Uvod u mikroračunarsku elektroniku

Predavanje IV

Sadržaj predavanja

- IF naredbe
- CASE naredbe
- NULL naredbe
- LOOP naredbe
- ASSERT i REPORT naredbe

Sekvencijalne naredbe

- U ovom predavanju videćemo kako se može manipulisati podacima unutar procesa
- Ovo se ostvaruje pomoću sekvencijalnih naredbi koje se tako zovu jer se izvršavaju sekvencijalno (jedna za drugom)
- Već smo se upoznali sa jednom osnovnom sekvencijalnom naredbom, naredbom dodele
- Naredbe koje ćemo uvesti u ovom predavanju odnose se na kontrolisanje akcije (ponašanja) unutar modela, pa se često zovu kontrolne strukture
- One dozvoljavaju selekciju između alternativnih tokova izvršavanja kao i ponavljanje izvršavanja

IF naredbe

IF naredbe I

- U većini modela, ponašanje zavisi od skupa uslova koji moraju ili ne smeju biti zadovoljeni

- Možemo koristiti *if* naredbu da bi smo opisali ovakvo ponašanje. Sintaksno pravilo za *if* naredbu je

if_naredba ⇐
[if_labela:]
if *bulov*_izraz **then**
{sekvencijalna_naredba}
{**elseif** *bulov*_izraz **then**
{sekvencijalna_naredba}}
[**else**
{sekvencijalna_naredba}]
end if [if_labela];

IF naredbe II

- Labela se može koristiti da se označi *if* naredba. Primer za jednostavnu *if* naredbu mogao bi biti

```
if en = '1' then
        stored_value := data_in;
end if;
```

- Izraz neposredno iza ključne reči *if* je uslov koji se koristi da bi se odlučilo da li treba preneti izvršavanje na blok naredbi posle ključne reči *then*

- Ako je uslov zadovoljen (promenljiva *en* ima vrednost jednaku 1) tok izvršavanja prenosi se na blok nardbi koji sledi neposredno iza ključne reči *then*, a ako uslov nije zadovoljen onda se taj blok preskače i prelazi se na izvršavanje naredbe koja sledi posle *if* naredbe

IF naredbe III

- Takođe možemo definisati ponašanje u slučaju da uslov nije ispunjen

```
if sel = 0 then
        result <= input_0;  -- izvršava se ako je sel = 0
else
        result <= input_1;  -- izvršava se ako je sel /= 0
end if;
```

IF naredbe IV

- Vrlo često javlja se potreba za proverom nekoliko raličitih uslova i izvršavanjem različitih naredbi u svakom slučaju
- Da bi smo to postigli možemo koristiti složeniji oblik *if* naredbe, na primer:

```
if mode = immediate then
        operand := immed_operand;
elsif opcode = load or opcode = add or opcode = substract then
        operand := address_operand;
end if;
```

- U ovom primeru prvo se testira prvi uslov i ako je on zadovoljen prelazi se na izvršavanje naredbe posle ključne reči *then*
- Ako uslov nije zadovoljen, proverava se drugi uslov i ako je zadovoljen, izvršava se naredba posle drugog *then*. Ako i drugi uslov nije zadovoljen izvršava se naredba posle ključne reči *else*.

IF naredbe V

- Broj naredbi koje se mogu nalaziti u nekoj od grana *if* naredbe nije ograničen na jednu, već ih može biti proizvoljno mnogo

- Na primer

```
if opcode = halt_opcode then
        PC := effective_address;
        executing := false;
        halt_indicator := true;
end if;
```

- Ako je uslov zadovoljen izvršavaju se sve tri naredbe jedna nakon druge

- Ako uslov nije zadovoljen ne izvršava se ni jedna naredba

IF naredbe VI

- U svakoj od grana *if* naredbe može se nalaziti bilo koja sekvencijalna naredba. To znači da su dozvoljene ugnježdene *if* naredbe, na primer:

```
if phase = wash then
        if cycle_select = delicate_cycle then
                agitator_speed <= slow;
        else
                agitator_speed <= fast;
        end if;
        agitator_on <= true;
end if;
```

- U ovom primeru, prvo se testira uslov *phase* = *wash*, i ako je zadovoljen prelazi se na izvršavanje ugnježdene *if* naredbe
- Znači dodela *agitator_speed* <= *slow* se izvršava samo ako su oba uslova zadovoljena, a dodela *agitator_speed* <= *fast* se izvršava samo ako je prvi uslov zadovoljen, a drugi uslov nije zadovoljen

IF naredbe VII

- Napišimo bihevijalni model jednostavnog termostata za grejač
- Uređaj se može modelovati kao entitet sa dva celobrojna ulaza, jedan koji specificira željenu temperaturu i drugi koji je spojen sa termometrom, i jednim izlazom tipa *boolean* koji uključuje i isključuje grejač
- Termostat uključuje grejač ako izmerena temperatura padne za dva stepena ispod željene temperature, a isključuje grejač ako izmerena temperatura poraste za dva stepena iznad željene temperature
- Desno su prikazani entitet i arhitekturno telo za termostat

```
entity thermostat is
    port (desired_temp, actual_temp: in integer;
          heater_on: out boolean);
end entity thermostat;

architecture example of thermostat is
begin
    controller: process (desired_temp,
                         actual_temp) is
    begin
        if actual_temp < desired_temp-2 then
            heater_on <= true;
        elsif actual_temp > desired_temp+2 then
            heater_on <= false;
        end if;
    end process controller;
end architecture example;
```

CASE naredbe

CASE naredbe I

- Ako imamo model čije ponašanje zavisi od vrednosti jednog izraza, možemo koristiti *case* naredbu

