SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Laboratorijske vježbe

ARHITEKTURE RAČUNALA 2

Programiranje u strojnom jeziku arhitekture x86

Dodatak uputama

Zagreb, prosinac 2012.

1 Stog, parametri, varijable, funkcije, primjeri

U ovom dodatku vježbi biti će objašnjeno kao se koristi stog prilikom poziva funkcija unutar x86 arhitekture. Prije toga korisno je pročitati kratak uvod u x86 asembler (npr. **ovdje** ili **ovdje**). Poznato je da se prilikom poziva funkcije na stog pohranjuju parametri funkcije, povratna adresa, te lokalne varijable. Pogledajmo sljedeći primjer funkcije pisane u C-u:

```
int primjer(int a, int b, int c)
{
  int x,y,z;
  ...
}
```

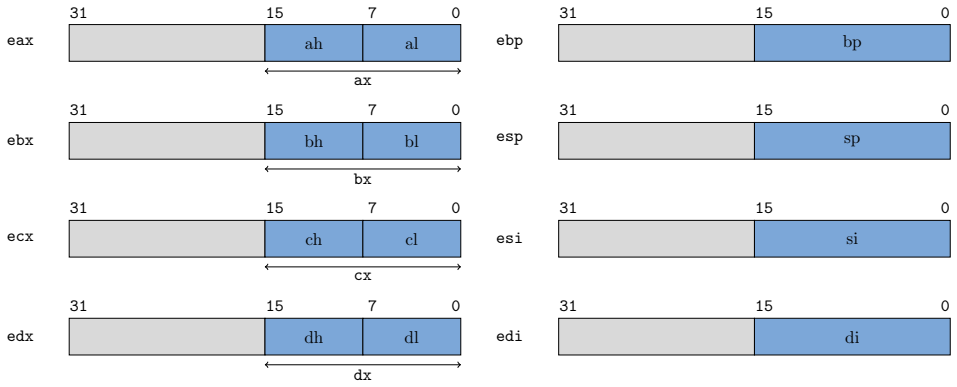
Za funkcijski poziv:

```
r = primjer(a, b, c);
```

Prevoditelj će generirati sljedeći asemblerski kod:

```
push c
push b
push a
call primjer
mov r, eax
```

Parametri funkcije pohranjuju se na stog s desna na lijevo (a je na najmanjoj adresi)[1]. Imena parametara i varijabli samo su simbolična (nisu prikazana takvim imenima i u asembleru, već memorijskim lokacijama npr. [ebp+4], [ebp+8]). Povratna vrijednost funkcije prenosi se preko registra eax, te se po povratku iz potprograma preslikava u varijablu r.



Slika 1: **Registri x86 programskog modela. Registri eax, ebx, ecx i edx su registri opće namjene. Ostali registri imaju posebna značenja. Sve registre možemo koristiti u strojnom programu, no unutar funkcijskih poziva (gdje ih koristimo), moramo ih prije korištenja staviti na stog kako bi se očuvala njihova stara vrijednost (pohranjivanje konteksta). Jedini registri čija stara vrijednost nije bitna (dakle njih ne moramo staviti na stog) su registri eax, ecx i edx.**

Uloge registara esp, ebp i eip su (eip nije prikazan na slici; također nije prikazan niti eflags registar. Prema njegovom imenu (eflags) lako zaključujemo da on sadrži vrijednosti zastavica procesora):

[1] U višim programskim jezicima parametri funkcije mogu biti složeni izrazi, koji se prije stavljanja na stog evaluiraju. Da li će se evaluacija izvesti s desna na lijevo ili obratno, nije standardizirano i nije povezano sa redoslijedom njihovog smještanja na stog.

esp je pokazivač na vrh stoga. On uvijek pokazuje na zadnji korišteni element na stogu (ne na prvi slobodni). Vrh stoga je memorijska lokacija sa najmanjom adresom (u trenutnom okviru stoga). Vrijednost esp može se mijenjati implicitno instrukcijama (npr. push, pop, call, ret) ili izravno instrukcijama (npr. add esp, $16, sub esp, $32).

ebp je registar koji se koristi za referenciranje na parametre funkcije i lokalne varijable unutar trenutnog okvira stoga. Njegovu staru vrijednost uvijek pohranjujemo na stog na početku izvođenja funkcije (cdecl prolog). Vrijednost ebp registra mijenjamo izravno.

eip sadrži adresu sljedeće instrukcije (PC). Instrukcije skoka direktno mijenjaju njegovu vrijednost. Instrukcija poziva potprograma pohranjuje njegovu staru vrijednost na stog. Njegova nova vrijednost postaje adresa potprograma kojeg pozivamo.

Instrukcija "call primjer" se zapravo prevodi kao:

```
push eip
jmp primjer
```

Pogledajmo sada funkciju main (koja poziva funkciju primjer) u C-u:

```
int main(int argc, int **argv)
{
  int r = 0;
  r = primjer(100,200,300);
}
```

Funkcija main u asembleru[2] izgleda ovako (gcc):

```
main:
 push   ebp                  \\pohranimo staru vrijednost na stog
 mov    ebp, esp
 sub    esp, 16              \\pomaknemo vrh stoga za 16B
 mov    DWORD PTR [ebp-4], 0  \\postavimo vrijednost lokalne varijable r
 mov    DWORD PTR [esp+8], 300 \\postavljamo parametre (ne sa push jer je esp vec pomaknut
    )
 mov    DWORD PTR [esp+4], 200
 mov    DWORD PTR [esp], 100
 call   primjer              \\pozovemo funkciju
 mov    DWORD PTR [ebp-4], eax \\povratnu vrijednost preslikamo u lokalnu varijablu
 mov    eax, 0
 leave
 ret
```

Možemo primijetiti kako funkcija main također ima cdecl prolog i epilog (leave). Također gcc prevoditelj ne postavlja parametre funkcije na stog sa push, već odredi koliko je memorije potrebno za lokalne varijable i parametre, te promijeni esp samo jednom. Nakon toga postavlja parametre u slobodne adrese. U našem slučaju imamo 3 parametra veličine 4 bajta. Također imamo 1 lokalnu varijablu veličine 4 bajta. Zbog toga će prevoditelj pomaknuti esp za 16 bajtova (tj. napraviti sub esp,$16). U ovom primjeru je slučajno broj 16 dobra vrijednost. Razlog tome je što segmenti stoga

[2]Koristimo Intel sintaksu (postoji još i AT&T sintaksa). "Čudan" simbol DWORD PTR u intel sintaksi označava veličinu operanda (DWORD = 4B). Taj operand je nužan u instrukcijama gdje na temelju operanda nije moguće utvrditi njegovu veličinu (npr. mov [ebp-4], 2; 2 može biti 1, 2, 4, ili 8 bajtova). U instrukcijama gdje je operand registar taj simbol se može izostaviti (npr. mov [ebp-4], eax; eax ima 32 bita). Postoje još i BYTE PTR = 1B, WORD PTR = 2B, QWORD PTR = 8B

(tj. adrese mem. lokacija) koje funkcija zauzima trebaju (moraju) biti poravnati sa 8 bajtova [3] (poravnati sa 8 znači da je početna adrese segmenta stoga djeljiva sa 8). Budući da je funkcija main početna adresa izvođenja našeg programa, možemo pretpostaviti da je esp inicijalno bio poravnat sa 8 bajtova (prije poziva funkcije main). Pozivom funkcije main postavila se povratna adresa na stog, te se u prologu sa "push ebp" esp smanjio za 4. Kako trebamo 16 bajtova za lokalnu varijablu te parametre, ukupni pomak bi bio 16+4+4 = 24 bajta. 24 je višekratnik od 8, pa stoga zaključujemo da esp ostaje poravnat sa 8. Katkad će prevoditelj u prologu funkcije generirati vrlo čudne instrukcije koje izgledaju ovako:

```
...
sub   esp, neki broj
and   esp, -16
...
```

Time zapravo pomičemo esp za broj bajtova koji nam je potreban za podatke, dok sa drugom instrukcijom doljnjih 4 bita postavljamo na 0 (osiguravamo da je esp djeljiv sa 16; tada je također djeljiv i sa 8). To efektivno znači smanjivanje vrijednosti esp (broj postaje manji ako mu zadnje bitove postavimo na 0), a to je upravo smjer rasta stoga. U epilogu funkcije staru vrijednost esp obnavljamo iz ebp:

```
...
mov   esp, ebp
pop   ebp
ret
```

Za sada je dovoljno znati da je razlog poravnavanju memorije sa 8 bajtova korištenje priručne memorije i veličina linije priručne memorije (procesor priručnu memoriju puni/ažurira blokovima od nekoliko bajtova (ne bajt po bajt), pri čemu su blokovi veličine 2^n). Utjecaj veličine p.m. (i njena brzina) na performansu izvođenja programa biti će tema 3. laboratorijske vježbe.

