Laboratorijska vežba 9

Rad sa datotekama u VHDL-u

Iako je rad sa datotekama potpuno nebitan prilikom pisanja modela sistema namenjenog za sintezu (jer ni jedan alat za automatsku sintezu hardvera nije u stanju da generiše odgovarajuće digitalno elektronsko kolo koje bi bilo u stanju da radi sa datotekama), datoteke se često koriste prilikom razvoja složenih verifikacionih okruženja, o čemu će biti reči kasnije. Datoteke pružaju efikasan metod za komunikaciju između verifikacionog inženjera i verifikacionog okruženja. Ovo se pogotovo odnosi na tekstualne datoteke, o kojima će biti reč u nastavku. Unutar verifikacinih okruženja, tekstualne datotetke koriste se za:

- inicijalizaciju modela memorija (za inicijalno popunjavanje njihovog sadržaja)
- smeštanje vremenskih oblika ulaznih sigala koje je potrebno dovesti na ulaze sistema koji se verifikuje
- smeštanje komandi i podešavanja koja se koriste za automatsko generisanje stimulusa unutar samog verifikacionog okruženja
- čuvanje različitih prikupljenih informacija od strane verifikacionog okruženja prilikom simulacije modela koji se verifikuje

U nastavku će biti prikazani primeri koji ilustruju rad sa datotekama unutar VHDL jezika.

Primer 1: VHL model memorije sa mogućnošću odabira sadržaja sa kojim će se izvršiti inicijalizacija

Često se prilikom testiranja složenih digitalnih sistema, pogotovo onih koji su bazirani na mikroprocesorima, javlja potreba za korišćenjem memorije u kojoj će biti smešten test program ili neki drugi kritični podaci. Jedan način kako da se ovakva memorija inicijalizuje jeste da se direktno inicijalizuje niz koji zapravo predstavlja memorijske lokacije. Iako je ovaj način jednostavan za implementaciju, ukoliko želimo da promenimo inicijalni sadržaj memorije moramo ga ručno izmeniti. Elegantnije rešenje bi se sastojalo u korišćenju datoteke koja sadrži inicijalni sadržaj memorije koji želimo koristiti. VHDL modul u kojem je opisana posmatrana memorija na početku bi učitao odabranu datoteku i sa njenim sadržajem inicijalizovao memoriju pre početka korišćenja.

Na primer, sledeći VHDL modul prikazuje jedan način modelovanja spoljašnje programske ROM memorije koja bi se mogla koristiti prilikom verifikacije modela mikrokontrolera 8051.

```
entity xrom is
    generic (filename: string);
    port (read_rom: in std_logic;                          -- signal that reads data from ROM
          xrom_addr: in std_logic_vector(15 downto 0); -- address bus (max. 64 KB)
          xrom_data: out std_logic_vector(7 downto 0)  -- data bus
         );
end entity xrom;

architecture beh of xrom is
```

```
    type mem_type is array (natural range <>) of std_logic_vector(7 downto 0);
    subtype ekst_rom_index is natural range 0 to 65535;
    subtype ekst_rom is mem_type(ekst_rom_index);
    signal xrom_array: ekst_rom;
begin
    xrom:
    process is
        ------------------------------------------------------------------------------------------------------------
        -- Purpose: Procedure that loads program into internal program memory. Program file
        -- must be produced with Franklin Macro Assembler. After running the a51 compiler,
        -- *.lst file will be generated. Only this file format is recognized by this loading
        -- procedure.
        ------------------------------------------------------------------------------------------------------------
        procedure Load_Program (file_name: string; memory: out mem_type) is
            file prog_file: text open read_mode is file_name;
            variable L: line;
            variable ch: character;
            variable line_number: natural := 0;
            variable byte_address: natural := 0;
            variable byte_count: natural := 0;
            variable indicator: natural := 0;
            type char_array is array (natural range <>) of character;
            variable temp: char_array (0 to 1);
            variable addr: char_array (0 to 3);
            -------------------------------------------------------------------------------------------------------
            -- Puropse: Function that converts loaded memory location represented as char
            -- array into a natural used to index the program memory array, xrom_array.
            -------------------------------------------------------------------------------------------------------
            function Char_to_Natural (addr_array: char_array) return natural is
                variable increment: natural;
                variable result: natural := 0;
                variable power: natural := 1;
            begin
                for i in addr_array'range loop
                    case addr_array(i) is
                        when '0' => increment := 0;
                        when '1' => increment := 1;
                        when '2' => increment := 2;
                        when '3' => increment := 3;
                        when '4' => increment := 4;
                        when '5' => increment := 5;
                        when '6' => increment := 6;
                        when '7' => increment := 7;
                        when '8' => increment := 8;
                        when '9' => increment := 9;
                        when 'A' => increment := 10;
                        when 'B' => increment := 11;
                        when 'C' => increment := 12;
                        when 'D' => increment := 13;
                        when 'E' => increment := 14;
                        when 'F' => increment := 15;
                        when others => report „PANIC ERROR!“ severity error;
                    end case;
                    result := result+power*increment;
                    power := power*16;
                end loop;
                return result;
            end function Char_to_Natural;
        begin
            while not endfile (prog_file) loop
                readline (prog_file, L);
                read (L, ch);
                read (L, temp(1));
```

```
                    read (L, temp(0));
                    byte_count := Char_to_Natural (temp);
                    read (L, addr(3));
                    read (L, addr(2));
                    read (L, addr(1));
                    read (L, addr(0));
                    byte_address := Char_to_Natural (addr);
                    read (L, temp(1));
                    read (L, temp(0));
                    indicator := Char_to_Natural (temp);
                    if indicator /= 0 then
                        exit;
                    end if;
                    for i in 1 to byte_count loop
                        read (L, ch);
                        case ch is
                            when ’0’ => memory(byte_address)(7 downto 4) := X“0“;
                            when ’1’ => memory(byte_address)(7 downto 4) := X“1“;
                            when ’2’ => memory(byte_address)(7 downto 4) := X“2“;
                            when ’3’ => memory(byte_address)(7 downto 4) := X“3“;
                            when ’4’ => memory(byte_address)(7 downto 4) := X“4“;
                            when ’5’ => memory(byte_address)(7 downto 4) := X“5“;
                            when ’6’ => memory(byte_address)(7 downto 4) := X“6“;
                            when ’7’ => memory(byte_address)(7 downto 4) := X“7“;
                            when ’8’ => memory(byte_address)(7 downto 4) := X“8“;
                            when ’9’ => memory(byte_address)(7 downto 4) := X“9“;
                            when ’A’ => memory(byte_address)(7 downto 4) := X“A“;
                            when ’B’ => memory(byte_address)(7 downto 4) := X“B“;
                            when ’C’ => memory(byte_address)(7 downto 4) := X“C“;
                            when ’D’ => memory(byte_address)(7 downto 4) := X“D“;
                            when ’E’ => memory(byte_address)(7 downto 4) := X“E“;
                            when ’F’ => memory(byte_address)(7 downto 4) := X“F“;
                            when others => report „PANIC ERROR!“ severity error;
                        end case;
                        read (L, ch);
                        case ch is
                            when ’0’ => memory(byte_address)(3 downto 0) := X“0“;
                            when ’1’ => memory(byte_address)(3 downto 0) := X“1“;
                            when ’2’ => memory(byte_address)(3 downto 0) := X“2“;
                            when ’3’ => memory(byte_address)(3 downto 0) := X“3“;
                            when ’4’ => memory(byte_address)(3 downto 0) := X“4“;
                            when ’5’ => memory(byte_address)(3 downto 0) := X“5“;
                            when ’6’ => memory(byte_address)(3 downto 0) := X“6“;
                            when ’7’ => memory(byte_address)(3 downto 0) := X“7“;
                            when ’8’ => memory(byte_address)(3 downto 0) := X“8“;
                            when ’9’ => memory(byte_address)(3 downto 0) := X“9“;
                            when ’A’ => memory(byte_address)(3 downto 0) := X“A“;
                            when ’B’ => memory(byte_address)(3 downto 0) := X“B“;
                            when ’C’ => memory(byte_address)(3 downto 0) := X“C“;
                            when ’D’ => memory(byte_address)(3 downto 0) := X“D“;
                            when ’E’ => memory(byte_address)(3 downto 0) := X“E“;
                            when ’F’ => memory(byte_address)(3 downto 0) := X“F“;
                            when others => report „PANIC ERROR!“ severity error;
                        end case;
                        byte_address := byte_address + 1;
                    end loop;
                end loop;
            end procedure Load_program;
begin
    Load_Program (filename, xrom_array);
    loop
        wait on read_rom, xrom_addr;
        if read_rom = '0' then
```

```
            xrom_data <= xrom_array(conv_integer (xrom_addr));
        else
            xrom_data <= "ZZZZZZZZ";
        end if;
    end loop;
  end process xrom;
end architecture beh;
```

Kao što se može videti analizirajući priloženi VHDL kod, spoljašnja programska memorija modelovana je kao ROM memorija sa dva ulazna porta: *read_rom*, 1-bitni signal koji predstavlja indikaciju da mikrokontroler 8051 pristupa ROM memoriji, i *xrom_addr*, 16-bitni signal koji predstavlja adresnu magistralu. ROM memorija ima samo jedan izlazni port, *xrom_data*, 8-bitni signal koji predstavlja magistralu podataka. Pored toga model ROM memorije ima i jedan generic, *filename*, koji služi za zadavanje imena datoteke čijim sadržajem će se popuniti ROM prilikom incijalizacije.

Razvijeni model pretpostavlja da će izvršni program kojim treba napuniti spoljašnju programsku memoriju biti zapisan u „Intel Standard HEX“ fajl formatu, koji je jedan od najpopularnijih i najčešće korišćenih formata prilikom rada sa 8051/52 mikrokontrolerima. Na primer, program kompajliran/asembliran korišćenjem Keil razvojnog okruženja imaće upravo ovaj format.

Intel Standard HEX fajl je zapravo ASCII fajl u kojem su podaci organizovani po linijama. Svaka linija ima sledeći format:

Pozicija	Opis
1	**Marker linije**: Prvi karakter svake linije je uvek ':', koji liniju identifikuje kao Intel HEX
2-3	**Dužina linije:** Ovo polje sadrži podatak o broju bajtova koji se odnose na podatke koje tekuća linija sadrži, predstavljen kao 2-cifreni heksadecimalni broj.
4-7	**Adresa:** Ovo polje sadrži početnu adresu od koje treba smeštati podatke iz linije u memoriji. Ova vrednost je uvek iz opsega 0-65535, predstavljena kao 4-cifreni heksadecimalni broj.
8-9	**Tip linije:** Ovo polje sadrži podatak o tipu linije. Moguće vrednosti su: **00** = Linija sadrži normalne podatke, **01** = Kraj datoteke, **02** = Proširena adresa.
10-?	**Podaci:** Ovo polje sadrži podatke koje je potrebno učitati u memoriju. Svaki podatak je predstavljen kao 2-cifreni heksadecimalni broj.
Poslednja 2 karaktera	**Čeksum:** Poslednja dva karaktera predstavljaju čeksum vrednost tekuće linije.

Na primer, konkretan izgled jednog Intel Standard HEX fajla mogao bi biti

```
:10000000740075200078207921760077007A007BD3
:10001000007C007D007E007F00900000040520062B
:100020000708090A0B0C0D0E0FA374FF0478FF08D4
:1000300077FF077520FF052090FFFFA375F002757D
:10004000D00375810475820575830675800775 90E8
:100050000875A00975B00A75B80B75A800758800F9
:1000600075890C758C0D758A0E758D0F758B1075D5
:100070000982175871305F005D0058105820583055 4
:100080008005900 5A005B005B805A8058805890577
:100090008C058A058D058B0598058775F0FF75D051
```

```
:1000A000FF7581FF7582FF7583FF7580FF7590FF77
:1000B00075A0FF75B0FF75B8FF7589FF758CFF756A
:1000C0008AFF758DFF758BFF7587FF05F005D005DD
:1000D0008105820583058005900 5A005B005B8055A
:0B00E00089058C058A058D058B0587BE
:00000001FF
```

Analizirajući *xrom* proces možemo videti da se prilikom prvog izvršavanja procesa poziva procedura *Load_Program*. Ova procedura učitava sadržaj odabrane datoteke (na osnovu imena datoteke prosleđenog preko ulaznog argumenta *file_name*) u programsku memoriju, modelovanu pomoću niza *xrom_array*. Unutar procedure otvara se odabrana datoteka i procesira linija po linija poštujući pravila iz Tabele 1.

Nakon inicijalizacije sadržaja memorije proces ulazi u beskonačnu petlju u kojoj čeka na događaj na jednom od dva ulazna signala, *read_rom*, *xrom_addr*. Kada se desi događaj na jednom od ta dva signala, izvršava se ostatak petlje, koji zapravo modeluje ponašanje ROM memorije sa *three-state* izlazom.

Primer 2: VHDL modela monitora magistrale (bus monitor)

Prilikom verifikacije složenih mikroprocesorskih sistema često se javlja potreba za nagledanjem protoka koji se odvija na sistemskoj magistrali. Ovo se može postići na taj način što će se napisati takozvani monitor magistrale (*bus monitor*) koji će kreirati *log* fajl u kojem će biti smeštena sva aktivnost koja se odvijala na magistrali. Pretpostavimo da se model mikroprocesorskog sistema koji se verifikuje sastoji od mikroprocesora, memorije i I/O kontrolera koji su povezani preko jedne, sistemske, magistrale pomoću sledećih signala:

```
signal address: std_logic_vector(15 downto 0);
signal data: std_logic_vector(7 downto 0);
signal rd, wr, io: std_logic;
signal ready: std_logic;
```

Monitor magistrale može se napisati unutar jednog procesa, *bus_monitor*. Unutar procesa deklariše se izlazna datoteka *log*, koja je tipa *text*, kao i pokazivačka promenljiva *trace_line*, tipa *line*, u kojoj se nalazi tekuća linija koja će biti upisana u *log* fajl. Proces se aktivira kada memorija ili I/O kontroler odgovore na zahteve magistrale za upisom ili čitanjem. *Bus_monitor* proces generiše formatiranu liniju teksta koristeći *write* operacije iz *textio* paketa. Takođe, vodi računa i o tome koliko linija je bilo upisano u *log* datoteku i dodaje zaglavlje nakon svakih 60 linija. Sadržaj *log* datoteke koju bi mogao generisati *bus_monitor* proces imao bi sledeći izgled

```
  Time   R/W  I/M  Address          Data
 0.4 us   R    M  0000000000000000 10011110
 0.9 us   R    M  0000000000000001 00010010
   2 us   R    M  0000000000010100 11100111
 2.7 us   W    I  0000000000000111 00000000
```

VHDL model monitora magistrale mogao bi imati sledeći izgled.

```
bus_monitor: process is
    constant header: string(1 to 44)
        := FF & "  Time   R/W  I/M  Address          Data";
    use std.textio.all;
```

```
        file log: text open write_mode is "buslog";
        variable trace_line: line;
        variable line_count: natural := 0;
    begin
        if line_count mod 60 = 0 then
            write (trace_line, header);
            writeline (log, trace_line);
            writeline (log, trace_line); -- empty line
        end if;

        wait until (rd = '1' or wr = '1') and ready = '1';
        write (trace_line, now, justified => right, field => 10, unit => us);
        write (trace_line, string'("    "));

        if rd = '1' then
            write (trace_line, 'R');
        else
            write (trace_line, 'W');
        end if;
        write (trace_line, string'("  "));

        if io = '1' then
            write (trace_line, 'I');
        else
            write (trace_line, 'M');
        end if;

        write (trace_line, string'("  "));
        write (trace_line, address);
        write (trace_line, ' ');
        write (trace_line, data);
        writeline (log, trace_line);

        line_count := line_count + 1;
    end process bus_monitor;
```

Zadaci za vežbu

Zadatak 1:

Pretpostavimo da naredna linija u tekstualnoj datoteci ima sledeći sadržaj

```
123 4.5 6789
```

Šta će biti povratne vrednosti sledećih *read* poziva?

```
readline (in_file, L);
read (L, bit_value); -- pročitaj vrednost tipa bit
read (L, int_value); -- pročitaj vrednost tipa integer
read (L, real_value); -- pročitaj vrednost tipa real
read (L, str_value); -- pročitaj vrednost tipa string(1 to 3)
```

Zadatak 2:

Napisati bihevijalni model adresnog dekodera sa sledećom *entity* deklaracijom:

```
entity address_decoder is
    generic (log_file_name: string);
    port (address: in natural;
          enable: in std_logic;
          ROM_sel, RAM_sel, IO_sel, int_sel: out std_logic);
end entity address_decoder:
```

Kada je *enable* port jednak ‘1’, dekoder na osnovu vrednosti ulaznog porta *address* treba da odluči koji od izlaznih signala treba aktivirati. Signal *ROM_sel* treba aktivirati ukoliko se *address* nalazi u opsegu 0 - 0x7FFF, *RAM_sel* treba aktivirati ukoliko se *address* nalazi u opsegu 0x8000 – 0x BFFF, *IO_sel* ukoliko se *address* nalazi u opsegu 0xC000 – 0xEFFF, a *int_sel* ukoliko se *address* nalazi u opsegu 0xF000 – 0xFFFF. Dekoder, pored toga što aktivira odgovarajuće izlazne signale, za svaku aktivaciju *enable* signala u log fajl treba da upiše koja je vrednost *address* porta bila prisutna i koja je periferija bila aktivirana. Log fajl treba da bude običan tekstualni fajl, sa sledećim formatom:

```
Address          Peripheral
-----------------------------------
0x1234            ROM
0xC334            IO
0xFF00            INT
0xA453            RAM
```

a) Napisati bihevijalni model adresnog dekodera.

b) Napisati jednostavan *testbench* pomoću kojega će biti moguće verifikovati ispravan rad dekodera.

