5.

Optimizacija dizajna

U cilju postizanja zahtevanih performansi u pogledu brzine, veličine, potrošnje, itd., na raspolaganju nam stoji čitav niz optimizacionih tehnika koje možemo koristiti prilikom projektovanja digitalnog sistema koji treba da implementira odabrani algoritam. Korišćenjem ovih tehnika polazni algoritam se transformiše u odgovarajuću ekvivalentnu formu, koja obezbeđuje efikasnije (brže) izvršavanje ili racionalnije korišćenje hardverskih resursa. Optimizacione tehnike mogu se podeliti u dve grupe:

- **Transformacije koda** (*Code Transformations*) – ove transformacije primenjuju se na pseudo-kod algoritma. Primenom ovih transformacija postojeća konkurentnost operacija i lokalnost podataka unutar algoritma čini se vidljivijom, što se zatim može iskoristiti, na različitim nivoima, u cilju postizanja efikasnije hardverske implementacije. Ovu grupu čine sledeće transformacije:
    - **Transformacije na nivou bitova** (*Bit-Level Transformations*) – Ova grupa transformacija ima za cilj da istakne tačan broj neophodnih bitova za reprezentaciju promenljivih i izvođenje operacija prisutnih u algoritmu. U ovu grupu spadaju sledeće transformacije: *Bit-Width Narrowing*, *Bit-Level Optimization*, *Floating to Fixed-Point Conversion*, itd. Većina ovih transformacija diskutovana je u okviru kursa Diskretni sistemi.
    - **Transformacije na nivou instrukcija** (*Instruction-Level Transformations*) – U ovu grupu spadaju različite algebarske transformacije koje se primenjuju nad instrukcijama ili grupama

instrukcija unutar pseudo-koda algoritma, kao što su: *Common Subexpression Elimination*, *Constant Folding*, *Constant Propagation*, *Operator Strength Reduction*, *Tree-Height Reduction*, *Code Motion*, itd.

- **Transformacije petlji** (*Loop-Level Transformations*) – Ove transformacije primenjuju se na petljama koje postoje unutar algoritma, sa ciljem da se postojeći paralelizam na nivou instrukcija (*Instruction Level Parallelism*, *ILP*) iskoristi u meri u kojoj to dopuštaju raspoloživi hardverski resursi (broj funkcionalnih jedinica, broj memorija, broj registarskih banki, njihova organizacija, itd.). Najpoznatiji predstavnici ove grupe transformacija su: *Loop Rolling/Unrolling*, *Loop Folding*, *Loop Tiling*, *Loop Strip-Mining*, *Loop Merging*, *Loop Distribution*, itd.
- **Transformacije podataka** (*Data-Oriented Transformations*) – Ova grupa transformacija reorganizuje podatke koji se obrađuju unutar algoritma na način koji obezbeđuje ili omogućava efikasnije izvođenje operacija nad njima. Ova grupa transformacija najčešće se primenjuje zajedno sa transformacijama petlji. Najpoznatiji predstavnici ove grupe transformacija su: *Data Distribution*, *Data Replication*, *Data Reuse*, *Scalar Replacement*, itd.

- **Optimizacije procesa mapiranja i izvršavanja** (*Mapping and Execution Optimizations*) – ove transformacije omogućavaju ostvarivanje različitih načina mapiranja postojećih operacija i promenljivih, koje postoje unutar algoritma, na raspoložive hardverske resurse (funkcionalne jedinice, registre, registarske banke, memorije, itd.), koji postoje u projektovanom digitalnom sistemu. Prilikom projektovanja digitalnog sistema koji će implementirati odabrani algoritam, na rasploganju nam stoji široki spektar mogućih implementacija, koje se međusobno razlikuju po broju, vrsti i načinu korišćenja hardverskih resursa. Primenom optimizacionih postupaka iz ove grupe moguće je ostvariti željeni balans korišćenja raspoloživih hardverskih resursa u prostoru i u vremenu. Najpoznatije optimizacione tehnike iz ove grupe koje se koriste u praksi su:
  - Deljenje resursa (*Resource Sharing*)
  - Protočna obrada (*Pipelining*)

5.1. Deljenje resursa (*Resource Sharing*)

Tehnika deljenja resursa bazira se na ideji korišćenja istog hardverskog resursa za implementaciju većeg broja različitih operacija koje su prisutne u algoritmu. Ukoliko se ove operacije ne izvršavaju u istom trenutku (istovremeno), mogu se mapirati na isti hardverski resurs. Na ovaj način se može ostvariti značajna ušteda potrebnih hardverskih resursa, bez značajnog degradiranja brzine rada projektovanog sistema.

Kako tehnika deljenja resursa zahteva korišćenje dodatnih hardverskih resursa (u vidu multiplekserskih mreža pomoću kojih se u različitim trenucima na deljeni resurs dovode različiti podaci), i dovodi do povećanja kritične putanje kroz sistem (što ima za posledicu redukciju maksimalne učestanosti na kojoj sistem može da stabilno radi), ima je smisla koristiti samo u slučaju kada se dele veliki, složeni hardverski resursi. Iako neki od savremenih alata za automatsku sintezu hardvera ovu optimizaciju primenjuju automatski, često se za postizanje optimalnih rezultata ona mora sprovesti ručno, od strane dizajnera. Postoje dva tipa deljenja resursa:

- **Deljenje operatora** (*Operator Sharing*) – ovo je najpoznatija varijanta deljenja resursa, kod koje se zapravo identifikuju operacije koje su istog tipa (na primer sve identifikovane operacije su operacije sabiranja) ali su međusobno isključive, u svakom trenutku aktivna je samo jedna od navedenih operacija. Ovo je čest slučaj prilikom korišćenja naredbi grananja unutar algoritma.
- **Deljenje funkcionalnosti** (*Functionality Sharing*) – ovo je složeniji tip deljenja resursa. Ukoliko u algoritmu postoje operacije/funkcije koje nisu istog tipa ali imaju neke srodne karakteristike, a ne izvršavaju se istovremeno, moguće je projektovati hardverski resurs koji je u stanju da izvrši svaku od navedenih operacija. Na primer, ukoliko u algoritmu imamo operacije sabiranja i oduzimanja, one se mogu implementirati samo pomoću sabirača, sa dodatnim kolom za negaciju drugog operanda, ukoliko su vrednosti operanada koji se sabiraju/oduzimaju predstavljene u sistemu drugog komplementa. Ovakva realizacija zahteva manje resursa od realizacije koja bi bila bazirana na korišćenju jednog sabirača i jednog oduzimača.

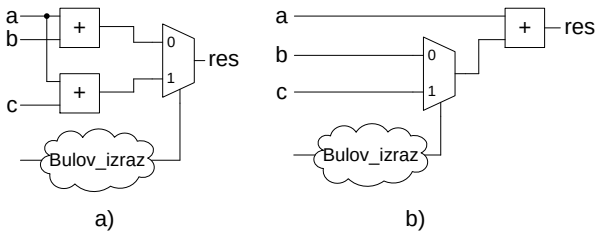
Kao što je već rečeno, samo deljenje resursa postiže se uvođenjem dodatnih multiplekserskih kola, koji se nalaze ili na ulazima deljenog resursa ili na njegovom izlazu, u zavisnosti od toga da li deljeni operator ima zajedničke ulazne, odnosno izlazne operande.

Na primer, posmatrajmo sledeću naredbu grananja

```
if (Bulov_izraz) then
   res = a+c;
else
   res = a+b;
```

U ovom slučaju možemo da delimo jedan sabirač, između dve naredbe dodele vrednosti, jer se one nalaze u međusobno isključivim granama **if** naredbe. U slučaju kada je *Bulov_izraz* zadovoljen izvršava se prva naredba dodele i promenljivoj *res* dodeljuje se vrednost $a+b$. Druga naredba dodele vrednosti se u ovom slučaju ne izvršava. Obrnuta, kada *Bulov_izraz* nije zadovoljen izvršava se druga naredbe dodele i ovoga puta se promenljivoj *res* dodeljuje vrednost $a+c$, a prva naredba dodele vrednosti se slučaju ne izvršava. Kako se u obe naredbe dodele vrednosti koristi isti operator, '+', umesto korišćenja dva sabirača možemo koristi samo jedan. Slika 5.1 prikazuje strukturu

digitalnog kola koje realizuje gornju **if** naredbu u slučaju kada se ne koristi deljenje resursa (slika 5.1a), i u slučaju kada se koristi deljenje resursa (slika 5.1b). Obratite pažnju da se dodatni multiplekseri za rutiranje koriste samo na ulazima sabirača, jer se u obe naredbe u kojima se koristi operator sabiranja izračunata vrednost dodeljuje istoj promenljivoj, odnosno deljeni operator ima zajednički izlazni operand.

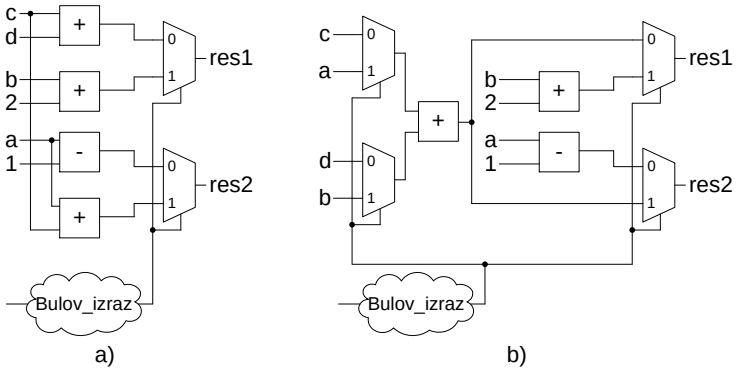


Slika 5.1. Deljenje operatora, zajednički izlazni operand

Kao ilustraciju deljenja operatora između operacija koje imaju različite i ulazne i izlazne operande, posmatrajmo sledeću naredbu grananja

```
if (Bulov_izraz) then
{
   res1 = b+2;
   res2 = a+b;
}
else
{
   res1 = c+d;
   res2 = a-1;
}
```

I u ovom slučaju možemo da delimo jedan sabirač, između dve naredbe dodele vrednosti promenljivim *res2* i *res1*, koje se nalaze u **then** i **else** granama **if** naredbe respektivno. Obratite pažnju da ne možemo deliti sabirač između dve naredbe dodele vrednosti promenljivim *res1* i *res2* koje se nalaze unutar **then** grane, jer se one izvršavaju istovremeno. Slika 5.2 prikazuje strukturu digitalnog kola koje realizuje gornju **if** naredbu u slučaju kada se ne koristi deljenje resursa (slika 5.2a), i u slučaju kada se koristi deljenje resursa (slika 5.2b). Kako su ovoga puta i ulazni i izlazni operandni operatora sabiranja koji delimo različiti, moramo koristiti dodatne multipleksere i na ulazima i na izlazu sabirača.



Slika 5.2. Deljenje operatora, različiti ulazni i izlazni operandi

Zadaci za vežbu

Zadatak 5.1.

Potrebno projektovati digitalni sistem koji može da izvrši sledeće četiri operacije: *a+b*, *a-b*, *a+1*, *a-1*. Ulazi *a* i *b* predstavljaju 16-bitni neoznačene brojeve, a željena operacija se bira pomoću 2-bitnog kontrolnog signala, *ctrl*.

a) Projektovati sistem korišćenjem dva sabirača, jednog inkrementera i jednog dekrementera. Napisati VHDL model za projektovani sistem.

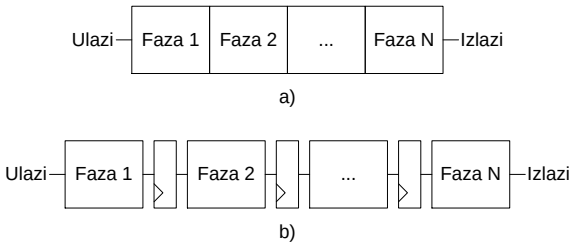
b) Projektovati sistem korišćenjem samo jednog sabirača. Napisati VHDL model za projektovani sistem.

c) Izvršiti sintezu oba dizajna korišćenjem Xilinx Vivado alata. Uporediti potrebne resurse i performanse za oba dizajna.

5.2. Protočna obrada podataka (*Pipelining*)

Protočna obrada je tehnika optimizacije dizajna kojom se povećavaju performanse projektovanog sistema, uz minimalno korišćenje dodatnih hardverski resursa. Osnovna ideja protočne obrade sastoji se u preklapanju obrade nekoliko ulaznih paketa kako bi se veći broj paketa mogao obraditi u istom intervalu vremena. Na osnovu prethodno rečenog možemo zaključiti da se uvođenjem protočne obrade u sistem, povećava njegova propusna moć.

Protočna obrada se tipično primenjuje na kombinacione mreže, gde se proces generisanja izlaznih signala iz mreže deli na faze, između kojih se umeću baferi (najčešće su to registri, ređe lečevi), kao što je prikazano na slici 5.3.



Slika 5.3. Princip protočne obrade: a) originalna kombinaciona mreža podeljena na N faza, b) ekvivalentna kombinaciona mreža sa protočnom obradom

Uloga bafera je da sačuvaju vrednosti svih unutrašnjih signala koji predstavljaju izlaze iz prethodne faze, kako bi se u narednom ciklusu mogli koristiti kao ulazi u narednu fazu.

Teorijsko povećanje propusne moći sistema sa *N* faza protočne obrade u odnosu na originalni sistem, iznosi *N* puta. U praksi se ova vrednost ne može dostići jer najčešće nije moguće izvršiti podelu originalnog kola na *N* faza sa identičnim kašenjem, a i umetnuti registri, zbog konačnih vrednosti vremena uspostavljanja na njihovim ulazima i vremena pojave novih vrednosti na njihovim izlazima, dovode do povećanja periode klok signala. Iz ova dva razloga perioda klok signala u realnim sistema sa protočnom obradom uvek zadovoljava sledeći uslov

$$T_{clk\_pipe} > \frac{T_{comb}}{N}.$$

Imajući ovo u vidu, i propusna moć realnog sistema sa protočnom obradom ovek je manja od teorijske

$$Throughput_{pipe} = \frac{1}{T_{clk\_pipe}} < \frac{N}{T_{comb}} = N \cdot Throughput_{comb}.$$

Obratite pažnju da se primenom protočne obrade može povećati samo propusna moć sistema. Kašnjenje u obradi pojedinačnih paketa ostaje isto kao i u slučaju originalne kombinacione mreže.

Iako se protočna obrada može implementirati unutar svake kombinacione mreže, da bi se ostvarila značajna povećanja propusne moći moraju biti zadovoljeni sledeći uslovi:

- Postoji dovoljna količina ulaznih podataka koji će obezbediti adekvatnu iskorišćenost sistema sa protočnom obradom
- Povećanje propusne moći sistema je glavni kriterijum što se tiče performansi
- Kombinaciona mreža se može podeliti na faze koje imaju podjednaka propagaciona kašnjenja
- Propagaciona kašnjenja svake od faza u protočnoj obradi su znatno veća od vremena uspostavljanja na ulazima registara (T_{setup}) i kašnjenja pojave novih vrednosti na izlazima registara (T_{cq})

Postupak uvođenja protočne obrade u originalnu kombinacionu mrežu sastoji se iz sledećih koraka:

1. Nacrta se blok dijagram originalne kombinacione mreže u obliku lanca sukcesivnih koraka u procesu transformacije ulaza u izlaze
2. Identifikuju se glavne komponente koje se koriste unutar svakog od koraka transformacije i procene se njihova propagaciona kašnjenja
3. Izvrši se podela lanca obrade u faze, grupisanjem susednih komponenti iz lanca u grupe sa sličnim propagacionim kašnjenjima
4. Identifikuju se signali koji prelaze granice između susednih faza
5. Umetnu se registri na svakom od identifikovanih signala

Sledeći primer ilustruje kako se ranije razmatrani „*Add-and-Shift*“ množač može modifikovati tako da koristi protočnu obradu podataka.

Primer 5.1: Modifikovani „*Add-and-Shift*“ množač sa protočnom obradom podataka

Množač dva neoznačena broja baziran na „*Add-and-Shift*“ algoritmu koji je bio razmatran ranije, u slučaju da se fiksira veličina ulaznih operanada, može se realizovati i pomoću „*Structural Data Flow*“ tehnike, razmatrane na predavanjima.

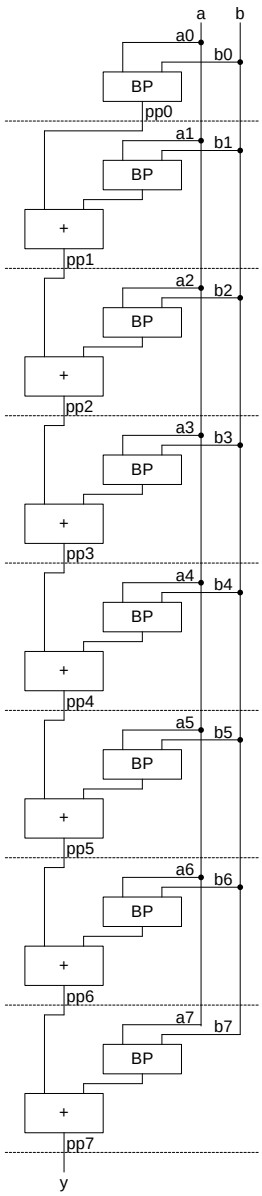
Kao što se može zaključiti na osnovu slike 4.10 i opisa koraka u „*Add-and-Shift*“ algoritmu, prikazanom na slici 4.11, množenje dva binarna broja se zapravo može svesti na sabiranje parcijalnog proizvoda sa pomerenim članovima b_i*A kako bi se dobio konačni rezultat. Sabirači čine glavne module u ovakvom dizajnu. U slučaju da su ulazni operandi predstavljeni sa *n* bita, s'obzirom da će veličina parcijalnog proizvoda biti *2n* bita, potrebno je ukupno *n-1 2n*-bitnih sabirača za realizaciju ovog sistema.

