Mikroprocesorski sistemi

Aleksandar B. Samardžić
(asamardzic@matf.bg.ac.yu)

Matematički fakultet
Univerzitet u Beogradu

Verzija 1.2.0, mart 2004.

Copyright © 2002, 2003, 2004 Aleksandar B. Samardžić.

Dokument je dozvoljeno kopirati, distribuirati i/ili modifikovati prema uslovima navedenim u *GNU Free Documentation License* licenci u verziji 1.1 ili bilo kojoj kasnijoj verziji objavljenoj od strane *Free Software Foundation*-a. Kopija licence se može naći na adresi http://www.gnu.org/licenses/licenses.html#TOCFDL.

Predgovor

Ovaj tekst je primarno namijenjen studentima koji pohađaju časove vježbi iz predmeta "Mikroračunari" na IV godini R smjera Matematičkog fakulteta Univerziteta u Beogradu. Predmet se prvenstveno bavi proučavanjem mikroprocesorskih sistema pa je toj temi posvećen i ovaj tekst. Prateći materijali za ovaj kurs, a prije svega kompletan programski kod koji je u ovom tekstu prezentiran, mogu se naći na *Web* strani predmeta ([1]).

Razmatranju mikroprocesorskih (i generalno računarskih) sistema se može pristupiti sa dva aspekta: sa aspekta arhitekture ili sa aspekta organizacije tih sistema. U prvom slučaju proučava se programerski interfejs datog mikroprocesorskog sistema odn. prije svega skup instrukcija, a zatim i registar-fajlovi, režimi adresiranja, konvencije koje treba slijediti u cilju optimizacije programa itd. Arhitektura pojedinih familija mikroprocesorskih sistema je obično vrlo specifična; npr. arhitektura *Intel IA-32* familije procesora je potpuno različita od arhitekture *Sun SPARC* familije. Organizacija mikroprocesorskih sistema se sa druge strane odnosi na način na koji su implementirane pojedine komponente tog sistema, npr. aritmetičko-logička (ALU) ili kontrolna jedinica, odnosno i sistem kao cjelina. Za razliku od arhitekture, organizacija većine mikroprocesorskih sistema je dosta slična, odn. obično postoje samo manje varijacije u načinima na koje su pojedine komponente implementirane.

Imajući u vidu primarnu grupu čitalaca kojoj je ovaj tekst namijenjen, orgranizaciji računara je posvećen manji, prvi dio teksta, čiji je cilj upoznavanje sa elementarnim digitalnim kolima koja se koriste kao osnovni elementi pri dizajnu mikroprocesorskih sistema. Drugi, veći dio teksta, je posvećen razmatranju jedne konkretne mikroprocesorske arhitekture, pri čemu je logičan izbor za razmatranje, kao trenutno ubjedljivo najzastupljenija, arhitektura *Intel*-ove *IA-32* familije procesora. U dodatku je u kratkim crtama predstavljena i *Intel*-ova *IA-64* arhitektura, od koje se očekuje da u narednim godina zamijeni *IA-32* arhitekturu.

Nastojalo se da pristup razmatranju organizacije, odn. arhitekture računara u ovom tekstu bude praktičan, tako da tekst prati veliki broj primjera. Primjeri koji se tiču organizacije računara odnose se na implementaciju el-

ementarnih sklopova na *Verilog HDL*[1] jeziku, dok se primjeri koji se tiču arhitekture računara odnose na implementaciju raznih algoritama na *IA-32* asembleru. U prvom slučaju, za prevođenje primjera treba koristiti *Icarus Verilog* prevodilac, a u drugom slučaju *GNU as* asembler. U vezi sa ovim treba pomenuti da, obzirom da je *Verilog* standardizovan jezik, odgovarajući primjeri treba bez problema da budu prevodivi i sa drugim prevodiocima; sa druge strane, sintaksa asemblera nije standardizovana, tako da su primjeri koji se odnose na *IA-32* asembler generalno prevodivi samo *GNU as* asemblerom. Dalje, u oba slučaja je pretpostavka da se sa alatima radi na *UNIX* platformi, tako da je preduslov za praćenje ovog teksta poznavanje rada na *UNIX*-u ([2]), a pogotovo poznavanje *GNU* programerskih alata ([3]). Takođe, pretpostavlja se i dobro poznavanje *C* jezika.

Autor želi da se zahvali recenzentima, prof. Milu Tomaševiću i Vladimiru Filipoviću, na pažljivom čitanju teksta i nizu korisnih sugestija. Takođe, autor se posebno zahvaljuje svim studentima R smjera Matematičkog fakulteta koji su tokom čitanja prijavljivali greške uočene u rukopisu.

[1] *hardware description language*

Sadržaj

Predgovor **i**

I Osnove organizacije mikroprocesorkih sistema **1**

1 Osnove logičkog dizajna **3**

1.1 Logičke primitive 3
1.2 Sinteza logičkih funkcija 4
1.3 Minimizacija logičkih funkcija 4
1.4 Izbor logičke baze 7
1.5 Primjer logičkog dizajna 8

2 Osnove *Verilog HDL* jezika **13**

2.1 Koncept modula 14
2.2 Testiranje funkcionalnosti modula 16
2.3 Grafički prikaz rezultata simulacije 20
2.4 Osnovna *Verilog* sintaksa 21

3 Elementarne logičke komponente **25**

3.1 Sabirači 25
3.2 Flip-flopovi 31
3.3 Registri 39
3.4 Brojači 42
3.5 Dekoderi i multiplekseri 44
3.6 Množači 45
3.7 *Data flow* modelovanje 48

4 Dizajn kompletnog sistema 51

4.1 Arhitektura sistema . . . 51

4.2 Organizacija sistema . . . 53

4.3 Model aritmetičko-logičke jedinice . . . 54

4.4 Model skupa registara opšte namjene . . . 58

4.5 Model *datapath* komponente . . . 59

4.6 Model kontrolne jedinice . . . 65

4.7 Model kompletnog mikroprocesora . . . 71

4.8 Simulacija dizajna . . . 72

4.9 Sinteza dizajna . . . 75

4.10 Napredne konstrukcije u *Verilog*-u . . . 76

II *IA-32* arhitektura 79

5 Uvod u *IA-32* arhitekturu 81

5.1 Skup registara . . . 82

5.2 Adresiranje operanada . . . 85

5.3 Tipovi podataka . . . 85

5.4 Skup instrukcija . . . 87

5.5 *GNU* asembler . . . 88

6 Instrukcije opšte namjene 95

6.1 Aritmetičko-logičke instrukcije . . . 95

6.2 Instrukcije za transfer podataka . . . 100

6.3 Instrukcije kontrole toka . . . 103

6.4 Bitske instrukcije . . . 113

6.5 Instrukcije za rad sa stringovima . . . 119

6.6 Instrukcije za rad sa flegovima . . . 122

6.7 Preostale instrukcije opšte namjene . . . 123

7 Potprogrami na *IA-32* arhitekturi **129**
7.1 Pozivanje potprograma 129
7.2 Prenošenje argumenata u potprograme 130
7.3 Vraćanje rezultata iz potprograma 131
7.4 Lokalne promjenljive u potprogramu 134
7.5 Standardni prolog i epilog potprograma 136
7.6 Čuvanje sadržaja registara u potprogramu 137
7.7 Diskusija opisanog načina pozivanja potprograma 137
7.8 Modularna organizacija programa 138

8 Povezivanje asemblerskog i *C* koda **141**
8.1 *C* konvencije za pozivanje potprograma 141
8.2 Sistemski pozivi iz asemblerskog koda 143
8.3 *Inline* asemblersko programiranje 145
8.4 *C* konvencije za smještanje podataka 147
8.5 Format i pokretanje izvršnih fajlova 147

9 Instrukcije za rad sa realnim brojevima **153**
9.1 Realni tipovi podataka 153
9.2 Numerički koprocesor 155
9.3 Instrukcije za transfer podataka 156
9.4 Instrukcije za učitavanje konstanti 158
9.5 Aritmetičke instrukcije 158
9.6 Instrukcije poređenja 160
9.7 Trigonometrijske instrukcije 164
9.8 Logaritamske i eksponencijalne funkcije 164
9.9 Kontrolne instrukcije 167
9.10 Mehanizam obrade grešaka koprocesora 168

10 Privilegovane instrukcije **171**
10.1 Mehanizam prekida 171
10.2 Ulazno-izlazne instrukcije 172

11 *MMX* instrukcije **177**

11.1 Skup registara za *MMX* instrukcije 177
11.2 Tretman prekoračenja kod *MMX* instrukcija 178
11.3 Instrukcije za transfer podataka 179
11.4 Aritmetičke instrukcije 179
11.5 Instrukcije poređenja 180
11.6 Instrukcije konverzije podataka 180
11.7 Logičke instrukcije 181
11.8 Instrukcije šiftovanja 182
11.9 Upotreba *MMX* instrukcija 182

12 *SSE* instrukcije **189**

12.1 Skup registara za *SSE* instrukcije 190
12.2 Instrukcije za transfer podataka 191
12.3 Aritmetičko-logičke instrukcije 192
12.4 Instrukcije poređenja 193
12.5 Instrukcije konverzije podataka 194
12.6 Ostale instrukcije uvedene *SSE* ekstenzijama 196
12.7 Upotreba *SSE* instrukcija 199

13 *SSE2* instrukcije **205**

13.1 Instrukcije sa rad sa realnim brojevima 205
13.2 Instrukcije sa rad sa cijelim brojevima 209
13.3 Ostale *SSE2* instrukcije 210
13.4 Upotreba *SSE2* instrukcija 210

A *IA-64* arhitektura **211**

A.1 Skup registara arhitekture 213
A.2 Format i način izvršavanja instrukcija 213
A.3 Skup instrukcija 214
A.4 Elementi arhitekture dizajnirani u cilju unapređenja performansi 215
A.5 Primjer programiranja u *IA-64* asembleru 216

Dio I

Osnove organizacije mikroprocesorkih sistema

Poglavlje 1

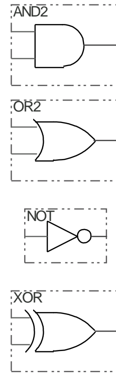
Osnove logičkog dizajna

1.1 Logičke primitive

Ključni postulat na kome se bazira dizajn i funkcionisanje mikroprocesorskih sistema jeste mogućnost svođenja svih operacija koje se inače na njima izvršavaju (npr. aritmetičkih, operacija sa stringovima i sl.) na logičke operacije. Na taj način, osnovni element na kome se bazira izgradnja ovih sistema jesu logičke primitive, odn. komponente koje implementiraju osnovne logičke operacije. Prema tome, da bi bilo moguće konstruisati neki digitalni sklop[1] neophodno je postojanje nekog fizičkog fenomena koji implementira logičke operacije. U prvo vrijeme, za tu svrhu su korišćeni elektro-mehanički elementi, a danas se koriste elektronski elementi (odn. preciznije tranzistori).

Digitalni sklopovi se obično sastoje od većeg broja logičkih primitiva (alternativno se koristi i izvorni termin *logic gates*), koje su međusobno povezane na odgovarajući način tj. digitalni sklopovi praktično predstavljaju logičke funkcije u smislu kombinacije logičkih primitiva. Jedan tip reprezentacije digitalnih sklopova jeste šematska reprezentacija, na kojoj se logičke primitive predstavljaju odgovarajućim standardnim simbolima, dok se konekcije među njima predstavljaju linijama. Standardni simboli za osnovne logičke primitive dati su na slici 1.1. Treba pomenuti i da se za negaciju koristi i nešto kompaktnija notacija kada se invertuje neki od ulaza ili izlaz logičke primitive - u tom slučaju negacija se predstavlja samo kružićem na ulazu odn. izlazu iz primitive.

[1] u smislu mikroprocesorskog sistema u cjelini ili pojedinačnih komponenti od kojih se ovaj sistem sastoji



Slika 1.1: Standardni simboli za elementarne logičke primitive.

1.2 Sinteza logičkih funkcija

Svaka logička primitiva, kao i svaka logička funkcija, je jednoznačno određena tablicom istinitosti, odn. tablicom vrijednosti izlaza za sve moguće kombinacije ulaza. Treba uočiti da nisu sve logičke primitive međusobno nezavisne - tako se npr. XOR primitiva može izraziti preko AND, OR i NOT primitiva na sledeći način[2]:

$$x_0 \oplus x_1 = \overline{x}_0 \cdot x_1 + x_0 \cdot \overline{x}_1$$

Tačnost gornjeg izraza se može naravno utvrditi konstruisanjem tablica istinitosti za lijevu i desnu stranu. Funkcija na desnoj strani ima formu koja je u matematičkoj logici poznata pod nazivom "suma proizvoda"; matematička logika takođe pokazuje da se svaka tablica istinitosti može direktno prevesti u ovakvu formu odnosno, obzirom da svakoj logičkoj funkciji odgovara tablica istinitosti, da se i svaka logička funkcija može predstaviti korišćenjem samo AND, OR i NOT primitiva. Skup primitiva preko koga se mogu izraziti sve druge primitive se naziva logičkom bazom, te u tom smislu skup primitiva AND, OR i NOT predstavlja logičku bazu. Posledica svega ovoga na dizajn digitalnih kola jeste da je onda dovoljno da gorepomenuti prirodni fenomen koji implementira logičke operacije implementira samo one operacije koje sačinjavaju bazu (tj. nije potrebno da implementira sve logičke operacije).

1.3 Minimizacija logičkih funkcija

Rezultat matematičke logike takođe je da logičke funkcije nisu jedinstvene tj. da je za datu funkciju uvijek moguće konstruisati neku drugu ekviva-

[2] u tekstu su korišćene standardne oznake za logičke operacije: $\cdot$ (ili bez oznake) za AND, $+$ za OR, nadvlaka za NOT i $\oplus$ za XOR; redosled prioriteta operatora je takođe standardan i NOT ima najveći prioritet, pa onda dolazi AND i na kraju OR

lentnu (u smislu da ima identičnu istinitosnu tablicu) funkciju. Tako se npr. poređenjem istinitosnih tablica može utvrditi da je funkciji $\overline{x}_0\overline{x}_1\overline{x}_2 + \overline{x}_0\overline{x}_1x_2 + \overline{x}_0x_1x_2 + x_0x_1x_2$ ekvivalentna funkcija $\overline{x}_0\overline{x}_1 + x_1x_2$. Sa aspekta logičkog dizajna ove funkcije međutim nisu ekvivalente, obzirom da je svakoj funkciji kada se implementira u vidu logičkog kola pridružena odgovarajuća cijena u pogledu faktora koji se datim dizajnom nastoje optimizovati. Sa tekućom, tranzistorskom tehnologijom, svakoj logičkoj primitivi je pridružena određena cijena u pogledu prostora koga primitiva zauzima kao i u pogledu kašnjenja pri generisanju izlaza nakon promjene ulaza. Oba ova faktora se nastoje minimizovati, ali uticaj oblika logičke funkcije na njih se ne može predstaviti nekom egzaktnom formulom[3]. Okvirno, može se reći da su u tranzistorskoj tehnologiji i površina koju zauzima neko logičko kolo i kašnjenje na tom kolu utoliko veći ukoliko je veći broj logičkih primitiva odn. ukoliko je broj ulaza u svaku primitivu veći. Tako npr. ako ugrubo pretpostavimo da je cijena koju unosi svaka logička primitiva jednaka 1, kao i da je cijena koju unosi neki ulaz u svaku logičku primitivu takođe jednaka 1, mogla bi se prva od dvije navedene ekvivalentne logičke funkcije okarakterisati cijenom 21, a druga funkcija cijenom 9. Na osnovu ovog primjera se može naslutiti koliko je za logički dizajn (bar u tranzistorskoj tehnologiji) važno pronaći minimalnu funkciju u skupu ekvivalentnih logičkih funkcija u smislu najmanjeg broja logičkih primitiva potrebnih za njenu implementaciju kao i najmanjeg broja ulaza u svaku logičku primitivu ponaosob.

Generalno, postoje dva metoda minimizacije logičkih izraza: algebarske transformacije i minimizacija preko tzv. *Karnaugh*-ovih mapa. U prvom slučaju se koriste logička pravila da bi se transformisao dati izraz u minimalnu formu. Jedan skup logičkih pravila (odn. preciznije algebarskih identiteta) koji su korisni za ovakve transformacije dat je tabelom 1.1.

Naziv	Algebarski identiteti
pravilo komutativnosti	$x_0 + x_1 = x_1 + x_0$, $x_0x_1 = x_1x_0$
pravilo asocijativnosti	$(x_0 + x_1) + x_2 = x_0 + (x_1 + x_2)$, $(x_0x_1)x_2 = x_0(x_1x_2)$
pravilo distributivnosti	$x_0 + x_1x_2 = (x_0 + x_1)(x_0 + x_2)$, $x_0(x_1 + x_2) = x_0x_1 + x_0x_2$
pravilo idempotencije	$x + x = x$, $xx = x$
pravilo involucije	$\overline{\overline{x}} = x$
pravilo komplementa	$x + \overline{x} = 1$, $x\overline{x} = 0$
De Morgran-ova pravila	$\overline{x_0 + x_1} = \overline{x}_0\overline{x}_1$, $\overline{x_0x_1} = \overline{x}_0 + \overline{x}_1$
	$1 + x = 1$, $0 \cdot x = 0$
	$0 + x = x$, $1 \cdot x = x$

Tabela 1.1: Osnovna logička pravila.

Tako se npr. korišćenjem redom pravila distributivnosti, komplementa i prav-

[3]utoliko prije što pomenuti faktori nisu ortogonalni, tj. postoje određene zavisnosti među njima

ila o neutralnom elementu može izvršiti svođenje između dvije ekvivalentne forme ranije pomenute funkcije:

$$\begin{aligned} & \overline{x}_0\overline{x}_1\overline{x}_2 + \overline{x}_0\overline{x}_1 x_2 + \overline{x}_0 x_1 x_2 + x_0 x_1 x_2 = \\ = \; & \overline{x}_0\overline{x}_1(x_2 + \overline{x}_2) + (x_0 + \overline{x}_0) x_1 x_2 = \\ = \; & \overline{x}_0\overline{x}_1 \cdot 1 + 1 \cdot x_1 x_2 = \\ = \; & \overline{x}_0\overline{x}_1 + x_1 x_2 \end{aligned}$$

Međutim, u opštem slučaju teško je odrediti pravi slijed transformacija koji svodi datu funkciju na minimalnu formu, te se stoga drugi metod, *Karnaugh*-ove mape, daleko češće koristi za rješavanje ovog problema. *Karnaugh*-ova mapa predstavlja običnu tablicu vrijednosti funkcije preuređenu tako da su varijacije vrijednosti nekih ulaza date duž redova tablice, a ostalih ulaza duž kolona tablice. Pritom, varijacije se navode takvim redom da se u dva susjedna reda odn. kolone mijenja vrijednost samo jednog ulaza. Tako bi npr. *Karnaugh*-ova mapa za pominjanu logičku funkciju $\overline{x}_0\overline{x}_1\overline{x}_2 + \overline{x}_0\overline{x}_1 x_2 + \overline{x}_0 x_1 x_2 + x_0 x_1 x_2$ mogla biti oblika:

$x_0 \downarrow \quad x_1x_2 \rightarrow$	00	01	11	10
0	1	1	1	0
1	0	0	1	0

Raspored promjenljivih po redovima odn. kolonama mape je proizvoljan, takođe je mapa mogla biti i oblika 4×2 umjesto 2×4. Ono što je bitno jeste da se pomenutim rasporedom, gdje se varijacije po redovima odn. kolonama mape razlikuju samo po vrijednosti jedne promjenljive, postiže da se algebarska pojednostavljenja uočavaju kao parovi jedinica u susjednim ćelijama. Pritom treba imati u vidu da se susjednim smatraju i poslednja i prva ćelija u datom redu odn. poslednja i prva ćelija u datoj koloni (obzirom da i za njih važi da se varijacije razlikuju samo po vrijednosti jedne promjenljive). Tako npr. prva i druga ćelija u prvom redu mape imaju vrijednost 1 što ukazuje na moguće pojednostavljenje parcijalne sume u logičkoj funkciji predstavljenoj tabelom koja odgovara ovim ćelijama - ova parcijalna suma iznosi $\overline{x}_0\overline{x}_1\overline{x}_2 + \overline{x}_0\overline{x}_1 x_2$, a sa mape se vidi da promjenljiva x_2 ne utiče na njenu vrijednost (obzirom da je za bilo koju vrijednost promjenljive x_2 ta suma jednaka 1); na taj način, ova parcijalna suma se sa tabele može odmah očitati kao $\overline{x}_0\overline{x}_1$. Na sličan način se sa tabele može druga parcijalna suma date logičke funkcije koja iznosi $\overline{x}_0 x_1 x_2 + x_0 x_1 x_2$ direktno očitati kao $x_1 x_2$. Pored pojave dvije susjedne jedinice u vrstama odn. kolonama mape (tj. pojave susjednih jedinica u konfiguracijama 2×1 odn. 1×2) moguće su i pojave susjednih jedinica (stalno treba voditi računa i o susjednosti poslednje i prve ćelije u redu odn. koloni) u konfiguracijama 2×2, 4×2 ili 2×4, kao i generalno u konfiguraciji $2^m \times 2^n$ ($m, n \geq 1$ i $mn \geq 2$) za logičke funkcije sa većim brojem promjenljivih. Takvim slučajevima odgovara još

prostiji oblik parcijalne sume, obzirom da ista ima konstantnu vrijednost za veći broj promjenljivih. Na taj način, može se reći da se minimalna forma date logičke funkcije iz *Karnaugh*-ove mape dobija tako što se sve jedinice na mapi grupišu u što manji broj pravougaonika (čije su dimenzije stepeni broja 2) što veće površine. Ovim drugim se postiže da logičke primitive u implementaciji dobijene funkcije sadrže minimalan broj ulaza, a i prvim i drugim da broj tih primitiva bude minimalan, odn. da se time praktično minimizuje cijena odgovarajućeg digitalnog sklopa u pogledu navedenih kriterijuma.

Jedna mogućnost vezana za minimizaciju logičkih funkcija jeste da postoje neke varijacije ulaznih promjenljivih koje se ne mogu pojaviti na ulazu. Tako npr. možemo zamisliti digitalno kolo koje provjerava da li je data decimalna cifra djeljiva sa 3. Ovo kolo bi imalo jedan izlaz, dok bi cifra na ulazu morala biti predstavljena sa 4 bita, pa bi onda kolo imalo 4 ulaza. Međutim, od 16 mogućih varijacija na ulazu, samo 10 je validno, dok se ostalih 6 ne mogu pojaviti; takvim varijacijama se zato mogu dodijeliti proizvoljne vrijednosti i onda te vrijednosti biramo tako da dalje minimizujemo logičku funkciju koju tabela predstavlja. Za pomenuti primjer bi *Karnaugh*-ova mapa bila oblika (ćelije koje odgovaraju varijacijama koje se ne mogu pojaviti na ulazu su označene sa d od *don't care* kako se inače obično označavaju takve varijacije):

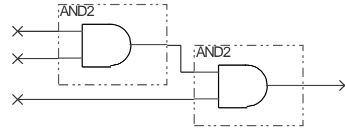
$x_3x_2 \downarrow$ $x_1x_0 \rightarrow$	00	01	11	10
00	0	0	1	0
01	0	0	0	1
11	d	d	d	d
10	0	1	d	d

U datom slučaju je zgodno za *don't care* varijacije 1100 i 1010 uzeti vrijednost 0, a za preostale *don't care* varijacije vrijednost 1, pa je onda minimizovana logička funkcija oblika $\overline{x}_2x_1x_0 + x_2x_1\overline{x}_0 + x_3x_0$.

1.4 Izbor logičke baze

U prethodnom dijelu teksta je pokazano kako skup logičkih primitiva *AND*, *OR* i *NOT* sačinjava logičku bazu. Ovo međutim nije minimalna baza u smislu najmanjeg broja primitiva koje sačinjavaju bazu - matematička logika pokazuje da minimalne baze sačinjavaju *NAND* ili *NOR* primitiva. Ove logičke baze su zgodne i zbog toga što je implementacija odgovarajućih primitiva posebno jednostavna u tranzistorskoj tehnologiji. Zbog svega toga se jedna ili druga primitiva najčešće koriste u sintezi digitalnih kola. Jedan nedostatak ovih primitiva u vezi sa tranzistorskom tehnologijom je međutim što nisu asocijativne; tranzistorska tehnologija postavlja ograničenja na broj ulaza u neku primitivu (ovaj ograničenje u pogledu broja ulaza se označava terminom *fan-in*) i to ograničenje u većini podvarijacija tranzistorske

tehnologije obično iznosi 5 ili 6. Ako su logičke primitive koje se koriste za sintezu *AND* ili *OR* onda je ovo ograničenje, ozbirom da su primitive asocijativne, jednostavno prevazići (doduše, uz uvođenje dodatnog kašnjenja) pomoću kaskade kao na slici 1.2. Ako su logičke primitive koje se koriste za sintezu *NAND* ili *NOR*, onda pravljenje ovakve jednostavne kaskade nije moguće, već je ekvivalentno kolo prilično komplikovanije.

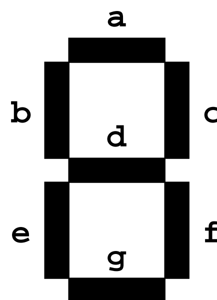


Slika 1.2: Implementacija troulazne *AND* primitive kada su na raspolaganju samo dvoulazne primitive.

Minimizacija logičkih funkcija i njihovo izražavanje u najpogodnijoj bazi su zadatak koji danas obično na sebe preuzimaju odgovarajući alati. Za programera je važno da pravilno predstavi, npr. pomoću odgovarajuće šeme, logičku funkciju koju dati digitalni sklop treba da obavlja, a onda alat u kome se vrši unos obavlja dalje optimizacije. Ipak, problematiku optimizacije je dobro poznavati radi kompletnosti razumijevanja čitavog procesa logičkog dizajna.

1.5 Primjer logičkog dizajna

Kao primjer dizajna nekog digitalnog sklopa biće razmotrena konstrukcija kontrolera sedmosegmentnog indikatora. Sedmosegmentni indikator predstavlja displej za prikaz decimalnih cifara koji se sastoji od 7 svijetala koja će biti označena kao na slici 1.3. Uključivanjem odgovarajućih segmenata mogu se predstaviti sve cifre od 0 do 9.



Slika 1.3: Sedmosegmentni indikator.

Kontroler sedmosegmentnog indikatora je digitalni sklop koji na ulazu prima cifru koju treba prikazati na indikatoru, a na izlazu generiše signale koji

kontrolišu pojedine segmente indikatora. Za predstavljanje decimalne cifre potrebna su 4 bita, tako da će signali x_3, x_2, x_1 i x_0 na ulazima kontrolera odgovarati ovim bitovima; signali na izlazu se mogu označiti sa a, b, c, d, e, f i g prema segmentima koje kontrolišu. Nakon toga treba formirati *Karnaugh*-ove mape za svaki izlaz ponaosob i pomoću njih odrediti logičku funkciju koja odgovara izlazu. Tako bi npr. za izlaz a *Karnaugh*-ova mapa bila oblika:

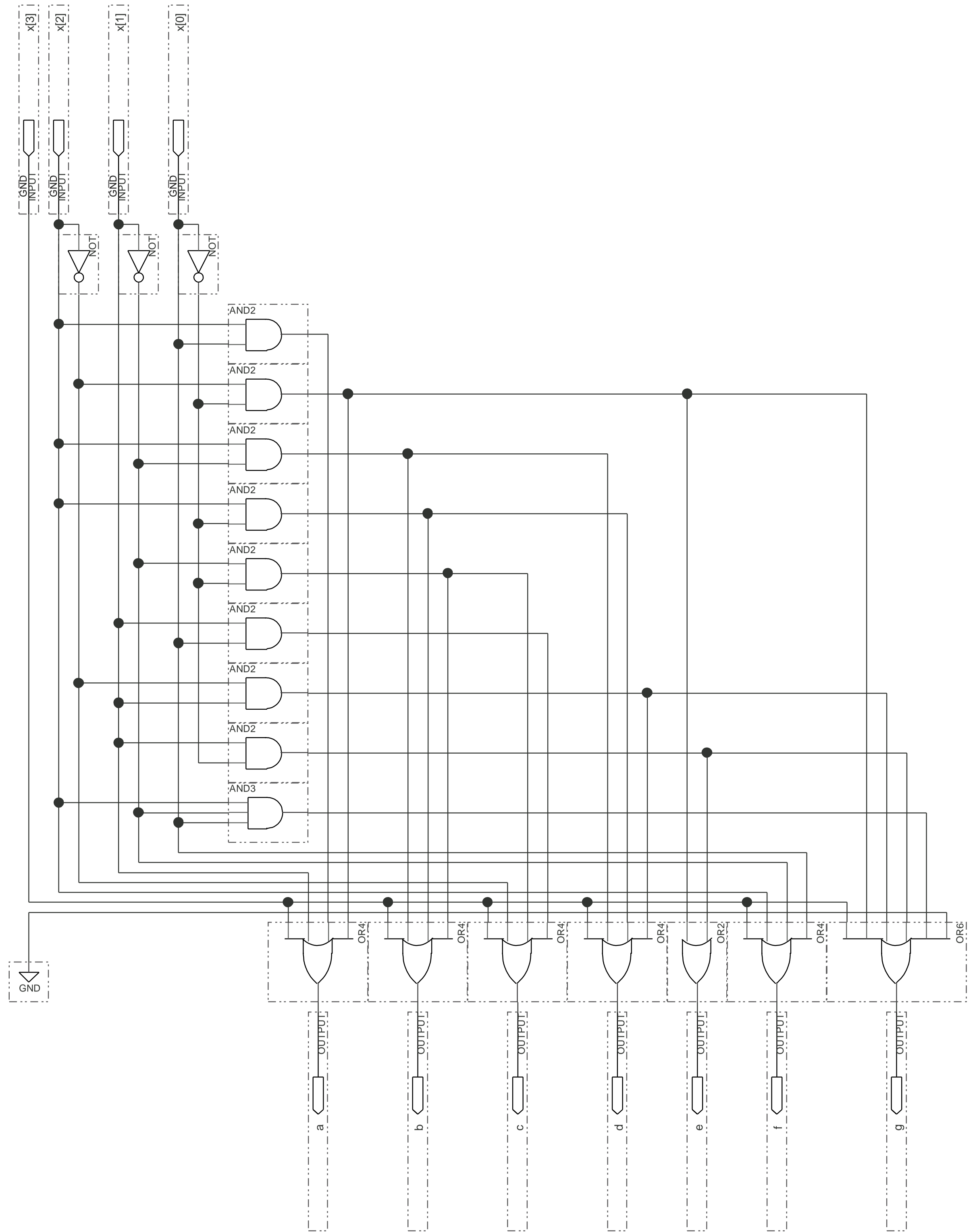
$x_3x_2 \downarrow \quad x_1x_0 \rightarrow$	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	d	d	d	d
10	1	1	d	d

odakle se dobija da je logička funkcija koja odgovara ovom izlazu $a = x_3 + x_1 + x_2x_0 + \overline{x}_2\overline{x}_0$. Na sličan način se određuju logičke funkcije koje odgovaraju ostalim izlazima:

$$
\begin{aligned}
b &= x_3 + x_2\overline{x}_1 + x_2\overline{x}_0 + \overline{x}_1\overline{x}_0 \\
c &= x_3 + \overline{x}_2 + x_1x_0 + \overline{x}_1\overline{x}_0 \\
d &= x_3 + x_2\overline{x}_1 + \overline{x}_2x_1 + x_1\overline{x}_0 \\
e &= \overline{x}_2\overline{x}_0 + x_1\overline{x}_0 \\
f &= x_3 + x_2 + \overline{x}_1 + x_0 \\
g &= x_3 + \overline{x}_2x_1 + \overline{x}_2\overline{x}_0 + x_1x_0 + x_2\overline{x}_2x_0
\end{aligned}
$$

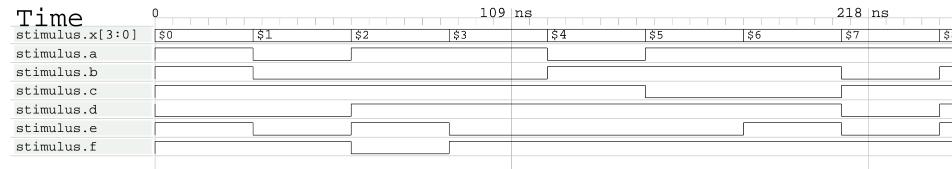
Na osnovu ovoga, može se napraviti digitalni sklop koji predstavlja kontroler i koji će biti oblika kao na slici 1.4. Treba uočiti kako u korišćenom programu za šematski unos nije podržana *OR* primitiva sa 5 ulaza pa je korišćena *OR* primitiva sa 6 ulaza čiji je jedan ulaz fiksiran na 0. Program u pitanju je inače *Quartus II* okruženje za digitalno projektovanje firme *Altera* koje uključuje editor za šematski unos, simulator i razne druge module; ovo okruženje je korišćeno za unos svih šema u ovom tekstu.

Po konstrukciji ovakvog digitalnog sklopa potrebno je provjeriti da li isti radi kako je zamišljeno. Program za šematski unos obično je praćen programom za simulaciju u kome se mogu na ulazu datog kola zadati neke varijacije signala da bi simulator potom izračunao signale na izlazu. Verifikacija datog sklopa se sprovodi poređenjem dobijenih izlaznih signala sa očekivanim vrijednostima. U posmatranom slučaju, verifikaciju je lako izvršiti zato što je broj varijacija ulaznih signala mali. Rezultati simulacije su prikazani na slici 1.5, varijacije ulaznog signala su date u prvom redu, a ostali redovi predstavljaju izlazne signale. Pregledom slike se može uvjeriti da se zaista za svaku cifru na izlazu uključuju odgovarajući segmenti indikatora. Treba uočiti kako je simulator (opet odgovarajući program iz paketa *Quartus II*)



Slika 1.4: Šema kontrolera sedmosegmentnog indikatora.

uračunao određena kašnjenja u propagaciji signala od ulaza prema izlazima; ova kašnjenja se odnose na određenu tranzistorsku tehnologiju za koju je simulator pretpostavio da će biti korišćena za implementaciju digitalnog sklopa i u konkretnom slučaju kašnjenja su između 7 i 9ns.

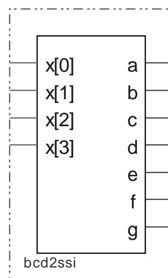


Slika 1.5: Rezulati simulacije kontrolera sedmosegmentnog indikatora.

Po uspješnoj verifikaciji, preostaje da se napravi integrisano pakovanje, tzv. čip koji implementira projektovani digitalni sklop. Čipove su u prvo vrijeme pravile po narudžbini specijalizovane fabrike kojima je dostavljan šematski ili neki drugi opis digitalnog sklopa. U poslednje vrijeme međutim za manje serije i u slučajevima kada potpuna optimizacija nije neophodna preovladava tzv. *FPGA*[4] metoda koja omogućava krajnjem korisniku da sam programira predizajnirane čipove. Tako se uz integrisana okruženja kakvo je *Quartus II* dobija i programator koji se priključuje na računar i koji omogućava programiranje određenog seta "praznih" čipova koji se mogu naručivati od proizvođača okruženja. Više o ovoj tehnologiji može se naći u [4].

U konkretnom slučaju kontrolera za sedmosegmentni indikator, logički sklop je suviše mali da bi zavređivao da bude programiran na posebnom čipu. Umjesto toga, sklop po verifikaciji može da se koristi kao primitiva (ravnopravno sa ostalim primitivama) za izgradnju kompleksnijih procesora, npr. kompletnog kalkulatora. Programi za šematski unos obično omogućavaju da se jednom dizajniranim sklopovima dodijeli grafički simbol koji onda može da se koristi ravnopravno sa ugrađenim simbolima za nove šeme. Simbol za takve sklopove je obično pravougaonik sa imenom sklopa i sa naznačenim ulazima i izlazima odnosno tzv. ulaznim i izlaznim pinovima. Tako bi simbol za dati kontroler mogao biti oblika kao na slici 1.6.

[4]skraćenica od *field programmable gate arrays*



Slika 1.6: Simbol za kontroler sedmosegmentnog indikatora.

Poglavlje 2

Osnove *Verilog HDL* jezika

Kako je vidljivo na primjeru iz prethodnog poglavlja, šematski način reprezentacije digitalnih sklopova, mada je veoma intuitivan, ima određenih nedostataka: potrebno je puno vremena da se unesu i rasporede sve primitive, za iole komplikovaniji sklop dobijaju se šeme velikih dimenzija itd. Zbog toga se danas logički sklopovi obično opisuju na drugi način; najčešći metod reprezentacije jeste pomoću specijalizovanih, tzv. *HDL*[1] programskih jezika. Ovi jezici imaju logičke primitive ugrađene kao ključne riječi, tako da se u njima može raditi na sličan način kao sa alatima za šematski unos u smislu da se složeniji sklopovi mogu opisivati povezivanjem elementarnih kola odgovarajućim konekcijima. *HDL* jezici podržavaju međutim i konstrukcije koje se susreću u višim programskim jezicima i koje omogućavaju da se na vrlo kompaktan način izrazi i neki kompleksniji dizajn.

Dva *HDL* jezika koji su danas ubjedljivo najviše korišćeni su *VHDL*[2] ([5]) i *Verilog* ([6]). U ovom tekstu za prezentaciju je odabran *Verilog*, primarno zbog toga što mu sintaksa podsjeća na sintaksu *C* jezika[3], tako da za početnike nije teška za savladati. Većina pominjanih integrisanih okruženja za digitalno projektovanje (pa tako i *Altera Quartus II*) podržavaju, pored šematskog unosa, i unos i simulaciju na oba ova jezika. Postoji takođe i dosta kvalitetan besplatan *Verilog* simulator koji može odlično poslužiti za ovladavanje jezikom - u pitanju je program *Icarus Verilog* koji je korišćen za testiranje svih *Verilog* primjera datih u ovom tekstu.

[1] skraćenica od *hardware description language*

[2] *V* u imenu dolazi od *VHSIC*, što je opet skraćenica od *very high speed integrated circuits*

[3] sintaksa *VHDL*-a je inače slična sintaksi jezika *Ada*

2.1 Koncept modula

Primarna podrška hijerarhijskoj metodologiji modelovanja u *Verilog*-u dolazi kroz mehanizam modula. Moduli su osnovne jedinice logičkog dizajna i oni predstavljaju implementaciju digitalnih sklopova. Fajl sa *Verilog* kodom se najčešće sastoji od jednog ili više modula. Mada *Verilog* podržava više modula po jednom fajlu, najbolja je praksa da se u jednom fajlu drži jedan modul; u tom slučaju ime fajla obično odgovara imenu modula, uz ekstenziju `.v` koja se obično koristi za fajlove sa *Verilog* kodom.

Definicija modula počinje ključnom riječju `module` koju slijedi ime modula (više o imenovanju identifikatora i ostalim osnovnim sintaksnim pravilima *Verilog*-a niže). Nakon ovoga ako modul ima terminale odn. ulazno/izlazne portove slijedi u zagradama lista zarezima razdvojenih imena portova digitalnog sklopa predstavljenog modulom. Naredba se, kao i većina naredbi u *Verilog*-u, završava sa `;`. Tako bi npr. deklaracija modula koji implementira kontroler sedmosegmentnog indikatora bila oblika:

```
module bcd2ssi (x, a, b, c, d, e, f, g);
```

Ovakvo zaglavlje modula slijede deklaracije portova. Za deklaraciju ulaznih portova koristi se ključna riječ `input`, za izlazne portove ključna riječ `output`, a za bidirekcione portove ključna riječ `inout`. Ukoliko je neki port vektor (odn. sadrži više bitova kao što je recimo slučaj sa portom `x` u gornjem primjeru) onda se unutar uglastih zagrada navode dvotačkom razdvojeni indeksi *MSB*[4] i *LSB*[5] bita takvog vektora. Tako bi deklaracija portova u gornjem primjeru bila oblika:

```
input [3:0] x;
output      a, b, c, d, e, f, g;
```

Deklaraciju portova slijedi tijelo modula, koje se sastoji od eventualnih deklaracija parametara i pomoćnih signala i promjenljivih i naravno od naredbi koje implementiraju funkcionalnost modula. Defincija modula se završava ključnom riječju `endmodule`, koju za razliku od ostalih *Verilog* naredbi ne slijedi `;`.

U konkretnom slučaju biće potrebno deklarisati signale koji predstavljaju invertovane ulaze, kao i signale koji predstavljaju izlaze iz *AND* kola i te deklaracije bi bile oblika:

```
wire [2:0]  not_x;
wire [8:0]  terms;
```

[4]bit najveće težine (*most significant bit*)
[5]bit najmanje težine (*least significant bit*)

Ključna riječ `wire` se koristi za deklarisanje veza između logičkih elemenata. Portovi koji su deklarisani sa `input`, `output` i `inout` su ustvari implicitno deklarisani kao `wire`. Sa druge strane, `wire` signali mogu, kao i portovi, biti vektori, kako je ovdje i slučaj.

Osnovni način modelovanja u *Verilog*-u se sastoji od instanciranja logičkih primitiva i povezivanja odgovarajućih signala na portove tih primitiva. Ovakav način modelovanja se označava terminom *gate level* modelovanje i potpuno odgovara šematskom unosu. *Verilog* između ostalih podržava i logičke primitive: `and`, `or`, i `not` (takođe i primitive `nand`, `nor`, `xor` i `xnor`). Instanciranje ovih primitiva se vrši tako što se navede odgovarajuća ključna riječ, potom opciono ime primitive, a onda unutar zagrada zarezima razdvojena imena signala koji se povezuju na portove primitive. Primitive `and` i `or` mogu imati tri ili više portova; prvi signal u listi se odnosi na izlaz, a ostali signali su ulazi. Primitiva `not` može imati dva ili više portova, pri čemu se poslednji signal u listi odnosi na ulaz, dok su prethodni signali izlazi. Ove primitive su dovoljne za potpun opis implementacije kontrolera sedmosegmentnog indikatora. Kompletan modul koji implementira kontroler bi onda bio oblika:

```
module bcd2ssi (x, a, b, c, d, e, f, g);
   input [3:0] x; /* BCD kodirana dekadna cifra. */
   output      a, b, c, d, e, f, g; /* Signali koji kontrolisu indikatore. */

   wire [2:0]  not_x; /* Invertovani ulazni signali. */
   wire [8:0]  terms; /* Produkti koji ulaze u izracunavanje izlaznih signala. */

   /* Invertuju se ulazni signali. */
   not (not_x[0], x[0]);
   not (not_x[1], x[1]);
   not (not_x[2], x[2]);

   /* Generisu se produkti koji su potrebni za racunanje izlaznih signala. */
   and (terms[0], x[2], x[0]);
   and (terms[1], not_x[2], not_x[0]);
   and (terms[2], x[2], not_x[1]);
   and (terms[3], x[2], not_x[0]);
   and (terms[4], not_x[1], not_x[0]);
   and (terms[5], x[1], x[0]);
   and (terms[6], not_x[2], x[1]);
   and (terms[7], x[1], not_x[0]);
   and (terms[8], x[2], not_x[1], x[0]);

   /* Generisu se izlazni signali. */
   or (a, x[3], x[1], terms[0], terms[1]);
   or (b, x[3], terms[2], terms [3], terms[4]);
   or (c, x[3], not_x[2], terms [4], terms[5]);
   or (d, x[3], terms[2], terms[3], terms[6]);
   or (e, terms[1], terms[7]);
   or (f, x[3], x[2], not_x[1], x[0]);
   or (g, x[3], terms[1], terms[6], terms[7], terms[8]);
endmodule
```

Prezentirana implementacija kontolera sedmosegmentnog indikatora je potpuno ekvivalenta šemi datoj na slici 1.4, s tim što je *Verilog* kod znatno

lakši za unos i održavanje. Kao i u svakom drugom programskom jeziku, i u *Verilog*-u je više nego preporučljivo detaljno komentarisati kod. Kao što se vidi iz gornjeg segmenta, komentari se u *Verilog*-u navode na isti način kao u *C*-u. Takođe su podržani i jednolinijski komentari koji počinju sa `//` (kao u *C++*-u).

Primarna karakteristika koja razlikuje *HDL* jezike od standardnih programskih jezika jeste što je kod ovih drugih tok izvršavanja obično sekvencijalan, tj. naredbe se izvršavaju tačno onim redosledom kojim su navedene. Kod *HDL* jezika to nije slučaj, već se programi na *HDL* jezicima ustvari sastoje od velikog broja procesa koji se kontinualno i konkurentno izvršavaju. Tako u gornjem primjeru svaka promjena signala na ulazu neke logičke primitive dovodi do ponovnog izračunavanja izlaza na toj primitivi i propagacije te vrijednosti u sve one tačke sa kojima je izlaz primitive povezan, što dalje može dovesti do novog izračunavanja vrijednosti izlaza nekih drugih logičkih primitiva i tako redom. Ova inherentna konkuretnost predstavlja i najveću poteškoću koju je potrebno savladati da bi se naučilo programiranje na *HDL* jezicima.

2.2 Testiranje funkcionalnosti modula

Kao i kod šematskog unosa, po kreiranju nekog modula na *Verilog*-u isti je potrebno testirati. U tu svrhu se najčešće u *Verilog*-u kreira poseban modul koji se obično označava kao modul stimulansa. Ovaj modul nema ulazno/izlaznih portova, već samo instancira modul koji se testira, povezuje na njega određene signale i onda mijenja vrijednosti signala koji su povezani na ulaze modula koji se testira štamapajući pritom promjene signala koji su povezani na njegove izlaze. Modul stimulansa nema portove, tako da bi njegova deklaracija počela sa:

```
module stimulus;
```

Obzirom da su sva imena portova i signala u *Verilog*-u jedinstvena u hijerarhiji, zgodno je za signale stimulansa koristiti ista imena kao i za portove modula koji se testira. Signali koji drže vrijednost koja im se dodijeli se u *Verilog*-u ne deklarišu kao `wire` već se u tu svrhu koristi posebna ključna riječ `reg`. Na taj način, deklaracije signala koji se vezuju na ulazne odn. izlazne portove `bcd2ssi` modula u modulu stimulansa bi bile oblika:

```
reg [3:0] x;
wire      a, b, c, d, e, f, g;
```

Nakon ovoga je potrebno instancirati modul koji se testira. Instanciranje modula se vrši na vrlo sličan način kao instanciranje primitive: prvo se navodi

ime modula, zatim treba navesti ime instance koje je za razliku od instanciranja primitiva obavezno i koje mora biti jedinstveno, a onda u zagradama slijedi lista signala koji se povezuju na odgovarajuće portove modula. Tako bi u datom primjeru instanciranje `bcd2ssi` bilo oblika:

```
bcd2ssi _bcd2ssi (x, a, b, c, d, e, f, g);
```

Ime instance je važno za formiranje hijerahije simbola. Svi simboli u nekom *Verilog* dizajnu su u principu globalni i može im se pristupiti odgovarajućim referenciranjem po hijerarhiji. Pritom se navode tačkama razdvojena imena instanci modula kroz hijerarhiju počev od najvišeg modula u hijerarhiji (obzirom da ovaj modul nije instanciran, za njega se zapravo navodi ime modula umjesto imena instance) do modula koji sadrži simbol; na kraju se dodaje još jedna tačka i ime simbola. Tako se npr. signal `x` u modulu `stimulus` može referencirati punim imenom `stimulus.x`, a recimo signal `not_x[0]` u instanci modula `bcd2ssi` imenom `stimulus._bcd2ssi.not_x[0]`. Referenciranje imena u modulu se može izvršiti i relativnom putanjom kroz hijerarhiju, pa se tako npr. iz modula stimulansa signal `not_x[0]` u instanci modula `bcd2ssi` može referencirati i samo sa `_bcd2ssi.not_x[0]`, dok za referenciranje signala koji pripadaju modulu nikakva putanja nije potrebna. Čitav ovaj mehanizam je veoma sličan referenciranju fajlova u fajl-sistemu, s tim što je ovdje moguće referencirati samo simbole koji se nalaze na istom ili nižim nivoima hijerarhije. Referenciranje simbola koji su u različitim nivoima hijerarhije od tekućeg se inače u *Verilog*-u najčešće vrši kada se želi specificirati štampanje tih signala na standardni izlaz; mada je ovaj mehanizam imena veoma moćan, u logičkom dizajnu (kao uostalom i u programiranju uopšte) komponentama treba pristupati samo preko definisanog interfejsa i ne treba pisati kod koji zavisi od poznavanja njihove interne strukture.

Pri povezivanju signala na portove nekog modula moraju se slijediti određena pravila u pogledu uparivanja tipa signala i porta. Tako se npr. na `input` port može povezati signal koji je deklarisan kao `wire` ili `reg`, dok se na izlazni i bidirekcioni port može povezati samo `wire` signal. Takođe, širina signala i porta koji se povezuju bi trebalo da bude ista, mada *Verilog* ne smatra sintaksnom greškom ako se ne poklapaju. *Verilog* takođe omogućava da se na neke portove uopšte ne povezuju signali, u kom slučaju treba prosto preskočiti ime signala u listi (tj. takvi portovi se prepoznaju po zarezu koji sa jedne strane ima ili drugi zarez ili zagradu).

Pored instanciranja modula u kome se svaki signal uparuje sa odgovarajućim portom prema poziciji u listi signala, *Verilog* omogućava i uparivanje po imenima. U tom slučaju se u listi za svaki port prvo stavlja tačka, onda navodi ime porta i potom u zagradi piše ime signala koji se povezuje na dati port. Tako npr. ako se iz nekog razloga želi da pri instanciranju `bcd2ssi` modula izlazni signali budu navedeni na početku liste, to bi se moglo uraditi na sledeći način:

```
bcd2ssi _bcd2ssi (.a(a), .b(b), .c(c), .d(d), .e(e), .f(f), .g(g), .x(x));
```

Poslednja komponenta tijela modula stimulansa biće proces koji mijenja vrijednosti signala `x` koji je povezan na ulaz kontrolera sedmosegmentnog indikatora, te koji prati promjene i štampa vrijednosti signala na izlazima kontrolera. Za implementaciju ovog procesa biće korišćene neke napredne *Verilog* konstrukcije, koje će biti detaljnije objašnjene kasnije u ovom tekstu. Na ovom mjestu treba ipak pojasniti osnovnu funkcionalnost tih konstrukcija, kako bi kod koji slijedi bio razumljiv. Na *Verilog*-u se proces koji se odvija kontinualno od početka simulacije eksplicitno (primitive `and`, `or` i `not` takođe predstavljaju procese, jedino što su u pitanju procesi drugog tipa i što su zadati implicitno) specificira ključnom riječju `initial` koju slijedi blok naredbi koje predstavljaju proces. Ukoliko je na nekom mjestu u *Verilog*-u predviđeno da stoji blok naredbi i ukoliko taj blok naredbi sadrži više od jedne naredbe, onda se te naredbe uokviruju ključnim riječima `begin` i `end`. Naredbe koje sačinjavaju proces se sastoje od sistemske direktive kojom se specificira koji će signali biti praćeni i štampani na standardnom izlazu kada se neki od njih promijeni, zatim niza naredbi za promjenu vrijednosti signala `x` i sistemske direktive kojom se zaustavlja simulacija. Vrijednosti signala u naredbama dodjele se sastoji od širine signala (`4`), oznake baze (`'b`) i same vrijednosti (`0000` do `1001`).

U kodu koji slijedi, prve dvije naredbe se izvršavaju kada počne simulacija, a za svaku sledeću naredbu je na početku naredbe dato kašnjenje. Kašnjenje u naredbi za promjenu vrijednosti signala se specificira ispred naredbe i to tako što se iza znaka `#` navede broj vremenskih jedinica. Vremenske jedinice se inače specificiraju direktivom `` `timescale `` koja se obično stavlja pri početku fajla i kojom se navode (razdvojene `/` karakterom) vremenska jedinica u kojoj su izražena kašnjenja navedena u fajlu, kao i vremenska jedinica koja predstavlja preciznost simulacije. Ako vremenske jedinice nisu zadate `` `timescale `` direktivom, uzima se da se vrijeme izražava u sekundama. Kašnjenja su uvijek relativna, što znači da npr. ako se za neku naredbu u procesu navede kašnjenje 10 vremenskih jedinica, a za sledeću naredbu opet kašnjenje od 10 vremenskih jedinica, da će se prva naredba izvršiti 10 vremenskih jedinica nakon datog trenutka, a druga 20 vremenskih jedinica nakon istog.

Kompletan kod modula stimulansa na taj način ima oblik:

```
/* Podesavanje vremenskih jedinica. */
`timescale 1ns/1ns

module stimulus;
   /* Ulazni signali. */
   reg [3:0] x;

   /* Izlazni signali. */
   wire      a, b, c, d, e, f, g;
```

```
    /* Instanciranje komponente. */
    bcd2ssi _bcd2ssi (x, a, b, c, d, e, f, g);

    /* Generisanje test vektora. */
    initial
      begin
         $monitor($time, ": x = %d, abcdefg=%b%b%b%b%b%b%b", x, a, b, c,
                  d, e, f, g);
         /*
          $dumpfile ("data.vcd");
          $dumpvars (1, stimulus);
          */
         x <= 4'b0000;
         #30 x <= 4'b0001;
         #30 x <= 4'b0010;
         #30 x <= 4'b0011;
         #30 x <= 4'b0100;
         #30 x <= 4'b0101;
         #30 x <= 4'b0110;
         #30 x <= 4'b0111;
         #30 x <= 4'b1000;
         #30 x <= 4'b1001;
         #30 $finish;
      end
endmodule
```

Pod pretpostavkom da je modul `bcd2ssi` sačuvan u fajlu `bcd2ssi.v`, a modul `stimulus` u fajlu `stimulus.v`, generisanje izvršnog fajla koji predstavlja simulaciju se *Icarus Verilog* prevodiocem može izvršiti naredbom:

```
iverilog -o bcd2ssi bcd2ssi.v stimulus.v
```

Opcije prevodioca su slične standardnim opcijama prevodilaca za više programske jezike na *UNIX* platformi[6]. Naravno, najzgodnije je naredbu za prevođenje, kao i ostale naredbe za održavanje projekta sačuvati u odgovarajućem *make* fajlu.

Kada se pokrene simulacija odn. upravo generisani izvršni fajl, na standardnom izlazu bivaju odštampane promjene signala `stimulus` modula onako kako je i specificirano:

```
  0: x =  0, abcdefg=1110111
 30: x =  1, abcdefg=0010010
 60: x =  2, abcdefg=1011101
 90: x =  3, abcdefg=1011011
120: x =  4, abcdefg=0111010
150: x =  5, abcdefg=1101011
180: x =  6, abcdefg=1101111
210: x =  7, abcdefg=1010010
240: x =  8, abcdefg=1111111
270: x =  9, abcdefg=1111011
300: x =  0, abcdefg=1110111
```

Analizom odštampanih rezulata može se verifikovati da sklop radi kako treba.

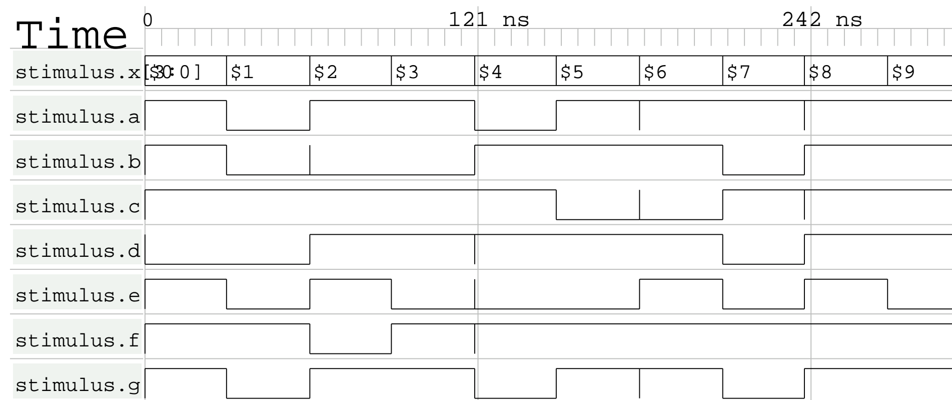
[6] za puni spisak opcija treba konsultovati *man* stranicu programa

2.3 Grafički prikaz rezultata simulacije

U datom primjeru nije previše komplikovano ispratiti listing promjene signala na izlazu; ukoliko ima više signala čije se vrijednosti štampaju i ukoliko je vrijeme snimanja duže, ovaj listing gubi na upotrebljivosti i bolje i lakše bi bilo promjene pratiti na grafičkom prikazu. U tom cilju, *Verilog* podržava snimanje signala u *VCD*[7] *ASCII* formatu koga većina programa za grafički prikaz signala podržava. Da bi se zamijenilo štampanje promjena vrijednosti signala na standardni izlaz sa njihovim snimanjem u *VCD* fajl dovoljno je u datom primjeru u modulu stimulansa `$monitor` direktivu zamijeniti direktivama:

```
$dumpfile ("data.vcd");
$dumpvars (1, stimulus);
```

Prva direktiva specificira fajl u koji će biti smješteni snimljeni signali, a druga direktiva specificira koliko nivoa hijerarhije signala i počev od kog nivoa će biti snimani. Kada se nakon ovih izmjena projekat ponovo prevede i pokrene simulacija, biće generisan `data.vcd` fajl sa promjenama signala u *VCD* formatu. Promjene signala se mogu grafički prikazati pomoću takođe besplatnog programa *GTKWave*; rezultati simulacije u ovom programu izgledaju kao na slici 2.1.



Slika 2.1: Rezulati simulacije prikazani programom *GTKWave*.

Ukoliko se koristi neko integrisano okruženje kao *Altera Quartus II* tada se test signali pripremaju i rezultati simulacije pregledaju na isti način za komponente opisane u *HDL* jeziku kako se to radi i za komponente unesene šematski. U svakom slučaju, pošto je simulacijom verifikovano pravilno funkcionisanje komponente, može se pristupiti programiranju čipa. *Icarus Verilog* ima rudimentarnu podršku za eksportovanje dizajna u format koji

[7]skraćenica od *value change dump*

podržavaju alati za programiranje iz integrisanog paketa za digitalno projektovanje firme *Xilinx*[8].

Treba uočiti da je, za razliku od simulacije *Quartus II* softverom kod koje se uzimaju u obzir kašnjenja na pojedinim logičkim elementima koji sačinjavaju dati sklop, simulacija generisana *Icarus Verilog* prevodiocem idealizovana odn. simulacija pretpostavlja da nema nikakvih kašnjenja i da se izlazni signali formiraju u istom trenutku kada dođe do promjene ulaznih signala. *Verilog* kao jezik omogućava specifikaciju kašnjenja na pojedinim logičkim elementima, o čemu će biti više riječi kasnije. Ovakva kašnjenja su međutim nepraktična za realističnu simulaciju zato što kašnjenja generalno zavise od korišćene tranzistorske tehnologije. Obzirom da *Quartus II* i ostali profesionalni alati uvijek vrše simulaciju nad modelom sklopa u datoj tehnologiji, za njih je moguće da izračunaju tačna kašnjenja pri simulaciji. Sa alatima tipa *Icarus Verilog*-a koji vrše simulaciju nad apstraktnim opisom komponente takvo nešto nije moguće, već se po eksportovanju modela u format za programiranje mora odgovarajućim alatom ponoviti simulacija da bi se verifikovalo da kašnjenja nisu prevelika da bi ugrozila ispravno funkcionisanje kola.

2.4 Osnovna *Verilog* sintaksa

Na ovom mjestu će biti nabrojana elementarna pravila *Verilog* sintakse; dosta od toga je kroz prethodni primjer neformalno uvedeno. Tekst koji slijedi je dosta suvoparan, ali je njegovo razumijevanje neophodno za dalji rad.

Osnovne leksičke konvencije za *Verilog* su slične odgovarajućim konvencijama za *C* jezik. *Verilog* pravi razliku između malih i velikih slova. Sve ključne riječi jezika pišu se malim slovima. Blanko znakovi, tabovi i oznake za novi red se ignorišu kroz kod osim ukoliko ne razdvajaju tokene; ovi znakovi se naravno ne ignorišu u stringovima. Komentari se, kako je već pomenuto mogu pisati između `/*` i `*/` ili između `//` i kraja reda.

Brojne konstante mogu biti sa ili bez specificirane širine (tj. broja bitova). U prvom slučaju prvo se navodi širina, zatim dolazi oznaka baze i na kraju vrijednost broja. Oznake za bazu su `'d` ili `'D` za decimalne, `'b` ili `'B` za binarne, `'o` ili `'O` za oktalne i `'h` ili `'H` za heksadecimalne brojeve. Brojne konstante bez specificirane širine prosto nemaju broj koji označava širinu na početku. Broj bitova konstanti bez specificirane širine je specifičan za dati simulator, ali mora biti najmanje 32. Ukoliko se u brojnoj konstanti pored oznake baze izostavi i specifikator baze, smatra se da je u pitanju decimalni broj. Neki primjeri brojnih konstanti u *Verilog*-u bi bili `4'b1111`, `16'hffff` i `8'd127` ili `'o65` i `15`.

[8] ova firma zajedno sa firmom *Altera* drži najveći dio *FPGA* tržišta

Brojne konstante bez specificirane širine mogu biti i negativne; znak `-` se pritom uvijek piše ispred baze. Brojne konstante sa specificiranom širinom su uvijek pozitivne. Brojne konstante mogu sadržati i proizvoljan broj `_` karaktera, koji se koriste za povećanje čitljivosti dugačkih brojeva, a pri interpretaciji broja se ignorišu.

Konstante koje predstavljaju stringove se navode između `"` karaktera. Stringovi se ne mogu protezati na više linija. Unutar stringova je dozvoljeno korišćenje specijalnih karaktera (`\n`, `\t` itd.) sa značenjem kao u *C* jeziku.

Imena identifikatora se mogu sastojati od alfanumeričkih znakova, kao i karaktera `_` i `$`. Identifikator ne smije počinjati brojem ili `$` karakterom.

Vrijednosti pojedinačnog signala mogu biti `0` ili `1`, ali takođe i `z` za tzv. stanje visoke impedanse (tj. vrijednost signala koji je odvojen od izvora napajanja) i `x` za nepoznatu vrijednost (npr. vrijednost nekog signala na početku simulacije). Karakter `?` je u ovom kontekstu ekvivalentan karakteru `z`. Tablice istinitosti kojima su u potpunosti opisane *Verilog* logičke primitive (`and`, `or` itd.) moraju uključiti sve varijacije ove 4 vrijednosti na ulazima kako bi primitiva bila u potpunosti opisana.

Osnovni tipovi podataka kojim se predstavljaju signali su `wire` i `reg`. Tip `wire` predstavlja prostu konekciju; podrazumijevana vrijednost signala ovog tipa je `z`. Da bi dobili neku drugu vrijednost signali ovog tipa se moraju povezati na izlaz nekog sklopa (koji onda za iste predstavlja tzv. *driver*). Tip `reg` sa druge strane predstavlja signale kojima nisu potrebni *driver*-i. Signalima tipa `reg` se može dodjeljivati vrijednost u *Verilog* kodu. Signali mogu, kako je već bilo pomenuto, biti grupisani u vektore, pri čemu se u deklaraciji iza tipa signala navode unutar uglastih zagrada indeksi krajnjih bitova razdvojeni `:` karakterom. Redosled indeksa je proizvoljan, ali se uvijek prvi indeks odnosi na *MSB* bit. Adresiranje pojedinačnih signala unutar vektora se vrši tako što se iza imena signala navede indeks u uglastim zagradama. Takođe je dozvoljeno adresirati i podopseg signala, pri čemu je sintaksa za specificiranje podopsega identična sintaksi za specificiranje opsega vektora u deklaraciji.

Verilog pored signala podržava i `integer`, `real` i `time` tipove. Semantika ovih tipova je identična semantici `reg` tipa, osim što su vrijednosti `reg` tipa uvijek neoznačeni, a vrijednosti nabrojanih tipova su označeni brojevi. Broj bitova `integer` tipa je specifičan od simulatora, ali mora biti najmanje 32. Isto važi za broj bitova vrijednosti `time` tipa. Jedinica u kojoj je izraženo vrijeme se postavlja pomenutom `‘timescale` direktivom. Sistemska direktiva `$time` daje tekuće vrijeme tokom simulacije.

Verilog omogućava deklaraciju polja veličina `reg`, `integer` i `time` tipa i vektora. Sintaksa deklaracije polja je identična sintaksi deklaracije vektora, s tim što se početni i krajnji indeks navode iza imena promjenljive (za razliku

od deklaracije vektora, gdje se ovi indeksi navode iza imena tipa). Važno je uočiti razliku između polja i vektora: vektor je jedan signal koji obuhvata više bitova, dok je polje skup više signala (koje mogu biti širine 1 ili više bita). Polja su posebno pogodna za deklaracije memorija; tako bi se npr. 512-bajtna memorija mogla deklarisati kao `reg [7:0] memory[0:511]`.

Verilog omogućava da se definišu konstante za neki modul preko ključne riječi `parameter`. Podrazumijevana vrijednost ovih konstanti se može promijeniti prilikom instanciranja modula, što omogućava konstrukciju parametrizovanih modula (npr. može se parametrizovati širina nekih portova modula i sl.).

Promjenljivim `reg` tipa dozvoljeno je dodjeljivati stringove, pri čemu bi ove promjenljive trebalo da budu dovoljne širine da mogu da prime sve karaktere stringa.

Verilog podržava i određen broj sistemskih direktiva koje omogućavaju obavljanje često potrebnih operacija. Ove direktive počinju `$` karakterom. Za ispis na standardni izlaz koristi se `$display` sistemska direktiva. Agumenti ove direktive se tretiraju slično argumentima `printf()` funkcije u *C* jeziku: stringovi se štampaju na standardnom izlazu s tim što se specijalni karakteri u njima zamjenjuju vrijednostima promjenljivih datih iza stringa u listi argumenata. Specijalni karakteri su `%d` ili `%D` za štampanje vrijednosti promjenljive u decimalnom formatu, `%b` ili `%B` za štampanje u binarnom formatu, `%o` ili `%O` za štampanje u oktalnom formatu, `%h` ili `%H` za štampanje u heksadecimalnom formatu, `%m` ili `%M` za štampanje tekuće hijerahije (dodatni argument iza stringa ovdje nije potreban) itd. Direktiva `$display` uvijek prelazi u novi red nakon štampanja argumenata. Sistemska direktiva `$monitor` je vrlo slična `$display` direktivi, s tim što je ovu direktivu dovoljno navesti samo jednom da bi uvijek bilo odštampano ono što je njome specificirano kad god se promjeni vrijednost nekog od signala koji su pomenuti u listi argumenata. Samo jedna lista zadata `$monitor` direktivom može biti aktivna u datom trenutku tokom simulacije, a štampanje se može isključivati i ponovo uključivati direktivama `$monitoroff` i `$monitoron`. Simulacija se može prekidati odn. završiti `$stop` i `$finish` sistemskim direktivama respektivno. Prva direktiva, ako je simulator sa kojim se radi interaktivan, omogućava debagovanje tokom simulacije. U spisak sistemskih direktiva treba dodati i pomenutu `$time` direktivu koja vraća tekuće vrijeme u simulaciji.

Pored sistemskih direktiva, *Verilog* podržava i pretprocesorske direktive. Ove direktive počinju ‘ karakterom i tu spadaju `‘define`, `‘include` i `‘ifdef-‘else-‘endif` direktive sa funkcionalnošću vrlo sličnom odgovarajućim direktivama *C* jezika (razlika je npr. što se makro definisan `‘define` direktivom u *Verilog*-u referencira tako što se ispred njegovog imena stavi ‘ karakter), kao i pomenuta `‘timescale` direktiva.

Ovim se završava pregled elementarne *Verilog* sintakse. Ostala sintaksna pravila će biti prezentirana kroz primjere u narednim poglavljima.

Poglavlje 3

Elementarne logičke komponente

U ovom poglavlju biće prezentirane bazične logičke komponente koje se koriste pri dizajnu mikroprocesorskih sistema. Takođe će kroz izlaganje biti dalje razmatrana sintaksa *Verilog* jezika.

3.1 Sabirači

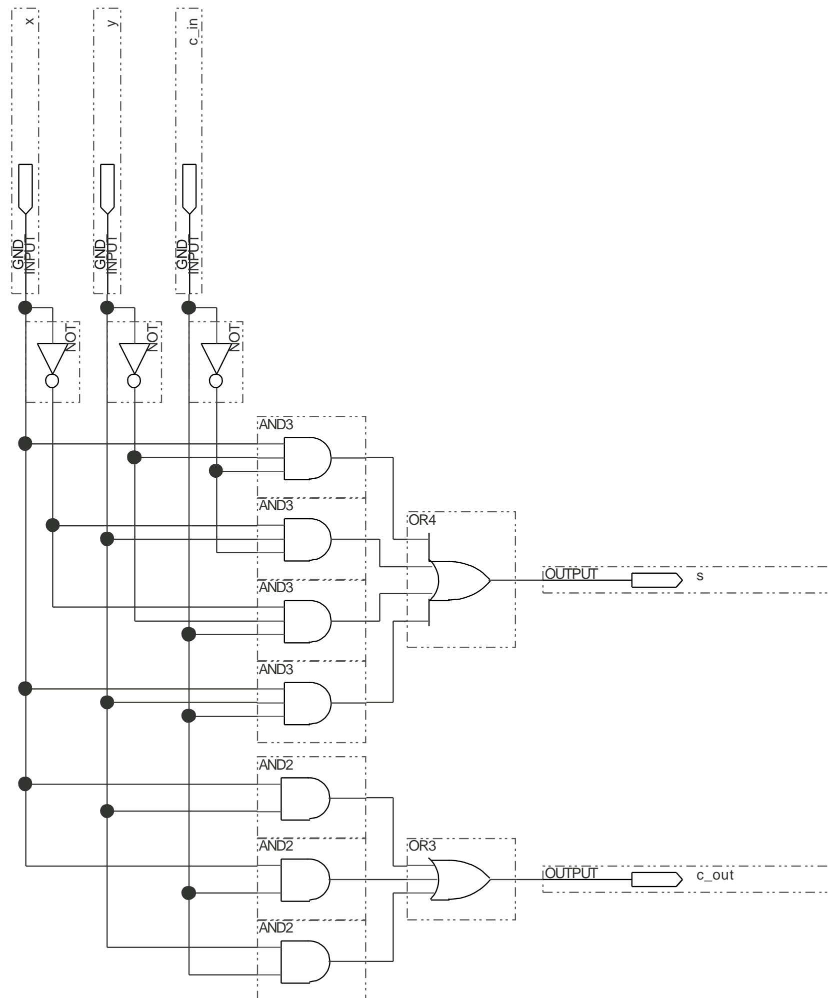
Svođenje aritmetičkih operacija na logičke počiva na implementaciji sabiranja preko logičkih operacija. Polusabiračem (*half-adder*) se naziva logičko kolo koje na ulazima prima dva jednobitna operanda `x` i `y`, a na izlazima daje njihovu sumu `s` i bit prenosa `c_out`. Ako se ispišu vrijednosti izlaza za sve moguće varijacije ulaza, dobija se sledeća tablica istinitosti:

x	y	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Na osnovu ove tablice jasno je da su logičke funkcije koje opisuju rad polusabirača oblika $s = x \oplus y$ i $c_{out} = xy$, tako da je vrlo jednostavno napraviti digitalnu komponentu koja implementira ovakvo kolo. Međutim, da bi takva komponenta bila funkcionalna, potrebno je dodati i signal ulaznog prenosa `c_in`, u kom slučaju se radi o tzv. punom sabiraču (*full adder*). Tablica istinitosti za ovu komponentu ima oblik:

x	y	c_{in}	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Pomoću *Karnaugh*-ovih mapa se mogu odrediti logičke funkcije koje opisuju rad kola i koje glase $s = \overline{x}\overline{y}c_{in} + \overline{x}y\overline{c}_{in} + x\overline{y}\overline{c}_{in} + xyc_{in}$ i $c_{in} = xy + xc_{in} + yc_{in}$. Na taj način, implementacija punog sabirača bi bila oblika kao na slici 3.1.



