

TDK-dolgozat

Horváth Norbert
Róth Arnold
Sajó Balázs

Debreceni Egyetem
Informatika Kar
Informatikai Rendszerek és Hálózatok Tanszék

x86-64 emulátor fejlesztés

Dr. Varga Imre
Egyetemi Docens

Horváth Norbert
Róth Arnold
Sajó Balázs
Mérnökinformatikus BSc 4. évfolyam

Debrecen, 2023 november

Tartalomjegyzék

Tartalomjegyzék	3
1 - Bevezetés.....	5
1.1 - Követelmények.....	6
1.2 - Konkurencia	7
1.3 - x86-64 architektúra történelem.....	8
1.4 - x86-64 felépítése	12
2 - A projekt felépítése	15
2.1 - Előfeldolgozás	15
2.1.1 - Forráskódok kezelése.....	15
2.1.2 - ELF bináris fájlok	16
2.1.3 - Futtatható állomány kezelése.....	17
2.1.4 - Objdump	18
2.1.5 - RAM fájl.....	19
2.2 - Emulátor architektúra	21
2.2.1 - Memóriakezelés	22
2.2.2 - Memóriaműveletek	24
2.2.3 - Az emulator működése	26
2.2.3.1 - Init	27
2.2.3.2 - Instruction fetch.....	29
2.2.3.3 - OperandFetch	30
2.2.3.3.1 - Konstansok.....	30
2.2.3.3.2 - Regiszterek.....	31
2.2.3.3.3 - Memóriaterületek	31
2.2.3.4 - Execute	32
2.2.3.5 - WriteBack.....	32
2.2.3.6 - Save state.....	33
2.2.3.7 - Fini	33
2.3 - Grafikus felhasználói felület.....	34
2.3.1 - A GUI-ról általában	35
2.3.2 - A felület felépítése	35
2.3.3 - További funkcionalitások.....	37
3 - Összefoglalás.....	39

3.1 - A fejlesztés felosztása.....	39
3.2 - Fejlesztésben felhasznált technológiák.....	40
3.3 - Továbbfejlesztési lehetőségek	41
Irodalomjegyzék	43
Ábrajegyzék.....	44
Függelék	45
Előfeldolgozás	45
RAM fájl szintaxis	45
Output fájl szintaxis	46
Támogatott assembly instrukciók	47

1 - Bevezetés

Egy mérnökinformatikus hallgatónak a tanulmányai során elengedhetetlen, hogy megismerkedjen a számítógép működésével, felépítésével. Előzetes tanulmányokból egy hallgató már megtanulhatta magasabb szinten, hogy milyen komponensekből épül fel egy ilyen rendszer, mint például processzor, memória, alaplapp stb., viszont egy mérnökinformatikusnak ismernie kell az ilyen rendszereket architektúráis szinten is: milyen architektúrát használ egy számítógép? Milyen egyéb architektúrák léteznek? Az egyes architektúrák hogyan épülnek fel, valamint hogyan működnek?

Manapság a számítógépekben az egyik leggyakrabban használt architektúra a 64 bites x86-64, avagy más nevén amd64-es architektúra. Különböző tantárgyak keretein belül elsajátíthatjuk, ahogyan az architektúra a hardveres szinten működik, későbbiekben elsajátíthatjuk, hogyan lehet ezen architektúrát szoftveresen is kihasználni az Assembly programozási nyelvet felhasználva. Assembly programozás órá keretein belül betekinthetünk, hogyan is lehet egy szoftvert megírni hardver közeli szinten, amely által felmerül az egyes utasítások után a hardver aktuális állapotának kérdésköre. Mikor alacsony szinten beszélünk kódról, olyankor előkerülnek az architektúrában az egyes memória területek, valamint az azokban tárolt értékek. A memóriaterületeken felül, ami talán még fontosabb felmerülő tényező az az egyes regiszterek, valamint azoknak állapot változásai. Sajnálatos módon, a regiszterekben és a memóriában tárolt értékeket közvetlenül kinyerni a program futása közben körülményes, esetenként lehetetlen.

Az egyes assembly programelemek (pl. utasítások, regiszterek, címzési módok stb.) előfordulási gyakoriságának elemzése [2] megmutatja például melyek a programkódokban leggyakrabban előforduló utasítások és azok hány százaléka igényel memóriahozzáférést. Ez azonban csak egyfajta statikus statisztika, amelynél sokkal nagyobb tudományos jelentősége van egy időfüggő tanulmánynak. Egy utasítás előfordulási száma a kódban nagyon eltérő lehet attól, hogy hányszor fog a valóságban végrehajtásra kerülni a program futása során. Gondoljunk csak az iterációkra, szubrutinhívásokra. A regiszterek tartalma gyorsabban elérhető, mint a memóriában tárolt változóké. Ha egy kódban csak 1-2 helyen hivatkozunk memóriaterületekre az nagyon hatékonynak tűnhet, de ha ez az 1-2 utasítás egy ciklusban szerepel, akkor a futásidő jelentős részében szükséges lesz a memória elérésére. Míg a kódban előfordulások megszámlálása viszonylag egyszerű fájlkezelési feladat, addig a dinamikus elemzés

futásidőben történő tanulmányozást követel meg. Ez viszont egy valós hardverben rettentően nehézkes. Egy szoftveresen emulált architektúra használata lehetővé teszi a program futása közben történő tudományos elemzéseket. Így vizsgálható többek között a programfordítás optimalizálásának hatékonysága vagy a regiszterkészlet kihasználtságának jellemzői, ami révén akár jobb futási teljesítmény elérését eredményező javaslatok is megfogalmazhatóak lesznek.

A dolgozatunk célja egy olyan eszköz, illetve program csomag előállítása, amely segítségével könnyen tudjuk vizsgálni az egyes programok futása közben a hardver állapotát. Ehhez egy emulátor szoftver készítése volt a cél, ami szoftveresen valósítja meg a hardver fontosabb tulajdonságait, mint például a regisztereket és a memóriát, amelyek állapotát program futtatás közben elérhetővé tesz a felhasználónak akár egyszerű szöveges fájlként, akár interaktív GUI-n keresztül. A megvalósítani kívánt emulátornak képesnek kell lennie egyszerűbb C nyelvű, assembly szintű és valós gépi kódú programok kezelésére, a grafikus felhasználói felületnek pedig az Intel szintaxisú assembly kódokat kell támogatnia.

1.1 - Követelmények

Mielőtt bele tudtunk volna kezdeni a projektbe, előtte szükséges volt lefektetnünk néhány célt, illetve követelményt, amelyeket szeretnénk volna elérni, valamint tartani a fejlesztés során.

Egy emulátor szoftvernél az első követelmény, amelyet le kell fektetni, hogy milyen architektúra emulálását végezze. Fentebb említettük, hogy a manapság egyik leggyakrabban használt architektúra az x86-64-es, valamint tanulmányaink során is ezzel ismerkedhetünk meg, így ez a döntés céljainkat tekintve egyértelmű volt.

Emulátor révén fontos, hogy a szoftverünk teljes mértékben a valós architektúrát emulálja. Ez alatt gondolhatunk a memória kezelésre, amelyet a szoftver végez, valamint a regiszter készletre, amelyeket imitál. Az emulátorunknál fontos, hogy teljesen valós memória címek kezelésére legyen képes, valamint fontos továbbá, hogy a regiszterek is a valódi architektúra szerint módosuljanak ahogyan az emulátor futtat egy adott programot.

Ha már megemlítettük a programok futtatását, itt megemlíthetjük a következő követelményt: az emulátorunk képes kell legyen olyan programok futtatására, amelyek már a piacon vannak. Itt főleg arra gondolunk, hogy az emulátor működéséhez ne legyen szükséges olyan kód írására, amely direkte az emulátort célozza meg.

Az emulátor fontos, hogy valamilyen formában eltárolja az egyes programok futásának információit. Az emulátornak a futása során elő kell állítania egy szöveges állományt ezekről az információkról, amelyet később egy másik program kutatási célzattal felhasználhat. Ilyen kutatásokra lehet példa, hogy egy adott program végrehajtása során egyes instrukciók, ciklusok, illetve hívási láncok hányszor fordulnak elő.

Fontos a program megvalósítása során, hogy fel lehessen használni kutatási célokra, viszont az elsődleges célja így is az oktatás elősegítése. Mivel a program főleg oktatási célzattal készül, így fontos, hogy ehhez biztosítsunk egy módszert, amely által egyszerű lesz vizsgálni az emulátor által végrehajtott utasítások működését. Ebből következik az utolsó követelmény, miszerint az emulátorhoz kell készüljön egy felhasználó barát grafikus felhasználói felület, amelyen a hallgatók, illetve az oktató egyszerűen meg tudja vizsgálni ahogyan a hardver állapota változik a szoftver futása közben, legyen szó ez alatt a regiszterekről, illetve a memóriáról.

1.2 - Konkurencia

Minden projekt kezdete előtt felmerül a kérdés; „Vajon létezik már hasonló megoldás a piacon?” Ezt a kérdést magunknak is fel kellett tegyük annak érdekében, hogy lássuk van-e esetleg már olyan termék, amelyet valamilyen módon fel tudunk használni saját céljaink eléréséhez, vagy teljesen a nulláról kell építkeznünk. Ez akkor lett volna igaz, amennyiben találtunk volna olyan megvalósítást, amelynek kódját kiegészítve jobbá tehattünk volna annak egészét. A kérdés megválaszolásához kutatómunkába kezdtünk, megvalósítások után kutatva.

Ezen munkálkodásunk során világossá vált, hogy habár a piacon már léteztek hasonló megoldások, azok sajnálatos módon még általunk kiegészítve sem feleltek volna meg összességükben elvárásainknak. Ilyenek voltak többek között az rkarris12 által megvalósított Assembly Emulator [3], a danrose499 által megvalósított x86 Assembly Emulator [4], a rafaelcarlos által megvalósított Assembly x86 Emulator [5], és a Unicorn Engine CPU emulátor keretrendszer [6]. Harris megvalósítása nem igazi memóriacímeket használt, emulátora nem volt képes igazi kódok (bináris/assembly) futtatására, emellett webalapú és AT&T szintaxisú volt. Dan kivitelezése habár Intel szintaxisú volt, csakis 8- és 16-bites értékekkel dolgozott és ezekkel tudott valós gépi kódokat futtatni, azonban Turbo Assembler mentén. Carlos emulátora túlzottan hardverközeli assembly-t értelmezett, kiegészítve képzeletbeli I/O eszközök

használatával, azonban Harris-hez hasonlóan webalapon, de 32 biten. A Unicorn Engine gépi kódot futtatott, amelyet assembly-re nehéz lett volna visszafejteni, így habár számunkra nem, de egyéb felhasználási esetekben ideális lehet. Sajnálattal tapasztaltuk továbbá, hogy sok megoldás nem nyílt forráskódú volt, így nem lehetett azok kódján módosítani elemzéseinkhez.

Ahogy azt a projektmunkánk elnevezése is sugallja, a valós hardver működésének legvalóságosabb imitálását tűztük ki célunk magunk elé. Ezért sem volt többek között elfogadható, hogy néhány fellelt megvalósítás nem valós memóriaarchitektúrát emulált, hanem egy saját igényeiket kielégítő tárolószervezetet. Emellett valahány projekt esetében nem a modern 64-bites architektúrára, hanem jobbra a manapság már elavultnak számító 32-bitesre építettek, vagy éppenséggel nem is Intel szintaxison alapultak ezen megvalósítások. Ezen tényezők összessége, valamint azon hiányosságok, hogy ezen működést imitáló programok némelyike képtelen volt futtatni valós kódokat, valamint sokszor nem rendelkeztek grafikus felülettel, arra sarkalltak minket, hogy egy saját megvalósításnak álljunk neki közösen.

A korábban felsorolt kontra érveket saját programunk esetében annak pro tulajdonságaivá kívántuk átforgatni. Végeredményben egy olyan programot alkottunk meg, amely natív módon képes Linux rendszerek alatt, a 64 bites architektúra emulálásán keresztül egyszerű programokat leíró valós kódokat futtatni, oly módon, hogy a valós memóriakezelésnek köszönhetően a felhasználó egy informatív grafikus felület segítségével képes lehessen elmélyülni, az általa vizsgálni kívánt programsorok futási sajátosságainak elemzésében. A rendelkezésre álló idő optimális strukturálása okán bizonyos komplexebb műveleteket, mint például a rendszerhívásokat és megszakításkezelést nem valósítottuk meg, hisz ezen műveletek helyes implementálásához a Linux kernelét hosszas kiismerés alá kellett volna vetnünk.

1.3 - x86-64 architektúra történelem

Kétség nem fér ahhoz, hogy az elmúlt fél évszázadban a technológia fejlődésben hatalmas ugrásokat figyelhettünk meg a tudományok legtöbb területén. A számítástechnika terén az elektroncsöveket kiváltó tranzistorok elterjedése elképesztő fejlődést hozott, hisz hatékonyságukban és kicsiny fizikai méretükben sokkalta túlteljesítették a feltalálók velük szemben felállított elvárásait. Ezzel a technológiai innovációval, valamint az előállítási módszerek finomodásával egyre több és több áramköri elemet voltak képesek a gyártók integrálni, egyre kisebb és kisebb méretű területen; létrejöttek az integrált áramkörök.

Gordon E. Moore nevéhez fűződik azon megfigyelés, miszerint az integrált áramkörök (IC) összetettsége a '70-es években még évente megduplázódott, manapság ez jobbra 2-3 évente történik meg. Az IC-k családjának idővel részesei lettek a mikroprocesszorok is, amelyek kifejlesztésének két úttörője, az Intel és az AMD az 1960-as évek legvégén kezdték meg pályafutásukat. A mikroprocesszorok a számítógép központi feldolgozó egységét (CPU) egy vagy legfeljebb néhány integrált áramkör formájában valósítják meg, amely elemek minden manapság használatos számítógép működésének alapját adják, a CPU-t körülölelő hardverrel.

