Asembler x86_64 – instrukcije opšte namene

— seminarski rad —

Arhitektura i operativni sistemi

Ljubica Peleksić i Milica Kojičić

5. novembar 2013.

Sažetak

Da bi bilo koji procesor mogao da se programira na asemblerskom jeziku, programer mora detaljno poznavati njegovu arhitekturu skupa instrukcija. Shodno tome, ovde ćemo se baviti arhitekturom x86_64 i to: organizacijom memorije, načinima adresiranja, a najvise skupom instrukcija. Instrukcije obično imaju jedan, dva ili tri operanda, kojima se može pristupiti neposrednim, direktnim, registarskim, indeksnim ili nekim drugim načinom adresiranja. Neki računari imaju veliki broj složenih načina adresiranja. U opštem slučaju postoje instrukcije za premeštanje podataka, za binarne i unarne operacije, uključujući i aritmetičke i logičke operacije, instrukcije za grananje, pozivanje procedura, rad u petlji, kao i ulazno-izlazne instrukcije. Tipične instrukcije premeštaju reč iz memorije u registar (ili obrnuto), sabiraju, oduzimaju, množe ili dele sadržaj dva registra ili sadržaj registra i memorijske reči, ili porede sadržaj registara ili memorije. U narednim odeljcima razmotrićemo one koje najviše koristimo pri programiranju u asembleru x86_64.

Sadržaj

1 UVOD

1.1 Asemblerski jezik

Svaki računar ima svoju arhitekturu skupa instrukcija (**Instruction Sat Architecture, ISA**) i ona obuhvata registre, instrukcije i druge mogućnosti vidljive programerima na niskom nivou. Ova arhitektura se naziva mašinski jezik. Program na ovom nivou apstrakcije je lista binarnih brojeva, po jedan za svaku instrukciju, koji saopštavaju koju instrukciju treba izvršiti i sa kojim operandima. Medjutim, veoma je teško programirati sa binarnim brojevima, pa svaki računar ima i svoj asemblerski jezik, simboličku predstavu arhitekture skupa instrukcija. Asemblerski jezik ili jednostavno asembler je niži simbolički jezik, prilagodjen radu računara. Binarni brojevi su sada zamenjeni mnemoničkim oznakamama, tako da je ovaj programski jezik mnogo razumljiviji. Osim kratkih reči (ADD, SUB, MUL) koje se koriste za mašinske instrukcije, asembleri dozvoljavaju korišćenje simboličkih imena za konstante i oznaka koje se odnose na instrukcije i memorijske adrese. Kada se program napisan na asemblerskom jeziku prosledi programu zvanom asembler, on ga pretvara u binarni program koji je pogodan za direktno izvršavanje, na stvarnom hardveru [5].

Zašto asembler?

Asembler je program programa, alat svih alata i on je ozbiljno oružje u rukama pravog programera [5].

Asemblerski jezik je najbliža forma komunikacije izmedju čoveka i mašine. Programi u asembleru se odlikuju mogućnošću slanja direktnih komandi procesoru kao i iskorišćavanju celog dijapazona računarske arhitekture. Koristeći asembler, programer može da tačno prati protok podataka i izvršavanje programa. Programeri često koriste asembler za kritične delove programa, ali i za prepravljanje koda za koji nemamo izvorni kod. Asemblerski kod će najčešće da se izvršava mnogo brže nego programi napisani u ostalim jezicima. Proizvodjači hardvera, kao što su Intel i AMD dodaju nove karakteristike svojim procesorima i većinom je jedini način da im se pristupi preko asemblera [4].

1.2 Arhitektura x86_64

Tokom godina, programeri su koristili x86 asembler da bi pisali kod kritičnih performansi. 64-bitni računari su zamenili 32-bitne pa se i sam asemblerski kod promenio. x64 je ime za 64-bitne nadogradnje, **Intelove** i **AMD**-ove, 32-bitne arhitekture skupa instrukcija. AMD je predstavio prvu verziju x64 koja se zvala x86_64, a zatim su je preimenovali u AMD64. Intel je svoju implementaciju nazvao IA-32e, a zatim EMT64. Postoje neznatne razlike izmedju dve verzije ali većina koda radi na obe verzije. Treba razlikovati ovu arhitekturu od arhitekture IA-64 (64 bita Intel Itanium). x86_64 je takodje poznata kao x64, x86-64 i AMD64 [1].

