Digitalni sistem se može opisati na različitim nivoima apstrakcije. Poželjno je imati zajednički radni okvir kako bi se jednostavno mogle razmjenjivati informacije među dizajnerima i softverskim alatima. Upravo je to svrha jezilka za opis hardvera (HDLs). Zadatak HDL-a je da vjerno opiše sistem na funkcionalnom ili strukturalnom nivou, na željenom nivou apstrakcije. Upotreba i semantika ovih jezika je značajno različita u odnosu na tradicionalne programske jezike.

Tradicionalni programski jezici vs HDLs

- Tradicionalni jezici:
  - Sekvencijalno izvršavanje
  - Jednostavan razvoj algoritma
  - Efikasan prevod na nivo mašinskih instrukcija
- Digitalni sistem karakterišu:
  - Veliki broj gradivnih elementa povezanih na određeni način
  - Paralelne operacije
  - Propagaciono kašnjenje

Programski jezik karakteriše sintaksa i semantika. Sintaksa podrazumijeva niz pravila koja se odnose na pisanje koda, dok je semantika značenje onoga što je napisano. Većina tradicionalnih programskih jezika kao što je C su modelovani na osnovu sekvencijalnog prosesa. Operacije se izvršavaju sekvencijalno, jedna za drugom. Redosljed operacija se ne može proizvoljno mijenjati. Ovaj model ima dvije važne prednosti. Čovjeku je jednostavno da razvije algoritam korak po korak, a i prevod iz algoritma na nivo mašinskih instrukcija je efikasan.

Sa druge strane, tipičan digitalni sistem se sastoji od velikog broja gradivnih elemenata, koji su povezani na određeni način. Kada se promijeni vrijednost ulaznog signala, svi blokovi do kojih signal dolazi se aktiviraju i inicira se set novih operacija. Ove operacije se izvršavaju istovremeno, i svaka od njih će trajati određeno vrijeme (propagaciono kašnjenje tog bloka). Ako je na izlazu određenog bloka stanje promijenjeno, to će inicirati aktivaciju svih blokova koji su sa njim povezani. Tradicionalni programski jezici nisu u stanju da modeluju ovakav sistem. Zato je bilo potrebno uvesti jezike za opis hardvera.

Tradicionalni programski jezici vs HDLs

- Tradicionalni programski jezik
  - Na osnovu zadatog ulaza generiše odgovarajući izlaz
  - Kod se prevodi na nivo mašinskih instrukcija
  - Program se pokreće na host računaru
- Uloge HDL programa:
  - Formalna dokumentacija
  - Ulaz za simulator
  - Ulaz za synthesizer

Program kodiran u tradicionalnim programskim jezicima je kreiran u cilju rješavanja specifičnog problema. Uzima odgovarajuće ulazne podatke i generiše odgovarajući izlaz na osnovu njih. Program se najprije kompajlira na nivo mašinskih instrukcija, a potom se pokreće na host računaru. Sa druge strane, primjena HDL programa je veoma različita. Program obavlja tri funkcije: formalnu dokumentaciju, ulaz za simulator, ulaz za sintetizer.
Sistem opisan u HDL-u je precizan i eksplicitan, tako da HDL progam može da posluži kao formalna specifikacija i dokumentacija sistema korisna za dizajnere i korisnike.
HDL program, zajedno sa generisanim testnim vektorima i kodom za prikupljanje podataka, formira testbench, koji je ulaz za HDL simulator. U toku izvršavanja, simulator tumači HDL kod i generiše odgovarajući odziv.
Softver za sintezu (synthesizer) na osnovu HDL programa realizuje kolo uz pomoć komponenti iz odgovarajućih biblioteka. Izlaz synthesizer-a je novi HDL program koji predstavlja strukturalni opis sintetizovanog kola.

HDL dizajn

- Koncepti:
  - Entitet
  - Povezivanje (connectivity)
  - Paralelizam (concurrency)
  - Timing
- Veriog, VHDL

Osnovne karakteristike digitalnog kola se definišu kroz koncept eniteta, povezivanja, paralelizma i timinga.

Entitet je osnovni gradivni element, modelovan na osnovu određenog dijela realnog kola.

Povezivanjem se modeluju provodne linije među odgovarajućim djelovima sistema.

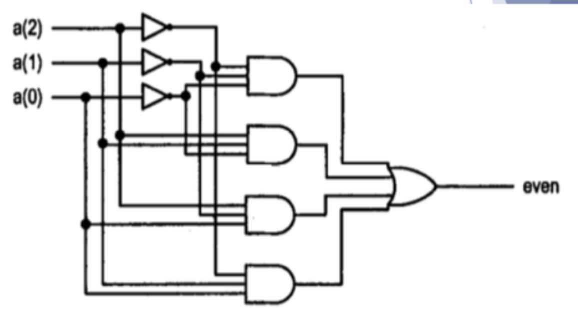
Entiteti povezani istom vezom su aktivni u istom trenutku, i mnoge operacije se obavljaju paralelno. Concurrency upravo opisuje ovo ponašanje.

