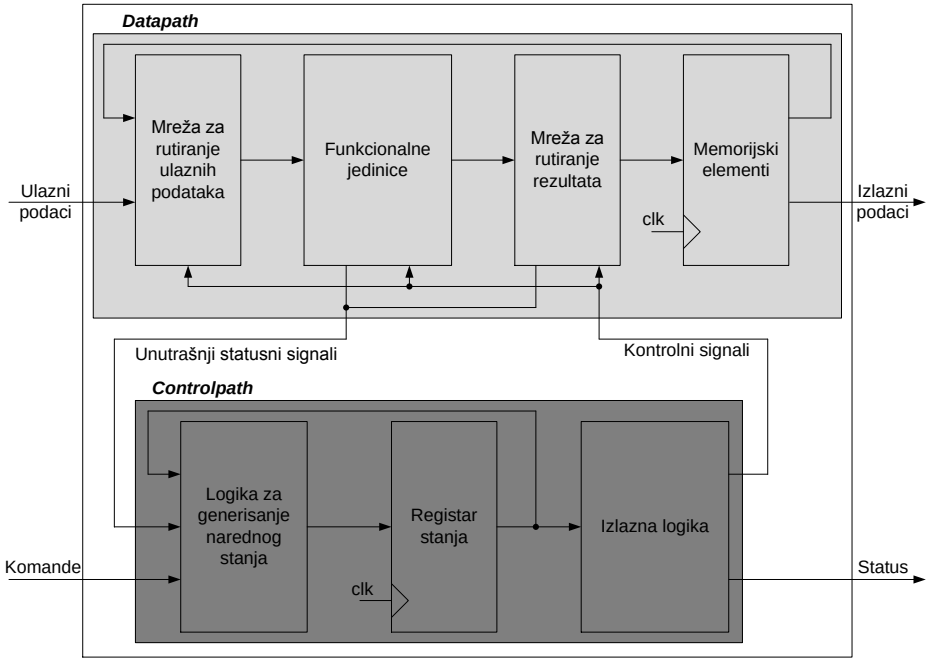
# 4.

# RT modelovanje

*Register Transfer* (RT) metodologija predstavlja formalizovan postupak projektovanja digitalnog elektronskog sistema koji implementira izabrani algoritam. Drugim rečima, pomoću RT metodologije moguće je proizvoljni algoritam prevesti u odgovarajuće digitalno elektronsko kolo, odnosno izvršiti hardversku implementaciju algoritma.

## 4.1. Postupak projektovanja digitalnih sistema primenom RT metodologije

RT metodologija opisuje rad digitalnog sistema preko sekvence transformacija podataka i njihovog prenosa između registara koji su prisutni u sistemu. Prilikom mapiraja algoritma u odgovarajući digitalni sistem promenljive koje se koriste u algoritmu mapiraju se u registre, a sve operacije (koje se u RT terminologiji zovu RT operacije) koje se u okviru algoritma izvode nad promenljivima mapiraju se u odgovarajuće hardverske funkcionalne jedinice, koje zajedno sa registrima formiraju *datapath* modul. Sekvencijalno izvršavanje operacija unutar algoritma prevodi se u sekvencu transformacija i premeštanja podataka (RT operacija) koju specificira konačni automat, koji zapravo predstavlja *controlpath* modul. Na osnovu prethodnog, može se primetiti da se primenom RT metodologije projektovani digitalni sistem na najvišem nivou hijerarhije uvek sastoji iz dva modula: *datapth* i *controlpath* modula. Generička struktura digitalnog sistema, projektovanog primenom RT metodologije, prikazana je na slici 4.1.




*Slika 4.1. Generička arhitektura digitalnog sistema projektovanog korišćenjem RT metodologije*

Sa slike 4.1 možemo videti da se digitalni sistem koji implementira željeni algoritam, projektovan primenom RT metodologije, sastoji iz sledećih modula:

- ***Datapath* modula** – zadužen za izvođenje svih RT operacija koje su identifikovane u algoritmu koji se implementira. Sastoji se iz tri modula:
    - **Memorijskih elemenata** – u standardnoj RT metodologiji radi se o registrima sa paralelnim ulazom i izlazom, koji služe za čuvanje svih međurezultata kao i konačnih rezultata koji se generišu tokom izvršavanja implementiranog algoritma
    - **Funkcionalnih jedinica** – koje realizuju sve operacije koje postoje u algoritmu koji se implementira. Po pravilu, funkcionalne jedinice koje se najčešće sreću su sabirači, oduzimači, množači, inkrementori, dekrementori, pomerači, komparatori, itd.
    - **Mreža za rutiranje** – služe za dovođenje izlaza odgovarajućih registara do odgovarajućih funkcionalnih jedinica, kao i za dovođenje izlaza funkcionalnih jedinica do ulaza odgovarajućih registara. Tipično su sastavljene od multipleksera, čiji selekcioni ulazi su pod kontrolom *controlpath* modula.

*Datapath* modul ima sledeće interfejse:

- **Ulazni interfejs podataka** – preko kojega se podaci koje je potrebno obraditi dovode do *datapath* modula
- **Izlazni interfejs podataka** – preko kojega se rezultati izvršavanja algoritma, implementiranog u digitalnom sistemu, prosleđuju spoljašnjem okruženju
- **Kontrolni interfejs** – pomoću kojega *controlpath* modul upravlja radom *datapath* modula
- **Statusni interfejs** – preko kojega se *controlpath* modulu šalju informacije o trenutnom statusu *datapath* modula. Na osnovu ovih statusnih signala konačni automat unutar *controlpath* modula može doneti odluku o narednim koracima obrade.

- ***Controlpath* modula** – koji je realizovan kao konačni automat. Unutar stanja i prelaza između stanja „mapirana" je kontrolna logika izvršavanja implementiranog algoritma. *Controlpath* modul određuje kada se koje operacije izvršavaju nad kojim podacima unutar *datapath* modula, i kada je proces izvršavanja implementiranog algoritma završen. *Controlpath* modul ima sledeće interfejse:
  - **Komandni interfejs** – ulazni interfejs pomoću kojega okruženje može upravljati radom digitalnog sistema u kojem je implementiran željeni algoritam. Preko ovog porta može se konfigurisati digitalni sistem, pokrenuti ili zaustaviti proces izvršavanja algoritma, itd.
  - **Izlazni statusni interfejs** – izlazni interfejs preko kojega digitalni sistem prosleđuje informacije o tekućem stanju obrade. Pomoću ovog interfejsa moguće je signalizirati okruženju da je sistem spreman za izvršavanje nove komande, status izvršenja tekuće komande, itd.
  - **Kontrolni interfejs** – pomoću kojega *controlpath* modul šalje upravljačke signale do *datapath* modula, na taj način upravljajući njegovim radom
  - **Ulazni statusni interfejs** – preko kojega *controlpath* modul prima statusne informacije od *datapath* modula, na osnovu kojih zatim može doneti odluku o narednoj akciji koju treba izvršiti

Proces projektovanja digitalnog sistema, koji treba da implementira odabrani algoritam, pomoću RT metodologije sastoji se iz sledećih pet koraka:

1. **Eliminacija naredbi ponavljanja -** ukoliko u algoritmu koji je potrebno implementirati postoje naredbe ponavljanja (**for**, **repeat** ili **while** petlje), potrebno ih je zameniti odgovarajućim **if-goto** naredbama.
2. **Definisanje interfejsa digitalnog sistema koji se projektuje** - analizom algoritma potrebno je uočiti ulazne i izlazne promenljive u algoritmu. Ulazne

promenljive biće mapirane u odgovarajuće portove ulaznog interfejsa podataka digitalnog sistema, dok će izlazne promenljive biti mapirane u portove izlaznog interfejsa podataka, kao što se može videti na slici 4.1. Pored ovih interfejsa, po potrebi, potrebno je uvesti i dodatne portove, pomoću kojih će biti realizovani komandni i statusni interfejs, koji su takođe prikazani na slici 4.1.

3. **Projektovanje *controlpath* modula** - na osnovu modifikovanog algoritma kreira se odgovarajući ASMD dijagram. Ovaj korak može biti trivijalan, ukoliko ne želimo da izvršimo optimizacije u pogledu brzine rada projektovanog sistema i/ili broja potrebnih hardverskih resursa, jer u tom slučaju ASMD dijagram blisko prati tok samog algoritma. Međutim, ukoliko želimo da iskoristimo deo ili ceo raspoloživi paralelizam unutar algoritma koji se implementira, ili pak želimo da optimizujemo korišćenje hardverskih resursa prema nekom kriterijumu (najčešće je reč o minimizaciji potrebnih resursa), razvoj odgovarajućeg ASMD dijagrama može biti vrlo zahtevan.

4. **Projektovanje *datapath* modula** – sastoji se iz četiri koraka:

    4.1. Prvi korak zahteva identifikaciju svih unutrašnjih promenljivih unutar algoritma koji se implementira, alokaciju i dimenzionisanje registara koji će biti pridruženi ovim promenljivim, kao i pravljenje liste svih RT operacija koje postoje unutar ASMD dijagrama.

    4.2. Vrši se grupisanje RT operacija prema odredišnim registrima. Sve RT operacije koje imaju isti odredišni registar smeštaju se u istu grupu.

    4.3. Za svaku grupu RT operacija formira se digitalno kolo na sledeći način:

        4.3.1. Formira se ciljni registar.

        4.3.2. Formiraju se kombinacione mreže za implementaciju svih funkcionalnih transformacija koje su prisutne u posmatranoj grupi RT operacija.

        4.3.3. Dodaju se multiplekserska i rutirajuća kola ukoliko je ciljni registar asociran sa većim brojem RT operacija.

    4.4. Dodaju se kombinacione mreže neophodne za formiranje potrebnih izlaznih statusnih signala iz *datapath* modula.

5. **Pisanje HDL modela** - objedinjavanjem svih kola, formiranih u koraku 4, dobija se konačni blok dijagram *datapath* modula, čime je njegovo projektovanje završeno. Na osnovu ovog blok dijagrama može se napisati odgovarajući HDL model *datapath* modula. Takođe se na osnovu razvijenog ASMD dijagrama u koraku 3, može napisati i HDL model *controlpath* modula. Nakon što se napišu HDL modeli *datapath* i *controlpath* modula može se napisati i HDL model kompletnog digitalnog kola koje implementira odabrani algoritam. Reč je o HDL modelu koji objedinjuje HDL modele *datapath* i *controlpath* modula.

Navedeni postupak implementacije proizvoljnog algoritma pomoću digitalnog elektronskog kola, primenom RT metodologije, biće ilustrovan na nekoliko primera.

## 4.2. Primeri projektovanja digitalnih sistema primenom RT metodologije

**Primer 4.1: Projektovanje „*Repetitive-Addition*“ množača neoznačenih brojeva**

Množač dva neoznačena broja, baziran na sukcesivnom sabiranju (u engleskoj literaturi poznat pod nazivom „*Repetitive-Addition Multiplier*“), baziran je na vrlo jednostavnom principu. Umesto množenja dva broja, *a*b*, isti rezultat možemo dobiti i ako saberemo broj *a b* puta, ili alternatnivno, broj *b a* puta

$$a*b=\underbrace{a+a+\ldots+a}_{b\ puta}=\underbrace{b+b+\ldots+b}_{a\ puta}.$$

Opisani postupak množenja, baziran na sukcesivnom sabiranju, može se formalizovati u formu sledećeg algoritma

```
if (a_in = 0 or b_in = 0) then
{
   r = 0;
}
else
{
   a = a_in;
   n = b_in;
   r = 0;
   while (n != 0)
   {
      r = r + a;
      n = n - 1;
   }
}
r_out = r;
```

*Slika 4.2. Množenje dva broja pomoću „Repetitive-Addition“ algoritma*

**Korak 1: Eliminacija naredbi ponavljanja iz algoritma**

U algoritmu sa slike 4.2. postoji jedna naredba ponavljanja. Reč je o *while* petlji. Kako ASMD dijagrami ne mogu modelovati naredbe ponavljanja, potrebno ih je zameniti sa odgovarajućim **if** i **goto** naredbama. Modifikovani „*Repetitive-Addition Multiplier*“ algoritam, u kojem je **while** naredba zamenjena sa odgovarajućim **if** i **goto** naredbama prikazan je na slici 4.3.

```
if (a_in = 0 or b_in = 0) then
{
   r = 0;
}
else
{
   a = a_in;
   n = b_in;
   r = 0;
   op:  r = r + a;
   n = n - 1;
   if (n = 0) then
   {
      goto stop;
   }
   else
   {
      goto op;
   }
}
r_out = r;
```

*Slika 4.3. Modifikovani „Repetitive-Addition“ algoritam, nakon eliminacije brojačkih naredbi*

**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 4.3., možemo videti da postoje dve ulazne promenljive, *a_in* i *b_in*. Najjednostavniji način za formiranje ulaznog interfejsa podataka jeste da se svakoj od ovih promenljivih pridruži po jedan višebitni ulazni port. Kako u postavci zadatka nije eksplicitno naglašeno koji su opsezi mogućih vrednosti ulaznih promenljivih *a_in* i *b_in*, možemo ih odabrati proizvoljno. U našem primeru koristićemo 8-bitne *std_logic_vector* ulazne portove *a_in* i *b_in*.

Izlazni interfejs podataka formira se tako što se u algoritmu koji se mapira u hardver identifikuju sve izlazne promenljive. U našem slučaju, analizom algoritma sa slike 4.3., možemo videti da postoji samo jedna izlazna promenljiva, *r_out*. Kao i u slučaju ulaznog interfejsa podataka, najjednostavniji način za formiranje izlaznog interfejsa podataka jeste da se svakoj od ovih od izlaznih promenljivih pridruži po jedan višebitni izlazni port. Za razliku od broja bita ulaznih višebitnih portova, koji se mogu birati proizvoljno, ako to nije drugačije naglašeno, u slučaju izbora širine izlaznih portova ona je obično vezana sa odabranom širinom ulaznih portova, preko neke funkcionalne veze. U našem primeru širina izlaznog porta *r_out*, preko kojega će se prenositi rezultat množenja dva broja, mora biti barem jednaka zbiru širina ulaznih portova preko kojih se prenose vrednosti operanda *a_in* i *b_in*

**length**(*r_out*) ≥ **length**(*a_in*) + **length**(*b_in*)

Ovo je stoga što je poznato da proizvod *n*-bitnog i *m*-bitnog binarnog broja u najgorem slučaju može biti *n*+*m*-bitni binarni broj. U ovom primeru funkcionalna veza koja povezuje dimenzije ulaznih portova za podatke sa dimenzijama izlaznih portova za podatke je relativno jednostavna. U opštem slučaju pronalaženje ove funkcionalne veze može biti vrlo težak zadatak.

Kako smo mi ograničili širinu ulaznih portova *a_in* i *b_in* na 8 bita, primenom gornje funkcionalne veze možemo zaključiti da minimalna širina izlaznog porta *r_out* mora biti barem 16 bita. Imajući ovo u vidu, izlazni port *r_out* biće implementiran kao 16-bitni *std_logic_vector* izlazni port.