x86_64 podrzava vece količine virtuelne i fizičke memorije, dozvoljavajući programima da čuvaju veće količine podataka u memoriji. x86_64 takodje ima 64-bitne registre opšte namene i mnoga druga poboljšanja. U potpunosti je unazad kompatibilna sa 16-bitnom i 32-bitnom arhitekturom. Zadržava se set instrukcija 16-bitne i 32-bitne arhitekture, ali uz dodatke. x86 arhitektura je proširila registre u 64-bitne na veoma sličan način na koji su 16-bitni registri proširени u 32-bitne. Adresiranje se ne menja značajno u

odnosu na 32-bitnu arhitekturu, samo se proširuje na 64 bita. Arhitektura x86_64 je korišćena pri implementaciji procesora Pentium 4, Celeron D, Xeon, Pentium Dual-Core, Pentium Extreme Edition, Core 2, Core i3,i5, i7 [1]...

1.3 Organizacija registara i načini adresiranja

U x86 asembleru postoji 9 primarnih registara: EAX, ECX, EDX, EBX, ESP, EIP, EBP, ESI i EDI. Svi oni mogu da čuvaju 32-bitne vrednosti. Prva četiri su registri opšte namene i koriste se (redom) kao akumulator, brojač, registar podataka i bazni registar. Njihova glavna uloga je privremeno skladistenje podataka i adresa tokom izvršavanja programa. Drugih 5 registra su takodje registri opšte namene ali se većinom koriste kao pokazivački i indeksni registri. Prva dva su od posebnog značaja pri izvršavanju programa. Pokazivač steka (ESP) pokazuje na poslednji argument koji je stavljen na stek. Važno je znati da stek raste ka nižim adresama. Drugi veoma važan registar je pokazivač instrukcija (EIP). On pokazuje na narednu instrukciju koju će procesor da izvrši. Kako ovo povezujemo sa x64 asemblerom? Svi ovi registri su, dakle, i dalje prisutni. Oni su, medjutim, prošireni sa maksimalna 32 bita na 64. Kao rezultat, oni su sada poznati kao **RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI i RIP** registri. Drugim rečima, E je zamenjeno sa R. Mada to nije u potpunosti tačno. E u imenima registara je i dalje validno u x64 asembleru. Ono nam omogućuje pristup pristup nižem 32-bitnom delu registra. Isto tako, za RAX, RCX, RDX i RBX registre vazi da se uklanjanjem slova r moze pristupiti najnizim 16 bitova (tako da RAX bude AX npr.) ili nizim od tih 16 preko AL registra, odnosno višim preko AH (dakle isto kao kod 32-bitnih registara). Kao dodatak, ovim 9 registrima, x86_64 arhitektura dodaje registre opšte namene **R8 do R15**. **RFLAGS** registar skladišti flegove; sadržaj mu se menja u zavisnosti od rezultata aritmetičkih operacija. Formiran je dodavanjem viša 32 bita na registar EFLAGS. Takodje je važno napomenuti postojanje registara koji se koriste za operacije u pokretnom zarezu, (eng. floating-point registri). Za skoro svaku instrukciju neophodni su podaci bilo iz memorije, bilo iz registara. Većina instrukcije sadrži dva operanda, odredišni i izvorišni.
Na primer: **MOV RAX,RBX**
Prvi operand je odredišni, a drugi izvorišni. Konstanta može da bude izvorište, tj. leva vrednost ali ne i odredište. Odredjene instrukcije zahtevaju samo jedan operand: postupno uvećavanje, negiranje, pomeranje itd. Na primer u instrukciji MUL samo registar RAX može da bude odredište [4]:
Na primer: **MUL RBX**

Procesor smešta brojeve po sistemu **little endian**, sto znači da se manje značajan deo reči smešta na nižu adresu. Načini adresiranja su zapravo načini na koje instrukcija može da pristupi registrima ili memoriji. Postoji više načina adresiranja. Oni koji se najčešće koriste su [5]:

1. Neposredno
2. Direktno
3. Indirektno registarsko
4. Registarko s pomerajem
5. Registarko indeksno

6. Registarsko indeksno s pomerajem
7. Podrazumevano

Neposredno adresiranje je način adresiranja u kome je operand u instrukciji konstantan bajt ili reč. Na primer:
CMP RAX, 50 (poredi vrednost registra RAX sa konstantom 50)

Direktno adresiranje podrazumeva da je operand u instrukciji adresa podatka. Na primer:
ADD RAX, (20) (dodaje registru RAX vrednost sa memorijske lokacije 20)

Indirektno registarsko znači da se adresa operanda nalazi u nekom od registara RBX, RSI ili RDI. U sva tri slučaja operand se nalazi u segmentu za podatke. Kada se konstanta postavi ispred registra, adresa se pronalazi dodavanjem sadržaja registra konstanti. Ovaj tip se naziva **registarsko adresiranje sa pomerajem**. Zgodan je za adresiranje bitova. Na primer:
SUB [RDI + 4], RAX (bazna adresa se nalazi u RDI, uz pomeraj od 4; od te vrednosti se oduzima sadržaj registra RAX)

Registarsko indeksno podrazumeva da je u RDI početna adresa niza, a brojač u RCX. Na primer:
CMP RSI, [RDI + RCX]
(poredi vrednost registra RSI sa vrednosti na adresi na koju pokazuje RDI+RCX) Poslednja dva tipa se mogu kombinovati u **registarsko indeksno s pomerajem**. Na primer:
SUB [RDI + 4 * RCX + 4], RAX (pomeraj je 4 bajta, bazna adresa se nalazi u RDI, indeks u RCX, a faktor skaliranja je 4; vrednost na ovoj lokaciji se umanjuje za vrednost registra RAX).

Na kraju, neke instrukcije koriste **podrazumevano adresiranje** gde se operand ili operandi podrazumevaju. Na primer:
PUSH RAX (stavlja sadržaj registra RAX na stek tako što smanjuje vrednost registra RSP, a zatim kopira sadržaj registra RAX na lokaciju na koju sada ukazuje RSP. U instrukciji PUSH upotreba registra RSP se podrazumeva).

2 INSTRUKCIJE OPŠTE NAMENE

2.1 Aritmetičke instrukcije

Aritmetičke instrukcije izvršavaju aritmetičke operacije i zahtevaju različit broj operanada, u zavisnosti od same operacije. Prvi operand je odredišni i mora biti registar ili memorijska lokacija. Drugi operand može biti ili memorijska lokacija, registar ili konstantna vrednost. Barem jedan od ova mora biti registar zato što operacije ne smeju da koriste memorijske lokacije kao oba operanda [2].

1. Instrukcija **ADD** vrši **sabiranje** i ima dva operanda:
   ADD RAX, 4 (sabiramo vrednost registra RAX sa konstantom 4

i rezultat će biti smešten u registar RAX)

2. Instrukcija **SUB** vrši **oduzimanje** i ima dva operanda:
   SUB RBX, X (oduzimamo vrednost sa memorijske lokacije X od vrednosti registra RBX i rezultat smeštamo u RBX)

3. Instrukcija **ADC** vrši **sabiranje s prenosom** i ima dva operanda:
   ADC RAX, 4 (sabiranje sa prenosom, vrednost eng. carry flag-a se dodaje na rezultat sabiranja)

4. Instrukcija **SBB** vrši **oduzimanje s pozajmicom** i ima dva operanda:
   SBB RBX, X (oduzimanje sa pozajmicom, vrednost eng. carry flag-a se oduzima od razlike operanada)

5. Razlikujemo dve instrukcije za mnozenje: **MUL** za neoznačene i **IMUL** za označene brojeve. Instrukcija MUL ima samo jedan operand i taj operand se množi sa sadržajem registra AL, AX, EAX ili RAX (u zavisnosti od širine operanda). Rezultat se smešta u registre AX, DX:AX, EDX:EAX, RDX:RAX (dvotačka označava da se prvi registar odnosi na više bitove podataka). IMUL može imati jedan operand i u tom slučaju je analogna instrukciji MUL, ali može imati i dva i tri operanda. Na primer:
   MUL RBX (sadržaj registra RAX se množi sa sadržajem registra RBX i rezultat se smešta u RDX:RAX)

