SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Diplomski rad br. 1878

DINAMIČKO GENERIRANJE I IZVOĐENJE STROJNOG KODA IZ PYTHONA

Mario Vidov

Zagreb, prosinac 2010.

Sadržaj

Uvod

Procesori se vrlo brzo razvijaju i dobivaju razne nove mogućnosti. Jedna od interesantnijih mogućnosti današnjih procesora je izvođenje vektorskih instrukcija. Vektorskim instrukcijama omogućeno je procesiranje većeg broja podataka odjednom. Moderni prevoditelji ne uspijevaju optimalno iskoristiti te vektorske instrukcije i to je jedna od glavnih činjenica iz kojih je proizašla motivacija za razvoj asemblera TDAsm koji je opisan u ovom radu. TDAsm omogućava da se asemblerski odsječci dinamički prevode u strojni kod i izvršavaju. Takvim pristupom možemo optimizirati kritične dijelove koda i umetnuti ih u aplikaciju iz visoko produktivnog okruženja.

U poglavlju 1. prikazan je kratki povjesni pregled što se tiče uvođenja vektorskih instrukcija za arhitekturu x86. Poglavlje 2, posvećeno je biblioteci CorePy koja također omogućava dinamičko generiranje i izvršavanje strojnog koda iz Pythona. U poglavlju 3. objašnjena je arhitektura asemblera TDAsm, te prevođenje i izvršavanje asemblerskih odsječaka koristeći biblioteku TDAsm. U poglavlju 4. prikazani su izvedbeni detalji asemblera TDAsm. U poglavlju 5. razmatrani su eksperimentalni rezultati dobiveni testiranjem implementacija algoritama koristeći biblioteke CorePy i TDAsm te programski jezik C++.

1. Vektorska instrukcijska proširenja arhitekture x86

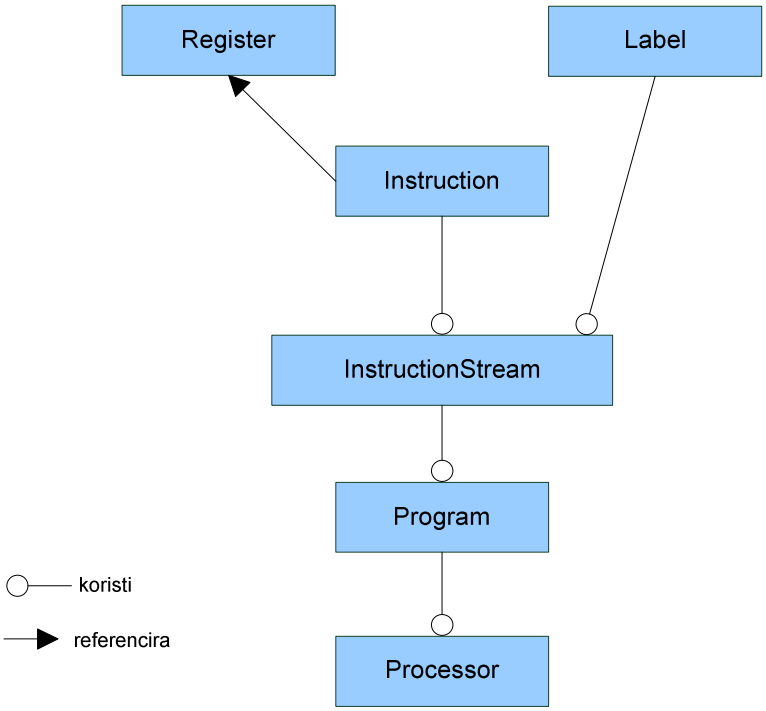
Slijedi kratak povijesni pregled arhitekture x86 što se tiče uvođenja vektorskih instrukcijskih setova. Intel je 1997. godine procesor Pentium proširio sa vektorskim instrukcijskim setom MMX. Vektorski instrukcijski set MMX uveo je vektorske registre MM0-MM7 koji su 64-bitni i podržavaju pakirane podatke što znači da je jedan 64-bitni registar mogao sadržavati jedan 64-bitni broj, dva 32-bitna broja, četri 16-bitna broja ili osam 8-bitnih brojeva, podržani su samo cjelobrojni tipovi podataka. Registri MM0-MM7 bili su dijeljeni sa registrima FPU jedinice. 1999 godine uveden je vektorskiinstrukcijski set SSE koji je uveo podršku za operacije s pomičnim zarezom i procesor je dobio 8 dodatnih 128-bitnih registra XMM0-XMM7. Godinu dana poslije uveden je vektorski instrukcijski set SSE2 koji je donio podršku za brojeve sa pomičnim zarezom dvostruke preciznosti. Početkom 2004 uveden je SSE3 instrukcijski set koji je donio podršku za horizontalno manipuliranje vektorima u odnsu na prijašnje instrukcijske setove koji su samo radili vertikalno. Intel je jos uveo SSSE3, SSE4.1 i SSE4.2 instrukcijske setove koji su samo dodali još instrukcija za baratanje XMM0-XMM7 registrima. AMD u svoje procesore nikad nije ugradio podršku za SSSE3, SSE4.1 i SSE4.2 tako da oni nikad nisu postali popularni i većina današnjih aplikacija se optimizira za SSE2. Početkom sljedeće godine pojaviti će se AVX instrukcijski set koji proširuje registre na 256 bita, instrukcije sa 3 operanda i razna druga poboljšanja. I Intel i AMD prihvatili su taj instrukcijski set. Procesor u 64-bitnom modu ima dodatnih osam 128-bitnih registra tako da imamo na raspolaganju registre XMM0-XMM15.

2. Biblioteka CorePy

U ovom poglavlju prikazati ćemo biblioteku CorePy koja također omogućava umetanje asemblerskih odsječaka u Python aplikaciju. Najveća razlika između CorePy i asemblera opisanog u ovom radu je način na koji se piše asemblerski kod. Da bismo mogli koristiti CorePy potrebno je poznavati njegovu arhitekturu koja je objašnjena u poglavlju 2.1. Nakon toga u poglavlju 2.2 biti će prikazano kako pisati asemblerski kod pomoću CorePy. U poglavlju 2.3 koristeći biblioteku CorePy prikazana je implementacija LCG generatora slučajnih brojeva.

2.1. Arhitektura biblioteke CorePy

Biblioteka CorePy sastoji se od kolekcije klasa koje su koncipirane tako da svaka pojedina klasa predstavlja neki dio procesorske arhitekture. Na slici 1. vidimo osnovne klase od kojih se sastoji biblioteka CorePy .



Slika 1. Arhitektura biblioteke CorePy. Processor izvodi program. Program koristi InstructionStream. InstructionStream se sastoji od instrukcija i labela. Instrukcija sadrži registre i konstante.

Razred `Instruction` predstavlja jednu instrukciju. Da bismo mogli instancirati taj objekt potrebni su nam i operandi instrukcije. Operandi mogu biti registri, labele, zastavice ili konstante. Za registre i labele koriste se odgovarajući razredi a za konstante i zastavice koriste se osnovni tipovi Pythona. Osim kao operand, labela se još koristi i za grananje u objektu tipa `InstructionStream`.

Prilikom instanciranja objekta tipa `Instruction` obaviti će se provjere da li su predani odgovarajući tipovi operanda i ako provjera nije uspješno prošla (npr. predali smo registar gdje smo trebali labelu ili smo predali konstantu sa prevelikom vrijednošću) generirati će se iznimka.

Pomoću metode `add` dodajemo instrukcije i labele u objekt razreda `InstructionStream` koji predstavlja naš odsječak koda. `InstructionStream`

objekti se mogu i spajati u jedan veći čime su olakšane neke optimizacije kao npr. odmotavanje petlje (loop unrolling).

Za prevođenje instrukcija u strojni kod zadužen je objekt tipa `Program` koji sadrži jedan ili više `InstructionStream`-ova. Prevođenje obavlja metoda `cache_code` koja ujedno i osigurava da generirani strojni kod bude u skladu sa arhitekturom i aplikacijskim binarnim sučeljem (ABI-application binary interface) za svaki operacijski sustav. Dobiveni niz bajtova kopira se u memorijski prostor koji može sadržavati izvršni kod. Pomoću ekstenzije napisane u C-u konvertiramo adresu gdje se nalazi izvršni kod u pokazivač na funkciju i taj pokazivač poslije koristimo za pokretanje tog strojnog koda.

Za pokretanje generiranog strojnog koda koristimo metodu `execution` objetka tipa `Procesor` kojoj ujedno prosljeđujemo parametre i dohvaćamo povratnu vrijednost.

Podržan je i asinkroni način pokretanja koda u kojem se strojni kod pokrene u novoj dretvi. U asinkronom načinu rada sa metodom `join` blokiramo dretvu i čekamo da se završi izvršavanje strojnog odsječka kojeg smo pokrenuli u drugoj dretvi.

Sve arhitekture koje podržava CorePy imaju neki set vektorskih instrukcija. Vektorske instrukcije obično zahtijevaju striktno poravnavanje memorije. Zbog toga CorePy ima pomoćnu klasu `ExtArray` koja garantira da će memorija biti poravnata. Razred `ExtArray` je dizajniran tako da imao isto sučelje kao i modul `array`.

2.2. Programiranje u CorePy

Sad kad smo se upoznali s osnovnom hijerarhijom klasa koje nam nudi CorePy za izradu asemblerskih programa prikazati ćemo korištenje tih klasa na nekoliko jednostavnih primjera. Svi primjeri biti će pisani za instrukcijsku arhitekturu x86-64.

CorePy podržava više procesorskih arhitektura tako da prvi korak koji moramo napraviti je importirati odgovarajuće okruženje.

```
# importiramo okruženje za x86-64 bitnu arhitekturu
import corepy.arch.x86_64.isa as x86
import corepy.arch.x86_64.platform as x86_env
```

Slijedeći korak je instanciranje objekta tipa `Program` koji predstavlja asemblerski odječak koji želimo umetnuti u aplikaciju. Sa metodom `get_stream` dobivamo instancu objekta tipa `InstructionStream` koji će sadržavati naš niz instrukcija. Objekt tipa `Program` može sadržavati više objekata tipa `InstructionStream`.

```
prgm = x86_env.Program()
code = prgm.get_stream()
```

Dodavanje instrukcija u `InstructionStream` vrši se pomoću metode `add`. Nakon što dodamo sve instrukcije u `InstructionStream`, pomoću operatora `+=` dodajemo te instrukcije u naš program.

```
# program ima samo jednu mov instrukciju
code.add(x86.mov(prgm.gp_return, 56))
prgm += code
```

Da bismo izvršili taj naš program moramo instancirati objekt tipa `Procesor` i pozvati metodu `execute`. Parametar `mode` u `execute` metodi označava tip povratne vrijednosti metode i osim vrijednosti ″int″ možemo staviti ″float″.

```
proc = x86_env.Processor()
result = proc.execute(prgm, mode='int')
print result
```

Zbog kompletnosti prikazati ćemo primjer u cijelosti da se jasnije vidi što je sve potrebno za minimalni primjer koji je savršeno funkcionalan iako ima samo jednu instrukciju i ne radi ništa pametno.

```
import corepy.arch.x86_64.isa as x86
import corepy.arch.x86_64.platform as x86_env
prgm = x86_env.Program()
code = prgm.get_stream()
code.add(x86.mov(prgm.gp_return, 56))
prgm += code
proc = x86_env.Processor()
result = proc.execute(prgm, mode='int')
print result
```

Da bismo pisali imalo složenije programe moramo znati manipulirati sa registrima, labelama i još nekoliko drugih elemenata koji su bitni za pisanje asemblerskih programa. Svaka arhitektura procesora ima različite registre i svaki programer koji programira u asembleru mora znati koje registre ima na raspolaganju. Arhitektura x86 ima 8 registara opće namjene, 8 registara za brojeve sa pomičnim zarezom i 8 vektorskih SSE registara. Arhitektura x86-64 povećala je broj registra opće namjene i vektorskih registara na 16. Da bismo dobili objekt koji predstavlja registar, razred `Program` pruža metodu `acquire_register` gdje mu kao parametar damo koji tip registra želimo. Broj registara je ograničen i ako zatražimo previše registara generirati će se iznimka. Kad završimo sa registrom moramo ga i osloboditi a to radimo sa metodom `release_register`.

```
# prgm je instanca Program objekta
r = prgm.acquire_register("gp64")
x86.cmp(r, 1) # primjer korištenja registra
prgm.release_register(r) # oslobađamo registar
```

Za arhitekturu x86 uobičajeno je registre koristiti po imenu jer neke instrukcije implictno koriste određene registre. Zbog toga corepy za arhitekturu x86 omogućava korištene registara po imenu i u daljnjim primjerima koristiti će se taj mehanizam za manipulaciju registrima.

```
from corepy.arch.x86_64.types.registers import *
x86.cmp(rax, 1) # rax, rbx, rcx, rdx, itd.... imamo na raspolaganju
```

Da bismo mogli imalo složenije programe pisati potrebna nam je neka vrsta grananja. U asembleru se za grananje koriste labele. Pozivom `get_label` metode nad objektom `Program` dobivamo labelu ali labela jos uvijek nije dio programa nego je moramo dodati sad `add` metodom kao i ostale instrukcije.

```
lbl_loop = prgm.get_label("LOOP")
code.add(lbl_loop)
...
...
code.add(x86.jmp(lbl_loop)) #skok na labelu LOOP
```

Pristup memoriji je učestaliji kod arhitekture x86 jer imamo mali broj registara na raspolaganju za pohranu privremenih vrijednosti, zbog toga nam x86 arhitektura dopušta da imamo jedan memorijski operand a drugi operand je registar za razliku od RISC arhitekture gdje operandi obavezno moraju biti registri. Imamo različite vrste adresiranja memorijskih operanada. Npr. možemo koristiti apsolutno adresiranje.

```
x86.mov(rax, MemRef(0xDF2DBCEF))
x86.add(MemRef(0xC2F4B62E), rax))
```

Najčešće se ipak koristi SIB (scale-index-base) adresiranje. Bazni registar je obavezan a indeksni registar je opcionalan koji se još množi sa skalarom koji može biti 1, 2, 4 ili 8.