Slika 3.1: Šema jednobitnog punog sabirača.

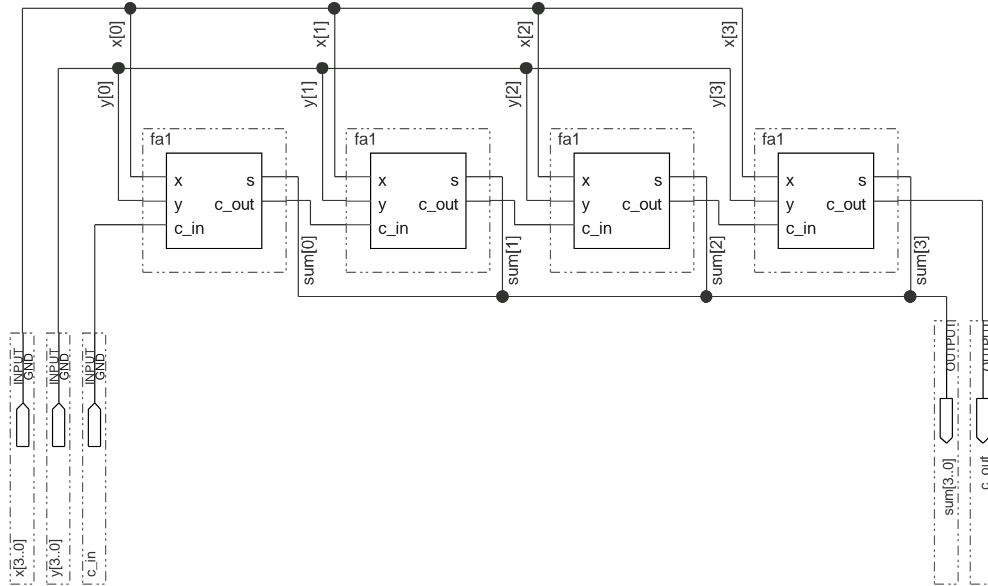
Mada je u pitanju jednostavno logičko kolo, opis istog na *Verilog*-u bi bio prilično dugačak obzirom da treba instancirati sedam *AND* primitiva i dvije *OR* primitive. *Verilog* međutim omogućava znatno kompaktniji zapis logičkih funkcija predstavljenih u obliku sume proizvoda. *Verilog* naime podržava bitske operatore `&`, `|` i `~` sa istom semantikom kao na *C* jeziku. Na taj način, logička funkcija $\overline{x}\overline{y}c_{in} + \overline{x}y\overline{c}_{in} + x\overline{y}\overline{c}_{in} + xyc_{in}$ se može kompaktno zapisati kao `(~x & ~y & c_in) | (~x & y & ~c_in) | (x & ~y & ~c_in) | (x & y & c_in)`, a logička funkcija $xy + xc_{in} + yc_{in}$ kao `(x & y) | (x & c_in) | (y & c_in)`. Prioritet ovih operatora je isti kao na *C*-u, tako da zagrade u datim izrazima nisu neophodne, ali su ipak ostavljene radi bolje čitljivosti. Svaki operator predstavlja odgovarajuće logičko kolo, a grupa od više istih operatora se može smatrati jednim logičkim kolom sa više ulaza. Pritom postoji direktna korespondencija između ovakvog opisa i opisa preko logičkih primitiva, tako da je sinteza trivijalna, a ovakav zapis je daleko čitljiviji.

Da bi se vrijednosti izračunate gornjim logičkim funkcijama dodijelile izlazima sabirača, biće iskorišćena *Verilog* naredba kontinualne dodjele. Ova naredba predstavlja kompaktan način da se zapiše proces koji mijenja vrijednosti datog signala kad god se promijeni neki od signala u izrazu koji opisuje logičku funkciju čiji je rezultat dati signal. Naredba kontinualne dodjele počinje ključnom riječju `assign` koju slijedi ime signala koji je rezultat logičke funkcije, zatim dolazi znak `=` i potom izraz koji predstavlja logičku funkciju. Tako bi npr. naredba kontinualne dodjele za signal `c_out` punog sabirača bila oblika `assign c_out = (x & y) | (x & c_in) | (y & c_in)`. Kada god se tokom simulacije promijeni neki od signala koji se nalaze na desnoj strani (tj. neki od signala `x`, `y` ili `c_in`), čitav izraz se ponovo izračunava i njegova vrijednost se dodjeljuje signalu `c_out`. Sve u svemu, ovom naredbom se dobija vrlo kompaktan zapis koji u stvari predstavlja čitav jedan logički sklop sa jednim izlazom i više ulaza u samo jednoj liniji koda, a za šta bi u šematskoj reprezentaciji ili u *Verilog* kodu koji bi koristio logičke primitive trebalo znatno više prostora.

Sada se modul koji implementira puni sabirač na *Verilog*-u može zapisati u obliku:

```
module fa1 (x, y, c_in, s, c_out);
   input  x, y; /* 1-bitni sabirci. */
   input  c_in; /* Ulazni prenos. */
   output s; /* 1-bitna suma. */
   output c_out; /* Izlazni prenos. */

   /* Izracunava se vrijednost sume. */
   assign s = (~x & ~y & c_in) | (~x & y & ~c_in) | (x & ~y & ~c_in) |
                                                       (x & y & c_in);

   /* Izracunava se izlazni prenos. */
   assign c_out = (x & y) | (x & c_in) | (y & c_in);
endmodule
```

Kaskadnim vezivanjem više jednobitnih punih sabirača može se dobiti sabirač za operande proizvoljne širine. Tako bi recimo šema 4-bitnog sabirača bila oblika kao na slici 3.2. Ovakav tip višebitnih sabirača se označava terminom *ripple-carry*.



Slika 3.2: Šema 4-bitnog *ripple-carry* sabirača.

Treba uočiti kako se pri šematskom unosu odgovarajućom anotacijom može naznačiti razdvajanje višebitnog signala na pojedinačne bitove odn. spajanje više jednobitnih signala u jedan višebitni.

Verilog model 4-bitnog *ripple-carry* sabirača bi bio oblika:

```
module rca4 (x, y, c_in, s, c_out);
   input [3:0]  x, y; /* 4-bitni sabirci. */
   input        c_in; /* Ulazni prenos. */
   output [3:0] s; /* 4-bitna suma. */
   output       c_out; /* Izlazni prenos. */

   wire [3:1]   carry; /* Unutrasnji bitovi prenosa. */

   /* Instanciraju se 1-bitni sabiraci i povezuju u kaskadu tako da formiraju
    4-bitni sabirac. */
   fa1 _fa1_0 (x[0], y[0], c_in, s[0], carry[1]);
   fa1 _fa1_1 (x[1], y[1], carry[1], s[1], carry[2]);
   fa1 _fa1_2 (x[2], y[2], carry[2], s[2], carry[3]);
   fa1 _fa1_3 (x[3], y[3], carry[3], s[3], c_out);
endmodule
```

Nedostatak *ripple-carry* sabirača je što je ukupno kašnjenje jednako zbiru kašnjenja na pojedinačnim sabiračima, što je za veću širinu operanada nepri-

hvatljivo[1]. Zato se pribjegava drugom rješenju za brze sabirače. Ako posmatramo i-ti stepen kaskade, može se logička funkcija za odgovarajući izlazni prenos napisati u obliku:

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

odnosno:

$$c_{i+1} = x_i y_i + c_i(x_i + y_i)$$

Ako se uvedu oznake $G_i = x_i y_i$ i $P_i = x_i + y_i$, onda se dati izraz svodi na:

$$c_{i+1} = G_i + P_i c_i$$

Izraz G_i se označava terminom *generate*, a izraz P_i terminom *propagate*. Ovi izrazi mogu biti izračunati u jednom koraku čim nove vrijednosti operanada postanu dostupne na ulazima sabirača. Ako se sada nastavi sa razvojem izraza za izlazni prenos, dobija se:

$$\begin{aligned} c_{i+1} &= G_i + P_i c_i \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1} \\ &= \ldots \\ &= G_i + P_i G_{i-1} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 c_0 \end{aligned}$$

Dakle, vrijednost izlaznog prenosa za svaku kaskadu može se izračunati u još dva koraka nakon inicijalnog izračunavanja svih vrijednosti G_i i P_i - jedan korak je za formiranje logičkih produkata i drugi korak za formiranje logičke sume. Na taj način, svi bitovi prenosa se formiraju u tri koraka i odmah može da počne izračunavanje bitova sume za koje je potrebno još tri koraka - jedan za invertovanje i dva za formiranje odgovarajućih logičkih produkata i suma; dakle rezultat se formira u 6 koraka. Na taj način se drastično smanjuje kašnjenje sabirača: rezonujući na isti način možemo izračunati da je za formiranje rezultata na 4-bitnom *ripple-carry* sabiraču potrebno 12 koraka.

Opisana implementacija sabirača se označava terminom *carry-lookahead*. Ograničavajući faktor za istu je *fan-in* logičkih primitiva, tako da nije moguće zadržati fiksno, malo kašnjenje za veći broj bitova operanada. Praktične implementacije se stoga ograničavaju na operande širine 4-6 bitova. Tako bi za 4-bitni *carry-lookahead* sabirač izrazi za međuprenose i izlazni prenos bili oblika:

$$\begin{aligned} c_1 &= G_0 + P_0 c_{in} \\ c_2 &= G_1 + P_1 G_0 + P_2 P_1 c_{in} \\ c_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_{in} \\ c_{out} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_{in} \end{aligned}$$

[1]interesantno je da u konkretnom slučaju 4-bitnog *ripple-carry* sabirača *Quartus II* uspijeva da optimizuje kolo tako da je kašnjenje zanemarljivo veće nego na jednobitnom sabiraču, ali to generalno ne mora biti slučaj

odnosno za implementaciju bi bile potrebne *AND* i *OR* primitive čiji je *fan-in* jednak 5, što je izvodljivo.

Verilog model 4-bitnog *carry-lookahead* sabirača bi bio oblika:

```
module cla4 (x, y, c_in, s, c_out);
   input [3:0]  x, y; /* 4-bitni sabirci. */
   input        c_in; /* Ulazni prenos. */
   output [3:0] s; /* 4-bitna suma. */
   output       c_out; /* Izlazni prenos. */

   wire [3:0]   G = x & y; /* "Generate" bitovi. */
   wire [3:0]   P = x | y; /* "Propagate" bitovi. */
   wire [3:1]   carry; /* Unutrasnji bitovi prenosa. */

   /* Izracunavaju se unutrasnji bitovi prenosa. */
   assign       carry = {G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) |
                (P[2] & P[1] & P[0] & c_in), G[1] | (P[1] & G[0]) |
                 (P[1] & P[0] & c_in), G[0] | (P[0] & c_in)};

   /* Izracunava se suma. */
   assign       s = x ^ y ^ {carry, c_in};

   /* Izracunava se izlazni prenos. */
   assign       c_out = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] &
                P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & c_in);
endmodule
```

Treba uočiti kako je pri deklaraciji signala `G` i `P` iskorišćena podrška za implicitnu naredbu kontinualne dodjele koju *Verilog* pruža. Naime, deklaracija oblika `wire [3:0] G = x & y;` je ekvivalenta deklaraciji `wire [3:0] G` koju bi u tijelu modula slijedila naredba kontinualne dodjele `assign G = x & y`. Dalje, treba primijetiti kako su bitski operatori korišćeni nad operandima čija je širina veća od 1 - ovakva konstrukcija se opet jednostavno implementira generisanjem po jedne instance logičke primitive za svaki bit operanada. Izraz za sumu je radi kompaktnijeg zapisa predstavljen bitskom *XOR* operacijom koju *Verilog* direktno podržava operatorom `^` (od bitskih operacija pored dosad pomenutih podržana je i bitska *XNOR* operacija za koju se mogu koristiti simboli `^~` ili `~^`). U istom izrazu, kao i u prethodnoj naredbi, korišćen je i operator grupisanja koji omogućava da se više signala poveže u jedan signal. Ovaj operator je predstavljen vitičastim zagradama unutar kojih se navodi lista signala koji se grupišu; operator predstavlja još jedan dobar primjer za konciznost koja se postiže korišćenjem *Verilog*-a.

Kada je potrebno napraviti šire sabirače, može se više *carry-lookahead* sabirača povezati u *ripple-carry* konfiguraciju. Alternativno, može se *carry-lookahead* mehanizam primijeniti za direktno izračunavanje bitova prenosa sabirača manje širine na isti način kako je to urađeno sa jednobitnim sabiračima i na taj način uz nešto veći utrošak logičkih primitiva postići maksimalna brzina izračunavanja.

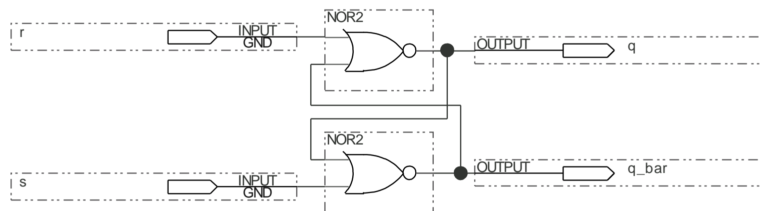
Čitava prethodna diskusija sabirača se odnosila na sabiranje nenegativnih odn. u računarskoj terminologiji neoznačenih brojeva. Negativni brojevi se

u računarskim sistemima predstavljaju isključivo tzv. drugim komplementom. Drugi komplement datog broja se dobija tako što se komplementiraju (invertuju) svi njegovi bitovi, a potom se takvom broju doda 1. Pokazuje se da se sabiranje cijelih odnosno tzv. označenih brojeva, ukoliko se negativni brojevi predstavljaju svojim drugim komplementom, može vršiti na isti način kao sabiranje pozitivnih brojeva; jedina je razlika što bit prenosa treba ignorisati (ali onda treba voditi računa o prekoračenju). Odavde slijedi da se i oduzimanje može implementirati na sabiračima, obzirom da se oduzimanje može shvatiti kao sabiranje prvog operanda sa drugim komplementom drugog operanda.

3.2 Flip-flopovi

Dosad razmatrani logički sklopovi bili su kombinatorne prirode tj. promjene vrijednosti na izlazima su zavisile samo od promjena vrijednosti na ulazima. U logičkom dizajnu se međutim puno koriste i sklopovi sekvencijalne prirode, tj. sklopovi kod kojih vrijednosti na izlazima zavise kako od vrijednosti na ulazima, tako i od prethodnih vrijednosti na izlazima. Najvažnija grupa takvih sklopova među prostijim logičkim kolima su *latch*-evi i flip-flopovi.

Latch predstavlja komponentu koja je u stanju da drži na izlazu neku vrijednost kada se ista jednom na odgovarajući način postavi. Najprostiji tip *latch*-a je *RS latch* koji se sastoji od dvije *NOR* primitive povezane kao na šemi na slici 3.3. Treba uočiti kako se vrijednosti signala sa izlaza uvode na ulaze *NOR* primitiva dajući tako kolu sekvencijalnu prirodu.



Slika 3.3: Šema *RS latch*-a.

Pretpostavimo da se na ulaze *latch*-a primijene vrijednosti $r = 1$ i $s = 0$. Tada će zbog jedinice na ulazu `r` na izlazu gornjeg *NOR* kola (tj. na izlazu `q` *latch*-a) biti 0, koja će zajedno sa nulom na ulazu `s` *latch*-a dati 1 na izlazu donjeg *NOR* kola (tj. na izlazu `q_bar` *latch*-a). Ova jedinica, kada se vrati na ulaz gornjeg *NOR* kola, neće ništa promijeniti na izlazu istog. Na taj način *latch* se nalazi u stabilnom stanju. Ako se sada signal `r` vrati na 0, izlazi `q` i `q_bar` se neće promijeniti, jer će izlaz `q_bar` i dalje držati 0 na izlazu gornjeg *NOR* kola. Potpuno ekvivaletno se sklop ponaša kada se na ulazu `s` postavi 1, a na ulazu `r` vrijednost 0 i potom vrati ulaz `s` na 0 - sklop

prelazi u stanje u kome su vrijednosti na izlazima $q = 1$ i $\overline{q} = 0$ i ostaje u tom stanju. Na taj način, primjenom kratkih impulsa na ulazima `r` odn. `s` pri čemu je na drugom ulazu 0 na izlazima `q_bar` odn. `q` se postavlja i održava jedinica, dok je na drugom izlazu uvijek 0. Imena ulaza potiču od ovakvog načina funkcionisanja *latch*-a: `r` dolazi od *reset*, a `s` od *set*.

Jedina kombinacija ulaza koja nije razmotrena je $r = 1$ i $s = 1$. Sa šeme se lako uvjeriti da u tom slučaju oba izlaza dobijaju vrijednost 0. Ovakva kombinacija vrijednosti na izlazima i nema neku primjenu, ali pravi problem sa njom nastaje ako se nakon ovoga u istom trenutku ulazi vrate na vrijednost 0. Tada će se na oba izlaza pojaviti jedinice, koje će onda prouzrokovati nule na izlazima, koje će opet prouzrokovati jedinice na izlazima i tako u krug. U realnim uslovima, komponente od kojih je napravljen *latch* su uvijek malo asimetrične, pa će tako jedno od *NOR* kola imati manje kašnjenje, što će dovesti kolo u neko stabilno stanje sa jedinicom na jednom izlazu i nulom na drugom; na kom izlazu će biti koja vrijednost ne može se međutim znati. Na taj način, najsigurnije je nikad ne doći u situaciju da budu dvije jedinice istovremeno na ulazima *latch*-a. Obzirom da se *latch*-evi u datoj konfiguraciji praktično nikad ne koriste sami za sebe, već uvijek u okviru složenijih komponenti koje obezbjeđuju da se nikad ne pojave dvije jedinice u nekom trenutku na ulazima, ovo nije praktičan problem.

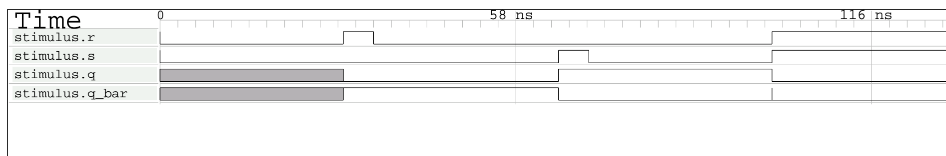
Slijedi *Verilog* modul koji implementira *RS latch*:

```
module rsl (r, s, q, q_bar);
   input  r, s; /* R i S pinovi. */
   output q, q_bar; /* Izlazni pinovi. */

   /* Generisu se izlazni signali. */
   nor /*#(5)*/ (q, r, q_bar);
   nor /*#(5)*/ (q_bar, q, s);
endmodule
```

Kod sekvencijalnih kola je, kada se koristi idealizovani simulator odn. simulator koji ne vrši simulaciju nad implementacijom sklopa u nekoj konkretnoj tehnologiji, zgodno uvesti kašnjenja da bi se pratila propagacija signala. U gornjem kodu su korišćene *Verilog* logičke primitive da bi bilo demonstrirano kako se zadaje kašnjenje za iste (treba uočiti da je specifikacija kašnjenja stavljena u komentare). *Verilog* podržava dosta složenu specifikaciju kašnjenja za logičke primitive i ista se navodi iza naziva primitive, a ispred eventualnog imena instance odn. zagrada sa listom signala koji se vezuju na portove. Kašnjenje odn. kašnjenja se navode unutar zagrada ispred kojih stoji znak `#`. Mogu se navesti zarezima razdvojena do tri kašnjenja. Prvo od njih je tzv. *rise* kašnjenje i odnosi se na tranziciju signala od vrijednosti 0 ka vrijednosti 1. Drugi tip kašnjenja je *fall* kašnjenje i odnosi se na tranziciju signala od vrijednosti 1 ka vrijednosti 0. Treći tip kašnjenja je *turn-off* kašnjenje i odnosi se na tranziciju signala na vrijednost visoke impedanse. Ako

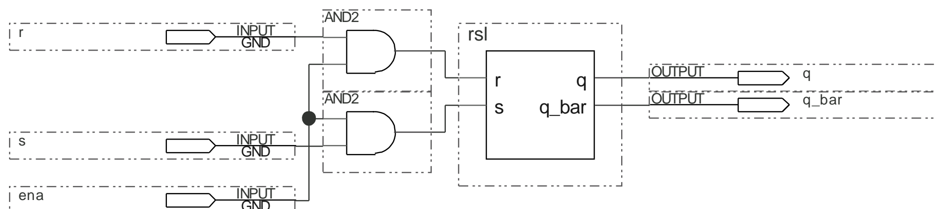
je navedeno samo jedno kašnjenje, ono se odnosi na sva tri tipa; ako su navedena dva kašnjenja, u pitanju su *rise* i *fall* kašnjenja, a *turn-off* kašnjenje uzima vrijednost manjeg od njih.

Icarus Verilog u tekućoj verziji ne obrađuje nažalost ovakve konstrukcije kako treba (to je razlog što je u gornjem kodu specifikacija kašnjenja zakomentarisana), tako da je rezultat simulacije prikazan na slici 3.4 generisan bez kašnjenja. Takva verzija modela biće korišćena i kasnije kada bude trebalo instancirati *RS latch* radi konstrukcije složenijih sklopova. Kako je već pomenuto, i jeste najbolje ne uključivati nikakva kašnjenja u *Verilog* modele već ostaviti odgovarajućim alatima da sami izračunaju kašnjenje za konkretnu implementaciju tih modela u datoj tranzistorskoj tehnologiji, ali je ipak šteta što u ovom slučaju to ne može da se uradi radi boljeg uvida u način rada *latch*-a.



Slika 3.4: Rezultati simulacije *RS latch*-a.

U mnogim primjenama je zgodno da do tranzicija signala na izlazima `q` odn. `q_bar` ne dolazi čim se promijene signali na ulazima `r` i `s`, već da se tranzicija dešava samo kada je aktivan neki drugi signal. Ovo se jednostavno postiže na način predstavljen šemom na slici 3.5. Dobijena komponenta se označava terminom *gated latch*.



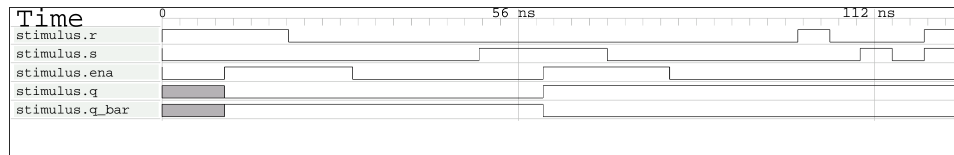
Slika 3.5: Šema *RS gated latch*-a.

Verilog implementacija *RS gated latch*-a bi bila oblika:

```
module rsgl (r, s, ena, q, q_bar);
   input  r, s; /* R i S pinovi. */
   input  ena; /* Kontrolni signal. */
   output q, q_bar; /* Izlazni pinovi. */

   wire   r_prim, s_prim; /* Ulazi RS latch-a. */

   /* Generisu se ulazi latch-a od kontrolnog signala i ulaza gated
    latch-a. */
```

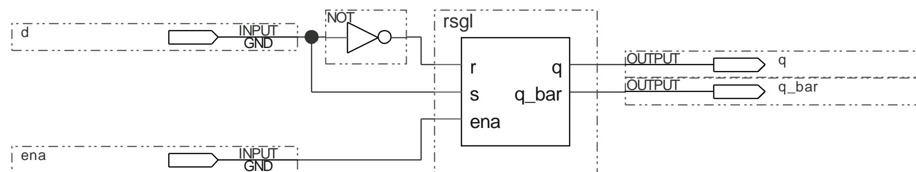
```
   assign r_prim = r & ena;
   assign s_prim = s & ena;

   /* Instancira se latch. */
   rsl _rsl(r_prim, s_prim, q, q_bar);
endmodule
```

Za generisanje pomoćnih signala `r_prim` i `s_prim` opet je umjesto instanciranja logičkih primitiva korišćen odgovarajući operator, što će biti dosljedno primjenjivano i u ostatku teksta. Rezultati simulacije *RS gated latch*-a su dati na slici 3.6. Treba uočiti kako varijacije signala `r` i `s` utiču na vrijednosti na izlazima samo ukoliko je kontrolni signal na visokom nivou - tako varijacije ulaznih signala u drugom dijelu simulacije nemaju nikakvog uticaja pošto je kontrolni signal na niskom nivou.



Slika 3.6: Rezultati simulacije *RS gated latch*-a.

Moguće nepouzdano ponašanje sklopa u situaciji kada su na oba ulaza jedinice je problem i kod *RS gated latch*-a. Stoga se obično na `r` ulaz ovog kola dovodi signal koji je invertovan u odnosu na signal koji je dat na ulazu `s`. Na taj način se dobija kolo koje se označava imenom *D gated latch* (*D* dolazi od *data*). Šema ovog kola je data na slici 3.7.

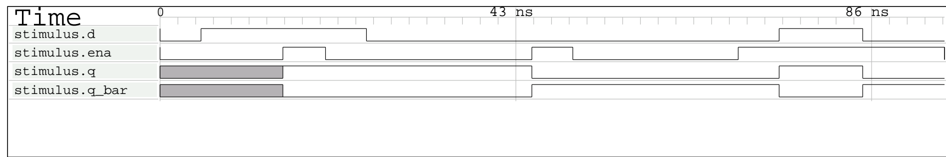


Slika 3.7: Šema *D gated latch*-a.

Verilog implementacija ovog sklopa glasi:

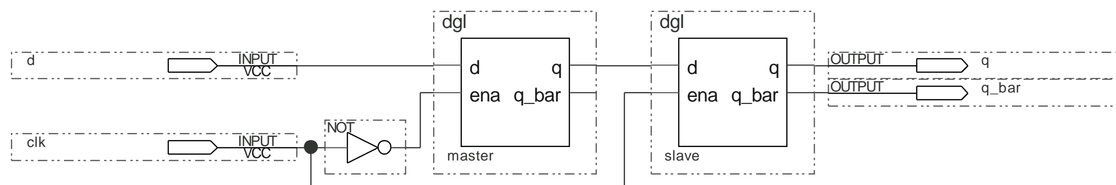
```
module dgl (d, ena, q, q_bar);
   input  d; /* D pin. */
   input  ena; /* Kontrolni signal. */
   output q, q_bar; /* Izlazni pinovi. */

   /* Instancira se RS gated latch. */
   rsgl _rsgl(~d, d, ena, q, q_bar);
endmodule
```

Rezultati simulacije *D gated latch*-a dati su na slici 3.8.



Slika 3.8: Rezultati simulacije *D gated latch*-a.

Rezultati simulacije pokazuju kako, kao i kod *RS gated latch*-a, vrijednosti na ulazu propagiraju prema izlazima tek kada kontrolni signal bude na visokom nivou. Sa druge polovine simulacije vidljivo je međutim ono što karakteriše oba sklopa: jednom kada je kontrolni signal na visokom nivou, sve varijacije ulaznih signala odmah propagiraju prema izlazima. Rješenje koje je neophodno za potpuno sinhroni rad jeste da se propagacija signala sa ulaza dešava samo u jednom trenutku, obično pri promjeni kontrolnog signala sa niskog na visoki nivo ili obrnuto. Komponenta koja funkcioniše na ovaj način se označava terminom flip-flop. Postoje dva načina da se ista implementira i to su *master-slave* i *edge-trigered* konfiguracije. Ovdje će biti razmotrena prva konfiguracija. Šema *master-slave D* flip-flopa sa tranzicijom na pozitivnoj ivici kontrolnog signala ima oblik kao na slici 3.9.



Slika 3.9: Šema *D* flip-flopa.

Slijedi *Verilog* implementacija ovog flip-flopa. Kod instanciranja *master latch*-a treba uočiti kako je iskorišćena mogućnost pomenuta u odjeljku o *Verilog* sintaksi da se port koji se ne koristi (tj. na koga se ne vezuje nijedan signal) ostavi praznim prilikom instanciranja:

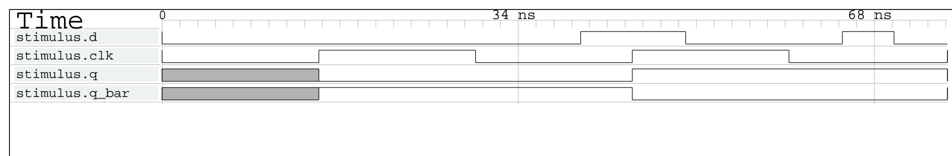
```
module dff (d, clk, q, q_bar);
   input  d; /* D pin. */
   input  clk; /* Signal takta. */
   output q, q_bar; /* Izlazni pinovi. */

   wire   d_prim; /* Izlaz master, ulaz slave gated latch-a. */

   /* Instancira se master gated latch. */
   dgl master (d, ~clk, d_prim, );

   /* Instancira se slave gated latch. */
   dgl slave (d_prim, clk, q, q_bar);
endmodule
```

Sinhronizacija se u mikroprocesorskom sistemu, odn. u njegovim pojedi-

načnim komponentama, obično vrši u odnosu na jedan globalni signal koji ciklično prelazi sa niskog na visoki nivo i obrnuto. Ovaj signal se označava kao signal takta. Broj ciklusa u jedinici vremena ovog signala naziva se učestanošću takta, koja istovremeno predstavlja i učestanost čitavog sistema. Na šemi D flip-flopa odn. u odgovarajućem *Verilog* kodu kontrolni signal je preimenovan u signal takta `clk`, obzirom da se na taj port flip-flopa zaista najčešće i povezuje signal takta.

Tokom negativnog dijela ciklusa takta, signal sa ulaza flip-flopa *latch*-a propagira na ulaz *slave latch*-a označen sa `d_prim`. Sa druge strane, ulaz *slave latch*-a je u tom dijelu ciklusa takta praktično odvojeni od izlaza flip-flopa. Tokom pozitivnog dijela ciklusa važi obrnuto: signal sa ulaza *slave latch*-a `d_prim` propagira na izlaze flip-flopa, dok je ulaz flip-flopa d odvojen od istog. To praktično znači da je signal `d_prim` stabilan tokom pozitivnog dijela ciklusa takta i da isključivo vrijednost koja se zatekla u toj tački tokom prelaza ciklusa takta sa negativne na pozitivnu vrijednost određuje vrijednosti izlaza.

Rezultati simulacije D flip-flopa su dati na slici 3.10.



Slika 3.10: Rezultati simulacije D flip-flopa.

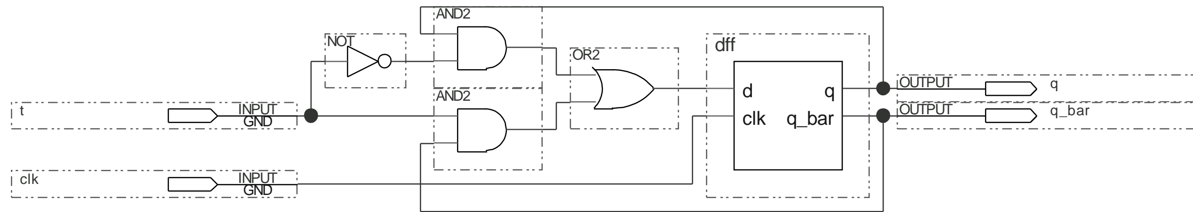
Za generisanje simulacije potrebno je u modulu stimulusa modelovati signal takta. Specificiranje svih promjena ovog signala u procesu `initial` tipa bi bilo naporno, već je umjesto toga bolje iskoristiti drugi tip procesa koji *Verilog* podržava, a to je proces `always` tipa. Ovaj proces predstavlja aktivnost koja se kontinualno ponavlja tokom simulacije; signal takta se može jednostavno modelovati korišćenje jednog `initial` procesa i jednog `always` procesa na sledeći način:

```
initial
  clk <= 1'b0;
always
  #15 clk <= ~clk;
```

Proces `initial` tipa služi za inicijalizaciju signala, a onda se procesom `always` tipa postiže da se ovaj signal svakih 15 jedinica simulacije invertuje, čime se upravo dobija željeni oblik signala takta.

Mada je D flip-flop ubjedljivo najčešće korišćeni tip flip-flopa, postoje još neki flip-flopovi koji su u upotrebi. Jedan od njih je tzv. JK flip-flop koji ovdje neće biti razmatran, a drugi je T flip-flop (T dolazi od *toggle*). Ovaj

flip-flop invertuje vrijednosti na izlazima uvijek kada je u trenutku pozitivne tranzicije signala takta na ulazu `t` visok nivo signala. Šema implementacije T flip-flopa je data na slici 3.11.



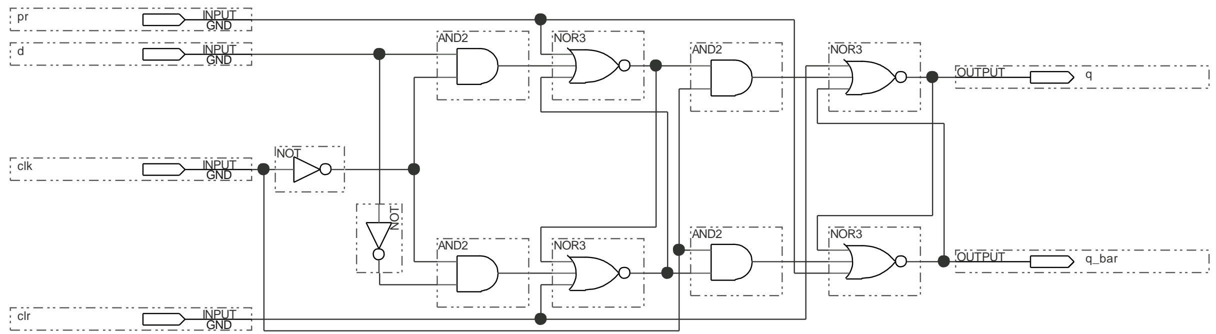
Slika 3.11: Šema T flip-flopa.

Ovakva implementacija ima međutim problem: ako se pokuša bilo kakva simulacija sa istom, može se vidjeti da izlazni signali stalno imaju nedefinisanu vrijednost. Razlog tome je što je pretpostavka da na početku simulacije svi signali imaju vrijednost visoke impedanse. Kada signali sa izlaza propagiraju na ulaze AND primitiva, oni na izlazima istih daju nedefinisanu vrijednost. Naime, već je pomenuto da tablice istinitosti svih primitiva obuhvataju i nedefinisanu i vrijedost visoke impedanse. Kada se jedan od ova dva signala nađe na ulazu bilo koje primitive, izlaz je skoro uvijek nedefinisana vrijednost. Jedini izuzeci su kada je drugi signal 0 i operacija AND (rezulat je uvijek 0) ili $NAND$ (rezultat je uvijek 1), odnosno kada je drugi signal 1 i operacija OR (rezultat je uvijek 1) ili NOR (rezultat je uvijek 0). U posmatranom slučaju, uvijek će na ulazu jedne AND primitive taj drugi signal biti 1, što će dati nepoznatu vrijednost i 0 na ulazu OR primitive i dalje nepoznatu vrijednost na ulazu D flip-flopa. Daljom propagacijom signala kroz flip-flop dobiće se nepoznate vrijednosti umjesto vrijednosti visoke impedanse na izlazima flip-flopa, što opet neće ništa promijeniti na izlazima AND primitiva i kolo će stalno ostati u takvom stanju.

Rješenje ovog problema je da se flip-flopovima dodaju posebni ulazi koji će moći direktno da postave izlaze na vrijednosti 0 ili 1, bez obzira na vrijednosti ostalih signala na ulazima. Ovakvi signali se označavaju kao *preset* i *clear* signali; da bi se ovi signali dodali potrebno je razviti flip-flop do logičkih primitiva i onda u odgovarajuće primitive dodati ove signale. Šema D flip-flopa sa *preset* i *clear* signalima je data na slici 3.12.

Odgovarajući *Verilog* kod slijedi:

```
module dpcff (d, clk, pr, clr, q, q_bar);
   input  d; /* D pin. */
   input  clk; /* Signal takta. */
   input  pr, clr; /* Preset i clear pinovi. */
   output q, q_bar; /* Izlazni pinovi. */

   wire   r_prim, s_prim; /* Izlazi master, ulazi slave gated latch-a. */
```

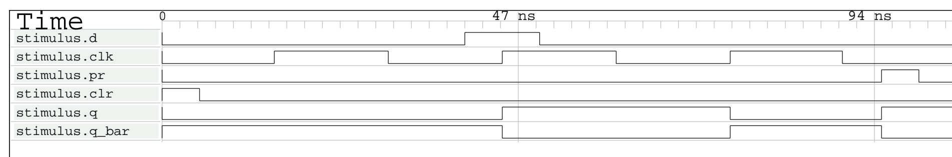


Slika 3.12: Šema *preset-clear D* flip-flopa.

```
    /* Generisu se signali r' i s'. */
    assign r_prim = ~(pr | (d & ~clk) | s_prim);
    assign s_prim = ~(clr | (~d & ~clk) | r_prim);

    /* Generisu se vrijednosti na izlazima flip-flopa. */
    assign q = ~(clr | (r_prim & clk) | q_bar);
    assign q_bar = ~(pr | (s_prim & clk) | q);
endmodule
```

Sklop funkcioniše identično običnom D flip-flopu, uz dodatak *preset* i *clear* signala (takođe treba uočiti da je zbog načina funkcionisanja ovih signala u ovom slučaju iskorišćen i `q_bar` izlaz *master latch*-a). Kada su ovi signali na niskom nivou, oni nemaju nikakvog uticaja na funkcionisanje sklopa. Kada je *preset* signal na visokom nivou, tačka `r_prim` se drži na nuli, baš kao i izlaz `q_bar`. Nula u tački `r_prim` drži kroz *AND* kolo na čijem je ulazu nulu na odgovarajućem ulazu *NOR* kola čiji je izlaz `q`. Obzirom da su i dva druga ulaza u ovo kolo koji su vezani na signale `q_bar` odn. `clr` jednaki 0 (pretpostavka je da neće istovremeno biti uključeni signali `pr` i `clr`), to se na izlazu `q` drži jedinica. Na isti način se može pokazati da su, kada se uključi signal `clr`, izlazi flip-flopa fiksirani na $q = 0$ i $\overline{q} = 1$.

Rezultati simulacije ovog flip-flopa su dati na slici 3.13.



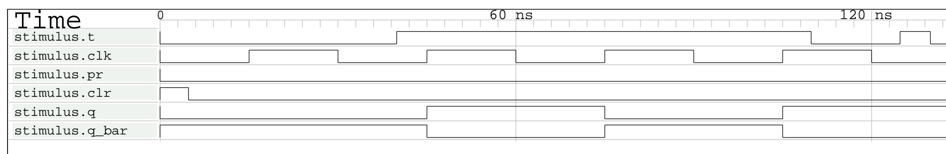
Slika 3.13: Rezultati simulacije *preset-clear D* flip-flopa.

Sada se može konstruisati T flip-flop kao na slici 3.11, sa razlikom da se umjesto običnog koristi *preset-clear D* flip-flop. *Verilog* opis ovog kola tada glasi:

```
module tff (t, clk, pr, clr, q, q_bar);
```

```
   input  t; /* T pin. */
   input  clk; /* Signal takta. */
   input  pr, clr; /* Preset i clear pinovi. */
   output q, q_bar; /* Izlazni pinovi. */

   wire   d; /* Ulaz preset-clear D flip-flopa. */

   /* Generise se signal d. */
   assign d = (~t & q) | (t & q_bar);

   /* Instancira se preset-clear D flip-flop. */
   dpcff _dpcff (d, clk, pr, clr, q, q_bar);
endmodule
```

Rezultati simulacije ovog kola su prikazani na slici 3.14. Treba primijetiti kako se u srednjem dijelu simulacije u svakom ciklusu takta mijenja vrijednost signala na izlazima zato što je ulaz `t` aktivan.



Slika 3.14: Rezultati simulacije *preset-clear T* flip-flopa.

3.3 Registri

Pojedinačni flip-flopovi mogu da čuvaju jedan bit podataka. Pri konstrukciji mikroprocesorkih sistema pogodno je grupisati više flip-flopova tako da imaju zajedničke kontrolne signale. Takva grupa flip-flopova se označava terminom registar. Slijedi *Verilog* implementacija 4-bitnog registra:

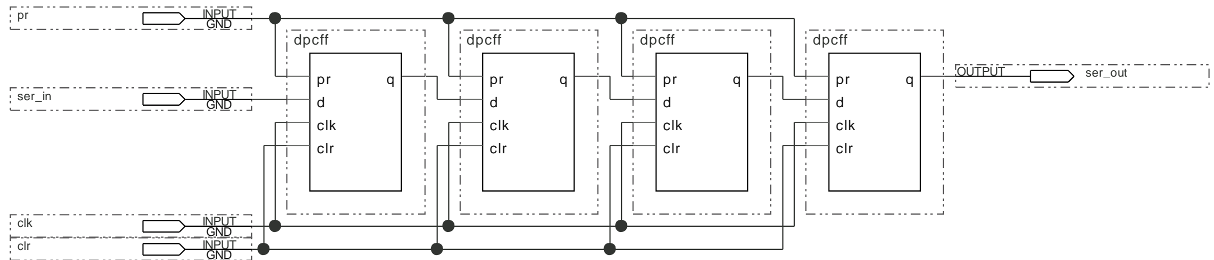
```
module reg4 (data_in, clk, pr, clr, data_out);
   input [3:0]  data_in; /* Ulazni port. */
   input        clk; /* Signal takta. */
   input        pr, clr; /* Signali za forsiranje vrijednosti izlaza. */
   output [3:0] data_out; /* Izlazni port. */

   /* Instanciranje D flip-flopova koji sacinjavaju registar. */
   dpcff _dpcff0 (data_in[0], clk, pr, clr, data_out[0], );
   dpcff _dpcff1 (data_in[1], clk, pr, clr, data_out[1], );
   dpcff _dpcff2 (data_in[2], clk, pr, clr, data_out[2], );
   dpcff _dpcff3 (data_in[3], clk, pr, clr, data_out[3], );
endmodule
```

Implementacija je veoma jednostavna: modul prosto povezuje 4 nezavisna flip-flopa u jednu cjelinu. Implementacija je mogla da se napravi i više generičkom da je korišćena ranije pominjana ključna riječ `parameter` da se definiše dužina (tj. broj bitova) registra. Tada bi u bloku deklaracija ulazni i izlazni port bili definisani na sledeći način:

```
parameter SIZE = 4;
input [SIZE-1:0]  data_in;
output [SIZE-1:0] data_out;
```

Prilikom instanciranja komponente moguće je promijeniti podrazumijevanu vrijednost parametra i na taj način bi se naizgled došlo do univerzalnog rješenja za registre proizvoljne dužine. Problem za takvu implementaciju je međutim u instanciranju pojedinačnih flip-flopova. *Verilog* naime nema ugrađenih mogućnosti za instanciranje polja komponenti. Sa druge strane, *VHDL* ima jako dobru podršku za tako nešto i dugo vremena to je bila jedna od glavnih prednosti tog jezika. Nova verzija *Verilog* standarda iz 2001. uključuje podršku za ovu konstrukciju. Međutim, većina *Verilog* alata još uvijek podržava standard iz 1995.; takav slučaj je i sa *Icarus Verilog*-om, tako da isti u ovom trenutku ne podržava polja instanci. Iz tog razloga je *Verilog* jezik prema standardu iz 1995. prezentiran i u ovom tekstu.

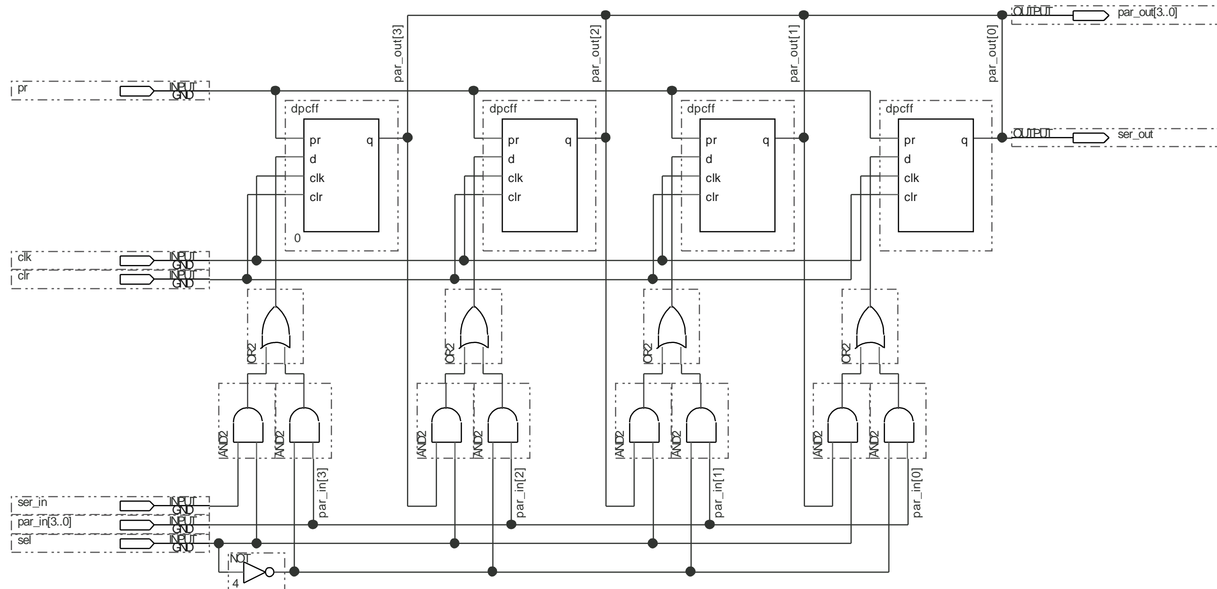
Često korišćena komponenta u mikroprocesorkim sistema je i pomjerački (*shift*) registar. Ovakvi registri omogućavaju pomjeranje podataka za po jedan bit ulijevo ili udesno u svakom ciklusu takta. Šema prostog 4-bitnog pomjeračkog registra za pomjeranje udesno je data na slici 3.15.



Slika 3.15: Šema jednostavnog pomjeračkog registra.

Obično je međutim zgodno da postoji i mogućnost direktnog pristupa sadržaju čitavog registra (baš kao kod običnih registara), pa se onda koristi nešto složenija konfiguracija kao na slici 3.16.

Kontrolni signal `sel` određuje da li se u registar učitava vrijednost (sa porta paralelnog ulaza) ili se vrši šiftovanje. Kada je ovaj signal na visokom nivou, izlaz desnog *AND* kola u svakom paru je 0, što znači da na ulaz svakog *D* flip-flopa kroz odgovarajuće *OR* kolo dolazi signal sa lijevog ulaza lijevog *AND* kola (jer je desni ulaz tog kola jednak 1), odn. čitav sklop praktično funkcioniše ekvivalentno onome prikazanom na slici 3.15 i vrši se pomjeranje. Ako je signal `sel` na niskom nivou, izlaz lijevog *AND* kola u svakom paru je 0 i na ulaz svakog flip-flopa dolazi signal koji je na desnom ulazu desnog *AND* kola (jer je lijevi ulaz istog jednak 1) odn. u registar se učitavaju vrijednosti koje se nalaze na paralelnom ulazu. U oba slučaja, sadržaj registra je stalno dostupan kroz port koji je označen kao paralelni izlaz. Treba uočiti kako



Slika 3.16: Šema pomjeračkog registra sa direktnim pristupom.

je važno što se na flip-flopovima tranzicija vrši samo u jednom trenutku tokom ciklusa takta - da su korišćeni *latch*-evi, moglo bi se desiti da sadržaj registra tokom šiftovanja "iscuri" odnosno da tokom jednog signala takta dođe do pomjeranja za više bitova.

Ukoliko se spoji bit serijskog izlaza na bit serijskog ulaza dobija se rotirajući registar. Kao i pomjerački registri, i ovi registri se dosta koriste pri konstrukciji mikroprocesorskih sistema.

Verilog kod za pomjerački registar bi bio oblika:

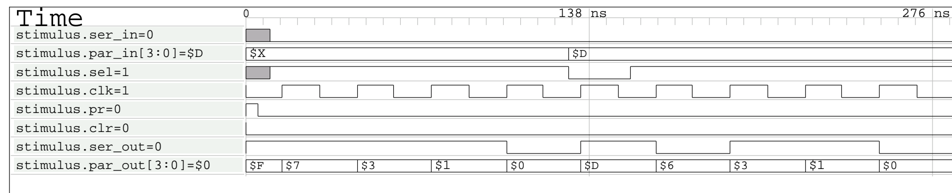
```
module rsr4 (ser_in, par_in, sel, clk, pr, clr, ser_out, par_out);
   input        ser_in; /* Serijski ulaz. */
   input [3:0]  par_in; /* Paralelni ulaz. */
   input        sel; /* Selekcija izmedju pomjeranja (1) i ucitavanja (0). */
   input        clk; /* Signal takta. */
   input        pr, clr; /* Signali za forsiranje izlaza. */
   output       ser_out; /* Serijski izlaz. */
   output [3:0] par_out; /* Paralelni izlaz. */

   /* Razdvojeni signali pomjeranja i ucitavanja. */
   wire         shift = sel, load = ~sel;

   /* Ulazi D flip-flopova koji sacinjavaju registar. */
   wire [3:0]   d = ({ser_in, par_out[3:1]} & {4 {shift}}) | ({4 {load}} &
                                                              par_in);

   /* Instanciraju se D flip-flopovi. */
   dpcff _dpcff3 (d[3], clk, pr, clr, par_out[3], );
   dpcff _dpcff2 (d[2], clk, pr, clr, par_out[2], );
   dpcff _dpcff1 (d[1], clk, pr, clr, par_out[1], );
   dpcff _dpcff0 (d[0], clk, pr, clr, par_out[0], );
```

```
   /* Generise se serijski izlaz. */
   assign      ser_out = par_out[0];
endmodule
```

Treba uočiti kako je u gornjem kodu za signale `shift` i `load`, kao i za signal `d` korišćena implicitna naredba kontinualne dodjele. *Verilog* omogućava da se za oba tipa naredbe kontinualne dodjele specificira kašnjenje pri dodjeli, čime se modeluje realno kašnjenje pri propagaciji signala na fizičkoj implementaciji digitalnog kola. Već je pomenuto da je bolje proračune kašnjenja prepustiti alatu za sintezu, te stoga specifikacija kašnjenja u naredbi kontinualne dodjele neće biti dalje pojašnjavana.

U gornjem kodu je u implicitnoj naredbi kontinualne dodjele za signal `d`, pored ranije već pomenutog operatora konkatenacije, korišćen i jedan novi operator - operator replikacije. Ovaj operator se sastoji od vitičastih zagrada sa brojem koga unutar još jednih vitičastih zagrada slijedi signal koji se replicira. Slično logičkim operatorima, ova dva operatora imaju prostu fizičku interpretaciju: operator konkatenacije prosto predstavlja fizičko grupisanje više signala, dok operator replikacije odgovara razdvajanju jednog signala na više linija.

Rezultati simulacije pomjeračkog registra su dati na slici 3.17.



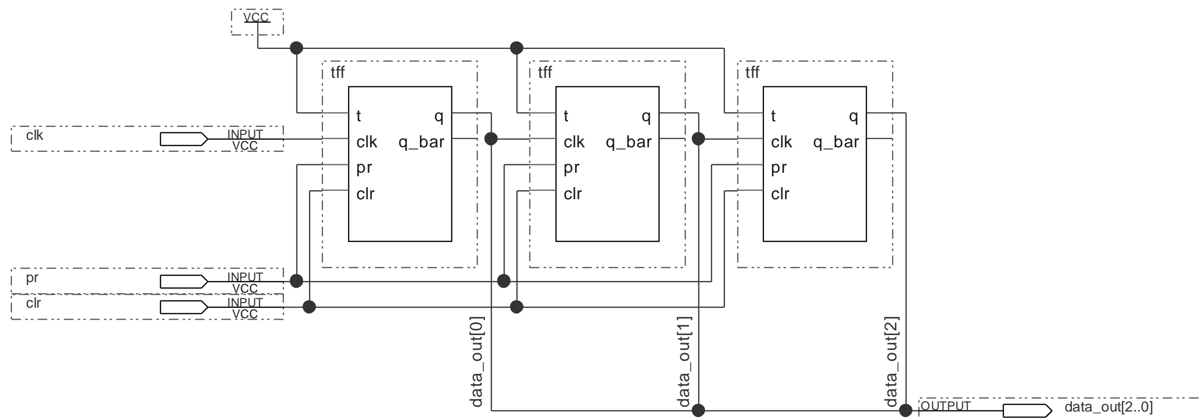
Slika 3.17: Rezulati simulacije pomjeračkog registra.

3.4 Brojači

Brojači su takođe često korišćena komponenta u mikroprocesorskim sistemima. Brojači se obično konstruišu koristeći T flip-flopove. Šema 3-bitnog brojača koji broji unatrag data je na slici 3.18.

Odgovarajući *Verilog* kod bi bio oblika:

```
module cnt3 (clk, pr, clr, data_out);
   input        clk; /* Signal takta. */
   input        pr, clr; /* Signali za forsiranje izlaza. */
   output [2:0] data_out; /* Izlazni signal. */

   /* Instanciraju se T flip-flopovi. */
   tff _tff0 (1'b1, clk, pr, clr, data_out[0], );
```



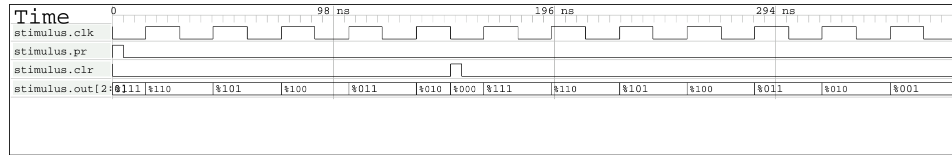
Slika 3.18: Šema 3-bitnog brojača.

```
   tff _tff1 (1'b1, data_out[0], pr, clr, data_out[1], );
   tff _tff2 (1'b1, data_out[1], pr, clr, data_out[2], );
endmodule
```

Treba obratiti pažnju na to kako su povezani flip-flopovi. Signal takta je povezan na `clk` ulaz samo prvog flip-flopa, dok je na odgovarajući ulaz svakog sledećeg flip-flopa povezan izlaz prethodnog flip-flopa. Obzirom da je ulazni signal prvog (kao i svakog narednog) flip-flopa postavljen na visok nivo, ovaj flip-flop će promijeniti stanje na izlazu sa 0 na 1 u jednom ciklusu takta i sa 1 natrag na 0 u narednom ciklusu takta. To praktično znači da će izlazni signal iz prvog flip-flopa biti istog oblika kao signal takta, samo što će ciklus tog signala biti dva puta duži. Analogno, signali na izlazu drugog odn. trećeg flip-flopa će takođe biti istog oblika kao signal takta, ali će dužina ciklusa tih signala biti 4 odn. 8 puta veća nego kod signala takta. Ukupan efekat ovakvog načina povezivanja je dvojak. Na jednoj strani, obzirom da viši bitovi brojača mijenjaju stanje tek kad niži bitovi prođu kroz sve moguće kombinacije, na izlazu brojača će se, pod pretpostavkom da su svi bitovi izlaza inicijalizovani na 1, redom izređati kombinacije bitova koje predstavljaju brojeve od 7 do 0. Sa druge strane, izlazni signali će biti ustvari signali tačno 2, 4 i 8 puta niže učestanosti od signala takta; zato se osim za brojanje, ovakve komponente koriste i za skaliranje učestanosti u mikroprocesorskom sistemu.

Takođe treba uočiti i da se kašnjenje na pojedinim flip-flopovima akumulira na sličan način kao što se akumuliralo kašnjenje kod *ripple-carry* sabirača. Iz ovog razloga, prikazana konfiguracija brojača se označava terminom *ripple* brojač.

Rezultati simulacije brojača su dati na slici 3.19.



Slika 3.19: Rezulati simulacije 3-bitnog brojača sa brojanjem naniže.

3.5 Dekoderi i multiplekseri

Dekoderi i multiplekseri su jednostavna kombinatorna kola koja se takođe često koriste u konstrukciji mikroprocesorskih sistema.

Dosta informacija u mikroprocesorskim sistemima se prenosi u kodiranoj formi (npr. kodovi instrukcija). Ove informacije je prije korišćenja obično potrebno dekodirati i u tu svrhu se koriste dekoderi. Dekoder je kolo sa n ulaza i 2^n izlaza. Pritom, uvijek je samo jedan od izlaza aktivan i to onaj čiji je indeks opisan trenutnom kombinacijom bitova na ulazu. U istinitosnim tablicama ili *Karnaugh*-ovim mapama za izlaze dekodera uvijek ima samo jedna jedinica i zato su logičke funkcije koje opisuju izlaze dekodera uvijek samo produkti; na taj način, ovakva kola je vrlo prosto konstruisati. Tako bi implementacija dekodera sa 2 ulaza i 4 izlaza u *Verilog*-u bila oblika:

```
module dec2x4 (data_in, data_out);
   input [1:0]  data_in; /* Kodirana informacija na ulazu. */
   output [3:0] data_out; /* Dekodirana informacija na izlazu. */

   /* Dekodiranje informacije sa ulaza na izlaz. */
   assign       data_out[0] = ~data_in[1] & ~data_in[0];
   assign       data_out[1] = ~data_in[1] & data_in[0];
   assign       data_out[2] = data_in[1] & ~data_in[0];
   assign       data_out[3] = data_in[1] & data_in[0];
endmodule
```

Okvirno, u dekodere se može ubrojati i kontroler sedmosegmentnog indikatora koji je ranije predstavljen.

Multiplekseri su, slično dekoderima, vrsta logičkih kola za selekciju signala. Multiplekseri naime imaju n kontrolnih signala i 2^n ulaza, takođe i jedan izlaz. U zavisnosti od vrijednosti kontrolnih signala (ovi signali se kod multipleksera često nazivaju i selektorskim signalima), vrijednost jednog od ulaza se prosleđuje na izlaz. Svaki ulaz odn. izlaz može biti višebitni. *Verilog* implementacija multipleksera sa dva ulaza proizvoljne širine bi bila oblika:

```
module mux2x1 (data_in0, data_in1, sel, data_out);
   parameter         SIZE = 4; /* Broj bitova ulaza odn. izlaza. */
   input [SIZE-1:0]  data_in0, data_in1; /* Ulazni 4-bitni signali. */
   input             sel; /* Bit selekcije ulaza za prosledjivanje na izlaz. */
   output [SIZE-1:0] data_out; /* Izlazni signal. */
```

```
    /* Prema vrijednosti bita selekcije, jedan od ulaznih signala
    propagira na izlaz. */
    assign            data_out = sel ? data_in1 : data_in0;
endmodule
```

Prvo treba primijetiti kako je ovdje korišćena ključna riječ `parameter` za definisanje broja bitova ulaza odn. izlaza čime je implementacija napravljena univerzalnom. Dalje, u naredbi kontinualne dodjele korišćen je uslovni operator. Semantika ovog operatora je ista kao na C-u. Njegova implementacija pomoću logičkih primitiva je jednostavna - tako je gornja naredba:

```
    assign            data_out = sel ? data_in1 : data_in0;
```

ekvivalentna sa:

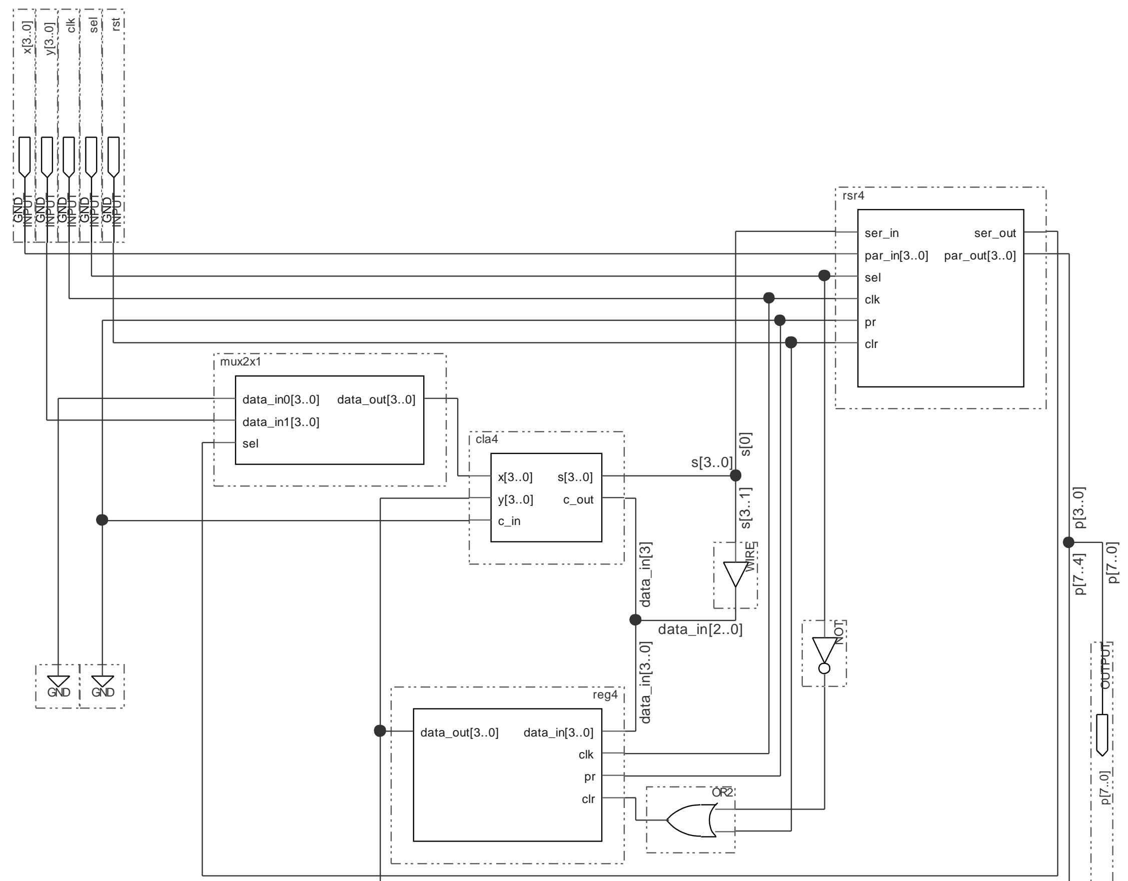
```
    assign            data_out = ({ SIZE {sel} } & data_in1) |
                                 ({ SIZE {~sel} } & data_in0);
```

3.6 Množači

Pri implementaciji množača koristi se većina dosad opisanih komponenti. Šema 4-bitnog množača je data na slici 3.20.

Odgovarajuća *Verilog* implementacija glasi:

```
module mult4 (x, y, clk, sel, rst, p);
   input [3:0]   x, y; /* 4-bitni operandi. */
   input         clk; /* Signal takta. */
   input         sel; /* Bit selekcije izmedju ucitavanja operanada i mnozenja. */
   input         rst; /* Pin za resetovanje mnozaca. */
   output [7:0]  p; /* 8-bitni izlaz. */

   wire [3:0]    add0, add1; /* Ulazi sabiraca. */
   wire [3:0]    sum; /* Izlaz sabiraca. */
   wire          carry; /* Izlazni signal prenosa sabiraca. */
   wire          curr; /* Tekuci bit mnozioca. */

   /* Instanciranje sabiraca. */
   cla4 _cla4 (add0, add1, 1'b0, sum, carry);

   /* Instanciranje registra koji sadrzi visi nibble rezultata tokom
    mnozenja. */
   reg4 _reg4 ({carry, sum[3:1]}, clk, 1'b0, ~sel | rst, add1);
   assign        p[7:4] = add1;

   /* Instanciranje pomjerackog registra koji sadrzi nizi nibble
    rezultata tokom mnozenja. */
   rsr4 _rsr4 (sum[0], x, sel, clk, 1'b0, rst, curr, p[3:0]);

   /* Selektovanje 0 ili drugog operanda za ulaz sabiraca u zavisnosti od
    tekuceg bita mnozioca. */
   mux2x1 #(4) _mux2x1 (4'b0, y, curr, add0);
endmodule
```



Slika 3.20: Šema 4-bitnog množača.

Sa ovog primjera je vidljivo da šematska metoda reprezentacije nije pogodna za veće dizajne. Crtanje šeme je komplikovano, dok je sa druge strane slika pretrpana i nije lako sa nje shvatiti način na koji sklop funkcioniše. Takođe, složenije načine razdvajanja i povezivanja signala je teško predstaviti, pa je tako bilo neophodno da se uvede bafer (primitiva koja samo prenosi na izlaz signale sa ulaza) između signala `s[3:1]` sa izlaza sabirača i `data_in[2..0]` sa ulaza registra. *Verilog* reprezentacija je daleko kompaktnija i sa iste je znatno lakše shvatiti kako radi množač.

Operandi množača se zadaju na portovima `x` i `y`. Rad sklopa se sastoji od dvije faze: prva faza se odnosi na učitavanje operanada i traje jedan ciklus takta, dok je druga faza samo množenje i traje 4 ciklusa takta. Faza se određuje na osnovu vrijednosti `sel` ulaza, pri čemu se vrijednost 0 ovog signala odnosi na učitavanje operanada, a vrijednost 1 na množenje. Komponenta koja instancira množač treba da obezbijedi da je ovaj signal jednak 0 tokom prvog ciklusa takta odnosno jednak 1 tokom naredna 4 ciklusa takta; takođe ta komponenta smije da koristi rezultat tek nakon isteka četvrtog ciklusa takta faze množenja. Ova komponenta može iskoristiti brojač u svrhu obezbjeđivanja pravilnog trajanja `sel` signala.

U fazi učitavanja operanada se zapravo operand `x` učitava u 4-bitni pomjerački registar (za pomjeranje udesno). Množenje se sprovodi onako kako bi se radilo i ručno: ide se zdesna ulijevo bit po bit kroz operand `x` i ukoliko je taj bit jednak 1 dodaje se proizvodu (koji je inicijalizovan na 0) operand `y` pomjeren za odgovarajući broj bitova ulijevo, a ukoliko je taj bit jednak 0 dodaje se proizvodu 0.

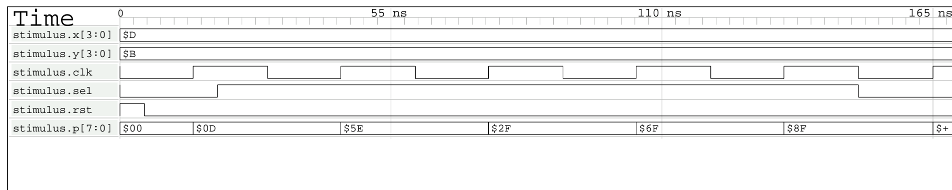
Operand `x` se pomjera udesno kroz pomjerački registar u kome je učitan, tako da je njegov tekući bit u svakom koraku ustvari serijski izlaz ovog registra. Ovaj signal predstavlja kontrolni signal multipleksera kome je na jednom ulazu 0, a na drugom ulazu operand `y`. Izlaz multipleksera je povezan na ulaz sabirača. Na taj način, kada je tekući bit operanda `x` jednak 0 na ulaz sabirača se kroz multiplekser propagira 0, a kada je tekući bit operanda `x` jednak 1 na ulaz sabirača se propagira operand `y`.

Drugi operand sabirača je tekuća vrijednost proizvoda. Samo najviša 4 bita proizvoda učestvuju u sabiranju u svakom koraku. Zato se ovi bitovi drže u posebnom 4-bitnom registru koji je vezan na drugi ulaz sabirača. Ovaj registar se resetuje na vrijednost 0 svaki put kada je signal `sel` na niskom nivou odnosno kada se vrši učitavanje operanada (registar se naravno resetuje na 0 i pri visokom nivou `rst` signala). Na ulaz ovog registra su povezani izlazi sabirača; obzirom da registar treba uvijek da sadrži najviša 4 bita proizvoda, to su zapravo na ulaze registra povezani bit prenosa sabirača (na bit najveće težine registra) kao i viša tri bita sume. Bit najmanje težine sume ne učestvuje više u sabiranju, već je njegova vrijednost konačna. Ovaj bit treba negdje smjestiti i pokazuje se da ga je veoma zgodno povezati na

serijski ulaz pomjeračkog registra. Na taj način, kako operand `x` prolazi kroz pomjerački registar tako se ovaj registar popunjava nižim bitovima finalnog proizvoda. Nakon 4 ciklusa takta, niža 4 bita proizvoda bivaju smješteni u pomjerački registar, a viša 4 bita u registar povezan na izlaze sabirača. Tako su izlazi ova dva registra zajedno praktično izlaz množača.