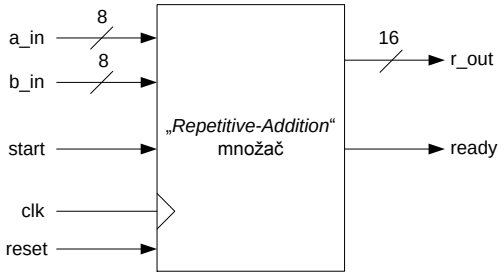
Komandni interfejs nam omogućava kontrolišemo i upravljamo radom digitalnog sistema pomoću koji implementira željeni algoritam (u našem slučaju to će biti „*Repetitive-Addition*“ algoritam množenja dva broja) u hardveru. Složenost komandnog interfejsa u dobroj meri zavisi i od složenosti samog algoritma koji se implementira. Kako je „*Repetitive-Addition*“ algoritam vrlo jednostavan, možemo koristiti najjednostavniji mogući komandni interfejs, koji se sastoji samo od jednog 1-bitnog ulaznog porta, *start*. Kada je ulazni port *start* postavljen na jedinicu, digitalni sistem treba da započne operaciju množenja dva binarna broja. Dok se ulazni port *start* na nuli, digitalni sistem treba da bude u stanju mirovanja.

Statusni interfejs nam omogućava da dobijemo informacije o trenutnom stanju digitalnog sistema koji implementira željeni algoritam. Složenost ovog interfejsa takođe u velikoj meri zavisi od složenosti samog algoritma koji se implementira. U slučaju implementacije „*Repetitive-Addition*“ algoritma 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 dva binarna broja). 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 dva binarna broja koje se još uvek u toku, tako da ne može započeti novu operaciju množenja).

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

- Ulazni interfejs podataka – sastoji se iz dva 8-bitna *std_logic_vector* ulazna porta, *a_in* i *b_in*
- Izlazni interfejs podataka – sastoji se iz jednog 16-bitnog *std_logic_vector* izlaznog porta, *r_out*
- 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 4.4. prikazan je kompletan interfejs digitalnog sistema koji implementira „*Repetitive-Addition*“ algoritam množenja dva broja.




*Slika 4.4. Interfejs digitalnog sistema koji implementira „Repetitive-Addition“ algoritam množenja dva broja*

**Korak 3: Projektovanje *controlpath* modula**

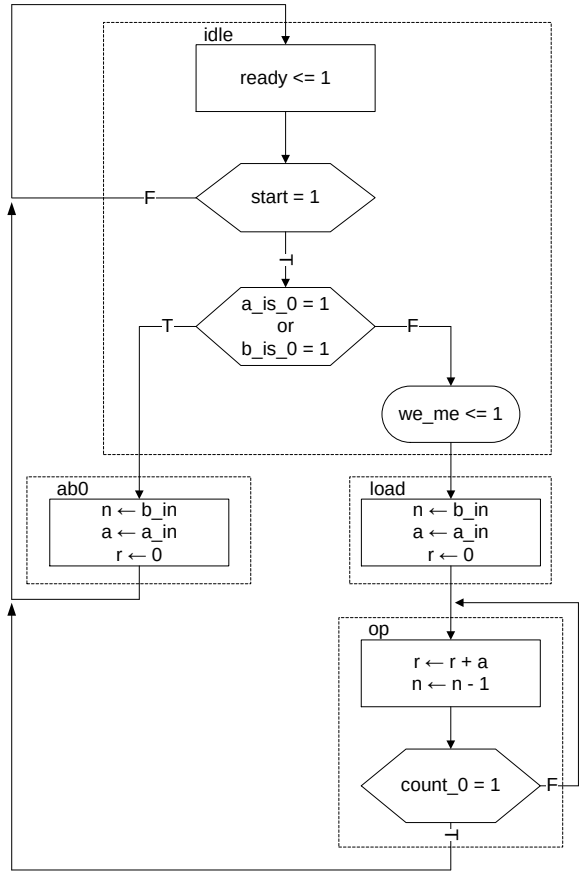
Kako mi nećemo vršiti nikakve optimizacije u pogledu brzine rada i korišćenja resursa, odgovarajući ASMD dijagram će blisko pratiti tok modifikovanog „*Repetitive-Addition*“ algoritma sa slike 4.3. Izgled ASMD dijagrama prikazan je na slici 4.5.

ASMD dijagram koristi ukupno tri registra, *n*, *a* i *r* za smeštanje tri unutrašnje promenljive koje postoje u algoritmu sa slike 4.5. Za razliku od pseudo-koda algoritma, u kojem se u svakom trenutku izvršava tačno jedna naredba, ASMD dijagram dozvoljava izvestan stepen paralelizma. Ukoliko se u nekom od *state* blokova ili uslovnih izlaznih blokova nalazi više od jedne RT operacije, to znači da će se sve ove operacije izvršiti u paraleli, u okviru iste periode takta. Takav slučaj može se videti na slici 4.5, u okviru *ab0*, *load* i *op state* blokova. Na primer, u okviru *op state* bloka nalaze se (planirane su) dve RT operacije, $r \leftarrow r + a$ i $n \leftarrow n - 1$. Pošto se nalaze u okviru istog *state* bloka, obe ove RT operacije će se izvršiti istovremeno, u paraleli. To znači da će unutar *datapth* modula morati da postoje i jedan sabirač i jedan dekrementor, kako bi se izvršavanje ovih operacija moglo realizovati konkurentno. U generalnom slučaju, moguće je planirati veći broj RT operacija za izvršavanje u okviru iste periode klok signala (smeštajući ih u isti *state* ili uslovni izlazni blok), sve dok ne postoji zavisnost između registara koji učestvuju u RT operacijama (*data dependency*) i ukoliko postoji dovoljan broj raspoloživih hardverskih funkcionalnih jedinica za njihovo konkurentno izvršavanje.

Prethodnom analizom zapravo smo dotaknuli problematiku optimizacije dizajna (u smislu brzine rada i potrebnih hardverskih resursa). Vidimo da se planiranjem većeg broja RT operacija za izvršavanje u okviru iste periode klok signala, može skratiti vreme izvršavanja algoritma. Međutim, paralelno izvršavanje većeg broja RT operacija zahteva i istovremeno postojanje većeg broja raspoloživih hardverskih resursa, što povećava veličinu dizajna. Kriterijumi skraćivanja vremena izvršavanja algoritma i

veličine projektovanog digitalnog sistema međusobno su suprotstavljeni. U glavi 5 biće prikazane standardne tehnike za optimizaciju dizajna.

```
if (a_in = 0 or b_in = 0) then
{
   r = 0;
}
else
{
   a = a_in;
   n = b_in;
   r = 0;
   op:  r = r + a;
        n = n - 1;
        if (n = 0) then
        {
           goto stop;
        }
        else
        {
           goto op;
        }
}
r_out = r;
```




*Slika 4.5. ASMD dijagram „Repetitive Addition" množača*

ASMD dijagram sa slike 4.5 ima četiri stanja. U *idle* stanju, ASMD proverava *start* signal. Ako je on aktivan, ASMD proverava da li je neki od ulaznih operanda (portovi *a_in* i *b_in*) jednak nuli. Za ovo se koriste unutrašnji statusni signali *a_is_0* i *b_is_0*, koji će biti generisani unutar *datapath* modula. Ukoliko je barem jedan od ulaznih operanada jednak nuli, operacija množenja je završena jer će i rezultat biti jednak nuli. ASMD u ovom slučaju prelazi u *ab0* stanje, postavlja vrednost rezultata množenja na nulu (*r ← 0* RT operacija u okviru *state* bloka *ab0* stanja) i vraća se u *idle* stanje.

U slučaju da su oba ulazna operanda različita od nule, ASMD prelazi u *load* stanje, u kojem učitava vrednosti ulaznih operanada *a_in* i *b_in* u registre *a* i *n* respektivno, inicijalizuje vrednosti *r* registra na nulu i prelazi u *op* stanje.

U okviru *op* stanja na tekuću vrednost *r* registra, dodaje se vrednost *a* registra. Ovaj proces se ponavlja sve dok vrednost *n* registra ne postane jednaka nuli. Ova informacija će biti prosleđena preko *count_0* unutrašnjeg statusnog signala, koji će biti generisan u okviru *datapath* modula. *Count_0* signal zapravo će biti izlaz komparatora sa nulom na čiji ulaz će biti doveden izlaz dekrementora. Detaljnije o ovome biće reči unutar koraka 5, koji se odnosi na projektovanje *datapath* modula.

Unutar *op* stanja zapravo je implementirana **while** petlja osnovnog „*Repetitive-Addition*“ algoritma za množenje dva broja. Na tekuću vrednost *r* registra, koji je pre ulaska u *op* stanje inicijalizovan na nulu, u svakom periodu klok signala dodaje se vrednost operanda *a_in*. Ovaj postupak se ponavlja *b_in* puta, jer je registar *n* inicijalizovan na vrednost *b_in* i dekrementuje se u svakoj periodi klok signala dok se nalazimo u *op* stanju, tako da će u trenutku napuštanja *op* stanja sadržaj *r* registra biti jednak vrednosti *a_in***b_in*.

Iako ASMD dijagram praktično prati pseudokod algoritma sa slike 4.5, postoje dve razlike. Prva razlika odnosi se na činjenicu da su operacije sabiranja i dekrementacije međusobno nezavisne, pa se mogu izvršiti u istom stanju *op*, u paraleli. Druga razlika odnosi se na vremensko odigravanje RT operacija. Usled toga što nove vrednosti odredišnih registara u RT operacijama postaju dostupne tek u narednoj periodi takta, ukoliko su nam potrebne u tekućoj periodi moramo koristiti *_next* vrednosti koje predstavljaju izlaze kombinacionih mreža u *datapath* modulu, umesto izlaza registara. Obratite pažnju da je ovo upravo slučaj u *op* stanju, gde se koriste *count_0* signal, umesto provere da li je sadržaj registra *n* jednak. O ovome je bilo više reči na predavanjima, pa se zainteresovani čitalac upućuje na materijal sa predavanja za više detalja.

Nakon što smo nacrtali ASMD dijagram, praktično smo završili projektovanje *controlpath* modula našeg digitalnog sistema.

### 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. Kao što je već ranije rečeno u „*Repetitive-Addition*“ algoritmu sa slike 4.3 postoje tri unutrašnje promenljive, *n*, *a* i *r*. Svakoj od njih asociraćemo po jedan registar. Što se tiče veličine ovih registara, s’obzirom da su ulazi *a_in* i *b_in* 8-bitni, usvojićemo da su registri *a* i *n* 8-bitni, a registar *r* je 16-bitan, jer se u njemu smešta rezultat množenja dva 8-bitna broja.

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 4.5 dolazimo do sledeće liste RT operacija:

- $n \leftarrow n, a \leftarrow a, r \leftarrow r$ (u *idle* stanju)
- $n \leftarrow b\_in, a \leftarrow a\_in, r \leftarrow 0$ (u *ab0* stanju)
- $n \leftarrow b\_in, a \leftarrow a\_in, r \leftarrow 0$ (u *load* stanju)
- $n \leftarrow n - 1, r \leftarrow r + a$ (u *op* stanju)

Obratite pažnju da u *idle* stanju takođe postoje RT operacije, ali su one trivijalne ($n \leftarrow n$, $a \leftarrow a$, $r \leftarrow r$), tako da se obično ne navode u ASMD dijagramu radi bolje preglednosti. Generalno, u svakom stanju u kojem se nekom od unutrašnjih registara ne menja vrednost uvek postoji odgovarajuća trivijalna RT operacija (*registar* ← *registar*). Ova RT operacija zapravo modeluje činjenicu da se vrednost registra ne menja dok se ASMD nalazi u posmatranom stanju. Radi bolje preglednosti ASMD dijagrama ove trivijalne RT operacije se uglavnom ne navode, ali se one takođe moraju uzeti u obzir prilikom projektovanja *datapath* modula. Njihovim izostavljanjem projektovaće se funkcionalno neispravan sistem.

Sledeći korak jeste da se RT operacije prisutne u ASMD dijagramu grupišu prema odredišnom registru.

RT operacije kojima je ciljni registar *a*:

1. $a \leftarrow a$ (u *idle* i *op* stanjima)
2. $a \leftarrow a\_in$ (u *load* i *ab0* stanjima)

RT operacije kojima je ciljni registar *n*:

1. $n \leftarrow n$ (u *idle* stanju)
2. $n \leftarrow b\_in$ (u *load* i *ab0* stanjima)
3. $n \leftarrow n - 1$ (u *op* stanju)

RT operacije kojima je ciljni registar *r*:

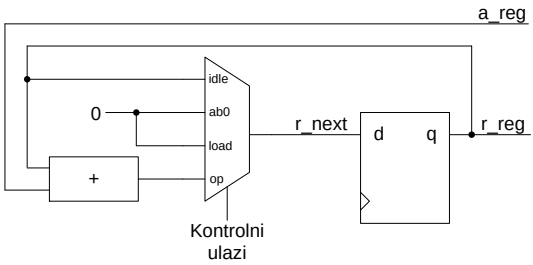
1. $r \leftarrow r$ (u *idle* stanju)
2. $r \leftarrow 0$ (u *laod* i *ab0* stanjima)
3. $r \leftarrow r + a$ (u *op* stanju)

Na osnovu ovih informacija moguće je formirati delove *datapath* modula koji su asocirani svakom od registara u sistemu. Na primer, deo *datapath* modula asociran *r* registru uključuje sledeće standardne kombinacione i sekvencijalne mreže:

- jedan 16-bitni registar, za čuvanje tekućeg sadržaja *r* registra, kada je konačni automat unutar *controlpath* modula u *idle* stanju,
- jedan 8-bitni sabirač, za izvršavanje operacija sabiranja tekućih vrednosti registara *r* i *a*, koja se izvršava kada je konačni automat unutar *controlpath* modula u *op* stanju,
- 16-bitne konstante 0, koja treba da se upiše u registar *r* kada je konačni automat unutar *controlpath* modula u *load* i *ab0* stanjima,

- jedan 16-bitni multiplekser 4-na-1, za dovođenje pravilne vrednosti koja treba da se upiše u registar *r* u narednom taktu, u zavisnosti od stanja u kome se nalazi konačni automat unutar *controlpath* modula.

Struktura dela *datapath* modula koji je asociran registru *r* prikazana je na slici 4.6.

