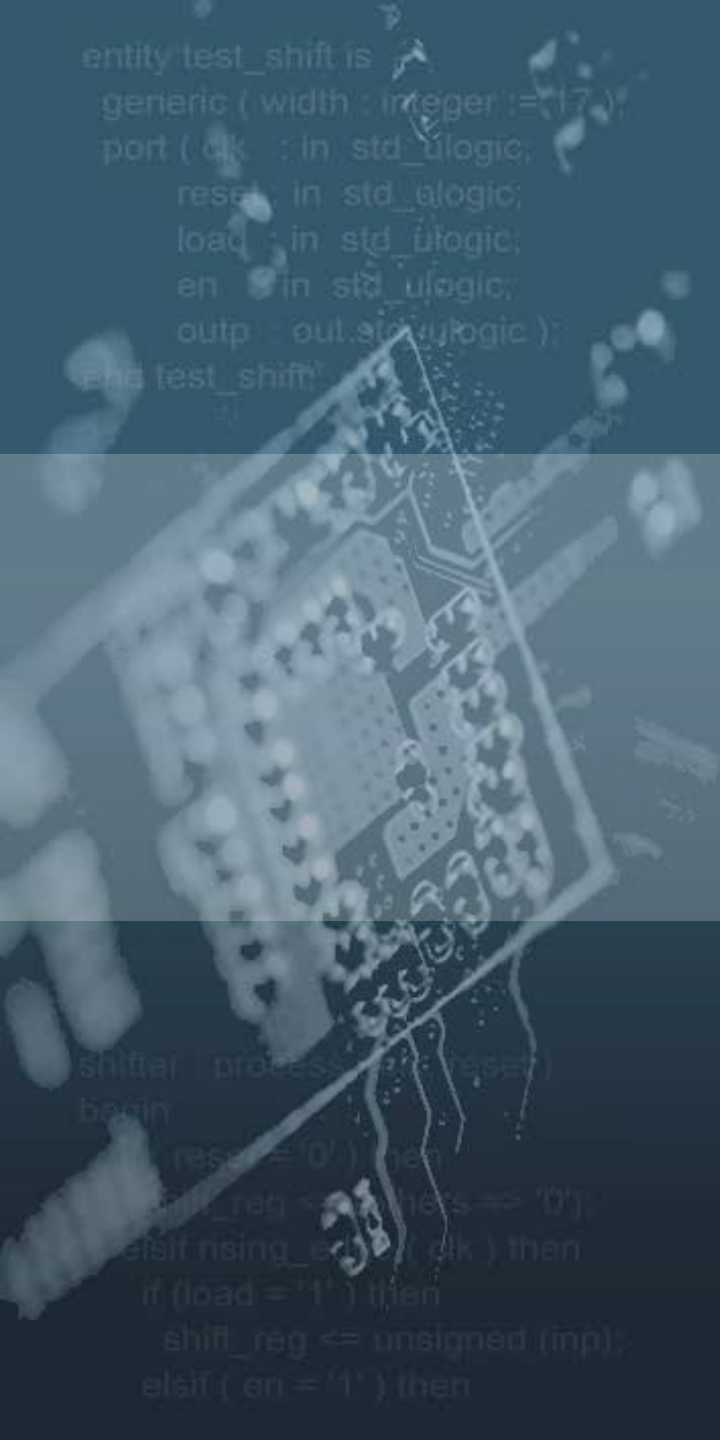
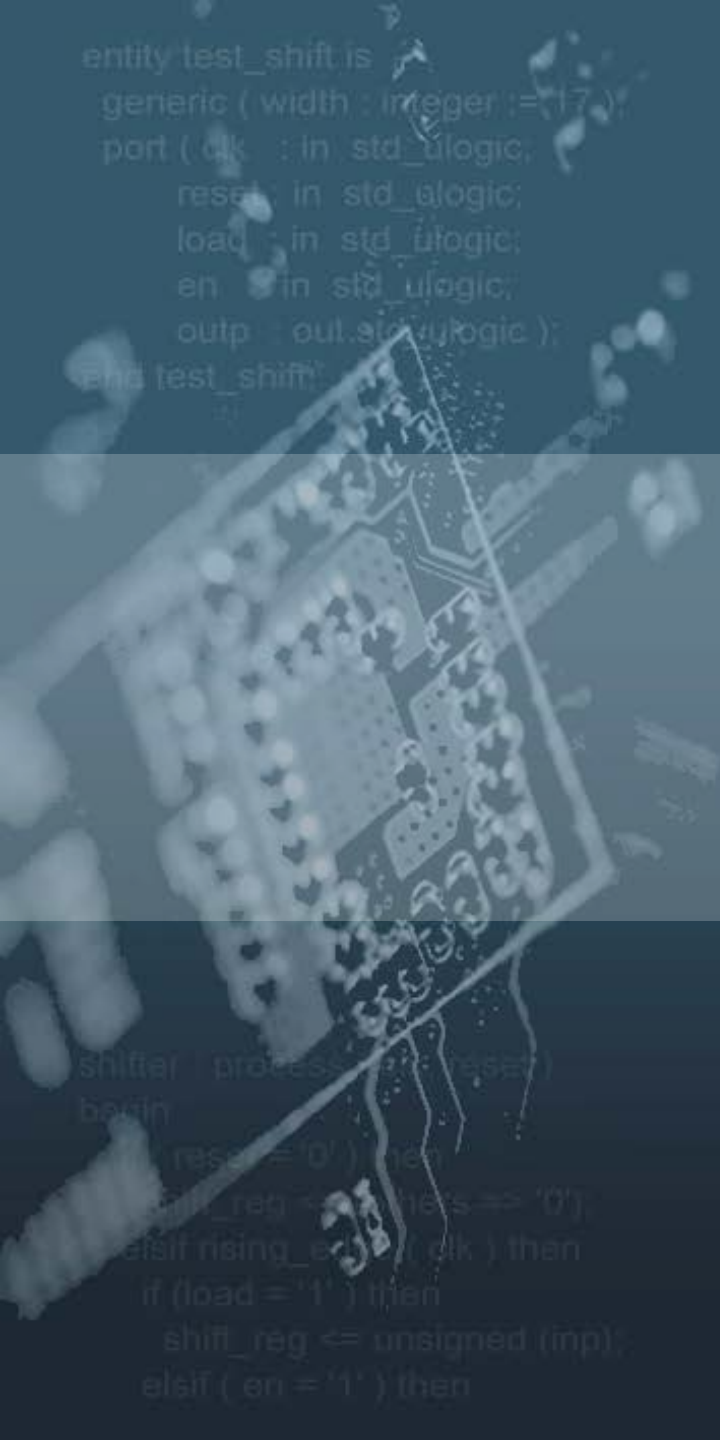