Verilog kod se sastoji od instanciranja odgovarajućih komponenti i njihovog povezivanja. Treba uočiti kako je inicijalizovan broj bitova ulaza multipleksera; inicijalizacija parametara nekog sklopa se vrši tako što se iza imena modula navede `#` karakter koga u zagradama slijedi lista vrijednosti koje se dodjeljuju parametrima onim redom kako su isti deklarisani u modulu. Mada je podrazumijevana vrijednost odgovarajućeg parametra multipleksera već jednaka 4, inicijalizacija je obavljena eksplicitno, za slučaj da se kasnije u modulu koji opisuje multiplekser eventualno promijeni vrijednost ovog parametra.

Rezultati simulacije množača na primjeru množenja brojeva 13 i 11 dati su na slici 3.21.



Slika 3.21: Rezulati simulacije 4-bitnog množača.

U vezi sa množačima treba na kraju pomenuti da postoje i razne šeme kojima se optimizuje rad istih; jedna često korišćena šema je *Booth*-ov algoritam ([7]). Ovakve šeme međutim znatno usložnjavaju konstrukciju množača, tako da ovdje neće biti razmatrane.

Dijeljenje se implementira pomoću odgovarajućeg sklopa čija je konstrukcija slična konstrukciji množača. Za razliku od sabiranja i oduzimanja, množenje i dijeljenje označenih brojeva nije potpuno ekvivalentno množenju i dijeljenju neoznačenih brojeva. Zato se obično množenje i dijeljenje označenih brojeva implementira tako što se na odgovarajućem sklopu izvrši operacija nad apsolutnim vrijednostima operanada, a posebnom logikom se na osnovu znakova operanada izračunava znak rezultata.

3.7 *Data flow* modelovanje

Kroz primjere u ovom poglavlju prezentirano je modelovanje digitalnih sklopova na jednom višem nivou apstrakcije od *gate level* modelovanja. Osnovne *Verilog* konstrukcije koje omogućavaju ovakav način modelovanja su

naredba kontinualne dodjele i operatori. Tehnika modelovanja kod koje se primarno koriste ove konstrukcije označava se terminom *data flow* modelovanje. Za razliku od *gate level* modelovanja koje direktno korespondira šematskom unosu, sa *data flow* modelovanjem implementacija digitalnih sklopova na *Verilog*-u počinje da liči na programiranje u višim programskim jezicima.

Lista *Verilog* operatora je data u tabeli 3.1.

<table>
<tr><th>Tip operatora</th><th>Simbol</th><th>Operacija</th></tr>
<tr><td>aritmetički</td><td>*</td><td>množenje</td></tr>
<tr><td></td><td>/</td><td>dijeljenje</td></tr>
<tr><td></td><td>+</td><td>sabiranje</td></tr>
<tr><td></td><td>-</td><td>oduzimanje</td></tr>
<tr><td></td><td>%</td><td>ostatak pri dijeljenju</td></tr>
<tr><td>logički</td><td>!</td><td>logička negacija</td></tr>
<tr><td></td><td>&&</td><td>logička konjukcija</td></tr>
<tr><td></td><td>||</td><td>logička disjunkcija</td></tr>
<tr><td>relacioni</td><td>></td><td>veće</td></tr>
<tr><td></td><td><</td><td>manje</td></tr>
<tr><td></td><td>>=</td><td>veće ili jednako</td></tr>
<tr><td></td><td><=</td><td>manje ili jednako</td></tr>
<tr><td>operatori jednakosti</td><td>==</td><td>jednako</td></tr>
<tr><td></td><td>!=</td><td>različito</td></tr>
<tr><td></td><td>===</td><td>case jednako</td></tr>
<tr><td></td><td>!==</td><td>case različito</td></tr>
<tr><td>bitski operatori</td><td>~</td><td>bitsko NOT</td></tr>
<tr><td></td><td>&</td><td>bitsko AND</td></tr>
<tr><td></td><td>|</td><td>bitsko OR</td></tr>
<tr><td></td><td>^</td><td>bitsko XOR</td></tr>
<tr><td></td><td>^~ ili ~^</td><td>bitsko XNOR</td></tr>
<tr><td>operatori redukcije</td><td>&</td><td>redukciono AND</td></tr>
<tr><td></td><td>~&</td><td>redukciono NAND</td></tr>
<tr><td></td><td>|</td><td>redukciono OR</td></tr>
<tr><td></td><td>~|</td><td>redukciono NOR</td></tr>
<tr><td></td><td>^</td><td>redukciono XOR</td></tr>
<tr><td></td><td>^~ ili ~^</td><td>redukciono XNOR</td></tr>
<tr><td>operatori šiftovanja</td><td><<</td><td>šiftovanje udesno</td></tr>
<tr><td></td><td>>></td><td>šiftovanje ulijevo</td></tr>
<tr><td>operatori grupisanja</td><td>{ }</td><td>konkatenacija</td></tr>
<tr><td></td><td>{ { } }</td><td>replikacija</td></tr>
<tr><td>uslovni operator</td><td>?:</td><td></td></tr>
</table>

Tabela 3.1: Tipovi i simboli operatora u *Verilog*-u.

Sa tabele se može vidjeti da je skup operatora veoma sličan operatorima na *C* jeziku, s tim što ovdje svi operatori moraju imati definisano ponašanje kada se u operandima jave vrijednosti visoke impedanse odn. nepoznate vrijednosti. *Case* operatori su ekvivalentni odgovarajućim operatorima jednakosti odn. nejednakosti, s tim što ovi operatori dozvoljavaju poređenje bitova koji imaju vrijednost visoke impedanse odn. neodređenu vrijednost. Obični operatori jednakosti odn. nejednakosti uvijek daju neodređenu vrijednost kao rezultat ako se neki ovakav bit pojavi u bilo kom operandu. Operatori redukcije su ekvivalentni odgovarajućim bitskim operatorima, ali ovi operatori imaju samo jedan operand i data logička operacija se sprovodi nad svim bitovima tog operanda[2].

Redosled prioriteta operatora je takođe vrlo sličan redosledu prioriteta na *C* jeziku. Tako najviši prioritet imaju unarni operatori, zatim dolaze operatori množenja, dijeljenja i ostatka pri dijeljenju, potom operatori sabiranja i oduzimanja, onda operatori šiftovanja, pa relacioni operatori, operatori jednakosti, operatori redukcije, logički operatori i na kraju uslovni operator.

Za veći broj operatora već je pokazano kako imaju jednostavnu logičku implementaciju. Slično važi i za složenije operatore. Tako se npr. operatori sabiranja i oduzimanja[3] implementiraju instanciranjem sabirača odgovarajuće širine. Slično važi i za množenje i dijeljenje. Operatori šiftovanja se implementiraju pomoću pomjeračkih registara. Operatori poređenja se mogu implementirati pomoću tzv. komparatora čija je konstrukcija dosta jednostavna, a takođe se može za njihovu konstrukciju koristiti sabirač. Na taj način, logička sinteza sklopa implementiranog tehnikom *data flow* modelovanja nije komplikovana. Pored ovoga, ova tehnika rezultira opisima digitalnih sklopova koji su znatno kompaktniji i lakši za održavanje i razumijevanje nego što je to slučaj kada se radi tehnikom *gate level* modelovanja. Stoga se danas *gate level* modelovanje vrlo rijetko koristi za konstrukciju digitalnih sklopova, već umjesto istog preovladava modelovanje na *data flow* nivou ili čak na još višem nivou apstrakcije, tzv. *behavioral* nivou modelovanja koje će biti prezentirano u narednom poglavlju.

[2]tako se npr. ovi operatori mogu iskoristiti da se kompaktno ispita da li dati operand ima bitova jednakih 0 ili 1 ili da se odredi njegova parnost

[3]obzirom da su negativni brojevi predstavljeni u drugom komplementu, oduzimanje se obavlja na isti način kao sabiranje

Poglavlje 4

Dizajn kompletnog sistema

U ovom poglavlju biće prezentiran model kompletnog (mada veoma jednostavnog) mikroprocesorskog sistema. Takođe će biti razmotrena i tehnika *behavioral* modelovanja na *Verilog*-u.

4.1 Arhitektura sistema

Mikroprocesor koji će biti modelovan će imati 16 registara opšte namjene. Ovi registri će biti 16-bitni. Za komunikaciju sa memorijom mikroprocesor će koristiti 16 adresnih linija i 16 linija za podatke. Memorijske adrese se odnose na čitave memorijske riječi, odn. neće biti moguće adresirati pojedinačne bajtove u memoriji. Na taj način, na mikroprocesor će biti potrebno povezivati memoriju koja se sastoji od određenog broja 16-bitnih riječi.

Skup instrukcija ovog mikroprocesora biće veoma rudimentaran - mikroprocesor će podržavati samo 14 instrukcija. Lista instrukcija je data u tabeli 4.1.

U tabeli je za pojašnjavanje značenja instrukcija korišćena *C*-olika notacija. Oznake `d`, `s0` i `s1` se odnose na neki od 16 registara procesora. Oznaka $\leftarrow$ predstavlja upis u registar odn. memorijsku lokaciju. Zagrade oko oznake za registar predstavljaju memorijsku lokaciju čija je adresa sadržana u registru. Oznaka `imm` se odnosi na 8-bitni označeni broj koji je direktno zadat u instrukciji. Oznaka `PC` se odnosi na *program counter* registar u kome mikroprocesor čuva memorijsku adresu tekuće instrukcije. U svakom koraku, mikroprocesor dovlači instrukciju sa adrese na koju ukazuje ovaj registar i potom je interpretira; skokovi se sprovode tako što se promijeni sadržaj ovog registra. Oznake `Z` i `C` se odnose na *zero* odn. *carry* flegove. Aritmetičko-logička jedinica u ove flegove nakon izvršene odgovarajuće instrukcije stavlja vrijednosti koje naznačavaju da li je rezultat jednak 0 odn. da li je bilo prenosa. Značenje prenosa je jasno za sabiranje, ali je takođe definisano i

Instrukcija	Operacija	Semantika operacije
`ADD d,s0,s1`	sabiranje	`d`←`s0+s1`
`INC d,s0`	inkrementiranje	`d`←`s0++`
`SUB d,s0,s1`	oduzimanje	`d`←`s0-s1`
`NOT d,s0`	bitska negacija	`d`←`~s0`
`AND d,s0,s1`	bitska konjukcija	`d`←`s0&s1`
`OR d,s0,s1`	bitska disjunkcija	`d`←`s0|s1`
`SLA d,s0`	pomjeranje ulijevo za 1	`d`←`s0<<1`
`SRA d,s0`	pomjeranje udesno za 1	`d`←`s0>>1`
`LD d,s0`	učitavanje iz memorije	`d`←`(s0)`
`ST d,s0`	upis u memoriju	`(d)`←`s0`
`LDI d,imm`	učitavanje broja	`d`←`imm`
`JMP imm`	bezuslovni skok	`goto PC+imm`
`JZ imm`	uslovni skok (ako je jednako)	`if (Z) goto PC+imm`
`JB imm`	uslovni skok (ako je manje)	`if (!Z && C) goto PC+imm`

Tabela 4.1: Spisak instrukcija mikroprocesora.

za ostale operacije; tako se npr. kod oduzimanja neoznačenih brojeva ovaj fleg odnosi na pozajmicu, kod pomjeranja se u ovaj fleg smješta bit koji je pomjeranjem izbačen i sl. Ukoliko se dva neoznačena broja oduzmu, na osnovu sadržaja `Z` i `C` flegova može se izvršiti njihova komparacija: ako je `Z` fleg postavljen operandi su naravno jednaki, a ako `Z` fleg nije postavljen, a `C` fleg jeste, pokazuje se da je prvi operand manji od drugog[1].

Kod operacija pomjeranja korišćeno je tzv. aritmetičko pomjeranje, koje karakteriše da se kod pomjeranja udesno u najviši bit upisuje ista vrijednost koja se i ranije u ovom bitu nalazila. Na taj način, održava se znak operanda. Postoji i drugi tip pomjeranja, tzv. logičko pomjeranje, kod koga se u ovaj bit uvijek upisuje 0. Kod oba tipa pomjeranja, pri pomjeranju ulijevo u bit najmanje težine operanda upisuje se 0.

Treba uočiti da je broj koji se navodi direktno u instrukcijima `LDI`, `JMP`, `JZ` i `JB` označeni 8-bitni broj, odn. da to može biti samo broj iz intervala $[-128, 127]$. Konstante koje se koriste pri računskim operacijama su obično male, a ako je potrebna neka veća konstanta onda istu treba smjestiti u memoriju i učitavati instrukcijom `LD`, tako da ovo nije veliko ograničenje za instrukciju `LDI`. Za skokove je međutim ovo dosta ograničavajući faktor, ali se ipak može prevazići razbijanjem dugačkog skoka na nekoliko kraćih. U vezi sa skokovima treba imati na umu da se uvijek pri implementaciji mikroprocesora registar `PC` na početku izvršavanja instrukcije uveća za 1 u očekivanju da se tok programa nastavi sekvencijalno odn. da naredna instrukcija bude instrukcija koja slijedi tekuću. Zato se vrijednost `imm` koja u instrukcijama

[1]obzirom da je `C` fleg postavljen, znači da je pri oduzimanju bila potrebna pozajmica, što znači da je prvi operand bio manji od drugog

skoka određuje dužinu (i pravac) skoka mora zadati ne u odnosu na tekuću instrukciju već u odnosu na narednu instrukciju.

Ovo ograničenje u pogledu dužine broja koji se navodi direktno u instrukciji dolazi od načina kodiranja instrukcije. Naime, predviđeno je da svaka instrukcija bude kodirana sa 16 bita. Prva 4 bita se odnose na kod operacije i 14 od 16 mogućih kombinacija kodiraju redom instrukcije date u tabeli 4.1. Tako je npr. `0000` kod za operaciju `AND`, `0001` kod za operaciju `INC` itd. Naredne 3 grupe od 4 bita se odnose respektivno na `d`, `s0` i `s1` registre; sa 4 bita se može adresirati tačno 16 registara. Ovo važi za tzv. troadresne instrukcije, odn. instrukcije kod kojih se zadaju oba operanda i odredište u koje se smješta rezultat (takve su instrukcije koje predstavljaju binarne aritmetičke odn. logičke operacije). Kod dvoadresnih instrukcija, kakve su instrukcije koje se odnose na unarne operacije ili instrukcije kojima se komunicira sa memorijom, bitovi koji se odnose na drugi operand su nevažni odn. uopšte se ne koriste prilikom interpretiranja instrukcije. Instrukcija `LDI` je jednoadresna i kod nje se sadržaj zadnjih 8 bitova interpretira kao `imm` broj. Isto važi za instrukcije skoka, s tim što su kod njih 4 bita između koda operacije i ovih zadnjih 8 bitova neiskorišćeni.

Mada je navedeni skup operacija zaista minimalan, pomoću njega se ipak može obaviti većina operacija koja se očekuje od mikroprocesorskog sistema. Operacije koje nedostaju se mogu implementirati kao sekvence koda u terminima postojećih - tako se npr. množenje i dijeljenje mogu implementirati pomoću aritmetičkih i instrukcija pomjeranja. Minimalan skup instrukcija je odabran da bi se implementacija mikroprocesora mogla lako sagledati, kao i da bi kodiranje instrukcija bilo jednostavno.

4.2 Organizacija sistema

Osnovne komponente od kojih se sastoji mikroprocesor su aritmetičko-logička jedinica, memorijska jedinica, ulazno-izlazna jedinica, kontrolna jedinica i skup registara. Aritmetičko-logička jedinica obavlja odgovarajuće operacije. Memorijska jedinica služi za komunikaciju sa memorijom, a ulazno-izlazna jedinica za komunikaciju sa periferijskim uređajima. Kontrolna jedinica pri izvršavanju svake instrukcije generiše signale koji upravljaju radom ostalih jedinica. Skup registara se sastoji od pomenutih 16 registara opšte namjene, ali takođe i od registara koje mikroprocesor interno koristi i koji nisu vidljivi programeru. Takav je recimo pomenuti `PC` registar, a pored njega mikroprocesor obavezno uključuje i *instruction register* (`IR`) u koji se smješta tekuća instrukcija po učitavanju iz memorije, kao i tzv. *processor status word* (`PSW`) registar koji objedinjuje flegove (u posmatranom slučaju, ovaj registar se sastoji samo od *zero* i *carry* flega).

Komunikacija između pojedinih komponenti mikroprocesorkog sistema se

odvija odgovarajućim linijama. Dizajn sistema je znatno pojednostavljen ako postoji neki zajednički skup linija na koji su povezane sve komponente. Generalno, u računarskim sistemima se skup linija koje služe za komunikaciju između komponenti sistema označava terminom magistrala, pa je to i ovdje slučaj. Unutar mikroprocesora može postojati jedan ili više takvih skupova linija, pa se u skladu sa tim procesori dijele na procesore sa jednom, dvije ili tri magistrale (za više magistrala obično nema potrebe). U posmatranom slučaju, mikroprocesor će sadržati dvije magistrale koje će biti označene sa `A` i `B`.

4.3 Model aritmetičko-logičke jedinice

Dizajn kompletnog mikroprocesorskog sistema će se sastojati od nekoliko modula, od kojih će prva biti prezentirana aritmetičko-logička jedinica. *Verilog* kod za ovaj modul glasi:

```
/* Kodovi operacija. */
`define ALU_ADD 3'b000
`define ALU_INC 3'b001
`define ALU_SUB 3'b010
`define ALU_NOT 3'b011
`define ALU_AND 3'b100
`define ALU_OR 3'b101
`define ALU_SLA 3'b110
`define ALU_SRA 3'b111

module alu (ena, in0, in1, op, out, zero, carry);
   input         ena; /* Signal za aktiviranje komponente. */
   input [15:0]  in0, in1; /* Operandi. */
   input [2:0]   op; /* Kod operacije. */
   output [15:0] out; /* Rezultat operacije. */
   output        zero, carry; /* Zero i carry flegovi. */

   /* Registarske deklaracije za izlazne portove. */
   reg [15:0]    out;
   reg           carry;

   /* Uvijek kada dodje do promjene nekog od ulaznih signala, ako je
    enable signal ukljucen izracunavaju se nove vrijednosti rezultata i
    carry flega u skladu sa zadatom operacijom. */
   always @(ena or in0 or in1 or op)
     if (ena)
       case (op)
         `ALU_ADD:
           {carry, out} <= in0 + in1;

         `ALU_INC:
           {carry, out} <= in0 + 16'd1;

         `ALU_SUB:
           {carry, out} <= in0 - in1;

         `ALU_NOT:
           {carry, out} <= {1'b0, ~in0};
```

```
        `ALU_AND:
          {carry, out} <= {1'b0, in0 & in1};

        `ALU_OR:
          {carry, out} <= {1'b0, in0 | in1};

        `ALU_SLA:
          {carry, out} <= {in0, 1'b0};

        `ALU_SRA:
          {out, carry} <= {in0[15], in0};
      endcase

    /* Po izracunavanju rezultata, postavlja se vrijednost zero flega. */
    assign zero = (out == 16'd0);
endmodule
```

Aritmetičko-logička jedinica ima dva 16-bitna ulaza `in0` i `in1` na kojima prima operande i 16-bitni izlaz `out` na koji prosleđuje izračunati rezultat. Pored ovih portova, ova jedinica ima i ulazni port `ena` koji kontroliše rad jedinice u smislu da se nova vrijednost rezultata izračunava samo ukoliko je na ovom portu signal visokog nivoa, zatim trobitni ulazni port `op` na koji dolazi kod operacije koja će biti primijenjena (za kodove pojedinih operacija su radi čitljivosti definisani odgovarajući makroi preko ranije pominjane `` `define `` *Verilog* konstrukcije), kao i izlazne portove `zero` i `carry` na kojima se po izračunavanju rezultata pojavljuju vrijednosti odgovarajućih flegova.

Implementacija modula je izvedena korišćenjem nekih *Verilog* konstrukcija koje spadaju u domen *behavioral* modelovanja. Osnovne konstrukcije kojima se izražava ovakav način modelovanja su ranije pominjane `initial` i `always` konstrukcije. Kada su ove konstrukcije uvedene u cilju opisa stimulansa koji su primjenjivani prilikom testiranja, rečeno je da svaka od njih predstavlja ustvari poseban proces koji se odvija nezavisno od i paralelno sa ostalim procesima koji opisuju datu komponentu. Procesi tipa `initial` se odvijaju samo jednom tokom simulacije i isti se rеđe koriste za opis digitalnih sklopova, obzirom da su repetitivne aktivnosti njihova primarna karakteristika. Sa druge strane, procesi tipa `always` su veoma pogodni za opis takve vrste aktivnosti i funkcionisanje aritmetičko-logičke jedinice uglavnom je izraženo upravo u vidu jednog takvog procesa.

Procesi tipa `always` uvijek imaju ugrađenu neku vremensku kontrolu; ukoliko ista ne bi postojala u ovakvom procesu, proces bi stalno iznova započinjao u istom trenutku. Vremenska kontrola koja je korišćena kada je ovakvim procesom opisivan signal takta bila je regularno kašnjenje. Kada se `always` proces koristi za opis implementacije nekog digitalnog sklopa, onda se obično kao vremenska kontrola koristi mehanizam događaja (*events*). Naime, iza `always` ključne riječi obično se navode događaji koji prouzrokuju početak izvršavanja procesa. Sintaksa za navođenje događaja zahtijeva da se prvo navede `@`, a potom u zagradi grupa signala čija izmjena dovodi do ponovnog izvršavanja

procesa. Ako ima više signala u grupi, njihova imena se razdvajaju ključnom riječju `or`. Za jednobitne signale može se specificirati i vrsta promjene vrijednosti koja će dovesti do ponovnog izvršavanja procesa. Naime, ako se želi da se proces izvrši samo u slučaju promjene vrijednosti signala sa 0 na 1 onda se ispred imena signala stavlja kvalifikator `posedge`, a ukoliko se želi da proces započne u slučaju promjene vrijednosti signala sa 1 na 0 onda se koristi kvalifikator `negedge`. Ukoliko u grupi ima više signala onda promjena bilo kog od njih započinje izvršavanje procesa (takva semantika bi se i očekivala od konteksta u kome se koristi `or`).

Proces `initial` ili `always` tipa se može sastojati od jedne ili više naredbi; u drugom slučaju naredbe moraju biti uokvirene sa `begin` i `end`. Naredbe koje se koriste za opis procesa su naredbe dodjele i naredbe kontrole toka. Naredbe dodjele o kojima je ovdje riječ su tzv. naredbe proceduralne dodjele i treba ih razlikovati od ranije pomenutih naredbi kontinualne dodjele. Naredbe kontinualne dodjele su karakteristične po tome da su stalno aktivne odn. da svaka promjena signala na desnoj strani ovakvih naredbi prouzrokuje ponovno izračunavanje izraza na desnoj strani i dodjelu izračunate vrijednosti signalu koji se nalazi na lijevoj strani ovakve naredbe dodjele. Naredbe proceduralne dodjele su sličnije naredbama dodjele u višim programskim jezicima - vrijednost izraza na desnoj strani se izračunava samo kada se u toku izvršavanja procesa naiđe na datu naredbu. U određenom smislu, naredba kontinualne dodjele je ekvivalentna naredbi proceduralne dodjele sadržanoj u procesu koji se aktivira svaki put kada se neki od signala u izrazu na desnoj strani promijeni. Tako bi npr. naredba kontinualne dodjele koja opisuje izračunavanje bita prenosa kod 1-bitnog punog sabirača:

```
assign c_out = (x & y) | (x & c_in) | (y & c_in)
```

bila ekvivalenta sledećem procesu:

```
always @(x or y or c_in)
  c_out = (x & y) | (x & c_in) | (y & c_in)
```

Međutim, treba imati na umu da naredba kontinualne dodjele može na lijevoj strani da ima samo signale `wire` tipa, a naredba proceduralne dodjele samo `reg` signale.

Postoje dva tipa naredbi proceduralne dodjele i to blokirajuće i neblokirajuće. Simbol za prvi tip dodjele je uobičajeno `=`, dok se za drugi tip naredbe proceduralne dodjele koristi oznaka `<=`. Naredba koja u datom bloku slijedi prvu od ova dva tipa naredbi se izvršava tek nakon što se završi dodjela, dok u drugom slučaju izvršavanje iduće naredbe počinje istovremeno sa posmatranom naredbom. Tako bi npr. nakon dodjela:

```
x = 1'b1;
y = x;
```

vrijednost signala y bila jednaka 1; ako bi umjesto blokirajućih bile korišćene neblokirajuće naredbe onda bi vrijednost signala y možda bila jednaka 1, a možda bi bila jednaka vrijednosti signala x prije nego što je istom dodijeljena vrijednost 1. Iz ovog primjera se vidi da su neblokirajuće naredbe zbog svog inherentnog paralelizma teže za korišćenje; međutim, obzirom da u aktivnosti digitalnih sklopova obično postoji visok stepen paralelizma ove naredbe su pogodnije za opis istih i treba nastojati da se što češće koriste.

Od naredbi za kontrolu toka, u implementaciji aritmetičko-logičke jedinice korišćene su naredbe `if` i `case`. Njihova semantika je veoma slična semantici *C* naredbi `if` odn. `switch`. Zapravo, u slučaju `if` naredbe semantika je identična (zajedno sa `else` dijelom naredbe), dok je u slučaju `case` naredbe razlika (osim u ključnoj riječi) u tome što se po završetku bloka naredbi za određenu vrijednost odmah završava naredba (tj. nije potreban neki ekvivalent `break` ključnoj riječi iz *C*-a), kao i što se skup vrijednosti za koje se izvršavaju iste naredbe razdvaja zarezima umjesto da se pojedinačno navode kao u *C*-u. Treba pomenuti i da postoje varijante `case` naredbe koje se označavaju sa `casez` i `casex` kod kojih vrijednosti visoke impedanse odn. i nepoznate vrijednosti na datim bitovima ne učestvuju u poređenju signala na osnovu koga se vrši grananje sa datom vrijednosti.

Ranije je pomenuto da su ulazni i izlazni portovi koji su deklarisani `input` odn. `output` ključnim riječima implicitno `wire` tipa. Obzirom da naredbe proceduralne dodjele na lijevoj strani mogu imati samo signale `reg` tipa, a pošto se ovakvim naredbama u implementaciji aritmetičko-logičke jedinice postavljaju vrijednosti rezultata i `carry` flega, to su u bloku deklaracija ova dva signala redeklarisani da budu `reg` tipa. Ovo se uvijek radi sa izlaznim signalima kojima treba postavljati vrijednost naredbama proceduralne dodjele.

Za postavljanje vrijednosti `zero` flega korišćena je standardna naredba kontinualne dodjele. Uvijek kada se promijeni vrijednost rezultata, ovaj fleg se postavi na 1 ili 0 u zavisnosti od toga da li je nova vrijednost rezultata jednaka 0 ili ne. Treba uočiti da se vrijednosti rezultata i flegova izračunavaju po promjeni ulaznih vrijednosti samo ukoliko je signal na ulaznom `ena` portu na visokom nivou.

Važno pitanje je kako se sintetizuju naredbe koje spadaju u domen *behavioral* modelovanja uvedene pri implementaciji aritmetičko-logičke jedinice. Procesi `always` tipa se prosto implementiraju kao posebne komponente unutar datog sklopa. Signali koji su dati kao grupa signala čije promjene izazivaju ponovno izvršavanje procesa predstavljaju ulaze u ovakve komponente. Naredbe proceduralne dodjele se implementiraju na sličan način kao naredbe kontinualne dodjele: formira se logičko kolo koje izračunava vrijednost izraza na desnoj strani naredbe dodjele i čiji je izlaz signal koji stoji na lijevoj strani ove naredbe. Ukoliko međutim u bloku imamo niz blokirajućih naredbi proce-

duralne dodjele onda je potrebno obezbijediti sekvencijalno izvršavanje ovih naredbi. U tom cilju moguće je iskoristiti brojač, pri čemu bi pri odbrojavanju redom bile izvršavane pojedinačne blokirajuće naredbe. Ovim se naravno usporava izvršavanje čitavog procesa, pa je to razlog više da se blokirajuće naredbe izbjegavaju.

Za implementiranje `if` i `case` naredbi je važno šta se nalazi u tijelu istih. Veoma često se, kao što je u gornjem primjeru slučaj sa `case` naredbom, grane ovih naredbi sastoje od dodjela različitih vrijednosti istom signalu. U tom slučaju se ove naredbe jednostavno implementiraju multiplekserima; `if` naredba je tada zapravo potpuno ekvivalentna uslovnom operatoru za koga je već rečeno da se standardno implementira kao 2×1 multiplekser. Ukoliko su grane ovih naredbi nezavisne, onda se svaka grana implementira kao posebno logičko kolo, pri čemu se tom kolu dodaje kontrolni signal koji ga aktivira (u smislu propagacije vrijednosti ulaznih signala prema izlazima) samo ukoliko je uslov za izvršavanje odgovarajuće grane ispunjen.

4.4 Model skupa registara opšte namjene

Implementacija skupa registara opšte namjene mikroprocesora je veoma jednostavna. Radi se o 16 običnih 16-bitnih registara, ali je njihova implementacija izdvojena u poseban modul da bi se pristup registrima prilagodio modelu mikroprocesora sa dvije magistrale koji će biti korišćen. Implementacija ove komponente nema novih *Verilog* konstrukcija:

```
module regfile (sel_out0, sel_out1, ena_in, sel_in, in, out0, out1);
   input [3:0]   sel_out0, sel_out1; /* Signali selekcije registara cije
                                        ce vrijednosti biti proslijedjene
                                        na izlaz. */
   input         ena_in; /* Kontrolni signal za upis u registar. */
   input [3:0]   sel_in; /* Signal selekcije registra u koji se upisuje
                            vrijedosti. */
   input [15:0]  in; /* Vrijednost koja se upisuje u gornji registar. */
   output [15:0] out0, out1; /* Vrijednosti selektovanih registara. */

   reg [15:0]    regs[0:15]; /* Skup registara opste namjene mikroprocesora. */

   /* Na izlaze se postavljaju vrijednosti selektovanih registara. */
   assign        out0 = regs[sel_out0];
   assign        out1 = regs[sel_out1];

   /* Ukoliko je dozvoljen upis, u odgovarajuci registar se upisuje
    vrijednost sa ulaza. */
   always @(ena_in or in or sel_in)
     if (ena_in)
       regs[sel_in] <= in;
endmodule
```

Modul ima dva 16-bitna izlaza i jedan 16-bitni ulaz koji će biti vezani na magistrale. Registre čije se vrijednosti prosleđuju na ove izlaze odn. registar

u koga se upisuje vrijednost na ulazu određuju 4-bitni selektorski signali. Izlazi se ažuriraju uvijek kada se odgovarajući selektorski signali promijene, dok je za upis ulazne vrijednosti u dati registar potrebno i da bude aktivan odgovarajući kontrolni signal.

4.5 Model *datapath* komponente

Prethodno implementirane komponente, blok registara opšte namjene i aritmetičko-logička jedinica, biće instancirani u komponenti koja objedinjuje implementaciju magistrala, namjenskih registara i rudimentarne memorijske jedinice. Ova komponenta se obično označava terminom *datapath*, obzirom da enkapsulira interni transfer podataka kroz mikroprocesor. Implementacija magistrala je jednostavna, radi se prosto o dva skupa od 16 žica kojima se razmjenjuju podaci između nabrojanih komponenti. Sa druge strane, obzirom da se veći broj signala može pojaviti na magistrali, potrebno je pažljivo dizajnirati odgovarajuće kontrolne signale koji će određivati koji se signali i kada propuštaju na magistrale.

Postoje dvije tehnike modelovanja u slučaju kada neki signal ima dva ili više izvora. U oba slučaja su neophodni neki kontrolni signali koji će određivati koji se izvor propušta na dati signal. Prva tehnika je eksplicitna i podrazumijeva korišćenje multipleksera. Kod druge tehnike se koriste vrijednosti visoke impedanse, tj. sa datog izvora se na posmatrani signal prosleđuje ili trenutna vrijednost ili vrijednost visoke impedanse, u zavisnosti od kontrolnog signala. Pritom se mora obezbijediti da u svakom trenutku samo jedan od izvora prosleđuje svoju vrijednost na posmatrani signal, a da svi ostali prosleđuju vrijednost visoke impedanse. Eksplicitna tehnika sa multiplekserima je jednostavnija za razumijevanje, pa će uglavnom biti korišćena u opisu magistrala mikroprocesora.

Mada će implementacija magistrala biti u ovom tekstu pojašnjena prije implementacije kontrolne jedinice, konstrukcija ova dva sklopa obično teče uporedo. Naime, da bi se implementirala magistrala potrebno je analizirati instrukcije procesora i razbiti ih na primitivnije operacije. Ove operacije podrazumijevaju određene transfere na magistrali, što dalje određuje šta se sve i na koji način povezuje na magistralu, a takođe i koji su kontrolni signali potrebni za pravilno sekvenciranje tih transfera.

Izvršavanje svih instrukcija počinje na isti način - prvo je potrebno instrukciju dovući iz memorije i dekodirati je. Već je pomenuto da će memorijska jedinica biti veoma jednostavna; ista će se sastojati od dva registra i jednog kontrolnog signala. U 16-bitni registar *memory address register* (`MAR`) se upisuje adresa memorijske lokacije kojoj se pristupa. Kontrolnom signalu se dodjeljuje vrijednost 1 ili 0 u zavisnosti od toga da li se želi upis ili čitanje iz memorije. Registar *memory data register* (`MDR`) je takođe 16-bitni i

u njega se u slučaju upisa u memoriju stavlja podatak koji se želi upisati, a u slučaju čitanja iz memorije u njega nakon pola ciklusa takta[2] biva upisan sadržaj memorijske riječi čija je adresa data u registru `MAR`. Obzirom da komunikacija između pojedinih registara teče isključivo preko magistrala, na početku svake instrukcije treba na jednu magistralu propustiti sadržaj `PC` registra i takođe vrijednost sa te magistrale propustiti u `MAR` registar. Na kontrolni signal koji određuje memorijsku operaciju treba staviti vrijednost 0 koja označava čitanje iz memorije. Pošto je sadržaj registra `PC` zapravo adresa naredne instrukcije, onda će se ta instrukcija nakon pola ciklusa takta naći u registru `MDR`.

Nakon što je iz memorije učitana naredna instrukcija, treba je prebaciti u `IR` registar, što će podrazumijevati još jedan transfer na magistrali sličan opisanom. Treba međutim primijetiti da se u prvom ciklusu može uraditi još nešto korisno: pošto je sadržaj registra `PC` već na magistrali, može se isti inkrementirati u aritmetičko-logičkoj jedinici da bi se na taj način izračunala adresa sledeće instrukcije (u slučaju sekvencijalnog izvršavanja programa). Ulazi aritmetičko-logičke jedinice su direktno povezani na magistrale, tako da sve što treba uraditi da bi se inkrementirao sadržaj `PC` registra koji se nalazi na magistrali jeste da se preko odgovarajućih kontrolnih signala aktivira aritmetičko-logička jedinica odn. da se specificira kod operacije koji odgovara operaciji inkrementiranja.

U drugom koraku se, kako je već rečeno, preko jedne magistrale sadržaj `MDR` registra prebacuje u `IR` registar, a druga magistrala se može iskoristiti da se rezultat iz aritmetičko-logičke jedinice odnosno sadržaj registra `PC` uvećan za 1 vrati natrag u registar `PC` tako da ovaj registar opet sadrži adresu naredne instrukcije. Aritmetičko-logička jedinica se pritom u ovom koraku isključuje; naime obzirom da su ulazi ove jedinice direktno vezani za magistrale, ako bi ova jedinica ostala uključena odmah po pojavi novih vrijednosti na magistralama došlo bi do preračunavanja vrijednosti na izlazu i prethodni rezultat bi bio izgubljen. Može se zaključiti da je dosad izlaz na magistralu bio potreban iz registra `PC`, iz `MDR` registra i sa izlaza aritmetičko-logičke jedinice, pri čemu druga dva signala moraju biti na različitim magistralama, a takođe se mora voditi računa da magistrala na koju se u prvom koraku propušta sadržaj registra `PC` bude vezana na ulaz `in0` aritmetičko-logičke jedinice, obzirom da je moguće inkrementirati samo vrijednost na tom ulazu. Da bi se sve ovo postiglo, pred ulaze na magistrale se kako je već nagoviješteno stavljaju multiplekseri. Ako pretpostavimo da je magistrala `A` vezana na ulaz `in0`, a magistrala `B` na ulaz `in1` aritmetičko-logičke jedinice, onda na multiplekser koji generiše signal za magistralu `A` zasad treba kao ulaze

[2] ovo je najveće pojednostavljenje u implementaciji memorijske jedinice - memorija obično radi na manjoj učestanosti od procesora što prouzrokuje da memorijski transferi traju više ciklusa takta procesora; ovo se pokušava ubrzati hijerarhijom *cache* memorija što onda bitno usložnjava konstrukciju memorijske jedinice

uvesti sadržaj registra `PC` i recimo sadržaj registra `MDR`, a na multiplekser koji generiše signal za magistralu `B` treba kao ulaz uvesti izlaz aritmetičko-logičke jedinice. Ovim multiplekserima će biti dodavano još ulaza kako se daljom analizom izvršavanja instrukcija ukaže potreba. Dalje treba uočiti i da su potrebni kontrolni signali za upis u registre `PC`, `IR` i `MAR`. Na taj način, analizom izvršavanja instrukcija formira se i skup kontrolnih signala, tako da implementacija kontrolne jedinice (bar što se tiče njenog interfejsa) treba da teče uporedo.

Za razliku od prva dva koraka, koji se inače u ovom obliku mogu označiti terminima *fetch* (jer se u prvom koraku startuje dovlačenje instrukcije iz memorije) odn. *decode* (jer u drugom koraku instrukcija stiže u `IR` registar čime je praktično i dekodirana obzirom na prost format instrukcija), koji su isti za sve instukcije, preostali koraci su specifični za svaku instrukciju ponaosob. Ovdje će biti analizirane tri instrukcije, koje su tipični predstavnici osnovnih grupa instrukcija koje podržava mikroprocesor, a to su aritmetičko-logičke, instrukcije za rad sa memorijom i instrukcije skoka.

Za izvršavanje `ADD` instrukcije potrebna su još dva koraka. U prvom koraku se iz skupa registara aktiviraju oni registri koji su u instrukciji navedeni kao operandi. Sadržaj ovih registara se šalje na ulaze aritmetičko-logičke jedinice preko magistrale. Prema tome, na multipleksere koji kontrolišu ulaze na magistralu treba dodati i izlaze iz skupa registara. Na aritmetičko-logičkoj jedinici se, isto kao kod inkrementiranja `PC` registra, uključuje kontrolni signal koji aktivira ovu komponentu, a takođe se i šalje kod operacije koji odgovara sabiranju. Ovaj korak u izvršavanju `ADD` instrukcije, odn. ekvivalentan korak za ostale instrukcije se može označiti terminom *execute* jer se u njemu zapravo obavlja ono što je centralno za datu instrukciju. Naredni korak predstavlja upis rezultata natrag u odgovarajući registar. Ovaj korak je onda logično označiti terminom *write*. Još u *execute* koraku treba na odgovarajući ulaz skupa registara dovesti signal koji selektuje odredišni registar, tako da u *write* koraku preostaje da se izlaz aritmetičko-logičke jedinice pusti na magistralu, kao i da se aktivira signal koji omogućuje upis u skup registara. Obzirom da je ranije odabrano da se izlaz aritmetičko-logičke jedinice šalje na magistralu `B`, to onda treba ulaz `in` skupa registara preko koga se unosi podatak u zadati registar priključiti na ovu magistralu. Signal koji selektuje registar u koji će biti upisana vrijednost sa magistrale se mora postaviti korak ispred upisa da ne bi prilikom aktiviranja signala upisa došlo do upisa u pogrešan registar. Ovo važi i za svaki drugi upis ako postoji više lokacija za upis (npr. za upis u memoriju) - izbor lokacije se mora izvršiti prije nego što se dozvoli upis.

Kod `ST` instrukcije se u trećem koraku na magistrale propuštaju vrijednosti registra koji određuje adresu memorijske lokacije na koju treba upisati podatak odn. registra koji sadrži podatak. Na magistralu dakle i u ovom slučaju dolaze signali sa izlaza skupa registara, tako da ne treba dodavati ništa na

multipleksere koji kontrolišu ulaze na magistrale. Sa druge strane, može se zaključiti i da je za ulaz u registar MDR potreban multiplekser, jer u ovaj registar dolaze signali kako od memorije (pri čitanju iz memorije kao npr. pri čitanju instrukcije) tako i sa magistrale (pri upisu u memoriju kao npr. u ST instrukciji). Signal koji određuje vrstu memorijske operacije u trećem koraku ST instrukcije se postavlja na učitavanje, da bi se ostavilo vremena da se preko adresnih linija aktivira odgovarajuća memorijska lokacija prije nego što počne upis. U četvrtom koraku se ovaj signal prosto prebacuje na upis, čime se zadata vrijednost smješta u memoriju.

Drugi dio izvršavanja intrukcije skoka JMP je sličan onome što se dešava u prva dva koraka. Sadržaj registra PC treba naime ažurirati novom adresom naredne instrukcije, pri čemu je ofset u odnosu na tekuću vrijednost (kako je već pomenuto, ofset se zadaje u odnosu na inkrementirani sadržaj PC registra) zadat kao dio instrukcije. U tom cilju se u trećem koraku na magistralu A propušta sadržaj registra PC (zato što je ova magistrala već korišćena za prenos vrijednosti ovog registra do aritmetičko-logičke jedinice), a na magistralu B odgovarajući ofset koji se očitava iz IR registra. Ovo znači da vrijednost ofseta treba dodati multiplekseru koji kontroliše izlaz na B magistralu. Pri opisu kodiranja instrukcija je navedeno da se ofset zadaje kao 8-bitna vrijednost, tako da ovu vrijednost treba proširiti na 16-bitnu vrijednost jer je magistrala 16-bitna. Proširivanje se vrši prosto tako što se u viših 8 bitova upiše bit najveće težine ofseta (koji predstavlja bit znaka). U trećem koraku se takođe na aritmetičko-logičku jedinicu šalju isti signali kao u slučaju ADD instrukcije: signal koji aktivira ovu jedinicu i signal koji određuje da je operacija koju treba izvršiti sabiranje. Četvrti korak je kod JMP instrukcije identičan dijelu drugog koraka: izlaz aritmetičko-logičke jedinice se preko magistrale B upisuje u registar PC što će kao posljedicu imati da će biti prekinut sekvencijalni tok izvršavanja instrukcija odn. da će biti izvršen skok.

Analizom ostalih instrukcija na sličan način se određuje koje je još veze potrebno napraviti prema magistralama i od magistrala. Pokazuje se npr. da je registar MDR potrebno povezati na multipleksere na ulazima obje magistrale: u drugom koraku za sve instrukcije se preko magistrale A sadržaj ovog registra prebacuje u IR registar, dok je za instrukciju LD potrebno da se sadržaj ovog registra proslijedi na magistralu B pošto ga treba upisati u odgovarajući registar u skup registara, a upis u skup registara ide isključivo preko magistrale B. Upis u svaki registar koji ne pripada skupu registara se kontroliše odgovarajućim kontrolnim signalom. Već je pomenuto da upis u registar MDR mora ići kroz multiplekser pošto postoje dva izvora vrijednosti za ovaj registar, isto važi i za PC registar koji, osim što se ažurira u svakoj instrukciji (u instrukcijama skoka eventualno i po dvaput), treba postaviti na 0 u slučaju aktiviranja signala za resetovanje mikroprocesora `rst`.

Osim ulaza za signal takta i signal za resetovanje, svi ulazi u *datapath* modul su signali od kontrolne jedinice. Registri IR i PSW, odn. preciznije njihove

komponente, su vidljivi kao izlazi ovog modula jer su isti potrebni kontrolnoj jedinici - prvi registar naravno za određivanje koje kontrolne signale treba generisati, a drugi registar jer su vrijednosti odgovarajućih flegova potrebne za instrukcije uslovnog skoka. Obzirom da *datapath* modul praktično obuhvata i memorijsku jedinicu, izlazni port ovog modula je i signal koji predstavlja memorijsku adresu. Takođe je prisutan i port preko koga se razmjenjuju podaci sa memorijom i ovaj port je ulazno-izlazni. Ova dva porta će biti direktno vidljivi kao portovi mikroprocesora kao cjeline. Obzirom da se izvori signala za port preko koga se razmjenjuju podaci sa memorijom nalaze sa dvije različite strane, unutar i izvan mikroprocesora, sa unutrašnje strane je za dodjelu vrijednosti ovom portu korišćena tehnika sa vrijednošću visoke impedanse. Slijedi *Verilog* kod koji implementira *datapath* modul:

```
module datapath (clk, rst, mem_op, a_sel, b_sel, regfile_out0_sel,
                 regfile_out1_sel, regfile_in_ena, regfile_in_sel,
                 alu_ena, alu_op, pc_ena, pc_sel, ir_ena, psw_ena,
                 mar_ena, mdr_ena, mdr_sel, opcode, dst, src0, src1,
                 zero, carry, addr, data);
   input         clk, rst; /* Signal takta i signal reseta. */

   input         mem_op; /* Signal koji odredjuje kakav tip pristupa memoriji
                          (upis ili citanje) je u pitanju. */
   input [1:0]   a_sel, b_sel; /* Signali koji odredjuju sta se propusta
                                na magistrale. */
   input [3:0]   regfile_out0_sel, regfile_out1_sel; /* Selektorski
                                                      signali za izlaze
                                                      skupa registara. */
   input         regfile_in_ena; /* Kontrolni signal za upis u registar iz skupa
                                  registara. */
   input [3:0]   regfile_in_sel; /* Selektorski signal za izbor registra
                                  iz skupa registara u koga ce biti
                                  upisana vrijednost. */
   input         alu_ena; /* Signal za aktiviranje aritmeticko-logicke
                           jedinice. */
   input [2:0]   alu_op; /* Kod operacije za aritmeticko-logicku jedinicu. */
   input         pc_ena, pc_sel, ir_ena, psw_ena; /* Kontrolni i selektorski
                                                   signali za registre
                                                   PC, IR i PSW. */
   input         mar_ena, mdr_ena, mdr_sel; /* Kontrolni i selektorski signali
                                             za registre MAR i MDR. */
   output [3:0]  opcode, dst, src0, src1; /* Komponente tekuce
                                           instrukcije: kod operacije,
                                           adrese odredisnog i izvornih
                                           registara. */
   output        zero, carry; /* Vrijednosti Z i C flegova PSW registra. */
   output [15:0] addr; /* Memorijska adresa. */
   inout [15:0]  data; /* Podatak koji se razmjenjuje sa memorijom. */

   wire [15:0]   bus_a, bus_b; /* Magistrale mikroprocesora. */
   reg [15:0]    pc, ir; /* Registri PC i IR. */
   reg [1:0]     psw; /* Registar PSW. */
   reg [15:0]    mar, mdr; /* Registri MAR i MDR. */

   wire [15:0]   regfile_out0, regfile_out1; /* Izlazi skupa registara
                                              opste namjene. */
   wire [15:0]   alu_res; /* Rezultat izracunavanja u aritmeticko-logickoj
                           jedinici. */
```

```
  wire          alu_zero, alu_carry; /* Flegovi aritmeticko-logicke jedinice. */

  /* Selektuju se signali koji se prosledjuju na magistrale. Za
   magistralu A selektorski signali su sledeci:
     00 - na magistralu se dovodi odgovarajuci izlaz skupa registara
     01 - na magistralu se dovodi sadrzaj MDR registra
     10, 11 - na magistralu se dovodi sadrzaj PC registra
   Za magistralu B selektorski signali su sledeci:
     00 - na magistralu se dovodi odgovarajuci izlaz skupa registara
     01 - na magistralu se dovodi sadrzaj MDR registra
     10 - na magistralu se dovodi broj koji je sadrzan u instrukciji
     11 - na magistralu se dovodi rezultat aritmeticko-logicke jedinice */
  assign        bus_a = a_sel[1] ? pc : (a_sel[0] ? mdr : regfile_out0);
  assign        bus_b = b_sel[1] ? (b_sel[0] ? alu_res : {{ 8 { ir[7] } },
                                                          ir[7:0]}) :
                                   (b_sel[0] ?  mdr : regfile_out1);

  /* Za svaki od registara PC, IR, PSW, MAR i MDR kreira se proces koji
   kontrolise upis vrijednosti u registar. Upis u registre PC i MDR je
   kontrolisan multiplekserima, obzirom da vrijednosti koje bivaju
   upisane u ove registre dolaze od dva izvora. */
  always @(pc_ena or pc_sel or bus_b)
    if (pc_ena)
      pc <= pc_sel ? bus_b : 16'd0;
  always @(ir_ena or bus_a)
    if (ir_ena)
      ir <= bus_a;
  always @(psw_ena or alu_zero or alu_carry)
    if (psw_ena)
      psw = {alu_carry, alu_zero};
  always @(mar_ena or bus_a)
    if (mar_ena)
      mar <= bus_a;
  always @(mdr_ena or mdr_sel or bus_b or data)
    if (mdr_ena)
      mdr <= mdr_sel ? bus_b : data;

  /* Pojedinacne komponente registara IR i PSW se cine dostpunim na
   izlazu komponente uvijek kada se sadrzaj odgovarajucih registara
   promijeni. */
  assign      {opcode, dst, src0, src1} = ir;
  assign      {carry, zero} = psw;

  /* Registri MAR i MDR se takodje direktno povezuju sa odgovarajucim
   portovima komponente. Obzirom da je port preko koga se razmjenjuju
   podaci sa memorijom ulazno/izlazni, na njega se prosledjuje
   vrijednost MDR registra samo ukoliko je u pitanju upis u
   memoriju. */
  assign      addr = mar;
  assign      data = mem_op ? mdr : 16'bz;

  /* Instancira se skup registara opste namjene. */
  regfile _regfile (regfile_out0_sel, regfile_out1_sel, regfile_in_ena,
                    regfile_in_sel, bus_b, regfile_out0, regfile_out1);

  /* Instancira se aritmeticko-logicka jedinica. */
  alu _alu (alu_ena, bus_a, bus_b, alu_op, alu_res, alu_zero, alu_carry);
endmodule
```

4.6 Model kontrolne jedinice

Svi izlazni portovi kontrolne jedinice predstavljaju kontrolne signale koji se povezuju na odgovarajuće ulaze *datapath* jedinice. Ulazni signali u kontrolnu jedinicu su, pored signala takta i signala reseta, signali od *datapath* jedinice koji su potrebni kontrolnoj jedinici radi generisanja i sekvenciranja kontrolnih signala; u pitanju su signali koji predstavljaju dekodiranu instrukciju, kao i vrijednosti flegova iz PSW registra.

U suštini, kontrolna jedinica funkcioniše kao konačni automat sa 4 stanja koja odgovaraju koracima u izvršavanju instrukcije. Svaki put na pozitivnoj ivici signala takta generišu se kontrolni signali koji odgovaraju tekućem stanju automata, kao i stanju signala koje kontrolna jedinica očitava od *datapath* jedinice (prije svega kodu instrukcije). Takođe se istovremeno konačni automat prevodi u naredno stanje.

Slijedi *Verilog* kod koji implementira kontrolnu jedinicu:

```
/* Koraci pri izvrsavanju instrukcija. */
`define FETCH 2'b00
`define DECODE 2'b01
`define EXECUTE 2'b10
`define WRITE 2'b11

/* Kodovi koji odgovaraju pojedinacnim instrukcijama. */
`define ADD 4'b0000
`define INC 4'b0001
`define SUB 4'b0010
`define NOT 4'b0011
`define AND 4'b0100
`define OR 4'b0101
`define SLA 4'b0110
`define SRA 4'b0111
`define LD 4'b1000
`define ST 4'b1001
`define LDI 4'b1010
`define JMP 4'b1011
`define JZ 4'b1100
`define JB 4'b1101

/* Konstante koje predstavljaju aktiviranje odn. deaktiviranje signala. */
`define ENABLE 1'b1
`define DISABLE 1'b0

/* Operacije sa memorijom. */
`define MEM_READ 1'b0
`define MEM_WRITE 1'b1

/* Selektorski signali za magistralu A. */
`define BUS_A_SEL_REGFILE_OUT0 2'b00
`define BUS_A_SEL_MDR 2'b01
`define BUS_A_SEL_PC 2'b10

/* Selektorski signali za magistralu B. */
`define BUS_B_SEL_REGFILE_OUT1 2'b00
`define BUS_B_SEL_MDR 2'b01
```

```
`define BUS_B_SEL_IMM 2'b10
`define BUS_B_SEL_ALU_RES 2'b11

/* Selektorski signal za PC registar. */
`define PC_SEL_0 1'b0
`define PC_SEL_BUS_B 1'b1

/* Selektorski signal za MDR registar. */
`define MDR_SEL_DATA 1'b0
`define MDR_SEL_BUS_B 1'b1

module control (clk, rst, opcode, dst, src0, src1, zero, carry, mem_op,
                a_sel, b_sel, regfile_out0_sel, regfile_out1_sel,
                regfile_in_ena, regfile_in_sel, alu_ena, alu_op, pc_ena,
                pc_sel, ir_ena, psw_ena, mar_ena, mdr_ena, mdr_sel);
   input        clk, rst; /* Signal takta i signal reseta. */
   input [3:0]  opcode, dst, src0, src1; /* Komponente tekuce
                                   instrukcije: kod operacije,
                                   adrese odredisnog i izvornih
                                   registara. */
   input        zero, carry; /* Vrijednosti Z i C flegova PSW registra. */

   output       mem_op; /* Signal koji odredjuje kakav tip pristupa memoriji
                         (upis ili citanje) je u pitanju. */

   output [1:0] a_sel, b_sel; /* Signali koji odredjuju sta se propusta
                               na magistrale. */

   output [3:0] regfile_out0_sel, regfile_out1_sel; /* Selektorski
                                                     signali za izlaze
                                                     skupa registara. */
   output       regfile_in_ena; /* Kontrolni signal za upis u registar iz
                                 skupa registara. */
   output [3:0] regfile_in_sel; /* Selektorski signal za izbor registra
                                 iz skupa registara u koga ce biti
                                 upisana vrijednost. */
   output       alu_ena; /* Signal za aktiviranje aritmeticko-logicke
                          jedinice. */
   output [2:0] alu_op; /* Kod operacije za aritmeticko-logicku jedinicu. */
   output       pc_ena, pc_sel, ir_ena, psw_ena; /* Kontrolni i selektorski
                                                   signali za registre
                                                   PC, IR i PSW. */
   output       mar_ena, mdr_ena, mdr_sel; /* Kontrolni i selektorski signali
                                            za registre MAR i MDR. */

   /* Registarske deklaracije za izlazne portove. */
   reg          mem_op;
   reg [1:0]    a_sel, b_sel;
   reg [3:0]    regfile_out0_sel, regfile_out1_sel;
   reg          regfile_in_ena;
   reg [3:0]    regfile_in_sel;
   reg          alu_ena;
   reg [2:0]    alu_op;
   reg          pc_ena, pc_sel, ir_ena, psw_ena;
   reg          mar_ena, mdr_ena, mdr_sel;

   reg [1:0]    step; /* Korak pri izvrsavanju instrukcije odn. tekuce
                       stanje konacnog automata. */

   /* Generisanje kontrolnih signala se vrsi na pozitivnoj ivici signala
    takta. */
   always @(posedge clk)
```

```
if (rst)
  begin
     /* Ako je aktivan signal reseta, u PC registar se upisuje 0, a
      naredni korak se postavlja na prvi korak. */
     pc_ena <= `ENABLE;
     pc_sel <= `PC_SEL_0;
     step <= `FETCH;
  end
else
  /* Implementacija konacnog automata. */
  case (step)
    `FETCH:
      /* U prvom koraku se sadrzaj PC registra prebacuje u MAR
       registar i zadaje se ucitavanje iz memorije. Takodje se u
       aritmeticko-logickoj jedinici inkrementira sadrzaj PC
       registra. */
      begin
         a_sel <= `BUS_A_SEL_PC;
         alu_ena <= `ENABLE;
         alu_op <= 3'b001;
         mar_ena <= `ENABLE;
         mdr_ena <= `ENABLE;
         mdr_sel <= `MDR_SEL_DATA;
         mem_op <= `MEM_READ;
         step <= `DECODE;
      end

    `DECODE:
      /* U drugom koraku se tekuca instrukcija, koja je u
       medjuvremenu iz memorije dovucena u MDR registar, prebacuje u IR
       registar. Inkrementirana vrijednost PC registra se upisuje
       natrag u PC registar. */
      begin
         a_sel <= `BUS_A_SEL_MDR;
         b_sel <= `BUS_B_SEL_ALU_RES;
         ir_ena <= `ENABLE;
         pc_ena <= `ENABLE;
         pc_sel <= `PC_SEL_BUS_B;
         step <= `EXECUTE;
      end

    `EXECUTE:
      /* U trecem koraku kontrolni signali koji se generisu zavise
       od toga koja je instrukcija u pitanju. */
      case (opcode)
        `ADD, `SUB, `AND, `OR:
          /* U slucaju binarnih aritmeticko-logickih operacija,
           sadrzaj operanada se iz skupa registara opste namjene
           propusta na magistrale, a aritmeticko-logicka jedinica
           se aktivira i prosledjuje joj se odgovarajuci kod
           operacije. Takodje se aktivira selektorski signal za
           upis rezultata natrag u skup registara opste namjene;
           ovaj signal je vazno aktivirati odmah, da bi u narednom
           koraku upis isao odmah u zeljeni registar. */
          begin
             a_sel <= `BUS_A_SEL_REGFILE_OUT0;
             b_sel <= `BUS_B_SEL_REGFILE_OUT1;
             regfile_out0_sel <= src0;
             regfile_out1_sel <= src1;
             regfile_in_sel <= dst;
             psw_ena <= `ENABLE;
             alu_ena <= `ENABLE;
```

```
            alu_op <= opcode[2:0];
            step <= `WRITE;
         end

      `INC, `NOT, `SLA, `SRA:
         /* U slucaju unarnih aritmeticko-logickih operacija,
          operanad se iz skupa registara opste namjene
          propusta na magistralu A, a ostali kontrolni signali se
          generisu isto kao za binarne operacije.. */
         begin
            a_sel <= `BUS_A_SEL_REGFILE_OUT0;
            regfile_out0_sel <= src0;
            regfile_in_sel <= dst;
            psw_ena <= `ENABLE;
            alu_ena <= `ENABLE;
            alu_op <= opcode[2:0];
            step <= `WRITE;
         end

      `LD:
         /* U slucaju instrukcije za ucitavanje iz memorije,
          vrijednost registra opste namjene koji sadrzi memorijsku
          adresu sa koje se vrsi ucitavanje se preko magistrale A
          prebacuje u MAR registar. Za MDR registar se dozvoljava
          upis, a za izvor signala se bira port preko koga se vrsi
          transfer podataka sa memorijom; sadrzaj ovog registra se
          takodje odmah propusta na magistralu B, tako da ce
          podatak postati dostupan na ovoj magistrali cim bude
          ucitan iz memorije. Za memorijsku operaciju se naravno
          selektuje citanje iz memorije. */
         begin
            a_sel <= `BUS_A_SEL_REGFILE_OUT0;
            b_sel <= `BUS_B_SEL_MDR;
            regfile_out0_sel <= src0;
            regfile_in_sel <= dst;
            mar_ena <= `ENABLE;
            mdr_ena <= `ENABLE;
            mdr_sel <= `MDR_SEL_DATA;
            mem_op <= `MEM_READ;
            step <= `WRITE;
         end

      `ST:
         /* U slucaju instrukcije za upis u memoriju, memorijska
          adresa odn. podatak koga treba upisati se iz
          odgovarajucih registara opste namjene preko magistrala
          prebacuju respektivo u registre MAR odn. MDR. Za
          registar MDR se selektuje da ulazna vrijednost bude
          citana sa odgovarajuce magistrale. Za memorijsku
          operaciju se naravno selektuje upis u memoriju. */
         begin
            a_sel <= `BUS_A_SEL_REGFILE_OUT0;
            b_sel <= `BUS_B_SEL_REGFILE_OUT1;
            regfile_out0_sel <= dst;
            regfile_out1_sel <= src0;
            mar_ena <= `ENABLE;
            mdr_ena <= `ENABLE;
            mdr_sel <= `MDR_SEL_BUS_B;
            step <= `WRITE;
         end

      `LDI:
```

```
          /* U slucaju instrukcije za ucitavanje konstante u
           registar, na magistralu B se propusta konstanta, a
           istovremeno se na skupu registara opste namjene
           selektuje registar u koga ce biti upisana ista. */
          begin
             b_sel <= `BUS_B_SEL_IMM;
             regfile_in_sel <= dst;
             step <= `WRITE;
          end

        `JMP, `JZ, `JB:
          /* U slucaju instrukcija uslovnog skoka se prvo
           provjerava da li je ispunjen uslov skoka i ako nije
           odmah se prelazi na obradu naredne instrukcije. */
          if ((opcode == `JZ && !zero) || (opcode == `JB && !(!zero
                                                     && carry)))
            step <= `FETCH;
          else
            /* Ako do skoka treba da dodje, onda se sadrzaj PC
             registra propusta na jednu magistralu, a na drugu
             magistralu vrijedost ofseta, aktivira se
             aritmeticko-logicka jedinica i selektuje se operacija
             sabiranja na istoj. */
            begin
               a_sel <= `BUS_A_SEL_PC;
               b_sel <= `BUS_B_SEL_IMM;
               alu_ena <= `ENABLE;
               alu_op <= 3'b000;
               step <= `WRITE;
            end

        default:
          step <= `FETCH;
      endcase

    `WRITE:
      case (opcode)
        `ADD, `INC, `SUB, `NOT, `AND, `OR, `SLA, `SRA:
          /* U slucaju aritmeticko-logickih instrukcija, rezultat
           se sa izlaza aritmeticko-logicke jedinice preko
           magistrale B upisuje u odgovarajuci registar u skupu
           registara opste namjene. */
          begin
             b_sel <= `BUS_B_SEL_ALU_RES;
             regfile_in_ena <= `ENABLE;
             step <= `FETCH;
          end

        `LD:
          /* U slucaju ucitavanja iz memorije, podatak se iz MDR
           registra upisuje u odgovarajuci registar u skupu
           registara opste namjene. */
          begin
             regfile_in_ena <= `ENABLE;
             step <= `FETCH;
          end

        `ST:
          /* U slucaju upisa u memoriju, aktivira se upis. */
          begin
             mem_op <= `MEM_WRITE;
             step <= `FETCH;
```

```
                end

              `LDI:
                /* U slucaju upisa konstante u registar, aktivira se upis
                 u skup registara opste namjene. */
                begin
                   regfile_in_ena <= `ENABLE;
                   step <= `FETCH;
                end

              `JMP, `JZ, `JB:
                /* U slucaju skoka, vrijednost na izlazu
                 aritmeticko-logicke jedinice se preko magistrale B
                 upisuje u PC registar. */
                begin
                   b_sel <= `BUS_B_SEL_ALU_RES;
                   pc_ena <= `ENABLE;
                   pc_sel <= `PC_SEL_BUS_B;
                   step <= `FETCH;
                end

              default:
                step <= `FETCH;
            endcase
        endcase

    /* Na negativnoj ivici signala takta deaktiviraju se svi signali koji
     dozvoljavaju upis u registre opste namjene odn. posebne registre, a
     takodje se deaktivira i aritmeticko-logicka jedinica. */
    always @(negedge clk)
      begin
         regfile_in_ena <= `DISABLE;
         alu_ena <= `DISABLE;
         pc_ena <= `DISABLE;
         ir_ena <= `DISABLE;
         psw_ena <= `DISABLE;
         mar_ena <= `DISABLE;
         mdr_ena <= `DISABLE;
     end
endmodule
```

Mada ovaj modul ima najduži kod, isti je u principu dosta jednostavan. Na početku je deklarisan veliki broj makroa koji treba da pospješe čitljivost ostatka koda. Implementacija modula se praktično sastoji od dva procesa. Prvi, veliki, proces predstavlja konačni automat i u njemu se vrši generisanje kontrolnih signala i tranzicija iz stanja u stanje. U drugom procesu se na svakoj negativnoj ivici takta signali koji dozvoljavaju upis u registre deaktiviraju, takođe se deaktivira i aritmetičko-logička jedinica. Treba uočiti da se kao posljedica ovoga korisne aktivnosti odvijaju samo tokom pozitivnog dijela ciklusa takta. Postoje komplikovanije šeme generisanja kontrolnih signala koje omogućavaju da se ciklus takta bolje iskoristi.

U implementaciji konačnog automata treba samostalno provjeriti da generisanje kontrolnih signala, prije svega za instrukcije `ADD`, `ST` i `JMP` koje su detaljno razmatrane u prethodnom poglavlju, a onda i za sve ostale instrukcije, teče kako treba i da je finalni efekat zaista izvršavanje date instrukcije

sa željenom semantikom.

4.7 Model kompletnog mikroprocesora

Po konstrukciji *datapath* i kontrolne jedinice, implementacija mikroprocesora je jednostavna i sastoji se u prostom deklarisanju signala koji se razmjenjuju između ove dvije jedinice i instanciranja istih.

Ulazni portovi u mikroprocesor su signali takta i reseta; ovi signali se prosleđuju objema jedinicama. Izlazni signali su signali koji služe za komunikaciju sa memorijom: signal koji kontroliše koja je operacija sa memorijom u pitanju uzima se direktno sa kontrolne jedinice, dok se adresne linije i linije za podatke uzimaju sa *datapath* jedinice.

Verilog kod za mikrorprocesor prema tome ima oblik:

```
module risc16(clk, rst, mem_op, address, data);
   input         clk, rst; /* Signal takta i signal reseta. */
   output        mem_op; /* Signal koji odredjuje kakav tip pristupa memoriji
                          (upis ili citanje) je u pitanju. */
   output [15:0] address; /* Memorijska adresa. */
   inout [15:0]  data; /* Podatak koji se razmjenjuje sa memorijom. */

   wire [1:0]    a_sel, b_sel; /* Signali koji odredjuju sta se propusta na
                                  magistrale. */
   wire [3:0]    regfile_out0_sel, regfile_out1_sel; /* Selektorski signali
                                                        za izlaze skupa
                                                        registara. */
   wire          regfile_in_ena; /* Kontrolni signal za upis u registar iz skupa
                                    registara. */
   wire [3:0]    regfile_in_sel; /* Selektorski signal za izbor registra iz
                                    skupa registara u koga ce biti upisana
                                    vrijednost. */
   wire          alu_ena; /* Signal za aktiviranje aritmeticko-logicke
                             jedinice. */
   wire [2:0]    alu_op; /* Kod operacije za aritmeticko-logicku jedinicu. */
   wire          pc_ena, pc_sel, ir_ena, psw_ena; /* Kontrolni i selektorski
                                                     signali za registre
                                                     PC, IR i PSW. */
   wire          mar_ena, mdr_ena, mdr_sel; /* Kontrolni i selektorski signali za
                                               registre MAR i MDR. */
   wire [3:0]    opcode, dst, src0, src1; /* Komponente tekuce instrukcije:
                                             kod operacije, adrese
                                             odredisnog i izvornih registara. */
   wire          zero, carry; /* Vrijednosti Z i C flegova PSW registra. */

   /* Instanciranje datapath jedinice. */
   datapath _datapath(clk, rst, mem_op, a_sel, b_sel, regfile_out0_sel,
                      regfile_out1_sel, regfile_in_ena, regfile_in_sel,
                      alu_ena, alu_op, pc_ena, pc_sel, ir_ena, psw_ena,
                      mar_ena, mdr_ena, mdr_sel, opcode, dst, src0,
                      src1, zero, carry, address, data);

   /* Instanciranje kontrolne jedinice. */
   control _control (clk, rst, opcode, dst, src0, src1, zero, carry,
                     mem_op, a_sel, b_sel, regfile_out0_sel,
```

```
                     regfile_out1_sel, regfile_in_ena, regfile_in_sel,
                     alu_ena, alu_op, pc_ena, pc_sel, ir_ena, psw_ena,
                     mar_ena,mdr_ena, mdr_sel);
endmodule
```

4.8 Simulacija dizajna

U cilju simulacije opisanog modela mikroprocesora potrebno je još minimalno obezbijediti generator signala takta i memoriju sa odgovarajućim programom. Ove komponente se mogu konstruisati kao posebni moduli, ali je ovdje odabrano da budu implementirane u okviru modula stimulansa. Tako je modul stimulansa oblika:

```
/* Podesavanje vremenskih jedinica. */
`timescale 1ns/1ns

/* Ime fajla sa sadrzajem memorije. */
`define DUMPFILE "core.dump"

module stimulus;
   /* Ulazni signali. */
   reg         clk, rst;

   /* Izlazni signali. */
   wire        mem_op;
   wire [15:0] addr;

   /* Ulazno-izlazni signali. */
   tri [15:0] data;

   /* Memorija. */
   reg [15:0]  mem[0:65535];

   /* Instanciranje komponente. */
   risc16 _risc16 (clk, rst, mem_op, addr, data);

   /* Generisanje signala takta. */
   initial
     clk <= 1'b0;
   always
     #15 clk <= ~clk;

   /* Resetovanje sistema. */
   initial
     begin
        rst <= 1'b1;
        #20 rst <= 1'b0;
     end

   /* Ucitavanje sadrzaja memorije iz fajla. */
   initial
     $readmemh (`DUMPFILE, mem);

   /* Ucitavanje podatka iz memorije. */
   assign data = mem_op ? 16'bz : mem[addr];

   /* Upis podatka u memoriju. */
```

```
    always @(mem_op or addr or data)
      if (mem_op)
        begin
          mem[addr] <= data;
          $display ($time, ": mem[%x] <- %x", addr, data);
        end
endmodule
```

Memorija je implementirana prosto kao polje od 2^{16} riječi dužine 16 bitova. Na početku modula se instancira mikroprocesor, a potom se generišu signal takta i inicijalni signal za resetovanje sistema na početku simulacije. Treba uočiti da je, obzirom da je mikroprocesor dizajniran da prihvata samo tzv. sinhrono resetovanje[3], dužina signala resetovanja dovoljna da obuhvati pozitivnu ivicu signala takta. Na početku simulacije vrši se i inicijalizacija memorije iz datog fajla. Za ovo je iskorišćena sistemska direktiva `$readmemh` koja iz fajla čije je ime prvi argument učitava sadržaj memorije koja je data kao drugi argument. U realnom sistemu, inicijalizacija memorije teče na nešto drugačiji način: jedan mali dio memorije je obično implementiran kao fiksna *ROM* memorija koja sadrži inicijalni kod za učitavanje operativnog sistema sa diska u ostatak memorije. Opisana *Verilog* sistemska direktiva je ipak veoma korisna za inicijalizaciju memorije u situacijama kada treba simulirati rad mikroprocesora jer obezbjeđuje da se simulacija mikroprocesora vrši bez da je neophodno da se dizajnira svaki detalj kompletnog računarskog sistema, a takođe omogućava i da se mikroprocesor testira različitim programima prosto tako što se ti programi upisuju u fajl koji će biti učitan kao sadržaj memorije. Na kraju modula stimulansa implementirani su čitanje odn. upis iz memorije preko `data` porta mikroprocesora. Za čitanje je, isto kao kod implementacije *datapath* jedinice, iskorišćena tehnika sa vrijednosti visoke impedanse, dok je upis u memoriju implementiran standardno preko odgovarajućeg procesa `always` tipa. Treba uočiti kako je `data` signal deklarisan kao `tri` - u pitanju je tip signala koji je ekvivalentan `wire` tipu, ali čijim se korišćenjem eksplicitno naglašava da mu se može dodijeliti vrijednost visoke impedanse. Pri svakom upisu u memoriju ispisuju se na standardni izlaz odgovarajuća adresa i podatak - ideja je da programi kojima će biti testiran mikroprocesor upisuju rezultate u memoriju, te da se ovim ispisom može brzo utvrditi da li ti programi rade kako treba.