Uz to još možemo koristiti i pomak koji iznosi 0 ako se ne navede. Na kraju za računanje efektivne adrese dobijemo formulu Base + (Index*Scale) + offset.

```
x86.div(MemRef(rbp, 64))                                # Adresa = rbp + 64
x86.and_(rdx, MemRef(rdi, index = r15))                 # Adresa = rdi + r15
x86.shl(MemRef(rsi, 0, rcx, 4, data_size = 32), 1)# Adr = rsi + (rcx * 4)
```

Prvi argument za MemRef je uvijek bazni registar a ostali argumenti su opcionalni. Ako ne navedemo imenovane argumente iza baznog slijedi ofset, indeksni registar i skalar. Kad navodimo memorijske operande ako ne odgovaraju podrazumijevanoj širini moramo eksplicitno navesti širinu kao u prethodnom primjeru. Podrazumijevana širina kod 32-bitne arhitekture je 32 a kod 64-bitne 64, znači kad budemo referencirali SSE registre koji su široki 128-bita uvijek ćemo morati eksplicitno postaviti parametar `data_size`. Kod 64-bitne arhitekture uvedeno je još relativno adresiranje. RIP je registar koji sadrži adresu slijedeće instrukcije koju treba izvesti. Efektivna adresa dobiva se pribrojavanjem 32-bitnog pomaka registru RIP.

```
x86.mov(r12, MemRef(rip, -286)) # r12 = *(rip - 286)
```

Parametri iz Pythona u Corepy prenose se pomoću stoga. Maksimalno se može prenijeti 8 parametara. U asembleru parametre dohvaćamo preko registra rbp. Na adresi rbp + 16 nalazi se prvi parametar a na adresi rbp + 72 posljednji. Objekt tipa `ExecParams` predajemo metodi `execute`.

```
params = env.ExecParams()
params.p1 = 1
params.p2 = 2
params.p3 = 3
params.p4 = 4
params.p5 = 5
params.p6 = 6
```

```
params.p7 = 7
params.p8 = 8


x86.mov(rax, MemRef(rbp, 16))
x86.add(rax, MemRef(rbp, 24))
x86.add(rax, MemRef(rbp, 32))
x86.add(rax, MemRef(rbp, 40))
x86.add(rax, MemRef(rbp, 48))
x86.add(rax, MemRef(rbp, 56))
x86.add(rax, MemRef(rbp, 64))
x86.add(rax, MemRef(rbp, 72))
```

Svaki program možemo pokrenuti sinkrono ili asinkrono. Ako program pokrenemo asinkrono da bismo blokirali trenutnu aktivnu dretvu i pričekali da se program izvrši u cijelosti moramo pozvati join metodu. U asinkronom načinu rada program možemo pauzirati i poslije nastaviti sa izvođenjem.

```
proc = env.Processor()
# code je instanca razreda Program
result = proc.execute(code) # sinkroni način pokretanja


prog_id = proc.execute(code, async = True) #asinkroni način pokretanja
proc.suspend(proc_id)  #pauziramo program
proc.resume(proc_id) # nastavljamo sa izvođenjem


proc.join(proc_id) # čekamo da program završi sa izvođenjem
```

Da bismo mogli bolje analizirati program koji smo napisali CorePy nam nudi metodu za ispis koda u ovisnosti o parametrima koje predamo. Ako `hex` i `binary` parametre postavimo na True osim asemblerskog ispisa dobiti ćemo heksadekadski i binarni broj koji predstavlja strojni kod instrukcije. CorePy interno generira strojni kod kao npr. kod prenošenja parametara i sa parametrima `pro` i `epi` (prolog i epilog) možemo vidjeti što je CorePy izgenerirao.

```
code.print_code(hex = True, binary = True, pro = True, epi = True)
```

2.3. Generator slučajnih brojeva

Prikaz implementacije LCG generatora slučajnih brojeva korištenjem biblioteke CorePy.

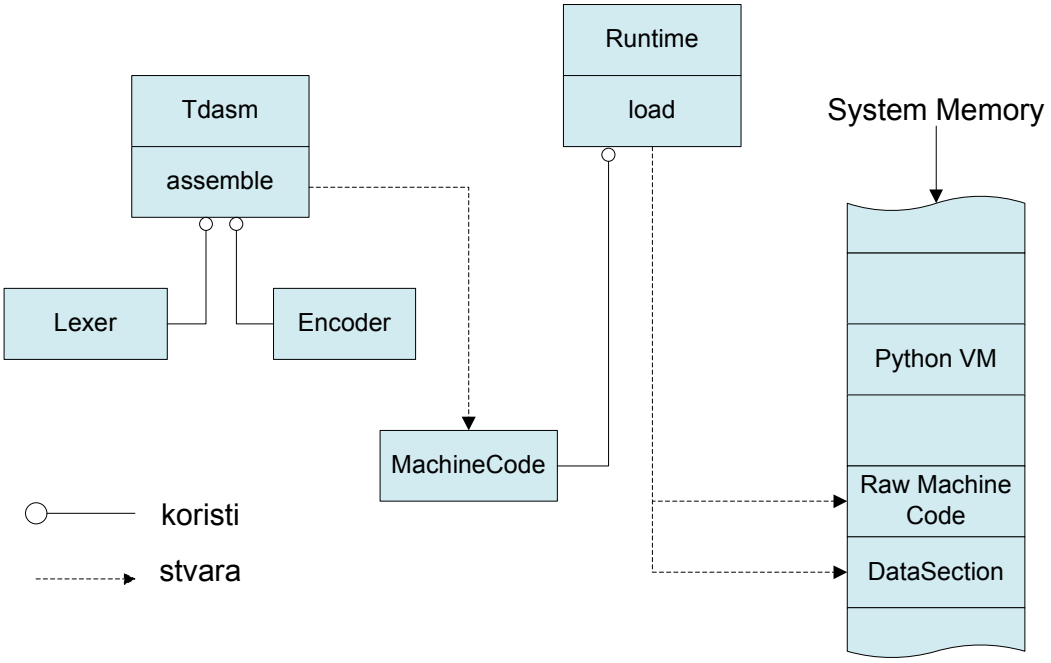
```
import random
import corepy.arch.x86_64.platform as x86_env
import corepy.arch.x86_64.isa as x86
from corepy.arch.x86_64.types.registers import *
from corepy.arch.x86_64.lib.memory import MemRef


from array import array


#parametri koje koje generator slucajnih brojeva
mult = array("I")
mult.append(214013)
mult.append(17405)
mult.append(214013)
mult.append(69069)
mult_adr, length = mult.buffer_info()


gadd = array("I")
gadd.append(2531011)
gadd.append(10395331)
gadd.append(13737667)
gadd.append(1)
gadd_adr, length = gadd.buffer_info()


mask = array("I")
mask.append(0xFFFFFFFF)
mask.append(0)
```

```
mask.append(0xFFFFFFFF)
mask.append(0)
mask_adr, length = mask.buffer_info()



seed = array("I")
seed.append(0xFFFFFFFF)
seed.append(0)
seed.append(0xFFFFFFFF)
seed.append(0)
seed_adr, length = seed.buffer_info()

params = x86_env.ExecParams()
params.p1 = mult_adr
params.p2 = gadd_adr
params.p3 = mask_adr
params.p4 = seed_adr

prgm = x86_env.Program()
code = prgm.get_stream()
code.add(x86.mov(rax, MemRef(rbp, 16)))
code.add(x86.movdqu(xmm1, MemRef(rax, data_size = 128)))
code.add(x86.mov(rax, MemRef(rbp, 24)))
code.add(x86.movdqu(xmm0, MemRef(rax, data_size = 128)))
code.add(x86.mov(rax, MemRef(rbp, 32)))
code.add(x86.movdqu(xmm2, MemRef(rax, data_size = 128)))
code.add(x86.mov(rax, MemRef(rbp, 40)))
code.add(x86.movdqu(xmm7, MemRef(rax, data_size = 128)))
code.add(x86.mov(rdi, MemRef(rbp, 48)))
code.add(x86.mov(rcx, MemRef(rbp, 56)))
code.add(x86.imul(rcx, rcx, 4))

lbl_loop = prgm.get_label("loop")
code.add(lbl_loop)
```

```
code.add(x86.sub(rcx, 16))


code.add(x86.pshufd(xmm4, xmm7, 0xB1))
code.add(x86.movdqa(xmm5, xmm7))
code.add(x86.pmuludq(xmm5, xmm1))
code.add(x86.pshufd(xmm1, xmm1, 0xB1))
code.add(x86.pmuludq(xmm4, xmm1))
code.add(x86.pand(xmm5, xmm2))
code.add(x86.pand(xmm4, xmm2))
code.add(x86.pshufd(xmm4, xmm4, 0xB1))
code.add(x86.por(xmm5, xmm4))
code.add(x86.paddd(xmm5, xmm0))
code.add(x86.movdqa(xmm7, xmm5))

code.add(x86.movdqu(MemRef(rdi, index=rcx, data_size = 128), xmm5))
code.add(x86.cmp(rcx, 0))

lbl_exit = prgm.get_label("exit")
code.add(x86.jle(lbl_exit))
code.add(x86.jmp(lbl_loop))
code.add(lbl_exit)

prgm += code
proc = x86_env.Processor()

def generate_random_numbers(arr):
    seed[0] = random.randint(0, 1000000000)
    seed[1] = random.randint(0, 1000000000)
    seed[2] = random.randint(0, 1000000000)
    seed[3] = random.randint(0, 1000000000)

    adr, length = arr.buffer_info()
    params.p5 = adr
    params.p6 = length
```

```
    proc.execute(prgm, params = params)
```

3. Korištenje bibliotekeTDAsm iz Pythona



Slika 2. Arhitektura asemblera TDAsm

Metoda `assemble` razreda `Tdasm` prihvaća asemblerski kod koji uz pomoć `Lexer` rastavlja na tokene. Kad je kod rastavljen na tokene asembler parsira jednu po jednu liniju koda i koristeći `Encoder` prevodi asemblerski kod u strojni kod. Prevedeni strojni kod sadržan je u objektu `MachineCode`. Da bismo mogli izvršiti strojni kod potrebno ga je učitati u radnu memoriju a za to je zadužena metoda `load` razreda `Runtime`. Jednom kad smo učitali strojni kod u memoriju on je spreman za izvođenje. Objekt `Runtime` prije nego što učita strojni kod u radnu memoriju alocira memorijski prostor za globalne varijable i napravi realokaciju nad strojnim kodom. Globalne varijable o kojima će više riječi biti kasnije služe za prenošenje parametara iz Pythona u asembler i dohvaćanje rezultata nakon što se izvrši kod. Postavljanje i dohvaćanje vrijednosti globalnih varijabli obavljamo pomoću objekta `DataSection`.

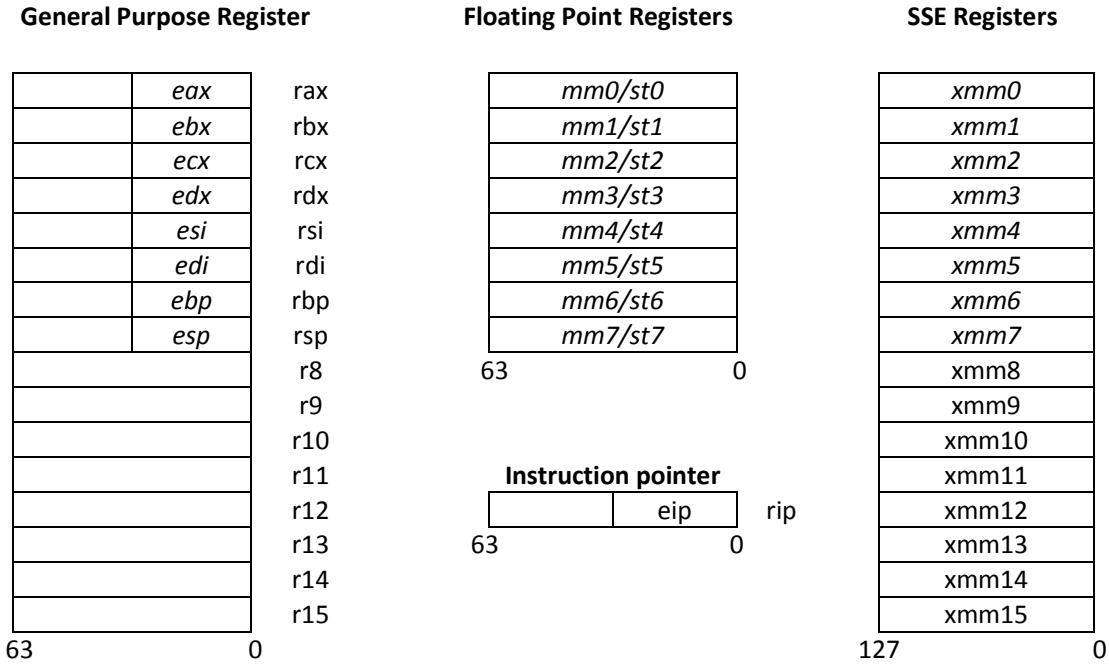
Jednom napisan asemblerski odsječakprevodimo u strojni kod i izvršavamo iz Pythona u sljedećih pet jednostavnih koraka:

1. Asemblerski kod prevodimo u strojni kod metodom `assemble` objekta `Tdasm`
2. Učitavamo strojni kod u memoriju metodom `load` objekta `Runtime`
3. Koristeći objekt `DataSection` postavljamo vrijednosti globalnih varijabli
4. Izvršavamo strojni kod pozivom metode `run` objekta `Runtime`
5. Koristeći objekt `DataSection` dohvaćamo vrijednosti globalnih varijabli

Iako cilj ovog rada nije učenje asemblera za arhitekturu x86 sve instrukcije koje će se pojavljivati u primjerima biti će objašnjene zbog lakšeg praćenja primjera. Slijedi jednostavni primjer koji zamjenjuje vrijednosti varijabli `x` i `y`.

```
from tdasm import Runtime, Tdasm
```

Uključujemo razrede `Runtime` i `Tdasm` koji su neophodni za prevođenje i izvršavanje asemblerskog koda. Asemblerski kod je običan Pythonski string. Kod naprednijh programa možemo taj string dinamički generirati i tako postići napredne optimizacije. Sad moramo napisati asemblerski kod koji kad zamjenjuje vrijednosti varijablama `x` i `y`. Da bismo napisali i najednostavniji program u asembleru moramo znati koje registre imamo na raspolaganju. Na slici su prikazani registri opće namjene za 32-bitnu i 64-bitnu arhitekturu.