Jedan od načina da se smanji veličina rezultujućeg digitalnog kola jeste da se sumiraju pomereni članovi b_i*A. Na ovaj način moguće je smanjiti veličinu potrebnih sabirača sa *2n* bita na *n+1* bita. Princip rada novog dizajna ilustrovan je na slici 5.4, u slučaju

4-bitnih operanada, a blok dijagram digitalnog sistema na slici 5.5, u slučaju da su ulazni operandi 8-bitni.

				a_3	a_2	a_1	a_0	Operand 1
X				b_3	b_2	b_1	b_0	Operand 2
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	
			$pp0_4$	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$	Parcijalni proizvod *pp0*
		+	a_3b_1	a_2b_1	a_1b_1	a_0b_1		
		$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$		Parcijalni proizvod *pp1*
	+	a_3b_2	a_2b_2	a_1b_2	a_0b_2			
	$pp2_4$	$pp2_3$	$pp2_2$	$pp2_1$	$pp2_0$			Parcijalni proizvod *pp2*
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
$pp3_4$	$pp3_3$	$pp3_2$	$pp3_1$	$pp3_0$				Parcijalni proizvod *pp3*
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	Proizvod

Slika 5.4. Alternativni način formiranja proizvoda



Slika 5.5. Blok dijagram modifikovanog sistema baziranog na „Add-and-Shift“ algoritmu u slučaju 8-bitnih ulaznih operanada

Dve glavne komponente u sistemu sa slike 5.5 su *n+1*-bitni sabirač i komponenta koja je zadužena za izračunavanje članova b_i*A, označena sa *BP* na slici 5.5. Ovaj modul zapravo vrši izračunavanje logičkog „I“ između b_i i pojedinačnih bitova operanda *A*.

Iako jednostavan za realizaciju, predloženi sistem ima jedan ozbiljan nedostatak, malu brzinu rada zbog dugačkog kritičnog propagacionog puta koji prolazi kroz sve sabirače module.

S'obzirom na izuzetno regularnu strukturu sistema sa slike 5.5, koji se lako može podeliti na 7 gotovo identičnih sekcija (razdvojenih isprekidanim linijama na slici 5.5), prirodno se nameće ideja o uvođenju protočne obrade radi povećanja brzine rada. Sistem sa slike 5.5 podelićemo na ukupno 7 faza, od kojih svaka sadrži po jedan sabirač i jedan *BP* modul, osim prve faze. Prva faza sadržaće dva *BP* modula i jedan sabirač. Između sukcesivnih faza protočne obrade potrebno je umetnuti odgovarajući broj registara koji treba da sačuvaju vrednosti svih signala koji prelaze iz jedne faze obrade u narednu.

Zadaci za vežbu

Zadatak 5.2.

Na osnovu blok dijagrama sistema sa slike 5.5:

a) Napisati VHDL model 8-bitnog „*Add-and-Shift*" množača koji ne koristi tehniku protočne obrade.

b) Nacrtati blok dijagram 8-bitnog „*Add-and-Shift*" množača koji koristi tehniku protočne obrade. Na osnovu blok dijagrama napisati VHDL model 8-bitnog „*Add-and-Shift*" množača koji koristi tehniku protočne obrade.

c) Napisati jednostavan testbenč pomoću kojega je moguće verifikovati ispravan rad projektovanih množača.

d) Izvršiti sintezu oba modela i uporediti dobijene rezultate u terminima potrebnih resursa i maksimalne brzine rada.

e) Za oba množača, koristeći podatke dobijene pod d), izračunati propusnu moć (broj operacija množenja koje se mogu izvršiti u jedinici vremena) i inicijalno kašnjenje.

Zadatak 5.3.

Broj stanja u protočnoj obradi moguće je lako kontrolisati dodavanjem, odnosno oduzimanjem bafer registara.

a) Nacrtati blok dijagram 8-bitnog „*Add-and-Shift*" množača koji koristi tehniku protočne obrade. Ovaj put je potrebno projektovati sistem koji će imati samo dve faze u protočnoj obradi. Na osnovu blok dijagrama napisati VHDL model 8-bitnog „*Add-and-Shift*" množača koji koristi tehniku protočne obrade.

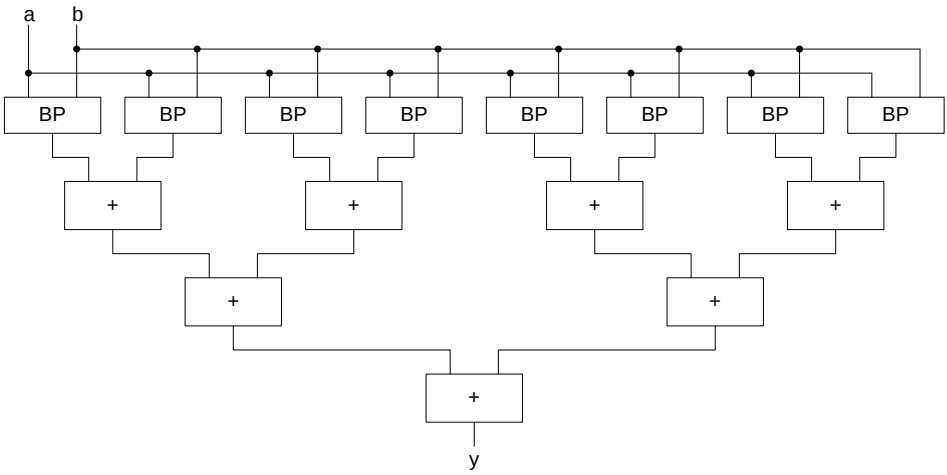
b) Napisati jednostavan testbenč pomoću kojega je moguće verifikovati ispravan rad projektovanog množača.

c) Izvršiti sintezu modela i uporediti dobijene rezultate sa onim iz zadatka 5.2.

d) Koristeći podatke dobijene pod c), izračunati propusnu moć (broj operacija množenja koje se mogu izvršiti u jedinici vremena) i inicijalno kašnjenje i uporediti ih sa onim dobijenim u zadatku 5.2.

Zadatak 5.4.

U slučaju projektovanja sistema koji se odlikuju kaskadnom vezom sekcija (što se lepo može videti na slici 5.5), moguće je izvršiti njihovo rearanžiranje kako bi se smanjilo inicijalno kašnjenje. U slučaju *n*-bitnog „*Add-and-Shift*“ množača, kritična putanja sastoji se od *n*-1 sabirača povezanih u kaskadnu mrežu, što se može videti na slici 5.5 u slučaju 8-bitnog množača. Kritičnu putanju moguće je smanjiti na samo *[log(n-1)]* sabirača koji su ovaj put organizovani u mrežu tipa stablo. Ova struktura, u slučaju 8-bitnog „*Add-and-Shift*“ množača, prikazana je na slici 5.6. Bitno je napomenuti da se ista reorganizacija može primeniti i na sistem sa protočnom obradom podataka.



Slika 5.6. Blok dijagram 8-bitnog „Add-and-Shift“ množača kod kojega su sabirači organizovani u strukturu tipa stabla

a) Na osnovu blok dijagrama sistema sa slike 5.6 napisati VHDL model 8-bitnog „*Add-and-Shift*“ množača sa strukturom tipa stablo.

b) Nacrtati blok dijagram 8-bitnog „*Add-and-Shift*“ množača tipa stablo koji koristi tehniku protočne obrade. Na osnovu blok dijagrama napisati VHDL model.

c) Napisati jednostavan testbenč pomoću kojega je moguće verifikovati ispravan rad projektovanih množača.

d) Izvršiti sintezu oba modela i uporediti dobijene rezultate u terminima potrebnih resursa i maksimalne brzine rada. Takođe, uporediti dobijene rezultate sa onima iz zadataka 5.2 i 5.3.

e) Za oba množača, koristeći podatke dobijene pod d), izračunati propusnu moć (broj operacija množenja koje se mogu izvršiti u jedinici vremena) i inicijalno kašnjenje. Uporediti dobijene vrednosti sa onima iz zadataka 5.2 i 5.3.

5.3. Optimizacija brojačkih naredbi

Kao što ranije već rečeno, transformacije petlji primenjuju se na petljama koje postoje unutar algoritma, sa ciljem da se postojeći paralelizam na nivou instrukcija (*Instruction Level Parallelism, ILP*) učini vidljivijim i iskoristi u onoj meri u kojoj to dopuštaju raspoloživi hardverski resursi (broj funkcionalnih jedinica, broj memorija, broj registarskih banki, njihova organizacija, itd.). Pomoću transformacije petlji po pravilu se povećavaju performanse projektovanog sistema (povećava se brzina obrade), pomoću paralelizacije izvršavanja tela petlje i efikasnim korišćenjem raspoloživih funkcionalnih i memorijskih jedinica. Najpoznatiji predstavnici ove grupe transformacija su: *Loop Rolling/Unrolling*, *Loop Tiling*, *Loop Strip-Mining*, *Loop Merging*, *Loop Distribution*, itd.

5.3.1. Razmotavanje petlje (*Loop Unrolling*)

Tehnika razmotavanja petlje jedna je od najčešće korišćenih tehnika za optimizaciju petlji, kada je izvršavanje petlje potrebno na optimalan način mapirati na raspoložive hardverske resurse platforme na kojoj se petlja izvršava. Osnovna ideja prilikom razmotavanja petlje je da se telo petlje replicira odgovarajući broj puta, pri čemu se vrednost iteratora petlje za svaku od razmotanih iteracija propagira unutar svih naredbi koje čine dotičnu iteraciju, kao što je prikazano na slici 5.7 u slučaju unutrašnje petlje po iteratoru *j*.

```
for (i=0; i < 3; i++)
{
   sum = 0;
   for (j=0; j < 3; j++)
      sum += a(i,j) * b(j,i);
   c(i) = sum;
}
```

a)

```
for (i=0; i < 3; i++)
{
   sum = 0;
   sum = a(i,0) * b(0,i);
   sum += a(i,1) * b(1,i);
   sum += a(i,2) * b(2,i);
   c(i) = sum;
}
```

b)

Slika 5.7. Razmotavanje petlje: a) originalni fragment koda b) fragment koda nakon potpunog razmotavanja unutrašnje petlje

Repliciranjem naredbi iz tela petlje otvaraju se mogućnosti za konkurentno izvršavanje većeg broja naredbi, na taj način ubrzavajući izvršavanje petlje. Stepen do koga se petlja može razmotati (*Loop Unrolling Factor*) zavisi od raspoloživog broja funkcionalnih jedinica, kao i od mogućnosti za istovremeni pristup većem broju podataka koji se nalaze smešteni u memoriji. Na slici 5.7 unutrašnja petlja (po iteratoru *j*) razmotana je u potpunosti, na taj način što smo telo petlje (naredbu *sum+= a[x][y] * b[y][x]*) replicirali ukupno tri puta. U transformisanom kodu sada imamo potencijal da ove tri replicirane naredbe izvršimo istovremeno, naravno ukoliko imamo na raspolaganju dovoljan broj sabirača i množača. U originalnom kodu bio nam je potreban samo jedan množač i sabirač, dok je nakon razmotavanja *j* petlje potrebno imati na raspolaganju tri množača i dva sabirača. Takođe, nakon razmotavanja *j* petlje moramo imati mogućnost da istovremeno iz memorije preuzmemo po tri vrednosti elemenata nizova *a* i *b*, što znači da se propusni opseg interfejsa između memorije i dela za obradu podataka mora utrostručiti. Kao što vidimo, razmotavanje petlje može značajno ubrzati njeno izvršavanje, ali po cenu značajnog povećanja potrebnih hardverskih resursa.

Pored toga što se na ovaj način povećava potencijal za iskorišćavanje paralelizma na nivou instrukcija, razmotavanje petlje takođe smanjuje vreme potrebno da se ažurira iterator petlje, jer se broj provera da li smo došli do kraja petlje eliminiše delimično (ako smo petlju samo delimično razmotali) ili u potpunosti (u slučaju potpunog razmotavanja petlje). Ovo se takođe može videti u primeru sa slike 5.7. Nakon razmotavanja *j* petlje u potpunosti, više nemamo potrebu za proverama da li je iterator *j* dostigao svoju maksimalnu vrednost.

U praksi vrlo često nije moguće razmotati petlju u potpunosti, već se radi takozvano delimično razmotavanje petlje (*Partial Loop Unrolling*). Prilikom delimičnog razmotavanja petlje sa faktorom *k*, telo petlje se replicira *k* puta, pri čemu je vrednost faktora razmotavanja *k* uvek manja od broja iteracija petlje koje je neophodno izvršiti. Na slici 5.8 prikazan je primer delimičnog razmotavanja petlje sa faktorom 2.

```
sum = 0;
for (i=0; i < n; i++)
   sum += a(i);
avg = sum/N;
```

a)

```
sum = 0;
for (i=0; i < n; i+=2)
{
   sum += a(i);
   if (i < N-1) then
      sum += a(i+1);
}
avg = sum/N;
```

b)

Slika 5.8. Razmotavanje petlje sa faktorom 2: a) originalni fragment koda b) fragment koda nakon delimičnog razmotavanja petlje sa faktorom 2

Primer sa slike 5.8 ilustruje način na koji je moguće izvršiti razmotavanje petlje u slučaju kada granice petlje nisu statičke (nisu poznate prilikom kompajliranja modela), kao što je to bio slučaj u primeru sa slike 5.7. U slučaju kada broj iteracija kroz petlju može dinamički da se menja (u toku rada sistema) prilikom razmotavanja petlje moraju se uključiti i dodatne kontrolne naredbe pomoću kojih se određuje da li je potrebno

izvršiti deo repliciranih tela petlje ili ne. U primeru sa slike 5.8 broj prolaza kroz petlju zavisi od vrednosti promenljive *n* i nije unapred poznat. Zbog toga on u opštem slučaju može da bude i paran i neparan, pa prilikom delimičnog razmotavanja petlje sa faktorom 2 moramo proveriti da li je potrebno izvršiti repliciranu naredbu iz originalnog tela pelja, *sum += a(x+1)*, prilikom svakog prolaska kroz delimično razmotanu petlju. Ovo je funkcija dodatne **if** naredbe iz tela razmotane petlje sa slike 5.8b. U slučaju kada je *n* neparno, prilikom poslednjeg prolaska kroz delimično razmotanu petlju ovu naredbu nije potrebno izvršiti. Ova provera predstavlja dodatni trošak u vremenu izvršavanja delimično razmotane petlje i samim tim umanjuje efekte ubrzanja rada koji se postižu razmotavanjem petlji. Upravo iz ovog razloga se prilikom delimičnog razmotavanja petlji sa unapred poznatim brojem prolaza *n*, faktor razmotavanja petlje *k* bira tako da količnik *n/k* bude ceo broj, ukoliko je to moguće. Na ovaj način se izbegava potreba za dodavanjem dodatnih kontrolnih naredbi u telo delimično razmotane petlje.

Naredni primer ilustruje način kako se performanse „*Naive*“ množača matrica, projektovanog ranije, mogu poboljšati korišćenjem tehnike razmotavanja petlje.

Primer 5.2: Množač matrica neoznačenih brojeva baziran na delimičnom razmotavanju petlje sa faktorom 2

U primeru 4.3 prikazan je postupak projektovanja hardverskog množača matrica, baziranog na „*Naive*“ algoritmu množenja dve matrice. Performanse projektovanog množača matrica procenjene su na $T_{naive} = n\_in \cdot p\_in \cdot m\_in + n\_ip \cdot p\_in + n\_in$ taktova, neophodnih za množenje dve matrice dimenzija *n_in*x*m_in* i *m_in*x*p_in*. U ovom primeru pokazaćemo kako se korišćenjem tehnike razmotavanja petlje ove performanse mogu poboljšati.