Uvod u mikroračunarsku elektroniku

Predavanje III

Sadržaj predavanja

- Naredbe dodele vrednosti signalu
- Konkurentna ASSERT naredba
- PROCESS naredba



Naredbe dodele vrednosti signalu

Naredba dodele vrednosti signalu I

- U svim dosadašnjim primerima, koristili smo jednostavnu formu naredbe dodele vrednosti signalu
- Ono o čemu još nije bilo reči jeste pitanje tajminga: kada signal uzima novu vrednost? Ovo je osnovno u modelovanju hardvera, jer se događaji odvijaju tokom vremena.
- Sintaksno pravilo za osnovnu naredbu dodele vrednosti signalu

  naredba_dodele_vrednosti_signalu ⇐
  [labela:] ime ⇐ [mehanizam_kašnjenja] talasni_oblik;
  talasni_oblik ⇐ (vrednosni_izraz [**after** vremenski_izraz]){, . . .}

- Vidimo da možemo definisati jedan ili više talasnih oblika, koji se sastoje od nove vrednosti i opcionog vremenskog kašnjenja
- Upravo ovo vremensko kašnjenje nam dozvoljava da tačno odredimo kada nova vrednost treba da se dodeli signalu

Naredba dodele vrednosti signalu II

- Na primer:

$$y \Leftarrow \textbf{not}\ \ or\_a\_b\ \textbf{after}\ 5\ ns;$$

- Ovim je određeno da signal *y* uzme novu vrednost 5 ns nakon trenutka u kome se izvršava naredba

- Kašnjenje se može tumačiti na dva načina, u zavisnosti od toga da li se model koristi čisto zbog svoje opisne vrednosti ili za simulaciju:
  - U prvom slučaju, kašnjenje se može shvatiti u apstraktnom smislu kao specifikacija propagacionog kašnjenja modula; svaki put kada se ulaz promeni, izlaz se menja 5 ns kasnije.
  - U drugom slučaju, kašnjenje se može shvatiti u operacionom smislu, u odnosu na mašinu koja simulira rad modula izvršavajući model. Tako, ako se gornja dodela izvrši u vremenu 250 ns, i *or_a_b* ima vrednost '1' u tom trenutku, tada će signal *y* uzeti vrednost '0' u trenutku 255 ns. Treba obratiti pažnju da se sama naredba dodele izvršava u *nultom modelovanom vremenu*.

Naredba dodele vrednosti signalu III

- Vreme na koje se misli kada se model izvršava je simulaciono vreme, odnosno, vreme u kojem je kolo koje se modeluje određeno da radi
- Ono je različito od realnog vemena izvršavanja na mašini na kojoj se vrši simulacija
- Simulaciono vreme merimo počevši od nule na početku simulacije i povećavamo ga u diskretnim koracima kako se događaji dešavaju unutar modela
- Ova tehnika se zove *simulacija diskretnog događaja* (discrete event simulation)
- Simulator diskretnog događaja mora imati simulacioni sat, i svaki put kada se izvrši naredba dodele, zahtevano kašnjenje se dodaje na tekuće simulaciono vreme da bi se odredilo kada nova vrednost treba da se dodeli signalu

Naredba dodele vrednosti signalu IV

- Kažemo da naredba dodele *planira transakciju* za signal, pri čemu se transakcija sastoji iz nove vrednosti i simulacionog vremena kada ta vrednost treba da se primeni na signal
- Kada simulaciono vreme dostigne vremenski ternutak u kojem postoji zakazana transakcija, signal se ažurira sa novom vrednošću
- Kažemo da ja signal *aktivan* tokom simulacionog ciklusa
- Ako je nova vrednost različita od stare vrednosti koju treba da zameni, kažemo da se desio *događaj* na posmatranom signalu
- Važnost ove razlike je u tome da procesi reaguju na događaje, a ne na transakcije

Naredba dodele vrednosti signalu V

- Sintaksno pravilo pokazuje da možemo planirati veći broj transakcija za signal, koje će biti primenjene sa različitim kašnjenjima
- Na primer, proces za drajver takt signala mogao bi izvršiti sledeću dodelu da generiše naredne dve ivice takt signala (pretpostavimo da je *T_pw* konstanta koja određuje širinu impulsa takta):

clk ⇐ '1' **after** T_pw, '0' **after** 2*T_pw;

- Ako se ova naredba izvrši u simulacionom vremenu 50 ns i *T_pw* ima vrednost 10 ns, jedna transakcija će biti planirana za vreme 60 ns kada treba signalu *clk* dodeliti vrednost '1', a druga za veme 70 ns kada signalu *clk* treba dodeliti vrednost '0'
- Ako pretpostavimo da *clk* ima vrednost '0' u trenutku kada se izvršava naredba, obe transakcije rezultuju sa događajima na signalu *clk*

Naredba dodele vrednosti signalu VI

- Možemo napisati deklaraciju procesa za takt generator koriteći gornju naredbu dodele vrednosti signalu da generišemo simetrični takt signal sa širinom impulsa *T_pw*

- Teškoća koju treba rešiti jeste da nateramo proces da se izvršava redovno svakog takt ciklusa

- Jedan način da to postignemo jeste da aktiviramo proces svaki put kada se desi promena na takt signalu, i planiramo naredne dve transakcije kada se takt signal promeni na vrednost '0':

```
clock_gen: process (clk) is
begin
        if clk = ‘0’ then
                clk <= ‘1’ after T_pw, ‘0’ after 2*T_pw;
        end if;
end process clock_gen;
```

Naredba dodele vrednosti signalu VII

- Obzirom da je proces osnovna jedinica za opis načina rada, ima smisla dozvoliti više od jedne naredbe za dodelu vrednosti za dati signal unutar procesa
- O ovome možemo misliti kao o opisu različitih načina na koje vrednost za signal može biti generisana od strane procesa u različitim trenucima
- Možemo napisati proces koji modeluje dvoulazni multiplekser kao na slici desno
- Vrednost *sel* porta koristi se za selekciju naredbe dodele vrednosti signalu koju treba izvršiti da bi se dobila izlazna vrednost

```
mux: process (a, b, sel) is
begin
    case sel is
        when ‘0’ =>
            z <= a after prop_delay;
        when ‘1’ =>
            z <= b after prop_delay;
    end case;
end process mux;
```

Naredba dodele vrednosti signalu VIII

- Kažemo da proces definiše *drajver* za signal ako i samo ako sadrži barem jednu naredbu dodele vrednosti za taj signal
- Tako da prethodni primer definiše drajver za signal *z*
- Ako proces sadrži naredbe dodele za nekoliko signala, on definiše drajvere za svaki od njih
- Drajver obezbeđuje vrednost koja treba da se primeni na signal. Važna stvar koju treba zapamtiti ja da za normalne signale, može postojati samo jedan drajver.
- Ovo znači da ne možemo napisati dva različita procesa koja oba sadrže naredbe dodele za isti signal. Ako želimo da modelujemo takve situacije kao na primer magistrale podataka ili 'ožičene ili' signale, moramo koristiti posebnu vrstu signala koji se zovu razrešeni signali.