Zadatak 3:

Napisati proces koji generiše signal *sig*, tipa *integer*, dodeljujući mu vrednosti koje su specificirane u ulaznoj datoteci čije je ime zadato pomoću parametra *filename*. Ulazna datoteka sadrži po dva podatka, organizovana u linije. Prvi podatak u svakoj liniji predstavlja vreme trajanja vrednosti signala izraženo u nanosekundama, a drugi samu vrednost. Na primer, izgled ulazne datoteke mogao bi biti sledeći:

```
1 5
10 35
7 44
...
```

Proces treba da učitava liniju po liniju, dodeli specificiranu vrednost izlaznom signalu *sig* i sačeka specificirano vreme, a zatim pređe na sledeću liniju. Kada završi sa obradom poslednje linije proces treba da se zaustavi do kraja tekuće simulacije.

Zadatak 4:

Napisati proces koji će smeštati istoriju promena vrednosti signala *sig*, tipa *std_logic_vector(7 downto 0)*, u izlaznu tekstualnu datoteku, čije je ime specificirano pomoću parametra *filename*. Proces treba da se aktivira svaki put kada dođe do promene na signalu *sig* i da u novoj liniji unutar log fajla zabeleži simulaciono vreme kada je došlo do promene vrednosti signala *sig* kao i novu vrednost signala *sig*.

Razvoj verifikacionih okruženja u VHDL-u

Složenost savremenih digitalnih sistema je tolika da je najveći problem sa kojim se inženjeri suočavaju prilikom njihovog razvoja utvrđivanje da li projektovani sistem ima željenu funkcionalnost i da li su sve stavke iz specifikacije korektno implementirane. Zadatak verifikacije jeste da da odgovore na ova pitanja, odnosno da utvrdi da li projektovani sistem zadovoljava svoju funkcionalnu specifikaciju. Pod funkcionalnom specifikacijom podrazumeva se detaljan opis željenih funkcija koja projektovani sistem treba da poseduje. Na osnovu funkcionalne specifikacije prave se dva dokumenta: dizajn specifikacija, koja opisuje kako će zahtevane funkcije biti implentirane i verifikacioni plan, koji opisuje kako će biti verifikovano da li projektovani sistem poseduje i korektno implementira zahtevanu funkcionalnost.

Postoje dva pristupa funkcionalnoj verifikaciji:

- Verifikacija bazirana na simulaciji – kod koje se ponašanje projektovanog sistema simulira pomoću posebno projektovanih softverskih alata, simulatora. Projektovani sistem se simulira zajedno sa simulacionim modelom njegovog okruženja (koje je modelovano pomoću odgovarajućeg testbenč fajla) čiji zadatak je da generiše odgovarajući stimulus koji se dovodi na ulaze sistema koji se testira. Pomoću simulatora se zatim izračunava odziv sistema na generisani stimulus i upoređuje sa očekivanim vrednostima. Ukoliko se očekivane i simulirane vrednosti razlikuju to predstavlja indikaciju prisustva neke greške (*bug*) unutar projektoavnog sistema.
- Formalna verifikacija – kod koje se matematički dokazuje prisustvo ili odsustvo odgovarajućih osobina koje projektovani sistem treba da poseduje. Kod ovog pristupa očekivana funkcionalnost koju projektovani sistem treba da poseduje se prevodi u odgovarajući skup tvrđenja (osobina). Ova tvrđenja se zapisuju pomoću posebnih jezika za opis osobina (*Property Specification Language, PSL*) i predstavljaju ulaz u odgovarajući alat za formalnu verifikaciju. Na osnovu modela projektovanog sistema i skupa tvrđenja koja on treba da zadovoljava alat za formalnu verifikaciju pokušava da dokaže zadovoljenost ili narušavanje svakog od navedenih tvrđenja.

Testbenč fajlovi predstavljaju osnovu funkcionalne verifikacije. Savremeni testbenč fajl obično obavlja sledeće funkcije:

- instancionira model sistema koji je potrebno verifikovati
- generiše stimulus koji se dovodi na ulaze instancionirano sistema koji se verifikuje
- prikuplja odziv sistema na tako generisani stimulus
- vrši proveru da li je prikupljeni odziv identičan sa očekivanim odzivom koje je sistem trebao da generiše
- prikuplja informacije o toku verifikacije čijom se kasnijom analizom može utvrditi kako napreduje proces verifikacije i kada se verifikacija može završiti
- generiše odgovarajuće izveštaje namenjene verifikacionim inženjerima

Testbenč fajlovi se međusobno razlikuju po načinima kako su gore navedene funkcije u njima implementirane. Jedna od osnovnih podela testbenč fajlova odnosi se na način kako se vrši provera očekivanih rezultata unutar testbenča. Prema ovom kriterijumu svi testbenč fajlovi mogu se podeliti u dve velike grupe:

- Testbenčeve bazirane na konceptu „zlatnih vektora“
- Testbenčeve bazirane ne referentnim modelima

Problematika verifikacije je vrlo kompleksna i izlazi iz okvira ovog kursa. Na višim godina postoje posebni predmeti posvećeni tehnikama funkcionalne i formalne verifikacije koji zainteresovanim studentia pružaju potrebna znanja koje jedan verifikacioni inženjer treba da poseduje. Na ovom mestu će samo biti ilustrovana osnovna struktura najjednostavnijih testbenčeva iz svake grupe.

Testbenčevi bazirani na „zlatnim vektorima“

Kod ovog tipa testbenča stimulus koji je neophodno dovesti na ulaze sistema koji se verifikuje, kao i očekivani odziv na svaki od ovih stimulusa unapred je definisan i smešten u skup takozvanih „zlatnih vektora“. Pod zlatnim vektorom podrazumeva se jedan par vrednosti ulaznih signala sistema koji se verifikuje i očekivanog izlaza sistema u tom slučaju. Obzirom da su ulazi i izlazi u sistem zadati u vektorskoj formi (kao niz individualnih vrednosti za svaki od ulaznih portova i niz očekivanih vrednosti na svakom od izlaznih portova sistema koji se verifikuje), jedan ovakav par čini jedan „zlatni vektor“. Za obavljanje verifikacije nije dovoljan samo jedan zlatni vektor već je neophodan veliki skup različitih zlatnih vektora. Ovaj skup zlatnih vektora najčešće se smešta u odgovarajuću datoteku koja se na početku simulacije učitava u verifikaciono okruženje. Prilikom simulacije testbenč uzima jedan po jedan zlatni vektor iz skupa i njegovu ulaznu komponentu dovodi na ulaze sistema koji je potrebno verifikovati. Nakon što se u procesu simulacije izračuna odziv sistema koji se verifikuje na ovu ulaznu komponentu zlatnog vektor, on se upoređuje sa izlaznom komponentom tekućeg zlatnog vektora. Ukoliko su oni identični proces verifikacije se nastavlja sa sledećim zlatnim vektorom. U slučaju da dolazi do odstupanja, proces verifikacije sa prekida, a identifikovano odstupanje se saopštava verifikacionom inženjeru kako bi mogao započeti proces lokalizacije greške (*bug tracing*) unutar sistema.

Sledeći primer prikazuje moguću strukturu testbenča baziranog na „zlatnim vektorima“ koji bi se mogao koristiti za funkcionalnu verifikaciju korektnog rada sekvencijalnog „Add-and-Shift“ množača koji smo razmatrali u vežbi 8.

Primer 1: Testbenč baziran na „zlatnim vektorima“ za verifikaciju sekvencijalnog „Add-and-Shift“ množača

U vežbi 8 postoja smo problem projektovanja digitalnog sistema koji će implementirati poznati „Add-and-Shift“ algoritam za množenje dva binarna neoznačena broja. U ovom primeru napisaćemo verifikaciono okruženje napisano u VHDL-u, bazirano na „zlatnim vektorima“, pomoću kojega se može izvršiti verifikacija projektovanog množača. Obzirom da ćemo i u naredne dve vežbe koristiti ovaj isti primer, kao prvi korak izvršićemo nekoliko modifikacija originalnog sistema korišćenog u vežbi 8.

Kao i u vežbi 8, projektovani sekvencijalni množač ima sledeće portove:

Ime porta	Tip	Funkcija
clk, reset	ulaz	Portovi za dovođenje globalnog sinhronizacionog signala i signala za inicijalizaciju sistema.
start_i	ulaz	Signal pomoću kojega se pokreće množač.
a_i, b_i	ulaz	Vrednosti dva operanda čiji proizvod želimo izračunati.
ready_o	izlaz	Izlazni signal pomoću kojega sistem signalizira da li je spreman za izvođenje nove operacije množenja ili je trenutno zauzet.
r_o	izlaz	Izlazni signal koji sadrži rezultat množenja dva operanda.

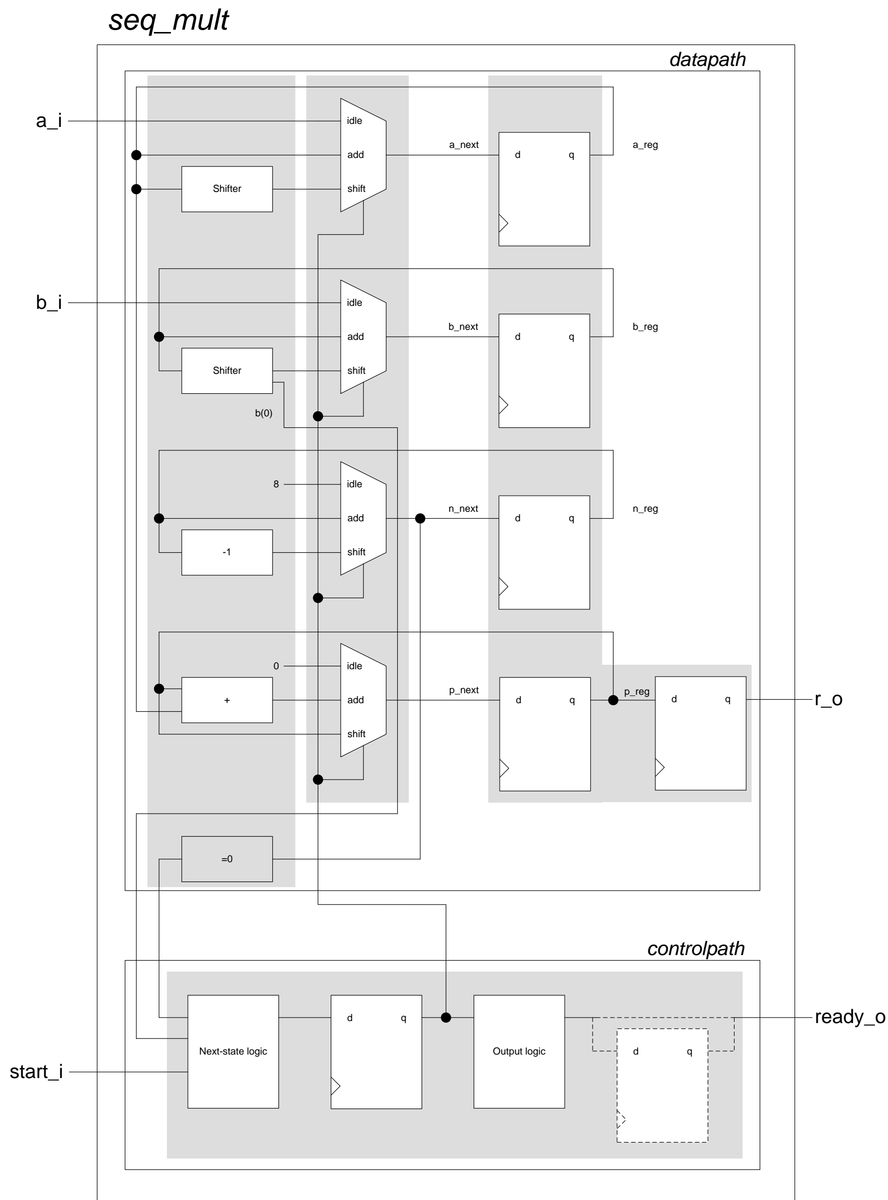
Prva razlika u odnosu na sistem projektovan u vežbi 8 odnosi se na način generisanja *r_o* signala. U vežbi 8 *r_o* signal bio je direktno povezan na izlaz *r* registra i odslikavao je svaku promenu njegovog sadržaja tokom rada množača, tako da se preko *r_o* porta lepo mogao videti način kako „Add-and-Shift" množač sekvencijalno izračunava finalni proizvod dva zadata broja. Za potrebe razvoja različitih tipova verifikacionih okruženja izvršićemo malu modifikaciju načina na koji se generiše *r_o* signal. Umesto da se izlaz *r* registra vodi direktno na *r_o* izlaz, ovaj put ćemo ga odvesti na ulaz još jednog registra. Svrha ovog novog registra jeste da od spoljnjeg sveta sakrije sam način na koji se generiše proizvod, čineći ga unutrašnjim procesom modula *seq_mult*. Na ovaj način više neće biti moguće pratiti postupak formiranja proizvoda dva broja preko *r_o* signala. Izlazni signal *r_o* ažuriraće se samo kada je izračunat konačni proizvod. Ovo je inače tipična situacija prilikom projektovanja složenih sistema, gde se na izlaznim portovima postavljaju samo konačni rezultati izvršavanja implementiranog algoritma obrade podataka, a ne i međurezultati.

Druge dve izmene odnose se na način generisanja izlaznih signala, *r_o* i *ready_o*. Uvođenjem generičke konstante, *gen_correct_model_g*, moguće je kontrolisati način na koji se generišu ova dva signala i na taj način uvesti „bagove" u dizajn po želji. Na ovaj način biće moguće analizirati kako verifikaciono okruženje signalizira pojavu ovih bagova. Generička konstanta *gen_correct_model_g* definisana je kao 2-bitni vektor. Postavljanjem individualnih bitova na '0' moguće je aktivirati neispravan način generisanja *r*_o izlaznog signala (bit 0 unutar *gen_correct_model_g*), odnosno *ready_o* (bit 1 unutar *gen_correct_model_g*).

Signal *r_o* sadrži konačnu vrednost množenja dva zadata broja. Ukoliko se generička konstanta *gen_correct_model_g(0)* postavi na vrednost '0', ispravno će biti izračunate vrednosti množenja samo ukoliko su one manje od 255. Ukoliko su veće od 255 vrednosti množenja će biti izračunate pogrešno.

Signal *ready_o* označava kraj izvođenja operacije množenja dva zadata broja i treba da se aktivira u trenutku kada je postupak množenja završen i rezultat se pojavljuje na *r_o* portu. Korišćenjem generičke konstante *gen_correct_model_g(1)* moguće je zakasniti aktiviranje *ready_o* signala za jedan takt, što će rezultovati u nepravilno implementiranom protokolu generisanja izlaznih signala i što bi verifikaciono okruženje trebalo da detektuje. Ova mogućnost će biti korišćena kasnije da bi se ilustrovao rad *monitora* izlaznog protokola, koji će biti deo projektovanog verifikacionog okruženja.