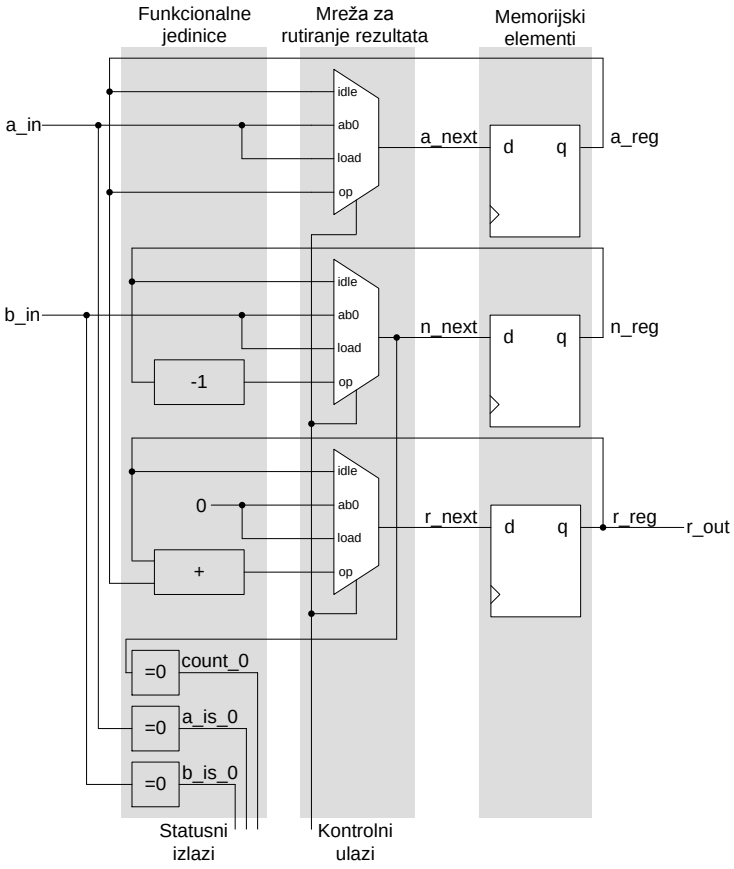



*Slika 4.6. Deo datapath modula asociran r registru*

Sa slike 4.6 može se videti da su ulazi u multiplekser označeni sa simboličkim imenima stanja ASMD dijagrama (*idle, ab0, load, op*). To znači da se kao selekcioni signal multipleksera zapravo može koristiti tekuće stanje konačnog automata iz *controlpath* modula! Ovo znači da je signal „Kontrolni ulazi“ sa slike 4.6 zapravo jednak signalu tekućeg stanja konačnog automata koji predstavlja izlaz iz registra stanja.

Struktura celog *datapath* modula prikazana je na slici 4.7. Kao što se sa slike 4.7 vidi, čitav *datapath* se sastoji iz tri bloka:

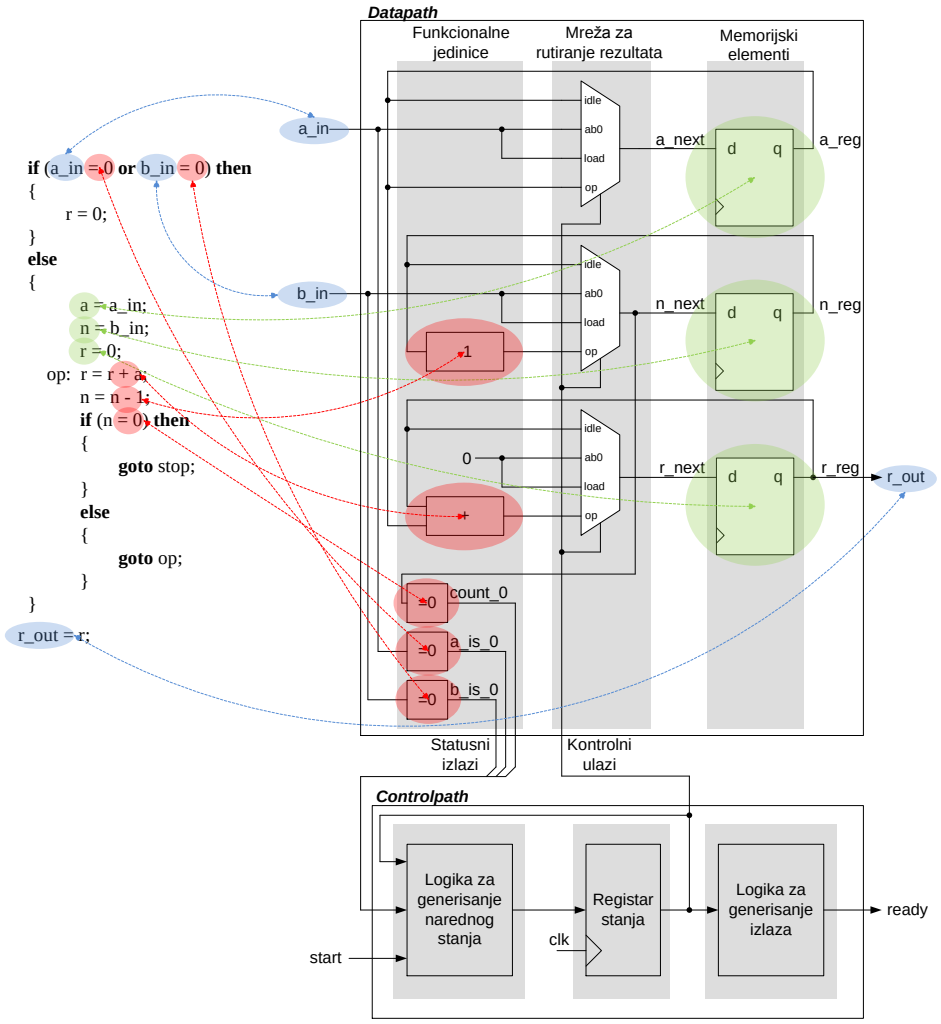
- **Mreže za rutiranje rezultata** – ovaj blok čine tri multipleksera koja su zadužena za rutiranje potrebnih vrednosti signala koje će biti upisane u tri registra, koji čine blok memorijskih elemenata. Koje vrednosti će biti upisani u ove registre zavisi od vrednosti signala „Kontrolni ulazi“, koji kao što je malopre rečeno zapravo predstavlja tekuće stanje konačnog automata koji se nalazi unutar *controlpath* modula.
- **Funkcionalnih jedinica** – ovaj blok čine sledeće kobinacione mreže: jedan 8-bitni sabirač, jedan 8-bitni dekrementor i tri 8-bitna komparatora sa nulom. Prva dva elementa služe za implementaciju svih aritmetičkih operacija koje su prisutne u modifikovanom „*Repetitive-Addition*“ algoritmu množenja. Preostala tri elementa služe za implementaciju relacionih operatora koji su prisutni u modifikovanom „*Repetitive-Addition*“ algoritmu množenja.
- **Memorijskih elemenata** – ovaj blok čine tri registra sa paralelnim ulazom i izlazom. Dva registra su 8-bitna, dok je treći 16-bitan. Ovi registri služe sa memorisanje tekućih vrednosti promenljivih koje su prisutne u modifikovanom „*Repetitive-Addition*“ algoritmu množenja.




*Slika 4.7. Struktura datapath modula „Repetitive-Addition“ množača*

Ukoliko uporedimo *datapath* modula „*Repetitive-Addition*“ množača sa slike 4.7 sa generičkim *datapath* modulom sa slike 4.1, možemo primetiti da u *datapath* modulu „*Repetitive-Addition*“ množača nedostaje mreža za rutiranje ulaznih podataka funkcionalnih jedinica. Ovo nije greška, već posledica toga da prilikom projektovanja ASMD dijagrama „*Repetitive-Addition*“ množača nismo izvršili nikakve optimizacije u pogledu korišćenja funkcionalnih jedinica. Konkretno, u našem dizajnu ne postoji situacija da se ista funkcionalna jedinica koristi u dva različita ASMD stanja, sa različitim ulaznim signalima, odnosno ne postoji delenje resursa (*Resource Sharing*). Zbog toga nema ni potrebe za mrežom za rutiranje ulaznih podataka, jer se na ulaz svake od prisutnih funkcionalnih jedinica uvek dovode isti podaci.

Na slici 4.8 prikazana je struktura kompletnog digitalnog sistema za implementaciju „*Repetitive-Addition*“ algoritma množenja dva broja.




*Slika 4.8. Način mapiranja ključnih delova „Repetitive-Addition“ algoritma u hardverske module primenom RT metodologije*

Na slici 4.8 je takođe prikazan i način na koji se ključni delovi „*Repetitive-Addition*“ algoritma mapiraju u odgovarajuće hardverske module primenom RT metodologije:

- Ulazni argumenti algoritma mapiraju se u ulazni interfejs podataka. Na slici 4.8 može se videti kako se ulazni argumenti *a_in* i *b_in* „*Repetitive-Addition*“ algoritma, označeni plavom bojom, mapiraju u ulazne portove podataka *a_in* i *b_in datapath* modula projektovanog sistema.
- Izlazni argumenti algoritma mapiraju se u izlazni interfejs podataka. Na slici 4.8 može se videti kako se izlazni argument *r_out* „*Repetitive-Addition*“

algoritma, označen plavom bojom, mapira u izlazni port podataka *r_out datapath* modula projektovanog sistema.

- Unutrašnje promenljive algoritma mapiraju se u registre. Na slici 4.8 zelenom bojom označene su tri unutrašnje promenljive koje postoje u „*Repetitive-Addition*“ algoritmu, *a*, *n* i *r*. U blok dijagramu projektovanog digitalnog sistema istom bojom označena su tri registra koja služe za čuvanje trenutnih vrednosti ove tri promenljive.
- Sve operacije unutar algoritma mapiraju se u odgovarajuće funkcionalne jedinice. Na slici 4.8 crvenom bojom označene su operacije koje postoje u „*Repetitive-Addition*“ algoritmu (tri operacije poređenja sa nulom, jedna operacija sabiranja i jedna operacija dekrementacije). U blok dijagramu projektovanog digitalnog sistema isto bojom označene su odgovarajuće funkcionalne jedinice koje implementiraju ove operacije u hardveru. Sa slike 4.8 možemo videti da svaka identifikovana operacija u „*Repetitive-Addition*“ algoritmu ima svoju ekskluzivnu funkcionalnu jedinicu, odnosno da u našem dizajnu digitalnog sistema za implementaciju „*Repetitive-Addition*“ algoritma nema deljenja resursa.
- Redosled izvršavanja operacija, definisan pseudo-kodom algoritma na levoj strani slike 4.8, unutar projektovanog digitalnog sistema implementiran je u stanjima i prelazima između stanja konačnog automata koji se nalazi u *controlpath* modulu.

**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 „*Repetitive-Addition*“ algoritam množenja dva broja u hardveru. Ovaj model se može napisati na različite načine. Na primer, mogli bismo koristiti hijerarhijski stil modelovanja, gde bismo u odvojenim entitetima modelovali *datapath* i *controlpath* module, a zatim bismo u još jednom dodatnom entitetu izveli njihovo povezivanje u celokupan sistem, korićenjem strukturnog stila modelovanja. Ovaj pristup ima smisla ukoliko je složenost *datapath* i *controlpath* modula velika. U slučaju kada su oni jednostavni, kao što je slučaj kod „*Repetitive-Addition*“ algoritma, ima smisla modelovati ih unutar jednog, zajedničkog entiteta. Takođe, prilikom pisanja *datapath* i *controlpath* modela, možemo birati između višeprocesnog, dvoprocesnog ili jednoprocesnog stila modelovanja, jer se i *datapath* i *controlpath* moduli sastoje iz većeg broja kombinacionih i sekvencijalnih mreža.

U nastavku je prikazan VHDL model projektovanog digitalnog sistema za implementaciju „*Repetitive-Addition*“ algoritma množenja dva broja koji modeluje *datapath* i *controlpath* module unutar istog entiteta, koristeći višeprocesni stil modelovanja.

```
entity repetitive_addition_mult is
  generic (WIDTH: positive := 8);
```

```
  port (
    -- Clocking and reset interface
    clk:    in std_logic;
    reset:  in std_logic;
    -- Input data interface
    a_in:   in std_logic_vector(WIDTH-1 downto 0);
    b_in:   in std_logic_vector(WIDTH-1 downto 0);
    -- Output data interface
    r_out: out std_logic_vector(2*WIDTH-1 downto 0);
    -- Command interface
    start:  in std_logic;
    -- Status interface
    ready: out std_logic);
end entity;

architecture mult_seg_arch of repetitive_addition_mult is
  type state_type is (idle, ab0, load, op);
  signal state_reg, state_next: state_type;
  signal a_is_0, b_is_0, count_0: std_logic;
  signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
  signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
  signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
  signal adder_out: unsigned (2*WIDTH-1 downto 0);
  signal sub_out: unsigned(WIDTH-1 downto 0);
begin
  -- control path: state register
  process (clk, reset)
  begin
    if reset = '1' then
      state_reg <= idle;
    elsif (clk'event and clk = '1') then
      state_reg <= state_next;
    end if;
  end process;

  -- control path: next-state/output logic
  process (state_reg, start, a_is_0, b_is_0, count_0)
  begin
    case state_reg is
      when idle =>
        if start = '1' then
          if (a_is_0 = '1' or b_is_0 = '1') then
            state_next <= ab0;
          else
            state_next <= load;
          end if;
        else
          state_next <= idle;
        end if;
      when ab0 =>
        state_next <= idle;
      when load =>
        state_next <= op;
      when op =>
        if count_0 = '1' then
          state_next <= idle;
        else
          state_next <= op;
        end if;
```

```
  end case;
end process;

-- control path: output logic
ready <= '1' when state_reg = idle else '0';

-- datapath: data register
process (clk, reset)
begin
  if reset = '1' then
    a_reg <= (others => '0');
    n_reg <= (others => '0');
    r_reg <= (others => '0');
  elsif (clk'event and clk='1') then
    a_reg <= a_next;
    n_reg <= n_next;
    r_reg <= r_next;
  end if;
end process;

-- datapath: routing multiplexer
process (state_reg, a_reg, n_reg, r_reg, a_in, b_in, adder_out, sub_out)
begin
  case state_reg is
    when idle =>
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
    when ab0 =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others = > '0');
    when load =>
      a_next <= unsigned (a_in);
      n_next <= unsigned (b_in);
      r_next <= (others => '0');
    when op =>
      a_next <= a_reg;
      n_next <= sub_out;
      r_next <= adder_out;
  end case;
end process;

-- datapath: functional units
adder_out <= (conv_std_logic_vector(0, WIDTH) & a_reg) + r_reg;
sub_out <= n_reg - 1;

-- datapath: status
a_is_0 <= '1' when a_in = conv_std_logic_vector(0, WIDTH) else '0';
b_is_0 <= '1' when b_in = conv_std_logic_vector(0, WIDTH) else '0';
count_0 <= '1' when n_next = conv_std_logic_vector(0, WIDTH) else '0';

-- datapath: output
r_out <= std_logic_vector(r_reg);
end mult_seg_arch;
```

*Slika 4.9. VHDL model digitalnog sistema za implementaciju „Repetitive-Addition“ algoritma množenja dva broja*

**Primer 4.2: Projektovanje „*Add-and-Shift*“ množača neoznačenih brojeva**

Množač dva neoznačena broja, baziran na sukcesivnom sabiranju, koji je bio razmatran u prethodnom primeru, iako baziran na jednostavnom algoritmu, nije pogodan za praktičnu upotrebu jer je vreme potrebno za izračunavanje rezultata množenja reda $O(2^n)$, gde je $n$ broj bita koji se koriste za predstavljanje vrednosti brojeva koji se množe. Na ovom mestu ćemo razmotriti drugi, efikasniji, algoritam množenja koji je baziran na metodi sabiranja i pomeranja („*Add-and-Shift*“).