Atributi signala I

- Pomoću atributa signala možemo dobiti informaciju o njihovoj istoriji transakcija i događaja. Ako imamo signal S i vrednost T tipa *time*, VHDL definiše sledeće atribute:
  - S'delayed(T) Signal koji uzima iste vrednosti kao i S ali je zakašnjen za vreme T
  - S'stable(T) Signal tipa *boolean* koji ima vrednost *true* ako nije bilo transakcija na S u vremenskom intervalu T, inače *false*
  - S'transaction Signal tipa *bit* koji menja vrednost sa '0' na '1' ili obrnuto svaki put kada postoji transakcija na S
  - S'event *True* ako postoji događaj na S u tekućem simulacionom ciklusu, inače *false*
  - S'active *True* ako postoji transakcija na S u tekućem simulacionom ciklusu, inače *false*
  - S'last_event Vremenski interval od poslednjeg događaja na S
  - S'last_active Vremenski interval od poslednje transakcije na S
  - S'last_value Vrednost S neposredno pre poslednjeg događaja na S

Atributi signala II

- Prva dva atributa mogu imati opcioni parametar tipa *time*. Ako ga izostavimo, pretpostavlja se da ima vrednost 0 fs. Ovi atributi se najčešće koriste u proveri tajminga unutar modela.

- Na primer, možemo proveriti da li signal *d* zadovoljava minimalno vreme uspostavljanja, *Tsu*, pre rastuće ivice na takt signalu *clk* tipa std_ulogic:

```
if clk'event and (clk = '1' or clk = 'H') and (clk'last_value = '0' or clk'last_value = 'L') then
        assert d'last_event >= Tsu
                report "Timing error: d changed within setup time of clk";
end if;
```

- Slično, možemo proveriti da li širina impulsa takt signala nije manja od minimalno dozvoljene, *Tpw_clk*:

```
assert (not clk'event) or clk'delayed'last'event >= Tpw_clk
        report "Clock frequency too high";
```

Atributi signala III

- Možemo iskoristiti sličan test za rastuću ivicu takt signala da modelujemo ivicom okidan modul, kao na primer flipflop
- Flipflop bi trebalo da smesti vrednost sa svog *D* ulaza na rastućoj ivici takt signala *clk*, ali da asinhrono postavi izlaz na nulu kada je signal *clr* jednak '1'
- Deklaracija entiteta i bihevijalno arhitekturno telo prikazani su desno

```
entity edge_trigg_Dff is
    port (D: in bit; clk: in bit; clr: in bit; Q: out
          bit);
end entity edge_trigg_Dff;

architecture behavioral of edge_trigg_Dff is
begin
    state_change: process (clk, clr) is
    begin
        if clr = '1' then
            Q <= '0' after 2 ns;
        elsif clk'event and clk = '1' then
            Q <= D after 2 ns;
        end if;
    end process state_change;
end architecture behavioral;
```

Konkurentne naredbe za dodelu vrednosti signalu

- VHDL obezbeđuje korisne skraćene notacije za *funkcionalno* modelovanje, odnosno modelovanje ponašanja u kojem je operacija koja se opisuje jednostavna kombinaciona (combinatorial) transformacija ulaza u izlaz

- Za razliku od običnih naredbi za dodelu vrednosti signalu, konkurentne naredbe dodele vrednosti signalu mogu biti uključene u arhitekturno telo

- Sintaksno pravilo glasi

konkurentna_dodela_vrednosti_signalu ⇐
[labela:] uslovna_dodela_vrednosti_signalu
|[labela:] selektovana_dodela_vrednosti_signalu

- Iz pravila vidimo da postoje dva oblika ove naredbe

Naredba uslovne dodele vrednosti signalu I

- Ova naredba predstavlja skraćeni zapis za kolekciju običnih naredbi dodele vrednosti signalu koje se nalaze u *if* naredbi, koja se opet nalazi u *process* naredbi

- Uprošćeno sintaksno pravilo za ovu naredbu je

naredba_uslovne_dodele_vrednosti_signalu ⇐
ime ⇐ [mehanizam_kašnjenja]
{talasni_oblik **when** bulov_izraz **else**}
talasni_oblik[**when** bulov_izraz];

- Naredba nam omogućava da odredimo koji od talasnih oblika treba da bude dodeljen signalu u zavisnosti od nekih uslova

Naredba uslovne dodele vrednosti signalu II

- Gornja naredba prikazana desno je funkcionalni opis multipleksera sa četiri ulaza podataka, *d0*, *d1*, *d2*, *d3* i dva kontrolna ulaza, *sel0* i *sel1* i izlazom *z*
- Svi ovi signali su tipa *bit*
- Ova naredba ima isto značenje kao i ekvivalentna process naredba, prikazana doel desno

```
zmux: z <= d0 when sel1 = ‘0’ and sel0 = ‘0’ else
         d1 when sel1 = ‘0’ and sel0 = ’1’ else
         d2 when sel1 = ‘1’ and sel0 = ‘0’ else
         d3 when sel1 = ‘1’ and sel0 = ‘1’;
```

```
zmux: process is
begin
   if sel1 = ‘0’ and sel0 = ‘0’ then
      z <= d0;
   elseif sel1 = ‘0’ and sel0 = ‘1’ then
      z <= d1;
   elseif sel1 = ‘1’ and sel0 = ‘0’ then
      z <= d2;
   elseif sel1 = ‘1’ and sel0 = ‘1’ then
      z <= d3;
   end if;
   wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Naredba uslovne dodele vrednosti signalu III

- Prednost korišćenja naredbe za uslovnu dodelu vrednosti signalu u odnosu na *process* naredbu jasno se vidi iz prethodnog primera
- Jednostavna kombinatorna transformacija je očigledna korisniku, nezamagljena detaljima koji su neophodni u *process* naredbi
- Ovo ne znači da su *process* naredbe loše, već da u jednostavnim slučajevima, nije loše sakriti nepotrebne detalje da bi smo model učinili jasnijim
- Analizirajući ekvivalentni proces može se primetiti jedna važna stvar, naredba uslovne dodele osetljiva je na ***sve*** signale navedene u talasnim oblicima i uslovima
- Kada bilo koji od njih promeni vrednost, naredba uslovne dodele se ponovo aktivira i planira novu transakciju za naznačeni signal

Naredba uslovne dodele vrednosti signalu IV

- Ako malo bolje pogledamo model multipleksera, vidimo da je poslednji uslov suvišan, jer su signali *sel0* i *sel1* tipa *bit*
- Ako ni jedan od prethodnih uslova nije ispunjen, signalu *z* bi uvek trebalo da se dodeli poslednji talasni oblik
- Tako da prethodni primer možemo zapisati i na način prikazan desno