Slika 3. Prikaz x86-64 registara

Osim registara opće namjene x86 arhitektura podržava i vektorske registre o kojima će više riječi biti na kraju poglavlja. Sad kad znamo koje registre imamo na raspolaganju upoznati ćemo se sa instrukcijom `mov` koja nam je potrebna za naš prvi primjer. Sintaksa instrukcije `mov` je:

```
mov  dest, src    ; dest = src
```

Instrukcija mov kopira izvorni operand u odredišni. Operandi mogu biti registri, numeričke konstante ili memorijski operandi. Slijedi nekoliko primjera korištenja instrukcije `mov`.

```
mov al, 33    ;  al = 33
mov ecx, eax  ; ecx = eax
mov eax, dword [n] ;eax = n, n je globalna varijabla
mov dword [n], ebx ; na lokaciju n upisujemo vrijednost registra ebx
mov dword [k], dword [n] ;greška ne možemo imati 2 memorijska operanda
```

Sad kad smo se upoznali sa instrukcijom `mov` i registrima, možemo pristupiti pisanju asemblerskog koda za naš primjer.

```
    ASM_SOURCE = """
        #DATA
        int32 x, y

        #CODE
        mov eax, dword [x] ; eax = x
        mov ebx, dword [y] ; ebx = y
        mov dword [x], ebx ; x = ebx
        mov dword [y], eax ; y = eax
        """
```

Kao što vidimo, asemblerski kod se sastoji od dva djela a to su `#DATA` i `#CODE`. U `#DATA` dijelu deklariramo globalne varijable u koje kasnije iz Pythona postavljamo i dohvaćamo vrijednosti. `#DATA` dijelu biti će detaljnije opisan u tekstu nakon ovog primjera. S direktivom `#CODE` kažemo asembleru da slijede asemblerske instrukcije. Izvršavanje programa kreće od prve instrukcije nakon prve direktive `#CODE`. Ako dohvaćamo neku memorijsku lokaciju obavezno moramo navesti širinu u ovom slučaju `dword` koji je 32 bita širok. Za pisanje asemblerskih instrukcija koristi se Intelova sintaksa, što znači da nam je uvijek prvi operand odredište. Komentar počinje sa ';' i proteže se do kraja retka. U komentarima vidimo ekvivalent te naredbe u programskom jeziku C. Sad kad smo napisali mali odsječak asemblerskog koda trebamo napraviti pet prethodno nabrojanih koraka da bismo ga preveli i izvršili.

```
from tdasm import Runtime, Tdasm


 ASM_SOURCE = """
        #DATA
        int32 x, y

        #CODE
        mov eax, dword [x] ; eax = x
        mov ebx, dword [y] ; ebx = y
        mov dword [x], ebx ; x = ebx
```

```
        mov dword [y], eax ; y = eax
        """


    if __name__ == "__main__":
        asm = Tdasm()
        machine_code = asm.assemble(ASM_SOURCE)
        r86 = Runtime()
        ds = r86.load("swap", machine_code)
        ds["x"] = -88
        ds["y"]=  44
        r86.run("swap")
        print (ds["x"], ds["y"])
```

Za prevođenje asemblerskog koda zovemo metodu `assemble` kojoj predajemo asemblerski kod. Ako je sve prošlo u redu metoda `assemble` vraća objekt razreda `MachineCode` koji sadrži naš prevedeni kod. Da bi kod bio spreman za izvođenje koristimo metodu `load` razreda `Runtime` koja taj kod učita u memoriju. Kad smo kod učitali u memoriju on je spreman za izvođenje. Prije nego šsto postavimo neke smislene vrijednosti u varijable nema smisla izvršavati program. Za postavljanje i dohvaćanje vrijednosti iz varijabli `x` i `y` koristimo objekt `DataSection` koji nam je vratila metoda `load`. Za pokretanje koristimo metodu `run` objekta Runtime koja kao parametar prima ime pod kojim smo ranije učitali taj kod u memoriju. Nakon što je završeno izvršavanje strojnog koda ispisujemo varijable `x` i `y` da se uvjerimo u ispravnost programa. Prije nego što krenemo sa složenijim primjerima, detaljnije ćemo se upoznati s pisanjem asemblerskog koda i manipuliranjem globalnih varijabli iz Pythona.

S direktivom `#DATA` kažemo asembleru da slijedi deklaracija globalnih varijabli. Osim gore korištenog cjelobrojnog tipa podataka `int32` asembler još podržava i cjelobrojne tipove `int8`, `uint8`, `int16`, `uint16`, `uint32`, `int64`, `uint64` i realne tipove `float` i `double`. Varijable prilikom deklaracije možemo i inicijalizirati. Numeričke konstante osim u dekadskom sustavu možemo pisati u binarnom i heksadecimalnom sustavu.

Tabela 1. Raspon vrijednosti numeričkih tipova podataka

Size	Name	Range
8	uint8	0 to 255
8	int8	-128 to 127
16	uint16	0 to 65 535
16	int16	-32 768 to 32 767
32	uint32	0 to 4 294 967 295
32	int32	-2 147 483 648 to 2 147 483 647
64	uint64	0 to 18 446 744 073 709 551 615
64	int64	–9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
32	float	-3.40292347E+38 to +3.40292347E+38
64	double	-1.7976931E+308 to +1.796931E+308

```
#DATA

uint8 x, y, z

int64 h = 3434456774334565  ;deklaracija i inicijalizacija

uint32 m = 0xFFAABBCC   ; heksadecimalna konstanta

uint8 b = 101110b           ; binarna konstanta
```

Osim varijabli možemo deklarirati i nizove.

```
#DATA

float r[12]  ; niz od 12 realnih brojeva jednostruke preciznosti

float g[6] = 2.3, 4.5, 2.9, 7.7, 1.1, 2.5 ; deklaracija i inicijalizacija
niza

uint32 k[8] = 44, 33, 22, 11  ; 0-3 članovi su inicijalizirani a ostaka niza
je neinicijaliziran
```

Manipulacija nizovima iz Pythona obavlja se slično kao i s običnim varijablama ali ima i dodatnih opcija.

```
# ds - objekt DataSection
ds[r] # dohvatiti ćemo cijeli niz koji će biti tuple
ds.get_member(r) #  isto je kao i ds[r]
ds.get_member(r, 5) # dohvatiti ćemo niz koji kreće od 6 člana pa do kraja
dg.get_member(r, 5, 3) #dohvatiti ćemo niz duljine 3 a kreće se od 6 člana
```

Sad kad znamo deklarirati varijable ostaje nam pisanje asemblerskog koda koji će manipulirati tim varijablama. Za pisanje instrukcija koriste se Intelova sintaksa znači da nam je prvi operand uvijek odredište. Asembler ne podržava pisanje makroa, odmotavanje petlji i slične mogućnosti koje su uobičajene kod boljih asemblera zbog toga što je asemblerski kod predstavljen običnim stringom kojeg dinamički generiramo iz Pythona i većinu tih stvari možemo emulirati Pythonom. Imena instrukcija i registara pišu se malim slovima a instrukcija ima maksimalno tri operanda. Memorijski operandi pišu se u uglatim zagradama i obavezno ispred uglate zagrade mora se naznačiti i širina operanda. Iako asembler u nekim slučajevima zavisno o instrukciji može sam zaključiti širinu memorijskog operanda, nekad to nije moguće i to je razlog čega se uvijek eksplicitno mora naznačiti širina memorijskog operanda.

Tabela 2. Širine memorijskih operanada u bitovima

Size	Name
8	byte
16	word
32	dword
64	qword
128	oword
128	dqword

Na primjeru vidimo kako ispravno pisati asemblerske instrukcije.

```
#DATA
uint32 varijabla
#CODE
add eax, ebx  ; ispravno
add Eax, Ebx  ; krivo - imena registara se pišu malim slovima
Add eax, ebx ; krivo - instrukcije se također pišu malim slovima

mov eax, dword [varijabla] ; ispravno
mov eax, [varijabla]   ; krivo - nije naznačena širina memorijskog operanda
mov al, byte [varijabla] ; ispravno
```

Arhitektura x86 podržava razne načine adresiranja memorije koji će tu biti prikazani. Za računanje efektivne adrese koristi se formula adresa = baza + skalar*indeks + pomak. U ovoj formuli baza i indeks predstavljaju registre opće namjene, skalar može biti 1, 2, 4 ili 8 a pomak je 8-bitni ili 32-bitni broj. Dodatno ograničenje postavlja se na indeksni registar koji ne može biti esp kod 32-bitne arhitekture odnosno rsp kod 64 bitne. Na primjeru je prikazano kako koristeći različite načine adresiranja učitavamo element niza iz memorije u registar.

```
#DATA
uint32 niz[20] =  20, 22, 25, 44, 33, 77, 55, 33, 11, 99, 88, 99, 11, 22

#CODE
mov eax, dword [niz]  ; eax = 20
mov eax, dword [niz + 12]  ; eax = 44
```

Kod 32-bitne arhitekture u gornja dva slučaja koristi se apsolutno adresiranje i niz predstavlja punu 32-bitnu adresu a kod 64-bitne arhitekture u gornja dva slučaja koristi se relativno adresiranje i niz predstavlja razliku između `RIP` i adrese nultog člana niza. `RIP` je registar koji sadrži adresu sljedeće instrukcije koju treba izvršiti. Gornja dva slučaja primjeri su direktnog adresiranja, da bi koristili indirektno adresiranje moramo prvo učitati adresu niza u registar a to možemo na dva načina pomoću instrukcija `lea` ili `mov`.

```
; lea - instrukcija koja učitava adresu operanda

; Pomoću lea instrukcije učitavamo adresu niza u registar

lea eax,  dword [niz]  ;  eax = &niz[0]   32-bitni kod

lea rax,  qword [niz]  ;   rax =  &niz[0]   64-bitni kod

lea rax, qword [niz + 12] ; rax = &(niz + 12) učitavanje adrese gdje počinje
4-ti član niza



; Da bi se olakšalo pisanje koda asembler dozvoljava posebnu sintaksu mov
instrukcije za učitavanje adrese varijable

mov eax, niz  ; eax = &niz[0]   32-bitni kod

mov rax, niz   ; rax =  &niz[0]   64-bitni kod

mov rax, niz + 12 ; nije podržano i ovo je greška
```

Slijede primjeri indirektnog adresiranja. U komentarima je prikazan koji se element niza učitao u registar.

```
#DATA

uint32 niz[20] = 20, 22, 25, 44, 33, 77, 55, 33, 11, 99, 88, 99, 11, 22

#CODE

mov rax, niz  ; rax = &niz[0]

mov ebx, dword [rax]  ; ebx = 20

mov ecx, dword [rax + 8] ; ecx = 25

mov rdx, 16

mov ebx, dword [rax + rdx] ; ebx = 33

mov ebx, dword [rax + 2*rdx] ; ebx = 11

mov ebx, dword [rax + 2*rdx + 8] ; ebx = 88
```

U drugom primjeru implementirati ćemo rutinu `kvadrat` koja u eax registru prima broj koji treba kvadrirati, a rezultat izvođenja također se sprema u eax registar. Za kvadriranje broja koristiti ćemo instrukciju `mul`, a za implementaciju rutine koriste se instrukcije `call` i `ret`. Kad se pozove instrukcija `call` procesor sprema adresu koja se nalazi u instrukcijskom registru na stog i izvođenje se nastavlja u rutini koju smo pozvali. Za povratak iz pozivajuće rutine zovemo instrukciju `ret` koja očekuje da se na vrhu stoga nalazi adresa koju je prethodno postavila instrukcija `call`.

Sintaksa instrukcija `mul`, `call` i `ret` je:

```
; r/m32 operand može biti registar ili memorijska lokacija
mul r/m32    ; eax = eax * r/m32
call labela  ; poziv rutine koja se zove labela
ret   ; služi za povratak iz rutine
```

Slijedi primjer koji kvadrira varijablu `x`. Glavni program učitava varijablu `x` u registar eax i pozivamo strojni potprogram koji rezultat sprema natrag u varijablu `x`.

```
ASM = """
     #DATA
     uint32 x;
     #CODE
     mov eax, dword [x] ;  eax = x
     call kvadrat       ; kvadriramo eax registar
     mov dword [x], eax  ; x = eax
     #END

     kvadrat:
     mul eax      ; eax = eax * eax
     ret
"""
```

Kao što je bilo rečeno na početku poglavlja, asembler prije nego krene sa izvođenjem instrukcija koje smo napisali umetne kod za spremanje vrijednosti registara opće namjene. Nakon izvođenja zadnje instrukcije koju smo napisali vrijednosti registara se vraćaju na početne vrijednosti. Ako u bilo dijelu strojnog programa želimo završiti sa izvođenjem moramo navesti direktivu `#END` koja vraća vrijednosti registara na početne vrijednosti i umeće instrukciju `ret` koja vrati kontrolu Pythonu. U gornjem primjeru da nismo stavili `#END` direktivu nakon zadnje `mov` instrukcije asembler bi dalje nastavio sa izvođenjem instrukcija iz potprograma `kvadrat`. Preporuča se uvijek na kraj glavnog programa staviti direktivu `#END` da bi se izbjegle ovakve pogreške.

