УНИВЕРЗИТЕТ
У НОВОМ САДУ

ФАКУЛТЕТ
ТЕХНИЧКИХ НАУКА

Трг Доситеја Обрадовића 6, 21000 Нови Сад, Југославија
Деканат: 021 350-413; 021 450-810; Централа: 021 350-122
Рачуноводство: 021 58-220; Студентска служба: 021 350-763
Телефакс: 021 58-133; e-mail: ftndean@uns.ns.ac.yu

PROJEKAT

iz predmeta:

Digitalni sistemi otporni na otkaz

TEMA PROJEKTA:

Projektrovanje ALU jedinice otporne na otkaz

TEKST ZADATKA:

Korišćenjem „Self Purging“ tehnike hibridne hardverske redundanse, modelovati u VHDL-u ALU jedinicu koja mora imati mogućnost tolerancije kvara. Parametrizovati dizajn na takav način da korisnik može odrediti broj redundantnih modula. Demonstrirati rad sistema na procesoru i uporediti rezultate sa običnom ALU jedinicom.

Mentor
Rastislav Struharik

Studenti:
Đorđe Mišeljić E1 5/2018
Nikola Kovačević E1 4/2018

U Novom Sadu, 6. septembra 2019 godine

1 Uvod

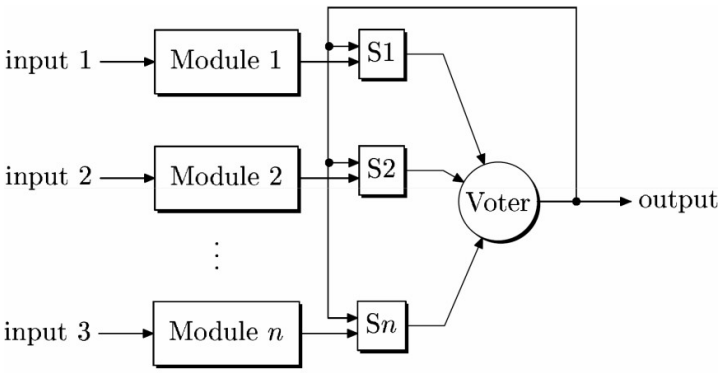
Sve češća upotrebe procesora u oblastima u kojima greška može prouzrokovati katastrofu je dovela do potrebe za procesorima koji mogu tolerisati greške (kako tranzijentne tako i hardverske). Tranzijentne greške se lako mogu detektovati i ispraviti tehnikama vremenske redundantnosti, dok je za uspostavljanje otpornosti na trajne kvarove potrebno implementirati neku od tehnika hardverske tolerancije na kvarove. Kao jedan od kompleksnijih modula sa najvećim brojem logičkih kola u procesoru, aritmetičko logička jedinica (ALU) ima najveću verovatnoću pojave trajnog kvara. Tipična "stuck-at-fault" greška u ALU jedinici može dovesti do odstupanja u rezultatima koji bi se teško primetili a lako doveli kontrolisani sistema u opsanost.

Postoje različiti načini za poboljšavanje pouzdanosti sistema. Jedan je korišćenje "Stand-by" redundancije u kojoj se pri detekciji kvara pokvareni modul zamenjuje ispravnim, koji je do tog trenutka bio isključen. Ovakve tehnike su često energetski efikasnije, ali u slučaju "hard" sistema za rad u relnom vremenu, kašnjenja ovog tipa nisu dopuštena već je potrebno pre oporavka i maskirati grešku. Sistemi sa masivnom redundancijom ostvaruju maskiranje gresaka višestrukim repliciranjem kritičnih modula. Kombinovanjem ove dve tehnike nastaju sistemi sa hibridnom redundanicjom.

Hibridno redundantni sistemi imaju mnoge prednosti u odnosu na pasivne i aktivne tehnike koje su prethodno pomenute, ali često zahtevaju implementaciju komplikovanih prekidačkih mehanizama za prosleđivanje tačnog rešenja. Ovi mehanizmi unose dodatnu potencijalu tačku otkaza. Self-purging tehnika ima sve pogodnosti hibridnih sistema dok je prekidački mehanizam jednostan a samim tim pravi pouzdaniji i manje zahtevan sistem. Jednstavnost prekidačkog mehanizma je posebno pogodna za implementaciju u procesorima jer je bitno uvesti što manje propagaciono kašnjenje u neku od faza pajplajna. Dodatna pogodnost kod uvođenja hibridne tehnike nad ALU jedinicom je da su većina današnjih procesora superskalarni i već poseduju nekoliko ALU jedinica kako bi izvršavali više instrukcija po periodi taktnog signala. Ovo nam omogućava da uz pomoć jednostvanog prekidačkog mehanizma i dodatnih multipleksera omogućimo da supreskalarni procesor iskoristi dodatne ALU jedinice za pouzdanost umesto performance. Iz ovog razloga će sistem, koji je tema ovog projekta, biti parametrizovan za lako implementiranje u bilo koji skalarni procesor sa proizvoljnim brojem ALU jedinica.

2 *Self-purging* sistem

Generalna struktura self-purging sistema sa jednim izlazom je prikazana na slici 1. Sistem se sastoji od *n* repliciranih modula čija se pouzdanost želi povećati, njima pridruženih prekidača i jednog glasača. U ovom sistemu ne postoji razlika između glavnog i pomoćnih modula već svaka od instanci ima pravo glasa u sistemu i ima sebi pridružen prekidač *Sn.* Prekidač isključuje modul iz sistema ukoliko njegovo rešenje odstupa od rešenja koje daje kompletan sistem, ovo je naznaka kvara u tom modulu. Glasač preuzima sve izlaze pojedinačnih modula i prosleđuje na izlaz ono koje je najviše zastupljeno u sistemu.



Slika 1. Generalna struktura self-purging sistema

Self-purging sistem sa *n* modula može da maskira *n-2* kvara u modulima. Kada se ostanu samo dva modula, sistem će biti u mogućnosti da detektuje sledeću grešku, ali neće biti u mogućnosti da odredi koji od dva modula je otkazao. Kako svi moduli rade paralelno, možemo predpostaviti da će greške biti međusobno nezavisne. Dovoljno je da dva modula rade kako bi sistem dao pravo rešenje na izlazu. Ako su glasač i prekidači savršeni, i ako smatramo da sve instance modula imaju jednaku pouzdanost $R_1=R_2=...=R_n=R$, onda sistem neće biti pouzdan ukoliko su svi moduli otkazali (verovatnoća $(1-R)^n$) ili ako radi samo jedan (verovatnoća $R(1-R)^{n-1}$). Iz ove dve verovatnoće možemo izvesti pouzdanost sistema kao:

$$R_{sp}= 1- ((1-R)^n+nR(1-R)^{n-1})$$

3 Implementacija sistema u VHDL-u

Model sistema je pisan u VHDL jeziku i parametrizovan je parametrom *NUM_MODULES* koji predstavlja broj instanci modula koji se replicira. Sistem čine tri fajla: *ALU_ft, treshold_voter i ft_pkg*.

Paket *ft_pkg* je uključen u oba modula i u njemu je definisan tip *array32_t* kao neograničen niz s*td_logic_vector*-a širine 32 bita. Ovaj tip je neophodan kako bi se sistem mogao parametrizovati. U paketu su takođe definisane funkcije koje će biti iskorištene u modulu koji vrši glasanje pa će više reči o njima biti u nastavku teksta.

U ALU_ft modulu su instancirane ALU jedinice, modelovana je prekidačka logika i instanciran je glasač, treshold_voter, čiji se bihevijalni opis nalazi u istoimenom fajlu. ALU_ft modul ima isti interfejs kao i obična ALU jedinica: taktni i reset signal *clk* i *reset*, 32-bitne ulaze *a_i* i *b_i*, 32-bitni izlaz *res_o*, 5-bitni ulaz koji govori operaciju *op_i* i signale nule i prekoračenja *zero_o* i *of_o*.

Za instanciranje ALU jedinica se koristi for generate naredba data u listingu 1. Ulazi *a_i*, *b_i* svih ALU jedinica se vezuju na ulaze *a_i* i *b_i* ALU_ft modula. Na ovaj način sve ALU jedinice imaju istu pobudu pa bi trebali imati i ista rešenja. Izlazi *res_o* se vezuju za pomoćni signal tipa *array32_t* koji se u trenutku instanciranja modula ograničava na *NUM_MODULES* članova, dok se jednobitni izlazi *zero_o* i *of_o* vezuju na pomoćne signale tipa *std_logic_vector* širine *NUM_MODULES*. Na ovakav način se lako može pristupiti signalima određene instance ALU jedinice pomoću indeksa *i*.

```
instantiate_alus:
for i in 0 to NUM_MODULES-1 generate
  alu_inst: entity work.ALU(behavioral)
  generic map (WIDTH => 32)
  port map (    a_i => a_i,
                b_i => b_i,
                op_i => op_i,
                res_o => alu_res_array_s(i),
                zero_o => zero_s(i),
                of_o => of_s(i));
end generate instantiate_alus;
```

Listing 1. Instanciranje ALU jedinica

Isklujučivanje neke od ALU jedinica nakon što se desi kvar se svodi na onemogućavanje da ista glasa pri donošenju konačnog rezultata. Iz ovog razloga je potrebno da prekidači imaju mogućnost postavljanja rezultata svake od ALU jedinica na nulu. Svaka od ALU jedinica ima sebi pridružen flip-flop koji signalizira da li je u aktivnom (rezultat validan) ili neaktivnom (kvar). Kako ALU jedinice čiji su rezultati na nuli ne bi imali uticaja pri donošenju rezultata, glasač mora biti realizovan kao "treshold" kolo. Treshold kolo prima više jednobitnih ulaza, sabira ih, i ukoliko je zbir veći od neke granice (treshold), na izlazu daje logičku jedinicu a u suprotnom slučaju nulu. Za 32-bitne signale je potrebno replicirati ovu logiku 32 puta za svaki od bita pojedinačno.

U VHDL modelu su flip-flopovi realizovani kao jedan registar širine *NUM_MODULES (videti kodni listing 2).* Pri resetu sistema se svi biti postavljaju na jedinice, što signalizira da su sve ALU jedinice funkcionalne. Kada se neki od bita postavi na nulu pomoću signala *alu_valid_s* (zbog otkaza neke ALU jedinice), njega je jedino moguće postaviti na jedinicu pomoću reseta. Na ovaj način se svi kvarovi tretiraju kao trajni.

```
unit_valid_register:
  process (clk) is
  begin
    if(rising_edge(clk))then
      if(reset='0')then
        alu_valid_reg <= (others=>'1');
      else
        alu_valid_reg <= alu_valid_reg and alu_valid_s;
      end if;
    end if;
  end process;
```

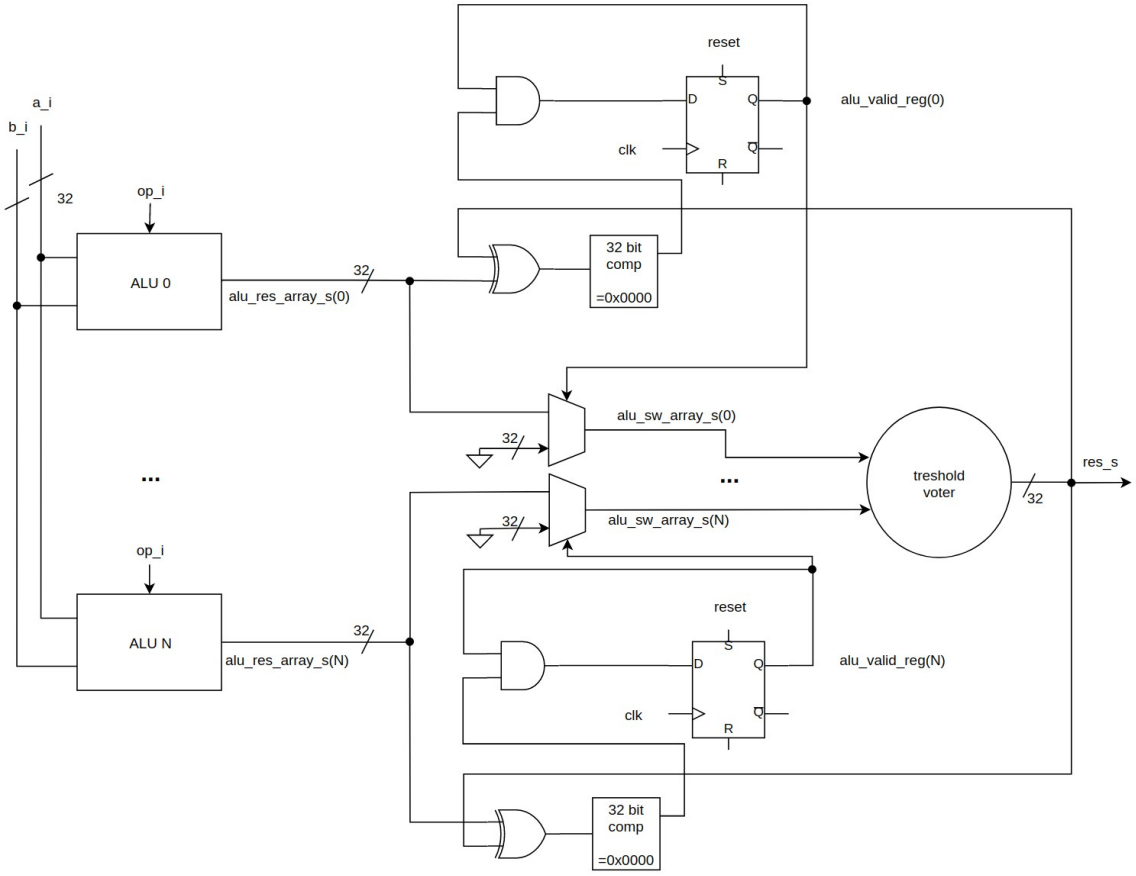
Listing 2. Registar koji čuva stanja ALU jedinica

Prekidačka logika je realizovana pomoću jedne for-loop naredbe koja iterira kroz ALU instance. Prva *if-else* naredba modeluje sam prekidač, koji u zavisnosti od vrednosti bita u registru koji odgovara toj ALU jedinici, ili prosleđuje njen rezultat ili ga postavlja na nulu. Druga *if-else* naredba kontroliše sledeće stanje registra. Prvo se proverava da li se rezultat ALU jedinice poklapa sa rezultatom koje daje glasač, što je najjednostavnije uraditi pomoću *xor* kola. Ukoliko se rezultat jednak onom koji daje glasač, ta ALU jedinica je i dalje validna pa njen bit u registru ostaje na jedinici. U suprotnom slučaju se taj bit resetuje na nulu.

```
switch_logic:
  process (alu_res_array_s,alu_valid_reg,voter_res_s)is
  begin
    for i in 0 to NUM_MODULES-1 loop

      if (alu_valid_reg(i) = '1')then
        alu_sw_array_s(i) <= alu_res_array_s(i);
      else
        alu_sw_array_s(i) <= (others => '0');
      end if;

      if((alu_res_array_s(i) xor voter_res_s) = std_logic_vector(to_unsigned(0,32))) then
        alu_valid_s(i) <= '1';
      else
        alu_valid_s(i) <= '0';
      end if;

    end loop;
  end process;
```

Listing 3. Prekidačka logika

Prethodno opisana prekidačka logika je prikazana na sledećem blok dijagramu.



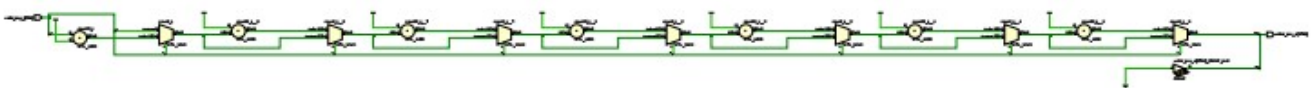
Slika 2. Uprošćena šema sistema

Kao što je već pomenuto, glasač prima *NUM_MODULES* 32-bitnih ulaza od ALU jedinica i drajvuje jedan 32-bitni izlaz koji predstavlja najzastupljenije rešenje. On dobija to rešenje na principu treshold kola. Svaki bit rešenja se dobija zasebno, naime za dobijanje vrednosti bita *res_o(0)*, glasač mora sabrati sve nulte bite ulaza (*alu_sw_array_s(i)(0)*) a zatim uporediti dobijeni broj sa granicom (*treshold*) koja je jednaka broju aktivnih ALU jedinica podeljn sa dva. Dakle, ukoliko više od pola aktivnih ALU jedinica daje logičku jedinicu za nulti bit, glasač postavlja nulti bit izlaza *res_s na* logičku jedinicu. Glasač stoga za određivanje treshold-a mora imati i pristup registru koji čuva validnost ALU jedinica, što je realizovano kao dodatan ulaz.

Kako bi se dobio broj jedinica u std_logic_vector-a u binarnom zapisu, što je neophodno za logiku glasača, u paketu *ft_pkg su* implementirane dve funkcije. Prva funkcija čiji je kod prikazan u kodnom listingu 4. prolazi sekvencijalno kroz std_logic_vector proizvoljne širine i za svaku logičku jedinicu inkrementira promenljivu *count*. Iako kod izgleda primamljivo, implementacija (prikazana na slici 3 za širinu vektora 8) nije optimalna. Propagaciono kašnjenje bi u ovom slučaju raslo linearno sa brojem modula. Zbog ovog je napravljena funkcija *count_ones_recursive*, prikazana u kodnom listingu 5. Ona rastavlja prosleđeni vektor na dva dela, a zatim poziva sebe nad dobijenim polovinama. Na ovaj način pravi strukturu binarnog stabla, čime se propagaciono kašnjenje uveliko smanjuje i raste brzinom $\log_2 n$ u odnosu na broj modula. Implementacija ove funkcije za širinu vektora 8 je prikazana na slici 4.

```
function count_ones_serial (vector : std_logic_vector) return std_logic_vector is
    constant RETURN_W : integer := (integer(ceil(log2(real(vector'length)))) + integer(1));
    variable count : std_logic_vector(RETURN_W-1 downto 0);
 begin

    count := (others => '0');

    for i in 0 to (vector'length-1) loop
      if(vector(i) = '1') then
        count := std_logic_vector(unsigned(count) + (to_unsigned(1,RETURN_W)));
      else
        count := std_logic_vector(unsigned(count) + (to_unsigned(0,RETURN_W)));
      end if;
    end loop;

    return count;
  end count_ones_serial;
```

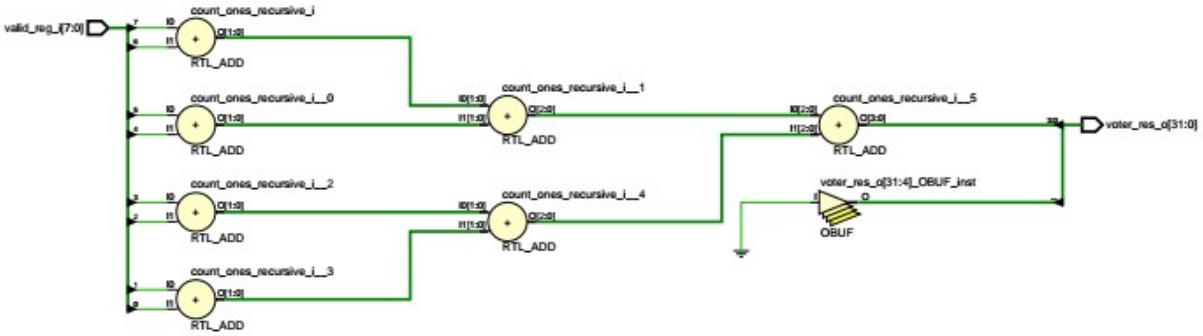
Listing 4. Kod funkcije count_ones_serial



Slika 3. Implementacija funkcije count_ones_serial

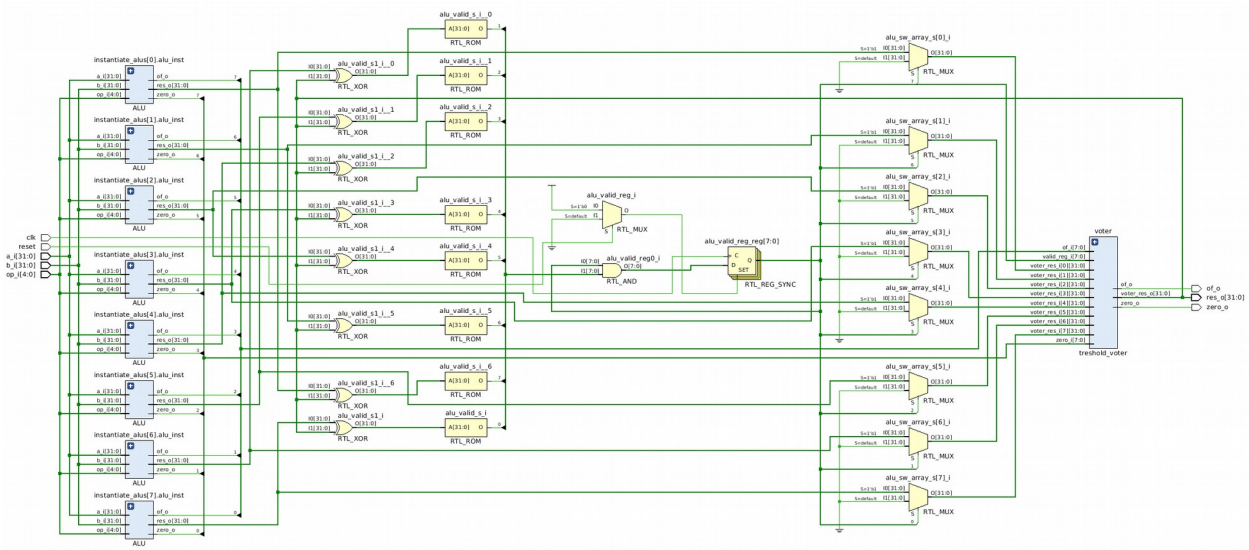
```
  function count_ones_recursive (vector : std_logic_vector) return std_logic_vector is
    constant RETURN_W : integer := (integer(ceil(log2(real(vector'length)))) + integer(1));
    variable count : std_logic_vector(RETURN_W-1 downto 0) := (others =>'0');
    variable res_upper,res_lower : unsigned(RETURN_W-1 downto 0) := (others =>'0');
    variable vector_upper : std_logic_vector(vector'length-vector'length/2-1 downto 0) := (others =>'0');
    variable vector_lower : std_logic_vector(vector'length/2-1 downto 0) := (others =>'0');
  begin
      if(vector'length = 1) then
        return std_logic_vector(to_unsigned(to_integer(unsigned(vector)),RETURN_W));
      else
        vector_upper := vector((vector'length-1) downto (vector'length/2));
        vector_lower := vector((vector'length/2-1) downto 0);

        res_upper := to_unsigned(to_integer(unsigned(count_ones_recursive(vector_upper))),
res_upper'length);
        res_lower := to_unsigned(to_integer(unsigned(count_ones_recursive(vector_lower))),
res_lower'length);

        count := std_logic_vector(res_upper + res_lower);

        return count;
      end if;
  end count_ones_recursive;
```

Listing 5. Kod funkcije count_ones_recursive



Slika 4. Implementacija funkcije count_ones_recursive

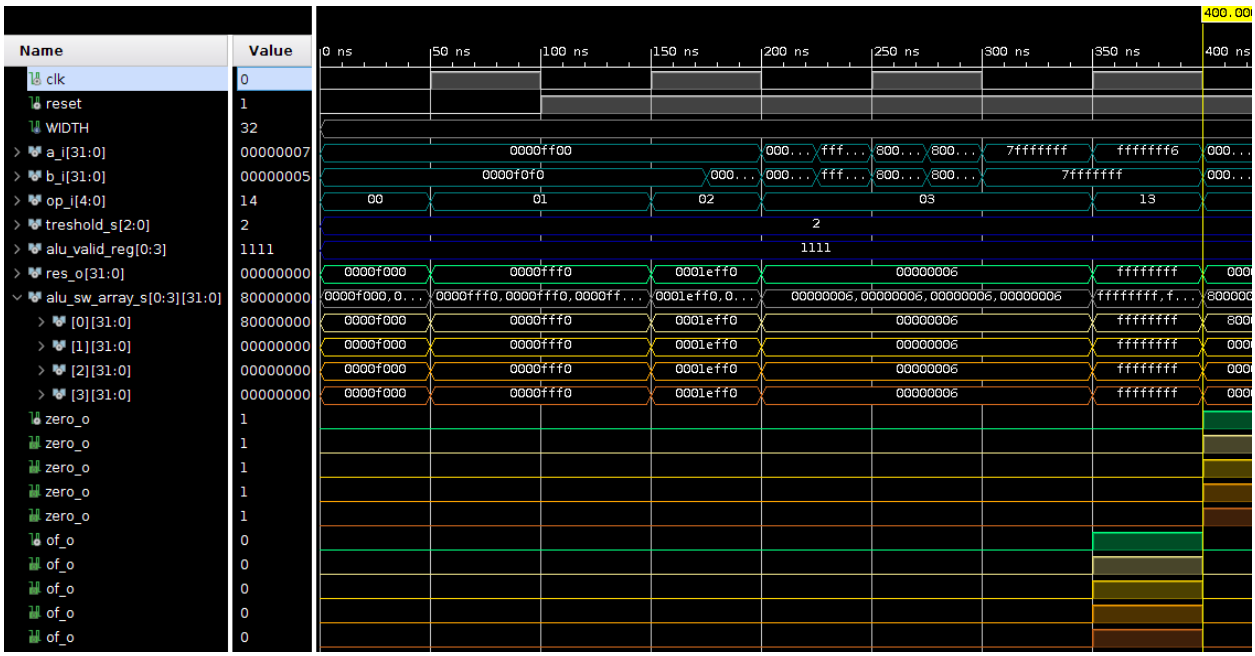
Budući da je glasač većinom repliciranje šematika sa prethodne slike 32 puta za svaki bit ponaosob, ovde neće biti prikazana detaljnija implementacija. Za 8 ALU jedinica je urađena sinteza kompletnog sistema što se može videti na slici 5. Može se primetiti da se šematik podudara sa slikom 2, s tim da je komparator u ovom slušaju modelovan pomoću ROM memorije.



Slika 5. Sinteza sistema za 8 ALU jedinica

4 Simulacija u Vivado okruženju

Kako bi se demonstrirao rad sistema, u test benču je instanciran sistem sa parametrom *NUM_MODULES* = 4. Na slikama 6a i 6b se mogu videti rezultati date simulacije. Signali *res_o, zero_o i of_o* čiji su signali prikazani zelenom bojom su izlazi sistema od 4 ALU jedinice. Na slici se takođe može primetiti niz od četiri 32-bitna signala *alu_sw_array_s* koji predstavlja izlaze ALU jedinica, kao i odgovarajući *zero_o* i *of_o* signali obojeni nijansama od žute boje. Inicijalno se rezultati ALU jedinica poklapaju pa ih i izlaz sistema *res_o* prati. Registar *alu_valid_reg* ima vrednost četiri jedinice što znači da su sve četiri ALU jedinice validne i da imaju pravo glasa. Signal *treshold_s* ima vrednost 2 što znači da je granica glasača 2, naime potrebno je da više od dve ALU jedinice daju isti rezultat kako bi on bio prihvaćen kao tačan.
Na slici 6a se vidi da sistem radi kao što je planirano do simulacionog vremena od 400 ns.

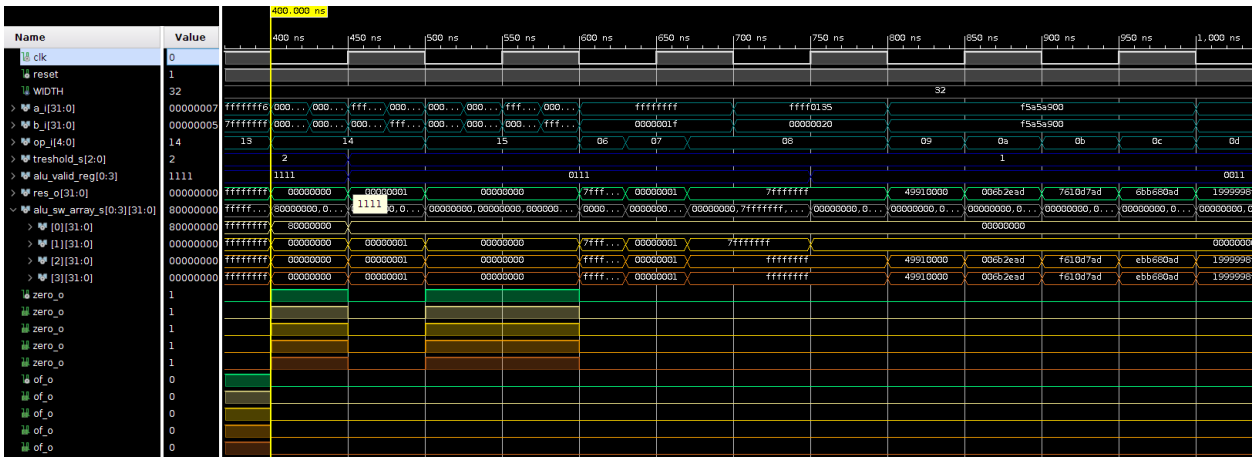


Slika 6a. Simulacija sa 4 ALU jedinice

Kako bi se demonstrirala prekidačka logika, u simulatoru su ubačene dve *stuck-at-fault* greške:

1) u 400ns 31. bit rezultata ALU jedinice 0 se zaustavlja na '1' do kraja simulacije
2) u 600ns 31. bit rezultata ALU jedinice 1 se zaustavlja na '0' do kraja simulacije

Kao što se može primetiti sa slike 6b. Otkaz ALU jedinice 0 se dešava u 400 ns i tada ona daje rešenje 0x800000000 do ostale daju rezultat 0x00000000. Na sledeću rastuću ivicu taktnog signala (450 ns), rezultat nulte ALU jedinice pada na 0 i njen bit validnosti u registru *alu_valid_reg* takođe pada na '0' što znači da od tog trenutka njen rezultat više neće biti uziman za računanje rezultata. Može se primetiti da treshold_s signal pada na 1, zato što su sada u sistemu ostale samo tri validne jedinice. Naime, od ovog trenutka je potrebno je da dve od tri ALU jedinice daju isti rezultat kako bi on bio prihvaćen kao konačni. U 600 ns se ponovo vidi neslaganje između ALU jedinice 1 i preostale dve, pa se na sledeću rastuću ivicu takta (650 ns) rezultat iste postavlja na 0 i vrednost registra *alu_valid_reg* se ažurira.

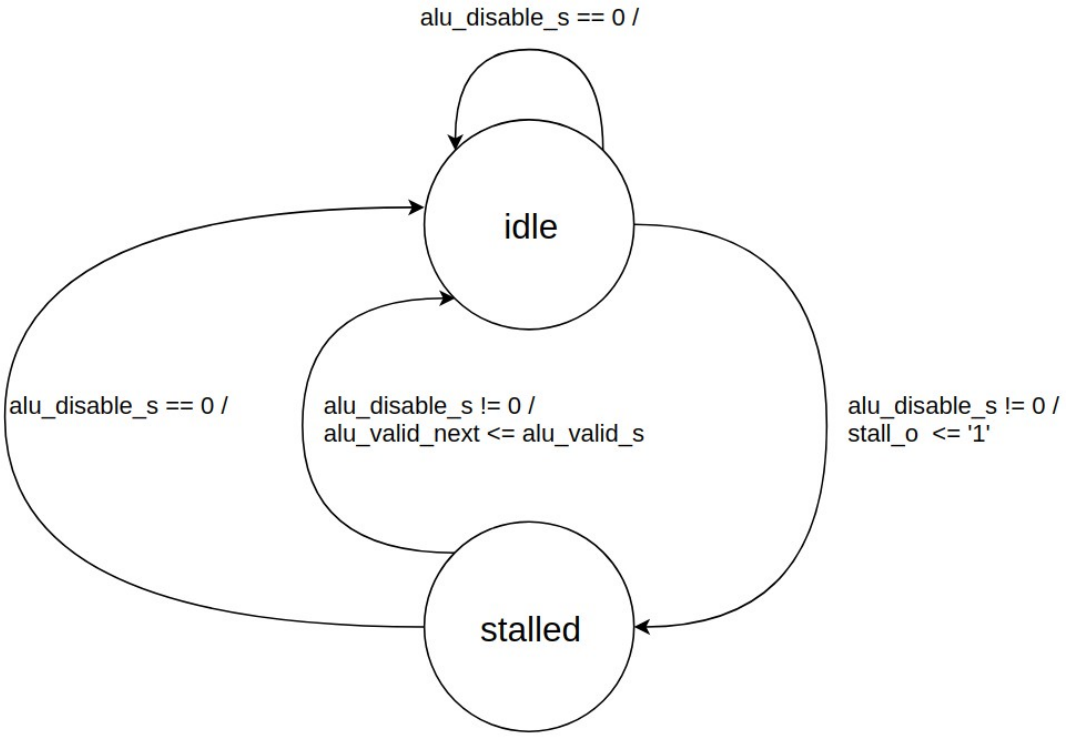


Slika 6b. Simulacija sa 4 ALU jedinice

5 Otpornost na tranzijentne greške

Prethodno opisani sistem bi i kratkotrajne (tranzijentne) greške smatrao hardverskim kvarovima, pa bi u slučaju promene rezultata jedne od ALU jedinica u toku rastuće ivice takta istu bespovratno isključio. U digitalnim sistemim se ovakve suvišne akcije mogu izbeći dodavanjem *"reissue"* signala, koji pri detkciji neslaganja medju modulima govori kontrolnoj jedinici da ponavi poslednju operaciju kako bi se potvrdilo da greška nije tranzijentna. U slučaju pajplajnovanog procesora, potrebno je da jedinica za razrešavnje hazarda zaustavi (stall-uje): PC (program counter), IF/ID registar (instruction fetch / instruction decode) i ID-EX registar (instruction decode - execute). Stoga u naš sistem dodajemo izlazni signal *stall_o* koji se prosleđuje direktno hazard jedinici. Potrebno je i modelovati jednostavan konačni automat koji generiše stall_o signal, i nadgleda da li je neka od ALU jedinica pred isključenje.

Dijagram automata je prikazan na slici 7. alu_disable_s je samo pomoćni signal koji je na nuli dok se rešenja aktivnih ALU jedinica podudaraju. Kada se desi greška na nekoj od aktivnih jedinica odgovarajući bit ovog signala će biti na jedinici, samim tim ovaj signal govori koja od ALU jedinica je pred isključenje. VHDL kod automata i pomoćnog signala *alu_disable_s* je dat u kodnom listingu 6.



Slika 7. Dijagram konačnog automata

```
fsm_reg:
process (clk,reset) is
begin
  if(rising_edge(clk))then
    if(reset='0')then
      state_reg <= idle;
    else
      state_reg <= state_next;
    end if;
  end if;
end process;

alu_disable_s <= (alu_valid_s nor (not alu_valid_reg));

fsm_comb:
process (alu_disable_s, state_reg) is
begin
  stall_o <= '0';
  alu_valid_next <= (others => '1');
  state_next <= idle;
  case state_reg is
    when idle =>
      if (alu_disable_s /= std_logic_vector(to_unsigned(0,NUM_MODULES))) then
        state_next <= stalled;
        stall_o <= '1';
      end if;
    when stalled =>
      state_next <= idle;
      if (alu_disable_s /= std_logic_vector(to_unsigned(0,NUM_MODULES))) then
        alu_valid_next <= alu_valid_s;
      end if;
    when others =>
  end case;
end process;
```

Listing 6. Automat koji rešava tranzijentne greške

Nakon ovih izmena blok dijagram sistema izgleda kao na slici 9.

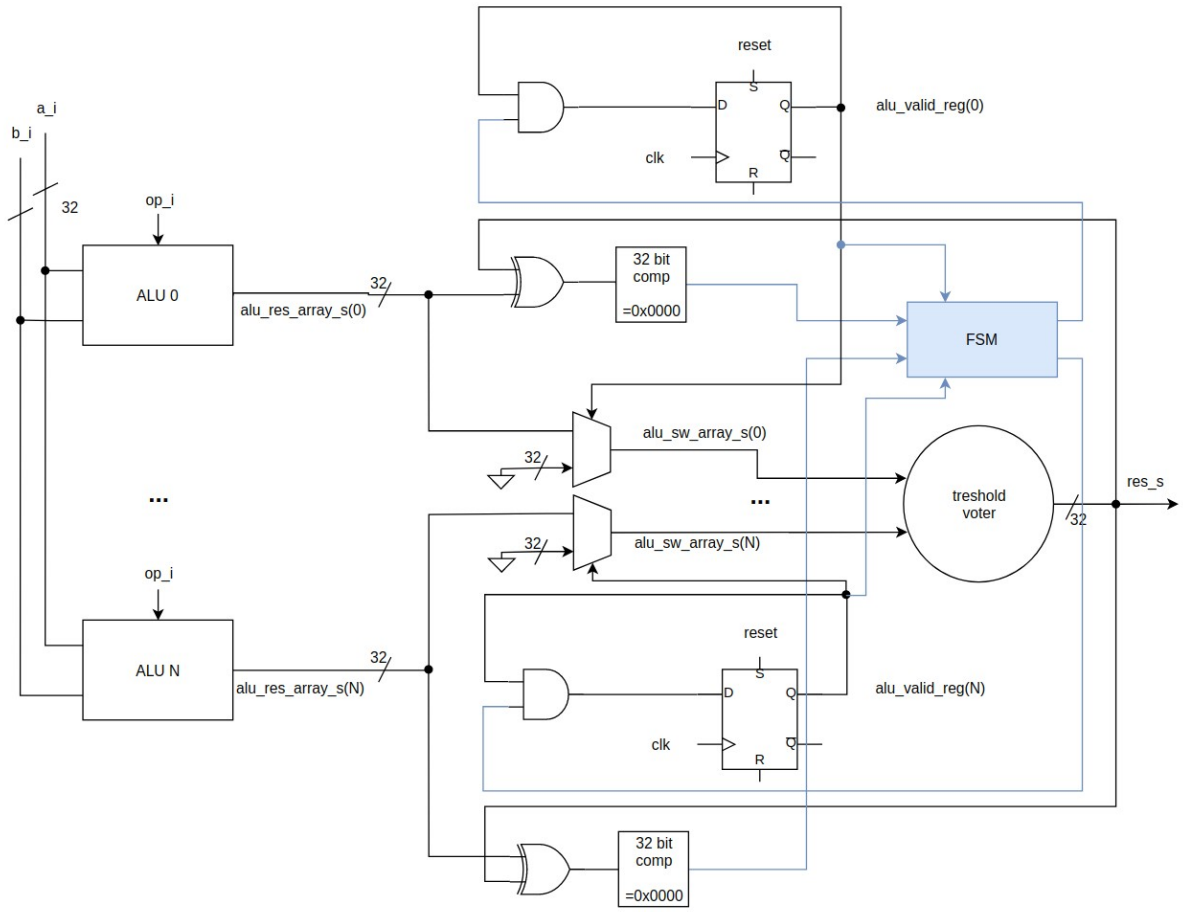
Kako bi se potvrdila funkcionalnost sistema, modul *ALU_ft* je instanciran u procesoru sa 5 pajplajn faza i setom instrukcija RV32I. Kao i u prethodnoj simulaciji, unete su dve *stuck-at-fault* greške:

1) tranzijentna (15.300ns - 15.500ns) ALU jedinica 0, sve linije *stuck-at-0*

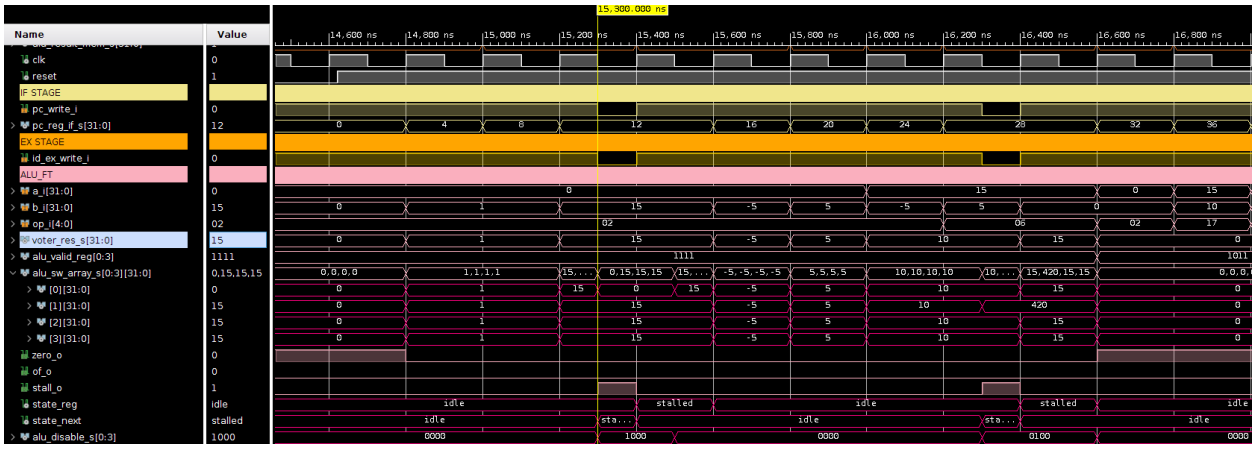
2) stalna (16.300ns - kraj simulacije) ALU jedinica 1, rezultat zaustavljen na 420

Kao što se može primetiti na slici 10, u 15.300ns se primećuje odstupanje ALU jedinice 0 i *stall_o* signal se postavlja na '1' čime se zaustavljaju sve pajplajn faze pre ALU jedinice. Iako se desi rastuća ivica takta u 15.400ns, procesor ponovo izvršava istu naredbu i čeka pre isključenja iste. Budući da se greška razreši pre sledeće rastuće ivice takta u 15.600ns, procesor nastavlja sa radom bez eliminisanja ALU jedinice 0.

Kada se desi sledeća greška u 16.300ns, procesor i u ovom slučaju zaustavlja pajplajn faze (16.400ns). Instrukcija se ponovo izvršava, ali budući da se greška ne razreši do sledeće rastuće ivice takta (16.500), smatra se da je ALU jedinica 1 otkazala. Ovo se može videti na signalu *alu_valid_reg* i postavljanju izlaza ove ALU jedinice na nulu.



Slika 9. Blok dijagram sa konačnim automatom

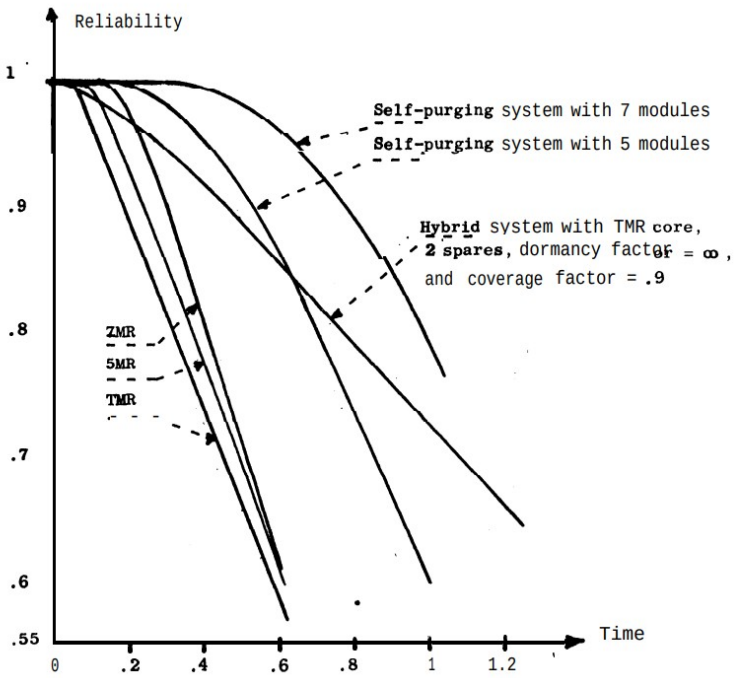


Slika 10. Simulacija procesora

6 Zaključak

Cilj ovog rada je bio da se pokaže jednostavnost prekidačke logike kod s*elf-purging* hibridne hardverske redundanse što omogućava laku implementaciju u sistemima uz minimalno zauzimanje FPGA resursa. Jednostavnost prekidača je ključna zato što ne predstavlja usko grlo kod pouzdanosti kompletnog sistema što se često dešava kod hibridnih i *stand-by* sistema.

Mogućnost maskiranja *n-2* greške uz detekciju *n-1* greške i uz dodatnu eliminaciju kratkotrajnih tranzijentnih grešaka, čine ovakav sistem pogodnim za različite primene u kojima je sigurnost od primarnog značaja a paralelan rad *n* modula ne narušava energetska ograničenja. Kada su vremena misije istog reda kao životni vek *simplex* sistema, ili kada je verovatnoća otkaza isključenog modula ista kao i za uključene, s*elf-purging* sistem je optimalan. Sva vremena misije duža od toga mogu bolje iskoristiti sisteme sa isključenim *stand-by* modulima (videti sliku 11). Još jedna prednost s*elf-purging* sistema je što se njena pouzdanost može tačno odrediti a ne samo aproksimirati što je slučaj kod većine drugih hibridnih i *stand-by* sistema.



Slika 11. Pouzdanost tokom vremena

Radili:
Đorđe Mišeljić E1 5/2018
Nikola Kovačević E1 4/2018

Literatura:

"A Highly Efficient Redundancy Scheme: Self-Purging Redundancy" - Jaques Losq, 1975. *Digital Systems Laboratory - Stanford Electronics Laboratories*
http://i.stanford.edu/pub/cstr/reports/csl/tr/73/62/CSL-TR-73-62.pdf

Potpun kod se može naći na sledećem repozitorijumu i grani *fault_tolerance*:
https://github.com/DjordjeMiseljic/RISC_VHDL.git