Az első mikroprocesszort az Intel mutatta be 1971-ben, amely az Intel 4004 nevet viselte, amely központi feldolgozó egység 4 bites volt, órajele pedig 750kHz körül tetőzött. Ekkortájt még nem éleződött ki az AMD és az Intel közötti konkurencia, így inkább egymás újításaiból táplálkozva haladtak fokozatosan az évek múlásával a 8 bites, illetve a 16 bites processzorok kifejlesztésének nyomán. 1978-ra az Intel 8086-os processzora, már 12-szer akkora integritást és órajelet tudhatott magáénak, mint az eredeti 4004-es, amelynek alapján igazolható Moore-törvénye is, hisz megfigyelhető egyfajta évenként másfélszeres komplexitás-növekedés.

A nagyobb processzorgyártóknak szükséges volt olyan architektúrákat kialakítaniuk az önmagukban csak szilíciumlapjának tekinthető processzorokra, amelyekkel megfelelően vezényelhetők azok működését felhasználástól függően. Megannyi processzorarchitektúra létezik, azonban ezeket két fő részre szokás osztani; összetett utasításkészletű- (CISC) és redukált utasításkészletű (RISC) processzorokra. Az összetett utasításkészlet esetén az utasítások bonyolultabbak a mikroprogramozott vezérlés okán, míg a redukált utasításkészlet esetén inkább a teljesítmény kerül előtérbe. TDK dolgozatunk szempontjából a CISC variánsról érdemes szót ejtenünk, hisz az x86-64 architektúra is ezen megközelítésen alapszik. Ezen típusú architektúránál összetett utasítások sokasága áll rendelkezésre a programozók számára, amelyek bonyolultabb címezési módokat, változó hosszúságú instrukciókat (ez nehezíti a pipeline megvalósítást) és kisebb regiszterkészletet vonnak maguk után. Az assembly programozás mindezek okán könnyebbé válik, hisz a magasabb órajelciklus-igényű összetett instrukciók egy szuszra oldhatnak meg komplexebb műveleteket a másodperc tört része alatt.

A korábban említett Intel 8086-os utódját az Intel 1985 végén dobta piacra Intel 80386 [13] megnevezéssel (lásd az 1. ábrán) , amely hosszú időközön át adta a számítógépek döntő



1 Az Intel 80386 processzor

Forrás: Wikipédia

többségének alapját, amely az imént taglalt CISC megközelítésre épített. Ezen processzor architektúrája esetén a regiszterek mérete, a legnagyobb felhasználható adategységek hossza, illetve a memóriacímek hossza is egyaránt 32-bites volt, emellett az előző generációhoz képest 16 instrukcióval bővült az utasítások száma, abból az okból kifolyólag, hogy a regiszterméreteket, a címezési és adattárolási módok sok esetben konvertálást igényeltek. Az architektúra vezető mérnöke John H. Crawford volt, akit a korábbi Intel 286-os processzor

architektúrájának fejlesztésével bíztak meg, ennek sikeressége okán ő vezethette az i386 mikrokódját programozó csapat munkáját a későbbiekben.

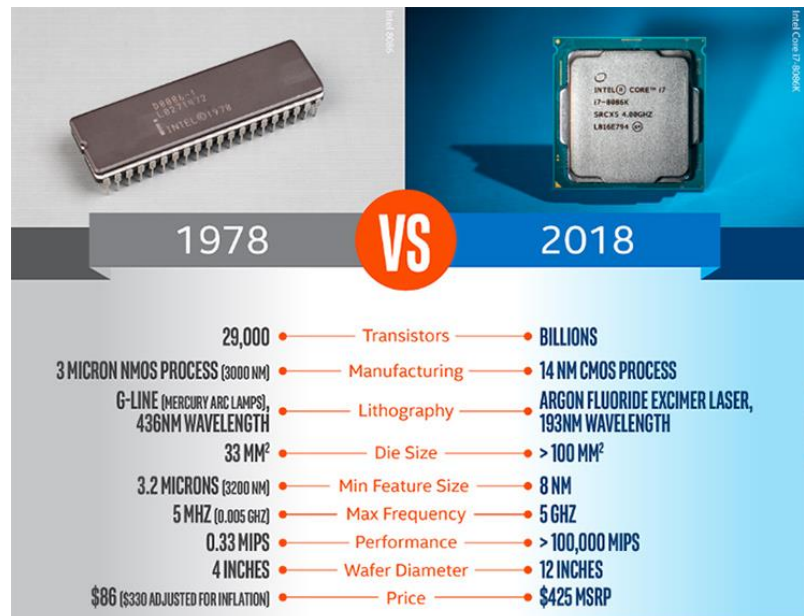
Nem tévedés azt állítani, hogy ezen architektúra egy jó darabig bőven megfelelt a legtöbb alkalmazási módnak, elvégre a 80-as évek közepén még gigantikusnak hatott a 4 gigabájt maximálisan kezelhető memóriamennyiség, hisz ekkoriban a legtöbb számítógép még csak pár megabájt rendszermemóriával rendelkezett. Azonban a fejlődés nem állt meg, hisz a 90-es évek közepétől kezdett elégtelenné válni a korábbi limit (főleg nagyvállalati környezetben), így megkezdődtek a fejlesztések a nagyobb memóriakapacitás kezelésének elérése érdekében.

Az ezredforduló közeledtével az Intel és az AMD továbbra is az x86 architektúrát alkották meg processzorcsaládok formájában egymással konkurens módon. Idővel azonban az AMD az i386 architektúrát továbbfejlesztve 1999-ben a nyilvánosság elé hozta az újdonsült architektúráját az x86-64-est [14]. Habár az AMD alkotta meg, az Intel és a HP magáénak tudta be az innovációt, kizárva ezzel az AMD-t a közvetlen sikerből. A legelső AMD64, avagy x86-64 architektúrára épülő processzort az AMD dobta piacra 2003 végén, amely az AMD Opteron elnevezést kapta.

Ebben az architektúrában már a 32-bites megvalósítást felváltó 64-bites kivitelezés nyert teret magának. Ezen újítással a processzor által logikailag kezelhető fizikai memória drasztikusan megnőtt, míg az ebben tárolható legnagyobb adategységek mérete is megduplázódott, s a regiszterek mennyisége és mérete is a duplájára nőtt. Természetesen itt is gondoltak a visszafelé kompatibilitásra, így egészen a 16 bites Intel 286-ra íródott programkódok magasabb hatékonysággal, míg az újonnan íródott x86-64 programok az új architektúra előnyeit kihasználva futhatnak a gazdagépeken. Ezen komplexitás-növekedés sajátosságainak nyomon követéséhez jöhet segítségül az általunk megalkotott x86-64 Assembly Emulátor.

Manapság már az x86-64 architektúrán alapuló

számítógépek minden háztartásban, munkahelyen és egyéb felhasználási területeken is fellelhetőek, legyen az hordozható, asztali, vagy szerver formátumban. Habár Moore-törvénye már nem teljesen áll helyt az IC-k komplexitásával kapcsolatban, túlzottan komplikált összehasonlítások nélkül már képtelenek vagyunk kifejezni a technológiai fejlődés mértékét a '70-es évekhez mérten. A 2. ábrán az Intel 8086-os processzora kerül összevetésre a cég 40. jubileumára kiadott az Intel i7-8086K processzorával, amely ábrán könnyen láthatók a drasztikus különbségek a két CPU között. Ahhoz, hogy a jövő programozói, illetve a jelenleg már programozással foglalkozó szakemberek minél hatékonyabb és a felhasználási területnek legmegfelelőbb szoftvereket írassanak meg, szükséges pontosan megérteniük azt, hogy egy adott utasítás milyen hatással van a mikroprocesszorra, és az azt körülölelő megannyi funkcionális elemre. Jelenleg még a kutatók nem látnak arra nyomós indokot, hogy 128 bitre fejlesszék át a jelenlegi x86-64 architektúrát, így még bizalommal vághat bele bárki a jelenleg az ARM architektúrák mellett trónoló processzorarchitektúra megismerésébe és megértésébe.



2 Az Intel 80386 és i7-8086K összehasonlítása

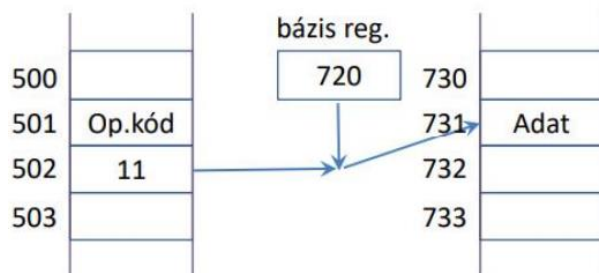
Forrás: HotHardware

1.4 - x86-64 felépítése

Ahhoz, hogy az olvasó számára világossá váljon, hogy mitől is egyedi az x86-64 architektúra, érdemes kielemezni sajátosságait, és hogy pontosan hogyan is épül fel. A legnagyobb áttörést az előző x86 architektúrához képest a címzési tér méretében érték el a fejlesztők, hiszen egy 64-bites módban futó program képes lehet megcímezni egy akár 2^{64} bájtnyi méretű területet (ez a 18-at követő 18 darab '0'), amely fizikai memória tekintetében 2^{52} bájtnyi területnek feleltethető meg. Ily módon 4 PB (4 millió gigabájt) memóriamennyiség volna megcímezhető, azonban a legmodernebb processzorok is csak pár TB (pár ezer gigabájt) értéket fednek le.

Az architektúra regiszterkészletével folytatva szükséges tisztázni azt, hogy a regiszter egy írható-olvasható, általában egyszerre csak 1 gépi szó tárolására alkalmas gyors tárolóegység a processzoron belül. A gyártási sajátosságok és a mikrokód együttes következményeként ilyesféle regiszterekből 16 darab áll rendelkezésre általános felhasználásra (Rx, illetve Rxx megnevezéssel), amelyek mindegyike 64-bit hosszúságú, de támogatják a bájt, szó, duplaszó és négyszó hosszúságú egészen végzett műveleteket is. Az általános felhasználású regisztereken felül megtalálhatók további 64-bit hosszúságú „mutató” regiszterek is, így például az utasításmutató regiszter (RIP), amely az éppen futó utasítás memóriacímére hivatott mutatni. Egymással szoros összefüggésben létezik a veremmutató-regiszter (RSP), amely a legutoljára a memóriában eltárolt adat memóriacímére mutat, és a bázismutató-regiszter (RBP), amely a függvényparaméterek és lokális változók elérését biztosítja, egy a veremmutatón alapuló offset eltárolásával. Szintén az architektúra szerves részét képezik a forrás- és célindex regiszterek (RSI és RDI), amelyek sorrendhelyesen az első, illetve a második argumentumok továbbítására használatosak függvényhívásokkor. Az állapotregiszterek (RFLAGS, korábban EFLAGS) a korábban végbement utasítás után létrejött „állapotot” hivatottak tükrözni, mint például, ha az utasítás eredménye nulla volt (ZF), vagy éppen negatív (SF), esetleg túlcsordulás (OF) történt-e, vagy carry (CF) képződött, azonban mindezen állapotokat külön szegmensekben tárolva, amely értékek a feltételes végrehajtások esetén mérvadók. Emulátorunkba nem foglaltuk bele a modern processzorokban megtalálható SSE és AVX kiterjesztéseket, így a változókkal megvalósított regiszterblokkunk ezen 16-16 SIMD (Single Instruction, Multiple Data) utasításokhoz használatos extra adatregiszterekkel nem rendelkezik [7].