Originalni „*Naive*“ algoritam množenja matrica, na osnovu kojega je projektovan digitalni sistema za množenje matrica u primeru 4.3, prikazan je ponovo na slici 5.9.

```
for (i = 0; i < n; i++)
   for (j = 0; j < p; j++)
   {
      temp = 0;
      for (k = 0; k < m; k++)
         temp = temp + a(i,k)*b(k,j);
      c(i,j) = temp;
   }
return c;
```

Slika 5.9. „Naive“ algoritam za množenje dve matrice

Kao što vidimo sa slike 5.9, „*Naive*“ algoritam množenja matrica sastoji se od tri ugnježdjene petlje, pa se na njega može primeniti tehnika razmotavanja petlji. U našem slučaju delimično ćemo razmotati spoljašnju petlju, po iteratoru *i*, sa faktorom 2. Nakon ove transformacije „*Naive*“ algoritam ima oblik prikazan na slici 5.10.

```
for (i = 0; i < n; i+=2)
   for (j = 0; j < p; j++)
   {
      temp1 = 0;
      temp2 = 0;
      for (k = 0; k < m; k++)
      {
         temp1 = temp1 + a(i,k)*b(k,j);
         if (i < n-1) then
            temp2 = temp2 + a(i+1,k)*b(k,j);
      }
      c(i,j) = temp1;
      if (i < n-1) then
         c(i+1,j) = temp2;
   }
return c;
```

Slika 5.10. „Naive" algoritam za množenje dve matrice nakon delimičnog razmotavanja spoljašnje petlje sa faktorom 2

Pošto broj prolaza kroz *i* petlju nije unapred poznat, već se specificira od strane korisnika, da bismo obezbedili pravilan rad algoritma u svim slučajevima, potrebna su dve dodatne **if** naredbe, pomoću kojih se određuje da li je potrebno izračunati vrednost promenljive *temp2* i kasnije da li je vrednost promenljive *temp2* potrebno upisati kao novu vrednost elementa $c_{i+1,j}$ ili ne. Kako je broj prolaza kroz delimično razmotanu *i* petlju u ovom slučaju prepolovljen, možemo očekivati da će ova optimizovana verzija „*Naive*" algoritma množenja matrica biti dva puta brža od implementacije prikazane u primeru 4.3.

Međutim, obratite pažnju da su nam sada neophodna dva množača i dva sabirača kako bismo mogli da izvršimo sve konkurentne operacije koje postoje u telu unutrašnje petlje po iteratoru *k*. Ovo je dva puta veći broj funkcionalnih jedinica od broja potrebnog u implementaciji prikazanoj u primeru 4.3. Ono što je možda još važnije, analizom algoritma sa slike 5.10 možemo primetiti da sada u svakoj iteraciji *k* petlje takođe moramo imati na raspolaganju vrednosti dva elementa matrice *A*, $a_{i,k}$ i $a_{i+1,k}$. Takođe, u svakoj iteraciji *j* petlje sada potencijalno upisujemo nove vrednosti dva elementa matrice *C*, $c_{i,j}$ i $c_{i+1,j}$. Ovo znači da se propusni opsezi interfejsa ka memorijama *A* i *C* takođe moraju udvostručiti u odnosu na varijantu prikazanu u primeru 4.3.

Korak 1: Eliminacija naredbi ponavljanja iz algoritma

Kao i u prethodnim primerima, da bi smo mogli da nacrtamo ASMD dijagram koji odgovara algoritmu koji implementiramo u hardveru, prvo je neophodno zameniti tri **for** petlje sa odgovarajućim **if** and **goto** naredbama. Nakon ove zamene, modifikovani „*Naive*" algoritam množenja matrica opisan je pomoću sledećeg pseudo-koda.

```
      i = 0;
l1:   j = 0;
l2:      temp1 = 0;
         temp2 = 0;
```

```
                k = 0;
l3:             temp1 = temp1 + a_in(i,k)*b_in(k,j);
                if (i < n-1) then
                   temp2 = temp2 + a_in(i+1,k)*b_in(k,j);
                k = k + 1;
             if (k = m_in) then
                goto l3e;
             else
                goto l3;
l3e:
             c_out(i,j) = temp1;
             if (i < n_in-1) then
                c_out(i+1,j) = temp2;
             j = j + 1;
          if (j = p_in) then
             goto l2e;
          else
             goto l2;
l2e:
       i = i + 2;
       if (i < n_in) then
          goto stop;
       else
          goto l1;

stop:
       nop
```

*Slika 5.11. Modifikovani „Naive“ algoritam za množenje dve matrice, kod kojega su **for** naredbe zamenjene **if-goto** naredbama*

Korak 2: Definisanje interfejsa digitalnog sistema

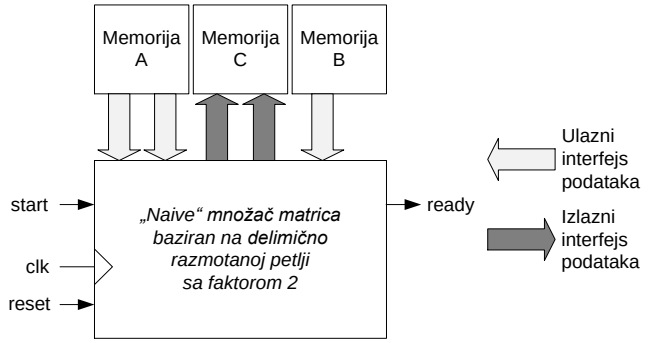
U ovom koraku potrebno je definisati četiri interfejsa digitalnog sistema koji se projektuje: ulazni interfejs podataka, izlazni interfejs podataka, komandni interfejs i statusni interfejs.

Ulazni interfejs podataka formira se tako što se u algoritmu koji se mapira u hardver identifikuju sve ulazne promenljive. U našem slučaju, analizom algoritma sa slike 5.11., možemo videti da postoji ukupno pet ulaznih promenljivih, matrice *A* i *B* (predstavljene pomoću promenljivih *a_in* i *b_in* u algoritmu, koje zapravo predstavljaju elemente tih matrica), kao i promenljive *n_in*, *p_in* i *m_in*, preko kojih su definisane dimenzije matrica *A* i *B*. Iako su *n_in*, *p_in* i *m_in* ulazne promenljive algoritma, logičnije je da one budu deo komandnog interfejsa, jer one ne predstavljaju podatke koje je potrebno obraditi već definišu veličine objekata koji se obrađuju pomoću projektovanog sistema.

Kako u postavci zadatka nije eksplicitno naglašeno koji su opsezi mogućih vrednosti elemenata matrica *A*, *B*, uvešćemo parametar *WIDTH*, preko kojega će biti moguće definisati širinu digitalne reči koja će se koristi za reprezentaciju njihovih vrednosti.

Takođe, pošto u postavci zadatka nisu definisane maksimalne dimenzije matrica *A* i *B*, uvešćemo parametar *SIZE*, koji će predstavljati maksimalnu dimenziju matrica *A* i *B*.

Kako su, zbog razmotavanja spoljašnje petlje, udvostručeni potrebni propusni opsezi interfejsa ka memorijama *A* i *C* (vidi algoritam sa slike 5.10), više se ne može koristiti arhitektura sistema prikazana na slici 4.21. Kako je u ovom slučaju u jednom taktu potrebno pročitati vrednosti dva elementa matrice *A* i upisati nove vrednosti dva elementa matrice *C*, ove memorije više ne mogu biti jednopristupne već se moraju koristiti dvopristupne memorije, kao što je prikazano na slici 5.12. Korišćenjem dvopristupnih memorija zapravo smo duplirali njihov propusni opseg.



Slika 5.12. Arhitektura sistema za implementaciju optimizovanog „Naive“ algoritma za množenje dve matrice

Memorijski interfejs za čitanje podataka iz dvopristupne memorije u kojoj je smeštena matrica *A* sada se sastoji iz sledećih portova:

- *a1_addr_o* – adresna magistrala porta 1, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
- *a1_data_i* – ulazna magistrala podataka porta 1, *std_logic_vector* tipa, širine *WIDTH* bita
- *a1_wr_o* – kontrolna magistrala porta 1, tipa *std_logic*.
- *a2_addr_o* – adresna magistrala porta 2, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
- *a2_data_i* – ulazna magistrala podataka porta 2, *std_logic_vector* tipa, širine *WIDTH* bita
- *a2_wr_o* – kontrolna magistrala porta 2, tipa *std_logic*.

Memorijski interfejs za čitanje podataka iz memorije u kojoj je smeštena matrica *B* sastoji se iz sledećih portova:

- *b_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita

- *b_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
- *b_wr_o* – kontrolna magistrala, tipa *std_logic*.

Izlazni interfejs podataka formira se tako što se u algoritmu koji se mapira u hardver identifikuju sve izlazne promenljive. U našem slučaju, postoji samo jedna izlazna promenljiva, matrica *C*. Kako će matrica *C* biti smeštena u drugoj dvopristupnoj memoriji, koristićemo još jedan memorijski interfejs, ovoga puta interfejs za upis podataka koji se sastoji iz sledećih portova:

- *c1_addr_o* – adresna magistrala porta 1, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
- *c1_data_o* – izlazna magistrala podataka porta 1, *std_logic_vector* tipa, širine 2**WIDTH*+*SIZE*-1 bita
- *c1_wr_o* – kontrolna magistrala porta 1, tipa *std_logic*.
- *c2_addr_o* – adresna magistrala porta 2, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
- *c2_data_o* – izlazna magistrala podataka porta 2, *std_logic_vector* tipa, širine 2**WIDTH*+*SIZE*-1 bita
- *c2_wr_o* – kontrolna magistrala porta 2, tipa *std_logic*.

NAPOMENA: Obratite pažnju da je širina izlazne magistrale data izrazom 2**WIDTH*+*SIZE*-1. Do ove vrednosti dolazi se na sledeći način. Vrednost elementa matrice *C* dobija se sabiranjem *m_in* proizvoda elemenata matrica *A* i *B*. Kako je svaki element matrica *A* i *B* predstavljen pomoću *WIDTH* bita, njihov proizvod mora biti reprezentovan sa 2**WIDTH* bita. Kako u najgorem slučaju možemo imati sumu od *SIZE* ovakvih vrednosti, a kako se prilikom sumiranja dva binarna broja širina rezultata povećava za jedan bit, maksimalni broj bita za koji se može povećati širina reči za reprezentaciju elementa matrice *C* iznosi dodatnih *SIZE*-1 bita.

Komandni interfejs nam omogućava kontrolišemo i upravljamo radom digitalnog sistema pomoću koji implementira željeni algoritam u hardveru. Kako je „*Naive*“ algoritam za množenje matrica vrlo jednostavan u pogledu njegovog korišćenja, možemo koristiti najjednostavniji mogući komandni interfejs, koji se sastoji od jednog 1-bitnog ulaznog porta, *start*. Kada je ulazni port *start* postavljen na jedinicu, digitalni sistem treba da započne operaciju množenja dve matrice. Dok je ulazni port *start* na nuli, digitalni sistem treba da bude u stanju mirovanja. Pored ovog porta, komandni interfejs sadrži i tri dodatna porta za definisanje vrednosti ulaznih promenljivih *n_in*, *p_in* i *m_in*. Najjednostavniji način je da se svakoj od ovih promenljivih pridruži po jedan višebitni *std_logic_vector* ulazni port, širine $\lceil ld(SIZE) \rceil$bita.

Statusni interfejs nam omogućava da dobijemo informacije o trenutnom stanju digitalnog sistema koji implementira željeni algoritam. U slučaju implementacije „*Naive*“ algoritma množenja matrica ponovo možemo koristiti najjednostavniji oblik statusnog interfejsa, koji se sastoji od jednog 1-bitnog izlaznog porta, *ready*. Ako je vrednost *ready* porta jednaka jedinici, to znači da je digitalni sistem spreman za izvršavanje nove komande (u našem primeru to znači da je spreman da započne množenje dve nove matrice). U slučaju kada je *ready* port postavljen na nulu, to predstavlja indikaciju da digitalni sistem nije u stanju da prihvati novu komandu (u našem slučaju to znači da je digitalni sistem zauzet množenjem dve matrice koje je još uvek u toku, tako da ne može započeti novu operaciju množenja).

Ovim je proces definisanja potrebnih interfejsa digitalnog sistema za hardversku implementaciju „*Naive*“ algoritma množenja matrica završen. Projektovani digitalni sistem imaće sledeće interfejse:

- Ulazni interfejs podataka – sastoji se iz sledećih portova:

  Portovi koji čine interfejs ka memoriji *A*:

  - *a1_addr_o* – adresna magistrala porta 1, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *a1_data_i* – ulazna magistrala podataka porta 1, *std_logic_vector* tipa, širine *WIDTH* bita
  - *a1_wr_o* – kontrolna magistrala porta 1, tipa *std_logic*
  - *a2_addr_o* – adresna magistrala porta 2, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *a2_data_i* – ulazna magistrala podataka porta 2, *std_logic_vector* tipa, širine *WIDTH* bita
  - *a2_wr_o* – kontrolna magistrala porta 2, tipa *std_logic*

  Portovi koji čine interfejs ka memoriji *B*:

  - *b_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *b_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
  - *b_wr_o* – kontrolna magistrala, tipa *std_logic*

- Izlazni interfejs podataka – sastoji se iz portova koji čine interfejs ka memoriji *C*:

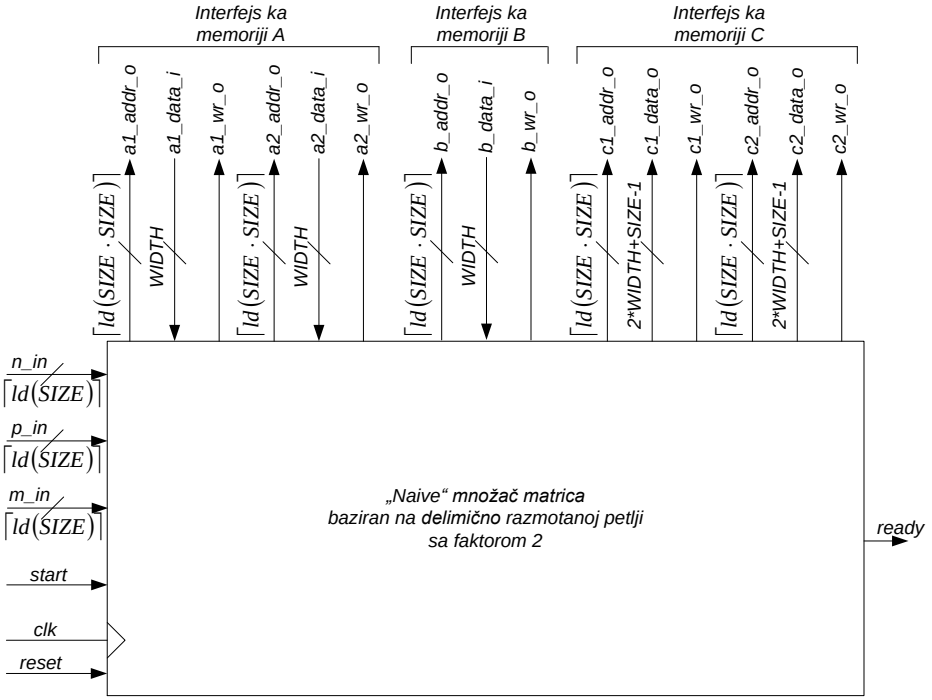
  - *c1_addr_o* – adresna magistrala porta 1, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *c1_data_o* – izlazna magistrala podataka porta 1, *std_logic_vector* tipa, širine 2**WIDTH*+*SIZE*-1 bita

- *c1_wr_o* – kontrolna magistrala porta 1, tipa *std_logic*
- *c2_addr_o* – adresna magistrala porta 2, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
- *c2_data_o* – izlazna magistrala podataka porta 2, *std_logic_vector* tipa, širine 2**WIDTH*+*SIZE*-1 bita
- *c2_wr_o* – kontrolna magistrala porta 2, tipa *std_logic*

- Komandni interfejs – sastoji se iz:
  - jednog 1-bitnog ulaznog porta, *start*, pomoću kojega se pokreće proces množenja matrica
  - tri ulazna porta, *n_in*, *p_in* i *m_in*, širine $\lceil ld(SIZE) \rceil$ bita, preko kojih se definišu tekuće dimenzije matrica *A* i *B* koje je potrebno pomnožiti

- Statusni interfejs – sastoji se iz jednog 1-bitnog izlaznog porta, *ready*

Pored ovih portova, digitalni sistem koji projektujemo mora posedovati i standardne portove za dovođenje klok i reset signala, *clk* i *reset*. Na slici 5.13. prikazan je kompletan interfejs digitalnog sistema koji implementira „*Naive*“ algoritam množenja dve matrice baziran na delimičnom razmotavanju spoljašnje petlje sa faktorom 2.



Slika 5.13. Interfejs digitalnog sistema koji implementira „Naive“ algoritam množenja dve matrice baziran na delimično razmotanoj spoljašnjoj petlji

Korak 3: Projektovanje *controlpath* modula

Uzimajući u obzir i način na koji je organizovan memorijski podsistem (vidi sliku 5.12) algoritam sa slike 5.11 lako se može prevesti u odgovarajući ASMD dijagram, koji je prikazan na slici 5.14.

ASMD dijagram ima četiri stanja. U *idle* stanju, ASMD proverava *start* signal. Ako je on aktivan, ASMD inicijalizuje brojački registar *i* (koja je zapravo brojač *i* iz prve **for** petlje originalnog algoritma), a zatim prelazi u *l1* stanje. U *l1* stanju vrši se inicijalizacija drugog brojačkog registra *j* (koja je zapravo brojač *j* iz druge **for** petlje originalnog algoritma), a zatim se prelazi u stanje *l2*.