Iako većina novijih procesora podržava segmentiranje stoga koje nije poravnato sa 8(16) bajtova, različiti operacijski sustavi to neće podržati. Neki će ispravno izvesti zadani program i javiti poruku poput (Segmentation fault), dok će drugi program iznenada prekinuti. Razlog tome su i neke instrukcije (program može sadržavati instrukcije iz više različitih instrukcijskih skupova) koje isključivo mogu pristupati podatcima čije su adrese višekratnici brojeva 8 ili 16. To su obično SIMD instrukcije. Kako bi izbjegli eventualne probleme, dobro je podatke na stogu zauzeti u blokovima po 8(16) bajtova (ako zauzmemo više nego što je potrebno, nije problem, program će ipak raditi brže). Tko želi saznati više o ovom problemu (optimizaciji pristupa memoriji, radu priručne memorije) izuzetno dobar manual može se pronaći **ovdje** (manual 2 (optimizing_assemply.pdf) poglavlje 11). Na istoj stranici nalazi se više materijala koji obrađuju optimizaciju (u asembleru i C++). Kada u programima koristimo blokove memorije (statičke ili dinamičke), dobra je optimizacijska navika da njihova početna adresa bude višekratnik od 8(16).

[3] Ovo vrijedi za sve 32-bitne arhitekture/operacijske sustave koji koriste x86 asemblerske kodove. Za 64-bitne arhitekture/operacijske sustave, prevoditelji će podatke poravnati sa 16 bajtova. Poravnanje sa sa 16B vrijedi uvijek na nekim operacijskim sustavima (Mac OSX i Linux).

Pogledajmo stanje na stogu:

Relativan pomak u B	Memorija	
...	...	
-16	100	← esp
-12	200	
-8	300	
-4	0	varijabla r
0	stari ebp	← ebp, (esp prije sub esp, $28)
+4	stari eip	(povratna adresa od main) (← esp prije main prologa)
+8	argc	
+12	argv	
...	...	

Na dnu stoga nalaze se parametri funkcije main (adrese +8 i +12) [u tablici kao referentnu adresu (0) uzimamo onu na koju pokazuje pokazivač na trenutni okvir stoga, dakle registar ebp. Ostale adrese predstavljaju relativan pomak u bajtovima u odnosu na tu adresu]. Prije cdecl prologa esp je pokazivao na adresu +4 (tamo se nalazi povratna adresa iz main). Zatim je ebp stavljen na stog (adresa 0). To je automatski pomaknulo i esp na adresu 0. Tada je novi ebp postao trenutni esp, te je esp smanjen za 16. Time je završio prolog funkcije main. Lokalna varijabla r nalazi se na adresi -4. Možemo uočiti standardnu organizaciju podataka na stogu (od veće adrese prema manjoj): najprije idu parametri funkcije, zatim povratna adresa i pokazivač na prijašnji okvir stoga, te lokalne varijable. Na adresama -8, -12, i -16 nalaze se parametri funkcije primjer. Ono što još trebamo napraviti kako bismo pokrenuli funkciju primjer je postaviti iznad njenih parametara sadržaj registra eip, te skočiti na početnu adresu od funkcije primjer. To upravo radi instrukcija "call primjer". Slijedi funkcija primjer u C-u, te u asembleru:

```
int primjer(int a, int b, int c)
{
  int x = 1;
  int y = 2;
  int z = 3;

  return a*x+b*y+c*z;
}
```

```
primjer:
 push   ebp                    \\pohranimo staru vrijednost na stog
 mov    ebp, esp
 sub    esp, 16                \\pomaknemo vrh stoga za 16 (3 varijable)
 mov    DWORD PTR [ebp-4], 1   \\x
 mov    DWORD PTR [ebp-8], 2   \\y
 mov    DWORD PTR [ebp-12], 3  \\z

 mov    eax, [ebp+8]           \\eax = a
 mov    edx, eax               \\edx = eax = a
 imul   edx, [ebp-4]           \\edx = a*x
 mov    eax, [ebp+12]          \\eax = b
 imul   eax, [ebp-8]           \\eax = b*y
 add    edx, eax               \\edx = a*x + b*y
 mov    eax, [ebp+16]          \\eax = c
 imul   eax, [ebp-12]          \\eax = c*z
 lea    eax, [eax+edx]         \\eax = eax + edx = a*x + b*y +c*z

 leave
```

```
ret
```

Stog prilikom izvođenja funkcije primjer izgleda kao u tablici ispod. Na početku funkcije izvodi se cdecl prolog, pri čemu se ebp pohranjuje na stog. Novi ebp se postavlja na trenutni esp (koji sada pokazuje na zadnje pohranjeni ebp), te se esp smanjuje za 16 (imamo 3 lokalne varijable od 4 bajta). Za novu referentnu adresu okvira stoga (0) u tablici postavljena je ona na koju pokazuje trenutni registar ebp unutar funkcije (u zagradama se nalaze stari relativni pomaci koji su se koristili u prethodnoj tablici).

Bez obzira što esp pokazuje na praznu lokaciju (u koju može stati još jedna varijabla od 4B), smatramo to polje zauzetim.

U ovoj vježbi, budući da koristimo x86 asembler, možemo zbog jednostavnosti uvesti pravilo da podaci na stogu trebaju biti poravnati sa 8 bajtova (prilikom poziva funkcija). To znači da ćemo vrijednost registra esp mijenjati u 8*n koracima, pri čemu je n cijeli broj.

Relativan pomak u B	Memorija	
...	...	
-16 (-40)	(prazno)	← esp
-12 (-36)	z	
-8 (-32)	y	
-4 (-28)	x	
0 (-24)	stari ebp	← ebp, (esp prije sub esp, $16)
+4 (-20)	stari eip	(povratna adresa od primjer)
+8 (-16)	100	a
+12 (-12)	200	b
+16 (-8)	300	c
+20 (-4)	0	varijabla r
+24 (0)	stari ebp	
+28 (+4)	stari eip	(povratna adresa od main)
+32 (+8)	argc	
+36 (+12)	argv	
...	...	

2 2 primjera

U nastavku se nalaze primjeri 2 funkcije u C-u i asembleru (asemblerski kod je direktan prijevod C funkcija, bez primjene optimizacija).