Na primer, množenje dva 4-bitna broja pomoću „*Add-and-Shift*“ metode prikazano je na slici 4.10.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ | Operand 1 |
| X | | | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ | Operand 2 |
| | | | | | $a_3b_0$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ | |
| | | | | $a_3b_1$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ | | |
| | | | $a_3b_2$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ | | | |
| + | | $a_3b_3$ | $a_2b_3$ | $a_1b_3$ | $a_0b_3$ | | | | |
| | $y_7$ | $y_6$ | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | Proizvod |

*Slika 4.10. Množenje dva 4-bitna broja pomoću „Add-and-Shift“ metode*

Množenje dva 4-bitna broja pomoću „*Add-and-Shift*“ metode uključuje sledeće korake:

1. Pomnoži cifre Operanda 2 ($b_3$, $b_2$, $b_1$, $b_0$) sa Operandom 1 ($A$) jednu po jednu da bi se dobili sledeći proizvodi: $b_3*A$, $b_2*A$, $b_1*A$, $b_0*A$. Proizvod $b_i*A$ se računa na sledeći način

$$b_i*A = (a_3 \cdot b_i, a_2 \cdot b_i, a_1 \cdot b_i, a_0 \cdot b_i)$$

   gde operator $\cdot$ predstavlja logičku operaciju konjunkcije.

2. Pomeri proizvod $b_i*A$ za $i$ pozicije u levo.
3. Saberi pomerene članove $b_i*A$ kako bi dobio konačni rezultat.

Opisana „*Add-and-Shift*“ metoda lako se može prevesti u odgovarajući sekvencijalni algoritam. Možemo procesirati jednu cifru Operanda 2 ($b_i$) u jednom trenutku i ponoviti postupak za sve cifre Operanda 2 ($B$). U svakoj iteraciji, izračunaćemo po jedan proizvod $b_i*A$, pomeriti ga za $i$ mesta u levo, i zatim ga dodati na tekuću vrednost proizvoda. S’obzirom da je $b_i$ zapravo binarna cifra, može imati samo dve vrednosti, 0 ili 1. Umesto da računamo proizvod $b_i*A$ možemo iskoristiti *if* naredbu da proverimo vrednost cifre $b_i$ i ukoliko je ona jednaka 1 dodati pomerenu vrednost Operanda 1 ($A$) na tekuću vrednost proizvoda. Pretpostavimo da su vrednosti dva broja koja je potrebno

pomnožiti zadata pomoću ulaza širine *WIDTH* bita *a_in* i *b_in*. Tada se „*Add-and-Shift*“ metoda može prikazati u vidu algoritma sa slike 4.11.

```
n = 0;
p = 0;
while (n != WIDTH)
{
   if (b_in(n) = 1) then
   {
      p = p + (a_in << n);
   }
   n = n + 1;
}
r_out = p;
```

*Slika 4.11. „Add-and-Shift“ algoritam za množenje dva binarna broja*

Hardverska implementacija operacija indeksiranja (na primer, *b_in(n)*) i uopštenog pomeranja (na primer, *a_in* << *n*) je „skupa“. Međutim, ove operacije se mogu izbeći ukoliko u svakoj iteraciji operande *a_in* i *b_in* pomerimo za jedno mesto u levo. Modifikovani „*Add-and-Shift*“ algoritam prikazan je na slici 4.12.

```
a = a_in;
b = b_in;
n = WIDTH;
p = 0;
while (n != 0)
{
   if (b(0) = 1) then
   {
      p = p + a;
   }
   a = a << 1;
   b = b >> 1;
   n = n - 1;
}
r_out = p;
```

*Slika 4.12. Modifikovani „Add-and-Shift“ algoritam za množenje dva binarna broja, pogodan za hardversku implementaciju*

U algoritmu sa slike 4.12 koriste se ukupno četiri unutrašnje promenljive. Promenljiva *p* se koristi za čuvanje tekućeg rezultata množenja, a promenljiva *n* za čuvanje broja iteracija koje su izvršene. Obratite pažnju da je smer brojanja iteracija obrnut u odnosu na algoritam sa Slike 4.11, takođe zbog lakše hardverske implementacije (poređenja sa nulom umesto sa *n*). Promenljiva *a* koristi se za čuvanje vrednosti Operanda 1 (*A*) koja se u svakoj iteraciji pomera za jedno mesto u levo. Promenljiva *b* koristi se za čuvanje vrednosti Operanda 2 (*B*). Vrednost promenljive *b* u svakoj iteraciji se pomera za jedno mesto u desno, što obezbeđuje da u *i*-toj iteraciji cifra $b_i$ uvek bude na LSB poziciji (odnosno, *b*(0)).

**Korak 1: Eliminacija naredbi ponavljanja iz algoritma**

Da bi smo mogli da nacrtamo ASMD dijagram koji odgovara algoritmu koji implementiramo u hardveru, prvo je neophodno zameniti **while** petlju sa odgovarajućim **if** and **goto** naredbama. Nakon ove zamene, modifikovani „*Add-and-Shift*“ algoritam opisan je pomoću sledećeg pseudo-koda.

```
    a = a_in;
    b = b_in;
    n = WIDTH;
    p = 0;
op: if (b(0) = 1) then
    {
       p = p + a;
    }
    a = a << 1;
    b = b >> 1;
    n = n - 1;
    if (n != 0) then
    {
       goto op;
    }
     r_out = p;
```

*Slika 4.13. Modifikovani „Add-and-Shift“ algoritam za množenje dva binarna broja, kod kojega je **while** naredba zamenjena **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 4.13., možemo videti da postoje dve ulazne promenljive, *a_in* i *b_in*. Najjednostavniji način za formiranje ulaznog interfejsa podataka jeste da se svakoj od ovih promenljivih pridruži po jedan višebitni ulazni port. Kako u postavci zadatka nije eksplicitno naglašeno koji su opsezi mogućih vrednosti ulaznih promenljivih *a_in* i *b_in*, možemo ih odabrati proizvoljno. U ovom primeru razvićemo parametrizovani model, pa ćeo koristiti *std_logic_vector* ulazne portove širine *WIDTH* bita, *a_in* i *b_in*. Parametar *WIDTH* omogućiće nam da za svaku instancu modula možemo specificirati različite širine ulaznih promenljivih.

Izlazni interfejs podataka formira se tako što se u algoritmu koji se mapira u hardver identifikuju sve izlazne promenljive. U našem slučaju, analizom algoritma sa slike 4.13., možemo videti da postoji samo jedna izlazna promenljiva, *r_out*. Kao i u slučaju ulaznog interfejsa podataka, najjednostavniji način za formiranje izlaznog interfejsa podataka jeste da se svakoj od ovih od izlaznih promenljivih pridruži po jedan višebitni izlazni port. Za razliku od broja bita ulaznih višebitnih portova, koji se mogu birati proizvoljno, ako to nije drugačije naglašeno, u slučaju izbora širine izlaznih portova ona

je obično vezana sa odabranom širinom ulaznih portova, preko neke funkcionalne veze. U našem primeru širina izlaznog porta *r_out*, preko kojega će se prenositi rezultat množenja dva broja, mora biti barem jednaka zbiru širina ulaznih portova preko kojih se prenose vrednosti operanda *a_in* i *b_in*

$$\textbf{length}(r\_out) \geq \textbf{length}(a\_in) + \textbf{length}(b\_in)$$

Ovo je stoga što je poznato da proizvod *n*-bitnog i *m*-bitnog binarnog broja u najgorem slučaju može biti *n*+*m*-bitni binarni broj. U ovom primeru funkcionalna veza koja povezuje dimenzije ulaznih portova za podatke sa dimenzijama izlaznih portova za podatke je relativno jednostavna. U opštem slučaju pronalaženje ove funkcionalne veze može biti vrlo težak zadatak.

Kako smo mi ograničili širinu ulaznih portova *a_in* i *b_in* na *WIDTH* bita, primenom gornje funkcionalne veze možemo zaključiti da minimalna širina izlaznog porta *r_out* mora biti barem 2**WIDTH* bita. Imajući ovo u vidu, izlazni port *r_out* biće implementiran kao *std_logic_vector* izlazni port širine 2**WIDTH* bita.

Komandni interfejs nam omogućava da kontrolišemo i upravljamo radom digitalnog sistema koji implementira željeni algoritam (u našem slučaju to će biti „*Add-and-Shift*“ algoritam množenja dva broja) u hardveru. Složenost komandnog interfejsa u dobroj meri zavisi i od složenosti samog algoritma koji se implementira. Kako je i „*Add-and-Shift*“ algoritam vrlo jednostavan, ponovo možemo koristiti najjednostavniji mogući komandni interfejs, koji se sastoji samo od jednog 1-bitnog ulaznog porta, *start*. Kada je ulazni port *start* postavljen na jedinicu, digitalni sistem treba da započne operaciju množenja dva binrarna broja. Dok se ulazni port *start* na nuli, digitalni sistem treba da bude u stanju mirovanja.

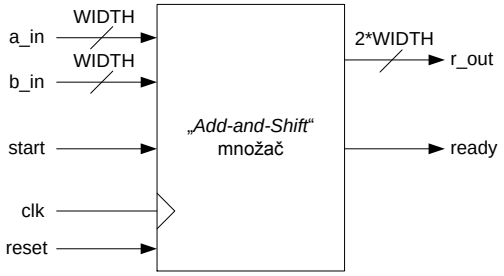
Statusni interfejs nam omogućava da dobijemo informacije o trenutnom stanju digitalnog sistema koji implementira željeni algoritam. Složenost ovog interfejsa takođe u velikoj meri zavisi od složenosti samog algoritma koji se implementira. U slučaju implementacije „*Add-and-Shift*“ algoritma ponovo možemo 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 dva binarna broja). 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 dva binarna broja koje se još uvek u toku, tako da ne može započeti novu operaciju množenja).

Ovim je proces definisanja potrebnih interfejsa digitalnog sistema za hardversku implementaciju „*Add-and-Shift*“ algoritma završen. Projektovani parametrizovani digitalni sistem imaće sledeće interfejse:

- Ulazni interfejs podataka – sastoji se iz dva *std_logic_vector* ulazna porta širine *WIDTH* bita, *a_in* i *b_in*
- Izlazni interfejs podataka – sastoji se iz jednog *std_logic_vector* izlaznog porta širine 2**WIDTH* bita, *r_out*

- 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 4.14. prikazan je kompletan interfejs digitalnog sistema koji implementira „*Add-and-Shift*“ algoritam množenja dva broja.

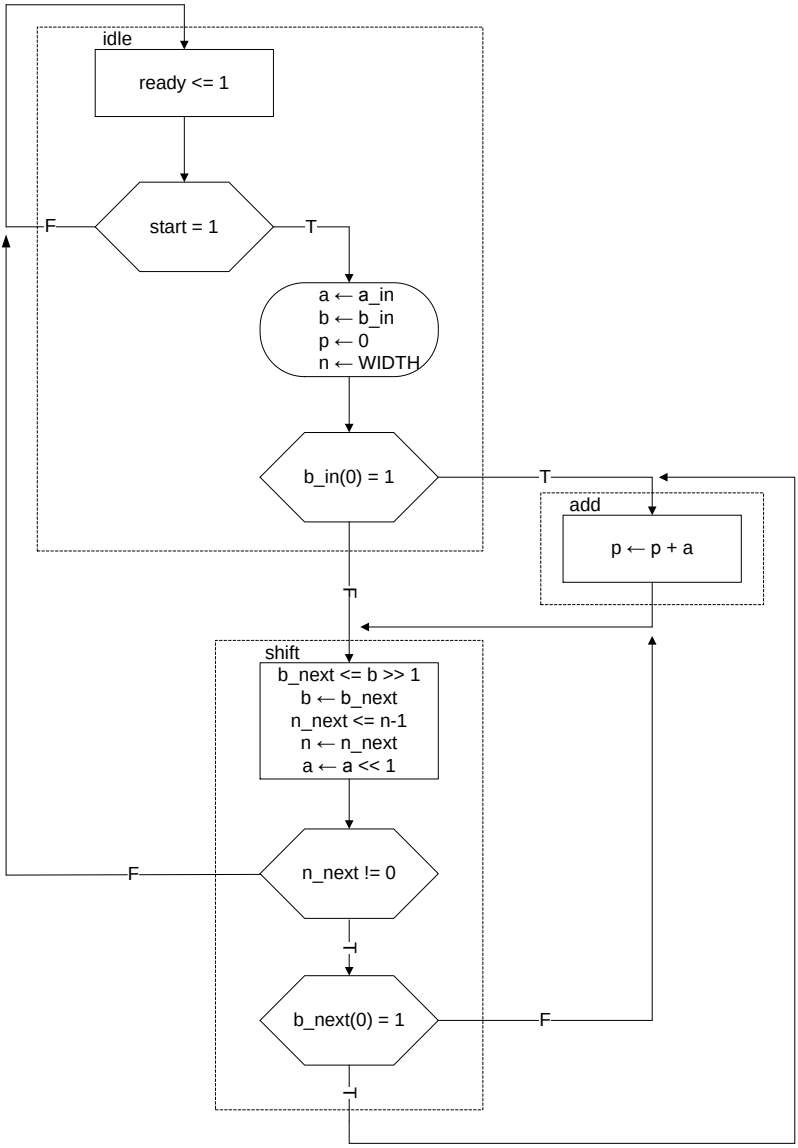



*Slika 4.14. Interfejs digitalnog sistema koji implementira „Add-and-Shift“ algoritam*

**Korak 3: Projektovanje *controlpath* modula**

Algoritam sa slike 4.13. lako se može prevesti u ASMD dijagram, koji je prikazan na slici 4.15.

ASMD dijagram ima tri stanja. U *idle* stanju, ASMD proverava *start* signal. Ako je on aktivan, ASMD učitava početne vrednosti u registre i prelazi ili u *add* ili u *shift* stanje. Ako je LSB bit Operanda 2 jednak ‘1’, ASMD prelazi u *add* stanje, u kojem se pomereni Operand 1 (*A*) dodaje na trenutnu vrednost proizvoda. U protivnom, ASMD prelazi u *shift* stanje, u kojem se Operand 1 (*A*) pomera za jedno mesto u levo, Operand 2 (*B*) se pomera za jedno mesto u desno, a brojač (*n*) se umanjuje za 1. Ovaj proces se ponavlja sve dok brojač (*n*) ne dođe do nule.




*Slika 4.15. ASMD dijagram „Add-and-Shift“ množača*

Iako ASMD dijagram praktično prati pseudokod algoritma sa Slike 4.13, postoje dve razlike. Prva razlika odnosi se na činjenicu da su operacije pomeranja i dekrementacije međusobno nezavisne, pa se mogu izvršiti u istom stanju, konkurentno. Druga razlika odnosi se na vremensko odigravanje RT operacija. Usled toga što nove vrednosti odredišnih registara u RT operacijama postaju dostupne tek u narednoj periodi takta, ukoliko su nam potrebne u tekućoj periodi moramo koristiti *_next* vrednosti koje

predstavljaju izlaze kombinacionih mreža u *datapath* modulu, umesto izlaza registara. Obratite pažnju da je ovo upravo slučaj u *shift* stanju, gde se koriste *b_next(0)* i *n_next* vrednosti, umesto *b(0)* i *n* vrednosti. O ovome je takođe bilo reči na predavanjima.

