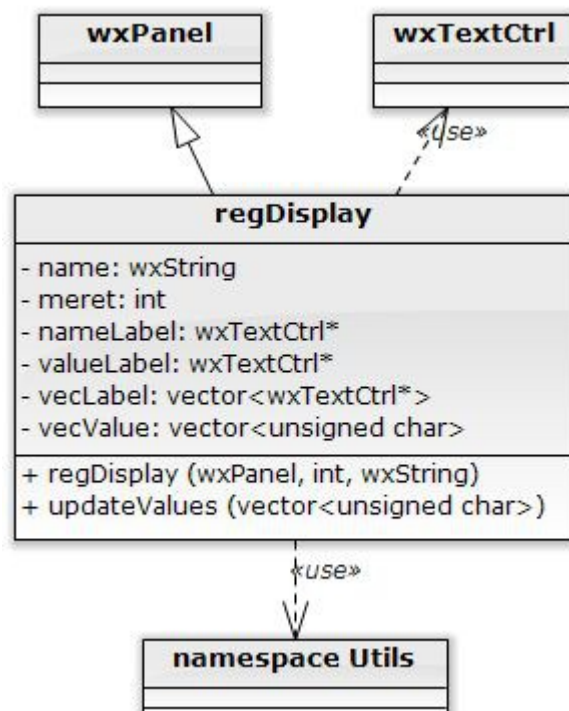
Adattagok:

- *name*: a kiírt név, konstans
- *value*: az aktuálisan felvett érték
- *nameLabel*: a nevet kijelző szövegdozoz
- *valLabel*: az értéket kijelző szövegdozoz

Metódusok:

- `flagDisplay(wxPanel, wxString)`: meghívja az őosztály konstruktorát, beállítja a panel szülőjét, beállítja a *name* adattagok, inicializálja a szövegdozozokat
- `set(bool)`: beállítja a *value*t, frissíti a *valLabel* tartalmát

regDisplay



A regiszterek tartalmának kijelzéséért felelős osztály. Az ábrán látható a megjelenése a programban, a szöveg a regiszter nevét jelöli, a szöveg alatti számok a regiszter bytejainak tartalmát (balról jobbra helyiérték szerint növekvően), a név melletti hexadecimális szám pedig a regiszter egészében tárolt értéket (tizenhatos számrendszerben).

EAX				0x000000ad				EBX				0x00000201			
173	0	0	0					1	2	0	0				
ECX				0x00000000				EDX				0x00000002			
0	0	0	0					2	0	0	0				

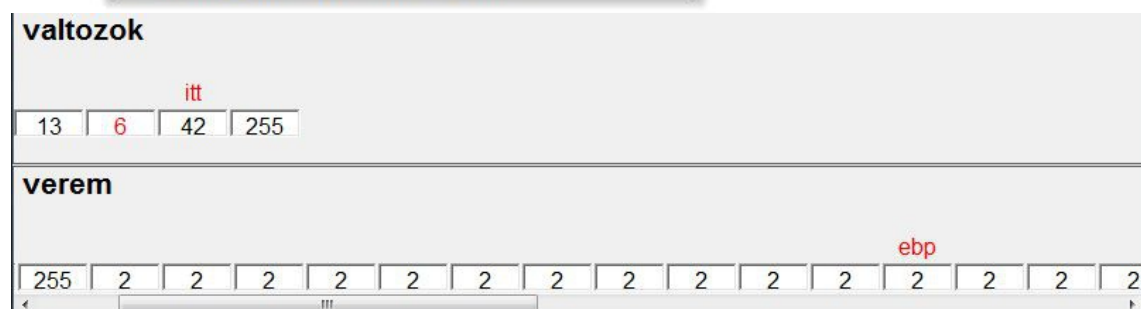
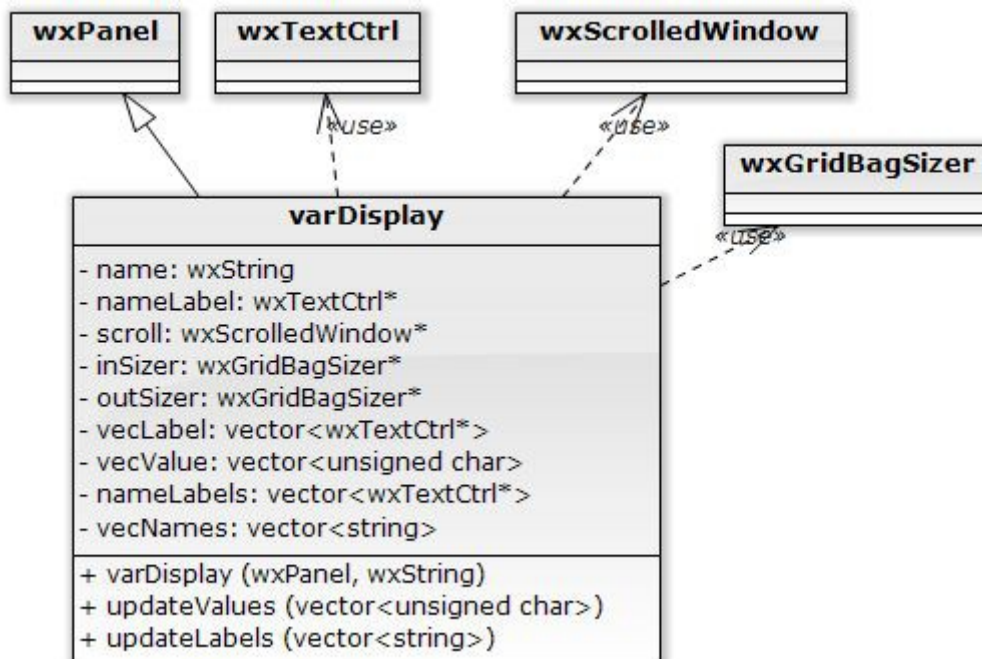
Adattagok:

- *name*: a regiszter kiírt neve, konstans
- *meret*: hány byteból áll a regiszter, konstans
- *nameLabel*: a nevet kijelző szövegdoboz
- *valueLabel*: a hexadecimális értéket kijelző szövegdoboz
- *vecLabel*: a byteokat kijelző szövegdobozokat tároló tömb
- *vecValue*: a byteok értékeit tároló tömb

Metódusok:

- `regDisplay(wxPanel, int, wxString)`: meghívja az ősosztály konstruktorát, beállítja a panel szülőjét, beállítja a *name* és a *meret* adattagokat, létrehozza a kijelző objektumokat
- `updateValues(vector<unsigned char>)`: módosítja a kijelzett byteokat a tömbben megadottakra, frissíti a *valueLabel* tartalmát is
 - nem ellenőrzi a tömb méretének helyességét, ennek biztosítása a programozó feladata

varDisplay



A változók és a futásidejű verem megjelenítésére szolgáló osztály. A vastag betűs szöveg jelöli, hogy az adott rész melyik célt szolgálja, a számok sorbarendezve jelölik az értékeket, fölöttük a szövegek jelölik a változók kezdetét és a verem pointerek aktuális helyzetét. Ha sok adatot tárol, akkor a téglalap alján megjelenik egy gördítősáv, aminek segítségével végig lehet nézni az egész tömb tartalmát

Adattagok:

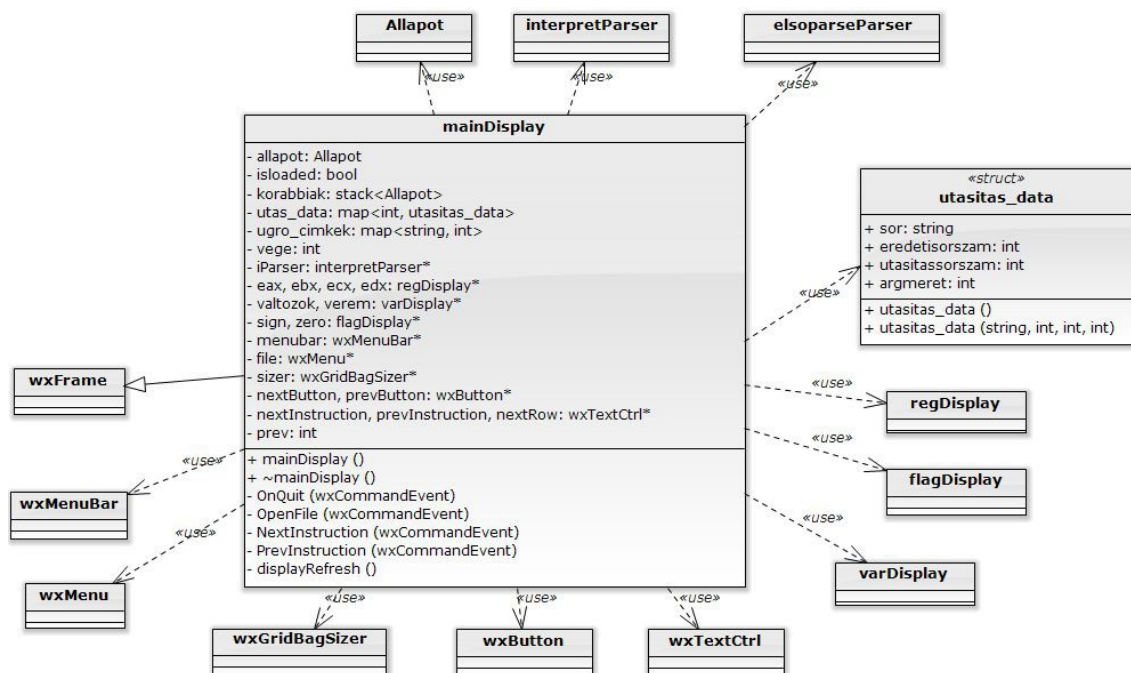
- *name*: az objektum kiírt neve, konstans
- *nameLabel*: az objektum nevét kijelző szövegdoboz
- *scroll*: a gördítősáv megjelenéséért felelős objektum
- *inSizer*: számok és a feliratok elrendezését tároló objektum