```
zmux: z <= d0 when sel1 = ‘0’ and sel0 = ‘0’ else
         d1 when sel1 = ‘0’ and sel0 = ’1’ else
         d2 when sel1 = ‘1’ and sel0 = ‘0’ else
         d3;
```

```
zmux: process is
begin
   if sel1 = ‘0’ and sel0 = ‘0’ then
      z <= d0;
   elseif sel1 = ‘0’ and sel0 = ‘1’ then
      z <= d1;
   elseif sel1 = ‘1’ and sel0 = ‘0’ then
      z <= d2;
   else
      z <= d3;
   end if;
   wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Naredba uslovne dodele vrednosti signalu V

- Vrlo čest slučaj prilikom funkcionalnog modelovanja jeste da se napiše naredba uslovne dodele bez ikakvih uslova, kao u sledećem primeru:

PC_incr: next_PC ⇐ PC+4 **after** 5 ns;

- Na prvi pogled ovo izgleda kao obična sekvencijalna naredba dodele vrednosti signalu koja bi morala da se nalazi u telu nekog procesa

- Međutim, ako pogledamo sintaksno pravilo za naredbu konkurentne dodele vrednosti signalu, možemo videti da se gornja naredba može prepoznati kao konkurentna naredba dodele vrednosti. U ovom slučaju ekvivalentni proces bi bilo:

```
PC_incr: process is
begin
    next_PC ⇐ PC+4 after 5 ns;
    wait on PC;
end process PC_incr;
```

Naredba uslovne dodele vrednosti signalu VI

- Drugi slučaj koji se javlja prilikom pisanja funkcionalnih modela je potreba za procesom koji planira početni skup transkacija, a zatim ne radi ništa do kraja simulacije

- Primer bi mogao biti generisanje reset signala. Jedan način da se to uradi bi mogao biti:

  reset_gen: reset ⇐ '1', '0' **after** 200 ns **when** extended_reset **else** '1', '0' **after** 50 ns;

- Kao što se vidi nema signala ni u talasnim oblicima ni u uslovima (pod pretpostavkom da je *extended_reset* konstanta)

- Ovo znači da se naredba izvršava jednom kada simulacija počinje, planira dve transakcije na signalu *reset* i zatim postaje neaktivna do kraja simulacije

Naredba uslovne dodele vrednosti signalu VII

- Ekvivalentan proces bio bi:

```
reset_gen: process is
begin
    if extended_reset then
        reset ⇐ ‘1’, ‘0’ after 200 ns;
    else
        reset ⇐ ‘1’, ‘0’ after 50 ns;
    end if;
    wait;
end process reset_gen;
```

- Obzirom da nisu navedeni signali, *wait* naredba nije osetljiva ni na jedan signal, pa se posle *if* naredbe proces zaustavlja sve do kraja simulacije

Naredba uslovne dodele vrednosti signalu VIII

- Jedan problem sa uslovnom dodelom vrednosti signalu jeste da se signalu uvek dodeljuje nova vrednost
- Ponekad možda ne želimo da promenimo vrednost signala, odnosno preciznije, ne želimo da planiramo novu transakciju na signalu
- U tom slučaju možemo koristiti ključnu reč *unaffected* umesto talasnog oblika, kao što je ilustrovano na desno

```
sheduler:
   request <= first_priority_request after scheduling_delay
                  when priority_waiting and server_status = ready else
               first_normal_request after scheduling_delay
                  when not priority_waiting and server_status = ready else
               unaffected
                  when server_status = busy else
               reset_request after scheduling_delay;
```

```
scheduler: process is
begin
   if priority_waiting and server_status = ready then
      request <= first_priority_request after scheduling_delay;
   elsif not priority_waiting and server_status = ready then
      request <= first_normal_request after scheduling_delay;
   elsif server_status = busy then
      null;
   else
      request <= reset_request after scheduling_delay;
   end if;
   wait on first_priority_request, priority_waiting, server_status,
           first_normal_request, reset_request;
end process scheduler;
```

Selektovana dodela vrednosti signalu I

- Ova naredba je slična naredbi uslovne dodele

- I ona predstavlja skraćeni zapis za određeni broj običnih dodela signalu unutar procesa. Ali u ovom slučaju skvivalentni proces sadrži *case* naredbu umesto *if* naredbe.

- Sintaksno pravilo je

naredba_selektovane_dodele_vrednosti_signalu ⇐
with izraz **select**
ime ⇐ [mehanizam_kašnjenja]
{talasni_oblik **when** izbori, }
talasni_oblik **when** izbori;

- Ova naredba dozvoljava nam da biramo između različitih talasnih oblika koji mogu biti dodeljeni signalu na osnovu vrednosti izraza

Selektovana dodela vrednosti signalu II

- Pogledajmo selektovanu dodelu signalu prikazanu na gore desno
- Ona ima isto značenje kao i proces dole desno.
- Naredba selektovane dodele vrednosti signalu osetljiva je na sve signale u selektorskim izrazima i talasnim oblicima.
- Ovo znači da je naredba selektovane dodele vrednosti signalu prikazana gore desno osetljiva i na signal *b* i postaće aktivna svaki put kada *b* promeni vrednost, čak i kada je vrednost *alu_function* jednaka *alu_pass_a*

```
alu: with alu_function select
   result <= a+b after Tpd        when alu_add | alu_add_unsigned,
             a-b after Tpd        when alu_sub | alu_sub_unsigned,
             a and b after Tpd    when alu_and,
             a or b after Tpd     when alu_or,
             a after Tpd          when alu_pass_a;
```