Potrebno je napisati funkciju koja rekurzivno računa sumu prirodnih brojeva manjih ili jednakih n:

```
int suma(int n)
{
  if(n <= 0) return 0;
  return n + suma(n-1);
}
```

```
suma_asm:
 push   ebp
 mov    ebp, esp
 sub    esp, 24

 cmp    DWORD PTR [ebp+8], 0
 jg     L1
 mov    eax, 0
 jmp    L2

L1:
 mov    eax, [ebp+8]
 sub    eax, 1
 mov    [esp], eax
 call   suma_asm
 add    eax, [ebp+8]

L2:
 mov esp, ebp
 pop ebp
 ret
```

Potrebno je napisati funkciju koja sve elemente zadanog cjelobrojnog polja povećava za 1:

```
void polje1(int *x, int n)
{
  for(int i = 0; i < n; ++i) x[i]++;
}
```

```
polje1_asm:
 push   ebp
 mov    ebp, esp
 sub    esp, 8

 mov    DWORD PTR [ebp-4], 0
 jmp    L10

L11:
 mov    eax, [ebp-4]
 sal    eax, 2
 add    eax, [ebp+8]
 mov    edx, [eax]
```

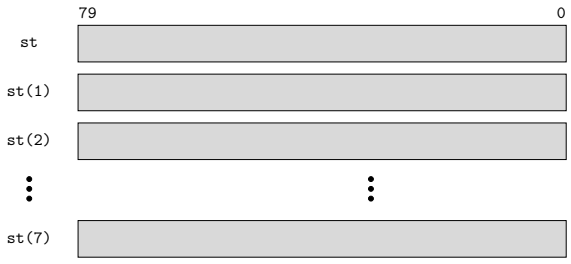
```
 add   edx, 1
 mov   [eax], edx
 add   DWORD PTR [ebp-4], 1

L10:
 mov   eax, [ebp-4]
 cmp   eax, [ebp+12]
 setl  al
 test  al, al
 jne   L11

 mov esp, ebp
 pop ebp
 ret
```

3 x87 instrukcijski podskup

Sa x87 označavamo podskup instrukcijskog skupa x86 koji sadrže instrukcije za operacije s brojevima sa pomičnim zarezom. Glavna komponenta FPU (floating point) jedinice procesora je FPU stog. To je poseban stog (nalazi se na procesorskom čipu) i neovisan je o osnovnom stogu koji se nalazi u RAM-u. Sastoji se od 8 elemenata duljine po 80 bita (slika 2).



Slika 2: **FPU stog. Može sadržavati do 8 elemenata.**

Elementi se dohvaćaju izravno preko imena st(0), st(1), ..., st(7) (ako piše samo st onda je to st(0)). Mnoge instrukcije implicitno dohvaćaju elemente (u tom slučaju to su elementi st i st(1) ili samo st). Postoje posebne instrukcije koje učitavaju najčešće konstante: fldz stavlja 0.0 na vrh stoga, fld1 stavlja 1.0. Sve ostale vrijednosti potrebno je učitati iz memorije. Niti jedna aritmetička x87 instrukcija ne izvodi operacije sa memorijskim elementima izravno (svi se prije toga moraju učitati na FPU stog). Pogledajmo:

```
fld1 /*stavljamo 1 u st, tj. radimo push na stog*/
     /*prethodni st sada ce biti st(1) ...*/

fld    DWORD PTR [eax] /* ucitavamo vrijednost elementa koji se */
                       /* nalazi na adresi koja je u registru eax */
                       /* npr. u varijablu a (float *a) stavimo u eax */
                       /* instrukcija ce u st staviti a[0]*/
fmulp  /* mnozimo elemente st i st(1) te rezultat pohranimo u st i st(1)*/
       /* zatim st maknemo sa stoga (pop), st(1) sada postaje st i sadrzi rezultat*/
```

Pogledajmo sljedeći primjer u kojem želimo napisati funkciju koja skalarno množi 2 vektora zadane duljine. C++ kod izgleda ovako:

```
#include<iostream>

extern "C" void mnozi_float_asm(float*, float*, float*, int);

int main()
{
	int n = 8;
	__attribute__ ((aligned(32))) float u[8] = {1,2,3,4,5,6,7,8};
	__attribute__ ((aligned(32))) float v[8] = {-1,0,5,2,3,6,11,5};
	__attribute__ ((aligned(32))) float w[8];

	//broji pripadne elemente polja u i v duljine n (rezultat je polje w)
	mnozi_float_asm(u,v,w,n);

	for(int i = 0; i < n; ++i)
	{
		std::cout << w[i] << " ";
	}

	std::cout << std::endl;
}
```

Jasno je što u programu želimo napraviti osim možda definicije polja. Takva definicija osigurava da polja (imamo statička polja, dakle nalaze se na stogu) imaju početne adrese koje su djeljive sa 32 (poravnate sa 32B). U ovom primjeru to nije nužno, no pri korištenju instrukcija u zadatku sa SSE biti će nužno. Češeće ćemo trebati dinamička polja koja su također poravnata sa npr. 16 ili 32B (dinamička polja zauzimaju se u dijelu memorije koji zovemo gomila ili heap). Postoje verzija funkcije malloc koja nam to osigurava:

```
#include <immintrin.h>
...
int n = 10;
float *a = (float*) _mm_malloc(sizeof(float)*n,32); //a je djeljiv sa 32
...
_mm_free(a);
```

Navedena malloc (i pripadna free) funkcija su intrinsične (ugrađene) funkcije koje su specifične za korišteni prevoditelj. Detaljnije o takvim funkcijama piše u odjeljku 4.

Funkcija množenja (.s datoteka) izgleda ovako:

```
mnozi_float_asm:
                    /* cdecl prolog: */
  push  ebp         /* spremi ebp */
  mov   ebp, esp    /* ubaci esp u ebp */
  push  ebx         /* ove registre stavljamo na stog*/
  push  esi
  push  edi

  xor   eax, eax    /* eax = 0, brojac */
  mov   ebx, [ebp+20] /* duljina polja n*/
  mov   esi, [ebp+8]  /* polje u */
  mov   edi, [ebp+12] /* polje v */
  mov   ecx, [ebp+16] /* polje w */

2:
```

```
fld  DWORD PTR [esi+eax*4] /* st = u[i] */
fld  DWORD PTR [edi+eax*4] /* st = v[i], st(1) = u[i] */

fmulp                      /* st = u[i]*v[i], pop st(1) */
fstp DWORD PTR [ecx+eax*4] /* w[i] = st, pop st */

inc eax             /* i = i + 1 */
cmp eax, ebx        /* i < n ?*/
jne 2b              /* Skok na labelu 2. Pise 2b. b dolazi od back */
                    /* Sintaksno pravilo gcc prevoditelja trazi da kada */
                    /* zelimo skociti na labelu koja je prije (na manjoj adresi) */
                    /* dodamo b na kraju. */

pop edi             /* dohvacamo stare vrijednosti */
pop esi
pop ebx
                    /* cdecl epilog: */
pop ebp             /* umjesto 'add esp,4, pop ebp' moze biti 'leave'*/
ret                 /* povratak iz potprograma */
```

Prevođenjem programa (`g++ -m32 main_gcc.cpp p_asm.s` i njegovim izvođenjem dobivamo očekivani ispis:

```
-1 0 15 8 15 36 77 40
```

Za vježbu pokušajte nacrtati stanje na FPU stogu za vrijeme izvođenja prikazane funkcije. U nastavku se nalaze tablice najčešće korištenih x87 instrukcija.

Tablica 1: Osnovne instrukcije za rad sa x87 stogom

Mnemonik	**Operandi**	**Značenje**
`finit`	-	inicijaliziraj FPU i očisti stog
`fld`	(addr)	Sadržaj *addr pokazivača push na FPU stog
`fld`	st(x)	Element st(x) push na FPU stog
`fld1`	-	1.0 push na stog
`fldz`	-	0.0 push na stog
`fst`	st(x)	st(x) = st
`fstp`	st(x)	st(x) = st, pop st
`fst`	(addr)	*addr = st
`fstp`	(addr)	*addr = st, pop st
`fxch`	-	zamjeni sadržaje st(1) i st
`fxch`	st(x)	zamjeni sadržaje st(x) i st

Tablica 2: Osnovne logičke x87 instrukcije

Mnemonik	**Operandi**	**Usporedi st sa**
`fcom`	-	st(1)
`fcom`	st(x)	st(x)
`fcom`	(mem)	*mem
`ftst`		0.0
`fcomp`	-	st(1), pop st
`fcomp`	st(x)	st(x), pop st
`fcomp`	(mem)	*mem, pop st
`fcompp`		st(1), pop st, pop st(1)
`fstsw`	ax	ax = FPU registar stanja. Zatim sa instrukcijama test ili bt ispitujemo ax. Ako je rezultat usporedbe st > onda su bitovi (14, 10, 8) u ax (0, 0, 0); ako je st < onda su ti bitovi (0, 0, 1); ako je st = onda su (1, 0, 0)