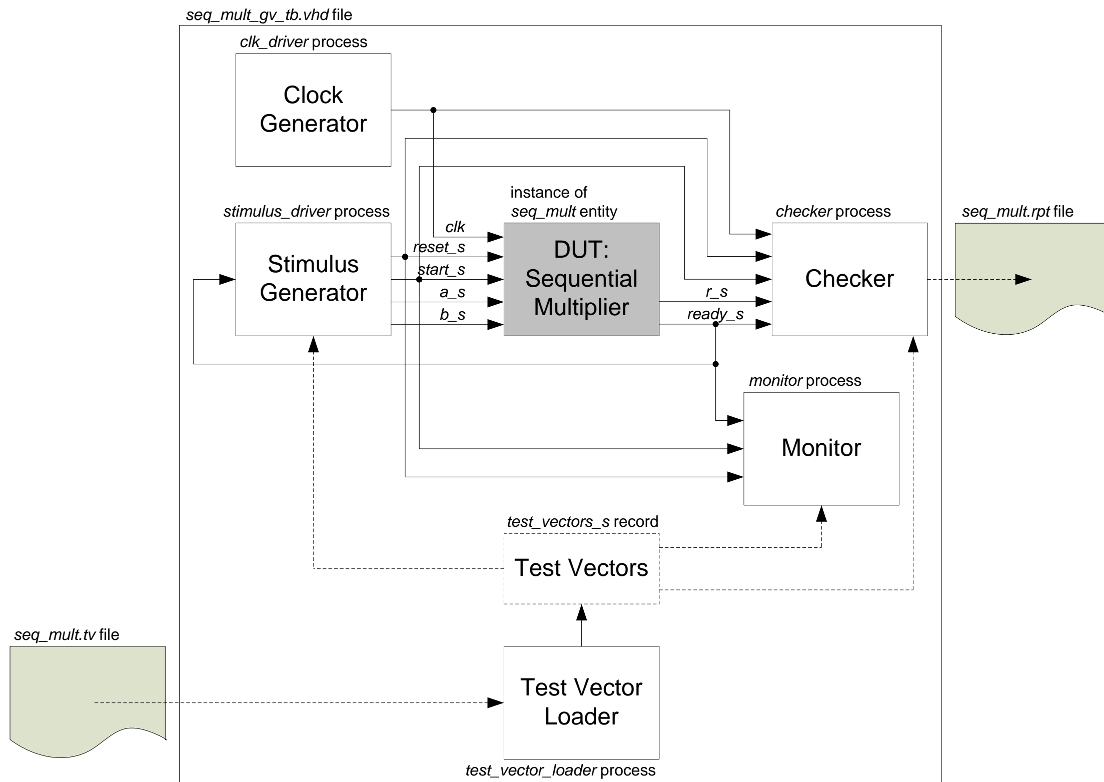
Kompletni blok dijagram projektovanog sekvencijalnog „Add-and-Shift" množača, sa ove dve modifikacije, prikazan je na Slici 1.



Slika 1. Blok dijagram modifikovanog „Add-and-Shift" množača koji je pogodan za ilustrovanje načina razvoja različitih verifikacionih okruženja

VHDL model modifikovanog „Add-and-Shift" množača nalazi se u datoteci *seq_mult.vhd*, koja se može pronaći u pratećoj arhivi, „Vezba 9 – Kreiranje verifikacionih okruzenja.rar" unutar „Sequential Multiplier - Golden Vectors" direktorijuma.

Pređimo sada na prikaz razvijenog verifikacionog okruženja za modifikovani „Add-and-Shift" množač, koje je bazirano na korišćenju „zlatnih vektora", pri čemu se DUT tretira kao „black box" komponenta. Blok dijagram verifikacionog okruženja prikazan je na Slici 2.



Slika 2. Blok dijagram verifikacionog okruženja za modifikovani „Add-and-Shift“ množač baziranog na korišćenju „zlatnih vektora“

Čitavo verifikaciono okruženje smešteno je unutar jednog VHDL fajla, *seq_mult_gv_tb.vhd*, koje se takođe može naći unutar prateće arhive.

Verifikaciono okruženje sastoji se iz većeg broja odvojenih procesa, od kojih svaki implementira jednu od standardnih verifikacionih komponenti, i odgovarajućih struktura podataka, neophodnih za pravilan rad:

Ime procesa	Funkcija
clk_driver	Proces zadužen za ispravno generisanje globalnog sinhronizacionog signala.
stimulus_generator	Proces koji generiše ulazne signale DUT-a, na osnovu zadatih vrednosti test vektora.
monitor	Proces koji nadgleda izlazne signale DUT-a i proverava da li je ispoštovan protokol generisanja izlaznih signala.
checker	Proces koji proverava da li DUT pravilno izračunao rezultat množenja dva zadata broja.
test_vector_loader	Proces koji učitava „zlatne vektore“.

Bitno je naglasiti da je svaka od verifikacionih komponenti mogla biti realizovana kao zasebni VHDL modul (*entity*) koji se nalazi smešten u odvojenom VHDL fajlu. Tada bi se u okviru *seq_mult_gv_tb* modula moralo izvršiti instancioniranje svake od ovih komponenti, slično instancioniranju DUT-a. Ovakav pristup ima smisla ukoliko su verifikacione komponente složene, pa svaku od njih razvija po jedan verifikacioni inženjer. Ovakav pristup odgovara hijerarhijskom pristupu projektovanja samog

digitalnog sistema, ali ovaj put primenjenom na projektovanje verifikacionog okruženja.

Za pravilan rad razvijenog verifikacionog okruženja, neophodno je postojanje jedne tekstualne datoteke, *seq_mult.tv*, u kojoj se nalaze smešteni „zlatni vektori“ koje je potrebno primeniti prilikom verifikacije DUT-a. Sam format ove datoteke je priozvoljan i određuje ga verifikacioni inženjer zadužen za njegov razvoj. U ovom primeru format datoteke *seq_mult.tv* ima sledeći oblik. „Zlatni vektori“ organizovani su po vrstama, pri čemu svaki „zlatni vektor“ zauzima jednu vrstu unutar datoteke. Svaki zlatni vektor sastoji se iz četiri odvojena polja:

Oznaka polja	Namena
A	Polje koje sadrži vrednost prvog operanda. Vrednost je zadata u heksadecimalnom brojnom sistemu, koristeći neoznačene brojeve.
B	Polje koje sadrži vrednost drugog operanda. Vrednost je zadata u heksadecimalnom brojnom sistemu, koristeći neoznačene brojeve.
Delay	Vreme, izraženo u periodama globalnog sinhronizacionog signala, koje mora proteći nakon završetka tekuće operacije množenja do iniciranja naredne operacije. Ova vrednost zapisana je u dekadnom brojnom sistemu.
Result	Očekivani rezultat koji se treba pojaviti na izlazima množača kada se na njegove ulaze dovedu operandi čije su vrednosti specificirane u poljima A i B.

Na primer, jedan mogući sadržaj *seq_mult.tv* datoteke mogao bi biti

```
%Test vectors file
%--------------------------------------------------------------------------
% A   B   Delay   Result
%--------------------------------------------------------------------------
  02  03    0     0006
  05  07    5     0023
  1F  0B   2      0155
```

Nakon opcionog zaglavlja koje čine sve linije koje počinju sa karakterom ‘%’, svaka naredna linija sadrži po jedan test vektor. Prve dve vrednosti predstavljaju vrednosti brojeva koje želimo pomnožiti, zapisane u heksadecimalnom formatu. Nakon ove dve vrednosti sledi vrednost, zapisana u dekadnom formatu. Poslednja vrednost predstavlja očekivani rezultat operacije množenja, predstavljen u heksadecimalnom formatu.

Tokom simulacije projektovano verifikaciono okruženje kreira jednu *log* datoteku, *seq_mult.rpt*, u koju se smeštaju razne informacije o toku simulacije. Tipično, korisnik ima mogućnost kontrole količine informacija koje će biti smeštene u ovu datoteku, od vrlo detaljnih do samo najvažnijih. Projektovano verifikaciono okruženje pruža mogućnost izbora između dva nivoa detalja: prvog, detaljnog, u kojem se sve informacije koje se prikupe tokom simulacije smeštaju u *log* datoteku, i drugog u kojem se u *log* datoteku smeštaju samo informacije vezane za situaciju kada se uoči pogrešan rad DUT-a prilikom aplikacije nekog test vektora.

Proučimo sada detaljnije sam način na koji je napisano verifikaciono okruženje.

Kao što se iz prethodne diskusije moglo zaključiti, razvijeno verifikaciono okruženje mora definisati čitav niz konstanti, signala i novih tipova podataka pomoću kojih će biti moguće realizovati sve planirane objekte. Ove deklaracije se nalaze u zaglavlju arhitekture *beh* modula *seq_mult_gv_tb*, a ovde su prikazane u nastavku.

```
constant CLK_PERIOD: natural := 200;
constant WIDTH: integer := 8;
constant MAX_TEST_VECTORS: integer := 100;

-- Declaration of the constant that will determine whether correct or incorrect
-- multiplier model should be used. Meaning of the individual bits in the
-- constant are:
-- bit 0 - If set to '0' multiplication results that are larger then 255 will be
--         calculated incorrectly. If set to '1' multiplier correctly calculates
--         all results.
-- bit 1 - If set to '0' ready_o signal will be delayed by one clock cycle. If
--         set to '1' correct ready_o signal is generated.
constant GEN_CORRECT_MODEL: std_logic_vector(1 downto 0) := "11";

-- Declaration of the constant that determines what will be written into report file
type report_type is (VERBOSE, ERR_ONLY);
constant REPORT_DETAIL: report_type := VERBOSE;

-- Declarations of signals necessary to implement testbench
signal clk_s: std_logic;
signal reset_s: std_logic := '0';
signal start_s: std_logic := '0';
signal a_s: std_logic_vector(WIDTH-1 downto 0);
signal b_s: std_logic_vector(WIDTH-1 downto 0);

signal ready_s: std_logic;
signal r_s: std_logic_vector(2*WIDTH-1 downto 0);

-- Declaration of the record type that will hold the test vectors
type test_vector is record
    a, b: std_logic_vector(WIDTH-1 downto 0);
    delay: natural;
    mult_duration: natural;
    exp_output: std_logic_vector(2*WIDTH-1 downto 0);
end record test_vector;

type test_vector_array is array (0 to MAX_TEST_VECTORS-1) of test_vector;

signal test_vectors_s: test_vector_array;
signal tv_count_s: integer := 0;

-- Open the file where test vectors are stored
file tv_file: text open read_mode is "..\seq_mult.tv";

-- Open the file where where report will be written
file report_file: text open write_mode is "..\seq_mult.rpt";
```

Konstanta *CLK_PERIOD* definiše trajanje periode globalnog sinhronizacionog signala koja će biti korišćena prilikom simulacije. Definisanje periode pomoću konstante, a ne njeno direktno korišćenje na mestima gde je to potrebno, omogućava korisniku da vrlo lako izvrši njenu izmenu i na taj način proveri rad DUT-a na različitim brzinama, što može biti neophodno.

Konstanta *WIDTH* određuje broj bitova pomoću kojih se predstavljaju vrednosti ulaznih operanada u množač. Konstanta *MAX_TEST_VECTORS* definiše maksimalan broj test vektora koji se može nalaziti u *seq_mult.tv* datoteci. Kao što će se videti

nešto kasnije, u ovom slučaju se kao struktura podataka koja se koristi za smeštanje test vektora koristi običan niz, umesto liste. Kao što je poznato niz je statička struktura podataka kod koje se unapred mora definisati njena maksimalna veličina, što je u ovom slučaju i urađeno, korišćenjem *MAX_TEST_VECTORS* konstante. Iako bi korišćenje dinamičke strukture (u ovom slučaju bi najpogodnija struktura bila jednostruko spregnuta lista) omogućilo da verifikaciono okruženje može da procesira proizvoljno veliki broj test vektora, ovde ona nije implementirana da bi se pojednostavio priloženi VHDL model.

Sledeća konstanta koja je definisana, *GEN_CORRECT_MODEL*, omogućava specificiranje da li želimo da koristimo ispravan model množača (*GEN_CORRECT_MODEL="11"*), ili pak želimo da koristimo model množača kod kojega je izlazni signal *ready_o* zakašnjen za jednu periodu takt signala (*GEN_CORRECT_MODEL="01"*) ili pak želimo model množača koji netačno računa vrednost proizvoda ukoliko je ona veća od 255 (*GEN_CORRECT_MODEL="10"*) ili pak model sa oba baga (*GEN_CORRECT_MODEL="00"*). Vrednost ove konstante biće prosleđena modelu množača preko generičke konstante, *gen_correct_model_g*, prilikom njegovog instancioniranja.

Poslednja konstanta koja je definisana, *REPORT_DETAIL*, omogućava nam da specificiramo koliko detalja treba da bude prisutno u *log* datoteci. U slučaju da ova konstanta ima vrednost *VERBOSE* sve informacije koje se prikupe tokom simulacije biće upisane u *seq_mult.rpt* datoteku. Ukoliko ova konstanta ima vrednost *ERR_ONLY*, u *seq_mult.rpt* datoteci nalaziće se samo informacije vezane za situacije kod kojih je detektovano nepravilno ponašanje DUT-a.

Nakon deklaracija ovih konstanti sledi čitav niz deklaracija signala koji služe za povezivanje različitih verifikacionih komponenti sa DUT-om.

Obzirom da se projektovano verifikaciono okruženje bazira na „zlatnim vektorima", neophodno je i postojanje odgovarajuće strukture podataka u kojoj će ovi vektori biti smešteni. O ovom verifikacionom okruženju to je funkcija objekta *test_vectors_s*. Kao što je već bilo rečeno, reč je zapravo o nizu podataka, od kojih je svaki tipa *test_vector*. Tip *test_vector* je zapravo struktura koja ima ukupno pet polja. Četiri polja (*a*, *b*, *delay* i *exp_output*) direktno korespondiraju odgovarajućim poljima iz *seq_mult.tv* datoteke, a polje *mult_duration* sadrži vrednost koja predstavlja broj taktova koji mora proteći da bi se završilo množenje dva zadata broja. Ovaj podatak će biti neophodan unutar *monitor* procesa da bi se moglo verifikovati poštovanje izlaznog protokola od strane DUT-a.

Na kraju deklarativnog dela nalaze se dve deklaracije pomoću kojih se omogućava pristup dvema datotekama: datoteci sa test vektorima, *seq_mult.tv*, preko *tv_file* objekta i *log* datoteci, *seq_mult.rpt*, preko *report_file* objekta.

Pogledajmo sada detaljnije strukturu svake od verifikacionih komponenti koje su prisutne u verifikacionom okruženju.

Proces *clk_driver*

Funkcija ovog procesa jeste da generiše periodični globalni sinhronizacioni signal sa unapred definisanom periodom ponavljanja. Ovo je standardni proces koji se periodično aktivira i prilikom svakog aktiviranja generiše narednu periodu sinhronizacionog signala.

```
clk_driver:
process
begin
    clk_s <= '0', '1' after (CLK_PERIOD/2) * 1 ns;
    wait for CLK_PERIOD * 1 ns;
end process clk_driver;
```

Proces *stimulus_generator*

Ovaj proces zadužen je za dovođenje narednog test vektora na ulaze DUT-a. Proces *stimulus_generator* u ovu svrhu pristupa objektu *test_vectors_s* koji zapravo sadrži niz test vektora koje je potrebno aplicirati na ulaze DUT-a. Struktura procesa prikazana je u nastavku.

```
stimulus_driver:
process
begin
    -- Reset the multiplier
    wait until falling_edge(clk_s);
    reset_s <= '1';

    -- Wait one clock cycle, then de-assert the reset_s signal
    wait until falling_edge(clk_s);
    reset_s <= '0';

    -- Start applying test vectors
    for i in 0 to tv_count_s - 1 loop
        wait until falling_edge(clk_s);

        a_s <= test_vectors_s(i).a;
        b_s <= test_vectors_s(i).b;

        -- Start the multiplier
        start_s <= '1';

        -- Wait one clock cycle, then de-assert the start_s signal
        wait until falling_edge(clk_s);
        start_s <= '0';

        -- Wait until multiplier multiplies the data
        wait until ready_s = '1';
        -- Wait predetermined number of clock cycles before appying next test vector
        for d in 1 to test_vectors_s(i).delay loop
            wait until rising_edge(clk_s);
        end loop;
    end loop;

    -- All test vectors have been applied, terminate the process
    wait;
end process stimulus_driver;
```

Na samom početku simulacije, vrši se resetovanja DUT-a, aktiviranjem *reset_s* signala. Trajanje reseta ograničeno je na jednu periodu globalnog sinhronizacionog signala, jer je u ovom slučaju to dovoljno dugo. U praksi je potrebno detaljno proučiti trajanje i način izvođenja inicijalizacije DUT-a jer ona može biti prilično složena i može trajati mnogo duže od jedne periode globalnog sinhronizacionog signala.

Nakon inicijalizacije DUT-a proce ulazi u petlju u kojoj se u svakoj iteraciji na ulaze DUT-a dovodi sledeći test vektor. Vrednosti narednog test vektora koje je potrebno dovesti na ulaze DUT-a preuzimaju se iz *test_vectors_s* objekta. Samo apliciranje

vrednosti izvodi se na opadajuću ivicu *clk_s* signala. Na ovaj način se izbegavaju potencijalni problemi sa *setup* i *hold* vremenima na ulazima u DUT-a. Nakon toga se startuje množač, aktiviranjem *start_s* signala u trajanju od jedne periode *clk_s* signala. Stimulus generator zatim čeka da množač završi operaciju množenja, što će biti signalizirano aktiviranjem *ready_s* izlaznog signala, a nakon toga čeka još specificirani broj perioda *clk_s* signala (broj perioda koji je potrebno čekati nalazi se unutar *test_vectors_s(i).delay* polja) pre nego što pristupi apliciranju sledećeg test vektora.