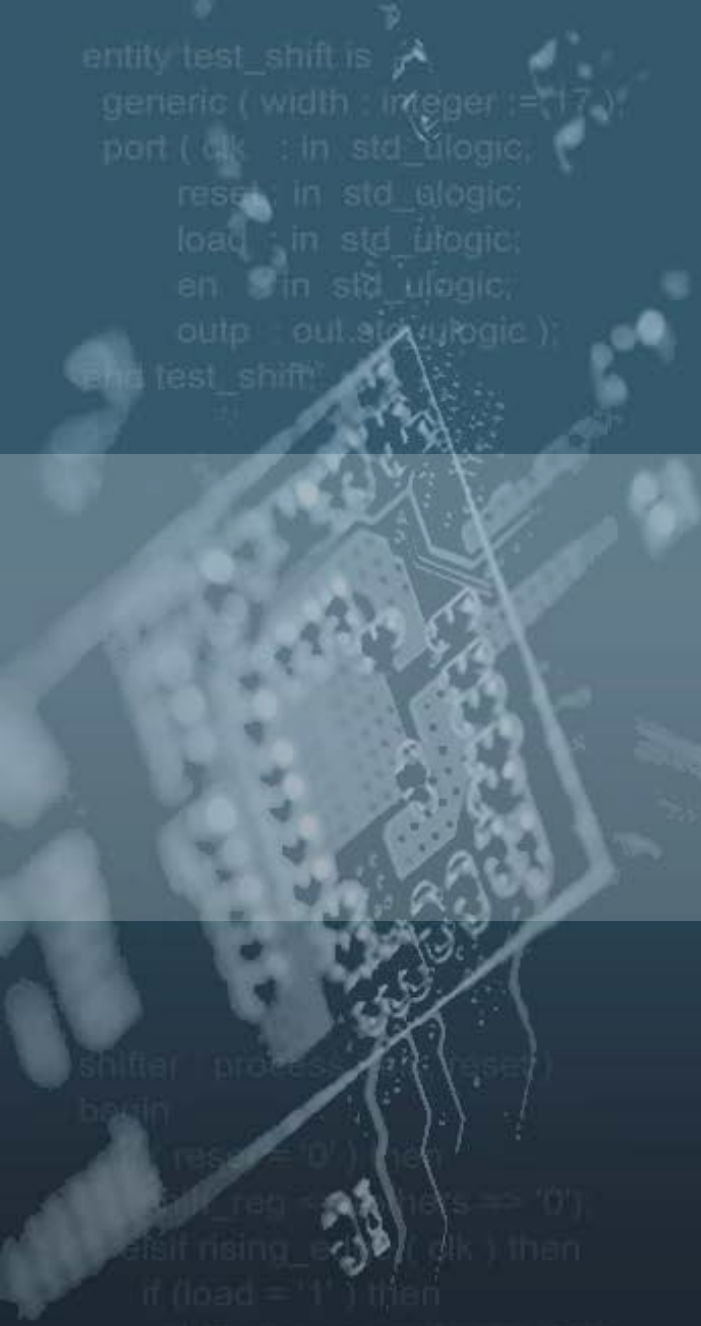
```
alu: process is
begin
   case alu_function is
      when alu_add | alu_add_unsigned => result <= a+b after Tpd;
      when alu_sub | alu_sub_unsigned  => result <= a-b after Tpd;
      when alu_and                     => result <= a and b after Tpd;
      when alu_or                      => result <= a or b after Tpd;
      when alu_pass_a                  => result <= a after Tpd;
   end case;
   wait on alu_function, a, b;
end process alu;
```

Selektovana dodela vrednosti signalu III

- Naredbu selektovane dodele signalu možemo koristiti da izrazimo kombinacionu logičku funkciju u tabelarnoj formi
- Desno je prikazana deklaracija entiteta i arhitekturno telo za sabirač
- Naredba selektovane dodele signalu ima, kao svoj selektorski izraz, bit vektor formiran spajanjem ulaznih signala
- Lista izbora navodi sve moguće vrednosti za ulaze, i za svaku od njih, date su vrednosti za izlaze *c_out* i *s*

```
entity full_adder is
    port (a, b, c_in: in bit; s, c_out: out bit);
end entity full_adder;

architecture truth_table of full_adder is
begin
    with bit_vector'(a, b, c_in) select
        (c_out, s) <= bit_vector'("00") when "000",
                      bit_vector'("01") when "001",
                      bit_vector'("01") when "001",
                      bit_vector'("10") when "010",
                      bit_vector'("01") when "011",
                      bit_vector'("10") when "100",
                      bit_vector'("10") when "101",
                      bit_vector'("11") when "111",
end architecture truth_table;
```



Konkurentna ASSERT naredba

Konkurentna ASSERT naredba I

- VHDL obezbeđuje još jednu konkurentu naredbu, konkuretnu *assert* naredbu, koja se može koristiti u modelovanju načina rada

- Sintaksno pravilo je

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

- Konkurentna *assert* naredba proverava da li je uslov ispunjen svaki put kada bilo koji od signala pomenutih u uslovnom izrazu promeni vrednost

- Ova naredba omogućava vrlo kompaktan način za uključivanje provera tajminga i korektnosti unutar modela