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. Kao što je već ranije rečeno u „*Add-and-Shift*" algoritmu sa slike 4.13 postoje četiri unutrašnje promenljive, *a, b, n* i *p*. Svakoj od njih asociraćemo po jedan registar. Što se tiče veličine ovih registara, s'obzirom da su ulazi *a_in* i *b_in* širine *WIDTH* bita, usvojićemo da su registri *a, b* i *n* takođe širine *WIDTH* bita, dok je registar *p* širine 2**WIDTH* bita, jer se u njemu smešta rezultat množenja dva binarna broja širine *WIDTH* bita.

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 4.15 dolazimo do sledeće liste RT operacija:

- $a \leftarrow a\_in, b \leftarrow b\_in, , n \leftarrow WIDTH, p \leftarrow 0$ (u *idle* stanju)
- $a \leftarrow a, b \leftarrow b, n \leftarrow n, \ p \leftarrow p + a$ (u *add* stanju)
- $a \leftarrow a << 1, b \leftarrow b >> 1, n \leftarrow n - 1, \ p \leftarrow p$ (u *shift* stanju)

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

RT operacije kojima je ciljni registar *a*:

1. $a \leftarrow a\_in$ (u *idle* stanju)
2. $a \leftarrow a$ (u *add* stanju)
3. $a \leftarrow a << 1$ (u *shift* stanju)

RT operacije kojima je ciljni registar *b*:

1. $b \leftarrow b\_in$ (u *idle* stanju)
2. $b \leftarrow b$ (u *add* stanju)
3. $b \leftarrow b >> 1$ (u *shift* stanju)

RT operacije kojima je ciljni registar *n*:

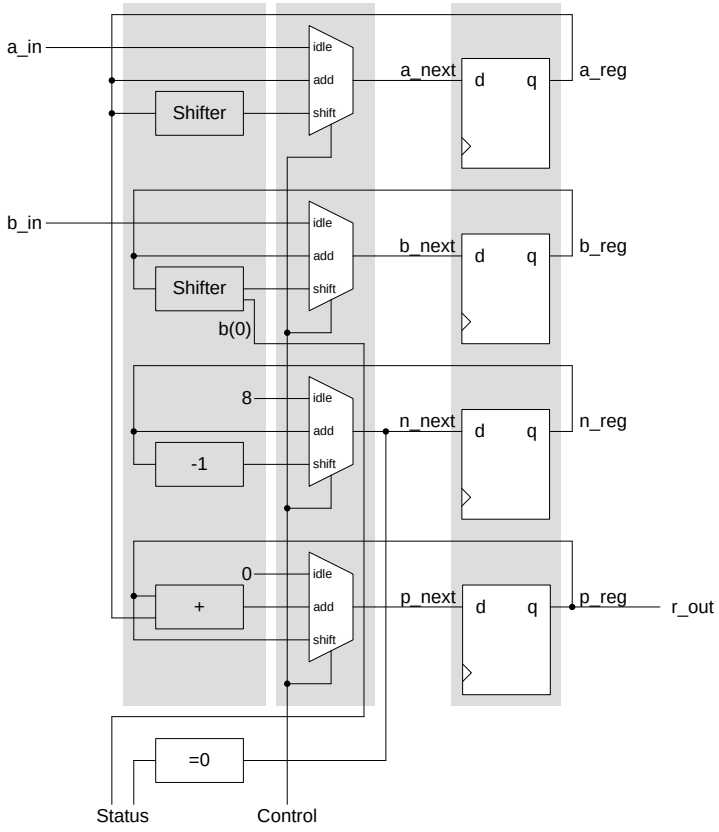
1. $n \leftarrow$ WIDTH (u *idle* stanju)
2. $n \leftarrow n$ (u *add* stanju)

3. $n \leftarrow n - 1$ (u *shift* stanju)

RT operacije kojima je ciljni registar *p*:

1. $p \leftarrow 0$ (u *idle* stanju)
2. $p \leftarrow p + a$ (u *add* stanju)
3. $p \leftarrow p$ (u *shift* stanju)

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 4.16.




*Slika 4.16. Datapath modul „Add-and-Shift“ množača*

Obratite pažnju da se blokovi koji implementiraju operacije pomeranja za jedno mesto u levo i u desno (*Shifter* moduli na slici 4.16) mogu realizovati bez korišćenja bilo kakvih resursa, odgovarajućim povezivanjem ulaznih i izlaznih signala.

### 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 „*Add-and-Shift*“ algoritam množenja u hardveru. Kao i u slučaju pisanja HDL modela „*Repetitive-Addition*“ množača i u slučaju „*Add-and-Shift*“ množača ovaj model se može napisati na različite načine. U nastavku je prikazan VHDL model projektovanog digitalnog sistema za implementaciju „*Add-and-Shift*“ algoritma množenja dva broja koji modeluje *datapath* i *controlpath* module unutar istog entiteta, koristeći dvoprocesni stil modelovanja.

```
entity add_and_shift_mult is
  generic (WIDTH: integer := 8)
  port (
    -- Clocking and reset interface
    clk:    in std_logic;
    reset:  in std_logic;
    -- Input data interface
    a_in:   in std_logic_vector(WIDTH-1 downto 0);
    b_in:   in std_logic_vector(WIDTH-1 downto 0);
    -- Output data interface
    r_out: out std_logic_vector(2*WIDTH-1 downto 0);
    -- Command interface
    start:  in std_logic;
    -- Status interface
    ready: out std_logic);
end entity;

architecture two_seg_arch of add_and_shift_mult is
  type state_type is (idle, add, shift);
  signal state_reg, state_next: state_type;
  signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
  signal b_reg, b_next: unsigned(WIDTH-1 downto 0);
  signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
  signal p_reg, p_next: unsigned(2*WIDTH-1 downto 0);
begin
  -- state and data registers
  process (clk, reset)
  begin
    if reset = '1' then
      state_reg <= idle;
      a_reg <= (others => '0');
      b_reg <= (others => '0');
      n_reg <= (others => '0');
      p_reg <= (others => '0');
    elsif (clk'event and clk = '1') then
      state_reg <= state_next;
      a_reg <= a_next;
      b_reg <= b_next;
      n_reg <= n_next;
```

```
    p_reg <= p_next;
  end if;
end process;

-- combinatorial circuits
process (state_reg, start, a_in, b_in, a_reg, b_reg, p_reg, n_reg, n_next, b_next)
begin
  -- default assignments
  a_next <= a_reg;
  b_next <= b_reg;
  n_next <= n_reg;
  p_next <= p_reg;
  ready <= '0';

  case state_reg is
    when idle =>
      ready <= '1';
      if start = '1' then
        a_next <= a_in;
        b_next <= b_in;
        p_next <= conv_std_logic_vector(0, 2*WIDTH);
        n_next <= conv_std_logic_vector(WIDTH, WIDTH);
        if (b_in(0) = '1') then
          state_next <= add;
        else
          state_next <= shift;
        end if;
      else
        state_next <= idle;
      end if;
    when add =>
      p_next <= p_reg + (conv_std_logic_vector(0, WIDTH)&a_reg);
      state_next <= shift;
    when shift =>
      b_next <= '0'&b_reg(WIDTH-1 downto 1);
      n_next <= n_reg - 1;
      a_next <= a_reg(WIDTH-2 downto 0)&'0';
      if n_next /= conv_std_logic_vector(0, WIDTH) then
        if b_next(0) = '1' then
          state_next <= add;
        else
          state_next <= shift;
        end if;
      else
        state_next <= idle;
      end if;
  end case;
end process;

-- system output
r_out <= std_logic_vector(p_reg);
end two_seg_arch;
```

Na kraju, procenimo brzinu rada „*Add-and-Shift*“ množača. Analizom „*Add-and-Shift*“ algoritma može se zaključiti da je potreban broj iteracija za kompletiranje operacije množenja jednak širini ulaznih operanada, *WIDTH*. Analizom ASMD dijagrama sa slike

4.15, može se zaključiti da svaka iteracija u najgorem slučaju zahteva dva perioda klok signala (u slučaju da je tekući bit Operanda 2 jednak '1' ASMD dijagram prolazi kroz dva stanja, *add* i *shift*, a u slučaju da je tekući bit Operanda 2 jednak '0' ASMD dijagram prolazi kroz jedno stanje, *shift*. Za *WIDTH*-bitne ulazne operande, za izračunavanje proizvoda potrebno je *2*WIDTH+1* taktova u najgorem slučaju (kada je vrednost Operanda 2 jednaka „1...1"), odnosno *WIDTH+1* taktova u najboljem slučaju (kada je vrednost Operanda 2 jednaka „0...0"). Ovo je daleko bolje od $2^{WIDTH}+1$ taktova, koliko je potrebno množaču baziranom na „*Repetitive-Addition*" algoritmu.

**Primer 4.3: Projektovanje množača matrica neoznačenih brojeva**

U ovom primeru pokazaćemo kako se primenom RT metodologije može projektovati digitalni sistem koji implementira operaciju množenja dve matrice. Matrične operacije su vrlo česte u numeričkim algoritmima, a matrično množenje je jedna od elementarnih matričnih operacija koja je prilično zahtevna po pitanju vremena potrebnog za njeno izvršavanje na procesorima opšte namene, te je od interesa projektovani poseban digitalni sistem koji će biti specijalizovan za izvođenje ove operacije. Ukoliko bi se ovaj digitalni sistem opremio odgovarajućim standardnim SoC (*System on Chip*) komunikacionim interfejsima (kao što su na primer AXI ili PLB), mogao bi se koristiti kao hardverski akcelerator matričnog množenja u mikroprocesorski baziranim sistemima.

Sama operacija matričnog množenja definisana je na sledeći način. Pretpostavimo da imamo dve matrice, matricu *A* dimenzija *n*x*m* i matricu *B* dimenzija *m*x*p*. Proizvod ove dve matrice je matrica *C*, dimenzija *n*x*p*, pri čemu se vrednost svakog elementa matrice *C*, $c_{i,j}$, računa pomoću sledeće formule

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}, \; 1 \le i \le n, 1 \le j \le p.$$

Kao što se može videti iz prethodnog iz definicionog izraza, postupak množenja dve matrice uključuje izračunavanje vrednosti $n \cdot p$ elemenata matrice *C*, $c_{i,j}$, pri čemu se vrednost svakog od elemenata dobija kao suma od ukupno *m* proizvoda elemenata $a_{i,k}$ i $b_{k,j}$ matrica *A* i *B*, koji se nalaze u *i*-toj vrsti matrice *A*, odnosno *j*-toj koloni matrice *B*.

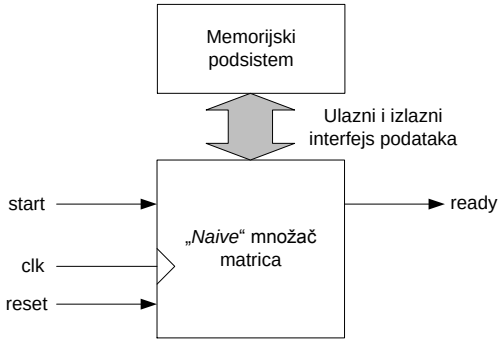
Opisani postupak množenja dve matrice lako se može prevesti u odgovarajući sekvencijalni algoritam. Možemo sekvencijalno prolaziti kroz elemente matrice *C*, i računati njihove vrednosti sekvencijalno sabirajući *m* proizvoda elemenata matrica *A* i *B*. Pretpostavimo da su dimenzije matrica *A* i *B* zadate preko parametara *n*, *m* i *p*. Operacija množenja ove dve matrice može se prikazati u vidu algoritma sa slike 4.17. Ovo je takozvani „*Naive*" algoritam množenja matrica.

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

*Slika 4.17. „Naive“ algoritam za množenje dve matrice*

Ulazne parametre algoritma sa slike 4.17 predstavljaju dve matrice *A* i *B*, kao i informacije o njihovim dimenzijama, *n*, *m* i *p*. Izlazni argument „*Naive*“ algoritma za množenje matrica predstavlja matrica *C*. Pre početka projektovanja digitalnog sistema koji implementira „*Naive*“ algoritam množenja matrica, moramo definisati način na koji će elementi ulaznih matrica *A* i *B*, kao i rezultujuće matrice *C* biti smešteni, i kako će im biti pristupano unutar digitalnog sistema. Najprirodniji način za njihovo smeštanje je korišćenje odgovarajućih memorija, kojima će projektovani digitalni sistem pristupati u toku svog rada. Organizacija čitavog sistema prikazana je na slici 4.18.

Ovaj primer je interesantan i zbog toga što ulazi u algoritam nisu više skalarni objekti, kao što je do sada bi slučaj, već je reč o složenijim strukturama podataka, matricama. Ovakve strukture podataka ne mogu biti smeštene unutar jednog registra, već zahtevaju postojanje odgovarajućih memorijskih jedinica. Kako se tipično ovakve složene strukture podataka obrađuju element po element, ova činjenica otvara mogućnost izbora načina na koji će se pristupati individualnim elementima, kao i načina na koji su oni organizovani unutar memorije. Od ovih izbora u mnogome će zavisiti i performanse sistema koji se projektuje.