Nakon što je doveo sve test vektore na ulaze DUT-a, stimulus generator se trajno deaktivira.

Proces *monitor*

Funkcija ovog procesa je da nadgleda izlaze množača i kontinualno proverava da li je ispoštovan izlazni protokol. Izlazni protokol o kojem je reč odnosi se na način generisanja *ready_o* signala. Ovaj signal predstavlja indikaciju da li je množač spreman da prihvati nove podatke (*ready_o* = '1'), ili je zauzet računanjem proizvoda (*ready_o* = '0'). Prema specifikaciji *ready_o* signal treba da bude aktivan od trenutka kada se završi tekuća operacija izračunavanja proizvoda i rezultat postavi na izlazni port *r_o* pa sve do trenutka kada se započne sledeća operacija računanja proizvoda kao rezultat aktiviranja ulaznog signala *start_i*. Sve dok se vrši računanje proizvoda tekuća dva broja, *ready_o* signal treba da bude neaktivan. Obzirom da vreme računanja proizvoda dva zadata broja zavisi od njihovih vrednosti, vreme tokom kojega je signal *ready_o* neaktivan je promenljivo. Proces *monitor* ima zadatak da utvrdi da li se signal *ready_o* generiše poštujući ova pravila. *Monitor* proces ima sledeću strukturu.

```
monitor:
process
    variable tv_count_v: natural := 0;
begin
    -- Wait until reset_s becomes active
    wait until reset_s = '1';
    -- Wait until reset_s is deactivated
    wait until reset_s = '0';

    loop
        -- At the start, before the start_s goes active for the first tie, or after mult_duration
        -- clock cycles, in case of active multiplication operation, ready_s signal must set to
        -- one and stay high until start_s goes active again.
        loop
            wait until falling_edge(clk_s);
            assert ready_s = '1'
                report "Output protocol violation detected! Ready_o signal should be one!"
                severity warning;
            wait until rising_edge(clk_s);
            exit when start_s = '1';
        end loop;

        -- After a multiplication is started it will take WIDTH clock cycles
        -- to complete the multiplication. During this period ready_s signal
        -- must be zero.
        for i in 1 to test_vectors_s(tv_count_v).mult_duration loop
            wait until falling_edge(clk_s);
            assert ready_s = '0'
                report "Output protocol violation detected! Ready_o signal should be zero!"
                severity warning;
        end loop;
```

```
        -- Current test vector has been processed, proceed with the next one
        tv_count_v := tv_count_v + 1;
    end loop;
end process monitor;
```

Na početku monitor čeka da se završi proces inicijalizacije DUT-a, nakon koga signal *ready*_s mora biti aktivan, jer je množač spreman da prihvati prve brojeve koje treba pomnožiti. Proces zatim ulazi u beskonačnu petlju. Unutar ove petlje prvo se ulazi u sledeću beskončnu petlju u kojoj se u svakoj periodi takt signala proverava da li je *ready_s* aktivan. Ovo je situacija u kojoj množač treba da se nalazi pre nego što se prvi put aktivira signal *start_s*, odnosno kada se završi sa množenjem zadata dva broja. Ovo tvrđenje može se ispitati korišćenjem VHDL naredbe ***assert***. Nakon ključne reči *assert* navodi se test koji mora biti ispunjen (u ovom slučaju to je izraz *ready_s = '1'*). Ukoliko uslov nije ispunjen izvršava se deo koji sledi iza ključne reči ***report***. Ova situacija traje sve dok se *start_s* signal ne aktivira kada se izlazi iz beskonačne petlje korišćenjem ***exit*** naredbe.

Nakon aktiviranja *start_s* signala počevši od sledećeg perioda *clk_s* signala, pa narednih *test_vectors_s(tv_count_v).mult_duration* perioda, *ready_s* signal mora imati vrednost '0', jer toliko je vremena potrebno da se pomnože dva zadata broja pomoću sekvencijalnog „Add-and-Shift“ množača. Ova provera izvršava se unutar *for loop* petlje.

Nakon isteka ovoga vremena završava se jedna iteracija izlaznog protokola, i čitav protokol počinje iz početka.

Proces *checker*

Ovaj proces proverava da li je rezultat množenja dva ulazna broja, koji se nalazi na izlazu *r_o* DUT-a korektan. Da bi se izvršila ova provera, *checker* proces koristi podatke smeštene unutar objekta *test_vectors_s*.

```
checker:
process
    variable l: line;
    variable errors_present_v: boolean := false;
begin
    write (l, string'("Starting simulation!"));
    writeline (report_file, l);
    writeline (report_file, l); -- write one blank line

    -- Wait for the first test vector to be applied
    wait until start_s = '1';

    for i in 0 to tv_count_s - 1 loop
        wait until ready_s = '1';

        if (REPORT_DETAIL = VERBOSE) then
            write (l, string'("Applying test vector number: "));
            write (l, i + 1);
            writeline (report_file, l);
            write (l, string'("A = "));
            write (l, conv_integer(test_vectors_s(i).a), justified => left, field => 3);
            write (l, string'("    B = "));
            write (l, conv_integer(test_vectors_s(i).b), justified => left, field => 3);
            write (l, string'("    Expected result = "));
            write (l, conv_integer(test_vectors_s(i).exp_output), justified => left, field => 5);
            write (l, string'("    Actual result = "));
            write (l, conv_integer(r_s), justified => left, field => 5);

            -- Actual checking is done here
```

```
                if (test_vectors_s(i).exp_output = r_s) then
                    write (l, string'("    OK"));
                else
                    write (l, string'("    ERROR"));
                end if;
                writeline (report_file, l);
                writeline (report_file, l); -- write one blank line
            else
                -- Actual checking is here
                if (test_vectors_s(i).exp_output /= r_s) then
                    errors_present_v := true;
                    write (l, string'("Error when applying test vector number: "));
                    write (l, i + 1);
                    writeline (report_file, l);
                    write (l, string'("A = "));
                    write (l, conv_integer(test_vectors_s(i).a), justified => left, field => 3);
                    write (l, string'("    B = "));
                    write (l, conv_integer(test_vectors_s(i).b), justified => left, field => 3);
                    write (l, string'("    Expected result = "));
                    write (l, conv_integer(test_vectors_s(i).exp_output), justified => left, field => 5);
                    write (l, string'("    Actual result = "));
                    write (l, conv_integer(r_s), justified => left, field => 5);
                    writeline (report_file, l);
                    writeline (report_file, l); -- write one blank line
                end if;
            end if;
        end loop;

        if (REPORT_DETAIL = ERR_ONLY and errors_present_v = false) then
            write (l, string'("No errors found during simulation!"));
            writeline (report_file, l);
        end if;

        -- All test vectors have been applied and results have been checked,
        -- wait one more clock cycle then end the simulation
        wait until rising_edge(clk_s);
        report "ALL TEST VECTORS HAVE BEEN APPLIED TO THE DUT! ENDING SIMULATION!"
            severity failure;
    end process checker;
```

Na početku proces generiše zaglavlje unutar *log* datoteke, a zatim čeka da se *start_s* signal aktivira prvi put, što predstavlja indikaciju da je započet postupak množenja prva dva broja.

Proces zatim ulazi u *for loop* petlju u kojoj će čekati da se izvrši množenje svakog od test vektora, pri čemu će za svaki od njih proveriti da li se na izlazu DUT-a generiše očekivani rezultat. Unutar petlje čeka se da *read_s* signal postane aktivan, što predstavlja indikaciju da je tekuća operacija množenja završena i da se rezultat nalazi na izlaznom portu *r_o*. *Checker* proces generiše dve vrste izveštaja, u zavisnosti od vrednosti konstante *REPORT_DETAIL*.

Ukoliko je izabran detaljan izveštaj, u *log* datoteku smeštaju se informacije o tekućem test vektoru (vrednosti koje su dovedene na ulaze *a_i* i *b_i* DUT-a) kao i očekivana i stvarna vrednost izlaznog porta *r_o*. Nakon toga vrši se poređenje očekivane i stvarne vrednosti porta *r_o* i u *log* datoteku se upisuje informacija da li su one jednake ili različite. Izgled *log* datoteke, u ovom slučaju mogao bi biti sledeći.

```
Starting simulation!

Applying test vector number: 1
A = 2     B = 3     Expected result = 6      Actual result = 6      OK

Applying test vector number: 2
A = 5     B = 7     Expected result = 35     Actual result = 35     OK

Applying test vector number: 3
A = 31    B = 11    Expected result = 341    Actual result = 341    OK
```

U slučaju drugog režima, u *log* datoteku smeštaju samo informacije o test vektorima kod kojih je uočeno nepravilno generisanje rezultata množenja. U ovom slučaju *log* datoteka mogla bi imati sledeći sadržaj, u slučaju da je uočeno neslaganje između očekivane i stvarne vrednosti na izlaznom portu *r_o* prilikom primene test vektora sa rednim brojem 3.

```
Starting simulation!

Error when applying test vector number: 3
A = 31    B = 11    Expected result = 341    Actual result = 339
```

Nakon što se obrade svi test vektori, prekida se simulacija i korisnik obaveštava o tome.

Proces *test_vector_loader*

Test_vector_loader proces služi za učitavanje test vektora iz *seq_mult.tv* datoteke u *test_vectors_s* objekat. Ovaj proces aktivira se samo jednom, na samom početku simulacije, kada se poziva procedura *Load_Test_Vectors* koja zapravo učitava test vektore. Kada se završi izvršavanje ove procedure proces se deaktivira do samog kraja simulacije.

```
test_vectors_loader:
process
begin
    Load_Test_Vectors (test_vectors_s, tv_count_s);
    wait;
end process test_vectors_loader;
```

Procedura *Load_Test_Vectors* učitava test vektore iz *seq_mult.tv* datoteke u *test_vectors_s* objekat. Procedura procesira jednu po jednu liniju iz *seq_mult.tv* datoteke. Ukoliko linija počinje sa karakterom '%' ona se preskače, u protivnom se učitava vrednost prvog i drugog operanda.

Sledeći korak predstavlja izračunavanje vremena koje je neophodno da bi se odabrana dva broja pomnožila pomoću sekvencijalnog „Add-and-Shift“ množača. Ovo vreme zavisi od broja jedinica koje su prisutne u binarnoj reprezentaciji vrednosti *B* operanda.

Zatim se učitava vrednost koja predstavlja broj taktova koji treba da proteknu pre nego što se pređe na sledeći test vektor. Na kraju, učitava se i očekivana vrednost operacije množenja.

```
procedure Load_Test_Vectors (
                              signal tv_o: out test_vector_array;
                              signal num_tv_o: out integer
                                     ) is

    variable l: line;
    variable ch: character;

    variable b_v: std_logic_vector(WIDTH-1 downto 0);
    variable delay_v: natural;
    variable tv_count_v: integer := 0;

    variable num_of_ones_v: natural := 0;
begin
    while (not (endfile (tv_file))) loop
        -- Load new test vector, skip any lines with comments
        readline (tv_file, l);
        read (l, ch);
        while (ch = '%') loop
            readline (tv_file, l);
            read (l, ch);
        end loop;

        -- Load A operand
        while (ch = ' ') loop
            read (l, ch);
        end loop;
        tv_o(tv_count_v).a(WIDTH-1 downto WIDTH-4) <= Char_to_StdLogicVector (ch);
        read (l, ch);
        tv_o(tv_count_v).a(WIDTH-5 downto 0) <= Char_to_StdLogicVector (ch);

        -- Load B operand
        read (l, ch);
        while (ch = ' ') loop
            read (l, ch);
        end loop;
        b_v(WIDTH-1 downto WIDTH-4) := Char_to_StdLogicVector (ch);
        read (l, ch);
        b_v(WIDTH-5 downto 0) := Char_to_StdLogicVector (ch);
        tv_o(tv_count_v).b <= b_v;

        -- Calculate the number of clock cycles it will take to multiply these numbers
        num_of_ones_v := 0;
        for i in WIDTH - 1 downto 0 loop
            if b_v(i) = '1' then
                num_of_ones_v := num_of_ones_v + 1;
            end if;
        end loop;
        tv_o(tv_count_v).mult_duration <= 2*num_of_ones_v + WIDTH-num_of_ones_v;

        -- Load delay value
        read (l, delay_v);
        tv_o(tv_count_v).delay <= delay_v;

        -- Load expected result
        read (l, ch);
        while (ch = ' ') loop
            read (l, ch);
        end loop;
        tv_o(tv_count_v).exp_output(2*WIDTH-1 downto 2*WIDTH-4) <= Char_to_StdLogicVector (ch);
        read (l, ch);
        tv_o(tv_count_v).exp_output(2*WIDTH-5 downto 2*WIDTH-8) <= Char_to_StdLogicVector (ch);
        read (l, ch);
        tv_o(tv_count_v).exp_output(2*WIDTH-9 downto 2*WIDTH-12) <= Char_to_StdLogicVector (ch);
        read (l, ch);
        tv_o(tv_count_v).exp_output(2*WIDTH-13 downto 0) <= Char_to_StdLogicVector (ch);

        tv_count_v := tv_count_v + 1;
    end loop;
    num_tv_o <= tv_count_v;
end procedure Load_Test_Vectors;
```

Prilikom učitavanja vrednosti ulaznih operanada kao i očekivane vrednosti rezultata množenja, neophodno je konvertovati string koji predstavlja željenu vrednost, zapisanu u heksadecimalnom formatu, u binarni format neophodan za simulaciju. Ovo

se može postići učitavajući jedan po jedan karakter i konvertujući ga u 4-bitni binarni broj, korišćenjem funkcije *Char_to_StdLogicVector*.

```
function Char_to_StdLogicVector (
                        ch: in character
                              ) return std_logic_vector is

    variable result: std_logic_vector (3 downto 0);
begin
    case ch is
        when '0' => result := X"0";
        when '1' => result := X"1";
        when '2' => result := X"2";
        when '3' => result := X"3";
        when '4' => result := X"4";
        when '5' => result := X"5";
        when '6' => result := X"6";
        when '7' => result := X"7";
        when '8' => result := X"8";
        when '9' => result := X"9";
        when 'A' => result := X"A";
        when 'B' => result := X"B";
        when 'C' => result := X"C";
        when 'D' => result := X"D";
        when 'E' => result := X"E";
        when 'F' => result := X"F";
        when 'X' => result := "XXXX";
        when others => report "Not a number! Char_to_StdLogicVector."
                        severity error;
    end case;
    return result;
end function Char_to_StdLogicVector;
```

Funkcija *Char_to_StdLogicVector* konvertuje jedan karakter od odgovarajuću 4-bitnu binarnu vrednost pomoću jedne ***case*** naredbe.

Zadaci za vežbu

Zadatak 1:

Preraditi postojeće verifikaciono okruženje tako da se svaka od verifikacionih komponenti opisuje pomoću posebnog VHDL modula, a unutar *seq_mult_gv_tb* se one instancioniraju i međusobno povezuju.

Zadatak 2:

Koristeći verifikaciono okruženje iz primera 1 kao polaznu osnovu, napisati verifikaciono okruženje bazirano na „zlatnim vektorima" koje se može iskoristiti za verifikaciju modela serijskog sabirača iz Vežbe 7 (zadatak 4).

Zadatak 3:

Koristeći verifikaciono okruženje iz primera 1 kao polaznu osnovu, napisati verifikaciono okruženje bazirano na „zlatnim vektorima" koje se može iskoristiti za verifikaciju modela detektora sekvence iz Vežbe 7 (zadatak 5).

Zadatak 4:

Koristeći verifikaciono okruženje iz primera 1 kao polaznu osnovu, napisati verifikaciono okruženje bazirano na „zlatnim vektorima" koje se može iskoristiti za verifikaciju modela dvopristupne memorije iz Vežbe 6 (zadatak 2).

Zadatak 5:

Koristeći verifikaciono okruženje iz primera 1 kao polaznu osnovu, napisati verifikaciono okruženje bazirano na „zlatnim vektorima" koje se može iskoristiti za verifikaciju modela konvertora BCD koda u 7-segmentni kod iz Vežbe 6 (zadatak 4).