- *outSizer*: a név és a gördítőablak elkülönítésére szolgáló, az elhelyezésüket tároló objektum
- *vecLabel*: a számok megjelenítését szolgáló objektumok tömbje
- *vecValue*: az aktuálisan tárolt számok tömbje
- *nameLabels*: a kiegészítő feliratokat megjelenítő objektumok tömbje
- *vecNames*: a kiegészítő feliratokat tároló tömb

Metódusok:

- *varDisplay(wxPanel, wxString)*: az őszosztály konstruktorát meghívja, beállítja az objektum szülőjét, beállítja a *name* adattagot, létrehozza az elrendező objektumokat
- *updateValues(vector<unsigned char>)*: beállítja a kiegészítő feliratokat
- *updateLabels(vector<string>)*: beállítja a kijelzett értékeket
 - a két update-eljárásnál a hasonló tömb-méreték biztosítása a programozó feladata, az osztály nem ellenőrzi őket

mainDisplay



A megjelenítésért, az események kezeléséért és általánosságban a futtatásért felelős osztály. Az eseményvezérlés működését a headerben a

wxDECLARE_EVENT_TABLE() és a wxDECLARE_EVENT() makrók, a forrásfájlban a wxDEFINE_EVENT() és az event-tábla makrók teszik lehetővé.

File						
EAX	0x00000006	EBX	0x00000000	SIGN	31. sor:	pop ecx
6	0	0	0	0		
ECX	0x00000000	EDX	0x00000000	ZERO		
0	0	0	0	1	Előző:	push eax
Változók						
x	y	yy	z			
1	2	5	15	0	0	0
Verem						
ebp	esp					
0	6					

Adattagok:

- *allapot*: a szimuláció aktuálisan eltárolt állapota
- *isloaded*: tárolja, hogy tovább lehet-e lépni az aktuális utasítás végrehajtásával
- *korábbiak*: a korábbi állapotokat tárolja, lehetővé téve az utasítások visszavonását
- *utas_data*: az elsoparseParser által beolvasott utasítás információkat tárolja
- *ugro_cimkek*: az elsoparseParser által beolvasott ugrási címkéket tárolja, az *iParser* ehhez pointeren keresztül hozzáférést kap (olvasáshoz)
- *vege*: az elsoparseParser által beolvasott utolsó utasítást jelző számot tárolja (ha az *allapot.get_kovetkezo()* eléri ezt az értéket, akkor nincs több végrehajtható művelet)
- *iParser*: az *allapot* objektumon szimulációt végző interpretParser
- *eax, ebx, ecx, edx*: a regiszterek megjelenítését végző objektumok
- *valtozok, verem*: a változók tömbjének és a futásidejű verem megjelenítését végző objektumok
- *sign, zero*: a flag-értékek kijelzését végző objektumok
- *menuBar*: a felső menüsáv
- *file*: a menüsáv egyetlen opciója

- *sizer*: a megjelenítő objektumokat rendezi el
- *nextButton*, *prevButton*: a szimuláció következő utasítását végrehajtó illetve utolsó utasítását visszavonó nyomógombok
- *nextInstruction*, *prevInstruction*: a következő és az utolsóként végrehajtott művelet szövegét megjelenítő szövegdobozok
- *nextRow*: a következő utasítás sorszámát (az eredeti fájlban) kiíró szövegdoboz
- *prev*: az aktuális utasítást megelőző sorszáma (a *prevInstruction* szövegéhez szükséges)