6. Što se deljenja tiče, ovde takodje imamo dve instrukcije: **DIV** se odnosi na deljenje neoznačenih, a **IDIV** na deljenje označenih brojeva. Instrukcija DIV ima jedan operand. Sadržaj registra AX, DX:AX, EDX:EAX, RDX:RAX se deli vrednošcu operanda koja je 8-bitna, 16-bitna, 32-bitna i 64-bitna i količnik se smešta (redom) u: AL, AX, EAX i RAX, a ostatak pri deljenju u AH, DX, EDX I RDX. IDIV je analogna instrukciji DIV i nema ekstenzije kao IMUL. Na primer:
   DIV RCX (sadržaj registra RDX:RAX se deli sadržajem registra RCX i količnik se smešta u RAX a ostatak u RDX)

7. Instrukcija **NEG** vrši **negaciju** i ima jedan operand kome menja znak. Na primer:
   NEG X (vrednosti na memorijskoj lokaciji x je promenjen znak)

8. Instrukcije **INC** i **DEC** vrše (redom) **uvecavanje i smanjivanje argumenta za jedan** i imaju po jedan operand. Inkrementacija radi mnogo brže nego **ADD ARG, 1**. Isto tako, dekrementacija radi mnogo brže nego **SUB ARG, 1**. Na primer:
   INC RDX (uvećava vrednost registra RDX za 1)
   DEC RDX (smanjuje vrednost registra RDX za 1)

2.2 Logičke instrukcije

Logičke instrukcije izvode logičke operacije nad operandima. Sve osim bitske negacije prihvataju dva operanda, dok negacija prihvata jedan. Razlikujemo sledeće bitovske logičke operacije [2]:

1. **Bitovska konjukcija (AND): AND RDX, RCX**
2. **Bitovska disjunkcija (OR): OR RDX, RCX**
3. **Bitovska ekskluzivna disjunkcija (XOR): XOR RDX, RCX**
4. **Bitovkska negacija (NOT): NOT RDX**

2.3 Instrukcije za šiftovanje i rotacije

Sve instrukcije za šiftovanje i rotacije prihvataju po jedan operand [2].

1. **Logičke instrukcije za šiftovanje**
   Logičko šiftovanje se koristi za neoznačene brojeve. U slučaju logickog pomeranja udesno, krajnje desni bit se zanemaruje, a početni bit rezultata se uvek popunjava nulom, a u slučaju logičkog pomeranja ulevo, početni bit argumenta se zanemaruje, a na završna mesta se upisuje nula.
   (a) **Šiftovanje udesno (SHR): SHR RDX**
   (b) **Šiftovanje ulevo (SHL): SHL RDX**
2. **Aritmetičke instrukcije za šiftovanje**
   Artimetičko pomeranje se koristi za označene brojeve u potpunom komplementu. U slučaju aritmetičkog pomeranja udesno, krajnje desni bit se zanemaruje, a početni bit se popunjava vodećim bitom argumenta koji predstavlja znak. A što se tiče pomeranja ulevo, početni bit argumenta se zanemaruje, dok se na završno mesto upisuju nule.
   (a) **Šiftovanje u desno (SAR): SAR RAX**
   (b) **Šiftovanje u levo (SAL): SAL RAX**
3. **Instrukcije za šiftovanje sa prenosom**
   Primenjujemo uobičajeno logičko šiftovanje, s tim što se bitovi koji se inače zanemaruju, šalju u eng. carry flag. Primer:
   (a) **Šiftovanje u desno sa prenosom (SCR): SCR RAX**
   (b) **Šiftovanje u levo sa prenosom (SCL): SCL RAX**
4. **Instrukcije za rotaciju**
   Rotacija udesno je šiftovanje udesno, s tim što bit koji gubimo sa početka vraćamo na kraj, dok je rotacija ulevo siftovanje ulevo, sa vraćanjem izgubljenog na kraj.
   (a) **rotacija udesno (ROR): ROR RAX**
   (b) **rotacija ulevo (ROL): ROL RAX**