Tesbenčevi bazirani na referentnim modelima

Glavni nedostatak testbenčeva baziranih na „zlatnim vektorima" ogleda se u činjenici da je pre početka procesa potrebno generisati odgovarajući skup „zlatnih vektora". Ovaj postupak se često radi ručno, od strane verifikacionog inženjera, i uključuje ne samo osmišljavanje interesantnih vrednosti ulaznog stimulusa, već i izračunavanje očekivanog ponašanja sistema koji se verifikuje na ovakav stimulus. Pošto se često radi ručno, ovaj proces je spor i podložan greškama koje zatim usporavaju dalji proces verifikacije.

Način kako da se ovaj nedostatak pristupa baziranog na „zlatnim vektorima" može prevazići je korišćenje testbenčeva baziranih na referentim modelima.

Pod referentnim modelom podrazumevamo model sistema koji se verifikuje napisan na visokom nivou apstrakcije. Referentni model treba da poseduje željenu funkcionalnost koja je trebalo da bude implementirana unutar sistema koji se verifikuje. Obzirom da se referentni model piše na visokom nivou apstrakcije on je znatno jednostavniji od modela sistema koji se verifikuje što olakšava njegov razvoj i smanjuje mogućnost unošenja nenamernih grešaka prilikom njegovog pisanja. Referentni model se unutar testbenča koristi kao refernca (otuda i njegovo ima) koja služi za poređenje odziva dobijenih od strane sistema koji se verifikuje na dovedenu pobudu.

Proces verifikacije unutar testbenča baziranog na referentnom modelu odvija se na sledeći način. Unutar testbenča instancionirane su dve komponente: model sistema koji želimo da verifikujemo i njegov referentni model. Prilikom procesa simulacije na ulaze oba modela dovode se iste sekvence ulaznih vektora, a odzivi referentnog modela i modela sistema koji se verifikuje se zatim porede kako bi se utvrdilo da li postoji neko odstupanje. Obzirom da oba modela modeluju istu funkcionalnost odstupanja ne bi trebalo da bude. Ukoliko se da je do odstupanja došlo, to predstavlja indikaciju o postojanju greške unutar nekog od dva modela. Simulacija se u tom trenutku prekida, a zadatak verifikacionog inženjera je da utvrdi u kojem od modela se nalazi greška.

Svi referentni modeli mogu se podeliti u dve velike grupe, prema tome koliko precizno modeluju funkcionalnost koju sistem koji se verifikuje treba da poseduje:

- Transakcione referentne modele („*transaction level*" referentni modeli) – ovi referentni modeli modeluju očekivano ponašanje sistema koji se verifikuje do nivoa transakcija, a ne individualnih perioda (ciklusa) klok signala. Transkcija predstavlja apstrakciju procesa obrade podataka unutar nekog sistema, pri čemu se vremenski period potreban za završetak obrade zanemaruje. Jedino što nas kod transakcija interesuje jeste rezultat obrade nekog ulaznog podataka (šta god ta obrada podrazumevala), a ne i vremenski redosled koraka koji

dovodi do tog rezultata. Na primer, u slučaju sekvencijalnog „Add-and-Shift" množača proces množenja dva broja traje određeni broj taktova, tokom kojega se akumulira parcijalni proizvod. Transakcioni referentni model modelovao bi samo krajnji rezultat ove operacije, a ne i sve međurezultate tokom procesa množenja.

- Referentne modele na nivou ciklusa (*„cycle accurate"* referentni modeli) – ovi modeli, za razliku od transakcionih, modeluju očekivano ponašanje sistema koji se verifikuje sve do nivoa individulanih ciklusa klok signala. Na primer, *„cycle accurate"* referentni model sekvencijalnog „Add-and-Shift" množača morao bi da korektno modeluje i pojavljivanje svake međuvrednosti koja se javlja na izlazima množača, nakon svakog takta klok signala, prilikom izvođenja operacije množenja.

U nastavku će biti prikazani testbenčevi bazirani na „transaction level" i „cycle accurate" referentnim modelima koji se mogu koristiti za verifikaciju modela sekvencijalnog „Add-and-Shift" množača razvijenog u vežbi 8.

Primer 1: Testbenč baziran na „transaction-level" referentnom modelu za verifikaciju sekvencijalnog „Add-and-Shift" množača

U prethodnom delu predstavili smo jedno moguće rešenje za razvoj verifikacionog okruženja sekvencijalnog „Add-and-Shift" množača koje je bazirano na „zlatnim vektorima". Ovaj pristup zahteva od verifikacionog inženjera da definiše „zlatne vektore" – kombinacije vrednosti koje je potrebno dovesti na ulaze DUT-a praćene očekivanim vrednostima koje će se pojaviti na izlazima DUT-a. Iako relativno jednostavan koncept, metodologija „zlatnih vektora" nije praktična za verifikaciju složenijih sistema jer zahteva veliki manuelni rad za pripremu samih vektora, koji je samim tim podložan greškama. Takođe ovaj pristup je i izuzetno vremenski zahtevan, što može predstavljati ozbiljan problem u savremenim projektima koji su po pravilo vrlo restriktivni po pitanju vremena koje stoji na raspolaganju verifikacionim inženjerima da kompletiraju čitav verifikacioni ciklus. U ovoj vežbi razmatraćemo pristupe razvoju verifikacionog okruženja koji su bazirani na korišćenju referentnih modela. Po pravilu, ovi pristupi su znatno efikasniji od pristupa baziranog na „zlatnim vektorima".

Verifikaciona okruženja bazirana na referentnim modelima po pravilu moraju imati modul koji je u stanju da „predviđa" odziv DUT-a na bilo koju ulaznu kombinaciju. Ovaj modul zove se referentni model. Referentni model zapravo predstavlja model funkcionalnosti DUT-a ali ovaj put opisan na znatno višem apstraktnijem nivou od RTL-a, koji je korišćen prilikom projektovanja DUT-a. Na ovaj način je razvoj, testiranje i eventualna modifikacija referentnog modela znatno olakšana. Sa druge strane, verifikacioni inženjer sada samo treba da definiše samo ulazne kombinacije koje želi dovesti na ulaze DUT-a, a odzive će biti moguće predvideti pomoću referentnog modela. Ovo znatno ubrzava i olakšava razvoj različitih test scenarija, što verifikacionom inženjeru omogućava da u vremenskom intervalu DUT podvrgne znatno većem broju različitih test scenarija nego što bi to bilo moguće u slučaju korišćenja verifikacionog okruženja baziranog na „zlatnim vektorima".

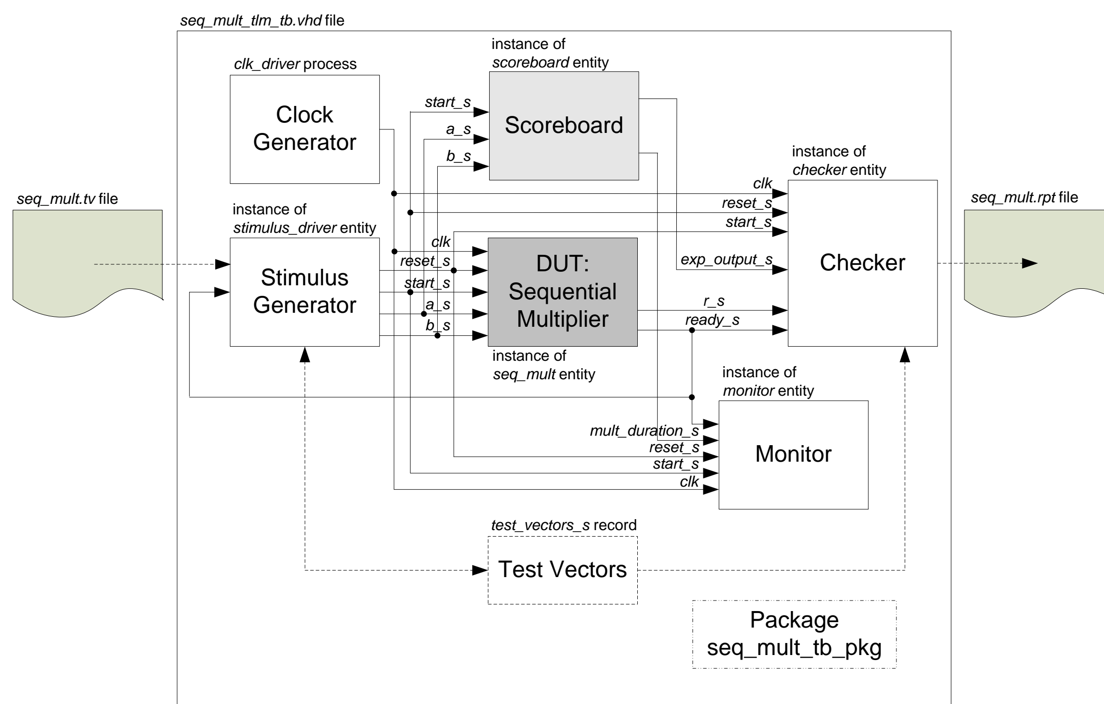
U praksi se najčešće koriste dve vrste referentnih modela:

1. *„transaction-level" referentni modeli* – ovi modeli predstavljaju vernu sliku funkcionalnosti DUT-a do nivoa pojedinačnih transakcija. Na ovaj način

apstrahovani su detalji rada DUT-a koji nisu bitni za samu funkcionalnost. Na primer, ukoliko je potrebno verifikovati rad nekog komunikacionog kontrolera to je moguće uraditi razvojem referentnog modela koji će korektno modelovati različite tipove *send* i *receive* transakcija. Svaka od ovih transakcija će se zapravo sastojati od čitavog niza vremenski precizno usklađenih operacija, ali ako nas interesuje samo konačan ishod svake transakcije, ovi detalji se mogu apstrahovati, odnosno, zanemariti.

2. *„cycle-accurate" referentni modeli* - kod ovih modela način rada DUT-a modelovan je do nivoa događaja koji se dešavaju nakon svake periode globalnog sinhronizacionog signala. Ovi modeli predstavljaju najtačnije modele digitalnih sistema koje je moguće razviti.

U ovom primeru prikazaćemo jedan mogući način razvoja verifikacionog okruženja, baziranog na „transaction-level" referentnom modelu, za modifikovani sekvencijalni „Add-and-Shift" množač koji omogućava kontrolisano uvođenje bagova u dizajn, kao što je objašnjeno u prethodnoj vežbi. Blok dijagram verifikacionog okruženja prikazan je na Slici 1. VHDL fajlovi koji čine ovo verifikaciono okruženje mogu se pronaći unutar arhive „Vezba 10 – Kreiranje verifikacionih okruzenja.rar", unutar *transaction_level_model* direktorijuma.



Slika 3. Blok dijagram verifikacionog okruženja za modifikovani „Add-and-Shift" množač, baziranog na korišćenju „transaction-level" referentnog modela

Kao što se sa slike 3 može videti, osnovna struktura verifikacionog okruženja baziranog na „transaction-level" referentnom modelu ne razlikuje se mnogo od verifikacionog okruženja baziranog na „zlatnim vektorima", razmatranog ranije. Međutim, ovaj put je čitavo verifikaciono okruženje smešteno je unutar većeg broja zasebnih VHDL modula koji su međusobno povezani unutar „top-level" modula, *seq_mult_tlm_tb.vhd*, koje se takođe može naći unutar prateće arhive.

Svaka od standardnih verifikacionih komponenti koja predstavlja deo razvijenog verifikacionog okruženja definisana je kao odvojeni modul (*entity*) koristeći poseban VHDL fajl.

Ime modula	Funkcija
stimulus_generator	Entitet koji generiše ulazne signale DUT-a, na osnovu zadatih vrednosti test vektora.
monitor	Entitet koji nadgleda izlazne signale DUT-a i proverava da li je ispoštovan protokol generisanja izlaznih signala.
checker	Entitet koji proverava da li je DUT pravilno izračunao rezultat množenja dva zadata broja.
scoreboard	Entitet koji predstavlja „transaction-level“ referentni model sekvencijalnog „Add-and-Shift“ množača.

Za pravilan rad razvijenog verifikacionog okruženja, neophodno je postojanje jedne tekstualne datoteke, *seq_mult.tv*, u kojoj se nalaze smešteni ulazni test vektori koje je potrebno primeniti prilikom verifikacije DUT-a. Sam format ove datoteke je priozvoljan i određuje ga verifikacioni inženjer zadužen za njegov razvoj. U ovom primeru format datoteke *seq_mult.tv* ima sledeći oblik. Ulazni test vektori organizovani su po vrstama, pri čemu svaki test vektor zauzima jednu vrstu unutar datoteke. Svaki test vektor sastoji se iz tri odvojena polja:

Oznaka polja	Namena
A	Polje koje sadrži vrednost prvog operanda. Vrednost je zadata u heksadecimalnom brojnom sistemu, koristeći neoznačene brojeve.
B	Polje koje sadrži vrednost drugog operanda. Vrednost je zadata u heksadecimalnom brojnom sistemu, koristeći neoznačene brojeve.
Delay	Vreme, izraženo u periodama globalnog sinhronizacionog signala, koje mora proteći nakon završetka tekuće operacije množenja do iniciranja naredne operacije. Ova vrednost zapisana je u dekadnom brojnom sistemu.

Na primer, jedan mogući sadržaj *seq_mult.tv* datoteke mogao bi biti

```
%Test vectors file
%-----------------------------------------------------------------------------
% A   B   Delay
%-----------------------------------------------------------------------------
  02  03    0
  05  07    5
  1F  0B    2
```

Nakon opcionog zaglavlja koje čine sve linije koje počinju sa karakterom ‘%’, svaka naredna linija sadrži po jedan test vektor. Prve dve vrednosti predstavljaju vrednosti brojeva koje želimo pomnožiti, zapisane u heksadecimalnom formatu. Nakon ove dve vrednosti sledi vrednost, zapisana u dekadnom formatu. Kao što se može primetiti, format ove datoteke gotovo je identičan onom iz Vežbe 10, sa jednom razlikom, u ovom slučaju ne postoji kolona „*Result*“, obzirom da će se očekivane vrednosti izračunati automatski, korišćenjem referentnog modela.

Tokom simulacije projektovano verifikaciono okruženje kreira jednu *log* datoteku, *seq_mult.rpt*, u koju se smeštaju razne informacije o toku simulacije. Tipično,

korisnik ima mogućnost kontrole količine informacija koje će biti smeštene u ovu datoteku, od vrlo detaljnih do samo najvažnijih. Projektovano verifikaciono okruženje pruža mogućnost izbora između dva nivoa detalja: prvog, detaljnog, u kojem se sve informacije koje se prikupe tokom simulacije smeštaju u *log* datoteku, i drugog u kojem se u *log* datoteku smeštaju samo informacije vezane za situaciju kada se uoči pogrešan rad DUT-a prilikom aplikacije nekog test vektora.

Proučimo sada detaljnije sam način na koji je napisano verifikaciono okruženje.