A regisztereket amiatt volt szükséges részleteiben ismertetni, hisz az architektúra működése



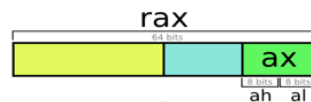
4 A bázis relatív címzés

Forrás: V. Imre Számítógépes Architektúrák diasor

során a korábban ismertetett rendeltetésüknek megfelelően részt vesznek a futás folyamatában. Itt érdemes megemlíteni a címzési módokat, valamint, hogy bizonyos utasítások hogyan vezényelhetők a négyszónál kisebb egész értékeken végzendő műveletekre. Ahogy azt a **Hiba! A h**

ivatkozási forrás nem található.. ábrán is nyomon követhetjük, kiválasztottunk egyetlen utasítást az x86-64 architektúrában fellelhető 981 közül, amelyet az Assembly programozási nyelv formai követelményeinek megfelelően prezentálunk [11]. Az utasítások egyetlen kódszóból (mnemonik) és egy vagy több operandusból épülnek fel. Az **Hiba! A hivatkozási f**
orrás nem található.. ábrán látható csoportosítás a címzési módokat hivatott bemutatni, amelyek segítségével az utasításban szereplő operandusokat a különböző tárolókból (regiszter, változó és konstans) mozgósíthatunk. Amennyiben egy ilyen utasítás folyamán csak egy adott szelvényét szeretnénk elérni a forrás- vagy éppen céloperandusnak, abban az esetben ezekre a

Regiszteres	<code>mov ax, bx</code>	Bázis+index relatív	<code>mov ax, [bx + di + 123]</code>
Közvetlen adat	<code>mov ax, 7Fh</code>	Skálázott	<code>mov ax, [si * 4 + 10h]</code>
Közvetlen cím	<code>mov ax, [1234]</code>	Szegmens előtag	<code>mov ax, [es:bx]</code>
Közvetett regiszteres	<code>mov ax, [bx]</code>		
Regiszter relatív	<code>mov ax, [bx + 100h]</code>		



3 A címzési módok és a regiszterek alregiszterei

Források: V. Imre Számítógépes architektúrák diasor, GSN Wordpress

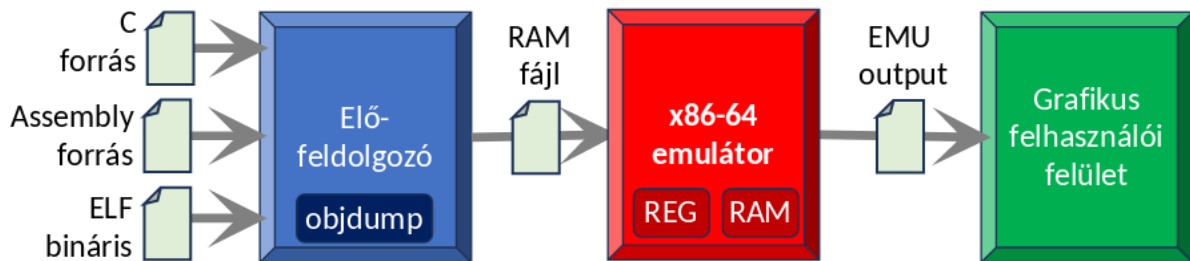
Hiba! A hivatkozási forrás nem található.. ábra jobb alsó sarkában lévő módokon hivatkozhatunk. Az RAX megnevezéssel a teljes 64 bitet címezzük, az EAX segítségével az alsó 32 bitet, az AX segítségével az alsó 16 bitet, míg az AH és AL szelektorokkal rendre a felső és alsó 8 bitet. Könnyen belátható tehát, hogy a visszafelé kompatibilitás szempontjából ezen szelektorok megőrizték eredeti megnevezésüket, ezért is vagyunk képesek 32-bites programokat 64-bites rendszereken futtatni. A címzési módok közül a legtriviálisabb megoldások közé sorolhatjuk a regiszteres, közvetlen adat, -cím és -regiszteres megvalósításokat, míg bázis relatív címzés esetén a **Hiba! A hivatkozási forrás nem található..** ábrán látható módon bonyolódik a megvalósítás. Míg a közvetlen címzési módnál nincs túlzott számolgatás az operandus helyét illetően, addig a bázis relatív esetben úgy kapjuk meg a keresett operandus címét, ha az eltolási értékhez hozzáadjuk a bázis regiszterben található

kezdőcímet. Itt érdemes visszaemlegetni a bázis- veremmutató regiszterek kapcsolatát, hisz így ezen érték származtatott értéknek tekinthető, amely korábban számítási kapacitást igényelt.

A korábban említett megközelítőleg 3600 alvariánssal rendelkező x86-64 utasítások közül az ugró-, forgató-, bitmanipuláló-, logikai-, adatmozgató- és az aritmetikai műveleteket megvalósító utasításokat implementáltuk saját architektúránk esetében, amelyek természetesen ugyanúgy megtalálhatók az eredeti architektúrában is, további közel 900 utasítás mellett. Gondolataink végére pontot téve, összességében az x86-64 architektúra egy robosztus találmány, amely regiszterek miriádjával szolgál, s a komplex címzési módok segítségével, valamint az utasítások regimentjével valósítják meg azt a működést, amely a legtöbb általunk használt számítógép működését adja, legyen az asztali vagy hordozható formában kivitelezve.

2 - A projekt felépítése

Az általunk megvalósított programcsomag három főbb komponensre bontható:



5 A projekt komponensei

A csomagunk első komponense az előfeldolgozó. Itt történik az általunk támogatni kívánt bemenetek előfeldolgozása, valamint az emulátor működéséhez szükséges bemenet előállítása.

A következő komponens, amely talán a legkomplikáltabb az összes közül az nem más, mint maga az emulátor. Programcsomagunk ezen komponense az előfeldolgozótól kapott bemenet alapján fogja lefuttatni a programunkat, és annak futásának részleteit kiírni egy kimeneti fájlba.

A csomag utolsó komponense egy különálló grafikus felhasználoi felület. Alapvetően az emulátor kimenetét önmagában felhasználhatjuk kutatási célokra, viszont oktatási célzattal egy ezt feldolgozó grafikus felhasználoi felületet is kínál a csomagunk.

A komponensekről részletesebb leírást a további fejezetekben adunk.

2.1 - Előfeldolgozás

A követelmények egyike, amelyet lefektettünk kijelenti, hogy az emulátor már létező kódok futtatására legyen képes, ez alatt érthetünk kész futtatható állományokat, valamint C, illetve Assembly programokat. Külön-külön támogatni mind a három opciót alapvetően nem érdemes, mivel ez rengeteg extra kód megírásához vezet, amelyet utána, ha az emulátorunk logikája változik sokkal nehezebb karbantartani. Ennek tudatában érdemes találni a három támogatni kívánt típus között egy közös nevezőt, amelyet egymagában képes támogatni az emulátorunk.

2.1.1 - Forráskódok kezelése

Az említett három fájltypus között van egy számunkra használható közös nevező: forráskód fájlokat át tudunk fordítani futtatható állománnyá.

Mikor átadunk egy fájlt az előfeldolgozó egységnek, az először megvizsgálja, hogy a három támogatott fájltypus közül melyiket adtuk meg neki. Az előfeldolgozó végrehajt egy elővizsgálatot a fájl kiterjesztésére, hogy az `.c` (C kódok esetében), vagy `.s` (Assembly kódok) kiterjesztésű-e. Amennyiben ez nem teljesül, az előfeldolgozó kiír egy informatív hibaüzenet és leáll. Ha a vizsgálat sikeres volt, akkor megpróbálja a kódot lefordítani futtatható állományra. Ehhez az előfeldolgozó segítségül hívja a Linux rendszerek alatt megtalálható GCC nevezetű programot.

A GCC (**G**NU **C**ompiler **C**ollection) eredetileg a GNU operációs rendszer fordításához készült, manapság viszont egy cross-platform szoftveres csomag, amelyben megtalálhatóak több programozás nyelvhez szükséges fordítók. Számunkra ezek közül a legfontosabb, hogy ez az egy eszköz képes lefordítani C, illetve Assembly kódot egyaránt.

Az előfeldolgozónk a `compile` lépés folyamán a GCC parancs meghívásával előállítja a számunkra szükséges futtatható állományt, amelyet alapvetően a programunk a `/tmp/asemu_executable` útvonalon tesz elérhetővé. Az általunk használt GCC parancs szintaxisok a függelékben megtalálhatóak, illetve a GCC további lehetséges paramétereiről olvashatunk annak Linux Manual Page-ében [8].

2.1.2 - ELF bináris fájlok

Mielőtt rátérhetnénk, hogy hogyan kezeli az előfeldolgozó a futtatható állományokat, előtte érdemes kitérni, hogy Linux alatt egy futtatható állomány hogyan is épül fel. Linux, illetve egyéb Unix-like rendszerek alatt a futtatható bináris fájlok az úgynevezett ELF (**E**xecutable and **L**inkable **F**ormat) szabvány szerint írhatóak le, amely egy cross-platform fájlformátum leírás futtatható állományok, object code illetve shared library fájlok számára. A projekt megvalósítása érdekében az ELF formátumon belül két szekciót kell kiemelni: az ELF Header-t, illetve a Szegmenseket.

ELF Header
Program Header Table
<i>Szegmens</i>
<i>Szegmens</i>
...
<i>Szegmens</i>
Section Header Table

6 ELF fájl elrendezése

Az első 64 bájtja a fájlnak (56 bájt 32 bites futtatható állományok esetében) az úgynevezett ELF Header. Itt a fájl kezeléséhez szükséges információkat találhatunk meg. Példa ilyenekre, hogy az adott állomány 32 vagy 64 bites-e, little endian vagy big endian elrendezést használnak-e a bájtok, mi az állomány cél operációs rendszere, cél processzor architektúrája. Az itt található

első 4 bájt az úgynevezett “magic number” ($0x7F454C46$, avagy `.ELF` ASCII formátumban, ahogyan azt a Wikipedia oldalán is olvashatjuk [12]), amely jelzi, hogy az adott fájl ELF formátumot használ-e.

A szegmensek azon rész, ahol maga a program adatai helyezkednek el, legyen az például a gépi kódú utasítások, illetve a konstans értékek. Számunkra négy szegmens érdekes a projekt szempontjából. Az első a `.text` szegmens, amely ahogyan az amerikai Yale Egyetem Executable and Linkable Format dokumentumában is olvashatjuk [15], a futtatható állomány gépi kódú utasításait tartalmazza. A következő a `.data` szegmens, amely az automatikusan inicializált változó deklarációkat tartalmazza, mint például a globális változókat. Az `.rodata` szegmens tárolja a csak olvasni kívánt konstans értékeket, például a sztringeket. Az utolsó érdekelt szegmens a `.bss`, amely az inicializálatlan változókat írja le.

Az említett szegmensek helyét, nevét, méretét, illetve egyéb adatokat a Program Header Table és a Section Header Table ír le az ELF formátum szerint.

2.1.3 - Futtatható állomány kezelése

Abban az esetben, ha az előfeldolgozó forráskód helyett már kész futtatható állományt kap meg bemenetként, olyankor is fontos, hogy előre megvizsgáljuk, hogy a fájl megfelelő-e. A fentebb említett ELF Header szekcióból négy értéket kell megvizsgálnunk, hogy megbizonyosodjunk róla megfelelő-e a fájl.

A legelső és egyben legtriviálisabb vizsgálat, hogy leellenőrizzük, hogy a “magic number” megtalálható-e.

A következő, amit meg kell vizsgálnunk, a cél operációs rendszer. Ehhez a $0x07$ offset-tel megtalálható bájtot kell megvizsgálnunk. Mivel a program Linux-os futtatható állományok kezelésére szolgál, ezért ennek az értéknek $0x03$ -nak kell lennie.

A $0x10$ -es offset-en található két bájt mondja meg az objektum fájl típusát. Ha az itt található érték $0x0002$, akkor tudjuk biztosra, hogy a fájl egy futtatható állomány.

Az utolsó, amit ellenőriznünk kell a $0x12$ -es offset-en megtalálható két bájt, amely a cél processzor architektúrát mondja meg. Mivel az emulátorunk az x86-64-es architektúrára alapoz, ezért nekünk ezen érték $0x003E$ kell legyen.

Ha további vizsgálatokat szeretnénk végezni az ELF Header-ön, az Executable and Linkable Format Wikipedia oldalon [12] egyszerűen és gyorsan meg tudjuk tekinteni a lehetséges értékeket, valamint azoknak helyét. Ha az ELF-ről nagyobb részletességgel kívánunk olvasni akkor a Yale Egyetem Executable and Linkable Format dokumentumában [15] megtehetjük.

Miután sikeresen megvizsgáltuk, hogy a bemeneti fájl megfelelő-e előjön egy újabb kihívás: hogyan szedjük ki a szükséges instrukciókat a bináris fájlból?

2.1.4 - Objdump

A projekt megvalósításához a GNU csomagból segítségül tudunk hívni egy újabb programot, amely az objdump. Az objdump az említett ELF formátumú objektum fájlokban tárolt információk megjelenítésére szolgál. Számunkra a legfontosabb funkciója, hogy képes a fájlban a Section Header Table által definiált egyes szegmensek gépi kódját visszafejteni az általunk megadott szintaxisú Assembly kódra. Esetünkben az Intel szintaxisra lesz szükség. A kimenetben az egyes szegmensek alatt található gépi kódok, a visszafejtett Assembly, valamint a kódok memória címe elkülönítve jelennek meg a kimenetben. Az előfeldolgozónk a decompile lépésben meghívja ezt a parancsot, és a kimenetét a /tmp/asemu_dump elérési úton teszi elérhetővé. A meghívott parancs szintaxisa megtalálható a függelékben, valamint a további lehetőségekről olvashatunk az objdump Linux Manual Page-ben [9].

```
244 | 400543: 00 04 00          add    BYTE PTR [rax+rax*1],al
245 | ...
246 |
247 | Disassembly of section .init:
248 |
249 | 0000000000401000 <_init>:
250 | 401000: f3 0f 1e fa          endbr64
251 | 401004: 48 83 ec 08          sub    rsp,0x8
252 | 401008: 48 8b 05 c9 2f 00 00  mov    rax,QWORD PTR [rip+0x2fc9]    # 403fd8 <__gmon_start__@Base>
253 | 40100f: 48 85 c0              test   rax,rax
254 | 401012: 74 02                je     401016 <_init+0x16>
255 | 401014: ff d0                call   rax
256 | 401016: 48 83 c4 08          add    rsp,0x8
257 | 40101a: c3                   ret
258 |
259 | Disassembly of section .text:
260 |
261 | 0000000000401020 <_start>:
262 | 401020: f3 0f 1e fa          endbr64
263 | 401024: 31 ed                xor    ebp,ebp
264 | 401026: 49 89 d1              mov    r9,rdx
265 | 401029: 5e                    pop    rsi
```

7 Objdump kimenete

Ezzel a kimenettel már el tudunk indulni, viszont felmerül egy újabb probléma: az objdump kimenete alapvetően nehezen feldolgozható. Nehézségek alatt gondolunk például arra, hogy előfordulhat egyes instrukcióknál kommenteket rak az objdump a sorok végére. További példa, hogy az olyan szegmenseknél, amelyek nem kódot tartalmaznak az objdump “túl jól működik”,

mivel az ott tárolt bájtokat is visszafejti Assembly kódra. Ezen felül további probléma az adattároló szegmenseknél, ha ismétlődő adatot talál, azokat nem mindig írja ki, hanem három darab ponttal helyettesíti.

Az emulátor működésének fejlesztését megnehezíti, ha ezeket a hibákat program futtatás közben kellene elhárítani, így mielőtt ezt a kimenetet átadnánk az emulátornak, érdemes egy előfeldolgozást végrehajtani rajta, amelyben megtisztítjuk az imént említett hibáktól.