- Sintaksa *case* naredbe glasi:

case_naredba ⇐
[*case*_labela]
case izraz **is**
(when alternativa => {sekvencijalna_naredba}**)**
{. . .}
end case [*case*_labela];

alternativa ⇐ **(**prost_izraz | diskretni_opseg | **others)** {| . . .}

- Labela se koristi za označavanje *case* naredbe

CASE naredbe II

- Pretpostavimo da modelujemo aritmetičko-logičku jedinicu (ALU), sa kontrolnim ulazom, *func*, koji je deklarisan kao nabrojivi tip:

```
type alu_func is (pass1, pass2, add, substract);
```

- Ponašanje ALU možemo opisati koristeći *case* naredbu:

```
case func is
    when pass1 =>
        result := operand1;
    when pass2 =>
        result := operand2;
    when add =>
        result := operand1+operand2;
    when substract =>
        result := opreand1-operand2;
end case;
```

CASE naredbe III

- *Case* naredba slična je *if* naredbi po tome što obe biraju između mogućih blokova sekvencijalnih naredbi. Razlika je u tome kako se biraju naredbe koje treba izvršiti.

- *If* naredba proverava uzastopne Bulove uslove jedan za drugim sve dok se ne pronađe jedan koji je zadovoljen. Blok naredbi koji odgovara tom uslovu zatim se izvršava.

- *Case* naredba ispituje jedan selektorski izraz da bi se dobila vrednost selektora. Ova se vrednost zatim upoređuje sa vrednostima izbora da bi se odredio blok naredbi koji je potrebno izvršiti.

- *If* naredba predstavlja opštiji mehanizam za odabir između mogućnosti, jer uslovi mogu biti proizvoljno složeni Bulovi izrazi.

- Međutim, kao što će se videti, *case* naredbe predstavljaju važan i koristan mehenizam modelovanja.

CASE naredbe IV

- Selektorski izraz mora rezultovati vrednošću diskretnog tipa, ili jednodimenzionalnim nizom karaktera, na primer stringom karaktera ili nizom bitova. Tako da možemo imati *case* naredbu koja bira alternativu na osnovu celobrojne vrednosti. Ako su *index_mode* i *instruction_register* deklarisane kao

```
subtype index_mode is integer range 0 to 3;
variable instruction_register: integer range 0 to 2**16-1;
```

- možemo napisati *case* naredbu koja koristi vrednost ovog tipa:

```
case index_mode'((instruction_register/2**12) rem 2**2) is
   when 0 =>
      index_value := 0;
   when 1 =>
      index_value := accumulator_A;
   when 2 =>
      index_value := accumulator_B;
   when 3 =>
      index_value := index_register;
end case;
```

CASE naredbe V

- Još jedno pravilo koje treba zapamtiti je da tip svakog izbora mora biti isti kao i tip koji se dobija nakon evaluacije selektorskog izraza

- U gornjem primeru bilo bi pogrešno uključiti alternativu

```
when 'a' ⇒ ...       -- POGREŠNO!
```

- jer vrednost izbora nije tipa *integer.*

- Možemo uključiti više od jednog izbora u svakoj alternativi na taj način što ćemo izbore razdvojiti simbolom '|'. Na primer ako je tip *opcodes* deklarisan kao

```
type opcodes is (nop, add, substract, load, store, jump, jumpsub, branch, halt);
```

- mogli bi smo napisati alternativu koja bi sadržala tri vrednosti kao izbore:

```
when load|add|substract =>
   operand := memory_operand;
```

CASE naredbe VI

- Ako želimo da uključimo alternativu koja će uključiti sve vrednosti koje nisu obuhvaćene ostalim alternativama u *case* naredbi, možemo iskoristiti poseban izbor *others*

- Na primer, ako je promenljiva *opcode* promenljiva tipa *opcodes*, možemo pisati

```
case opcode is
    when load|add|substract =>
        operand := memory_operand;
    when store|jump|jumpsub|branch =>
        operand := address_operand;
    when others =>
        operand := 0;
end case;
```

- Ako je vrednost promenljive *opcode* bilo koja od vrednosti koje nisu obuhvaćene prethodnim alternativama, poslednja alternativa biva odabrana

CASE naredbe VII

- Preostali oblik izbora koji još nije pomenut je izbor diskretnog opsega, definisan kao

diskretni_opseg ⇐
*diskretni*_indikator_ podtipa
| prost_izraz **(to** | **downto)** prost_izraz

indikator_diskretnog_podtipa ⇐
oznaka_tipa
[**range** prost_izraz (**to** | **downto**)prost_izraz]

- Ovi oblici dozvoljavaju nam da definišemo opseg vrednosti u alternativi
- Ako je vrednost selektorskog izraza unutar tog opsega izvršava se blok naredbi koji pripada toj alternativi

CASE naredbe VIII

- Najjednostavniji način za definisanje opsega je da se napišu leva i desna granica opsega, razdvojene sa ključnom reči smera

- Na primer gornja *case* naredba može se napisati kao

```
case opcode is
        when  add to load =>
                operand := memory_operand;
        when branch downto store =>
                operand := address_operand;
        when others =>
                operand := 0;
end case;
```

CASE naredbe IX

- Drugi način za specifikaciju diskretnog opsega jeste da se iskoristi ime diskretnog tipa i eventualno *range* ograničenje da bi se suzio skup vrednosti na podtip tog tipa. Na primer ako deklarišemo podtip tipa *opcodes* kao

```
subtype control_transfer_opcodes is opcodes range jump to branch;
```

- drugu alternativu možemo zapisati kao

```
when control_transfer_opcodes | store =>
        operand := address_operand;
```

- Treba primetiti da se diskretni opseg kao izbor može koristiti samo ako je selektorski izraz diskretnog tipa

- Ne možemo koristiti opseg ako je selektorski izraz nizovnog tipa, npr. tipa *bit_vector* . Ako opseg specificiramo pišući granice i smer, smer nema ulogu osim one da odredi granice opsega.