2.4 Instrukcije za transfer podataka

Kopiranje podataka sa jednog mesta na drugo je najosnovnije od svih operacija. Pod kopiranjem podrazumevamo stvaranje novog objekta koji ima identičan raspored bitova kao original. Instrukcije za premeštanje podataka bi bilo bolje zvati instrukcije za dupliranje podataka, ali se izraz premeštanje podataka već previše odomaćio. Postoje dva razloga zbog kojih se podaci kopiraju s jednog mesta na drugo. Jedan je dodeljivanje vrednosti promenljivama, a drugi razlog je efikasnije pristupanje i korišćenje (brojne instrukcije mogu da pristupe promenljivama samo ako se one nalaze u registrima) [2].

1. Instrukcija **MOV** kopira vrednost drugog operanda u prvi, pri čemu se nijedan fleg ne menja. Na primer:
   MOV RAX, 6 (kopiramo konstantu 6 u registar RAX)
2. Instrukcija **XCHG** za zamenu podataka zamenjuje vrednosti prvog i drugog operanda, pri čemu se takodje nijedan fleg ne menja. Na primer:
   XCHG RAX, RBX (zamenjujemo vrednosti u registrima RAX i RBX)
3. Takodje postoje instrukcije koje pri kopiranju drugog operanda u prvi, nepopunjena mesta u prvom argumentu popunjavaju nulama (instrukcija je korisna za kopiranje malih neoznačenih vrednosti u veći registar) ili znakom drugog argumenta (instrukcija je korisna za kopiranje malih označenih brojeva). Na primer:
   MOVZ RAX, 5 (prvi slučaj)
   MOVS RAX, 5 (drugi slučaj)
4. Instrukcija **MOVSB** nema nijedan operand i kopira jedan bajt sa lokacije koja se nalazi u RSI registru na lokaciju koja se nalazi u RDI. Instrukcija takodje ne menja flegove.
5. **MOVSW** je instrukcija koja nema nijedan operand i kopira dva bajta (jednu reč) iz RSI u RDI.

2.5 Instrukcije za kontrolu toka

U skoro svim programskim jezicima može da se pri izvršavanju instrukcija menja njihov redosled, pa ni asembler nije izuzetak. Pokazivač instrukcija (RIP) registar sadrži adresu naredne instrukcije koja će da se izvrši. Da bi se tok kontrole promenio, programer mora da modifikuje vrednost registra RIP. Upravo tu, glavnu ulogu igraju instrukcije za kontrolu toka [2].

1. **Instrukcije za poredjenje**
   (a) **TEST** je instrukcija koja izvršava bitovsku konjukciju nad dva operanda i postavlja flegove, ali ne čuva rezultat Na primer:
   TEST RAX, RBX
   (b) **CMP** je instrukcija koja vrši oduzimanje izmedju dva operanda, postavlja flegove, ali ne čuva rezultat. Na primer:
   CMP RAX, RBX
2. **Skokovi**
   (a) **Bezuslovni skok**
   Instrukcija **JMP** postavlja u registar RIP adresu naredne instrukcije koja se izvršava. Na primer:
   JMP LOC (skače na izvršavanje instrukcije koja se nalazi na adresi LOC)

   (b) **Skok ako je jednako**
   Instrukcija **JE** postavlja u RIP adresu naredne instrukcije, ako su operandi prethodne CMP instrukcije jednaki.
   MOV RCX, 5
   MOV RDX, 5
   CMP RCX, RDX
   JE EQUAL
   # ako se ne skoči na labelu EQUAL, onda to znači da 5 i 5 nisu jednaki.

EQUAL:
ako se skočilo ovde, onda to znači da su 5 i 5 jednaki.

(c) **Skok ako nije jednako**
Instrukcija **JNE** stavlja u RIP registar adresu naredne instrukcije, ako operandi prethodne CMP instrukcije nisu jednaki.

(d) **Skok ako je veće**

i. Instrukcija **JG** u registar RIP stavlja adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije veći od drugog. (izvodi se poredjenje označenih brojeva).

ii. Instrukcija **JGE** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije veći ili jednak drugom (izvodi se poredjenje označenih brojeva).

iii. Instrukcija **JA** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije veći od drugog. (izvodi se poredjenje neoznačenih brojeva).

iv. Instrukcija **JAE** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije veći ili jednak drugom (izvodi se poredjenje neoznačenih brojeva).