Za testiranje mikroprocesora treba obezbijediti odgovarajuće programe i smještati ih u fajl iz koga će biti učitavana memorija. Potom se pokreće simulacija i posmatraju rezultati upisa u memoriju. Prema potrebi se mogu dodati direktive za snimanje vrijednosti signala u fajl i kasnije analizirati isti kao što je to rađeno za elementarne komponente; kod složenijih komponenti kakav je mikroprocesor ovakav metod analize rada komponente je manje upotrebljiv obzirom na veliki broj signala.

[3] tj. da bi signal reseta imao efekta isti mora da bude aktivan na određenoj ivici signala takta

Slijedi primjer programa koji koristi mikroprocesor da sabere 2 i 2:

```
@000
b003

@002
0002
0002
a002
8100
a003
8200
0312
a001
9030
b0f8
```

Obzirom da je korišćena `$readmemh` direktiva, očekuje se da sadržaj memorije bude dat u heksadecimalnom formatu, pri čemu svaki red fajla odgovara jednoj memorijskoj riječi. Postoji i `$readmemb` direktiva, kod koje se sadržaj memorije zadaje u binarnom formatu. Prazni redovi u ovom fajlu se ignorišu, a dozvoljeno je specificirati i memorijske adrese (za slučaj da se ne počinje od adrese 0 ili da se neki opseg adresa preskače) tako što se navede `@` i onda adresa. Na taj način, u gornjem fajlu je zadato da se na adresi 0 nalazi podatak `b003`, a onda počev od adrese 2 pa do adrese 11 vrijednosti date u završnih 10 redova. Memorijske lokacije koje nisu inicijalizovane imaju nepoznatu vrijednost u svim svojim bitovima.

Da bi se razumio gornji program, treba konsultovati kodove instrukcija. Na memorijskoj lokaciji 0 upisana je `JMP` instrukcija sa ofsetom skoka 3. Obzirom da se ofset računa ne u odnosu na tekuću instrukciju, već u odnosu na narednu instrukciju, to znači da će skok zapravo završiti na memorijskoj lokaciji 4. Lokacija sa adresom 1 je ostavljena prazna i u nju će biti upisan rezultat sabiranja. Lokacije 2 i 3 čuvaju vrijednosti koje se sabiraju, odn. u obje ove lokacije je upisan broj 2. Pravi program zapravo počinje od adrese 4 i mogao bi se u nekom hipotetičkom asembleru datog mikroprocesora zapisati u obliku:

```
LDI R0, 2
LD R1, (R0)
LDI R0, 3
LD R2, (R0)
ADD R3, R1, R2
LDI R0, 1
ST (R0), R3
JMP -8
```

Dakle, u registre `R1` i `R2` se učitavaju sabirci sa memorijskih adresa 2 odn. 3, potom se izračunava zbir i smješta u registar `R3`, da bi se iz istog potom kopirao i u memoriju, na adresu 1. Program se potom bezuslovnim skokom

vraća na instrukciju na memorijskoj adresi 4, odn. izračunavanje počinje ispočetka. Ono što bude odštampano tokom simulacije je oblika (ispis je mnogo duži, zapravo ne prestaje dok se nasilno ne prekine rad simulatora):

```
 975: mem[0001] <- xxxx
 975: mem[0001] <- 0004
1935: mem[0001] <- 0004
2895: mem[0001] <- 0004
3855: mem[0001] <- 0004
```

Dakle, vidi se da svaki put kada se završi sa računanjem, u memorijsku lokaciju sa adresom 1 biva upisan rezultat odn. broj 4.

Prikazani program mora da se stalno vrti u petlji obzirom da nije podržana instrukcija za zaustavljanje procesora. Treba uočiti da zapravo i u kompletnom, složenom sistemu imamo operativni sistem kao program slične strukture. Dalje, najveći broj mikroprocesora se zapravo koristi unutar mikrokontrolerskih sklopova koji ni nemaju operativni sistem već se upravo ovako kako je ovdje prikazano na njima stalno u petlji vrti samo jedan program[4].

Prvi naredni korak u dizajnu sistema baziranog na datom mikroprocesoru bi svakako bila konstrukcija asemblera koji bi omogućio da se izbjegne korišćenje mašinskog jezika za pisanje programa; potom dolazi pisanje prevodilaca za više programske jezike, eventualno pisanje operativnog sistema itd. Ove aktivnosti međutim više ne spadaju u domen hardverskog dizajna; ono na čemu se u tom pogledu dalje radi jesu ostale komponente sistema (napredniji model memorije, sistemska magistrala, interfejsi prema periferijskim uređajima itd.). Prikazani model mikroprocesora ostaje međutim, bez obzira na jednostavnost, potpun i kompletan model mikroprocesorskog sistema sa svom potrebnom osnovnom funkcionalnošću.

4.9 Sinteza dizajna

Po kompletiranju nekog dizajna uvijek je interesantno kako će proći sinteza. U datom primjeru za sintezu je opet korišćen *Quartus II* softver. Bilo je neophodno napraviti dvije izmjene da bi *Quartus II* prihvatio dizajn. U modulu koji implementira registre opšte namjene *Quartus II* nije prihvatao da za indeksiranje polja registara koristi odgovarajuće selektorske signale, već je traženo da kao indeksi stoje konstante. Ovo je prevaziđeno tako što su odgovarajuće naredbe dodjele razbijene na `case` konstrukcije sa po 16

[4]npr. kod mikrokontrolera koji kontroliše klima-uređaj ovaj program prvo očitava temperaturu, potom provjerava da li je potrebno uključiti ili isključiti uređaj ili eventualno podesiti njegov režim rada (uključiti ili isključiti grijače, povećati ili smanjiti brzinu ventilatora i sl.), eventualno obavlja te operacije i onda se opet vraća na početak, na mjerenje temperature

različitih dodjela, odn. praktično eksplicitno je predstavljen multiplekser koji je prvobitnom naredbom dodjele zapravo bio impliciran. Drugi problem je bio što u kontrolnoj jedinici nije prihvatano da se istim signalima dodjeljuje vrijednost u dva procesa (u pitanju su svi signali kojima se mijenja vrijednost u drugom procesu). Zato su ova dva procesa spojena u jedan koji biva aktiviran signalom takta i koji sadrži dva početna procesa kao grane `if` naredbe čiji je uslov vrijednost signala takta: kada je signal takta nakon promjene na visokom nivou izvršava se kod iz procesa koji implementira konačni automat, a kada je signal takta nakon promjene na niskom nivou izvršava se drugi proces. Unesene izmjene ne mijenjaju ništa u ponašanju mikroprocesora. Ovakve izmjene je inače često potrebno izvršiti da bi dati alat za sintezu prihvatio opis dizajna na *Verilog*-u. Alati za sintezu naime još uvijek nisu toliko usavršeni da mogu da prihvataju sve *Verilog* konstrukcije. Problematične konstrukcije obično spadaju u domen *behavioral* modelovanja. Tehnika modelovanja koja se po složenosti nalazi između *dataflow* i *behavioral* modelovanja težeći drugom ali pribjegavajući prvom kada alati za sintezu ne prihvataju neke konstrukcije se označava terminom *register transfer level* (*RTL*) modelovanje. Može se reći da se ovakva tehnika modelovanja danas najčešće koristi.

Po unesenim opisanim izmjenama, *Quartus II* je uspješno sintetizovao model mikroprocesora na osnovu datog *Verilog* koda. Na *Altera*-inom *EP20K60EBC356-1* čipu model mikroprocesora je zauzeo 47% površine i 17% pinova i maksimalna moguća učestanost je bila reda veličine 1.5MHz. Sinteza je na relativno jakom računaru obavljena za nešto manje od 4 minuta.

4.10 Napredne konstrukcije u *Verilog*-u

Pomenuto je da osnovne jezičke konstrukcije za tehniku *behavioral* modelovanja u *Verilog*-u predstavljaju naredba proceduralne dodjele i naredbe kontrole toka. Od ovih drugih objašnjene su naredbe grananja; preostaje da se nešto kaže o petljama.

Postoje 4 naredbe u *Verilog*-u kojima se implementiraju petlje. To su `while` i `for` naredbe sa potpuno istom sintaksom i semantikom kao na *C*-u, te `repeat` i `forever` naredbe. Naredba `repeat` se koristi kada neka petlja treba da se izvrši tačno određeni broj puta. Ova naredba je ekvivalentna brojačkoj `for` petlji kod koje se brojač ne koristi u tijelu petlje. Naredba `forever` se koristi za beskrajnu petlju. Tako bi se recimo signal takta mogao generisati i samo jednim procesom `initial` tipa na sledeći način:

```
initial
  begin
     clk <= 1'b0;
     forever #15 clk <= ~clk;
  end
```

Svakom bloku naredbi u *Verilog*-u može biti pridruženo ime tako što se iza `begin` naredbe kojom počinje blok stavi dvotačka i zatim ime bloka. Izvršavanje naredbi u bloku se može zaustaviti pomoću ključne riječi `disable` iza koje se navodi ime bloka, tako da je to način da se prekine beskrajna petlja.

Sinteza petlji se razlikuje po tome da li su u pitanju brojačke petlje ili petlje generalnog tipa. U oba slučaja se tijelo petlje implementira u vidu posebnog sklopa, s tim što se u prvom slučaju završetak rada kontroliše pomoću brojača, a u drugom slučaju pomoću kontrolnog signala koji odgovara uslovu petlje.

Verilog takođe podržava grupisanje naredbi koje spadaju u domen *behavioral* modelovanja u potprograme. Postoje dva tipa potprograma i to funkcije i *task*-ovi, između kojih je razlika otprilike kao između funkcija i procedura u *Fortran*-u. Funkcije naime vraćaju tačno jednu vrijednost i moraju imati bar jedan argument. Svi argumenti funkcija su ulazni, tj. funkcija ne može vraćati vrijednost preko argumenata. Funkcija takođe ne može sadržati nikakve naredbe koje podrazumijevaju kašnjenje, odn. praktično mogu služiti samo za implementaciju kombinatorne logike. Funkcije mogu pozivati druge funkcije, ali ne i *task*-ove. *Task*-ovi ne vraćaju vrijednost i mogu imati proizvoljan broj argumenata koji opet mogu biti ulazni, izlazni odn. ulazno-izlazni (tj. *task*-ovi mogu vraćati jednu ili više vrijednosti preko svojih argumenata). Pored ovoga, *task*-ovi nemaju nijedno od ograničenja koje važe za funkcije. Obje konstrukcije se implementiraju kao posebni logički sklopovi sa portovima koji odgovaraju argumentima - funkcije i *task*-ovi služe samo radi lakše dekompozicije dizajna i nemaju posebno značenje za sintezu.

Treba reći da pored naredbi koje su pojašnjene u ovom tekstu *Verilog* podržava i razne druge konstrukcije koje olakšavaju napredne tehnike modelovanja. Tako npr. *Verilog* podržava i modelovanje na nivou tranzistorskih tehnologija, zatim model jačine signala za razrešavanje konflikta kada u neku tačku dolaze signali od dva izvora, potom dopunjavanje jezika sopstvenim logičkim primitivama, kao i puni interfejs za pristup internom stanju simulatora. Ove konstrukcije zbog ograničenog prostora nisu mogle biti predstavljene u ovom tekstu, pa se čitalac koji je zainteresovan za iste upućuje na ranije pomenutu literaturu ([6]).

Dio II

IA-32 arhitektura

Poglavlje 5

Uvod u *IA-32* arhitekturu

Intel-ova *IA-32* arhitektura vuče korijene od *Intel*-ovog procesora *8086* koji je ugrađivan u prve *PC* računare. Punu fizionomiju međutim ova arhitektura dobija tek sa *80386* procesorom kao prvim 32-bitnim procesorom iz ove familije procesora. Ova arhitektura je potom zadržana na svim kasnijim procesorima, uključujući *80486*, *Pentium*, *PentiumPro*, *Pentium MMX*, *Pentium II*, *Pentium III* i *Pentium IV* procesore. Novije generacije su imale različitu organizaciju (odn. u *Intel*-ovoj terminologiji mikroarhitekturu), koja je zajedno sa povećanjem radnog takta doprinosila povećanju performansi, ali je arhitektura u smislu skupa registara i skupa instrukcija uz određene ekstenzije ostajala ista. Detalji organizacije pojedinačnih generacija procesora se mogu naći u [8]. U ovom tekstu će uglavnom biti razmatran osnovni skup instrukcija koga podržavaju svi procesori koji implementiraju *IA-32* arhitekturu; kada bude govora o nekim instrukcijama koje su specifične za jednu ili više generacija procesora, to će biti i naglašeno. Programi koji budu prezentirani u tekstu će podrazumijevati izvršavanje na *Pentium II* ili novijim procesorima, obzirom da su iste generacije danas najraširenije. Podrška za pojedinačne generacije procesora je inače precizno navedena za svaku instrukciju u odgovarajućem referentnom materijalu ([9]).

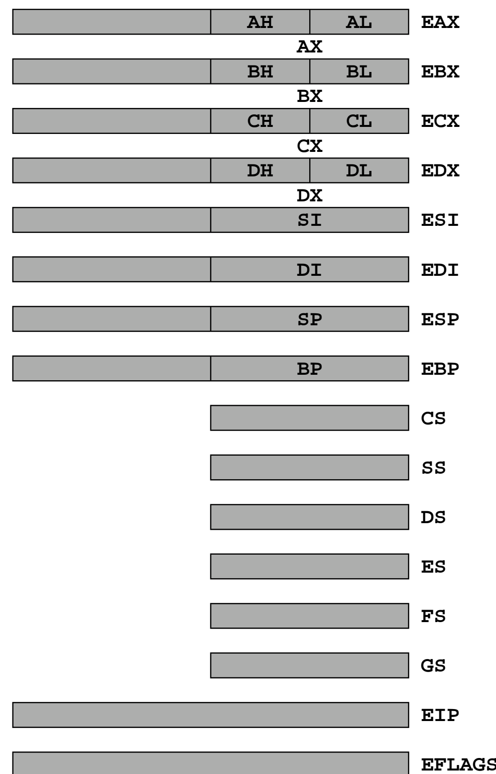
IA-32 arhitektura, kao i većina modernih arhitektura, podržava izvršavanje u tzv. sistemskom i aplikativnom režimu; u prvom režimu se izvršava operativni sistem, a u drugom obični programi. Ovaj mehanizam je u *Intel*-ovoj terminologiji implementiran preko tzv. nivoa privilegija. *IA-32* arhitektura podržava 4 nivoa označena sa *level 3*, *level 2*, *level 1* i *level 0*. Svi registri i instrukcije procesora su dostupni samo ukoliko se neki kod izvršava u nivou 0, u ostalim nivoima pojedini registri i instrukcije nisu dostupni[1]. Operativni sistem može po volji koristiti nivoe privilegija da implementira različite režime izvršavanja. Tako npr. *UNIX* tradicionalno implementira samo pomenuta

[1] ovakvi registri i instrukcije se označavaju terminom “privilegovani”

dva režima izvršavanja za koje se obično koriste nivoi 0 i 3; inicijalizacija sistema se obavlja u nivou 0, da bi kernel nakon izvršene inicijalizacije pokretao konzole u nivou 3 tako da se onda dalje i svi korisnički programi izvršavaju u nivou 3. U smislu režima izvršavanja može se napraviti razlika između sistemskog odn. aplikativnog asemblerskog programiranja; ukoliko se razmatra samo asemblersko aplikativno programiranje odn. pisanje koda koji će se izvršavati u korisničkom režimu, nije se neophodno upoznavati sa privilegovanim registrima odn. instrukcijama. Obzirom da će u ovom tekstu biti uglavnom razmatrano upravo aplikativno asemblersko programiranje, većina privilegovanih registara i instrukcija *IA-32* arhitekture zaista neće biti ni pomenuta; u vezi ove materije može se po potrebi konsultovati odgovarajuća literatura ([10]).

5.1 Skup registara

Skup osnovnih registara *IA-32* arhitekture je dat na slici 5.1.



Slika 5.1: Osnovni registri *IA-32* arhitekture.

Prvu grupu registara čine tzv. registri opšte namjene - u pitanju je 8 registara dužine 32 bita koji se označavaju imenima `EAX`, `EBX`, `ECX`, `EDX`, `ESI`,

EDI, ESP i EBP. Ovi registri se koriste za čuvanje operanada aritmetičkih i logičkih operacija, kao i pri izračunavanju memorijskih adresa. Nižih 16 bita ovih registara mogu biti individualno adresirani imenima AX, BX, CX, DX, SI, DI, SP i BP. Takođe, za prva 4 registra se svakom bajtu od tih nižih 16 bita može nezavisno pristupati imenima AH odn. AL, BH odn. BL, CH odn. CL i DH odn. DL. Mada se ovi registri označavaju terminom registri opšte namjene, zapravo svaki od njih ima za neke instrukcije specifičnu namjenu, što je sumirano tabelom 5.1. Više o ovome biće riječi kasnije, kada se budu razmatrale pojedinačne instrukcije.

Registar	Implicitna upotreba
EAX	akumulator pri aritmetičko-logičkim operacijama
EBX	pointer za pristupe memoriji
ECX	brojač u petljama
EDX	pointer za ulazno-izlazne operacije
ESI	pointer za operacije sa stringovima
EDI	pointer za operacije sa stringovima
ESP	pointer za operacije sa stekom
EBP	pointer za operacije sa stekom

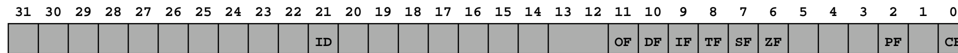
Tabela 5.1: Korišćenje registara opšte namjene.

Sledeća grupa registara su tzv. segmentni registri. U pitanju je 6 registara dužine 16 bita koji su označeni imenima CS, SS, DS, ES, FS i GS. *IA-32* arhitektura podržava segmentnu i linearnu organizaciju pristupa memoriji[2]; slično kao za nivoe privilegija, tako i ovdje operativni sistem bira način na koji će ogranizovati pristup memoriji. Segmentni registri služe za podršku segmentnoj organizaciji pristupa memorije; opet po tradiciji *UNIX* koristi linearnu organizaciju pristupa memoriji tako da u asemblerskom programiranju na *UNIX*-u nema nikakve potrebe da se pristupa ovim registrima.

Dva preostala registra su tzv. *instruction pointer* registar EIP i statusni registar EFLAGS. U pitanju su 32-bitni registri; prvi registar čuva adresu naredne instrukcije odn. odgovara *program counter* registru iz prvog dijela ovog teksta, dok drugi registar sadrži vrijednosti raznih flegova i odgovara *processor status word* registru iz prvog dijela. Ovim registrima se ne može pristupiti direktno, već se njihov sadržaj mijenja izvršavanjem određenih instrukcija: vrijednost EIP registra se npr. mijenja instrukcijama skoka, dok se vrijednosti flegova u EFLAGS registru mijenjaju po izvršenju svake aritmetičke ili logičke instrukcije. Značajniji flegovi EFLAGS registra su prikazani na slici 5.2.

Sa CF je označen *carry* fleg koji se postavlja ako je pri sabiranju odn. oduzimanju tretirajući operande kao neoznačene brojeve bilo prenosa odn. poza-

[2]za oba tipa pristupa je takođe podržana stranična organizacija virtuelne memorije



Slika 5.2: Pojedinačni flegovi `EFLAGS` registra.

jmice. Sa `PF` je označen *parity* fleg koji se postavlja ako bajt najmanje težine rezultata ima paran broj bitova jednak 1. Dalje postoji *zero* fleg (`ZF`) koji se postavlja ako je rezultat jednak 0, te *sign* fleg (`SF`) koji se postavlja na bit znaka rezultata. Potom nailazi *overflow* fleg (`OF`) koji ima sličnu namjenu kao *carry* fleg, ali za operande koji su označeni brojevi. Svi dosad nabrojani flegovi spadaju u statusne flegove; registar `EFLAGS` sadrži međutim i kontrolne flegove koji se mogu postaviti u cilju promjene nekog aspekta ponašanja procesora. Tako postoji *interrupt* fleg (`IF`) koji određuje da li procesor prihvata prekide koji se mogu isključiti, potom *direction* fleg (`DF`) kojim se određuje da li se stringovi obrađuju unaprijed ili unatrag ili recimo *trap* fleg (`TF`) čijim se postavljanjem generiše prekid nakon svake instrukcije. Na slici je prikazan i *identification* fleg (`ID`) koji takođe spada u grupu statusnih flegova. Mogućnost promjene vrijednosti ovog flega određuje da li procesor podržava `CPUID` instrukciju koja vraća informacije o tipu procesora.

Obzirom da je u vezi sa nekim flegovima pomenut termin prekida, treba ga i pojasniti. Prekidi (*interrupts*) predstavljaju mehanizam kojim se informiše procesor o nekim događajima koji zahtijevaju trenutnu obradu. Procesor po prijemu prekida privremeno zaustavlja izvršavanje tekućeg programa da bi se izvršila tzv. prekidna rutina koja obrađuje događaj koji je u pitanju. Procesor se po završetku prekidne rutine vraća izvršavanju tekućeg programa. Prekidi mogu biti generisani od strane samog procesora da signaliziraju neke neregularne situacije nastale tokom izvršavanja programa, npr. dijeljenje nulom, nevalidan kod instrukcije (obično nisu svi kodovi instrukcija zauzeti) itd. - ovaj tip prekida se obično označava terminom *exception*. Prekidi dalje mogu dolaziti od spoljašnjih uređaja. Procesori *IA-32* arhitekture imaju dva pina, `NMI` i `INTR` kojim ulazno-izlazni uređaji mogu signalizirati da im je potrebna neka obrada od strane procesora (npr. tastatura da je pritisnut neki taster, te da procesor treba da pokupi kod tastera). Procesor ignoriše signal prekida koji dolazi preko `INTR` pina ako je postavljen `IF` fleg u `EFLAGS` registru. Ovaj fleg se zato obično postavlja za vrijeme nekih važnih operacija za koje je bitno da se brzo završe (često za vrijeme izvršavanja prekidnih rutina visokog prioriteta), a na `INTR` pin se vezuju ulazno-izlazni uređaji nižeg prioriteta. Signal prekida koji dolazi preko `NMI` (*non-maskable interrupt*) pina se ne može ignorisati. Poslednji tip prekida su prekidi koji su izazvani posebnim instrukcijama; ovi prekidi se označavaju terminom softverski prekidi. Ako istovremeno dođe do više prekida, onda se prekidi izršavaju po redosledu prioriteta, pri čemu je prioritet određen brojem prekida i utoliko je viši ukoliko je broj prekida niži.

5.2 Adresiranje operanada

IA-32 instrukcije mogu imati 0, 1 ili više operanada. Operand može biti zadat direktno u instrukciji ili se može nalaziti u registru, memoriji ili ulazno-izlaznom portu. Osim memorijskih, svi ostali tipovi operanada se adresiraju direktno, odn. recimo navođenjem imena registra za operand koji se nalazi u registru ili navođenjem broja ulazno-izlaznog porta za operand koji se nalazi u portu.

IA-32 arhitektura omogućava pristup memoriji na nivou pojedinačnih bajtova. Ako se koristi linearni režim pristupa memoriji, svakom programu je na raspolaganju 4GB (odn. 2^{32} bajtova) memorije. Adresa memorijskih operanada se može sastojati od više komponenti. Ove komponente se redom označavaju kao *base*, *index*, *scale* i *displacement*; ako ih respektivno označimo sa b, i, s i d onda se adresa memorijskog operanda izračunava kao $b+i\cdot s+d$ (pritom, bilo koji od sabiraka može biti izostavljen, takođe i *scale* komponenta). Sa *displacement* se označava broj koji je direktno naveden u instrukciji, baš kao kada se sam operand direktno zadaje. Vrijednosti *base* i *index* predstavljaju zapravo neki od registara opšte namjene (pri čemu postoji ograničenje da se registar `ESP` ne može koristiti kao *index*), dok vrijednost *scale* može biti samo broj 2, 4 ili 8.

Kombinovanjem različitih komponenti mogu se dobiti načini adresiranja memorijskih operanada koji su pogodni za specifične primjene. Tako se npr. korišćenjem *displacement*-a za adresu početka polja, a *index*-a za indeks tekućeg elementa može vrlo efikasno pristupati elementima polja ako je njihova dužina jednaka 1, 2, 4 ili 8 bajtova.

Mada opisani način memorijskog adresiranja pruža veliku fleksibilnost, isti ima i svoju cijenu. Ovakvim načinom adresiranja se naime znatno komplikuje kodiranje instrukcija, a takođe obavljanje računskih operacija radi određivanja efektivne adrese otežava preklapanje izvršavanja instrukcija koje u novijim generacijama procesora predstavlja osnovni način ubrzavanja rada. Takođe, komplikovana memorijska adresiranja mogu smanjiti čitljivost asemblerskog koda. Zbog ovih razloga će u ovom tekstu biti dosljedno primjenjivani samo jednostavni tipovi memorijskih adresiranja gdje će biti korišćen ili samo *displacement* ili samo *base* odn. gdje će efektivna adresa ili biti data direktno ili će biti sadržana u nekom od registara opšte namjene.

5.3 Tipovi podataka

Instrukcije *IA-32* arhitekture uzimaju za operande vrijednosti dužine 1, 2, 4, 8 ili 16 bajtova. U *IA-32* terminologiji ove vrijednosti se respektivno označavaju kao *bytes*, *words*, *doublewords*, *quadwords* i *double quadwords*. Treba uočiti da je termin *word* malo nedosljedno korišćen - obično se ovim

terminom u arhitekturi računara označava osnovni tip podataka sa kojim barata procesor, a koji je opet određen širinom registara procesora. Kod *IA-32* arhitekture su svi procesori koji je implementiraju 32-bitni, ali je termin *word* rezervisan za 16-bitni podatak; razlog ovoj nedoslednosti je što arhitektura vuče korijene od 16-bitne arhitekture procesora *8086*.

IA-32 arhitektura je predstavnik grupe tzv. *little-endian* arhitektura. Ovu grupu arhitektura karakteriše da višebajtni podatak smještaju u memoriju tako da na najnižu adresu ide bajt najmanje težine[3]. O ovome je važno voditi računa kada se neki podaci prenose sa *IA-32* arhitekture na neku drugu arhitekturu ili obrnuto.

IA-32 arhitektura ne zahtijeva da podaci u memoriji budu "uravnati", odn. recimo *word* podatak ne mora u memoriji biti smješten na lokaciji djeljivoj sa 2. Ipak, pristup memoriji se uvijek vrši uravnato na adresama djeljivim sa 4 tako da je zbog performansi preporučljivo držati podatke dužine veće ili jednake od 4 uravnatim. Naime, ako recimo *doubleword* podatak nije uravnat na adresu djeljivu sa 4, tada će za njegovo očitavanje biti potrebna dva pristupa memoriji. Sa druge strane, da bi *word* podatak bio učitan u jednom pristupu memoriji, isti ne mora biti na parnoj adresi, ali zato mora biti kompletno sadržan unutar *doubleword*-a koji počinje na adresi djeljivoj sa 4.

Osim po dužini, procesor ne pravi nikakvu drugu razliku među podacima kojima barata; svaki od njih se tretira kao prosti niz bitova. Interpretaciju ovim nizovima bitova daju instrukcije koje se nad njima primjenjuju. U tom smislu, može se reći da su tipovi podataka kojima procesor manipuliše sledeći:

- neoznačeni cijeli brojevi
- označeni cijeli brojevi
- realni brojevi
- pointeri
- polja bitova
- stringovi
- tzv. *SIMD*[4] tipovi podataka
- *BCD*[5] brojevi

[3]grupa arhitektura sa suprotnom konvencijom, tj. arhitekture kod kojih na nižu adresu u memoriji ide bajt više težine, se označavaju terminom *big-endian*

[4]*single instruction multiple data*

[5]*binary coded decimals*

Neoznačeni cijeli brojevi su prosto neoznačeni binarni brojevi sadržani u datom *byte*, *word*, *doubleword* ili *quadword* podatku. Za označene cijele brojeve važi slično, ali se u slučaju kada je bit najveće težine jednak 1, ostatak bitova interpretira kao negativan broj u drugom komplementu.

Realni brojevi će biti detaljnije razmatrani u poglavlju o instrukcijama za rad sa realnim brojevima.

Pointeri predstavljaju memorijske adrese i u linearnom režimu pristupa memoriji u pitanju su prosto redni brojevi bajtova u memoriji.

Polje bitova predstavlja kontinualan niz bitova dužine do 32 bita. Ovaj niz može početi na bilo kom bitu bilo kog bajta u memoriji.

String predstavlja kontinualan niz bitova, bajtova, *word*-ova ili *double word*-ova dužine do $2^{32} - 1$. Ovaj niz takođe može početi na bilo kom bitu bilo kog bajta u memoriji u slučaju niza bitova, odn. na bilo kom bajtu u memoriji u slučaju ostalih nizova.

SIMD tipovi podataka su uvedeni u novijim generacijama procesora *IA-32* arhitekture i o njima će biti više riječi u poglavljima o odgovarajućim (*MMX*, *SSE* odn. *SSE2*) instrukcijama.

BCD podaci podrazumijevaju predstavljanje brojeva u decimalnom brojnom sistemu gdje svaki bajt sadrži jednu cifru u slučaju tzv. *unpacked* formata odn. dvije cifre u slučaju *packed* formata. Ovaj tip podataka se danas rijetko koristi, tako da odgovarajuće instrukcije neće biti razmatrane u ovom tekstu.

5.4 Skup instrukcija

Skup instrukcija koje podržava *IA-32* arhitektura se može podijeliti na sledeće grupe:

- instrukcije opšte namjene
- instrukcije za rad sa realnim brojevima
- sistemske odn. privilegovane instrukcije
- *SIMD* instrukcije

Sve ove grupe instrukcija će biti redom razmatrane u narednim poglavljima ovog teksta. Treba odmah reći da će instrukcije biti uglavnom samo pobrojane, uz navođenje najosnovnijih podataka. Skup instrukcija *IA-32* arhitekture je veoma neregularan, kako u pogledu podržanih instrukcija, tako i u pogledu dozvoljenih tipova operanada i same semantike istih. Međutim,

nabrajanje relevantnih podataka za svaku instrukciju bi oduzelo previše prostora, a i nepotrebno bi se duplirala postojeća dokumentacija. Zato odgovarajući segment ovog teksta treba shvatiti samo kao upoznavanje sa onim što je u skupu instrukcija *IA-32* arhitekture podržano, a pri programiranju na asembleru stalno treba konsultovati referentnu dokumentaciju ([9]).

Treba znati i da su instrukcije opšte namjene, instrukcije za rad sa realnim brojevima i sistemske instrukcije podržane na svim procesorima koji implementiraju *IA-32* arhitekturu, dok su *SIMD* instrukcije podržane samo na poslednjim generacijama ovih procesora. Na taj način, programi koji sadrže instrukcije iz ove grupe mogu se pokretati samo na odgovarajućim procesorima.

5.5 *GNU* asembler

GNU asembler *as* je primarno razvijen kao dio *GNU gcc*[6] paketa; *GNU C* prevodilac npr. interno koristi ovaj asembler da bi generisao objektni fajl na osnovu datog fajla sa izvornim *C* kodom. Međutim, *as* se isto tako može koristiti i za pisanje programa isključivo na asembleru. Svi primjeri koji slijede u ovom tekstu su razvijeni na *as*-u; u početku će primjeri zaista biti čisto asemblerski programi, a kasnije će dosta pažnje biti posvećeno tome kako se asemblerski kod koristi zajedno sa *C* kodom. Naime, obzirom na složenost novijih procesora i kvalitet današnjih prevodilaca, pisanje programa isključivo na asembleru sve manje ima smisla. Sa druge strane, pisanje kritičnih segmenata koda na asembleru radi maksimalne efikasnosti i dalje je ponekad neophodno, tako da je u smislu praktičnog korišćenja asemblera jako važno naučiti kako se kod napisan na njemu povezuje sa kodom napisanim na višim programskim jezicima, prije svega na *C*-u.

Mada u principu instrukcije u asemblerskom programu direktno odgovaraju mašinskim instrukcijama, asembleri se po sintaksi često dosta razlikuju. Te razlike se mogu manifestovati npr. u navođenju direktnih operanada, u imenovanju registara, u načinu određivanja tipa memorijskih operanada, u redosledu navođenja operanada i sl. *GNU* asembler podržava dvije različite sintakse: *AT&T* sintaksu i *Intel*-ovu sintaksu. Prvi tip sintakse je istovremeno i podrazumijevana sintaksa *GNU* asemblera; ime dolazi od toga što je sintaksa slična sintaksi asemblera koji je isporučivan uz *AT&T UNIX*. Prednost ove sintakse je što je generička - pored *IA-32* arhitekture, *GNU* asembler podržava i mnoge druge arhitekture, tako da kad se jednom nauči ova sintaksa lako je preći na neku drugu arhitekturu. Sa druge strane, prednost *Intel*-ove sintakse je što se ta sintaksa koristi u *Intel*-ovoj referentnoj dokumentaciji ([8], [9], [10]) i što većina drugih *IA-32* asemblera koristi

[6] *GNU compiler collection*

sličnu sintaksu. Ovi razlozi su prevagnuli da u ovom tekstu programi budu prezentirani korišćenjem *Intel*-ove sintakse. Treba međutim zapamtiti da se *GNU* asembleru u svakom programu mora staviti do znanja da je korišćena *Intel*-ova sintaksa tako što će se u prvoj liniji programa navesti direktiva `.intel_syntax`. Ovoj direktivi će uvijek biti pridruživan i argument `noprefix` koji naznačava da imena registara ne treba da imaju `%` prefiks, što asembler inače očekuje.

Važan element za uvježbavanje programiranja na asembleru je svakako mogućnost ispisa na standardni izlaz odn. čitanja sa standardnog ulaza. Jedan od načina da se ovo postigne jeste preko odgovarajućih sistemskih poziva operativnom sistemu. Kako se iz asemblera vrše sistemski pozivi biće objašnjeno kasnije u ovom tekstu, a za ulazno-izlazne operacije će biti korišćen komforniji pristup - biće naime pozivane funkcije iz *C* standardne biblioteke. Na taj način, od samog početka će primjeri upućivati na uporedo korišćenje asemblera sa *C*-om, što i jeste jedna od najvažnijih stvari koja kroz njih treba da se nauči.

Slijedi kompletan primjer asemblerskog programa. Po već ustaljenom običaju kad je u pitanju uvodni program pri učenju nekog novog programskog jezika, radi se o programu koji štampa pozdravnu poruku na standardni izlaz. Program demonstrira osnovnu strukturu programa pisanog na *GNU* asembleru koja će biti zadržana i u ostalim primjerima. Slijedi kod, a potom i detaljnija objašnjenja sintakse *GNU* asemblera (kao i kod odgovarajućeg poglavlja o sintaksi *Verilog*-a, ovaj tekst je pomalo suvoparan, ali je njegovo razumijevanje neophodno za dalji rad):

```
.intel_syntax noprefix

.data
fmt:    .asciz  "Hello, world!\n"       # Poruka koju treba ispisati.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ispisuje se poruka.
        lea     eax, fmt
        push    eax
        call printf
        add     esp, 4

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Program počinje već pomenutom `.intel_syntax` direktivom sa argumentom

`noprefix`. Asemblerske direktive se ne prevode u mašinski kod; one utiču na način kako se prevode asemblerske instrukcije i asemblerski program u cjelini, služe za definisanje podataka i sekcija u asemblerskom programu, implementiraju neke funkcije pretprocesora itd. Veći broj direktiva *GNU* asemblera će biti prezentiran kroz ovaj tekst. Treba naglasiti da su skup direktiva i njihova sintaksa takođe visoko specifične od asemblera koji se koristi; u *GNU* asembleru se direktive lako prepoznaju po tome što počinju tačkom.

Početnu direktivu u gornjem programu slijedi još jedna direktiva - u pitanju je `.data` direktiva koja služi za definisanje odgovarajuće sekcije. U istu svrhu se koristi i `.text` direktiva koja se nalazi nešto niže u kodu. U najširem smislu, sekcija predstavlja niz sukcesivnih lokacija u memoriji bez "rupa". Sekcije olakšavaju linkovanje izvršnog fajla. Naime, većina operativnih sistema ima svoje konvencije u pogledu memorijskih adresa na kojima počinju podaci ili izvršni kod; takođe se vrlo često izvršni fajl linkuje iz više objektnih fajlova. Podjela koda na sekcije omogućava linkeru da lako razvrsta gdje šta treba da stavi u finalnom izvršnom fajlu. Početak svake sekcije se u asembleru navodi odgovarajućom direktivom. Sekcije se protežu od ove direktive do naredne direktive kojom počinje neka sekcija ili do kraja fajla. Tri najvažnije sekcije se naznačavaju direktivama `.data` za inicijalizovane podatke, `.bss` za neinicijalizovane podatke i `.text` za kod.

U `.data` sekciji se deklarišu podaci čije vrijednosti treba da budu inicijalizovane u memoriji. Svaka deklaracija se sastoji od tri dijela: labele koja imenuje podatak, direktive koja određuje tip podatka i vrijednosti koja se dodjeljuje podatku. Labele se u *GNU* asembleru deklarišu tako što se navede odgovarajući jedinstveni identifikator i potom odmah iza istog stavi dvotačka; referenciranje labela se vrši samo preko identifikatora. Identifikatori se sastoje od slova i brojeva, a mogu sadržati i karaktere `_`, `.` i `$`; pritom, identifikator ne može počinjati brojem. Treba voditi računa i da *GNU* asembler, za razliku od nekih drugih asemblera, pravi razliku između malih i velikih slova kod identifikatora. U asemblerskoj terminologiji je inače uobičajeno da se identifikatori nazivaju simbolima, pa će taj naziv u ostatku teksta biti korišćen. *GNU* asembler ne pravi razliku između malih i velikih slova u imenima instrukcija, ali će kroz ovaj tekst biti korišćena konvencija da se instrukcije pišu velikim slovima u tekstu da bi bile uočljivije, a malim slovima u listinzima programa.

Za deklarisanje tipova podataka *GNU* asembler podržava veći broj asemblerskih direktiva. Lista ovih direktiva je data u tabeli 5.2 (treba napomenuti i da neke od direktiva imaju alternativna imena koja ovdje nisu navedena). Može se primijetiti da se imena koja *GNU* asembler koristi za tipove podataka razlikuju od odgovarajućih imena u *IA-32* terminologiji, tako da o tome treba voditi računa.

Direktiva	Tip podataka
`.ascii`	string
`.asciz`	string automatski terminiran nulom
`.byte`	8-bitni cio broj
`.double`	64-bitni realan broj
`.float`	32-bitni realan broj
`.int`	32-bitni cio broj
`.octa`	128-bitni cio broj
`.quad`	64-bitni cio broj
`.short`	16-bitni cio broj

Tabela 5.2: Asemblerske direktive za deklarisanje tipova podataka.

Po navođenju direktive za dati tip podataka mogu se navesti jedna ili više konstanti koje predstavljaju podatak datog tipa; ako ih ima više, onda se razdvajaju zarezima.

Konstante koje su karakteri se navode između jednostrukih navodnika, kao u *C*-u. Specijalni karakteri se takođe navode kao u *C*-u, kao kombinacija obrnute kose crte i slova; pritom, podržan je samo podskup specijalnih karaktera od onih koji su podržani na *C*-u. Konstante koje su stringovi se navode između dvostrukih navodnika, opet kao u *C*-u. Za razliku od *C*-a, ovakve konstante asembler ne terminira nulom automatski - to se radi samo ukoliko je tip podatka koji predstavlja string jednak `.asciz`.

Cjelobrojne i realne konstante se navode na potpuno isti način kao na *C*-u (osim sufiksa koji na *C*-u opciono određuju širinu broja, a koji na asembleru nisu podržani).

Labele u *GNU* asembleru predstavljaju proste memorijske adrese. Prema tome i ovome što je upravo rečeno o konstantama, deklaracija stringa `fmt` iz gornjeg koda ima efekat da se niz bajtova koji predstavljaju karaktere stringa smještа na sukcesivne lokacije u memoriji na početku `.data` sekcije. Pritom, taj niz bajtova se još terminira jednim bajtom čija je vrijednost nula zbog toga što je konstanta deklarisana da bude tipa `.asciz`.

Naredbe asemblera se razdvajaju ili dvotačkom ili novim redom. Naredba može biti direktiva ili asemblerska instrukcija. Naredba u kojoj je deklarisan string `fmt` demonstrira i kako se pišu komentari u *GNU* asembleru. Komentari počinju `#` karakterom i protežu se do kraja reda. *GNU* asembler podržava takođe i višelinijske komentare koji imaju istu sintaksu kao komentari u *C*-u.

Direktivu `.text` slijedi direktiva `.global` koja se odnosi na simbol `main`. Direktivom `.global` se naznačava da je neki simbol vidljiv i izvan objektnog fajla u koga bude preveden izvorni kod koji sadrži simbol. Simbol `main` predstavlja labelu kojom počinje kod. Na asembleru se funkcije, baš kao i

promjenljive, naznačavaju labelama. Jednostavnosti radi, odabrano je da se zasad za startovanje asemblerskog programa koristi standardna *C* biblioteka, tako da simbol `main` zapravo odgovara `main()` funkciji na *C*-u i ovaj simbol mora biti globalno vidljiv jer je potreban linkeru. Kasnije će u toku teksta biti objašnjeno kako se asemblerski programi mogu pokretati direktno, bez pomoći *C* standardne biblioteke.

Asemblerski kod koji sačinjava program sastoji se od tri cjeline. Na početku odn. na kraju nalaze se standardni prolog i epilog funkcije. Ovaj kod će biti detaljnije objašnjen kada bude riječi o pozivanju potprograma na asembleru, zasad će biti samo prepisivan iz programa u program. Tijelo programa se sastoji od poziva `printf()` funkcije iz *C* standardne biblioteke. Opet je u pitanju kod koji će biti potpuno jasan tek kad bude obrađeno pozivanje potprograma, ali obzirom da će sličan kod biti dosta korišćen u međuvremenu za ispis na standardni izlaz odn. za čitanje sa standardnog ulaza, isti će ovdje biti donekle pojašnjen.

Pri pozivanju procedura na asembleru, argumenti se prenose preko steka. Stek je veoma korisna struktura podataka, do te mjere da svi savremeni procesori imaju direktnu podršku za isti; *IA-32* arhitektura nije izuzetak u tom pogledu. Stek predstavlja dio memorije kome se pristupa po *LIFO*[7] principu; procesori obično imaju posebne instrukcije za pristup memoriji na ovaj način, a takođe u nekom registru čuvaju adresu vrha steka. Na *IA-32* arhitekturi, instrukcija `PUSH` služi za stavljanje podatka na stek, a instrukcija `POP` za skidanje podatka sa steka. Prenošenje argumenata proceduri se sastoji od stavljanja istih na stek i to obrnutim redom. U ovom slučaju funkcija `printf()` se poziva sa jednim argumentom, i taj argument je adresa stringa. Dakle, na stek treba da se stavi adresa stringa; prethodno je adresu potrebno prebaciti u neki registar jer `PUSH` i `POP` instrukcije rade samo sa registrima. Početak stringa je označen labelom `fmt`, ali sintaksa asemblera je takva da referenciranje ovog simbola u ostatku koda označava podatak koji se nalazi na toj adresi. Zato se za učitavanje adrese na kojoj počinje string mora iskoristiti specijalna instrukcija `LEA`. Standardna konvencija za *Intel*-ovu sintaksu je da se u asemblerskoj instrukciji prvo navodi odredišni, a potom ostali operandi, pa je tako i u ovom slučaju prvo naveden registar `EAX` u koji se smješta adresa stringa, a potom simbol `fmt`. Instrukcija `LEA` smješta adresu stringa u registar `EAX`, a potom se sadržaj ovog registra stavlja na stek instrukcijom `PUSH`. Potom treba pozvati `printf()` funkciju. Instrukcija koja u asembleru služi za pozivanje procedure je `CALL` instrukcija; ovu instrukciju slijedi ime simbola koji predstavlja funkciju. *GNU C* prevodilac kada prevodi funkciju započinje asemblerski kod labelom istog imena kao funkcija. Standardna *C* biblioteka koja se isporučuje uz ovaj prevodilac slijedi istu konvenciju i zato je za pozivanje `printf()` funkcije dovoljno navesti simbol `printf`. Funkcija `printf()` će potom da odštampa dati string, pa preostaje

[7] *last in first out*

da se skine argument sa steka. Ovo je po konvencijama o kojima će opet biti više riječi kada bude razmatrano pozivanje funkcija dužnost programa koji poziva funkciju. Skidanje argumenta sa steka može biti izvršeno `POP` instrukcijom, ali taj argument se zapravo više ne koristi tako da nema potrebe vraćati ga u neki registar. Umjesto toga, može se iskoristiti činjenica da se na *IA-32* arhitekturi adresa vrha steka drži u `ESP` registru. Stek na ovoj arhitekturi raste prema nižim adresama u memoriji, tako da je za skidanje 4-bajtnog podatka sa steka dovoljno sadržaju `ESP` registra dodati 4. To je i urađeno instrukcijom `ADD`. Ova instrukcija, kao i većina drugih aritmetičko-logičkih instrukcija koje implementiraju binarne operacije, ima dva operanda pri čemu je prvi operand istovremeno i odredišni operand odn. po izvršenoj operaciji rezultat se smješta u njega zamjenjujući onu vrijednost koja se ranije tamo nalazila. Na taj način se `ADD` instrukcijom ažurira sadržaj `ESP` registra.

Program koji je prezentiran može se prevesti iz dva koraka. Po konvenciji, fajlovi sa asemblerskim kodom nose ekstenziju `.s`, tako da se program može sačuvati recimo u fajlu `hello.s`. Ovaj fajl se može prevesti u objektni fajl sa imenom `hello.o` naredbom:

```
as -o hello.o hello.s
```

Opcijom `-o` se, baš kao kod *C* kompajlera, navodi ime izlaznog fajla. *GNU* asembler podržava još neke opcije, od kojih je najvažnija `-gstabs` koja specificira da se u objektni fajl upišu informacije koje omogućavaju debagovanje. *GNU* debager *gdb* podržava debagovanje asemblerskog koda ukoliko je isti preveden sa uključenom ovom opcijom.

Linkovanje je, obzirom da program koristi standardnu *C* biblioteku za startovanje, najlakše obaviti pozivanjem programa *gcc* prostim:

```
gcc -o hello hello.o
```

Naravno, ove komande, kao i ostale komande za održavanje projekta, najbolje je izdvojiti u odgovarajući *makefile*.

Poglavlje 6

Instrukcije opšte namjene

6.1 Aritmetičko-logičke instrukcije

Lista aritmetičkih instrukcija koje podržava *IA-32* arhitektura je data u tabeli 6.1. Logičke instrukcije su date u tabeli 6.2.

Mnemonik	Operacija
`ADD`	sabiranje
`ADC`	sabiranje sa prenosom
`SUB`	oduzimanje
`SBB`	oduzimanje sa pozajmicom
`INC`	inkrementiranje
`DEC`	dekrementiranje
`CMP`	poređenje
`NEG`	promjena znaka
`MUL`	množenje neoznačenih brojeva
`IMUL`	množenje označenih brojeva
`DIV`	dijeljenje neoznačenih brojeva
`IDIV`	dijeljenje označenih brojeva

Tabela 6.1: Aritmetičke instrukcije.

Mnemonik	Operacija
`AND`	bitska konjunkcija
`OR`	bitska disjunkcija
`NOT`	bitska negacija
`XOR`	bitska operacija *XOR*

Tabela 6.2: Logičke instrukcije.

IA-32 arhitektura podržava jako mali broj instrukcija sa tri operanda. Zato,

kako je već pomenuto, instrukcije koje implementiraju binarne operacije najčešće upisuju rezultat u prvi operand. Ovo znači i da prvi operand bilo koje ovakve instrukcije ne može biti konstanta. Dalje, samo jedan operand može biti memorijski operand; drugi operand u tom slučaju mora biti registar ili konstanta. Ova pravila je jako važno stalno imati na umu[1].

Instrukcije `ADD` i `SUB` vrše sabiranje odn. oduzimanje datih operanada. Instrukcije `ADC` i `SBB` takođe vrše sabiranje odn. oduzimanje, ali uzimajući u obzir i vrijednost *carry* flega kao prenos odn. pozajmicu: u prvom slučaju se na zbir operanada dodaje vrijednost *carry* flega, a u drugom slučaju se ta vrijednost oduzima od razlike operanada. Ove instrukcije su pogodne za implementaciju sabiranja odn. oduzimanja cijelih brojeva širine veće od 32 bita. Instrukcije `INC` odn. `DEC` vrše inkrementiranje odn. dekrementiranje datog operanda. Ove instrukcije se, kao i u višim programskim jezicima, najviše koriste za ažuriranje brojača u petljama. Sve nabrojane instrukcije koje se tiču sabiranja i oduzimanja rade podjednako nad neoznačenim odn. označenim operandima.

Kao i većina instrukcija *IA-32* arhitekture, `SUB` instrukcija mijenja vrijednosti flegova u `EFLAGS` registru. Ove vrijednosti se mogu potom koristiti kao uslovi u instrukcijama kontrole toka programa. Instrukcija `SUB` je posebno pogodna za izračunavanje uslova pri kontroli toka, obzirom da se ispitivanjem razlike dva broja isti mogu porediti. Međutim, instrukcija `SUB`, kao i ostale instrukcije koje implementiraju binarne operacije, uvijek upisuje rezultat preko jednog od operanada, što je obično nezgodno ako se brojevi žele samo porediti. Stoga *IA-32* arhitektura podržava `CMP` instrukciju, koja radi isto što i `SUB` instrukcija, s tom razlikom što ne smješta rezultat nigdje već samo ažurira vrijednosti flegova. Instrukciju `CMP` gotovo uvijek slijedi instrukcija uslovnog skoka. Obzirom da se oduzimanje neoznačenih i označenih brojeva izvodi na isti način, flegovi koji se ispituju u instrukciji skoka koja slijedi `CMP` instrukciju određuju da li su brojevi poređeni kao neoznačeni ili označeni.

Instrukcija `NEG` tretira operand kao označeni broj i mijenja mu znak.

Obzirom da se množenje odn. dijeljenje neoznačenih i označenih brojeva implementira na različite načine, *IA-32* arhitektura ima po dvije različite instrukcije za množenje odn. dijeljenje neoznačenih i označenih brojeva.

Množenje neoznačenih brojeva se izvodi instrukcijom `MUL`, dok se za množenje označenih brojeva koristi instrukcija `IMUL`. Razlika između ovih instrukcija je i u broju operanada. Instrukcija `MUL` ima samo jedan operand koji može biti vrijednost u registru ili memoriji. Ovaj operand se množi sa sadržajem registra `AL`, `AX` odn. `EAX` (u zavisnosti od širine operanda), pri čemu rezultat

[1] nabrojana ograničenja inače dolaze od nastojanja da se instrukcije učine kraćim - registri opšte namjene kod *IA-32* arhitekture se mogu kodirati sa 3 bita, dok je npr. za kodiranje memorijskih operanada zbog složenog načina adresiranja potrebno znatno više bitova

respektivno biva smješten u registre `AX`, `DX:AX` odn. `EDX:EAX`. Kada se navodi par registara razdvojen dvotačkom, to znači da se prvi registar odnosi na više bitove podatka, a drugi registar na niže bitove. Instrukcija `IMUL` podržava identičnu formu sa jednim operandom, ali takođe podržava i forme sa dva odn. tri operanda. U ovim slučajevima, odredišni operand je iste širine kao i činioci, tako da može doći do prekoračenja; takođe operandi mogu biti samo 16-bitni i 32-bitni, ne i 8-bitni. U formi sa tri operanda, jedan činilac mora biti konstanta [2].

Instrukcija `DIV` služi za dijeljenje neoznačenih cijelih brojeva. Ova instrukcija je komplementarna `MUL` instrukciji; instrukcija `DIV` naime dijeli vrijednosti u registrima `AX`, `DX:AX` odn. `EDX:EAX` sa datom 8-bitnom, 16-bitnom odn. 32-bitnom vrijednosti iz registra ili memorije. Količnik se respektivno smješta u registre `AL`, `AX` odn. `EAX`, a ostatak respektivno u registre `AH`, `DX` odn. `EDX`. Instrukcija `IDIV` je potpuno ekvivalentna instrukciji `DIV`, osim što radi nad označenim brojevima; ova instrukcija dakle nema ekstenzije kao `IMUL` instrukcija.

Logičke instrukcije obavaljaju odgovarajuće bitske logičke operacije nad svojim operandima.

U aritmetičko-logičke instrukcije se mogu ubrojiti i instrukcije za rad sa brojevima u *BCD* formatu, koje ovdje neće biti razmatrane.

Slijedi listing programa koji prikazuje neke od nabrojanih instrukcija. Za većinu grupa instrukcija koje budu predstavljene u ovom tekstu biće ovako dat i primjer programa koji demonstrira upotrebu nekih (obično značajnijih) instrukcija iz date grupe; instrukcije koje ne budu demonstrirane na ovaj način biće uglavnom demonstrirane nekim kasnijim sličnim primjerom. U ovom slučaju, program učitava dva označena cijela broja sa standardnog ulaza i izračunava i štampa njihov zbir, razliku, proizvod i količnik:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d%d"          # Ulazni format.
wrfmt:  .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  x, 4                    # Prvi operand.
.lcomm  y, 4                    # Drugi operand.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha
```

[2] ove neregularnosti opet dolaze od nastojanja da se kodiranje instrukcije održi koliko-toliko kompaktnim

```
## Ucitavaju se operandi.
lea     eax, y
push    eax
lea     eax, x
push    eax
lea     eax, rdfmt
push    eax
call scanf
add     esp, 12

## Izracunava se i ispisuje zbir operanada.
mov     eax, x
add     eax, y
push    eax
lea     eax, wrfmt
push    eax
call printf
add     esp, 8

## Izracunava se i ispisuje razlika operanada.
mov     eax, x
sub     eax, y
push    eax
lea     eax, wrfmt
push    eax
call printf
add     esp, 8

## Izracunava se i ispisuje proizvod operanada (smatra se da ce
## proizvod stati u 32 bita).
mov     eax, x
imul    dword ptr y
push    eax
lea     eax, wrfmt
push    eax
call printf
add     esp, 8

## Izracunava se i ispisuje kolicnik operanada.
mov     eax, x
cdq
idiv    dword ptr y
push    eax
lea     eax, wrfmt
push    eax
call printf
add     esp, 8

## Epilog funkcije.
popa
xor     eax, eax
leave
ret
```

U ovom programu korišćena je `.bss` sekcija. Memorijske promjenljive se u ovoj sekciji deklarišu pomoću `.lcomm` direktive koju slijede zarezom razdvojeni ime promjenljive i broj bajtova koje ista u memoriji zauzima. Memorijske promjenljive iz `.bss` sekcije su neinicijalizovane.

Za učitavanje operanada sa ulaza iskorišćena je funkcija `scanf()` iz *C* standardne biblioteke. Ovoj funkciji se prenose tri argumenta: format za učitavanje i adrese na koje treba smjestiti vrijednosti operanada. Kako je već pomenuto, argumenti se po konvenciji stavljaju na stek obrnutim redom. Za ispis rezultata se poziva `printf()` funkcija na način identičan kao u prethodnom primjeru; razlika je jedino što se ovdje funkciji prenose dva argumenta.

Obzirom da kod sabiranja i oduzimanja jedan operand mora biti u registru, dok opet kod množenja i dijeljenja taj operand implicitno mora biti u registru `EAX`, izvršavanje svake računske operacije počinje prebacivanjem prvog operanda u registar `EAX`. Za ovo prebacivanje se koristi `MOV` instrukcija; ova instrukcija spada u grupu instrukcija za transfer podataka koje će biti naredne razmatrane.

Instrukcije `IMUL` (u formi sa jednim operandom) i `IDIV` prihvataju 8-bitni, 16-bitni i 32-bitni operand. Kada se kao operand navodi registar, onda je njegova širina implicitno određena; kada se međutim kao operand navodi vrijednost iz memorije, onda se ne zna da li sa date memorijske lokacije treba uzeti 8-bitnu, 16-bitnu ili 32-bitnu vrijednost. U ovom i sličnim slučajevima se širina operanda u *GNU* asembleru naznačava tzv. prefiksom koji može imati vrijednosti `BYTE PTR` za 8-bitni operand, `WORD PTR` za 16-bitni operand, `DWORD PTR` za 32-bitni operand i `QWORD PTR` za 64-bitni operand.

Pri dijeljenju se dijeljenik nalazi u `EDX:EAX` registrima. Zato je potrebno po učitavanju prvog operanda u registar `EAX` upisati u registar `EDX` odgovarajuću vrijednost. U tu svrhu je iskorišćena instrukcija `CDQ` koja proširuje 32-bitnu vrijednost iz `EAX` registra u 64-bitnu vrijednot u `EDX:EAX` registrima. Ova instrukcija će takođe biti razmatrana u okviru instrukcija za transfer podataka.

Slijedi još jedan primjer programa koji demonstrira korišćenje aritmetičko-logičkih operacija:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%x,%x"         # Ulazni format.
wrfmt:  .asciz  "%x,%x\n"       # Izlazni format.

.bss
.lcomm  x, 8                    # Prvi sabirak.
.lcomm  y, 8                    # Drugi sabirak.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se prvi sabirak.
```

```
lea     eax, x
push    eax
lea     eax, x+4
push    eax
lea     eax, rdfmt
push    eax
call scanf
add     esp, 12

## Ucitava se drugi sabirak.
lea     eax, y
push    eax
lea     eax, y+4
push    eax
lea     eax, rdfmt
push    eax
call scanf
add     esp, 12

## Izracunava se zbir.
mov     eax, x
add     eax, y
mov     edx, x+4
adc     edx, y+4

## Ispisuje se zbir.
push    eax
push    edx
lea     eax, wrfmt
push    eax
call printf
add     esp, 12

## Epilog funkcije.
popa
xor     eax, eax
leave
ret
```

Program sa standardnog ulaza učitava dva 64-bitna broja, pri čemu se za svaki broj u heksadecimalnom formatu navode zarezom razdvojeni prvo 32 viša, a zatim 32 niža bita. Program potom sabira ove brojeve u dva koraka: prvo sabira niža 32 bita `ADD` instrukcijom, a potom i viša 32 bita, ali ovaj put `ADC` instrukcijom, uzimajući na taj način u obzir prenos iz niža 32 bita. Program potom štampa rezultat na standardnom izlazu u istom formatu u kom su brojevi i učitani.

U slučajevima kada operandi imaju još veći broj bitova zgodno je sabiranje vršiti uvijek `ADC` instrukcijom u petlji; pritom, za inicijalizaciju `carry` flega na vrijednost 0 prije ulaska u petlju može se iskoristiti `CLC` instrukcija.

6.2 Instrukcije za transfer podataka

Instrukcije za transfer podataka služe, kako im i ime kaže, za kopiranje podataka sa jedne lokacije na drugu. Lista instrukcija ovog tipa koje *IA-32*

arhitektura podržava je data u tabeli 6.3.

Mnemonik	Operacija
MOV	prenos podataka
CMOV	uslovni prenos podataka
XCHG	razmjena podataka
BSWAP	promjena redosljeda bajtova
XADD	razmjena i sabiranje podataka
CMPXCHG	testiranje i razmjena podataka
CMPXCHG8	testiranje i razmjena 64-bitnih podataka
PUSH	stavljanje na stek
POP	skidanje sa steka
PUSHA	stavljanje registara opšte namjene na stek
POPA	skidanje registara opšte namjene sa steka
CBW	konverzija 8-bitne u 16-bitnu vrijednost
CWD	konverzija 16-bitne u 32-bitnu vrijednost
CWDE	konverzija 16-bitne u 32-bitnu vrijednost
CDQ	konverzija 32-bitne u 64-bitnu vrijednost
MOVSX	prenos podataka uz proširivanje očuvanjem znaka
MOVZX	prenos podataka uz proširivanje popunjavanjem nulama

Tabela 6.3: Instrukcije za transfer podataka.

Ubjedljivo najčešće korišćena instrukcija za transfer podataka je `MOV` instrukcija. Za operande ove instrukcije važi slično što i za operande instrukcija koje implementiraju binarne operacije: samo jedan operand može biti memorijska lokacija, a izvorni operand može biti i konstanta. Veoma slične `MOV` instrukciji su instrukcije iz grupe instrukcija uslovnog transfera. Imena ovih instrukcija počinju sa `CMOV` i ove instrukcije obavljaju prenos samo ako je odgovarajući uslov koji je izražen sufiksom ispunjen. Uslovi se tiču stanja flegova u `EFLAGS` registru odn. praktično rezultata prethodne operacije. Sufiksi kod `CMOV` instrukcija su slični sufiksima instrukcija uslovnog skoka koje su prikazane na tabelama 6.5 i 6.6. Inače, instrukcije uslovnog transfera su podržane tek od *PentiumPro* procesora i njihova jedina upotreba je praktično optimizacija malih grananja koja sadrže dodjelu.

Instrukcija `XCHG` razmjenjuje sadržaj svoja dva operanda u jednoj atomičnoj operaciji. Ova instrukcija je korisna za implementiranje semafora i sličnih struktura podataka kod sinhronizacije procesa.

Instrukcija `BSWAP` obrće redosled bajtova datog operanda i može se iskoristiti za konverziju između ranije pominjanih *little-endian* i *big-endian* šema. Ova instrukcija ne postoji na *80386* procesoru.

Instrukcija `XADD` razmjenjuje sadržaj operanada, a onda izračunava njihov zbir i smješta ga u odredišni operand. Instrukcije `CMPXCHG` i `CMPXCHG8` imaju

tri operanda; ove instrukcije porede dva operanda, pa ako su jednaki zamjenjuju drugi operand sa trećim, a ako nisu jednaki učitavaju drugi operand u prvi. Instrukcija `CMPXCHG` radi sa 32-bitnim operandima i prvi operand je implicitno registar `EAX`. Instrukcija `CMPXCHG8` radi sa 64-bitnim operandima i prvi i treći operand su implicitno registri `EDX:EAX` odn. `ECX:EBX`. Instrukcije `XADD`, `CMPXCHG` i `CMPXCHG8` imaju sličnu namjenu kao instrukcija `XCHG`, ali nisu podržane na svim procesorima *IA-32* arhitekture.

Instrukcije `PUSH` i `POP` služe sa stavljanje odn. skidanje vrijednosti sa steka. Na stek se mogu stavljati 16-bitne ili 32-bitne vrijednosti iz registara ili memorije, kao i 8-bitne, 16-bitne ili 32-bitne konstante. Međutim, zbog onog što je rečeno o uravnanju u vezi sa pristupom memoriji, najbolje je na stek stavljati samo 32-bitne vrijednosti. Registar `ESP` u svakom trenutku sadrži adresu podatka koji je na vrhu steka. Kako je već pomenuto, *IA-32* arhitektura predviđa da stek raste prema naniže. Na taj način, `PUSH` instrukcija se sastoji od umanjivanja vrijednosti `ESP` registra za odgovarajući broj (tj. za veličinu u bajtovima podatka koji se stavlja na stek) i stavljanja datog podatka na novu adresu na koju pokazuje ovaj registar, dok se `POP` instrukcija sastoji od uvećavanja vrijednosti `ESP` registra.

Instrukcije `PUSHA` i `POPA` služe za stavljanje odn. skidanje sa steka vrijednosti iz svih 8 registara opšte namjene odjednom. Ove instrukcije su korisne kada treba brzo sačuvati stanje procesora pri nekoj promjeni konteksta. Treba se prisjetiti da su ove instrukcije korišćene u prologu odn. epilogu `main()` funkcije u svakom od prethodnih primjera.

Operacije sa stekom su efikasnije od `MOV` instrukcije iz dva razloga: obzirom da je odredišna adresa implicitno određena, kodovi instrukcija su kraći, a takođe kod ovih operacija se automatski ažurira odredišna adresa za sledeći pristup memoriji. Zato pri pisanju asemblerskih programa ukoliko podaci treba da se odlože u memoriju onda to treba po mogućnosti raditi na stek.

U proste instrukcije konverzije podataka se ubrajaju instrukcije `CBW`, `CWD`, `CWDE` i `CDQ`. Ove instrukcije respektivno proširuju vrijednosti iz registara `AL`, `AX`, `AX`[3] i `EAX` u registre `AX`, `DX:AX`, `EAX` i `EDX:EAX`. Ove instrukcije operand tretiraju kao označeni broj; obzirom da se negativni brojevi predstavljaju u drugom komplementu, pokazuje se da se proširenje sastoji u kopiranju bita znaka polaznog operanda u sve nove bitove.

Složenije instrukcije konverzije su instrukcije `MOVSX` i `MOVZX`. Ove instrukcije su generalnije od prethodno nabrojanih instrukcija konverzije: one rade sa dva operanda, pri čemu odredišni operand može biti bilo koji od registara opšte namjene.

---

[3] instrukcija `CWD` je postojala na 16-bitnim procesorima koji su prethodili *IA-32* arhitekturi, tako da je na ovoj arhitekturi uvedena instrukcija `CWDE` koja radi nad istim operandom (registrom `AX`) ali koja ima drugo odredište

6.3 Instrukcije kontrole toka

Instrukcije kontrole toka se mogu podijeliti na instrukcije bezuslovnog skoka, instrukcije uslovnog skoka, instrukcije za brojačke petlje i instrukcije za generisanje prekida. Lista instrukcija bezuslovnog skoka je data u tabeli 6.4.

Mnemonik	Operacija
`JMP`	bezuslovni skok
`CALL`	poziv potprograma
`RET`	povratak iz potprograma

Tabela 6.4: Instrukcije bezuslovnog skoka.

Instrukcija bezuslovnog skoka učitava u `EIP` registar adresu koja je zadata kao operand, čime se praktično izvršavanje programa nastavlja instrukcijom koja se nalazi na toj adresi. Instrukcije `CALL` i `RET` biće detaljnije razmotrene u poglavlju o potprogramima. U grupu instrukcija bezuslovnog skoka bi se mogle ubrojiti i instrukcije `INT` i `IRET`, ali će ove instrukcije ipak biti razmatrane respektivno u segmentu o instrukcijama za generisanje prekida odn. u segmentu o sistemskim instrukcijama, gdje ih je logičnije ubrojiti.

Instrukcije uslovnog skoka provjeravaju određene flegove iz `EFLAGS` registra tj. provjeravaju rezultat prethodne operacije i vrše skok samo ako je dati uslov zadovoljen. Ove instrukcije su predstavljene na tabelama 6.5 i 6.6. Tabele se respektivno odnose na slučajeve kada se operandi prethodne operacije uzimaju kao neoznačeni odn. označeni brojevi. Većina instrukcija uslovnog skoka je data u redundantnim parovima, u kom slučaju su mnemonici u tabeli razdvojeni `/` karakterom. U zadnjim kolonama ovih tabela su naznačene i vrijednosti flegova za koje je dati uslov ispunjen. Oznake $+$ odn. $\oplus$ u ovoj koloni se odnose na logičke operacije *OR* odn. *XOR*.

Slijedi program koji demonstrira korišćenje naredbi bezuslovnog i uslovnog skoka. Opet je u pitanju varijanta kalkulatora, ali ovaj put program u petlji učitava sa standardnog ulaza izraz koji se sastoji od dva operanda i operatora u infiksnoj notaciji, te izračunava i štampa njegov rezultat. Ukoliko operand nije validan (odn. ako nije zadata neka od 4 osnovne aritmetičke operacije), program nastavlja sa učitavanjem narednog izraza. Opisani program bi bio oblika:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d%c%d"        # Ulazni format.
wrfmt:  .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  x, 4                    # Prvi operand.
```

Mnemonik	Operacija	Stanje flegova
`JA/JNBE`	skok ako je veće	$CF + ZF = 0$
`JAE/JNB`	skok ako je veće ili jednako	$CF = 0$
`JB/JNAE`	skok ako je manje	$CF = 1$
`JBE/JNA`	skok ako je manje ili jednako	$CF + ZF = 1$
`JC`	skok ako ima prenosa	$CF = 1$
`JE/JZ`	skok ako je jednako (jednako 0)	$ZF = 1$
`JNC`	skok ako nema prenosa	$CF = 0$
`JNE/JNZ`	skok ako nije jednako (različito od 0)	$ZF = 0$
`JP/JPE`	skok ako je parnost parna	$PF = 1$
`JNP/JPO`	skok ako je parnost neparna	$PF = 0$
`JCXZ`	skok ako je registar `CX` jednak 0	
`JECXZ`	skok ako je registar `ECX` jednak 0	

Tabela 6.5: Instrukcije uslovnog skoka za operacije sa neoznačenim brojevima.

