UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET
KATEDRA ZA ELEKTRONIKU
SMER: ELEKTRONSKA KOLA I SISTEMI
PREDMET: DSP ALGORITMI I PROGRAMIRANJE
PROFESOR: **Prof. Dr. Mile Stojčev**

PROJEKAT

32-bitni Barrel Shifter

STUDENTI:

Radoslav Aleksić 11716,
Miloš Radovanović 11864

U Nišu, 4.7.2010. godine

Sadržaj:

1. Uvod (strana 3)

2. Operacija pomeranja i rotacije (strana 4)

3. Pomerači i rotatori (strana 6)

4. Barrel Shifter-i sa inverzijom podataka zasnovani na multiplekserima (strana 9)

5. Realizacija Barrel Shifter-a sa inverzijom podataka zasnovanog na multiplekserima (strana 12)

6. Rezultati simulacije (strana 23)

7. Kompletan VHDL kod (strana 28)

8. Reference i literatura (strana 36)

1. UVOD

Barrel Shifter je digitalno kolo koje može da pomera (shift) digitalnu reč (data word) za određeni broj bitova u jednom ciklusu takta.

Uobičajenu primenu nalazi u harderskoj implementaciji aritmetike u pokretnom zarezu. Kod operacija sabiranja i oduzimanja koje se obavljaju u pokretnom zarezu, brojevi nad kojima se vrši operacija moraju da budu poravnjani, što podrazumeva pomeranje (shift) manjeg broja u desno, uvećavajući tako njegov eksponent, dok se on ne izjednači sa eksponentom većeg broja. Za ovu potrebu, Barrel Shifter treba da u jednom ciklusu pomeri manji broj za bilo koji broj bitova. Ukoliko bi se koristio jednobitni pomerač, za pomeraj od n bitova bilo bi potrebno n ciklusa takta.

U ovom radu projektovan je i opisan 32-bitni Barrel Shifter koji u jednom ciklusu takta može da izvrši sledećže operacije (sa pomerajem od n bitova):

1. Desno logičko pomeranje,
2. Desno aritmetičko pomeranje,
3. Rotaciju u desno,
4. Levo logičko pomeranje,
5. Levo aritmetičko pomeranje,
6. Rotaciju u levo.

Dizajn je optimizovan da deli harver za različite operacije. Takođe, prikazana je tehnika i implementacija kola koje određuje prekoračenje i nulti rezultat paralelno sa izvršenjem operacija pomeranja.

2. Operacije pomeranja i rotacije

U ovom radu, A je definisano kao ulazni 32-bitni operand, B je veličina pomeraja/rotacije (5-bitni broj), a Y je pomeren/rotiran 32-bitni rezultat. Primer izvršenja operacija koje obavlja pomerač/rotator iz ovog rada ilustrovan je u tabeli 2.1. U ovoj tabeli bit vektor ulaznog operanda A je prikazan kao a31,a30,a29,a28...a3,a2,a1,a0, a količina pomeraja/rotacije B iznosi 5 bitova.

- Operacija B-bitnog logičkog pomeranja u desno obavlja pomeranje u desno za B bitova i postavlja prvih B bitova rezultata na '0';
- Operacija B-bitnog aritmetičkog pomeranja u desno obavlja pomeranje u desno za B bitova i postavlja prvih B bitova rezultata na vrednost MSB-a, koji predstavlja bit znaka broja A;
- Operacija B-bitne rotacije u desno obavlja pomeranje u desno za B bitova i postavlja prvih B bitova rezultata na na vrednost poslednjih B bitova podatka A;
- Operacija B-bitnog logičkog pomeranja u levo obavlja pomeranje u levo za B bitova i postavlja poslednjih B bitova rezultata na '0';
- Operacija B-bitnog aritmetičkog pomeranja u levo obavlja pomeranje u levo za B bitova i postavlja prvih B bitova rezultata na vrednost '0';
- Operacija B-bitne rotacije u levo obavlja pomeranje u levo za B bitova i postavlja poslednjih B bitova rezultata na na vrednost prvih B bitova podatka A;

Operacija	Y
5-bitno logičko pomeranje u desno	00000a31a30a29a28.....a8a7a6a5
5-bitno aritmetičko pomeranje u desno	a31a31a31a31a31a30a29a28...a8a7a6a5
5-bitan rotacija u desno	a4a3a2a1a0a31a30a29a28.....a8a7a6a5
5-bitno logičko pomeranje u levo	a26a25a24a23.....a2a1a0 0 0 0
5-bitno aritmetičko pomeranje u levo	a31a25a24a23.....a2a1a0 0 0 0
5-bitna rotacija u levo	a26a25a2.....a2a1a0a31a30aa29a28a27

Tabela 2.1. Primeri pomeranja za A = a31a30a29a28.....a4a3a2a1a0 i B = 5.

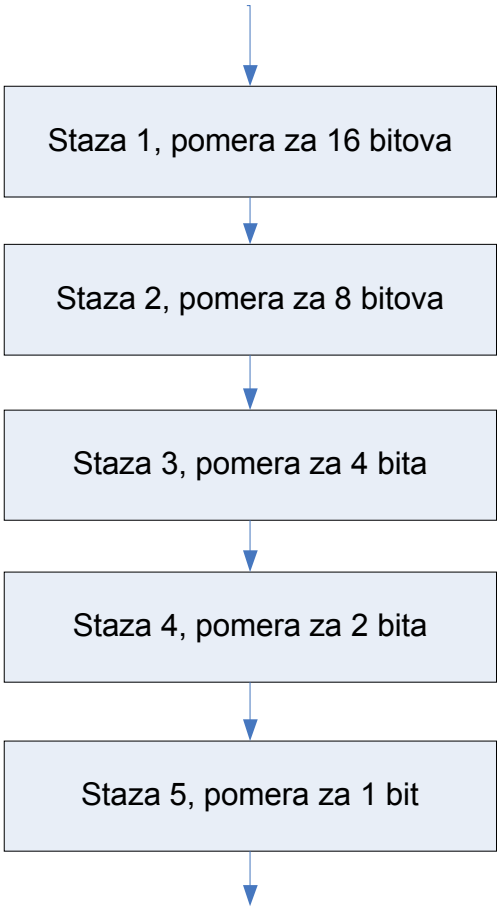
3-bitni opkod			Operacija
Left	Rotate	Arithmetic	
0	0	0	Desno logičko pomeranje
0	0	1	Desno aritmetičko pomeranje
0	1	X	Rotacija u levo

1	0	0	Levo logičko pomeranje
1	0	1	Levo aritmetičko pomeranje
1	1	X	Rotacija u desno

Tabela 2.2. Bitovi koji kontrolišu operaciju.

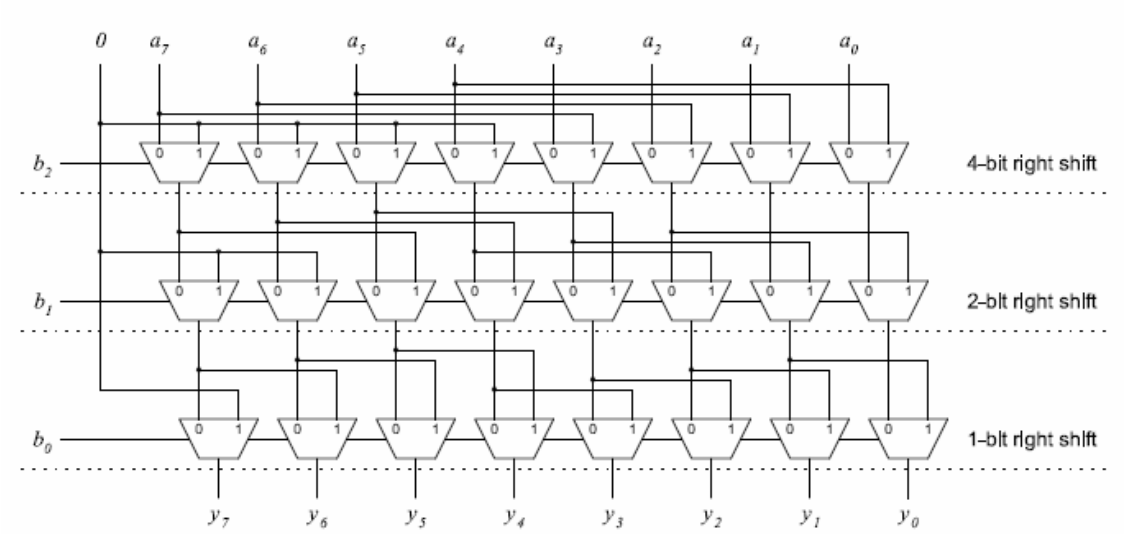
3. Pomerači i rotatori

32-bitni barrel pomerač upotrebljava $\log_2(32) = 5$ staza pomeranja. Svaki bit količine pomeraja B kontroliše drugu stazu pomerača. Podaci u određenoj stazi b_k su pomereni za 2^k bitova ukoliko je b_k = '1'; u suprotnom nema pomeraja. Slika 3.1 pokazuje staze pomeranja za 32-bitni barrel pomerač koji pomera podatke u desno.