Da bi se olakšalo razvijanje složenijih programa, asembler podržava pisanje rutina u xml datoteke. Jedna xml datoteka sadrži jednu rutinu. Slijedi isti primjer kao i prethodni ali ovaj put će strojni potprogram biti u datoteci *kvadrat.xml*.

```
<kvadrat>
     <description>
            Rutina za kvadriranje
            Ulaz: eax
            Rezultat: eax
     </description>
     <source>
            mul eax
            ret
     </source>
</kvadrat>
```

kvadrat.xml

Polje description služi za opis rutine i može se izostaviti. Polje source je neophodno jer sadrži implementaciju rutine. Primjer izgleda isto kao i prije jedino što sad nema rutinu za kvadriranje.

```
ASM = """
     #DATA
     uint32 x;
     #CODE
     mov eax, dword [x] ;  eax = x
     call kvadrat                  ; kvadriramo eax registar
     mov dword [x], eax  ; x = eax
     #END
"""
```

Da bi asembler mogao razriješiti poziv rutine kvadrat, asembler mora znati gdje se nalazi xml datoteka. Asembler ima metodu `set_path` koja kao parametar prima putanju gdje se nalaze xml datoteke sa implementacijama strojnih rutina.

```
if __name__ == "__main__":

        asm = Tdasm()

        asm.set_path("putanja do direktorija gdje se nalaze implementirane
rutine")

        machine_code = asm.assemble(ASM)

        r86 = Runtime()

        ds = r86.load("kvadrat", machine_code)

        ds["x"] = 55

        r86.run("kvadrat")

        print (ds["x"])
```

Jedna od glavnih motivacija za razvoj ovog asemblera je iskorištavanje naprednih mogućnosti procesora poput vektorskog instrukcijskog seta SSE. SSE instrukcije ne koriste registre opće namjene nego imaju vektorske registre koji su široki 128-bita. Na 32-bitnoj arhitekturi imamo 8 vektorska registra a na 64-bitnoj broj je povećan na 16. Registri se označavaju sa xmm0 - xmm7 odnosno na 64-bita imamo xmm0 - xmm15. Jedan vektorski registar može sadržavati 4 broja jednostruke preciznosti ili 2 broja dvostruke preciznosti ili 4 cjelobrojna 32-bitna broja ili 8 cjelobrojnih 16-bitnih brojeva ili 16 cjelobrojnih 8-bitnih brojeva. Naš drugi primjer je izračunati drugi korijen nad nizom brojeva jednostruke preciznosti. Prije nego što prikažemo program koji to radi upoznati ćemo se sa SSE instrukcijama. Kao što smo u prethodnim primjerima koristili instrukciju `mov` za kopiranje sadržaja iz memorije u registar tako postoji velik broj različitih instrukcija `mov` koje prebacuju sadržaj iz memorije u vektorski registar. To su jedne od najvažnijih instrukcija za postizanje vrhunskih performansi tako da ćemo se tu detaljnije upoznati sa njima. Svi Intelovi i AMD-ovi 64-bitni procesori podržavaju SSE2. U zavisnosti o tipu podatka koji prebacujemo iz memorije u registar imamo različite `mov` instrukcije. Slijedi odsječak koda koji prikazuje kako određeni tip podatka učitati u vektorski registar.

```
#DATA

uint32 v1[4] = 33, 44, 55, 66

float v2[4] = 44.5, 11.11, 33.45, 22.95

double v3[2] = 11.34567755, 11111.4456666

#CODE

;movups - memorija nije poravnata i movaps - memorija je poravnata na 128-
bita

;u komentarima je prikazano kako je broj smješten u vektorskom registru

movups xmm0, oword [v2] ; xmm0 = 22.95, 33.45, 11.11, 44.5

movaps xmm0, oword [v2] ; xmm0 = 22.95, 33.45, 11.11, 44.5

movdqu xmm0, oword [v1]; xmm0 = 66, 55, 44, 33

movdqa xmm0, oword [v1]; xmm0 = 66, 55, 44, 33

movupd xmm0, oword [v3]; xmm0 = 11111.4456666, 11.34567755

movapd xmm0, oword [v3]; xmm0 = 11111.4456666, 11.34567755
```

Kao što vidimo, sve `mov` instrukcije dolaze u dvije varijante: jedna za čitanje iz memorije koja nije poravnata na 128-bita i druga za poravnato čitanje. Da bismo postigli vrhunske performanse kad god je moguće treba koristiti instrukcije koje zahtijevaju da je memorija poravnata. Asembler se sam brine da globalne varijable budu ispravno poravnate i možete koristiti instrukcije koje zahtijevaju poravnavanje. Kad nismo sigurni da li je memorija ispravno poravnata obavezno treba koristiti instrukcije koje ne zahtijevaju da memorija bude poravnata. Također je moguće učitati vrijednost u određeni dio registra za što opet postoji velik broj `mov` instrukcija.

```
#DATA

uint32 v1[4] = 33, 44, 55, 66

float v2[4] = 44.5, 11.11, 33.45, 22.95

double v3[2] = 11.34567755, 11111.4456666

#CODE

; xxxx - vrijedost koja je bila prije mov instrukcije

movhps xmm0, qword [v2]  ; xmm0 = 11.11, 44.5, xxxx, xxxx

movlps xmm0, qword [v2]  ;  xmm0 = xxxx, xxxx, 11.11, 44.5
```

```
movss  xmm0, dword [v2]  ;   xmm0 = xxxx, xxxx, xxxx, 44.5
movhpd xmm0, qword [v3];  xmm0 = 11.34567755, xxxxxxxx
movlpd  xmm0, qword [v3];  xmm0 =  xxxxxxxxxx, 11.34567755
movsd   xmm0, qword [v3];  xmm0 =   xxxxxxxxxx, 11.34567755
movq   xmm0, qword [v1] ; xmm0 = xxxxxxxx, xxxxxxxx, 44, 33
movd  xmm0, dword [v1];  xmm0 = xxxxxx, xxxxxx, xxxxxx, 33
```

Sve nabrojane `mov` instrukcije možete koristiti i za spremanje iz registra u memoriju. Ovo nije kompletna lista `mov` instrukcija ali je dovoljna za naša primjere. SSE podržava osnovne aritmetičke operacije a to su: zbrajanje, oduzimanje, množenje, dijeljenje, korjenovanje i određivanje recipročne vrijednosti. Sve aritmetičke operacije imaju dvije izvedbe, jednu koja radi nad sva četiri elementa vektorskog registra te drugu koja radi samo s jednim elementom vektorskog registra. Prikazati ćemo primjer sa množenjem a jedina razlika u odnosu na ostale aritmetičke operacije je naziv instrukcije.

```
#DATA
float v1[4] = 3.4, 5.6, 3.8, 4.2
float v2[5] = 6.6, 7.7, 8.8, 9.9
#CODE
movaps xmm0, oword [v1]
movpas xmm1, oword [v2]
mulps   xmm1, xmm0 ;  xmm1 = xmm1 * xmm0  ,  xmm1=41.58, 33.44, 43.12, 22.44
mulss   xmm1, xmm0 ;xmm1[0]=xmm1[0]*xmm0[0], xmm1=41.58, 33.44, 43.12, 76.296
```

Slijedi primjer u kojem nad svim elementima niza računamo drugi korijen koristeći SSE instrukcije. Parametri koje python prosljeđuje su broj elemenata niza i adresa prvog elementa. U ovom primjeru očekuje se da broj elemenata u nizu bude djeljiv s 4.

```
ASM = """
     #DATA
     uint32 n ; broj elemenata niza
     uint64 adresa  ; adresa prvog elementa
```

```
     #CODE
     mov rax, qword [adresa]
     mov ecx, dword [n]

     dalje:
     movups xmm0, oword [rax]
     sqrtps  xmm0, xmm0  ; xmm0 =  sqrt(xmm0)  , korjenovanje
     ; mulps xmm0, xmm0  ; xmm0 = xmm0 * xmm0 ,kvadriranje
     movups oword [rax], xmm0

     add rax, 16 ; uvećamo adresu za 16
     sub ecx, 4  ;  smanjimo broj elementata za 4
     jnz dalje   ;  ako ecx nije jednak nuli nastavljamo sa petljom
     #END
"""
```

Razvijanje programa za SSE3 ili SSE4 je dosta problematično zbog toga što Intelovi i AMD-ovi procesori ne podržavaju iste instrukcijske podskupove. Zbog toga se većina aplikacija danas optimizira za SSE2. Asembler podržava Intelove instrukcijeske setove SSE3, SSSE3, SSE4.1 i SSE4.2. Da bi se olakšalo programerima asembler dopušta da jednu rutinu implmentiramo na više načina pomoću različitih instrukcijskih skupova npr. SSE2, SSSE3, SSE4.1. U trenutku prevođenja programa asembler sam izabire odgovarajuću implementaciju u ovisnosti o tome koji instrukcijski set podržava procesor na kojem se odvija prevođenje. Prioritet dobivaju podržane implementacije koje koriste noviji instrukcijski set što znači ako imamo jednu rutinu koja je implementirana pomoću SSE2 i SSE4.1 asembler će odabrati implementaciju koja koristi SSE4.1. Na primjeru ćemo prikazati kako korištenjem različitih instrukcijskih setova implementirati rutinu za računanje skalarnog produkta vektora. Jedina razlika ovakvih rutina u odnosu na rutine koje imaju samo jednu implementaciju je ta što postoji više source tagova. Za primjer je uzeto računanje skalarnog produkta zbog toga što su noviji vektorski instrukcijski setovi donijeli nove instrukcije za efikasnije računanje skalarnog produkta.

```
<dot_product>
     <description>
            Implementation of dot product.
            Input: xmm0 = vector1 , xmm1 = vector2
            Result: low-dword of xmm0 is result
            xmm0 = a4, a3, a2, a1
            xmm1 = b4, b3, b2, b1
     </description>
     <source req="sse2" inline="true">
#CODE
    mulps xmm0, xmm1     ;a4*b4, a3*b3, a2*b2, a1*b1
    movhlps xmm1, xmm0   ;X, X, a4*b4, a3*b3
    addps xmm0, xmm1     ;X, X, a2*b2 + a4*b4, a1*b1 + a3*b3
    pshufd xmm1, xmm0, 1 ;X, X, X, a2*b2 + a4*b4
    addss xmm0, xmm1     ;a1*b1 + a3*b3 + a2*b2 + a4*b4
     </source>
<source req="sse3" inline="true">
#CODE
    mulps xmm0, xmm1   ;a4*b4, a3*b3, a2*b2, a1*b1
    haddps xmm0, xmm0  ; a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
    movaps xmm1, xmm0  ;a4*b4+a3*b3, a2*b2+a1*b1, a4*b4+a3*b3, a2*b2+a1*b1
    psrlq xmm0, 32     ; 0,a4*b4+a3*b3, 0, a4*b4+a3*b3
    addss xmm0, xmm1   ; a1*b1 + a3*b3 + a2*b2 + a4*b4
     </source>
<source req="sse41" inline="true">
#CODE
     dpps xmm0, xmm1, 0xf1 ; 0, 0, 0, a1*b1 + a3*b3 + a2*b2 + a4*b4
     </source>
</dot_product>
```

dot_product.xml

Vidimo xml datoteku gdje je implementirana rutina za računanje skalarnog produkta pomoću instrukcijskih podskupa SSE2, SSE3 i SSE4.1. Svaki pojedino *source* polje sadrži *req* opciju koja kaže koji minimalni set instrukcija procesor

mora podržavati. Zbog dodatne efikasnosti uključena je i opcija *inline* da bi se izbjegao poziv funkcije. Korištenje ovakvih rutina ne razlikuje se od korištenja običnih rutina.

```
ASM = """
#DATA
float vec1[4] = 2.3, 3.3, 4.4, 5.5
float vec2[4] = 1.2, 3.3, 2.45, 5.66
float result[4]

#CODE
movaps xmm0, oword [vec1]
movaps xmm1, oword [vec2]
call dot_product
movaps oword [result], xmm0
"""
```

U većini slučajeva želimo napisati više rutina u asembleru koje koriste iste globalne varijable. Npr. želimo implementirati generator slučajnih brojeva koji ima dvije rutine, jednu za inicijalizaciju stanja generatora a drugu koja koristi to stanje za generiranje slučajnog broja. Da bismo to postigli moramo paziti da te dvije rutine imaju istu podatkovnu `#DATA` sekciju. Prilikom učitavanja prve rutine u memoriju metoda `load` objekta `Runtime` alocirati će memorijski prostor potreban za sve globalne varijable a prilikom učitavanja druge rutine trebamo metodi `load` proslijediti već postojeći objekt tipa `DataSection` koji nam je vraćen nakon prvog poziva.

```
ASM1 = """
#DATA
uint32 seed
uint32 x, y, z ........
#CODE
...
...
"""
ASM2="""
#DATA
uint32 seed
uint32 h, m, k, .......
#CODE
...
...
"""
```