*Slika 4.18. Veza između memorijskog sistema i sistema za implementaciju „Naive“ algoritma za množenje dve matrice*

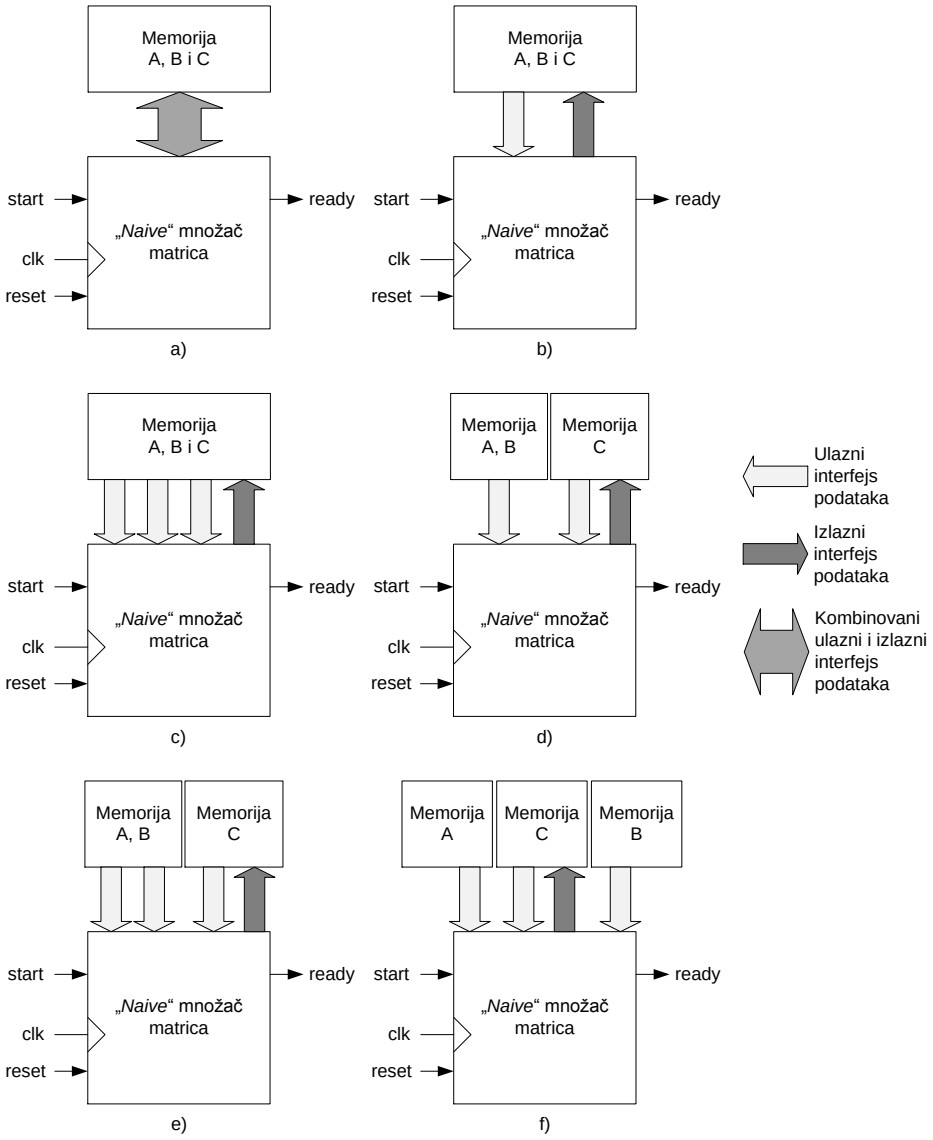
Primetimo da su u ovom slučaju ulazni i izlazni interfejsi podataka daleko složeniji. Ovaj put oni se ne mogu sastojati samo od jednog višebitnog ulaznog, odnosno izlaznog

porta, već moraju biti organizovani kao višeportni interfejsi koji treba da obezbede komunikaciju sa memorijom.

U našem primeru analizom možemo utvrditi da u slučaju „*Naive*“ algoritma množenja matrica imamo potrebu za pristup elementima tri različite složene strukture podataka (matrice u ovom slučaju). Ovaj pristup može se organizovani na različite načine, neki od kojih su prikazani na slici 4.19. Obratite pažnju da je prilikom analize performansi različitih arhitektura sa slike 4.19 pretpostavljeno da se memorijski podsistem uvek sastoji iz memorija sa sinhronim upisom i čitanjem.

Na slici 4.19a prikazana je varijanta ostvarivanja veze između memorijskog podsistema i „*Naive*“ množača matrica koja se bazira na korišćenju jedne jednopristupne memorije. Ova organizacija odgovara standarnoj organizaciji veze između mikroprocesora i memorijskog podsistema. U okviru iste memorije smeštene su sve tri matrice, *A*, *B* i *C*. U svakom taktu, memoriji se može pristupiti ili radi čitanja jednog elementa ili radi upisa jednog elementa određene matrice. Ova organizacija je najjednostavnija za realizaciju, ali postavlja velika ograničenja u pogledu efikasnosti izvršavanja „*Naive*“ algoritma množenja. Implementacija „*Naive*“ algoritma u ovom slučaju je ograničena brzinom pristupa elementima matrica *A*, *B* i *C*. Potrebno je tri takta da preuzmemo vrednosti elemenata $a_{i,k}$, $b_{k,j}$ i $c_{i,j}$ matrica *A*, *B* i *C*, jedan takt da ih pomnožimo i dodamo na tekuću vrednost elementa $c_{i,j}$ i još jedan takt da novu vrednost elementa $c_{i,j}$ upišemo nazad u memoriju, što čini ukupno pet taktova za izvršavanje tela unutrašnje petlje algoritma sa slike 4.17. Iako nema algoritamskih ograničenja koja bi nas sprečila da u istom taktu preuzmemo vrednosti elemenata $a_{i,k}$, $b_{k,j}$, i $c_{i,j}$, kao i da upišemo novu vrednost elementa $c_{i,j}$, postoje arhitekturna ograničenja (jednopristupna memorija) koja nas u tome sprečavaju.

Na slici 4.19b prikazana je varijanta sa dvopristupnom memorijom, koja ima jedan pristup za čitanje i jedan pristup za upis podataka. I dalje je potrebno tri takta da preuzmemo vrednosti elemenata $a_{i,k}$, $b_{k,j}$ i $c_{i,j}$ matrica *A*, *B* i *C*, jedan takt da ih pomnožimo i dodamo na tekuću vrednost elementa $c_{i,j}$, ali više nije potreban dodatni takt za upis nove vrednosti elementa $c_{i,j}$ u memoriju. Pošto memorija poseduje dodatni pristup za upis podataka, upis nove vrednosti elementa $c_{i,j}$ u memoriju može se izvršiti u istom taktu u kojem se vrši množenje i sabiranje. Korišćenjem ove arhitekture izvršavanje tela unutrašnje petlje algoritma sa slike 4.17 sada iznosi četiri takta. U poređenju sa pet taktova u slučaju korišćenja arhitekture sa slike 4.19a, arhitektura sa slike 4.19b donosi ubrzanje od 20% u izvršavanju algoritma množenja. Cena koja je pritom plaćena jeste da nam je sada neophodna dvopristupna memorija.




*Slika 4.19. Neki od mogućih načina realizacije veze između memorijskog podsistema i sistema za implementaciju „Naive“ algoritma za množenje dve matrice*

Razrađujući ideju arhitekture sa slike 4.19b do krajnjih granica, dolazimo do arhitekture prikazane na slici 4.19c. Ova arhitektura se bazira na korišćenju četvoropristupne memorije, sa tri pristupa za čitanje i jednim pristupom za upis podataka. Sada se sve operacije preuzimanja vrednosti elemenata $a_{i,k}$, $b_{k,j}$ i $c_{i,j}$ matrica $A$, $B$ i $C$ mogu izvesti u jednom taktu, dok je još jedan takt neophodan za ažuriranje vrednosti elementa $c_{i,j}$.

Korišćenjem ove arhitekture trajanje izvršavanja tela unutrašnje petlje svedeno je na samo dva takta, što predstavlja ubrzanje od 60% u odnosu na arhitekturu sa slike 4.19a. Međutim, ova arhitektura zahteva korišćenje četvoropristupne memorije, koja je izuzetno skupa po pitanju potrebnih hardverskih resursa za njenu realizaciju.

Kako je u „*Naive*“ algoritmu množenja matrica obrazac pristupa podacima smeštenim unutar memorijskog podsistema vrlo regularan i jasno particionisan, nema potrebe da se sve strukture smeštaju unutar iste, višepristupne memorije. Umesto korišćenja jedne memorije, matrice *A*, *B* i *C* mogu se smestiti u odvojene memorijske module. Na ovaj način svaki od korišćenih memorijskih modula može biti jednopristupna ili maksimalno dvopristupna memorija, koje predstavljaju standardne komponente. Obratite pažnju da se ova vrsta optimizacije broja memorijskih pristupa ne može sprovesti u opštem slučaju, i da zavisi od oblika obrazaca pristupa memoriji koji postoje u algoritmu koji se implementira.

Na slici 4.19d prikazana je arhitektura koja koristi dve memorije. U jednopristupnoj memoriji nalaze se smešteni elementi matrica *A* i *B*, dok se u dvopristupnoj memoriji nalaze smešteni elementi matrice *C*. Vreme potrebno da se izvrši telo unutrašnje petlje „*Naive*“ algoritma množenja matrica sa slike 4.17 sada iznosi tri takta, što predstavlja ubrzanje od 40% u odnosu na arhitekturu sa slike 4.19a.

Na slici 4.19e prikazana je arhitektura koja je takođe bazirana na korišćenju dve memorije ali su ovog puta obe memorije dvopristupne. U prvoj memoriji nalaze se smešteni elementi matrica *A* i *B*, dok se u drugoj memoriji nalaze smešteni elementi matrice *C*. Vreme potrebno da se izvrši telo unutrašnje petlje „*Naive*“ algoritma množenja matrica sa slike 4.17 sada iznosi dva takta, što predstavlja ubrzanje od 60% u odnosu na arhitekturu sa slike 4.19a. Obratite pažnju da je arhitektura sa slike 4.19e ekvivalentna arhitekturi sa slike 4.19c u pogledu performansi, ali je daleko jednostavnija za hardversku implementaciju jer ne zahteva korišćenje četvoropristupne memorije.

Konačno, na slici 4.19f prikazana je još jedna moguća arhitektura, koja se sastoji iz tri memorije. Sada je svaka matrica smeštena u posebni memorijki modul. Ova arhitektura zapravo predstavlja varijaciju arhitekture sa slike 4.19e kod koje je jedna dvopristupna memorija zamenjena sa dve jednopristupne memorije.

**NAPOMENA**: Prethodna analiza imala je za cilj da pokaže raznolikost mogućih rešenja prilikom hardverske implementacije algoritma. Za razliku od softverske implementacije, koja je ograničena na korišćenje fiksne hardverske platforme, sa svim njenim dobrim i lošim stranama, hardverska implementacija uvek ima veliki stepen slobode u izboru optimalne arhitekture. Različite arhitekture koje stoje na raspolaganju prilikom hardverske implementacije algoritma čine takozvani **prostor dizajna** (*Design Space*) za posmatrani algoritam. Svaka od raspoloživih arhitektura predstavlja jednu tačku unutar prostora dizajna, sa svojim karakteristikama u pogledu brzine rada, potrebnog broja harverskih resursa, potrošnje, pouzdanosti, cene, itd. Posao digitalnog dizajnera je da izvrši pretragu ovog prostora stanja (*Design Space Exploration*) i odabere arhitekturu koja predstavlja najbolju opciju u pogledu zahtevanih performansi sistema koji se projektuje.

**NAPOMENA**: Takođe obratite pažnju da će odabrani način formiranja ulaznog i izlaznog interfejsa podataka imate veliki uticaj na izgled ASMD dijagrama koji opisuje rad digitalnog sistema koji se projektuje. Različiti načini prenosa podataka od i ka memorijskom podsistemu rezultovaće u različitim upravljačkim akcijama, njihovom redosledu i trajanju.