Mnemonik	Operacija	Stanje flegova
`JG/JNLE`	skok ako je veće	$(SF \oplus OF) + ZF = 0$
`JGE/JNL`	skok ako je veće ili jednako	$SF \oplus OF = 0$
`JL/JNGE`	skok ako je manje	$SF \oplus OF = 1$
`JLE/JNG`	skok ako je manje ili jednako	$(SF \oplus OF) + ZF = 1$
`JNO`	skok ako nema prekoračenja	$OF = 0$
`JNS`	skok ako je pozitivno	$SF = 0$
`JO`	skok ako ima prekoračenja	$OF = 1$
`JS`	skok ako je negativno	$SF = 1$

Tabela 6.6: Instrukcije uslovnog skoka za operacije sa označenim brojevima.

```
.lcomm  op, 1                   # Operator.
.lcomm  y, 4                    # Drugi operand.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se izraz.
again:
        lea     eax, y
        push    eax
        lea     eax, op
        push    eax
        lea     eax, x
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 16

        ## Odredjuje se koja je operacija u pitanju.
        mov     al, op
        cmp     al, '+'
        je addition
        cmp     al, '-'
        je subtraction
        cmp     al, '*'
        je multiplication
        cmp     al, '/'
        je division
        jmp again

        ## Izracunava se zbir.
addition:
        mov     eax, x
        add     eax, y
        jmp print

        ## Izracunava se razlika.
subtraction:
        mov     eax, x
        sub     eax, y
        jmp print

        ## Izracunava se proizvod (smatra se da ce proizvod stati u 32
        ## bita).
multiplication:
        mov     eax, x
        imul    dword ptr y
        jmp print

        ## Izracunava se kolicnik.
division:
        mov     eax, x
        cdq
        idiv    dword ptr y

        ## Stampa se rezultat i prelazi se na ucitavanje narednog
```

```
        ## izraza.
print:
        push    eax
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8
        jmp again

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Učitavanje izraza teče na sličan način kao u prethodnom primjeru sa aritmetičkim operacijama. Potom se ispituje koja je operacija u pitanju. Ovdje se više puta koristi `CMP` instrukcija da se operator uporedi redom sa slovnim konstantama koje označavaju osnovne aritmetičke operacije. Ova instrukcija postavlja vrijednosti flegova u `EFLAGS` registru i te vrijednosti ispituje instrukcija uslovnog skoka koja uvijek slijedi `CMP` instrukciju. Korišćena je `JE` instrukcija uslovnog skoka tako da do skoka dolazi ako su vrijednosti koje su poređene jednake. Na taj način, čim se odredi koja je operacija u pitanju, skače se na odgovarajuću labelu. Ukoliko oznaka za operator ne odgovara nijednoj od osnovnih aritmetičkih operacija, naredbom bezuslovnog skoka `JMP` vraća se na ponovno učitavanje izraza. Ukoliko je operator validan, izvršava se odgovarajuća operacija i izračunava se rezultat. Ovaj dio koda je identičan kao u prethodnom primjeru. Kada se izračuna rezultat, naredbom bezuslovnog skoka `JMP` prelazi se na labelu ispod koje se nalazi kod za ispis istog. Ispis rezultata je riješen identično kao u prethodnom primjeru. Po obavljenom ispisu, naredbom bezuslovnog skoka `JMP` kontrola se vraća na učitavanje izraza.

Na gornjem primjeru se vidi kako se instrukcije skoka mogu iskoristiti za implementaciju grananja odn. i petlji. Mada primjer demonstrira beskonačnu petlju, može se uočiti kako bi se dodavanjem odgovarajućeg uslovnog skoka čije bi odredište bilo izvan petlje implementirale uobičajene petlje. Poseban tip petlje koja se često sreće u višim programskim jezicima jeste brojačka petlja, u kojoj je uslov koji se ispituje vezan za vrijednost neke brojačke promjenljive koja se obično u petlji inkrementira ili dekrementira. *IA-32* arhitektura podržava ovaj tip petlje preko posebnih naredbi koje su nabrojane u tabeli 6.7. I ovaj skup instrukcija ima nekih redundantnih instrukcija koje su kao i ranije prikazane u istom redu tabele, razdvojene `/` znakom.

Navedene instrukcije, kao i ostale instrukcije za kontrolu toka, imaju jedan operand koji predstavlja odredišnu adresu. Instrukcija `LOOP` dekrementira sadržaj `ECX` registra (ranije je pomenuto da se ovaj registar na *IA-32* arhitekturi koristi kao brojač) i ukoliko nova vrijednosti nije jednaka 0, skače na adresu datu operandom. U suprotnom, izvršavanje programa se nastavlja

Mnemonik	Operacija
`LOOP`	obična instrukcija brojačke petlje
`LOOPE/LOOPZ`	instrukcija brojačke petlje sa izlaskom ako je $ZF = 0$
`LOOPNE/LOOPNZ`	instrukcija brojačke petlje sa izlaskom ako je $ZF = 1$

Tabela 6.7: Instrukcije za brojačke petlje.

instrukcijom koja slijedi `LOOP` instrukciju. Preostale 4 instrukcije rade na identičan način, osim što pored dekrementirane vrijednosti registra `ECX` provjeravaju i sadržaj *zero* flega: sa instrukcijom `LOOPE` (`LOOPZ`) do skoka ne dolazi ni ako je ovaj fleg jednak 0, a sa instrukcijom `LOOPNE` (`LOOPNZ`) ako je ovaj fleg jednak 1. Ove instrukcije su pogodne za brojačke petlje sa dodatnim uslovom izlaska iz istih.

Postoji više načina na koje se mogu iskoristiti gornje instrukcije da bi se implementirale petlje, ali se obično iste stavljaju na kraj petlje, pri čemu je odredište skoka početak petlje. Ovo podrazumijeva da se kroz petlju prolazi bar jednom, što u nekim slučajevima nije poželjno. Tada je zgodno ispred petlje iskoristiti naredbu `JECXZ`, pa ako se ispostavi da je brojač već na početku petlje jednak 0 može se preskočiti čitava petlja.

Slijedi primjer programa koji koristi opisane instrukcije da izračuna uniformnu normu vektora koga učitava sa standardnog ulaza:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
wrfmt:  .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  value, 4                # Lokacija za smjestanje ucitanih podataka.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se duzina vektora. Duzina se stavlja u registar ecx,
        ## koji ce drzati broj elemenata vektora koje jos treba
        ## ucitati. Pretpostavka je da ce duzina vektora biti
        ## nenegativan broj.
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        mov     ecx, value

        ## Tekuca vrijednost uniformne norme ce biti drzana u registru
```

```
        ## ebx. Inicijalizuje se sadrzaj ovog registra na 0 i provjerava
        ## da li je duzina vektora veca od 0.
        xor     ebx, ebx
        jecxz print

next_element:
        ## Ucitava se naredni element. Brojac se tokom ucitavanja cuva
        ## na steku zato sto scanf() funkcija unistava sadrzaj ecx
        ## registra.
        push    ecx
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        pop     ecx

        ## Izracunava se apsolutna vrijednost elementa i poredi sa
        ## tekucom vrijednoscu norme. Vrijednost norme se u skladu sa
        ## ovim poredjenjem eventualno postavlja na apsolutnu vrijednost
        ## elementa.
        mov     eax, value
        cmp     eax, 0
        jge compare
        neg     eax
compare:
        cmp     eax, ebx
        jng continue
        mov     ebx, eax

continue:
        ## Nastavlja se sa petljom.
        loop next_element

print:
        ## Ispisuje se vrijednost norme na standardni izlaz.
        push    ebx
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Program na početku učitava dužinu vektora i smješta je u registar `ECX` koji se koristi kao brojač. Registar `EBX` će čuvati tekuću vrijednost uniformne norme. Ovaj registar se inicijalizuje na 0 i to korišćenjem `XOR`, a ne `MOV` instrukcije koja vjerovatno prva pada na pamet - ovaj idiom se često susreće u asemblerskim programima za *IA-32* arhitekturu i dolazi od toga što je mašinski kod instrukcije u prvom slučaju kraći. Po inicijalizaciji registra `EBX`, instrukcijom `JECXZ` se provjerava da li je dužina vektora jednaka 0, u kom slučaju se preskače petlja i odmah prelazi na štampanje. Ukoliko to nije slučaj, u petlji se učitava jedan po jedan element vektora, poredi njegova apsolutna vrijed-

nost sa tekućom vrijednošću norme i eventualno ažurira vrijednost norme. Treba uočiti kako se u petlji vrijednost brojača odn. sadržaj registra `ECX` čuva na steku tokom poziva `scanf()` funkcije; ovo stoga što se po konvencijama o kojima će biti više riječi u poglavlju o potprogramima sadržaj ovog registra ne čuva tokom poziva procedure. Na kraju petlje se `LOOP` instrukcijom dekrementira brojač i provjerava da li su učitani svi elementi vektora. Ako to nije slučaj, skače se natrag na početak petlje, a u suprotnom se nastavlja sa instrukcijom koja slijedi `LOOP` instrukciju odn. prelazi se na štampanje rezultata.

Slijedi primjer programa koji učitava sa standardnog ulaza jedan broj, a onda i polje brojeva i određuje da li je dati broj element polja:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
msg:    .asciz  "y\n"           # Izlazna poruka.

.bss
.lcomm  value, 4                # Lokacija za smjestanje ucitanih podataka.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se element koji se trazi i smjesta u registar ebx.
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        mov     ebx, value

        ## Ucitava se duzina polja i smjesta u registar ecx, koji ce
        ## drzati broj preostalih elemenata. Pretpostavka je da ce
        ## duzina polja biti nenegativan broj.
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        mov     ecx, value

        ## Provjerava se da li je duzina polja veca od 0.
        jecxz done

next_element:
        ## Ucitava se jedan po jedan element polja i poredi sa trazenim
        ## elementom. Broj elemenata koji je preostao da se ucita se u
        ## medjuvremenu cuva na steku zato sto scanf() funkcija unistava
        ## sadrzaj ecx registra.
```

```
        push    ecx
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        pop     ecx

        ## Poredi se trazeni element sa tekucim i, ukoliko su razliciti,
        ## nastavlja se sa petljom.
        cmp     ebx, value
        loopne next_element

        ## Ukoliko se iz petlje izaslo zato sto su ucitani svi elementi,
        ## zavrsava se funkcija.
        jne done

        ## U suprotnom se stampa poruka da je trazeni element pronadjen.
        lea     eax, msg
        push    eax
        call printf
        add     esp, 4

done:
        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Struktura programa je slična prethodnom primjeru: u petlji se učitava jedan po jedan element polja, pri čemu se za svaki element ispituje da li je jednak datom broju. Ono što je interesantno jeste da je za izlazak iz petlje iskorišćena `LOOPNE` instrukcija; na taj način, pretraživanje se prekida ili kada brojač dođe na 0 (odn. kada se ispitaju svi elementi polja) ili kada se poređenjem koje prethodi ovoj instrukciji ustanovi da je tekući element jednak traženom broju. Po izlasku iz petlje potrebno je ustanoviti kako je završena petlja, jer je u prvom slučaju rezultat pretraživanja negativan, a u drugom afirmativan. Ovdje se može zgodno iskoristiti `JNE` instrukcija da se još jednom provjeri rezultat poslednjeg poređenja. Treba uočiti da je to moguće zato što `LOOPNE` instrukcija ne mijenja flegove. O ovome se mora pažljivo voditi računa; referentna dokumentacija ([9]) za svaku instrukciju navodi da li i koje flegove mijenja.

Slijedi još jedan primjer programa koji koristi instrukcije petlji - ovaj put demonstrirana je dvostruka petlja na primjeru izračunavanja uniformne norme matrice:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
wrfmt:  .asciz  "%d\n"          # Izlazni format.
```

```
.bss
.lcomm  value, 4                # Lokacija za smjestanje ucitanih podataka.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se dimenzija matrice i prebacuje u registar
        ## edx. Pretpostavka je da ce ova dimenzija biti nenegativan
        ## broj.
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        mov     edx, value

        ## Tekuca vrijednost uniformne norme ce biti drzana u registru
        ## edi. Inicijalizuje se sadrzaj ovog registra na 0 i provjerava
        ## da li je dimenzija matrice veca od 0.
        xor     edi, edi
        cmp     edx, 0
        je print

        ## U spoljasnjoj petlji se prolazi kroz sve redove matrice,
        ## izracunava se suma apsolutnih vrijednosti njihovih elemenata i
        ## poredi sa tekucom vrijednoscu norme, pa se eventualno norma
        ## postavlja na vrijednost ove sume. Suma apsolutnih vrijednosti
        ## elemenata reda se drzi u registru ebx koji se inicijalizuje na
        ## 0. Registar ecx se koristi kao brojac u spoljasnjoj petlji,
        ## ali takodje i u unutrasnoj petlji. Zato se njegova vrijednost
        ## prije ulaska u unutrasnju petlju cuva na steku, a potom
        ## incijalizuje na dimenziju matrice.
        mov     ecx, edx
next_row:
        push    ecx
        mov     ecx, edx
        xor     ebx, ebx

next_element:
        ## U unutrasnjoj petlji se prvo ucitava naredni
        ## element. Sadrzaji registara ecx i edx se tokom ucitavanja
        ## cuvaju na steku zato sto ih scanf() funkcija unistava.
        push    ecx
        push    edx
        lea     eax, value
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8
        pop     edx
        pop     ecx

        ## Potom se izracunava apsolutna vrijednost elementa i dodaje
        ## sumi apsolutnih vrijednosti elemenata tekuceg reda.
        mov     eax, value
```

```
        cmp     eax, 0
        jge addition
        neg     eax
addition:
        add     ebx, eax
        loop next_element

        ## Poredi se suma apsolutnih vrijednosti elemenata tekuceg reda
        ## sa tekucom vrijednoscu norme i eventualno se postavlja
        ## vrijednost norme na ovu sumu. Skida se sa steka vrijednost
        ## brojaca za spoljasnju petlju i prelazi se na naredni red.
        cmp     edi, ebx
        jge continue
        mov     edi, ebx
continue:
        pop     ecx
        loop next_row

print:
        ## Ispisuje se vrijednost norme na standardni izlaz.
        push    edi
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Na početku programa se učitava dimenzija matrice, a potom se u dvostrukoj petlji učitavaju elementi matrice (pretpostavka je da se elementi matrice unose po redovima). Kad god se učita neki element, dodaje se njegova apsolutna vrijednost na sumu apsolutnih vrijednosti elemenata tekućeg reda. Na kraju spoljašnje petlje se poredi ova suma sa tekućom vrijednosti norme i eventualno se ažurira norma. Ključna stvar koju treba uočiti u programu jeste kako se sadržaj registra `ECX` koji je brojač u spoljašnjoj petlji drži na steku tokom unutrašnje petlje zato što se ovaj registar takođe koristi kao brojač i u unutrašnjoj petlji. Stek predstavlja veoma pogodnu strukturu podataka za čuvanje sadržaja registara iz spoljašnje petlje ukoliko se isti registri (a to je, obzirom da *IA-32* arhitektura ima malo registara, obično slučaj) koriste i u unutrašnjoj petlji.

Instrukcije za generisanje prekida su date u tabeli 6.8.

Mnemonik	Operacija
`INT`	softverski prekid
`INTO`	prekid pri računskom prekoračenju
`BOUND`	prekid pri prekoračenju granica polja

Tabela 6.8: Instrukcije za generisanje prekida.

Instrukcija `INT` ima jedan operand i u pitanju je broj prekida; ova instrukcija prosto poziva odgovarajuću prekidnu rutinu. Instrukcija `INTO` poziva odgovarajuću prekidnu rutinu ako je *overflow* fleg postavljen; ova instrukcija se može iskoristiti u vezi sa aritmetičkim instrukcijama koje mogu dovesti do prekoračenja. Instrukcija `BOUND` provjerava da li je dati broj između druga dva data broja i generiše odgovarajući prekid ako to nije slučaj; instrukcija se može iskoristiti npr. za provjeru da li se dati indeks elementa nekog polja nalazi u dozvoljenim granicama.

6.4 Bitske instrukcije

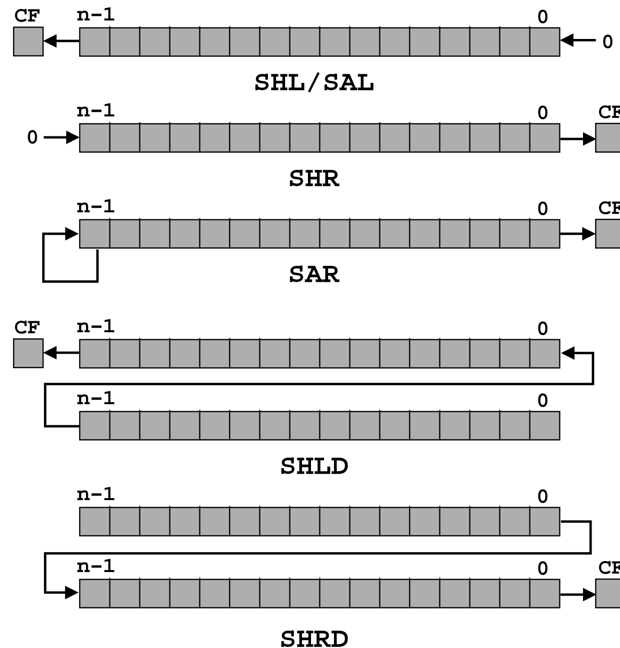
Instrukcije za manipulaciju bitovima se mogu podijeliti na instrukcije šiftovanja, instrukcije rotacije i bitske instrukcije u užem smislu. Instrukcije šiftovanja su nabrojane u tabeli 6.9. Efekat ovih instrukcija je prikazan na slici 6.1.

Mnemonik	Operacija
`SHL/SAL`	logičko i artimetičko šiftovanje ulijevo
`SHR`	logičko šiftovanje udesno
`SAR`	aritmetičko šiftovanje udesno
`SHLD`	šiftovanje ulijevo iz drugog podatka
`SHRD`	šiftovanje udesno iz drugog podatka

Tabela 6.9: Instrukcije šiftovanja.

Logičko i aritmetičko šiftovanje ulijevo su ekvivalentni: bitovi koji predstavljaju operand se pomjeraju za dati broj mjesta ulijevo, a bitovi koji pritom ostaju upražnjeni se popunjavaju nulama. U oba slučaja, bit koji je poslednji šiftovan sa pozicije najveće težine se smješta u *carry* fleg. Logičko šiftovanje udesno funkcioniše na isti način, samo što je ovdje u pitanju pomjeranje udesno; takođe, ovdje se u *carry* fleg po izvršenoj operaciji smješta bit koji je poslednji šiftovan sa pozicije najmanje težine. Kod aritmetičkog šiftovanja udesno razlika je u tome što se upražnjene pozicije uvijek popunjavaju bitom sa pozicije najveće težine odn. bitom znaka. Nabrojane instrukcije mogu imati jedan ili dva operanda. U prvom slučaju pomjeranje je za 1 bit, a u drugom slučaju drugi operand određuje za koliko se bitova vrši pomjeranje; pritom drugi operand može biti ili registar `CL` ili 8-bitna konstanta. Treba uočiti i da se instrukcije logičkog i aritmetičkog šiftovanja mogu iskoristiti za efikasno množenje odn. dijeljenje datog broja stepenom dvojke.

Instrukcije `SHLD` i `SHRD` vrše pomjeranje tako što upražnjene bitove popunjavaju bitovima najveće odn. najmanje težine iz drugog operanda (koji ostaje neizmijenjen). Ove instrukcije imaju tri operanda i treći operand određuje za koliko se bitova vrši pomjeranje. I ovdje ovaj operand može biti samo reg-



Slika 6.1: Efekat instrukcija šiftovanja nad operandom dužine n bitova.

istar `CL` ili 8-bitna konstanta. Poslednji bit koji bude istisnut iz broja koji se šiftuje se takođe smješta u *carry* fleg. Instrukcije `SHLD` i `SHRD` su pogodne za šiftovanje bitskih stringova.

Instrukcije rotacije su nabrojane u tabeli 6.10. Efekat ovih instrukcija je prikazan na slici 6.2.

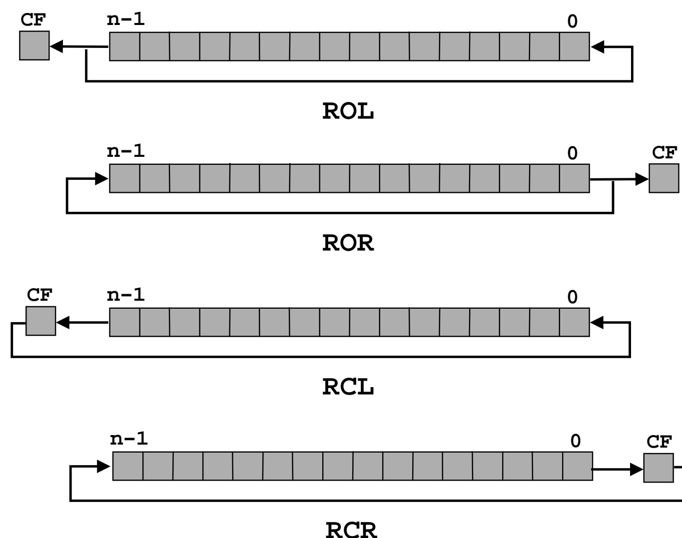
Mnemonik	Operacija
`ROL`	rotacija ulijevo
`ROR`	rotacija udesno
`RCL`	rotacija ulijevo kroz *carry* fleg
`RCR`	rotacija udesno kroz *carry* fleg

Tabela 6.10: Instrukcije rotacije.

Rotacije su slične šiftovanju, s tom razlikom što se u svakom koraku bit koji biva istisnut upisuje na drugom kraju operanda. Ovaj bit se takođe, kao kod šiftovanja, uvijek upisuje u *carry* fleg. Kod rotacija kroz *carry* fleg, u operand se na upražnjeno mjesto upisuje sadržaj ovog flega. Drugi operand instrukcija rotacije, koji određuje za koliko se pozicija vrši rotacija, može imati iste vrijednosti kao kod instrukcija šiftovanja.

Bitske instrukcije u užem smislu su nabrojane u tabeli 6.11.

Instrukcije za testiranje bita upisuju vrijednost bita operanda na specifici-



Slika 6.2: Efekat instrukcija rotacije nad operandom dužine n bitova.

Mnemonik	Operacija
`BT`	testiranje bita
`BTS`	testiranje i postavljanje bita na vrijednost 1
`BTR`	testiranje i postavljanje bita na vrijednost 0
`BTC`	testiranje i komplementiranje vrijednosti bita
`BSF`	pretraživanje prema bitu najveće težine
`BSR`	pretraživanje prema bitu najmanje težine
`SET`	postavljanje bajta prema vrijednosti flega
`TEST`	bitska konjunkcija bez upisa rezultata

Tabela 6.11: Bitske instrukcije u užem smislu.

ranoj lokaciji u *carry* fleg. Osim prve instrukcije iz ove grupe, tri preostale instrukcije pritom i mijenjaju vrijednost ovog bita u operandu tako što ga respektivo postavljaju na 1, 0 ili ga komplementiraju.

Instrukcije `BSF` i `BSR` pretražuju bitsku reprezentaciju date vrijednosti prema bitu najveće odn. najmanje težine i upisuju indeks prvog bita koji je jednak 1 u zadati registar.

Instrukcija `SET` predstavlja zapravo grupu instrukcija koje se razlikuju po sufiksu; ove instrukcije postavljaju zadati bajt na 0 ili 1 u zavisnosti od vrijednosti flegova. Koji se flegovi pritom uzimaju u obzir određeno je sufiksom; ovi sufiksi su ekvivalentni sufiksima u instrukcijama uslovnog skoka.

Instrukcija `TEST` se odnosi prema instrukciji `AND` isto kao što se instrukcija `CMP` odnosi prema instrukciji `SUB` - instrukcija obavlja odgovarajuću operaciju, ali se rezultat ne upisuje nigdje već se samo ažuriraju vrijednosti flegova.

Kao primjer programa koji koristi bitske operacije biće prikazan program koji obrće zadati broj bitova najmanje težine datog broja (ovakav problem se susreće kod pisanja programa za brzu Fourier-ovu transformaciju). Program bi bio oblika:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d%x"          # Ulazni format.
wrfmt:  .asciz  "%x\n"          # Izlazni format.

.bss
.lcomm  n, 4                    # Broj bitova koji se premecu.
.lcomm  x, 4                    # Broj ciji se bitovi premecu.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitavaju se broj bitova koji se premecu i broj ciji se
        ## bitovi premecu.
        lea     eax, x
        push    eax
        lea     eax, n
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 12

        ## U registar eax se prebacuje broj ciji se bitovi
        ## premecu. Registar ecx ce biti koriscen kao brojac i u njega se
        ## stavlja broj bitova koji se premecu. Pretpostavka je da je
        ## ovaj broj u opsegu [0,32]. U registar ebx ce biti smjestani
        ## premetnuti bitovi.
        mov     eax, x
```

```
        mov     ecx, n
        xor     ebx, ebx

        ## Provjerava se da li je broj bitova koje treba premetnuti
        ## jednak 0 i, ako jeste, odmah se ide na stampanje rezultata.
        jecxz print

again:
        ## U petlji se izbacuje bit najmanje tezine iz registra eax i
        ## odmah isti bit upisuje u bit najmanje tezine registra ebx.
        shr     eax, 1
        rcl     ebx, 1
        loop again

        ## Vracaju se premetnuti bitovi u registar eax.
        mov     ecx, n
        shl     eax, cl
        or      eax, ebx

print:
        ## Stampa se premetnuti broj.
        push    eax
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Po učitavanju operanada (pretpostavka je da je broj bitova koje treba obrnuti u opsegu $[0, 32]$), u petlji se izdvaja jedan po jedan bit sa kraja registra `EAX` u koji je smješten dati broj i upisuje obrnutim redosledom u registar `EBX`. Ovo se postiže instrukcijama šiftovanja i rotacije kroz *carry* fleg: registar `EAX` se šiftuje udesno tako da njegov bit najmanje težine ulazi u *carry* fleg, odakle se instrukcijom `RCL` prebacuje u bit najmanje težine registra `EBX`, pri čemu se naravno istovremeno sadržaj ovog registra pomjera za 1 bit ulijevo. Kada se na taj način u registru `EBX` formiraju obrnuti niži bitovi zadatog broja, iste treba ponovo povezati sa višim bitovima broja koji se sada nalaze u nižim bitovima registra `EAX`. U tom cilju se registar `EAX` šiftuje ulijevo za dati broj bitova, čime se viši bitovi datog broja vraćaju na svoje mjesto. Preostaje da se primijeni bitsko *OR* da se kombinuju viši bitovi iz registra `EAX` i niži bitovi iz registra `EBX`.

Slijedi još jedan primjer programa koji koristi bitske operacije. U pitanju je program koji učitava nenegativan broj sa standardnog ulaza i izračunava prvi nenegativan stepen dvojke koji je veći ili jednak datom broju:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
```

```
wrfmt:  .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  x, 4                    # Broj za koji se odredjuje stepen dvojke.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se broj.
        lea     eax, x
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8

        ## Provjerava se da li je broj jednak 0. Pretpostavka je da ce
        ## biti unesen nenegativan broj.
        mov     eax, x
        cmp     eax, 0
        je zero

        ## Odredjuje se indeks bita najvece tezine u broju koji je
        ## razlicit od 0.
        bsr     ebx, eax

        ## Postavlja se ovaj bit na 0 i provjerava se da li je ostatak
        ## broja jednak 0. Ako je to slucaj, indeks odredjen u
        ## prethodnom koraku je trazeni stepen dvojke.
        btr     eax, ebx
        cmp     eax, 0
        je print

        ## U suprotnom, inkrementira se indeks i tako se dolazi do
        ## trazenog stepena dvojke.
        inc     ebx
        jmp print

zero:
        ## Ako je broj u startu jednak 0, postavlja se odgovarajuci
        ## stepen dvojke na 0.
        mov     ebx, 0

print:
        ## Stampa se izracunati stepen dvojke.
        push    ebx
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Ako je učitani broj jednak 0, odmah se rezultatu dodjeljuje vrijednost 0. U suprotnom se instrukcijom `BSR` određuje indeks bita najveće težine datog broja koji je jednak 1. Ovaj bit se u datom broju potom instrukcijom `BTR` postavlja na 0 i onda se provjerava da li je ono što je ostalo od broja jednako 0. Ako jeste, sam broj predstavlja stepen dvojke i izračunati indeks određuje taj stepen; u suprotnom, traženi stepen se dobija tako što se indeks inkrementira za 1.

6.5 Instrukcije za rad sa stringovima

IA-32 arhitektura podržava niz instrukcija koje omogućavaju efikasno obavljanje operacija nad stringovima u smislu definicije stringova date ranije u sekciji o tipovima podataka. Lista ovih instrukcija je data u tabeli 6.12.

Mnemonik	Operacija
`MOVS`	kopiranje elementa stringa
`CMPS`	poređenje elemenata stringova
`SCAS`	poređenje elementa stringa sa datom vrijednošću
`LODS`	učitavanje elementa stringa u registar
`STOS`	smještanje sadržaja registra u element stringa

Tabela 6.12: Instrukcije za rad sa stringovima.

Navedene instrukcije imaju uglavnom implicitne operande. Adrese elemenata stringova koji učestvuju u operaciji moraju biti u registrima `ESI` odn. `EDI`. Veličina operanda odn. operanada je određena sufiksom instrukcije, koji može imati vrijednost `B` za 8-bitne operande, `W` za 16-bitne operande ili `D` za 32-bitne operande. Vrijednost sa kojom se poredi tekući element stringa kod `SCAS` instrukcije se nalazi u `AL`, `AX` odn. `EAX` registru, zavisno od sufiksa instrukcije. Isti registri se respektivno koriste za smještanje vrijednosti tekućeg elementa stringa učitanog `LODS` instrukcijom odn. sadržaj ovih registara se upisuje u tekući element stringa `STOS` instrukcijom.

Po izvršenju odgovarajuće operacije, sadržaj registara `ESI` i/ili `EDI` se ažurira tako da pokazuju na naredni ili prethodni element stringa, što je kontrolisano vrijednošću *direction* flega u `EFLAGS` registru. Ako je ovaj fleg jednak 0, onda se sadržaj registara `ESI` odn. `EDI` uvećava za odgovarajući broj bajtova; ako je vrijednost *direction* flega jednaka 1 onda se sadržaj ovih registara umanjuje. Zato obavezno prije korišćenja instrukcija za rad sa stringovima treba instrukcijama `CLD` odn. `STD` postaviti vrijednost *direction* flega na 0 odn. 1 (ove instrukcije inače spadaju u grupu instrukcija za rad sa flegovima, pa će dalje u tekstu u odgovarajućoj sekciji i biti nabrojane).

Slijedi primjer programa koji demonstrira upotrebu nekih od instrukcija za rad sa stringovima. U pitanju je program koji učitava sa standardnog ulaza

i poredi dva stringa (bez korišćenja `strcmp()` funkcije iz standardne biblioteke):

```
.intel_syntax noprefix

.data
wrfmt:  .asciz  "%c\n"          # Izlazni format.

.bss
.lcomm  s0, 80                  # Prvi string.
.lcomm  s1, 80                  # Drugi string.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se prvi string. Radi jednostavnosti je za ucitavanje
        ## stringa koriscena gets() funkcija; pretpostavka je da uneseni
        ## stringovi nece biti duzi od 79 karaktera. Ovu funkciju zbog
        ## mogucnosti buffer overflow-a inace NIKAD ne treba
        ## upotrebljavati, vec umjesto nje treba koristiti fgets()
        ## funkciju.
        lea     eax, s0
        push    eax
        call gets
        add     esp, 4

        ## Ucitava se drugi string.
        lea     eax, s1
        push    eax
        call gets
        add     esp, 4

        ## Brise se direction fleg, obzirom da ce sve instrukcije za rad
        ## sa stringovima prolaziti kroz stringove prema naprijed.
        cld

        ## Odredjuje se duzina prvog stringa. String je terminiran
        ## nulom, tako da se u al registar stavlja 0, a u edi registar
        ## adresa prvog karaktera u stringu i onda se scas instrukcijom
        ## ispituju redom karakteri stringa dok se ne naidje na karakter
        ## koji je jednak 0. Duzina stringa se akumulira u registru ecx.
        xor     al, al
        lea     edi, s0
        xor     ecx, ecx
next_char0:
        scasb
        je compare
        inc     ecx
        jmp next_char0

compare:
        ## Izracunata duzina se inkrementira za 1 da bi pri
        ## poredjenju bio uzet u obzir i terminirajuci karakter.
        ## Potom se porede stringovi. U registrima esi i edi se drze
        ## adrese tekucih karaktera stringova. U petlji se u svakom
        ## koraku cmps instrukcijom porede ova dva karaktera.
```

```
        inc     ecx
        lea     esi, s0
        lea     edi, s1
next_char1:
        cmpsb
        loope next_char1

        ## Ako je petlja zavrsena jer su pronadjena dva razlicita
        ## karaktera, u registar al se smjesta karakter 'n', a u
        ## suprotnom se u isti registar smjesta karakter 'y'.
        jne not_equal
        xor     eax, eax
        mov     al, 'y'
        jmp print
not_equal:
        xor     eax, eax
        mov     al, 'n'

print:
        ## Stampa se rezultat poredjenja.
        push    eax
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Mada je pomenuto u komentarima, treba još jednom skrenuti pažnju na to da je `gets()` funkcija u gornjem programu korišćena radi jednostavnosti i da je inače nikad ne treba koristiti. Program pomoću ove funkcije učitava stringove, a potom određuje dužinu prvog stringa. Ovdje opet u tu svrhu nije korišćena `strlen()` funkcija iz standardne biblioteke da bi bila demonstrirana upotreba `SCAS` instrukcije. Potom se u petlji `CMPS` instrukcijom porede stringovi karakter po karakter. Na taj način se utvrđuje da li su isti identični ili ne, da bi na kraju na standardnom izlazu bio odštampan rezultat poređenja.

Korišćenje instrukcija za rad sa stringovima u petljama je veoma često. Zbog toga *IA-32* arhitektura podržava tzv. prefikse koji se mogu dodati ovim instrukcijama i koji omogućavaju da se petlje u kojima se koriste ove instrukcije kompaktno zapišu. Ovi prefiksi su nabrojani na tabeli 6.13. Prefiks sa nekom instrukcijom za rad sa stringovima ima isto značenje kao i ta instrukcija unutar odgovarajuće `LOOP` instrukcije; kao i kod `LOOP` instrukcija, i ovdje se u svakom koraku sadržaj registra `ECX` dekrementira.

Na taj način, petlja u kojoj se porede tekući karakteri datih stringova u gornjem primjeru:

```
next_char1:
```

Prefiks	Značenje
`REP`	ponavljanje instrukcije sve dok je $ECX \neq 0$
`REPE/REPZ`	ponavljanje instrukcije sve dok je $ECX \neq 0$ i $ZF \neq 0$
`REPNE/REPNZ`	ponavljanje instrukcije sve dok je $ECX \neq 0$ i $ZF = 0$

Tabela 6.13: Prefiksi instrukcija za rad sa stringovima.

```
cmpsb
loope next_char1
```

bi se mogla zapisati u obliku:

```
repe cmpsb
```

Obzirom da je drugi zapis znatno kompaktniji, prefikse u vezi sa instrukcijama za rad sa stringovima treba koristiti kad god je to moguće.

6.6 Instrukcije za rad sa flegovima

Jedna grupa instrukcija opšte namjene odnosi se na baratanje flegovima iz `EFLAGS` registra. Ove instrukcije su nabrojane u tabeli 6.14.

Mnemonik	Operacija
`STC`	postavljanje *carry* flega na vrijednost 1
`CLC`	postavljanje *carry* flega na vrijednost 0
`CMC`	komplementiranje vrijednosti *carry* flega
`STD`	postavljanje vrijednosti *direction* flega na 1
`CLD`	postavljanje vrijednosti *direction* flega na 0
`STI`	postavljanje vrijednosti *interrupt* flega na 1
`CLI`	postavljanje vrijednosti *interrupt* flega na 0
`LAHF`	učitavanje bajta najmanje težine `EFLAGS` registra u `AH` registar
`SAHF`	učitavanje sadržaja `AH` registra u bajt najmanje težine `EFLAGS` registra
`PUSHF`	stavljanje nižih 16 bitova `EFLAGS` registra na stek
`POPF`	skidanje 16-bitne vrijednosti sa steka u nižih 16 bitova `EFLAGS` registra
`PUSHFD`	stavljanje `EFLAGS` registra na stek
`POPFD`	skidanje 32-bitne vrijednosti sa steka u `EFLAGS` registar

Tabela 6.14: Instrukcije za rad sa flegovima.

Značenje instrukcija je jasno iz tabele; neke od ovih instrukcija su već korišćene i ranije u ovom tekstu. Treba pomenuti da su instrukcije za postavljanje vrijednosti *interrupt* flega privilegovane. Takođe, instrukcije `POPF` i `POPFD` mijenjaju samo djelimično sadržaj `EFLAGS` registra ukoliko se ne izvršavaju u privilegovanom režimu.

6.7 Preostale instrukcije opšte namjene

U ovoj sekciji će biti nabrojane instrukcije opšte namjene koje ne spadaju ni u jednu prethodno pomenutu grupu. Lista ovih instrukcija je data u tabeli 6.15.

Mnemonik	Operacija
LEA	izračunavanje efektivne adrese
XLATB	instrukcija za *lookup* u tabeli
CPUID	instrukcija za detekciju procesora
NOP	instrukcija koja ne radi ništa
UD2	nedefinisana instrukcija

Tabela 6.15: Preostale instrukcije opšte namjene.

Instrukcija LEA je već dosta korišćena; ova instrukcija izračunava efektivnu adresu date memorijske promjenljive i smješta tu adresu u zadati registar opšte namjene.

Instrukcija XLATB pristupa elementu polja bajtova u memoriji čija je adresa prvog elementa data u registru EBX, a indeks elementa u registru AL. Instrukcija učitava sadržaj ovog elementa u registar AL. Ova instrukcija je pogodna recimo za konverziju kodova karaktera i jedan takav primjer će biti prikazan niže.

Instrukcija CPUID popunjava registre EAX, EBX, ECX i EDX sa raznim informacijama o procesoru. Koje će tačno informacije biti očitane, zavisi od vrijednosti koja se prije pozivanja instrukcije nalazila u registru EAX. Tako npr. ukoliko se u EAX registar stavi 0, pa onda izvrši CPUID instrukcija, registri EBX, EDX i ECX će za *Intel*-ove procesore[4] sadržati string GenuineIntel. Posebno interesantan rezultat se dobija ako se u EAX registar stavi 1, pa onda izvrši CPUID instrukcija - registar EAX će tada sadržati oznake modela i verzije procesora[5]. CPUID instrukcija se najviše koristi da bi se odredilo da li se program izvršava na generaciji procesora koja podržava određene instrukcije. Treba imati u vidu i da je CPUID instrukcija uvedena tek na kasnijim modelima *80486* procesora, tako da je podršku za ovu instrukciju, kako je već pomenuto kada su razmatrani flegovi, potrebno prethodno detektovati pomoću *identification* flega.

Instrukcija NOP je instrukcija koja nema nikakvog efekta, osim što uvećava sadržaj EIP registra tako da ovaj pokaže na instrukciju koja slijedi NOP instrukciju. Ova instrukcija se može iskoristiti kad god je potrebno uvesti neko čekanje procesora.

[4] pored *Intel*-a postoje još neke firme koje proizvode procesore kompatibilne sa *IA-32* arhitekturom kao npr. *AMD*, *Transmeta* itd.

[5] nažalost, ove oznake nisu standardizovane između različitih proizvođača

Svi kodovi instrukcija na procesorima *IA-32* arhitekture nisu zauzeti - neki kodovi su rezervisani za buduće ekstenzije. Ukoliko procesor ipak u nizu instrukcija dobije neki kod koji nije validan, generiše se odgovarajući prekid. Instrukcija `UD2` omogućava da se eksplicitno generiše ovaj prekid i na taj način testira prateća prekidna rutina.

Slijedi primjer programa koji demonstrira kako se može iskoristiti `XLATB` instrukcija za šifrovanje poruke:

```
.intel_syntax noprefix

.data
table:                          # Tabela konverzije
        .fill   48, 1, 0
        .byte   '4','5','9','0','8','2','1','3','6','7'
        .fill   7, 1, 0
        .byte   'G','V','H','Z','U','S','O','B','M','I','K','P','J'
        .byte   'C','A','D','L','F','T','Y','E','Q','N','W','X','R'
        .fill   6, 1, 0
        .byte   'g','v','h','z','u','s','o','b','m','i','k','p','j'
        .byte   'c','a','d','l','f','t','y','e','q','n','w','x','r'
        .fill   133, 1, 0
wrfmt:  .asciz  "%s\n"          # Izlazni format.

.bss
.lcomm  msg, 80                 # Poruka koja se sifruje.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se poruka sa standardnog ulaza. Radi jednostavnosti
        ## je za ucitavanje stringa koriscena gets() funkcija;
        ## pretpostavka je da uneseni stringovi nece biti duzi od 79
        ## karaktera. Ovu funkciju zbog mogucnosti buffer overflow-a
        ## inace NIKAD ne treba upotrebljavati, vec umjesto nje treba
        ## koristiti fgets() funkciju.
        lea     eax, msg
        push    eax
        call gets
        add     esp, 4

        ## Izracunava se duzina poruke. U registar al se stavlja
        ## vrijednost koja se trazi odn. 0 (kojom je terminiran string).
        ## U registar ecx se stavlja maksimalna duzina stringa, a u
        ## registar edi adresa prvom karaktera u stringu. Potom se scasb
        ## instrukcija ponavlja dok se ne dodje do kraja stringa, a onda
        ## se od 79 (za 1 manje od vrijednosti prvobitno ucitane u
        ## registar ecx zato sto se pri primjeni rep prefiksa uvijek
        ## prvo dekrementira ovaj registar, pa tek onda ispituju uslovi
        ## izlaska iz petlje) oduzme sadrzaj registra ecx i na taj nacin
        ## se dolazi do duzine stringa.
        cld
        xor     al, al
        mov     ecx, 80
        lea     edi, msg
```

```
        repnz scasb
        sub     ecx, 79
        neg     ecx

        ## Ako je duzina poruke 0, odmah se prelazi na stampanje.
        jecxz print

        ## Sifruje se poruka. U registru ebx se drzi adresa prvog
        ## karaktera tabele sifrovanja, a registri esi i edi sadrze
        ## adresu tekuceg karaktera poruke.
        lea     ebx, table
        lea     esi, msg
        mov     edi, esi

next_char:
        ## Ucitava se tekuci karakter iz poruke i pronalazi njegova
        ## zamjena u tabeli, koja se potom upisuje namjesto karaktera u
        ## poruci, Ako je zamjena jednaka 0, znaci da nije u pitanju
        ## alfanumericki karakter i onda se samo prelazi na naredni
        ## karakter u poruci.
        lodsb
        xlatb
        cmp     al, 0
        je special
        stosb
        jmp continue
special:
        inc     edi
continue:
        loop next_char

print:
        ## Ispisuje se sifrovana poruka na standardni izlaz.
        lea     eax, msg
        push    eax
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Program učitava poruku sa standardnog ulaza (opet koristeći `gets()` funkciju radi jednostavnosti). Potom program izračunava dužinu poruke koristeći `SCAS` instrukciju sa odgovarajućim `REP` prefiksom da pronađe terminirajuću nulu u stringu. Treba uporediti ovaj metod sa određivanjem dužine stringa bez korišćenja `REP` prefiksa koja je ranije demonstrirana (treba takođe opet uočiti da bi za ovu operaciju inače bila korišćena `strlen()` funkcija iz standardne biblioteke).

Po određivanju dužine poruke, vrši se šifrovanje. U petlji se učitava jedan po jedan karakter iz poruke i pomoću `XLATB` instrukcije određuje koja mu šifra odgovara. Šifriranje je određeno tabelom koja je data u `.data` sekciji. Predviđeno je da se šifriraju samo alfanumerički karakteri, tako da tabela sadrži

šifre samo za iste, dok ostalim karakterima u tabeli odgovara vrijednost 0, koja označava da dati karater ne treba šifrovati. Tabela je razbijena na više redova u `.data` sekciji što je potpuno validno - labela se uvijek odnosi na adresu prvog narednog podatka odn. instrukcije, a sukcesivni podaci u svakoj sekciji se stavljaju u sukcesivne lokacije u memoriji. Za popunjavanje grupa nula u tabeli iskorišćena je `.fill` direktiva. Argumenti ove direktive su redom broj memorijskih elemenata koje treba popuniti, veličina u bajtovima svakog memorijskog elementa i vrijednost koju u svaki element treba upisati (zapravo, tumačenje argumenata je dosta složenije, ali kada je veličina elementa jednaka 1, onda se svodi na navedeno). Ukoliko se za tekući karakter ispostavi da je šifra očitana iz tabele različita od 0, šifra se upisuje u poruku namjesto datog karaktera, u suprotnom se samo prelazi na naredni karakter. Nakon što je kompletna poruka šifrirana, ista se štampa na standardni izlaz.

Slijedi primjer programa koji demonstrira kako se `CPUID` instrukcijom može očitati identifikacioni string proizvođača procesora:

```
.intel_syntax noprefix

.data
wrfmt:  .asciz  "%s\n"          # Izlazni format.

.bss
.lcomm  id, 13                  # Identifikacioni string proizvodjaca procesora.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Provjerava se da li procesor podrzava cpuid
        ## instrukciju. Sadrzaj eflags registra se preko steka prebacuje u
        ## registar eax, a onda se testira i komplementira vrijednost
        ## bita koji odgovara identification flegu.  Prethodna vrijednost
        ## flega se pamti u bl registru.  Potom se nova vrijednost vraca
        ## u eflags registar, da bi se odmah zatim na isti nacin kao
        ## gore ocitala nova vrijednost eflags registra u registar eax
        ## odnosno identification flega u registar bh.  Na kraju se
        ## porede stara i nova vrijednost flega iz registara bl i bh.
        ## Ukoliko se vrijednosti razlikuju, znaci da je promjena
        ## vrijedosti identification flega uspjela, odn. da procesor
        ## podrzava cpuid instrukciju.
        pushfd
        pop     eax
        btc     eax, 21
        setc    bl
        push    eax
        popfd
        pushfd
        pop     eax
        bt      eax, 21
        setc    bh
        cmp     bh, bl
```

```
        je done

        ## Ocitava se identifikacioni string proizvodjaca procesora.
        xor     eax, eax
        cpuid

        ## Prebacuje se identifikacioni string u memoriju. Prva 4
        ## karaktera ovog stringa se po izvrsenju cpuid instrukcije
        ## nalaze u registru ebx, naredna 4 u registru edx i poslednja 4
        ## u registru ecx.
        mov     id, ebx
        mov     id+4, edx
        mov     id+8, ecx
        mov     byte ptr id+12, 0

        ## Stampa se identifikacioni string.
        lea     eax, id
        push    eax
        lea     eax, wrfmt
        push    eax
        call printf
        add     esp, 8

done:
        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

Program prvo pokušava da promijeni vrijednost *identification* flega u `EFLAGS` registru. Ako u tome uspije znači da procesor podržava `CPUID` instrukciju, koju program potom poziva pošto je prethodno u `EAX` registar stavio vrijednost 0 što znači da se želi očitati identifikacioni string proizvođača. Po 4 karaktera ovog stringa bivaju smješteni respektivno u registre `EBX`, `EDX` i `ECX`, pri čemu je uvijek prvi karakter u bajtu najmanje težine registra i tako redom do četvrtog karaktera koji je u bajtu najveće težine. Identifikacioni string proizvođača se potom formira u odgovarajućem polju karaktera u memoriji tako što se sadržaj ovih registara kopira u odgovarajuće djelove polja. Obzirom na opisani raspored karaktera u registrima i na *little-endian* arhitekturu koja smješta bajtove manje težine registra na niže adrese u memoriji, sadržaj svakog registra se može prebaciti u memoriju jednim 32-bitnim memorijskim transferom i svaki karakter će pritom doći na svoje mjesto. Identifikacioni string se terminira nulom. Treba uočiti kako u odgovarajućoj `MOV` instrukciji nije implicitno određeno kolika je širina transfera, pa je onda iskorišćen `BYTE PTR` prefiks nad odgovarajućim operandom. Nakon što je identifikacioni string proizvođača upisan u memoriju, isti se štampa na standardni izlaz.

Na kraju treba pomenuti da bi se u grupu instrukcija opšte namjene koja je razmatrana u ovoj sekciji mogle ubrojati `ENTER` i `LEAVE` instrukcije koje se koriste u potprogramima. Ove instrukcije će međutim biti prikazane u narednom poglavlju, koje je u potpunosti posvećeno potprogramima.

Poglavlje 7

Potprogrami na *IA-32* arhitekturi

Osnovni razlog za korišćenje potprograma u asembleru je isti kao i u višim programskim jezicima - da se izdvoji kod koji se više puta javlja u programu na jedno mjesto i na taj način napravi program lakšim za održavanje. U vezi sa pozivanjem potprograma važno je razmotriti pitanja načina pozivanja, prenošenja argumenata i vraćanja rezultata iz potprograma.

7.1 Pozivanje potprograma

Pozivanje potprograma se ne razlikuje od običnog bezuslovnog skoka. Međutim, potprogram po svom završetku mora da zna da se vrati na instrukciju koja se nalazi iza instrukcije kojom je potprogram pozvan. Postoje različiti načini da se ovo obezbijedi, jedan način je da se upiše povratna adresa u neki registar koji potprogram tokom svog izvršavanja neće mijenjati, da bi se onda po svom završetku jednom instrukcijom skoka koristeći ovu adresu vratio na mjesto na kome je poziv napravljen. Međutim, takav način čuvanja adrese ne odgovara rekurzivnim programima, jer bi pri rekurzivnom pozivu bila uništena povratna adresa za tekući poziv. Na *IA-32* arhitekturi najpogodnije je čuvati povratnu adresu na steku. *IA-32* arhitektura direktno podržava ovakav način čuvanja povratne adrese pri pozivu potprograma kroz posebnu instrukciju `CALL` koja je slična `JMP` instrukciji, ali prije nego što se izvrši skok ovdje se povratna adresa stavlja na stek. Komplementarna sa ovom instrukcijom je `RET` instrukcija kojom se završava potprogram i koja prvo skida sa steka povratnu adresu, a potom vrši bezuslovni skok na tu adresu. Ovaj metod pozivanja potprograma je jednostavan i omogućava lako ugnježdavanje tih poziva; zbog toga je ovo uobičajeni način pozivanja potprograma na *IA-32* arhitekturi. Ono što je važno kod ovog metoda jeste

da mora da se obezbijedi da sadržaj steka na kraju potprograma bude isti kao na njegovom početku, tj. da ako ima `PUSH` instrukcija u potprogramu da ih prati odgovarajući broj `POP` instrukcija, kao i da se eventualno nekom greškom ne prebriše nekim drugim podatkom povratna adresa.

7.2 Prenošenje argumenata u potprograme

Za prenos argumenata u potprogram važi slično rezonovanje kao za čuvanje povratne adrese. Jedan način prenosa argumenata bi bio preko registara. U pitanju je najbrži mogući način za prenos argumenata; međutim problem sa *IA-32* arhitekturom je što ima malo registara tako da na ovaj način nije moguće prenijeti veliki broj argumenata, a takođe ako se argumenti prenose preko registara i ako su njihove vrijednosti potrebne tokom većeg dijela potprograma onda ostane malo registara za rad samog potprograma. Razmatranjem i drugih alternativa (smještanje argumenata u memoriju na neku fiksnu adresu, smještanje argumenata u memoriju na proizvoljnu adresu i prenošenje te adrese preko registra i sl.), opet se pokazuje da je na *IA-32* arhitekturi najzgodniji način prenosa argumenata potprogramu preko steka. Prije poziva potprograma argumenti se redom stavljaju na stek, da bi nakon toga `CALL` instrukcija stavila na stek povratnu adresu. Programer je slobodan da izabere da li će na stek stavljati podatke ili adrese podataka; naravno, ako potprogram treba da izmijeni podatak onda se mora na stek staviti njegova adresa. Takođe treba napomenuti da zbog poravnanja steka treba voditi računa da se samo 32-bitne vrijednosti stavljaju na stek.

Potprogram pristupa argumentima direktno na steku, bez korišćenja `POP` instrukcije. Razlog ovome je što se argumenti stavljaju na stek prije povratne adrese koja mora ostati na steku do kraja potprograma. Potencijalni problem je međutim što se `ESP` registar unutar potprograma može promijeniti, tako da se argumenti ne mogu lako adresirati relativno u odnosu na sadržaj ovog registra. Zato je standardni mehanizam pristupa argumentima potprograma preko `EBP` registra - ovaj registar se odmah po ulasku u potprogram stavlja na stek i u njega se kopira vrijednost `ESP` registra. Sadržaj registra `EBP` se onda više ne mijenja tokom potprograma, pa se argumentima može lako pristupati relativno u odnosu na isti. Nema posebnog razloga da se `EBP` registar odabere za ovu svrhu, ali je takva konvencija, a i nad `EBP`-om nisu moguće sve operacije koje su moguće nad registrima opšte namjene tako da je manja šteta da se on koristi za ovu namjenu. Na kraju potprograma treba sa steka skinuti sačuvani sadržaj `EBP` registra i onda može normalno da se pozove `RET` instrukcija koja će na steku pronaći povratnu adresu.

Po završetku potprograma, argumente treba skinuti sa steka. Ovo je najlakše uraditi tako što će se prosto dodati ukupna dužina argumenata na `ESP` registar - to je brže nego da se argumenti skidaju sekvencom `POP` instrukcija,

a i `POP` instrukcija bi zahtijevala da se ti argumenti negdje smještaju. Ovo može uraditi ili potprogram ili kod koji je pozvao potprogram. Potprogram može skinuti argumente sa steka samo ukoliko je broj argumenata fiksan. Za ovakve slučajeve, *IA-32* arhitektura podržava posebnu formu `RET` instrukcije koja prima jedan operand, pri čemu je taj operand broj koji treba dodati na sadržaj `ESP` registra nakon što se sa njega skine povratna adresa. Ako potprogram može da primi promjenljiv broj argumenata (kao npr. *C* funkcije `printf()` ili `scanf()`) onda samo kod koji je pozvao potprogram može da zna koliko je argumenata na steku, pa taj kod treba i da ih skine. Pošto je drugo rješenje generalnije, ono se češće i koristi.

7.3 Vraćanje rezultata iz potprograma

Što se tiče vraćanja rezultata iz potprograma, pošto je obično u pitanju jedna vrijednost (ili nijedna), tu je najraširenija konvencija da se rezultat vraća preko registra, pri čemu se najčešće za tu svrhu koristi registar `EAX`. U svim prethodnim primjerima, `main()` funkcija je vraćala 0 na ovaj način - ovo je postizano instrukcijom `XOR EAX, EAX` u epilogu svake od njih.

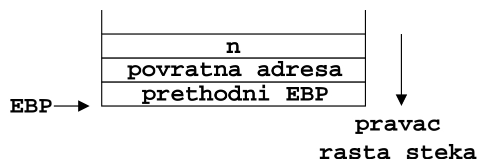
Slijedi primjer programa koji demonstrira opisane tehnike pozivanja potprograma:

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
wrtfmt: .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  n, 4                    # Broj ciji se faktorijel racuna.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se broj. Pretpostavka je da ce biti unesen
        ## nenegativan broj.
        lea     eax, n
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8

        ## Poziva se funkcija za racunanje faktorijela.
        mov     eax, n
        push    eax
        call fact
```

```
        ## Stampa se rezultat.
        push    eax
        lea     eax, wrtfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret

        ## Funkcija fact() izracunava faktorijel datog broja. Argumenti
        ## funkcije su:
        ##   [ebp+8] - broj ciji se faktorijel racuna
        ## Pretpostavke su da je broj ciji se faktorijel racuna
        ## nenegativan i da vrijednost faktorijela moze stati u oznaceni
        ## 32-bitni broj.
fact:
        ## Prolog funkcije.
        push    ebp
        mov     ebp, esp

        ## Ako je broj ciji se faktorijel trazi manji ili jednak 1,
        ## vraca se 1 kao rezultat funkcije.
        mov     eax, [ebp+8]
        cmp     eax, 1
        jg recursion
        mov     eax, 1
        jmp     done

recursion:
        ## U suprotnom se rekurzivno poziva funkcija pri cemu joj se kao
        ## argument prenosi za 1 umanjen tekuci argument.
        dec     eax
        push    eax
        call fact

        ## Mnozi se tekuci argument sa rezultatom rekurzivnog
        ## poziva. Pretpostavlja se da ce rezultat stati u 32 bita.
        imul    dword ptr [ebp+8]

done:
        ## Epilog funkcije.
        pop     ebp
        ret     4
```

Program učitava broj sa standardnog ulaza i izračunava i štampa njegov faktorijel. Vrijednost faktorijela se izračunava rekurzivnim potprogramom `fact()`. Potprogrami se u asembleru naznačavaju normalnim labelama. Na početku potprograma se na stek stavlja sadržaj registra `EBP`, a potom se u ovaj registar kopira sadržaj registra `ESP`. Na taj način, vrh steka u tom trenutku ima oblik kao na slici 7.1.

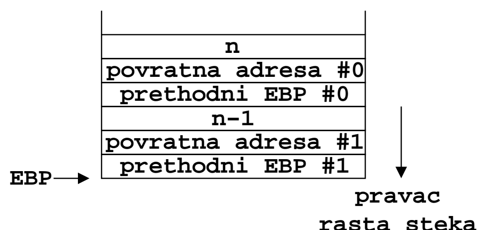
Sa slike je jasno da se adresa argumenta dobija tako što se sadržaju `EBP` registra doda 8. Ovakvo adresiranje spada u složenija memorijska adresiranja koja uključuju registar kao *base* odn. *index* komponentu adrese. Radi



Slika 7.1: Izgled steka nakon osnovnog poziva funkcije `fact()`.

nedvosmislenosti u ovakvim situacijama, *GNU* asembler zahtijeva da se oko adrese stave uglaste zagrade, pa se tako npr. u ovom slučaju argument kroz potprogram adresira kao `[EBP+8]`.

Potprogram potom provjerava da li je argument veći od 1 i ako nije odmah upisuje 1 u registar `EAX` kao vrijednost koja će biti vraćena kao rezultat (pretpostavka je da je broj koji je prenesen kao argument nenegativan). U suprotnom, vrši se rekurzivni poziv potprograma. Argument se umanjuje za 1 i stavlja se na stek, a onda se opet poziva potprogram. Vrh steka nakon ovog poziva ima oblik kao na slici 7.2.



Slika 7.2: Izgled steka nakon rekurzivnog poziva funkcije `fact()`.

Indeksi na ovoj slici u oznakama za povratnu adresu odn. prethodnu vrijednost registra `EBP` se odnose na poziv. Može se uočiti kako svaki rekurzivni poziv kreira na steku jedan blok podataka koji se odnosi na tekući poziv potprograma. Ovaj blok se obično označava terminom *stack frame*.

Sa rekurzivnim pozivima se nastavlja sve dok vrijednost argumenta potprograma `fact()` ne postane jednaka 1. Tada se, kako je gore već opisano, rekurzija preskače i kroz registar `EAX` se vraća 1 kao rezultat. Kada se poziv potprograma završi, odgovarajući *frame* biva uništen. Naime, prvo se `POP` instrukcijom povrati sadržaj `EBP` registra od prije poziva potprograma. Zatim se `RET` instrukcijom skine povratna adresa sa steka. Takođe se, obzirom da ova instrukcija ima operand sa vrijednošću 4, dodaje `ESP` registru broj 4 čime se poništava argument na steku.

U tom trenutku `EBP` registar ponovo ukazuje na *stack frame* prethodnog poziva, tako da je `[EBP+8]` opet vrijednost argumenta u tom rekurzivnom pozivu koja se sada može pomnožiti sa izračunatom vrijednošću faktorijela za broj 1 koja se nalazi u registru `EAX`. Rezultat množenja ostaje u registru

EAX i preko njega se vraća kao rezultat potprograma (ignorišu se viša 32 bita rezultata iz registra EDX, jer je pretpostavka da broj čiji se faktorijel računa nije previše veliki, tako da će njegov faktorijel moći da stane u 32 bita). Po povratku u prethodni poziv, opisani slijed događaja se ponavlja sve dok se ne dođe do osnovnog poziva potprograma. Stek tada opet ima oblik kao na slici 7.1. Još jednom se množi argument sa steka sa vrijednošću koja je vraćena iz rekurzivnog poziva i potom se tekući poziv završava na isti način kao i rekurzivni pozivi prije njega. Glavni program tada dobija vrijednost faktorijela u registru EAX i može da je odštampa na standardni izlaz.

7.4 Lokalne promjenljive u potprogramu

Stek je takođe vrlo pogodan za čuvanje lokalnih promjenljivih potprograma; ovim promjenljivima se u tom slučaju može pristupati preko EBP registra na sličan način kao argumentima. U tom cilju je potrebno na početku potprograma, nakon što se kreira *stack frame*, alocirati prostor za lokalne promjenljive na steku, što se može uraditi prosto tako što se od ESP registra oduzme broj bajtova koji je potreban za lokalne promjenljive. Lokalne promjenljive se takođe smatraju dijelom *stack frame*-a. Na završteku potprograma, prije nego što se krene u vraćanje prethodnog sadržaja registra EBP, treba ESP registru dodati broj bajtova koje su zauzimale lokalne promjenljive, čime se iste praktično uništavaju.

Slijedi primjer programa koji koristi stek za lokalne promjenljive:

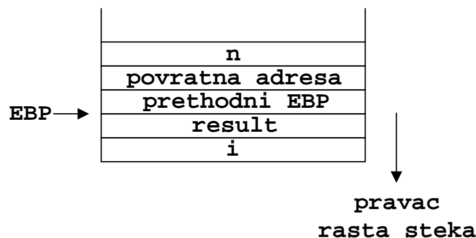
```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
wrtfmt: .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  n, 4                    # Broj ciji se faktorijel racuna.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se broj.
        lea     eax, n
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
        add     esp, 8

        ## Poziva se funkcija za racunanje faktorijela.
```

```
        mov     eax, n
        push    eax
        call fact

        ## Stampa se rezultat.
        push    eax
        lea     eax, wrtfmt
        push    eax
        call printf
        add     esp, 8

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret

        ## Funkcija fact() izracunava faktorijel datog broja. Argumenti
        ## funkcije su:
        ##   [ebp+8] - broj ciji se faktorijel racuna
        ## Lokalne promjenljive u funkciji su:
        ##   [ebp-4] - tekuca vrijednost rezultata
        ##   [ebp-8] - brojac u petlji
        ## Pretpostavke su da je broj ciji se faktorijel racuna
        ## nenegativan i da vrijednost faktorijela moze stati u oznaceni
        ## 32-bitni broj.
fact:
        ## Prolog funkcije.
        push    ebp
        mov     ebp, esp
        sub     esp, 8

        ## U registar ecx se prebacuje broj ciji se faktorijel racuna.
        mov     ecx, [ebp+8]

        ## Inicijalizuju se rezultat i brojac u petlji.
        ##   /* result = 1; i = 2; */
        mov     dword ptr [ebp-4], 1
        mov     dword ptr [ebp-8], 2

next:
        ## Provjerava se uslov izlaska iz petlje.
        ##   /* while (i <= n) { */
        cmp     [ebp-8], ecx
        jg done

        ## Mnozi se rezultat sa tekucom vrijednoscu brojaca.
        ##   /* result *= i; */
        mov     eax, [ebp-4]
        imul    dword ptr [ebp-8]
        mov     [ebp-4], eax

        ## Inkrementira se brojac.
        ##   /* i++; } */
        inc     dword ptr [ebp-8]
        jmp next

done:
        ## Stavlja se rezultat u registar eax.
        mov     eax, [ebp-4]

        ## Epilog funkcije.
```

```
add     esp, 8
pop     ebp
ret     4
```

Program predstavlja iterativnu verziju potprograma za računanje faktorijela. Funkcija `main()` je ista kao u prethodnom primjeru, izmijenjena je samo funkcija `fact()`. Ova funkcija koristi dvije lokalne promjenljive - jednu za akumulisanje faktorijela i drugu kao brojač u petlji.

Po ulasku u funkciju, alocira se 8 bajtova na steku za lokalne promjenljive. Stek nakon ove alokacije ima oblik kao na slici 7.3.



Slika 7.3: Izgled steka nakon alokacije lokalnih promjenljivih.

Na slici su lokalne promjenljive označene imenima koja su za njih korišćena u pseudokodu u komentarima programa. Promjenljiva koja predstavlja rezultat izračunavanja faktorijela se inicijalizuje na 1, a brojač na 2 (ako je broj manji od 2, njegov faktorijel je jednak 1). Potom se u uslovu petlje poredi brojač sa zadatim brojem (koji se drži u registru `ECX`), pa ako još nije premašen zadati broj, množi se rezultat sa tekućom vrijednošću brojača i inkrementira se brojač. Po završetku petlje rezultat se prebacuje u registar `EAX`, uništavaju se lokalne promjenljive sa steka i normalno se završava funkcija.

Treba uočiti da se ova iterativna verzija funkcije za računanje faktorijela mogla napisati i samo korišćenjem registara, što inače treba činiti kad god je to moguće. Pošto je pristup memoriji znatno sporiji od pristupa registrima, lokalne promjenljive treba uvoditi samo kad je zaista neophodno.

7.5 Standardni prolog i epilog potprograma

Opisani način rada sa parametrima i lokalnim promjenljivim programa je toliko uobičajen da *IA-32* arhitektura sadrži posebne instrukcije koje pojednostavljuju prolog i epilog potprograma. Tako instrukcija `ENTER L, 0` (gdje `L` mora biti cjelobrojna konstanta) zamjenjuje sekvencu:

```
push    ebp
mov     ebp, esp
sub     esp, L
```

u prologu, dok instrukcija `LEAVE` zamjenjuje sekvencu:

```
mov     esp, ebp
pop     ebp
```

u epilogu potprograma.

Drugi operand `ENTER` instrukcije može biti različit od 0, u kom slučaju procesor prije odvajanja prostora za lokalne promjenljive stavlja na stek pointere na *stack frame*-ove prethodnih ugnježdenih poziva potprograma. Na taj način, lokalne promjenljive i argumenti ovih potprograma su vidljivi tekućem potprogramu.

Primjeri korišćenja instrukcija `ENTER` i `LEAVE` se mogu naći u `main()` funkciji u svim ranije prezentiranim programima. Takođe, ove instrukcije će u ostatku teksta biti dosljedno korišćene za prolog i epilog potprograma.

7.6 Čuvanje sadržaja registara u potprogramu

Dobro dizajniran potprogram je dužan da sačuva sadržaj svih registara koje mijenja. To se obično radi nizom `PUSH` instrukcija nakon prologa koga prati niz `POP` instrukcija prije epiloga funkcije. Pritom treba voditi računa da se sačuvani sadržaj vrati u registre obrnutim redom. Duži potprogrami mogu za ovu svrhu iskoristiti i ranije pomenute `PUSHA` i `POPA` instrukcije.

Obzirom da se za čuvanje svih registara na steku tokom izvršavanja potprograma ipak izgubi određeno vrijeme, postoje različite konvencije u pogledu toga koji se registri čuvaju, a koji ne. Jedna takva konvencija biće predstavljena nešto dalje u tekstu kada bude riječi o *C* konvencijama za pozivanje potprograma.

7.7 Diskusija opisanog načina pozivanja potprograma

Osnovna prednost predstavljenog standardnog načina pozivanja potprograma na *IA-32* arhitekturi je što tako napisani potprogrami ne pristupaju nikakvim globalnim promjenljivim (takođe i što prostor za lokalne promjenljive i parametre zauzimaju samo tokom izvršavanja potprograma). Zbog toga se ovakav kod označava terminom *reentrant*, što označava da kod može biti pozivan rekurzivno, kao i da isti kod istovremeno može izvršavati više niti istog procesa.

Adresiranje argumenata potprograma i lokalnih promjenljivih direktno na steku pomoću `EBP` registra je elegantno, ali ne i previše čitljivo rješenje.

Većina asemblera obezbjeđuje direktive za davanje alternativnih imena (tzv. *alias*-a) nekim izrazima, pa se onda to može iskoristiti za davanje deskriptivnih imena argumentima (`[EBP+8]`, `[EBP+12]` itd.) odn. lokalnim promjenljivim (`[EBP-4]`, `[EBP-8]` itd.). *GNU* asembler nažalost nema podršku za ove direktive[1], tako da je jedina mogućnost da se uvede ovako nešto da se koristi *C* preprocesor. Ako se asemblerski fajl propušta kroz preprocesor (uobičajena ekstenzija za asemblerske fajlove kojima je to potrebno je .S), programeru je na raspolaganju `#define` direktiva sa svim njenim mogućnostima; mogu se praviti i razni komplikovani makroi što je takođe često mogućnost koju drugi asembleri ističu kao prednost[2]. Nedostatak ovog metoda je što se prilikom debagovanja vidi kod nakon preprocesiranja, tj. ne vide se npr. imena koja su eventualno specificirana `#define` direktivama.

7.8 Modularna organizacija programa

Asemblerski kod može biti razbijen na više fajlova. Pri pisanju ovakvog koda važno je samo voditi računa da simboli iz datog fajla koji treba da budu pristupačni u drugim fajlovima budu nabrojani iza `.global` direktive; kod asemblera su naime svi simboli privatni za dati fajl osim ako se ne eksportuju na ovaj način. Sa druge strane, *GNU* asembler smatra sve simbole koje ne pronađe u datom fajlu eksternim i ostavlja linkeru da razriješi takve reference (primjer za to je bilo svako prethodno korišćenje `scanf()` ili `printf()` funkcije). Treba znati da je ovo suprotno od konvencije u većini drugih asemblera gdje se eksterni simboli moraju navesti pod tzv. *extern* direktivom inače će asembler prijaviti sintaksnu grešku; u *GNU* asembleru postoji odgovarajuća direktiva (zove se `.extern`) ali ista nema nikakvog efekta.

Prethodni primjer sa iterativnim računanjem faktorijela bi mogao biti razbijen na dva fajla tako da u jednom fajlu bude izdvojena funkcija `fact()`:

```
.intel_syntax noprefix

.text
.global fact

        ## Funkcija fact() izracunava faktorijel datog broja. Argumenti
        ## funkcije su:
        ##   [ebp+8] - broj ciji se faktorijel racuna
        ## Lokalne promjenljive u funkciji su:
        ##   [ebp-4] - tekuca vrijednost rezultata
        ##   [ebp-8] - brojac u petlji
        ## Pretpostavke su da je broj ciji se faktorijel racuna
        ## nenegativan i da vrijednost faktorijela moze stati u oznaceni
        ## 32-bitni broj.
fact:
```

[1] ustvari ima, ali to u tekućim verzijama ne funkcioniše kako treba

[2] za ovo takođe postoje odgovarajuće direktive u *GNU* asembleru, ali opet te direktive u ovom trenutku ne rade kako treba

```
        ## Prolog funkcije.
        enter   8, 0

        ## U registar ecx se prebacuje broj ciji se faktorijel racuna.
        mov     ecx, [ebp+8]

        ## Inicijalizuju se rezultat i brojac u petlji.
        ##   /* result = 1; i = 2; */
        mov     [ebp-4], dword ptr 1
        mov     [ebp-8], dword ptr 2

next:
        ## Provjerava se uslov izlaska iz petlje.
        ##   /* while (i <= n) { */
        cmp     [ebp-8], ecx
        jg done

        ## Mnozi se rezultat sa tekucom vrijednoscu brojaca.
        ##   /* result *= i; */
        mov     eax, [ebp-4]
        imul    dword ptr [ebp-8]
        mov     [ebp-4], eax

        ## Inkrementira se brojac.
        ##   /* i++; } */
        inc     dword ptr [ebp-8]
        jmp next

done:
        ## Stavlja se rezultat u registar eax.
        mov     eax, [ebp-4]

        ## Epilog funkcije.
        leave
        ret
```

dok bi u drugom fajlu bio izdvojen glavni program (tj. `main()` funkcija):

```
.intel_syntax noprefix

.data
rdfmt:  .asciz  "%d"            # Ulazni format.
wrtfmt: .asciz  "%d\n"          # Izlazni format.