2.1.5 - RAM fájl

Az előfeldolgozás utolsó lépéseként készül el az általunk memória fájlnak, illetve RAM fájlnak elnevezett megtisztított objdump kimenet.

```
tmp7 = asemu_ram
1  main: 401106
2  .text
3  401106 mov eax 0x3 _ b803000000 mov eax,0x3
4  40110b mov ebx 0x4 _ bb04000000 mov ebx,0x4
5  401110 add eax ebx _ 01d8 add eax,ebx
6  401112 mov eax 0x100 _ b800010000 mov eax,0x100
7  401117 push rax _ _ 50 push rax
8  401118 add eax 0x1 _ 83c001 add eax,0x1
9  40111b push rax _ _ 50 push rax
10 40111c add eax 0x18 _ 83c018 add eax,0x18
11 40111f push rax _ _ 50 push rax
12 401120 pop rcx _ _ 59 pop rcx
13 401121 pop rcx _ _ 59 pop rcx
14 401122 pop rcx _ _ 59 pop rcx
15 401123 ret _ _ _ c3 ret
16 .rodata
17 402000 01
18 402001 00
19 402002 02
20 402003 00
21 .data
22 404000 00
23 404001 00
24 404002 00
25 404003 00
26 404004 00
27 404005 00
```

8 RAM fájl

A RAM fájlba az objdump kimenetéből már csak az számunkra fontos négy szegmens kerül be az ELF fájlból. Megvalósítást tekintve a fájlt két nagyobb logikai részre tudjuk felosztani.

Az első szekcióját a fájlunk fejlesztés folyamán kód szegmensnek neveztük. Itt azokat a szegmenseket kezeltük le, amelyek végrehajtható Assembly instrukciókat tartalmaznak (esetünkben a .text szegmens). A RAM fájlba ezeknél a szegmenseknél négy dolgot szeretnénk beírni egy sorba: a memória címét az utasításnak, az Assembly kódot leegyszerűsítve, a gépi kódot, illetve az eredeti megtisztított Assembly kódot. Az Assembly kód megtisztítása során, a

sorok végén lévő esetleges kommenteket, illetve a mnemonikok és operandusok elején található esetleges prefixeket távolítjuk el. Erre, mint példa az alábbiakban található sort tisztítjuk meg:

```
add rax,es:QWORD PTR [rip+0x02] <_init+0x32>
```

Ebből a sorból lesz az alábbi végeredmény:

```
add rax,QWORD PTR [rip+0x2]
```

Az egyszerűsített Assembly kódot a letisztított kódból állítjuk elő. Ebben az esetben a felosztást úgy kezeljük, hogy egy mnemonik van, illetve három operandus. Ha az egyes instrukciók kevesebb operandust tartalmaznak, akkor a hiányzóak helyére egy alsóvonást teszünk. Erre példa:

```
cmp rax,rcx
```

Amelyből ez lesz:

```
cmp rax rcx _
```

Ha valamelyik operandus egy memória címre mutatott érték, akkor az érték hosszát lerövidítjük annak kezdőbetűjére az egyszerűbb kezelhetőség érdekében.

```
mov rdx,QWORD PTR [0x33]
```

Itt a végeredmény így fog kinézni:

```
mov rdx Q[0x33] _
```

A gépi kód tisztítása talán az egyik legegyszerűbb, mivel itt csak a whitespace karaktereket távolítjuk el az egyes bájtok közül.

A .text szegmensnél fontos megtalálni a program belépési pontjának memória címét, hogy az emulátor tudja honnan is kell futtatnia a megadott kódot. Ezt az objdump kimenetében `main` label-lel ellátva találhatjuk meg, és miután megtaláltuk, a RAM fájl tetejére elhelyezzük.

A RAM fájl második nagyobb logikai szekcióját data szekciónak nevezzük. Itt kezeljük az olyan szegmenseket az ELF fájlból, amelyek csak adatokat szolgáltatnak (`.bss`, `.data`, `.rodata`). Ezeknél a szegmenseknél nekünk csak a memória címekre, illetve a tárolt adatra volt szükségünk, így a visszafejtett Assembly kódokat figyelmen kívül hagytuk. Az objdump mivel ezeket is igyekszik visszafejteni Assembly-re sok esetben több bájtot is berak egy sorba.

Mi az egyszerűbb kezelés érdekében ezeket a bájtokat a RAM fájlban szeparáljuk külön sorokba azok memóriacímével. Vannak olyan esetek ezeknél a szegmenseknél, mikor egyes sorokba csak három darab pontot ír az objdump, ilyenkor, ha az két olyan sor között helyezkedik el, amelynek van memória címe, akkor a két cím között lévő teret 0 értékekkel töltjük fel a RAM fájlban. Ha a vég memória cím kisebb, mint a kezdeti, illetve mikor nem találunk második memória címet, ami alapján kiindulhatunk, akkor az ilyen esetekben 100 darab 0 értéket helyezünk el a RAM fájlban.

A RAM fájlt az előfeldolgozó a /tmp/asemu_ram elérési úton teszi elérhetővé. A szintaxis leírása a RAM fájl egyes szegmenseinek megtalálható a függelékben.

2.2 - Emulátor architektúra

Emulátorunk felépítése egy átlagos Intel x86-64 processzor szerkezetéhez hasonlít. Kivételt képez a memóriakezelés, de ezt a későbbiekben fogjuk kifejteni. A regiszterkészlet teljes mértékben megegyezik azzal, amilyennel egy valós processzor rendelkezik, ezeket 64 bites egész típusú globális változókkal valósítottuk meg.

RAX	64 bites egész típusú, általános célú A regiszter
RBX	64 bites egész típusú, általános célú B regiszter
RCX	64 bites egész típusú, általános célú C regiszter
RDX	64 bites egész típusú, általános célú D regiszter
RSI	64 bites egész típusú, forrás index regiszter
RDI	64 bites egész típusú, cél index regiszter
RSP	64 bites egész típusú, verem mutató regiszter
RBP	64 bites egész típusú, bázis mutató regiszter
R8 - R15	64 bites egész típusú, általános célú 8-15 regiszterek
RIP	64 bites egész típusú, program számláló regiszter

Egyedüli különbség, az RFLAGS, másnéven státuszregiszter felépítése. A státuszregiszter teljes tartalmára nem volt szükségünk, ezért a számunkra fontos biteket külön változóként definiáltuk, a későbbi hivatkozások megkönnyítése miatt. Ezek a változók 32 bites egész típusúak.

OF túlcsordulást jelző bit

CF átvitelt jelző bit

SF előjel bit

ZF zéró flag bit

Architektúránk rendelkezik egy aritmetikai és logikai egységgel (ALU), amely felépül kettő darab 64 bites bemeneti változóból, amely az `ALUin1` és `ALUin2` nevet kapták, és egy kimeneti változóból, amely az `ALUout` elnevezést viseli, és szintén 64 bites. Az aritmetikai és logikai egységhez tartozik továbbá két darab maszkként használatos 64 bites változó, `OpMaskDest`, amely a céloperandusra vonatkozó maszk, és `OpSourceDest`, ami a forrásoperandusra vonatkozik. A maszkolás szerepe és fontossága a későbbiekben lesz ismertetve.

Definiáltunk két struktúrát, amely a memóriánkat fogja reprezentálni. A hagyományostól eltérő módon, külön kezeljük az utasítás, és külön az adat memóriát, amelyeknek emiatt külön struktúrára is volt szükségük.

2.2.1 - Memóriakezelés

A Neumann-elvű számítógépeknek három alapvető funkcionális egységből kell állniuk; központi egységből, háttértárakból és perifériákból. Összességében ezen komponensek együttes működése adja a mai modern számítógépek működését is. A központi egység az aritmetikai-logikai egység, a regiszterblokk, a gyorsítómemória, a matematikai társprocesszor és az operatív tár összessége. Az operatív tár a Neumann-architektúra szempontjából a program- és adatmemóriát egyetlen egységként kezeli, szemben a Harvard-architektúrával, ahol ezen két részegység különbözik egymástól. Programunk emulátor mivolta okán elengedhetetlen volt egy az eredetihez hasonló memóriaarchitektúra megalkotása, amely képes volt imitálni a valós memóriaműködést a lehető legvalóságosabb módon. Sokunk bizonyára a számítógép memóriáját egyetlen hosszú, egymást követő kisebb tárterületek együtteseként képzelel, azonban valójában ennél komplexebb megoldásra gondoltak a feltalálók.

A RAM, vagyis a véletlen elérésű memória azon tulajdonsága után kapta elnevezését, miszerint bármely memóriaterületét azonos idő alatt képes elérni a számítógép, amennyiben ezen területről olvasni, vagy éppen ezen területre írni készülne. Könnyű így belátni, hogy minekután nincsenek elérési időbeli különbségek a rácsszerű memóriamegvalósítás okán, így habár az adatok fizikailag távol lehetnek jelen egymástól elektronikus jelek formájában, a logikai

```

32
33  /* RAM management */
34  typedef struct MemDataUnit{
35      long long int addr;
36      unsigned char byte;
37      struct MemDataUnit *next;
38  } MemDataUnit;
39
40  typedef struct MemInstUnit{
41      long long int addr;
42      char mnemonic[32];
43      char operand1[32];
44      char operand2[32];
45      char assembly[256];
46      unsigned char machinecode[32];
47      int instlength;
48      struct MemInstUnit *next;
49  } MemInstUnit;
50
51  MemDataUnit *MemData;
52  MemInstUnit *MemInst;
53

```

9 Az adat- és utasításmemória linkelemei,
ezen láncolt listák fejmutatói

összeköttetéseknek köszönhetően ezen fizikai távolságokat semmisnek tekinthetjük.

Emulátorunk egy a fix tömbök gondolatával ellentétes láncolt listák formájában valósítja meg az úgynevezett operatív tárat. Ezen felül oktatóprogram mivoltából az operatív tár korábban említett két tárterületét külön-külön láncolt lista segítségével kezeli annak érdekében, hogy leendő a hallgatók könnyebben értelmezhessék a memória működését. A szóban forgó láncolt listák egyike az úgynevezett utasításmemóriát testesíti meg, míg a másik az adatmemóriát (lásd 9. ábrán látható kódrészlet). A

láncolt lista egy olyan a felhasználó igényeinek legjobban

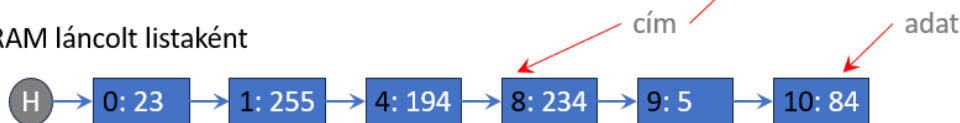
megfelelő struktúrákból álló láncolat, amely a tömbökhöz hasonló logikailag egységesnek tekinthető tárterületet valósít meg. Ezen láncolat elemei között a fentebb már említett logikai összeköttetéseket mutatók segítségével valósítjuk meg. Míg a tömbök esetében indexelt elemekkel találkozhatunk, addig a láncolt listák esetében a következő elemre mutató mutatókkal navigálhatunk szekvenciális módon a „tömb” elemei között. Minekután minden elem saját „memóriacímmel” rendelkezik, így a láncolt listában egymást követő elemek ezen címekre mutatnak rá a megelőző elemek mutatói. Ahogy azt a 9. ábra is jól mutatja, az adatmemóriát megvalósító láncolt lista elemeknek egyetlen 1 bájtnyi adatot tároló mezője van, míg az utasításmemóriát megvalósító struktúrában a mnemonic (kódszó), az utasítások végrehajtásához szükséges operandusok, az utasítás assembly megnevezése, a gépikód, illetve az utasítások hossza is helyet kaptak. Szintén az ábrán láthatók az úgynevezett fejmutatók, amelyek a láncolt listák kezdőelemére hivatottak mutatni a program inicializálását követően. Ezen fejmutatók nem változnak, ha csak nem a lista legelső eleme felülíródik egy új kezdőelemmel.

A láncolt listák használata azért is volt a leghelyesebb módszertan, mivel így csak a ténylegesen tárolandó adatoknak foglalódnak le az emulált memóriaterületek, ezzel hatékonyan kihasználva a számítógép fizikai memóriáját, nem úgy, mint a fix tömbök esetében, ahogy azt a 10. ábrán való összehasonlítással is nyomon követhetünk. A memória felépítését követően érdemes megemlíteni a memóriaműveleteket, amelyek egyedülálló módon imitálják a valós működést.

RAM tömbként

0	1	2	3	4	5	6	7	8	9	10	11
23	255	??	??	194	??	??	??	234	5	84	??

RAM láncolt listaként



10 A fix tömbös és láncolt listás megvalósítások összevetése

2.2.2 - Memóriaműveletek

A memóriaműveleteket nagy vonalakban az olvasási és írási műveletekre szokás felbontani. Az olvasási folyamat során egy adott memóriaterületre korábban elmentett értéket hívhatunk vissza, míg az írási művelet esetében egy már felhasznált memóriaterületet írhatunk felül, vagy egy üres memóriaterületre vihetünk fel eltárolandó adatot. Ezt a működést a `ReadMem()` és a `WriteMem()` függvények viszik véghez emulátorunk működésének folyamán.