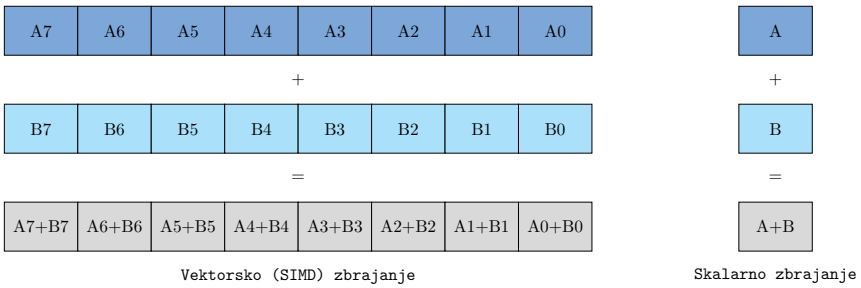
Tablica 3: Osnovne aritmetičke x87 instrukcije

Mnemonik	Operandi	Značenje
`fadd(fmul)`	-	zamjeni st i st(1) njihovom sumom (produktom)
`fadd(fmul)`	(mem)	zamjeni st sa sumom (produktom) st i *mem
`fadd(fmul)`	st, st(x)	zamjeni st sa sumom (produktom) st i st(x)
`fadd(fmul)`	st(x), st	zamjeni st(x) sa sumom (produktom) st i st(x)
`fadd(fmul)`	st(x), st	zamjeni st(x) sa sumom (produktom) st i st(x), pop st
`fsub`	-	zamjeni st i st(x) sa st(1)-st
`fsub`	(mem)	zamjeni st sa st - *mem
`fsub`	st, st(x)	st-= st(x)
`fsub`	st(x), st	st(x)-= st
`fsubp`	st(x), st	st(x)-= st, pop st
`fsubr`	-	zamjeni st i st(1) sa st-st(1)
`fsubr`	(mem)	zamjeni st sa *mem-st
`fsubr`	st(x), st	st(x) = st-st(x)
`fsubr`	st, st(x)	st = st(x)-st
`fsubpr`	st(x), st	st(x) = st-st(x), pop st
`fabs`	-	st = abs(st)
`fchs`	-	st = -st

4 Vektorske instrukcije tipa SIMD

SIMD (eng. Single Instruction Multiple Data) Instrukcije tipa SIMD omogućuju istodobno izvođenje operacije nad većim brojem podataka.

Kao ilustraciju možemo naveti primjer zbrajanja dva vektora (operacija C = A + B) (slika 3).



Slika 3: **Operacija zbrajanja elemenata vektora `A` i `B` na SIMD (vektorskom) procesoru i skalarno zbrajanje.** Ako su latencije SIMD i skalarnih instrukcija jednake, SIMD operacija je 8 puta brža.

Skalarna izvedba takvog zbrajanja izvodila bi zbrajanje jednog elementa vektora `A` sa jednim elementom vektora `B`, te pohranjivala rezultat u odgovarajući element vekotra `C`. Istovijetna SIMD instrukcija bi istovremeno zbrojila `n` elemenata vektora `A` sa `n` elemenata vektora `B`, te rezultat pohranila u odgovarajućih `n` elemenata vektora `C`. Stoga možemo reći da je SIMD, ili vektorska, instrukcija `n` puta brža (`n` predstavlja broj elemenata koji se mogu pohraniti u vektorski registar SIMD instrukcije).

Primjetimo i jedan manji detalj: vektori podataka (`A,B,C`) su pohranjeni kao nizovi slijednih memorijskih lokacija koji mogu imati proizvoljnu duljinu i koja nije nužno djeljiva sa `n`. Stoga ako SIMD instrukcija izvodi operaciju sa po `n` slijednih elemenata, ona neće moći u potpunosti izvesti tu operaciju. Taj problem se može rješiti tako da se vektori produlje dodatnim (praznim) elementima kako bi njihova duljina postala djeljiva sa `n`. Operacija se zatim može izvesti i na tim dodatnim elementima, no te elemente nećemo uzeti u obzir prilikom čitanja rezultata. Drugo rješenja je izvoditi SIMD operacije onoliko puta koliko je duljina vektora podataka djeljiva sa `n`, a za ostale elemente operaciju izvesti skalarno. Moderni procesori koriste sljedeće SIMD instrukcijske skupove:

- SSE (eng. Streaming SIMD Extensions)
- MMX (eng. MultiMedia Extension)
- AVX (eng. Advanced Vector Extension)

Na primjer, istrukcijske skup koje podržava procesor na Linux operacijskom sustavu možemo ispitati tako da ispišemo datoteku `cpuinfo`:

```
$ cat /proc/cpuinfo
```

Kasnije ćemo pokazati kako iste podatke dobiti korištenjem instrukcije `cpuid` unutar C programa. Na Windows operacijskom sustavu to možemo postići korištenjem programa **CpuZ**. Na primjer: procesor Intel Core 2 Duo P8600 podržava instrukcijske skupove (od ovdje spomenutih): mmx, sse, sse2, ssse3, sse4_1. Noviji procesor Intel Core i7-2600 podržava instrukcijske skupove: mmx, sse, sse2, ssse3, sse4_1, sse4_2 i avx. U nastavku ćemo ukratko reći nešto o instrukcijskim skupovima mmx, sse i najnovijem avx, te prikazati nekoliko instrukcija iz svakog od njih u obliku kraćih programskih odsječaka. Vrlo kratki asemblerski programski odsječci mogu se smjesiti zajedno sa C/C++ kodom u obliku `inline` zapisa (ti zapisi imaju prefiks **__asm__** ili **__asm**), no duže asemblerske odsječke bolje je staviti u zasebnu datoteku (ima ekstenziju .s). Ukoliko se piše nekoliko slijednih **asm** zapisa postoji opasnost da optimizirajući prevoditelj promijeni njihov redoslijed prilikom optimizacije koda te time promijeni smisao programa (bolje je pisati manje, ali cjelovite blokove asemblerskog koda). Zbog sličnog razloga, može se pojaviti i problem sa korištenjem labela i naredbi skoka. U asemblerskim primjerima koje ćemo u nastavku pokazati poštovati ćemo to pravilo. U sljedećem podpoglavlju prikazujemo kako je programski moguće odrediti podržane instrukcijske skupove procesora korištenjem instrukcije `cpuid`.

4.1 Određivanje podržanih instrukcijskih skupova procesora instrukcijom `cpuid`

`Cpuid` instrukcija (eng. CPU IDentification) služi za programsko određivanje informacija o tipu i karakteristikama procesora koji se nalazi u sustavu. Instrukcija `cpuid` nema parametre, već indirektno koristi registar `eax` koji se postavlja na određenu vrijednost neposredno prije poziva instrukcije `cpuid`. Povratne vrijednosti instrukcije, ovisno o vrijednosti registra `eax`, nalazit će se u jednom od registara `eax`, `ebx`, `ecx` ili `edx`. Povratne vrijednosti interpretiraju se u ovisnosti o početnoj vrijednosti registra `eax`. Na primjer, ako ispitujemo karakteristike procesora, povratne vrijednosti će se nalaziti u registrima `edx` i `ecx`. Svaki bit tih registara predstavlja jednu karakteristiku procesora koja je prisutna ako je on postavljen na `1` ili nije ako je postavljen na `0`. Popis značenja pojedinih bitova povratnih vrijednosti registara `edx` i `ecx` nalazi se na stranicama proizvođača procesora. Za Intel procesore, priručnici su dostupni **ovdje**.