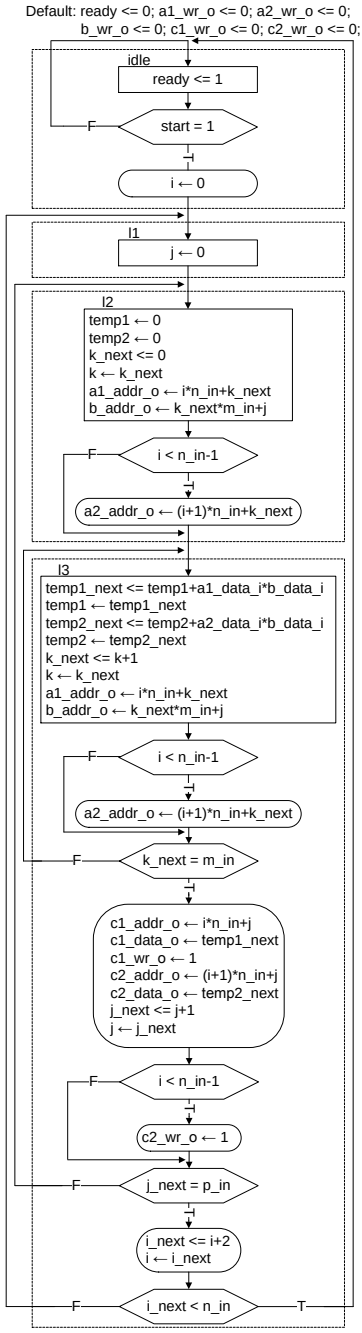
U stanju *l2* inicijalizuje se poslednji brojački registar *k* (koja je zapravo brojač *i* iz prve **for** petlje originalnog algoritma), inicijalizuju se registri *temp1* i *temp2* u kojima će biti smeštene privremene vrednosti elemenata matrice *C*, i izračunavaju se početne adrese elemenata matrica *A* i *B*, a nakon toga se prelazi u stanje *l3*. Obratite pažnju da je ovaj put potrebno izračunati početne adrese za dva elementa matrice *A*, pri čemu je potrebno voditi računa da li je adresa koja se prosleđuje na portu 2 memorije *A* validna.

U stanju *l3* se na tekuće vrednosti registara *temp1* i *temp2* dodaju vrednosti izračunatih proizvoda tekućih elemenata matrica *A* i *B* i izračunavaju adrese narednih elemenata matrica *A* i *B* koji će biti korišćeni u narednoj iteraciji. Zatim se inkrementuje vrednost brojačkog registra *k* i proverava da li je ona dostigla vrednost *m_in*.

U slučaju da nije, ASMD ostaje u stanju *l3*, a ako jeste vrši se upis izračunatih vrednosti tekuća dva elementa matrice *C* na odgovarajuća mesta u memoriji, jer je proces njihovog računanja kompletiran. Opet je potrebno proveriti da li je potrebno upisati vrednost elementa $c_{i+1,j}$ matrice *C*, jer on možda izlazi iz dozvoljenog opsega prve dimenzije matrice *C*, definisane preko ulaznog porta *n_in*. Takođe se inkrementuje i vrednost brojačkog registra *j* i proverava da li je dostigla vrednost *p_in*.

U slučaju da nije, vraćamo se u stanje *l2*, i započinjemo računanje vrednosti sledećeg elementa u tekućoj vrsti matrice *C*. Ako je vrednost registra *j* jednaka *p_in*, inkrementuje se vrednost brojačkog registra *i* (ali ovoga puta sa faktorom 2), i proverava da li je ona jednaka vrednosti *n_in*.

Ukoliko jeste, postupak množenja matrica je završen i ASDM se vraća u *idle* stanje. Ukoliko vrednost *n_in* još uvek nije dostignuta, ASMD se vraća u stanje *l1*, gde započinje računanje vrednosti elemenata matrice *C* iz sledeće dve vrste.



Slika 5.14. ASMD dijagram „Naive“ množača matrica sa delimičnim razmotavanjem spoljašnje petlje sa faktorom 2

Nakon što smo nacrtali ASMD dijagram, praktično smo završili projektovanje *controlpath* modula našeg dizajna. Ono što je preostalo jeste da se izvrši projektovanje *datapath* modula.

Korak 4: Projektovanje *datapath* modula

Prvi korak prilikom projektovanja *datapath* modula jeste da identifikujemo sve registre koji su prisutni u sistemu i njima asocirane RT operacije. U „*Naive*“ algoritmu za množenje matrica sa slike 5.11 postoji pet unutrašnjih promenljivih, *i, j, k, temp1* i *temp2*. Svakoj od njih asociraćemo po jedan registar. Što se tiče veličina ovih registara, registri i, *j* i *k* su širine $\lceil ld(SIZE) \rceil$bita, dok su registri *temp1* i *temp2* širine 2**WIDTH+SIZE-1* bita, jer se u njih smešta rezultat akumulisanih proizvoda elemenata matrica *A* i *B*, tokom računanja vrednosti elemenata matrice *C*.

Nakon što smo odredili broj i veličinu registara u sistemu, možemo pristupiti pravljenju liste svih RT operacija koje postoje u projektovanom ASMD dijagramu. Analizom ASMD dijagrama sa slike 5.14 dolazimo do sledeće liste RT operacija:

- u *idle* stanju: $i \leftarrow 0, j \leftarrow j, k \leftarrow k, temp1 \leftarrow temp1, temp2 \leftarrow temp2$
- u *l1* stanju: $i \leftarrow i, j \leftarrow 0, k \leftarrow k, temp1 \leftarrow temp1, temp2 \leftarrow temp2$
- u *l2* stanju: $i \leftarrow i, j \leftarrow j, k \leftarrow 0, temp1 \leftarrow 0, temp2 \leftarrow 0$
- u *l3* stanju: $i \leftarrow i, i \leftarrow i+1, j \leftarrow j, j \leftarrow j+1, k \leftarrow k+1,$
  $temp1 \leftarrow temp1+a1\_data\_i*b\_data\_i,$
  $temp2 \leftarrow temp2+a2\_data\_i*b\_data\_i,$

Sledeći korak jeste da se svakom od registara pridruže asocirane RT operacije.

RT operacije kojima je ciljni registar *i*:

1. u *idle* stanju: $i \leftarrow 0$
2. u *l1* stanju: $i \leftarrow i$
3. u *l2* stanju: $i \leftarrow i$
4. u *l3* stanju: $i \leftarrow i+1$ (ako je *k_next* = *m_in* i *j_next* = *p_in*) ,
   $i \leftarrow i$ (inače)

RT operacije kojima je ciljni registar *j*:

1. u *idle* stanju: $j \leftarrow j$
2. u *l1* stanju: $j \leftarrow 0$
3. u *l2* stanju: $j \leftarrow j$

4. u *l3* stanju: $j \leftarrow j$ (ako je $k\_next < m\_in$);
   $j \leftarrow j+1$ (ako je $k\_next = m\_in$)

RT operacije kojima je ciljni registar *k*:

1. u *idle* stanju: $k \leftarrow k$
2. u *l1* stanju: $k \leftarrow k$
3. u *l2* stanju: $k \leftarrow 0$
4. u *l3* stanju: $k \leftarrow k + 1$

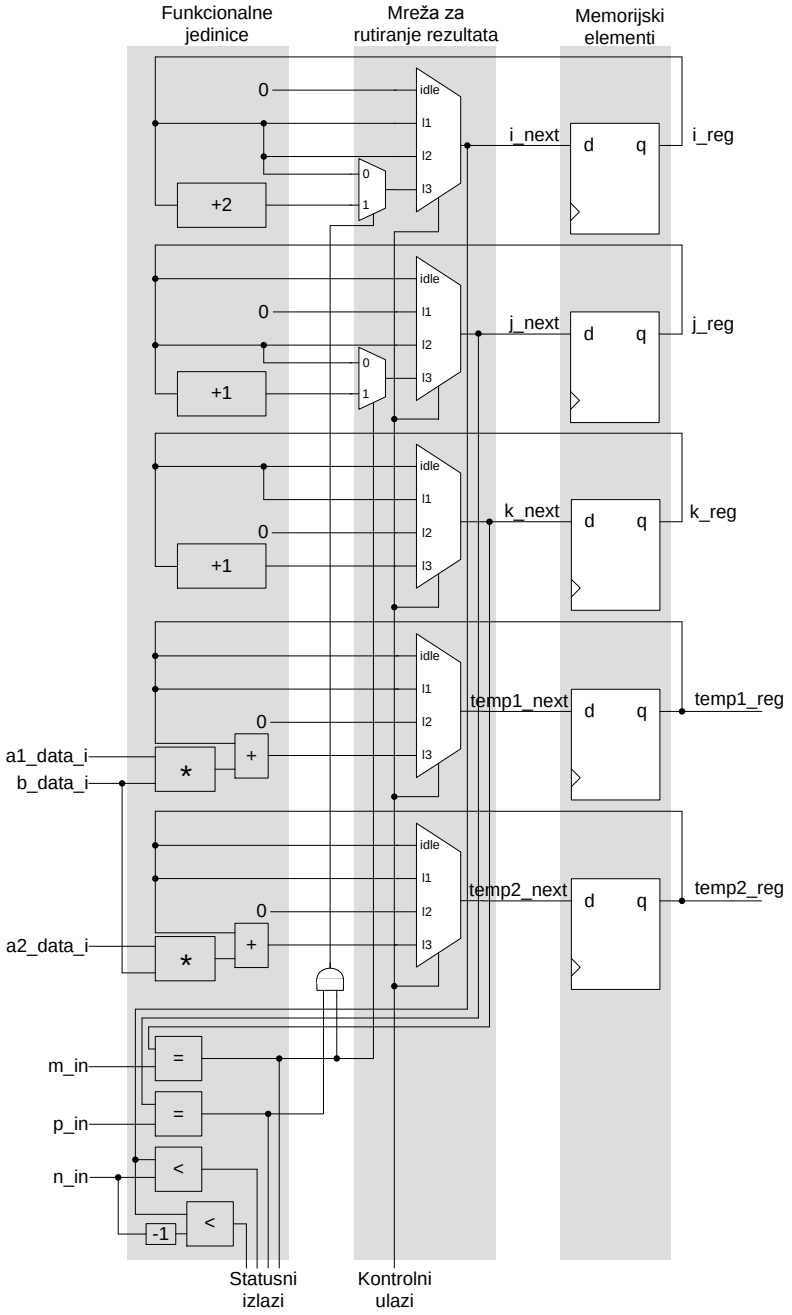
RT operacije kojima je ciljni registar *temp1*:

1. u *idle* stanju: $temp1 \leftarrow temp1$
2. u *l1* stanju: $temp1 \leftarrow temp1$
3. u *l2* stanju: $temp1 \leftarrow 0$
4. u *l3* stanju: $temp1 \leftarrow temp1+a1\_data\_i*b\_data\_i$

RT operacije kojima je ciljni registar *temp2*:

1. u *idle* stanju: $temp2 \leftarrow temp2$
2. u *l1* stanju: $temp2 \leftarrow temp2$
3. u *l2* stanju: $temp2 \leftarrow 0$
4. u *l3* stanju: $temp2 \leftarrow temp2+a2\_data\_i*b\_data\_i$

Na osnovu ovih informacija moguće je formirati delove *datapath* modula koji su asocirani svakom od registara u sistemu. Kompletna struktura *datapath* modula prikazana je na slici 5.15.



Slika 5.15. Datapath modul „Naive“ množača matrica sa delimično razmotanom spoljašnjom petljom sa faktorom 2

Korak 5: Pisanje HDL modela

Nakon što smo projektovali *datapth* i *controlpath* module, poslednji korak predstavlja pisanje odgovarajućeg HDL modela čitavog digitalnog sistema koji implementira „*Naive*“ algoritam množenja matrica sa delimično razmotanom spoljašnjm petljom u hardveru. Kao što je to bio slučaj u ranijim primerima i ovaj model se može napisati na različite načine. U nastavku je prikazan VHDL model projektovanog digitalnog sistema za implementaciju „*Naive*“ algoritma množenja dve matrice koji modeluje *datapath* i *controlpath* module unutar istog entiteta, koristeći dvoprocesni stil modelovanja.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.utils_pkg.all;

entity matrix_mult_lu2 is
  generic (
    WIDTH: integer := 8;
    SIZE: integer := 3
        );
  port (
    ---------- Clocking and reset interface ----------
    clk:                in std_logic;
    reset:              in std_logic;
    ---------- Input data interface ----------
    -- Matrix A memory interface, port 1
    a1_addr_o:          out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    a1_data_i:          in std_logic_vector(WIDTH-1 downto 0);
    a1_wr_o:            out std_logic;
    -- Matrix A memory interface, port 2
    a2_addr_o:          out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    a2_data_i:          in std_logic_vector(WIDTH-1 downto 0);
    a2_wr_o:            out std_logic;
    -- Matrix B memory interface
    b_addr_o:           out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    b_data_i:           in std_logic_vector(WIDTH-1 downto 0);
    b_wr_o:             out std_logic;
    -- Matrix dimensions definition interface
    n_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
    p_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
    m_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
    ---------- Output data interface ----------
    -- Matrix C memory interface, port 1
    c1_addr_o:          out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    c1_data_o:          out std_logic_vector(2*WIDTH+SIZE-1 downto 0);
    c1_wr_o:            out std_logic;
    -- Matrix C memory interface, port 2
    c2_addr_o:          out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    c2_data_o:          out std_logic_vector(2*WIDTH+SIZE-1 downto 0);
    c2_wr_o:            out std_logic;
    ---------- Command interface ----------
    start:              in std_logic;
    ---------- Status interface ----------
    ready:              out std_logic);
end entity;
```

```
architecture two_seg_arch of matrix_mult_lu2 is
   type state_type is (idle, l1, l2, l3);
   signal state_reg, state_next: state_type;
   signal i_reg, i_next: unsigned(log2c(SIZE)-1 downto 0);
   signal j_reg, j_next: unsigned(log2c(SIZE)-1 downto 0);
   signal k_reg, k_next: unsigned(log2c(SIZE)-1 downto 0);
   signal temp1_reg, temp1_next: unsigned(2*WIDTH+SIZE-1 downto 0);
   signal temp2_reg, temp2_next: unsigned(2*WIDTH+SIZE-1 downto 0);
begin
   -- State and data registers
   process (clk, reset)
   begin
     if reset = '1' then
       state_reg <= idle;
       i_reg <= (others => '0');
       j_reg <= (others => '0');
       k_reg <= (others => '0');
       temp1_reg <= (others => '0');
       temp2_reg <= (others => '0');
     elsif (clk'event and clk = '1') then
       state_reg <= state_next;
       i_reg <= i_next;
       j_reg <= j_next;
       k_reg <= k_next;
       temp1_reg <= temp1_next;
       temp2_reg <= temp2_next;
     end if;
   end process;

   -- Combinatorial circuits
   process (state_reg, start, a1_data_i, a2_data_i, b_data_i, i_reg, j_reg, k_reg,
            temp1_reg, temp2_reg, i_next, j_next, k_next, temp1_next, temp2_next)
     variable conv_v: std_logic_vector(2*log2c(SIZE)-1 downto 0);
   begin
     -- Default assignments
     i_next <= i_reg;
     j_next <= j_reg;
     k_next <= k_reg;
     temp1_next <= temp1_reg;
     temp2_next <= temp2_reg;
     a1_addr_o <= (others => '0');
     a2_addr_o <= (others => '0');
     a1_wr_o <= '0';
     a2_wr_o <= '0';
     b_addr_o <= (others => '0');
     b_wr_o <= '0';
     c1_addr_o <= (others => '0');
     c2_addr_o <= (others => '0');
     c1_wr_o <= '0';
     c2_wr_o <= '0';
     ready <= '0';

     case state_reg is
       when idle =>
         ready <= '1';
         if start = '1' then
           i_next <= to_unsigned(0, log2c(SIZE));
           state_next <= l1;
```

```
  else
    state_next <= idle;
  end if;


when l1 =>
  j_next <= to_unsigned(0, log2c(SIZE));
  state_next <= l2;

when l2 =>
  temp1_next <= to_unsigned(0, 2*WIDTH+SIZE);
  temp2_next <= to_unsigned(0, 2*WIDTH+SIZE);
  k_next <= to_unsigned(0, log2c(SIZE));
  conv_v := std_logic_vector(i_reg*unsigned(n_in)+k_next);
  a1_addr_o <= conv_v(a1_addr_o'range);
  if (i_reg < unsigned(n_in) - 1) then
    conv_v := std_logic_vector((i_reg+1)*unsigned(n_in)+k_next);
    a2_addr_o <= conv_v(a2_addr_o'range);
  end if;
  conv_v := std_logic_vector(k_next*unsigned(m_in)+j_reg);
  b_addr_o <= conv_v(b_addr_o'range);
  state_next <= l3;

when l3 =>
  temp1_next <= temp1_reg + unsigned(a1_data_i)*unsigned(b_data_i);
  temp2_next <= temp2_reg + unsigned(a2_data_i)*unsigned(b_data_i);
  k_next <= k_reg + 1;
  conv_v := std_logic_vector(i_reg*unsigned(n_in)+k_next);
  a1_addr_o <= conv_v(a1_addr_o'range);
  if (i_reg < unsigned(n_in) - 1) then
    conv_v := std_logic_vector((i_reg+1)*unsigned(n_in)+k_next);
    a2_addr_o <= conv_v(a2_addr_o'range);
  end if;
  conv_v := std_logic_vector(k_next*unsigned(m_in)+j_reg);
  b_addr_o <= conv_v(b_addr_o'range);
  if (k_next = unsigned(m_in)) then
    conv_v := std_logic_vector(i_reg*unsigned(n_in)+j_reg);
    c1_addr_o <= conv_v(c1_addr_o'range);
    c1_data_o <= std_logic_vector(temp1_next);
    c1_wr_o <= '1';
    conv_v := std_logic_vector((i_reg+1)*unsigned(n_in)+j_reg);
    c2_addr_o <= conv_v(c2_addr_o'range);
    c2_data_o <= std_logic_vector(temp2_next);
    if (i_reg < unsigned(n_in) - 1) then
      c2_wr_o <= '1';
    end if;
    j_next <= j_reg + 1;
    if (j_next = unsigned(p_in)) then
      i_next <= i_reg + 2;
      if (i_next < unsigned(n_in)) then
        state_next <= l1;
      else
        state_next <= idle;
      end if;
    else
      state_next <= l2;
    end if;
  else
    state_next <= l3;
```