CASE naredbe X

- Važna stvar u vezi izbora u *case* naredbi je da oni moraju biti zapisani koristeći **lokalno statičke** vrednosti. Ovo znači da vrednosti izbora moraju biti određene u fazi analize tokom obrade dizajna.

- Svi gornji primeri zadovoljavaju ovaj uslov, ali ako na primer imamo celobrojnu promenljivu N, deklarisanu kao

```
variable N: integer := 1;
```

- ako bi *case* naredbu napisali na sledeći način

```
case izraz is                          --  primer nepravilne case naredbe
        when N | N+1 => ...
        when N+2 to N+5 => ...
        when others => ...
end case;
```

- vrednosti izbora zavise od promenljive *N*. Obzirom da se ta vrednost može promeniti tokom izvršavanja programa, ovi izbori nisu lokalno statički.

CASE naredbe XI

- Međutim, ako bi deklarisali *C* kao konstantni ceo broj, na primer

```
constant C: integer := 1;
```

- onda bi sledeća *case* naredba bila legalna

```
case izraz is
        when C | C+1 => ...
        when C+2 to C+5 => ...
        when others => ...
end case;
```

- Analizirajući model možemo utvrditi da prva alternativa uključuje izbore 1 i 2, druga brojeve između 3 i 6, a treća preostale vrednosti izraza, pa je ova *case* naredba legalna

CASE naredbe XII

- Prethodni primeri prikazuju alternative koje sadrže po samo jednu naredbu koju treba izvršiti u slučaju da ta alternativa bude izabrana

- Kao i u slučaju *if* naredbe, možemo uključiti proizvoljan broj sekvencijalnih naredbi bilo kog tipa u svakoj alternativi

- Ovo omogućava pisanje ugnježdenih *case* naredbi, *if* naredbe ili bilo koji drugi oblik sekvencijalnih naredbi u alternativama

- U praksi potrebno je zapamtiti sledeće stvari kada se koriste *case* naredbe:
  - sve moguće vrednosti selektorskog izraza moraju biti pokrivene izborima vrednosti izbora moraju biti **lokalno statičke**, i
  - ako se koristi izbor *others* on se mora nalaziti u poslednjoj alternativi i mora biti jedini uslov u toj alternativi.

CASE naredbe XIII

- Napišimo bihevijalni model multipleksera se selektorskim ulazom *sel*; četiri ulaza podataka *d0*, *d1*, *d2* i *d3*; i izlazom *z*
- Ulazi i izlazi su tipa *IEEE standard-logic*, a selektorski ulaz je tipa *sel_range*, za koji pretpostavljamo da je deklarisan negde drugde kao

  type sel_range **is range** 0 **to** 3;
- Deklaracija entiteta koja definiše portove i arhitekturno telo prikazani su desno
- Arhitekturno telo sadrži samo deklaraciju procesa
- Obzirom da se izlaz multipleksera mora promeniti kada se promene selektorski ulazi ili ulazi podataka, proces mora biti osetljiv na sve ulaze. Proces koristi *case* naredbu da bi odabrao koji od ulaza podataka treba dodeliti izlazu.

```
library ieee; use ieee.std_logic_1164.all;
entity mux4 is
    port (sel: in sel_range; d0, d1, d2, d3: in std_logic;
          z: out std_logic);
end entity mux4;

architecture demo of mux4 is
begin
    out_select: process (sel, d0, d1, d2, d3) is
    begin
        case sel is
            when 0 =>
                z <= d0;
            when 1 =>
                z <= d1;
            when 2 =>
                z <= d2;
            when 3 =>
                z <= d3;
        end case;
    end process out_select;
end architecture demo;
```

NULL naredbe

NULL naredbe I

- Ponekad, prilikom pisanja modela, javlja se situacija da kada je neki uslov zadovoljen, ne treba preduzeti nikakvu akciju
- Ovaj slučaj često se javlja kada se koriste *case* naredbe, obzirom da moramo opisati situacije zasve moguće vrednosti selektorskog izraza
- Kada se u nekoj od alternativa javi slučaj da nema nikakve akcije, on se opisuje pomoću *null* naredbe
- Sintaksno pravilo za *null* naredbu ima oblik

null_naredba ⇐ [labela:] **null**;

- Opciona labela služi za označavanje naredbe

NULL naredbe II

- Primer korišćenja *null* naredbe u okviru *case* naredbe mogao bi biti:

```
case opcode is
    when add =>
        Acc := Acc+operand;

    when substract =>
        Acc := Acc-operand;

    when nop =>
        null;
end case;
```

NULL naredbe III

- *Null* naredbu možemo koristiti u bilo kojem mestu gde se zahteva neka sekvencijalna naredba
- *Null* naredba može se koristiti tokom razvojne faze modela
- Ako na primer znamo da će nam biti potreban neki entitet kao deo sistema, ali još nismo u mogućnosti da napišemo detaljan model za njega, možemo napisati model koji ne radi ništa
- Takav model sastoji se od procesa u čijem telu se nalazi *null* naredba:

```
control_section: process (lista_osetljivosti) is
begin
        null;
end process control_section;
```

LOOP naredbe

LOOP naredbe I

- Često se javlja potreba za pisanjem bloka naredbi čije izvršenje se ponavlja
- Da bi smo opisali takvu situaciju koristimo *loop* naredbe
- Postoji nekoliko različitih oblika *loop* naredbi u VHDL; najjednostavnija je petlja koja ponavlja blok naredbi beskonačan broj puta, koja se često zove beskonačna petlja
- Sintaksno pravilo za ovu vrstu petlje je

loop_naredba ⇐
[loop_labela:]
loop
{sekvencijalna_naredba}
end loop [loop_labela:];