Kezdjük az olvasást megvalósító `ReadMem()` „eljárással”. Ahogy az a 11. ábrán is látható ezen műveletet megvalósító függvény egy 64-bites egész értékkel tér vissza az olvasás folyamatának befejeztével. Az olvasási művelet során ciklikus módon keressük a kiolvasni kívánt memóriacímet az adatmemóriát megvalósító láncolt listában oly módon, hogy szekvenciálisan

végig haladunk a tömb összes elemén a listát lezáró „NULL”

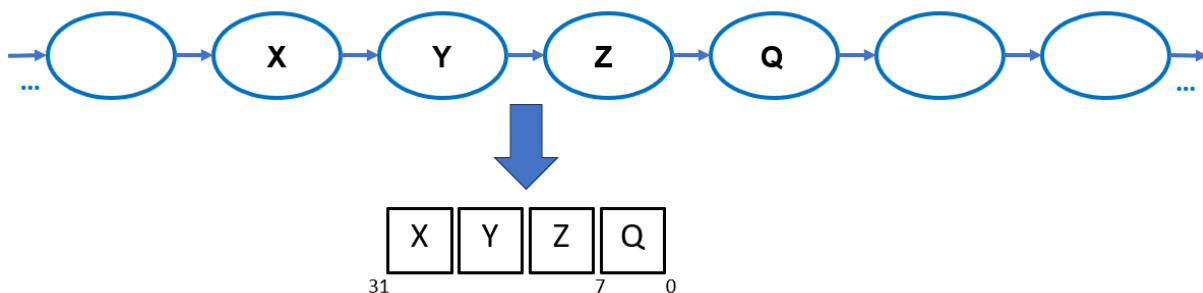
```

82 long long int ReadMem(long long int address, int size);
83 // Search for the given address in the data memory linked list
84 // If necessary compose multi-byte data from the consecutive data
85 // Return with the given data

```

11 Az olvasófüggvény szignatúrája

mutató irányába. Amennyiben ezen zárójelvényig nem találja meg a keresett memóriacímet az alprogram, abban az esetben nincs visszatérési érték, és a program egy egyedi hibával leáll. Az esetek többségében a program futásának elején megvalósuló inicializáláskor a memóriában az összes adat rendelkezésre áll egy helyesen megírt program esetén, amelynek működését a felhasználó elemezheti. Ekkor a függvény szignatúrájában fellelhető méretet mutató változónak



12 A láncolt lista linkelemeinek összegyűrése egyetlen 4 bájtos adattá

megfelelően egy ciklus, a bitműveletek és a megfelelő eltolások segítségével a hasznos adatot behelyezi a visszatérési értékbe. Maximum 8 bájtnyi adatot képes kiolvasni, ebben az esetben nyolcszor fut le a belső ciklus, minden alkalommal a megfelelő mennyiséggel eltolva és logikai „VAGY” művelettel összevetve a beírandó és a már beírt adatokat a visszatérési értékben. A 12. ábrán az a folyamat követhető nyomon, amikor a láncolt listák linkelemeit egyetlen 4 bájtos adattá gyűrjük össze. Lássuk mennyiben különbözik az írási műveleteket megvalósító művelet.

Az írást a WriteMem() valósítja meg a társfüggvényéhez hasonlóan. Az írási

```

87 void WriteMem(long long int address, int size, long long int data);
88 // Search for the given address in the data memory linked list
89 // Save all the data bytes with the corresponding addresses into the linked list
90 // Use little-endian byte order

```

műveletek
abból a
szempontból

13 Az írófüggvény szignatúrája

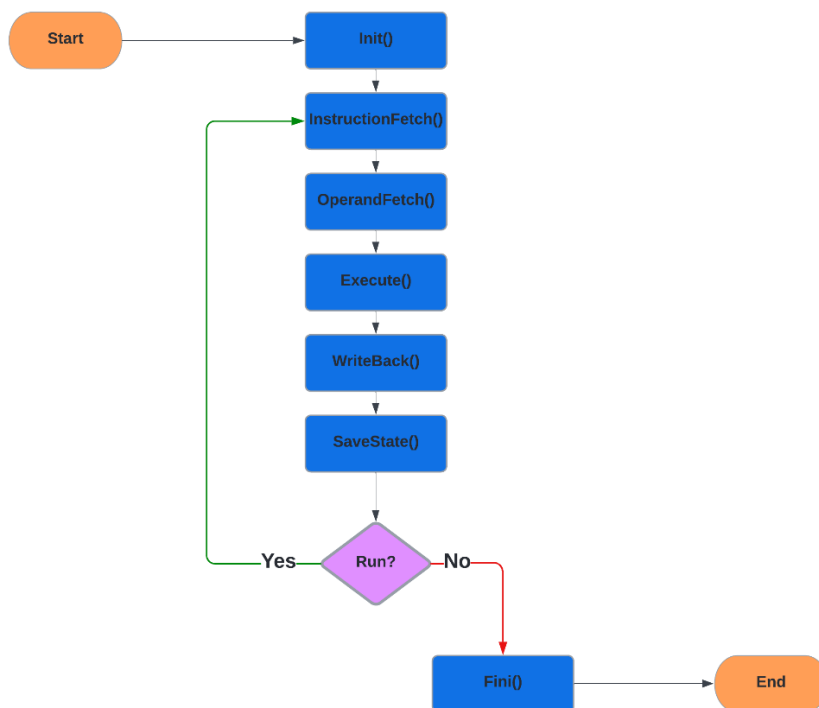
sokkal összetettebb műveletek, minekután ebben az esetben szükség lehet új memóriaterületek hozzáfűzésére, amelynek következtében a fejmutatók a lista újdonsült első elemére mutathatnak, vagy éppen a memóriaterület végének kibővítését vonhatják magukkal, illetve szerencsésebb esetekben a memóriaterület közepére kerülhetnek beszúrásra. Ezért is a függvény a jelenleg keresett memóriaterületeken kívül szekvenciális módon minden memóriaterület-ugráskor elmenti a jelenleg „megtalált” memóriaterületnél eggyel kisebb cím mutatóját is. Ezt egészen addig teszi egy ciklus, ameddig nem találja meg a keresett memóriacímet, mint rákövetkezőt, vagy nem fut rá a korábban is említett „NULL” mutatóra. Könnyű belátni, hogy ezen megközelítés az összes írási eshetőséget lefedi, így térjünk rá a memóriaterületek beszúrására. Amennyiben meglévő memóriaterületre szúrunk be adatot, a megfelelő eltolások segítségével beillesztjük a bájtnyi méretű „cellákba” a függvény szignatúrájában (lásd a 13. ábrán) is látható adatváltozó értékszelvényeit. Amennyiben még nem meglévő memóriaterületre írunk, abban az esetben először beállítjuk egy újonnan lefoglalt memóriaelemre a beírandó adatokat, majd a korábban rákövetkező elemre irányítjuk a lefoglalt memóriaterület mutatóit, valamint a korábban megelőző memóriaterület mutatóit is

aktualizáljuk aszerint, hogy közre vegyék az újonnan, az 1 bájtnyi adat számára lefoglalt területet. Sokbájtos adat esetén annyszor írunk be 1 bájtnyi adatot, amekkora az erre átadott változóban eltárolt hasznos adat mértéke.

Így, hogy jobb betekintést nyerhettünk ezen függvények működésébe érdemes leszögezni azt, hogy ezen műveletek a hardverben elektronikailag vannak kivitelezve, míg nekünk szoftveres környezetben ezt a működést függvények valósítják meg. Azért volt szükséges a funkcionális megvalósítás, hogy minél valóságghűbb módon működési hasonlatosságot emulálhassunk a programunk futásakor. A változók maximális 8 bájtos, vagy 64-bites mérete az x86-64 architektúrán alapuló architektúránk esetében azt a technológiai előnyt hivatottak éreztetni, hogy mennyivel is hatékonyabb, illetve fejlettebb ezen megvalósítás a 32-bites x86-hoz képest.

2.2.3 - Az emulátor működése

Az emulátor megvalósításához több összetett függvény megírására volt szükség. A működés szemléltetésére készítettünk egy erre alkalmas folyamatábrát, amely nagyvonalakban bemutatja az implementáció logikáját.



14 Emulátor működésének folyamatábrája

A processzorok működésük során alapvetően egy úgynevezett fetch-execute ciklust ismételve. Amint az a folyamatábrán is látható, programunk is ezt a ciklusszerű megvalósítást alkalmazza. Ezt a lépéssorozatot egy egyszerű megközelítésben 4 + 1 fázisra bontottuk. Az első 4 fázis (`InstructionFetch()`, `OperandFetch()`, `Execute()`, `WriteBack()`) szigorúan a processzor működéséhez kapcsolódik, az utolsó fázis (`SaveState()`) azonban ilyen szempontból elhanyagolható, mivel ez arra szolgál, hogy az emulátor kimenetet produkáljon. A ciklus megkezdése előtt szükség van egy inicializáló eljárásra is. Deklarálnunk kellett egy `Run` nevezetű, 32 bites egész típusú változót, amelynek logikailag igaz értéke a processzor futásának feltétele, ha ez hamis értékkel rendelkezik az emulált fetch-execute ciklus leáll.

2.2.3.1 - Init

A korábban említett inicializáló eljárás minden olyan műveletet elvégez, amely lehetővé teszi azt, hogy processzorunk meg tudja kezdeni a működését. Ez az eljárás az `Init()`, amely többek között elvégzi a már korábban előállított RAM fájl teljes körű feldolgozását, inicializálja a regiszterként használt globális változókat, és a memóriaként használatos láncolt listákat. A regiszterkészlet inicializálása két fő szempontból is kulcsfontosságú. Először is a regiszterek megvalósítására szolgáló változóink csak deklarálva voltak, szóval memóriaszemetet tartalmaznak. Ez komoly problémákat okozhat, mivel, ha így kezdenénk meg az utasítások végrehajtását, számunkra ismeretlen értékekkel dolgoznánk, amelyek eredményeként megsérülhet az adat integritás. Továbbá a verem mutató regiszter (`RSP`) inicializálása is elengedhetetlen, mivel a verem az adat memóriához fog tartozni, amelyet a korábban ismertetett láncolt listás módszerrel valósítottunk meg. Az adat memóriához fogjuk hozzá fűzni a verem szegmenst, a verem mutató regiszterben található memóriacímtől. Létrehoztunk egy `RSPinit` nevű, globális 64 bites egész típusú változót, amely a verem mutató regiszter kezdeti értékét tárolja. Ennek fontosságát a későbbiekben fogjuk megérteni.

A programszámláló regiszter inicializálása rendhagyó módon történik. Ez a regiszter a végrehajtani kívánt utasítás memóriacímét tárolja, ezért inicializálása azzal a memóriacímmel történik, ahol az emulálni kívánt program végrehajtása kezdődik. Ez az érték fellelhető a RAM fájlban.

A következő lépés a `Run` nevezetű változónk logikailag igaz értékkel való feltöltése. Ez a változó a korábbiakban ismertetettek alapján arra szolgál, hogy globálisan irányítsa emulátorunk futását.

Ezen lépések után következhet az adat és utasítás memória inicializálása. Az `Init()` függvény soronként beolvassa a RAM fájlt, és annak megfelelően, hogy a RAM fájl melyik szegmenséről érkezett a beolvasott sor, eltárolja azt `MemDataUnit` adategységben, ha adatmemóriáról van szó, vagy `MemInstUnit` adategységben, ha az utasítás memóriáról van szó. Miután megtörtént a sor dekódolása, és az értékek megfelelő eltárolása, hozzáfüzi őket a láncolt listákhoz, amelyeknek fejmutatói a `MemInst` és `MemData` nevezetű mutatók.

A beolvasás, és az adatok helyes konverziójának ellenőrzése érdekében létrehoztunk egy `PrintList()` nevezetű függvényt, amely egy sztringet vár paraméterként, ha a paraméter értéke `"data"`, akkor az adatmemóriát, ha a paraméter értéke `"inst"`, akkor az utasítás memóriát listázza soronként a program standard kimenetén. Egyéb paraméter esetén a `PrintList()` hibaüzenettel leáll.

Az `Init()` lefutása után végrehajtódott az összes szükséges művelet az `fetch-execute` ciklus megkezdése előtt. Ciklusunk futásának feltétele a `Run` nevezetű változó, amely többféleképpen vehet fel logikailag hamis értéket, amely a ciklus és az emulátor futásának befejezéséhez vezet. A legtöbb ilyen lehetőség kialakulásának a verem mutató regiszter aktuális értékéhez van köze. Ha a veremben szeretnénk tárolni adatokat, akkor az úgy történik, hogy a verem mutató aktuális értékéből kivonjuk a tárolni kívánt adat méretét bájtokban, és erre az összefüggő memóriaterületre mentjük el az adatot. A verem mutató egy olyan memóriaterületre mutat, amelyen már van érték tárolva. Mivel így módon használjuk a vermet, ha a verem mérete nő, a verem mutató értéke csökken. Ha a verem mutató a kezdeti értéknél nagyobbra nő, az azt jelenti, hogy programunk olyan memóriaterületet próbál használni, amely már nem tartozik a hatásköre alá, vagyis a futásnak le kell állnia. Ezt a kezdeti értéket az `RSPinit` változó tartalmazza. Szorosan kötődnek a veremkezeléshez a `call` és `ret` utasítások. A `call` utasítás egyetlen operandusú, ez általában a meghívandó függvényre mutató címke. Ezen címke mögött egy memóriacím áll, amelyet mi már kinyertünk az előfeldolgozás során. Az utasítás elmenteni a programszámláló regiszter aktuális értékét, amely már a függvény utáni első utasításra mutat, azért, mert az `InstructionFetch()` függvényünk ahogy feldolgozta a pillanatnyi

utasítást, lépteti a programszámláló regisztert, tehát ez nem a jelenleg végrehajtani kívánt utasításra fog mutatni. Ezt az értéket a verembe mentjük el, majd felülírjuk a a programszámláló regiszter tartalmát a függvényre mutató címmel. A végrehajtott függvény egy `ret` utasítással kell hogy végződjön. A `ret` utasítás, a verem tetején lévő címmel felülírja a programszámláló regisztert. Ez az érték a `call` utáni utasítás. Ily módon valósítható meg egy függvényhívás.