```
        end if;
      end case;
    end process;
end two_seg_arch;
```

NAPOMENA: *utils_pkg* paket, koji koristi prethodni model isti je kao i u primeru 4.3, pa ovde neće biti prikazan njegov sadržaj.

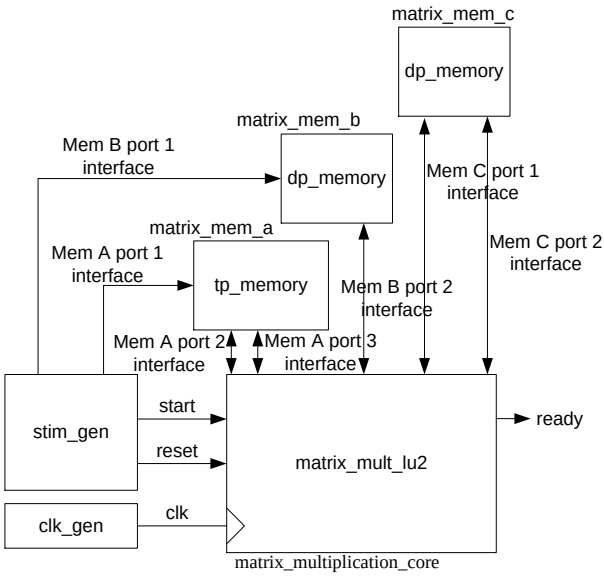
Na kraju, procenimo brzinu rada „*Naive*“ množača matrica baziranog na delimičnom razmotavanju spoljašnje petlje sa faktorom 2. Analizom ASMD dijagrama sa slike 5.14, može se zaključiti da računanje vrednosti svakog elementa matrice *C* zahteva *m_in* taktova (jer u svakom taktu računamo dva proizvoda odgovarajućih elementata matrica *A* i *B* i dodajemo po jedan od njih na tekuće vrednosti akumuliranih suma, promenljive *temp1* i *temp2*). Nakon završetka računanja vrednosti tekuća dva elementa matrice *C*, pre započinjanja računanja vrednosti naredna dva elementa troši se još jedan takt (ponovo se vraćamo u *l2* stanje kako bismo inicijalizovali brojač *k*). Dodatni takt troši se svaki put kada započinjenom računanje vrednosti elemenata matrice *C* iz sledeće dve vrste. Međutim, kako smo razmotali spoljašnju petlju sa faktorom 2, u svakom prolazu kroz nju računamo po dve vrednosti elemenata matrice *C* istovremeno! Imajući sve ovo u vidu, potreban broj taktova da se pomoću projektovanog „*Naive*“ množača matrica sa delimičnim razmotavanjem spoljašnje petlje sa faktorom 2 pomnože dve matrice dimenzija *n*x*m* i *m*x*p* iznosi

$$T_{naive\_lu2} = \frac{n\_in}{2} \cdot \left(p\_in \cdot \left(m\_in + 1\right) + 1\right) = \frac{T_{naive}}{2}$$

taktova. Na osnovu prethodnog izraza možemo zaključiti da smo delimičnim razmotavanjem spoljašnje petlje originalnog „*Naive*“ algoritma prepolovili vreme potrebno da se pomnože dve matrice, što je i bilo očekivano.

Korak 6: Verifikacija razvijenog HDL modela

Radi kompletnosti rešenja, u nastavku će biti prikazano i verifikaciono okruženje koje bi moglo da se iskoristi za verifikaciju „*Naive*“ množač matrica baziranog na delimičnom razmotavanju spoljašnje petlje sa faktorom 2. Struktura korišćenog verifikacionog okruženja može se videti na slici 5.15, a sam VHDL model verifikacionog okruženja prikazan je u nastavku.



Slika 5.15. Struktura verifikacionog okruženja "Naive" množača matrica sa delimičnim razmotavanjem petlje

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use work.utils_pkg.all;

entity matrix_mult_lu2_tb is
end entity;

architecture beh of matrix_mult_lu2_tb is
  constant DATA_WIDTH_c: integer := 8;
  constant SIZE_c: integer := 5;
  constant N_c: integer := 3;
  constant M_c: integer := 3;
  constant P_c: integer := 3;

  type mem_t is array (0 to SIZE_c*SIZE_c-1) of
                                    std_logic_vector(DATA_WIDTH_c-1 downto 0);

  constant MEM_A_CONTENT_c: mem_t :=
                                  (conv_std_logic_vector(1, DATA_WIDTH_c),
                                   conv_std_logic_vector(2, DATA_WIDTH_c),
                                   conv_std_logic_vector(3, DATA_WIDTH_c),
                                   conv_std_logic_vector(4, DATA_WIDTH_c),
                                   conv_std_logic_vector(5, DATA_WIDTH_c),
                                   conv_std_logic_vector(6, DATA_WIDTH_c),
                                   conv_std_logic_vector(7, DATA_WIDTH_c),
                                   conv_std_logic_vector(8, DATA_WIDTH_c),
                                   conv_std_logic_vector(9, DATA_WIDTH_c),
```

```
                                        others => (others => '0'));
  constant MEM_B_CONTENT_c: mem_t :=
                                       (conv_std_logic_vector(10, DATA_WIDTH_c),
                                        conv_std_logic_vector(11, DATA_WIDTH_c),
                                        conv_std_logic_vector(12, DATA_WIDTH_c),
                                        conv_std_logic_vector(13, DATA_WIDTH_c),
                                        conv_std_logic_vector(14, DATA_WIDTH_c),
                                        conv_std_logic_vector(15, DATA_WIDTH_c),
                                        conv_std_logic_vector(16, DATA_WIDTH_c),
                                        conv_std_logic_vector(17, DATA_WIDTH_c),
                                        conv_std_logic_vector(18, DATA_WIDTH_c),
                                        others => (others => '0'));

  signal clk_s: std_logic;
  signal reset_s: std_logic;

  signal n_in_s: std_logic_vector(log2c(SIZE_c)-1 downto 0);
  signal m_in_s: std_logic_vector(log2c(SIZE_c)-1 downto 0);
  signal p_in_s: std_logic_vector(log2c(SIZE_c)-1 downto 0);

  -- Matrix A memory interface
  signal mem_a_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal mem_a_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal mem_a_wr_s: std_logic;
  signal a1_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal a1_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal a1_wr_s: std_logic;
  signal a2_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal a2_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal a2_wr_s: std_logic;
  -- Matrix B memory interface
  signal mem_b_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal mem_b_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal mem_b_wr_s: std_logic;
  signal b_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal b_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal b_wr_s: std_logic;
  -- Matrix C memory interface
  signal c1_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal c1_data_out_s: std_logic_vector(2*DATA_WIDTH_c+SIZE_c-1 downto 0);
  signal c1_wr_s: std_logic;
  signal c2_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal c2_data_out_s: std_logic_vector(2*DATA_WIDTH_c+SIZE_c-1 downto 0);
  signal c2_wr_s: std_logic;

  signal start_s: std_logic := '0';
  signal ready_s: std_logic;
begin

  clk_gen: process
  begin
    clk_s <= '0', '1' after 100 ns;
    wait for 200 ns;
  end process;

  stim_gen: process
  begin
    -- Apply system level reset
    reset_s <= '1';
```

```
  wait for 500 ns;
  reset_s <= '0';

  wait until falling_edge(clk_s);
  -- Load the data into the matrix A memory
  mem_a_wr_s <= '1';
  for i in 0 to N_c-1 loop
    for j in 0 to M_c-1 loop
      mem_a_addr_s <= conv_std_logic_vector(i*M_c+j, mem_a_addr_s'length);
      mem_a_data_in_s <= MEM_A_CONTENT_c(i*M_c+j);
      wait until falling_edge(clk_s);
    end loop;
  end loop;
  mem_a_wr_s <= '0';

  -- Load the data into the matrix B memory
  mem_b_wr_s <= '1';
  for i in 0 to M_c-1 loop
    for j in 0 to P_c-1 loop
      mem_b_addr_s <= conv_std_logic_vector(i*P_c+j, mem_b_addr_s'length);
      mem_b_data_in_s <= MEM_B_CONTENT_c(i*P_c+j);
      wait until falling_edge(clk_s);
    end loop;
  end loop;
  mem_b_wr_s <= '0';

  -- Start the multiplication process
  start_s <= '1';
  wait until falling_edge(clk_s);
  start_s <= '0';

  -- Wait until matrix multiplication module signals that the operation has been
  -- complted
  wait until ready_s = '1';

  -- End stimulus generation
  wait;
end process;

-- Matrix A memory
matrix_a_mem: entity work.tp_memory(beh)
  generic map (
    WIDTH           => DATA_WIDTH_c,
    SIZE            => SIZE_c)
  port map (
    clk             => clk_s,
    reset           => reset_s,
    p1_addr_i       => mem_a_addr_s,
    p1_data_i       => mem_a_data_in_s,
    p1_data_o       => open,
    p1_wr_i         => mem_a_wr_s,
    p2_addr_i       => a1_addr_s,
    p2_data_i       => (others => '0'),
    p2_data_o       => a1_data_in_s,
    p2_wr_i         => a1_wr_s,
    p3_addr_i       => a2_addr_s,
    p3_data_i       => (others => '0'),
    p3_data_o       => a2_data_in_s,
    p3_wr_i         => a2_wr_s);
```

```
-- Matrix B memory
matrix_b_mem: entity work.dp_memory(beh)
  generic map (
    WIDTH              => DATA_WIDTH_c,
    SIZE               => SIZE_c)
  port map (
    clk                => clk_s,
    reset              => reset_s,
    p1_addr_i          => mem_b_addr_s,
    p1_data_i          => mem_b_data_in_s,
    p1_data_o          => open,
    p1_wr_i            => mem_b_wr_s,
    p2_addr_i          => b_addr_s,
    p2_data_i          => (others => '0'),
    p2_data_o          => b_data_in_s,
    p2_wr_i            => b_wr_s);

-- Matrix C memory
matrix_c_mem: entity work.dp_memory(beh)
  generic map (
    WIDTH              => 2*DATA_WIDTH_c+SIZE_c,
    SIZE               => SIZE_c)
  port map (
    clk                => clk_s,
    reset              => reset_s,
    p1_addr_i          => c1_addr_s,
    p1_data_i          => c1_data_out_s,
    p1_data_o          => open,
    p1_wr_i            => c1_wr_s,
    p2_addr_i          => c2_addr_s,
    p2_data_i          => c2_data_out_s,
    p2_data_o          => open,
    p2_wr_i            => c2_wr_s);

-- DUT
n_in_s <= conv_std_logic_vector(N_c, log2c(SIZE_c));
p_in_s <= conv_std_logic_vector(P_c, log2c(SIZE_c));
m_in_s <= conv_std_logic_vector(M_c, log2c(SIZE_c));
matrix_multiplication_core: entity work.matrix_mult_lu2(two_seg_arch)
  generic map (
    WIDTH              => DATA_WIDTH_c,
    SIZE               => SIZE_c)
  port map (
     ---------- Clocking and reset interface ----------
    clk                => clk_s,
    reset              => reset_s,
      ---------- Input data interface ----------
    -- Matrix A memory interface
    a1_addr_o          => a1_addr_s,
    a1_data_i          => a1_data_in_s,
    a1_wr_o            => a1_wr_s,
    a2_addr_o          => a2_addr_s,
    a2_data_i          => a2_data_in_s,
    a2_wr_o            => a2_wr_s,
    -- Matrix B memory interface
    b_addr_o           => b_addr_s,
    b_data_i           => b_data_in_s,
    b_wr_o             => open,
```

```
    -- Matrix dimensions definition interface
    n_in            => n_in_s,
    p_in            => p_in_s,
    m_in            => m_in_s,
     ------------------ Output data interface ------------------
    -- Matrix C memory interface
    c1_addr_o       => c1_addr_s,
    c1_data_o       => c1_data_out_s,
    c1_wr_o         => c1_wr_s,
    c2_addr_o       => c2_addr_s,
    c2_data_o       => c2_data_out_s,
    c2_wr_o         => c2_wr_s,
     --------------------- Command interface --------------------
    start           => start_s,
     ---------------------- Status interface ----------------------
    ready           => ready_s);
end architecture beh;
```

Prikazano verifikaciono okruženje koristi dve instance dvopristupne memorije, za smeštanje vrednosti elemenata ulazne matrice *B* i izlazne matrice *C*, čiji VHDL model je prikazan u primeru 4.3, ali takođe koristi i jednu instancu tropristupne memorije za smeštanje vrednosti elemenata ulazne matrice *A*. Radi kompletnosti, u nastavku je prikazan i model ove tropristupne memorije.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

use work.utils_pkg.all;

entity tp_memory is
  generic (
    WIDTH: integer := 8;
    SIZE: integer := 3
     );
  port (
    clk:            in std_logic;
    reset:          in std_logic;
    -- Port 1 interface
    p1_addr_i:      in std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    p1_data_i:      in std_logic_vector(WIDTH-1 downto 0);
    p1_data_o:      out std_logic_vector(WIDTH-1 downto 0);
    p1_wr_i:        n std_logic;
    -- Port 2 interface
    p2_addr_i:      in std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    p2_data_i:      in std_logic_vector(WIDTH-1 downto 0);
    p2_data_o:      out std_logic_vector(WIDTH-1 downto 0);
    p2_wr_i:        in std_logic;
    -- Port 3 interface
    p3_addr_i:      in std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    p3_data_i:      in std_logic_vector(WIDTH-1 downto 0);
    p3_data_o:      out std_logic_vector(WIDTH-1 downto 0);
    p3_wr_i:        in std_logic);
end entity;
```