LOOP naredbe II

- U većini programskih jezika, beskonačna petlja je nepoželjna, jer njeno korišćenje znači da se program nikada neće završiti

- Međutim, prilikom modelovanja digitalnih sistema, beskonačna petlja može biti korisna, jer veliki broj uređaja neprekidno ponavlja neku operaciju sve dok se ne isključi napajanje

- Uobičajeno je da model za takav uređaj uključuje *loop* naredbu u telu procesa; petlja u svom telu sadrži *wait* naredbu

LOOP naredbe III

- Desno je prikazan model brojača koji počinje brojanje od nule, i inkrementuje svoju vrednost na svakom prelazu takt signala sa 0 na 1
- Kada brojač stigne do 15, prilikom sledeće tranzicije takt signala vraća se na nulu
- Arhitekturno telo za brojač sadrži proces koji prvo inicijalizuje *count* izlaz na nulu, a zatim čeka na tranziciju takt signala pre nego što inkrementuje vrednost *count*

```
entity counter is
    port (clk: in bit; count: out natural);
end entity counter;

architecture behavior of counter is
begin
    incrementer: process is
        variable count_value: natural := 0;
    begin
        count <= count_value;
        loop
            wait until clk = '1';
            count_value := (count_value+1) mod 16;
            count <= count_value;
        end loop;
    end process incrementer;
end architectura behavior;
```

EXIT naredbe I

- Često se javlja potreba da se izađe iz petlje kada je neki uslov zadovoljen. Da bi smo izašli iz petlje koristimo *exit* naredbu. Sintaksno pravilo za *exit* naredbu glasi

  exit_naredba ⇐ [labela:] **exit** [loop_labela] [**when** Bulov_izraz];

- Opciona labela na početku *exit* naredbe služi za označavanje naredbe. Najjednostavniji oblik *exit* naredbe je

  exit;

- Kada se ova naredba izvrši, sve preostale naredbe unutar petlje se preskaču, a prelazi se na izvršavanje naredbe koja sledi posle ključnih reči *end loop*. Tako da unutar petlje možemo pisati

  if uslov **then**
  exit;
  end if;

- gde je uslov Bulov izraz.

EXIT naredbe II

- Obzirom da je ovo najčešći oblik korišćenja *exit* naredbe VHDL obezbeđuje skraćeni oblik za njegovo zapisivanje korišćenjem *when* člana

- Na primer, možemo napisati sledeće

loop

...

exit when *uslov*;

...

end loop;

... – izvršavanje se prenosi ovde kada je *uslov* zadovoljen

EXIT naredbe III

- Sada ćemo preraditi prethodni model brojača tako da on sadrži *reset* ulaz koji, kada ima vrednost ‘1’, postavlja izlaz *count* na nulu
- Izlaz ima vrednost nula sve dok je *reset* jednak ‘1’ i nastavalja sa brojanjem na sledećoj tranziciji takt signala nakon što se *reset* ulaz promeni na ‘0’
- Prerađena deklaracija entiteta, prikazana je desno, i sadrži novi ulazni port

```
entity counter is
   port (clk, reset: in bit; count: out natural);
end entity counter;

architecture behavior of counter is
begin
   incrementer: process is
      variable count_value: natural := 0;
   begin
      count <= count_value;
      loop
         loop
            wait until clk =’1’ or reset = ‘1’;
            exit when reset = ‘1’;
            count_value := (count_value+1) mod 16;
            count <= count_value;
         end loop;
         -- u ovoj tački, reset = ‘1’
         count_value := 0;
         count <= count_value;
         wait until reset = ‘0’;
      end loop;
   end process incrementer;
end architecture behavior;
```

EXIT naredbe IV

- Mogu se javiti slučajevi kada je potrebno preneti izvršavanje izvan unutrašnje petlje i izvan spoljašnje petlje (situacija sa ugnježdenim petljama)

- Ovo se može uraditi na taj način što će se označiti spoljašnja petlja labelom, i korišćenjem te labele u *exit* naredbi

- Na primer:

```
ime_petlje: loop
        ...
        exit ime_petlje;
        ...
end loop;
```

- Na ovaj način smo označili petlju labelom *ime_petlje*, pa možemo naznačiti koju petlju želimo da napustimo u *exit* naredbi

EXIT naredbe V

- Sledeći primer ilustruje izlaženje iz ugnježdenih petlji

```
spoljnja: loop
        unutrašnja: loop
                ...
                exit spoljnja when uslov_1;      -- exit 1
                ...
                exit when uslov_2;               -- exit 2
        end loop unutrašnja;
        ...                                      -- mesto A
        exit spoljnja when uslov_3;              -- exit 3
        ...
end loop spoljnja;
...                                              -- mesto B
```

NEXT naredbe I

- Druga vrsta naredbe koju možemo koristiti da bi smo kontrolisali izvršavanje petlji je *next* naredba
- Kada se ova naredba izvrši, tekuća iteracija u petlji se završava bez izvršavanja naredbi u telu petlje koje slede posle *next* naredbe i naredna iteracija započinje. Sintaksno pravilo je

  next_naredba ⇐ [labela:] **next** [loop_labela] [**when** Bulov_izraz];

- Najprostiji oblik *next* naredbe je: **next**;
- koji započinje narednu iteraciju petlje u čijem se telu nalazi. Možemo uključiti uslov koji mora biti ispunjen da bi se izvršila *next* naredba:

  next when uslov;

- a možemo i uključiti labelu petlje da naznačimo za koju petlju želimo da završimo iteraciju:

  next loop_labela;

- ili: **next** loop_labela **when** uslov;

NEXT naredbe II

- *Next* naredba koja završava iteraciju u petlji u čijem se telu nalazi može se zameniti sa odgovarajućom *if* naredbom. Na primer naredne dve petlje su ekvivalentne:

```
loop
    naredba_1;
    next when uslov;
    naredba_2;
end loop;
```

```
loop
    naredba_1;
    if not uslov then
        naredba_2;
    end if;
end loop;
```

- Međutim, ugnježdene petlje koje sadrže *next* naredbe koje se odnose na spoljnje petlje ne mogu se tako lako napisati korišćenjem nekih drugih naredbi

- Načelno, poželjno je izbegavati komplikovane *loop*/*next* strukture jer one mogu biti sbunjujuće, otežavajući čitanje i razumevanje modela

- Ako se nađemo u situaciji da ih moramo koristiti, potrebno je još jednom proveriti logiku ponašanja modela, jer je možda moguće pronaći jednostavniju formulaciju za njeno opisivanje.