Ako postavimo registar `eax = 0x00000000`, instrukcija `cpuid` ima sljedeće povratne vrijednosti: `eax` će biti postavljen na maksimalni broj koji se može postaviti u registar `eax` prije poziva instrukcije `cpuid`, dok će registri `ebx`, `edx` i `ecx` sadržavati po 4 znaka kodnog imena proizvođača procesora. Pogledajmo primjer (ovo je gcc inline programski odsječak sa asemblerskim kodom pisan u AT&T sintaksi. Ako ne razumijete značenje svih detalja, nije presudno; pogledajte što radi. AT&T sintaksa obrnutim redoslijedom piše izvorišni i odredišni registar):

```
unsigned char arr[13];
int max_broj;

__asm__ volatile (
```

Tablica 4: Vrijednosti registra `eax` prije poziva instrukcije `cpuid` i informacije koje za pojedine vrijednosti dobivamo.

Vrijednost registra eax	**Informacija koju daje cpuid instrukcija**
0x00000000	identifikator proizvođača
0x00000001	bitovi značajki procesora
0x00000002	opisnici priručne memorije i TLB polja
0x00000003	serijski broj procesora
0x80000000	najveća podržana vrijednost `eax` za "visoke" funkcije (vrijednosti koje počinju sa 0x8)
0x00000001	dodatni bitovi značajki procesora
0x80000002, 0x80000003 0x80000004	brend ime procesora
0x00000005	značajke L1 priručne memorije
0x00000006	značajke L2 priručne memorije
0x00000007	informacije o taktu, naponu i temperaturi procesora
0x00000008	informacije o fizičkom i virtualnom adresnom prostoru

```
    "xorl %%eax, %%eax \n\t"   //postavi a registar na 0x0
    "cpuid \n\t"               //pozovi cpuid instrukciju
    "movl %%eax, %0 \n\t"      //pohrani rezultat iz registra a
    "leal %1, %%eax \n\t"      //ucitaj adresu pocetka niza
    "movl %%ebx,0(%%eax) \n\t" //pohrani prva 4 bajta (znaka)
    "movl %%edx,4(%%eax) \n\t" //sljedeca 4 bajta
    "movl %%ecx,8(%%eax) \n\t" //sljedeca 4 bajta
    "movb $0,12(%%eax) \n\t"   //zavrsna 0
    : "=r"(max_broj)           //povratne varijable
    : "m"(*arr)                //ulazne varijable
    : "eax","ebx","ecx","edx"  //koristeni registri
  );

printf("Kod proizvodaca = %s maksimalni eax = %d\n",arr,max_broj);
```

Obrazložimo ukratko značenje ovog programskog odsječka. Ovo je takozvani inline asemblerski kod koji je dodan u datoteku sa C/C++ programom. Prilikom prevođenja prevoditelj će ga umetnuti u datoteku zajedno sa strojnim kodom ostalih elemenata višeg progamskog jezika. Sintaksa dopušta korištenje varijabli višeg programskog jezika, unutar asemblerskog koda čime je omogućena jednostavna komunikacija sa kodom višeg programskog jezika. Prisjetimo se, sve varijable ili objekti koje koristimo unutar funkcija C programa nalaze se ili na stogu ili na gomili memorijskog prostora pridruženog procesu koji izvodi program [4]. Prevoditelj prilikom prevođenja inline asemblerskog koda zamjenjuje njihovo ime adresom (ako se radi o memorijskim operandima; njih obično dohvaćamo preko pokazivača) ili vrijednošću, te ih (ovisno o instrukcijama) učitava u neki od registara procesora. Prikazani kod, pohraniti će 12 znakova u polje `arr` koje smo proslijedili kao parametar, te jednu cjelobrojnu vrijednost u varijablu `max_broj`. Funkcija `printf` daje ispis:

```
Kod proizvodaca = GenuineIntel maksimalni eax = 13
```

Ispis će se razlikovati ovisno o procesoru kojeg imamo u sustavu.

Informaciju o brendu procesora dobivamo tako da u registar `eax` upišemo konstantu 0x80000000

[4] Postoje i registarske varijable koje se mogu nalaziti samo u registru procesora. U C/C++ programu to označavamo ključnom riječ `register` ispred tipa varijable. Prevoditelju to služi samo kao naputak, ne i pravilo. Prevoditelj, u cilju optimizacije performanse, može i samostalno neku varijablu učiniti registarskom

te pozovemo instrukciju `cpuid`. Ako informacija o brendu postoji, potrebno je 3 puta pozivati instrukciju `cpuid` sa vrijednostima registra `eax` redom: `0x80000002`, `0x80000003`, `0x80000004` pri čemu ćemo svaki puta kao povratnu vrijednost dobivati po 16 okteta u registrima `eax`, `ebx`, `ecx` i `edx`. Na poslijetku, ako dobivenih 48 okteta ispišemo kao niz znakova dobit ćemo sljedeći ispis:

```
Intel(R) Core(TM) i7-2720QM CPU @ 2.20GHz
```

Ispitajmo sada da li procesor koji koristimo podržava instrukcijske skupove MMX, SSE i AVX, budući da ćemo u sljedećim potpoglavljima prikazati kratke programske odsječke koji koriste instrukcije iz tih skupova. Prema tablici 4 u registar `eax` upisujemo vrijednost 1.

```
int c,d;

__asm__ volatile (
    "movl $1, %%eax \n\t"       //postavi a registar na 0x1
    "cpuid \n\t"                //pozovi cpuid instrukciju
    : "=c" (c), "=d"(d)         //povratne varijable
    :                           //ulazne varijable
    :
  );

cout<<boolalpha<<"Procesor podrzava Hyper-threading = "<<(((1<<28)&d)!=0)<<endl;

cout<<"Podrzani instrukcijski skupovi:"<<endl;
cout<<boolalpha<<"MMX = "<<(((1<<23)&d)!=0)<<endl;
cout<<boolalpha<<"SSE = "<<(((1<<25)&d)!=0)<<endl;
cout<<boolalpha<<"SSE2 = "<<(((1<<26)&d)!=0)<<endl;
cout<<boolalpha<<"SSSE3 = "<<(((1<<9)&c)!=0)<<endl;
cout<<boolalpha<<"SSE4.1 = "<<(((1<<19)&c)!=0)<<endl;
cout<<boolalpha<<"SSE4.2 = "<<(((1<<20)&c)!=0)<<endl;
cout<<boolalpha<<"AVX = "<<(((1<<28)&c)!=0)<<endl;
```

Za procesor `Intel(R) Core(TM) i7-2720QM CPU @ 2.20GHz` dobivamo ispis:

```
Procesor podrzava Hyper-threading = true
Podrzani instrukcijski skupovi:
MMX    = true
SSE    = true
SSE2   = true
SSSE3  = true
SSE4.1 = true
SSE4.2 = true
AVX    = true
```

Vidimo da su podržani svi SIMD instrukcijski skupovi budući da se radi o novijem modelu procesora proizvedenom 2011. godine. To nam sugerira da ćemo sve SIMD primjere moći prevesti i izvesti na tom procesoru. Ako bi prijašnji programski odsječak izveli na starijem procesoru (2009.) koji daje informaciju o brendu `Intel(R) Core(TM)2 Duo CPU P8600 @ 2.40GHz`, dobili bismo sljedeći ispis:

```
Procesor podrzava Hyper-threading = true
Podrzani instrukcijski skupovi:
MMX    = true
SSE    = true
SSE2   = true
SSSE3  = true
```

```
SSE4.1 = true
SSE4.2 = false
AVX    = false
```

Vidimo da taj procesor ne podržava SIMD instrukcije iz skupa AVX. Ukoliko bismo ipak pokušali izvesti program koji sadrži AVX instrukcije, program bi se prekinuo uz (očekivanu) poruku o grešci: `Illegal instruction`.

4.2 Intrinsične funkcije

Intrinsične funkcije (eng. intrinsic functions) su funkcije koje implementira prevoditelj izravno zamjenjujući funkcijski poziv u višem programskom jeziku nizom strojnih instrukcija. Intrinsične funkcije se prevode kao inline funkcije pa prilikom njihovog izvođenja ne postoji funkcijski poziv koji bi stvorio usporenje (eng. function call overhead). S obzirom da su te funkcije pisane specifično za zadani programski jezik, prevoditelj i arhitekturu, omogućuju bolju optimizaciju koda. Često se nazivaju i ugrađene funkcije (eng. builtin functions). Ovakve funkcije obično se koriste za ostvarivanje paralelizacije u jezicima u kojima paralelizacija ili vektorizacija nije izravno podržana. Na primjer OpenMP api (služi za ostvarivanje paralelizacije programa na višejezgrenom procesoru) ostvaren je uporabom intrinsičnih funkcija.