Kao što se iz prethodne diskusije moglo zaključiti, razvijeno verifikaciono okruženje mora definisati čitav niz konstanti, signala i novih tipova podataka pomoću kojih će biti moguće realizovati sve planirane objekte. U Vežbi 10 ove deklaracije su se nalazile u zaglavlju arhitekture *beh* modula *seq_mult_gv_tb*, a ovde ćemo prikazati alternativan način, korišćenjem VHDL *package*-a. Sve konstante, strukture podataka i funkcije koje su neophodne definisane su u okviru paketa *seq_mult_tb_pkg.vhd* čiji je sadržaj prikazan u nastavku.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

package seq_mult_tb_pkg is
    constant CLK_PERIOD: natural := 200;
    constant OP_WIDTH: integer := 8;
    constant MAX_TEST_VECTORS: integer := 100;

    -- Declaration of the constant that will determine whether correct or incorrect
    -- multiplier model should be used. Meaning of the individual bits in the
    -- constant are:
    -- bit 0 - If set to '0' multiplication results that are larger then 255 will be
    --       calculated incorrectly. If set to '1' multiplier correctly calculates
    --       all results.
    -- bit 1 - If set to '0' ready_o signal will be delayed by one clock cycle. If
    --       set to '1' correct ready_o signal is generated.
    constant GEN_CORRECT_MODEL: std_logic_vector(1 downto 0) := "11";

    -- Declaration of the constant that determines what will be
    -- written into report file
    type report_type is (VERBOSE, ERR_ONLY);
    constant REPORT_DETAIL: report_type := VERBOSE;

    -- Declaration of the record type that will hold the test vectors
    type test_vector is record
        a, b: std_logic_vector(OP_WIDTH-1 downto 0);
        delay: natural;
    end record test_vector;

    type test_vector_array is array (0 to MAX_TEST_VECTORS-1) of test_vector;

    signal test_vectors_s: test_vector_array;
    signal tv_count_s: integer := 0;

    -- Filename of the input test vectors file
    constant TV_FILENAME: string := "..\seq_mult.tv";

    -- Filename of the report file
    constant REPORT_FILENAME: string := "..\seq_mult.rpt";

    function Char_to_StdLogicVector (ch: in character) return std_logic_vector;
```

```
procedure Load_Test_Vectors (
                              filename: in string;
                              signal tv_o: out test_vector_array;
                              signal num_tv_o: out integer
                             );

end;
```

Konstanta *CLK_PERIOD* definiše trajanje periode globalnog sinhronizacionog signala koja će biti korišćena prilikom simulacije. Definisanje periode pomoću konstante, a ne njeno direktno korišćenje na mestima gde je to potrebno, omogućava korisniku da vrlo lako izvrši njenu izmenu i na taj način proveri rad DUT-a na različitim brzinama, što može biti neophodno.

Konstanta *WIDTH* određuje broj bitova pomoću kojih se predstavljaju vrednosti ulaznih operanada u množač. Konstanta *MAX_TEST_VECTORS* definiše maksimalan broj test vektora koji se može nalaziti u *seq_mult.tv* datoteci. Kao što će se videti nešto kasnije, u ovom slučaju se kao struktura podataka koja se koristi za smeštanje test vektora koristi običan niz, umesto liste. Kao što je poznato niz je statička struktura podataka kod koje se unapred mora definisati njena maksimalna veličina, što je u ovom slučaju i urađeno, korišćenjem *MAX_TEST_VECTORS* konstante. Iako bi korišćenje dinamičke strukture (u ovom slučaju bi najpogodnija struktura bila jednostruko spregnuta lista) omogućilo da verifikaciono okruženje može da procesira proizvoljno veliki broj test vektora, ovde ona nije implementirana da bi se pojednostavio priloženi VHDL model.

Sledeća konstanta koja je definisana, *GEN_CORRECT_MODEL*, omogućava specificiranje da li želimo da koristimo ispravan model množača (*GEN_CORRECT_MODEL="11"*), ili pak želimo da koristimo model množača kod kojega je izlazni signal *ready_o* zakašnjen za jednu periodu takt signala (*GEN_CORRECT_MODEL="01"*) ili pak želimo model množača koji netačno računa vrednost proizvoda ukoliko je ona veća od 255 (*GEN_CORRECT_MODEL="10"*) ili pak model sa oba baga (*GEN_CORRECT_MODEL="00"*). Vrednost ove konstante biće prosleđena modelu množača preko generičke konstante, *gen_correct_model_g*, prilikom njegovog instancioniranja.

Poslednja konstanta koja je definisana, *REPORT_DETAIL*, omogućava nam da specificiramo koliko detalja treba da bude prisutno u *log* datoteci. U slučaju da ova konstanta ima vrednost *VERBOSE* sve informacije koje se prikupe tokom simulacije biće upisane u *seq_mult.rpt* datoteku. Ukoliko ova konstanta ima vrednost *ERR_ONLY*, u *seq_mult.rpt* datoteci nalaziće se samo informacije vezane za situacije kod kojih je detektovano nepravilno ponašanje DUT-a.

Nakon deklaracija ovih konstanti sledi čitav niz deklaracija signala koji služe za povezivanje različitih verifikacionih komponenti sa DUT-om.

Obzirom da se projektovano verifikaciono okruženje bazira na referentnom modelu pri čemu će se ulazni test vektori učitati iz odgovarajuće datoteke kreirane od strane verifikacionog inženjera, neophodno je i postojanje odgovarajuće strukture podataka u kojoj će ovi vektori biti smešteni. O ovom verifikacionom okruženju to je funkcija objekta *test_vectors_s*. Kao i u Vežbi 10, reč je zapravo o nizu podataka, od kojih je svaki tipa *test_vector*. Tip *test_vector* je zapravo struktura koja ovaj put ima tri polja koja direktno korespondiraju odgovarajućim poljima iz *seq_mult.tv* datoteke.

U nastavku se nalaze dve deklaracije konstanti pomoću kojih se omogućava pristup dvema datotekama: datoteci sa test vektorima, *seq_mult.tv*, preko *tv_file* objekta i *log* datoteci, *seq_mult.rpt*, preko *report_file* objekta.

Na kraju paketa se nalaze deklaracije funkcije koja služi za konverziju *char* tipa u *std_logic_vector* i procedure pomoću koje se učitavaju test vektori. Ova dva podprograma definisana su u nastavku VHDL fajla, u odgovarajućem telu paketa. Obzirom da su ova dva podprograma gotovo identična onim koji su korišćeni u Vežbi 10, nećemo ih ovde ponavljati.

Čitavo verifikaciono okruženje, zajedno sa DUT-om, opisano je u okviru *seq_mult_tlm_tb* entiteta. Njegova struktura prkazana je u nastavku.

```
entity seq_mult_tlm_tb is
end seq_mult_tlm_tb;

architecture beh of seq_mult_tlm_tb is
    -- Declarations of signals necessary to implement testbench
    signal clk_s: std_logic;
    signal reset_s: std_logic := '0';
    signal start_s: std_logic := '0';
    signal a_s: std_logic_vector(OP_WIDTH-1 downto 0);
    signal b_s: std_logic_vector(OP_WIDTH-1 downto 0);

    signal ready_s: std_logic;
    signal r_s: std_logic_vector(2*OP_WIDTH-1 downto 0);
    signal exp_output_s: std_logic_vector(2*OP_WIDTH-1 downto 0);
    signal mult_duration_s: natural;
begin
    ---------------------------------------------------------------------------
    -- Process that drives global clock signal
    ---------------------------------------------------------------------------
    clk_driver:
    process
    begin
        clk_s <= '0', '1' after (CLK_PERIOD/2) * 1 ns;
        wait for CLK_PERIOD * 1 ns;
    end process clk_driver;
    ---------------------------------------------------------------------------
    -- Stimulus generator component.
    ---------------------------------------------------------------------------
    stim_gen:
    entity work.stimulus_generator(beh)
        generic map (
                      tv_filename_g => TV_FILENAME
                     )
        port map (clk => clk_s,
                  ready_i => ready_s,
                  reset_o => reset_s,
                  start_o => start_s,
                  a_o => a_s,
                  b_o => b_s
                 );
    ---------------------------------------------------------------------------
    -- Monitor component.
    ---------------------------------------------------------------------------
    monitor:
    entity work.monitor(beh)
        port map (clk => clk_s,
                  reset_i => reset_s,
                  start_i => start_s,
                  ready_i => ready_s,
                  mult_duration_i => mult_duration_s
                 );
    ---------------------------------------------------------------------------
    -- Checker component.
```

```
--------------------------------------------------------------------------
checker:
entity work.checker(beh)
     generic map (
                     rpt_filename_g => REPORT_FILENAME
                    )
      port map (clk => clk_s,
                  reset_i => reset_s,
                  start_i => start_s,
                  ready_i => ready_s,
                  r_i => r_s,
                  exp_output_i => exp_output_s
                 );
--------------------------------------------------------------------------
-- Scoreboard component.
--------------------------------------------------------------------------
scoreboard:
entity work.scoreboard(beh)
     port map (start_i => start_s,
                 a_i => a_s,
                 b_i => b_s,
                 exp_output_o => exp_output_s,
                 mult_duration_o => mult_duration_s
                );
--------------------------------------------------------------------------
-- DUT component. Sequential multiplier
--------------------------------------------------------------------------
dut:
entity work.seq_mult(shift_add_arch)
     generic map (
                     gen_correct_model_g => GEN_CORRECT_MODEL
                    )
     port map (clk => clk_s,
                 reset_i => reset_s,
                 start_i => start_s,
                 a_i => a_s,
                 b_i => b_s,
                 ready_o => ready_s,
                 r_o => r_s
                );
end architecture beh;
```

Kao što se može videti, ovaj put je čitavo verifikaciono okruženje modelovano korišćenjem strukturnog pristupa, instancioniranjem i povezivanjem odgovarajućih verifikacionih komponenti i DUT-a, kao što je prikazano na slici 3.

Pogledajmo sada detaljnije strukturu svake od verifikacionih komponenti koje su prisutne u verifikacionom okruženju.

Proces *clk_driver*

Funkcija ovog procesa jeste da generiše periodični globalni sinhronizacioni signal sa unapred definisanom periodom ponavljanja.

Modul *stimulus_generator*

Modul *stimulus_gnerator* zadužen je za dovođenje narednog test vektora na ulaze DUT-a. S*timulus_generator* u ovu svrhu pristupa objektu *test_vectors_s* koji zapravo sadrži niz test vektora koje je potrebno aplicirati na ulaze DUT-a. Struktura modula prikazana je u nastavku.

```
entity stimulus_generator is
    generic (
            tv_filename_g: string
            );
    port (clk:      in std_logic;
          ready_i: in std_logic;

          reset_o: out std_logic;
          a_o:     out std_logic_vector(OP_WIDTH-1 downto 0);
          b_o:     out std_logic_vector(OP_WIDTH-1 downto 0);
          start_o: out std_logic
     );
end stimulus_generator;

architecture beh of stimulus_generator is
begin
    stimulus_driver:
    process
    begin
        -- Load the input test vectors
        Load_Test_Vectors (tv_filename_g, test_vectors_s, tv_count_s);

        -- Reset the multiplier
        wait until falling_edge(clk);
        reset_o <= '1';
        -- Wait one clock cycle, then de-assert the reset_o signal
        wait until falling_edge(clk);
        reset_o <= '0';

        -- Start applying test vectors
        for i in 0 to tv_count_s - 1 loop
            wait until falling_edge(clk);
            a_o <= test_vectors_s(i).a;
            b_o <= test_vectors_s(i).b;

            -- Start the multiplier
            start_o <= '1';

            -- Wait one clock cycle, then de-assert the start_o signal
            wait until falling_edge(clk);
            start_o <= '0';

            -- Wait until multiplier multiplies the data
            wait until ready_i = '1';

            -- Wait predetermined number of clock cycles before appying next test
            -- vector
            for d in 1 to test_vectors_s(i).delay loop
                wait until rising_edge(clk);
            end loop;
        end loop;

        -- All test vectors have been applied, terminate the process
        wait;
    end process stimulus_driver;
end architecture beh;
```

Na samom početku simulacije, pomoću procedure *Load_Test_Vectors*, vrši se učitavanje ulaznih test vektora u niz *test_vectors_s*, a zatim se vrši resetovanje DUT-a, aktiviranjem *reset_s* signala. Trajanje reseta ograničeno je na jednu periodu globalnog sinhronizacionog signala, jer je u ovom slučaju to dovoljno dugo.

Nakon inicijalizacije DUT-a ulazi se u *for* petlju u kojoj se u svakoj iteraciji na ulaze DUT-a dovodi sledeći test vektor. Vrednosti narednog test vektora koje je potrebno dovesti na ulaze DUT-a preuzimaju se iz *test_vectors_s* objekta. Samo apliciranje vrednosti izvodi se na opadajuću ivicu *clk_s* signala. Na ovaj način se izbegavaju potencijalni problemi sa *setup* i *hold* vremenima na ulazima u DUT-a. Nakon toga se startuje množač, aktiviranjem *start_s* signala u trajanju od jedne periode *clk_s* signala. Stimulus generator zatim čeka da množač završi operaciju množenja, što će biti signalizirano aktiviranjem *ready_s* izlaznog signala, a nakon toga čeka još specificirani broj perioda *clk_s* signala (broj perioda koji je potrebno čekati nalazi se unutar *test_vectors_s(i).delay* polja) pre nego što pristupi apliciranju sledećeg test vektora.

Nakon što je doveo sve test vektore na ulaze DUT-a, stimulus generator se trajno deaktivira.

Kao što se može videti iz strukture modula, ona je gotovo identična sa strukturom *stimulus_generator* procesa koji je korišćen u testbenču baziranom na „zlatnim vektorima".

Modul *monitor*

Funkcija ovog modula je da nadgleda izlaze množača i kontinualno proverava da li je ispoštovan izlazni protokol. Izlazni protokol o kojem je reč odnosi se na način generisanja *ready_o* signala. Ovaj signal predstavlja indikaciju da li je množač spreman da prihvati nove podatke (*ready_o* = '1'), ili je zauzet računanjem proizvoda (*ready_o* = '0'). Prema specifikaciji *ready_o* signal treba da bude aktivan od trenutka kada se završi tekuća operacija izračunavanja proizvoda i rezultat postavi na izlazni port *r_o* pa sve do trenutka kada se započne sledeća operacija računanja proizvoda kao rezultat aktiviranja ulaznog signala *start_i*. Sve dok se vrši računanje proizvoda tekuća dva broja, *ready_o* signal treba da bude neaktivan. Obzirom da vreme računanja proizvoda dva zadata broja zavisi od njihovih vrednosti, vreme tokom kojega je signal *ready_o* neaktivan je promenljivo. Modul *monitor* ima zadatak da utvrdi da li se signal *ready_o* generiše poštujući ova pravila.

Modul m*onitor* ima strukturu koja je gotovo identična strukturi procesa *monitor* iz testbenča baziranog na „zlatnim vektorima", sa jednom bitnom razlikom. U testbenču baziranom na „zlatnim vektorima", podatak o trajanju tekuće operacije množenja bio je deo objekta *test_vectors_s* (*.mult_duration*) i izračunavao se tokom učitavanja test vektora u okviru procedure *Load_Test_Vectors*. U verifikacionom okruženju baziranom na referentnom modelu ovaj podatak izračunavaće se tokom simulacije, pojedinačno za svaki test vektor koji je doveden na ulaze DUT-a u modulu *scoreboard*, a zatim će biti prosleđen modulu *monitor* preko *mult_duration_i* porta, što se može videti i na slici 3.

```
entity monitor is
    port (clk:             in std_logic;
          reset_i:         in std_logic;
          start_i:         in std_logic;
          ready_i:         in std_logic;
          mult_duration_i: in natural
         );
end monitor;

architecture beh of monitor is
begin
    monitor:
    process
```

```
        variable tv_count_v: natural := 0;
    begin
        -- Wait until reset_i becomes active
        wait until reset_i = '1';
        -- Wait until reset_i is deactivated
        wait until reset_i = '0';

        loop
            -- At the start, before the start_i goes active for the first tie, or after mult_duration
            -- clock cycles, in case of active multiplication operation, ready_i signal must set
            -- to one and stay high until start_i goes active again.
            loop
                wait until falling_edge(clk);
                assert ready_i = '1'
                    report "Output protocol violation detected! Ready_o signal should be one!"
                        severity warning;
                wait until rising_edge(clk);
                exit when start_i = '1';
            end loop;

            -- After a multiplication is started it will take mult_duration clock cycles
            -- to complete the multiplication. During this period ready_i signal
            -- must be zero.
            for i in 1 to mult_duration_i loop
                wait until falling_edge(clk);
                assert ready_i = '0'
                  report "Output protocol violation detected! Ready_o signal should be zero!"
                      severity warning;
        end loop;

            -- Current test vector has been processed, proceed with the next one
            tv_count_v := tv_count_v + 1;
        end loop;
    end process monitor;
end architecture beh;
```

Na početku monitor čeka da se završi proces inicijalizacije DUT-a, nakon koga signal *ready_*s mora biti aktivan, jer je množač spreman da prihvati prve brojeve koje treba pomnožiti. Proces zatim ulazi u beskonačnu petlju. Unutar ove petlje prvo se ulazi u sledeću beskončnu petlju u kojoj se u svakoj periodi takt signala proverava da li je *ready_s* aktivan. Ovo je situacija u kojoj množač treba da se nalazi pre nego što se prvi put aktivira signal *start_s*, odnosno kada se završi sa množenjem zadata dva broja. Ovo tvrđenje može se ispitati korišćenjem VHDL naredbe ***assert***. Nakon ključne reči *assert* navodi se test koji mora biti ispunjen (u ovom slučaju to je izraz *ready_s = '1'*). Ukoliko uslov nije ispunjen izvršava se deo koji sledi iza ključne reči ***report*.** Ova situacija traje sve dok se *start_s* signal ne aktivira kada se izlazi iz beskonačne petlje korišćenjem ***exit*** naredbe.

Nakon aktiviranja *start_s* signala počevši od sledećeg perioda *clk_s* signala, pa narednih *test_vectors_s(tv_count_v).mult_duration* perioda, *ready_s* signal mora imati vrednost '0', jer toliko je vremena potrebno da se pomnože dva zadata broja pomoću sekvencijalnog „Add-and-Shift" množača. Ova provera izvršava se unutar *for loop* petlje.

Nakon isteka ovoga vremena završava se jedna iteracija izlaznog protokola, i čitav protokol počinje iz početka.

Modul *checker*

Funkcija ovog modula je da izvrši proveru da li je rezultat množenja dva ulazna broja, koji se nalazi na izlazu *r_o* DUT-a korektan. Da bi se izvršila ova provera, *checker* modul koristi estimaciju rezultata množenja, dobijenu pomoću referentnog modela, *exp_output_i*.