(e) **Skok ako je manje**

i. Instrukcija **JL** u registar RIP stavlja adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije manji od drugog. (izvodi se poredjenje označenih brojeva).

ii. Instrukcija **JLE** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije manji ili jednak drugom (izvodi se poredjenje označenih brojeva).

iii. Instrukcija **JB** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije manji od drugog. (izvodi se poredjenje neoznačenih brojeva).

iv. Instrukcija **JBE** stavlja u RIP adresu naredne instrukcije, ako je prvi operand prethodne CMP instrukcije manji ili jednak drugom (izvodi se poredjenje neoznačenih brojeva).

(f) **Skok ako postoji prekoračenje**
Instrukcija **JO** u registar RIP stavlja adresu naredne instrukcije, ako je postavljen bit prekoračenja pri prethodnoj aritmetičkoj operaciji.

(g) **Skok u slučaju nule**

i. Instrukcija **JNZ** u registar RIP stavlja adresu naredne instrukcije, ako nije postavljen eng. zero flag pri prethodnoj aritmetičkoj operaciji. JNZ instrukcija je identična instrukciji JNE.

ii. Instrukcija **JZ** stavlja u RIP adresu naredne instrukcije,ako je postavljen eng. zero flag pri prethodnoj aritmetičkoj operaciji. JZ je identična JE.

3. **Instrukcije za poziv procedura**
   Instrukcija **CALL** bezuslovno skače na adresu koja je zadata kao operand, ali prethodno stavlja na vrh steka adresu povratka iz RIP registra.
   Instrukcija **RET** završava potrogram, ona skida sa steka povratnu adresu, stavlja je u RIP registar i vrši bezuslovan skok na tu adresu. Vrlo je važno voditi računa da ako se na stek postavljaju neki drugi podaci, da se oni pre poziva RET instrukcije skinu sa steka, kao i da se greškom ne izmeni sadržaj povratne adrese na steku.

4. **Instrukcije brojačke petlje**
   Sve instrukcije primaju jedan argument koji je adresa memorijske lokacije, umanjuju vrednost registra RCX i skaču na zadatu adresu u zavisnosti od uslova.

   (a) Instrukcija **LOOP**
   Ima samo jedan argument koji predstavlja adresu memorijske lokacije. Ona umanjuje vrednost registra RCX za jedan i skače na zadatu adresu sve dok vrednost registra RAX ne bude nula. LOOP ne postavlja flegove. Na primer:

   MOV RCX, 5
   START_LOOP:
   #kod bi se ovde izvršavao 5 puta
   LOOP START_LOOP:

   (b) Instrukcija **LOOPX**
   Ove LOOP instrukcije umanjuju vrednost registra RCX i skače na adresu zadatu operandom ako je njihov uslov zadovoljen, tj. ako je postavljen odgovarajući fleg.
   i. **LOOPE** skače ako je jednako
   ii. **LOOPZ** skače ako je nula
   iii. **LOOPNE** skače ako nije jednako
   iv. **LOOPNZ** skače ako nije nula

5. Instrukcije **ENTER** i **LEAVE**
   Instrukcija **ENTER** pravi stek okvir sa odredjenom količinom prostora alociranom na steku. Obično se taj prostor rezerviše za lokalne promenljive koje se koriste u procedurama. Na primer:
   ENTER 0,0 (u ovom slučaju se ne rezervise prostor za lokalne promenljive na steku)
   Instrukcija **LEAVE** uništava trenutni stek okvir i restaurira prethodni okvir. Na primer:
   MOV RSP, RBP
   POP RBP

