Funkcionalna verifikacija hardvera

Vežba 2
Objektno orijentisani aspekti SystemVerilog jezika

Sadržaj

Ova vežba se bavi nizovima u SystemVerilog jeziku, načinima kastovanja kao i objektno-orijentisanim aspektima jezika. Objašnjeni su osnovni pojmovi u OOP terminologiji, konvencije pri korišćenju i dati su primeri upotrebe. Ukratko su opisane interfejs i *package* konstrukcije.

1 Nizovi

U ovom poglavlju su objašnjeni nizovi u SystemVerilog jeziku. Dat je pregled osnovnih vrsta i načina korišćenja, objašnjena razlika između *packed* i *unpacked* nizova i objašnjen način upotrebe ugrađenih metoda. Dat je i pregled redova (engl. *queue*) u SystemVerilog-u.

1.1 Nizovi fiksne veličine

Nizovi fiksne veličine se deklarišu navođenjem gornjeg i donjeg limita (po uzoru na Verilog) ili navođenjem veličine (po uzoru na C). Sledeće dve deklaracije su ekvivalentne i predstavljaju niz od 8 int promenljivih.

```
int example [0 : 7];
int example2 [8];
```

Multidimenzioni nizovi se isto deklarišu navođenjem dimenzija nakon imena niza, npr:

```
int example3 [8] [15];
```

Niz se može inicijalizovati koristeći sintaksu ispod:

```
bit example4 [7:0] = '{0,0,0,0,0,0,0,1};
```

Čest način manipulisanja listama je *for*, odnosno *foreach* petlja. Prilikom iteracije kroz niz koristeći *for* petlju, pogodna je funkcija $*size()* koja vraća veličinu niza. *Foreach* petlja, sa druge strane, ovo radi automatski. Prilikom korišćenja *foreach* petlje navede se niz i index, i zatim se vrši iteracija kroz sve elemente niza. Npr. sledeći primeri dupliraju vrednost svih članova niza:

```
for(int i = 0; i < $size(example); i++) begin
  example[i] *= 2;
end

foreach(example[i])
  example[i] *= 2;
```

Foreach petlja za multidimenzionu listu ima malo drugačiju sintaksu. Umesto navođenja dimenzija u odvojenim zagradama, dimenzije se navode u istoj uglastoj zagradi odvojene zarezima. Npr:

```
foreach (example3 [i, j]) example3[i][j] *= 2;
```

U nastavku je dato još par primera upotrebe nizova kako bi se bolje upoznali sa sintaksom.

```
module fixed_size_arrays;

   int x[4] = '{1, 2, 3, 4};
   int y[4] = '{5, 6, 7, 8};

    initial begin
       if(x == y)
         $display("Nizovi su jednaki");
       else
         $display("Nizovi nisu jednaki");
       y = x;
       // y poprima vrednosti x niza
       y[0] = 0; // promena prvog elementa
       // poredjenje dela niza
       if(x[1:2] == y[1:2])
         $display("Elementi 1 i 2 su jednaki");
       else
         $display("Elementi 1 i 2 nisu jednaki");
   end

   // Rezultat izvrsavanja:
```

```
  //
  // Nizovi nisu jednaki
  // Elementi 1 i 2 su jednaki

endmodule : fixed_size_arrays
```

Kod 1: nizovi fiksne velicine

1.2 *Packed* i *unpacked* nizovi

Ako se prisetimo prve vežbe i deklaracije željene veličine promenljivih, možemo uočiti razliku između deklaracije promenljive željene veličine sa nizom fiksne veličine, objašnjenim u prethodnom poglavlju. Zapravo obe ove deklaracije predstavljaju niz podataka, ali dve različite vrste – *packed* i *unpacked*. Na primeru ispod možemo videti razliku u sintaksi.

```
bit [7 : 0] x; // packed niz
bit x [8]; // unpacked niz
```

Za neke tipove podataka je pogodno pristupati celom podatku, ali i podeliti ga u manje elemente kojima se može pojedinačno pristupati (npr. pristupati bajtu oređnom ili svakom bitu pojedinačno). *Packed* niz ovo omogućava. On se može tretirati i kao niz i kao jedna vrednost. Za razliku od *unpacked* niza, dimenzije packed niza se navode pre imena, kao deo tipa promenljive. Takođe, obavezno je navesti dimenzije u formatu [MSB : LSB] (nije moguće navesti samo veličinu). *Packed* nizovima se može pristupati kao običnoj promenljivoj. Može im se dodeljivati vrednost, koristiti se u raznim izrazima i sl., ali se može pristupati i pojedinačnim elementima, npr:

```
logic [3 : 0] x, y = 0;
x = 4'hA;
y = x + 4'h8;
x[2] = y[0];
y[3 : 1] = 3'b010;
```

Glavna razlika između *packed* i *unpacked* nizova je u načinu čuvanja vrednosti u memoriji. *Packed* niz se čuva u kontinualnom nizu bita, bez praznih mesta, što ne mora biti slučaj sa *unpacked* nizom. Sledeće tabele ilustruju način smeštanja nizova u memoriji, ali i mogućnost kombinovanja *packed* i *unpacked* dimenzija:

bit [3:0] [7:0] packed_example

							7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
							packed_example[3]								packed_example[2]								packed_example[1]								packed_example[0]							

bit [7:0] unpacked_example [4]

		7	6	5	4	3	2	1	0	unpacked_example[0]
		7	6	5	4	3	2	1	0	unpacked_example[1]
		7	6	5	4	3	2	1	0	unpacked_example[2]
		7	6	5	4	3	2	1	0	unpacked_example[3]

bit [2:0] [7:0] mix_example [4]
dat je i primer pristupanja određenim elementima

mix_example[0][2][5]