.bss
.lcomm  n, 4                    # Broj ciji se faktorijel racuna.

.text
.global main

main:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Ucitava se broj.
        lea     eax, n
        push    eax
        lea     eax, rdfmt
        push    eax
        call scanf
```

```
add     esp, 8

## Poziva se funkcija za racunanje faktorijela.
mov     eax, n
push    eax
call fact
add     esp, 4

## Stampa se rezultat.
push    eax
lea     eax, wrtfmt
push    eax
call printf
add     esp, 8

## Epilog funkcije.
popa
xor     eax, eax
leave
ret
```

Treba uočiti kako je u prvom fajlu funkcija `fact()` eksportovana `.global` direktivom. Dalje, treba uočiti da su, kako je najavljeno, sada korišćene `ENTER` i `LEAVE` instrukcije za prolog odn. epilog potprograma. Takođe, u funkciji `fact()` je demonstrirana druga konvencija u pogledu skidanja argumenata sa steka - `RET` naredba sada nema argumenata, tako da je dužnost glavnog programa koji poziva `fact()` funkciju da počisti argumente sa steka po povratku iz iste.

Prilikom prevođenja projekta koji se sastoji od više asemblerskih fajlova, asemblerom treba prevesti svaki od tih fajlova ponaosob u odgovarajući objektni fajl, a nakon toga linkerom od ovih objektnih fajlova napraviti jedan izvršni fajl. Linker pritom spaja sve sekcije sa istim imenom (npr. sve `.data` ili sve `.text` sekcije) u jednu sekciju i vrši sve neophodne relokacije. Na potpuno isti način funkcioniše prevođenje projekta koji se sastoji od izmiješanih asemblerskih fajlova i fajlova na višem programskom jeziku (odn. i samo od fajlova na višem programskom jeziku), s tom razlikom što fajlove na višem programskom jeziku mora prvo prevodilac preko asemblerskog prevesti u objektni kod.

Poglavlje 8

Povezivanje asemblerskog i *C* koda

Ranije je već pomenuto da pisanje kompletnih progama na asembleru danas ima malo smisla. Asemblersko programiranje je primarno od interesa u vezi sa programiranjem na nekom višem programskom jeziku, pri čemu se obično glavnina programa piše na višem programskom jeziku, a onda se samo kritični djelovi koda programiraju na asembleru. Posebno je *C* jezik, zbog slične ciljne grupe problema čijem je rješavanju namijenjen (sistemsko programiranje), pogodan za ovakav način korišćenja zajedno sa asemblerom. Obično su osnovne jedinice u kojima se pišu programski moduli potprogrami, pa će stoga ovo poglavlje biti posvećeno prije svega pozivanju asemblerskih potprograma iz *C*-a odn. pozivanju funkcija napisanih na *C*-u iz asemblera. Takođe će biti razmotreni i sistemski pozivi iz asemblera. Pred kraj poglavlja će biti prikazano i ubacivanje sekvenci asemblerskog koda kroz *C* kod odn. tzv. *inline* asemblersko programiranje, a takođe će biti napravljena jedna digresija da se nešto kaže o formatu izvršnih fajlova.

8.1 *C* konvencije za pozivanje potprograma

U prethodnom poglavlju je predstavljena podrška koju *IA-32* arhitektura pruža za pozivanje potprograma. Vidjelo se da praktično za svaki aspekt pozivanja potprograma postoji dva ili više rješenja. Prevodioci koji prevode kod napisan u višim programskim jezicima na asemblerski kod se odlučuju za neka od tih rješenja i onda ih dosledno primjenjuju. Ukoliko se zna kakve konvencije poštuje neki prevodilac, onda se može lako miješati asemblerski kod sa kodom napisanim na višem programskom jeziku. U ovoj sekciji će biti nabrojane konvencije koje skoro bez izuzetka poštuju *C* prevodioci za *IA-32* arhitekturu.

C program stavlja argumente potprograma na stek i to počev od poslednjeg argumenta prema prvome. Skidanje argumenata sa steka dužan je da obavi kod koji poziva potprogram. Rezultat potprograma se vraća u registru `EAX` ukoliko je u pitanju cio broj ili pokazivač, odn. u registru `ST(0)` numeričkog koprocesora ukoliko je u pitanju realan broj. *C* prevodioci označavaju adresu na kojoj počinje kod potprograma labelom koja ima isto ime kao potprogram[1], tako da se to ime može koristiti prilikom pozivanja. Na *C*-u je konvencija da potprogrami očuvaju sadržaj registara `EBX`, `ESI`, `EDI` i `EBP` (prva tri registra ne smiju biti promijenjena jer ih *C* prevodioci koriste za registarske promjenljive, dok registar `EBP` treba da bude sačuvan zbog pristupa tekućeg *frame*-a na steku), kao i segmentnih registara.

Većina *C* prevodilaca obezbjeđuje ili preko odgovarajućih prekidača u komadnoj liniji ili dodavanjem nestandardnih ključnih riječi u kod alternativne načine da se neka od ovih pravila zaobiđu ukoliko je tako nešto baš neophodno. Tako npr. *gcc* prevodilac omogućuje da se neke funkcije deklarišu tako da budu prevedene na način da same skidaju svoje argumente sa steka; takođe da neke funkcije budu deklarisane tako da budu prevedene sa prenošenjem argumenata preko registara umjesto preko steka. Ovakva rješenja međutim treba izbjegavati.

U prethodnom dijelu ovog teksta bilo je dosta primjera pozivanja *C* funkcija iz asemblera, iz kojih se može vidjeti da je za tako nešto dovoljno poznavati i slijediti nabrojane konvencije. Pozivanje asemblerskih potprograma iz *C* koda je podjednako jednostavno, jedino o čemu treba voditi računa jeste da se asemblerski potprogram napiše da slijedi opisane konvencije. Tako je npr. funkcija `fact()` u poslednjem primjeru napisana da potpuno slijedi opisane konvencije, pa se onda glavni program umjesto na asembleru može napisati na *C*-u:

```
#include <stdio.h>

extern int      fact(int);

/*
 * Program sluzi za testiranje funkcije fact(). Program ucitava broj sa
 * standardnog ulaza, a zatim poziva funkciju fact() da izracuna faktorijel
 * datog broja i stampa rezultat na standardni izlaz.
 */
int
main()
{
        int             n;      /* Broj ciji se faktorijel racuna. */

        /* Ucitava se broj sa standardnog ulaza. */
        scanf("%d", &n);

        /* Izracunava se faktorijel i stampa na standardni izlaz. */
        printf("%d\n", fact(n));
```

[1] neki *Windows* prevodioci ne slijede ovu konvenciju

```
        return 0;
}
```

Na početku programa je funkcija `fact()` deklarisana kao eksterna; potom se ova funkcija poziva iz *C* koda na sasvim uobičajen način.

Primjeri u ostatku ovog teksta uglavnom će se sastojati od *C* koda za učitavanje argumenata i ispis rezultata (obzirom da to nisu vremenski kritične operacije, iste se rijetko izvode iz asemblerskog koda kao što je dosad rađeno) i asemblerskog potprograma koji će implementirati rješavanje datog problema. *C* kod zbog svoje jednostavnosti i uštede prostora uglavnom neće biti navođen u tekstu, već može da se nađe na odgovarajućoj *Web* strani [1].

8.2 Sistemski pozivi iz asemblerskog koda

Sistemski pozivi predstavljaju interfejs prema kernelu operativnog sistema; direktno pozivanje funkcija kernela iz asemblerskog koda umjesto njihovog pozivanja preko standardne biblioteke je često poželjno radi maksimalne brzine. Ovdje međutim ne postoje konvencije koje bi slijedili svi operativni sistemi. Na taj način, ono što bude rečeno u ovoj sekciji odnosiće se respektivno samo na *Linux* odn. *FreeBSD* operativni sistem.

Pri izvršenju sistemskog poziva, procesor treba da pređe iz korisničkog u sistemski režim izvršavanja. Postoji precizno kontrolisani mehanizam za prelazak iz korisničkog nivoa u sistemski nivo i to preko tzv. *call gate*-ova. Ovaj mehanizam je međutim vezan za segmentni podsistem za organizaciju memorije koju *UNIX*-i ignorišu, tako da se na *UNIX*-ima koristi alternativni metod za prelazak na sistemski nivo, a to je upravo preko softverskih prekida, obzirom da se rutine za obradu prekida uvijek izvršavaju u sistemskom nivou. Sistemski pozivi *UNIX* kernelu idu preko softverskog prekida, i to preko `INT 0x80` instrukcije.

Konvencija na *Linux*-u je da se argumenti prenose preko registara i to tako što se u registar `EAX` stavlja broj sistemskog poziva, a onda u registre `EBX`, `ECX`, `EDX`, `ESI` i `EDI` redom argumenti poziva (ako ima više od 5 argumenata sistemskog poziva, isti se stavljaju u memoriju, a u registar `EBX` se stavlja adresa memorijske lokacije na kojoj počinje lista argumenata). Po povratku iz sistemskog poziva, registar `EAX` će sadržati povratnu vrijednost.

Na *FreeBSD*-u je konvencija da se argumenti sistemskom pozivu prenose preko steka[2], baš kao i običnoj funkciji. Takođe kao i kod poziva obične funkcije, dužnost je koda koji vrši sistemski poziv da po završenom pozivu

[2] treba napomenuti da *FreeBSD* ima dodatnu podršku i za prenošenje argumenata sistemskom pozivu na isti način kao i *Linux*, preko registara

počisti stek. Broj sistemskog poziva se i ovdje stavlja u *EAX* registar. *FreeBSD* zahtijeva i da se na stek, nakon argumenata, stavi još jedan dodatni 32-bitni podatak (bilo koja vrijednost); ovaj podatak treba takođe da bude očišćen sa steka nakon što se sistemski poziv završi.

Brojevi sistemskih poziva su izlistani na *Linux*-u u fajlu `/usr/include/asm/unistd.h`, a na *FreeBSD*-u u fajlu `/usr/include/sys/syscall.h`. Iz ovih fajlova se recimo može vidjeti da je broj `exit()` sistemskog poziva jednak 1 ili npr. broj `write()` sistemskog poziva 4 itd.

Slijedi primjer programa koji štampa poruku na standardni izlaz koristeći `write` sistemski poziv:

```
.intel_syntax noprefix

.data
msg:    .asciz  "Hello, world!\n" # Poruka koju treba ispisati.

.text
.global main

main:
        ## Prolog funkcije.
        enter 0, 0
        pusha

        ## Poziva se write sistemski poziv da ispise poruku. Argumenti
        ## poziva su: broj deskriptora fajla u koji se upisuje string
        ## (za standardni izlaz taj broj je 1), adresa prvog karaktera
        ## stringa i duzina stringa (u ovom slucaju 14). Broj write
        ## sistemskog poziva je 4.
        mov     eax, 4
        mov     ebx, 1
        lea     ecx, msg
        mov     edx, 14
        int     0x80

        ## Epilog funkcije.
        popa
        xor     eax, eax
        leave
        ret
```

U pitanju je *Linux* verzija programa; na *FreeBSD*-u bi trebalo zamijeniti kod koji vrši sistemski poziv sa:

```
        push    dword ptr 14
        lea     eax, msg
        push    eax
        push    dword ptr 1
        mov     eax, 4
        push    eax
        int     0x80
        add     esp, 16
```

8.3 *Inline* asemblersko programiranje

Kao i većina drugih *C* prevodilaca, *gcc* prevodilac podržava direktno unošenje asemblerskog koda kroz *C* kod (tzv. *inline* asemblersko programiranje). Tačan metod korišćenja asemblerskog koda na ovaj način uvijek zavisi od prevodioca. U *gcc*-u, takav blok koda se navodi direktivom `__asm__`. Iza ove ključne riječi se u zagradama umeće pod navodnicima blok asemblerskog koda i eventualno specifikacije ulaznih i izlaznih podataka za ovaj blok, kao i registara koji će biti izmijenjeni; svaka od opcionih specifikacija počinje dvotačkom.

Specifikacija ulaznih odn. izlaznih podataka, kao i registara koji će biti izmijenjeni, se sastoji od 0, 1 ili više polja; ako ima dva ili više polja u jednoj specifikaciji, ista su razdvojena zarezom. Redosled specifikacija je fiksan i uvijek prvo dolazi specifikacija izlaznih podataka, zatim specifikacija ulaznih podataka i na kraju specifikacija izmijenjenih registara. Kako je već rečeno, jedna ili više ovih specifikacija sa kraja se mogu izostaviti.

Svako polje u specifikaciji ulaznih odn. izlaznih podataka se sastoji od stringa koga slijedi *C* izraz u zagradama. Kod specifikacije izlaznih podataka string mora početi karakterom `=`. String u ovim specifikacijama određuje gdje će biti smješten odgovarajući operand u asemblerskom kodu. U *gcc* dokumentaciji se može naći spisak mogućih stringova i njihova značenja; treba uočiti da se na ovaj način alokacija registara potpuno prepušta prevodiocu dok programer zapravo piše kod u nekoj vrsti meta-asemblera. Neke od mogućih vrijednosti stringova koji specificiraju lokaciju operanda su recimo `"r"` za registar opšte namjene ili `"g"` za memorijsku lokaciju ili bilo koji registar osim registara opšte namjene. Kada se podaci specificiraju na ovaj način, onda se kroz asemblerski kod (takođe i u bloku za specifikaciju izmijenjenih registara kod onih podataka koji se drže u registrima) mogu referencirati pomoću identifikatora `%0`, `%1` i tako redom do `%9` pri čemu redni brojevi odgovaraju redosledu navođenja podataka u specifikaciji.

U *gcc inline* asembleru može se koristiti samo *AT&T* sintaksa, pa je to dobar povod da se nabroje razlike između ove i *Intel*-ove sintakse. U *AT&T* sintaksi nije uobičajeno da se koristi `noprefix` argument direktive koja određuje sintaksu (zapravo, pošto je *AT&T* sintaksa podrazumijevana za *GNU* asembler, ova direktiva se obično ni ne navodi), tako da se očekuje da imena registara imaju `%` prefiks. Obzirom da podaci koji se razmjenjuju između *C* i *inline* asemblerskog koda već koriste prefiks `%`, imena registara se u *inline* asemblerskom kodu referenciraju sa prefiksom `%%`. Konstante se takođe pišu sa prefiksom, pri čemu se za prefiks koristi `$` karakter. *AT&T* sintaksa koristi suprotan redosled navođenja operanada, pa se tako odredišni operand navodi kao poslednji. Veličina memorijskih operanada se u *AT&T* sintaksi navodi preko odgovarajućeg sufiksa instrukcije. Sufiksi `b`, `w`, `l` i `q` se respektivno koriste za 8-bitni, 16-bitni, 32-bitni i 64-bitni memorijski pristup. Kada se u

AT&T sintaksi navodi memorijska adresa koja se sastoji od više komponenti onda se prvo piše *displacement*, pa onda unutar zagrada zarezima razdvojene *base*, *index* i *scale* komponente, dok se u *Intel*-ovoj sintaksi unutar uglastih zagrada piše puni izraz koji predstavlja memorijsku adresu[3].

Slijedi primjer programa koji koristi *inline* asembler da sabere dva broja:

```
#include <stdio.h>

int
main()
{
        int             x, y;   /* Sabirci. */
        int             res;    /* Suma. */

        /* Ucitavaju se sabirci. */
        scanf("%d%d", &x, &y);

        /* Obavlja se sabiranje. */
        __asm__("
                mov %1, %0
                add %2, %0
                "
                :"=r"(res)
                :"r"(x), "r"(y)
                :"0"
                );

        /* Stampa se suma. */
        printf("%d\n", res);

        return 0;
}
```

Program učitava sabirke sa standardnog ulaza, a zatim obavlja sabiranje pomoću dvije asemblerske instrukcije. Specifikacija iza asemblerskog koda određuje da se sve promjenljive drže u registrima, te da se promjenljive `res`, `x` i `y` mogu kroz asemblerski kod respektivno referencirati sa `%0`, `%1` i `%2`. Pritom, prva od njih je navedena kao izlazna, dok su preostale dvije promjenljive ulazne promjenljive za dati blok asemblerskog koda. U specifikaciji registara koji mijenjaju vrijednost navedeno je da vrijednost mijenja jedino registar koji odgovara `res` promjenljivoj (registri koji su prethodno zauzeti pojedinim promjenljivim mogu se dalje u bloku specifikacija referencirati direktno brojem koji odgovara toj promjenljivoj). Po završenom segmentu asemblerskog koda, suma biva smještena u promjenljivu `res` i program štampa vrijednost te promjenljive.

Više o *inline* asemblerskom programiranju se može naći u dokumentaciji *gcc* prevodioca.

---

[3]tako bi npr. adresi `[ebx+esi*4+matrix]` u *Intel*-ovoj sintaksi odgovaralo `matrix(%ebx,%esi,4)` u *AT&T* sintaksi

8.4 *C* konvencije za smještanje podataka

Na ovom mjestu vrijedi nabrojati i konvencije koje slijede *C* prevodioci u pogledu smještanja podataka. Globalne promjenljive se smještaju u `.data` i `.bss` sekcijama i to inicijalizovane odn. neinicijalizovane promjenljive respektivno. Isto važi i za statičke promjenljive, s tom razlikom što prevodilac obezbjeđuje da pristup ovim promjenljivim ima samo kod iz funkcije u kojoj su deklarisane. Lokalne (tzv. automatske) promjenljive se smještaju na stek. Ukoliko je programer neku promjenljivu označio atributom `register` onda će *C* prevodilac pokušati da je drži u nekom od registara za koje je ranije pomenuto da služe u tu svrhu. Konvencije za smještanje podataka je važno poznavati, između ostalog i jer se recimo na taj način mogu bolje razumjeti neke programerske greške (npr. greške koje se tiču prekoračenja granica niza).

8.5 Format i pokretanje izvršnih fajlova

U prethodnim primjerima je linkovanjem sa standardnom *C* bibliotekom postizano da se u izvršne fajlove uključi odgovarajući kod za pokretanje programa. Asemblerski programi su naime pravljeni tako da obavezno sadrže `main()` funkciju i nakon njihovog prevođenja u objektni fajl dalje su linkovani na isti način kao *C* programi. U ovoj sekciji će zato biti pojašnjeno šta se tačno dešava prilikom pokretanja bilo *C* programa, bilo asemblerskog programa prevedenog na ovaj način. Takođe će biti pokazano i kako se mogu napraviti asemblerski programi koji će biti linkovani direktno.

Kao primjer na kome će biti rađeno tokom izlaganja uzeće se da je zadatak da se minimizuje generisani kod za sledeći jednostavni *C* program koji samo vraća rezultat iz `main()` funkcije i ne radi ništa drugo[4]:

```
int
main()
{
        return 42;
}
```

Da program radi kako treba, može se provjeriti tako što će isti biti pokrenut, a potom odštampan njegov izlazni kod:

```
./answer
echo $?
```

[4]potreban je što je moguće jednostavniji program, a programi koji vraćaju 0 i 1 već postoje - u pitanju su *UNIX* komande `false` i `true`

Dužina izvšnog fajla `answer`, prevedenog bez upisa informacija potrebnih debageru (tj. bez `-g` flega) iznosi 4245 bajtova[5], što nije puno, ali opet jeste previše za program čiji je izvorni kod dugačak svega 27 bajtova. Kroz dalji tekst biće razmotreni načini da se dužina ovog izvršnog fajla smanji.

Format izvršnih fajlova pod većinom *UNIX*-a je tzv. *ELF* (*executable and linker format*) format[6]. Kao i svaki drugi fajl-format, i ovaj format specificira da se fajlovi (izvršni) sastoje od nekoliko cjelina, pri čemu su glavne (ne-opcione) cjeline zaglavlje i sekcije. Sekcije mogu sadržati djelove programa, npr. kod, inicijalizovane i neinicijalizovane podatke ili neke pomoćne informacije, npr. tabelu simbola ili tabelu relokacije[7]. Neinicijalizovani podaci se izdvajaju u posebnu sekciju zato što ta sekcija može biti prazna, tj. jedina informacija koja u vezi sa njom treba da se zapamti jeste njena dužina; prilikom učitavanja programa se ovi podaci postavljaju na 0. Neke od ovih sekcija[8] *gcc* prevodilac dodaje u izvršni fajl mada iste nisu neophodne, pa je prvi korak u smanjenju veličine polaznog fajla oslobađanje od istih. To se za tabelu simbola i relokacione informacije može postići opcijom `-s` prevodioca; kada se polazni fajl prevede uz uključenu ovu opciju, dobija se izvršni fajl čija je dužina samo 2740 bajtova. Ako se sada nad izvršnim fajlom primijeni komanda `nm` koja lista simbole iz datog izvršnog fajla, vidjeće se da je prijavljeno da ovaj fajl nema simbola. Ovo se može provjeriti i u *gdb* debageru - dok je u prvom slučaju moguće postaviti prekidnu tačku na simbol `main`[9], u drugom slučaju ta prekidna tačka se ne može postaviti.

Sledeće što pada na pamet u cilju smanjenja dužine izvršnog fajla jeste uključivanje optimizacije. Međutim, dodavanjem opcije `-Os` za maksimalnu optimizaciju po veličini programa prilikom prevođenja dobija se fajl iste dužine kao i ranije (zaista nema ničeg u datom programu što bi se moglo optimizovati). Zato je naredni korak da se program implementira na asembleru; odgovarajući asemblerski program bi bio oblika:

```
.intel_syntax noprefix

.text
.global main

main:
        ## Prolog funkcije.
        enter 0, 0
        pusha
```

[5] svi podaci o dužini navedeni u ovom poglavlju odnose se na fajlove generisane verzijom 2.95.4 *gcc* prevodioca

[6] ovaj format se inače koristi i na nekim drugim arhitekturama pored *IA-32* arhitekture

[7] *ELF* format se ne odnosi samo na izvršne fajlove, već i na objektne fajlove kao i na dijeljene biblioteke

[8] detaljne informacije o sadržaju nekog *ELF* fajla se mogu dobiti naredbom `readelf`

[9] mada mimo toga nije moguće neko detaljnije debagovanje, pošto pri prevođenju nije bilo uključeno upisivanje u izvršni fajl svih ostalih informacija koje omogućuju povezivanje izvršnog koda sa izvornim kodom

```
        ## Epilog funkcije.
        popa
        mov     eax, 42
        leave
        ret
```

Ako se ovaj program prevede, opet sa uključenim opcijama `-s` i `-Os` prilikom linkovanja, dužina izvršnog fajla je 2668 bajtova. Ovo je nešto bolje nego ranije, što je dobar primjer da uvijek ima dobitaka, makar i minimalnih, ako se programira na asembleru umjesto na *C*-u. Dalje malo poboljšanje se može dobiti ako se uoči da se u `main()` funkciji ne koriste nikakvi argumenti niti lokalne promjenljive, niti se mijenja bilo koji od registara, tako da se prolog i epilog funkcije mogu izbaciti odn. program se može redukovati na:

```
.intel_syntax noprefix

.text
.global main

main:
        ## Vraca se 42 kao rezultat iz funkcije
        mov     eax, 42
        ret
```

Međutim, i nakon ovih izmjena, dobija se izvršni fajl kraći za samo 8 bajtova. Dakle, zaključak sa pisanjem verzije programa na asembleru je da time ne može da se napravi velika optimizacija; uostalom, ako se originalni *C* fajl prevede samo do asembliranja (navođenjem opcije `-S` prilikom prevođenja) može se vidjeti da se dobija asemblerski kod koji je vrlo sličan gornjim fajlovima odn. optimalan.

U pogledu daljeg smanjenja veličine izvršnog fajla ostaje da se optimizacije vrše u onom kodu koji biva umetnut od strane linkera sa zadatkom da obavi inicijalizaciju programa. Ovdje se dolazi do pitanja redosleda dešavanja prilikom pokretanja izvršnog programa. Kada se u *UNIX shell*-u zada ime izvršnog programa, *shell* pravi `fork()` sistemski poziv. *Parent* proces, tj. *shell* nastavlja sa `waitpid()` sistemskim pozivom (osim ako nije na kraju gornje naredbe stajao `&` karakter), dok *child* proces poziva `execve()` sistemski poziv. Ovaj sistemski poziv puni u memorijski prostor *child* procesa sekcije sa kodom i podacima iz odgovarajućeg izvršnog fajla. Takođe, stek pointer *child* procesa se reinicijalizuje i na njega se stavljaju prvo *environment* promjenljive, a zatim argumenti komandne linije (oboje unatraške) koji su argumenti `execve()` poziva, a potom se na stek stavljaju polja pokazivača na ove promjenljive odn. argumente (opet unatraške), pri čemu se svako od tih polja završava nulom. Nakon ovoga `execve()` funkcija se praktično završava i pravi se bezuslovni skok na adresu koja je navedena kao početna

adresa u *ELF* zaglavlju izvršnog programa (ova adresa je u izvršnom fajlu označena `_start` simbolom). Nakon toga kontrolu preuzima kod koji se nalazi u izvršnom fajlu.

Opis dešavanja se odnosi na izvršni fajl koji je kreiran iz izvornog koda u bilo kom programskom jeziku; karakteristično je da se u njemu nigdje ne pominje `main()` funkcija, a to je naravno zato što postoji veliki broj jezika u kojima uopšte ne postoji koncept `main()` funkcije. Kod koji je potreban za pozivanje `main()` funkcije u *C*-u i srodnim jezicima biva kreiran od strane *gcc front-end*-a koji je dosad korišćen za linkovanje i koji je takođe postavljao adresu ovog koda kao početnu adresu u *ELF* zaglavlju . Prije pozivanja `main()` funkcije ovaj kod na stek stavlja njene argumente i to pokazivač na polje stringova sa vrijednostima *environment* promjenljivih, pokazivač na polje stringova sa argumentima komandne linije i na kraju broj argumenata komandne linije (tj. dužinu prethodnog polja). Ovim argumentima se može pristupiti ako se deklarišu kao argumenti `main()` funkcije u *C* programu, a iz asemblera na način kako se uobičajeno pristupa argumentima procedure. Nakon ovoga obavljaju se dodatne inicijalizacije; ove inicijalizacije su minimalne za *C* program[10], međutim one su ipak prisutne.

Imajući sve ovo u vidu, korak ka daljoj optimizaciji bi moglo biti izbacivanje koda za pozivanje `main()` funkcije odn. bilo kakvog koda koji traži linkovanje sa standardnom *C* bibliotekom. To praktično znači da asemblerski program treba početi `_start` labelom i da linkovanje treba izvršiti direktno linkerom. Pitanje koje ostaje otvoreno jeste kako onda završiti asemblerski program obzirom da isti više ne počinje kao poziv procedure već kao bezuslovni skok; odgovor je na sreću veoma prost - koristiće se `exit()` sistemski poziv čiji je jedini argument upravo ono što ovaj asemblerski program treba da vrati. Asemblerski kod bi tako sada bio oblika (na *Linux*-u):

```
.intel_syntax noprefix

.text
.global _start

_start:
        ## Poziva se exit sistemski poziv.
        mov     eax, 1
        mov     ebx, 42
        int     0x80
```

Prevođenje objektnog fajla u izvršni sada se može izvršiti linkerom na sledeći način (pod pretpostavkom da je ime objektnog fajla `answer4.o`):

```
ld -s -Os -o answer4 answer4.o
```

[10] za neke srodne jezike mogu biti i duže - tako se npr. u *C++*-u ovdje odgovarajućim mehanizmom pozivaju konstruktori globalnih objekata

Nakon ovoga je dužina izvršnog fajla svega 420 bajtova. Sitni dobici se sada mogu postići ako se asemblerske instrukcije iskodiraju tako da njihovi mašinski kodovi budu kraći. Tako je npr. instrukcija `MOV EAX, 1` duža od sekvence koju čine `XOR EAX, EAX` i `INC EAX`. Takođe, obzirom da pri pokretanju programa, pored svega što je gore navedeno da se uradi, još i prije skoka na `_start` labelu svi registri opšte namjene budu postavljeni na 0, može se umjesto `MOV EBX, 42` instrukcije staviti `MOV BL, 42`. Na ovaj način se dobija kod koji je kraći još za 4 bajta. Pomoću `readelf` programa može se vidjeti da je dužina kod segmenta u tom slučaju zapravo 8 bajtova, ostatak fajla predstavljaju informacije koje su zahtijevane *ELF* formatom. Dalje smanjenje dužine izvršnog fajla se može postići ručnim izbacivanjem nekih od njih, jer se sve te informacije ne koriste na svakoj platformi. Ove tehnike međutim neće ovdje biti dalje razmatrane.

Poglavlje 9

Instrukcije za rad sa realnim brojevima

9.1 Realni tipovi podataka

Realni brojevi se na računaru predstavljaju u binarnom brojnom sistemu odn. nizom bitova, baš kao i svi ostali tipovi podataka. Konverzija realnog broja iz decimalnog u binarni brojni sistem teče odvojeno za cijeli i razlomljeni dio. Cijeli dio se konvertuje na uobičajeni način, tako što se uzastopno dijeli broj sa 2 dok se ne dobije 0, a onda se ostaci pri dijeljenu pročitaju unatrag (u ovome treba prepoznati zapravo šiftovanje za 1 bit udesno u svakom koraku, pri čemu ostatak pri dijeljenju ustvari predstavlja istisnuti bit najmanje težine). Razlomljeni dio se množi sa 2 i u svakom koraku se pamti cjelobrojni dio (koji može biti 0 ili 1), a razlomljeni dio se ponovo množi sa 2 sve dok se ne dobije 0. Cjelobrojni djelovi proizvoda se ovaj put čitaju redom da bi se dobila reprezentacija razlomljenog dijela datog broja u binarnom brojnom sistemu (opet, u ovome treba prepoznati šiftovanje za 1 bit ulijevo u svakom koraku, pri čemu se sa lijeve strane decimalne tačke izdvaja jedan po jedan bit razlomljenog dijela). Da bi se ovako dobijenim brojevima manipulisalo na računaru, mora se, baš kao i za cijele brojeve, ograničiti broj bitova kojima će isti biti predstavljeni. Kao i u decimalnom brojnom sistemu, i u binarnom brojnom sistemu se ne mogu svi realni brojevi predstaviti konačnim brojem bitova, tako da se onda na računaru može predstaviti samo podskup skupa realnih brojeva. Štaviše, za razliku od skupa cijelih brojeva koji se takođe ne može predstaviti na računaru u cjelini, ali je bar opseg koji se može predstaviti kontinualan, skup realnih brojeva koji se mogu predstaviti na računaru nije kontinualan.

Postoje razni izbori za broj bitova kojima će biti predstavljeni realni brojevi i razne metode kojima se može podijeliti taj skup bitova na cijeli i razlomljeni dio. *IA-32* arhitektura u potpunosti podržava *IEEE-754* standard za rad

sa realnim brojevima u pogledu načina predstavljanja istih. Prema ovom standardu, realni brojevi se predstavljalju nizom od 32 (tzv. *single precision* brojevi) ili 64 (tzv. *double precision* brojevi) bita, pri čemu su bitovi u tom nizu podijeljeni na tri cjeline. Bit najveće težine je bit znaka i ima, kao kod označenih cijelih brojeva, vrijednost 0 za pozitivne odn. 1 za negativne brojeve. Naredna grupa od 8 bitova u 32-bitnom formatu, odn. 11 bitova u 64-bitnom formatu predstavlja eksponent. Eksponent se čita kao neoznačeni broj, a da bi se dobila stvarna vrijednost eksponenta od toga broja treba oduzeti odgovarajuću konstantu koja u 32-bitnom formatu iznosi 127, a u 64-bitnom formatu 1023. Ovakva reprezentacija se označava kao "pomjereni" (*biased*) eksponent. Preostala 23 bita u 32-bitnom odn. 52 bita u 64-bitnom formatu predstavljaju mantisu. Mantisa se obično zadaje u normalizovanom obliku, odn. pretpostavlja se da niz bitova koji je dat mantisom dolazi iza decimalne tačke, ispred koje je implicitna jedinica koja se ne upisuje u binarnu reprezentaciju realnog broja. Na taj način, ako se bit znaka označi sa s, niz bitova koji predstavljaju eksponent sa e i niz bitova koji predstavljaju mantisu sa m, 32-bitni realan broj[1] koji je predstavljen konkatenacijom bitova s, e i m iznosi $(-1)^s \cdot 1.m \cdot 2^{e-127}$.

IEEE standard za rad sa realnim brojevima izdvaja neke kombinacije bitova za predstavljanje specijalnih vrijednosti. Upravo predstavljeni način interpretacije bitskog zapisa realnog broja ne važi naime za vrijednosti eksponenta čiji su svi bitovi 0 odn. 1. Ukoliko su svi bitovi eksponenta i mantise jednaki 0, odgovarajući realan broj je 0 (treba uočiti kako bit znaka može da varira, pa tako postoje $+0$ i -0). Ukoliko su svi bitovi eksponenta jednaki 0, a mantisa različita od 0, radi se o tzv. denormalizovanom broju. Kod ovih brojeva se ne podrazumijeva jedinica koja stoji ispred mantise, tako da se, obzirom da za ovakve brojeve stoji najmanja moguća vrijednost eksponenta, radi o jako malim brojevima koji inače ne bi mogli biti predstavljeni na osnovni način. Ukoliko su svi bitovi eksponenta jednaki 1, a mantisa jednaka 0, radi se o nizu bitova kojim se predstavljaju vrijednosti $+\infty$ odn. $-\infty$. Na kraju, ukoliko su svi bitovi eksponenta jednaki 1, a mantisa različita od nule, radi se o reprezentaciji vrijednosti koja nije broj (*NaN* odn. *not a number*), npr. $\frac{0}{0}$.

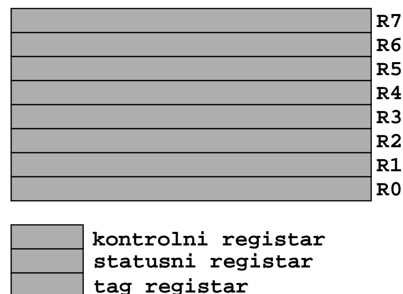
Kao posledica opisanog načina predstavljanja realnih brojeva, izračunavanja koja uključuju iste obično nisu tačna. O ovome treba posebno voditi računa kada se porede rezultati dobijeni različitim računskim operacijama nad realnim brojevima - u opštem slučaju ti rezultati, mada možda matematički identični, ne moraju biti jednaki. Zato je obično pri ispitivanju nekih uslova sa realnim brojevima potrebno dozvoliti i malo odstupanje.

[1] 64-bitni broj se izračunava na isti način, osim što umjesto 127 u izrazu stoji 1023

9.2 Numerički koprocesor

Rad sa realnim brojevima bio je u prvim verzijama procesora *80386*, kao i u nekim verzijama procesora *80486*, izdvojen u poseban procesor, tzv. numerički koprocesor. U svim ostalim procesorima *IA-32* arhitekture ova jedinica je integrisana sa glavnim procesorom, ali se zadržalo korišćenje termina numerički koprocesor za njeno označavanje uporedo sa novijim terminom *FPU* (*floating point unit*).

Skup registara numeričkog koprocesora dat je na slici 9.1.



Slika 9.1: Skup registara numeričkog koprocesora.

Ovaj skup registara se sastoji od 8 registara za podatke dužine 80 bita, te kontrolnog, statusnog i tzv. *tag* registra. Registri za podatke čuvaju realne brojeve u formatu sa tzv. proširenom (*extended*) preciznosti, gdje se 15 bitova odvaja za eksponent i 64 za mantisu. Sve operacije u numeričkom koprocesoru se obično odvijaju u ovom formatu, a kada se u instrukcijama operandi učitavaju iz ili upisuju u memoriju, koprocesor vrši odgovarajuću konverziju. Instrukcije koprocesora inače prihvataju kao operande registre koprocesora ili memorijske promjenljive koje mogu biti realni brojevi u *single*, *double* i *extended* (ovi se rjeđe koriste, jer ih obično viši programski jezici ne podržavaju) preciznosti, te 16-bitni, 32-bitni i 64-bitni cijeli brojevi (takođe i *BCD* brojevi, ali to je manje interesantno). Za razliku od instrukcija opšte namjene, ovdje se kao operandi ne mogu javiti konstante.

Ključna osobina numeričkog koprocesora na *IA-32* arhitekturi je da se registri za podatke koriste kao stek. Naime, u statusnom registru jedno 3-bitno polje određuje koji je registar trenutno vrh steka i većina instrukcija barata implicitno sa tim registrom. U asemblerskom kodu koristi se oznaka `ST(0)` ili prosto `ST` za vrh steka, a onda redom `ST(1)`, `ST(2)` itd. za ostale registre ispod (ustvari iznad, jer stek raste naniže, od registra označenog brojem 7 prema registru označenom brojem 0) vrha steka.

Statusni registar sadrži flegove koji ukazuju da li je došlo do neke greške prilikom izračunavanja, zatim broj registra koji trenutno predstavlja vrh steka (tj. stek-pointer), kao i 4 flega koji se odnose na rezultat prethodne

operacije (ovi flegovi za razliku od flegova iz `EFLAGS` registra imaju različito tumačenje u zavisnosti od operacije). Moguće greške prilikom izračunavanja koje postavljaju flegove statusnog registra su pogrešna operacija, prisustvo denormalizovanog operanda, dijeljenje sa nulom, prekoračenje, potkoračenje i neprecizno izračunavanje. Važna karakteristika odgovarajućih flegova u statusnom registru je da kad jednom budu postavljeni zbog prisustva greške, ostaju u tom stanju dok ih programer eksplicitno ne resetuje.

Kontrolni registar sadrži bitove kojima se dozvoljava odn. zaprečava generisanje odgovarajućeg prekida ako dođe do greške prilikom izračunavanja, te polja kojima se kontroliše preciznost i režim zaokruživanja. Greške kod izračunavanja su iste one greške koje su upravo nabrojane u vezi sa statusnim registrom - programer dakle može da odabere da li će greške biti obrađivane automatski odgovarajućom prekidnom rutinom ili će svaki put ako želi da ustanovi postojanje greške morati da očitava statusni registar. Polje za kontrolu preciznosti omogućava da se specificira da se računske operacije u koprocesoru izvršavaju u *single* odn. *double* umjesto *extended* preciznosti, što ima smisla samo u vezi sa nekim višim programskim jezicima koji ne podržavaju *IEEE* standard. Polje za kontrolu režima zaokruživanja omogućava da se izabere između zaokruživanja prema najbližem broju (što je podrazumijevani režim), zaokruživanja naniže ili naviše i zaokruživanja prema nuli.

Tag registar opisuje sadržaj registara za podatke - za svaki od ovih registara odvojena su 2 bita *tag* registra koji kodiraju da li se u odgovarajućem registru za podatke nalazi validna vrijednost, nula, specijalna vrijednost (denormalizovan broj, beskonačna vrijednost ili *NaN*) ili je registar prazan.

Za pristup sadržaju statusnog i kontrolnog registra postoje i posebne instrukcije, dok se *tag* registru može pristupiti samo pomoću opštih instrukcija za pristup grupama registara unutar koprocesora. Više o svim ovim instrukcijama niže u sekciji o kontrolnim instrukcijama.

9.3 Instrukcije za transfer podataka

Instrukcije koprocesora za transfer podataka su nabrojane u tabeli 9.1. Sve instrukcije koprocesora počinju slovom `F`.

Instrukcija `FLD` učitava realan broj i stavlja ga na vrh steka registara za podatke. Kako je već rečeno operand može biti u *single*, *double* ili *extended* preciznosti i prilikom učitavanja u registar automatski se konvertuje u *extended* preciznost. Za operand se u ovoj i sličnim instrukcijama mora obavezno navesti prefiks koji određuje širinu istog; za operande u *single* odn. *double* preciznosti prefiksi glase `DWORD PTR` odn. `QWORD PTR`, dok *GNU* asembler

Mnemonik	Operacija
`FLD`	učitavanje realnog broja iz memorije
`FILD`	učitavanje cijelog broja iz memorije
`FST`	upis realnog broja u memoriju
`FIST`	upis cijelog broja u memoriju
`FSTP`	upis realnog broja u memoriju i skidanje sa steka
`FISTP`	upis cijelog broja u memoriju i skidanje sa steka
`FXCH`	razmjena sadržaja registara
`FCMOV`	uslovni prenos podataka

Tabela 9.1: Instrukcije numeričkog koprocesora za transfer podataka.

u *Intel*-ovoj sintaksi nema prefiks za operand u *extended* preciznosti[2]. Po izvršenju instrukcije `FLD`, registar koji je bio na vrhu steka odn. koji je bio adresiran sa `ST(0)` se adresira sa `ST(1)`, registar koji je bio adresiran sa `ST(1)` sada je `ST(2)` i tako redom.

Instrukcija `FILD` je slična `FLD` instrukciji, osim što radi sa cjelobrojnim operandom - i ovdje se operand učitava iz memorije, konvertuje u realan broj u `extended` preciznosti i stavlja na vrh registarskog steka. Za specificiranje širine operanda takođe treba obavezno koristiti odgovarajuće prefikse; adresiranje registara se naravno isto mijenja kao kod instrukcije `FLD`.

Instrukcije `FST` i `FIST` konvertuju vrijednost sa vrha registarskog steka u odgovarajući realan odn. cio broj i smještaju taj broj na zadatu memorijsku lokaciju. I kod ovih instrukcija se širina memorijskog operanda mora specificirati prefiksom.

Instrukcije `FSTP` i `FISTP` su potpuno ekvivalentne `FST` i `FIST` instrukcijama, osim što vrijednost sa vrha steka, po smještanju u memoriju, skidaju sa steka. Prema tome, po izvršenim ovim instrukcijama mijenja se adresiranje registara tako da registar koji je bio označavan sa `ST(1)` sada postaje `ST(0)`, registar `ST(2)` postaje `ST(1)` itd. Isto važi za sve instrukcije koprocesora koje skidaju vrijednost sa steka, tako da ovo dalje neće biti ponavljano za svaku od njih ponaosob; ove instrukcije je inače lako prepoznati jer svaka od njih ima sufiks `P`.

Instrukcija `FLD` može imati kao operand i neki registar sa registarskog steka u kom slučaju se sadržaj tog registra kopira na vrh steka. Slično, instrukcije `FST` i `FSTP` mogu takođe imati kao operand neki registar u kom slučaju se sadržaj registra sa vrha steka kopira u taj registar.

Instrukcija `FXCH` razmjenjuje sadržaj registra sa vrha steka i registra koji se specificira kao operand.

[2]u *AT&T* sintaksi se koristi sufiks `T` za instrukcije koje barataju memorijskim operandima u *extended* preciznosti

Instrukcija FCMOV prebacuje vrijednost iz specificiranog registra u registar ST(0) na vrhu steka ukoliko je uslov određen sufiksom instrukcije ispunjen. Sufiksi instrukcije FCMOV su slični kao za CMOV instrukciju; takođe, instrukcija FCMOV se koristi u sličnim situacijama kao CMOV instrukcija.

9.4 Instrukcije za učitavanje konstanti

Instrukcije koprocesora za učitavanje konstanti su nabrojane u tabeli 9.2.

Mnemonik	Operacija
FLDZ	učitavanje broja 0
FLD1	učitavanje broja 1
FLDPI	učitavanje broja π
FLDL2T	učitavanje broja $\log_2 10$
FLDL2E	učitavanje broja $\log_2 e$
FLDLG2	učitavanje broja $\log_{10} 2$
FLDLN2	učitavanje broja $\log_e 2$

Tabela 9.2: Instrukcije numeričkog koprocesora za učitavanje konstanti.

Značenje instrukcija je jasno sa tabele. Sve ove instrukcije stavljaju novu vrijednost na vrh steka, te prema tome mijenjaju adresiranje registara na isti način kao FLD instrukcija.

9.5 Aritmetičke instrukcije

Osnovne aritmetičke instrukcije su nabrojane u tabeli 9.3.

Instrukcije sabiranja, oduzimanja, množenja i dijeljenja uzimaju za operande ili dva registra sa registarskog steka ili jedan registar i realan ili cio broj iz memorije. Instrukcije sa sufiksom P, kako je već ranije rečeno, po izvršenoj operaciji skidaju vrijednost sa vrha steka. Za oduzimanje i dijeljenje, obzirom da su u pitanju nekomutativne operacije, postoje i instrukcije sa sufiksom R kod kojih je redosled operanada obrnut u odnosu na instrukcije bez sufiksa.

Ove instrukcije su veoma važne, tako da će varijante operanada biti detaljnije prikazane na primjeru FADD instrukcije (ostale instrukcije primaju iste operande, samo je aritmetička operacija koja se izvršava druga). Prve dvije varijante se odnose na 32-bitni i 64-bitni memorijski operand u kom slučaju se taj operand sabira sa vrijednošću registra ST(0) i rezultat se smješta opet u registar ST(0). Naredne dvije varijante imaju za operande registar ST(0) i neki drugi registar ST(i), pri čemu je u prvom slučaju prvi odn. odredišni

Mnemonik	Operacija
FADD/FADDP	sabiranje realnih brojeva
FIADD	sabiranje realnog i cijelog broja
FSUB/FSUBP	oduzimanje realnih brojeva
FISUB	oduzimanje realnog i cijelog broja
FSUBR/FSUBRP	oduzimanje realnih brojeva sa obrnutim operandima
FISUBR	oduzimanje realnog i cijelog broja sa obrnutnim operandima
FMUL/FMULP	množenje realnih brojeva
FIMUL	množenje realnog i cijelog broja
FDIV/FDIVP	dijeljenje realnih brojeva
FIDIV	dijeljenje realnog i cijelog broja
FDIVR/FDIVRP	dijeljenje realnih brojeva sa obrnutim operandima
FIDIVR	dijeljenje realnog i cijelog broja sa obrnutnim operandima
FABS	izračunavanje apsolutne vrijednosti
FCHS	promjena znaka operanda
FSQRT	izračunavanje kvadaratnog korijena
FPREM	izračunavanje djelimičnog ostatka
FPREM1	izračunavanje djelimičnog ostatka po *IEEE* standardu
FRNDINT	zaokruživanje na cio broj
FXTRACT	razdvajanje eksponenta i mantise

Tabela 9.3: Aritmetičke instrukcije numeričkog koprocesora.

operand registar `ST(0)`, a u drugom slučaju registar `ST(i)`; vrijednosti registara se sabiraju i rezultat se smješta u odredišni registar. Postoje i dvije varijante sa `P` sufiksom, pri čemu prva od njih ima dva operanda i to neki registar `ST(i)` kao odredišni operand, a registar `ST(0)` kao drugi operand, dok druga varijanta nema operanada. U prvom slučaju se sabiraju vrijednosti registara `ST(i)` i `ST(0)`, rezultat se smješta u registar `ST(i)` i potom se vrijednost iz registra `ST(0)` skida sa steka. U drugom slučaju operandi su implicitni: sadržaj registara `ST(0)` i `ST(1)` se sabira i smješta u registar `ST(1)` i onda se registar `ST(0)` skida sa steka. Instrukcija `FIADD` koja prihvata cjelobrojne operande postoji samo u dvije varijante koje su ekvivalentne prvim dvjema varijantama instrukcije `FADD` - 16-bitni ili 32-bitni cio broj iz memorije se sabira sa sadržajem registra `ST(0)` i rezultat se opet smješta u registar `ST(0)`. Instrukcije sa prefiksom `R` imaju sve nabrojane varijante operanada i odredišni operand im je uvijek isti kao u odgovarajućoj instrukciji bez `R` prefiksa, ali je kod ovih instrukcija redosled operanada u operaciji izmijenjen, što pravi razliku kod nekomutativnih operacija.

Značenje instrukcija `FABS`, `FCHS` i `FSQRT` je jasno sa tabele. Instrukcije `FPREM` i `FPREM1` računaju ostatak pri dijeljenju dva data operanda; instrukcija `FPREM` računa taj ostatak na način kako su to radili *8087* i *80287* matematički

koprocesori, dok instrukcija FPREM1 vrši računanje po *IEEE* standardu[3]. Instrukcija FRNDINT vraća realan broj koji predstavlja cjelobrojnu vrijednost najbližu datom operandu u skladu sa tekućim režimom zaokruživanja. Instrukcija FXTRACT izdvaja eksponent i mantisu vrijednosti sa vrha steka i upisuje ih opet na vrh steka u dva registra kao realne brojeve.

9.6 Instrukcije poređenja

Instrukcije koprocesora za poređenje i klasifikaciju su nabrojane u tabeli 9.4.

Mnemonik	Operacija
FCOM/FCOMP/FCOMPP	poređenje realnih brojeva i postavljanje flegova koprocesora
FUCOM/FUCOMP/FUCOMPP	isto kao prethodno uz “neuređeno” poređenje
FICOM/FICOMP	poređenje realnog i cijelog broja i postavljanje flegova koprocesora
FCOMI/FCOMIP	poređenje realnih brojeva i postavljanje flegova EFLAGS registra
FUCOMI/FUCOMIP	isto kao prethodno uz “neuređeno” poređenje
FTST	poređenje sa nulom
FXAM	klasifikacija broja

Tabela 9.4: Instrukcije numeričkog koprocesora za poređenje i klasifikaciju.

Poređenje realnih brojeva se razlikuje od poređenja cijelih brojeva jer ovi brojevi mogu biti i neuređeni; takav je slučaj kada je jedan od brojeva koji se porede *NaN* - ono što nije broj ne može biti ni u jednoj od tri osnovne relacije (manje, jednako ili veće) sa nekim drugim brojem.

Instrukcije FCOM, FCOMP i FCOMPP porede vrijednost u registru ST(0) sa datim operandom i postavljaju odgovarajuće flegove u statusnom registru koprocesora. Ako su operandi neuređeni, generiše se greška tipa pogrešna operacija. Verzija instrukcije sa sufiksom P skida jedan, a verzija sa sufiksom PP dva registra sa steka po izvršenom poređenju. Ove dvije instrukcije se mogu iskoristiti i za skidanje nepotrebnih vrijednosti sa vrha steka (alternativno se to može uraditi i sekvencom instrukcija FFREE i FINCSTP o kojima će biti više riječi u sekciji o kontrolnim instrukcijama koprocesora).

Instrukcije FUCOM, FUCOMP i FUCOMPP rade isto što i odgovarajuće instrukcije bez slova U, s tom razlikom što se ne generiše greška ako su operandi neuređeni. Slično, instrukcije FICOM i FICOMP rade isto što i instrukcije FCOM odn. FCOMP, osim što je ovdje operand cio broj u memoriji.

[3] ovaj aspekt standarda nije bio preciziran u vrijeme kada su dizajnirani pomenuti koprocesori

Instrukcije `FCOMI` i `FCOMIP` su uvedene tek na *PentiumPro* procesorima. Ove instrukcije obavljaju poređenje na isti način kao `FCOM` odn. `FCOMP` instrukcije, ali ne postavljaju flegove statusnog registra koprocesora već flegove `EFLAGS` registra. Na taj način, instrukcije uslovnog skoka mogu biti izvršene direktno nad rezultatima ovih instrukcija. Treba međutim voditi računa da postavljanje flegova više odgovara poređenju neoznačenih cijelih brojeva (za detalje vidjeti dokumentaciju ovih instrukcija u [9]), tako da onda treba koristiti odgovarajuću grupu instrukcija uslovnog skoka.

Instrukcija `FTST` radi isto što i `FCOM` s tim što je broj sa kojim se vrši poređenje 0.

Instrukcija `FXAM` vrši klasifikaciju vrijednosti iz registra `ST(0)` - u zavisnosti od toga da li je u registru običan realan broj, denormalizovan broj, nula, beskonačna vrijednost, *NaN* ili je registar prazan, postavljaju se flegovi statusnog registra koprocesora (već je pomenuto da vrijednosti ovih flegova imaju različita značenja, u zavisnosti od operacije koja ih mijenja).

Slijedi primjer funkcije koja demonstrira većinu do sada predstavljenih instrukcija koprocesora:

```
.intel_syntax noprefix

.data
four:   .int    4       # Broj 4.
two:    .int    2       # Broj 2.

.text
.global quadric

        ## Funkcija quadric() odredjuje realne korijene kvadratne
        ## jednacine a*x^2+b*x+c=0. Argumenti funkcije su:
        ##   [ebp+8] - koeficijent jednacine uz x^2
        ##   [ebp+16] - koeficijent jednacine uz x^1
        ##   [ebp+24] - koeficijent jednacine uz x^0
        ##   [ebp+32] - adresa lokacije na koju treba smjestiti prvi korijen
        ##   [ebp+36] - adresa lokacije na koju treba smjestiti drugi korijen
        ## Funkcija vraca broj pronadjenih realnih korijena. Ukoliko
        ## jednacina nema realnih korijena, lokacije na koje pokazuju
        ## cetvrti i peti argument ostaju neizmijenjene. Ukoliko
        ## jednacina ima jedan realni korijen, njegova vrijednost biva
        ## upisana u lokaciju na koju pokazuje cetvrti argument.
quadric:
        ## Prolog funkcije.
        enter   0, 0
        push    edi

        ## Izracunava se diskriminanta jednacine b^2-4*a*c
        fld     qword ptr [ebp+16]      # Stek: b
        fld     st                      # Stek: b, b
        fmulp                           # Stek: b^2
        fild    dword ptr four          # Stek: b^2, 4
        fld     qword ptr [ebp+8]       # Stek: b^2, 4, a
        fmulp                           # Stek: b^2, 4*a
        fld     qword ptr [ebp+24]      # Stek: b^2, 4*a, c
        fmulp                           # Stek: b^2, 4*a*c
```

```
        fsubp                           # Stek: b^2-4*a*c

        ## Ispituje se da li je diskriminanta jednaka 0 i ako jeste odmah se
        ## izlazi iz funkcije.
        fldz                            # Stek: b^2-4*a*c, 0
        fcomip  st, st(1)               # Stek: b^2-4*a*c
        je single_root
        jb multiple_roots

no_roots:
        ## Brise se diskriminanta sa steka.
        ffree   st
        fincstp

        ## Postavlja se rezultat na 0 i izlazi se iz funkcije.
        xor     eax, eax
        jmp done

single_root:
        ## Brise se diskriminanta sa steka.
        ffree   st
        fincstp

        ## Izracunava se korijen jednacine -b/(2*a).
        fld     qword ptr [ebp+16]      # Stek: b
        fchs                            # Stek: -b
        fld     qword ptr [ebp+8]       # Stek: -b, a
        fimul   dword ptr two           # Stek: -b, 2*a
        fdivp                           # Stek: -b/(2*a)

        ## Smjesta se korijen na odgovarajucu memorijsku
        ## lokaciju. Registar edi ce drzati adresu te lokacije.
        mov     edi, [ebp+32]
        fstp    qword ptr [edi]

        ## Postavlja se rezultat na 1 i izlazi se iz funkcije.
        mov     eax, 1
        jmp done

multiple_roots:
        ## Izracunavaju se kvadratni korijen diskriminante i vrijednost
        ## imenioca u izrazu za korijene jednacine.
        fsqrt                           # Stek: sqrt(b^2-4*a*c)=S
        fld     qword ptr [ebp+8]       # Stek: S, a
        fimul   dword ptr two           # Stek: S, 2*a
        fxch                            # Stek: 2*a, S

        ## Izracunava se prvi korijen jednacine.
        fld     qword ptr [ebp+16]      # Stek: 2*a, S, b
        fchs                            # Stek: 2*a, S, -b
        fadd    st, st(1)               # Stek: 2*a, S, -b+S
        fdiv    st, st(2)               # Stek: 2*a, S, (-b+S)/(2*a)

        ## Smjesta se prvi korijen na odgovarajucu memorijsku lokaciju.
        ## Registar edi ce drzati adresu te lokacije.
        mov     edi, [ebp+32]
        fstp    qword ptr [edi]         # Stek: 2*a, S

        ## Izracunava se drugi korijen jednacine.
        fld     qword ptr [ebp+16]      # Stek: 2*a, S, b
        faddp                           # Stek: 2*a, b+S
        fchs                            # Stek: 2*a, -b-S
```

```
        fdivrp                          # Stek: (-b-S)/(2*a)

        ## Smjesta se drugi korijen na odgovarajucu memorijsku lokaciju.
        ## Registar edi ce drzati adresu te lokacije.
        mov     edi, [ebp+36]
        fstp    qword ptr [edi]

        ## Postavlja se rezultat na 2 i izlazi se iz funkcije.
        mov     eax, 2

done:
        ## Epilog funkcije.
        pop     edi
        leave
        ret
```

Funkcija prima kao argumente koeficijente kvadratne jednačine kao realne brojeve u *double* preciznosti, kao i dva pokazivača na memorijske lokacije na koje treba eventualno smjestiti realna rješenja te jednačine. Funkcija određuje koliko jednačina ima realnih rješenja i vraća broj rješenja kao rezultat, a takođe ako jednačina ima neka realna rješenja izračunava ta rješenja i smješta ih u memoriju, na lokacije određene pokazivačima koji su preneseni kao argumenti u funkciju.

Na početku funkcije se izračunava diskriminanta jednačine. Treba uočiti kako je kroz kod komentarima prikazan sadržaj steka. Ovakav način komentarisanja, gdje se nakon svake operacije ispisuje sadržaj steka (obično tako da, kao u ovom primjeru, vrh steka bude krajnje desno), je uobičajen za segmente koda koji barataju koprocesorom i znatno olakšava njihovo razumijevanje, te treba nastojati da se isti usvoji i dosljedno primjenjuje.

Pronalaženje optimalne sekvence instrukcija stek-mašine kakav je numerički koprocesor za izračunavanje datog izraza zahtijeva određeno iskustvo. Od pomoći može biti da se izraz napiše u postfiksnoj notaciji, ali takođe treba imati na umu da zbog postojanja varijanti instrukcija sa `R` sufiksom praktično sve operacije mogu zamijeniti mjesta operandima, što u nekim slučajevima može dovesti do pojednostavljivanja postfiksnog izraza. U datom primjeru je izračunavanje diskriminante urađeno direktnim prevođenjem odgovarajućeg postfiksnog izraza u instrukcije stek-mašine, dok su u ostatku koda uvedene određene optimizacije.

Treba uočiti kako je u svim instrukcijama koje koriste operande iz memorije korišćen prefiks koji određuje širinu memorijskog operanda. Upotreba ovog prefiksa je, kako je već naglašeno, obavezna u svim takvim situacijama.

Po izračunavanju diskriminante, poredi se ista sa nulom da bi se odredilo koliko jednačina ima korijena. Za poređenje je korišćena `FCOMI` instrukcija koja postavlja flegove u `EFLAGS` registru. Na taj način, po izvršenom poređenju mogu se koristiti obične instrukcije uslovnog skoka, u ovom slučaju `JE` i `JB` instrukcije.

Ukoliko jednačina nema korijena, funkcija se odmah završava. Treba uočiti kako se pritom čisti registarski stek sekvencom instrukcija `FFREE` i `FINCSTP` o kojima će više riječi biti kasnije - registarski stek treba po završetku korišćenja koprocesora uvijek da se ostavi u istom stanju u kome je i bio prethodno. Ukoliko jednačina ima realnih korijena, onda se ti korijeni novim nizom operacija na steku izračunavaju i smještaju u odgovarajuće memorijske lokacije. Funkcija u svakom slučaju vraća broj korijena na uobičajeni način, preko registra `EAX`.

9.7 Trigonometrijske instrukcije

Instrukcije koprocesora koje implementiraju trigonometrijske funkcije su nabrojane u tabeli 9.5.

Mnemonik	Operacija
`FSIN`	izračunavanje sinusa
`FCOS`	izračunavanje kosinusa
`FSINCOS`	istovremeno izračunavanje sinusa i kosinusa
`FPTAN`	izračunavanje tangensa
`FPATAN`	izračunavanje inverznog tangensa

Tabela 9.5: Instrukcije numeričkog koprocesora za trigonometrijske funkcije.

Sve navedene instrukcije uzimaju operande sa vrha registarskog steka i smještaju rezultate na isti. Operandi za prve četiri instrukcije moraju biti zadati u radijanima i moraju biti u opsegu $[-2^{63}, 2^{63}]$. Podržani skup trigonometrijskih funkcija je minimalan, ostale trigonometrijske funkcije treba izračunavati korišćenjem trigonometrijskih identiteta.

Instrukcije `FSIN` i `FCOS` zamjenjuju sadržaj registra na vrhu steka sa sinusom odn. kosinusom istog. Instrukcija `FSINCOS` zamjenjuje sadržaj registra na vrhu steka njegovim sinusom, a potom stavlja na stek i vrijednost njegovog kosinusa.

Instrukcija `FPTAN` zamjenjuje sadržaj registra na vrhu steka njegovim tangensom i potom stavlja 1 na vrh steka.

Instrukcija `FPATAN` računa inverzni tangens količnika registara `ST(1)` i `ST(0)` u radijanima, smješta ga u registar `ST(1)` i potom skida vrijednost sa vrha steka.

9.8 Logaritamske i eksponencijalne funkcije

Lista instrukcija koprocesora koje implementiraju logaritamske i eksponencijalne funkcije je data tabelom 9.6.

Mnemonik	Operacija
`FYL2X`	izračunavanje vrijednosti $y \cdot \log_2 x$
`FYL2XP1`	izračunavanje vrijednosti $y \cdot \log_2(x+1)$
`F2XM1`	izračunavanje vrijednosti $2^x - 1$
`FSCALE`	množenje stepenom dvojke

Tabela 9.6: Instrukcije numeričkog koprocesora za logaritamske i eksponencijalne funkcije.

Kao i kod trigonometrijskih funkcija, i ovdje je skup instrukcija minimalan, a funkcije koje nisu direktno podržane treba računati koristeći odgovarajuće identitete. Oznake x i y u pojašnjenju značenja instrukcija u tabeli se odnose na vrijednosti iz registara `ST(0)` odn. `ST(1)`.

Instrukcije `FYL2X` i `FYL2XP1` smještaju rezultat u registar `ST(1)` i skidaju jednu vrijednost sa steka. Instrukcija `FYL2XP1` obezbjeđuje bolju preciznost kada je vrijednost x bliska 0. Operand u registru `ST(0)` kod instrukcije `FYL2X` mora biti veći od 0, a kod instrukcije `FYL2XP1` u opsegu $[-(1-\frac{\sqrt{2}}{2}), 1-\frac{\sqrt{2}}{2}]$.

Instrukcija `F2XM1` zamjenjuje vrijednost iz registra `ST(0)` izračunatom vrijednošću funkcije. Operand ove instrukcije mora biti u opsegu $[-1, 1]$.

Instrukcija `FSCALE` dodaje na eksponent vrijednosti u registru `ST(0)` cjelobrojni dio vrijednosti iz registra `ST(1)`. Ova instrukcija se može koristiti za brzo množenje datog broja stepenom dvojke.

Slijedi primjer funkcije koja demonstrira korišćenje trigonometrijskih, logaritamskih i eksponencijalnih instrukcija:

```
.intel_syntax noprefix

.text
.global root

        ## Funkcija root() racuna n-ti korijen datog kompleksnog broja.
        ## Argumenti funkcije su:
        ##   [ebp+8], [ebp+16] - realni i imaginarni dio datog broja
        ##   [ebp+24] - broj korijena koga treba izracunati
        ##   [ebp+28], [ebp+32] - adrese na koje treba smjestiti realni
        ##                        i imaginarni dio n-tog korijena
        ## Pretpostavlja se da je broj korijena pozitivan. Funkcija se u
        ## potpunosti podrvrgava C konvencijama pozivanja.
root:
        ## Prolog funkcije.
        enter   0, 0
        push    ebx

        ## Kompleksni broj re+im*i ce biti prvo preveden u polarne
        ## koordinate r*e^(i*fi), a potom ce njegov n-ti korijen biti
        ## izracunat kao r^(1/n)*(cos(fi/n)+i*sin(fi/n)).