Az emulálni kívánt program is egy `ret` utasítással végződik. Ha a hívási és veremkezelési konvencióknak megfelelően, a veremben elhelyezett értékeket, használatuk után felszabadítjuk, akkor a program befejezésekor ugyanannak az értéknek kell lennie a veremben, mint induláskor. Ha helyesen kezeltük a vermet, az utolsó utasítás végrehajtásakor a verem mutató regiszter értéke megegyezik a kezdeti értékkel. A `ret` utasítás a programszámláló regiszterbe tölti a verem tetején kezdődő 8 bájtos adatot, vagyis a verem mutatónk a kezdeti értéknél nagyobbra nő. Ez a program leállításához vezet.

A futás leállítása történhet a `ReadMem()` függvény által is. A memóriaolvasó függvényünk két paramétert kap, egy memóriacímet, és a beolvasni kívánt adat méretét bájtokban. Ha a függvény nem talál az adott címen akkora összefüggő adatot, amelyinre szüksége lenne, a futás leáll.

Továbbá a programunk leáll, ha olyan utasítással találkozik, amely nincs az eddig implementált utasítások között.

2.2.3.2 - Instruction fetch

Minden szükséges művelet végrehajtásra került a ciklus indítása előtt, szóval az megkezdheti működését. Az első eljárás a végrehajtási sorban az `InstructionFetch()` nevezetű. Az `RIP` által mutatott címről betölti a korábban már említett, láncolt listával megvalósított utasítás memóriában megtalálható instrukciót, majd lépteti a programszámláló regisztert. Architektúránk rendelkezik olyan változókkal, amelyek egy utasítás minden elemi részét képesek tárolni, ezek a változók, és szerepük a következők:

`Mnemonic` – az utasítás neve, azt leíró kulcsszó

`Operand1` – a művelet első operandusa

`Operand2` – a művelet második operandusa

`Assembly` – a teljes utasítás assembly nyelven

`MachineCode` – az assembly utasítás gépi kódja

`InstLength` – az assembly utasítás gépi kódjának hossza bájtokban

Az `InstructionFetch()` függvény az utasítás memóriában, a fejmutatótól indulva keres. Azon a memóriacímen lévő utasítást keresi, amelynek címe megegyezik a programszámláló regiszterben lévő értékkel. Ha ezt megtalálta, felülírja a fentiekben említett változókat a talált utasítás szerinti értékekkel, majd az `RIP` regiszter értékét növeli a jelenleg beolvasott utasítás hosszával. Ha nincs találat, hibaüzenetet produkálva leállítja a teljes program működését. Felmerülhet kérdésként, hogy miért nem a `Run` változót állítja logikailag hamis értékre. Ennek az a magyarázata, hogy a `Run` változó értékének ellenőrzése a ciklus elején történik, ha ez logikailag igaz, akkor egy teljes fetch-execute ciklus végrehajtódik, és programunk kimenetet is produkálna, holott meg sem találtuk azt az utasítást, amit végre szerettünk volna hajtani.

2.2.3.3 - `OperandFetch`

A végrehajtási sorban az `OperandFetch()` függvény a következő. Az `InstructionFetch` által frissített `Operand1` és `Operand2` változókat dekódolja, majd betölti a megfelelő értékeket az `ALU` regisztereibe a megfelelő címzési módot használva. Ezen változókra jelenleg sztringként lehet hivatkozni. Ezen sztringek mögött értékek állnak, amelyek lehetnek memóriában tároltak, konstansok, vagy akár regiszterekben találhatóak. A függvénynek képes kellett lennie mindhárom esetben felismerni, hogy melyik opcióval van dolga, és ennek megfelelően különböző módszerekkel kinyerni az értékeket, amelyeket az `ALU` bemeneti regisztereibe tölt. Léteznek olyan utasítások is, amelyek nem rendelkeznek operandusokkal (`ret`, `nop`),

2.2.3.3.1 - Konstansok

Kezdjük a legegyszerűbb változattal, amikor is az operandus egy konstans, más néven literál. Az emulátor működéséhez használt RAM fájl szintaxis alapján, amely a függelékben megtalálható, a konstans két változatban létezhet, operandusként, `0x` prefixxel ellátva, vagy nélküle. Mindkét esetben 16-os számrendszer a használatos.

Ennek felismerésére egy `FetchLiteral()` nevű függvényt használtunk, amely eldönti, hogy melyik konstans formátum van használatban, és ennek megfelelően számszerűsíti a

sztring tartalmát. Ha a konverzió sikeres volt, visszatér egy logikailag igaz, másképpen egy logikailag hamis értékkel.

2.2.3.3.2 - Regiszterek

Ezen változat sem igényelt komplexebb logikát, egyszerű elágazásokkal eldöntöttük, hogy a sztringként hivatkozott operandus melyik regiszterhez is tartozik. Egyedüli kihívása ennek a verzióknak az volt, hogy a 64 bites regiszterként használt változóinkból csak azon a biteken lévő értékeket töltsük be az ALU bemeneti regisztereibe, amelyekre hivatkozik az utasítás. Itt jön képbe a maszkolás. Tegyük fel, hogy szükségünk van az EBX regiszter tartalmára. Ebben az esetben, az RBX 64 bites regiszter legkisebb helyi értékein lévő 32 bitjére van szükségünk. Ezt úgy állítjuk elő, hogy az RBX nevű változót bitenként összeeszeljük egy 64 bit hosszúságú, felső 32 bitjén 0, alsó 32 bitjén 1 értékkel ellátott maszkkal. Ezt a műveletet minden lehetséges eshetőségre képesek vagyunk megtenni, akár 16, akár 8 biten tárolt értékre van szükségünk az adott regiszterből. Az ilyenkor használt maszkokat nevesített konstansként definiáltuk. A korábban leírt műveleteket függvénybe szerveztük, melynek neve `FetchRegister()`.

2.2.3.3.3 - Memóriaterületek

Előfordulhat azonban, hogy az operandus a memóriában található. Ennek meghatározása volt a legösszetettebb feladat. A használt szintaxis alapján, ebben az esetben a sztring (`Operand1`, vagy `Operand2`) mindenképpen egy betűvel kezdődik, amely megadja a hivatkozott memóriaterület nagyságát (Q - 8 bájt, D - 4 bájt, W - 2 bájt, B - 1 bájt), majd ezt követi egy kifejezés szögletes zárójelek között, amely megoldásával megkapjuk az érték első bájtjára mutató címet. Ettől a címtől kezdve a memóriaolvásó függvényünk segítségével beolvassunk adott méretnyi bájtot, majd ezt töltjük be az ALU bemeneti regisztereibe. A kifejezés megoldási komplexitásának szemléltetése egy példával:

`D [rbp+rcx*4-20]`

Az operandus egy dupla szó hosszúságú memóriában tárolt adat, amelynek címét úgy határozhatjuk meg, hogy az RCX regiszter értékének négyszeresét hozzáadjuk az RBP regiszter aktuális értékéhez majd ebből még kivonunk 20-at. Amint látható ebben a verzióban is szükség volt regiszterekben tárolt értékek kinyerésére, valamint konstansok felismerésére, amelyekre a korábbiakban is használt `FetchRegister` és `FetchLiteral` függvényeket használtuk fel.

Az operandusok dekódolása közben, a megfelelő helyen és időben beállítódnak az OpMaskSource és OpMaskDest nevű változók. Ezek annak megfelelően vesznek fel értéket, hogy milyen maszkot kellett használni a cél, illetve forrás operandusok dekódolása közben.

2.2.3.4 - Execute

Minden egyes olyan utasításhoz, amelyet emulátorunk kezelni képes létrehoztunk egy ugyanolyan nevű eljárást, amely végrehajtja azt. Az Execute() nevű eljárásunk tulajdonképpen kiválasztja ezt az eljárást. Elágazásokat használva összehasonlítja a Mnemonic nevű változó tartalmát az implementált utasításokkal, majd ha egyezést talál végrehajtja az implementált függvényt. Jelenleg emulátorunk 71 utasítást képes kezelni, és ezen utasítások lefedik az “Assembly programozás” c. tárgy keretein belül tanultakat. A lista meg Például az összeadás utasításnak létrehoztuk az add() nevezetű metódust. Nézzük meg ezt egy kicsit közelebbről. Az add utasítás két operandusú, első operandusa a céloperandus, második operandusa a forrásoperandus. A két értéket kell összeadni, amely nem tűnik nagy kihívásnak, azonban ez assembly nyelven eléggé komplex feladat. Először is a céloperandus, amelyben az eredmény lesz tárolva, lehet 8, 16, 32 és 64 bites is. Minden esetet másképpen kell kezelni, mivel az összeadás művelet aritmetikai művelet, és hatással van a státuszregiszter bitjeire, többek között a túlcsordulást és átvitelt jelző bitekre. A státuszregiszter bitjeit 32 bites egész típusú globális változóként deklaráltuk. Átvitel bit akkor keletkezik összeadásnál, ha az első operandus nagyobb az adott számú biten ábrázolható legnagyobb érték, és a második operandus különbségénél. A túlcsordulás megtörténik, ha a két operandus ugyanolyan előjellel rendelkezik, és az összeg előjele különbözik ezen előjeltől. Ezt a két feltételt minden céloperandus variációra ellenőriznünk kellett, elágazásokat használva. Továbbá az összeadás hatással van az előjelbitre és a zéró flag bitre is. Ezek ellenőrzésére létrehoztunk két eljárást, amelyeknek át kell adni az előállt eredményt, és a céloperandus méretét, mert ehhez viszonyítva írják felül 1-es vagy éppen 0 értékkel az említett biteket megtestesítő globális változókat.

2.2.3.5 - WriteBack

A WriteBack() által történik a céloperandus felülírása, amennyiben ez szükséges. Léteznek olyan utasítások, ahol nincs szükség az eredmény tárolására, csak a státuszregiszter bitjei módosulnak (cmp, test). A verem kezelő eljárások (push, pop) adatot tárolnak, azonban

ezt maguk végzik el, ők sem használják fel a `WriteBack()` eljárást. Értelemszerűen az operandus nélküli utasítások sem tárolják el eredményüket. Ezekben az esetekben módosításokat nem végezve fut le a `WriteBack()`.

Minden más esetben, az eredmény az ALU kimeneti regiszterében lesz fellelhető. Első lépésként ezt az értéket maszkoljuk a céloperandus maszkjával, amely az `OperandFetch` függvényben állt elő (`OpMaskDest`). Szükséges megállapítani, hogy a felülírni kívánt terület regiszterhez, vagy memóriához tartozik. Ennek megállapítása szintén a `FetchRegister` segítségével történik, mivel ennek a függvénynek a visszatérési értéke logikailag igaz, ha paramétereként kapott sztring típusú operandus egy regiszter, vagy annak egy kisebbik része. Fontos paraméter az adat mérete is. Ezt az egyik korábbi lépésben, az `OperandFetch` függvényben határoztuk meg, mivel ott volt a legcélszerűbb. Ha előállt a méret, a felülírni kívánt terület, és maga az adat is, a `WriteBack()` elmenti azt a megfelelő módon. Mivel egy 64 bites regiszterként használt változónak hivatkozható az alsó 32, 16, illetve 8 bitje, ezért felülírásnál is csak a hivatkozott bitek íródhatnak felül. Ezt úgy oldottuk meg, hogy létrehoztunk egy azon biteken egyes értéket tartalmazó maszkot, amelyekre a végrehajtott utasítás is hivatkozik, majd ezzel a maszkkal összeéseltük az előállt eredményt, és végül az így kapott értéket ismét összeéseltük, de már a felülírni kívánt területet tartalmazó regiszterként használt változóval. Ily módon csak a hivatkozott terület íródott felül, a regiszter további tartalma érintetlen maradt.

2.2.3.6 - Save state

A hagyományos fetch-execute ciklus végrehajtása megtörtént, azonban, hogy mi is láthassuk mi történik a végrehajtás alatt, létrehoztuk a `SaveState()` függvényt. Ennek a függvénynek az a szerepe, hogy minden egyes utasítás végrehajtása után eltárolja a pillanatnyi állapotokat. Dokumentálva lesznek a regiszterek értékei, a stack méret, és a stackben lévő bájtok fordított sorrendben, valamint a teljes assembly utasítás szöveges formában. A függvény létrehoz egy `output` nevezetű fájlt, majd ebbe írja ki a függelékben megtalálható szintaxis alapján a fent említett értékeket. Programunk rendelkezik grafikus felülettel is, amelynek a bemenete lesz ez a fájl, amely tulajdonképpen egy kivonat a végrehajtás alatt megtörtént változásokról.

2.2.3.7 - Fini

Ha a `Run` változónk logikailag hamis értéket vesz fel, a ciklus végrehajtása befejeződik. A ciklus után szükséges elmenteni rendszerünk végső állapotát, amely hasonlóképpen történik,

mint a SaveState-ben. A Fini függvény ugyanazokat az adatokat tárolja el az output fájlban, mint a SaveState, továbbá felszabadítja a futás alatt lefoglalt memóriaterületeket, majd terminálja az emulátor program működését.