Konkurentna ASSERT naredba II

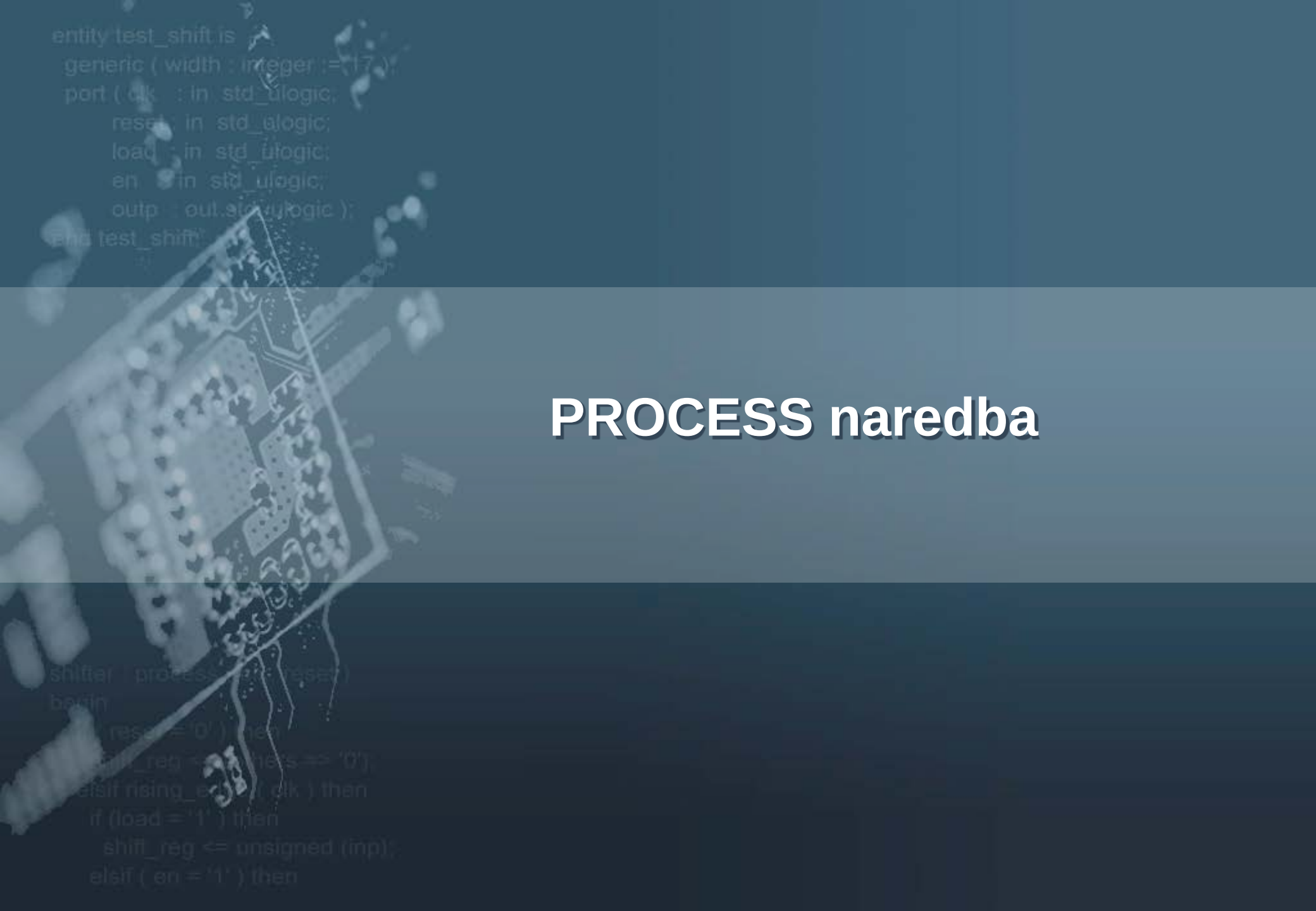
- Možemo iskoristiti konkurentnu *assert* naredbu da proverimo korektnu upotrebu set/reset flip-flopa sa dva ulaza *s* i *r*, i dva izlaza *q* i *q_n*, svi tipa *bit*
- Uslovi korišćenja zahtevaju da *s* i *r* ne budu istovremeno ‘1’ u isto veme
- Narušavanje ovog pravila proverava se pomoću konkurentne *assert* naredbe koja počinje labelom *check*
- Entitet i arhitekturno telo VHDL modela SR flip-flopa pirkazani su desno

```
entity S_R_flipflop is
    port (s, r: in bit; q, q_n: out bit);
end entity S_R_flipflop;

architecture functional of S_R_flipflop is
begin
    q <= ‘1’ when s = ‘1’ else
          ‘0’ when r = ‘1’;

    q_n <= ‘0’ when s = ‘1’ else
             ‘1’ when r = ‘1’;

    check: assert not (s = ‘1’ and r = ‘1’)
        report “Nepravilno korišćenje S/R flipflopa: s
                  i r su oba ‘1’”;
end architecture functional;
```



PROCESS naredba

PROCESS naredba I

- Sintaksno pravilo kojim se definiše *process* naredba glasi:

process_naredba ⇐ [process_labela:]
process [(ime_signala{, . . .})] **[is]**
{deklarativni_deo}
begin
{sekvencijalna_naredba}
end process [proces_labela];

- *Process* naredba je konkurentna naredba koja može biti uključena u arhitekturno telo da bi implementirala celokupni ili samo deo načina rada modula

- Iako je opciona, dobra praksa je uključivanje labele za svaki proces. Labela olakšava fazu otklanjanja grešaka (debagovanje) tokom simulacije sistema, jer većina simulatora obezbeđuje način za identifikaciju procesa pomoću njihove labele. Nakon što smo identifikovali proces možemo se koncentrisati na pronalaženje greške unutar njega.

PROCESS naredba I

- Deklarativni deo može sadržati deklaracije konstanti, promenljivih i tipova, kao i ostale deklaracije o kojima će tek biti reči. Treba naglasiti da se obične promenljive mogu deklarisati samo unutar procesa. Promenljive se koriste da predstavljaju stanje procesa.
- Sekvencijalne naredbe koje čine telo procesa biće opisane na jednom od narednih predavanja, plus naredbe dodele vrednosti signalu i *wait* naredba
- Kada proces postane aktivan tokom simulacije, on počinje sa izvršavanjem od prve sekvencijalne naredbe i nastavlja sve dok ne dođe do poslednje. Zatim počinje izvršavanje od početka.
- Ovo bi rezultovalo beskonačnom petljom, bez napretka u simulaciji, kada ne bi bilo *wait* naredbi, koje zaustavljaju izvršavanje procesa sve dok se ne dogode neki relevantni događaji
- *Wait* naredbe su jedine naredbe za čije je izvršavanje potrebno ne nulto simulaciono vreme

PROCESS naredba II

- Proces može sadržati *listu osetljivosti*, u zagradi nakon ključne reči *process*
- Lista osetljivosti sadrži skup signala koje proces 'nadgleda' očekujući pojavu događaja
- Ako je lista osetljivosti izostavljena, proces bi trebao da sadrži jednu ili više *wait* naredbi
- Sa druge strane ako lista osetljivosti postoji, onda se u telu procesa ne sme nalaziti ni jedna *wait* naredba
- Umesto toga, postoji implicitna *wait* naredba neposredno pre ključnih reči *end process*, koja sadrži signale navedene u listi osetljivosti

WAIT naredba I

- Nakon što smo videli kako možemo promeniti vrednosti signala tokom vremena, sledeći korak je da odredimo kada procesi treba da reaguju na promene vrednosti odabranih signala

- Ovo se može postići korišćenjem *wait* naredbi. *Wait* naredba je sekvencijalna naredba definisana sledećim sintaksnim pravilom

  wait_naredba ⇐ [labela:] **wait** [**on** ime_signala{, . . .}][**until** bulov_izraz]
  [**for** vremenski_izraz];

- Namena *wait* naredbe je da zaustavi izvršavanje procesa u čijem se telu nalazi

- Lista osetljivosti, uslovni deo i “timeout” deo određuju kada proces treba da ponovo postane aktivan (nastavi sa izvršavanjem)

- Možemo uključiti bilo koju kombinaciju ovih delova ili izostaviti sva tri dela prilikom korišćenja *wait* naredbe. Sada ćemo objasniti svaki od ova tri dela pojedinačno.