        ## Racuna se reciprocna vrijednost broja korijena.
        fld1                            # Stek: 1
        fidiv   dword ptr [ebp+24]      # Stek: 1/n
```

```
## Racuna se polarna koordinata r=sqrt(re*re+im*im).
fld     qword ptr [ebp+16]      # Stek: 1/n, im
fmul    st                      # Stek: 1/n, im*im
fld     qword ptr [ebp+8]       # Stek: 1/n, im*im, re
fmul    st                      # Stek: 1/n, im*im, re*re
faddp                           # Stek: 1/n, re*re+im*im
fsqrt                           # Stek: 1/n, sqrt(re*re+im*im)=r

## Racuna se n-ti korijen koordinate r. U komentarima je sa
## int() oznacen cjelobrojni, a sa frac() razlomljeni dio
## realnog broja.
fyl2x                   # Stek: 1/n*log2(r)=log2(r^(1/n))=L
fld     st              # Stek: L, L
frndint                 # Stek: L, int(L)
fsub    st(1), st       # Stek: L-int(L)=frac(L), int(L)
fxch                    # Stek: int(L), frac(L)
f2xm1                   # Stek: int(L), 2^frac(L)-1
fld1                    # Stek: int(L), 2^frac(L)-1, 1
faddp                   # Stek: int(L), 2^frac(L)
fscale                  # Stek: int(L), 2^int(L)*2^frac(L)=2^(int(L)+frac(L))=2^L
fxch                    # Stek: 2^L, int(L)
fcomp                   # Stek: 2^L=2^log2(r^(1/n))=r^(1/n)

## Racuna se polarna koordinata fi.
fld     qword ptr [ebp+16]      # Stek: r^(1/n), im
fld     qword ptr [ebp+8]       # Stek: r^(1/n), im, re
fpatan                          # Stek: r^(1/n), atan(im/re)=fi

## Racuna se vrijednost fi/n, kao i sinus i kosinus toga ugla.
fidiv   dword ptr [ebp+24]      # Stek: r^(1/n), fi/n
fsincos                         # Stek: r^(1/n), sin(fi/n), cos(fi/n)

## Racuna se realna koordinata korijena datog kompleksnog broja
## r^(1/n)*cos(fi/n) i smjesta na odgovarajucu lokaciju u memoriji.
fmul    st, st(2)               # Stek: r^(1/n), sin(fi/n), r^(1/n)*cos(fi/n)
mov     ebx, [ebp+28]
fstp    qword ptr [ebx]         # Stek: r^(1/n), sin(fi/n)

## Racuna se imaginarna koordinata korijena datog kompleksnog broja
## r^(1/n)*sin(fi/n) i smjesta na odgovarajucu lokaciju u memoriji.
fmulp                           # Stek: r^(1/n)*sin(fi/n)
mov     ebx, [ebp+32]
fstp    qword ptr [ebx]

## Epilog funkcije.
pop     ebx
leave
ret
```

Funkcija prima kao argumente dva realna broja u *double* preciznosti koji predstavljaju realni i imaginarni dio datog kompleksnog broja, zatim cio broj koji predstavlja korijen ovog kompleksnog broja koga treba izračunati i na kraju dva pokazivača na memorijske lokacije u koje treba smjestiti realni i imaginarni dio traženog korijena.

Na početku funkcije se računa modul r kompleksnog broja, a zatim se nalazi odgovarajući korijen ovog modula. Na ovaj dio koda treba obratiti posebnu pažnju jer demonstrira kako se korišćenjem postojećih logaritamskih i ek-

sponencijalnih instrukcija računaju funkcije koje nisu direktno podržane. Naime, da bi se izračunao $\sqrt[n]{r}$ treba uočiti identitet $\sqrt[n]{r} = 2^{log_2(\sqrt[n]{r})} = 2^{\frac{1}{n}log_2 r}$. Za izračunavanje veličine $L = \frac{1}{n}log_2 r$ potom se koristi instrukcija `FYL2X`, a onda se za izračunavanje 2^L koristi instrukcija `F2XM1`. Instrukcija `F2XM1` međutim, kako je već rečeno, prihvata operande samo u opsegu $[-1, 1]$, tako da se broj koji stoji u eksponentu razbija na cjelobrojni i razlomljeni dio, a onda se instrukcija `F2XM1` koristi samo nad razlomljenim dijelom, dok se uračunavanje cjelobrojnog dijela u eksponentu vrši instrukcijom `FSCALE`. Po izračunavanju korijena modula datog kompleksnog broja na steku ostaje cjelobrojni dio veličine L; treba uočiti kako je za skidanje sa steka ove za dalje izračunavanje nepotrebne vrijednosti ovdje iskorišćena instrukcija `FCOMP`.

Nakon što je izračunat odgovarajući korijen modula kompleksnog broja izračunava se faza korijena, a zatim i njen kosinus i sinus koji se množe sa korijenom modula čime se dobijaju realni i imaginarni dio korijena datog broja. Ovaj dio koda dobro demonstrira upotrebu trigonometrijskih funkcija. Rezultat se po završenom izračunavanju upisuje u memoriju, na lokacije određene odgovarajućim argumentima funkcije.

9.9 Kontrolne instrukcije

Kontrolne instrukcije koprocesora su izlistane u tabeli 9.7.

Mnemonik	Operacija
`FINIT/FNINIT`	inicijalizacija koprocesora
`FLDCW`	učitavanje kontrolnog registra koprocesora
`FSTCW/FNSTCW`	kopiranje sadržaja kontrolnog registra koprocesora
`FSTSW/FNSTSW`	kopiranje sadržaja statusnog registra koprocesora
`FCLEX/FNCLEX`	resetovanje flegova u statusnom registru
`FLDENV`	učitavanje sadržaja svih pomoćnih registara koprocesora
`FSTENV/FSTENV`	kopiranje sadržaja svih pomoćnih registara koprocesora
`FRSTOR`	učitavanje sadržaja svih registara koprocesora
`FSAVE/FNSAVE`	kopiranje sadržaja svih registara koprocesora
`FINCSTP`	inkrementiranje stek-pointera za registarski stek
`FDECSTP`	dekrementiranje stek-pointera za registarski stek
`FFREE`	oslobađanje registra
`FNOP`	prazna instrukcija
`WAIT/FWAIT`	instrukcija za sinhronizaciju

Tabela 9.7: Kontrolne instrukcije numeričkog koprocesora.

Instrukcije `FINIT` i `FNINIT` inicijalizuju koprocesor i postavljaju sve njegove registre na podrazumijevane vrijednosti. Razlika je što instrukcija `FNINIT`, kao i sve ostale instrukcije sa prefiksom `FN`, prije nego što izvrši datu operaciju

ne provjerava da li postoji neka greška od ranije; instrukcije bez ovog prefiksa vrše provjeru i eventualno obrađuju greške.

Instrukcija `FLDCW` učitava sadržaj kontrolnog registra koprocesora iz memorije. Instrukcije `FSTCW` i `FNSTCW` upisuju sadržaj ovog registra u memoriju dok instrukcije `FSTSW` i `FNSTSW` upisuju sadržaj statusnog registra u memoriju ili u `AX` registar.

Instrukcije `FCLEX` i `FNCLEX` resetuju sadržaj svih flegova u statusnom registru.

Instrukcije `FSTENV` i `FNSTENV` kopiraju sadržaj svih registara koprocesora osim registarskog steka u memoriju. Za kopiranje sadržaja svih registara u memoriju koriste se instrukcije `FSAVE` i `FNSAVE`; ove instrukcije takođe inicijalizuju sadržaj svih registara na podrazumijevane vrijednosti baš kao i `FINIT` i `FNINIT` instrukcije. Sa druge strane, instrukcije `FLDENV` i `FRSTOR` učitavaju u registre koprocesora vrijednosti iz memorije; prva instrukcija učitava iz memorije sadržaj svih registara osim registarskog steka, dok druga instrukcija učitava sadržaj svih registara.

Instrukcije `FINCSTP` odn. `FDECSTP` inkrementiraju odn. dekrementiraju broj registra koji određuje vrh steka u statusnom registru. Instrukcija `FFREE` markira dati registar kao prazan u *tag* registru. Za oslobađanje registra sa vrha steka, potrebno je primijeniti instrukcije `FFREE` i `FINCSTP`; ranije je pokazano da se alternativno u istu svrhu može iskoristiti instrukcija `FCOMP`.

Instrukcije `WAIT` i `FWAIT`[4] forsiraju obradu eventualnih postojećih grešaka na koprocesoru prije nego što se izvrši naredna, bilo koja, instrukcija procesora. Instrukcije koprocesora koje nemaju prefiks `FN` podrazumijevaju naime obradu grešaka, ali tek kada u nizu instrukcija dođe red na njih. Međutim, može da se desi da između instrukcije koja prouzrokuje grešku i naredne instrukcije koprocesora u programu ima drugih instrukcija, i to takvih da koriste ili mijenjaju vrijednost pri čijem je izračunavanju prijavljena greška. Da bi se izbjegli takvi problemi, dvije navedene instrukcije se mogu umetnuti odmah ispred tog niza drugih instrukcija da bi eventualna greška bila obrađena prije nego što se krene u izvršavanje istog.

9.10 Mehanizam obrade grešaka koprocesora

Ranije je već pomenuto da prilikom izvršavanja većine instrukcija koprocesora može biti generisana greška iz jedne od sledećih 6 grupa grešaka: pogrešna operacija, prisustvo denormalizovanog operanda, dijeljenje sa nulom, prekoračenje, potkoračenje i neprecizno izračunavanje. Koprocesor prijavljuje grešku tako što postavlja jedan od 6 flegova u statusnom registru. Takođe, ukoliko nije zapriječen setovanjem odgovarajućeg flega u kontrolnom

[4] u pitanju su ustvari dva različita mnemonika za istu mašinsku instrukciju

registru, biva generisan i prekid; šta će tačno uraditi prekidna rutina u tom slučaju je zavisno od operativnog sistema.

Jasno je da su greške iz navedenih grupa različite ozbiljnosti. Tako npr. neprecizno izračunavanje nije kritičan problem; npr. sve transcendentalne instrukcije generišu ovu grešku. Sa druge strane, pogrešna operacija je svakako greška koja zahtijeva neku reakciju. Koliko pažnje će biti posvećeno obradi grešaka svakako zavisi od tipa programa koji se piše. Ukoliko je u pitanju recimo neka numerička biblioteka, onda itekako ima smisla pažljivo proučiti ponašanje koprocesora u ovakvim situacijama i implementirati detaljne provjere pojavljivanja grešaka i dosljednu obradu istih. U većini drugih primjena, dovoljno je osloniti se na podrazumijevano ponašanje koprocesora u pogledu obrade ovakvih situacija; takav pristup je primijenjen i kroz kod koji je prezentiran u ovom tekstu.

Poglavlje 10

Privilegovane instrukcije

Kako je već pomenuto, *IA-32* arhitektura obuhvata određen broj sistemskih instrukcija, odn. instrukcija koje se mogu izvršavati samo u privilegovanom režimu. *IA-32* arhitektura predviđa naime direktnu podršku u procesoru za određen skup funkcionalnosti operativnog sistema; tako npr. kako je već pominjano postoji podrška za segmentnu i straničnu organizaciju memorije, zatim za različite nivoe privilegija pri izvršavanju procesa, takođe i za *scheduling* procesa; pod podrškom se ovdje podrazumijeva skup namjenskih registara i instrukcija. Obzirom da su skup privlegovanih registara odn. instrukcija uglavnom od značaja samo za implementaciju operativnih sistema, isti ovdje neće biti detaljnije razmatrani. Kao primjer korišćenja ovakvih registara odn. instrukcija biće razmotren mehanizam prekida, a onda će biti nešto rečeno i o ulazno-izlaznim instrukcijama koje takođe spadaju u grupu privilegovanih instrukcija ali se ponekad javlja potreba da se koriste i iz aplikativnih programa.

10.1 Mehanizam prekida

O prekidima je već bilo riječi na više mjesta u ovom tekstu. *IA-32* arhitektura podržava 256 prekida, pri čemu su prva 32 prekida rezervisani za interne prekide i tzv. *non-maskable* (preko `NMI` pina procesora) prekid. Procesor u registru `IDTR` drži adresu jedne tabele, tzv. *IDT* (*interrupt descriptor table*) u koju operativni sistem prilikom inicijalizacije za svaki prekid upisuje adresu potprograma koga treba izvršiti nakon što dođe do prekida. Poziv rutine za obradu prekida je sličan `CALL` instrukciji, osim što se na stek pored povratne adrese stavlja i sadržaj `EFLAGS` registra. Odgovarajuća instrukcija za završetak prekidne rutine je `IRET` koja je ekvivalentna `RET` instrukciji osim što vraća sa steka i prethodni sadržaj `EFLAGS` registra. Obzirom da se interni i eksterni prekidi dešavaju potpuno asinhrono (tj. do njih može doći u bilo

kom trenutku), od vitalne je važnosti da rutine za obradu prekida očuvaju sadržaj svih ostalih registara.

Pomenuti `IDTR` registar procesora je, baš kao i ostali sistemski registri, dostupan preko odgovarajućih sistemskih instrukcija samo kodu koji se izvršava u privilegovanom režimu. Instrukcije kojima se pristupa ovom registru su `LIDT` za učitavanje vrijednosti u registar iz memorije odn. `SIDT` za kopiranje vrijednosti iz registra u memoriju.

10.2 Ulazno-izlazne instrukcije

Na *IA-32* arhitekturi ulazno-izlazni i memorijski adresni prostori su razdvojeni; za portove ulazno-izlaznih uređaja je odvojen 16-bitni adresni prostor kome se pristupa posebnim instrukcijama (treba međutim napomenuti da se sistem može dizajnirati i tako da neki od I/O uređaja imaju memorijski mapirane portove). Instrukcije za pristup portovima su nabrojane u tabeli 10.1.

Mnemonik	Operacija
`IN`	učitavanje vrijednosti sa porta
`OUT`	upis vrijednosti u port
`INS`	učitavanje niza vrijednosti sa portova
`OUTS`	upis niza vrijednosti u portove

Tabela 10.1: Ulazno-izlazne instrukcije.

Kod instrukcija `IN` odn. `OUT`, učitavanje odn. upis uvijek idu u odn. iz `EAX` registra (ili njegovog dijela). Bez obzira na to, ovaj registar (odn. njegov dio) se uvijek navodi kao prvi argument instrukcije `IN` odn. drugi argument instrukcije `OUT`, čime je određeno da li je u pitanju 8-bitni, 16-bitni ili 32-bitni transfer. Adresa porta se može navesti kao direktni operand ukoliko je u pitanju neki od prvih 256 portova ili se može staviti u registar `DX` (koji se onda takođe obavezno navodi kao operand) za svaki adresabilni port. Kada je u pitanju 16-bitni transfer adresa porta mora biti djeljiva sa 2, a kada je u pitanju 32-bitni transfer adresa porta mora biti djeljiva sa 4.

Instrukcije `INS` odn. `OUTS` su ekvivalentne instrukcijama `LODS` odn. `STOS` za rad sa stringovima. Instrukcije `INS` i `OUTS` postoje u dvije varijacije, pri čemu jedna od varijacija ima dva operanda, a druga nema nijedan. U prvom slučaju jedan operand je `DX` registar u kome se nalazi adresa porta, dok je drugi operand registar ili memorijska lokacija i njegova veličina određuje da li će u pitanju biti 8-bitni, 16-bitni ili 32-bitni transfer. Odredišna (u slučaju `INS` instrukcije) odn. izvorišna (u slučaju `OUTS` instrukcije) adresa je određena sadržajem registara `EDI` odn. `ESI`. Druga varijacija ovih instrukcija podrazumijeva da je adresa porta data u registru `DX` dok sufiks instrukcije

određuje širinu transfera: `B` stoji za 8-bitni transfer, `W` za 16-bitni i `D` za 32-bitni. Uz `INS` i `OUTS` instrukcije može se koristiti `REP` prefiks koji ima isto značenje kao kod instrukcija za rad sa stringovima.

Na *UNIX*-u, korisnički procesi inicijalno nemaju pravo pristupa portovima. Dio podataka koji se čuvaju o procesu jeste i tzv. *I/O permission bitmap* u kojoj svakom portu odgovara po jedan bit koji određuje da li neki proces ima pravo pristupa određenom portu ili ne; ova bitmapa se pri kreiranju procesa inicijalizuje tako da proces nema pravo pristupa nijednom portu. Prava pristupa se mogu promijeniti odgovarajućim sistemskim pozivom koji je specifičan od varijacije *UNIX*-a (`ioperm()` za *Linux*, `i386_set_ioperm()` za *FreeBSD*). Ovaj sistemski poziv se međutim može uspješno izvršiti samo ako proces koji ga poziva ima *root* privilegije.

Slijedi primjer funkcije koja koristi ulazno-izlazne instrukcije za pristup portovima da bi pustila ton određene učestanosti na internom zvučniku računara:

```
.intel_syntax noprefix

.text
.global play

        ## Funkcija play() pusta ton odredjene ucestanosti i trajanja na
        ## zvucniku. Argumenti funkcije su:
        ##   [ebp+8] - ucestanost tona (u 0.01Hz)
        ##   [ebp+12] - duzina trajanja tona  (u ms)
        ## Dozvoljeni opseg za prvi argument je [20Hz,20kHz]. Funkcija
        ## se u potpunosti podvrgava C konvencijama
        ## pozivanja. Pretpostavka je da proces iz koga se poziva
        ## funkcija ima pravo pristupa portovima 0x42, 0x43 i 0x61.
play:
        ## Prolog funkcije.
        enter   0, 0

        ## U kontrolni registar 8254 kola cija je adresa 0x43 salje se
        ## komanda za upis odgovarajuceg dijeljenika ucestanosti u
        ## brojac 2. Dva bita najvece tezine u ovoj komandi odredjuju da
        ## se komanda odnosi na brojac 2, dva naredna bita da je u
        ## pitanju operacija upisa dva bajta u ovaj registar, pri cemu
        ## ce prvo biti upisan nizi, a potom visi bajt. Tri naredna bita
        ## komande odredjuju da ovaj brojac treba da radi u tzv. Mode3
        ## rezimu odn. da generise kontinualan signal date
        ## ucestanosti. Poslednji bit komande odredjuje da vrijednost
        ## koja je upisana u brojac treba da bude tretirana kao
        ## 16-tobitni broj.
        mov     al, 0xb6
        out     0x43, al

        ## Sada treba u odgovarajuci port za podatke (0x42) upisati broj
        ## kojim se dijeli ta ucestanost da bi se dobila zeljena
        ## ucestanost. Obzirom da je zeljena ucestanost data u stotim
        ## djelovima Hz, umjesto ulazne ucestanosti se pri odredjivanju
        ## djelioca koristi 100 puta veci broj. Treba obratiti paznju
        ## kako se 16-tobitni djelilac upisuje tako sto se prvo na ovaj
        ## port posalje njegov nizi, a potom visi bajt (opseg
        ## dozvoljenih ucestanosti obezbjedjuje da djelilac uvijek bude
```

```
        ## 16-tobitni broj).
        mov     eax, 119318000
        xor     edx, edx
        mov     ecx, [ebp+8]
        div     ecx
        out     0x42, al
        mov     al, ah
        out     0x42, al

        ## Zvucnik se ukljucuje tako sto se u zadnja dva bita porta 0x61
        ## upisu jedinice.
        in      al, 0x61
        or      al, 0x03
        out     0x61, al

        ## Nakon aktiviranja zvucnika proteklo vrijeme se mjeri tako sto
        ## se prati bit 4 porta 0x61 - ovaj bit svakih 15.085us mijenja
        ## vrijednost. Broj promjena ovog bita tokom datog intervala se
        ## odredjuje tako sto se duzina intervala izrazi u ns (sto se
        ## postize mnozenjem sa 10^6, obzirom da je duzina intervala
        ## data u ms), a potom podijeli sa periodom promjene ovog bita
        ## izrazenim takodje u ns.
        push    eax
        mov     eax, [ebp+12]
        mov     ecx, 1000000
        mul     ecx
        mov     ecx, 15085
        div     ecx

        ## U petlji se stalno provjerava da li se bit 4 porta 0x61
        ## promijenio tako sto se ocita port, maskiraju ostali bitovi i
        ## uporedi ta vrijednost sa sacuvanom isto tako dobijenom
        ## vrijednoscu nakon prethodne promjene.
        mov     ecx, eax
        mov     ah, 0
again:
        in      al, 0x61
        and     al, 0x10
        cmp     al, ah
        je again
        mov     ah, al
        loop again

        ## Zvucnik se iskljucuje tako sto se u zadnja dva bita porta
        ## 0x61 upisu nule.
        pop     eax
        and     al, 0xfc
        out     0x61, al

        ## Epilog funkcije.
        leave
        ret
```

Učestanost tona koji se čuje sa ugrađenog zvučnika određena je preko kola koje se označava imenom programabilni tajmer ili kolo *8254*. Ovo kolo zapravo sadrži tri brojača odn. dijeljenika učestanosti, a prima ulazni takt na učestanosti 1193180Hz. Prvi od tri brojača služi za generisanje signala prekida procesoru u pravilnim razmacima, pri čemu se obično prekidna rutina koristi za *scheduling* i slične periodične aktivnosti. Treći brojač je povezan

sa ugrađenim zvučnikom i učestanost na izlazu ovog brojača određuje učestanost tona koji će se čuti sa zvučnika. Potrebno je dakle u ovaj brojač upisati takvu vrijednost dijeljenika koja će na izlazu dati traženu učestanost za dati ton. U tom cilju se u kontrolni registar (port 0x43) kola *8254* `OUT` instrukcijom šalje odgovarajuća komanda za upis dijeljenika učestanosti za treći brojač, a potom se u registar sa podacima (port 0x42) sa dvije `OUT` instrukcije upisuju prvo niži pa viši bajt djelioca.

Zatim se prelazi na uključivanje zvučnika, koji se kontroliše preko dva bita najmanje težine porta 0x61. Vrijednost ovog porta se prvo očitava `IN` instrukcijom, modifikuju se zadnja 2 bita da bi se naznačilo da treba uključiti zvučnik, a potom se `OUT` instrukcijom vrati modifikovana vrijednost u port 0x61, čime zvučnik biva uključen. Sada preostaje samo da se isključi zvučnik nakon određenog vremena. Mjerenje proteklog vremena se može izvršiti na razne načine, a ovdje je korišćena dosta primitivna metoda da se kontinualno prati bit sa indeksom 4 porta 0x61 koji periodično mijenja vrijednost. Mjerenje nije precizno, jer ništa ne garantuje da neki drugi proces neće zauzeti procesor na duže vrijeme nego što traje ciklus promjene vrijednosti ovog bita, ali može da posluži za demonstraciju. Nakon što dati vremenski interval istekne (po ovakvom načinu mjerenja), u zadnja 2 bita porta 0x61 se upisuju nule, čime se zvučnik isključuje.

Na odgovarajućoj *Web* strani ([1]) može se naći i *C* program koji učitava sa standardnog ulaza željenu učestanost tona i dužinu intervala, potom poziva odgovarajući sistemski poziv kojim se dozvoljava pristup portovima 0x42, 0x43 i 0x61 koje koristi gornja funkcija i na kraju poziva samu funkciju da pusti ton na zvučniku. Treba uočiti da ovaj program može da se pokrene samo sa *root* privilegijama, u suprotnom promjena prava pristupa pomenutim portovima neće biti uspješno obavljena, tako da gornja funkcija neće moći da obavi svoj zadatak.

Dati primjer demonstrira kako korišćenje ulazno-izlaznih instrukcija nije komplikovano. Međutim, za ulazno-izlazno programiranje potrebno je izuzetno poznavanje periferijskih uređaja koji se programiraju. Takođe, pošto se ovakav kod obično javlja u kontekstu pisanja odgovarajućih drajvera, potrebno je dobro poznavanje mehanizma kojim drajver nameće odgovarajući interfejs periferijskom uređaju na datom operativnom sistemu (npr. na *UNIX*-u se od drajvera očekuje da učini da uređaj što je moguće više liči na običan fajl).

Poglavlje 11

MMX instrukcije

Mada je svaka nova generacija procesora *IA-32* arhitekture donosila neka proširenja u pogledu skupa instrukcija, prva zaista krupna ekstenzija na *IA-32* arhitekturu je bilo dodavanje *MMX* (*multimedia extensions*) instrukcija na kasnijim verzijama *Pentium* procesora[1]. *MMX* ekstenzije uvode uslovno rečeno 8 novih 64-bitnih cjelobrojnih registara i određen broj novih instrukcija primarno namijenjenih za paralelnu obradu nekoliko podataka malih cjelobrojnih tipova. Prilikom dizajniranja *MMX* ekstenzija pošlo se od malog broja aplikacija (multimedija, komunikacije i sl.) i analizirane su unutrašnje petlje ključnih algoritama u takvim aplikacijama. Pokazalo se da su osnovne karakteristike analiziranih algoritama mali cjelobrojni tipovi podataka (8-bitni ili 16-bitni), visok nivo paralelizma u operacijama nad takvim podacima, te dominantna upotreba sabiranja i množenja od aritmetičkih operacija. Onda su *MMX* instrukcije osmišljene da mogu da se iskoriste da znatno ubrzaju takve algoritme.

11.1 Skup registara za *MMX* instrukcije

MMX ekstenzije su podrazumijevale i uvođenje 8 novih 64-bitnih registara označenih imenima `MM0`, `MM1` i tako redom do `MM7` nad kojima barataju *MMX* instrukcije. Osnovna karakteristika ovog skupa registara jeste da zapravo nisu u pitanju novi registri, već su za iste iskorišćena polja za mantisu registara matematičkog koprocesora. Zbog ovakve organizacije, nije moguće istovremeno koristiti *MMX* instrukcije i raditi u pokretnom zarezu - razlog za ovakvu odluku u dizajnu je bilo to što je uočeno da ciljna grupa aplikacija nema izračunavanja u pokretnom zarezu. Ono što je važno jeste da se uvijek po završetku korišćenja ovih registara kao *MMX* registara, a prije njihovog

[1] *PentiumPro* generacija nije međutim imala ove ekstenzije, dok su počev od *Pentium II* generacije *MMX* ekstenzije standardni dio arhitekture

ponovnog korišćenja kao registara matematičkog koprocesora mora izvršiti *MMX* instrukcija `EMMS`, koja reinicijalizuje stanje matematičkog koprocesora.

U *MMX* instrukcijama se pomenutim registrima pristupa direktno tj. na isti način kao registrima opšte namjene odn. ovi registri se ne smatraju organizovanim u stek kao što je to slučaj kada se radi sa registrima koprocesora. *MMX* registri se mogu koristiti na više načina: u njima se može držati osam 8-bitnih, četiri 16-bitna ili dva 32-bitna broja nad kojima se mogu paralelno izvršavati određene operacije (u kojim slučajevima se ti brojevi označavaju kao tzv. pakovani operandi) ili se registri mogu tretirati kao jedan 64-bitni podatak.

Inače, prefiks većine *MMX* instrukcija je `P`. Prije korišćenja *MMX* instrukcija treba obavezno provjeriti da li ih procesor podržava, što se može uraditi tako što se izvrši `CPUID` instrukcija sa sadržajem `EAX` registra jednakim 1, a potom provjeri bit 23 registra `EDX` - ukoliko je ovaj bit jednak 1, procesor podržava *MMX* instrukcije. Pored ovoga bi trebalo provjeriti i da li je isključena emulacija numeričkog koprocesora, ali obzirom da na poslednjim generacijama procesora *IA-32* arhitekture nema razloga da tako ne bude, ova se provjera ponekad preskače.

11.2 Tretman prekoračenja kod *MMX* instrukcija

Važna karakteristika *MMX* instrukcija je da ne generišu prekoračenja. Ove instrukcije naime rade ili tako da ignorišu prekoračenja (tzv. *wraparound*) ili rade u tzv. režimu saturacije, kada se u slučaju prekoračenja za rezultat uzima najbliža vrijednost koja se može predstaviti datim tipom podataka. Neka se npr. vrši sabiranje dva označena 8-bitna podatka čije su vrijednosti 100 i 50. U uobičajenom režimu rada bi bilo prijavljeno prekoračenje i rezultat bi bio negativan. U *wraparound* režimu međutim prekoračenje ne bi bilo prijavljeno, a rezultat bi bio ostavljen kao negativan. U režimu saturacije bi rezultat bio postavljen na 127, kao vrijednost koja je najbliža stvarnom rezultatu od svih vrijednosti koje se datim tipom mogu predstaviti.

U režimu saturacije razlikuju se označena i neoznačena saturacija, u zavisnosti od toga sa kojim se tipom podataka radi. Razlika se naznačava sufiksom instrukcija, pri čemu se koriste sufiksi `S` za označenu, odn. `US` za neoznačenu saturaciju.

Režim saturacije je inače, baš kao i *MMX* ekstenzije generalno, uveden zato što je često potreban u domenu aplikacija kojima su ove ekstenzije primarno namijenjene.

11.3 Instrukcije za transfer podataka

Transfer podataka između *MMX* registara, te između ovih registara i memorije odn. registara opšte namjene se vrši običnim `MOV` instrukcijama. Karakteristično je međutim da se u ovom slučaju u *GNU* asembleru širina transfera ne naznačava prefiksom operanda, već sufiksom instrukcije. Tako se za 32-bitne transfere koristi sufiks `D`, a za 64-bitne sufiks `Q`. Sufiks je obavezan čak i kada operandi impliciraju širinu transfera (npr. kada se transfer vrši iz jednog *MMX* registra u drugi).

11.4 Aritmetičke instrukcije

Lista aritmetičkih *MMX* instrukcija je data u tabeli 11.1.

Mnemonik	Operacija
`PADDB/PADDW/PADDD`	sabiranje pakovanih cjelobrojnih podataka
`PSUBB/PSUBW/PSUBD`	oduzimanje pakovanih cjelobrojnih podataka
`PADDSB/PADDSW`	sabiranje pakovanih označenih vrijednosti sa saturacijom
`PSUBSB/PSUBSW`	oduzimanje pakovanih označenih vrijednosti sa saturacijom
`PADDUSB/PADDUSW`	sabiranje pakovanih neoznačenih vrijednosti sa saturacijom
`PSUBUSB/PSUBUSW`	oduzimanje pakovanih neoznačenih vrijednosti sa saturacijom
`PMULLW/PMULHW`	izračunavanje niže odn. više polovine proizvoda pakovanih podataka
`PMADDWD`	množenje pakovanih podataka i sabiranje proizvoda

Tabela 11.1: Aritmetičke *MMX* instrukcije.

Odredišni operand aritmetičkih instrukcija uvijek je *MMX* registar, a drugi operand može biti ili drugi *MMX* registar ili 64-bitni podatak iz memorije.

Instrukcije `PADD` i `PSUB` vrše sabiranje i oduzimanje u *wraparound* režimu i mogu da rade nad svim pakovanim tipovima podataka. Sabiranje i oduzimanje mogu da rade i u režimu saturacije (kako neoznačene, tako i označene) - ove operacije obavljaju instrukcije sa odgovarajućim sufiksima, koje mogu da rade samo nad 8-bitnim i 16-bitnim pakovanim podacima. Za množenje postoje dvije instrukcije i to `PMULLW` i `PMULHW`; množenje radi samo nad pakovanim 16-bitnim podacima i prva instrukcija smješta nižih 16 bita rezultata u odredišni pakovani 16-bitni operand dok druga instrukcija radi isto to za viših 16 bita rezultata. Postoji i instrukcija `PMADDWD` koja takođe množi dva

pakovana 16-bitna operanda, ali koja onda sumira rezultate dva množenja nižih odh. viših operanada i smješta ih u niža odn. viša 32 bita odredišta[2]. Treba reći i da se sve instrukcije množenja odnose na označene operande.

11.5 Instrukcije poređenja

MMX instrukcije za poređenje su nabrojane u tabeli 11.2.

Mnemonik	Operacija
PCMPEQB/PCMPEQW/PCMPEQD	poređenja na jednakost
PCMPGTB/PCMPGTW/PCMPGTD	poređenja na relaciju "veće"

Tabela 11.2: *MMX* instrukcije poređenja.

Od relacija poređenja podržane su samo poređenje na jednakost (instrukcijom `PCMPEQ`) i poređenje na relaciju "veće" (instrukcijom `PCMPGT`); ostale relacije se mogu izvesti kombinovanjem ovih instrukcija. Instrukcija `PCMPGT` radi nad označenim operandima. Ukoliko je uslov poređenja zadovoljen onda se svi bitovi rezultata koji odgovaraju datim operandima postavljaju na 1, a u suprotnom se postavljaju na 0. Za tipove operanada instrukcija poređenja važi isto što i za operande aritmetičkih instrukcija.

11.6 Instrukcije konverzije podataka

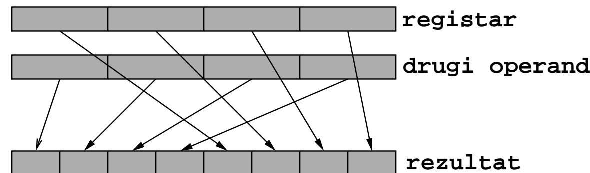
Instrukcije konverzije služe za preraspoređivanje pojedinačnih elemenata u okviru grupe pakovanih podataka. Spisak ovih instrukcija je dat na tabeli 11.3.

Mnemonik	Operacija
PACKSSWB/PACKSSDW	skraćenje podataka sa označenom saturacijom
PACKUSWB	skraćenje podataka sa neoznačenom saturacijom
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ	proširenje višeg dijela podatka na čitav podatak
PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ	proširenje nižeg dijela podatka na čitav podatak

Tabela 11.3: *MMX* instrukcije konverzije podataka.

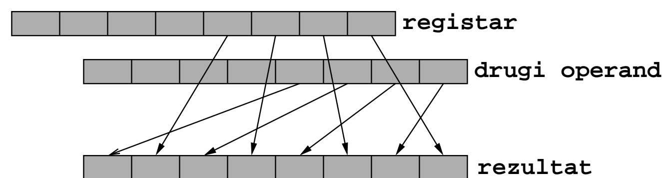
[2]opet je ovako specijalizovana instrukcija uvedena zato što je uočeno da je odgovarajuća operacija česta u posmatranom skupu aplikacija

Instrukcije PACKSSDW, PACKSSWB i PACKUSWB, vrše skraćenje i spajanje operanada i uvijek rade u režimu saturacije. Za označene brojeve, mogu se spajati 16-bitni i 32-bitni podaci, a za neoznačene brojeve samo 16-bitni podaci. Spajanje se vrši tako što se podaci iz dva operanda prosto odsijeku (u režimu saturacije) na dvostruko kraći tip podataka, a potom spoje u odredišnom operandu i to tako što se podaci koji su originalno bili u odredišnom operandu stave u niža 32 bita tog operanda, a podaci koji su originalno bili u drugom operandu u njegova viša 32 bita. Efekat skraćenja sa 16-bitnih na 8-bitne pakovane podatke je prikazan na slici 11.1.



Slika 11.1: Pakovanje podataka *MMX* instrukcijama.

Instrukcije PUNPCKLBW, PUNPCKLWD i PUNPCKLDQ uzimaju pakovane podatke iz niža 32 bita operanada i stavljaju ih u odredišni operand, pri čemu svaka dva podatka iz ta niža 32 bita operanada čine novi, dvostruko duži podatak na odgovarajućem mjestu u odredišnom operandu. Tako recimo instrukcija PUNPCKLBW ostavlja bitove 0-7 odredišnog operanda na svom mjestu, a na pozicije 8-15 odredišnog operanda stavlja bitove 0-7 drugog operanda; istovremeno se originalni bitovi 8-15 odredišnog operanda stavljaju na pozicije 16-23, a na pozicije 24-31 stavljaju se bitovi 8-15 drugog operanda itd. Efekat ove instrukcije je prikazan na slici 11.2. Instrukcije PUNPCKHBW, PUNPCKHWD i PUNPCKHDQ rade istu stvar samo što umjesto niža 32 uzimaju u obzir viša 32 bita operanada.



Slika 11.2: Raspakivanje podataka *MMX* instrukcijama.

Za tipove operanada operacije konverzije važi isto što i za aritmetičke odn. instrukcije poređenja.

11.7 Logičke instrukcije

Lista logičkih *MMX* instrukcija je data u tabeli 11.4.

Mnemonik	Operacija
`PAND`	paralelna bitska konjukcija
`PANDN`	paralelna bitska *NAND* operacija
`POR`	paralelna bitska disjunkcija
`PXOR`	paralelna bitska *XOR* operacija

Tabela 11.4: Logičke *MMX* instrukcije.

Ove instrukcije izvršavaju datu operaciju nezavisno nad pojedinačnim bitovima, tako da nemaju nikakve varijacije u pogledu dužine operanada. Za tipove operanada bitskih instrukcija važi isto što i za prethodne grupe instrukcija.

11.8 Instrukcije šiftovanja

MMX instrukcije za šiftovanje su nabrojane u tabeli 11.5.

Mnemonik	Operacija
`PSLLW/PSLLD/PSLLQ`	šiftovanje ulijevo
`PSRLW/PSRLD/PSRLQ`	logičko šiftovanje udesno
`PSRAW/PSRAD`	aritmetičko šiftovanje udesno

Tabela 11.5: *MMX* instrukcije šiftovanja.

Sve instrukcije šiftovanja mogu da rade nad pakovanim 16-bitnim i 32-bitnim podacima, a osim aritmetičkog šiftovanja udesno i nad 64-bitnim podacima. Tako su varijacije za logičko šiftovanje ulijevo (koje je istovremeno i aritmetičko šiftovanje ulijevo - kodovi za ove operacije kod *MMX* instrukcija nisu razdvojeni) `PSLLW`, `PSLLD` i `PSLLQ`. Kada se radi nad pakovanim podacima (tj. u prva dva slučaja), onda se kao i kod svih drugih operacija nad više takvih podataka ustvari istovremeno nezavisno izvršava specificirana operacija. Kodovi operacija za logičko odn. aritmetičko šiftovanje udesno su `PSRL` odn. `PSRA`. Operand koji određuje za koliko mjesta se vrši šiftovanje kod sve tri ove operacije može biti 8-bitna konstanta, drugi *MMX* registar (koji se tretira kao 64-bitni broj) ili 64-bitni broj u memoriji.

11.9 Upotreba *MMX* instrukcija

Obzirom da *MMX* instrukcije dijele stanje sa instrukcijama numeričkog koprocesora, pri uporedom korišćenju ovih skupova instrukcija u istom programu treba biti posebno pažljiv.

Prilikom upisa u registar neke *MMX* instrukcije prepisuje se mantisa odgovarajućeg registra iz registarskog steka. Takođe, pritom se u sve bitove eksponenta upisuju jedinice; dakle, sadržaj registra u smislu realnog podatka koji se ranije u njemu nalazio biva potpuno uništen. Zato se, ukoliko je potrebno sačuvati stanje registara koprocesora tokom izvršavanja sekvence *MMX* instrukcija i povratiti ga po završetku te sekvence, moraju iskoristiti odgovarajuće instrukcije koprocesora: `FSAVE` prije sekvence *MMX* instrukcija i `FRSTOR` nakon iste.

Štaviše, čak i ako nije potrebno čuvati stanje koprocesora tokom izvršavanja sekvence *MMX* instrukcija, potrebno je po završetku te sekvence preduzeti odgovarajuće korake da bi se mogle koristiti instrukcije koprocesora. Naime, svaka *MMX* instrukcija postavlja *tag* registra koprocesora na vrijednosti koje naznačavaju da se svaki registar koristi, a vrijednost koja označava vrh steka u statusnom registru na 0. Dakle, ako bi se u nizu *MMX* instrukcija izvršila neka instrukcija za rad sa realnim brojevima koja stavlja nešto na registarski stek, ista bi naišla na pun stek. Zato se nakon niza *MMX* instrukcija, a prije nego što se počne sa izvršavanjem instrukcija koprocesora, mora izvršiti instrukcija `EMMS` koja u *tag* registru naznačava da su svi registri prazni, a broj koji označava vrh steka u statusnom registru vraća na 7, čime se opet naznačava da je stek prazan.

Za čuvanje i vraćanje sadržaja *MMX* registara mogu se iskoristiti pomenute instrukcije koprocesora `FSAVE` i `FRSTOR`.

Korišćenje *MMX* instrukcija biće demonstrirano sledećom funkcijom koja vrši kompoziciju slika:

```
.intel_syntax noprefix

.text
.global blend

        ## Funkcija blend() vrsi kompoziciju dviju slika u skladu sa
        ## datim faktorom kompozicije. Ukoliko se faktor kompozicije
        ## predstavi razlomkom p/q, onda se svaka komponenta Cc datog
        ## piksela kompozitne clike racuna kao Cc=(p*C0+(q-p)*C1)/q,
        ## pri cemu su C0 i C1 odgovarajuce komponente piksela prve i
        ## druge slike. Pritom je svaki piksel predstavljen sa 4 bajta u
        ## memoriji gdje prvo dolazi R, zatim G, pa B komponenta dok je
        ## cetvrti bajt nevazan. Argumenti funkcije su:
        ##   [ebp+8] - brojilac faktora kompozicije
        ##   [ebp+12] - imenilac faktora kompozicije izrazen kao stepen dvojke
        ##   [ebp+16] - broj piksela u svakoj od slika
        ##   [ebp+20] - adresa na kojoj pocinju pikseli prve slike
        ##   [ebp+24] - adresa na kojoj pocinju pikseli druge slike
        ##   [ebp+28] - adresa na kojoj pocinju pikseli kompozitne slike
        ## Funkcija se u potpunosti podvrgava C konvencijama pozivanja.
        ## Pretpostavke funkcije su da brojilac faktora kompozicije nije
        ## veci od imenioca, te da imenilac nije veci od 128. Takodje je
        ## pretpostavka funkcije da se kod izvrsava na procesoru koji
        ## podrzava cpuid instrukciju; u tom slucaju funkcija moze da
        ## provjeri da li su MMX instrukcije podrzane.
```

```
blend:
        ## Prolog funkcije.
        enter   0, 0
        pusha

        ## Provjerava se da li procesor podrzava MMX instrukcije.
        mov     eax, 1
        cpuid
        test    edx, 0x800000
        jz done

        ## U registar mm0 se upisuje brojilac faktora kompozicije p. Ova
        ## vrijednost se upisuje u oba niza 16-bitna podregistra
        ## registra mm0.
        mov     eax, [ebp+8]
        movd    mm0, eax
        punpcklwd mm0, mm0

        ## U registar mm1 se upisuje razlika imenioca faktora
        ## kompozicije q (koji se prvo izracuna siftovanjem u registru
        ## bl) i brojioca tog faktora p. I ovdje se broj q-p upisuje u
        ## oba niza 16-bitna podregistra registra mm1.
        xor     ebx, ebx
        mov     bl, 1
        mov     ecx, [ebp+12]
        shl     bl, cl
        sub     ebx, eax
        movd    mm1, ebx
        punpcklwd mm1, mm1

        ## Kombinuju se registri mm0 i mm1 u registar mm0 tako da u
        ## najnizem 16-bitnom podregistru registra mm0 bude broj p, u
        ## narednom podregistru broj q-p, onda u narednom opet broj p i
        ## u najvisem ponovo broj q-p.
        punpcklwd mm0, mm1

        ## Imenilac faktora kompozicije izrazen kao stepen dvojke se
        ## pamti u registru mm7. U ovom slucaju registar mm7 se tretira
        ## kao cjelina tj. kao 64-bitni broj.
        movd    mm7, ecx

        ## U registar mm6 se upisuje 0.
        pxor    mm6, mm6

        ## U registar ecx se upisuje broj piksela i ovaj registar ce
        ## sluziti kao brojac. U registar esi se upisuje adresa prve
        ## slike, a u registar edi adresa kompozitne slike. U registar
        ## ebx se stavlja adresa druge slike.
        cld
        mov     ecx, [ebp+16]
        mov     esi, [ebp+20]
        mov     ebx, [ebp+24]
        mov     edi, [ebp+28]

next_pixel:
        ## Ucitava se naredni piksel prve slike i prebacuje u registar
        ## mm1 tako sto se u svaki njegov 16-bitni podregistar upise
        ## odgovarajuca komponenta piksela.
        lodsd
        movd    mm1, eax
        punpcklbw mm1, mm6
```

```
        ## Ucitava se naredni piksel druge slike i prebacuje u registar
        ## mm2 na isti nacin kao sto je odgovarajuci piksel prve slike
        ## stavljen u registar mm1. Prilikom ucitavanja piksela,
        ## privremeno se njegova adresa prebacuje iz registra ebx u
        ## registar esi.
        xchg    esi, ebx
        lodsd
        xchg    esi, ebx
        movd    mm2, eax
        punpcklbw mm2, mm6

        ## Nize dvije komponente piskela se kombinuju u registru mm3,
        ## tako da u najnizi 16-tobitni podregistar ovog registra dodje
        ## najniza komponenta piksela prve slike, u naredni 16-tobini
        ## podregistar najniza komponenta piksela druge slike, potom
        ## druga komponenta piksela prve slike i na kraju druga
        ## komponenta piksela druge slike. Ove komponente se u paraleli
        ## mnoze sa faktorima p odn. q-p cime se za svaku od njih
        ## formira izraz p*C0+(q-p)*C1. Za izracunavanje odgovarajucih
        ## komponenti kompozitne slike potrebno je jos podijeliti ove
        ## izraze sa q, sto se postize siftovanjem; ove komponente po
        ## zavrsetku ovog izracunavanja se nalaze respektivno u nizem odn.
        ## visem 32-bitnom podregistru registra mm3.
        movq    mm3, mm1
        punpcklwd mm3, mm2
        pmaddwd mm3, mm0
        psrld   mm3, mm7

        ## Na isti nacin kao nize dvije, odredjuju se vise dvije
        ## komponente piksela kompozitne slike, koje bivaju upisane u
        ## registru mm1.
        punpckhwd mm1, mm2
        pmaddwd mm1, mm0
        psrld   mm1, mm7

        ## Pakuju se finalne vrijednosti tako da svaka komponenta
        ## piksela stane u jedan bajt.
        packssdw mm3, mm1
        packuswb mm3, mm6

        ## Rezultujuci piksel se upisuje u kompozitnu sliku.
        movd    eax, mm3
        stosd

        ## Prelazi se na naredni piksel.
        loop next_pixel

        ## Naznacava se kraj rada sa MMX instrukcijama.
        emms

done:
        ## Epilog funkcije.
        popa
        leave
        ret
```

Pri kompoziciji, za svaku komponentu C svakog piksela odredišne slike kombinuju se odgovarajuće komponente C_0 i C_1 odgovarajućih piksela dviju slika koje ulaze u kompoziciju. Kombinovanje se vrši uz skaliranje odgovarajućim faktorom kompozicije $s \in [0, 1]$ tako da jednačina za rezultujuću komponentu

kompozitne slike glasi:

$$C = sC_0 + (1 - s)C_1$$

Komponente piksela slike su obično predstavljene cijelim brojevima u opsegu $[0, 255]$ pri čemu 0 predstavlja minimalni, a 255 maksimalni intenzitet za datu komponentu. Da bi *MMX* registri mogli da se koriste za ovakvo izračunavanje, mora se gornja operacija svesti na cjelobrojnu aritmetiku. U tom cilju se predstavlja faktor kompozicije kao količnik dva cijela broja $s = \frac{p}{q}$ i jednačina se preuređuje na oblik:

$$C = \frac{pC_0 + (q - p)C_1}{q}$$

Obzirom da *MMX* instrukcije ne podržavaju dijeljenje, da bi se sve operacije u vezi sa kompozicijom obavile u *MMX* registrima uzeto je da će imenilac faktora kompozicije uvijek biti stepen broja 2, tako da se dijeljenje može obaviti šiftovanjem.

Na početku funkcije se provjerava da li procesor podržava *MMX* instrukcije, a potom se vrijednosti p i $q - p$ kombinuju u jednom *MMX* registru na način koji će kasnije biti pogodan za paralelno izračunavanje. Po inicijalizaciji još nekih *MMX* registara na vrijednosti koje će biti često korišćene tokom izračunavanja, ulazi se u petlju u kojoj se kombinuje jedan po jedan piksel datih slika. U prvom koraku u petlji se komponuju dvije niže komponente slika, a u drugom koraku dvije više komponente. Operandi se pritom kombinuju na način koji omogućava da se sama kompozicija obavi samo jednom `PMADDWD` instrukcijom. Pri kombinovanju operanada tokom pripreme za kompoziciju, kao i kasnije tokom grupisanja komponenti rezultujućeg piksela, intenzivno se koriste operacije raspakivanja i pakovanja. Formule po kojima se računa garantuju da će komponente rezultujućeg piksela takođe stati u opseg $[0, 255]$ odn. u jedan bajt. Izračunati piksel rezultujuće slike se po izračunavanju smješta na odgovarajuću lokaciju u memoriji i petlja se nastavlja nad narednim pikselima polaznih slika.

Obzirom da se *MMX* instrukcijama obrađuju istovremeno sve 4 komponente piksela, dok bi običnim instrukcijama bilo moguće obrađivati ih samo jednu po jednu, jasno je da će u datom primjeru korišćenje *MMX* ekstenzija donijeti znatne dobitke u brzini izvršavanja. Generalno, primjena ovih ekstenzija je međutim ograničena na jedan veoma uski skup aplikacija. Zbog iregularnosti *MMX* instrukcija teško je neke opšte algoritme paralelizovati njihovim korišćenjem. Ove instrukcije se takođe ne uklapaju dobro u programerski model koji nude ostale *IA-32* instrukcije. To su i razlozi što malo koji prevodilac uopšte koristi ove instrukcije tako da se njihova upotreba svodi na biblioteke koje rješavaju probleme iz seta aplikacija koje ove instrukcije pokrivaju i

koje su ručno kodirane u asembleru. Treba uočiti da je to rijedak primjer situacije kada programiranje u asembleru nema adekvatnu alternativu.

Poglavlje 12

SSE instrukcije

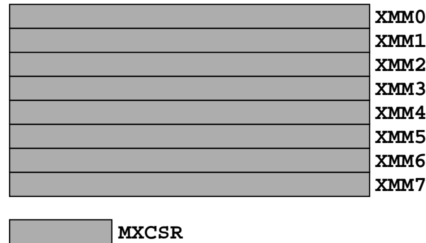
Sa trendom uvođenja paralelnih instrukcija se nastavilo i nakon uvođenja *MMX* seta instrukcija, pa su tako u *Pentium III* generaciji uvedene instrukcije koje rade paralelno nad više podataka koji su realni brojevi. Ove instrukcije se označavaju terminom *SSE* (*streaming SIMD extensions*). Dalja unapređenja ovog skupa instrukcija su došla sa uvođenjem *Pentium IV* generacije procesora; proširenja koje donosi ovaj procesor se označavaju terminom *SSE2*.

Iako se instrukcije iz nabrojanih skupova izvršavaju paralelno nad manjim grupama podataka, te se u *Intel*-ovoj terminologiji označavaju terminom *SIMD*, treba imati na umu da je ipak što se paralelizma tiče u pitanju dosta primitivna forma paralelnog procesiranja u odnosu na onaj tip procesiranja na koji se ovaj termin obično odnosi.

SSE ekstenzije su konceptualno slične *MMX* ekstenzijama u smislu da se radi o skupu instrukcija namijenjenom prevashodno tačno određenoj ciljnoj grupi aplikacija; kod *SSE* ekstenzija se išlo prije svega na ubrzavanje klase aplikacija koje računaju u pokretnom zarezu (npr. 3D grafika, digitalna obrada signala itd.). Slično *MMX* instrukcijama, koje mogu istovremeno da izvrše više operacija nad cjelobrojnim operandima, *SSE* instrukcije omogućavaju da se 4 operacije izvrše istovremeno nad realnim operandima u jednostrukoj tačnosti (tj. dužine 32 bita). Može se međutim reći da je skup *SSE* instrukcija bolje zaokružen u odnosu na *MMX* instrukcije, odn. da *SSE* predstavlja bolju implementaciju ugradnje nekih *SIMD* koncepata u *IA-32* kao inherentno *SISD* (*single instruction, single data*) arhitekturu, tako da se *SSE* instrukcije mogu dobro iskoristiti i mimo ciljne grupe aplikacija, odn. za paraleno procesiranje u pokretnom zarezu opšteg tipa.

12.1 Skup registara za *SSE* instrukcije

Za razliku od dizajna *MMX* instrukcija, koje dijele skup registara nad kojima operišu sa instrukcijama za rad u pokretnom zarezu, za potrebe *SSE* instrukcija je skup registara procesora proširen sa 8 novih 128-bitnih registara. Ovi registri se označavaju imenima `XMM0`, `XMM1` i tako redom do `XMM7`. Takođe, *SSE* ekstenzije podrazumijevaju i prisustvo 32-bitnog statusno-kontrolnog registra `MXCSR`. Svi *SSE* registri su prikazani na slici 12.1.



Slika 12.1: Skup registara koje uvode *SSE* ekstenzije.

SSE instrukcije uvijek tretiraju sadržaj *SSE* registara kao 4 realna 32-bitna (*single precision*) broja. Pritom, većina instrukcija postoji u dvije varijacije, pri čemu se u prvoj varijaciji izvršava data operacija paralelno nad sva 4 realna broja sadržana u datom registru odn. registrima (ova varijacija se prepoznaje po sufiksu instrukcije `PS`), dok se u drugoj varijaciji data operacija izvršava samo nad realnim brojem sadržanim u 32 bita najmanje težine datog registra odn. registara (ova varijacija se prepoznaje po sufiksu instrukcije `SS`)[1].

Statusno-kontrolni registar `MXCSR` objedinjava u sebi funkcionalnost koju u numeričkom procesoru imaju odvojeni statusni i kontrolni registar. Ovaj registar sadrži naime 6 flegova koji se postavljaju prilikom pojave odgovarajućih grešaka tokom izračunavanja, i to: pogrešna operacija, prisustvo denormalizovanog operanda, dijeljenje sa nulom, prekoračenje, potkoračenje i neprecizno izračunavanje. Takođe, postoji i 6 odgovarajućih bitova u ovom registru kojima se može pojedinačno zapriječiti generisanje prekida kada se pojavi bilo koja od ovih grešaka tokom izračunavanja. Dva bita statusno-kontrolnog registra su odvojena za kontrolu zaokruživanja, koja funkcioniše na isti način kao sa numeričkim koprocesorom. Preostala dva aktivna bita ovog registra omogućavaju da se tokom izračunavanja denormalizovani rezultat odn. operandi automatski pretvore u 0; ovakav režim rada nije u skladu sa *IEEE-754* standardom (kome se inače *SSE* ekstenzije potpuno podvrgavaju), ali je pogodan za ubrzavanje izračunavanja u aplikacijama kod kojih numer-

[1] ukoliko je u ovom slučaju odredišni operand *SSE* registar, viši bitovi tog registra ostaju neizmijenjeni

ička preciznost nije od kritične važnosti (a upravo takve su aplikacije kojima su namijenjene *SSE* ekstenzije).

Kao i za *MMX* ekstenzije, tako se i za *SSE* ekstenzije prije korišćenja odgovarajućih registara odn. instrukcija mora provjeriti da li ih procesor na kome se dati program izvršava podržava. U tu svrhu se može iskoristiti `CPUID` instrukcija pokrenuta sa vrijednošću 1 u *EAX* registru; ukoliko je nakon toga bit 25 *EDX* registra jednak 1, to znači da su *SSE* ekstenzije podržane.

12.2 Instrukcije za transfer podataka

SSE instrukcije za transfer podataka služe za prebacivanje podataka između *SSE* registara, kao i iz memorije u *SSE* registar i obrnuto. Lista ovih instrukcija je data u tabeli 12.1.

Mnemonik	Operacija
`MOVAPS`	prebacivanje 4 uravnata podatka (32-bitna realna broja)
`MOVUPS`	prebacivanje 4 neuravnata podatka
`MOVSS`	prebacivanje jednog podatka
`MOVLPS`	prebacivanje 2 podatka između memorije i nižeg dijela registra
`MOVHPS`	prebacivanje 2 podatka između memorije i višeg dijela registra
`MOVLHPS`	prebacivanje 2 podatka iz nižeg u viši dio registra
`MOVHLPS`	prebacivanje 2 podatka iz višeg u niži dio registra
`MOVMSKPS`	upisivanje bitova znaka sva 4 podatka u registar opšte namjene
`STMXCSR`	snimanje statusno-kontrolnog registra u memoriju
`LDMXCSR`	učitavanje statusno-kontrolnog registra iz memorije
`FXSAVE`	snimanje svih registara numeričkog procesora i *SSE* registara u memoriju
`FXRSTOR`	učitavanje svih registara numeričkog procesora i *SSE* registara iz memorije

Tabela 12.1: *SSE* instrukcije za transfer podataka.

Instrukcije `MOVAPS` odn. `MOVUPS` prebacuju 4 podatka dužine 32 bita između dva *SSE* registra, odn. između jednog registra i memorije. Ukoliko je jedan operand memorijska adresa, za prvu instrukciju ta adresa mora biti djeljiva sa 16 ili će u suprotnom biti generisana greška, dok za drugu instrukciju to ne mora biti slučaj[2].

Instrukcija `MOVSS` prebacuje samo jedan 32-bitni podatak između *SSE* registara odn. između *SSE* registra i memorije. Treba uočiti `SS` sufiks instrukcije

[2]ali je zato transfer brži u prvom slučaju

koji, kako je već pomenuto, označava da se operacija vrši samo nad jednim 32-bitnim podatkom (za razliku od PS sufiksa koji, kao kod dvije prethodne instrukcije, označava da se operacija vrši istovremeno nad sva 4 podatka). Kao i kod drugih instrukcija ovog tipa, uvijek se radi sa podatkom koji se nalazi u 32 bita najmanje težine odgovarajućeg *SSE* registra.

Instrukcije MOVLPS i MOVHPS služe za prebacivanje 2 podatka između *SSE* registra i memorije. U prvom slučaju u transfer su uključena 2 podatka iz niža, a u drugom 2 podatka iz viša 64 bita odgovarajućeg *SSE* registra.

Instrukcije MOVLHPS i MOVHLPS služe za prebacivanje 2 podatka između 2 *SSE* registra, pri čemu se u prvom slučaju prebacuju niža 64 bita polaznog registra u viša 64 bita odredišnog registra, a u drugom slučaju je obrnuto.

Instrukcija MOVMSKPS upisuje bitove znaka (bitove najveće težine) svakog od 4 podatka iz datog *SSE* registra u 4 bita najmanje težine datog registra opšte namjene. Ova vrijednost se obično nakon toga koristi za neki vid grananja.

Instrukcije STMXCSR odn. LDMXCSR služe za snimanje sadržaja statusno-kontrolnog registra MXCSR na datu memorijsku lokaciju, odn. za učitavanje sadržaja ovog registra iz memorije.

Instrukcije FXSAVE odn. FXRSTOR služe za brzo snimanje odn. učitavanje kompletnog stanja registara numeričkog koprocesora odn. *SSE* registara u memoriju, odn. iz memorije. Ove instrukcije predstavljaju ekstenzije ranije pominjanih instrukcija FSAVE odn. FRSTOR koje rade iste stvari, ali samo sa registrima numeričkog koprocesora.

12.3 Aritmetičko-logičke instrukcije

Skup aritmetičko-logičkih *SSE* instrukcija je dat u tabeli 12.2.

Aritmetičko-logičke *SSE* instrukcija su vrlo regularne. Od artimetičkih operacija podržano je sabiranje, oduzimanje, množenje, dijeljenje, te određivanje recipročne vrijednosti, kvadratnog korijena, recipročne vrijednosti kvadratnog korijena, minimuma i maksimuma operanada. Za sve ove instrukcije operandi mogu biti dva *SSE* registra ili jedan registar i memorijska lokacija. Sve instrukcije postoje u varijaciji sa PS sufiksom u kom slučaju se operacija paralelno izvršava nad 4 para operanada, kao i u varijaciji sa SS sufiksom kada se operacija izvršava nad samo jednim parom operanada. Logičke instrukcije (baš kao i odgovarajuće *MMX* instrukcije) rade samo nad sva 4 podatka istovremeno, i to tako što se odgovarajuća logička operacija pojedinačno primjenjuje nad svakim od ukupno 128 bitskih parova.

Važno je napomenuti da sve instrukcije iz ove grupe koje rade paralelno nad 4 operanda, kao i sve ostale *SSE* instrukcije tog tipa koje će biti predstavljene u nastavku, u slučaju kada je drugi operand neka memorijska adresa očekuju

Mnemonik	Operacija
`ADDPS/ADDSS`	sabiranje operanada
`SUBPS/SUBSS`	oduzimanje operanada
`MULPS/MULSS`	množenje operanada
`DIVPS/DIVSS`	dijeljenje operanada
`RCPPS/RCPSS`	izračunavanje recipročne vrijednosti operanada
`SQRTPS/SQRTSS`	izračunavanje kvadratnog korijena operanada
`RSQRTPS/RSQRTSS`	izračunavanje recipročne vrijednosti kvadratnog korijena operanada
`MAXPS/MAXSS`	izračunavanje većeg od operanada
`MINPS/MINSS`	izračunavanje manjeg od operanada
`ANDPS`	bitska konjukcija operanada
`ANDNPS`	negacija bitske konjukcije operanada
`ORPS`	bitska disjunkcija operanada
`XORPS`	bitska operacija *XOR* nad operandima

Tabela 12.2: Aritmetičko-logičke *SSE* instrukcije.

da ta adresa bude djeljiva sa 16. Pošto to najčešće nije slučaj, onda se obično odgovarajući sadržaj prebaci iz memorije u neki *SSE* registar instrukcijom `MOVUPS`, da bi se potom odgovarajuća operacija izvršila nad registrima kao operandima.

12.4 Instrukcije poređenja

SSE instrukcije poređenja su nabrojane u tabeli 12.3.

Mnemonik	Operacija
`CMPPS`	istovremeno poređenje 4 para podataka
`CMPSS`	poređenje jednog para podataka
`COMISS`	poređenje jednog para podataka i postavljanje flegova `EFLAGS` registra
`UCOMISS`	isto kao prethodno uz "neuređeno" poređenje

Tabela 12.3: *SSE* instrukcije poređenja.

Instrukcije `CMPPS` i `CMPSS` imaju tri operanda, pri čemu je prvi operand *SSE* registar, drugi operand je takođe *SSE* registar ili memorijska lokacija, dok je treći operand 8-bitna vrijednost koja određuje prema kojoj relaciji se vrši poređenje prva dva operanda. Kao i u svim slučajevima instrukcija sa varijacijama u `PS` i `SS` sufiksima, prva varijacija vrši istovremeno poređenje 4 podatka, dok druga varijacija vrši poređenje samo podataka koji se nalaze u 32 bita najmanje težine operanada. Rezultat svakog poređenja je (kao kod

MMX instrukcija poređenja) niz od 32 jedinice (ako je data relacija zadovoljena) odn. nule (ako relacija nije zadovoljena) koji se upisuje na mjesto odredišnog operanda. Vrijednosti koje određuju vrstu poređenja su date u tabeli 12.4.

Vrijednost	Vrsta poređenja
0	jednako
1	manje
2	manje ili jednako
3	poređenje na "neuređenost"
4	nije jednako
5	nije manje
6	nije manje ili jednako
7	poređenje na "uređenost"

Tabela 12.4: Vrijednosti trećeg operanda `CMPPS` i `CMPSS` instrukcija.

Neuređeni brojevi su diskutovani u vezi sa instrukcijama poređenja numeričkog koprocesora. Poređenje na "neuređenost" je zadovoljeno ako je bar jedan operand neuređen broj, dok je poređenje na "uređenost" zadovoljeno samo ako su oba operanda uređeni brojevi.

Instrukcije `COMISS` i `UCOMISS` imaju dva operanda (dva *SSE* registra ili *SSE* registar i memorijsku lokaciju) i vrše poređenje podataka iz 32 bita najmanje težine operanada uz upisivanje rezultata poređenja u odgovarajuće bitove `EFLAGS` registra na sličan način kao što to radi cjelobrojna instrukcija poređenja. Kao što je rečeno za slične instrukcije koprocesora `FCOMI` i `FUCOMI`, postavljanje flegova u `EFLAGS` registru više odgovara poređenju neoznačenih cijelih brojeva, pa onda nakon ovakvih poređenja treba koristiti odgovarajuću grupu instrukcija uslovnog skoka. Razlika između dvije navedene instrukcije je što prva instrukcija generiše grešku tipa pogrešna operacija ako su operandi neuređeni, dok druga instrukcija ne generiše ovu grešku u tom slučaju.

12.5 Instrukcije konverzije podataka

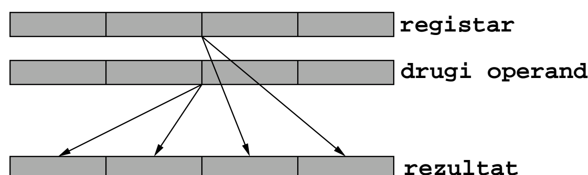
SSE instrukcije za preuređivanje podataka, kao i za konverziju između realnih i cjelobrojnih podataka su nabrojane u tabeli 12.5.

Instrukcija `SHUFPS` ima tri operanda, pri čemu je odredišni operand *SSE* registar, drugi operand opet *SSE* registar ili memorijska lokacija, a treći operand 8-bitna vrijednost. Po 2 bita iz ove 8-bitne vrijednosti određuju koji će se podatak iz svakog od operanada naći u odgovarajuća 32 bita u odredišnom operandu. Pritom, 2 para od po 2 bita u 4 niža bita ove vrijednosti se odnose na podatke iz prvog operanda, a 2 para od po 2 bita u 4 viša

Mnemonik	Operacija
SHUFPS	kopiranje po 2 podatka iz svakog operanda u niža odn. viša 64 bita odredišnog operanda
UNPCKHPS	naizmjenično kopiranje po 2 podatka iz viša 64 bita operanada u odredišni operand
UNPCKLPS	naizmjenično kopiranje po 2 podatka iz niža 64 bita operanada u odredišni operand
CVTPI2PS	konverzija 2 cjelobrojna podatka iz *MMX* registra ili memorije u 2 realna podatka u *SSE* registru
CVTSI2SS	konverzija jednog cjelobrojnog podatka iz registra opšte namjene ili memorije u realan podatak u *SSE* registru
CVTPS2PI	konverzija 2 realna podatka iz *SSE* registra ili memorije u 2 cjelobrojna podatka u *MMX* registru
CVTSS2SI	konverzija jednog realnog podatka iz *SSE* registra ili memorije u cjelobrojni podatak u registru opšte namjene

Tabela 12.5: *SSE* instrukcije za preuređivanje i konverziju podataka.

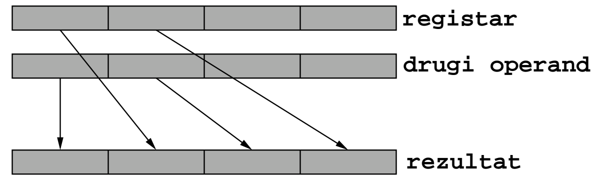
bita ove vrijednosti se odnose na podatke iz drugog operanda. Efekat ove instrukcije je prikazan na slici 12.2.



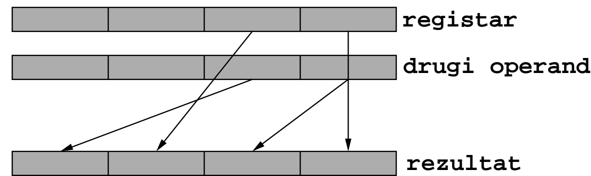
Slika 12.2: Preuređivanje podataka `SHUFPS` instrukcijom.

Instrukcije `UNPCKHPS` odn. `UNPCKLPS` primaju kao odredišni operand *SSE* registar, a kao drugi operand *SSE* registar ili memorijsku lokaciju i smještaju u odredišni operand po 2 podatka iz viših odn. nižih 64 bita operanada. Smještanje podataka je pritom naizmjenično: podatak koji zauzima niža 32 od data 64 bita u prvom operandu ide na 32 bita najmanje težine u odredišnom operandu, podatak koji zauzima niža 32 od data 64 bita u drugom operandu ide na naredna 32 bita u odredišnom operandu i tako redom. Efekat ove dvije instrukcije je prikazan na slikama 12.3 odn. 12.4.

Instrukcije `CVTPI2PS` i `CVTPS2PI` vrše konverziju između 2 realna i 2 cjelobrojna podatka, dok instrukcije `CVTSI2SS` i `CVTSS2SI` vrše konverziju između jednog realnog i jednog cjelobrojnog podatka. Kod svih instrukcija, polazni operand može biti u memoriji ili u odgovarajućem registru, dok je odredišni operand uvijek registar. Za realne podatke je taj registar uvijek *SSE* registar, dok je kod cjelobrojnih podataka u slučaju prve dvije instrukcije to neki *MMX* registar, a u slučaju druge dvije instrukcije neki od registara opšte



Slika 12.3: Preuređivanje podataka `UNPCKHPS` instrukcijom.



Slika 12.4: Preuređivanje podataka `UNPCKLPS` instrukcijom.

namjene[3]. U slučaju da prilikom konverzije treba da dođe do gubitka preciznosti, zaokruživanje se vrši u skladu sa tekućim režimom zaokruživanja određenim odgovarajućim bitovima statusno-kontrolnog registra.

12.6 Ostale instrukcije uvedene *SSE* ekstenzijama

SSE ekstenzije su donijele i dvije nove grupe instrukcija koje ne barataju *SSE* registrima i to su instrukcije koje proširuju skup *MMX* instrukcija i instrukcije za kontrolu keširanja podataka. Lista dodatnih *MMX* instrukcija je prikazana u tabeli 12.6.

Instrukcije `PAVGB` i `PAVGW` računaju poluzbir neoznačenih pakovanih 8-bitnih ili 16-bitnih podataka. Odredišni operand ovih instrukcija je *MMX* registar, dok je drugi operand ili *MMX* registar ili memorijska lokacija. Poluzbir se računa tako što se saberu operandi i zbir uveća za 1, a zatim se tako dobijena vrijednost šiftuje udesno za 1 bit.

Instrukcije `PEXTRW` i `PINSRW` prevazilaze problem nemogućnosti razmjene podataka između *MMX* registara i registara opšte namjene koji je postojao u osnovnom skupu *MMX* instrukcija. Obje instrukcije vrše transfer 16-bitnog podatka i imaju tri operanda. Prva dva operanda su *MMX* registar i registar opšte namjene, pri čemu je odredišni operand kod prve instrukcije registar opšte namjene, a kod druge *MMX* registar[4], dok je treći operand 8-bitna vrijednost koja određuje koji je od 4 podatka iz *MMX* registra uključen

[3]treba primijetiti da ova grupa instrukcija donosi prednost dizajnu skupa *SSE* instrukcija u odnosu na dizajn skupa *MMX* instrukcija, u kome inicijalno nema instrukcija za razmjenu podataka između *MMX* registara i registara opšte namjene

[4]kod druge instrukcije drugi operand može biti i memorijska lokacija

Mnemonik	Operacija
`PAVGB/PAVGW`	računanje poluzbira pakovanih cjelobrojnih podataka
`PEXTRW`	kopiranje podatka iz *MMX* registra u registar opšte namjene
`PINSRW`	kopiranje podatka iz registra opšte namjene ili memorije u *MMX* registar
`PMAXUB/PMAXSW`	računanje maksimuma pakovanih podataka
`PMINUB/PMINSW`	računanje minimuma pakovanih podataka
`PMOVMSKB`	upisivanje bitova znaka pakovanih podataka u registar opšte namjene
`PMULHUW`	izračunavanje više polovine proizvoda pakovanih podataka
`PSADBW`	izračunavanje sume apsolutnih vrijednosti razlika pakovanih podataka
`PSHUFW`	preuređivanje pakovanih podataka

Tabela 12.6: *MMX* instrukcije u *SSE* ekstenzijama.

u transfer. Instrukcija `PINSRW` takođe anulira sve ostale podatke u datom *MMX* registru.

Instrukcije `PMAXUB`, `PMAXSW`, `PMINUB` i `PMINSW` imaju 2 operanda i to *MMX* registar kao odredišni operand, a *MMX* registar ili memorijsku lokaciju kao drugi operand. Ove instrukcije računaju maksimum odn. minimum pakovanih neoznačenih 8-bitnih odn. označenih 16-bitnih podataka.

Instrukcija `PMOVMSKB` je pandan `MOVMSKPS` *SSE* instrukciji - ova instrukcija uzima bitove znaka svakog od 8-bitnih podataka iz datog *MMX* registra i upisuje ih u 8 bitova najmanje težine datog registra opšte namjene.

Instrukcija `PMULHUW` radi isto što i `PMULHW` instrukcija iz osnovnog skupa *MMX* instrukcija, osim što tretira operande kao neoznačene umjesto označenih brojeva.

Instrukcija `PSADWB` računa apsolutne vrijednosti razlika svakog para 8-bitnih podataka iz operanada (operandi su isti kao npr. za `PAVGB` instrukciju), a zatim sumira ove razlike i smješta sumu u 16 bita najmanje težine odredišnog registra, dok ostatak ovog registra postavlja na 0.

Instrukcija `PSHUFW` je donekle slična `SHUFPS` *SSE* instrukciji. Ova instrukcija ima tri operanda, pri čemu su prva dva operanda ista kao npr. kod `PAVGW` instrukcije, a treći operand je 8-bitna vrijednost. Svaka 2 bita iz ove 8-bitne vrijednosti određuju koji će se 16-bitni podatak iz drugog operanda naći u odgovarajućih 16 bitova u odredišnom operandu.

Instrukcije za kontrolu keširanja koje su dodate zajedno sa *SSE* ekstenzijama su nabrojane u tabeli 12.7.

Mnemonik	Operacija
`MOVNTQ`	smještanje sadržaja *MMX* registra u memoriju mimoilaženjem *cache*-eva
`MOVNTPS`	smještanje sadržaja *SSE* registra u memoriju mimoilaženjem *cache*-eva
`MASKMOVQ`	smještanje dijela sadržaja *MMX* registra u memoriju mimoilaženjem *cache*-eva
`PREFETCHT0/PREFETCHT1/PREFETCHT2`	dovlačenje temporalnih podataka iz memorije u dati nivo *cache*-a
`PREFETCHNTA`	dovlačenje netemporalnih podataka iz memorije u *cache*-a
`SFENCE`	serijalizacija prethodnih upisa u memoriju

Tabela 12.7: Instrukcije za kontrolu keširanja u *SSE* ekstenzijama.

U cilju ubrzavanja pristupa memoriji, svi savremeni procesori, pa tako i procesori iz *IA-32* arhitekture upošljavaju više nivoa *cache* memorije između procesora i glavne memorije. Postojanje hijerarhije *cache* memorija povlači za sobom potrebu postojanja složene logike u procesoru koja pokušava da na optimalan način iskoristi ovu hijerarhiju, a takođe i da obezbijedi koherenciju u slučaju kada određenim segmentima memorije pored procesora pristupaju i druge komponente sistema. Jedan aspekt optimalnog korišćenja hijerarhije *cache* memorija podrazumijeva podjelu sadržaja memorije na blokove lokacija temporalnog i netemporalnog tipa. Lokacije temporalnog tipa su one lokacije za koje kad im procesor pristupi postoji velika vjerovatnoća da će im uskoro ponovo pristupiti (kao npr. niz instrukcija u nekoj petlji) dok su lokacije netemporalnog tipa one lokacije kojima procesor u nekom dužem vremenskom intervalu pristupa samo jednom (kao npr. odbirci nekog zvučnog signala koji se preslušava). Jasno je da temporalne blokove ima puno smisla držati u *cache* memoriji, dok suprotno važi za netemporalne blokove. Postoje razni mehanizmi kojima procesor nastoji da odredi (ili kojima se procesoru može ukazati) koji su blokovi memorije temporalnog, a koji netemporalnog tipa. Obzirom da su podaci koji se obrađuju *SSE* instrukcijama (kao i *MMX* instrukcijama) uglavnom netemporalnog tipa, *SSE* ekstenzije uvode i određen broj instrukcija kojima programer eksplicitno naznačava procesoru kakvog su tipa podaci kojima te instrukcije barataju i na taj način mu pomaže da bolje iskoristi hijerarhiju *cache* memorija.

Instrukcija `MOVNTQ` smješta sadržaj datog *MMX* registra u memoriju baš kao `MOVQ` instrukcija, ali se korišćenjem ove instrukcije naznačava procesoru da su u pitanju netemporalni podaci, te da ih treba smjestiti direktno u radnu memoriju i ne zadržavati ih na bilo kom nivou *cache* memorije. In-

strukcija `MOVNTPS` radi potpuno istu stvar za dati *SSE* registar (dakle, ova instrukcija je ekvivalentna `MOVAPS` odn. `MOVUPS` instrukcijama, ali se njome naznačava da su podaci koji se prebacuju u memoriju netemporalnog tipa). Instrukcija `MASKMOVQ` je slična `MOVNTQ` instrukciji, ali se ovdje u memoriju ne smješta sadržaj čitavog *MMX* registra već samo određeni bajtovi iz njega; koji će bajtovi biti smješteni u memoriju određeno je bitom znaka *MMX* registra koji predstavlja drugi operand, dok je adresa memorijske lokacije na koju se smještaju ovi podaci određena sadržajem `EDI` registra. Instrukcijom `MASKMOVQ` se takođe naglašava da su podaci koji se smještaju u memoriju netemporalnog tipa.

Instrukcijama sa prefiksom `PREFETCH` se prenosi blok veličine sa kojim barataju *cache* memorije (obično 32 bajta) koji sadrži adresu specificiranu kao jedini argument ovih instrukcija u nivo *cache* memorije određen sufiksom instrukcije. Sufiksima `T0`, `T1` i `T2` se naznačava da se radi o bloku temporalnog, dok se sufiksom `NTA` naznačava da se radi o bloku netemporalnog tipa. Broj na kraju prva tri sufiksa se odnosi na nivo *cache* memorije u koju se želi dovući dati blok, ali ne predstavlja eksplicitnu oznaku nivoa već njegovo značenje zavisi od date implementacije *IA-32* arhitekture. Tako npr. za *Pentium III* procesore sufiks `T0` se odnosi na prvi ili drugi, a sufiksi `T1` i `T2` na drugi nivo *cache* memorije, dok se za *Pentium IV* svi sufiksi odnose na drugi nivo. Slično je i sa blokovima netemporalnih podataka - instrukcija `PREFETCHNTA` na *Pentium III* procesoru dovlači odgovarajuće podatke u prvi, a na *Pentium IV* procesoru u drugi nivo *cache* memorije.

Ukoliko procesor prepozna neke podatke kao temporalne, onda se obično prilikom upisa u takve memorijske lokacije ažurira samo odgovarajući sadržaj u određenim nivoima *cache* memorije, a ne i u glavnoj memoriji sa idejom da će možda taj sadržaj biti uskoro ponovo promijenjen, čime bi se onda uštedilo na upisima u glavnu memoriju. Instrukcija `SFENCE` služi da svi takvi ranije izvršeni parcijalni upisi budu propagirani u glavnu memoriju. Ova instrukcija je važna za *SSE* ekstenzije jer se podaci koji se obrađuju *SSE* instrukcijama obično potom prosleđuju nekom drugom procesoru (npr. grafičkom ili *DSP* procesoru), tako da mora postojati sredstvo da se osigura da je sadržaj memorije u koju su smještani procesirani podaci potpuno ažuriran; `SFENCE` instrukcija predstavlja takvo sredstvo.

12.7 Upotreba *SSE* instrukcija

Kako je već pomenuto, *SSE* ekstenzije imaju znatno širu upotrebljivost u odnosu na *MMX* ekstenzije i mogu se iskoristiti za opšta izračunavanja u pokretnom zarezu, a ne samo za aplikacije iz domena kome su primarno bile namijenjene. Upotreba *SSE* instrukcija biće demonstrirana procedurom koja implementira dati *FIR* (*finite-duration impuls response*)

filter nad datim ulaznim podacima. *FIR* filter je određen vektorom koeficijenata $[c_0\ c_1\ \ldots\ c_{m-1}]$ $(m > 0)$ koji se primjenjuju na ulazni niz podataka $[x_0\ x_1\ \ldots\ x_{n-1}]$ $(n \geq m)$ da generiše izlazni niz podataka $[y_{m-1}\ y_m\ \ldots\ y_{n-1}]$, pri čemu se svaki element izlaznog niza y_i računa prema formuli:

$$y_i = \sum_{j=0}^{m-1} c_j x_{i-j}$$

Obzirom da se isti filter u praksi uvijek primjenjuje na veći broj nizova ulaznih podataka, obično se radi jednostavnosti implementacije niz koeficijenata filtera obrne. Ovdje to neće biti rađeno da bi bile demonstrirane *SSE* instrukcije za preuređivanje podataka, ali će zadatak biti pojednostavljen utoliko što će se smatrati da procedura koja implementira filter umjesto adrese prvog elementa izlaznog niza y prima adresu $m-1$-og elementa toga niza[5]. U tom slučaju, ova procedura bi bila oblika:

```
.intel_syntax noprefix

.text
.global fir

        ## Funkcija fir() primjenjuje FIR filter sa datim koeficijentima
        ## na dati ulazni niz. Argumenti funkcije su:
        ##   [ebp+8] - broj koeficijenata
        ##   [ebp+12] - niz koeficijenata
        ##   [ebp+16] - broj elemenata ulaznog niza
        ##   [ebp+20] - ulazni niz
        ##   [ebp+24] - izlazni niz
        ## Pretpostavlja se da je broj koeficijenata veci od 0, kao i da
        ## je broj elemenata ulaznog niza veci ili jednak ovom broju.
        ## Funkcija ne vraca nikakvu vrijednost.
fir:
        ## Prolog funkcije.
        enter   0, 0
        push    ebx
        push    esi
        push    edi

        ## Provjerava se da li procesor podrzava SSE instrukcije.
        mov     eax, 1
        cpuid
        test    edx, 0x2000000
        jz done

        ## Cuva se sadrzaj SSE registara (i registara koprocesora).
        mov     edx, esp
        and     esp, 0xfffffff0
        sub     esp, 512
        fxsave  [esp]
```

[5] pretpostavlja se da se izvan ove procedure prvih $m-1$ elemenata izlaznog niza postavlja recimo na 0

```
        ## U registru ecx se izracunava broj elemenata izlaznog niza;
        ## ovaj registar ce biti koriscen kao brojac.
        mov     ecx, [ebp+16]
        sub     ecx, [ebp+8]
        inc     ecx

        ## Registar esi ce pokazivati na tekuci element ulaznog niza i
        ## incijalizuje se na adresu iza poslednjeg elementa ulaznog
        ## niza koji ce ucestvovati u izracunavanju prvog elementa
        ## izlaznog niza. Registar edi ce pokazivati na tekuci element
        ## izlaznog niza i postavlja se na adresu prvog elementa toga
        ## niza.
        mov     esi, [ebp+20]
        mov     eax, [ebp+8]
        shl     eax, 2
        add     esi, eax
        mov     edi, [ebp+24]

next_element:
        ## Registar ecx ce u unutrasnjoj petlji, u kojoj se racuna jedan
        ## element izlaznog niza, takodje sluziti kao brojac i
        ## inicijalizuje se na broj koeficijenata filtera.  Registar ebx
        ## ce pokazivati na tekuci koeficijent filtera i inicijalizuje
        ## se na adresu prvog koeficijenta filtera.
        push    ecx
        mov     ecx, [ebp+8]
        mov     ebx, [ebp+12]

        ## Svaki element izlaznog niza jednak je sumi proizvoda
        ## koeficijenata filtera i odgovarajucih elemenata ulaznog
        ## niza. Ovi proizvodi ce biti racunati po 4 paralelno, a
        ## parcijalne sume ce biti cuvane u registru xmm0.
        xorps   xmm0, xmm0

next_four:
        ## Provjerava se da li ima jos 4 proizvoda koeficijenata filtera
        ## i elememata ulaznog niza za paralelno izracunavanje.  Ako to
        ## nije slucaj, prelazi se na pojedinacno izracunavanje proizvoda.
        cmp     ecx, 4
        jl next_one

        ## Ucitavaju se 4 naredna koeficijenta filtera u registar xmm1 i
        ## 4 naredna elementa ulaznog niza u registar xmm2.  Registar
        ## xmm2 se potom preuredjuje da bi elementi ulaznog niza dosli
        ## na odgovarajuca mjesta u pogledu izracunavanja proizvoda.
        movups  xmm1, [ebx]
        add     ebx, 16
        sub     esi, 16
        movups  xmm2, [esi]
        shufps  xmm2, xmm2, 0x1b

        ## Izracunavaju se 4 proizvoda i dodaju parcijalnim sumama u
        ## registru xmm0.
        mulps   xmm1, xmm2
        addps   xmm0, xmm1

        ## Prelazi se na narednu iteraciju.
        sub     ecx, 4
        jmp next_four

next_one:
        ## Provjerava se da li ima jos proizvoda koeficijenata filtera
```

```
        ## i elememata ulaznog niza za izracunavanje.  Ako to nije
        ## slucaj, prelazi se na sumiranje parcijalnih suma.
        jecxz calc_sum

        ## Ucitava se naredni koeficijent filtera u registar xmm1 i
        ## mnozi sa odgovarajucim elementom ulaznog niza, a proizvod se
        ## potom dodaje jednoj parcijalnoj sumi u registru xmm0.
        movss   xmm1, [ebx]
        add     ebx, 4
        sub     esi, 4
        mulss   xmm1, [esi]
        addss   xmm0, xmm1

        ## Prelazi se na narednu iteraciju.
        dec     ecx
        jmp next_one

calc_sum:
        ## Na osnovu 4 parcijalne sume iz registra xmm1, izracunava se
        ## finalna suma, koja predstavlja tekuci element izlaznog niza i
        ## ta vrijednost se smjesta na odgovarajucu adresu.
        movhlps xmm1, xmm0
        addps   xmm0, xmm1
        movaps  xmm1, xmm0
        shufps  xmm1, xmm1, 0x55
        addss   xmm0, xmm1
        movss   [edi], xmm0

        ## Azurira se registar esi tako da opet pokazuje iza poslednjeg
        ## elementa ulaznog niza koji ce ucestvovati u izracunavanju
        ## narednog elementa izlaznog niza.
        mov     eax, [ebp+8]
        inc     eax
        shl     eax, 2
        add     esi, eax

        ## Prelazi se na izracunavanje narednog elementa izlaznog niza.
        pop     ecx
        add     edi, 4
        loop next_element

        ## Vraca se prethodni sadrzaj u registre koprocesora i SSE
        ## registre.
        fxrstor [esp]
        mov     esp, edx

done:
        ## Epilog funkcije.
        pop     edi
        pop     esi
        pop     ebx
        leave
        ret
```

Na početku procedure se provjerava da li su *SSE* ekstenzije podržane, a zatim se instrukcijom `FXSAVE` odlaže sadržaj registara numeričkog koprocesora i *SSE* registara u prethodno odvojen prostor na steku. Pritom, instrukcija `FXSAVE` zahtijeva da je adresa na koju se vrši odlaganje djeljiva sa 16; način na koji je ovo implementirano u gornjoj proceduri podrazumijeva da se sadržaj

registra `ESP` od prije ove alokacije prostora na steku sačuva u registru `EDX` i onda vrati iz ovog registra u `ESP` registar pred kraj procedure.

Potom se u petlji računa jedan po jedan element izlaznog niza prema gornjoj formuli. Dok god ima više od 4 preostala elementa u sumi, po 4 elementa sume se računaju istovremeno tako što se u *SSE* registre dovedu odgovarajući koeficijenti odn. elementi ulaznog niza (važno je uočiti da se elementi ulaznog niza nakon učitavanja iz memorije moraju preurediti), a zatim paralelno pomnože instrukcijom `MULPS` i onda paralelno dodaju instrukcijom `ADDPS` odgovarajućim parcijalnim sumama koje se takođe akumuliraju u jednom *SSE* registru. Kada ostane manje od 4 elementa da se uračuna u sumu, proizvodi se računaju pojedinačno i dodaju parcijalnoj sumi koja se nalazi u 32 bita najmanje težine registra koji sadrži parcijalne sume, pri čemu se sad koriste instrukcije `MULSS` odn. `ADDSS`. Na kraju se 4 parcijalne sume saberu i rezultat odloži na odgovarajuću lokaciju izlaznog niza.

Prikazani primjer dobro demonstrira tipičnu sekvencu koraka prilikom rješavanja nekog problema korišćenjem *SSE* ekstenzija. Naime, obzirom da dimenzija problema obično nije umnožak broja 4, glavnina problema se prvo rješava paralelno, koristeći *SSE* instrukcije sa sufiksom *PS*, da bi se potom rješavanje problema kompletiralo korišćenjem istog slijeda operacija, ali sada sa *SSE* instrukcijama sa `SS` sufiksom.

Poglavlje 13

SSE2 instrukcije

Kako je već pomenuto, uvođenje *Pentium IV* procesora značilo je i pojavu novog proširenja *IA-32* arhitekture u vidu *SSE2* skupa instrukcija. *SSE2* ekstenzije, mada obimne, ipak ne donose tako radikalne izmjene kao *MMX* ili *SSE* ekstenzije. Ove ekstenzije naime ne podrazumijevaju uvođenje novih registara, već instrukcije koje su njima dodate koriste postojeće *SSE* i *MMX* registre. Takođe, nove instrukcije uglavnom predstavljaju proširenja postojećih *SSE* instrukcija da podrže paralelne operacije nad realnim brojevima u dvostrukoj preciznosti odn. *MMX* instrukcija da podrže operande u *SSE* registrima.

Provjera da li su *SSE2* ekstenzije podržane se vrši na isti način kao za *MMX* ili *SSE* ekstenzije - potrebno je izdati `CPUID` instrukcija sa vrijednošću 1 u *EAX* registru i ukoliko je nakon toga bit 26 registra *EDX* jednak 1, znači da su *SSE2* ekstenzije podržane na datom procesoru.

13.1 Instrukcije sa rad sa realnim brojevima

Jedan od novih tipova podataka koga uvode *SSE2* ekstenzije su pakovani realni brojevi u dvostrukoj preciznosti. *SSE* registri mogu sadržati po 2 ovakva broja i najvažniji podskup *SSE2* instrukcija se odnosi na paralelne i pojedinačne operacije nad ovakvim podacima. Ove instrukcije su potpuno ekvivalentne odgovarajućim *SSE* instrukcijama i mogu se podijeliti na nekoliko grupa.

SSE2 instrukcije za transfer podataka su prikazane u tabeli 13.1. Ove instrukcije se odnose na transfer 64-bitnih realnih brojeva između *SSE* registara odn. između *SSE* registra i memorije i potpuno su ekvivalente odgovarajućim *SSE* instrukcijama predstavljenim u tabeli 12.1. Obzirom da *SSE2* ekstenzije ne uvode nove registre, instrukcije `STMXCSR` i `LDMXCSR` odn. `FXSAVE`

i `FXRSTOR` se koriste na uobičajen način za odlaganje sadržaja statusno-kontrolnog odn. svih *SSE* registara u memoriju, te za učitavanje sadržaja tih registara iz memorije.

Mnemonik	Operacija
`MOVAPD`	prebacivanje 2 uravnata podatka (64-bitna realna broja)
`MOVUPD`	prebacivanje 2 neuravnata podatka
`MOVSD`	prebacivanje jednog podatka
`MOVLPD`	prebacivanje jednog podatka između memorije i nižeg dijela registra
`MOVHPD`	prebacivanje jednog podatka između memorije i višeg dijela registra
`MOVMSKPD`	upisivanje bitova znaka oba podatka u registar opšte namjene

Tabela 13.1: *SSE2* instrukcije za transfer podataka.

SSE2 aritmetičko-logičke instrukcije su prikazane u tabeli 13.2. Ove instrukcije su ekvivalentne odgovarajućim *SSE* instrukcijama prikazanim u tabeli 12.2. Treba međutim primijetiti da instrukcije `RCPPS`, `RCPSS`, `RSQRTPS` i `RSQRTSS` nemaju ekvivalente u *SSE2* skupu instrukcija. Sve navedene instrukcije inače primaju dva operanda, od kojih jedan eventualno može biti memorijski. Kao i *SSE* instrukcije, tako i paralelne aritmetičko-logičke (i sve ostale slične) *SSE2* instrukcije zahtijevaju da memorijski operand bude na adresi djeljivoj sa 16. Takođe, interakcija *SSE2* instrukcija sa statusno-kontrolnim registrom je potpuno ista kao kod odgovarajućih *SSE* instrukcija.

Mnemonik	Operacija
`ADDPD/ADDSD`	sabiranje operanada
`SUBPD/SUBSD`	oduzimanje operanada
`MULPD/MULSD`	množenje operanada
`DIVPD/DIVSD`	dijeljenje operanada
`SQRTPD/SQRTSD`	izračunavanje kvadratnog korijena operanada
`MAXPD/MAXSD`	izračunavanje većeg od operanada
`MINPD/MINSD`	izračunavanje manjeg od operanada
`ANDPD`	bitska konjukcija operanada
`ANDNPD`	negacija bitske konjukcije operanada
`ORPD`	bitska disjunkcija operanada
`XORPD`	bitska operacija *XOR* nad operandima

Tabela 13.2: *SSE2* aritmetičko-logičke instrukcije.

SSE2 instrukcije poređenja su date u tabeli 13.3. Ove instrukcije su potpuno ekvivalentne odgovarajućim *SSE* instrukcijama datim u tabeli 12.3.

SSE2 ektenzije donose veliki broj instrukcija za konverziju podataka, koje su nabrojane u tabeli 13.4.

Mnemonik	Operacija
CMPPD	istovremeno poređenje 2 para podataka
CMPSD	poređenje jednog para podataka
COMISD	poređenje jednog para podataka i postavljanje flegova EFLAGS registra
UCOMISD	isto kao prethodno uz "neuređeno" poređenje

Tabela 13.3: *SSE2* instrukcije poređenja.

Instrukcije SHUFPD, UNPCKHPD i UNPCKLPD su ekvivalentne *SSE* instrukcijama SHUFPS, UNPCKHPS odn. UNPCKLPS. Instrukcija SHUFPD prima tri operanda, od kojih je treći vrijednost čiji bit najmanje težine određuje da li se gornja ili donja polovina odredišnog registra prebacuje u njegovu donju polovinu, dok naredni bit određuje da li se donja ili gornja polovina drugog operanda prebacuje u gornju polovinu odredišnog registra. Kod instrukcija UNPCKHPD i UNPCKLPD se viša odn. niža polovina odredišnog registra prebacuje u njegovu nižu polovinu, dok se u viši dio odredišnog registra stavlja viša odn. niža polovina drugog operanda.

Slično, instrukcije CVTPI2PD, CVTSI2SD, CVTPD2PI i CVTSD2SI su potpuno ekvivalentne CVTPI2PS, CVTSI2SS, CVTPS2PI odn. CVTSS2SI za konverziju podataka između *SSE* registara i *MMX* registara odn. između *SSE* registara i registara opšte namjene. Takođe, dodate su i instrukcije CVTTPS2PI, CVTTSS2SI, CVTTPD2PI i CVTTSD2SI koje vrše iste konverzije kao i odgovarajuće instrukcije koje imaju samo jedno slovo T u prefiksu, s tim što je kod ovih instrukcija zaokruživanje uvijek prema nuli, bez obzira na režim zaokruživanja određen statusno-kontrolnim registrom.

SSE2 ekstenzije takođe sadrže instrukcije za konverziju između 32-bitnih i 64-bitnih realnih brojeva. Instrukcija CVTPS2PD konvertuje 2 32-bitna realna broja iz datog *SSE* registra ili memorije u 2 64-bitna realna broja u odredišnom *SSE* registru. Slično, instrukcija CVTSS2SD konvertuje 32-bitni realan broj iz datog *SSE* registra ili memorije u 64-bitni realan broj u odredišnom *SSE* registru. Obrnute konverzije, iz 64-bitnih u 32-bitne realne podatke vrše instrukcije CVTPD2PS odn. CVTSD2SS.

Na kraju, *SSE2* ekstenzije predviđaju da se cjelobrojni podaci mogu držati u *SSE* registrima, pa tako postoje i instrukcije za konverziju između realnih i takvih cjelobrojnih podataka. Instrukcija CVTPS2DQ vrši konverziju 4 32-bitna realna broja koji se nalaze u *SSE* registru ili memoriji u 4 32-bitna cijela broja koje pakuje u odredišni *SSE* registar. Obrnutu konverziju radi instrukcija CVTDQ2PS, dok instrukcija CVTTPS2DQ radi isto što i CVTTPS2DQ, ali uz fiksno zaokruživanje prema nuli. Instrukcije CVTPD2DQ, CVTDQ2PD i CVTTPD2DQ su respektivno ekvivalentne instrukcijama CVTPS2DQ, CVTDQ2PS i CVTTPS2DQ, s tim što se konverzija odnosi na 64-bitne umjesto 32-bitnih realnih podataka i što su u konverziju uključena po 2 umjesto po 4 podatka.

Mnemonik	Operacija
`SHUFPD`	kopiranje po jednog podatka iz svakog operanda u niža odn. viša 64 bita odredišnog operanda
`UNPCKHPD`	naizmjenično kopiranje podataka iz viša 64 bita operanada u odredišni operand
`UNPCKLPD`	naizmjenično kopiranje podataka iz niža 64 bita operanada u odredišni operand
`CVTPI2PD`	konverzija 2 cjelobrojna podatka iz *MMX* registra ili memorije u 2 64-bitna realna podatka u *SSE* registru
`CVTSI2SD`	konverzija jednog cjelobrojnog podatka iz registra opšte namjene ili memorije u 64-bitni realan podatak u *SSE* registru
`CVTPD2PI`	konverzija 2 64-bitna realna podatka iz *SSE* registra ili memorije u 2 cjelobrojna podatka u *MMX* registru
`CVTSD2SI`	konverzija jednog 64-bitnog realnog podatka iz *SSE* registra ili memorije u cjelobrojni podatak u registru opšte namjene
`CVTTPS2PI`	konverzija 2 32-bitna realna podatka iz *SSE* registra ili memorije u 2 cjelobrojna podatka u *MMX* registru uz zaokruživanje prema nuli
`CVTTSS2SI`	konverzija jednog 32-bitnog realnog podatka iz *SSE* registra ili memorije u cjelobrojni podatak u registru opšte namjene uz zaokruživanje prema nuli
`CVTTPD2PI`	konverzija 2 64-bitna realna podatka iz *SSE* registra ili memorije u 2 cjelobrojna podatka u *MMX* registru uz zaokruživanje prema nuli
`CVTTSD2SI`	konverzija jednog 64-bitnog realnog podatka iz *SSE* registra ili memorije u cjelobrojni podatak u registru opšte namjene uz zaokruživanje prema nuli
`CVTPS2PD`	konverzija 2 32-bitna realna podatka iz *SSE* registra ili memorije u 2 64-bitna realna podatka u *SSE* registru
`CVTSS2SD`	konverzija 32-bitnog realnog podatka iz *SSE* registra ili memorije u 64-bitni realni podatak u *SSE* registru
`CVTPD2PS`	konverzija 2 64-bitna realna podatka iz *SSE* registra ili memorije u 2 32-bitna realna podatka u *SSE* registru
`CVTSD2SS`	konverzija 64-bitnog realnog podatka iz *SSE* registra ili memorije u 32-bitni realni podatak u *SSE* registru
`CVTPS2DQ`	konverzija 4 32-bitna realna podatka iz *SSE* registra ili memorije u 4 32-bitna cjelobrojna podatka u *SSE* registru
`CVTTPS2DQ`	isto kao prethodno, uz zaokruživanje prema nuli
`CVTDQ2PS`	konverzija 4 32-bitna cjelobrojna podatka iz *SSE* registra ili memorije u 4 32-bitna realna podatka u *SSE* registru
`CVTPD2DQ`	konverzija 2 64-bitna realna podatka iz *SSE* registra ili memorije u 2 32-bitna cjelobrojna podatka u *SSE* registru
`CVTTPD2DQ`	isto kao prethodno, uz zaokruživanje prema nuli
`CVTDQ2PD`	konverzija 2 32-bitna cjelobrojna podatka iz *SSE* registra ili memorije u 2 64-bitna realna podatka u *SSE* registru

Tabela 13.4: *SSE2* instrukcije za preuređivanje i konverziju podataka.

13.2 Instrukcije sa rad sa cijelim brojevima

Osnovna novina koju *SSE2* ekstenzije donose u pogledu rada sa cijelim brojevima jeste što su sve *MMX* instrukcije (kao i odgovarajuće ekstenzije uvedene u sklopu *SSE* instrukcija) proširene tako da mogu podjednako da rade sa podacima u *SSE* registrima kao i nad podacima u *MMX* registrima. Tako su npr. instrukcije `PADDB`, `PADDW` i `PADDD` proširene da mogu, isto kao što mogu da paraleno sabiraju 8 parova 8-bitnih ili 4 para 16-bitnih ili 2 para 32-bitnih cjelobrojnih podataka iz *MMX* registara (odn. *MMX* registra i memorije), da sabiraju 16 parova 8-bitnih ili 8 parova 16-bitnih ili 4 para 32-bitnih cjelobrojnih podataka iz *SSE* registara (odn. *SSE* registra i memorije). Ovo je dosljedno izvedeno za sve *MMX* instrukcije, čime je postignuto da ove instrukcije mogu da operišu nad dvostruko više podataka ako se isti nalaze u *SSE* registrima, a takođe se ovim dodatno rješava problem preklapanja *MMX* registara i registara koprocesora kao jedan od osnovnih nedostatka dizajna *MMX* ekstenzija.

Pored ovoga, *SSE2* ekstenzije uvode i neke nove *MMX* instrukcije. Ove instrukcije su nabrojane u tabeli 13.5.

Mnemonik	Operacija
`MOVDQA`	transfer 2 cjelobrojna 64-bitna podatka između *SSE* registara odn. između *SSE* registra i memorije
`MOVDQA`	isto kao prethodno, bez zahtjeva za uravnanjem memorijske adrese
`PADDQ`	sabiranje pakovanih 64-bitnih cjelobrojnih podataka
`PSUBQ`	oduzimanje pakovanih 64-bitnih cjelobrojnih podataka
`PMULUDQ`	množenje pakovanih 32-bitnih i 64-bitnih neoznačenih cjelobrojnih podataka
`PSHUFLW`	preuređivanje 4 16-bitna podatka iz niže polovine datog *SSE* registra
`PSHUFHW`	preuređivanje 4 16-bitna podatka iz više polovine datog *SSE* registra
`PSHUFHD`	preuređivanje 4 32-bitna podatka iz datog *SSE* registra
`PSLLDQ`	šifovanje ulijevo za dati broj bajtova
`PSRLDQ`	šifovanje udesno za dati broj bajtova
`PUNPCKHQDQ`	izdvajanje viših polovina operanada u odredišni operand
`PUNPCKLQDQ`	izdvajanje nižih polovina operanada u odredišni operand
`MOVQ2DQ`	prebacivanje 64-bitnog cjelobrojnog podatka iz *MMX* registra u nižu polovinu *SSE* registra
`MOVDQ2Q`	prebacivanje 64-bitnog cjelobrojnog podatka iz niže polovine *SSE* registra u *MMX* registar

Tabela 13.5: *MMX* instrukcije u *SSE2* ekstenzijama.

13.3 Ostale *SSE2* instrukcije

Kao i *SSE* ekstenzije, tako i *SSE2* ekstenzije uključuju i određen broj instrukcija za kontrolu keširanja podataka. Ove instrukcije su nabrojane u tabeli 13.6.

Mnemonik	Operacija
`CLFLUSH`	izbacivanje bloka podataka iz svih nivoa *cache*-a
`MOVNTDQ`	smještanje sadržaja *SSE* registra kao 64-bitnih cjelobronojih podataka u memoriju mimoilaženjem *cache*-eva
`MOVNTDQ`	smještanje sadržaja *SSE* registra kao 64-bitnih realnih podataka u memoriju mimoilaženjem *cache*-eva
`MOVNTI`	smještanje sadržaja registra opšte namjene u memoriju mimoilaženjem *cache*-eva
`MASKMOVDQU`	smještanje dijela sadržaja *SSE* registra u memoriju mimoilaženjem *cache*-eva
`LFENCE`	serijalizacija prethodnih učitavanja iz memorije
`MFENCE`	serijalizacija svih prethodnih memorijskih transfera

Tabela 13.6: *SSE2* instrukcije za kontrolu keširanja podataka.

13.4 Upotreba *SSE2* instrukcija

Obzirom da je asemblersko programiranje korišćenjem *SSE2* instrukcija potpuno ekvivalentno programiranju korišćenjem *SSE* instrukcija, neće biti davan poseban primjer programa koji koristi ove instrukcije. Kao zaključak, može se reći da skup *SSE2* instrukcija dobro zaokružuje *SSE* instrukcije u smislu podrške za rad sa realnim brojevima dvostruke preciznosti, tako da su ova dva skupa instrukcija veoma upotrebljiva za asemblersko programiranje. Sa druge strane, mada mogućnost korišćenja *MMX* operacija nad cjelobrojnim podacima u *SSE* registrima koju donose *SSE2* ekstenzije predstavlja značajno unapređenje osnovnog skupa *MMX* instrukcija, ovaj skup instrukcija ostaje vrlo neregularan i težak za asemblersko programiranje.

Dodatak A

IA-64 arhitektura

Osnovna karakteristika skupa arhitektura kojima pripada *IA-32* arhitektura jeste kompleksan (u slučaju *IA-32* arhitekture i veoma neregularan) skup instrukcija zbog čega se ovakve arhitekture označavaju terminom *CISC* (*complex instruction set computers*). Ovakve arhitekture su uglavnom razvijane tokom 70.-tih godina (tipičan predstavnik arhitektura koje su prethodile *IA-32* arhitekturi je *DEC*-ova *VAX* arhitektura); ideja uvođenja velikog broja kompleksnih instrukcija bila je da se olakša posao prevodiocima. Vrijeme je međutim pokazalo da prevodioci nisu u stanju da efikasno iskoriste ni dio ovog kompleksnog skupa instrukcija, dok je opet taj kompleksan skup instrukcija pravio brojne prepreke unapređenjima takvih arhitektura. Zato se od sredine 80.-tih godina intenzivira razvoj jednog novog skupa arhitektura gdje je osnovna ideja bila da se napravi mali i dosljedan skup instrukcija koje će biti lako implementirane direktno u hardveru (bez mikrokodiranja), pa na taj način i veoma brze. Ovakve arhitekture se označavaju terminom *RISC* arhitekture (od *reduced instruction set computers*)[1]. *RISC* arhitekture karakteriše fiksna dužina instrukcija (radi lakšeg dekodovanja), minimizovan broj instrukcija sa bočnim efektima, namjenske instrukcije za rad sa memorijom i prihvatanje samo registara ili konstanti kao operanada (zbog ovoga se ove arhitekture označavaju i terminom *load-store* arhitekture). Ove osobine su omogućile implementaciju dubokih *pipeline*-ova, što je u kombinaciji sa više izvršnih jedinica (tzv. superskalarno izvršavanje) otvorilo realne mogućnosti izvršavanja većeg broja instrukcija u jednom ciklusu takta u dužim sekvencama, a što je bilo nedostižno za *CISC* arhitekture[2].

IA-32 arhitektura je ipak uspjela da odoli izazovu *RISC* arhitektura prvenstveno zbog kontrole dominantnog segmenta tržišta, a zatim i zbog stalnih

[1] tipični predstavnici ovog skupa arhitektura su *MIPS*, te *Sun*-ova *SPARC*, *HP*-ova *PA-RISC* i *DEC*-ova *Alpha* arhitektura

[2] i dan danas tekući procesori iz ovih arhitektura imaju znatno niže cikluse takta od odgovarajućih procesora *IA-32* arhitekture, ali zbog navedenih osobina rade brže od njih

napora da se dostignuća *RISC* arhitektura, koliko je to bilo moguće, primijene i na *IA-32* arhitekturu. Zato poslednje generacije procesora *IA-32* arhitekture prestavljaju zapravo smješu *RISC* i *CISC* elemenata; međutim zbog inherentne kompleksnosti *IA-32* arhitekture mogućnosti njenog daljeg unapređenja su praktično dovedene do kranjih granica.

Uočavajući da *IA-32* arhitektura prelazi svoj zenit, *Intel* se okreće traženju arhitekture koja bi je naslijedila. Obzirom da se u međuvremenu polako počinje sa prelazom na 64-bitne arhitekture, logičan izbor je bio da se osmisli takva arhitektura. *Intel* međutim nema resursa da razvije arhitekturu od nule, tako da nalazi partnera u *HP*-u i oni zajedno razvijaju arhitekturu koja je 64-bitni nasljednik *HP*-ove *PA-RISC* arhitekture; ova arhitektura dobija ime *IA-64*. Prvi procesori predstavnici ove arhitekture bili su *Itanium* i *Itanium2*; u narednim godinama se očekuje izbacivanje na tržište niza srodnih procesora koji bi trebalo da potpuno istisnu *Pentium* procesore.

IA-64 arhitektura predstavlja modernu *RISC* arhitekturu, ali sa nizom inovativnih osobina zbog kojih zavređuje da se svrsta u posebnu novu kategoriju arhitektura koju *Intel* označava terminom *EPIC* (*explicitly parallel instruction set computing*). Naime, ključna karakteristika ove arhitekture jeste da se kroz tzv. grupe instrukcija eksplicitno istovremeno aktiviraju više procesnih jedinica na koje se pojedine instrukcije iz grupe šalju paralelno na izvršavanje. Ovaj koncept vuče porijeklo iz grupe tzv. *VLIW* (*very large instruction word*) arhitektura koje su u određenoj mjeri uporedo razvijane sa *CISC* i *RISC* arhitekturama, ali nikad nisu komercijalno zaživjele. Na ovaj način, kompleksni resursi koji su vršili preraspoređivanje instrukcija po procesnim jedinicama se eliminišu iz procesora oslobađajući time prostor za druga korisna unapređenja, dok se taj posao raspoređivanja instrukcija na izvršavanje prebacuje na prevodilac odn. na asemblerskog programera. Procesori iz *IA-64* arhitekture mogu odn. moći će da istovremeno paralelno izvršavaju po tri instrukcije iz grupe, a takođe zbog superskalarnog dizajna i da šalju na izvršavanje najmanje po dvije grupe instrukcija u jednom ciklusu takta tako da su vršne performanse arhitekture minimalno 6 instrukcija po ciklusu takta.

Procesori *IA-64* arhitekture imaju i režim kompatibilnosti sa *IA-32* procesorima[3]. U tom režimu, na ovim procesorima se može izvršavati sav postojeći *IA-32* kod; postojanje ovog režima je bio obavezni preduslov da bi se mogla očekivati migracija na *IA-64* platforme. Režim kompatibilnosti je uglavnom implementiran potpuno odvojeno od ostatka procesora tako da će, jednom kada se završi sa migracijom na novu arhitekturu, odgovarajući resursi biti prosto izbačeni.

[3]takođe i još jedan režim kompatibilnosti sa *PA-RISC* procesorima

A.1 Skup registara arhitekture

IA-64 arhitektura ima masivne registarske resurse što omogućava da se većina izračunavanja obavlja u registrima i time minimizuje broj pristupa memoriji, čime se znatno unapređuju performanse. Postoji 128 cjelobrojnih registara (oznake u asembleru su `r0-r127`) koji su 64-bitni; nekoliko od ovih registara ima posebnu namjenu, ali generalno oni se mogu tretirati kao registri opšte namjene. Ovi registri su podijeljeni u dvije grupe, pri čemu prva 32 registra predstavljaju tzv. statičke registre dok se preostalih 96 registara koriste u vidu posebne vrste steka koja znatno olakšava i ubrzava prenošenje argumenata pri pozivanju procedura, a takođe mogu da funkcionišu i u tzv. režimu rotacije koji služi za ubrzavanje izvršavanja koda u kratkim petljama. Postoji dalje 128 registara za rad u pokretnom zarezu (oznake `f0-f127`) koji su opet podijeljeni tako da su prva 32 registra statički, dok su preostalih 96 registara rotirajući. Postoji zatim skup od 64 predikatska registra (oznake `p0-p63`) koji su 1-bitski i koji generalizuju koncept flegova iz ranijih arhitektura, tj. na osnovu njihovog sadržaja se može određivati hoće li se ili neće izršavati neka instrukcija. Postoji onda skup od 8 registara skoka (oznake `b0-b7`) koji drže adrese skoka pri indirektnim skokovima. Postoji na kraju skup od 128 registara koji se označavaju imenom aplikacioni registri i od kojih svaki ima posebnu namjenu (i posebnu oznaku); tako npr. ovdje imamo registar koji služi kao brojač u petljama (`ar.lc`) ili registar koji čuva stanje steka registara opšte namjene za prethodnu funkciju (`ar.pfs`). Pored ovih registara postoje i neke specijalne grupe registara kao npr. registri koji čuvaju informacije o procesoru, registri koji služe za mjerenje performansi i naravno sistemski registri tj. registri kojima se može pristupiti samo privilegovanim instrukcijama.

A.2 Format i način izvršavanja instrukcija

Format instrukcija kod *IA-64* arhitekture je sličan formatu instrukcija kod *RISC* arhitektura u smislu da je binarni kod instrukcije fiksne dužine. Svaka instrukcija je naime dužine 41 bit, a tri instrukcije čine grupu instrukcija (tzv. *bundle*) koja je dužine 128 bita pri čemu zadnjih 5 bita kodiraju na kojim se jedinicama mogu izvršavati pojedine instrukcije kao i da li se i koliko njih mogu paralelno izvršavati. Pri programiranju na asembleru, programer ima dva izbora u pogledu specifikacije sekvenciranja instrukcija: programer može da eksplicitno specificira svaku grupu instrukcija sa ovim dodatnim podacima[4] ili programer može samo da naznači koje se instrukcije mogu

[4] asembler naravno mora pritom da podržava odgovarajuće direktive, kao što je to slučaj sa *GNU* asemblerom

paralelno izvršavati[5], a da ostavi asembleru da ih grupiše na odgovarajući način. Ovaj drugi način je svakako dosta fleksibilniji i očekuje se da će ga programeri češće koristiti. Što se tiče pravila na osnovu kojih se određuje koje se instrukcije mogu naći zajedno u sekvenci, jedino ograničenje jeste da ne smije postojati instrukcija koja čita ili piše u neki registar u koji je neka prethodna instrukcija iz iste sekvence već pisala[6].

A.3 Skup instrukcija

Format pojedinačnih instrukcija *IA-64* arhitekture ima oblik:

```
(qp) mnemonic dst = src0, src1
```

Ovdje je `qp` predikatski registar čiji sadržaj određuje da li će se instrukcija izvršavati ili neće; dio `(qp)` na početku instrukcije nije obavezan i ako ga nema onda se instrukcija prosto uvijek izvršava. Dalje, `mnemonic` je naravno skraćenica koja predstavlja ime instrukcije. Ono što je karakteristično za *IA-64* arhitekturu je da mnemonici mogu imati veliki broj sufiksa kojima se ili precizira vrsta instrukcije ili daju procesoru razne naznake u pogledu njenog izvršavanja; ovakvi sufiksi se u *IA-64* terminologiji nazivaju *completers* i svaki od njih se odvaja tačkom od prethodnog dijela mnemonika. Odredišna adresa je prestavljena sa `dst` dok su `src0` i `src1` naravno adrese operanada; znak jednakosti je obavezan. *IA-64* je tipična *load-store* arhitektura (što je opet jedna od glavnih karakteristika *RISC* arhitektura), odnosno operandi najvećeg broja instrukcija mogu biti samo ili registri ili konstante, dok postoji posebna mala grupa instrukcija koja služi za razmjenu podataka između registara i memorije.

Skup instrukcija *IA-64* arhitekture je blizak uobičajenom skupu instrukcija koji se može naći na *RISC* arhitekturama. Tako se mogu nabrojati sledeće osnovne grupe instrukcija:

- aritmetičke instrukcije (`add`[7], `sub`, `xmpy`)
- logičke instrukcije (`and`, `or`, `xor`, `andcm`)
- instrukcije pomjeranja (`shl`, `shr`, `shr.u`)

---

[5] ovo se u *GNU* asembleru radi prosto tako što se stavi odgovarajuća oznaka u vidu dvostruke tačke-zareza na kraj sekvence instrukcija koje se mogu paralelno izvršavati; takođe neke instrukcije, kao npr. instrukcije skoka, impliciraju kraj sekvence

[6] ove nedozvoljene zavisnosti se označavaju terminima *RAW* (*read after write*) odn. *WAW* (*write after write*)

[7] u skladu sa odgovarajućom *Intel*-ovom referentnom dokumentacijom, imena registara i instrukcija *IA-64* arhitekture će, za razliku od *IA-32* arhitekture, biti navođena malim slovima

- instrukcije poređenja (`cmp`)
- instrukcije za pristup memoriji (`ld`, `st`, `xchg`)
- instrukcije za transfer podataka među registrima (`mov`)
- instrukcije skoka (`br`)
- SIMD instrukcije (`padd`, `psub` itd.)
- instrukcije za rad sa realnim brojevima

Skup instrukcija *IA-64* arhitekture takođe karakterišu i neke druge osobine koje su tipične za *RISC* arhitekture, kao npr. da nema instrukcije dijeljenja već da se dijeljenje mora vršiti softverski. Postoje i neke specifičnosti karakteristične samo za ovu arhitekturu kao npr. da se i cjelobrojno množenje obavlja u jedinici za rad u pokretnom zarezu.

A.4 Elementi arhitekture dizajnirani u cilju unapređenja performansi

Pored paralelizma na nivou instrukcija koji predstavlja glavnu inovaciju *IA-64* arhitekture u smislu unapređenja performansi, postoji i niz drugih elemenata arhitekture koji su namjenski dizajnirani upravo sa istim ciljem. Ovdje će biti pojašnjen mehanizam pozivanja procedura, a svakako među takvim naprednim osobinama treba pomenuti i podršku za spekulaciju po kontroli odn. podacima kao i mehanizam rotacije registara koji omogućava paralelizaciju petlji.

Kako je već pomenuto, registri opšte namjene `r32-r127` funkcionišu na principu steka. Svaka procedura pristupa ovim registrima koristeći logička imena `in0`, `in1` itd. odn. `loc0`, `loc1` itd. ili `out0`, `out1` itd. Pritom, registri sa prefiksom `in` predstavljaju argumente procedure, registri sa prefiksom `loc` predstavljaju lokalne promjenljive, a registri sa prefiksom `out` predstavljaju argumente procedura koje data procedura poziva. Na početku svake procedure treba direktivom `alloc` specificirati koliko data procedura ima argumenata, odn. koliko joj je potrebno registara za lokalne promjenljive i argumente procedura koje će ona pozivati. Procesor potom dodjeljuje gorepomenuta logička imena trenutno raspoloživim sukcesivnim registrima iz grupe registara `r32-r127`. Pritom, logička imena za `in` registre odgovaraju fizičkim registrima koji su bili imenovani kao `out` registri u kodu koji je pozvao datu proceduru i na taj način je omogućen veoma jednostavan i efikasan mehanizam prenošenja argumenata kroz registre. Naravno, ako postoji veliki broj ugnježdenih poziva procedura, 96 registara neće biti dovoljno; u tom slučaju procesor automatski prebacuje sadržaj registara koji odgovaraju polaznim

procedurama u memoriju oslobađajući tako prostor za registre ugnježdenih procedura. Obzirom da se registri koje je koristila data procedura oslobađaju čim se procedura završi, može se reći da čitav mehanizam funkcioniše kao neki stek, ali čiji je vrh uvijek u registrima tako da je rad sa njime veoma efikasan.

A.5 Primjer programiranja u *IA-64* asembleru

Neke karakteristike *IA-64* arhitekture su demonstrirane narednim primjerom:

```
.text

        ## Funkcija lsearch() trazi datu vrijednost u datom
        ## polju. Argumenti funkcije su:
        ##   in0 - trazena vrijednost
        ##   in1 - duzina polja
        ##   in2 - polje koje se pretrazuje
        ## Funkcija vraca indeks prvog elementa u polju koji je jednak
        ## datoj vrijednosti, odn. -1 ako takav ne postoji. Funkcija u
        ## potpunosti slijedi IA-64 konvencije u pogledu upotrebe
        ## registara.
.global lsearch
lsearch:
        # Alocira se prostor na registarskom steku za funkciju i cuva
        # se sadrzaj registra brojaca u petljama.
        alloc           r14 = ar.pfs, 3, 0, 0, 0
        mov             r15 = ar.lc

        # Provjerava se da li je duzina polja veca od 0 i ako nije
        # odmah se zavrsava funkcija.
        cmp.le.unc      p7, p0 = in1, r0
        (p7) br.cond.dpnt.few not_found
        ;;

        # Umanjuje se za 1 broj koji predstavlja duzinu niza.
        add             in1 = -1, in1
        ;;

        # Inicijalizuje se rezultat na -1 i brojac u petlji na duzinu
        # niza umanjenu za 1.
        mov             ret0 = -1
        mov             ar.lc = in1
        ;;

next_element:
        # Ucitava se naredni element niza i inkrementira indeks
        # tekuceg elementa.
        ld4             r16 = [in2], 4
        add             ret0 = 1, ret0
        ;;

        # Poredi se tekuci element niza sa trazenom vrijednoscu i
        # ukoliko su jednaki, izlazi se iz petlje. U suprotnom se
        # nastavlja sa narednom iteracijom.
        cmp4.eq.unc     p7, p0 = r16, in0
```

```
        (p7) br.cond.dptk.few found
        br.cloop.dptk.few next_element
        ;;

not_found:
        # Rezultat se postavlja na -1 kao indikator da trazena
        # vrijednost nije pronadjena u nizu.
        mov             ret0 = -1
        ;;

found:
        # Restaurise se sadrzaj registra brojaca u petljama.
        mov             ar.lc = r15

        # Zavrsava se funkcija.
        br.ret.sptk.many rp
```

Primjer predstavlja funkciju za linearno pretraživanje polja cijelih brojeva. Funkcija prima kao prvi argument broj koji se traži, a zatim kao preostala dva argumenta dužinu polja koje se pretražuje i adresu njegovog prvog elementa.

Na početku funckije se `alloc` instrukcijom specificira koliko registara funkcija koristi za svoje argumente, za lokalne promjenljive i za prenos argumenata funkcijama koje poziva, kao i za rotirajuće registre. U ovom slučaju funkcija koristi 3 registra preko kojih dobija argumente i ne koristi registre za lokalne promjenljive, prenos argumenata ili kao rotirajuće registre. Instrukcijom `alloc` se takođe specificira u kom registru se tokom izvršavanja funkcije čuva sadržaj ranije pominjanog `ar.pfs` registra koji opisuje stanje registarskog steka za prethodnu funkciju.

Slično softverskim konvencijama koje postoje u pogledu čuvanja sadržaja registara na *IA-32* arhitekturi, tako je i na *IA-64* arhitekturi predviđeno da se sadržaj određenih registara mora sačuvati tokom poziva funkcija, a nekih opet ne. Konkretno, konvencija je da se od statičkih registara opšte namjene moraju sačuvati registri `r4`, `r5`, `r6`, `r7`, `r12` i `r13`, dok se svi ostali registri mogu po volji koristiti. Slične konvencije postoje i za ostale registre, pa je tako recimo predviđeno da se sadržaj aplikacionih registara `ar.pfs` i `ar.lc` takođe mora sačuvati tokom poziva funkcije, naravno ako ih funkcija koristi. Svaka funkcija koristi registar `ar.pfs`, tako da je za njega predviđeno da se obavezno kopira u neki drugi registar u `alloc` instrukciji. U ovom slučaju je sa čuvanje njegovog sadržaja iskorišćen `r14` registar, obzirom da se zna da funkcija `lsearch()` ne poziva nijednu drugu funkciju, tako da sadržaj tog registra neće biti promijenjen. Slično, registar `r15` je iskorišćen za čuvanje `ar.lc` registra - sadržaj ovog registra se mora sačuvati jer funkcija sadrži petlju koja koristi ovaj registar kao brojač.

Nakon ovakvog prologa funkcije, provjerava se da li je dužina polja veća od 0. U tu svrhu se koristi `cmp` instrukcija. Treba uočiti kako se registar koji sadrži dužinu polja, koja je prenesena kao argument funkciji `lsearch()` referencira kao `in1`. Ovaj registar odgovara nekom od registara opšte namjene,

ali funkcija ne mora da zna koji je to tačno registar, već mu uvijek pristupa pomoću ovog alternativnog imena. Sufiksi `cmp` instrukcije određuju tip poređenja (`le`, odn. poređenje na relaciju "manje ili jednako"), kao i način na koji se ažuriraju predikatski registri u zavisnosti od rezultata poređenja (`unc` označava da se predikatski registri inicijalizuju na 0, a zatim da se u prvi registar upiše rezultat poređenja, a u drugi registar njegov komplement). Rezultat poređenja se uvijek upisuje u dva registra, u ovom slučaju to su registri `p7` i `p0`. Registar `p7` je odabran zato što je opet u pitanju registar čiji se sadržaj ne mora očuvati tokom poziva funkcije. Registar `p0` kako je već rečeno uvijek ima vrijednost 1 i upisi u taj registar se zanemaruju, tako da se ovaj registar može iskoristiti kao odredišni u instrukcijama poređenja kada se ustvari ne želi upisati neki rezultat poređenja ni u koji registar.

Instrukciju poređenja slijedi instrukcija skoka `br` na kraj funkcije ukoliko se pokaže da je rezultat poređenja afirmativan (tj. da je dužina polja manja ili jednaka 0). Ova instrukcija skoka demonstrira predikaciju instrukcija - instrukcija će biti izvršena samo ako je vrijednost predikatskog registra `p7` jednaka 1 (odn. ako je uslov prethodnog poređenja ispunjen). Pored sufiksa `cond` koji označava uslovni skok (treba primijetiti redundantnost između ovog sufiksa i predikata), instrukcija ima i neke sufikse kojima programer naznačava procesoru prirodu skoka. Sufiks `dpnt` ukazuje procesoru da treba da koristi dinamičku predikciju tj. da procesor održava statistiku koliko puta do skoka dolazi i da započinje spekulativno izvršavanje one grane koda za koju je veća vjerovatnoća da će zaista biti izvršena; pritom, inicijalno se očekuje da do skoka neće doći tako da procesor prije nego što prikupi dovoljno podataka treba da nastavlja sa izvršavanjem u tom pravcu. Sufiks `few` se odnosi na broj instrukcija koje je potrebno dovući u odgovarajuću *cache* memoriju ukoliko do skoka dođe; ovdje prefiks specificira, obzirom da u slučaju da do skoka dođe ubrzo nakon istog slijedi još jedan skok (zbog povratka iz funkcije), da taj broj bude mali.

Funkcija nakon ovoga inicijalizuje brojač u petlji koja će proći kroz sve elemente niza na indeks poslednjeg elementa. Sa druge strane, indeks tekućeg elementa u petlji će biti praktično držan u registru preko koga se vraća rezultat iz funkcije i koji se označava sa `ret0`; ovaj registar se inicijalizuje na -1. Potom se u petlji u registar `r16` (još jedan registar čiji sadržaj funkcija nije dužna da sačuva) učitava jedan po jedan element niza instrukcijom za učitavanje 32-bitnog broja `ld4`, pri čemu se svaki put adresa inkrementira za 4 (inkrement se specificira kao drugi operand instrukcije za učitavanje iz memorije). Istovremeno se ažurira indeks tekućeg elementa. Zatim se vrši poređenje; ako je tražena vrijednost, koja je prenesena funkciji preko registra `in0` jednaka tekućem elementu koji se nalazi u registru `r16`, izlazi se iz petlje. Provjera je izvedena instrukcijom skoka koja slijedi instrukciju poređenja na sličan način kao što je na početku funkcije implementirana provjera da li je dužina niza veća od 0.

Petlja se završava instrukcijom skoka sa sufiksom `cloop` koji naznačava da je u pitanju brojačka petlja odn. da treba provjeriti da li je brojač jednak 0, pa ako je taj uslov ispunjen onda treba izaći iz petlje, a ako nije dekrementirati brojač i vratiti se na početak petlje.

Ukoliko se petlja završi time što brojač postane jednak 0, znači da se tražena vrijednost ne nalazi u polju i u registar preko koga se vraća rezultat se ponovo stavlja -1 kao indikator takvog rezultata. Potom se vraća registar `ar.lc` na vrijednost koju je imao pri ulasku u funkciju i završava se funkcija još jednom instrukcijom skoka koja ima sufiks `ret` što označava da se odnosi na povratak iz procedure, te sufikse `sptk` koji označava da treba koristiti statičku predikciju (spekulatirno izvršavanje se uvijek nastavlja u jednoj grani) i da se očekuje da će do skoka doći odn. `many` koji naznačava da u `cache` memoriju treba dovući dosta instrukcija sa odredišne adrese skoka obzirom da se kontrola u potpunosti prenosi tamo.

Treba uočiti kako su kroz kod raspoređene grupe instrukcija koje se mogu paralelno izvršavati i korelisati njihov raspored sa ranije navedenim pravilima u vezi istog.

IA-64 arhitektura podrazumijeva sasvim drugačiji pristup asemblerskom programiranju od *IA-32* arhitekture; programiranje u paralelnom modelu izvršavanja je znatno komplikovanije od programiranja u sekvencijalnom modelu. Zato je za očekivati da će asemblersko programiranje još više izgubiti na značaju u korist sofisticiranih optimizujućih prevodilaca. Ipak, učenje asemblerskog programiranja će nesumnjivo ostati važna stepenica pri upoznavanju načina funkcionisanja mikroprocesorskih sistema.

Literatura

[1] http://www.matf.bg.ac.yu/r4mr/

[2] J. Peek, G. Todino, J. Strang; "Learning the UNIX Operating System"; O'Reilly & Associates, 2002.

[3] A. Samardžić; "GNU programerski alati"; Matematički fakultet, 2001.

[4] M. J. S. Smith; "Application-Specific Integrated Circuits"; Addison-Wesley, 1997.

[5] R. Lipsett, C. F. Schaefer, C. Ussery; "VHDL: Hardware Description and Design"; Kluwer Academic Publishers, 1989.

[6] S. Palnitkar; "Verilog HDL - A Guide to Digital Design and Synthesis"; SunSoft Press, 1996.

[7] V. C. Hamacher, Z. G. Vranesic, S. G. Zaky; "Computer organization"; McGraw-Hill, 2002.

[8] "IA-32 Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture"; Intel Corporation, 2002.

[9] "IA-32 Intel Architecture Software Developer's Manual - Volume 2: Instruction Set Reference"; Intel Corporation, 2002.

[10] "IA-32 Intel Architecture Software Developer's Manual - Volume 3: System Programming Guide"; Intel Corporation, 2002.

Indeks

8086, 81, 86
8087, 159
80287, 159
80386, 81, 101, 155
80486, 81, 123, 155

A

ADC, 96
ADD, 93, 96
adresiranje operanada, 85
algebarske transformacije, 5
Alpha arhitektura, 211
always, 36
and, 15
aplikativni režim, 81
arhitektura sistema, 51
aritmetičke instrukcije, 158
aritmetičko-logičke instrukcije, 95, 192
__asm__, 145
assign, 27
AT&T sintaksa, 88

B

begin, 18
behavioral modelovanje, 55
bitske instrukcije, 113
Booth-ov algoritam, 48
BOUND, 113
brojači, 42
BSF, 116
BSR, 116
.bss, 90
BSWAP, 101
BYTE PTR, 99

C

C konvencije za pozivanje potprograma, 141
C konvencije za smještanje podataka, 147
cache memorija, 198
CALL, 92, 103, 129
call gate-ovi, 143
carry-lookahead sabirač, 29
case, 57
casex, 57
casez, 57
CBW, 102
CDQ, 99, 102
CISC arhitekture, 211
CLC, 100
CLD, 119
CMOV, 101
CMP, 96
CMPPS, 193
CMPSS, 193
CMPXCHG, 101
CMPXCHG8, 101
COMISS, 194
CPUID, 123
CVTDQ2PD, 207
CVTDQ2PS, 207
CVTPD2DQ, 207
CVTPD2PI, 207
CVTPD2PS, 207
CVTPI2PD, 207
CVTPI2PS, 195
CVTPS2DQ, 207

CVTPS2PD, 207
CVTPS2PI, 195
CVTSD2SI, 207
CVTSD2SS, 207
CVTSI2SD, 207
CVTSI2SS, 195
CVTSS2SD, 207
CVTSS2SI, 195
CVTTPD2DQ, 207
CVTTPD2PI, 207
CVTTPS2DQ, 207
CVTTPS2PI, 207
CVTTSD2SI, 207
CVTTSS2SI, 207
CWD, 102
CWDE, 102

D

D gated latch, 34
.data, 90
data flow modelovanje, 48
DEC, 96
'define, 23
deklaracije parametara, 14
deklaracije portova, 14
deklaracije pomoćnih signala, 14
deklaracije promjenljivih, 14
dekoderi, 44
denormalizovani brojevi, 154
dijeljenje sa nulom, 156
disable, 77
$display, 23
DIV, 97
dizajn kompletnog sistema, 51
don't care slučajevi, 7
double precision brojevi, 154
DWORD PTR, 99

E

eksponencijalne funkcije, 164
eksponent, 154
ekvivalentne funkcije, 5
elementarne logičke komponente, 25
ELF format, 148
'else, 23
EMMS, 178, 183
end, 18
'endif, 23
endmodule, 14
ENTER, 127, 136
EPIC arhitekture, 212
epilog potprograma, 136
extended precision brojevi, 155

F

F2XM1, 165
FABS, 159
FADD, 158
fan-in ograničenje, 7
FCHS, 159
FCLEX, 168
FCMOV, 158
FCOM, 160
FCOMI, 161
FCOMIP, 161
FCOMP, 160
FCOMPPP, 160
FCOS, 164
FDECSTP, 168
FFREE, 160, 168
FIADD, 159
FICOM, 160
FICOMP, 160
FILD, 157
FINCSTP, 160, 168
$finish, 23
FINIT, 167
FIST, 157
FISTP, 157
FLD, 156
FLDCW, 168
FLDENV, 168
flip-flopovi, 31
FNCLEX, 168
FNINIT, 167
FNSAVE, 168

FNSTCW, 168
FNSTENV, 168
FNSTSW, 168
for, 76
forever, 76
format *IA-64* instrukcija, 213
format izvršnih fajlova, 147
FPATAN, 164
FPREM, 159
FPREM1, 159
FPTAN, 164
FRNDINT, 160
FRSTOR, 168, 183
FSAVE, 168, 183
FSCALE, 165
FSIN, 164
FSINCOS, 164
FSQRT, 159
FST, 157
FSTCW, 168
FSTENV, 168
FSTP, 157
FSTSW, 168
FTST, 161
FUCOM, 160
FUCOMP, 160
FUCOMPP, 160
FWAIT, 168
FXAM, 161
FXCH, 157
FXRSTOR, 192, 206
FXSAVE, 192, 205
FXTRACT, 160
FYL2X, 165
FYL2XP1, 165

G

gate level modelovanje, 15
.global, 91
GNU asembler
  direktive, 90
  labele, 90
GNU asembler
  deklarisanje tipova podataka, 90
  identifikatori, 90
  komentari, 91
GNU asembler, 88
grafički prikaz rezultata simulacije, 20
GTKWave program, 20

H

HDL jezici, 13
HDL procesi, 16

I

I/0 permission bitmap, 173
i386_set_ioperm(), 173
IA-32 arhitektura, 81
IA-64 arhitektura, 211
Icarus Verilog program, 13
IDIV, 97
IEEE-754 standard, 153
if, 57
‘ifdef, 23
IMUL, 96
IN, 172
INC, 96
incijalizacija parametara, 48
‘include, 23
initial, 18
inline asemblersko programiranje, 145
inout, 14
input, 14
INS, 172
instanciranje modula, 16
instrukcije šiftovanja, 182
instrukcije kontrole toka, 103
instrukcije konverzije podataka, 180, 194
instrukcije opšte namjene, 95
instrukcije poređenja, 160, 180, 193
instrukcije za rad sa cijelim brojevima, 209
instrukcije za rad sa flegovima, 122

instrukcije za rad sa realnim brojevima, 153, 205
instrukcije za rad sa stringovima, 119
instrukcije za transfer podataka, 156, 191
instrukcije za učitavanje konstanti, 158
`INT`, 103, 113
`integer`, 22
Intel-ova sintaksa, 88
`.intel_syntax`, 89
`INTO`, 113
`ioperm()`, 173
`IRET`, 103, 171
Itanium, 212
Itanium2, 212
izbor logičke baze, 7
izvršavanje *IA-64* instrukcija, 213

J

`JECXZ`, 107

K

Karnaugh-ove mape, 5
kombinatorni sklopovi, 31
koncept modula, 14
kontroler sedmosegmentnog indikatora, 8
kontrolne instrukcije, 167

L

`.lcomm`, 98
`LDMXCSR`, 192, 205
`LEA`, 92, 123
`LEAVE`, 127, 137
`LIDT`, 172
little-endian arhitektura, 86
load-store arhitekture, 211
`LODS`, 119
logaritamske funkcije, 164
logička baza, 4
logičke instrukcije, 181
logičke primitive, 3
lokalne promjenljive u potprogramu, 134
`LOOP`, 106
`LOOPE`, 107
`LOOPNE`, 107
`LOOPNZ`, 107
`LOOPZ`, 107

M

mantisa, 154
`MASKMOVQ`, 199
master-slave D flip-flop, 35
mehanizam događaja, 55
mehanizam obrade grešaka koprocesora, 168
mehanizam prekida, 84, 171
memorijske lokacije
    temporalnog tipa, 198
memorijske lokacije
    netemporalnog tipa, 198
mikroarhitektura, 81
minimalna funkcija, 5
minimalne baze, 7
minimizacija logičkih funkcija, 4
MIPS arhitektura, 211
MMX instrukcije, 177
množači, 45
model aritmetičko-logičke jedinice, 54
model *datapath* komponente, 59
model kompletnog mikroprocesora, 71
model kontrolne jednice, 65
model skupa registara opšte namjene, 58
modul stimulansa, 16
modularna organizacija programa, 138
`module`, 14
`$monitor`, 20
`$monitoroff`, 23
`$monitoron`, 23

MOV, 99, 101, 179
MOVAPS, 191
MOVHLPS, 192
MOVHPS, 192
MOVLHPS, 192
MOVLPS, 192
MOVMSKPS, 192
MOVNTPS, 199
MOVNTQ, 198
MOVSS, 191
MOVSX, 102
MOVUPS, 191
MOVZX, 102
MUL, 96
multiplekseri, 44

N

nand, 15
naredba kontinualne dodjele, 27
naredba proceduralne dodjele, 56
naredba proceduralne dodjele
  blokirajuća, 56
  neblokirajuća, 56
naredbe modula, 14
NEG, 96
negedge, 56
neprecizno izračunavanje, 156
nivoi privilegija, 81
NOP, 123
noprefix, 89
nor, 15
not, 15
not a number broj, 154
numerički koprocesor, 155

O

or, 15
organizacija sistema, 53
osnove logičkog dizajna, 3
osnovna *Verilog* sintaksa, 21
ostale instrukcije opšte namjene, 123
ostale *SSE* instrukcije, 196
ostale *SSE2* instrukcije, 210
OUT, 172
output, 14
OUTS, 172

P

PA-RISC arhitektura, 211
PACKSSDW, 181
PACKSSWB, 181
PACKUSWB, 181
PADD, 179
parameter, 23
PAVGB, 196
PAWGW, 196
PCMPEQ, 180
PCMPGT, 180
Pentium, 81, 177, 212
Pentium II, 81
Pentium III, 199
Pentium IV, 189, 205
Pentium II, 81
Pentium III, 81, 189, 199
Pentium IV, 81, 199
Pentium MMX, 81
PentiumPro, 81, 101, 161
PEXTRW, 196
PINSRW, 196
pipeline, 211
PMADDWD, 179
PMAXSW, 197
PMAXUB, 197
PMINSW, 197
PMINUB, 197
PMOVMSKB, 197
PMULHUW, 197
PMULHW, 179
PMULLW, 179
pogrešna operacija, 156
pokretanje izvršnih fajlova, 147
polusabirač, 25
pomjerački registar, 40
"pomjereni" eksponent, 154
POP, 92, 102
POPA, 102

POPFD, 122
POPF, 122
portovi modula, 14
posedge, 56
potkoračenje, 156
potprogrami, 129
povezivanje asemblerskog i *C* koda, 141
pozivanje potprograma, 129
PREFETCH, 199
prekoračenje, 156
prekoračenje kod *MMX* instrukcija, 178
prenošenje argumenata u potprograme, 130
preset-clear D flip-flop, 37
pristup argumentima potprograma, 130
prisustvo denormalizovanog operanda, 156
privilegovane instrukcije, 171
prolog potprograma, 136
PSADWB, 197
PSHUFW, 197
PSLLD, 182
PSLLQ, 182
PSLLW, 182
PSRA, 182
PSRL, 182
PSUB, 179
puni sabirač, 25
PUNPCKHBW, 181
PUNPCKHDQ, 181
PUNPCKHWD, 181
PUNPCKLBW, 181
PUNPCKLDQ, 181
PUNPCKLWD, 181
PUSH, 92, 102
PUSHA, 102

Q

Quartus II program, 9
QWORD PTR, 99

R

razlike između *AT&T* i *Intel*-ove sintakse, 145
režim saturacije, 178
$readmemb, 74
$readmemh, 73
real, 22
realizacija dizajna, 11
realni tipovi podataka, 153
reentrant kod, 137
reg, 16
registar EFLAGS, 83
registar EIP, 83
register transfer level modelovanje, 76
registri, 39
repeat, 76
RET, 103, 129
ripple brojač, 43
ripple-carry sabirač, 28
RISC arhitekture, 211
rotirajući registar, 41
RS latch, 31
RS gated latch, 33

S

sabirači, 25
SBB, 96
segmentni registri, 83
sekvencijalni sklopovi, 31
SET, 116
SFENCE, 199
SHLD, 113
SHRD, 113
SHUFPS, 207
SHUFPS, 194
SIDT, 172
signal takta, 36
single precision brojevi, 154
simulacija dizajna, 9
sinteza logičkih funkcija, 4
sistemski pozivi iz asemblerskog koda, 143

sistemski režim, 81
skup instrukcija, 87
skup instrukcija *IA-64* arhitekture, 214
skup *MMX* registara, 177
skup registara *IA-32* arhitekture, 82
skup registara *IA-64* arhitekture, 213
skup registara numeričkog koprocesora, 155
skup registara za *SSE* instrukcije, 190
softverski prekidi, 143
SPARC arhitektura, 211
SSE instrukcije, 189
SSE2 instrukcije, 205
stack frame, 133
standardni simboli, 3
`STD`, 119
stek, 92
`STMXCSR`, 192, 205
`$stop`, 23
`STOS`, 119
`SUB`, 96
suma proizvoda, 4
superskalarno izvršavanje, 211

T

T flip-flop, 36
tablice istinitosti, 4
`TEST`, 116
testiranje funkcionalnosti modula, 16
`.text`, 90
tijelo modula, 14
`$time`, 23
`time`, 22
`‘timescale`, 18
tipovi podataka, 85
`tri`, 73
trigonometrijske instrukcije, 164

U

`UCOMISS`, 194
`UD2`, 124
ulazno-izlazne instrukcije, 172
`UNPCKHPD`, 207
`UNPCKHPS`, 195
`UNPCKLPD`, 207
`UNPCKLPS`, 195
uparivanje po imenima, 17
uparivanje signala i porta, 17
upotreba *MMX* instrukcija, 182
uvod u *IA-32* arhitekturu, 81

V

VAX arhitektura, 211
VCD format, 20
Verilog
    bitski vektori, 22
    brojne konstante, 21
    funkcije, 77
    imena identifikatora, 22
    komentari, 16
    konstante, 23
    leksičke konvencije, 21
    napredne konstrukcije, 76
    operatori, 49
    ostali tipovi podataka, 22
    polja signala, 22
    pretprocesorske direktive, 23
    referenciranje simbola, 17
    sistemske direktive, 23
    specifikacija kašnjenja, 18
    string konstante, 22
    task-ovi, 77
    tipovi signala, 22
    vrijednosti signala, 22
Verilog HDL jezik, 13
VLIW arhitekture, 212
vraćanje rezultata iz potprograma, 131

W

`WAIT`, 168
`while`, 76
`wire`, 15
`WORD PTR`, 99

wraparound režim, 178

X

XADD, 101
XCHG, 101
XLATB, 123
xnor, 15
xor, 15