WHILE petlje I

- Možemo proširiti osnovnu *loop* naredbu u formu *while* petlje, kod koje se testira neki uslov pre svake iteracije

- Ako je uslov zadovoljen izvršava se još jeda iteracija, a ako nije zadovoljen petlja se završava. Sintaksno pravilo za *while* petlju glasi

while_petlja ⇐
[loop_labela:]
while uslov **loop**
{sekvencijalna_naredba}
end loop [loop_labela];

- Jedina razlika između ove forme i osnovne forme je u tome da smo dodali ključnu reč *while* i uslov pre ključne reči *loop*

- Sve stvari koje su rečene u vezi osnovnog oblika *loop* naredbe važe i za *while* petlju.

- Možemo pisati bilo koje sekvencijalne naredbe u telu petlje uključujući i *exit* i *next* naredbe, i možemo označiti petlju pišući labelu ispred ključne reči *while*

WHILE petlje II

- Postoje tri bitne tvari u vezi *while* petlji

- Prva je da se uslov testira pre svake iteracije, uključujući i prvu iteraciju

- Ovo znači da ako uslov nije zadovoljen pre početka izvršavanja petlje, petlja se neće izvršiti. Na primer, ako imamo *while* petlju

```
while index > 0 loop
    ...        -- naredba_A: radi nešto sa promenljivom index
end loop;
...            -- naredba_B
```

- Ako je vrednost promenljive *index* manja ili jednaka nuli pre nego što se petlja započne, onda znamo da se *naredba_A* unutar petlje neće ni jednom izvršiti, već će se izvršavanje odmah preneti na *naredbu_B*

WHILE petlje III

- Druga stvar je da ukoliko nemamo *exit* naredbe unutar *while* petlje, petlja se završava jedino kada uslov prestane da bude ispunjen
- Slično, u odsustvu *next* naredbe, iteracija se izvršava samo kada je uslov zadovoljen, tako da znamo da je uslov zadovoljen kada započinjemo novu iteraciju
- U gornjem primeru, znamo da *index* mora biti veći od nule kada izvršavamo *naredbu_A*, i takođe da je *index* manji ili jednak nuli kada stignemo do *naredbe_B*
- Ovo znanje nam može biti od koristi kada proveravamo ispravnost modela koji pišemo
- Treća stvar je da kada pišemo naredbe unutar tela *while* petlje, moramo obezbediti da uslov u jednom trenutku prestane da bude ispunjen ili da *exit* naredbom završimo petlju
- U protivnom *while* petlja se nikad neće završiti. Da smo hteli koristiti beskonačnu petlju, iskoristili bi smo jednostavniju formu *loop* naredbe.

WHILE petlje IV

- Možemo razviti model za entitet *cos* koji se može koristiti kao deo specijalizovanog sistema za obradu signala
- Entitet ima jedan ulaz, *theta*, koji je realni broj koji predstavlja ugao izražen u radijanima, i jedan izlaz, *result*, koji predstavlja vrednost kosinusne funkcije za ugao *theta*
- Za izračunavanje kosinusne funkcije koristićemo razvoj u red

$$\cos\vartheta = 1 - \frac{\vartheta^2}{2!} + \frac{\vartheta^4}{4!} - \frac{\vartheta^6}{6!} + \dots$$

- dodajući sukcesivne članove reda sve dok članovi ne postanu manji od milionitog dela konačnog rezultata

```
entity  cos is
   port (theta: in real; result: out real);
end entity cos;

architecture series of cos is
begin
   summation: process (theta) is
      variable sum, term: real;
      variable n: natural;
   begin
      sum := 1.0;
      term := 1.0;
      n := 0;
         while abs term > abs (sum/1.0E6) loop
            n := n+2;
            term := (-term)*theta**2/real(((n-1)*n));
            sum := sum+term;
         end loop;
      result <= sum;
   end process summation;
end architecture series;
```

FOR petlje I

- Drugi način na koji možemo proširiti osnovnu *loop* naredbu jeste *for* petlja.

- *For* petlja uključuje specifikaciju koliko puta treba izvršiti telo petlje. Sintaksno pravilo za *for* petlju je

  for_petlja ⇐
  [loop_labela:]
  for identifikator **in** diskretni_opseg **loop**
  {sekvencijalna_naredba}
  end loop [loop_labela];

- Već smo videli da *diskretni_opseg* ima formu

  prost_izraz **(to** | **downto)** prost_izraz

- predstavljajući sve vrednosti između leve i desne granice, uključujući i njih. Identifikator se naziva *parametar petlje*, i za svaku iteraciju uzima narednu vrednost iz diskretnog opsega, počinjući sa levim elementom.