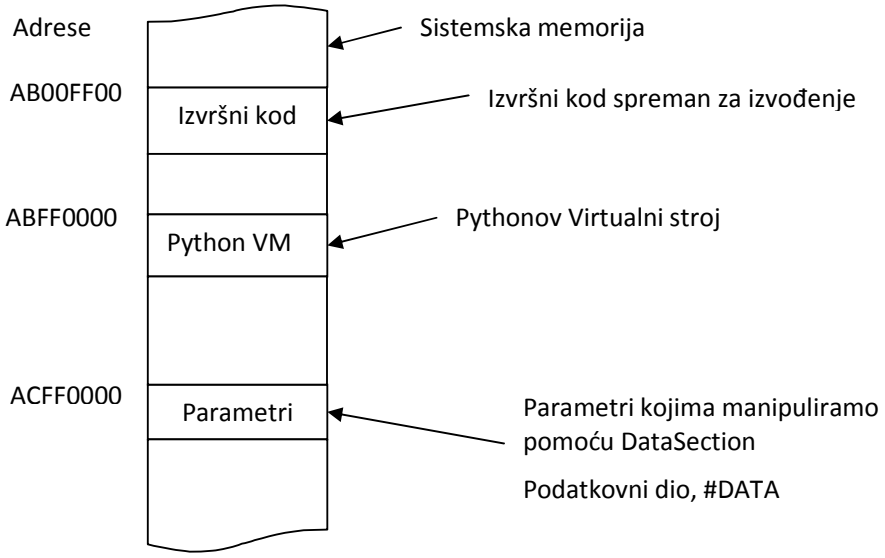
Želimo prevesti ove dvije rutine tako da dijele zajedničke varijable u našem primjeru želimo da i jedna i druga rutina imaju pristup istoj varijabli `seed`. Zbog toga što metoda `load` prilikom prvog poziva mora alocirati prostor za sve globalne varijable moramo spojiti podatkovnidio rutina. To radimo nakon što smo preveli asemblerski u strojni kod pomoću metode `combine_data_sections`.

```
if __name__ == "__main__":
        asm = Tdasm()
       ;machine_code uz strojni kod sadrzi i deklaracije globalnih varijabli
        machine_code = asm.assemble(ASM1)
        machine_code2 = asm.assemble(ASM2)

      ;sinkorniziramo da oba dva objekta imaju isti podatkovni dio
       machine_code.combine_data_sections(machine_code2)

        r86 = Runtime()
```

```
        ;nije bitno koju rutinu prvo ucitamo
        ds = r86.load("init", machine_code) ;init je proizvoljno ime
        ;kao treci parametar moramo predati postojeci ds
        ds = r86.load("random", machine_code2, ds)


        r86.run("init")
        r86.run("random")
```

4. Izvedbeni detalji asemblera TDAsm

Prenošenje parametara iz Pythona u asembler i dohvaćanje rezultata riješeno je tako da se alocira prostor u memoriji za parametre. Objekt `Runtime` to radi prije učitavanja strojnog koda jer da bi se mogla napraviti realokacija moraju biti poznate adrese parametara u memoriji. Prije nego što pokrenemo učitan kod postavljamo vrijednosti parametara metodama objekta `DataSection`. Na slici vidimo kako izgleda memorija kad je strojni kod spreman za izvršavanje.

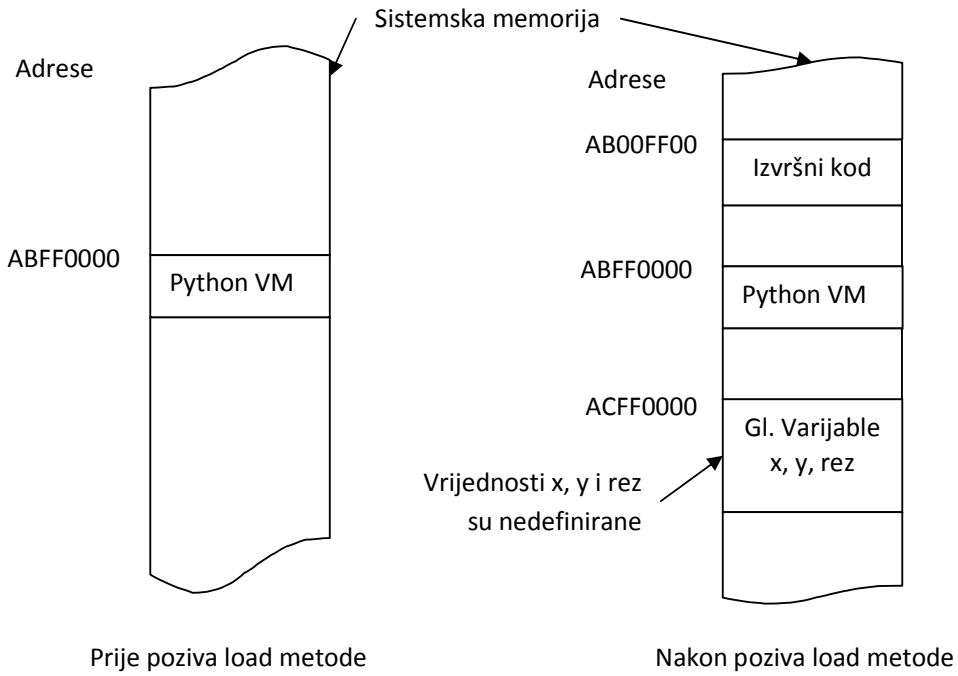


Slika 4. Stanje memorije nakon učitavanja strojnog koda

Da bi se još više pojasnilo kako python prenosi parametre i dohvaća rezultate slijedi primjer gdje će se objasniti što se događa sa memorijom u svakom koraku.Direktivom `#DATA` kažemo asembleru da slijedi deklaracija globalnih varijabli(parametara). U našem primjeru imamo tri globalne varijable `x`, `y` i `rez`. Kad pokrenemo izvršavanje ovog primjera zbrojiti će se vrijednosti varijabli `x` i `y` i to će se spremiti u varijablu `rez`.

```
ASM_KOD  = """
#DATA ;
uint32 rez, x, y
#CODE ;asemblerski kod koji koristi x, y, rez varijable
mov eax, dword  [x]
add eax, dword [y]
mov dword [rez], eax
"""


asm = Tdasm()
machine_code = asm.assemble(ASM_KOD)
r86 = Runtime()
ds = r86.load("zbroji", machine_code)
```

Metoda `load` razreda`Runtime` učitava strojni kod u izvršni dio memorije i vraća `DataSection` objekt preko kojeg postavljamo i dohvaćamo vrijednosti globalnih varijabli. Na slici vidimo što se zbiva sa memorijom prije i poslije poziva metode `load`.

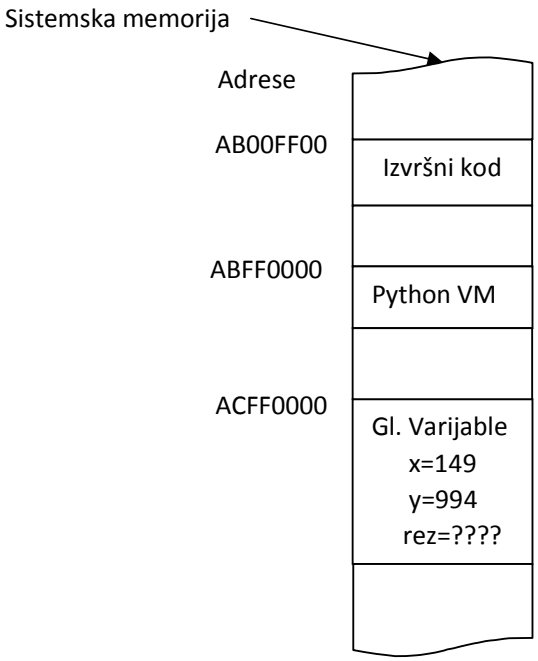


Slika 5. Sadržaj memorije prije i poslije učitavanja strojnog koda

Nakon poziva metode `load` strojni kod je spreman za izvršavanje, ali da bi rezultat izvođenja imao smisla treba prethodno zadati vrijednosti globalnih varijabli.

```
ds["x"] = 149
ds["y"] = 994
```

Stanje memorije nakon postavljanja vrijednosti u globalne varijable.

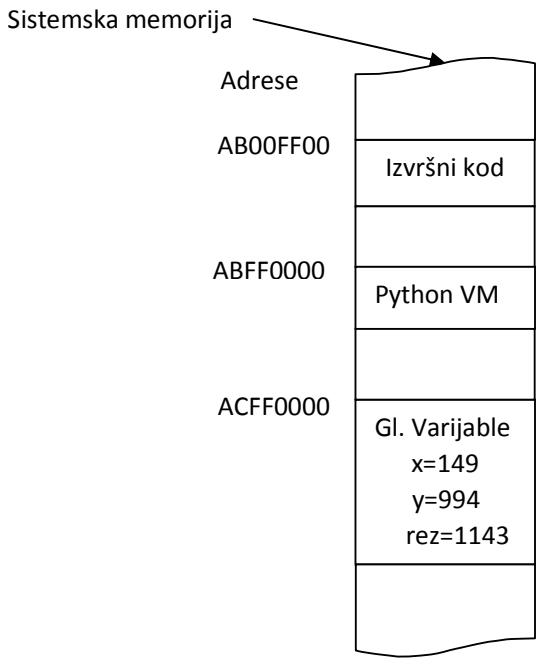


Slika 6. Sadržaj memorije nakon učitavanja parametara x i y

Da bi prenošenje parametara bilo što jednostavnije ili bolje reći u duhu Pythona objekt tipa DataSection pamti kojeg je tipa koji parametar i sam radi konverzije između tipa Pythona i onog u globalnoj memoriji. Da su `x` i `y` varijable u našem primjeru tipa `uint16`, `uint64`, `int16` ili `int64`, `DataSection` bi ispravno napravio konverzije. Zbog performansi objekt za pristup parametrima ne provjerava da li broj izvan dozvoljenih vrijednosti što znači da ako imamo tip podatka `uint8` a predamo vrijednost veću od 255 neće se prijaviti greška nego ćemo imati krivi rezultat nakon izvođenja programa. Da bismo to spriječili moramo sami provjeriti da li je parametar u dozvoljenom intervalu. Sad kad smo postavili parametre na željene vrijednosti pozivamo metodu `run` razreda `Runtime` da pokrenemo izvršavanje koda koji smo učitali u izvršnu memoriju.

```
r86.run("zbroji")
```

Prikaz stanja memorije nakon što je završilo izvršavanje koda.



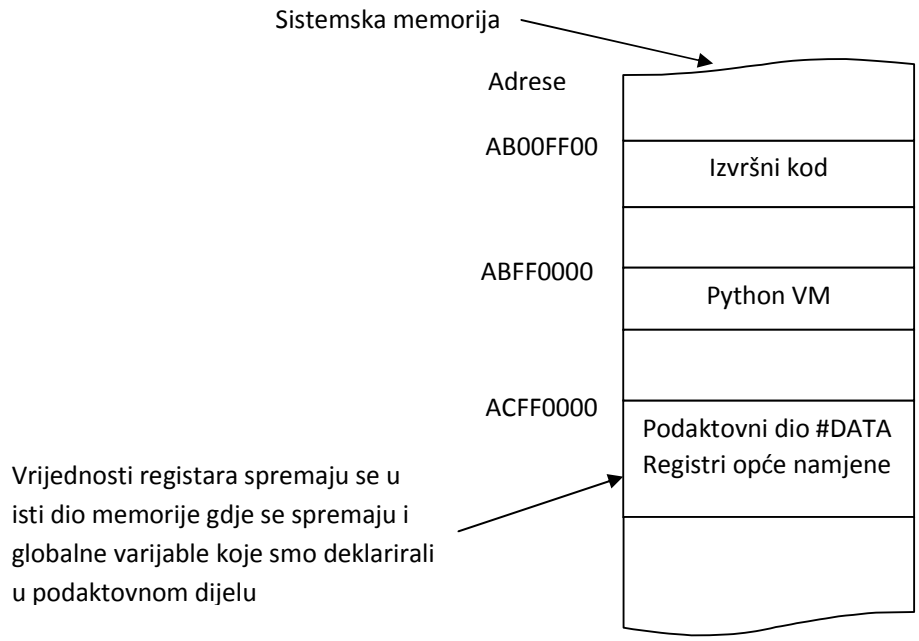
Slika 7. Sadržaj memorije nakon izvršavanja strojnog koda

Nakon što je završeno izvršavanje programa rezultat izvođenja dohvaćamo na isti način na koji smo i prenosili parametre u memoriju.

```
rezultat = ds["rez"]
```

Kad smetodom `run` pokrenemo izvođenje programa dužnost programera je da se pobrine da nakon što se strojni kod izvrši stanje procesora bude isto kao i prije početka izvršavanja. Ako promijenimo vrijednost registra i ne vratimo je na početnu vrijednost nakon izvršavanja strojnog koda Pythonov virtualni stroj neće moći nastaviti normalno funkcionirati. Da bi se olakšalo programerima asembler sam alocira prostor u memoriji za spremanje osam registra opće namjene i prije nego što se krene sa izvršavanjem koda koji je napisao programer umetnute su instrukcije koje prebacuju vrijednosti iz registara u taj prostor. Isto tako prije nego što se završi sa izvođenjem programa umetnute su instrukcije koje vraćaju registre na njihove početne vrijednosti. Zbog toga što su početne vrijednosti registara spremljene u memoriju a ne na stog koristeći direktivu `#END` koja ubacuje instrukcije za vraćenje početnih vrijednosti registara i instrukciju `ret` koja vraća kontrolu Python-ovom virtualnom stroju moguće je prekinuti izvođenje strojnog

koda u bilo kojem trenutku. Direktiva `#END` je ekvivalent `exit` funkcije u programskom jeziku C. Za spremanje drugih registara programer se mora sam pobrinuti. U praksi se pokazalo da nije potrebno spremiti vektorske XMM registre.