S obzirom da su intrinsične funkcije programeru prikazane kao funkcije višeg programskog jezika, programski kod će biti znatno čitljivji i imati će bolju organizaciju, te u nekim slučajevima i bolju performansu nego inline asemlerski odsječci (zbog uključenih optimizacija). Nedostatak programskog koda koji koristi intrinsične funkcije je smanjena prenosivost (eng. portability).

Popis intrinsičnih funkcija za GCC prevoditelj i sve arhitekture za koje postoji GCC prevoditelj dostupan je ovdje (GCC prevoditelj ih zove ugrađene funkcije): **GCC**.

Za Microsoft C++ prevoditelj i arhitekture ARM, x86 i x64 popis je dostupan ovdje: **Microsoft C++**.

Za Intel C++ prevoditelj i Intel arhitekture popis se nalazi ovdje: **Intel C++**.

Intrinsične funkcije uključuju mnoge aritmetičke funkcije (npr. abs, sqrt, ceil, max, log), trigonometrijske funkcije (npr. cos, cosh, acos, atan2), funkcije za operacije nad znakovima ili nizovima znakova (npr. iswalpha, iswdigit, iswlower, fprintf, strcat) ili funkcije za rad s memorijom (npr. malloc, memset, memcpy).

Intrinsične funkcije definirane i za instrukcijske skupove MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, te instrukcija za kriptiranje podataka (eng. Advanced Encryption Standard, AES). Pogledajmo primjer:

```
__attribute__ ((aligned(32))) float a[]={1,2,3,4,5,6,7,8};
__m256 ymm3 = _mm256_load_ps(a);
```

Intrinsična funkcija `_mm256_load_ps` prevodi se kao instrukcija `vmovaps` (AVX instrukcija):

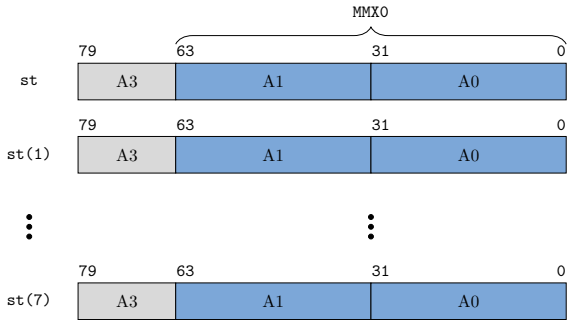
```
movaps (%eax),ymm0
```

Ta funkcija (instrukcija) učitava 8 decimalnih brojeva jednostruke preciznosti u vektorski registar. Adresa početka polja proslijeđuje se kao argument funkcije (radi se o polju `a`, njegova adresa se

u našem primjeru nalazi u registru `eax`). Također primijetimo da smo prije polja a stavili atribut `aligned(32)`. Taj atribut govori prevoditelju da polje `a` na stogu poravna sa 32 bajta (tj. početna adresa polja `a` biti će djeljiva sa 32). To je nužno, jer instrukcija `movaps` zahtijeva da podaci budu poravnati sa 32 bajta. Napomenimo i to, da se atribut `aligned` može koristiti samo za statička polja (kao u našem slučaju), dok se za dinamička polja koriste naprednije funkcije za zauzimanje memorije i poravnavanje adresa (npr. `_mm_malloc`). Možemo primjetiti i to da smo u C programu varijablu označili sa `ymm3`, dok će ona stvarno koristiti registar `ymm0`. To je zato što je u C programu niz znakova `ymm3` ime varijable kojeg možemo zadati u skladu sa pravilima C jezika (npr. mogli smo napisati i abc). U asembleru `ymm0` je ime registra kojeg moramo označiti njegovim imenom. Prevoditelj sam određuje koje varijable će pridružiti pojedinim registrima.

4.3 Instrukcijski skup MMX

Instrukcijski skup MMX (eng. Multi Media eXtensions) je proširenje standardnog Intelovog instrukcijskog skupa instrukcijama tipa SIMD uvedeno 1996. godine. MMX instrukcijski skup uvodi 57 novih operacijskih kodova (instrukcija), 64-bitni tip podataka označen sa `quadword` (dvostruka dvostruka riječ), te 8 novih 64-bitnih registara ostvarenih u obliku elemenata stoga x87 FPU jedinice (jedinice za aritmetiku sa pomičnim zarezom) nazvanih `MMX0 - MMX7`. To znači da će svaka promjena elemenata stoga prilikom skalarnih operacija koje koriste brojeve s pomičnim zarezom izazvati i promjenu jednog ili dva MMX registra. Vrijedi i obratno. Svaka promjena nekog od MMX registara mijenja i stanje stoga x87 FPU jedinice. Registre MMX adresiramo direktno njihovim imenom (npr. `MMX0`, `MMX3`). Elemente stoga x87 FPU jedinice možemo adresirati unutar x87 instrukcije tako da navedemo njihov indeks (npr. `st` je 1. element stoga, `st(3)` je 4. element stoga). Korištenje stoga u x87 instrukcijama mora biti u skladu sa dozvoljenim operacijama za rad sa stogom (npr. uključena zastavica `pop` x87 instrukcije ukoliko je stog prazan rezultirala bi prekidom izvođenja programa; MMX registar ne može biti prazan). Navedimo i to da su svi elementi stoga x87 FPU jedinice 80-bitni, pa su MMX registri zapravo sadržani u njima (slika 4).



Slika 4: **Registri instrukcijskog skupa MMX**: sadrži ukupno 8 64-bitnih registara, koji su ujedno i elementi stoga x87 FPU jedinice. Svaki element stoga (oznaka `st(x)` ima širinu 80-bita.

MMX registar može sadržavati:

- 1 64-bitni cijeli broja (`long ili long long`),
- 2 32-bitna cijela broja (`int`),
- 4 16-bitna cijela broja (`short int`), te
- 8 znakova (`char`).

Možemo uočiti da MMX instrukcijski skup ne podržava brojeve s pomičnim zarezom kao elemente svojih registara (iako bi to mogli očekivati s obzirom da koristi registre FPU jedinice).

Primjeri nekoliko čestih instrukcija iz instrukcijskog skupa MMX prikazano je u tablici 5.

Kratak primjer programa koji koristi MMX instrukcije (skalarni produkt vektora `a` i `b`):

```
short DotProduct2(short *a, short *b, int size)
{
	short p, rez;
	//__m64 je ime za tip podataka koji odgovara registru mmx (64-bita)
	__m64 num3, sum;
	__m64 *ptr1, *ptr2;

	sum = _mm_setzero_si64();    //inicijaliziraj sumu na 0

	for(int i=0; i<size; i+=4)          //vektori su tipa short
	{
		ptr1 = (__m64*)&a[i];        //dohvatimo 64-bita vektora a
		ptr2 = (__m64*)&b[i];        //i vektora b

		num3 = _mm_madd_pi16(*ptr1, *ptr2); //ovo je instrukcija multiply-add
		//ona racuna  l = a[i] * b[i] + a[i+1]*b[i+1]
		//            u = a[i+2] * b[i+2] + a[i+3]*b[i+3]
		//l se pohranjuje u donjih 32-bita registra num3
		//u se pohranjuje u gornja 32-bita registra num3

		sum = _mm_add_pi32(sum, num3);  //uvecaj gornju i donju parcijalnu sumu
		//registri sum i num3 sadrze 2 32-bitna broja
	}
	//krajnji rezultat dobivamo tako da zbrojimo
	p = _mm_cvtsi64_si32(sum);          //donjih 32 bita (donja parcijalna suma)
	sum = _mm_srli_si64(sum, 32);       //aritmeticki pomaknemo gornja 32 bita desno
	rez = _mm_cvtsi64_si32(sum);        //odredimo gornju parcijalnu sumu

	rez += p;                              //zbrojimo parcijalne sume

	_m_empty(); //isprazni FPU stog

	return rez;
}
```