FOR petlje II

- Na primer

```
for  count_value in 0 to 127 loop
    count_out <= count_value;
    wait for 5 ns;
end loop;
```

- identifikator *count_value* uzima vrednosti 0, 1, 2, itd. , i za svaku vrednost izvršavaju se naredba dodele i *wait* naredba. Znači signal count_out dobijaće vrednosti 0, 1, 2, sve do 127 u intervalima od po 5 ns.
- Videli smo da se diskretni opseg može specificirati korišćenjem diskretnog tipa ili imena podtipa koji je možda ograničen na podopseg vrednosti. Na primer, ako imamo nabrojivi tip

```
type controller_state is (initial, idle, active, error);
```

- možemo napisati petlju koja prolazi kroz svaku od mogućih vrednosti tog tipa

```
for state in controller_state loop
        ...
end loop;
```

FOR petlje III

- Unutar tela petlje, parametar petlje predstavlja konstantu čiji je tip osnovni tip diskretnog opsega
- Ovo znači da možemo koristiti njegovu vrednost u izrazima, ali ne možemo mu dodeljivati vrednost
- Za razliku od drugih konstanti ne moramo ga deklarisati
- Parametar petlje je implicitno deklarisan za posmatranu petlju
- On postoji samo kada se petlja izvršava, i ne postoji pre ili posle toga
- Na primer, sledeći proces ilustruje kako se ne može koristiti parametar petlje

```
erroneous: process is
    variable i, j: integer;
begin
    i:= loop_param;              -- GREŠKA!
    for loop_param in 1 to 10 loop
        loop_param := 5;         -- GREŠKA!
    end loop;
    j := loop_param;             -- GREŠKA!
end process erroneous;
```

FOR petlje IV

- Zbog načina na koji je definisan, parametar petlje unutar petlje sakriva bilo koji objekat sa istim imenom koji je definisan izvan petlje. Na primer, u procesu:

```
hiding_example: process is
   variable a, b: integer;
begin
   a := 10;
   for a in 0 to 7 loop
      b := a;
   end loop;
   --  a = 10 i b = 7
   ...
end process hiding_example;
```

- Promenljivoj *a* je na početku dodeljena vrednost 10, a zatim se počinje sa izvršavanjem petlje, pri čemu se kreira parametar petlje sa istom oznakom, *a*
- Unutar petlje dodela promenljivoj *b* koristi parametar petlje, tako da je krajnja vrednost *b* jednaka 7
- Posle petlje parametar petlje više ne postoji, tako da ako bi smo koristili ime *a*, ono bi se odnosilo na promenljivu *a* čija je vrednost 10

FOR petlje V

- Kao što je rečeno, petlja pravi iteraciju pri čemu parametar petlje uzima uzastopne vrednosti iz diskretnog opsega počinjući sa krajnjim levim elementom
- Bitna stvar koju treba naglasiti je da ako specificiramo *nulti opseg*, telo petlje neće se uopšte izvršiti
- Nulti opseg može se javiti ako specificiramo rastući opseg sa levom granicom koja je veća od desne, ili opadajući opseg sa levom granicom koja je manja od desne. Na primer, petlja

```
for i in 10 to 1 loop
   ...
end loop;
```

- završava se odmah, bez izvršavanja tela petlje. Ako stvarno želimo da se petlja izvršava, a da *i* uzima vrednosti 10, 9, 8, itd. , treba napisati sledeće

```
for i in 10 downto 1 loop
   ...
end loop;
```

- Poslednja stvar koju treba istaći u vezi *for* petlji, je da kao i osnovne *loop* petlje, i one mogu sadržati proizvoljne sekvencijalne naredbe uključujući i *exit* i *next* naredbe, i da je moguće označavanje *for* petlji labelom pre ključne reči *for*.

FOR petlje VI

- Možemo preraditi *cos* model sa slajda 44 da računa rezultat sumirajući prvih 10 članova razvoja u red
- Deklaracija entiteta ostaje ista, a prerađeno arhitekturno telo, prikazano desno, sadrži proces koji koristi *for* petlju umesto *while* petlje
- Kao i ranije, promenljive *sum* i *term* na početku se postavljaju na 1.0, predstavljajući prvi član razvoja
- Promenljiva *n* je zamenjena sa parametrom *for* petlje
- Petlja se ponavlja devet puta, računajući preostalih devet članova reda

```
architecture fixed_length_series of cos is
begin
    summation: process (theta) is
        variable sum, theta: real;
    begin
        sum := 1.0;
        term := 1.0;
            for n in 1 to 9 loop
                term := (-term)*theta**2/real(((2*n-1*2*n));
                sum := sum+term;
            end loop;
        result <= sum;
    end process sumation;
end architecture fixed_length_series;
```

Pregled LOOP naredbi I

- U prethodnim poglavljima opisane su različite forme *loop* naredbi
- Na kraju ćemo dati ukupni pregled *loop* naredbi, da bi smo naglasili par osnovnih stvari koje vredi zapamtiti
- Kao prvo, sintaksno pravilo sa sve vrste *loop* naredbi glasi

loop_naredba ⇐
[loop_labela:]
[**while** uslov | **for** identifikator **in** diksretni_opseg] **loop**
{sekvencijalna_naredba}
end loop [loop_labela];

- Drugo, u odsustvu *exit* i *next* naredbi, *while* petlja se izvršava sve dok je uslov zadovoljen, a *for* petlja se izvršava sve dok parametar petlje ne uzme sve vrednosti iz diskretnog opsega
- Ako je uslov u *while* petlji inicijalno neispunjen, ili diskretni opseg u *for* petlji predstavlja nulti opseg, ne izvršava se niti jedna iteracija

Pregled LOOP naredbi II

- Treće, parametar petlje u *for* petlji ne može se eksplicitno deklarisati i on je konstanta unutar tela petlje
- Takođe on u telu petlje skriva bilo koji drugi objekat sa istim imenom koji je deklarisan izvan petlje
- Konačno, *exit* naredba može se iskoristiti za prekidanje svake petlje, a *next* naredba se može iskoristiti za završavanje tekuće iteracije i započinjanje naredne
- Ove dve naredbe mogu se sadržati labele petlji da bi prekinule ili završile iteraciju neke spoljašnje petlje u ugnježdenim petljama