2.3 - Grafikus felhasználói felület

Az emulátor által előállított kimeneti fájl kiváló bemenetként szolgálhat esetlegesen más programok számára, amelyek kutatási célzattal megvizsgálják ezen kimenetet, például futási elemzéshez. A problémák akkor merülnek fel, amikor figyelembe vesszük ezen projektnek a legfőbb célját: ezen programot elsősorban oktatásban szeretnénk hasznosítani. Még egy jóltaglalt szöveges fájl könnyű bemenetként szolgál egy program számára, ha egy diákot, illetve oktatót tekintünk, mint fogyasztó onnantól már nehezen értelmezhetőnek tűnhet a fájl. Ennek szemléltetésére szolgál a 15. ábra.

```

4198662 68781716 68781878 68781878 90197966376698770 1 90197966376698742 90197966376698
166 0 0 90073312864405654 0 90184319943919616 68781156 90197986613158262 0 0 0
0 0 1 0 - BEGIN
4198667 3 68781878 68781878 90197966376698770 1 90197966376698742 90197966376698166 0
0 90073312864405654 0 90184319943919616 68781156 90197986613158262 0 0 0 0 0
1 0 b80300000 mov eax,0x3
4198669 3 68781878 68781878 90197966376698770 1 90197966376698742 90197966376698158 0
0 90073312864405654 0 90184319943919616 68781156 90197986613158262 0 0 0 0 0
1 8 0 0 0 0 0 0 3 6a03 push 0x3
4198671 3 68781878 68781878 90197966376698770 1 90197966376698742 90197966376698150 0
0 90073312864405654 0 90184319943919616 68781156 90197986613158262 0 0 0 0 0
1 16 0 0 0 0 0 0 3 0 0 0 0 0 0 0 4 6a04 push 0x4
4198673 3 68781878 68781878 90197966376698770 1 90197966376698742 90197966376698142 0
0 90073312864405654 0 90184319943919616 68781156 90197986613158262 0 0 0 0 0
1 24 0 0 0 0 0 0 3 0 0 0 0 0 0 0 4 0 0 0 0 0 5
6a05 push 0x5

```

15 Output fájl

Ha egy diáknak odaadjuk ezen kimenetet, hogy esetlegesen értelmezze, vonjon le belőle következtetéseket nem lesz egyszerű dolga, mivel a sorokban megtalálható számok közül nem feltétlen tudja egyszerűen megmondani, hogy melyik szám pontosan mit jelöl. Ha az illető, aki szeretné használni az emulátort meg is tudja tanulni a szintaxis leírás alapján melyik érték mit jelöl, még akkor sem lesz teljesen egyszerű értelmeznie a kimenetet. Egyik nehezítő tényező, hogy némelyik regiszter értéke a 64 bites architektúra miatt nagyon nagy lehet, amely miatt, ha egy szövegszerkesztőben nézzük a fájlt nem fog feltétlenül minden sor kiférni egy képernyőn, ezt tovább nehezítve, ha esetleg nem csak egy ilyen érték van egy sorban.

Másik nehezítő tényező a szintaxis megtanulására, hogy egy sorban nem feltétlenül igaz, hogy mindig ugyanannyi érték fordul elő. Fentebb már említettük a stack méretét eltároló értéket. Ha a bemeneti kódban előfordul stack manipuláció, amely során belekerülnek a stackbe értékek, akkor a sorban megtalálható stack mérettől függő mennyiségű stackben található egy bajtot

reprezentáló érték kerül még be az adott sorba. Ha ezt a programot szeretnénk oktatási célzattal effektíven felhasználni, akkor szükségünk lesz valamilyen megoldásra, amely által könnyebben értelmezhető lesz a fájl.

2.3.1 - A GUI-ról általában

Ahogy azt említettük, a kimeneti fájl egy másik program számára kiválóan értelmezhető, mivel egy adott szintaxis alapján vannak strukturáltan rendezve benne az értékek. Ahhoz, hogy a kimenet értelmezését egyszerűbbé tegyük a hallgatók számára pontosan egy ilyen utat választottunk: létrehozunk egy olyan programot, amely az emulátor kimenetét képes értelmezni, valamint annak adatait képes könnyebben értelmezhető módon megjeleníteni. A programcsomagunk harmadik, és egyben utolsó komponenseként létrejött az úgynevezett grafikus felhasználói felület, avagy rövidebb nevén GUI (az angol Graphical User Interface kifejezésből ered).

Még az emulátor fejlesztésénél a C programozási nyelv megfelelő választás volt annak gyorsasága és rendszerközelisége miatt, egy szöveges fájlt feldolgozó grafikus programnál ezekre nem volt szükségünk. További ellenérv a C-vel szemben, hogy grafikus felületek elkészítéséhez szükséges könyvtárakból nem sok található, mivel a legtöbb inkább a C++ nyelvet célozza meg. Végző soron a GUI elkészítéséhez a Python programozási nyelvre esett a választás. A Python nyelv egyik nagy előnye a C-vel szemben a hatalmas mennyiségű függvénykönyvtár, amely elérhető a nyelvhez (ennek a dokumentumnak az írásakor közel 5 millió csomag elérhető a hivatalos Pythonhoz kitalált pip csomagkezelőt használva), ezáltal több választási lehetőség megnyílván grafikus könyvtárak tekintetében. Az elérhető grafikus könyvtárak közül végül a PyQt5 nevezetű könyvtárra esett a választás, mivel ez bizonyult az egyik legegyszerűbben felhasználhatónak, mint telepítés, mint programozási szemszögből, valamint ez a könyvtár olyan grafikus technológiákon alapul, amely a mai modern Linux rendszereken alaptól megtalálható. További előnye a Pythonnak a nyelv egyszerűsége, gondolva itt az egyes problémák megoldásához szükséges kód mennyiségére.

2.3.2 - A felület felépítése

A GUI fő felületét, ahogyan azt a 16. ábrán is látjuk ránézésre három részre tudjuk bontani, amelyeket tekinthetünk akár “oszlopoknak” is. Logikai felépítés alapján talán a legegyszerűbb, ha jobbról balra haladunk az egyes részek leírásával.

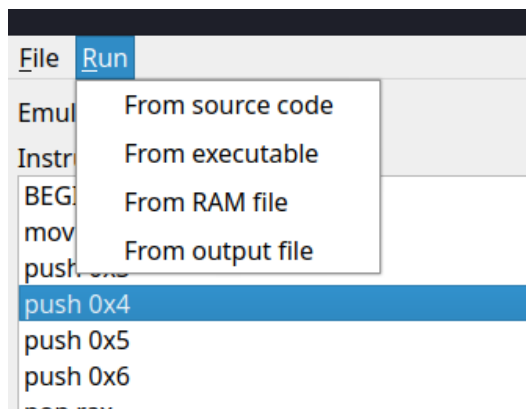
szabadon lépegetni, így ezáltal fog változni a felületen látható regiszterek, illetve a stack értékei. Itt fontos megjegyezni, hogy a regiszter, illetve stack értékek mindig az instrukció végrehajtás UTÁNI értékeket jelenítik meg. Alapvetően ezzel semmi probléma nem merül fel, viszont a legelső instrukció előtt nem tudjuk így, hogy a regiszterek valamit a stack milyen értékeket tárolnak. Ebből kiindulva, a felületen, illetve az emulátor kimenetben megjelenik két extra sor: az első még a legelső instrukció elé kerül be, itt az Assembly kód helyett egy BEGIN szó jelenik meg, amely még az emulátor kiinduló regiszter és stack értékeit jeleníti meg. Ezen felül megjelenik a legutolsó instrukció után egy END szóval megjelölt sor, amely a legutolsó instrukció utáni értékek megjelenítésére szolgál.

A kilistázott instrukcióknál fontos megemlíteni, hogy a listába nem a program kódjában megtalálható instrukciók kerülnek betöltésre ahogyan azok a kódban le vannak írva, hanem a program végrehajtása során lefutott instrukciók listája időrendi sorrendben. Korábban említettük Szucsányi Erik Péter szakdolgozatát [2], amely az Assembly kódokban megtalálható egyes instrukciók előfordulását vizsgálja. Míg az ő dolgozata magában a kódban lévő instrukciók előfordulását vizsgálja, addig az általunk elkészített program a futás során végrehajtott instrukciók vizsgálatára alkalmazható.

2.3.3 - További funkcionalitások

A felületnek meg tudunk adni már az emulátor által elkészített kimeneti fájlt, hogy azt jelenítse meg. Alapvetően ez is volt a célunk, viszont sokkal felhasználó barátabb lehet a felhasználói felület, ha képes lenne az emulátor direkt futtatására is.

Ahogy látható a 17. ábrán, a felületen található egy menüsáv, amelyen Run névvel található egy menüpont. Alapvetően ez alá került az a menüpont is, amellyel az elkészült kimenetet tudjuk megjeleníteni Load from output néven. Ezen felül találhatunk itt még három másik menüt is. Talán a leggyakrabban használt, az a Load from source code. Itt Assembly, illetve C kódokat tudunk kikeresni, azaz .s és .c kiterjesztésű fájlokat, amelyeket átad a GUI az emulátornak. A következő a Load from executable. Itt már elkészült futtatható állományokat tudunk kikeresni, amelyet az emulátor futtatni tud. Az utolsó a Load from RAM file, ezzel az előfeldolgozás alatt elkészült RAM fájlokat tudunk átadni az emulátornak.



17 A GUI menüsávja, menüpontjai

Mint reflektáltunk rá, az előbb említett három menüpont az emulátort fogja hívni különböző paraméterek alapján, és az általa elkészült kimenetet fogják megjeleníteni. Ezt figyelembe véve ahhoz, hogy ezek működjenek előfeltétel, hogy először a GUI ismerje az emulátor elérési útját.

A menüsávon található egy másik menüpont is: a `File` menü., amely alatt található egy `Load emulator` menüpont. Ezzel a menüvel tudjuk megmondani milyen elérési út alatt található meg az emulátor. A menüpont meg fog nekünk nyitni egy fájl kiválasztó ablakot, amellyel az emulátor futtatható állományát ki tudjuk keresni. Annak érdekében, hogy itt kizárólag az emulátort tudjuk betölteni szükség volt egy kisebb védelmi logika kialakítására. A GUI itt csak olyan programok kiválasztását engedélyezi, amelyeknek `asemu` a neve. Emellett az emulátornak van egy parancssori argumentuma (a `--gui-info`) amely annyit tesz, ha az emulátort ezzel az argumentummal futtatjuk, akkor kiír `stdout`-ra egy `asemu` szöveget. Mikor a GUI megpróbálja betölteni a megadott fájlt, akkor azt lefuttatja és megnézi, hogy az kiírja-e a fenti szöveget. Ha igen, akkor a GUI eltárolja az emulátor elérési útját későbbi használatra. Az emulátor betöltését a GUI indításkor automatikusan is megpróbálja megkísérelni. Ilyenkor az emulátort a projekt struktúra alapján relatív hivatkozással próbálja meg elérni (ez leírható az alábbi útvonallal: `../asemu`). A betöltött emulátor elérési útja a felületen is megjelenik, így tudhatjuk biztosan, hogy ez sikeresen megtörtént.

3 - Összefoglalás

Célkitűzéseink eme nagy horderejű csapatmunka megkezdésekor már-már kőbe voltak vésve; alkossunk meg egy olyan oktatóprogramot, amelynek segítségével a leendő informatikus hallgatók az Assembly programozási nyelvet interaktív módon sajátíthatják el az egyetemi órák keretein belül, emellett legyen felkészítve esetleges tudományos célokra való alkalmazásra is. Nagyvonalakban mindkét cél megvalósult hiszen egy olyan jól átlátható grafikus felülettel is rendelkező emulátort sikerült megalkotnunk, amely az x86-64 architektúrát emulálva valós C/Assembly kódokból felépülő egyszerűbb programokat képes futtatni valós memóriakezelés mellett. A folyamatosan frissülő szöveges fájlkimenetének segítségével lekövethetővé teszi ezen programok működését, amelyet a saját grafikus felületünkön túl más programok is saját bemeneteiknek hasznosíthatnak. Habár sok nehézséget leküzdöttünk, programunk nem képes rendszerhívásokra és megszakításkezelésre, illetve dinamikus linkelést igénylő utasításokat csakis statikus linkelés mentén képes lefuttatni, azonban ezen hiányosságokért kárpótolnak emulátorunk képességei. Az egyetemi órákon szükséges programkódok mindegyikét képes futtatni, az alap x86-64 architektúra összes regisztere a részét képezi, a valós stack memóriaszegmenset magába foglalja, képes statikus és dinamikus változókat kezelni, továbbá bonyolult fejlesztői eszközöket, mint tömböket, szubrutin hívásokat és rekurziót is helyén értelmez. Az bizonyos, hogy nagy fába vágtuk a fejszénket, amely potenciális továbbfejlesztési lehetőségekben gyümölcsözik, így TDK dolgozatunk témájaként fantasztikus választás volt.