Slika 8. Stanje memorije poslije izvršenja umetnutih instrukcija koje spremaju vrijednosti općih registara

Asembler podržava skoro sve instrukcije koje podržavaju današnji x86 procesori, većina instrukcija koje nisu implementirane sistemske su instrukcije. Glavni prioritet asemblera je optimiziranje matematičkih operacija a ne pisanje sistemskih programa. Sve potrebne informacije za kodiranje jedne instrukcije nalaze se u xml datoteci koja se zove isto kao i instrukcija npr. *mov.xml*, *add.xml* itd. Ime datoteke nam predstavlja ime instrukcije i jednostavnim preimenovanjem datoteke promijenili smo i ime instrukcije. Na ovaj način možemo ako želimo dati proizvoljna imena instrukcijama ili možemo uz postojeće ime instrukcije dodati i novo ime tako da napravimo kopiju xml datoteke i promijenimo joj ime. Na slici vidimo primjer `div` instrukcije u xml datoteki.

```
<div>

     <description>

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers
(divi-

dend) by the source operand (divisor) and stores the result in the AX
(AH:AL),

DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-

purpose register or a memory location.

     </description>

     <options mnemonic="div r/m"></options>

     <in op1="r/m8" opcode="F6" mod="/6" prefix="REX"></in>

     <in op1="r/m16" opcode="F7" mod="/6" prefix="66"></in>

     <in op1="r/m32" opcode="F7" mod="/6"></in>

     <in op1="r/m64" opcode="F7" mod="/6" i32="false" prefix="REX.W"></in>

</div>
```

Polje *description* opisuje što radi instrukcija i nije obavezan ali je koristan zbog toga što asembler ima nekoliko pomoćnih funkcija koje ispisuju informacije o instrukciji. Isto vrijedi i za polje *options*. Polje *in* je neophodno jer tu pišu sve informacije potrebne za kodiranje instrukcije. Za one koje zanima više preporuča se pogledati Intelov i AMD-ov priručnik za procesore.

4.1. Promjena organizacije memorije zbog 64-bitne arhitekture

Asembler je prvotno razvijen za 32-bitnu x86 arhitekturu procesora. Kod prelaska na 64 bita procesori su dobili jedan nov način adresiranja memorije, relativno adresiranje, gdje se pomak zbraja sa trenutnom vrijednosti instrukcijskog pokayivača. Instrukcijski pokayivač sadrži adresu sljedeće instrukcije koju treba izvesti. Pomoću tog adresiranja strojni kod postaje lakše prenosiv ali i on ima svojih ograničenja. Na slici je prikazano od kojih se sve polja sastoji instrukcija.

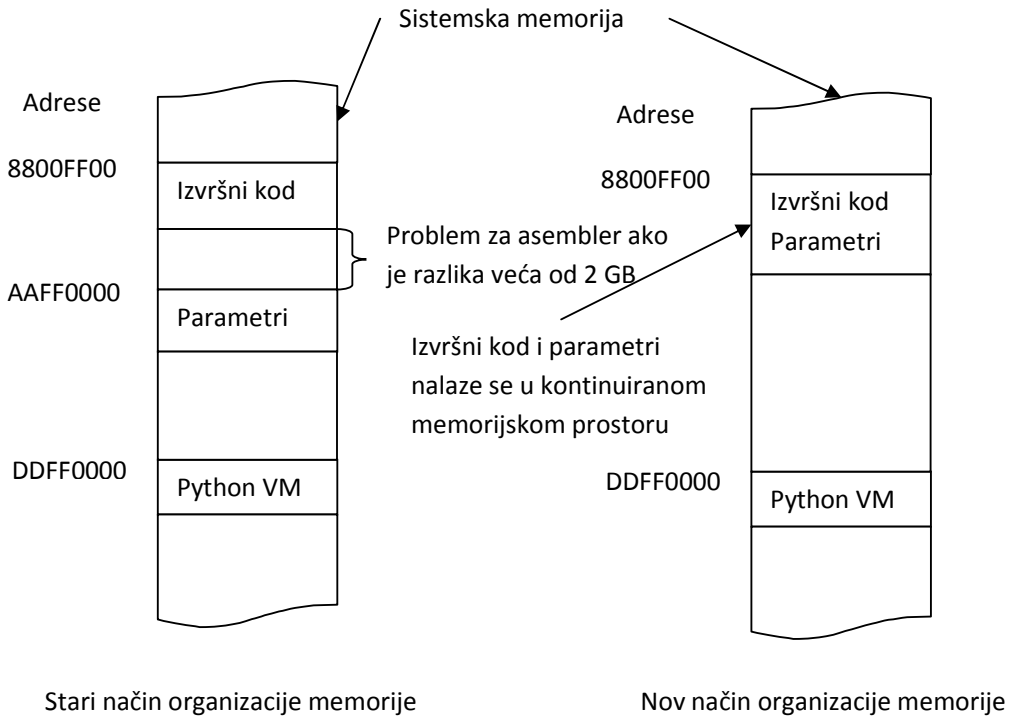
0-4	1,2 or 3	1	1	1,2 or 4	1,2 or 4
Instruction Prefix	Opcode	ModR/M	SIB	Displacement	Immediate

32 – bit format instrukcije

0-4	1	1,2 or 3	1	1	1,2 or 4	1,2 or 4
Instruction Prefix	REX	Opcode	ModR/M	SIB	Displacement	Immediate

64 – bit format instrukcije

Za ovu diskusiju bitan je samo jedan parametar a to je pomak (displacement). Kod 32-bitne arhitekture polje je široko 32 bita i prelaskom na 64-bitnu arhitekturu polje se nije povećalo nego je i dalje ostalo na 32 bita. Pošto 32-bitna arhitektura podržava maksimalno 4GB radne memorije 32-bitni ofset bio je dovoljan za adresiranje cijele radne memorije i zbog toga se koristio apsolutni način adresiranja gdje je ofset sadržavao adresu nekog bajta. Kod 64-bitne arhitekture možemo imati puno više od 4GB radne memorije i 32-bitni ofset više nije dovoljan za adresiranje cijele radne memorije. Zbog toga procesori u 64-bitnom modu ofset koriste za relativno adresiranje. U relativnom adresiranju pomoću 32-bitnog pomaka možemo adresirati+/-2GB memorije. Ovo ograničenje predstavlja problem za asembler zbog toga što on nezna gdje će u memoriji biti alociran prostor za parametre a gdje za strojni kod i može se dogoditi da razlika između ta dva memorijska prostora bude veće od 2GB. Da bi se izbjegli mogući problemi promijenjena je organizacija memorije i sad se strojni kod i parametri nalaze u kontinuiranom memorijskom prostoru kako je prikazano na slici.



Slika 9. Stanje memorije nakon učitavanja strojnog koda na x86-64 arhitekturi

Maksimalna veličina prostora za parametre je 2GB što je i više nego dovoljno.

5. Eksperimentalne evaluacije na stvarnim primjerima

U ovom poglavlju biti ce prikazana implementacija nekoliko konkretnih primjera. Svi primjeri bit će pisani za 64-bitnu arhitekturu. U prvom primjeru implementiran je jednostavan generator slučajnih brojeva korištenjem instrukcijskog podskupa SSE. Svrha prvog primjera je prikazati kako efikasno prenositi veće količine podataka između Pythona i asemblera korištenjem modula `array`. Za implementaciju generatora slučajnih brojeva potrebne su nam dvije rutine jedna za inicijalizaciju stanja generatora i druga za generiranje slučajnog broja. Prvo definiramo sve potrebne globalne varijable.

```
DATA_SECTION = """

    #DATA

    uint32 cur_seed[4]

    uint32 mult[4] = 214013, 17405, 214013, 69069

    uint32 gadd[4] = 2531011, 10395331, 13737667, 1

    uint32 mask[4] = 0xFFFFFFFF, 0, 0xFFFFFFFF, 0


    uint32 sign_mask[4] = 0x7FFFFFFF, 0x7FFFFFFF, 0x7FFFFFFF, 0x7FFFFFFF

    float flt[4] = 0.00000000465661287524, 0.00000000465661287524,
0.00000000465661287524, 0.00000000465661287524

    uint32 length ; duzina niza

    uint64 array ;adresa prvog elementa u nizu


    """
```

Većinu varijabli interno koristi generator slučajnih brojeva a nas najviše zanimaju dvije varijable a to su `length` i `array`. Ideja je da u Pythonu napišemo metodu koja vraća niz od n slučajnih brojeva. Brojevi koji se generiraju spremaju se u niz koji je predstavljen modulom `array`. Prvo ćemo napisati jednostavniju metodu koja koristeći `rdtsc` instrukciju inicijalizira stanje generatora slučajnih brojeva.

```
#  inicijaliziramo stanje generatora
#  rdtsc - u eax registar
INIT_RAND = DATA_SECTION + """

    #CODE
    rdtsc
    mov dword [cur_seed], eax
    mov dword [cur_seed + 8], eax
    add eax, 1
    mov dword [cur_seed + 4], eax
    mov dword [cur_seed + 12], eax
    #END
    """
```

Sad nam ostaje implementirati metodu koja puni niz sa slučajnim brojevima.

```
RAND_FUNCTION = DATA_SECTION  +  '''

    #CODE
    movdqa xmm0, oword [gadd]
    movdqa xmm1, oword [mult]
    movdqa xmm2, oword [mask]

    mov rdi, qword [array]
    mov ecx, dword [length]
    imul ecx, ecx, 4

    loop:
    sub ecx, 16

    pshufd xmm4, oword [cur_seed], 10110001b
    movdqa xmm5, oword [cur_seed]
    pmuludq xmm5, xmm1
    pshufd xmm1, xmm1, 10110001b
```

```
    pmuludq xmm4, xmm1
    pand xmm5, xmm2
    pand xmm4, xmm2
    pshufd xmm4, xmm4, 10110001b
    por xmm5, xmm4
    paddd xmm5, xmm0
    movdqa oword [cur_seed], xmm5
    movdqu oword [edi + ecx], xmm5

    cmp ecx, 0
    jle exit
    jmp loop
    exit:

    #END
    '''
```

Sad kad smo napisali rutine u asembleru ostaje nam implementirati metodu u Pythonu koja vraća niz slučajnih brojeva. Prevođenje asemblerskog koda u strojni kod i učitavanje strojnog koda u memoriju dovoljno je obaviti samo jednom.

```
from tdasm import Tdasm, Runtime
import array
asm = Tdasm()
init_code = asm.assemble(INIT_RAND)
rand_code = asm.assemble(RAND_FUNCTION)
run = Runtime()
ds = run.load("init", init_code)
ds = run.load("rand", rand_code, ds)
run.run("init") #inicijaliziramo stanje generatora


def random_ints(n):
    arr = array.array('I', B'\x00' * (n*4)) # alociranje i inicijaliziranje
niza
```

```
    address, length = arr.buffer_info()
    ds["array"] = address
    ds["length"] = length
    run.run("rand")
    return arr
```

Velik broj aplikacija zahtijeva slučajne brojeve u intervalu između 0.0 - 1.0. Funkcija za generiranje tih brojeva ista je kao i za generiranje cjelobrojnih vrijednosti jedino nakon što generiramo cjelobrojni slučajni broj konvertiramo ga u interval [0,1].

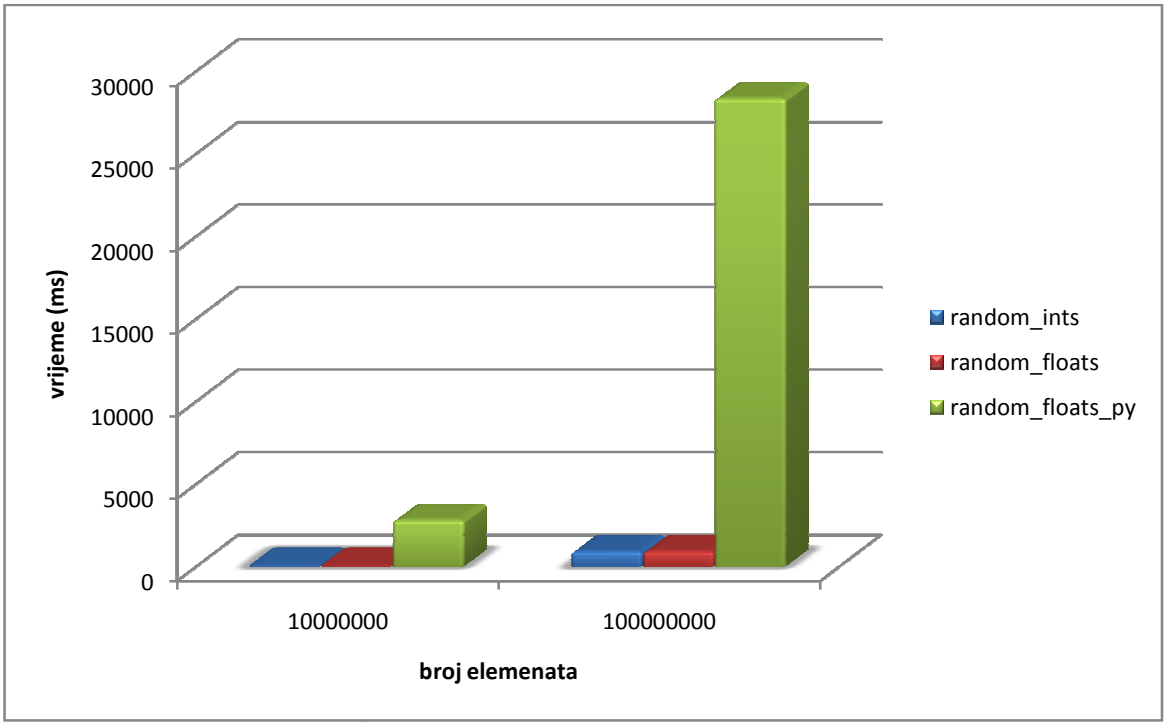
```
RAND_FUNCTION_FLOATS = DATA_SECTION  +  '''
    #CODE
    movdqa xmm0, oword [gadd]
    movdqa xmm1, oword [mult]
    movdqa xmm2, oword [mask]

    mov rdi, qword [array]
    mov ecx, dword [length]
    imul ecx, ecx, 4

    loop:
    sub ecx, 16

    pshufd xmm4, oword [cur_seed], 10110001b
    movdqa xmm5, oword [cur_seed]
    pmuludq xmm5, xmm1
    pshufd xmm1, xmm1, 10110001b
    pmuludq xmm4, xmm1
    pand xmm5, xmm2
    pand xmm4, xmm2
    pshufd xmm4, xmm4, 10110001b
    por xmm5, xmm4
    paddd xmm5, xmm0
```

```
    movdqa oword [cur_seed], xmm5

    ;konverzija cjelobrojne vrijednosti u interval 0.0-1.0
    pand xmm5, oword [sign_mask]
    cvtdq2ps xmm6, xmm5
    mulps xmm6, oword [flt]
    movups oword [edi + ecx], xmm6

    cmp ecx, 0
    jle exit
    jmp loop
    exit:

    #END

    '''

from tdasm import Tdasm, Runtime
import array
asm = Tdasm()
init_code = asm.assemble(INIT_RAND)
rand_code = asm.assemble(RAND_FUNCTION)
rand_float_code = asm.assemble(RAND_FUNCTION_FLOATS)
run = Runtime()
ds = run.load("init", init_code)
ds = run.load("rand", rand_code, ds)
ds = run.load("rand_floats", rand_float_code, ds)
run.run("init") #inicijaliziramo stanje generatora

def random_floats_asm(n):
    arr = array.array('f', B'\x00' * (n*4))

    address, length = arr.buffer_info()
    ds["array"] = address
```

```
    ds["length"] = length
    run.run("rand_floats")
    return arr
```