```
architecture beh of tp_memory is
  type mem_t is array (0 to 2**log2c(SIZE*SIZE)-1) of std_logic_vector(WIDTH-1 downto 0);
  signal mem_s: mem_t;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (reset = '1') then
      mem_s <= (others => (others => '0'));
    else
      if (p1_wr_i = '1') then
        mem_s(conv_integer(p1_addr_i)) <= p1_data_i;
      end if;
      if (p2_wr_i = '1') then
        mem_s(conv_integer(p2_addr_i)) <= p2_data_i;
      end if;
      if (p3_wr_i = '1') then
        mem_s(conv_integer(p3_addr_i)) <= p3_data_i;
      end if;
        p1_data_o <= mem_s(conv_integer(p1_addr_i));
        p2_data_o <= mem_s(conv_integer(p2_addr_i));
        p3_data_o <= mem_s(conv_integer(p3_addr_i));
    end if;
    end if;
  end process;
end architecture beh;
```

Na slici 5.16 prikazani su talasni oblici karakterističnih signala, koji predstavljaju rezultat simulacije verifikacionog okruženja i modela „*Naive*“ množača matrica baziranog na delimičnom razmotavanju spoljašnje petlje sa faktorom 2, u slučaju množenja kvadratnih matrica *A* i *B*, dimenzija 3x3. Matrice *A* i *B* definisane su na sledeći način

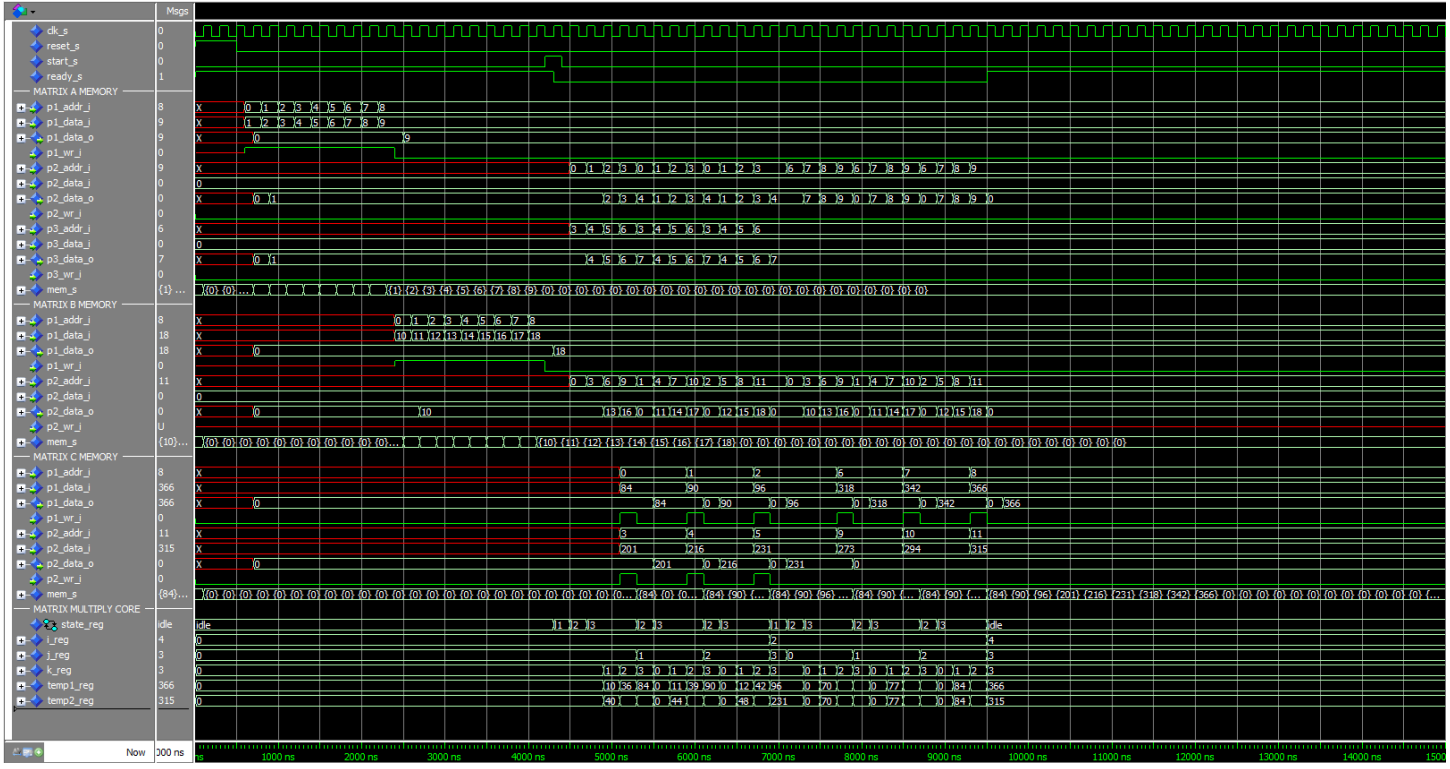
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \qquad B = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}.$$

Rezultat množenja ove dve matrice trebalo bi da bude sledeća matrica

$$C = A \cdot B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix} = \begin{bmatrix} 84 & 90 & 96 \\ 201 & 216 & 231 \\ 318 & 342 & 366 \end{bmatrix}.$$

Inspekcijom konačnih vrednosti unutrašnjih signala *mem_s* sva tri memorijska modula („*Matrix A Memory*“, „*Matrix B Memory*“ i „*Matrix C Memory*“) sa slike 5.16, možemo se uveriti da projektovani digitalni sistem korektno implementira „*Naive*“ algoritam množenja dve matrice. Obratite takođe pažnju na računanje po dve vrednosti elemenata iz sukcesivnih kolona matrice *C*, što se lepo može videti na slici 5.16. Takođe, kako je

broj prolaza kroz spoljašnju petlju u ovom slučaju jednak 3, u poslednjem prolazu se računaju vrednosti samo elemenata iz jedne vrste (koristi se samo port 1 memorije za smeštanje matrice *C*, dok je port 2 neaktivan).



Slika 5.16. Rezultat simulacije rada „Naive“ množača matrica sa delimičnim razmotavanjem spoljašnje petlje sa faktorom 2, na primeru množenja dve matrice dimenzija 3x3

Zadaci za vežbu

Zadatak 5.5.

Primenom RT metodologije projektovati digitalni sistem koji realizuje algoritam „*Naive*“ množača matrica kod koga je spoljašnja petlja delimično razmotana sa faktorom 4. Napisati i osnovno verifikaciono okruženje pomoću kojega se može verifikovati korektan rad projektovanog sistema.

Zadatak 5.6.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za određivanje indeksa elementa niza brojeva *x* koji je najbliži srednjoj vrednosti članova niza, prikazanog u nastavku. Prilikom implementacije obe petlje je potrebno razmotati sa faktorom 2.

```
srv = 0;
for (i=0; i<n_in-1; i++)
   srv = srv + x_in[i];
srv = srv/n_in;
pos = 0;
min_dif = abs(x_in[0]-srv);
for (i=1; i<n_in; i++)
{
   dif = abs(x_in[i]-srv);
   if (dif < min_dif) then
   {
      pos = x_in[i];
      min_dif = dif;
   }
}
return pos;
```

Prilikom hardverske implementacije polazni interfejs projektovanog hardverskog bloka treba da prati smernice iz zadatka 4.5, sa potrebnim izmenama memorijskog interfejsa kako bi se obezbedio dovoljan protok podataka iz memorije ka projektovanom digitalnom sistemu (protok će biti povećan zbog delimičnog razmotavanja petlji).

Napisati i osnovno verifikaciono okruženje pomoću kojega se može verifikovati korektan rad projektovanog sistema.

Zadatak 5.7.

Primenom RT metodologije izvršiti hardversku implementaciju *„Selection Sort“* algoritma za sortiranje niza brojeva *x* prikazanog u nastavku. Prilikom implementacije unutrašnju petlju potrebno je razmotati sa faktorom 2.

```
for (i=0; i<n_in-1; i++)
   for (j=i+1; j<n_in; j++)
      if (dir_in = 0) then
         if (x_in1[j] < x_in2[i]) then
         {
            x_out1[j] = x_in2[i];
            x_out2[i] = x_in1[j];
         }
      else
         if(x_in1[j] > x_in2[i]) then
         {
            x_out1[j] = x_in2[i];
            x_out2[i] = x_in1[j];
         }
```

Prilikom hardverske implementacije polazni interfejs projektovanog hardverskog bloka treba da prati smernice iz zadatka 4.6, sa potrebnim izmenama memorijskog interfejsa kako bi se obezbedio dovoljan protok podataka iz memorije ka projektovanom digitalnom sistemu (protok će biti povećan zbog delimičnog razmotavanja petlji).

Napisati i osnovno verifikaciono okruženje pomoću kojega se može verifikovati korektan rad projektovanog sistema.

5.3.2 Preklapanje petlje (*Loop Folding*)

„*Loop Folding*“ je tehnika pomoću koje se skraćuje vreme izvršavanja petlje preklapajući operacije iz susednih iteracija petlje. Broj operacija koje se moraju izvršiti unutar jedne iteracije petlje ostaje isti, ali se sada deo operacija odnosi na tekuću iteraciju petlje, a deo na narednu. Na ovaj način moguće je izvestan deo operacija u tekućoj iteraciji petlje izračunati unapred u prethodnoj iteraciji, a zatim koristiti ove unapred izračunate vrednosti u tekućoj iteraciji, čime se potencijalno skraćuje trajanje kritične putanje kroz *datapath* modul, a samim tim i povećava maksimalna učestanost rada čitavog sistema, skraćujući na taj način vreme potrebno za izvršavanje petlje. Na slici 5.17 prikazana je primena „*Loop Folding*“ tehnike.

```
sum = 0;
for (i=0; i < n; i++)
   sum += a(i)*b(i);
```

a)

```
sum = 0;
temp = a(0)*b(0); // prolog
for (i=1; i < n; i++)
{
   sum += temp;
   temp = a(i)*b(i);
}
sum += temp; // epilog
```

b)

Slika 5.17. „Loop folding“ tehnika: a) originalni fragment koda b) fragment koda nakon primene „Loop Folding“ tehnike

U fragmentu koda sa slike 5.17a, unutar svake iteracije petlje potrebno je izvršiti jednu operaciju množenja i jednu operaciju sabiranja. Ukoliko želimo da ove dve operacije izvršimo u istom taktu, unutar *datapath* modula moramo imati redno vezane množač i sabirač što će značajno smanjiti maksimalnu učestanost rada projektovanog sistema. Međutim, primenom „*Loop Folding*“ tehnike možemo razdvojiti operacije množenja i sabiranja u dve sukcesivne iteracije, kao što je prikazano na slici 5.17b. Sada se u svakoj iteraciji petlje izvode dve nezavisne operacije. Na tekuću vrednost sume dodaje se proizvod dva elementa nizova *a* i *b* izračunat u prethodnoj iteraciji, a istovremeno se računa i proizvod elemenata nizova *a* i *b* koji će biti korišćen u narednoj iteraciji. Na ovaj način su ove dve operacije međusobno nezavisne, pa se mogu izvršiti konkurentno. U ovom slučaju će kritična putanja biti redukovana na propagaciono kašnjenje množača, pa će i maksimalna učestanost rada ovog sistema biti veća što će na kraju rezultovati u kraćem izvršavanju petlje.

Drugi primer korišćenja „*Loop Folding*“ tehnike prikazan je na slici 5.18.

```
for (i=0; i < n; i++)
   c(i) = a(i)*b(i);
```

a)

```
temp1 = a(0); // prolog
temp2 = b(0); // prolog
for (i=1; i < n; i++)
{
   c(i-1) = temp1*temp2;
   temp1 = a(i);
   temp2 = b(i);
}
c(N-1) = temp1*temp2; // epilog
```

b)

Slika 5.18. „Loop folding“ tehnika: a) originalni fragment koda b) fragment koda nakon primene „Loop Folding“ tehnike

Pretpostavimo da su nizovi *a*, *b*, i *c* smešteni u tri odvojene jednopristupne memorije sa sinhronim upisom i čitanjem. U ovom slučaju je za izvršavanje jedne iteracije petlje originalnog fragmenta sa slike 5.18a potrebno dva takta. U prvom taktu generišu se adrese elemenata *a(i)* i *b(i)*. U sledećem taktu ove vrednosti su raspoložive i može se napraviti njihov proizvod koji se zatim upisuje u memoriju gde je smešten niz *c*. Imajući ovo u vidu, ukupno trajanje izvršavanja petlje sa slike 5.18a iznosi $2 \cdot n$ taktova. Primenom „*Loop Folding*“ tehnike moguće je preklopiti proces čitanja vrednosti elemenata *a(i)* i *b(i)* sa računanjem proizvoda prethodno pročitanih vrednosti *a(i-1)* i *b(i-1)*, kao što je prikazano na slici 5.18b. Kako su ovo nezavisne operacije mogu se izvršavati konkurentno. Trajanje jedne iteracije petlje sada iznosi samo jedan takt, a trajanje čitavog fragmenta $n+2$ taktova, što je gotovo dva puta kraće od originalne implementacije.

NAPOMENA: Ako se pažljivije pogledaju ASMD dijagrami u primerima 4.3 i 5.2 može se primetiti da je u njima već korišćena „*Loop Folding*“ tehnika za preklapanje pristupa memorijama matrica *A* i *B* i izvođenja operacije ažuriranja tekuće vrednosti elementa matrice *C*.

NAPOMENA: Obratite pažnju da se prilikom primene „*Loop Folding*“ tehnike neminovno javlja potreba za dodavanjem dodatnih operacija koje prethode petlji (takozvane **prolog** operacije) i slede nakon petlje (takozvane **epilog** operacije). Ove operacije su neophodne za pravilan rad sistema nakon primene „*Loop Folding*“ tehnike. Prolog operacije služe za inicijalno izvođenje operacija čiji rezultati će biti korišćeni prilikom prvog prolaska kroz petlju. Slično, epilog operacije služe za kompletiranje čitavog postupka računanja, implementiranog u originalnoj petlji, izvođenjem potrebnih operacija koje koriste rezultate operacija koje su bile izvedene u poslednjoj iteraciji petlje. Prolog i epilog operacije zahtevaju dodatno vreme za izvršavanje, koje nije prisutno u originalnoj implementaciji, i na taj način delimično umanjuju efekte primene „*Loop Folding*“ tehnike.

Sledeći primer ilustruje kako se originalni „*Naive*“ algoritam za množenje matrica može poboljšati primenom „*Loop Folding*“ tehnike na najdublju petlju *k* originalnog algoritma.

Primer 5.3: Množač matrica neoznačenih brojeva baziran na korišćenju „*Loop Folding*“ tehnike

U ovom primeru pokazaćemo kako se primenom „*Loop Folding*“ tehnike na telo najdublje petlje (*k* petlja) originalnog „*Naive*“ algoritma za množenje matrica može projektovati digitalni sistem koji predstavlja efikasniju implementaciju operacije množenja dve matrice.

Originalni „*Naive*“ algoritam množenja matrica, na osnovu kojega je projektovan digitalni sistema za množenje matrica u primeru 4.3, prikazan je ponovo na slici 5.19.

```
for (i = 0; i < n; i++)
   for (j = 0; j < p; j++)
   {
      temp = 0;
      for (k = 0; k < m; k++)
         temp = temp + a(i,k)*b(k,j);
      c(i,j) = temp;
   }
return c;
```

Slika 5.19. „Naive“ algoritam za množenje dve matrice

Ovaj put možemo primetiti da se unutar svake iteracije *k* petlje zahteva izvođenje jedne operacije množenja i jedne operacije sabiranja. Ukoliko ne želimo da ove dve operacije izvršimo u odvojenim taktovima moramo kaskadno povezati jedan množač i jedan sabirač, kao što je prikazano na slici 4.25. Ova operacija kaskadnog vezivanja većeg broja funkcionalnih jedinica se u engleskoj literatori naziva „*Operator Chaining*“. Međutim, usled kaskadnog vezivanja množača i sabirača, maksimalna učestanost rada projektovanog sistema biće umanjena.

Primenom „*Loop Folding*“ tehnike na telo najdublje petlje, mogu se preklopiti operacije množenja elemenata matrica *A* i *B* iz tekuće iteracije i akumuliranja vrednosti u pomoćnoj promenljivoj *temp*, kao što je prikazano na slici 5.20.

```
for (i = 0; i < n; i++)
   for (j = 0; j < p; j++)
   {
      temp = 0;
      mul_next = a(i,0)*b(0,j);
      for (k = 1; k < m; k++)
      {
         temp = temp + mul_next;
         mul_next = a(i,k)*b(k,j);
      }
      c(i,j) = temp+ mul_next;
   }
return c;
```

Slika 5.20. „Naive“ algoritam za množenje dve matrice nakon primene „Loop Folding“ tehnike

Sada se u svakoj iteraciji *k* petlje na tekuću vrednost promenljive *temp* akumulira proizvod elemenata matrica *A* i *B* izračunat u prethodnoj iteraciji, a istovremeno se računa proizvod elemenata matrica *A* i *B* koji će biti korišćen u sledećoj iteraciji. Na ovaj način su operacije množenja i sabiranja koje su bile povezane u originalnom „*Naive*“ algoritmu množenja matrica sa slike 5.19 postale nezavisne, i mogu se izvršiti konkurentno. Na ovaj način više nema potrebe za kaskadnim vezivanjem množača i sabirača, pa će i maksimalna učestanost rada ovako projektovanog sistema biti veća.

Korak 1: Eliminacija naredbi ponavljanja iz algoritma

Kao i u prethodnim primerima, da bi smo mogli da nacrtamo ASMD dijagram koji odgovara algoritmu koji implementiramo u hardveru, prvo je neophodno zameniti tri **for** petlje sa odgovarajućim **if** and **goto** naredbama. Nakon ove zamene, modifikovani „*Naive*“ algoritam množenja matrica baziran na primeni „*Loop Folding*“ tehnike opisan je pomoću sledećeg pseudo-koda.