   Sadržaj RSP registra je adresa vrha steka, ali se on tokom rada funkcije može promeniti, sto nam u tom slučaju otežava adresiranje argumenata funkcije. Standardni način pristupa argumentima funkcije

je preko registra RBP. Po ulasku u funkciju, ovaj registar se stavlja na stek (da bi se njegova vrednost sačuvala) i u njega se kopira vrednost registra RSP, tj. adresa vrha steka će nam se sad nalaziti i u RBP registru. Sadržaj RBP registra se posle ne menja u toku rada funkcije i preko njega mi pristupamo njenim argumentima. Na vrhu steka je sadržaj RBP registra, ispod njega je adresa povratka iz funkcije, a onda redom argumenti funkcije kojima se pristupa tako što na adresu RBP dodamo odredjen broj bajtova u zavisnosti gde se na steku ti argumenti nalaze (jer stek raste ka nižim adresama). Npr. ako hoćemo da pristupimo prvom argumentu funkcije dodajemo 8 bajtova na RBP, za drugi argument dodajemo 12 bajtova itd. Na kraju funkcije mora se skinuti sadržaj registra RBP sa steka i tek onda može da se pozove RET instrukcija. Gore navedenim naredbama postavili smo sadržaj RBP registra u registar RSP (u kom se čuva adresa vrha steka) i skinuli registar RBP sa steka kako bi RET instrukcija skinula povratnu adresu i kako bi program mogao da nastavi da se izvršava od te adrese.

6. **Druge instrukcije kontrole**
   (a) **HLT** instrukcija zaustavlja procesor
   (b) **NOP** znači da nema operacije. Ova instrukcija ne radi nista, tj. samo troši instrukcioni ciklus.Često je zamenjuje instrukcija: **XCHG RAX, RAX.**
   (c) **LOCK** instrukcija ističe procesorov signal LOCK koji je istaknut tokom operacije koja ide uz instrukciju.
   (d) **WAIT** instrukcija čeka da procesor završi svoju poslednju operaciju.

2.6 Instrukcije za rad sa stekom

Instrukcije primaju po jedan argument, osim instrukcija PUSHA i POPA koje nemaju argumente [2].

1. **PUSH**
   Ova instrukcija smanjuje za 1 pokazivač na stek i podatak koji se nalazi u argumentu stavlja na lokaciju na koju pokazuje pokazivač na stek.

2. **POP**
   Ova instukcija stavlja podatke koji se nalaze na lokaciji na koju pokazuje pokazivač na stek u argument, a zatim povecava pokazivač na stek za 1.

   MOV RAX, 5
   MOV RBX, 6
   PUSH RAX (posle ove instrukcije, stek bi bio: [5])
   PUSH RBX (posle ove instrukcije, stek bi bio: [6] [5])
   POP RAX (ova instrukcija vrednost koja je poslednja stavljena na stek[6], kopira u RAX. Posle ove operacije, stek bi bio: [5])
   POP RBX (vrednost registra RBX bi bila 5, a stek bi bio prazan)

3. **PUSHF**
   Ova instrukcija smanjuje pokazivač na stek za 1, a zatim na lokaciju na koju pokazuje pokazivač na stek, stavlja sadržaj fleg registra.

4. **POPF**
   Ova instrukcija u fleg regitar stavlja sadrzaj memorijske adrese na koju pokazuje pokazivač na stek, a zatim pokazivač na stek uvećava za 1.
5. Instrukcijama **PUSHA** i **POPA** se mogu brzo sačuvati svi registri opšte namene

2.7 Instrukcije za rad sa fleg-ovima

Dok fleg registar služi da prijavi rezultat izvršenih instrukcija (prekoračenje, prenos itd.), takodje sadrži i flegove koji utiču na operacije procesora. Ovi flegovi bivaju postavljeni i poništeni posebnim instrukcijama [2].

1. **Fleg za prekide** govori procesoru treba li ili ne da prihvati hardverski prekid.
   (a) **STI**
   Ova instrukcija postavlja fleg za prekide; u tom slučaju procesor prihvata zahtev za prekid.
   (b) **CLI**
   Ova instrukcija poništava fleg za prekide.
2. **Fleg za prenos** se obično postavlja nakon aritmetičkih instrukcija, ali može biti postavljen ili poništen i ručno.
   (a) **STC**
   Ova instrukcija postavlja fleg za prenos.
   (b) **CLC**
   Ova instrukcija poništava fleg za prenos.
   (c) **CMC**
   Ova instrukcija komplementira fleg za prenos.
3. **Ostali flag-ovi**
   (a) **SAHF**
   Ova instrukcija čuva sadržaj AH registra u nižim bajtovima fleg registra.
   (b) **LAHF**
   Ova instrukcija stavlja u AH registar ono što se nalazi u nižim bajtovima fleg registra.