Tablica 5: Primjeri instrukcija instrukcijskog skupa MMX

Intrinsična funkcija	**Asemblerska instrukcija**	**značenje**
`_mm_empty()`	`emms`	uklanjanje svih elemenata sa x87 FPU stoga
`__m64 _mm_add_pi32(__m64 m1, __m64 m2)`	`paddd`	`DEST[31..0] = m1[31..0] + m2[31..0];` `DEST[63..32] = m1[63..32] + m2[63..32];`

`__m64 _mm_madd_pi16(__m64 m1, __m64 m2)`	`pmaddwd`	`DEST[31..0] = m1[15..0] * m2[15..0] +`
		`m1[31..16] * m2[31..16];`
		`DEST[63..32] = m1[47..32] * m2[47..32] +`
		`m1[63..48] * m2[63..48];`

`int _mm_cvtsi64_pi32(__m64 m1)`	`movd`	`DEST = m1[31-0];`

Naglasimo da je poslije korištenja instrukcija iz skupa MMX nužno očistiti stanje na stogu x87 FPU jedinice (postaviti sve zastavice koje označavaju prisutnost elemenata na stogu na 0). To se

postiže navođenjem intrinsične funckije `_mm_empty()` (koja se prevodi u instrukciju `emms`) nakon svih MMX instrukcija.

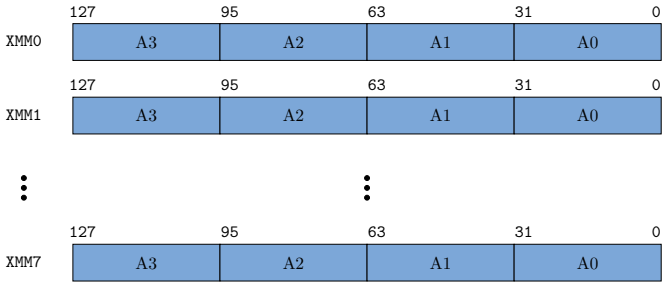
Možemo zaključiti da su glavni nedostaci MMX instrukcijskog skupa nemogućnost istovremenog korištenja FPU jedinice i MMX instrukcija, te nemogućnost rada s podacima s pomičnim zarezom kao elementima `XMM` registarskog skupa. To je i za očekivati s obzirom da je MMX instrukcijski skup bio prvi skup koji je uveo SIMD instrukcije u Intel arhitekturu procesora. Danas svi moderniji prevoditelji prilikom optimizacije koriste instrukcije iz novijih SIMD instrukcijskih skupova, a to su skupovi SSE i AVX. MMX instrukcije koriste samo starije arhitekture procesora, a u novijima su dostupne samo zbog kompatibilnosti sa starijim programima.

4.4 Instrukcijski skup SSE

SSE (eng. Streaming SIMD Extensions) je sljedeće instrukcijsko proširenje x86 instrukcijskog skupa. Uvedeno je 1999. godine i do sada je imalo nekoliko dodatnih proširenja SSE2, SSE3 i SSE4. SSE koristi 128-bitne registre bitne xmm0 - xmm7. Zadnja dva slova mnemonika instrukcije znače da li je instrukcija paralelna ili nije (Packed or Single), te preciznost instrukcije (eng. Precision). Preciznost označava veličinu podataka s kojima instukcija barata, a ona može biti: Single, Double, Word, Quadword or Binary.

XMM registar može sadržavati:

- 4 broja s pomičnim zarezom jednostruke preciznosti (`32bit, float`),
- 2 broja s pomičnim zarezom dvostruke preciznosti (`64bit, double`),
- 2 64-bitna cijela broja (`long ili long long`),
- 4 32-bitna cijela broja (`int`),
- 8 16-bitnih cijelih brojeva (`short int`), te
- 16 znakova (`char`).



Slika 5: **Registri instrukcijskog skupa SSE**: SSE instrukcijski skup ima 8 128-bitnih registara

Nekoliko najčešćih instrukcija iz skupa SSE prikazano je u tablici 6.

Primjer programa koji računa `saxpy` operaciju (eng. Single-precision real Alpha X Plus Y; $y \leftarrow \alpha x + y$) koristeći SSE intrinsične funkcije:

Tablica 6: Primjeri instrukcija instrukcijskog skupa SSE

Intrinsična funkcija	Asemblerska instrukcija	značenje
`__m128 _mm_cmpeq_ps(__m128 m1, __m128 m2)`	`cmpeqps`	`DEST[31..0] = (m1[31..0] == m2[31..0]) ? 0xffffffff : 0x00000000; ... DEST[127..96] = (m1[127..96] == m2[127..96]) ? 0xffffffff : 0x00000000;`
`__m128 _mm_max_ps(__m128 m1, __m128 m2)`	`maxps`	`DEST[31..0] = max(m1[31..0], m2[31..0]); ... DEST[127..96] = max(m1[127..96], m2[127..96]);`
`int _mm_cvtss_si32(__m128 m1)`	`cvtss2si`	`DEST[31..0] = (int)m1[31..0];`

```
void SaxpySSE128(float *x, float *y, float alpha, int size)
{
	__m128 xmm0 = _mm_set1_ps(alpha);   //ucitaj skalar alfa

	for (int i = 0; i < size; i+=4)
	{
		__m128 xmm1 = _mm_load_ps(x+i); //ucitaj 4 elementa iz x
		__m128 xmm2 = _mm_load_ps(y+i); //ucitaj 4 elementa iz y
		xmm1 = _mm_mul_ps(xmm1, xmm0);  //pomnozi x sa alfa
		xmm1 = _mm_add_ps(xmm1, xmm2);  //zbroji alfa*x sa y
		_mm_store_ps(y+i, xmm1);        //pohrani rezultat u y
	}
}
```

Ovo je primjer u kojem koristimo intrinsične funkcije. Kako bi te funkcije zapisali u asemblerskom kodu? (uzmite u obzir pri deklaraciji polja da ona moraju biti poravnata sa 32B, inače funkcija neće raditi).

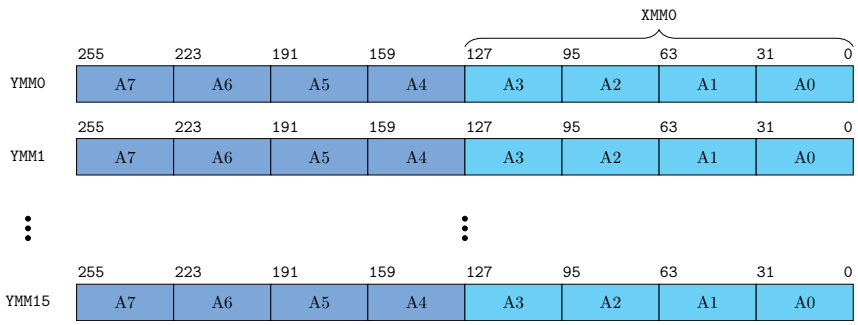
4.5 Instrukcijski skup AVX

Intel AVX (eng. Advanced Vector Extensions) instrukcijski skup sadrži instrukcije tipa SIMD i predstavlja proširenje instrukcijskih skupova MMX i SSE. Bitne značajke AVX instrukcijskog skupa su:

- uz nove instrukcije dodano je 16 novih 256 bitnih registra YMM0-YMM15 (ostavljena je mogućnost budućeg proširenja 512 i 1024 bitnim registrima).
- AVX instrukcije imaju mogućnost troadresnog adresiranja (SSE je omogućavao samo dvoadresno) tako da su sada moguće instrukcije koje izvode operacije poput `A = B + C` (operacije ne mijenjaju izvorne registre `B` i `C`).
- neke instrukcije imaju mogućnost četveroadresnog adresiranja (omogućuju složenije operacije koje smanjuju broj potrebnih instrukcija programa)
- unutar istog programskog koda mogu se nalaziti i starije instrukcije iz skupova SSE i MMX
- dodane su instrukcije koje mogu koristiti memorijske operande čije adrese nisu poravnate sa 32 bajta

AVX instrukcije omogućuju operacije nad podacima sa pomičnim zarezom jednostruke (4B) i dvostruke (8B) preciznosti. Cjelobrojne operacije nisu trenutno podržane, no biti će uvrštene u sljedeće instrukcijsko proširenje AVX2 (prvi procesor koji će ga podržavati je procesor sljedeće Intel arhitekture čije je trenutno kodno ime Haswell i planirano pojavljivanje 2014. godine).