WAIT naredba II

- Lista osetljivosti, koja počinje sa ključnom reči *on*, dozvoljava nam da specificiramo listu signala na koje proces treba da reaguje

- Ako u *wait* naredbu uključimo samo listu osetljivosti, proces će postati aktivan svaki put kada bilo koji od navedenih signala promeni vrednost, odnosno, kada se desi događaj na bilo kojem od signala iz liste

- Ovaj tip *wait* naredbi koristan je u procesima koji modeluju blok kombinatorne logike, jer svaka promena na nekom od ulaza može rezultovati u novim vrednostima na izlazima, na primer:

```
half_add: proces is
begin
    sum ⇐ a xor b after T_pd;
    carry ⇐ a and b after T_pd;
    wait on a, b;
end process half_add;
```

WAIT naredba III

- Ovaj oblik procesa je toliko uobičajen, da VHDL obezbeđuje skraćenu notaciju koju smo do sada već videli u mnogim primerima

- Proces sa listom osetljivosti u svojem zaglavlju ekvivalentan je sa procesom koji ima *wait* naredbu na svom kraju, čija je lista osetljivosti identična sa listom osetljivosti procesa

- Tako da se proces *half_add* može zapisati kao:

```
half_add: proces (a, b) is
begin
    sum ⇐ a xor b after T_pd;
    carry ⇐ a and b after T_pd;
end process half_add;
```

WAIT naredba IV

- Vratimo se modelu dvoulaznog multipleksera. Proces u tom modelu osetljiv je na sva tri ulazna signala.
- Ovo znači da će se aktivirati na promenu bilo kog od ulaza podataka, iako je samo jedan od njih selektovan u svakom trenutku
- Ako smo zabrinuti zbog ove male neefikasnosti tokom simulacije, možemo proces napisati drugačije, koristeći *wait* naredbe da bi smo bili selektivniji u vezi sa signalima na koje je proces osetljiv svaki put kada postane neaktivan
- Prerađeni model prikazan je desno
- U ovom modelu, kada se odabere ulaz *a*, proces čeka samo na promene signala *a*. Bilo kakve promene na signalu *b* se ignorišu.
- Slično, ako je odabran ulaz *b*, proces čeka na promene na signalima *sel* i *b*, ignorišući promene na signalu *a*.

```
entity mux2 is
    port (a, b, sel: in bit; z: out bit);
end entity mux2;

architecture behavioral of mux2 is
    constant prop_delay: time := 2 ns;
begin
    slick_mux: process is
    begin
        case sel is
            when '0' =>
                z <= a after prop_delay;
                wait on sel, a;
            when '1' =>
                z <= b after prop_delay;
                wait on sel, b;
        end case;
    end process slick_mux;
end architecture behavioral;
```

WAIT naredba V

- Uslovni deo u *wait* naredbi, koji počinje sa ključnom reči *until*, omogućava nam da navedemo uslov koji mora biti ispunjen da bi proces ponovo postao aktivan. Na primer, *wait* naredba

wait until clk = '1';

- zaustavlja tekući proces sve doe se vrednost signala *clk* ne promeni na '1'.

- Uslovni izraz se testira dok je proces zaustavljen da bi se odredilo da li treba aktivirati proces

- Posledica ovoga je da čak i kada je uslov ispunjen kada se *wait* naredba izvršava, proces će ipak biti zaustavljen sve dok se odgovarajući signali ne promene i uslov ponovo bude ispunjen

- Ako *wait* naredba nema listu osetljivosti, uslov se proverava svaki put kada se desi događaj na bilo kojem od signala pomenutim u uslovu

WAIT naredba VI

- Proces koji je modelovao takt generator sa slajda 9 može se napisati drugačije koristeći *wait* naredbu sa uslovnim delom, kao što je prikazano desno
- Svaki put kada proces izvrši *wait* naredbu, *clk* signal ima vrednost '0'
- Međutim proces se ipak zaustavlja, i uslov se testira svaki put kada postoji događaj na signalu *clk*
- Kada se *clk* promeni na '1', ništa se ne dešava, ali kada se opet promeni na '0', proces se aktivira i planira transakcije za naredni ciklus

```
clock_gen: process is
begin
    clk <= ‘1’ after T_pw, ‘0’ after 2*T_pw;
    wait until clk = ‘0’;
end process clock_gen;
```

WAIT naredba VII

- Ako *wait* naredba ima listu osetljivosti i uslovni deo, uslov se proverava samo kada se desi događaj na nekom od signala iz liste osetljivosti. Na primer, ako se proces zaustavi na sledećoj *wait* naredbi:

wait on clk **until** reset = '0';

- uslov se proverava na svakoj promeni vrednosti signala *clk*, bez obzira na moguće promene signala *reset*.
- "Timeout" deo koji počinje sa ključnom reči *for*, dozvoljava nam da specificiramo maksimalni interval simulacionog vremena tokom kojeg proces može biti obustavljen
- Ako uključimo listu osetljivosti ili uslovni deo, oni mogu izazvati da se proces nastavi i pre isteka navedeno vremena u timeout delu. Na primer, *wait* naredba

wait until trigger = '1' **for** 1ms;

- zaustavlja tekući proces sve dok se *trigger* ne promeni sa '0' na '1', ili dok ne prođe 1 ms simulacionog vremena od trenutka zaustavljanja procesa.

WAIT naredba VIII

- Možemo još jednom preraditi proces za takt generator, ovaj put koristeći *wait* naredbu sa *timeout* delom, kao što je pokazano dole

- U ovom slučaju za timeout specificiramo periodu takta, nakon koje se proces ponovo aktivira

```
clock_gen: process is
begin
    clk <= ‘1’ after T_pw, ‘0’ after 2*T_pw;
    wait for 2*T_pw;
end process clock_gen;
```

WAIT naredba IX

- Ako se vratimo na sintaksno pravilo vidimo da je dozvoljeno napisati

wait;

- Ovaj oblik zaustavlja tekući proces da kraja simulacije
- Iako se ovo isprva može činiti čudnim, vrlo često se pokazuje korisnim
- Jedno mesto gde se može iskoristiti je unutar procesa čija je namena da generiše stimuluse za simulaciju
- Takav proces treba da generiše sekvencu transakcija na signalima koji su spojeni sa drugim delovima modela i zatim da stane

WAIT naredba X

- Na primer, proces

```
test_gen: process is
begin
    test0 ⇐ '0' after 10 ns, '1' after 20 ns, '0' after 30 ns, '1' after 40 ns;
    test1 ⇐ '0' after 10 ns, '1' after 30 ns;
    wait;
end process test_gen;
```

- generiše sve četiri moguće kombinacije vrednosti na signalima *test0* i *test1*