Timing je povezan sa paralelizmom. Specificira početak i kraj svake operacije u vremenu, i daje neku vrstu rasporeda kada će se šta izvršiti.

Cilj HDL-a je da opiše i modeluje digitalni sistem što je moguće vjernije.

Primjer: even-parity

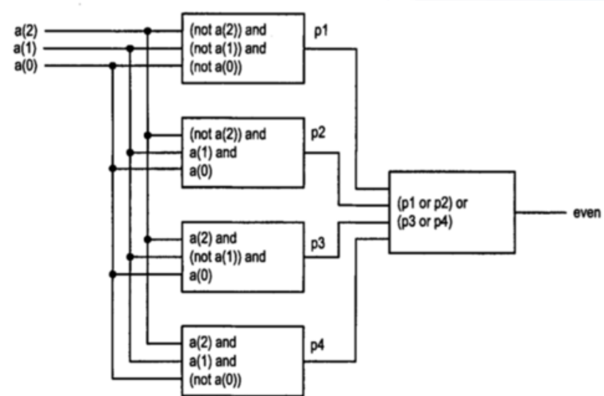
a(2)	a(1)	a(0)	even
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



Kolo detektuje paran broj jedinica na ulazu (nula ili dvije).

Primjer: even-parity

```
entity even_detector is

    Port ( a : in  std_logic_vector(2 downto 0);
           even : out  std_logic);

end even_detector;

architecture Behavioral of even_detector is

   signal p1, p2, p3, p4: std_logic;

begin

   even <= (p1 or p2) or (p3 or p4);
   p1 <= not(a(2)) and not(a(1)) and not(a(0));
   p2 <= not(a(2)) and a(1) and a(0);
   p3 <= a(2) and not(a(1)) and a(0);
   p4 <= a(2) and a(1) and not(a(0));

end Behavioral;
```



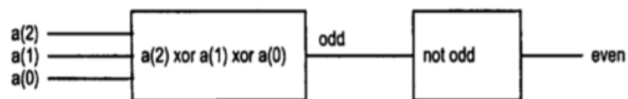
Entitet specificira ulazne i izlazne portove kola.

Ahitektura specificira interne operacije ili organizaciju kola.

Prpagaciono kašnjenje treba uvijek imati na umu.

Primjer: even-parity sa XOR kolom

```
architecture arc_even_detector of even_detector is
    signal odd: std_logic;
begin
    even <= not odd;
    odd <= a(2) xor a(1) xor a(0);
end arc_even_detector;
```



Isti problem se može riješiti i uz pomoć XOR kola.

Strukturalni opis

```
entity xor2 is
     port
     (
          i1, i2: in std_logic;
          o1: out std_logic
     );
end xor2;
architecture arc_xor2 of xor2 is
begin
     o1 <= i1 xor i2;
end arc_xor2;
```

```
entity not1 is
     port
     (
          i1: in std_logic;
          o1: out std_logic
     );
end not1;
architecture arc_not1 of not1 is
begin
     o1 <= not i1;
end arc_not1;
```

```
entity even_detector is
port
     (
          a : in  std_logic_vector(2 downto 0);
          even : out  std_logic
     );
end even_detector;
architecture arc_even_detector_s of even_detector
is
component xor2
     port
     (
          i1, i2: in std_logic;
          o1: out std_logic
     );
end component;
component not1
     port
     (
          i1: in std_logic;
          o1: out std_logic
     );
end component;
     signal sig1, sig2: std_logic;
begin
     unit1: xor2
     port map (i1 => a(0), i2 => a(1), o1 => sig1);
     unit2: xor2
     port map (i1 => sig1, i2 => a(2), o1 => sig2);
     unit3: not1
     port map (i1 => sig2, o1 => even);
end arc_even_detector_s;
```

Kod strukturalnog opisa, kolo se konstruiše od manjih gradivnih elemenata. Opisom se specificira koji elementi se koriste i na koji način su povezani. Iako smo u prethodnom primjeru svaki od izraza koji se paralelno izvršavaju tretirali kao poseban element, formlana VHDL strukturalna dskripcija se vrši kroz koncept komponenti (components). Komponenta može da bude postojeći, ili hipotetički element. Najprije je potrebno deklarisati komponentu, a potom instancirati (koristiti) u tijelu arhitekture u skladu sa potrebama.
Strukturalni opis i upotreba komponenti su veoma korisni. Olakšavaju hijerarhijski dizajn. Kompleksan sistem se može podijeliti u nekoliko manjih subsistema od kojih je svaki predstavljen komponentom i zasebno dizajniran. Dalje, podržavaju predizajniranje kola. Na ovaj način, složeni IP core-ovi kao i ćelije iz biblioteka se mogu koristtiti kao “crne kutije”.