AVX registri `YMM0 - YMM15` ostvareni su kao proširenja registara `XMM` (slika 6). To znači da će se postavljanjem registra `XMM` na zadanu vrijednost postaviti i donja polovica pripadnog registra `YMM` (vrijedi i obratno postavljanjem registra `YMM` postaviti će se i pripadni registar `XMM`).



Slika 6: **Registri instrukcijskog skupa AVX**: AVX instrukcijski skup koristi 16 256-bitnih registara koji su ostvareni kao proširenja registara `XMM`. Time je i broj `XMM` registara povećan na 16.

Nekoliko tipičnih instrukcija iz skupa AVX zajedno sa pripadnim intrinsičnim funkcijama prikazano je u tablici 7.

Tablica 7: Primjeri instrukcija instrukcijskog skupa AVX

Intrinsična funkcija	**Asemblerska instrukcija**	**značenje**

`__m256 _mm256_add_ps(__m256 m1, __m256 m2)`	`vadddps`	Elementi polja su tipa float.
		`DEST[31..0] = m1[31..0] + m2[31..0];`
		`DEST[63..32] = m1[63..32] + m2[63..32];`
		`...`
		`DEST[255..224] = m1[255..224] + m2[255..224];`

`__m256 _mm256_sqrt_ps(__m256 m1)`	`vsqrtps`	Elementi polja su tipa float.
		`DEST[31..0] = sqrt(m1[31..0]);`
		`DEST[63..32] = sqrt(m1[63..32]);`
		`...`
		`DEST[255..224] = sqrt(m1[255..224]);`

`__m256d _mm256_load_pd(const double *a)`	`vmovapd`	Elementi polja su tipa double.
		`DEST[63..0] = a[0];`
		`...`
		`DEST[255..192] = a[3];`

`void _mm256_store_ps(float *a, __m256 m1)`	`vmovaps`	Elementi polja su tipa float.
		`a[0] = m1[31..0]; a[1] = m1[63..32];`
		`...`
		`a[7] = m1[255..224];`

5 Prevođenje programa i mogući problemi (GCC prevodioc)

Vježbu se može napraviti korištenjem proizvoljne (32-bitne ili 64-bitne) distribucije Linuxa koja ima gcc, Unixa (npr. fakultetskom računalu Pinus), Mac OSXa ili Windowsa (sa instaliranim gcc prevodiocem **[MinGW]**). Također, ako radite pod Windowsima možete koristiti i virtualne linux strojeve (npr. **[VmWare]** i **[Ubuntu]**). Modifikacija asemblerskog koda koju je potrebno napraviti prilikom korištenja MSVC prevodioca dana je na početku uputa.

U ovom dodataku uputama pokazujemo nekoliko mogućih problema koji mogu nastati korištenjem gcc prevodioca.

U prilogu se nalaze datoteke main.cpp i p_asm.s. U njima se nalaze glavni program i potprogrami kako su opisani u uputi. Ove datoteke mogu se koristiti u a) zadatku, a mogu poslužiti i kao početak rješavanja ostalih zadataka. Program se prevodi i povezuje naredbom:

```
$ g++ -g -o main main p_asm.s main.cpp
```

Program pokrećemo:

```
$ ./main
```

Za navedene parametre funkcija, dobivamo sljedeći ispis:

```
ASM: 48
C++: 48
```

Ovaj primjer se ispravno prevodi na 32-bitnom operacijskom sustavu.

Ukoliko se prilikom prevođenja programa pojavi sljedeća greška:

```
p_asm.s: Assembler messages:
p_asm.s:10: Error: operand type mismatch for ‘push’
p_asm.s:28: Error: operand type mismatch for ‘pop’
```

Razlog je taj što se najvjerojatnije koristi 64-bitni operacijski sustav, a samim time i 64-bitni asembler, koji koristi drugačiju konvenciju (nije cdecl), te neispravno interpretira parametre funkcije. Također, 64-bitni asembler ima drugačiji instrukcijski set, 64-bitne registre i podaci na stogu su poravnati sa 16B.

Postoje 2 rješenja: Možemo prilikom prevođenja specificirati prevoditelju da želimo prevoditi 32-bitni kod:

To možemo učiniti dodavanjem opcije -m32 (po defaultu je 64 bita -m64)

```
$ g++ -g -m32 -o main p_asm.s main.cpp
```

No tada možda imamo sljedeći problem:

```
/usr/bin/ld: skipping incompatible /usr/lib/gcc/x86_64-linux-gnu/
4.4.5/libstdc++.a when searching for -lstdc++
/usr/bin/ld: cannot find -lstdc++
collect2: ld returned 1 exit status
```

Standardne library datoteke koje trenutno imamo su 64-bitne, no trebaju nam 32-bitne. Možemo ih instalirati ovako:

```
$ sudo apt-get install g++-multilib
```

Te zatim ponoviti:

```
$ g++ -g -m32 -o main p_asm.s main.cpp
```

i program će se uspješno prevesti.

Ili možemo preurediti asemblerski kod funkcije koristeći arhitekturu x86_64 (promjene su značajne jer koristimo 64-bitnu konvenciju (Microsoft x64 ili AMD64), te koristimo 64-bitne registre i instrukcije). Za naš primjer a) funkcija u asembleru izgledati će ovako:

```
/* oznaka sintakse: */
.intel_syntax noprefix

/* neka simbol potprogram\_asm bude vidljiv izvana: */
.global potprogram_asm

/*labela potprograma:*/
potprogram_asm:    /*AMD64 ABI konvencija*/
                   /* u ovom slucaju prolog nam nije potreban
                         jer ne koristimo lokalne varijable */
  mov  rax, rdi    /*rdi = a*/
  add  rax, rsi    /*rsi = b*/
  imul rax, rdx    /*rdx = c*/

                   /*rezultat je u rax*/
  ret              /* povratak iz potprograma */
```

Naglasak ove vježbe je na x86 arhitekturi, stoga nije nužno proučavati x64 konvencije. Ipak, ako nekoga zanima može pogledati sljedeće radove/stranice:

[1] Introduction to x64 Assembly
[2] x86 calling conventions
[3] The history of calling conventions, part 5: amd64
[4] The reasons why 64-bit programs require more stack memory

Postoje 2 konvencije (Microsoft x64 calling convention (za Windowse) i System V AMD64 ABI convention (za Linux, BSD, Mac)). Konvencije su međusobno vrlo slične. Treba obratiti pozornost na način prenošenja parametara u svakoj on njih: prva prenosi 4 parametra preko registara rcx/xmm0, rdx/xmm1, r8/xmm2, r9/xmm3, dok ih druga prenosi do 6 preko rdi, rsi, rdx, rcx, r8, r9, xmm0-7. Ostali parametri prenose se preko stoga.

Napomena: Ako pokušamo program prevesti na Mac OSX operacijskom sustavu (koji je također 64-bitan) dobiti ćemo sljedeće:

```
$ g++ -m32 -g -o main p_asm.s main.cpp
p_asm.s:5:Unknown pseudo-op: .global
p_asm.s:5:Rest of line ignored. 1st junk character valued 112 (p).
```

Program se neće uspješno prevesti jer prevoditelj ne prepoznaje operaciju .global, te kao i u Windowsima labela potprogram dobiva podvlaku kao prefix. Potrebno je napraviti sljedeće izmjene:

```
.global potprogram_asm => .globl _potprogram_asm
potprogram_asm: => _potprogram_asm
```