Metoda `random_floats` skoro je ista kao i `random_ints` jedina razlika je u tome što alociramo niz koji sadrži realne brojeve i zovemo asemblersku rutinu koja generira realne brojeve. Za generiranje slučajnih brojeva implementiran je algoritam LCG koji je jednostavan i brz. Da bismo demonstrirali brzinu instrukcijskog podskupa SSE napraviti ćemo nekoliko različitih implementacija za generiranje niza slučajnih brojeva. Kao prvi test napraviti ćemo implementaciju u pythonu pomoću modula `random`.

```
import random
def random_floats_python(n):
    arr = array.array('f', B'\x00' * (n*4))
    for x in range(n):
        arr[x] = random.random()
    return arr
```



Slika 10. Prikaz vremena trajanja generiranja slučajnih brojeva koristeći implementaciju napisanu pomoću TDAsm i Python implementaciju

Generiranje cjelobrojnih 32-bitnih vrijednosti u Pythonu još je sporije u odnosu na generiranje realnih brojeva te zbog toga nije uključen taj test. Python za implementaciju slučajnih brojeva koristi Mersenne Twister algoritam koji je kvalitetniji ali sporiji od LCG algoritma a za generiranje realnog broja dvostruke preciznosti koristi dva 32-bitna cjelobrojna slučajna broja i uz sve to umetanje brojeva u niz obavlja se u Pythonu. Zbog svih ovih navedenih razloga jasno je zašto je ovakva razlika u rezultatima. Da bismo dobili objektivne rezultate isti algoritam koji je implementiran u asembleru implementirati ćemo u programskom jeziku C++ i napisati ekstenziju za Python tako da možemo usporediti brzine iz Pythona. Za prevođenje ekstenzije koristiti će se Visual Studio 2008 i biti će uključene optimizacije prevoditelja za postizanje maksimalnih preformansi. Tu ćemo prikazati metode napisane u C++ za generiranje cjelobrojnih brojeva i realnih brojeva. Metode primaju pokazivač na prvi element niza, broj elemenata u nizu i početnu vrijednost za generator slučajnih brojeva.

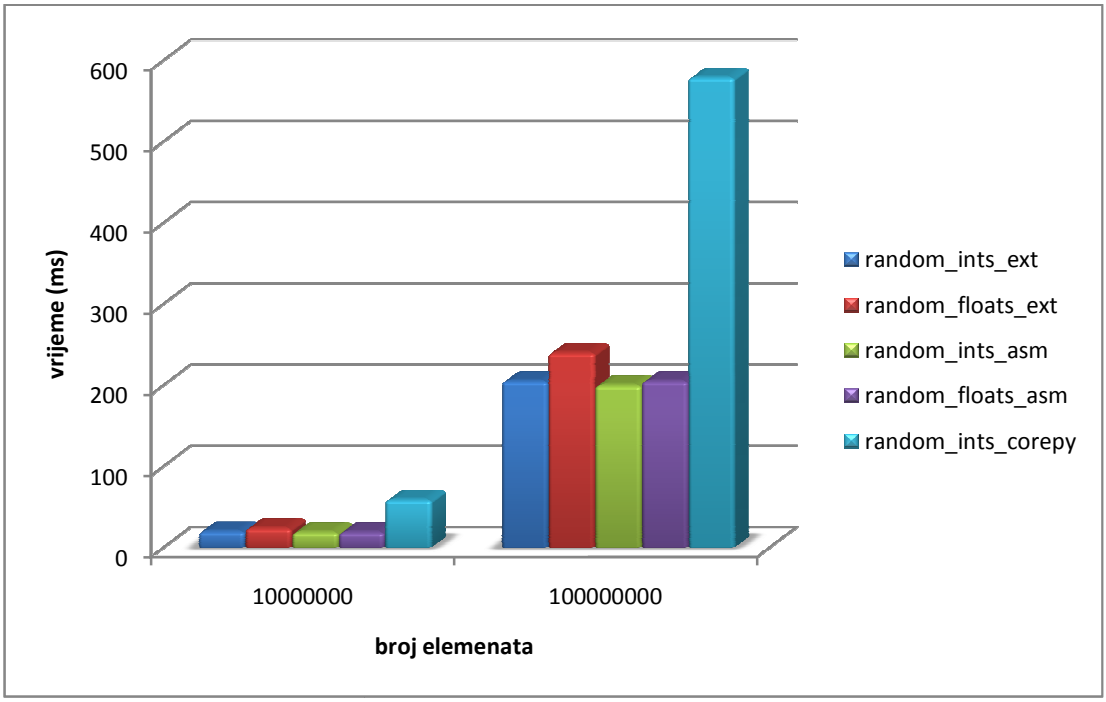
```
void generate_ints(unsigned int* numbers, int n, unsigned int seed)
{
     unsigned int g_seed = seed;
```

```
	for(int x=0; x < n; x++)
	{
		g_seed = (214013*g_seed+2531011);
		numbers[x] = g_seed;
	}
}

void generate_floats(float *numbers, int n, unsigned int seed)
{
	unsigned int g_seed = seed;
	for(int x=0; x<n; x++)
	{
		g_seed = (214013*g_seed+2531011);
		numbers[x] = 0.000000000465661287524f * (g_seed & 0x7FFFFFFF);
	}
}
```

Tu ćemo sad prikazati sve četri metode implementirane u Pythonu, dvije metode koriste rutinu napisanu u asembleru a druge dvije ekstenziju koju smo napisali u C++. Metode kao parametar primaju `array` koji moraju ispuniti slučajnim brojevima.

```
import rnd_ext
def random_ints_ext(arr):
    address, length = arr.buffer_info()
    rnd_ext.GenerateInts(address, n, 675687)
    return arr

def random_floats_ext(arr):
    address, length = arr.buffer_info()
    rnd_ext.GenerateFloats(address, n, 675687)
    return arr

def random_ints_asm(arr):
```

```
    address, length = arr.buffer_info()
    ds["array"] = address
    ds["length"] = length
    run.run("rand")
    return arr


def random_floats_asm(arr):
    address, length = arr.buffer_info()
    ds["array"] = address
    ds["length"] = length
    run.run("rand_floats")
    return arr
```



Slika 11. Prikaz vremena trajanja generiranja slučajnih brojeva koristeći ekstenziju napisanu u C++ i implementacije napisane pomoćuCorePy i TDAsm

Sa grafa vidimo da je brzina generiranja cjelobrojnih vrijednosti približno ista i za asemblersku rutinu i C++ ekstenziju, dok jedino za generiranje realnih brojeva vidimo prednost korištenja asemblerske rutine. Vrlo često programeri pokušaju napisati mali odsječak koda pomoću SSE instrukcija i razočaraju se kad vide da kompajler izgenerira kod koji je jednako brz ili brži nego što su oni ručno napisali.

Jedan od najčešćih uzroka zašto je to tako je memorija. Današnji procesori su puno brži od memorije tako da je pristup memoriji vrlo skupa operacija. Ovaj naš primjer ima upravo taj problem što se može provjeriti tako da se zakomentiraju sve SSE instrukcije koje generiraju slučajni broj i jedino ostavi instrukcija koja upisuje slučajni broj u niz. Tu ćemo i prikazati kritični dio koda da se jasnije vidi uzrok problema.

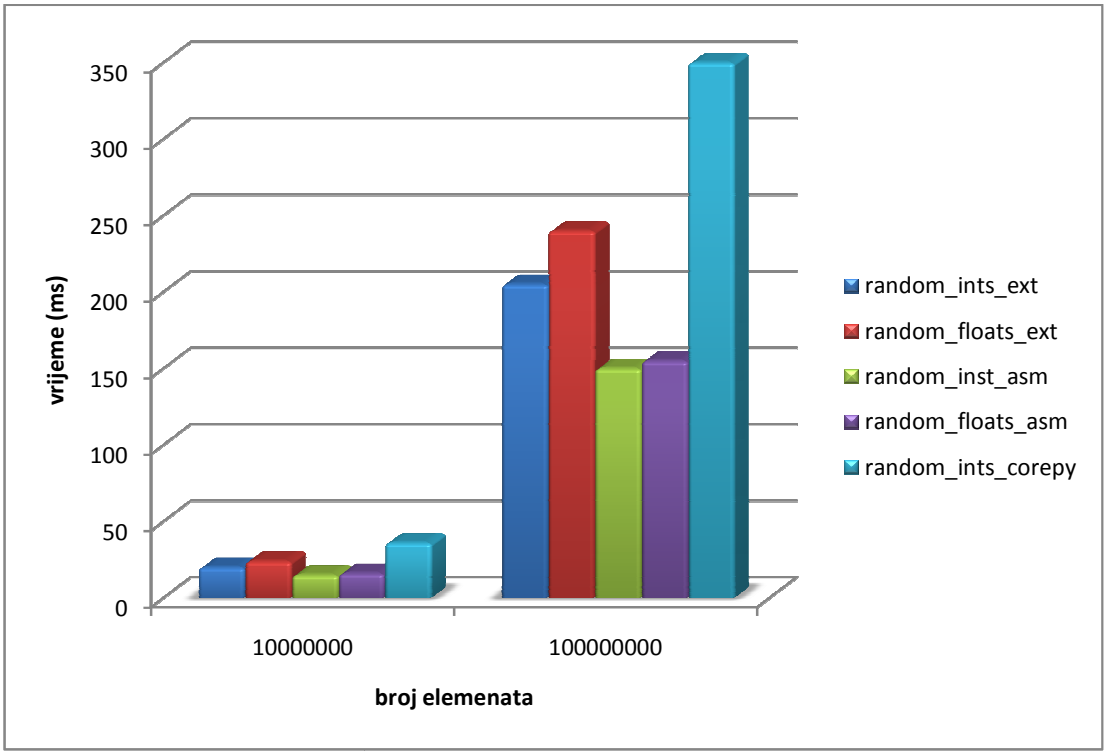
```
loop:
    sub ecx, 16
;zakomentiramo linije koje generiraju slucajni broj jedino
;ostavimo liniju koja upisjuje rezultat u memoiju
    ;pshufd xmm4, oword [cur_seed], 10110001b
    ;movdqa xmm5, oword [cur_seed]
    ;pmuludq xmm5, xmm1
    ;pshufd xmm1, xmm1, 10110001b
    ;pmuludq xmm4, xmm1
    ;pand xmm5, xmm2
    ;pand xmm4, xmm2
    ;pshufd xmm4, xmm4, 10110001b
    ;por xmm5, xmm4
    ;paddd xmm5, xmm0
    ;movdqa oword [cur_seed], xmm5
    movdqu oword [edi + ecx], xmm5 ; upis u memoriju!!!!!!


    cmp ecx, 0
    jle exit
    jmp loop
    exit:
```

Ako pokrenete ovako zakomentirani kod vidjeti ćete da će rezultat izvođenja biti približno isti i da se preko 95% vremena troši na upis rezultata u memoriju. Ovo je razlog zašto u našem primjeru kompajlirani kod i optimizirani SSE kod traju približno isto jer je upis rezultata u memoriju usko grlo. Jasno odmah se postavlja pitanje da li ima smisla pisati kod koristeći SSE instrukcije za koji znamo da će nam usko grlo biti memorijska propusnost. Ima smisla jer ovo i je jedan od razloga

zašto se i dan danas neke stvari pišu u asembleru. Arhitektura procesora omogućava nam razne optimizacije da bismo što efikasnije koristili memoriju. Uvijek kad se rade optimizacije postoje i ograničenja tako da moramo paziti da je adresa niza poravnata na 128-bita. Tu ćemo prikazati ključnu izmjenu koju je potrebno napraviti da bismo ubrzali primjer.

```
 ;movdqu oword [edi + ecx], xmm5 ; zamijenimo sa movntdq
 movntdq oword [edi + ecx], xmm5
; kod generiranja realnih brojeva
;movups oword [edi + ecx], xmm6 ; zamijenimo sa movntps
 movntps oword [edi + ecx], xmm6
```

Sve što je potrebno u našem primjeru je zamijeniti `movdqu`, `movups` sa `movntdq`, `movntps`. Instrukcije `movntdq`, `movntps` osim što upisuju rezultat u memoriju kažu procesoru da se u bliskoj budućnosti te se vrijednosti neće koristiti. Zbog tog naputka procesor će zaobići mehanizam stavljanja tih vrijednosti u priručnu memoriju. Na slici vidimo graf nakon što smo zamijenili instrukcije.