```
      i = 0;
l1:   j = 0;
l2:      temp = 0;
         mul_next = a_in(i,0)*b_in(0,j);
         k = 1;
l3:         temp = temp + mul_next;
            mul_next = a_in(i,k)*b_in(k,j);
            k = k + 1;
         if (k = m_in) then
            goto l3e;
         else
            goto l3;
l3e:
```

```
                c_out(i,j) = temp+mul_next;
                j = j + 1;
            if (j = p_in) then
                goto l2e;
            else
                goto l2;
l2e:
        i = i + 1;
        if (i = n_in) then
            goto stop;
        else
            goto l1;

stop:
        nop
```

*Slika 5.21. Modifikovani „Naive“ algoritam za množenje dve matrice baziran na primeni „Loop Folding“ tehnike, kod kojega su **for** naredbe zamenjene **if-goto** naredbama*

Korak 2: Definisanje interfejsa digitalnog sistema

U ovom koraku potrebno je definisati četiri interfejsa digitalnog sistema koji se projektuje: ulazni interfejs podataka, izlazni interfejs podataka, komandni interfejs i statusni interfejs.

Kako prilikom primene „*Loop Folding*“ tehnike na originalni „*Naive*“ algoritam nije došlo do promene ulaznih i izlaznih parametara algoritma, interfejs digitalnog sistema koji će biti projektovan na osnovu algoritma sa slike 5.21 biće u potpunosti isti kao i interfejs digitalnog sistema iz primera 4.3:

- Ulazni interfejs podataka – sastoji se iz sledećih portova:

  Portovi koji čine interfejs ka memoriji *A*:

  - *a_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *a_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
  - *a_wr_o* – kontrolna magistrala, tipa *std_logic*

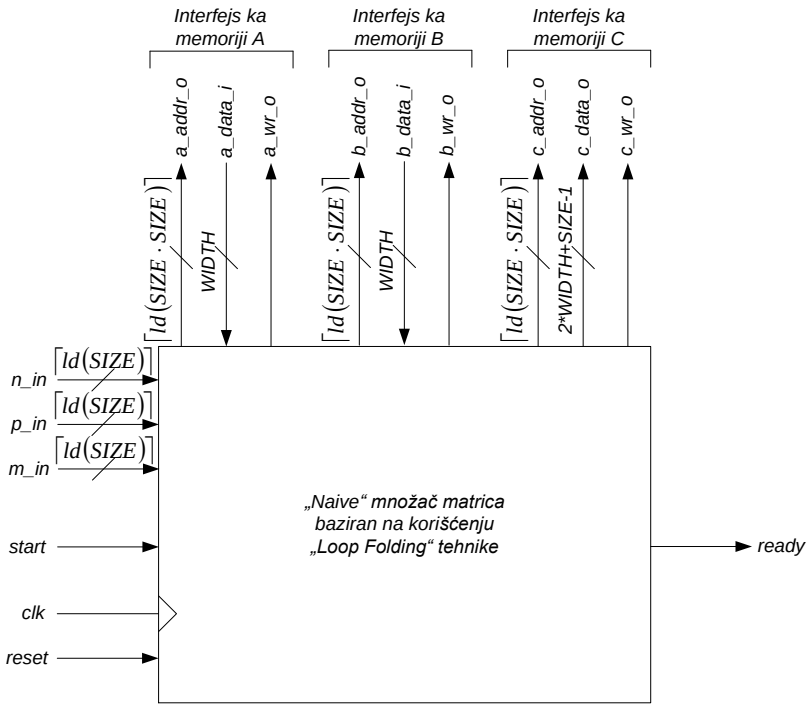
  Portovi koji čine interfejs ka memoriji *B*:

  - *b_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
  - *b_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
  - *b_wr_o* – kontrolna magistrala, tipa *std_logic*

- Izlazni interfejs podataka – sastoji se iz portova koji čine interfejs ka memoriji *C*:
    - *c_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE) \rceil$ bita
    - *c_data_o* – izlazna magistrala podataka, *std_logic_vector* tipa, širine 2**WIDTH*+*SIZE*-1 bita
    - *c_wr_o* – kontrolna magistrala, tipa *std_logic*

- Komandni interfejs – sastoji se iz jednog 1-bitnog ulaznog porta, *start*
- Statusni interfejs – sastoji se iz jednog 1-bitnog izlaznog porta, *ready*

Pored ovih portova, digitalni sistem koji projektujemo mora posedovati i standardne portove za dovođenje klok i reset signala, *clk* i *reset*. Na slici 5.22. prikazan je kompletan interfejs digitalnog sistema koji implementira „*Naive*“ algoritam množenja dve matrice baziran na korišćenju „*Loop Folding*“ tehnike.

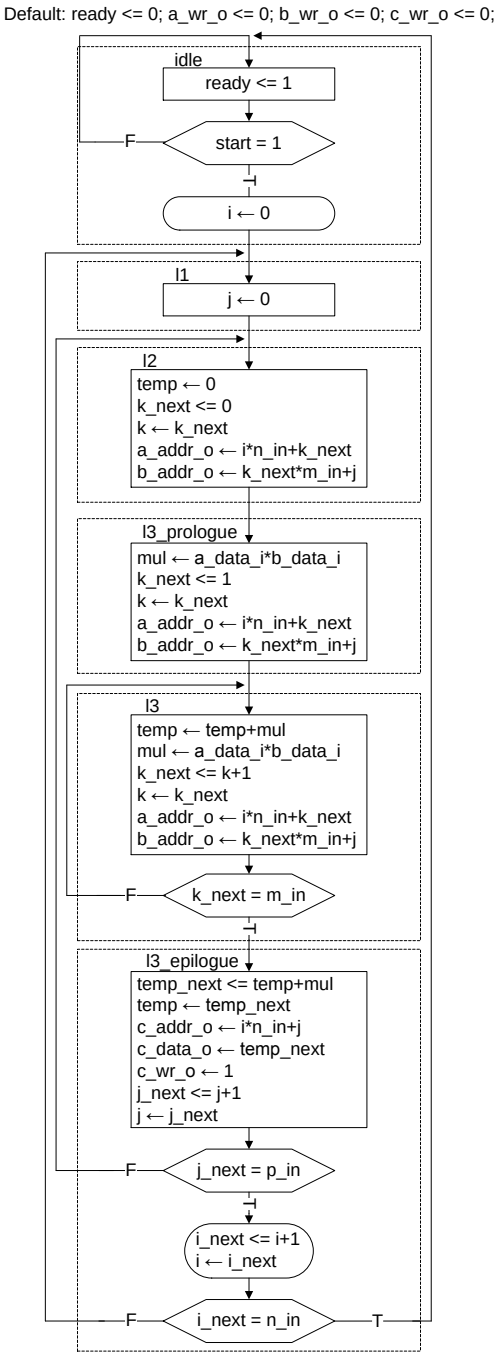


Slika 5.22. Interfejs digitalnog sistema koji implementira „Naive“ algoritam množenja dve matrice optimizovan korišćenjem „Loop Folding“ tehnike

Korak 3: Projektovanje *controlpath* modula

Uzimajući u obzir i način na koji je organizovan memorijski podsistem (vidi sliku 4.21) algoritam sa slike 5.21 lako se može prevesti u odgovarajući ASMD dijagram, koji je prikazan na slici 5.23.

ASMD dijagram ima šest stanja. U *idle* stanju, ASMD proverava *start* signal. Ako je on aktivan, ASMD inicijalizuje brojački registar *i* (koja je zapravo brojač *i* iz prve **for** petlje originalnog algoritma), a zatim prelazi u *l1* stanje. U *l1* stanju vrši se inicijalizacija drugog brojačkog registra *j* (koja je zapravo brojač *j* iz druge **for** petlje originalnog algoritma), a zatim se prelazi u stanje *l2*. U stanju *l2* inicijalizuje se poslednji brojački registar *k* (koja je zapravo brojač *i* iz prve **for** petlje originalnog algoritma), inicijalizuje registar *temp* i izračunavaju se početne adrese elemenata matrica *A* i *B*, a nakon toga se prelazi u stanje *l3_prologue*. U stanju *l3_prologue* vrši se izračunavanje proizvoda prva dva elementa matrica A i B, takođe se generišu i adrese narednih elemenata čiji proizvod će biti izračunat u sledećoj iteraciji, a zatim se prelazi u stanje *l3*. U stanju *l3* se na tekuću vrednost registra *temp* dodaje vrednost izračunatog proizvoda u prethodnoj iteraciji (registar *mul*). Istovremeno se računa proizvod naredna dva elementa matrica *A* i *B*. Zatim se inkrementuje vrednost brojačkog registra *k* i proverava da li je ona dostigla vrednost *m_in*. U slučaju da nije, ASMD ostaje u stanju *l3*, a ako jeste prelazi se u *l3_epilogue* stanje. U *l3_epilogue* stanju na akumuliranu sumu proizvoda dodaje se proizvod elemenata izračunat u poslednjoj iteraciji *k* petlje, a zatim se vrši upis izračunate vrednosti tekućeg elementa matrice *C* na odgovarajuće mesto u memoriji matrice *C*, jer je proces njegovog računanja kompletiran. Zatim se inkrementuje vrednost brojačkog registra *j* i proverava da li je dostigla vrednost *p_in*. U slučaju da nije, vraćamo se u stanje *l2*, i započinjemo računanje vrednosti sledećeg elementa u tekućoj vrsti matrice *C*. Ako je vrednost registra *j* jednaka *p_in*, inkrementuje se vrednost brojačkog registra *i*, i proverava da li je ona jednaka vrednosti *n_in*. Ukoliko jeste, postupak množenja matrica je završen i ASDM se vraća u *idle* stanje. Ukoliko vrednost *n_in* još uvek nije dostignuta, ASMD se vraća u stanje *l1*, gde započinje računanje vrednosti elemenata matrice *C* iz sledeće vrste.



Slika 5.23. ASMD dijagram „Naive“ množača matrica optimizovanog korišćenjem „Loop Folding“ tehnike

Nakon što smo nacrtali ASMD dijagram, praktično smo završili projektovanje *controlpath* modula našeg dizajna. Ono što je preostalo jeste da se izvrši projektovanje *datapath* modula.

Korak 4: Projektovanje *datapath* modula

Prvi korak prilikom projektovanja *datapath* modula jeste da identifikujemo sve registre koji su prisutni u sistemu i njima asocirane RT operacije. U „*Naive*“ algoritmu za množenje matrica sa slike 5.21 postoji pet unutrašnjih promenljivih, *i, j, k, temp* i *mul*. Svakoj od njih asociraćemo po jedan registar. Što se tiče veličine ovih registara, registri i, *j* i *k* su širine $\lceil ld(SIZE) \rceil$bita, registar *temp* je širine 2**WIDTH*+*SIZE-1* bita, jer se u njemu smešta rezultat akumulisanih proizvoda elemenata matrica *A* i *B*, tokom računanja vrednosti elemenata matrice *C*. Registar *mul* je širine 2**WIDTH* bita, jer se u njega smešta proizvod elemenata matrica *A* i *B*.

Nakon što smo odredili broj i veličinu registara u sistemu, možemo pristupiti pravljenju liste svih RT operacija koje postoje u projektovanom ASMD dijagramu. Analizom ASMD dijagrama sa slike 5.23 dolazimo do sledeće liste RT operacija:

- u *idle* stanju: $i \leftarrow 0, j \leftarrow j, k \leftarrow k, temp \leftarrow temp, mul \leftarrow mul$
- u *l1* stanju: $i \leftarrow i, j \leftarrow 0, k \leftarrow k, temp \leftarrow temp, mul \leftarrow mul$
- u *l2* stanju: $i \leftarrow i, j \leftarrow j, k \leftarrow 0, temp \leftarrow 0, mul \leftarrow mul$
- u *l3_prologue* stanju: $i \leftarrow i, j \leftarrow j, k \leftarrow 1, temp \leftarrow temp,$ $mul \leftarrow a\_data\_i * b\_data\_i$
- u *l3* stanju: $i \leftarrow i, j \leftarrow j, k \leftarrow k+1, temp \leftarrow temp+mul,$ $mul \leftarrow a\_data\_i * b\_data\_i$
- u *l3_epilogue* stanju: $i \leftarrow i, i \leftarrow i+1, j \leftarrow j+1, k \leftarrow k,$ $temp \leftarrow temp+mul, mul \leftarrow mul$

Sledeći korak jeste da se svakom od registara pridruže asocirane RT operacije.

RT operacije kojima je ciljni registar *i*:

1. u *idle* stanju: $i \leftarrow 0$
2. u *l1* stanju: $i \leftarrow i$
3. u *l2* stanju: $i \leftarrow i$
4. u *l2_prologue* stanju: $i \leftarrow i$
5. u *l3* stanju: $i \leftarrow i$

6. u *l2_epilogue* stanju: $i \leftarrow i+1$ (ako je *j_next* = *p_in*) ,
   $i \leftarrow i$ (inače)

RT operacije kojima je ciljni registar *j*:

1. u *idle* stanju: $j \leftarrow j$
2. u *l1* stanju: $j \leftarrow 0$
3. u *l2* stanju: $j \leftarrow j$
4. u *l3_prologue* stanju: $j \leftarrow j$
5. u *l3* stanju: $j \leftarrow j$
6. u *l3_epilogue* stanju: $j \leftarrow j+1$

RT operacije kojima je ciljni registar *k*:

1. u *idle* stanju: $k \leftarrow k$
2. u *l1* stanju: $k \leftarrow k$
3. u *l2* stanju: $k \leftarrow 0$
4. u *l3_prologue* stanju: $k \leftarrow 1$
5. u *l3* stanju: $k \leftarrow k + 1$
6. u *l3_epilogue* stanju: $k \leftarrow k$

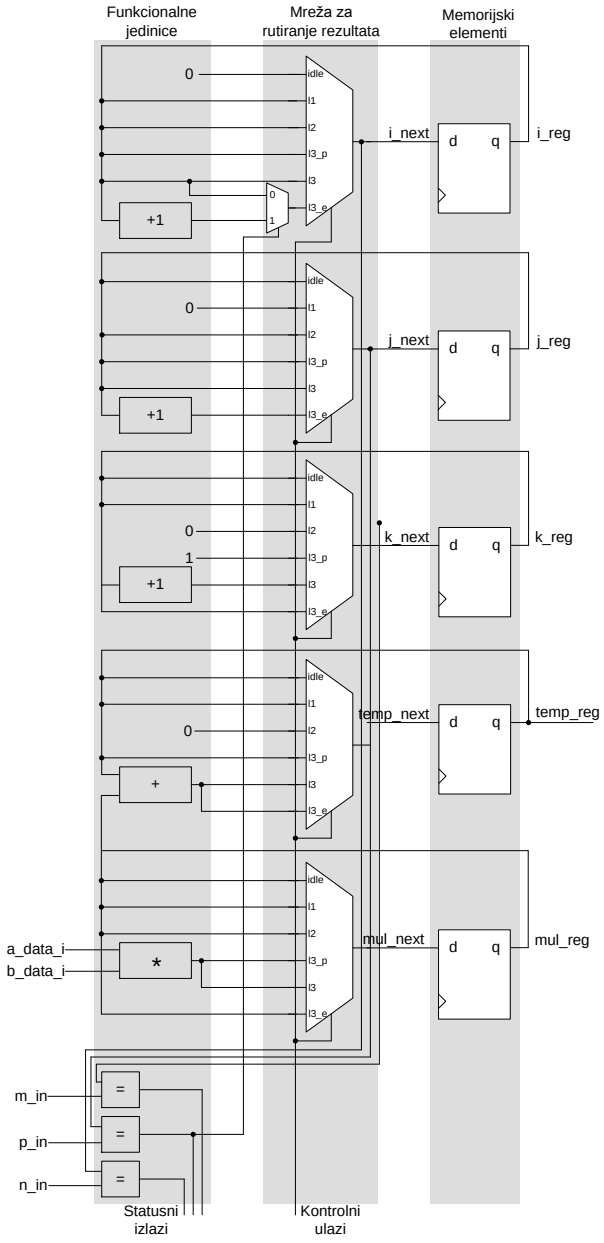
RT operacije kojima je ciljni registar *temp*:

1. u *idle* stanju: $temp \leftarrow temp$
2. u *l1* stanju: $temp \leftarrow temp$
3. u *l2* stanju: $temp \leftarrow 0$
4. u *l3_prologue* stanju: $temp \leftarrow temp$
5. u *l3* stanju: $temp \leftarrow temp+mul$
6. u *l3_epilogue* stanju: $temp \leftarrow temp+mul$

RT operacije kojima je ciljni registar *mul*:

1. u *idle* stanju: $mul \leftarrow mul$
2. u *l1* stanju: $mul \leftarrow mul$
3. u *l2* stanju: $mul \leftarrow mul$
4. u *l3_prologue* stanju: $mul \leftarrow a\_data\_i*b\_data\_i$
5. u *l3* stanju: $mul \leftarrow a\_data\_i*b\_data\_i$
6. u *l3_epilogue* stanju: $mul \leftarrow mul$

Na osnovu ovih informacija moguće je formirati delove *datapath* modula koji su asocirani svakom od registara u sistemu. Kompletna struktura *datapath* modula prikazana je na slici 5.24.



Slika 5.24. Datapath modul „Naive“ množača matrica optimizovanog korišćenjem „Loop Folding“ tehnike

Korak 5: Pisanje HDL modela

Nakon što smo projektovali *datapth* i *controlpath* module, poslednji korak predstavlja pisanje odgovarajućeg HDL modela čitavog digitalnog sistema koji implementira „*Naive*“ algoritam množenja matrica, optimizovan korišćenjem „*Loop Folding*“ tehnike, u hardveru. Kao što je to bio slučaj u ranijim primerima i ovaj model se može napisati na različite načine. U nastavku je prikazan VHDL model projektovanog digitalnog sistema za implementaciju „*Naive*“ algoritma množenja dve matrice, optimizovanog korišćenjem „*Loop Folding*“ tehnike, koji modeluje *datapath* i *controlpath* module unutar istog entiteta, koristeći dvoprocesni stil modelovanja.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.utils_pkg.all;

entity matrix_mult_lf is
  generic (
    WIDTH: integer := 8;
    SIZE: integer := 3
      );
  port (
  ---------------- Clocking and reset interface ---------------
    clk:                in std_logic;
    reset:              in std_logic;
      ------------------- Input data interface -------------------
    -- Matrix A memory interface
    a_addr_o:           out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    a_data_i:           in std_logic_vector(WIDTH-1 downto 0);
    a_wr_o:             out std_logic;
    -- Matrix B memory interface
    b_addr_o:           out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    b_data_i:           in std_logic_vector(WIDTH-1 downto 0);
    b_wr_o:             out std_logic;
    -- Matrix dimensions definition interface
    n_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
    p_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
    m_in:               in std_logic_vector(log2c(SIZE)-1 downto 0);
      ------------------- Output data interface ------------------
    -- Matrix C memory interface
    c_addr_o:           out std_logic_vector(log2c(SIZE*SIZE)-1 downto 0);
    c_data_o:           out std_logic_vector(2*WIDTH+SIZE-1 downto 0);
    c_wr_o:             out std_logic;
      ---------------------- Command interface ---------------------
    start:              in std_logic;
      ---------------------- Status interface ----------------------
    ready:              out std_logic);
end entity;

architecture two_seg_arch of matrix_mult_lf is
  type state_type is (idle, l1, l2, l3, l3_prologue, l3_epilogue);
  signal state_reg, state_next: state_type;
  signal i_reg, i_next: unsigned(log2c(SIZE)-1 downto 0);
  signal j_reg, j_next: unsigned(log2c(SIZE)-1 downto 0);
  signal k_reg, k_next: unsigned(log2c(SIZE)-1 downto 0);
  signal temp_reg, temp_next: unsigned(2*WIDTH+SIZE-1 downto 0);
```

```
  signal mul_reg, mul_next: unsigned(2*WIDTH-1 downto 0);
begin
    -- State and data registers
  process (clk, reset)
  begin
    if reset = '1' then
      state_reg <= idle;
        i_reg <= (others => '0');
        j_reg <= (others => '0');
        k_reg <= (others => '0');
        temp_reg <= (others => '0');
        mul_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
      state_reg <= state_next;
        i_reg <= i_next;
        j_reg <= j_next;
        k_reg <= k_next;
        temp_reg <= temp_next;
        mul_reg <= mul_next;
    end if;
  end process;

    -- Combinatorial circuits
  process (state_reg, start, a_data_i, b_data_i, i_reg, j_reg, k_reg, temp_reg, mul_reg,
              i_next, j_next, k_next, temp_next, mul_next)
  begin
    -- Default assignments
    i_next <= i_reg;
    j_next <= j_reg;
    k_next <= k_reg;
    temp_next <= temp_reg;
    a_addr_o <= (others => '0');
    a_wr_o <= '0';
    b_addr_o <= (others => '0');
    b_wr_o <= '0';
    c_addr_o <= (others => '0');
    c_data_o <= (others => '0');
    c_wr_o <= '0';
    ready <= '0';

    case state_reg is
      when idle =>
        ready <= '1';
        if start = '1' then
          i_next <= to_unsigned(0, log2c(SIZE));
          state_next <= l1;
        else
          state_next <= idle;
        end if;

      when l1 =>
        j_next <= to_unsigned(0, log2c(SIZE));
        state_next <= l2;

      when l2 =>
        temp_next <= to_unsigned(0, 2*WIDTH+SIZE);
        k_next <= to_unsigned(0, log2c(SIZE));
        a_addr_o <= std_logic_vector(i_reg*unsigned(n_in)+k_next);
        b_addr_o <= std_logic_vector(k_next*unsigned(m_in)+j_reg);
```

```
        state_next <= l3_prologue;

      when l3_prologue =>
        mul_next <= unsigned(a_data_i)*unsigned(b_data_i);
        k_next <= to_unsigned(1, log2c(SIZE));
        a_addr_o <= std_logic_vector(i_reg*unsigned(n_in)+k_next);
        b_addr_o <= std_logic_vector(k_next*unsigned(m_in)+j_reg);
        state_next <= l3;

      when l3 =>
        temp_next <= temp_reg + mul_reg;
        mul_next <= unsigned(a_data_i)*unsigned(b_data_i);
        k_next <= k_reg + 1;
        a_addr_o <= std_logic_vector(i_reg*unsigned(n_in)+k_next);
        b_addr_o <= std_logic_vector(k_next*unsigned(m_in)+j_reg);
        if (k_next = unsigned(m_in)) then
          state_next <= l3_epilogue;
        else
          state_next <= l3;
        end if;

      when l3_epilogue =>
        temp_next <= temp_reg + mul_reg;
        c_addr_o <= std_logic_vector(i_reg*unsigned(n_in)+j_reg);
        c_data_o <= std_logic_vector(temp_next);
        c_wr_o <= '1';
        j_next <= j_reg + 1;
        if (j_next = unsigned(p_in)) then
          i_next <= i_reg + 1;
          if (i_next = unsigned(n_in)) then
            state_next <= idle;
          else
            state_next <= l1;
          end if;
        else
          state_next <= l2;
        end if;
    end case;
  end process;
end two_seg_arch;
```