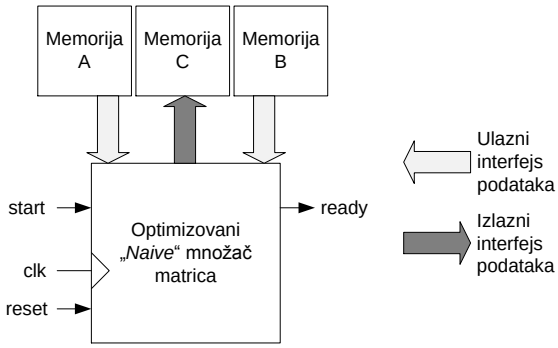
Na osnovu prethodne analize mogućih načina sprezanja digitalnog sistema koji implementira „*Naive*“ algoritam množenja matrica i memorijskog podsistema, može se primetiti da se malom promenom samog algoritma može ostvariti značajna ušteda u broju potrebnih pristupa memorijskom podsistemu. Naime, umesto da se $c_{i,j}$ elementu matrice *C* pristupa u svakom prolasku kroz unutrašnju petlju, možemo uvesti pomoćnu promenljivu *temp* čiju vrednost ćemo ažurirati tokom prolaska kroz unutrašnju petlju. Nakon što smo izračunali konačnu vrednost elementa $c_{i,j}$, nju ćemo upisati nazad u memoriju. Sve međuvrednosti ćemo čuvati lokalno, unutar promenljive *temp*. Kako će prilikom projektovanja digitalnog sistema primenom RT metodologije ova unutrašnja promenljiva biti mapirana u odgovarajući registar, njegovo ažuriranje nas „ne košta“ ništa u terminima dodatnih ciklusa ili dodatnih memorijskih interfejsa. Na ovaj način smo broj pristupa memoriji radi ažuriranja vrednosti jednog elementa matrice *C* sveli sa *m* čitanja i upisa na samo jedan upis. Modifikovani „*Naive*“ algoritam za množenje matrica prikazan je na slici 4.20.

```
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 4.20. Optimizovani „Naive“ algoritam za množenje dve matrice, sa redukovanim brojem pristupa matrici C*

**NAPOMENA**: Unutrašnju promenljivu *temp* možemo smatrati za neku vrstu minimalne *cache* memorije (veličine jednog registra). Umesto da sistem prilikom rada neprekidno komunicira sa memorijskim podsistemom, što je skupo u terminima brzine i hardverskih resursa, on većinu vremena komunicira sa lokalnom, brzom *cache* memorijom.

Nakon ove optimizacije „*Naive*“ algoritma za množenje matrica može se osmisliti još jedna arhitektura čitavog sistema, prikazana na slici 4.21.




*Slika 4.21. Arhitektura sistema za implementaciju optimizovanog „Naive“ algoritma za množenje dve matrice*

Iako su performanse arhitekture sa slike 4.21 jednake performansama arhitektura sa slika 4.19c, 4.19e i 4.19f, njena velika prednost je da ona zahteva korišćenje samo jednopristupnih memorija. Upravo iz tog razloga ona će biti korišćena u nastavku ovog primera.

### 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:       temp = 0;
          k = 0;
l3:         temp = temp + 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;
          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 4.22. 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 4.22., 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 maksimalni broj elemenata u bilo kojoj dimenziji matrica *A* i *B*.

Elementima *a_in* i *b_in* matrica *A* i *B* pristupaćemo preko dva odvojena memorijska interfejsa za čitanje podataka, jer će matrice *A* i *B* biti smeštene u odvojenim jednopristupnim memorijama, kao što je prikazano na slici 4.21.

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

- *a_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE * SIZE) \rceil$ bita
- *a_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
- *a_wr_o* – kontrolna magistrala, 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 $\lceil ld(SIZE * SIZE) \rceil$ 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 trećoj jednopristupnoj memoriji, koristićemo još jedan memorijski interfejs, ovoga puta interfejs za upis podataka koji se sastoji iz sledećih portova:

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

**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 koji implementira željeni algoritam u hardveru. Kako je „*Naive*“ algoritam za množenje matrica i dalje 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*:

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

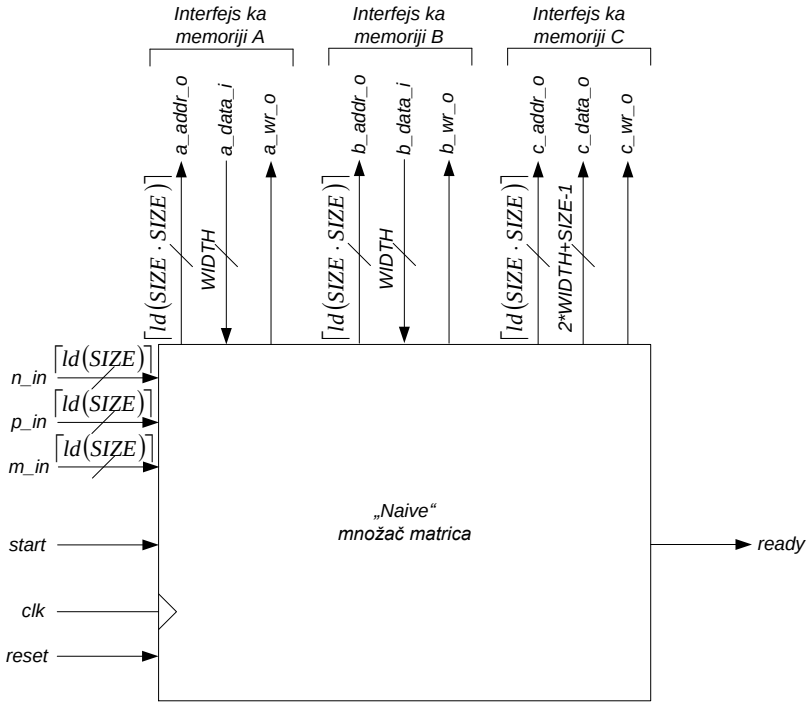
  - *b_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE*SIZE) \rceil$ bita
  - *b_data_i* – ulazna magistrala podataka, *std_logic_vector* tipa, širine *WIDTH* bita
  - *b_wr_o* – kontrolna magistrala, tipa *std_logic*

- Izlazni interfejs podataka – sastoji se iz portova koji čine interfejs ka memoriji *C*:
  - *c_addr_o* – adresna magistrala, tipa *std_logic_vector*, širine $\lceil ld(SIZE*SIZE) \rceil$ 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*, 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 4.23. prikazan je kompletan interfejs digitalnog sistema koji implementira „*Naive*“ algoritam množenja dve matrice.



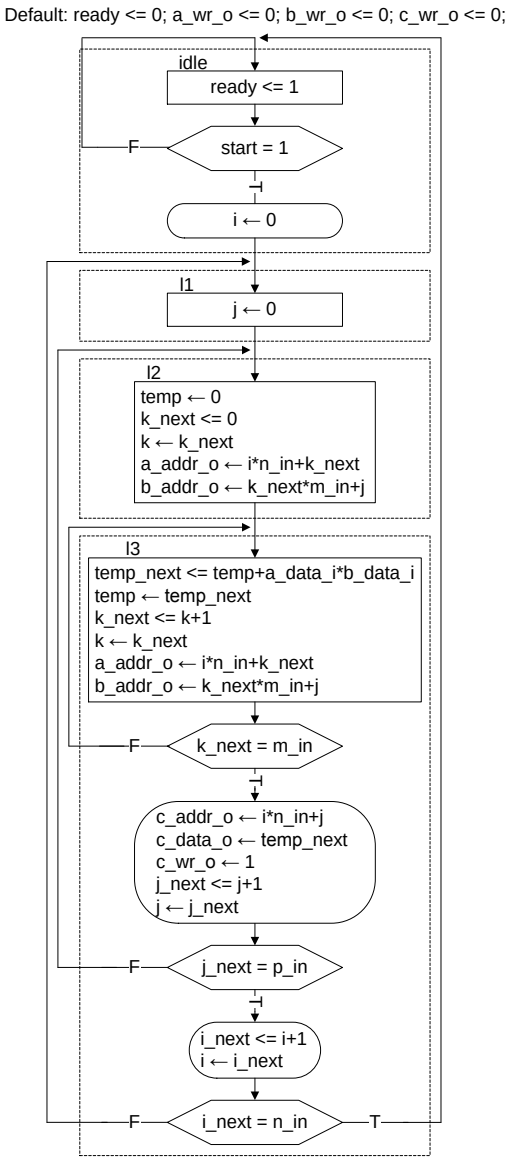

*Slika 4.23. Interfejs digitalnog sistema koji implementira „Naive“ algoritam množenja dve matrice*

**Korak 3: Projektovanje *controlpath* modula**

Uzimajući u obzir i način na koji je organizovan memorijski podsistem (vidi sliku 4.21) algoritam sa slike 4.22 lako se može prevesti u odgovarajući ASMD dijagram, koji je prikazan na slici 4.24.

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), inicijalizuje registar *temp* i izračunavaju se početne adrese elemenata matrica *A* i *B*, a nakon toga se prelazi u stanje *l3*. U stanju *l3* se na tekuću vrednost registra *temp* dodaje vrednost izračunatog proizvoda tekućih elemenata 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 vrši se upis izračunate vrednosti tekućeg elementa matrice *C* na odgovarajuće mesto u memoriji, 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 4.24. ASMD dijagram „Naive“ množača matrica*

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 zamnoženje matrica sa slike 4.22 postoje četiri unutrašnje promenljive, *i, j, k* i *temp*. 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, dok je registar *temp* 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*.

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 4.24 dolazimo do sledeće liste RT operacija:

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

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*:

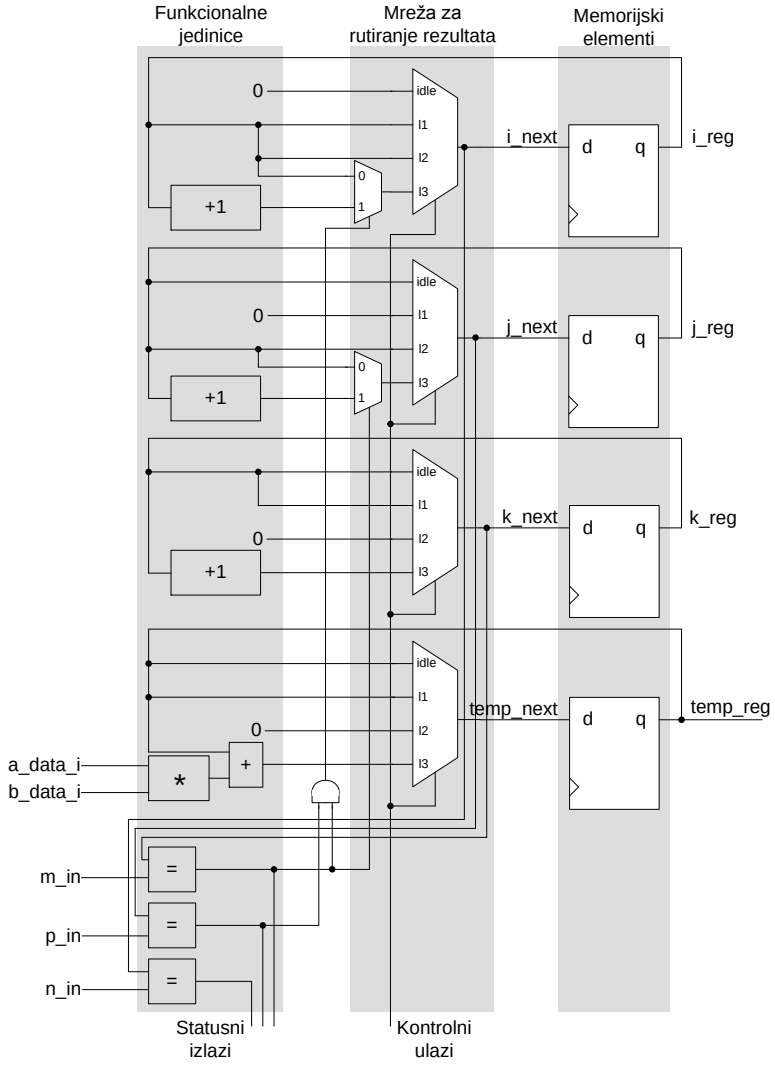
1. u *idle* stanju: $k \leftarrow k$
2. u *l1* stanju: $k \leftarrow k$

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

RT operacije kojima je ciljni registar *temp*:

1. u *idle* stanju: *temp* ← *temp*
2. u *l1* stanju: *temp* ← *temp*
3. u *l2* stanju: *temp* ← 0
4. u *l3* stanju: *temp* ← *temp*+*a_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 4.25.




*Slika 4.25. Datapath modul „Naive“ množača matrica*

**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 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 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 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 temp_reg, temp_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');
      temp_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;
  end if;
end process;

-- Combinatorial circuits
process (state_reg, start, a_data_i, b_data_i, i_reg, j_reg, k_reg, temp_reg, i_next, j_next, k_next,
          temp_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;

    when l3 =>
      temp_next <= temp_reg + 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
        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;
    else
      state_next <= l3;
    end if;
  end case;
 end process;
end two_seg_arch;
```

Obratite pažnju da prethodni model koristi *utils_pkg* VHDL paket, u kome se nalazi definicija funkcije *log2c*, koja se koristi za računanje minimalnog broj potrebnog bitova za reprezentaciju željenog opsega vrednosti pozitivnih brojeva. Sadržaj *utils_pkg* VHDL paketa prikazan je u nastavku.

```
-- package declaration
library ieee;
use ieee.std_logic_1164.all;

package utils_pkg is
   function log2c (n: integer) return integer;
end utils_pkg;

--package body
package body utils_pkg is
 function log2c (n: integer) return integer is
   variable m, p: integer;
 begin
   m := 0;
   p := 1;
   while p < n loop
     m := m + 1;
     p := p * 2;
   end loop;
   return m;
 end log2c;
end utils_pkg;
```

Na kraju, procenimo brzinu rada „*Naive*“ množača matrica. Analizom „*Naive*“ algoritma množenja matrica može se zaključiti da je potreban broj iteracija za kompletiranje operacije množenja dve matrice jednak proizvodu dimenzija matrica *A* i *B*, *n_in*·*m_in*·*p_in*. Ovo je dobro poznat rezultat iz teorije algoritama, gde je vremenska složenost „*Naive*“ algoritma množenja dve matrice procenjena na $O(n^3)$, u slučaju množenja dve kvadratne matrice, dimenzija *n*x*n*. Analizom ASMD dijagrama sa slike 4.24, može se zaključiti da računanje vrednosti svakog elementa matrice *C* zahteva *m_in* taktova (jer u svakom raktu računamo jedan proizvod odgovarajućih elementata matrica *A* i *B* i dodajemo ga na tekuću vrednost akumulirane sume, promenljivu *temp*). Nakon

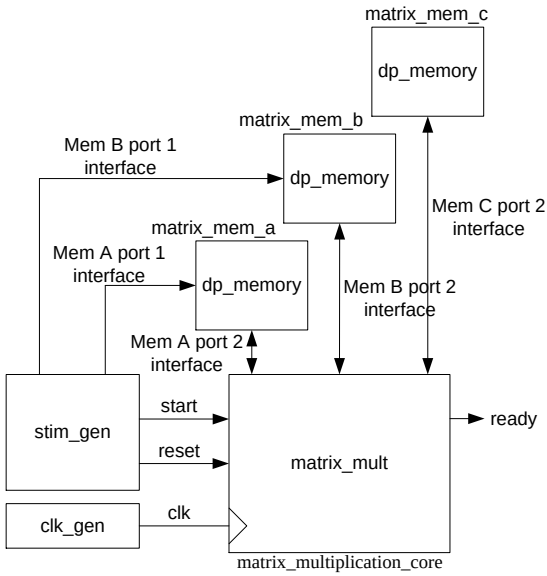
završetka računanja vrednosti tekućeg elementa matrice *C*, pre započinjanja računanja vrednosti narednog elementa troši se još jedan takt (ponovo se vraćamo u *l2* stanje kako bismo inicijalizovali brojač *k*). Dodatni takt troši se svaki put kada započinjenom računanje vrednosti elemenata matrice *C* iz sledeće vrste. Imajući sve ovo u vidu, potreban broj taktova da se pomoću projektovanog „*Naive*“ množača matrica pomnože dve matrice dimenzija *n*x*m* i *m*x*p* iznosi

$$T_{naive} = n\_in \cdot (p\_in \cdot (m\_in + 1) + 1) = n\_in \cdot p\_in \cdot m\_in + n\_ip \cdot p\_in + n\_in$$

taktova. Iz prethodnog izraza možemo primetiti da u implementiranoj arhitekturi postoji dosta dodatnih operacija (koje zahtevaju *n_in*· *p_in*+ *n_in* taktova), koje se uglavnom odnose na ažuriranje tri indeksna brojača *i, j* i *k*. Iako teorija algoritama, prilikom procene vremenske složenosti algoritma, zanemaruje ove dodatne opearacije (što je jedna od karakteristika korišćene veliko-O notacije), u praktičnim aplikacijama ove dodatne operacije mogu imati vrlo značajnu ulogu i njihova minimizacija je svakako od interesa. Na primer, pretpostavimo da množimo dva kvadratne matrice, dimenzija 3x3. Korišćenjem prikazanog „*Naive*“ množača za izvođenje ove operacije biće potrebno ukupno 39 taktova. Ako bismo mogli projektovati digitalni sistema kod koga bi bile eliminisane sve dodatne operacije, potreban broj taktova pao bi na 27, što predstavlja ubrzanje rada od 30%! Zainteresovani studenti ohrabruju se da pokušaju da modifikuju ASMD dijagram sa slike 4.24 kako bi smanjili ovaj broj dodatnih taktova.

**Korak 6: Verifikacija razvijenog HDL modela**

U prethodnim primerima nismo prikazivali verifikaciono okruženje koje bi moglo da se iskoristi za verifikaciju razvijenih digitalnih sistema. Kako je „*Naive*“ množač matrica značajno složeniji sistem od prethodno prikazanih, u ovom slučaju prikazaćemo i izgled jednog mogućeg verifikacionog okruženja za njega. Struktura korišćenog verifikacionog okruženja može se videti na slici 4.26., a nakon nje prikazan je i VHDL model u okviru kojega je implementirano verifikaciono okruženje „*Naive*“ množača dve matrice.




*Slika 4.26. Struktura verifikacionog okruženja "Naive" množača matrica*

```
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_tb is
end entity;

architecture beh of matrix_mult_tb is
  constant DATA_WIDTH_c: integer := 8;
  constant SIZE_c: integer := 3;
  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));
  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));

  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 a_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal a_data_in_s: std_logic_vector(DATA_WIDTH_c-1 downto 0);
  signal a_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 c_addr_s: std_logic_vector(log2c(SIZE_c*SIZE_c)-1 downto 0);
  signal c_data_out_s: std_logic_vector(2*DATA_WIDTH_c+SIZE_c-1 downto 0);
  signal c_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 operation has been complted
  wait until ready_s = '1';

  -- End stimulus generation
  wait;
end process;

-- Matrix A memory
matrix_a_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_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       => a_addr_s,
    p2_data_i       => (others => '0'),
    p2_data_o       => a_data_in_s,
    p2_wr_i         => a_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         => (others => '0'),
      p1_data_i         => (others => '0'),
      p1_data_o         => open,
      p1_wr_i           => '0',
      p2_addr_i         => c_addr_s,
      p2_data_i         => c_data_out_s,
      p2_data_o         => open,
      p2_wr_i           => c_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(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
    a_addr_o            => a_addr_s,
    a_data_i            => a_data_in_s,
    a_wr_o              => a_wr_s,
    -- Matrix B memory interface
    b_addr_o            => b_addr_s,
    b_data_i            => b_data_in_s,
    b_wr_o              => b_wr_s,
    -- 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
    c_addr_o            => c_addr_s,
    c_data_o            => c_data_out_s,
    c_wr_o              => c_wr_s,
    ---------- Command interface ----------
    start               => start_s,
    ---------- Status interface ----------
    ready               => ready_s);
end architecture beh;
```

Prikazano verifikaciono okruženje koristi i tri instance dvopristupne memorije, za smeštanje vrednosti elemenata ulaznih matrica *A* i *B*, kao i vrednosti matrice *C* koja predstavlja rezultat množenja matrica *A* i *B*. Radi kompletnosti, u nastavku je prikazan i VHDL model ove dvopristupne memorije.