Slika 12. Prikaz vremena generiranja slučajnih brojeva primjenom TDAsm, CorePy i C++ nakon zamjene movdqu sa movntdq instrukcijom

Za slijedeći primjer prikazati ćemo implementaciju konvolucije korištenjem SSE2 instrukcijskog seta. Jednadžba prikazuje dvodimenzionalnu diskretnu konvoluciju.

$$r(i,j) = \sum_n \sum_m s(i-n, j-m)h(n,m)$$

Ako imamo filter h veličine 3x3 za računanje elementa r(i, j) koristimo formulu.

$$\begin{aligned} r(i,j) = {} & h_{0,0} * s(i-1, j-1) + h_{1,0} * s(i, j-1) + h_{2,0} * s(i+1, j-1) + h_{0,1} * s(i-1, j) \\ & + h_{1,1} * s(i,j) + h_{2,1} * s(i+1, j) + h_{0,2} * s(i-1, j+1) + h_{1,2} * s(i, j+1) \\ & + h_{2,2} * s(i+1, j+1) \end{aligned}$$

U primjeru ćemo koristiti filter veličine 5x5. Da bi se lakše razumjela implementacija asemblerske rutine, prvo ćemo prikazati implementaciju u C++ jeziku. Funkcija kao ulaz prima dvodimenzionalni niz realnih brojeva i filtrira taj niz koristeći filter veličine 5x5. Rubne dijelove ulaznog niza ne filtriramo.

```
void convolution5x5(float *input, float *output, int width, int height, float
*h)

{

     float temp = 0.0f;

     float temp_buf[2048];

     for(int y=2; y<height-2; y++)

     {

            for(int x=2; x<width-2; x++)

            {

                  temp = input[(y-2) * width + (x-2)] * h[0] + input[(y-2) *
width + (x-1)] * h[1] + input[(y-2) * width + (x)] * h[2];

                  temp += input[(y-2) * width + (x+1)] * h[3] + input[(y-2) *
width + (x+2)] * h[4];



                  temp += input[(y-1) * width + (x-2)] * h[5] + input[(y-1) *
width + (x-1)] * h[6] + input[(y-1) * width + (x)] * h[7];

                  temp += input[(y-1) * width + (x+1)] * h[8] + input[(y-1) *
width + (x+2)] * h[9];
```

```
                    temp += input[(y) * width + (x-2)] * h[10] + input[(y) *
width + (x-1)] * h[11] + input[(y) * width + (x)] * h[12];

                    temp += input[(y) * width + (x+1)] * h[13] + input[(y) *
width + (x+2)] * h[14];



                    temp += input[(y+1) * width + (x-2)] * h[15] + input[(y+1)
* width + (x-1)] * h[16] + input[(y+1) * width + (x)] * h[17];

                    temp += input[(y+1) * width + (x+1)] * h[18] + input[(y+1)
* width + (x+2)] * h[19];



                    temp += input[(y+2) * width + (x-2)] * h[20] + input[(y+2)
* width + (x-1)] * h[21] + input[(y+2) * width + (x)] * h[22];

                    temp += input[(y+2) * width + (x+1)] * h[23] + input[(y+2)
* width + (x+2)] * h[24];



                    //output[y * width + x] = temp; brze je ako koristimo
meduspremnik

                    temp_buf[x] = temp;

            }

            for(int x=2; x<width-2; x++)

            {

                    output[y*width + x] = temp_buf[x];

            }

     }

}
```

Nakon što smo prikazali implementaciju u jeziku C++ prikazati ćemo implementaciju korištenjem SSE2 instrukcijskog seta.

```
CONVOLUTION5x5 = """

    #DATA

    ; ulazni parametri adresa ulaznog i izlaznog niza dimenzije niza

    uint64 adr_in, adr_out

    uint32 width, height



    ;filter h velicine 5x5 je parametar

    ;filter je zadan u dva dijela zbog SSE instrukcija da bi imali poravnatu
memoriju
```

```
    float filter[20] = 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04,
0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04, 0.04

    float filter2[5] = 0.04, 0.04, 0.04, 0.04, 0.04


    float temp[10000] ;meduspremnik

    float testv[4]

    float result




    #CODE

    mov r15, temp

    ;mnozenje sa registrom je brze pa neke dijelove filtera stavljamo u
registre

    movss xmm11, dword [filter2]

    movss xmm12, dword [filter2 + 4]

    movss xmm13, dword [filter2 + 8]

    movss xmm14, dword [filter2 + 12]

    movss xmm15, dword [filter2 + 16]


    mov eax, dword [width]

    mov ebx, dword [height]

    mov ecx, eax

    ; sirna niza u bajtovima

    imul eax, eax, 4


mov rsi, qword [adr_in] ; registar rsi sadrzi adresu ulaznog niza

    mov rdi, qword [adr_out]; registar rdi sadrzi adresu izlaznog niza


    add rdi, rax ; begin of next line

    add rdi, rax ; begin of next line

    add rdi, 8 ; prvi element izlaznog niza element(2, 2)


    sub ecx, 4 ;rubne dijelove niza ne filtriramo

    sub ebx, 4 ; pa oduzmemo 2 elmenta sa svakog ruba
```

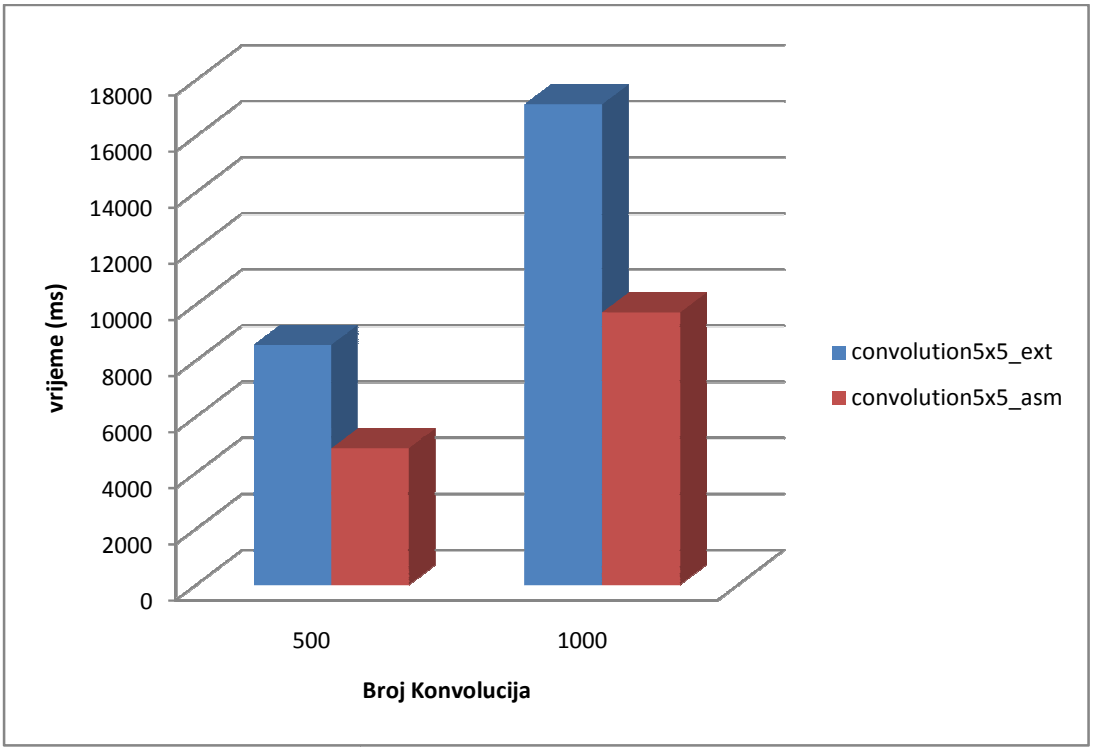
```
    imul r14, rax, 3


    ;ucitavamo 5*4 matricu ulaznog niza i matricu pokaknemo u desno za 4
bajta
    ;zbog toga sto za svaki sljedeci element trebamo matricu pomicati za 4
bajta
    ;ulijevo da bismo ucitali novi element
    movups xmm6, oword [rsi]
    psrldq xmm6, 4
    movups xmm7, oword [rsi + rax]
    psrldq xmm7, 4
    movups xmm8, oword [rsi + 2*rax]
    psrldq xmm8, 4
    movups xmm9, oword [rsi + r14]
    psrldq xmm9, 4
    movups xmm10, oword [rsi + 4*rax]
    psrldq xmm10, 4


    looping: ; glavna petlja
    ;zbog toga sto je filter sirok 5 elemenata a nama u SSE registar moze
    ;najvise stati 4 elemenata, 5 element racunamo odvojeno
    movss xmm0, dword [rsi + 16]
    mulss xmm0, xmm11
    movss xmm1, dword [rsi + rax + 16]
    mulss xmm1, xmm12
    movss xmm2, dword [rsi + 2*rax + 16]
    mulss xmm2, xmm13
    addss xmm0, xmm1
    movss xmm3, dword [rsi + r14 + 16]
    mulss xmm3, xmm14
    movss xmm4, dword [rsi + 4*rax + 16]
    mulss xmm4, xmm15
    addss xmm2, xmm3
    movss xmm5, xmm4
    addss xmm5, xmm0
```

```
    addss xmm5, xmm2


    ; mnozimo
    pslldq xmm6, 4
    movss xmm0, dword [rsi]
    movss xmm6, xmm0
    movaps xmm0, oword [filter]
    mulps xmm0, xmm6



    pslldq xmm7, 4
    movss xmm1, dword [rsi + rax]
    movss xmm7, xmm1
    movaps xmm1, oword [filter + 16]
    mulps xmm1, xmm7

    pslldq xmm8, 4
    movss xmm2, dword [rsi + 2*rax]
    movss xmm8, xmm2
    movaps xmm2, oword [filter + 32]
    mulps xmm2, xmm8

    pslldq xmm9, 4
    movss xmm3, dword [rsi + r14]
    movss xmm9, xmm3
    movaps xmm3, oword [filter + 48]
    mulps xmm3, xmm9

    pslldq xmm10, 4
    movss xmm4, dword [rsi + 4*rax]
    movss xmm10, xmm4
    movaps xmm4, oword [filter + 64]
    mulps xmm4, xmm10
```

```
    addps xmm1, xmm2
    addps xmm3, xmm4
    addps xmm0, xmm1
    addps xmm0, xmm3
    movhlps xmm4, xmm0
    addps xmm0, xmm4
    pshufd xmm4, xmm0, 1
    addss xmm0, xmm5  ; prethodni produkti
    addss xmm0, xmm4

    ;upisujemo rezultat u meduspremnik
    movss dword [r15 + 4*rcx], xmm0 ; r15 is adresa meduspremnika

    ;unutarnja petlja
    add rsi, 4 ;uvecamo adresu ulaznog niza
    sub ecx, 1 ;gledamo da li smo zavrsili jedan redak
    jnz looping
    mov ecx, dword [width]
    sub ecx, 4
    add rsi, 16
    ;sub ebx, 1 ; height - 1

    mov r12d, dword [width]
    sub r12, 4
    xor r10, r10
    test_loop:
    movss xmm0, dword [r15 + 4*r12]
    ;movss dword [rdi + 4*r12], xmm0
    movss dword [rdi + 4*r10], xmm0
    add r10, 1
    sub r12, 1
    jnz test_loop

    add rdi, rax ;mov destination to next line rax = sirina u bajtovima
```

```
    movups xmm6, oword [rsi]
    psrldq xmm6, 4
    movups xmm7, oword [rsi + rax]
    psrldq xmm7, 4
    movups xmm8, oword [rsi + 2*rax]
    psrldq xmm8, 4
    movups xmm9, oword [rsi + r14]
    psrldq xmm9, 4
    movups xmm10, oword [rsi + 4*rax]
    psrldq xmm10, 4

    sub ebx, 1 ; height - 1
    jnz looping

    #END

"""
```

Za tesitranje brzine koristiti će se ulazni niz dimenzija 1024*768 i mjerenje će se obaviti sa različitim brojem konvolucija.



Slika 13. Prikaz vremena računanja konvolucije s filterom 5x5 korištenjem C++ i implementacije u asembleru optimizirane sa instrukcijskim setom SSE2

Sa grafa vidimo da je implementacija konvolucije pomoću instrukcijskog skupa SSE2 skoro duplo brža nego C++ implementacija.

6. Zaključak

Za veliku većinu aplikacija prevođenijezici poput C++ sasvim su dovoljni. Međutim postoji i određen broj aplikacija kao npr. aplikacije za obradu videa, kriptiranje, razne numeričke simulacije i sličnegdje su potrebne maksimalne preformanse. Za postizanje maksimalnih preformansi nužno je koristiti napredne mogućnosti arhitekture poput vektorskih instrukcijskih setova. Početkom sljedeće godine procesori dobivaju instrukcijski set AVX koji proširuje vektorske registre sa 128-bita na 256-bitai instrukcije će umjesto sa dosadašnjih dva operanda moći raditi sa tri operanda što će omogućiti dodatna ubrzanja.

Iako su u radu su prikazana dva primjera gdje je implementacija u asembleru brža od C++ implementacije optimiziranje u asembleru ima smisla ako možemo vektorizirati problem inače se ne isplati jer je vrlo teško napisati kod koji će biti brži od današnjih modernih prevoditelja. Osim vektorskih instrukcija današnji napredniji procesori još imaju iinstrukcije zakodiranje i dekodiranje AES algoritma gdje se vrlo lagano postiže značajno ubrzanje u odnosu na prevođene jezike koji softverski izvršavaju algoritam.

TDAsm asemblertakođer se može koristiti iz programskog jezika C/C++ jer komunikacija Pythona i C/C++ ide u oba smjera.

7. Literatura

1. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, prosinac 2010. http://www.intel.com/Assets/PDF/manual/253665.pdf
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M, prosinac 2010. http://www.intel.com/Assets/PDF/manual/253667.pdf
3. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z, prosinac 2010. http://www.intel.com/Assets/PDF/manual/253667.pdf
4. R. Blum, Professional Assembly Language, Wiley Publishing, Inc., Indianapolis, Indiana, 2005.
5. http://www.corepy.org (2010)

Popis slika

Popis tablica