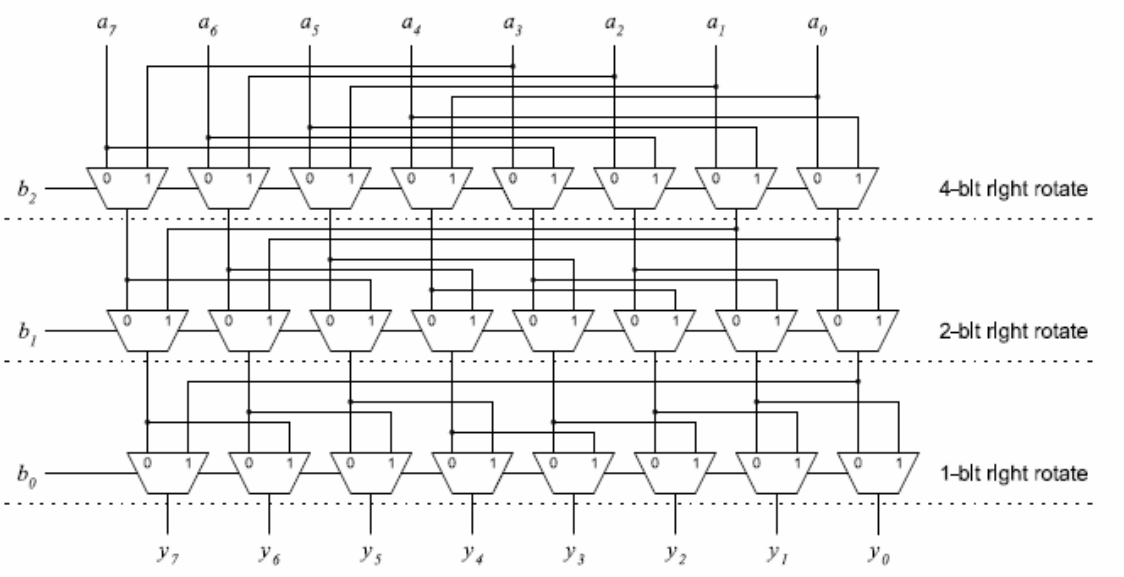
Slika 3.1. Staze kod 32-bitnog *Barrel* pomerača.

Na slici 3.2 prikazan je blok dijagram 8-bitnog logičkog pomerača u desno radi razumevanja. On koristi 3 faza sa pomeranjem od 4, 2 i 1 bitova.



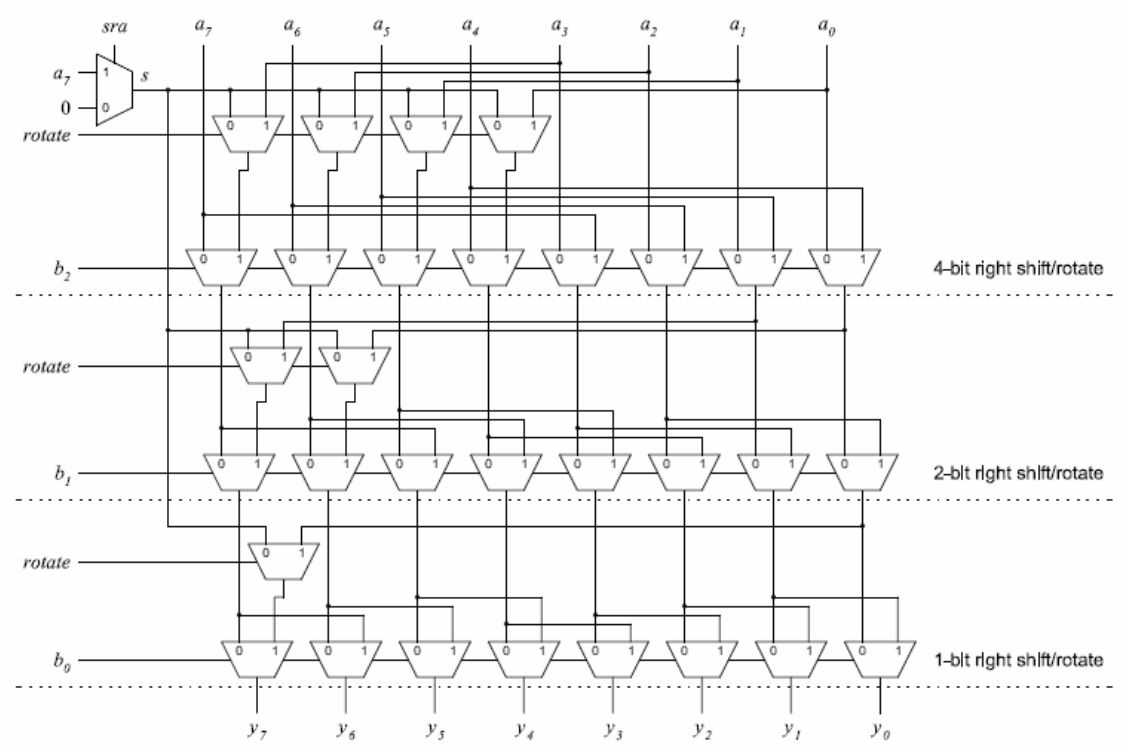
Slika 3.2. 8-bitni logčki pomerač u desno.

Slična jedinica koja obavlja rotacije, umesto pomeranja u desno, može se napraviti modifikacijom konekcija multiplekserima za bitove veće važnosti. Na slici 3.3 prikazan je blok dijagram 8-bitnog rotatora u desno. Desni rotator i logički desni pomerač obezbeđuju različite ulaze multiplekserima za bitove veće važnosti. Kod rotatora, pošto su svi bitovi sa ulaza sprovedeni na izlaz, ne postoji potreba za linijama koje nose '0'. Umesto tih linija ubačene su linije koje za omogućavanje rutiranja poslednjih 2^k bitova na prvih 2^k bitova u fazi koju kontroliše bit b_k.



Slika 3.3. 8-bitni rotator u desno.

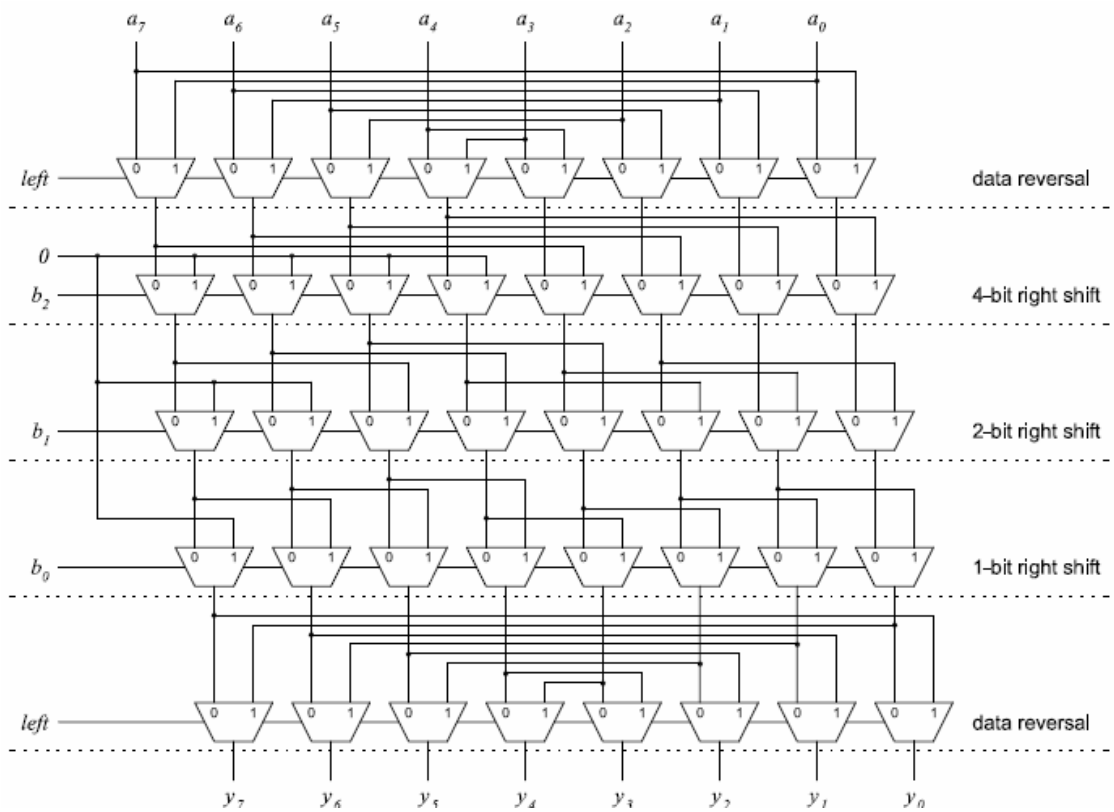
Desni logički pomerač se može proširiti da obavlja i operaciju aritmetičkog pomeranja u desno, kao i operaciju desne rotacije pomoću dodatnih multipleksera. Ovaj pristup je prikazan na slici 3.4 za 8-birni pomerač/rotator u desno sa 3 faze : 4-bitnom, 2-bitnom i 1-bitnom. Inicijalno, jednim multiplekserom se bira između '0' za logičko desno pomeranje i a_{n-1} za aritmetičko pomeranje u desno, čime se dobija signal s. U fazi kontrolisanoj bitom b_k, 2^k multipleksera biraju između signala s za pomeranjei 2^k nižih bitova podatka za rotaciju.



Slika 3.4. 8-bitni pomerač/rotator zasnovan na arhitekturi sa multiplekserima.

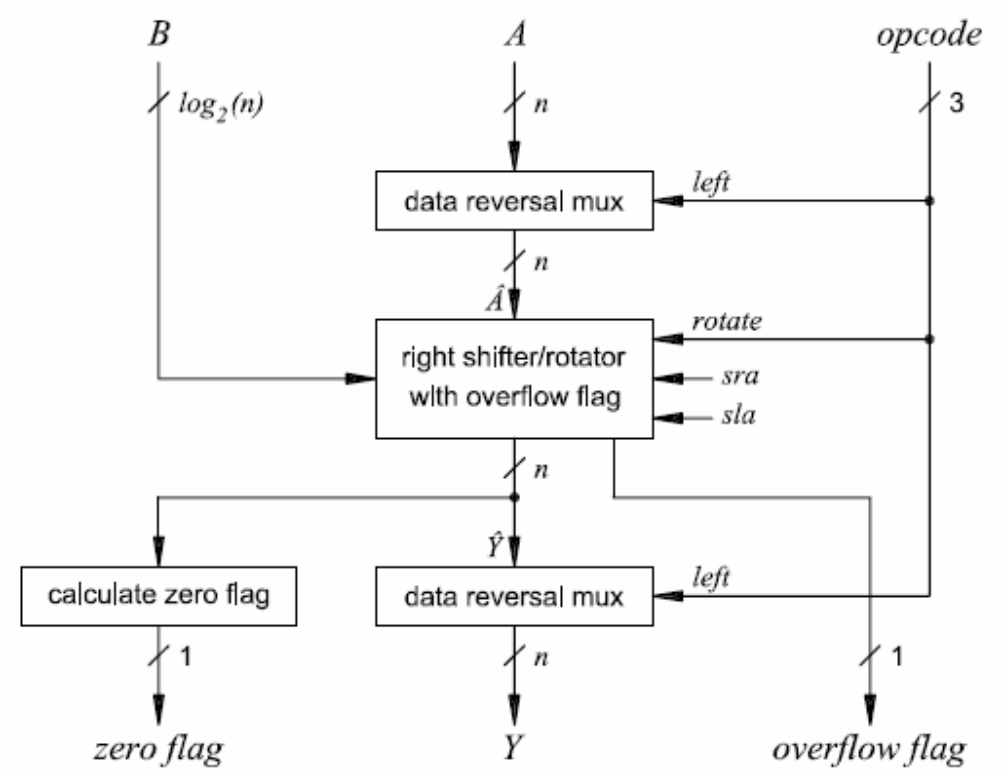
4. Barrel Shifter-i sa inverzijom podataka zasnovani na multiplekserima

Desni pomerač se može proširiti da obavlja i operaciju levog pomeranja dodavanjem reda od n multipleksera pre i posle operacije pomeranja u desno [1]. Kada se obavlja operacija pomeranja u levo, ovi multiplekseri vrše inverziju podatka pre i posle pomeranja u desno. Kada se obavlja pomeranje u desno, nad podacima se ne vrši inverzija. Primera radi, 8-bitni logički pomerač sa inverzijom podataka koji može da vrši pomeranje u levo i u desno prikazan je na slici 4.1.



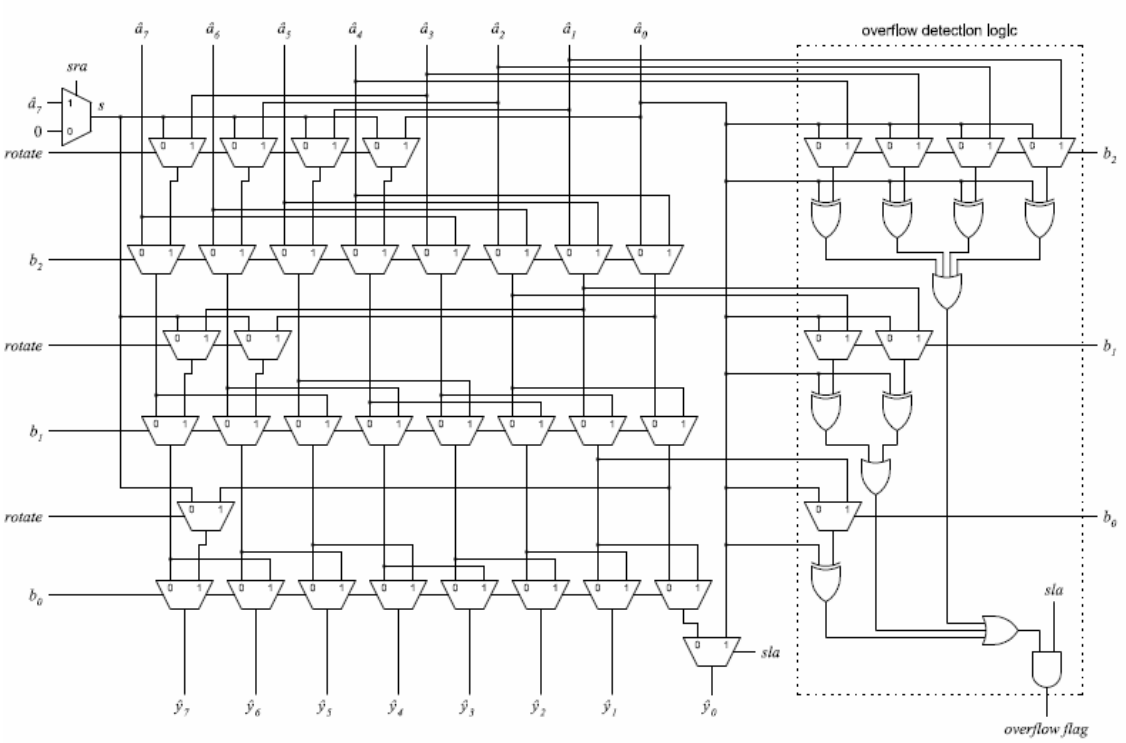
Slika 4.1. 8-bitni logički pomerač sa inverzijom podataka.

Pređašnja tehnika se može preurediti u barrel shifter koji može da obavlja operacije logičkog i aritmetičkog pomeranja u desno i levo, kao i rotacije u desno i u levo. Takav jedan pomerač je prikazan na slici 4.2. Inicijačlno, red od n multipleksera vrši inverziju redosleda podatka ukoliko je kontrolni signal left postavljen na '1' da bi se oformio invertovani ulazni podataka $\hat{A}$. Zatim, n-bitni pomerač/rotator obavlja operaciju desnog pomeranja ili rotacije na podatku $\hat{A}$ da bi se dobio podatak $\hat{Y}$. Konačno, red n multipleksera invertuje podatke ukoliko je left = '1' da bi se dobio konačan rezultat Y.



Slika 4.2. *Barrel Shifter* sa inverzijom podataka zasnovan na multiplekserima.

Nacrt prikazan na slici 4.2 nazvan Barrel Shifter sa inverzijom podataka zasnovan na multiplekserima, takođe detektuje i prekoračenje (Overflow) i rezultat nule. Prekoračenje se može dogoditi isključivo kada se obavlja aritmetičko pomeranje u levo i kada se jedan ili više izbačenih bitova razlikuje od bita znaka. Metoda za paralelnu detekciju prikazana je na slici 4.3.

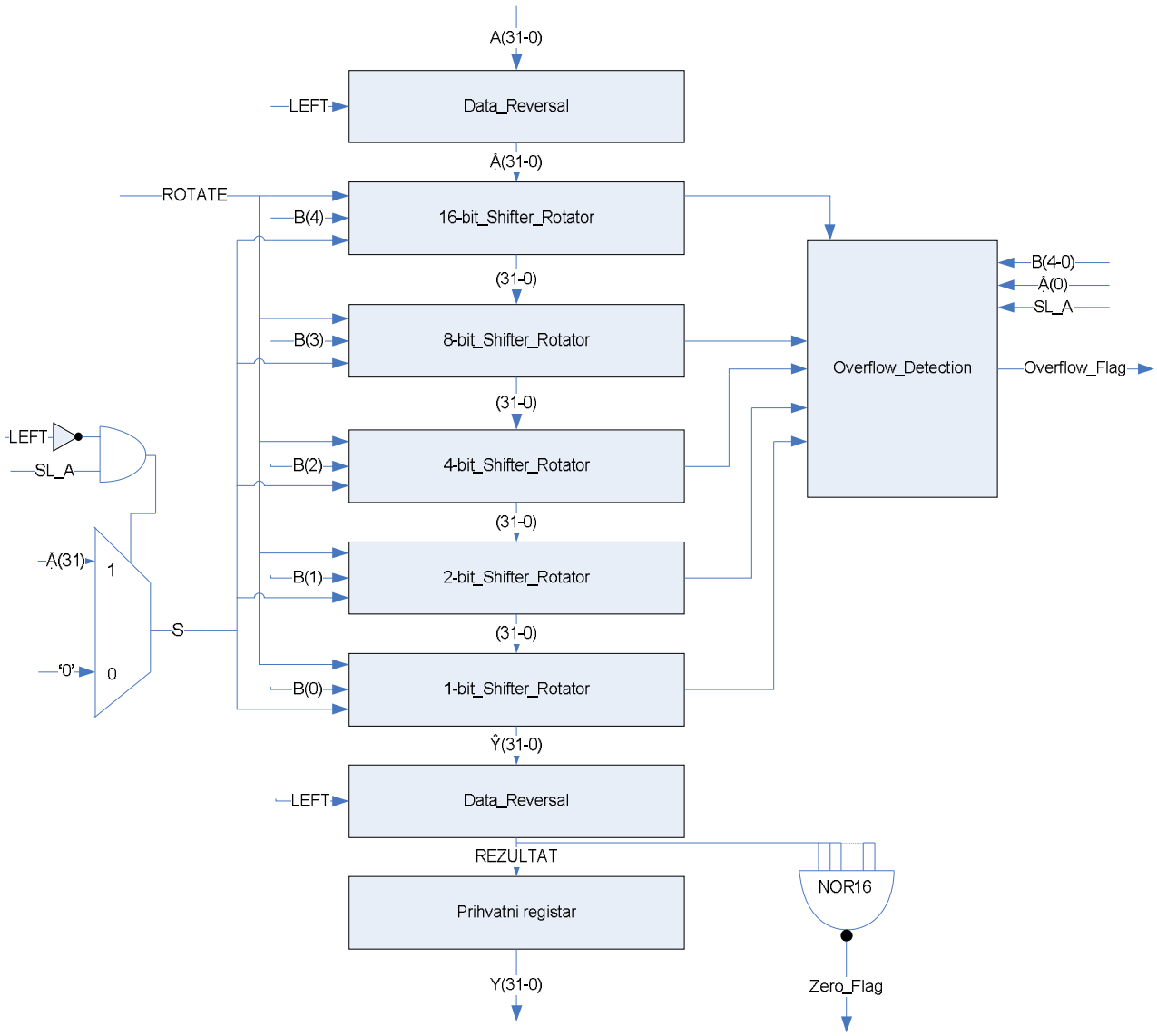


Slika 4.3. 8-bitni pomerač/rotator sa logikom za detekciju prekoračenja.

U svakoj stazi, bitovi koji se izbacuju se XOR-uju sa bitom znaka; ukoliko se nijedan bit ne izbacuje (ne vrši se pomeranj u toj stazi), bit znaka se XOR-uje sam sa sobom. Izlazi XOR gejtova se zatim OR-uju zajedno da bi proizveli fleg prekoračenja (Overflow Flag), čija vrednost iznosi ’1’ kada se diogodi prekoračenje. Dodatni multiplekser postavlja $\hat{y}_0$ na $\hat{a}_0$ kada je sla = ’1’. Fleg nule (Zero Flag), koji je ’1’ kada je Y = 0, dobija se izvršenjem logičke operacije NOR nad svim bitovima podatka $\hat{Y}$.

5. Realizacija Barrel Shifter-a sa inverzijom podataka zasnovanog na multiplekserima u VHDL-u

Blok dijagram celokupnog sistema pomerača prikazan je na slici 5.1. Opis komponenti na VHDL-u kao i simulacija urađena je u progmskom alatu Active HDL. Celokupni pomerač je rastavljen na funkcionalne celine, koje su opisane i testirane, a tek nakon toga instancirane u glavni (Top-Level) dokument.



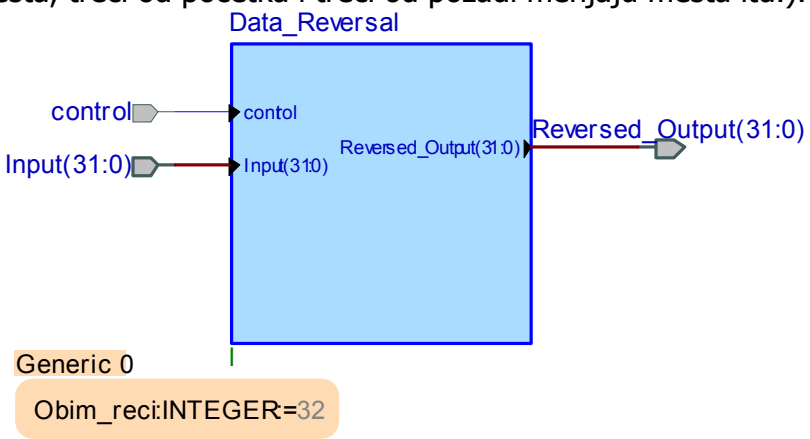
Slika 5.1. Blok dijagram projektovanog *Barrel Shifter*-a.

Na blok dijagramu sa slike 5.1 nalaze se blokovi koji su funkcionalno opisani u ovom poglavlju. U vršnom fajlu Barrel_shifter opisan su još i: Prihvatni registar koji se sinhrono resetuje ulaznim signalom RST i rastućom ivicom sistemskog takta CLK. Upis i reset se vrše na rastuću ivicu sistesmkog takta. U njemu se čuva rezultat 32-bitne operacije pomeranja ili rotacije. Signal **s** predstavlja vrednost bitova koji se upisuju na mesto prvih pomerenih bitova u stazama za pomeranje (prvih 16 bitova kod 16-bit_shift_rotate, na primer). **S** dobija vrednost znaka ulaznog podatka za desno aritmetičko pomeranje, u ostalim situacijama (aritmetčko levo i logičko levo i desno pomeranje) dobija vrednost '0'. U ovom fajlu opisana je i logika za određivanje flega nule (**Zero_Flag**). Logika zapravo prestavlja 16-ulazno NILI kolo na koje se dovodi vrednost rezultata Y.

Kompletan opis svih komponenti u VHDL-u prikazan je u poglavlju 7.

5.1. Opis komponente Data_Reversal

Instancira se dvaput u vršnom dokumentu Barrel_Shifter.vhd. Namena mu je da invertuje podatke pre pomeranja ili rotiranja i nakon pomeranja ili rotiranja ukoliko se obavlja operacija pomeranja ili rotiranja u levo (kao što je objašnjeno u Poglavlju 4). Na signal **contol** se dovodi ulazni signal **LEFT**, koji je na nivou logičke '1' ukoliko se obavlja operacija pomeranja u levo, odnosno na nivou logičke '0' za operacije pomeranja u desno. Kolo dakle predstavlja red multipleksera (32) koji biraju između ulaza **input** ukoliko je **control** = '0' ili invertovanog ulaza (MSB i LSB menjaju mesta, drugi i predposlednji bit u nizu menjaju mesta, treći od početka i treći od pozadi menjaju mesta itd.).



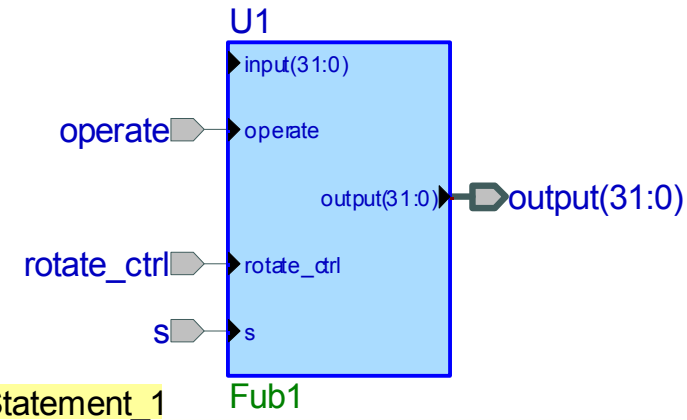
Statement_1

```
process (control, Input)
  begin
    if (control = '1') then
      FOR i IN 0 TO Obim_reci-1 LOOP
       Reversed_output (Obim_reci - 1 - i) <= Input (i);
      END LOOP;
    else
      Reversed_Output <= Input;
    end if;
  end process;
```

5.2. Opis komponente 16-bit_Shifter_Rotator

Komponenta izvršava operaciju pomeranja ili rotiranja u zavisnosti od stanja ulaznog signala **rotate_ctrl**; ukoliko je ’1’, obavlja se funkcija rotiranja, ukoliko je ’0’ izvrašva se operacija pomeranja. Dozvola izvršenja operacije obavlja se signalom **operate** na čiju se pobudu dovodi najviši bit ulaza za količinu pomeraja B (b4). Ukoliko je signal **operate** = ’1’, onda se izvršava operacija, ukoliko ne, ulaz se prosleđuje na izlaz nepromenjen.

Funkcija operacije pomeranja povezuje prvih 16-bitova ulaza na poslednjih 16 bitova izlaza, a na prvih 16 bitova se dovodi vrednost **s**, koji predstavlja spoljni signal kojim se vrši odabir između logičkog i aritmetičkog pomeranja. Ukoliko se izvršava operacije pomeranja, umesto postavljanja bita **s** za vrednost prvih 16 bitova izlaza, dovode se 16 najlakših bitova ulaza koji se prilikom pomerajna istiskuju.



Na ulaz ovog kola dovodi se signal propušten kroz Dara_Reversal blok, a izlaz se povezuje na ulaz bloka 8-bit_Shifter_Rotator.

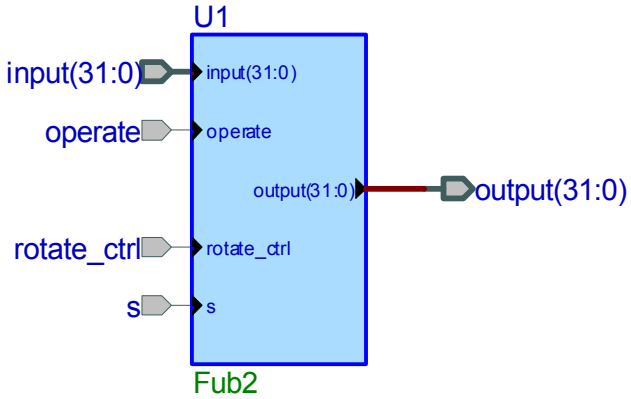
Statement_1

```
process (input, s, rotate_ctrl, operate)
begin
  if (operate = '1') then
    if (rotate_ctrl = '1') then
      output(31 downto 16) <= input (15 do...
      output(15 downto 0) <= input (31 dow...
    elsif (rotate_ctrl = '0') then
      output (15 downto 0) <= input (31 do...
      FOR i IN 0 TO 15 LOOP
        output(31 - i) <= s;
      END LOOP;
    end if;
  else
    output <= input;
  end if;

end process;
```

5.3. Opis komponente 8-bit_Shifter_rotator

Ova komponenta funkcioniše na sličan način kao i predhodno opisana. Na ulaz **operate** se dovodi bit b3 iz ulazne veličine pomeraja B. Ulazni podatak **input** se dobija iz 16-bit_shifter_rotator-a, a izlaz se vezuje na sledeću stazu pomeranja 4-bit_shifter_rotator. U ovoj stazi se poslednji bajt podataka odbacuje, ukoliko se vrši operacija pomeranja, odnosno dovodi na mesto prvog bajta izlaznog podatka za izvršenje operacije rotiranja. Svi ostali bitovi se pomeraju za 8 bitova u desno.



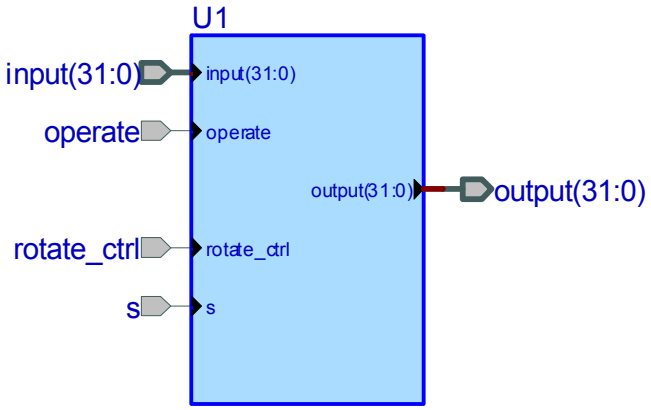
Statement_1

```
process (rotate_ctrl, operate, input, s)
  begin
  if (operate = '1') then
    if (rotate_ctrl = '1') then
      output(31 downto 24) <= input (7 downto 0);
      output(23 downto 16) <= input (31 downto 24);
      output(15 downto 8) <= input (23 downto 16);
      output(7 downto 0) <= input (15 downto 8);
    elsif (rotate_ctrl = '0') then
      output (7 downto 0) <= input (15 downto 8);
      output(15 downto 8) <= input (23 downto 16);
      output(23 downto 16) <= input (31 downto 24);
      FOR i IN 0 TO 7 LOOP
        output(31 - i) <= s;
      END LOOP;
    end if;
  else
    output <= input;
  end if;

end process;
```

5.4. Opis komponente 4-bit_shifter_rotator

Komponenta vrši funkciju ekvivalentu predhodnim, sa izuzetkom da se na ulaz **operate** u ovom slučaju dovodi bit b2 veličine B i da se poslednjig 4 bita odbacuje, odnosno dovodi na prvih 4 bita u zavisnosti da li se izvršava operacija pomeranja ili rotacije. Ulazni singal **input** povezan je sa izlazom 8-bit_shift_rotate, dok se izlaz dovodi na ulazni podatak staze 2-bit_shift_rotate.



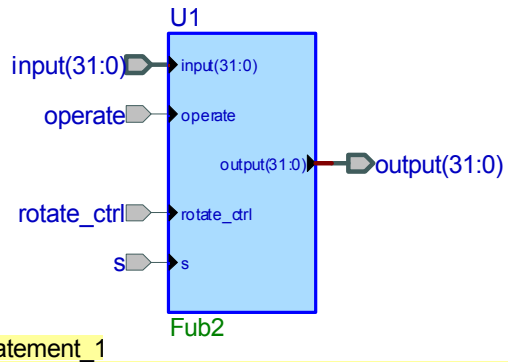
Statement_1

```
process (rotate_ctrl, operate, input, s)
  begin
  if (operate = '1') then
    if (rotate_ctrl = '1') then
      output(31 downto 28) <= input (3 downto 0);
      output(27 downto 24) <= input (31 downto 28);
      output(23 downto 20) <= input (27 downto 24);
      output(19 downto 16) <= input (23 downto 20);
      output(15 downto 12) <= input (19 downto 16);
      output(11 downto 8) <= input (15 downto 12);
      output(7 downto 4) <= input (11 downto 8);
      output(3 downto 0) <= input (7 downto 4);
    elsif (rotate_ctrl = '0') then
      output(27 downto 24) <= input (31 downto 28);
      output(23 downto 20) <= input (27 downto 24);
      output(19 downto 16) <= input (23 downto 20);
      output(15 downto 12) <= input (19 downto 16);
      output(11 downto 8) <= input (15 downto 12);
      output(7 downto 4) <= input (11 downto 8);
      output(3 downto 0) <= input (7 downto 4);
      FOR i IN 0 TO 3 LOOP
        output(31 - i) <= s;
      END LOOP;
    end if;
  else
    output <= input;
  end if;
end process;
```

5.5. Opis komponente 2-bit_shifter_rotator

Ova staza vrši istu funkcio kao i predhodne staze sa izuzetkom da se sada poslednja 2 bita odbacuju za funkciju pomeranja, odnosno dovode na poziciju prva 2 bita za slučaj operacije rotacije, a operacija pomeranja ili rotacije se dozvoljava bitom b1.

Kao ulazni podatak dovodi se izlaz iz 4-bit_shifter_rotator, a izlazni signal se vodi na poslednju stazu pomeranja/rotacije 1-bit_shifter_rotator.

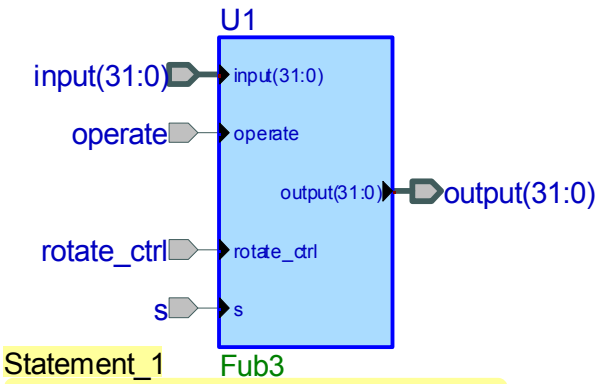


Statement_1

```
process (rotate_ctrl, operate, input, s)
  begin
  if (operate = '1') then
    if (rotate_ctrl = '1') then
      output(31 downto 30) <= input (1 downto 0);
      output(29 downto 28) <= input (31 downto 30);
      output(27 downto 26) <= input (29 downto 28);
      output(25 downto 24) <= input (27 downto 26);
      output(23 downto 22) <= input (25 downto 24);
      output(21 downto 20) <= input (23 downto 22);
      output(19 downto 18) <= input (21 downto 20);
      output(17 downto 16) <= input (19 downto 18);
      output(15 downto 14) <= input (17 downto 16);
      output(13 downto 12) <= input (15 downto 14);
      output(11 downto 10) <= input (13 downto 12);
      output(9 downto 8) <= input (11 downto 10);
      output(7 downto 6) <= input (9 downto 8);
      output(5 downto 4) <= input (7 downto 6);
      output(3 downto 2) <= input (5 downto 4);
      output(1 downto 0) <= input (3 downto 2);
    elsif (rotate_ctrl = '0') then
      output(29 downto 28) <= input (31 downto 30);
      output(27 downto 26) <= input (29 downto 28);
      output(25 downto 24) <= input (27 downto 26);
      output(23 downto 22) <= input (25 downto 24);
      output(21 downto 20) <= input (23 downto 22);
      output(19 downto 18) <= input (21 downto 20);
      output(17 downto 16) <= input (19 downto 18);
      output(15 downto 14) <= input (17 downto 16);
      output(13 downto 12) <= input (15 downto 14);
      output(11 downto 10) <= input (13 downto 12);
      output(9 downto 8) <= input (11 downto 10);
      output(7 downto 6) <= input (9 downto 8);
      output(5 downto 4) <= input (7 downto 6);
      output(3 downto 2) <= input (5 downto 4);
      output(1 downto 0) <= input (3 downto 2);
      output (31) <= s;
      output (30)   <= s;
    end if;
  else
    output <= input;
  end if;
end process;
```

5.6. Opis komponente 1-bit_shifter_rotator

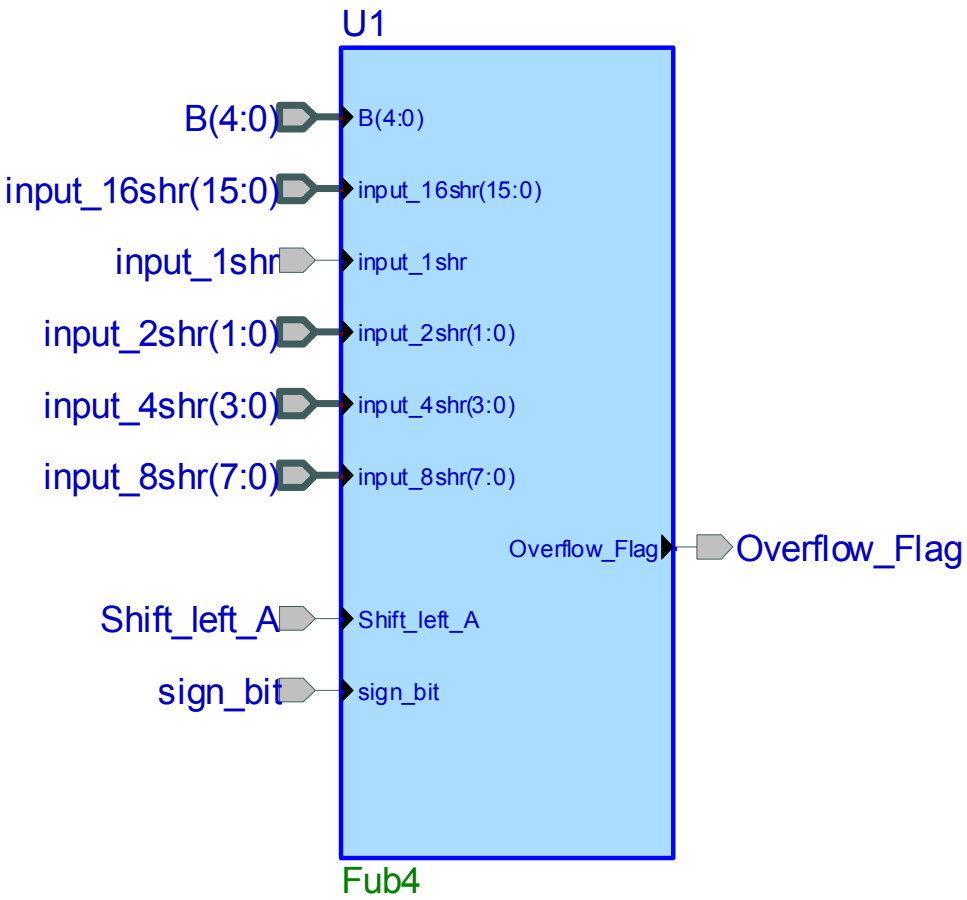
Opet je funkcija ista kao u predhodnim stazama sa promenom da je na **operate** ulaz povezan LSB veličine B, a svi bitovi su pomereni za jednu poziciju u desno osim LSB bita ulaznog podatka koji se odbacuje za funkciju pomeranja, odnosno dovodi na mesto MSB bita izlaznog podatka za funkciju rotacije.




```
process (s, rotate_ctrl, s, input, ope...
begin
  if (operate = '1') then
    if (rotate_ctrl = '1') then
      output (31) <= input (0);
      for i IN 0 TO 30 LOOP
        output (i) <= input (i+1);
      end LOOP;
    elsif (rotate_ctrl = '0') then
      output (31) <= s;
      for i IN 0 TO 30 LOOP
        output (i) <= input (i+1);
      end LOOP;
    end if;
  else
    output <= input;
  end if;
end process;
```

5.7. Opis koponente Overflow_Detection

Namena ove komponente je da odredi da li je došlo do prekoračenja pri levom aritmetičkom pomeranju. Ukoliko je došlo do prekoračenja, rezultat nije važeći, a spoljašnjost se obaveštava o tome izlaznim signalom **Overflow_Flag**. Na ulaz ovog kola se dovode svi bitovi koji se odbacuju iz staza za pomeranje/rotaciju, a zatim se oni upotređuju sa bitom znaka. Ukoliko se ijedan bit koji je izbačen razlikuje od znaka bita, **Overflow_Flag** se postavlja na '1', a dobijeni rezultat na izlazu Barrel_Shifter-a nije važeći.



Statement_1

```
process (input_16shr, input_8shr,input_4shr,
begin
    if (B(4) = '1') then
      muxout16 <= input_16shr;
    elsif (B(4) = '0') then
      FOR i IN 0 TO 15 LOOP
        muxout16(i) <= sign_bit;
      END LOOP;
    end if;
    if (B(3) = '1') then
      muxout8 <= input_8shr;
    elsif (B(3) = '0') then
      FOR i IN 0 TO 7 LOOP
        muxout8(i) <= sign_bit;
      END LOOP;
    end if;
    if (B(2) = '1') then
      muxout4 <= input_4shr;
    elsif (B(2) = '0') then
      FOR i IN 0 TO 3 LOOP
        muxout4(i) <= sign_bit;
      END LOOP;
    end if;
    if (B(1) = '1') then
      muxout2 <= input_2shr;
    elsif (B(1) = '0') then
      FOR i IN 0 TO 1 LOOP
        muxout2(i) <= sign_bit;
      END LOOP;
    end if;
    if (B(0) = '1') then
      muxout1 <= input_1shr;
    elsif (B(0) = '0') then
      muxout1 <= sign_bit;
    end if;
  end process;
```

Statement_2

```
OF16: FOR i IN 0 TO 15 GENER...
      xor16_out(i) <= muxout16(i) xor
  END GENERATE;
```

Statement_3

```
OF_16 <= xor16_out(15) or xor16...
  or xor16_out(8) or xor16_out(7) or
  or xor16_out(1) or xor16_out(0);
```

Statement_4

```
OF8:FOR i IN 0 TO 7 GENERATE
      xor8_out(i) <= muxout8(i) xor ...
  END GENERATE;
```

Statement_5

```
OF_8 <= xor8_out(7) or xor8_out(6
  or xor8_out(1) or xor8_out(0);
```

Statement_6

```
OF4: FOR i IN 0 TO 3 GENERATE
      xor4_out(i) <= muxout4(i) xor ...
  END GENERATE;
```

Statement_7

```
OF_4 <= xor4_out(3) or xor4_out(2
```

Statement_8

```
xor2_out (1) <= muxout2 (1) xor si...
```

Statement_9

```
xor2_out (0) <= muxout2 (0) xor si...
```

Statement_10

```
OF_2 <= xor2_out (1) or xor2_out(0
```

Statement_11

```
OF_1 <= muxout1 xor sign_bit;
```

Statement_12

```
Overflow_Flag <= (OF_16 or OF_8
```

6. Rezultati simulacije

Simulacije je izvršena u programskom alatu Active-HDL. Simulacija je pokazala potpunu ispravnost projektovane komponente, što potvrđuju rezultati priloženi u ovom poglavlju.

A prestavlja ulazni podatak, Y predstavlja izlazni podatak, dok B predstavlja veličinu pomeraja (u testu predstavljen kao decimalni broj).

Test1: Aritmetičko pomeranje u desno. Opkod: ROTATE = 0, SL_A = 1, LEFT = 0.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B
0.000	0	0	0	1	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	1	0	0	1	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	8	0	0	1	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUU...	5
100.000 ns	0	0	1	1	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	0	1	1	1	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	1	1	1	1	0	0	10000000000111000000000000000000	00000000000000000000000000000000	5
200.000 ns	0	1	0	1	0	0	10000000000111000000000000000000	00000000000000000000000000000000	5
250.000 ns	0	0	0	1	0	0	10000000000111000000000000000000	00000000000000000000000000000000	5
350.000 ns	0	1	0	1	0	0	10000000000111000000000000000000	00000000000000000000000000000000	5
350.000 ns	1	1	0	1	0	0	10000000000111000000000000000000	11111100000000001110000000000000	5
450.000 ns	0	0	0	1	0	0	10000000000111000000000000000000	11111100000000001110000000000000	5

Test2: Logičko pomeranje u desno. Opkod: ROTATE = 0, SL_A = 0, LEFT = 0.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y
0.000	0	0	0	0	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUUU...
0.000	1	0	0	0	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUUU...
100.000 ns	0	0	1	0	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUUU...
150.000 ns	0	1	1	0	0	0	10000000000111000000000000000000	UUUUUUUUUUUUUUUUUUUUUUUUU...
150.000 ns	1	1	1	0	0	0	10000000000111000000000000000000	00000000000000000000000000000000
200.000 ns	0	1	0	0	0	0	10000000000111000000000000000000	00000000000000000000000000000000
250.000 ns	0	0	0	0	0	0	10000000000111000000000000000000	00000000000000000000000000000000
350.000 ns	0	1	0	0	0	0	10000000000111000000000000000000	00000000000000000000000000000000
350.000 ns	1	1	0	0	0	0	10000000000111000000000000000000	00000100000000001110000000000000
450.000 ns	0	0	0	0	0	0	10000000000111000000000000000000	00000100000000001110000000000000

Test3: Rotacija u desno. Opkod: ROTATE = 1, SL_A = X, LEFT = 0.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B
0.000	0	0	0	0	1	0	10000000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	1	0	0	0	1	0	10000000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
100.000 ns	0	0	1	0	1	0	10000000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	0	1	1	0	1	0	10000000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	1	1	1	0	1	0	10000000000111000000000000011011	00000000000000000000000000000000	5
200.000 ns	0	1	0	0	1	0	10000000000111000000000000011011	00000000000000000000000000000000	5
250.000 ns	0	0	0	0	1	0	10000000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	0	1	0	0	1	0	10000000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	1	1	0	0	1	0	10000000000111000000000000011011	11011100000000001110000000000000	5
450.000 ns	0	0	0	0	1	0	10000000000111000000000000011011	11011100000000001110000000000000	5

Test4: Aritmetičko pomeranje u levo za slučaj kada nema prekoračenja. Opkod: ROTATE = 0, SL_A = 1, LEFT = 1.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B	Overflow_Flag
0.000	0	0	0	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	1	0	0	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	2	0	0	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	4	0	0	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	8	0	0	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	0
100.000 ns	0	0	1	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	0
150.000 ns	0	1	1	1	0	1	11111111000011100000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	0
150.000 ns	1	1	1	1	0	1	11111111000011100000000000000011011	00000000000000000000000000000000	5	0
200.000 ns	0	1	0	1	0	1	11111111000011100000000000000011011	00000000000000000000000000000000	5	0
250.000 ns	0	0	0	1	0	1	11111111000011100000000000000011011	00000000000000000000000000000000	5	0
350.000 ns	0	1	0	1	0	1	11111111000011100000000000000011011	00000000000000000000000000000000	5	0
350.000 ns	1	1	0	1	0	1	11111111000011100000000000000011011	11000011100000000000001101100000	5	0
450.000 ns	0	0	0	1	0	1	11111111000011100000000000000011011	11000011100000000000001101100000	5	0

Test5: Aritmetičko pomeranje u levo za slučaj kada dolazi do prekoračenja. Opkod: ROTATE = 0, SL_A = 1, LEFT = 1.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B	Overflow_Flag
0.000	0	0	0	1	0	1	10101000000111000000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	1	0	0	1	0	1	10101000000111000000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	U
0.000	7	0	0	1	0	1	10101000000111000000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	1
100.000 ns	0	0	1	1	0	1	10101000000111000000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	1
150.000 ns	0	1	1	1	0	1	10101000000111000000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5	1
150.000 ns	1	1	1	1	0	1	10101000000111000000000000000011011	00000000000000000000000000000000	5	1
200.000 ns	0	1	0	1	0	1	10101000000111000000000000000011011	00000000000000000000000000000000	5	1
250.000 ns	0	0	0	1	0	1	10101000000111000000000000000011011	00000000000000000000000000000000	5	1
350.000 ns	0	1	0	1	0	1	10101000000111000000000000000011011	00000000000000000000000000000000	5	1
350.000 ns	1	1	0	1	0	1	10101000000111000000000000000011011	10000011100000000000001101111111	5	1
450.000 ns	0	0	0	1	0	1	10101000000111000000000000000011011	10000011100000000000001101111111	5	1

Komentar: Došlo je do prekoračenja, rezultat nije važeći.

Test 6: Logičko pomeranje u levo. Opkod: ROTATE = 0, SL_A = 0, LEFT = 1.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B
0.000	0	0	0	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	1	0	0	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	2	0	0	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	4	0	0	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
100.000 ns	0	0	1	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	0	1	1	0	0	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	1	1	1	0	0	1	11111000000111000000000000011011	00000000000000000000000000000000	5
200.000 ns	0	1	0	0	0	1	11111000000111000000000000011011	00000000000000000000000000000000	5
250.000 ns	0	0	0	0	0	1	11111000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	0	1	0	0	0	1	11111000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	1	1	0	0	0	1	11111000000111000000000000011011	00000011100000000000001101100000	5
450.000 ns	0	0	0	0	0	1	11111000000111000000000000011011	00000011100000000000001101100000	5

Test 7: Rotacija u levo. Opkod: ROTATE = 1, SL_A = X, LEFT = 1.

Rezultat testiranja:

Time	D	clk	RST	SL_A	ROTATE	LEFT	A	Y	B
0.000	0	0	0	0	1	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	1	0	0	0	1	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
0.000	4	0	0	0	1	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
100.000 ns	0	0	1	0	1	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	0	1	1	0	1	1	11111000000111000000000000011011	UUUUUUUUUUUUUUUUUUUUUUUUU...	5
150.000 ns	1	1	1	0	1	1	11111000000111000000000000011011	00000000000000000000000000000000	5
200.000 ns	0	1	0	0	1	1	11111000000111000000000000011011	00000000000000000000000000000000	5
250.000 ns	0	0	0	0	1	1	11111000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	0	1	0	0	1	1	11111000000111000000000000011011	00000000000000000000000000000000	5
350.000 ns	1	1	0	0	1	1	11111000000111000000000000011011	00000011100000000000001101111111	5
450.000 ns	0	0	0	0	1	1	11111000000111000000000000011011	00000011100000000000001101111111	5

Test 8: Testiranje ispravnosti flega nule (Zero_Flag) Opkod: ROTATE = 0, SL_A = 0, LEFT = 0.

Rezultat testiranja:

Time	Delta	clk	RST	SL_A	ROTATE	LEFT	A	Y	B	Zero_Flag
0.000	0	0	0	0	0	0	00000003	UUUUUUUU	03	U
0.000	1	0	0	0	0	0	00000003	UUUUUUUU	03	0
0.000	4	0	0	0	0	0	00000003	UUUUUUUU	03	0
0.000	9	0	0	0	0	0	00000003	UUUUUUUU	03	1
100.000 ns	0	0	1	0	0	0	00000003	UUUUUUUU	03	1
150.000 ns	0	1	1	0	0	0	00000003	UUUUUUUU	03	1
150.000 ns	1	1	1	0	0	0	00000003	00000000	03	1
200.000 ns	0	1	0	0	0	0	00000003	00000000	03	1
250.000 ns	0	0	0	0	0	0	00000003	00000000	03	1
350.000 ns	0	1	0	0	0	0	00000003	00000000	03	1
450.000 ns	0	0	0	0	0	0	00000003	00000000	03	1

KOMENTAR: testiranjem svih mogućih situacija pokazana je potpuna ispravnost projektovanog kola.

7. Kompletan VHDL kod

Opis top-level entiteta: Barrel_Shiftrer.vhd.

```
ibrary IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Barel_Shifter is
	port(
		clk : in STD_LOGIC;
		RST : in STD_LOGIC;
		SL_A : in STD_LOGIC;
		ROTATE : in STD_LOGIC;
		LEFT : in STD_LOGIC;
		A : in STD_LOGIC_VECTOR(31 downto 0);
		B : in STD_LOGIC_VECTOR(4 downto 0);
		Y : out STD_LOGIC_VECTOR (31 downto 0);
		Overflow_Flag : out STD_LOGIC;
		Zero_Flag : out STD_LOGIC
	  );
end Barel_Shifter;

--}} End of automatically maintained section

architecture Barel_Shifter of Barel_Shifter is

component Data_Reversal is
	GENERIC(Obim_reci : INTEGER := 32);
	port(
		control : in STD_LOGIC;
		Input : in STD_LOGIC_VECTOR(31 downto 0);
		Reversed_Output : out STD_LOGIC_VECTOR(31 downto 0)
	  );
end component;
component \16-bit_right_shift-rotator\ is
	port(
		rotate_ctrl, operate : in STD_LOGIC;
		s : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
	  );
end component;
component \8-bit_right_shift-rotator\ is
	port(
		rotate_ctrl, s : in STD_LOGIC;
		operate : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
	  );
end component;
component \4-bit_right_shift-rotate\ is
	port(
		operate : in STD_LOGIC;
		s : in STD_LOGIC;
		rotate_ctrl : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
	  );
end component;
component \2-bit_shift-rotate\ is
	port(
		s : in STD_LOGIC;
		operate : in STD_LOGIC;
		rotate_ctrl : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
```

```
		);
end component;
component \1-bit_shift-rotate\ is
		port(
			s : in STD_LOGIC;
			rotate_ctrl : in STD_LOGIC;
			operate : in STD_LOGIC;
			input : in STD_LOGIC_VECTOR(31 downto 0);
			output : out STD_LOGIC_VECTOR(31 downto 0)
		);
end component;
component Overflow_detection is
		port(
			input_1shr : in STD_LOGIC;
			--operation : in STD_LOGIC;
			sign_bit : in STD_LOGIC;
			Shift_left_A : in STD_LOGIC;
			input_16shr : in STD_LOGIC_VECTOR(15 downto 0);
			input_8shr : in STD_LOGIC_VECTOR(7 downto 0);
			input_4shr : in STD_LOGIC_VECTOR(3 downto 0);
			input_2shr : in STD_LOGIC_VECTOR(1 downto 0);
			B : in STD_LOGIC_VECTOR(4 downto 0);
			Overflow_Flag : out STD_LOGIC
		);
end component;
signal Reversed_input, Sh16_out, Sh8_out, Sh4_out, Sh2_out, Sh1_out, REZULTAT : std_logic_vector (31 downto 0);
signal s, reversed_sign : std_logic;
begin
s <= Reversed_input(31) when (SL_A = '1' and LEFT = '0') else
		'0';
		reversed_sign <= Reversed_input (0) when SL_A = '1' else Sh1_out (0);
		Detektorprekoracenja: Overflow_detection
		port map
		(input_16shr => Reversed_input (16 downto 1),
		input_8shr => sh16_out (8 downto 1),
		input_4shr => sh8_out (4 downto 1),
		input_2shr => sh4_out (2 downto 1),
		input_1shr => sh2_out(1),
		B => B,
		sign_bit => Reversed_input (0),
		Overflow_Flag => Overflow_flag,
		Shift_left_A => SL_A
		);
		Reversenaulazu: Data_Reversal
		port map
		(control => LEFT,
		input => A,
		Reversed_Output => Reversed_input);

		Shifter16bits : \16-bit_right_shift-rotator\
		port map
		(rotate_ctrl => ROTATE,
		operate => B(4),
		s => s,
		input => Reversed_input,
		output => Sh16_out
		);
		Shifter8bits : \8-bit_right_shift-rotator\
		port map
		(rotate_ctrl => ROTATE,
		s => s,
		operate => B(3),
		input => Sh16_out,
		output => Sh8_out
		);
		Shifter4bits : \4-bit_right_shift-rotate\
		port map
		(rotate_ctrl => ROTATE,
		s => s,
		operate => B(2),
```

```
		input => Sh8_out,
		output => Sh4_out
		);
		Shifter2bits : \2-bit_shift-rotate\
		port map
		(rotate_ctrl => ROTATE,
		operate => B(1),
		s => s,
		input => Sh4_out,
		output => Sh2_out
		);
		Shifter1bit : \1-bit_shift-rotate\
		port map
		(operate => B(0),
		rotate_ctrl => ROTATE,
		s => s,
		input => Sh2_out,
		output => Sh1_out
		);

		Rreversenaizlazu: Data_Reversal
		port map
		(control => LEFT,
		input(31 downto 1) => Sh1_out(31 downto 1),
		input(0) => reversed_sign,
		Reversed_Output => REZULTAT
		);
		Zero_Flag <= '1' when REZULTAT = "00000000000000000000000000000000" else '0';
		process (clk)
		begin
			if (clk'event and clk = '1') then
				if (RST = '1') then
					Y <= (OTHERS => '0');
				else
					Y <= REZULTAT;
				end if;
			end if;
		end process;

		 -- enter your statements here --

end Barel_Shifter;
```

Opis komponente: Data_Reversal.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Data_Reversal is
		GENERIC(Obim_reci : INTEGER := 32);
		 port(
			control : in STD_LOGIC;
			Input : in STD_LOGIC_VECTOR(31 downto 0);
			Reversed_Output : out STD_LOGIC_VECTOR(31 downto 0)
		    );
end Data_Reversal;

architecture Data_Reversal of Data_Reversal is
begin
		process  (control, Input)
		begin
			if (control = '1') then
				FOR i IN 0 TO Obim_reci -1 LOOP
				Reversed_output (Obim_reci - 1 - i) <= Input (i);
				END LOOP;
			else
				Reversed_Output <= Input;
```

```
                end if;
        end process;


end Data_Reversal;
```

Opis komponente: 1-bit_shift_rotate.vhd.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity \1-bit_shift-rotate\ is
        port(
                s : in STD_LOGIC;
                rotate_ctrl : in STD_LOGIC;
                operate : in STD_LOGIC;
                input : in STD_LOGIC_VECTOR(31 downto 0);
                output : out STD_LOGIC_VECTOR(31 downto 0)
            );
end \1-bit_shift-rotate\;

--}} End of automatically maintained section

architecture \1-bit_shift-rotate\ of \1-bit_shift-rotate\ is
begin

process (s, rotate_ctrl, s, input, operate)
begin
        if (operate = '1') then
                if (rotate_ctrl = '1') then
                        output (31) <= input (0);
                        for i IN 0 TO 30 LOOP
                                output (i) <= input (i+1);
                        end LOOP;
                elsif (rotate_ctrl = '0') then
                        output (31) <= s;
                        for i IN 0 TO 30 LOOP
                                output (i) <= input (i+1);
                        end LOOP;
                end if;
        else
                output <= input;
        end if;
end process;
end \1-bit_shift-rotate\;
```

Opis komponente: 2-bit_Shift_Rotate.vhd.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity \2-bit_shift-rotate\ is
        port(
                s : in STD_LOGIC;
                operate : in STD_LOGIC;
                rotate_ctrl : in STD_LOGIC;
                input : in STD_LOGIC_VECTOR(31 downto 0);
                output : out STD_LOGIC_VECTOR(31 downto 0)
            );
end \2-bit_shift-rotate\;

--}} End of automatically maintained section

architecture \2-bit_shift-rotate\ of \2-bit_shift-rotate\ is
begin

process (rotate_ctrl, operate, input, s)
```

```
	begin
	if (operate = '1') then
		if (rotate_ctrl = '1') then
			output(31 downto 30) <= input (1 downto 0);
			output(29 downto 28) <= input (31 downto 30);
			output(27 downto 26) <= input (29 downto 28);
			output(25 downto 24) <= input (27 downto 26);
			output(23 downto 22) <= input (25 downto 24);
			output(21 downto 20) <= input (23 downto 22);
			output(19 downto 18) <= input (21 downto 20);
			output(17 downto 16) <= input (19 downto 18);
			output(15 downto 14) <= input (17 downto 16);
			output(13 downto 12) <= input (15 downto 14);
			output(11 downto 10) <= input (13 downto 12);
			output(9 downto 8) <= input (11 downto 10);
			output(7 downto 6) <= input (9 downto 8);
			output(5 downto 4) <= input (7 downto 6);
			output(3 downto 2) <= input (5 downto 4);
			output(1 downto 0) <= input (3 downto 2);
		elsif (rotate_ctrl = '0') then
			output(29 downto 28) <= input (31 downto 30);
			output(27 downto 26) <= input (29 downto 28);
			output(25 downto 24) <= input (27 downto 26);
			output(23 downto 22) <= input (25 downto 24);
			output(21 downto 20) <= input (23 downto 22);
			output(19 downto 18) <= input (21 downto 20);
			output(17 downto 16) <= input (19 downto 18);
			output(15 downto 14) <= input (17 downto 16);
			output(13 downto 12) <= input (15 downto 14);
			output(11 downto 10) <= input (13 downto 12);
			output(9 downto 8) <= input (11 downto 10);
			output(7 downto 6) <= input (9 downto 8);
			output(5 downto 4) <= input (7 downto 6);
			output(3 downto 2) <= input (5 downto 4);
			output(1 downto 0) <= input (3 downto 2);
			output (31) <= s;
			output (30)		<= s;
		end if;
	else
		output <= input;
	end if;
end process;

end \2-bit_shift-rotate\;
```

Opis komponente: 4-bit_Shift_Rotate.vhd.

```
ibrary IEEE;
use IEEE.STD_LOGIC_1164.all;

entity \4-bit_right_shift-rotate\ is
	port(
		operate : in STD_LOGIC;
		s : in STD_LOGIC;
		rotate_ctrl : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
	);
end \4-bit_right_shift-rotate\;

--}} End of automatically maintained section

architecture \4-bit_right_shift-rotate\ of \4-bit_right_shift-rotate\ is
begin

process (rotate_ctrl, operate, input, s)
	begin
	if (operate = '1') then
		if (rotate_ctrl = '1') then
```

```
                output(31 downto 28) <= input (3 downto 0);
                output(27 downto 24) <= input (31 downto 28);
                output(23 downto 20) <= input (27 downto 24);
                output(19 downto 16) <= input (23 downto 20);
                output(15 downto 12) <= input (19 downto 16);
                output(11 downto 8) <= input (15 downto 12);
                output(7 downto 4) <= input (11 downto 8);
                output(3 downto 0) <= input (7 downto 4);
        elsif (rotate_ctrl = '0') then
                output(27 downto 24) <= input (31 downto 28);
                output(23 downto 20) <= input (27 downto 24);
                output(19 downto 16) <= input (23 downto 20);
                output(15 downto 12) <= input (19 downto 16);
                output(11 downto 8) <= input (15 downto 12);
                output(7 downto 4) <= input (11 downto 8);
                output(3 downto 0) <= input (7 downto 4);
                FOR i IN 0 TO 3 LOOP
                        output(31 - i) <= s;
                END LOOP;
        end if;
    else
        output <= input;
    end if;
end process;

end \4-bit_right_shift-rotate\;
```

Opis komponente: 8-bit_Shift_Rotate.vhd.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity \8-bit_right_shift-rotator\ is
     port(
          rotate_ctrl, s : in STD_LOGIC;
          operate : in STD_LOGIC;
          input : in STD_LOGIC_VECTOR(31 downto 0);
          output : out STD_LOGIC_VECTOR(31 downto 0)
       );
end \8-bit_right_shift-rotator\;

--}} End of automatically maintained section

architecture \8-bit_right_shift-rotator\ of \8-bit_right_shift-rotator\ is
begin
    process (rotate_ctrl, operate, input, s)
    begin
    if (operate = '1') then
        if (rotate_ctrl = '1') then
                output(31 downto 24) <= input (7 downto 0);
                output(23 downto 16) <= input (31 downto 24);
                output(15 downto 8) <= input (23 downto 16);
                output(7 downto 0) <= input (15 downto 8);
        elsif (rotate_ctrl = '0') then
                output (7 downto 0) <= input (15 downto 8);
                output(15 downto 8) <= input (23 downto 16);
                output(23 downto 16) <= input (31 downto 24);
                FOR i IN 0 TO 7 LOOP
                        output(31 - i) <= s;
                END LOOP;
        end if;
    else
        output <= input;
    end if;

end process;

end \8-bit_right_shift-rotator\;
```

Opis komponente: 16-bit_Shift_Rotate.vhd.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity \16-bit_right_shift-rotator\ is
	port(
		rotate_ctrl, operate : in STD_LOGIC;
		s : in STD_LOGIC;
		input : in STD_LOGIC_VECTOR(31 downto 0);
		output : out STD_LOGIC_VECTOR(31 downto 0)
	);
end \16-bit_right_shift-rotator\;

--}} End of automatically maintained section

architecture \16-bit_right_shift-rotator\ of \16-bit_right_shift-rotator\ is
begin

process (input, s, rotate_ctrl, operate)
begin
	if (operate = '1') then
		if (rotate_ctrl = '1') then
			output(31 downto 16) <= input (15 downto 0);
			output(15 downto 0) <= input (31 downto 16);
		elsif (rotate_ctrl = '0') then
			output (15 downto 0) <= input (31 downto 16);
			FOR i IN 0 TO 15 LOOP
				output(31 - i) <= s;
			END LOOP;
		end if;
	else
		output <= input;
	end if;

end process;

end \16-bit_right_shift-rotator\;
```

Opis komponente: Overslow_Detection.vhd.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Overflow_detection is
	port(
		input_1shr : in STD_LOGIC;
		--operation : in STD_LOGIC;
		sign_bit : in STD_LOGIC;
		Shift_left_A : in STD_LOGIC;
		input_16shr : in STD_LOGIC_VECTOR(15 downto 0);
		input_8shr : in STD_LOGIC_VECTOR(7 downto 0);
		input_4shr : in STD_LOGIC_VECTOR(3 downto 0);
		input_2shr : in STD_LOGIC_VECTOR(1 downto 0);
		B : in STD_LOGIC_VECTOR(4 downto 0);
		Overflow_Flag : out STD_LOGIC
	);
end Overflow_detection;

--}} End of automatically maintained section

architecture Overflow_detection of Overflow_detection is
signal muxout16, xor16_out : std_logic_vector (15 downto 0);
signal muxout8, xor8_out : std_logic_vector (7 downto 0);
signal muxout4, xor4_out : std_logic_vector (3 downto 0);
signal muxout2, xor2_out : std_logic_vector (1 downto 0);
signal muxout1, OF_16, OF_8, OF_4, OF_2, OF_1 : std_logic;
begin
```

```
process (input_16shr, input_8shr,input_4shr,input_2shr,input_1shr,sign_bit,B,Shift_left_A)
begin
                        if (B(4) = '1') then
                                muxout16 <= input_16shr;
                        elsif (B(4) = '0') then
                                FOR i IN 0 TO 15 LOOP
                                        muxout16(i) <= sign_bit;
                                END LOOP;
                        end if;
                        if (B(3) = '1') then
                                muxout8 <= input_8shr;
                        elsif (B(3) = '0') then
                                FOR i IN 0 TO 7 LOOP
                                        muxout8(i) <= sign_bit;
                                END LOOP;
                        end if;
                        if (B(2) = '1') then
                                muxout4 <= input_4shr;
                        elsif (B(2) = '0') then
                                FOR i IN 0 TO 3 LOOP
                                        muxout4(i) <= sign_bit;
                                END LOOP;
                        end if;
                        if (B(1) = '1') then
                                muxout2 <= input_2shr;
                        elsif (B(1) = '0') then
                                FOR i IN 0 TO 1 LOOP
                                        muxout2(i) <= sign_bit;
                                END LOOP;
                        end if;
                        if (B(0) = '1') then
                                muxout1 <= input_1shr;
                        elsif (B(0) = '0') then
                                muxout1 <= sign_bit;
                        end if;
        end process;
        OF16: FOR i IN 0 TO 15 GENERATE
                xor16_out(i) <= muxout16(i) xor sign_bit;
        END GENERATE;
        OF_16 <= xor16_out(15) or xor16_out(14) or xor16_out(13) or xor16_out(12) or xor16_out(11) or
xor16_out(10) or xor16_out(9)
        or xor16_out(8) or xor16_out(7) or xor16_out(6) or xor16_out(5) or xor16_out(4) or xor16_out(3) or
xor16_out(2)
        or xor16_out(1) or xor16_out(0);

        OF8:FOR i IN 0 TO 7 GENERATE
                xor8_out(i) <= muxout8(i) xor sign_bit;
        END GENERATE;
        OF_8 <= xor8_out(7) or xor8_out(6) or xor8_out(5) or xor8_out(4) or xor8_out(3) or xor8_out(2)
        or xor8_out(1) or xor8_out(0);

        OF4: FOR i IN 0 TO 3 GENERATE
                xor4_out(i) <= muxout4(i) xor sign_bit;
        END GENERATE;
        OF_4 <= xor4_out(3) or xor4_out(2) or xor4_out(1) or xor4_out(0);
        xor2_out (1) <= muxout2 (1) xor sign_bit;
        xor2_out (0) <= muxout2 (0) xor sign_bit;
        OF_2 <= xor2_out (1) or xor2_out(0);--ISPRAVKA!
        OF_1 <= muxout1 xor sign_bit;

        Overflow_Flag <= (OF_16 or OF_8 or OF_4 or OF_2 or OF_1) and Shift_left_A;



end Overflow_detection;
```

8. Reference i literatura

1. A. Ito, "Barrel Shifter," U.S. Patent 4,829,460, May 1989.
2. J. Muwafi, G. Fettweis, and H. Neff, "Circuit for Rotating, Left Shifting, or Right Shifting Bits," U.S.Patent 5.978,822, December 1995.
3. A. Yamaguchi, "Bidirectional Shifter," U.S. Patent 5,262,971, November 1993.