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

use work.utils_pkg.all;

entity dp_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: in 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);
end entity;

architecture beh of dp_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;
        p1_data_o <= mem_s(conv_integer(p1_addr_i));
        p2_data_o <= mem_s(conv_integer(p2_addr_i));
      end if;
    end if;
  end process;
end architecture beh;
```

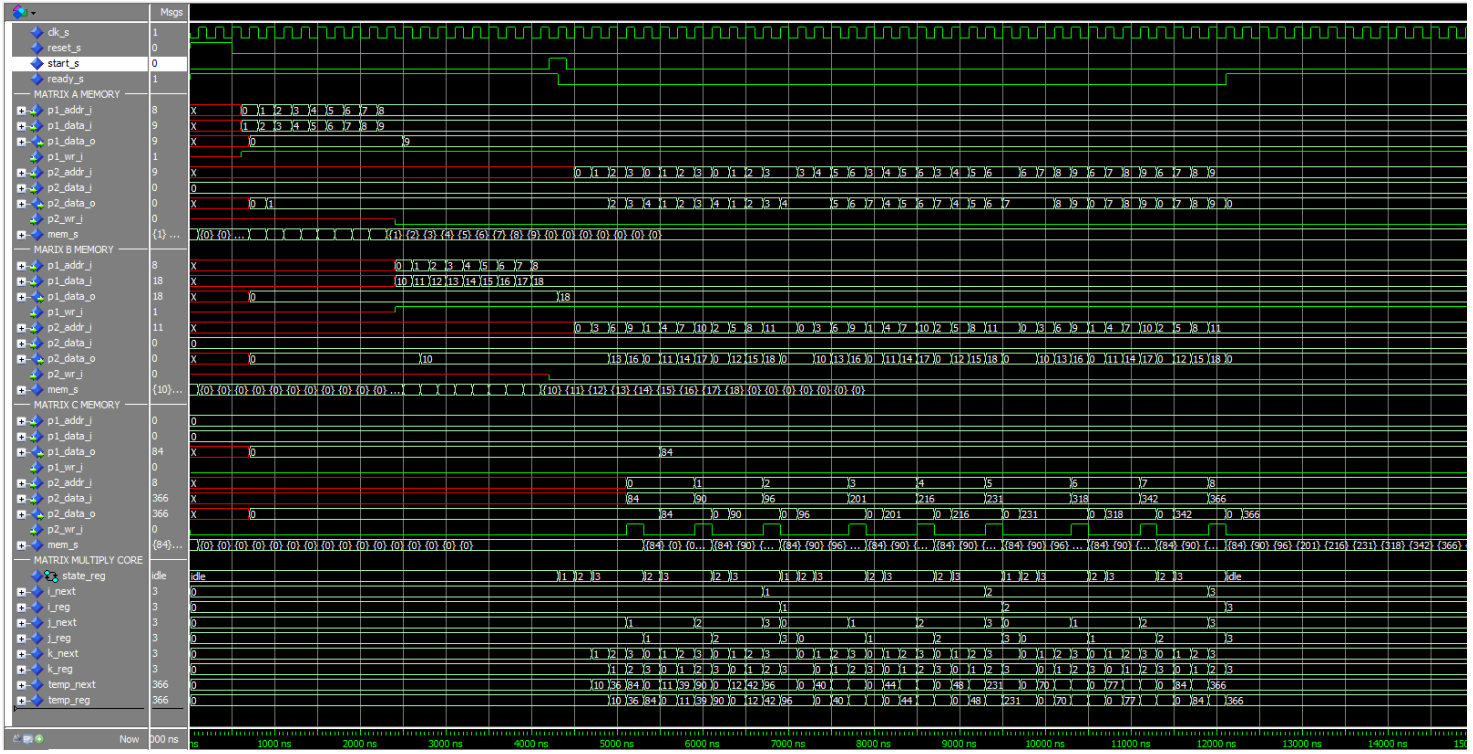
Na slici 4.27 prikazani su talasni oblici karakterističnih signala, koji predstavljaju rezultat simulacije verifikacionog okruženja i modela „*Naive*“ množača matrica, u slučaju množenja kvadratnih matrica *A* i *B*, dimenzija 3x3. Matrice *A* i *B* definisane su na sledeći način

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

Rezultat množenja ove dve matrice trebalo bi da bude sledeća matrica *C*

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

Inspekcijom konačnih vrednosti unutrašnjih signala *mem_s* sva tri memorijska modula („*Matrix A Memory*“, „*Matrix B Memory*“ i „*Matrix C Memory*“) sa slike 4.27., možemo se uveriti da projektovani digitalni sistem korektno implementira „*Naive*“ algoritam množenja dve matrice.




*Slika 4.27. Rezultat simulacije rada „Naive“ množača matrica, na primeru množenja dve matrice dimenzija 3x3*

## Zadaci za vežbu

Zadatak 4.1.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za izračunavanje kvadratnog korena zadatog broja prikazanog na slici 4.28.

```
sqrt(x)
{
   op = x;
   res = 0;
   one = 1 << 30;
   while (one > op)
      one >>= 2;
   while (one != 0) {
      if (op >= res + one) {
         op = op - (res + one);
         res = res +  2 * one;
      }
      res >>= 1;
      one >>= 2;
   }
   return res;
}
```

*Slika 4.28. Algoritam za izračunavanje kvadratnog korena zadatog broja*

Prilikom hardverske implementacije pretpostaviti da je ulazni broj *x* predstavljen sa 32 bita i da su pomoćne promenljive *op*, *res* i *one* takođe predstavljene sa 32 bita. Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije korenovanja i izlazni *ready* signal, koji označava završetak tekuće operacije korenovanja. Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema za izračunavanje kvadratnog korena zadatog broja.

Zadatak 4.2.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za deljenje dva binarna broja prikazanog na slici 4.29.

```
if (a_in > b_in) then
{
   op1 = a_in;
   op2 = b_in;
}
else
{
   op1 = b_in;
   op2 = a_in;
}
```

```
q = 0;
r = 0;
while (op1 >= op2)
{
   op1 = op1 – op2;
   q = q + 1;
}
r = op1;
return r;
```

*Slika 4.29. Algoritam za deljenje dva binarna broja*

Prilikom hardverske implementacije pretpostaviti da su dva binarna broja zadata kao 8-bitni neoznačeni binarni brojevi, *a_in* i *b_in*. Kao rezultat izvršavanja algoritma sa slike 4.29, na izlazu sistema treba da se pojave količnik, izlaz *q*, i ostatak, izlaz *r*. Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije deljenja i izlazni *ready* signal, koji označava završetak tekuće operacije deljenja. Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema za delenje dva binarna broja.

Zadatak 4.3.

Primenom RT metodologije izvršiti hardversku implementaciju Butovog algoritma (*Booth's algorithm*) za množenje dva 8-bitna binarna označena broja, *a_in* i *b_in* prikazanog na slici 4.30.

```
A = a_in&"000000000";
S = (-a_in)&"000000000";
P = "00000000"&b_in&'0';
for (i=0; i<8; i++)
{
   if (P(1:0) = "01")
      temp = P+A;
   else if (P(1:0) = "10")
      temp = P+S;
   else if (P(1:0) = "00")
      temp = P;
   else
      temp = P;
   P = temp >> 1; // Koristiti aritmetičko pomeranje u desno!!!
   res = P(16:1);
}
return res;
```

*Slika 4.30. Butov algoritam za množenje dva binarna označena broja*

Kao rezultat izvršavanja algoritma sa slike 4.30, na izlazu sistema treba da se pojavi proizvod, izlaz *res*. Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije množenja i izlazni *ready* signal, koji označava

završetak tekuće operacije množenja. Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema za množenje dva binarna broja.

Zadatak 4.4.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za nalaženje pozicije zadatog elementa *key*, unutar prethodno sortiranog niza brojeva *x*, korišćenjem „*Binary Search*“ algoritma, prikazanog na slici 4.31.

```
while (left ≤ right)
{
   middle = (left+right)/2;
   if x[middle] = key then
      return key;
   if x[middle] > key then
      right = middle – 1;
   if x[middle] < key then
      left = middle + 1;
}
return not_found;
```

*Slika 4.31. „Binary Search“ algoritam za nalaženje pozicije elementa key u nizu x*

Prilikom hardverske implementacije pretpostaviti da je element *key*, čiju je poziciju potrebno pronaći, dovodi preko 8-bitnog neoznačenog porta, *key_in*. Leva i desna granica niza, *left*, *right*, takođe se dovode korišćenjem dva 8-bitna neoznačena porta, *left_in*, *right_in*. Kao rezultat izvršavanja algoritma sa slike 4.31, na izlazu sistema treba da se pojavi pozicija elementa *key_in*, na izlaznom portu *pos_out*. Pored porta *pos_out* potrebno je uvesti i dodatni 1-bitni izlazni port *el_found_out*, koji će signalizirati da li je element pronađen u nizu ili nije. Prilikom projektovanja takođe uvesti ulazni *start* signal koji označava početak izvođenja operacije pretrage i izlazni *ready* signal, koji označava završetak tekuće operacije pretrage.

Prilikom hardverske implementacije pretpostaviti da je sortirani niz 8-bitnih neoznačenih brojeva *x* koji je potrebno pretražiti smešten u jednopristupnoj memoriji. Memorija ima jedan pristup, *mem_data*, koji se može koristiti i za čitanje i za upis. Adresa lokacije kojoj se želi pristupiti prosleđuje se preko 8-bitnog ulaza za adresiranje, *adr_out*, a ulaz za kontrolu upisa, odnosno čitanja, *rw_out*, određuje koju operaciju želimo da izvršimo. Pretpostaviti da je upis u memoriju sinhroni i da se odvija na rastuću ivicu globalnog sinhronizacionog signala, dok je čitanje asinhrono, pri čemu se na *mem_data* izlazu pojavljuje trenutni sadržaj memorijske lokacije koja je adresirana. Napisati odgovarajući testbenč pomoću kojega je moguće verifikovati korektan rad projektovanog sistema.

Zadatak 4.5.

Primenom RT metodologije izvršiti hardversku implementaciju algoritma za određivanje indeksa elementa niza brojeva *x* koji je najbliži srednjoj vrednosti članova niza, prikazanog na slici 4.32.

```
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;
```

*Slika 4.32. Algoritam za određivanje indeksa elementa niza x koji je najbliži srednjoj vrednosti članova niza x*

Prilikom hardverske implementacije pretpostaviti da je niz 8-bitnih označenih brojeva *x* koji je potrebno pretražiti smešten u jednopristupnoj memoriji. Memorija ima jedan pristup koji se može koristiti i za čitanje i za upis. Adresa lokacije kojoj se želi pristupiti prosleđuje se preko ulaza za adresiranje, a ulaz za kontrolu upisa, odnosno čitanja određuje koju operaciju želimo da izvršimo. Pretpostaviti da je upis u memoriju 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:

- ulazni port, *x_in* preko koga se prosleđuju podaci koji su pročitani iz memorije,
- adresni port, *adr_out* preko koga se prosleđuju adrese lokacija kojima se želi pristupiti (pretpostaviti da su ovi portovi 8-bitni),
- signal dozvole upisa, odnosno čitanja iz memorije, *wr_out* (kada je signal na visokom logičkom nivou vrši se upis u memoriju),
- jedan ulazni port, *n_in*, preko kojega se zadaje veličina niza koji je potrebno obraditi (veličina ovog porta je takođe 8 bita),
- jedan izlazni port, *pos*, preko kojega se prosleđuje pozicija elementa u nizu čija je vrednost najbliža srednjoj vrednosti članova niza (ovo je takođe 8-bitni port).

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

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

Zadatak 4.6.

Primenom RT metodologije izvršiti hardversku implementaciju *„Selection Sort“* algoritma za sortiranje niza brojeva x prikazanog na slici 4.33.

```
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];
         }
```

*Slika 4.33. „Selection Sort“* a*lgoritam za sortiranje niza brojeva x*

Prilikom hardverske implementacije pretpostaviti da je niz 8-bitnih neoznačenih brojeva *x* koji je potrebno sortirati smešten u dvopristupnoj memoriji. Dvopristupna memorija ima dva pristupa pri čemu se svaki od njih može koristiti za čitanje i za upis. Svaki od pristupa ima svoj ulaz za adresiranje i ulaz za kontrolu upisa, odnosno čitanja. Pretpostaviti da je upis u memoriju sinhroni i da se odvija na rastuću ivicu globalnog sinhronizacionog signala, dok je čitanje asinhrono, 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:

- dva ulazna porta, *x_in1* i *x_in2* preko kojih se prosleđuju podaci koji su pročitani iz dvopristupne memorije,
- dva izlazna porta, *x_out1* i *x_out2* preko kojih se prosleđuju podaci koje je potrebno upisati u dvopristupnu memoriju,
- dva adresna porta, *adr1_out* i *adr2_out* preko kojih se prosleđuju adrese lokacija kojima se želi pristupiti (pretpostaviti da su ovi portovi 8-bitni),
- dva signala dozvole upisa, odnosno čitanja, *wr1_out* i *wr2_out*, za dva pristupa memorije (kada je signal na visokom logičkom nivou vrši se upis u memoriju),

- jedan ulazni port, *n_in*, preko kojega se zadaje veličina niza koji je potrebno sortirati (veličina ovog porta je takođe 8 bita),
- jedan ulazni port, *dir_in*, preko kojega se zadaje smer sortiranja (ovo je jednobitni port).

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

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

Zadatak 4.7.

Fibonačijeva fukcija definisana je sledećom rekurzivnom funkcijom:

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

Potrebno je implementirati ovu funkciju u hardveru. Pretpostavimo da je *n* 6-bitni ulazni signal. Obratiti pažnju da je vrednost funkcije *fib(63)* jednaka 6557470319842.

a) Nacrtati ASMD dijagram i *data path* sistema koji implementira Fibonačijevu funkciju.

b) Na osnovu ASMD dijagrama i *data path*-a napisati VHDL model sistema.

c) Napisati jednostavan testbenč pomoću kojega je moguće verifikovati ispravan rad projektovanog sistema.