Metódusok:

- *mainDisplay()*: meghívja az őosztály konstruktorát, létrehozza a megjelenítő objektumokat, elrendezi őket
- *~mainDisplay()*: destruktorként, ha létre lett hozva *iParser*, akkor kitörli
- *OnQuit(wxCommandEvent)*: kilépési esemény, a *file* menü „Kilépés” gombja hatására fut le, kilépteti a programot
- *OpenFile(wxCommandEvent)*: a *file* menü „Fájl megnyitása” opciójának hatására fut le, fájl dialógus ablak segítségével bekér a felhasználótól egy fájlnevet, azon végrehajtja az *elsoparseParser* kezdeti elemzését és inicializálja az *allapot* és *iParser* változókat
 - lekezeli a fájl megnyitása és elemzése során fellépő hibaüzeneteket, a felhasználót ezekről felugró ablakkal informálja
- *NextInstruction(wxCommandEvent)*: a „Végrehajt” feliratú *nextButton* nyomógomb megnyomására fut le, az *allapot*ot eltárolja a *korabbiak* verembe, az *iParser* használatával végrehajtja a soron következő utasítást, frissíti a kijelzést
 - az esetlegesen fellépő hibákat (*Allapot::Exceptions*, *interpretParser::Exceptions*) elkapja és a felhasználót felugró ablakkal tájékoztatja
 - ha nincs több végrehajtható utasítás, kikapcsolja a *nextButton* gombot
- *PrevInstruction(wxCommandEvent)*: a *korabbiak* tetejéről leveszi az utolsó művelet végrehajtása előtt odahelyezett *Allapot*ot és betölti annak tartalmát

- a *prevButton* és *nextButton* gombok státuszát megfelelően beállítja

Tesztelés

namespace Utils

Tesztfájl programkódja: ***utils_main.cpp***

Lefordítása „make utils” utasítással, a készített fájl neve *teszt_utils.exe*

Tesztelés: a *vecc2sint()* és a *sint2vecc()* függvények működése.

- 0-at megfelelő vektorra konvertálnak, az visszakonvertálva is 0.
- -1-et konvertálva és visszakonvertálva az eredmény -1.
- -2-őt konvertálva, visszakonvertálva, 8-at kivonva belőle majd újra oda-visszakonvertálva az eredmény -10
- -12-vel ugyanez, csak 8-at hozzáadva, az eredmény -4

Ezzel jól látható, hogy a vektor átkonvertálása számmá és azon művelet végzés megfelelővé teszi arra, hogy a veremhez tartozó regisztereket átalakítsa, mivel az átalakítás után a hozzáadás csökkenti az abszolútértékét (így *esp* esetén a veremben található byteok számát), ami az assembly nyelv működésével egybevág.

Allapot

Tesztfájl programkódja: ***allapot_main.cpp***

Lefordítása „make állapot” utasítással, a készített fájl neve *teszt_allapot.exe*

Tesztelés:

- flagek működése: *set_sign()*, *get_sign()*, *set_zero()*, *get_zero()*
 - alapértelmezetten az értékük 0 (hamis), igazra állítva az értékük 1 (igaz)
- regiszterek működése: *set_reg()*, *get_reg()*
 - vektor értékekkel feltöltése, *eax* regiszterbe átmásolása *set_reg()* utasítással, az *eax* regiszter lekérdezése egy másik vektorba, majd annak kiírása
 - ugyanez megismétlése az *ah* regiszterrel, ekkor *eax* felső részében még megtalálhatók a korábban belemásolt értékek
 - *ebx*, *ecx*, *edx* regiszterek módosítása, majd a megfelelő rész-regiszterek lekérdezése

- ezután az *ebx* regiszter lekérdezése, demonstrálva, hogy az *ecx* és az *edx* módosítása nem hatott rá, végül a teljes állapot kiírása
- verem működése: *verem_push()*, *verem_pop()*, *set_reg()* *esp/ebp*-re, *get_reg()* *esp/ebp*-re, *verem_teteje()*, *get_var()*
 - verembe értékek pusholása, majd verem lekérdezése, utána az adatok poppolása, és összehasonlítás az eredetivel
 - üres veremből poppolás, dobott *URES_VEREM* kivétel elkapása
 - *esp* regiszter *get_reg()*-gel lekérdezve, konvertálva, a kapott számból 8 kivonása, visszakonvertálva és *set_reg()*-gel visszatöltve, a verem helyesen 8 byteot tartalmaz, majd kétszer 4 byte poppolása
 - *ebp*-be *esp* kimásolása (üres verem), 3-szor 4 byte pusholása, majd *(ebp – 4)* és *(ebp – 8)* lekérdezése *get_var()*-ral
 - *esp* regiszter lekérdezése, 16 hozzáadása, visszatöltése *NEGATIV_VEREM_MERET* kivételt dobott (mivel a verem -4 byteot tartalmazna)
- változók tesztje: *get_var()*, *set_var()*, *elso_byte()*
 - *valtozok* vektor és *valtozo_elso* inicializálása
 - első byteok lekérdezése
 - *set_var()*, majd *get_var()* meghívása ugyanarra az első bytera
 - *set_var()* meghívása úgy, hogy átcsússzon egy másik változó memóriaterületére (a tömbön belül), majd a változók kiírása
 - ugyanez megismételve úgy, hogy a másik változónak csak az aljába csússzon bele
 - hibakezelések ellenőrzése, 10 byteos tömbnek a -1. bytetól, illetve 11. byteig lekérdezése, írási kísérlete *HATARON_KIVULI_VALTOZO*-t dobott
 - ezek után az állapot kiírása – a regiszterek és a verem változatlanok
- verem limit teszt:
 - végtelen ciklussal 4 byteos vektor pusholása a verembe, ez 67108863 lépés után dobott *TELE_VEREM* kivételt, összesen 268435452 byte került a verembe

elsoparseParser

Tesztfájl programkódja: ***elsoparse.cpp***

Lefordítása „make elsoparse” utasítással, a készített fájl neve `teszt_elsoparse.exe`

Tesztelés: a lefuttatjuk a parsert és kiíratjuk a gyűjtött adatokat, amennyiben van megadva a programnak parancssori argumentum, akkor arra a fájlra futtatja le, ha nincs, akkor a `teszt1.asm` fájlra.

- lefuttatjuk a parsert a bemeneti fájlra
- lekérdezzük a parsertől a kigyűjtött adatokat: `get_utasitasok()`, `get_valtozokezdet()`, `get_ugrocimke()`, `get_valtozok()`, `get_elsoutasitas()`, majd iterátorok segítségével végigmegyünk a kapott mapeken és vektoron, és kiírjuk az értékeket
- a kiíratott értékeket összehasonlítjuk a bemenetnek megadott assembly fájl alapján keletkezendő értékekkel
- közben figyeljük, hogy a parser dob-e kivételt, ha igen, akkor kiírjuk a hiba üzenetét

interpretParser

Tesztfájl programkódja: ***interpret.cpp***

Lefordítása „make interpret” utasítással, a készített fájl neve `teszt_interpret.exe`

Tesztelés: lefuttatjuk az elsoparseParsert, az általa gyűjtött adatokon és egy Allapot objektumon végigfuttatjuk az interpretParsert. Amennyiben meg van adva parancssori argumentum, akkor arra a fájlra futtatjuk le a szimulációt, ha nincs, akkor a `teszt1.asm` fájlra.

- lefuttatjuk az elsoparseParsert a bemeneti fájlra, a kapott adatokat kigyűjtjük és inicializálunk vele egy Allapot objektumot
- létrehozuk az interpretParsert, és inicializáljuk az állapottal és az ugrás címkékkel
- amíg az Allapotban tárolt következő utasítás sorszám nem egyezik meg a kezdeti elemzés során kapott utolsó utasítás számával, addig a következő utasításra lefuttatjuk az interpretParsert

- a futás után kiíratjuk az Allapot tartalmát és Enter leütéséig várunk a ciklus következő futásával
- menet közben figyeljük, hogy dob-e hibát az Allapot vagy az interpretParser, amennyiben ilyen tapasztalunk, a program kiírja a kapott hiba típusát és kilép

A grafikus felület rész-osztályai

Tesztfájlok programkódjai: ***ui_parts_main.cpp***, ***ui_parts_appmain.cpp***

Lefordítása „make ui_parts” utasítással, a készített fájl neve **teszt_ui_parts.exe**

Tesztelés: egy módosított mainDisplay osztályt hozunk létre

- létrehozuk a kijelző osztályokat és elrendezzük őket
- létrehozuk a menüsávot és belerakjuk a „Fájl megnyitása” opciót
- egy módosított displayRefresh() függvénnyel feltöltjük a regDisplayeket adatokkal, majd ugyanezt a flagDisplayekre és a varDisplayekre
 - a varDisplayekre egy függvényhíváson belül több módosítást is küldünk, hogy ellenőrizzük a méret változtatásával járó viselkedést
- nincs kapcsolat az elsoparseParserrel és az interpretParserrel, csak a megjelenítést ellenőrizzük

A teljes program

Lefordítás „make” paranccsal, a készített fájl neve **szimulacio.exe**

Tesztelés a mainDisplay osztályon keresztül, annak felhasználói lehetőségeit használva, mind az előre-, mind a visszalépést alkalmazva.

Tesztesetek:

- kezdeti elemzés során észlelt hibás bemeneti fájlok:
 - hibas_argmeret.asm: eltérő argumentum méretek
 - hibas_argmeret_nincs.asm: nem megadott argumentumméretek
 - hibas_cimke_nincs.asm: nem létező címkére hivatkozás
 - hibas_ket_valtozo.asm: ugyanazzal az azonosítóval kétszer definiált változó
 - hibas_lexerror.asm: lexikális hiba
 - hibas_nincs_valtozo.asm: hivatkozott, nem definiált változó
 - hibas_noglobal.asm: nincs belépési pont (**global**) megadva

- hibas_sok_cimke.asm: ugyanaz a címke kétszer megadva
- hibas_syntaxerror.asm: szintaktikus hiba
- hibas_twoglobal.asm: két belépési pont van megadva
- futásidőben hibás bemeneti fájlok:
 - hibas_div0.asm: 0-val osztás
 - hibas_esp.asm: verem mélységének negatívra állítása
 - hibas_ures_pop.asm: üres veremből pop
 - hibas_valtozok.asm, hibas_valtozok2.asm: érvénytelen változó referencia
 - hibas_verem_ref.asm, hibas_verem_ref2.asm: érvénytelen verem referencia
- helyes tesztekre
 - factorial.asm, factorial4.asm: 3 illetve 4 faktoriálisának rekurzív kiszámítása
 - jump_teszt.asm: feltételes ugrások helyességének ellenőrzése
 - valtozok_teszt.asm: változók, veremhivatkozások
 - teszt1.asm: általános tesztelés, változók inicializálása, egyszerű műveletek
 - cimke_teszt.asm: változónévvel megegyező címke
 - logic_div_teszt.asm: logikai műveletek, egyszerű osztás
 - div_overflow.asm: osztás úgy, hogy *edx* regiszterben van magas helyiérték
 - mul_overflow.asm: szorzás úgy, hogy *edx* regiszterbe túlcsordul
 - nagyverem.asm: hosszú program, a verembe 400 byte kerül, összesen 500 állapotot tárol el

Továbbifejlesztési lehetőségek

Több művelet, regiszter és flag támogatása.

Felhasználói felület fejlesztése: a hosszabb változónevek is kiferjenek, egyszerre kettőnél több utasítást jelenítsen meg, egy kattintással egynél több lépést hajtson végre, billentyűlenyomással lehessen navigálni, beépített súgó.

Mentési lehetőség: a szimuláció jelenlegi állapotának fájlba kimentése és betöltése.

Memória spórolás: ne tároljon minden egyes Allapotot egy veremben, hanem 5-10 műveletenként mentse csak el, visszalépéskor pedig az utolsónak eltároltótól hajtson végre több műveletet, hogy pontosan egy lépéssel legyen hátrébb.

A CD-lemez tartalma

„futtathato” mappa: a futtatható állományok és a nekik szükséges .dll-ek

„testfiles” mappa: teszt bemeneti fájlok

„forrasfajlok” mappa: forrásállományok

- „cpp” mappa: a forrásfájlok és a Makefile
- „headers” mappa: header állományok
- „elsoparse” mappa: az elsoparseParser előállításához szükséges források
- „interpret” mappa: az interpretParser előállításához szükséges források
 - ez utóbbi két mappában a „.l” kiterjesztés a flex bemenete, a „.y” kiterjesztés a bisonc++ bemenete, a „.h” és a „.ih” a bisonc++ által egyszer legenerált, később általam módosított header (és részleges implementáció) fájlok, a „.yy.cc” a flex által generált állomány, a „.cc” a bisonc++ által generált állomány

„dokumentacio” mappa: az alábbi dokumentáció, „.pdf” formátumban is és a felhasznált képek