- Ako bi *wait* naredba bila izostavljena, proces bi nastavio sa izvršavanjem, ponavljajući naredbe dodele vrednosti bez zaustavljanja i simulacija ne bi mogla da napreduje

Delta kašnjenja I

- Vratimo se ponovo na temu kašnjenja u naredbi dodele vrednosti signalu. U mnogim primerima do sada, ovaj deo je izostavljan.
- Ovo je ekvivalentno sa specificiranjem kašnjenja od 0 fs. Nova vrednost trebala bi da se dodeli signalu u tekućem simulacionom vremenu.
- Međutim, bitno je naglasiti da se vrednost signala neće promeniti odmah nakon izvršenja naredbe dodele
- Umesto toga, naredba dodele *planira* transakciju na signalu, koja će biti primenjena nakon što se proces zaustavi
- Stoga proces 'ne vidi' rezultat izvršenja naredbe dodele sve do trenutka kada ponovo postane aktivan
- Zbog ovoga kašnjenje od 0 fs u naredbi dodele zove se *delta kašnjenje*

Delta kašnjenja II

- Da bi se shvatilo zašto delta kašnjenje funkcioniše na baš takav način, neophodno je analizirati simulacioni ciklus

- On se sastoji iz dve faze:
  - faze *ažuriranja signala*, koja je praćena fazom
  - *izvršavanja procesa*

- U fazi ažuriranja signala, simulaciono vreme se pomera do onog trenutka u kojem je planirana najranija transakcija, i vrednosti u svim transakcijama (može ih biti više od jedne) planiranim za taj trenutak se primenjuju na odgovarajuće signale. Ovo može dovesti do pojave događaja na nekim signalima.

- U fazi izvršavanja procesa, svi procesi koji reaguju na te događaje (imaju odgovarajuće signale u svojim listama osetljivosti) postaju aktivni i izvršavaju se sve dok se ponovo ne zaustave na *wait* naredbama. Simulator zatim prelazi na novi simulacioni ciklus.

Delta kašnjenja III

- Pogledajmo šta se događa kada proces izvrši naredbu dodele vrednosti signalu sa delta kašnjenjem, na primer:

$$\text{data} \Leftarrow \text{X'00';}$$

- Pretpostavimo da se naredba izvršava u simulacionom vremenu *t*, tokom faze izvršavanja procesa u tekućem simulacionom ciklusu

- Kao rezultat izvršenja naredbe dodele vrednosti signalu planira se transakcija za dodeljivanje vrednosti *X'00'* signalu *data* u trenutku *t*

- Transakcija se ne primenljuje odmah, jer je simulator još uvek u fazi izvršavanja procesa

Delta kašnjenja IV

- Zbog toga proces nastavlja sa izvršavanjem, a vrednost signala *data* ostaje ne promenjena
- Kada se svi aktivni procesi zaustave, simulator počinje naredni simulacioni ciklus i ažurira simulaciono vreme
- Obzirom da je najranija transakcija planirana za trenutak *t*, simulaciono vreme ostaje nepromenjeno
- Simulator sada primenjuje vrednost *X'00'* u planiranoj transakciji na signalu *data*, a zatim nastavlja sve procese koji reaguju na ovu promenu

Delta kašnjenja V

- Pisanje modela sa delta kašnjenjima korisno je kada radimo na visokom nivou apstrakcije i još nismo zainteresovani za detaljni tajming u modelu

- Međutim mora se obratiti pažnja na uobičajenu grešku početnika u VHDL koji koriste delta kašnjenja: zaboravlja se da proces “ne vidi” efekat naredbe dodele vrednosti signalu trenutno

- Na primer, mogli bi napisati proces koji sadrži sledeće naredbe:

```
s <= '1';
if s = '1' then
...
```

- i očekivati da će proces izvršiti *if* naredbu pretpostavljajući da će *s* imati vrednost '1'. Zatim bi smo proveli sate i sate nastojeći da utvrdimo zašto se to ne događa, sve dok se ne bi setili da *s* još uvek ima svoju staru vrednost sve do novog simulacionog ciklusa.

Delta kašnjenja VI

- Desno je prikazan okvir apstraktnog modela računarskog sistema
- Procesor i memorija spojeni su preko adresne i madistrale podataka
- Sinhronizuju svoj rad pomoću *mem_read* i *mem_write* kontrolnih signala i *mem_ready* status signala
- Pošto nisu specificirana kašnjenja u naredbama dodele vrednosti signalima, sinhronizacija se dešava tokom nekoliko delta ciklusa, kao što je prikazano na slici na sledećem slajdu

```
architecture abstract of computer_system is
   subtype word is bit_vector(31 downto 0);
   signal address: natural; signal read_data, write_data: word;
   signal mem_read, mem_write, mem_ready : bit := '0';
begin
   cpu: process is
      variable instr_reg: word; variable PC: natural;
   begin
      loop
         address <= PC;  mem_read <= '1';
         wait until mem_ready = '1';
         instr_reg := read_data;  mem_read <= '0';
         wait until mem_ready = '0';
         PC := PC+4; ...     --  izvrši instrukciju
      end loop;
   end process  cpu;

   memory: process is
      type memory_array is array (0 to 2**14-1) of word;
      variable store: memory_array := (...);
   begin
      wait until mem_read = '1' or mem_write = '1';
      if mem_read = '1' then
         read_data <= store(address/4); mem_ready <= '1';
         wait until mem_ready = '0';
         mem_ready <= '0';
      else
         ...          izvrši pristup upisa
      end if;
   end process memory;
end architecture abstract;
```

Delta kašnjenja VII

- Kada počne simulacija, *cpu* proces počinje sa izvršavanjem svojih naredbi, a *memory* proces se odmah zaustavlja
- *Cpu* planira transakcije da bi dodelio adresu naredne instrukcije signalu *address* i vrednost '1' signalu *mem_read*, a zatim se zaustavlja
- U sledećem simulacionom ciklusu, ovi signali se ažuriraju i *memory* proces se aktivira, jer je on čekao na događaj na signalu *mem_read. Memory* proces planira dodelu *read_data* signalu i dodelu vednosti '1' *mem_ready* signalu, a zatim se zaustavlja.
- U trećem ciklusu, ovi signali se ažuriraju i *cpu* proces se aktivira. On planira dodelu vrednosti '0' *mem_read* signalu i zatim se zaustavlja.
- U četvrtom ciklusu, *mem_read* se ažurira i *memory* proces se aktivira, planira dodelu vrednosti '0' signalu *mem_ready* da bi završio proces sinhronizacije.
- Konačno, u petom ciklusu, *mem_ready* se ažurira i *cpu* proces se aktivira i izvršava zahvaćenu instrukciju.