3.1 - A fejlesztés felosztása

Az általunk választott téma feldolgozása nagy, illetve komplex feladatnak bizonyult, amely miatt indokolt volt a csapatban való közös munka. Még a projekt megtervezése közös munka volt, a fejlesztést a projekt elkészítésének sebessége érdekében felosztottuk hármunk között. Fejlesztés során a csapat két nagyobb halmazra bomlott, amely során az előfeldolgozással, illetve a grafikus felhasználó felülettel való munkálatokat Horváth Norbert végezte, miközben Róth Arnold és Sajó Balázs az emulátor megvalósításán dolgozott. Mivel az emulátor önmagában is egy nagyobb komponensnek számít, ezért itt is tovább lehetett osztani a munkálatokat. Minekután az x86-64 Assembly programozási nyelv 981 utasítással, és ezek megközelítőleg 3600 alvariánsával operál, így szükséges volt egy teljesíthető mennyiségű célt leszögeznünk az emulátort fejlesztő alcsapatnak. Ehhez segítségünkre volt a tanár úr által az

egyetemi órák keretein belül elvárt utasítások listája, amelyet ezen műveletek kiegészítéséhez szükséges további utasításokkal múltunk felül. Az utasításokból Arnold az ugró-, a forgató- és a bitmanipuláló utasításokért volt felelős a memória- és az output műveleteket megvalósító függvények megírása mellett. Balázs az instrukció- és operanduskereső, az állapot-visszamentő és az inicializáló folyamatokat megvalósító függvények megírásán felül a logikai, az adatmozgató és az aritmetikai utasítások megvalósításával foglalatzkodott. Természetesen ezen alcsapat munkájának voltak közös pontjai is, mint például az önállóan megírt kódsorok logikai egyeztetése internetes hívásokon keresztül, illetve az emulátor utasításainak futtatására szolgáló függvény, habár egyéni, de közös fejlesztése is. Összességében sikeres volt a kooperáció, hisz habár két alcsapatként, de mindig egymást segítve tudtunk megoldásokat találni akár egymás-, akár saját problémáinkat illetően. A webes hívások a Discord chatprogramon folytak, az egyéb csapatbeli kommunikációhoz pedig Messenger csoportot hoztunk létre. Idővel el kellett gondolkoznunk a dokumentáció megírásához használt technológiákat illetően is, amelyhez végtére is a Microsoft Office Word programot használtuk. Ezen szövegszerkesztő manapság már rendelkezik egy online felületű felhőalapú dokumentumszerkesztővel is, amelynek segítségével a saját fejlesztéseinkről offline megírt szövegezést ezen közös online TDK dokumentáció részévé tehattük. Mindezen közös munka gyümölcseként megalkottuk az emulátort, és egy kerek dolgozatot.

3.2 - Fejlesztésben felhasznált technológiák

A technológiák közül talán az első, amelyet érdemes megemlíteni, az a felhasznált programozási nyelvek. Ahogyan reflektáltunk rá, az előfeldolgozó, illetve az emulátor elkészítéséhez a C programozási nyelvet használtuk, mivel ez a nyelv híres annak sebességéről és rendszerközeliségéről, valamint ezt a programozási nyelvet a korábbi tanulmányaink miatt már mind ismertük. A C programozási nyelv tanulására kiváló könyv Brian W. Kernighan és Dennis M. Ritchie The C Programming Language Second Edition című könyve [1]. A kód fordítására, mint előfeldolgozón belül, mint magunknak a program fordítására a GCC nevezetű fordítócsomagot használtuk. A UI elkészítéséhez a Python programozási nyelvet használtuk, valamint vele együtt a PyQt5 grafikus könyvtárat. A PyQt5 könyvtár használatához kiváló segítségnek bizonyult annak dokumentációja [10].

A fejlesztéshez általánosságban a Visual Studio Code-ot használtuk, mint szövegszerkesztő. Ezen szoftver kiválóan támogatja a fejlesztésünket annak plugin könyvtárával, amellyel a szükséges nyelvek támogatásának elérése egyszerű volt.

Mivel csapatban dolgoztunk ezért fontos volt, hogy találjunk egy közös platformot, ahol a kódunkat tárolhattuk, illetve amely tud segíteni minket a közös fejlesztés megvalósításában. Ehhez a Git verziókezelő szoftvert használtuk, amelyet összekötöttünk egy távoli GitHub repository-val. Ezzel elérhettük, hogy a fejlesztett kódunkat a többi csapattag mindig elérje, valamint így van egy távoli biztonsági mentés is a kódunkról.

3.3 - Továbbfejlesztési lehetőségek

Egy szoftveres termék fejlesztése során soha nem jön el az a pont, mikor azt mondjuk “kész vagyunk”. Egy terméken mindig lesznek fejlesztési lehetőségek, legyen az később beérkező igény a program képességeinek kiegészítésére, esetlegesen felmerülő hibák javítása, vagy lehet még szó akár optimalizációs lehetőségekről is. Mivel ilyen téren a mi szoftverünk se képez kivételt, ezért példának felsorolnánk néhány ilyen lehetőséget, amelyet találtunk:

Az emulátor készítése során az egyik cél, amelyet kitűztünk, hogy a tanórákon elsajátítható mnemonikok mindegyikét támogassa a program. Ezen célunkat sikeresen el is értük, viszont ez az x86-64 architektúrában megtalálható mnemonikok 7%-át teszi ki, mivel az órára kb. 70 mnemonik támogatására volt szükség, viszont az architektúrában kb. 1000 egyedi mnemonik megtalálható.

A fejlesztés során az emulátort csak kisebb programokkal teszteltük, így nem merült fel bennünk a kérdés, hogy az emulátorunk futásának mik a fizikai limitjei. Mivel a program futásához szükséges összes instrukciót eltároljuk folyamatosan a memóriába, ezért nagyobb programoknak hatalmas lábnyoma lehet a memóriában. Fejlesztés során nem merült fel a kérdés, hogy esetleg limitáljuk-e a futtatható programok méretét, illetve alkalmazzunk-e más fajta optimalizációkat ennek elkerülésére.

Annak ellenére, hogy az emulátorunk a függelékben megtalálható Assembly instrukciókat mind támogatja, valamint valós memóriakezelést használ, vannak korlátai, hogy milyen programokat futtathatunk vele. Az emulátorunk nem képes olyan programok futtatására, amelyek megszakításkezelést alkalmaznak. Nem képes az emulátor rendszerhívásokat véghez vinni, így például a gyakran használt `printf()` és `scanf()` függvények hívására nem alkalmas.

Alapvetően az emulátor egyszerűbb user-space-ben működő programok futtatására való, viszont ígyis vannak függvények, amelyeket nem minden esetben tud kezelni: erre példa a C-ben megtalálható `pow` függvény, amelyet egy külső könyvtár használatával tudunk meghívni, Ha a fordítás során dinamikus linkelést használunk a könyvtár meghívásához, akkor az emulátor nem képes futtatni az adott függvényt, viszont statikus linkelés során, mikor a kód belekerül a futtatható állományba működni fog.

Irodalomjegyzék

1. B. W. Kernighan/D. M. Ritchie (1988). *The C Programming Language, 2nd edition*
2. Debreceni Egyetem Informatikai Kar. Szucsányi E. P. szakdolgozat (2022). *Assembly programelemek előfordulásának statisztikai elemzése*
3. GitHub. rkarris12. *Assembly Emulator*:
https://github.com/rkharris12/assembly_emulator [Letöltve: 2022. december]
4. Github. danrose499. *x86 Assembly Emulator*:
<https://github.com/danrose499/x86Emulator> [Letöltve: 2022. december]
5. GitHub. rafaelcarlos. *Assembly x86 Emulator*:
<https://github.com/carlosrafaelgn/Asm86> [Letöltve: 2022. december]
6. Github. wtdcode. Unicorn Engine:
<https://github.com/unicorn-engine/unicorn> [Letöltve: 2022. december]
7. Intel. (2023). *Intel 64 and IA-32 Architectures Software Developer's Manual*:
<https://irh.inf.unideb.hu/~vargai/download/assembly/Intel.pdf> [Letöltve: 2023. október]
8. Man7. gcc: <https://man7.org/linux/man-pages/man1/gcc.1.html> [Letöltve: 2023. október]
9. Man7. objdump: <https://man7.org/linux/man-pages/man1/objdump.1.html> [Letöltve: 2023. október]
10. PyQT5 dokumentáció: <https://doc.qt.io/qtforpython-5/contents.html> [Letöltve: 2023. július]
11. R. Seyfarth (2011). *Introduction to 64 Bit Assembly Programming Language for Linux*
12. Wikipedia. *Executable and Linkable Format*:
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format [Letöltve: 2023. október]
13. Wikipedia. i386: <https://en.wikipedia.org/wiki/I386> [Letöltve: 2023. szeptember]
14. Wikipedia. x86-64: <https://en.wikipedia.org/wiki/X86-64> [Letöltve: 2023. szeptember]
15. Yale University. *Executable and Linkable Format (ELF)*:
http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf [Letöltve: 2023. október]

Ábrajegyzék

1 Az Intel 80386 processzor	10
2 Az Intel 80386 és i7-8086K összehasonlítása	11
3 A címzési módok és a regiszterek alregiszterei.....	13
4 A bázis relatív címzés.....	13
5 A projekt komponensei.....	15
6 ELF fájl elrendezése	16
7 Objdump kimenete	18
8 RAM fájl.....	19
9 Az adat- és utasításmemória linkelemei, ezen láncolt listák fejmutatói.....	23
10 A fix tömbös és láncolt listás megvalósítások összevetése	24
11 Az olvasófüggvény szignatúrája.....	24
12 A láncolt lista linkelemeinek összegyűrése egyetlen 4 bájtos adattá	25
13 Az írófüggvény szignatúrája.....	25
14 Emulátor működésének folyamatábrája	26
15 Output fájl.....	34
16 A GUI felülete	36
17 A GUI menüsávja, menüpontjai	38

Függellék

Előfeldolgozás

Assembly kód fordításához szükséges parancs:

```
gcc -no-pie <bemeneti fájl> -o <kimeneti fájl>
```

C kód fordításához szükséges parancs:

```
gcc <bemeneti fájl> -o <kimeneti fájl>
```

Objdump futtatásához szükséges parancs:

```
objdump <bemeneti fájl> -D -M intel > <kimeneti fájl>
```

RAM fájl szintaxis

```
"main: " <hexa> "\n.text\n" <codeline> ".rodata\n" <dataline>
".data\n" <dataline> ".bss\n" <dataline>
```

<hexa>:

<hexa_digit> [<hexa_digit>] ...

<hexa_digit>:

<digit> | "a" | "b" | "c" | "d" | "e" | "f"

<digit>:

"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<dataline>:

<literal> "\t" <hexa_digit> <hexa_digit> "\n"

<codeline>:

<literal> "\t" <instruction> "\t" <machinecode> "\t" <assembly>
"\n"

<instruction>:

<mnemonic> "\t" <operand> "\t" <operand> "\t" <operand>

<mnemonic>:

"mov" | "add" | "sub" | "adc" | "sbb" | "inc" | "dec" | "cmp" |
"ret" | "jmp" | "push" | "pop" | "lea" | "and" | "or" | "xor" |
"not" | "neg" | "imul" | "idiv" | "mul" | "div" | "shl" | "shr"
| "sal" | "sar" | "jz" | "jnz" | "je" | "jne" | "jg" | "jge" |
"jl" | "jle" | "ja" | "jae" | "jb" | "jbe" | "cbw" | "cwb" |
"cdq" | "cdqe" | "nop" | "call" | "sete" | "setne" | "seto" |
"setno" | "setc" | "setnc" | "setz" | "setnz" | "test" | "xchg"
| "rol" | "ror" | "rcr" | "rcl" | "endbr64" | "movsx" | "movzx"
| "leave" | "(bad)"

```

<operand>:
<literal> | <register> | <memory> | "_"

<literal>:
<hexa> | ( "0x" <hexa> )

<register>:
"rax" | "eax" | "ax" | "ah" | "al" | "rbx" | "ebx" | "bx" | "bh"
| "bl" | "rcx" | "ecx" | "cx" | "ch" | "cl" | "rdx" | "edx" | "dx"
| "dh" | "dl" | "rdi" | "edi" | "di" | "dil" | "rsi" | "esi" |
"si" | "sil" | "rbp" | "ebp" | "bp" | "bpl" | "rsp" | "esp" |
"sp" | "spl" | | "r8" | "r8d" | "r8w" | "r8b" | | "r9" | "r9d"
| "r9w" | "r9b" | | "r10" | "r10d" | "r10w" | "r10b" | | "r11"
| "r11d" | "r11w" | "r11b" | | "r12" | "r12d" | "r12w" | "r12b"
| | "r13" | "r13d" | "r13w" | "r13b" | | "r14" | "r14d" | "r14w"
| "r14b" | "r15" | "r15d" | "r15w" | "r15b" | "r16" | "r16d" |
|r16w" | "r16b" | "rip" | "eip" | "ip"

<memory>:
<size> ( <literal> | ( "[" [ <register> ] [ "+" <register> [
"*1" | "*2" | "*4" | "*8" ] ] [ ( "+" | "-" ) <literal> ] "]" )
)

<size>:
"B" | "W" | "D" | "Q"

<machinecode>:
<hexa_digit> <hexa_digit> [ <hexa_digit> <hexa_digit> ] ...

<assembly>:
<character> [ <character> ] ...

<character>:
"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
| "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F"
| "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
"Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "0"
| "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "," |
"[" | "]" | "+" | "-" | "*" | ":" | " "

```

Output fájl szintaxis

```

rip "\t" rax "\t" rbx "\t" rcx "\t" rdx "\t" rdi "\t" rsi "\t"
rsp "\t" rbp "\t" r8 "\t" r9 "\t" r10 "\t" r11 "\t" r12 "\t" r13
"\t" r14 "\t" r15 "\t" CF "\t" OF "\t" SF "\t" ZF "\t" <StackSize>
("\t" <StackBytes_InReverseOrder>)... "\t" <machine_code> "\t "
<assembly_instruction> "\n"

```

Támogatott assembly instrukciók

1. adc	25. jl	49. sal
2. add	26. jle	50. sar
3. addsd	27. jmp	51. sbb
4. and	28. jnc	52. seta
5. call	29. jne	53. setae
6. cbw	30. jnz	54. setb
7. cdq	31. jz	55. setbe
8. cdqe	32. lea	56. setc
9. cmp	33. leave	57. sete
10. cwd	34. loop	58. setg
11. dec	35. mov	59. setge
12. div	36. movabs	60. setl
13. enbdr64	37. movsx	61. setle
14. idiv	38. movzx	62. setnc
15. imul	39. mul	63. setne
16. inc	40. neg	64. setnz
17. ja	41. nop	65. setz
18. jae	42. not	66. shl
19. jb	43. or	67. shr
20. jbe	44. pop	68. sub
21. jc	45. push	69. test
22. je	46. ret	70. xchg
23. jg	47. rol	71. xor
24. jge	48. ror	