7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	mix_example[0]
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	mix_example[1]
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	mix_example[2]
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	mix_example[3]

mix_example[3][1]

1.3 Dinamički nizovi

Dinamički niz je jednodimenzioni, *unpacked* niz čija se veličina može menjati tokom simulacije, odnosno ne mora biti poznata u vreme kompajliranja. Deklariše se korišćenjem praznih uglastih

zagrada []. Dinamički niz ne zauzima prostor u memoriji do god se on eksplicitno ne kreira korišćenjem metode *new*[<veličina>]. Pored ove metode, često se koriste i metode *size()* koja vraća trenutnu veličinu niza, i metoda *delete()* koja briše niz (rezultat je prazan niz). U nastavku je dat primer korišćenja dinamičkih nizova.

```
module dyn_array;

  // deklaracija
  int dyn_example[];

  initial begin

    // alokacija memorije
    dyn_example = new[3]; // 3 elementa

    //  inicijalizacija
    foreach (dyn_example[i])
      dyn_example[i] = i;
    $display("dyn_example = %p", dyn_example);

    // dodavanje elemenata
    // 5 elemenata (3 postojeca + 2 nova)
    dyn_example = new[5](dyn_example);
    $display("dyn_example = %p", dyn_example);

    // 100 elemenata (prethodne vrednosti su izgubljene)
    dyn_example = new[100];
    $display("dyn_example = %p", dyn_example);

    // brisanje niza
    dyn_example.delete();
    $display("dyn_example = %p", dyn_example);
  end

  // Rezultat izvrsavanja:
  //
  // dyn_example = '{0, 1, 2}
  // dyn_example = '{0, 1, 2, 0, 0}
  // dyn_example = '{0, 0, ..., 0}
  // dyn_example = '{}

endmodule : dyn_array
```

Kod 2: Dinamicki nizovi

1.4 Asocijativni nizovi

Ukoliko je veličina niza nepoznata ili je redak prostor podataka (*sparse data space*), dobro rešenje pružaju asocijativni nizovi. Ovi nizovi čuvaju elemente u *sparse* matrici i zauzimaju memoriju tek kada se koristi element. Takođe, indeks niza ne mora biti celobrojni, već može biti bilo kog tipa. Asocijativni niz implementira *lookup* tabelu željenog tipa, gde indeks predstavlja *lookup* ključ. Primer upotrebe asocijativih nizova bila bi, npr., verifikacija memorije velikog opsega. Tokom simulacije se možda vrši pristup na samo nekoliko adresa i u asocijativnom nizu se čuvaju vrednosti tih lokacija, umesto čitave memorije što znatno štedi na utrošenom memorijskom prostoru.

Sintaksa deklaracije asocijativnih nizova je:

```
<tip_podatka> <id_niza> [<tip_indeksa>];
```

gde <tip_podatka>predstavlja tip elemenata niza, a za <tip_indeksa>se navodi ili tip podataka kome pripadaju indeksi niza ili "*", što označava bilo koji celobrojni izraz proizvoljne veličine. Npr.

```
int example_array1 [*]; // niz int-ova
```

```
// niz 8-bitnih vektora ciji
// su indeksi string-ovi
bit [7 : 0] example_array2 [string];

// niz logic elemenata koji
//su indeksirani sa example_class klasom
logic example_array3 [example_class];
```

SystemVerilog pruža veliki broj metoda za rad sa asocijativnim nizovima. One su:

- *num()* – vraća broj elemenata u nizu
- *delete(index)* – briše element sa specificiran datim indeksom
- *exists(index)* – vraća 1 ukoliko postoji element na datom indeksu, 0 u suprotnom
- *first(var)* – pridružuje vrednost prvog indeksa promenljivoj *var* i vraća 0 ukoliko je niz prazan, 1 u suprotnom
- *last(var)* – pridružuje vrednost poslednjeg indeksa promenljivoj *var* i vraća 0 ukoliko je niz prazan, 1 u suprotnom
- *next(var)* – ukoliko postoji, pridružuje vrednost sledećeg indeksa promenljivoj *var* i vraća 1, u suprotnom *var* ostaje nepromenjena i vraća se 0
- *prev(var)* – ukoliko postoji, pridružuje vrednost prethodnog indeksa promenljivoj *var* i vraća 1, u suprotnom *var* ostaje nepromenjena i vraća se 0

U nastavku je dat primer upotrebe asocijativnih nizova. Uočiti zašto se iteracija kroz sve elemente niza vrši sa *foreach*, a ne *for* petljom. Alternativno, moguće je iterirati kroz sve elemente koristeći metode *next()* ili *prev()* u petlji.

```
module assoc_array;

  int example_array[int];
  int idx;

  initial begin
    // dodeli vrednosti pojedinim elementima
    idx = 2;
    repeat(5) begin
      example_array[idx] = idx;
      idx *= 5;
    end
    $display("example_array = %p", example_array);

    // prodi kroz sve elemente
    foreach (example_array[i]) begin
      $display(example_array[i]);
    end

    // primeri upotrebe metoda
    if(example_array.exists(3)) begin
      // telo se nece izvrsiti
      // jer ne postoji element sa indeksom 3
      $display("Nece se ispisati .");
    end
    if(example_array.first(idx)) begin
      // telo ce se izvrsiti , a idx dobija vrednost 2
      $display("Primer metode first: idx = %0d", idx);
    end
  end

  // Rezultat izvrsavanja:
```

```
  //
  // example_array = '{2:2, 10:10, 50:50, 250:250, 1250:1250 }
  //              2
  //             10
  //             50
  //            250
  //           1250
  // Primer metode first: idx = 2

endmodule : assoc_array
```

Kod 3: Asocijativni nizovi

1.5 Redovi

Redovi (engl. *queues*) u SystemVerilog-u predstavljaju kombinaciju dinamičkih nizova i povezanih lista (engl. *linked lists*). Kao i kod dinamičkih nizova, bilo kom elementu se može pristupati preko indeksa (bez troškova prolaska kroz sve prethodne elemente kao kod liste), ali se i, kao kod povezanih lista mogu dodavati ili izbacivati elementi bilo gde u redu (bez troškova kopiranja i kreiranja nove strukture kao kod dinamičkog niza). Redovi se deklarišu kao *unpacked* nizovi, ali se navodi znak "$" kao veličina, odnosno:

```
<data_type> <queue_name> [$];
```

Što se tiče indeksiranja, "0" predstavlja prvi element, a "$" poslednji. Vitičaste zagrade se koriste za inicijalizaciju redova, ali i za dodavanje ili brisanje elemenata. Operatori sa rad sa redovima su slični operatorima *unpacked* nizova.

Postoje i ugrađene metode za rad sa redovima:

- *size()* – vraća broj elemenata u redu
- *insert(indeks, vrednost)* - ubacuje element na mesto sa datim indeksom
- *delete(indeks)* - briše element sa datim indeksom. Ako indeks nije zadat, briše se ceo red
- *push_front(vrednost)* - ubacuje element na početak reda
- *push_back(vrednost)* - ubacuje element na kraj reda
- *pop_front()* - briše i vraća element sa početka reda
- *pop_back()* - briše i vraća element sa kraja reda

U nastavku su dati primeri korićenja redova i izvršeno je poređenje upotrebe operatora sa upotrebom metoda. Naredni kodni fragmenti su su ekvivalentni.

```
module queue_operators;

  int x = 2;

  // deklaracija
  int q1[$] = {2, 4, 8}; // i  inicijalizacija

  initial begin
    // ubacivanje elemenata

    q1 = {q1, 10}; // na kraj reda
    q1 = {3, q1};  // na pocetak reda
    q1 = {q1[0 : x-1], 5, q1[x : $]}; // ubacuje 5 poziciju x
    $display("q1 = %p", q1);

    // brisanje elemenata
```

```
    x = q1[0]; // x dobija vrednost prvog el.
    q1 = q1[1 : $]; // brise prvi element
    $display("q1 = %p", q1);

    // x dobija vrednost poslednjeg elementa
    x = q1[$];
    // brise poslednji element
    q1 = q1[0:$-1];
    $display("q1 = %p", q1);

    q1 = {}; // brise ceo red
    $display("q1 = %p", q1);
  end

  // Rezultat izvrsavanja:
  //
  // q1 = '{3, 2, 5, 4, 8, 10}
  // q1 = '{2, 5, 4, 8, 10}
  // q1 = '{2, 5, 4, 8}
  // q1 = '{}

endmodule : queue_operators
```

Kod 4: Redovi - operatori

```
module queue_methods;

  int x = 2;

  // deklaracija
  int q2[$] = {2, 4, 8}; // i  inicijalizacija

  initial begin
    // ubacivanje elemenata

    q2.push_back(10); // na kraj reda
    q2.push_front(3); // na pocetak reda
    q2.insert(x, 5); // ubacuje 5 poziciju x
    $display("q2 = %p", q2);

    // brisanje elemenata

    // prvi element se uklanja iz reda i
    // x dobija njegovu vrednost
    x = q2.pop_front();
    $display("q2 = %p", q2);

    // poslednji element se uklanja iz reda i
    // x dobija njegovu vrednost
    x = q2.pop_back();
    $display("q2 = %p", q2);

    q2.delete(); // brise ceo red
    $display("q2 = %p", q2);
  end

  // Rezultat izvrsavanja:
  //
  // q2 = '{3, 2, 5, 4, 8, 10}
  // q2 = '{2, 5, 4, 8, 10}
  // q2 = '{2, 5, 4, 8}
  // q2 = '{}

endmodule : queue_methods
```

Kod 5: Redovi - metode

Treba napomenuti da iako postoje povezane liste u SystemVerilog-u, njihova upotreba se ne preporučuje zato što su redovi mnogo efikasniji i lakši za korišćenje, pa se ovde neće detaljnije obrađivati.

1.6 Metode

U SystemVerilog-u postoji mnoštvo metoda za rad sa nizovima i redovima. Ove metode se mogu koristiti na bilo kom *unpacked* nizu, odnosno na nizovima fiksne veličine, dinamičkim i asocijativnim nizovima i redovima. Sintaksa je:

```
array_method_call ::=
    expression.array_method_name { attribute_instance }
      | [ ( list_of_arguments ) ] [ with ( expression ) ]
```

with klauza je opciona i prihvata izraz u zagradama.

U nastavku je dat pregled ovih metoda.

- Metode lokacije (povratna vrednost je red):
  - *find()* - vraća sve elemente koji zadovoljavaju dati izraz
  - *find_ index()* - vraća indekse svih elemenata koji zadovoljavaju dati izraz
  - *find_ first()* - vraća prvi element koji zadovoljava dati izraz
  - *find_ first_ index()* - vraća indeks prvog elementa koji zadovoljava dati izraz
  - *find_ last()* - vraća poslednji element koji zadovoljava dati izraz
  - *find_ last_ index()* - vraća indeks poslednjeg elementa koji zadovoljava dati izraz
  - *min()* - vraća element sa minimalnom vredošću
  - *max()* - vraća element sa maksimalnom vrednošću
  - *unique()* - vraća sve elemente čija je vrednost jedinstvena
  - *unique_ index()* - vraća indekse svih elemenata čija je vrednost jedinstvena
- Metode uređivanja:
  - *reverse()* - preokreće sve elemente
  - *sort()* - sortira elemente u rastućem redosledu
  - *rsort()* - sortira elemente u opadajućem redosledu
  - *shuffle()* - randomizuje redosled elemenata
- Metode redukcije:
  - *sum()* - vraća sumu svih elemenata
  - *product()* - vraća proizvod svih elemenata
  - *and()* - vraća *bitwise and* od svih elemenata
  - *or()* - vraća *bitwise or* od svih elemenata
  - *xor()* - vraća *bitwise xor* od svih elemenata

Za pojedine metode je *with* klauza opciona, za pojedine obavezna, a za neke nedozvoljena. Sve *find* metode moraju sadržati i *with* klauzu, dok je za *min*, *max* i *unique* metode ona opciona. Za *sort*, *rsort* i sve metode redukcije je takođe opciona. Ali je za metode *reverse* i *shuffle* nedozvoljena. U nastavku je dato nekoliko primera:

```
module array_methods;

  int example[$] = {1, 15, 6, 3};
  int res [$];
  int x;

  initial begin
    // nadji sve elemente manje od 10
    res = example.find(x) with (x < 10); // res dobija 1, 6, 3
    $display("Elementi manji od 10 su: %p", res);

    // nadji najmanji element
    res = example.min; // res dobija 1
    $display("Najmanji element je: %p", res);

    // sortiranje niza
    example.sort; // example postaje {1, 3, 6, 15}
    $display("Sortiran niz: %p", example);

    // sumiranje
    x = example.sum; // x = 1 + 3 + 6 + 15
    $display("Suma elemenata niza: %0d", x);

    // xor - ovanje
    x = example.xor with (item + 2); // x = 3 ^ 5 ^ 8 ^ 17
    $display("Xor elemenata niza: %0d", x);
  end

  // Rezultat izvrsavanja:
  //
  // Elementi manji od 10 su: '{1, 6, 3}
  // Najmanji element je: '{1}
  // Sortiran niz: '{1, 3, 6, 15}
  // Suma elemenata niza: 25
  // Xor elemenata niza: 31

endmodule : array_methods
```

Kod 6: Primer upotrebe metoda

1.7 Zadaci

Zadatak Kod 7 prikazuje upotrebu redova u SystemVerilog-u (fajl "queue_examples.sv" u pratećim materijalima). Proučiti način upotrebe redova, korišćenje operatora i metoda. Koji će biti rezultat izvršavanja datog koda?

```
module queue_examples;

  int x = 2;
  int y = 2;

  int q1[$] = {2, 4, 8};
  int q2[$] = {2, 4, 8};

  initial begin

    $display("Inicijalno stanje reda q1: %p", q1);

    q1 = {q1, 10};
    q1 = {3, q1};
    q1 = {q1[0 : x-1], 5, q1[x : $]};
    $display("Posle ubacivanja elemenata u q1: %p", q1);

    x = q1[0];
    $display("x dobija vrednost %0d", x);
    q1 = q1[1 : $];
```

```
        $display("Posle brisanja elemenata iz q1: %p", q1);

        x = q1[$];
        $display("x dobija vrednost %0d", x);
        q1 = q1[0:$-1];
        $display("Posle brisanja elemenata iz q1: %p", q1);

        q1 = {};
        $display("Posle brisanja celog reda q1: %p", q1);

        // ------------------------------------------------

        $display(" Inicijalno  stanje reda q2: %p", q2);

        q2.push_back(10);
        q2.push_front(3);
        q2.insert(y, 5);
        $display("Posle ubacivanja elemenata u q2: %p", q2);

        y = q2.pop_front();
        $display("y dobija vrednost %0d", y);
        $display("Posle brisanja elemenata iz q2: %p", q2);

        y = q2.pop_back();
        $display("y dobija vrednost %0d", y);
        $display("Posle brisanja elemenata iz q2: %p", q2);

        q2.delete();
        $display("Posle brisanja celog reda q2: %p", q2);
    end

endmodule : queue_examples
```

Kod 7: Redovi - zadatak

2 Kastovanje

Zbog velikog broja tipova podataka, ponekad je potrebno dodeliti promenljivu jednog tipa promenljivoj drugog tipa odnosno uraditi *cast*. Postoje dve vrste *cast*-ovanja: statički i dinamički.

2.1 Statički kast

Tip podatka se može promeniti korišćenjem *cast* (“'”) operacije. Potrebno je navesti željeni tip i izraz koji želimo da promenimo. Npr.

```
int'(2.5 + 2.3); // realni broj postaje int
```

2.2 Dinamički kast

SystemVerilog sadrži sistemski task $*cast* koji se koristi za dinamičko *cast*-ovanje. Dinamički *cast* omogućava proveru *out-of-bounds* vrednosti. $*cast* se može pozvati na dva načina: kao funkcija ili kao task. Sintaksa poziva je:

```
function int $cast( singular dest_var, singular source_exp );
task $cast( singular dest_var, singular source_exp );
```

Razlika je u reakciji na pogrešne dodele. Ako se poziva kao task, $*cast* pokušava da dodeli *source* izraz *destination* promenljivoj i ukoliko je dodela neuspešna javlja se *runtime error*, a promenljiva ostaje nepromenjena. Ako se poziva kao funkcija, $*cast* će u neuspelom slučaju vratiti 0 da signalizira grešku, odnosno vratiće 1 pri uspešnoj operaciji.

Bitno je uočiti razliku između statičnog i dinamičkog *cast*-a. $*cast* vrši proveru u vreme izvršavanja (engl. *at runtime*). Provera tipa se ne vrši od strane kompajlera. Što nije slučaj kod statičnog *cast*-a. Razlike i način odabira odgovarajuće vrste se najbolje uočava na primeru koristeći *enum*:

```
module cast;

  typedef enum { red, green, blue, yellow, white, black } Colors;
  Colors col;

  initial begin
    // dynamic
    if (!$cast(col, 2 + 3)) // col = black
      $error("Invalid cast");
    $display("Vrednost col = %s", col.name);

    if (!$cast(col, 2 + 8)) // 10: invalid cast
      $error("Invalid cast");
    $display("Vrednost col = %s", col.name); // vrednost ostaje nepromenjena

    // static
    // compile-time cast
    // uvek je uspesan u toku izvrsavanja i nema mogucnost
    // provere greske ukoliko zadata vrednost lezi van
    // opsega enum-a
    col = Colors'(2 + 1);
    $display("Vrednost col = %s", col.name);

    col = Colors'(2 + 8);
    $display("Vrednost col = %s", col.name);
  end

  // Rezultat izvrsavanja:
  //
  // Vrednost col = black
  // ** Error: Invalid cast
  //    Time: 0 ps Scope: cast File: v2_cast.sv Line: 13
```

```
    // Vrednost col = black
    // Vrednost col = yellow
    // Vrednost col =

endmodule : cast
```

Kod 8: Kastovanje

3 Objektno orijentisano programiranje

Objektno orijentisano programiranje omogućava kreiranje kompleksnih tipova podataka i grupisanje istih sa metodama. U pogledu verifikacije, olakšava kreiranje testbenčeva i modela na višem nivou apstrakcije čime se postiže veća produktivnost, lakše održavanje koda i laka ponovna upotreba. Objekto orijentisani aspekti SysemVerilog-a veoma liče na osobine C++ jezika. U ovom poglavlju je dat pregled osnovnih pojmova i dati primeri upotreba klasa u SystemVerilog jeziku.

3.1 Osnovni pojmovi

Klasa je tip koji sadrži podatke i rutine za manipulisanje tim podacima. U nastavku je dat primer klase u SystemVerilog-u. Klasa *transaction* sadrži podatke o podacima i adresi i ima metodu za prikaz. Labele nakon definicija funkcija ili klasa, odnosno bilo kog bloka, su opcione ali korisne jer čine kod mnogo čitljivijim.

```
class transaction;

   bit [1 : 0] addr;
   bit [7 : 0] data_i;

   function void display_transaction();
      $display("\taddr = %0h", this.addr);
      $display("\tdata_i = %0h", this.data_i);
   endfunction : display_transaction

endclass : transaction
```

Kod 9: Primer klase

Terminologija:

- *Class* - klasa je bazični blok koji sadrži podatke i procedure
- *Object* - objekat je instanca klase
- *Handle* - pokazivač na objekat
- *Property* - polje (promenljiva) u klasi (npr. *addr* i *data* iz prethodnog primera)
- *Method* - metoda je proceduralni kod koji manipuliše promenljivama smešten u funkciju ili task (npr. *display_transaction* iz prethodnog primera)

3.1.1 Objekti

Klasa definiše tip podatka, a objekat je instanca te klase. Za razliku od modula, objekti su dinamični. Mogu se kreirati, uništavati, pristupati im se bilo kada u toku izvršavanja koda. Objekat se koristi tako što se prvo deklariše promenljiva odgovarajućeg tipa koja čuva pokazivač na taj tip, a zatim se kreira objekat pozivanjem funkcije *new* i dodeljuje toj promenljivoj. Npr.:

```
transaction tr;
tr = new;
```

Objektima u SystemVerilog-u se uvek pristupa preko pokazivača. Međutim postoji dosta razlika u odnosu na pokazivače u C-u, odnosno mnogo više ograničenja (jedan od razloga za korišćenje reči *handle* umesto *pointer*). C *pointer*-i dozvoljavaju aritmetičke operacije, mogu se definisati za proizvoljne tipove podataka, može im se dodeljivati adresa dok su sve ove operacije zabranjene za SystemVerilog *handle*. Takođe, kastovanje je dosta ograničeno za razliku od C-a.

Važno je zapamtiti da se objekat kreira jedino pozivom funkcije *new*. U narednom primeru *tr1* i *tr2* pokazuju na isti objekat u memoriji i bilo koje promene na *tr1* će uticati na *tr2* i obrnuto.

```
transaction tr1, tr2;
tr1 = new;
tr2 = tr1;
tr2.addr = 3;
tr1.addr = 1;
$display(tr2.addr); // prikazuje vrednost 1
```

Promenljivima unutar objekata i metodama se pristupa pomoću ".". Npr.

```
tr.data_i = 4;
tr.display_transaction();
```

3.1.2 Konstruktor

Kada se objekat kreira, poziva se funkcija *new()* asoricana sa datom klasom. Ova funkcija se zove konstruktor i ne sme specificirati povratni tip. Svaka klasa ima podražumevani konstruktor koji alocira memoriju za dati objkat (slično kao *malloc* u C-u) i inicijalizuje promenljive. Promenljive koje su tipa sa 2 stanja dobijaju vrednost 0, a one sa 4 stanja vrednost X. Naravno, moguće je i definisati sopstveni konstruktor, umesto korišćenja podrazumevanog, koji će odraditi željene operacije. Moguće je i prosleđivati vrednost konstruktorima kako bi se omogućila veća kontrola tokom izvršavanja, npr. konstruktor u klasi *transaction* može biti:

```
function new(logic [7:0] data_value = 8hAA,logic [2:0] addr_value = 0);
  data_i = data_value;
  addr = addr_value;
endfunction : new
```

Za razliku od C++ jezika, u SystemVerilog-u može postojati samo jedan konstruktor. Takođe, neinicijalizovan handle ima specijalnu vrednost *null*.

3.1.3 *Static*

Kao i u C++-u, i SystemVerilog podržava statična polja i metode. Statična polja se koriste kada je potrebno da neki član ima istu vrednost u svim instancama. One su zajedničke za sve objekte date klase odnosno vrednost polja je ista u svim objektima. Primer upotrebe bio bi npr. polje koje broji instance date klase odnosno aktuelne objekte u programu. Statične metode mogu pristupati samo statičnim poljima i metodama u klasi (pokušaj pristupa ne-statičnim poljima rezultuje kompilacionom greškom). Takođe, statične metode ne mogu biti virtualne (objašnjeno u narednim vežbama).

Statična polja i metode se deklarišu koristeći ključnu reč *static*, npr.:

```
class staticClassExample;
  static int count;
  int local_count;

  static function void increaseCount();
    count++;
    // local_count++; nije dozvoljeno
    $display("Count value = %0d\n", count);
  endfunction : increaseCount

endclass : staticClassExample
```

3.1.4 *This*

Ključna reč *this* se koristi da bi se nedvosmisleno specificiralo polje ili metoda trenutne instance klase. Korišćenje ove ključne reči je uglavnom opciono i često izostavljeno, iako veoma doprinosi čitljivosti koda. Najčešće se sreće u konstruktorima. Npr.:

```
class thisExample;
  int x;

  function new(int x = 5);
    this.x = x;
    // this.x se odnosi na polje u klasi
    // x se odnosi na prosledjeni arugment
  endfunction : new

endclass : thisExample
```

3.1.5 *Extern*

Slično kao i u C++ jeziku, i u SystemVerilogu je moguće definisati metodu izvan klase. Ovo se postiže korišćenjem ključne reči *extern*. Za obimne metode korisno je definisati ih izvan klase kako bi sam kod bio čitljiviji. Da bi se metoda definisala izvan klase potrebno je u klasi navesti prototip metode kojem prethodi ključna reč *extern*, a prilikom same definicije metode dodati ime klase i "::" (*scope resolution operator*) ispred imena metode. Npr:

```
class externExample;
  int x;
  extern task taskExample();
endclass : externExample

task externExample::taskExample();
  // do something
endtask : taskExample
```

Prototip metode u klasi i definiciji se moraju poklapati. Međutim, razni simulatori različito reaguju na podrazumevane vrednosti argumente. Neki dozvoljavaju navođenje podrazumevanih argumenata na oba mesta, a neki ne. Ali pošto su podrazumevane vrednosti argumenata bitne samo za poziv metode, a ne i za njenu implementaciju, neophodno ih je navesti samo prilikom navođenja prototipa u klasi.

3.2 Enkapsulacija

Enkapsulacija je tehnika skrivanja podataka unutar klase. Skriveni podaci su dostupni samo u unutrašnjosti klase i može im se pristupati jedino preko metoda klase. Enkapsulacija omogućava zaštitu članova klase jer je pristup dozvoljen jedino "poznatim" korisnicima odnosno metodama čije je ponašanje poznato, čime se može vršiti provera i ograničavanje vrednosti pojedinih članova. Način enkapsulacije u SystemVerilog-u veoma liči na C++. Podrazumevani pristup je *public*. Ukoliko se eksplicitno ne navede pravo pristupa članu klase, on je *publilc* i tom članu (polju ili metodi) se može pristupati kako iz unutrašnjosti, tako i iz spoljašnjosti klase, bez ikakvih ograničenja, koristeći "." operator. Pored *public* pristupa, postoje još dva prava pristupa: *local* i *protected*. *Local* pristup je nalik *private* pristupu u C++ jeziku. Članu klase koji je definisan kao *local* se može pristupati jedino iz unutrašnjosti klase (ne može im se pristupati ni iz klase koja nasleđuje datu klasu). *Protected* pristup se isto sreće u C++ jeziku i omogućava pristup i u nasleđenim klasama, ali i dalje im se ne može pristupati iz spoljašnjosti. Kontrola pristupa se vrši dodavanjem kjučnih reči *local* ili *protected* ispred deklarisanja člana klase, npr:

```
// primer klase
class encapsulationExample;

   integer a = 5; // a je public
   local bit b = 1;
   protected int c;

   task displayExample();
      $display("a = %0d", a);
      $display("b = %0d", b);
```

```
        $display("c = %0d", c);
    endtask : displayExample

endclass : encapsulationExample

// primer upotrebe
module encapsulationUseExample;

    encapsulationExample example = new;

    initial begin
        example.displayExample(); // dozvoljeno; pristup preko metode klase
        $display("a = %0d", example.a); // dozvoljeno; pristup public polju
        // $display("b = %0d", example.b); // nije dozvoljeno; greska pri kompajliranju
    end

    // Rezultat izvrsavanja:
    //
    // a = 5
    // b = 1
    // c = 0
    // a = 5

endmodule : encapsulationUseExample
```

Kod 10: Enkapsulacija

SystemVerilog je jezik za verifikaciju i, za razliku od klasičnih programskih jezika kao što je C++, korišćenje enkapsulacije u praksi je retko. Lak pristup i jednostavna kontrola elementima u verifikacionom okruženju je često bitnija od dugoročne stabilnosti softvera, čak je često i potrebno ubacivati pogrešne vrednosti kako bi se detaljno verifikovao dizajn. Zbog toga, u SystemVerilog-u, članovi klase najčešće imaju podrazumevani, *public*, pristup.

3.3 Nasleđivanje

Nasleđivanje je veza između klasa koja omogućava preuzimanje sadržaja iz jedne klase (*parent* klasa) i uz eventualne modifikacije i dodatke kreira izvedenu (*child*) klasu. *Child* klasa nasleđuje sva polja i metode iz parent klase, može ih modifikovati, ali može i dodavati nova polja i metode. Npr. ukoliko uzmemo primer *transaction* klase sa početka poglavlja, tokom razvoja dizajna moguće je da se ukaže potreba za dodavanjem još jednog člana koji čuva npr. *en* ili *rw*. Tada bi, umesto kreiranja nove klase, nasledili *transaction* klasu i dodali potreban sadržaj. Nasleđivanje se realizuje korišćenjem ključne reči *extends*.

```
class new_transaction extends transaction;

    bit [7 : 0] data_o;
    bit         rw;
    bit         en;

    function void display_transaction();
        super.display_transaction();
        $display("\tdata_o = %0h", this.data_o);
        $display("\trw = %0h", this.rw);
        $display("\ten = %0h", this.en);
    endfunction : display_transaction

endclass : new_transaction
```

Kod 11: Nasleđivanje

3.3.1 Redefinisane metode

Metode u *child* klasi mogu biti nasleđene iz *parent* klase bez modifikacija, mogu biti redefinisane ili mogu biti potpuno nove. Podrazumevano se sve metode iz *parent* klase nasleđuju u *child* klasi, a zatim se iste mogu redefinisati ukoliko je to potrebno - jednostavno se navede nova definicija metode koja će se koristiti u *child* klasi umesto originalne.

3.3.2 *Super*

Ključna reč *super* se koristi u *child* klasi kako bi se referencirali članovi *parent* klase. Neophodno ju je koristiti jedino kada je metoda redefinisana u *child* klasi, a potrebno je pristupiti originalnoj metodi u *parent* klasi.

Pogledati primer redefinisanja metode *display_transaction* u *new_transaction* klasi.

3.3.3 Konstruktor

Kada se *child* klasa instancira, poziva se kontruktor čija je prva akcija da pozove konstruktor *parent* klase čime se postiže pravilan redosled pozivanja konstruktora od bazne klase do svih nasleđenih klasa. Ukoliko konstruktor u *parent* klasi ima argumente, konstruktor u *child* klasi mora postojati i mora pozivati *parent* konstruktor na prvoj liniji (*super.new(...)*).

4 Interfejs i *package*

Interfejsi i *package*-i su dve često korišćene konstrukcije u SystemVerilog-u. Sa porastom kompleksnosti okruženja postaju neophodni radi čitljivosti i ponovne upotrebe koda, ali i olakšavanja same verifikacije. U ovom poglavlju je dat kratak pregled, dok će se potrebne funkcionalnosti detaljno obraditi u narednim vežbama.

4.1 Interfejsi

Interfejsi služe za povezivanje dizajna i testbenča. Cilj je obuhvatiti svu komunikaciju na jednom mestu. Svi signali koji povezuju DUT i testbenč treba da se nalaze u interfesima, čime se postiže jasna struktura koda. Interfejsi povećavaju fleksibilnost koda, olakšavaju pisanje protokol čekera i olakšavaju upotrebu jer se interfesi mogu instancirati i koristiti kao promenljive. Kao i moduli, i interfejsi mogu sadržati parametre, funkcije, taskove, promenljive, ... Definišu se ključnom reči *interface* i mogu sadržati ulazne, izlazne i ulazno-izlazne portove. Interfejsi se instanciraju kao i moduli i to samo jednom, a zatim se pokazivač prosleđuje svim objektima koji treba da koriste ovaj interfejs.

```
interface memory_if(input clk, input rst);

    logic [1 : 0]   addr;
    logic           rw;
    logic           en;
    logic [7 : 0]   data_i;
    logic [7 : 0]   data_o;

endinterface : memory_if

// instanciranje u modulu
module top;

    bit clk;
    bit rst;

    memory_if mem_if(clk, rst);

    // ...

endmodule : top
```

Kod 12: Interfejs

Umesto da se pojedinačno kreiraju i instanciraju svi signali (*addr*, *data*, ...), oni se grupišu u interfejs koji se tada instancira u top modulu, povezuje sa DUT-om i dalje prosleđuje svim objektima kojima je potreban pristup (drajverima, monitorima, ...).

4.2 *Package*

Razvojem verifikacionog okruženja, raste i broj fajlova koji se koriste. Pošto se, po konvenciji, svaka klasa definiše u posebnom fajlu, ubrzo može postati nezgodno pratiti sve potrebne fajlove u projektu. Takođe, ukoliko postoje funkcije koje se koriste na više mesta ili parametri ili osobine potrebne u više klasa ili modula, korisno je definisati ih samo na jednom mestu. *Package* ovo omogućava.

Podacima u *package*-u se može pristupati na dva načina: preko *scope resolution operatora* ("::") ili pomoću *import* naredbe. U nastavku je dat primer:

```
// in example1_pkg.sv
package example1_pkg;
   typedef enum {READ, WRITE} dir_e;
endpackage
```

```
// in example2_pkg.sv
package example2_pkg;
    typedef enum {FALSE, TRUE} bool_e;
endpackage

// in top.sv
`include "example1_pkg.sv"
`include "example2_pkg.sv"
module top;

    import example1_pkg::*;

    dir_e direction;
    example2_pkg::bool_e value;

endmodule : top
```

Kod 13: Package

5 Zadaci

Zadatak U nastavku je dat primer testbenča koji se koristi za verifikaciju memorije. Sastoji se od nekoliko klasa i modula. Svi fajlovi se nalaze u pratećim materijalima za ovu vežbu.

- kod 14 - primeri *transaction* klase
- kod 15 - primer drajver klase koja generiše signale
- kod 16 - *package*; include svih fajlova na jednom mestu
- kod 17 - interfejs; služi za povezivanje DUT-a i okruženja
- kod 18 - jednostavan primer memorije koji služi kao DUT
- kod 19 - glavni modul; u njemu se vrši povezivanje DUT-a i okruženja, generišu se *clk* i *reset* signali i startuje drajver
- kod 20 - skripta za pokretanje simulacije

Analizirati sve fajlove, uočiti način povezivanja, primer nasleđivanja u *transaction* klasi, kao i upotrebu jedne klase u drugoj (*transaction* u drajver klasi).

```
`ifndef TRANSACTION_SV
 `define TRANSACTION_SV

class transaction;

   bit [1 : 0] addr;
   bit [7 : 0] data_i;

   function void display_transaction();
      $display("\taddr = %0h", this.addr);
      $display("\tdata_i = %0h", this.data_i);
   endfunction : display_transaction

endclass : transaction

class newTransaction extends transaction;

   bit [7 : 0] data_o;
   bit         rw;
   bit         en;

   function void display_transaction();
      super.display_transaction();
      $display("\tdata_o = %0h", this.data_o);
      $display("\trw = %0h", this.rw);
      $display("\ten = %0h", this.en);
   endfunction : display_transaction

endclass : newTransaction

`endif
```

Kod 14: Transackije

```
`ifndef DRIVER_SV
 `define DRIVER_SV

class driver;

   newTransaction tr;

    virtual memory_if mem_if;
```

```
    function new(virtual memory_if mem_if);
        this.mem_if = mem_if;
    endfunction : new

    task run();
        tr = new();
        drive_transaction(tr);
        tr.addr = 2'hA;
        tr.en = 1'b1;
        drive_transaction(tr);
        tr.data_i = 8'hAA;
        tr.rw = 1'b1;
        drive_transaction(tr);
    endtask : run

    task drive_transaction(newTransaction tr);
        $display("Driving transaction:");
        tr.display_transaction();
        @(posedge mem_if.clk);
        mem_if.addr <= tr.addr;
        mem_if.data_i <= tr.data_i;
        mem_if.en <= tr.en;
        mem_if.rw <= tr.rw;
    endtask : drive_transaction

endclass : driver

`endif
```

Kod 15: Drajver

```
`ifndef MEMORY_PKG_SV
 `define MEMORY_PKG_SV

package memory_pkg;

 `include "v2_tr.sv"
 `include "v2_driver.sv"

endpackage : memory_pkg

`endif
```

Kod 16: Package

```
`ifndef MEMORY_IF_SV
 `define MEMORY_IF_SV

interface memory_if(input clk, input rst);

    logic [1 : 0]   addr;
    logic           rw;
    logic           en;
    logic [7 : 0]   data_i;
    logic [7 : 0]   data_o;

endinterface : memory_if

`endif
```

Kod 17: Interfejs

```
`ifndef MEMORY_SV
 `define MEMORY_SV
```

```
module memory #(
                parameter ADDR_WIDTH = 2,
                parameter DATA_WIDTH = 8
                )
   (
    input logic                          clk,
    input logic                          rst,
    input logic [ADDR_WIDTH-1:0]  addr_i,
    input logic                          rw_i,
    input logic                          en_i,
    input logic [DATA_WIDTH-1:0]  data_i,
    output logic [DATA_WIDTH-1:0] data_o
    );

   logic [DATA_WIDTH-1:0]        mem [2**ADDR_WIDTH];

   always @(posedge clk or posedge rst) begin
      if(rst) begin
         for(int i = 0; i < 2**ADDR_WIDTH; i++) begin
            mem[i] = 0;
         end
      end
      else begin
         if(en_i) begin
            if(rw_i) begin
               mem[addr_i] <= data_i;
            end
            else begin
               data_o <= mem[addr_i];
            end
         end
      end
   end

endmodule : memory

`endif
```

Kod 18: Memorija

```
`ifndef TOP_SV
`define TOP_SV

`include "v2_memory.sv"
`include "v2_memory_if.sv"
`include "v2_memory_pkg.sv"

module top;

   import memory_pkg::*;

   bit clk;
   bit rst;

   memory_if mem_if(clk, rst);

   memory DUT (
               .clk     (clk),
               .rst     (rst),
               .addr_i (mem_if.addr),
               .rw_i    (mem_if.rw),
               .en_i    (mem_if.en),
               .data_i (mem_if.data_i),
               .data_o (mem_if.data_o)
               );

   driver drv = new(mem_if);
```

```
    initial  begin
       clk  = 0;
       rst  = 1;
       #5 rst =0;
       #500 $finish();
    end

    always #5 clk = ~clk;

    initial  drv.run();

endmodule : top

`endif
```

Kod 19: Top

```
# create library
if  [ file  exists  work] {
     vdel  -all
}
vlib  work

# compile everything
vlog  +acc  -sv v2_top.sv

# run simulation
vsim top
```

Kod 20: Run

Zadatak Napisati klasu monitor koja nadgleda signale sa interfejsa. Svaku ispravnu transackiju koju uoči (*en* visok) treba da sačuva u redu i da na kraju ispiše sve uočene transakcije.