```
entity checker is
    generic (
            rpt_filename_g: string
            );
    port (clk:          in std_logic;
          reset_i:      in std_logic;
          start_i:      in std_logic;
          ready_i:      in std_logic;
          r_i:          in std_logic_vector(2*OP_WIDTH-1 downto 0);
          exp_output_i: in std_logic_vector(2*OP_WIDTH-1 downto 0)
         );
end checker;
architecture beh of checker is
begin
    checker:
    process
        variable l: line;
        variable errors_present_v: boolean := false;

        file report_file_f: text;
    begin
        -- Open the report file
        file_open (report_file_f, rpt_filename_g, write_mode);

        write (l, string'("Starting simulation!"));
        writeline (report_file_f, l);
        writeline (report_file_f, l); -- write one blank line

        -- Wait for the first test vector to be applied
        wait until start_i = '1';

        for i in 0 to tv_count_s - 1 loop
            wait until ready_i = '1';

            if (REPORT_DETAIL = VERBOSE) then
                write (l, string'("Applying test vector number: "));
                write (l, i + 1);
                writeline (report_file_f, l);
                write (l, string'("A = "));
                write (l, conv_integer(test_vectors_s(i).a), justified => left, field => 3);
                write (l, string'("    B = "));
                write (l, conv_integer(test_vectors_s(i).b), justified => left, field => 3);
                write (l, string'("    Expected result = "));
                write (l, conv_integer(exp_output_i), justified => left, field => 5);
                write (l, string'("    Actual result = "));
                write (l, conv_integer(r_i), justified => left, field => 5);
                -- Actual checking is here
                if (exp_output_i = r_i) then
                    write (l, string'("    OK"));
                else
                    write (l, string'("    ERROR"));
                end if;
                writeline (report_file_f, l);
                writeline (report_file_f, l); -- write one blank line
            else
                -- Actual checking is done here
                if (exp_output_i /= r_i) then
                    errors_present_v := true;
                    write (l, string'("Error when applying test vector number: "));
                    write (l, i + 1);
                    writeline (report_file_f, l);
                    write (l, string'("A = "));
```

```
                write (l, conv_integer(test_vectors_s(i).a), justified => left, field => 3);
                write (l, string'("    B = "));
                write (l, conv_integer(test_vectors_s(i).b), justified => left, field => 3);
                write (l, string'("    Expected result = "));
                write (l, conv_integer(exp_output_i), justified => left, field => 5);
                write (l, string'("    Actual result = "));
                write (l, conv_integer(r_i), justified => left, field => 5);
                writeline (report_file_f, l);
                writeline (report_file_f, l); -- write one blank line
            end if;
        end if;
    end loop;

    if (REPORT_DETAIL = ERR_ONLY and errors_present_v = false) then
        write (l, string'("No errors found during simulation!"));
        writeline (report_file_f, l);
    end if;

    -- All test vectors have been applied and results have been checked,
    -- wait one more clock cycle then end the simulation
    wait until rising_edge(clk);
    -- Close the report file
    file_close (report_file_f);

    -- End the simulation
    report "ALL TEST VECTORS HAVE BEEN APPLIED TO THE DUT! ENDING SIMULATION!"
        severity failure;
  end process checker;
end architecture beh;
```

Sama struktura *checker* modula identična je strukturi procesa *checker* iz testbenča baziranog na „zlatnim vektorima“ sa jednom razlikom. Očekivana vrednost operacije množenja ovaj put nije deo strukture *test_vectors_s*, kao što je bio slučaj u testbenču baziranom na „zlatnim vektorima“, već se ona izračunava tokom simulacije pomoću referentnog modela.

Na početku proces generiše zaglavlje unutar *log* datoteke, a zatim čeka da se *start_s* signal aktivira prvi put, što predstavlja indikaciju da je započet postupak množenja prva dva broja.

Proces zatim ulazi u *for loop* petlju u kojoj će čekati da se izvrši množenje svakog od test vektora, pri čemu će za svaki od njih proveriti da li se na izlazu DUT-a generiše očekivani rezultat. Unutar petlje čeka se da *read_s* signal postane aktivan, što predstavlja indikaciju da je tekuća operacija množenja završena i da se rezultat nalazi na izlaznom portu *r_o*. *Checker* modul generiše dve vrste izveštaja, u zavisnosti od vrednosti konstante *REPORT_DETAIL*, na isti način kao u testbenču baziranom na „zlatnim vektorima“.

Nakon što se obrade svi test vektori, prekida se simulacija i korisnik obaveštava o tome.

Modul *scoreboard*

Najveća novina u okviru verifikacionog okruženja, u odnosu na verifikaciono okruženje prikazano u testbenču baziranom na „zlatnim vektorima“, jeste modul *scoreboard*. Ovaj modul sadrži „transaction-level“ referentni model sekvencijalnog „Add-and-Shift“ množača, koji se koristi za izračunavanje očekivane vrednosti operacije množenja dva zadata broja. Ova vrednost se zatim koristi u modulu *checker* tokom provere ispravnog rada DUT-a. Referentni model takođe izračunava vreme potrebno da se završi tekuća operacija množenja. Ovaj podatak biće korišćen u okviru modula *monitor*. Struktura *scoreboard* modula prikazana je u nastavku.

```
entity scoreboard is
    port (start_i: in std_logic;
          a_i:     in std_logic_vector(OP_WIDTH-1 downto 0);
          b_i:     in std_logic_vector(OP_WIDTH-1 downto 0);

          exp_output_o:    out std_logic_vector(2*OP_WIDTH-1 downto 0);
          mult_duration_o: out natural
         );
end scoreboard;

architecture beh of scoreboard is
begin
    ---------------------------------------------------------------------------
    -- Process that implements the transaction-level model of the multiplier.
    -- expected duration of the current multiplication operation.
    ---------------------------------------------------------------------------
    tl_model:
    process
    begin
        loop
            -- Wait until start_i becomes active, indicating that new multiplication
            -- operation is started
            wait until start_i = '1';

            -- Calculate the expected result of the multiplication
            exp_output_o <= a_i * b_i;
        end loop;
    end process tl_model;

    ---------------------------------------------------------------------------
    -- Process that calcualted expected duration of the current multiplication
    -- operation.
    ---------------------------------------------------------------------------
    calc_mult_duration:
    process
        variable num_of_ones_v: natural := 0;
    begin
        loop
            -- Wait until start_i becomes active, indicating that new multiplication
            -- operation is started
            wait until start_i = '1';

            -- Calculate the number of clock cycles it will take to multiply
            -- specified operands
            num_of_ones_v := 0;
            for i in OP_WIDTH - 1 downto 0 loop
                if b_i(i) = '1' then
                    num_of_ones_v := num_of_ones_v + 1;
                end if;
            end loop;
            mult_duration_o <= 2*num_of_ones_v + OP_WIDTH-num_of_ones_v;
        end loop;
    end process calc_mult_duration;
end architecture beh;
```

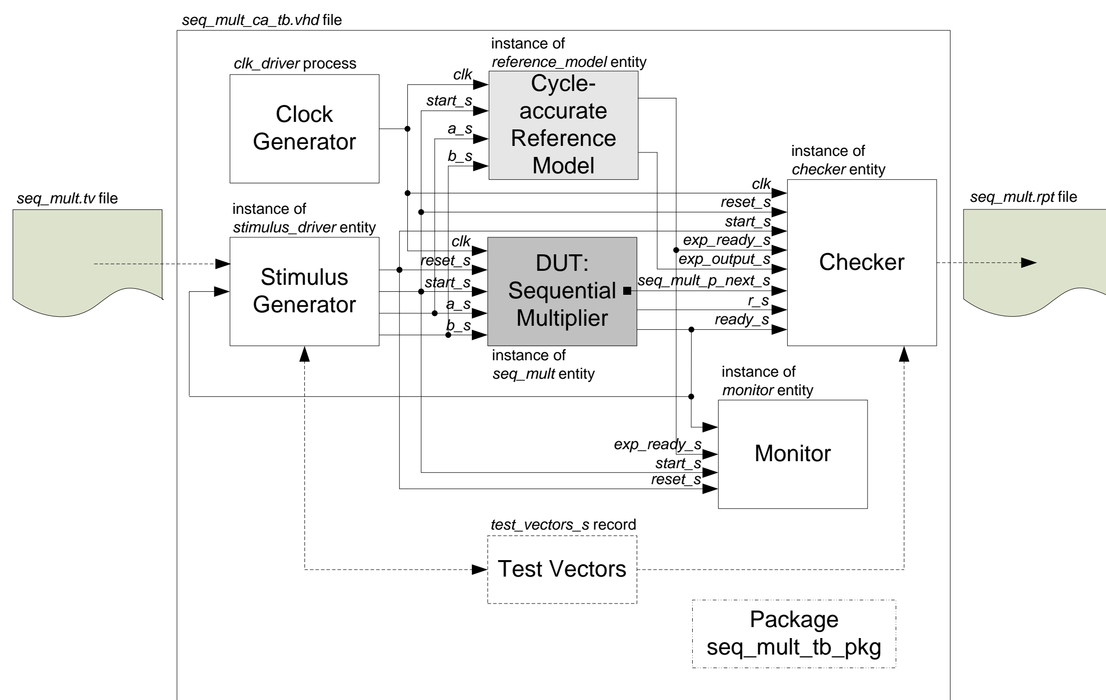
Scoreboard modul sastoji se iz dva odvojena procesa. Proces *tl_model* zapravo predstavlja „transaction-level" referentni model sekvencijalnog „Add-and-Shift" množača. Funkcija ovoga modela je izračuna očekivanu vrednost tekuće operacije množenja. Obzirom da je reč o „transaction-level" modelu, on je u stanju da izračuna samo krajnju vrednost tekuće operacije množenja. Svaka operacija množenja može se zapravo posmatrati kao jedna transakcija, pri čemu njeno trajanje može imati proizvoljnu vrednost. „Transaction-level" model ne može izračunati sve

međuvrednosti koje će se pojavljivati nakon svakog takta unutar sekvencijalnog „Add-and-Shift“ množača prilikom množenja. Ovo je razlog zbog čega ovaj proces nije osetljiv na *clk* signal.

Calc_mult_duration proces izračunava dužinu trajanja tekuće operacije množenja. Ovaj podatak se koristi unutar modula *monitor* da bi se nadgledalo da li DUT poštuje izlazni protokol generisanja *ready_o* signala.

Primer 2: Testbenč baziran na „cycle-accurate“ referentnom modelu za verifikaciju sekvencijalnog „Add-and-Shift“ množača

Prethodni primer ilustrovao je način na koji je moguće izvršiti verifikaciju sekvencijalnog „Add-and-Shift“ množača korišćenjem „transaction-level“ referentnog modela. Kao što je već rečeno, „transaction-level“ model u stanju je da predvidi samo krajnji ishod operacije množenja (koja se u ovom primeru može smatrati jednom transakcijom). Ukoliko želimo precizniji model, koji će biti u stanju da predvidi ponašanje DUT-a nakon svakog takta globalnog sinhronizacionog signala moramo razviti „cycle-accurate“ referentni model. Jasno je da je razvoj ovakvog modela daleko složeniji od odgovarajućeg „transaction-level“ modela, ali je sada moguće pratiti i verifikovati ponašanje DUT-a nakon svakog takta. Na ovaj način, ukoliko se detektuje greška daleko je lakše utvrditi njeno poreklo. Ovo je i glavni razlog zbog kojega se koriste „cycle-accurate“ referentni modeli. Blok dijagram verifikacionog okruženja baziranog na „cycle-accurate“ referentnom modelu prikazan je na Slici 4. VHDL fajlovi koji čine ovo verifikaciono okruženje mogu se pronaći unutar arhive „Vezba 9 – Kreiranje verifikacionih okruzenja.rar“, unutar *cycle_accurate_model* direktorijuma.



Slika 4. Blok dijagram verifikacionog okruženja za modifikovani „Add-and-Shift“ množač, baziranog na korišćenju „cycle-accurate“ referentnog modela

Po svojoj strukturi, verifikaciono okruženje bazirano na „transaction-level“ modelu suštinski se ne razlikuje se od verifikacionog okruženja baziranog na „cycle-accurate“

modelu. Jedine razlike postojaće unutar *checker* i *monitor* komponenti. Naravno, umesto „transaction-level“ referentnog modela koji se nalazio unutar *scoreboard* komponente, sada imamo „cycle-accurate“ referentni model, koji se nalazi unutar *reference_model* komponente.

Pogledajmo detaljnije strukturu samo onih modula koji se razlikuju u odnosu na verifikaciono okruženje bazirano na „transaction-level“ referentnom modelu.

Modul *monitor*

Kao i ranije, funkcija ovog modula je da nadgleda izlaze množača i kontinualno proverava da li je ispoštovan izlazni protokol. Struktura modula prikazana je u nastavku.

```
entity monitor is
    port (clk:     in std_logic;
          reset_i: in std_logic;
          start_i: in std_logic;
          ready_i: in std_logic;
          exp_ready_i: in std_logic
         );
end monitor;

architecture beh of monitor is
begin
    ---------------------------------------------------------------------------
    -- Process that monitors the output values from the multiplier and checks
    -- for output protocol compliance.
    ---------------------------------------------------------------------------
    monitor:
    process
        variable tv_count_v: natural := 0;
    begin
        -- Wait until reset_i becomes active
        wait until reset_i = '1';
        -- Wait until reset_i is deactivated
        wait until reset_i = '0';

        loop
            wait until falling_edge(clk);
            if (ready_i = '1' and exp_ready_i = '0') then
                report "Output protocol violation detected! Ready_o signal should be zero!"
                    severity warning;
            end if;
            if (ready_i = '0' and exp_ready_i = '1') then
                report "Output protocol violation detected! Ready_o signal should be one!"
                    severity warning;
            end if;
        end loop;
    end process monitor;
end architecture beh;
```

Ono što se može uočiti na prvi pogled, je da je struktura *monitor* modula ovaj put znatno jednostavnija. Razlog zbog čega je to tako je što nam „cycle-accurate“ referentni model daje tačno predviđanje ponašanja *ready_o* signala DUT-a. Monitor stoga treba samo da kontinualno upoređuje ovu predkciju (ulazni port *exp_ready_i*) sa *ready_o* izlazom DUT-a (ulazni port *ready_i*) i signalizira ukoliko dođe do nekog neslaganja.

Modul *checker*

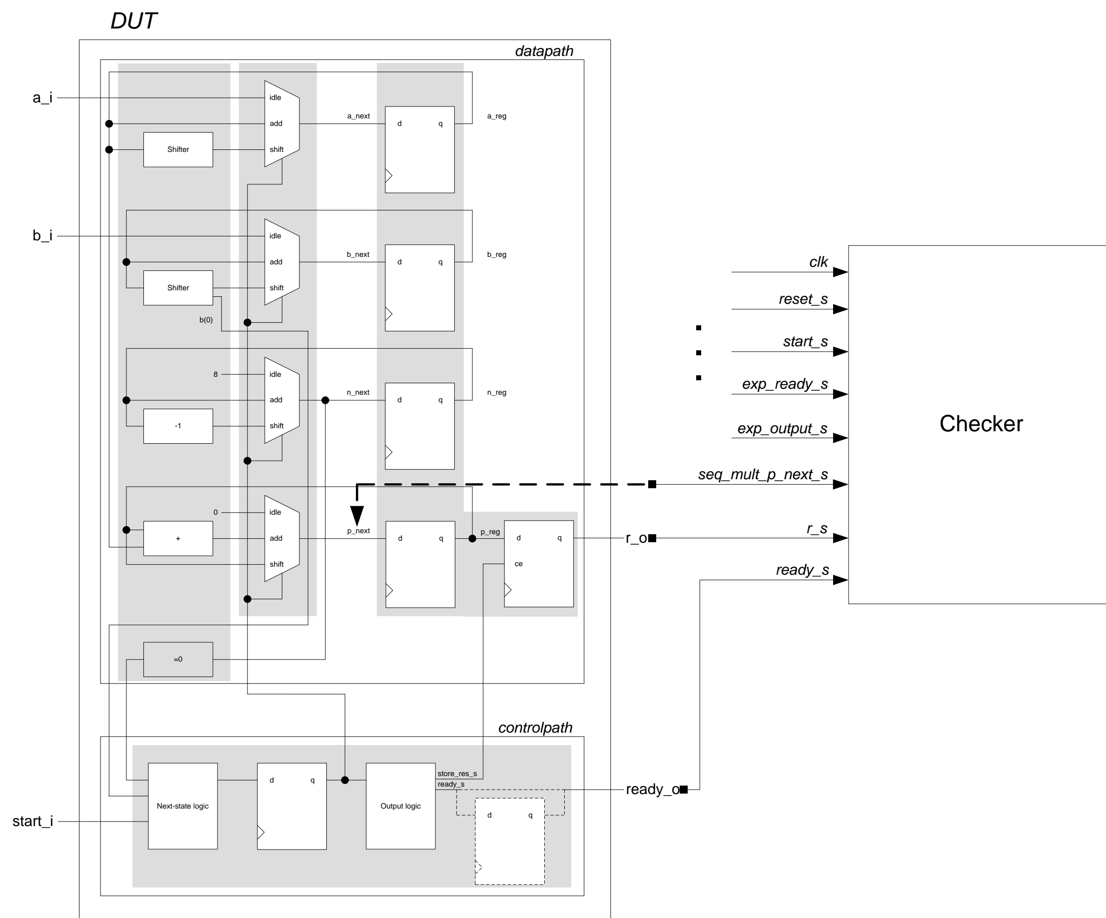
Funkcija ovog modula je da izvrši proveru da li je rezultat množenja dva ulazna broja, koji se nalazi na izlazu *r_o* DUT-a korektan. Da bi se izvršila ova provera, *checker* modul koristi estimaciju rezultata množenja, dobijenu pomoću referentnog modela, *exp_output_i*.