ASSERT i REPORT naredbe

ASSERT i REPORT naredbe I

- Jedan od razloga zašto se pišu modeli jeste da se verifikuje da dizajn ispravno funkcioniše
- Možemo delimično testirati model korišćenjem test signala na ulazima i proveravajući da li izlazi imaju željene vrednosti
- Ako to nije slučaj, onda se suočavamo sa problemom određivanja mesta greške unutar dizajna
- Ovaj zadatak može se olakšati korišćenjem *assertion* naredbi koje proveravaju da li su uslovi od interesa zadovoljeni unutar modela
- *Assert* naredba je sekvencijalna naredba, što znači da se može pojaviti bilo gde je dozvoljeno pojavljivanje sekvencijalnih naredbi. Potpuno sintaksno pravilo za *assert* naredbu je

assert_naredba ⇐

[labela:] **assert** bulov_izraz [**report** izraz][**severity** izraz];

ASSERT i REPORT naredbe II

- Najjednostavniji oblik *assert* naredbe uključuje ključnu reč *assert* koja je praćena uslovom za koji očekujemo da je ispunjen kada se naredba bude izvršavala
- Ako uslov nije ispunjen, kažemo da se desio *assert prekršaj*
- Ako se to desi tokom simulacije, simulator nam to prijavljuje
- Tokom sinteze, uslov u *assertion* naredbi može biti protumačen kao uslov za koji sintetizator (synthesizer) može pretpostaviti da je zadovoljen

ASSERT i REPORT naredbe III

- Tokom formalne verifikacije, uslov može biti protumačen kao uslov koji verifikator treba da dokaže

- Na primer, ako napišemo

assert initial_value <= max_value;

- i *intial_value* je veći od *max_value*, kada se naredba izvrši tokom simulacije, simulator će nas obavestiti da je nastupio prekršaj.

- Tokom sinteze, sintetizator može pretpostaviti da je navedeni uslov zadovoljen, i shodno tome izvršiti optimizaciju

- Tokom formalne verifikacije, verifikator može pokušati da dokaže da je *initial_value* <= *max_value* za sve moguće ulazne stimuluse i puteve izvršavanja koji vode do *assert* naredbe

ASSERT i REPORT naredbe IV

- Ako imamo nekoliko *assert* naredbi u modelu, korisno je znati koja od njih je prekršena
- Možemo od simulatora dobiti dodatnu informaciju uključujući *report* član u *assert* naredbu, na primer:

```
assert initial_value <= max_value report "initial value too large";
```

- String koji smo uključili koristi se prilikom formiranja poruke o narušavanju *assert* naredbe
- Možemo da napišemo bilo koji izraz u *report* delu, pod uslovom da je njegov rezultat string, na primer:

```
assert current_character >= ‘0’ and current_character <= ‘9’
    report "Input number" & input_string & "contains a non-digit";
```

- Ovde se poruka dobija spajanjem tri stringa

ASSERT i REPORT naredbe V

- Predefinisani nabrojivi tip *severity_level* definisan je kao:

```
type severity_level is (note, warning, error, failure);
```

- Možemo uključiti vrednost ovog tipa u *severity* član *assert* naredbe

- Ova vrednost pokazuje stepen ozbiljnosti greške

- Vrednost *note* može se iskoristiti za prenos informativne poruke od strane simulatora, na primer:

```
assert free_memory >= low_water_limit
    report "Low on memory, about to start garbage collect"
        severity note;
```

ASSERT i REPORT naredbe VI

- *Severity* nivo *warning* može se iskoristiti ako se desi neka neuobičajena situacija u kojoj model može nastaviti da se izvršava, ali može proizvesti neočekivane rezultate, na primer:

```
assert packet_length /= 0
    report "Empty network packet received"
        severity warning;
```

- *Severity* nivo *error* može se koristiti da ukaže da je nešto definitivno pošlo loše i da je neophodno sprovesti korektivne akcije, na primer:

```
assert clock_pulse_width >= min_clock_width
    severity error;
```

- Konačno, *severity* nivo *failure* može se koristiti ako se otkrije nedoslednost koja nikad ne bi smela da se pojavi, na primer:

```
assert (last_positio-first_position+1) = number_of_entries
    report "Inconsistency in buffer model"
        severity failure;
```

ASSERT i REPORT naredbe VII

- Videli smo da *assert* naredbu možemo pisati sa jednim ili oba od *report* i *severity* delova
- Ako su prisutna oba dela, *report* deo dolazi prvi
- Ako izostavimo *report* deo, podrazumevani string u poruci greške je "Assertion violation."
- Ako izostavimo *severity* deo, podrazumevana vrednost je *error*
- Ova vrednost se najčešće koristi od strane simulatora da se odredi da li treba nastaviti sa simulacijom nakom narušavanja assertion naredbe
- Većina simulatora dozvoljava korisniku da odredi *severity* nivo nakon kojeg se simlacija prekida
- Obično, narušavanje *assert* naredbe znači da se entitet koristi neispravno kao deo većeg dizajna, ili da je model entiteta pogrešno napisan. Ilustrovaćemo oba slučaja.