2.8 Ulazno-izlazne instrukcije

Instrukcije primaju po dva operanda [2].

1. **IN**
   Instrukcija IN skoro uvek radi sa operandima AX i DX (tj. EAX, EDX ili RAX, RDX). DX često drzi adresu porta za čitanje, a AX prima podatke od porta. U zaštićenom režimu operativnih sistema, IN instrukcija je zaključana, pa je obični korisnici ne mogu koristiti.

2. **OUT**
   Instrukcija OUT je veoma slična instrukciji IN. OUT stavlja podatke iz registra koje je u prvom operandu, na port dat u drugom operandu. U zaštićenom režimu, instrukcija OUT je zaključana, pa ni nju obični korisnici ne mogu koristiti.

2.9 Sistemske instrukcije

1. **SYSENTER**
   Ova instrukcija uvodi procesor u zaštićeni režim.
2. **SYSEXIT**
   Ova instrukcija izvlači procesor iz zaštićenog režima i uvodi ga u režim za korisnika.

2.10 Instrukcije za generisanje prekida

Sistem prekida predstavlja jedan od mehanizama za upravljanje tokom rada procesora. U osnovne uloge sistema prekida spadaju komunikacija sa U-I uredjajima i upotreba osnovnih usluga operativnog sistema. Prekidi se mogu izazvati hardverski ili softverski. Kod procesora x86 **INT** je instrukcija koja poziva odgovarajuću prekidnu proceduru. Ima jedan operand - broj prekida. Kada se opsluživanje prekida dovrši, program nastavlja sa izvršavanjem od mesta na kom je prekinut [2]. Na primer: **INT 0x0A** (poziva prekid broj 10)

3 ZAKLJUČAK

Pri samom učenju asemblera, ono od čega krećemo jeste zapravo učenje o skupu instrukcija. Većina x86 instrukcija je validna u 64-bitnoj nadogradnji, s tim da su neke retko korišćene instrukcije izbačene iz upotrebe, a pridodate su nove kao na primer [3]:

1. **CDQE** koja konvertuje eng. dword (EAX) u eng. qword (RAX).
2. **CQO** koja konvertuje eng. qword (RAX) u eng. oword (RDX:RAX).
3. **MOVSQ** koja kopira qword iz RSI registra u RDI.
4. **CMPSQ** koja poredi qword iz RDI sa RAX registrom.
5. **SCASQ** koja učitava qword u RDI, a zatim uporedjuje qword iz RDI sa RAX registrom.
6. **LODSQ** koja popunjava RAX registar sa qword-om iz RSI registra.
7. **STOSQ** koja skladišti qword u RDI iz RAX registra.
8. **MOVSXD** koja proširuje eng. dword u qword tako što preostala mesta popunjava znakom dword-a.
9. **JRCXZ** koja skače ako je RCX nula.
10. **POPFQ** koja skida sa steka RFLAGS registar.
11. **PUSHFQ** koja stavlja na stek RFLAGS registar.
12. **IRETQ** 64-bitni povratak iz prekida.

Srce svakog računara je skup instrukcija koje može da izvršava. Da bismo stvarno razumeli računar, neophodno je da prvo dobro razumemo njegov skup instrukcija. Nije retko da ih računar u svom repertoaru ima i preko 200. Ovde su prikazane one koje se najčešće koriste i predstavljaju dobru osnovu za dalje i detaljnije bavljenje istim.

Literatura

[1] x86-64. on-line at: http://en.wikipedia.org/wiki/X86-64.

[2] x86 Assembly. on-line at: http://en.wikibooks.org/wiki/X86_Assembly.

[3] x86 instruction listings. on-line at: http://en.wikipedia.org/wiki/X86_instruction_listings.

[4] Chris Lomont. Introduction to x64 Assembly, 2010. on-line at: http://software.intel.com/en-us/articles/introduction-to-x64-assembly.

[5] Andrew Tanenbaum. *Arhitektura i organizacija računara*. Mikro knjiga, Beograd, 2007. http://www.mikroknjiga.rs/store/prikaz.php?ref=978-86-7555-314-4.