```
entity checker is
    generic (
            rpt_filename_g: string
           );
    port (clk:           in std_logic;
          reset_i:       in std_logic;
          start_i:       in std_logic;
          ready_i:       in std_logic;
          exp_ready_i: in std_logic;
          r_i:           in std_logic_vector(2*OP_WIDTH-1 downto 0);
          p_next_i:      in std_logic_vector(2*OP_WIDTH-1 downto 0);
          exp_output_i: in std_logic_vector(2*OP_WIDTH-1 downto 0)
         );
end checker;

architecture beh of checker is
begin
    ------------------------------------------------------------------------------
    -- Process that compares output values from the multiplier with golden vectors.
    ------------------------------------------------------------------------------
    checker:
    process
        variable cycle_num_v: natural := 0;
        variable tv_count_v: natural := 0;
        variable errors_present_v: boolean := false;
        variable l: line;

        file report_file_f: text;
    begin
        -- Open the report file
        file_open (report_file_f, rpt_filename_g, write_mode);

        write (l, string'("Starting simulation!"));
        writeline (report_file_f, l);

        loop
            wait until falling_edge(clk);
            -- If new test vector is applied, write the info about the test vector in the report file
            if (start_i = '1') then
                cycle_num_v := 1;
                writeline (report_file_f, l); -- write one blank line
                write (l, string'("-----------------------------------------------------------------------------------------"));
                writeline (report_file_f, l);
                write (l, string'("Applying test vector number: "));
                write (l, tv_count_v + 1);
                writeline (report_file_f, l);
                write (l, string'("A = "));
                write (l, conv_integer(test_vectors_s(tv_count_v).a), justified => left, field => 3);
                write (l, string'("    B = "));
                write (l, conv_integer(test_vectors_s(tv_count_v).b), justified => left, field => 3);
                writeline (report_file_f, l);
                write (l, string'("-----------------------------------------------------------------------------------------"));
                writeline (report_file_f, l);
                tv_count_v := tv_count_v + 1;
            end if;

            if (tv_count_v > 0) then
                if (REPORT_DETAIL = VERBOSE) then
                    if (exp_ready_i = '0') then
                        write (l, string'("Cycle "));
                        write (l, cycle_num_v, justified => left, field => 4);
                        write (l, string'(": "));
                        write (l, string'("Actual temporary result = "));
                        write (l, conv_integer(p_next_i), justified => left, field => 6);
                        write (l, string'("Expected temporary result = "));
                        write (l, conv_integer(exp_output_i), justified => left, field => 6);
                        if (p_next_i = exp_output_i) then
                            write (l, string'(" Status: OK "));
                        else
                            write (l, string'(" Status: ERROR "));
```

```
                        end if;
                        writeline (report_file_f, l);
                    end if;
                    if (exp_ready_i = '1') then
                        write (l, string'("Multiplication finished! Checking final output: "));
                        if (r_i = exp_output_i) then
                            write (l, string'("OK"));
                        else
                            write (l, string'("ERROR"));
                        end if;
                        writeline (report_file_f, l);
                        -- If all test vectors have been applied, exit the loop and end the simulation
                        if (tv_count_v = tv_count_s) then
                            exit;
                        end if;
                    end if;
                end if;
                if (REPORT_DETAIL = ERR_ONLY) then
                    if (exp_ready_i = '0') then
                        if (p_next_i /= exp_output_i) then
                            errors_present_v := true;
                            write (l, string'("Cycle "));
                            write (l, cycle_num_v, justified => left, field => 4);
                            write (l, string'(": "));
                            write (l, string'("Actual temporary result = "));
                            write (l, conv_integer(p_next_i), justified => left, field => 6);
                            write (l, string'("Expected temporary result = "));
                            write (l, conv_integer(exp_output_i), justified => left, field => 6);
                            write (l, string'(" Status: ERROR "));
                            writeline (report_file_f, l);
                        end if;
                    end if;
                    if (exp_ready_i = '1') then
                        if (r_i /= exp_output_i) then
                            errors_present_v := true;
                            write (l, string'("Multiplication finished! Checking final output: ERROR"));
                            writeline (report_file_f, l);
                        end if;
                        -- If all test vectors have been applied, exit the loop and end the simulation
                        if (tv_count_v = tv_count_s) then
                            exit;
                        end if;
                    end if;
                end if;
            end if;
            cycle_num_v := cycle_num_v + 1;
        end loop;


        if (REPORT_DETAIL = ERR_ONLY and errors_present_v = false) then
            writeline (report_file_f, l);
            write (l, string'("Simulation finished! No errors found during simulation!"));
            writeline (report_file_f, l);
        end if;

        -- All test vectors have been applied and results have been checked,
        -- wait one more clock cycle then end the simulation
        wait until rising_edge(clk);

        -- Close the report file
        file_close (report_file_f);

        -- End the simulation
        report "ALL TEST VECTORS HAVE BEEN APPLIED TO THE DUT! ENDING SIMULATION!"
            severity failure;
    end process checker;
end architecture beh;
```

Ovaj put je struktura *checker* modula nešto drugačija. Obzirom da *checker* ovaj put mora da proverava rad DUT-a nakon svake periode globalnog sinhronizacionog signala. Obzirom da se izlazni port *r_o* DUT-a ažurira samo nakon što se izračuna konačna vrednost proizvoda dva zadata broja nije dovoljno posmatrati samo njega. Ovaj put *checker* mora da „zaviri“ i u unutrašnjost DUT-a, odnosno verifikaciono okruženje bazirano na „cycle-accurate“ referentnom modelu posmatra DUT-a kao „sivu kutiju“ (odnosno koristi *grey box* pristup). Analizom načina na koji je

projektovan sekvencijalni „Add-and-Shift“ množač može se zaključiti da je ovom prilikom neophodno posmatrati i unutrašnji signal *p_next*, koji predstavalja narednu vrednost koja će biti smeštena u *p* registar. Ova situacija prikazana je na Slici 5.



Slika 5. Način povezivanja *cheker* modula sa DUT-om

Kao što je ranije rečeno, VHDL ne pruža mogućnost da se sa višeg nivoa hijerarhije pristupi objektu koji je definisan unutar nekog, nižeg nivoa hijerarhije. To znači da ne postoji druga mogućnost da *checker* modul pristupi *p_next* signalu, osim da se modifikuje port lista DUT-a i signal *p_next* učini izlaznim portom. Ovo očigledno nije dobro rešenje. Srećom, neki VHDL simulatori obezbeđuju drugačije mehanizme pomoću kojih je ipak moguće pristupiti ovim unutrašnjim signalima, bez potrebe za modifikovanjem DUT-a. Na primer, u okviru ModelSim simulatora ovo se može postići korišćenjem *Signal Spy* procedura. Ove procedure omogućavaju nam da nadgledamo (*spy*), čak i da modifikujemo VHDL objekte koji su definisani na nekom od nižih nivoa hijerarhije. Način korišćenja *Signal Spy* procedura je krajnje jednostavan.

Prvi korak predstavlja uključivanje ModelSim biblioteke *modelsim_lib* u okviru koje se nalazi paket *util* koji treba koristiti. Ovo se postiže pomoću sledeće dve linije u okviru VHDL modula koji želi da koristi *Signal Spy* proceduru.

```
library modelsim_lib;
use modelsim_lib.util.all;
```

Prvi korak predstavlja definisanje odgovarajućeg signala unutar verifikacionog okruženja koji će zapravo predstavljati vezu sa unutrašnjim signalom kome želimo da pristupamo. Jasno je da tip ovoga signal mora biti isti sa tipom unutrašnjeg signala kome pristupamo. U okviru našeg verifikacionog okruženja, u VHDL datoteci *seq_mult_ca_tb.vhd*, definisan je sledeći signal koji će omogućiti pristup signalu *p_next*, koji je definisan unutar DUT-a.

```
-- Internal signal from the DUT that we need to monitor
signal seq_mult_p_next_s: std_logic_vector(2*OP_WIDTH-1 downto 0);
```

Signal *seq_mult_p_next_s* predstavljaće „vezu“ sa unutrašnjim signalom *p_next*. Još nam preostaje da izvršimo samo mapiranje. Ovo se postiže pomoću sledećeg procesa definisanog unutar *seq_mult_ca_tb* modula.

```
---------------------------------------------------------------------------
-- Spy process required in order to be able to monitor DUT internal signal, p_next.
---------------------------------------------------------------------------
spy_process:
process
begin
    init_signal_spy("/seq_mult_ca_tb/dut/p_next","/seq_mult_ca_tb/seq_mult_p_next_s",1, 1);
    wait;
end process spy_process;
```

U okviru procesa *spy_proces* vrši se povezivanje signala *seq_mult_p_next_s* sa unutrašnjim signalom *p_next* pozivom procedure *init_spy_signal*. Ova procedura zahteva ukupno četiri parametra.

Prvi parametar predstavlja hijerarhijsku putanju do željenog unutrašnjeg signala. U našem slučaju *p_next* signal nalazi se unutar *dut* objekta, koji se pak nalazi unutar *seq_mult_ca_tb* objekta, koji zapravo predstavlja čitavo verifikaciono okruženje, odnoso vrh hijerarhije.

Drugi parametar predstavlja hijerarhijsko ime signala koji će biti veza sa unutrašnjim signalom kome želimo da pristupimo. U našem slučaju to je signal *seq_mult_p_next_s* koji je definisan u okviru modula *seq_mult_ca_tb*.

Značenje preostala dva parametra trenutno nije od interesa i oni se mogu postaviti na vrednost 1.

Nakon poziva *init_signal_spy* procedure formira se trajna veza između polaznog i odredišnog signala. Svaka promena polaznog signala verno se reprodukuje na odredišnom signalu. Za detalje u vezi sa *init_spy_signal* procedurom, zainteresovani čitalac se upućuje na dokumentaciju ModelSim simulatora (*ModelSim User's Manual*).

Obzirom da simulatori kompanije Xilinx još uvek ne pružaju mogućnost nadgledanja unutaršnjih signala DUV-a iz testbenča, jedina mogućnost koja nam preostaje jeste da modifikujemo entity deklaraciju „Add-and-Shift“ množača i dodamo još jedan izlazni port preko kojega ćemo moći pristupiti unutrašnjem signalu *p_next*. Da bi se jasno naznačilo da je jedina namena ovog dodatnog porta omogućavanje efikasnije verifikacije, tipično je da se njegovo ime izabere na način koji će oslikavati ovu činjenicu. Ime koje se najčešće koristi za ovaj tip portova je *debug*. U našem konkretnom primeru, u okviru entity deklracije „Add-and-Shift“ množača dodaćemo još jedan dodatni 16-bitni izlazni port *debug_o*, kao što je prikazano u nastavku.

```
entity seq_mult is
    generic (
        gen_correct_model_g: std_logic_vector(1 downto 0)
       );
    port (clk:      in std_logic;
        reset_i:    in std_logic;
        start_i:    in std_logic;
        a_i:        in std_logic_vector(7 downto 0);
        b_i:        in std_logic_vector(7 downto 0);

        -- Additional port used for debugging
        debug_o: out std_logic_vector(15 downto 0);

        ready_o:  out std_logic;
        r_o:        out std_logic_vector(15 downto 0)
          );
end seq_mult;
```

Dodatni port potreban za verifikaciju

Pored ove modifikacije, potrebno je izvršiti još jednu. U okviru arhitekturnog tela „Add-and-Shift" množača potrebno je dodati još jednu naredbu konkurentne dodele signalu, pomoću kojega će na *debug_o* izlazni port biti povezan odgovarajući unutrašnji signal. U našem slučaju na *debug_o* izlazni port potrebno je povezati *p_next* unutrašnji signal, kao što je prikazano u nastavku.

```
architecture shift_add_arch of seq_mult is
    constant WIDTH: integer := 8;   -- width of the input operands
    constant C_WIDTH: integer := 4; -- width of the counter
    constant C_INIT: unsigned(C_WIDTH-1 downto 0) := "1000";

    type state_type is (idle, add, shift);
    signal state_reg, state_next: state_type;

    signal b_reg, b_next: unsigned(WIDTH-1 downto 0);
    signal a_reg, a_next: unsigned(2*WIDTH-1 downto 0);
    signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
    signal p_reg, p_next: unsigned(2*WIDTH-1 downto 0);

    signal ready_s: std_logic;
    signal store_res_s: std_logic;
begin
    -- Connect the debug port to selected internal signal
    debug_o <= p_next;

    -- State and data registers
    process (clk, reset_i)
    ...
```

Povezivanje *debug_o* porta na odgovarajući unutrašnji signal

Ovaj put *checker* modul, nakon završetka svake periode globalnog sinhronizacionog signala vrši upoređivanje očekivane vrednosti (port *exp_output_i*) sa vrednošću unutrašnjeg signala *p_next* (preko porta *p_next_i* na koji je povezan *seq_mult_p_next_s* signal, koji je sa druge strane povezan na *debug_o* izlazni port „Add-and-Shift" množača). U slučaju da je tekuća operacija množenja završena (*exp_ready_i* port ima vrednost '1'), proverava se da li i *r_o* izlaz DUT-a, koji je

povezan na *r_i* ulaz *checker* modula, ima očekivanu vrednost. Kao i ranije postoje dva nivoa detalja smeštanja informacija u *log* datoteku koji se kontrolišu pomoću konstante *REPORT_DETAIL*.

Modul *reference_model*

Ovaj modul predstavlja „cycle-accurate“ referentni model sekvencijalnog „Add-and-Shift“ množača. Struktura referentnog modela prikazana je u nastavku.

```
entity reference_model is
    port (clk:      in std_logic;
          start_i: in std_logic;
          a_i:      in std_logic_vector(OP_WIDTH-1 downto 0);
          b_i:      in std_logic_vector(OP_WIDTH-1 downto 0);

          exp_ready_o:  out std_logic;
          exp_output_o: out std_logic_vector(2*OP_WIDTH-1 downto 0)
          );
end reference_model;

architecture beh of reference_model is
begin
    ----------------------------------------------------------------------------
    -- Process that implements the cycle accurate reference model of the multiplier.
    ----------------------------------------------------------------------------
    ref_model:
    process
        variable a_v: natural := 0;
        variable exp_output_v: natural := 0;
    begin
        exp_ready_o <= '1';
        exp_output_o <= (others => '0');
        loop
            -- Wait for the start_i to go high, indicating the start of new
            -- multiplication cycle
            wait until start_i = '1';
            exp_output_o <= (others => '0');
            wait until rising_edge(clk) and start_i = '1';

            -- Calculate the expected output after each clock cycle
            exp_output_v := 0;
            a_v := conv_integer (a_i);
            for i in 0 to OP_WIDTH-1 loop
                exp_output_v := exp_output_v + conv_integer(b_i(i)) * a_v;
                a_v := 2 * a_v;

                exp_ready_o <= '0';
                exp_output_o <= conv_std_logic_vector (exp_output_v, 2*OP_WIDTH);
                if (b_i(i) = '1') then
                    wait until rising_edge(clk);
                    wait until rising_edge(clk);
                else
                    wait until rising_edge(clk);
                end if;
            end loop;
            exp_ready_o <= '1';
        end loop;
    end process ref_model;
end architecture beh;
```

Sam „cycle-accurate“ referentni model je nešto složeniji od „transaction-level“

referentnog modela, što se moglo i očekivati. Referentni model ovaj put mora verno reprodukovati način rada samoga DUT-a nakon svake periode globalnog sinhronizacionog signala. Međutim, ono što treba da naglasiti da je i pored toga njegova veličina i složenost neuporedivo manja od odgovarajućeg RTL modela koji smo bili prinuđeni da razvijemo prilikom projektovanja samog DUT-a.