ASSERT i REPORT naredbe VIII

- Set/reset (SR) flipflop ima dva ulaza, *S* i *R*, i jedan izlaz *Q*
- Kada je *S* jednak ‘1’, izlaz uzima vrednost ‘1’, a kada je *R* jednak ‘1’, izlaz uzima vrednost ‘0’.
- Ulazi *S* i *R* ne smeju istovremeno biti jednaki ‘1’. Ako se to desi, izlazna vrednost se ne može specificirati.
- Desno je prikazan bihevijalni model SR flipflopa koji sadrži proveru za taj nedozvoljeni slučaj.
- Arhitekturno telo sadrži proces osetljiv na *S* i *R* ulaze
- Unutar procesa nalazi se *assert* naredba koja zahteva da *S* i *R* ne budu istovremo jednaki ‘1’
- Ako su *S* i *R* istovremeno jednaki ‘1’, simulator prikazuje poruku o grešci sa nivoom *error*

```
entity SR_flipflop is
    port (S, R: in bit; Q: out bit);
end entity SR_flipflop;

architecture checking of SR_flipflop is
begin
    set_reset: process (S, R) is
    begin
        assert S = ‘1’ nand R = ‘1’;
        if S = ‘1’ then
            Q <= ‘1’;
        end if;
        if R = ‘1’ then
            Q <= ‘0’;
        end if;
    end process set_reset;
end architecture checking;
```

ASSERT i REPORT naredbe IX

- Da bi smo ilustrovali korišćenje *assert* naredbe kao načina za proveru korektnosti napisanog modela, pogledajmo model prikazan desno, za entitet koji ima tri celobrojna ulaza, *a*, *b* i *c*, i jedan izlaz *z*, koji ima vrednost najvećeg ulaza
- Arhitekturno telo sadrži proces koji pak sadrži ugnježdene *if* naredbe
- U ovom primeru uveli smo 'slučajnu' grešku u modelu. Ako simuliramo ovaj model i postavimo vrednosti *a* = 7, *b* = 3 i *c* = 9 na portove entiteta, očekujemo da će vrednost promenljive *result* i izlaznog porta biti 9.
- *Assert* naredba tvrdi da vrednost promenljive *result* mora biti veća ili jednaka od svih ulaza.
- Međutim, naša greška u kodiranju dovodi do toga da će vrednost 7 biti dodeljena promenljivoj *result*, pa će *assert* biti narušen
- Ovo bi nas pak navelo da pažljivije proverimo model, i ispravimo grešku.

```
entity max3 is
   port (a, b, c: in integer; z: out integer);
end entity max3;

architecture check_error of max3 is
begin
   maximizer: process (a, b, c)
      variable result: integer;
   begin
      if a > b then
         if a > c then
            result := a;
         else
            reslut := a; -- Greška! Trebalo bi da bude result := c;
         end if;
      elseif b >c then
          result := b;
      else
         result := c:
      end if;
      assert result >= a and result >= b and  result >= c
         report “nekonzistentan rezultat za maksimum”
            severity failure;
      z <= result;
   end process maximizer;
end architecture check_error;
```

ASSERT i REPORT naredbe X

- Još jedno mesto gde se *assert* naredba može korisno upotrebiti, jeste u proveri vremenskih ograničenja koja model mora zadovoljiti

- Na primer, većina sinhronih uređaja zahteva da trajanje taktnog impulsa bude duže od nekog minimuma

- Da bi smo izračunali trajanje impulsa možemo koristiti predefinisanu funkciju *now*

ASSERT i REPORT naredbe XI

- Registar okidan ivicom ima ulaze i izlaze podataka koji su tipa *real* i takt ulaz tipa *bit*
- Kada se takt signal promeni sa ‘0’ na ‘1’, ulaz podataka se proverava, a očitana vrednost smešta i prenosi se kroz izlaz
- Pretpostavimo da takt ulaz mora biti ‘1’ barem 5 ns.
- Desno je prikazam model ovog registra uključujući i proveru širine taktnog impulsa
- Arhitekturno telo sadrži proces koji je osetljiv na promene taktnog signala
- Kada se taktni signal promeni sa ‘0’ na ‘1’, ulazi se smeštaju, i trenutno simulaciono vreme se pamti u promenljivoj *pulse_start*
- Kada se taktni signal promeni sa ‘1’ na ‘0’, razlika između *pulse_start* i tekućeg simulacionog vremena proverava se pomoću *assert* naredbe

```
entity edge_triggered_register is
   port (clock: in bit; d_in: in real; d_out: out real);
end entity edge_treiggered_register;

architecture check_timing of edge_triggered_register is
begin
   store_and_check: process (clock) is
      variable stored_value: real;
      variable pulse_start: time;
   begin
      case clock is
        when ‘1’ =>
            pulse_start := now;
            stored_value := d_in;
            d_out := stored_value;
         when ‘0’ =>
            assert now = 0 ns or (now-pulse_start) >= 5 ns
               report “clock pulse too short”;
      end case;
   end process store_and_check;
end architecture check_timing;
```

ASSERT i REPORT naredbe XII

- VHDL obezbeđuje i *report* naredbu, koja je slična *assert* naredbi

- Sintaksno pravilo za *report* naredbu je

  report_naredba ⇐

  [labela:] **report** izraz [**severity** izraz];

- Razlike su u tome što nema uslova i ako *severity* nivo nije specificiran, podrazumevan nivo je *note*

- *Report* naredba može se posmatrati kao *assertion* naredba kod koje je uslov uvek nezadovoljen, a *severity* nivo je *note*

- *Report* naredba je korisna kao sredstvo praćenja toka izvršavanja programa prilikom otklanjanja grešaka u programu

ASSERT i REPORT naredbe XIII

- Pretpostavimo da pišemo složeni model i nismo sigurni da smo ispravno shvatili logiku modela

- Možemo koristiti *report* naredbe da bi procesi unutar modela ispisivali poruke, na osnovu kojih možemo videti kada su koji procesi aktivni i šta rade

- Na primer

```
transmit_element: process (transmit_data) is
        ...  – deklaracije promenljvih
begin
        report “transmit_element: data =”
                & data_type’image (transmit_data);
        ...
end process transmit_element;
```