NAPOMENA: *utils_pkg* paket, koji koristi prethodni model isti je kao i u primeru 4.3, pa ovde neće biti prikazan njegov sadržaj.

Na kraju, procenimo brzinu rada „*Naive*“ množača matrica optimizovanog korišćenjem „*Loop Folding*“ tehnike. Analizom ASMD dijagrama sa slike 5.23, može se zaključiti da računanje vrednosti svakog elementa matrice *C* zahteva *m_in*+1 taktova (kroz stanje l3 prolazimo *m_in*-1 puta, plus po jedan prolaz kroz stanja *l3_prologue* i *l3_epilogue*). Nakon završetka računanja vrednosti tekućeg elementa matrice *C*, pre započinjanja računanja vrednosti narednog elementa troši se još jedan takt (ponovo se vraćamo u *l2* stanje kako bismo inicijalizovali brojač *k*). Dodatni takt takođe se troši i svaki put kada započinjenom računanje vrednosti elemenata matrice *C* iz sledeće vrste (kada se vraćamo u stanje *l1*). Imajući sve ovo u vidu, potreban broj taktova da se pomoću

projektovanog „*Naive*“ množača matrica optimizovanog korišćenjem „*Loop Folding*“ tehnike pomnože dve matrice dimenzija *nxm* i *mxp* iznosi

$$T_{naive\_lu2} = n\_in \cdot \left(p\_in \cdot \left(m\_in + 2 \right) + 1 \right) > T_{naive}$$

taktova.

Na osnovu prethodnog izraza možemo zaključiti je trajanje množenja dve matrice pomoću „*Naive*“ algoritma optimizovanog korišćenjem „*Loop Folding*“ tehnike duže od vremena potrebnog da se pomnože dve matrice korišćenjem dizajna iz primera 4.3, što je i bilo očekivano jer se dodatno vreme troši na prolog i epilog stanja prilikom preklapanja petlje po iteratoru *k, l3_prologue* i *l3_epilogue*. Ovo produženje nije veliko i iznosi *n_in·p_in* taktova, što u slučaju velikih matrica postaje zanemarljivo u odnosu na član *n_in·p_in·m_in*. Sa druge strane kritična putanja unutar *datapath* modula „*Naive*“ množača optimizovanog korišćenjem „*Loop Folding*“ tehnike, prikazanog na slici 5.24, sadrži samo množač, za razliku od kritične putanje *datapath* modula originalnog „*Naive*“ množača, prikazanog na slici 4.25, koja sadrži i množač i sabirač.

Procenimo koliko povećanje maksimalne učestanosti rada „*Naive*“ množača optimizovanog korišćenjem „*Loop Folding*“ tehnike možemo da očekujemo usled ovog skraćenja kritične putanje. Kako kašnjenje *Carry-Lookahead* sabirača raste logaritamski sa širinom ulaznih operanada, a kašnjenje *Array* množača raste linearno, kritične putanje u ova dva slučaja približno imaju sledeće vrednosti (zanemarujući kašnjenja kroz multipleksere)

$$T_{naive\_lf} = 2n, \quad T_{naive} = 2n + 4\log_4 n \, .$$

Ako pretpostavimo da su elementi matrica *A* i *B* zadati kao 16-bitni brojevi, onda je kritična putanja „*Naive*“ množača optimizovanog korišćenjem „*Loop Folding*“ tehnike kraća od kritične putanje originalnog „*Naive*“ množača matrica

$$\frac{T_{naive}}{T_{naive\_lf}} = \frac{2n + 4\log_4 n}{2n} = \frac{2 \cdot 16 + 4\log_4 16}{2 \cdot 16} = \frac{40}{32} = 1.25$$

puta.

Ovo znači da će „*Naive*“ množač matrica optimizovan korišćenjem „*Loop Folding*“ tehnike moći da radi na 1.25 puta većoj učestanosti (što predstavlja ubrzanje od 25%) od originalnog „*Naive*“ množača matrica.

Korak 6: Verifikacija razvijenog HDL modela

Verifikaciono okruženje koje bi moglo da se iskoristi za verifikaciju „*Naive*“ množača matrica optimizovanog korišćenjem „*Loop Folding*“ tehnike identično je sa verifikacionim okruženjem koje je bilo korišćeno za verifikaciju ispravnog rada originalnog „*Naive*“ algoritma, u primeru 4.3.

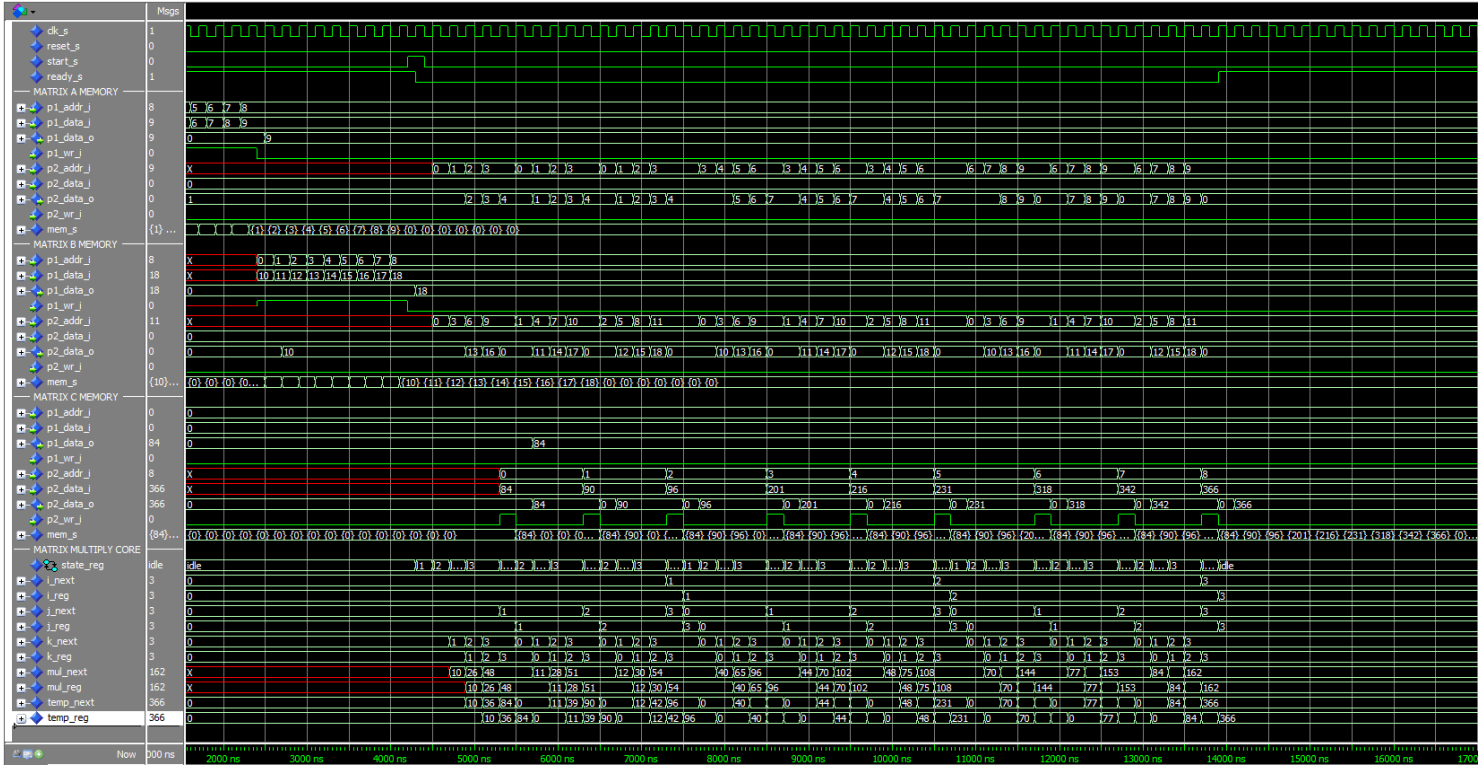
Stoga će na ovom mestu samo biti prikazan izgled talasnih oblika karakterističnih signala, prikazanih na slici 5.25, koji predstavljaju rezultat simulacije verifikacionog okruženja i modela „*Naive*“ množača matrica optimizovanog korišćenjem „*Loop Folding*“ tehnike, u slučaju množenja kvadratnih matrica *A* i *B*, deminenzija 3x3. Matrice *A* i *B* definisane su kao i u prethodnim primerima na sledeći način

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \qquad B = \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}.$$

Rezultat množenja ove dve matrice trebalo bi da bude sledeće matrica

$$C = A \cdot B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix} = \begin{bmatrix} 84 & 90 & 96 \\ 201 & 216 & 231 \\ 318 & 342 & 366 \end{bmatrix}.$$

Inspekcijom konačnih vrednosti unutrašnjih signala *mem_s* sva tri memorijska modula („*Matrix A Memory*“, „*Matrix B Memory*“ i „*Matrix C Memory*“) sa slike 5.25, možemo se uveriti da projektovani digitalni sistem korektno implementira „*Naive*“ algoritam množenja dve matrice.



Slika 5.25. Rezultat simulacije rada „Naive“ množača matrica optimizovanog korišćenjem „Loop Folding“ tehnike, na primeru množenja dve matrice dimenzija 3x3

Zadaci za vežbu

Zadatak 5.8.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za određivanje skalarnog proizvoda dva *n*-dimenzionalna vektora. Prilikom implementacije koristiti „*Loop Folding*“ tehniku.

```
sp = 0;
for (i=0; i < n; i++)
   sp += a(i)*b(i);
```

Prilikom hardverske implementacije pretpostaviti da su elementi vektora *a* i *b* predstavljeni kao 8-bitni označeni brojevi smešteni odvojenim jednopristupnim memorijama. Pretpostaviti da je upis u memorije sinhroni i da se odvija na rastuću ivicu globalnog sinhronizacionog signala, dok je čitanje takođe sinhrono, pri čemu se na odgovarajućem izlazu pojavljuje trenutni sadržaj memorijske lokacije koja je adresirana.

Hardverski modul koji je potrebno projektovati treba da ima sledeće portove:

- ulazne portove, *a_in* i *b_in* preko kojih se prosleđuju elementi vektora *a* i *b* koji su pročitani iz memorija,
- adresne portove, *a_adr_out* i *b_adr_out* preko kojih se prosleđuju adrese elemenata kojima se želi pristupiti (pretpostaviti da su i ovi portovi 8-bitni),
- signal dozvole upisa, odnosno čitanja iz memorija, *a_wr_out* i *b_wr_out* (kada je signal na visokom logičkom nivou vrši se upis u memoriju),
- jedan ulazni port, *n_in*, preko kojega se zadaje dimenzija vektora koje je potrebno obraditi (veličina ovog porta je takođe 8 bita),
- jedan izlazni port, *sp_out*, preko kojega se prosleđuje izračunati skalarni proizvod vektora (odrediti minimalni broj potrebnih bitova za reprezentaciju vrednosti skalarnog proizvoda koji se prosleđuje kroz ovaj port).

Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije računanja skalarnog proizvoda i izlazni *ready* signal, koji označava završetak tekuće operacije računanja skalarnog proizvoda.

Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema.

Zadatak 5.9.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za određivanje naredne vrednosti izlaznog signala FIR diskretnog sistema *n*-tog reda. Prilikom implementacije koristiti „*Loop Folding*“ tehniku.

```
y = 0;
for (i=0; i < n; i++)
   y += b(i)*x(n-i);
```

Prilikom hardverske implementacije pretpostaviti da su koeficijenti filtra (niz *b*) i vrednosti prethodnih *n* odbiraka ulaznog signala (niz *x*) predstavljeni kao 8-bitni označeni brojevi smešteni odvojenim jednopristupnim memorijama. Pretpostaviti da je upis u memorije sinhroni i da se odvija na rastuću ivicu globalnog sinhronizacionog signala, dok je čitanje takođe sinhrono, pri čemu se na odgovarajućem izlazu pojavljuje trenutni sadržaj memorijske lokacije koja je adresirana.

Hardverski modul koji je potrebno projektovati treba da ima sledeće portove:

- ulazne portove, b_*in* i *x_in* preko kojih se prosleđuju koeficijenti FIR filtra *b* i odbirci ulaznog signala *x*, koji su pročitani iz memorija,
- adresne portove, *b_adr_out* i *x_adr_out* preko kojih se prosleđuju adrese koeficijenta i ulaznog odbirka kojima se želi pristupiti (pretpostaviti da su i ovi portovi 8-bitni),
- signal dozvole upisa, odnosno čitanja iz memorija, *b_wr_out* i *x_wr_out* (kada je signal na visokom logičkom nivou vrši se upis u memoriju),
- jedan ulazni port, *n_in*, preko kojega se zadaje red FiR filtra (veličina ovog porta je takođe 8 bita),
- jedan izlazni port, *y_out*, preko kojega se prosleđuje izračunata vrednost sledećeg obirka izlaznog signala (odrediti minimalni broj potrebnih bitova za reprezentaciju vrednosti odbirka izlaznog signala koji se prosleđuje kroz ovaj port).

Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije računanja vrednosti odbirka izlaznog signala filtra i izlazni *ready* signal, koji označava završetak tekuće operacije računanja vrednosti odbirka izlaznog signala filtra.

Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema.