Funkcionalna verifikacija hardvera

Vežba 11
Prikupljanje pokrivenosti

Sadržaj

Vežba 11 je posvećena prikupljanju pokrivenosti. Dat je kratak pregled tipova pokrivenosti, prednosti i mana i objašnjen način implementacije koristeći SystemVerilog. Pokazane su i mogućnosti alata u pogledu prikupljanja pokrivenosti.

1 Uvod

Jedan od težih zadataka u procesu verifikacije je odlučiti kada je verifikacija završena. Da bi smo došli do tog zaključka mora se odgovoriti na dva pitanja: da li su sve osobine dizajna, koje su identifikovane u verifikacionom planu, verifikovane? I da li postoje delovi koda u dizajnu koji se nikad nisu koristili? Da bi smo dali odgovore na ova pitanja uvodi se nova metrika - pokrivenost (engl. *coverage*). Sa porastom veličine i kompleksnosti sistema, merenje pokrivenosti je postalo neophodan deo svakog verifikacionog ciklusa. Pored zaključka o kraju verifikacije, pokrivenost pomaže i pri analizi toka verifikacije dajući odgovore na pitanja nalik: kada je testirana jedna osobina, da li se u isto vreme proverila i neka druga osobina? Da li je proces uporen iz nekog razloga? Da li je moguće smanjiti broj testova kako bi se ubrzao proces, a da se i dalje sve potrebne osobine verifikuju?

Ukratko, *coverage* je metrika koja se koristi za merenje progresa i završetka verifikacije. Daje informacije o tome kada se neki deo dizajna aktivirao tokom simulacije i, što je možda i važnije, da li postoje delovi dizajna koji nikad nisu bili aktivirani. Sa ovim informacijama je moguće podesiti stimulus kako bi ostvarili ciljeve.

Dve najčešće korišćene *coverage* metrike su:

- Strukturna pokrivenost (engl. *code coverage*) - implicitna
- Funkcionalna pokrivenost (engl. *functional coverage*) - eksplicitna

Pojedinačne metrike nisu dovoljne da bi smo došli do odgovarajućih zaključaka. Npr. moguće je ostvariti 100% pokrivenosti koda, a da neke osobine uopšte nisu verifikovane. Takođe je moguće ostvariti 100% funkcionalne pokrivenosti, a manji procenat pokrivenosti koda. Zato se uvek koriste obe metrike, uz detaljno razrađen plan o prikupljanju pokrivenosti.

U nastavku vežbe objašnjene su obe metrike, sa posebnim akcentom na funkcionalnoj pokrivenosti.

2 Strukturna pokrivenost

Strukturna pokrivenost ili pokrivenost koda daje informacije o stepenu aktivacije sors koda tokom verifikacije čime se omogućava praćenje struktura koje se nikad ne aktiviraju. Glavna prednost ove metrike je što je implicitna odnosno kreiranje modela je automatsko. Za korišćenje pokrivenosti koda nije potrebno dodavati poseban kod i ne zahteva poseban pristup tokom verifikacije.

Mana ovog tipa pokrivenosti je što je moguće imati 100% pokrivenosti, a da i dalje postoje greške u dizajnu. Npr. stimulus je aktivirao liniju koja sadrži grešku, ali efekti nisu propagirani dovoljno da bi je odgovarajuće provere uhvatile. Drugo ograničenje je što ne daje indikacije o samoj funkcionalnosti. Npr. moguće je da postoji deo funkcionalnosti DUT-a koji nikad nije verifikovan, a možda nije čak ni implementiran. Iako bi tada imali 100% pokrivenosti koda, proces verifikacije ne bi bio uspešan. I pored ovih problema, zbog automatskog prikupljanja, pokrivenost koda se jako puno koristi i daje dobre indikacije o toku same verifikacije.

Postoji više tipova strukturne pokrivenosti. U nastavku su ukratno opisane. Za detalje pogledati odgovarajuće predavanje.

- *Toggle coverage* - meri koliko puta je svaki bit registra ili signala promenio vrednost. Analiza svih ovih podataka može biti redudantna, ali je i dalje korisna u nekim situacijama npr. kod povezanosti dva IP bloka
- *Line coverage* - meri koje linije koda su aktivirane tokom simulacije i koliko puta. Ova metrika često okriva da postoji deo koda koji se jako retko aktivira zbog manja stimulusa ili da postoji deo koda koji se nikad ne koristi npr. zbog konfiguracije IP-a
- *Statement coverage* - meri koje naredbe su aktivirane i koliko puta. Često je korisnije od merenja linija jer jedna naredba može sadržati više linija koda, ali i više naredbi može postojati u jednoj liniji
- *Block coverage* - varijacija *statement coverage*-a koja meri da li se blok koda izvršio. Blok čine naredbe unutar uslovnih naredbi ili unutar proceduralne definicije. Ideja je da kada se dođe do određenog bloka, sve naredbe unutar njega će biti izvršene
- *Branch coverage* - meri da li su uslovi unutar kontrolnih strukura (*if*, *case*, *while*, *for*, *repeat*, ...) evaluirane i kao tačne i kao netačne.
- *Expression coverage* - meri da li je svaki operand unutar uslova evaluiran i kao tačan i kao netačan
- *Finite-state machine coverage* - pošto je alat u mogućnosti da identifikuje FSM u RTL-u, moguće je prikupljati informacije o pokrivenosti, npr. koliko puta se ušlo u svako stanje ili da li su svi prelazi viđeni

Pogledati četvro poglavlje za informacije o alatu i kako doći do podataka o ovom tipu pokrivenosti.

3 Funkcionalna pokrivenost

Cilj funkcionalne verifikacije je utvrditi da li dizajn implementira sve osobine i funkcioniše na način opisan u funkcionalnoj specifikaciji. Međutim do zaključka o tome da li je neka funkcionalnost stvarno implemntirana i da li je verifikovana, ne možemo doći na osnovu praćenja pokrivenosti koda. Zbog toga se uvodi nova, eksplicitna metrika - funkcionalna pokrivenost. Cilj ove metrike je merenje progresa verifikacije u odnosu na funkcionalne zahteve dizajna.

Jedan od problema korićenja *constrained-random* pristupa generisanja stimulusa je što ne znamo tačno koje funkcionalnosti se verifikuju (šta je tačno dovedeno na ulaz DUT-a) bez da ručno analiziramo *waveform*-e tokom simulacije. Međutim, praćenje funkcionalne pokrivenosti nam omogućava upravo ovo - određivanje funkcionalnosti koje su verifikovane bez vizuelne analize samih signala.

Mane ove metrike su što, pošto nije implicitna, ne može biti automatski implementirana. Implementacija dobrog modela pokrivenosti zahteva dosta vremena pošto je potrebno prvenstveno napraviti dobar plan, identifikovati sve osobine od interesa i odrediti način na koji će se prikupljati podaci o pokrivenosti. Nakon toga, potrebno je izvršiti samu implementaciju u okruženju.

U SystemVerilog jeziku postoje dve glavne konstrukcije pomoću kojih je moguće implementirati ovu metriku:

- *Cover groups* - obuhvata prikupljanje vrednosti različitih promenljivih. Ove vrednosti se prikupljaju bilo nadgledanjem interfejsa, registara, kontrolnih signala... Vrednosti koje se mere se uzimaju u jednom vremenskom trenutku.
- *Cover properties* - opisuje temporalne osobine između sekvenci različitih događaja. Najčešće se koriste za proveru *handshake* sekvenci u nekom protokolu (npr. posle *req* sledi *ack* i sl.)

Na ovom kursu ćemo se zadržati na modelovanju grupa, što je i najčešći način impementacije modela pokrivenosti. Za informacije o *cover propety* zainteresovani studenti neka pogledaju odgovarajuće poglavlje u "Cookbook"-u.

U nastavku je dat jednostavan primer sakupljanja funkcionalne pokrivenosti za memoriju. U primeru postoji jedna grupa čiji se uzorak uzima na svakoj rastućoj ivici en signala i uzimaju se vrednosti adrese (u tri opsega), parnosti (za parnu i neparnu) i *rw* (čitanje ili upis). Nakon simulacije ovog koda, možemo videti informacije o funkcionalnoj pokrivenosti - koje vrednosti adresa su viđene, koji tip parnosti i da li se vršio upis ili čitanje. Deo izveštaja je takođe prikazan. Iz njega možemo zaključiti da su viđene adrese iz donjeg opsega, pri čemu je viđena samo *read* operacija uz *even* parnost.

```
module simple_coverage();

  logic [7:0] data;
  logic [7:0] addr;
  logic       par;
  logic       rw;
  logic       en;

  // covergroup
  covergroup memory @ (posedge en);
    option.per_instance = 1;
    address : coverpoint addr {
      bins low  = {0,50};
      bins med  = {51,150};
      bins high = {151,255};
    }
    parity : coverpoint par {
      bins even = {0};
      bins odd  = {1};
```

```
    }
    read_write : coverpoint rw {
      bins read = {0};
      bins write = {1};
    }
  endgroup

  // instance of covergroup
  memory mem = new();

  // drive stimulus
  task drive (input [7:0] a, input [7:0] d, input r);
    #5;
    en   <= 1;
    addr <= a;
    rw   <= r;
    data <= d;
    par  <= ^d;
    #5;
    en   <= 0;
    rw   <= 0;
    data <= 0;
    par  <= 0;
    addr <= 0;
    rw   <= 0;
  endtask

  // stimulus generation
  initial begin
    en = 0;
    repeat (10) begin
      drive ($random, $random, $random);
    end
    #10 $finish;
  end

endmodule : simple_coverage
```

Kod 1: Primer sakupljanja funkcionalne pokrivenosti

Name	Coverage	Goal	% of Goal	Status
/simple_coverage				
TYPE memory	44.4%	100	44.4%	
CVP memory::address	33.3%	100	33.3%	
CVP memory::parity	50.0%	100	50.0%	
CVP memory::read_writ...	50.0%	100	50.0%	
INST \/simple_coverag...	44.4%	100	44.4%	
CVP address	33.3%	100	33.3%	
bin low	9	1	100.0%	
bin med	0	1	0.0%	
bin high	0	1	0.0%	
CVP parity	50.0%	100	50.0%	
bin even	9	1	100.0%	
bin odd	0	1	0.0%	
CVP read_write	50.0%	100	50.0%	
bin read	9	1	100.0%	
bin write	0	1	0.0%	

Slika 1: Primer sakupljanja funkcionalne pokrivenosti

U nastavku je dat detaljan opis strukture grupe i njenih mogućnosti.

3.1 Implementacija

Covergroup konstrukcija je tip definisan od strane korisnika. Samo jednom se definiše, a moguće je instancirati ih više puta. Ima dosta sličnosti sa klasama. Jedom kada se definiše, instance se kreiraju pozivanjem *new*. Sama grupa se može definisati u *package*-u, modulu, program, interfesju ili klasi.

Grupa se definiše korišćenjem *covergroup* i *endgroup* ključnih reči, a instancira pozivom *new*. Npr.:

```
covergroup cg_example
  // ...
endgroup
cg_example cg = new();
```

Napomena: grupa za pokrivenost mora biti instancirana u klasama kako bi sakupljala podatke. Česta greška je zaboraviti poziv *new*. U tom slučaju neće biti prijavljena greška i grupa se neće pojavljivati u finalnom izveštaju.

U UVM okruženju, grupe za praćenje pokrivenosti se uglavnom definišu i kreiraju u odgovarajućim komponentama i na odgovarajućem hijerarhijskom nivou. Uglavnom su sadržane u odgovarajućim komponentama za analizu: monitorima, *scoreboard*-u ili posebnim *coverage collector* komponentama čija je jedina uloga sakupljanje funkcionalne pokrivenosti.

Covergroup-a može sadržati:

- Događaj - definiše kada se uzima uzorak. Ukoliko se izostavi korisnik mora eksplicitno pozvati metodu
- tačke pokrivenosti (engl. *coverpoint*)- može biti promenljiva ili izraz
- *Cross coverage* - između dve ili više tačaka pokrivenosti
- Opcije - kontrolišu ponašanje grupe
- Opcione formalne argumente

Svi navedeni konstrukti su detaljnije objašnjeni u narednim poglavljima.

3.1.1 *Trigerovanje*

Osim definisanja strukture grupe, potrebno je i odlučiti u kom trenutku će se uzimati uzorci tj. kada su podaci spremni za semplovanje. Postoji više načina na koje je ovo moguće specificirati. U nastavku su objašnjeni najčešće korišćeni.

Prilikom definisanja grupe, moguće je i definisati događaj na kojem će se uzimati uzorci. Npr:

```
event e1;
covergroup cg_example @(e1);
  coverpoint x;
endgroup
// ...
cg_example cg = new();
```

U ovom slučaju podaci se prikupljaju prilikom svakog događaja *e1*.

Drugi način trigerovanja grupe je eksplicitnim pozivanjem funkcije *sample()*. Npr:

```
covergroup cg_example;
  coverpoint x;
endgroup
// ...
```

```
cg_example cg = new();
// ...
cg.sample();
// ...
```

Pozivanje funkcije *sample* nad instancom se može obaviti u bilo kom trenutku.

3.1.2 Sakupljanje podataka

Informacije o pokrivenosti se sakupljaju korišćenjem tačaka pokrivenosti (engl. *coverpoint*). Oni zapravo predstavljaju promenljivu ili izraz koji služi za prosleđivanje vrednosti od interesa. Kada se tačka definiše, kreira se određeni broj *bin*-ova koji služe za praćenje samih vrednosti i koliko puta su određene vrednosti viđene tokom simulacije. *Bin*-ovi predstavljaju osnovnu jedinicu mere funkcionalne pokrivenosti. Npr. ako pratimo pokrivenost jednobitne promenljive, za nju je moguće kreirati najviše dva *bin*-a - za vrednost nula i za vrednost jedan. Svaki put kada se uzme uzorak, *bin* koji pokriva trenutnu vrednost se inkrementuje i tako vodi računa o viđenosti podataka tokom simulacije.

Bin-ovi se mogu kreirati na više načina i mogu sadržati različite vrednosti. Moguće je kreirati po jedan bin za svaku vrednost ili da jedan bin pokriva opseg vrednosti. Takođe je moguće eksplicitno odrediti broj bin-ova i vrednosti ili automatski kreirati željeni broj *bin*-ova. U nastavku su opisane razne, i najčešće korišćene, mogućnosti koje SystemVerilog pruža u pogledu sakupljanja podataka za praćenje funkcionalne pokrivenosti. Pored opisanih postoji veliki broj dodatnih mogućnosti jezika, ali one izlaze iz okvira ovog kursa. Za detalje pogledati odgovarajuću literaturu.

Automatski (implicitni) *bin*-ovi Ukoliko se eksplicitno ne naglase, SystemVerilog automatski kreira *bin*-ove za tačke pokrivenosti. Broj kreiranih bin-ova će zavisiti od same promenljive ili izraza. Npr. za N-bitni izraz postoji 2N mogućnih vrednosti odnosno kreiraće se 2N bin-ova. Ovo će biti ispunjeno do god broj kreiranih bin-ova ne prelazi zadatu granicu. Podrazumevana granica je 64 bin-a, ali je ovo moguće kontrolisati modifikovanjem odgovarajuće opcije (pogledati poglavlje ispod).

Kada broj mogućih vrednosti prevazilazi broj *bin*-ova, vrednosti će se ravnomerno raspodeliti po opsezima. Na primer, promenljiva od 16-bita ima ukupno 65,536 mogućih vrednosti (216). Ukoliko ostavimo broj automatsko kreiranih *bin*-ova na podrazumevanoj vrednosti od 64, u ovom slučaju će se kreirati 64 bin-a pri čemu svaki pokriva 1024 vrednosti.

Sintaksno, *bin*-ovi će se automatski kreirati kada se eksplicitno ne navedu. Npr:

```
bit en;
covergroup cg_example;
  coverpoint en; // autobin {0} i autobin {1}
endgroup
```

Type: cg_example

- **CVP**: cg_example::en
  - **bin** auto ['b0]
  - **bin** auto ['b1]

Eksplicitni *bin*-ovi U većini slučajeva je potrebno tačno odrediti vrednosti *bin*-ova, pri čemu im se zadaje i ime. Ovo je preporučeni način korišćenja, kada god je to moguće, pošto daje bolji pregled i olašava dalju analizu.

U narednom primeru kreiraju se tri *bin*-a za sakupljanje informacija o vrednosti adrese. Adresa može imati ukupno 256 vrednosti. Kreiraju se tri *bin*-a: *low* koji prikuplja vrednosti 0-50, *med* 51-150 i *high* za 151-255.

```
logic [7:0] addr;
covergroup cg_addr;
  cp_address : coverpoint addr {
    bins low  = {0,50};
    bins med  = {51,150};
    bins high = {151,255};
  }
endgroup
```

Napomena: za navođenje vrednosti se korsite vitičaste zagrade, a ne *begin..end* zato što nije u pitanju proceduralni kod.

Prilikom navođenja vrednosti moguće je koristiti razne kombinacije - više pojedinačnih vrednosti, više opsega itd. Posebna vrednost je *default* odnosno *bin* za sve vrednosti koje nisu prethodno navedene. Npr:

```
logic [7:0] addr;
covergroup cg_addr;
  cp_address: coverpoint addr {
    bins b1 = {0,2,7};
    bins b2 = {11:20};
    bins b3 =  {[30:40],[50:60],77};
    bins b4 =  {[79:99],[110:130],140};
    bins b5 = {160,170,180};
    bins b6 = {200:255};
    bins b7 = default;
  }
endgroup
```

Ignorisani i nedozvoljeni *bin*-ovi Za neke tačke pokrivenosti se neće sve vrednosti dogoditi. Npr. možda se koriste samo donjih 3 bita promenljive od 5 bita. U takvim slučajevima nikada nećemo stići do 100% pokrivenosti, ukoliko se koriste automatski *bin*-ovi. U ovakvim i sličnim situacijama korisno je navesti vrednosti koje nisu od interesa i koje treba da se ignorišu. SystemVerilog ovo omogućava korišećem *ignore_bins*. Navedene vrednosti se neće prikupljati. Npr:

```
logic [7:0] addr;
covergroup cg_addr;
  cp_address: coverpoint addr {
    ignore_bins ignore_vals = {0,1,2,3};
  }
endgroup
```

Takođe, mogu postojati nedozvoljene vrednosti koje ne samo da treba ignorisati već treba i javiti grešku ukoliko se dese. Iako je najbolje raditi provere u odgovarajućim monitorima ili *scoreboard*-u, moguće je definisati i nedozvoljene *bin*-ove koji će javiti grešku ukoliko se primeti navedena vrednost.

```
logic [7:0] addr;

covergroup cg_addr;
  cp_address: coverpoint addr {
    illegal_bins ignore_vals = {0,1,2,3}; // javi gresku ukoliko se dese
  }
endgroup
```

3.1.3 *Cross coverage*

Tačka pokrivenosti čuva viđene vrednosti za jednu promenljivu ili izraz. Često je potrebno imati informacije o tome koje vrednosti su višene na više promenljivih u jednom trenutku. Npr. ne samo koja adresa je viđena već da li je viđena prilikom upisa ili čitanja. Za ovakve situacije koristi se *cross coverage*. Na ovaj način omogućava se merenje pokrivenosti dva ili više *coverpoint*-a. U opštem slučaju ako jedna promenljiva ima N vrednosti, a druga M, potrebno je NxM bin-ova da bi se čuvale sve kombinacije.

Unutar *cross* konstrukcije nije moguće direktno koristiti promenljive ili izraze, već samo prethodno navedene *covepoint*-e. Npr:

```
logic [7:0] addr;
bit dir;

covergroup cg_addr;
  cp_address : coverpoint addr;
  cp_dir : coverpoint dir;
  cx_addr_dir : cross cp_address, cp_dir;
endgroup
```

CROSS: cx_addr_dir

- **bin** <auto['b00000000:'b00000011],auto['b0]>
- **bin** <auto['b00000100:'b00000111],auto['b0]>
- **bin** <auto['b00001000:'b00001011],auto['b0]>
- **bin** <auto['b00001100:'b00001111],auto['b0]>
- **bin** <auto['b00010000:'b00010011],auto['b0]>
- ...

U ovom slučaju za *cross* će se automatski kreirati *bin*-ovi. Voditi računa da će ovakva kod rezultovati u jako velikom broju *bin*-ova, što često nije željena situacija (na slici je prikazan samo mali deo od ukupno kreiranih *bin*-ova). Tada se sakuplja velika količina podataka koje je uglavnom teško analizirati. Zbog toga je i za *cross* moguće specificirati željene *bin*-ove na prethodno opisane načine, pri čemu se mogu koristiti i *illegal_bins* i *ignore_bins* konstrukcije. Npr:

```
logic [7:0] addr;
bit dir;

covergroup cg_addr;
  cp_address : coverpoint addr {
    bins low  = {0,50};
    bins med  = {51,150};
    bins high = {151,255};
  }
  cp_dir : coverpoint dir;
  cx_addr_dir : cross cp_address, cp_dir {
    bins read_addr  = binsof(cp_dir) intersect {0};
    bins write_addr = binsof(cp_dir) intersect {1};
  }
endgroup
```

CVP: cp_addr

- **bin** low
- **bin** med
- **bin** high

CVP: cp_dir

- **bin** auto [b'0]
- **bin** auto [b'1]

CROSS: cx_addr_dir

- **bin** read_addr
- **bin** write_addr

U ovom slučaju sakupljaće se dva *bin*-a, jedan za operaciju upisa, a drugi za čitanje. *Intersect* služi za odabir vrednosti koje želimo u *cross*-u. Ovo se često koristi uz *ignore_bins*. Za prethodni primer mogli bi imati:

```
cx_addr_dir : cross cp_address, cp_dir {
  ignore_bins read = binsof(cp_dir) intersect {0};
  ignore_bins write_addr_zero = binsof(cp_dir) intersect {1} && binsof(cp_address) intersect {0};
}
```

CVP: cp_addr

- **bin** low
- **bin** med
- **bin** high

CVP: cp_dir

- **bin** auto [b'0]
- **bin** auto [b'1]

CROSS: cx_addr_dir

- **bin** <med, auto['b1]>
- **bin** <high, auto['b1]>
- **ignore_bin** read
- **ignore_bin** write_addr_zero

U ovom slučaju ćemo ignorisati sve *bin*-ove gde *dir* ima vrednost 0, ali i one gde *dir* ima vrednost 1, a *addr* nisku vrednost. Na ovaj način se može lako kontrolisati automatsko kreiranje *bin*-ova, odnosno smanjiti broj kreiranih *bin*-ova čime će ubrzati simulacija i olakšati analiza dobijenih rezultata.

3.1.4 Opcije

Prilikom definisanja *cover* grupe moguće je navesti i dodatne opcije koje kontrolišu ponašanje grupe, tačaka pokrivenosti i *cross coverage*-a. Postoje dve grupe opcija: one koje se odnose na neku specifičnu instancu i one koje se odnose na sve instance. Pošto je tipično verifikaciono okruženje dosta veliko i sadrži veliki broj grupa za praćenje pokrivenosti, prikupljanje ovih podataka može biti veoma zahtevan posao. Mogućnost da se npr. postavlja veći prioritet nekih grupa ili kontroliše cilj svake grupe može veoma olakšati ovaj posao.

U tabeli 1 je dat opis ovih opcija:

Od svih navedenih, najčešće se koriste weight, *goal*, *at_least* i *per_instance* opcije. Da bi se specificirala opcija, potrebno je dodeliti joj odgovarajuću vrednost na odgovarajućem nivou, npr:

Opcija	**Podrazumevana vrednost**	**Opis**
weight	1	Ako se postavi na nivou grupe, specificira težinu instance ove grupe prilikom računanja ukupne pokrivenosti instanci u simulaciji. Ako se postavi na nivou *coverpoint*-a (ili *cross*-a), specificira težinu *coverpoint*-a (ili *cross*-a) prilikom računanja ukupne pokrivenosti date grupe
goal	90	Cilj za instancu grupe ili *coverpoint*-a (ili *cross*-a) u instanci
name	unique name	Ime instance
comment	""	Komentar uz instancu
at_least	1	Minimalni broj puta koji je protrebno "pogoditi" *bin* pre nego što se deklariše kao "pogođen"
detect_overlap	0	Da li da se prikazuje upozorenje ukoliko ima preklapanja između opsega dva *bina* u *coverpoint*-u
auto_bin_max	64	Maksimalni broj automatsko kreiranih *bin*-ova
per_instance	0	Svaka instanca učestvuje u ukupnim podacima o pokrivenosti date grupe. Kada je ova vrednost *true*, informacije o pokrivenosti za instancu se takođe prate i uključuju u izveštaj. Kada je *false*, ne moraju se čuvati podaci za pojedinačnu instancu

Tabela 1: *Covergroup* opcije

```
covergroup cg_example ()
  option.comment = "Example covergroup";
  option.per_instance = 1;
  option.goal = 100;
  option.weight = 50;
  cp_addr : coverpoint addr {
    option.auto_bin_max = 100;
  }
  cp_data : coverpoint data {
    option.auto_bin_max = 10;
  }
endgroup
```

Napomena: voditi računa da QuestaSim i većina simulatora neće prikazivati informacije o pojedinačnim *bin*-ovima ukoliko *per_instance* opcija nije postavljena na 1.

4 *Coverage* u Vivadu

U ovom poglavlju su opisane komande u Vivado alatu koje služe za prikupljanje pokrivenosti. Izveštaj o pokrivenosti generuše xcrg (*Xilinx Coverage Report Generator*) alat i da bi on to uradio neophodno je nakon izvršene simulacije ukucati sledeću komandu u.tcl konzolu:

```
xcrg -report_format html -dir xcrg -report_format html -dir <putanja do xsim direktorijuma> -report_dir
    <putanja do direktorijuma gde ce izvestaj biti sacuvan>
```

Prvi argument "-report_format" specificira tip fajla u kome će se nalaziti izveštaj. U prethodnoj komandi taj tip je html, ali moglo je umesto toga da se napiše "text" da bi se izveštaj sačuvao u tekstualnom formatu (Radi preglednosti preporuka je da se koristi html format). Drugi argument, "-dir xcrg", je putanja do direktorijuma u kome se nalaze rezultati simulacije. On se nalazi unutar .sim direktorijuma koji se može pronaći u direktorijumu Vivado projekta. Treći argumen "report_dir" jeste putanja do lokacije na kojoj želite da sačuvate izveštaj. Sledeći primer ilustruje kako bi trebalo da izgleda prethodno opisana komanda:

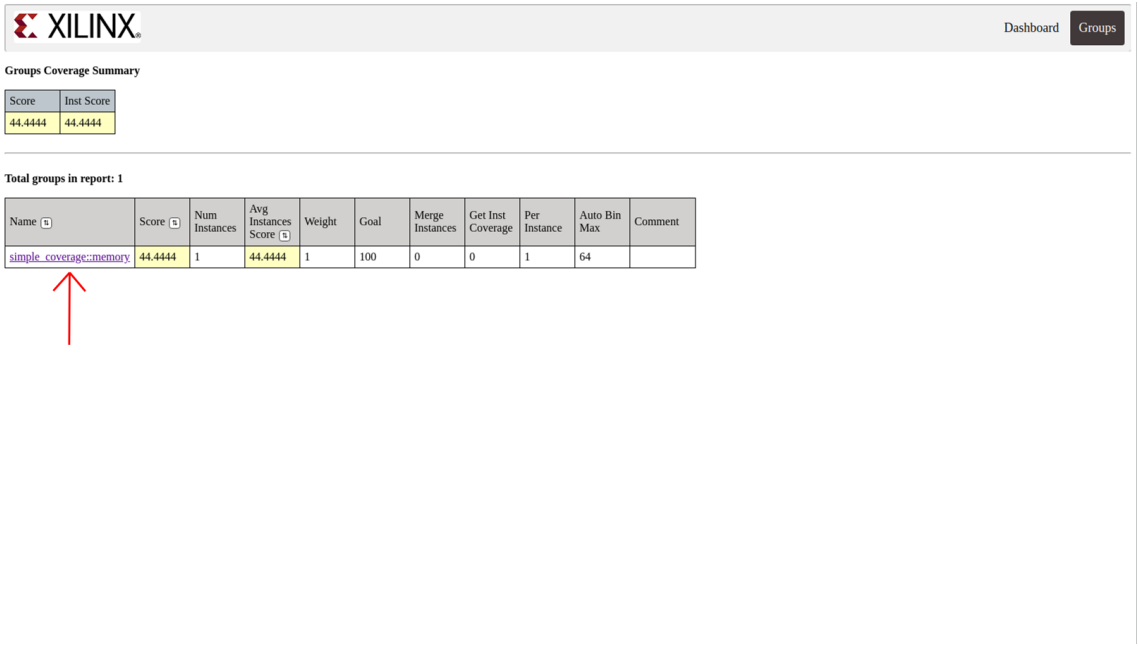
```
xcrg -report_format text -dir /home/nikola/Documents/saradnik/Funkcionalna_verifikacija/vivado_projects/
    v11_vivado_priprema/v11_vivado_priprema.sim/sim_1/behav/xsim -report_dir /home/nikola/Documents
    /
```

Nakon što se ova .tcl komanda izvrši na zadatoj putanji će se pojaviti nekoliko fajlova. Najbitniji od njih jeste dashboard.html fajl koji kada se otvori prikazuje sledeće:

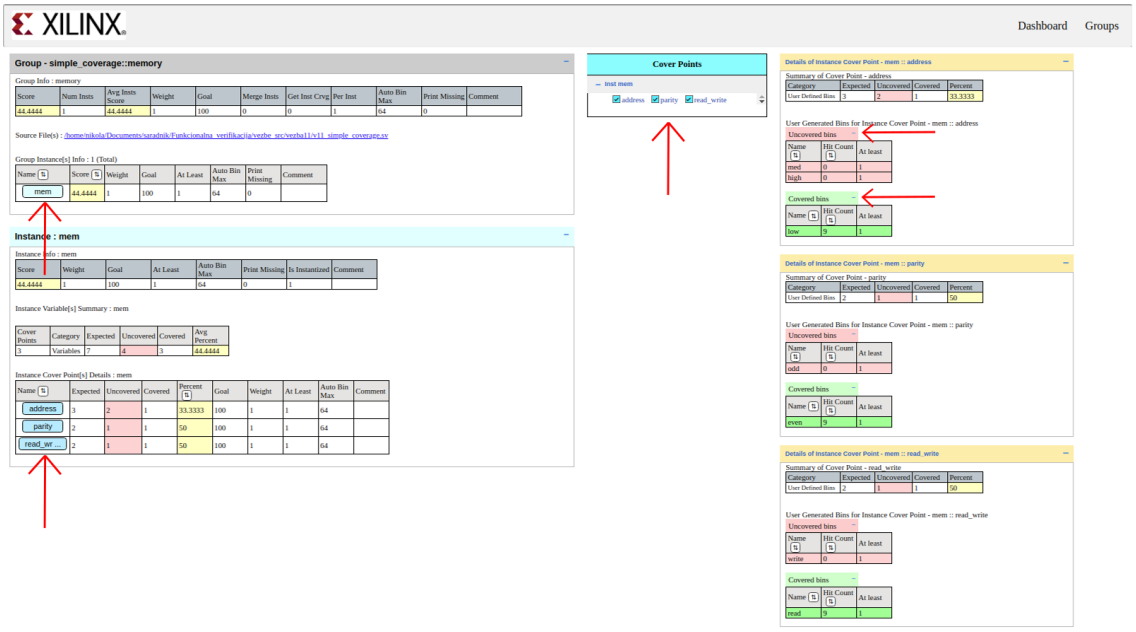
Slika 2: Sadržaj *dashboard* fajla

U njemu su prikazane informacije o verziji Vivado simulatora, pokrenutoj komandi, datumu, broju testova, itd. Iz tog prozora može da se pređe u Groups prozor (crvena strelica na slici 2) u kome se nalazi spisak svih *cover* grupa koje postoje u verifikacionom okruženju (slika 3).

Pritiskom na opciju označenu crvenom strelicom (*simple_ coverage*) otvoriće se detaljan izveštaj o toj *cover* grupi (slika 4)



Slika 3: Sadržaj *groups* fajla



Slika 4: Sadržaj pojedinačnog *group* fajla

Dodatno, nakon što se otvori prozor prikazan na slici 4 moguće je sakriti ili okriti neke delove izveštaja pritiskom na opcije označene crvenom strelicom.

Napomena: Poslednja verzija Vivado alata ne podržava analizu strukturalne pokrivenosti. Ukoliko je to neophodno, mora se iskoristiti neki drugi alat. Takođe, detaljnije informacije u tome kako se koristi xcrg alat možete pronaći u sledećim dokumentacijama:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug900-vivado-logic-simulation.pdf.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug937-vivado-design-suite-simulation-tuto pdf.

5 Zadaci

Zadatak Za primer modula datog na početku vežbe i u dodatnim materijalima sakupiti podatke o pokrivenosti koji se odnose na sledeći uslov u verifikacionom planu: "Na *data* signalu će se pojaviti granične vrednosti (0 i 255)". Definisati *bin*-ove za ove dve vrednosti, kao i jedan *bin* za sve preostale vrednosti.

Zadatak Proširiti prethodni zadatak tako da pokrije i sledeći zahtev: "Operacije upisa i čitanja (*rw* signal) treba da budu obavljene sa obe vrednosti parnosti (*par* signal)".

Zadatak Proširiti prethodni zadatak tako da pokrije i sledeći zahtev: "Potrebno je upisati granične vrednosti podataka (0 i 255) na granične adrese (0 i 255)".

Zadatak Proširiti prethodni zadatak tako da pokrije i sledeći zahtev: "Pratiti vrednosti pročitanih podataka, pri čemu podaci sa *par*=1 nisu od interesa".