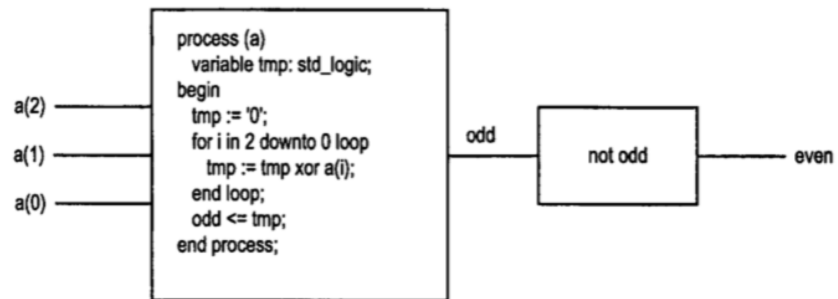
Apstraktni funkcionalni opis

```
process (sensitivity_list)
   variable declaration;
begin
   sequential statements;
end process;
```

Kada je u pitanju složen i obiman sistem, implementacija može da bude veoma vremenski zahtjevna. Način na koji čovjek razmišlja i kreira algoritam je u osnovi sekvencijalne prirode. Iz tog razloga, VHDL ima mogućnost kodiranja koje preslikava sekvencijalnu semantiku, uključujući upotrebu varijabli i sekvencijalno izvršavanje. Ovo svojstvo se smatra izuzetkom u odnosu na regularnu VHDL semantiku i enkapsulirano je u tzv **process**. Ovaj način kodiranja se često označava kao funkcionalni opis. U principu, za funkcionalni opis nema precizne definicije. Prema VHDL-u, svi kodovi, osim za čisto instanciranje komponenti se smatraju funkcionalnim.

Sensitivity lista se sastoji od seta signala. Kada se neki od signala u listi izmijeni, proces se aktivira. U okviru procesa, semantika je slična tradicionalnim programskim jezicima. Koriste se varijable i izvršavanje koda je sekvencijalno. Varijable i petlje nemaju odgovarajući fizički par, kao što je slučaj sa signalima i paralelnim izvršavanjem.

Primjer sa process: even-parity

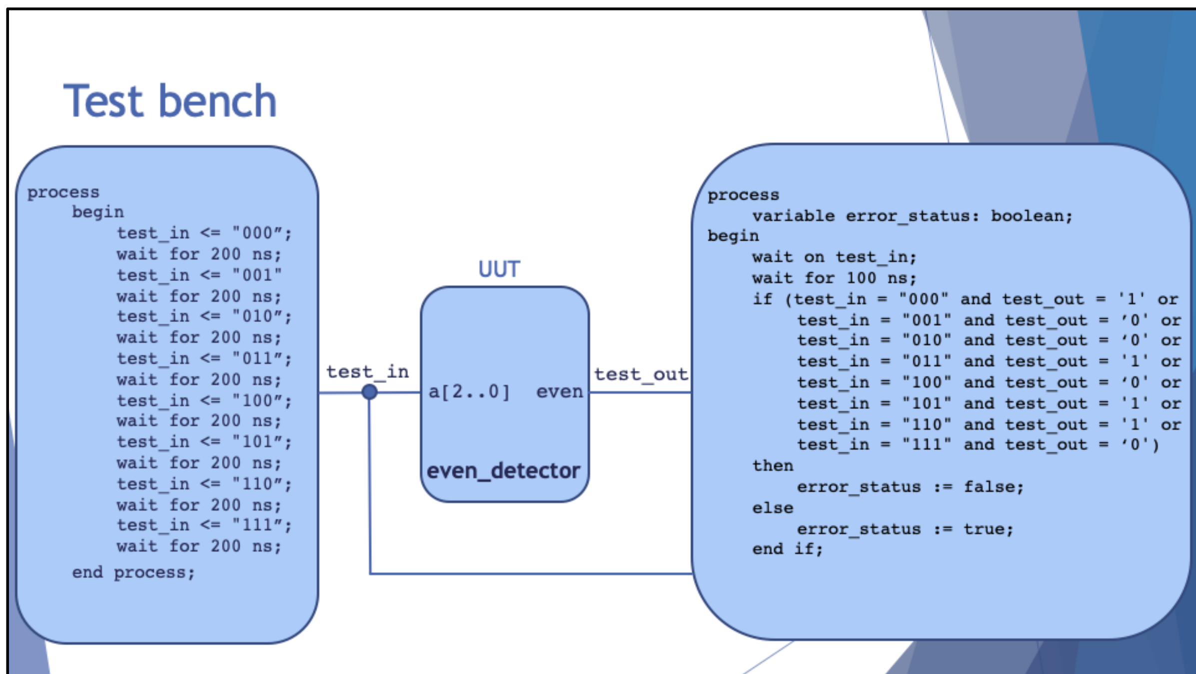


Primjer sa process: even-parity

a(2)
a(1)
a(0)

```
process(a)
    variable sum, r: integer;
begin
    sum := 0;
for i in 2 downto 0 loop
    if a(i) = '1' then
        sum := sum + 1;
    end if;
end loop;

r := sum mod 2;

if (r = 0) then
    even <= '1';
else
    even <= '0';
end if;
end process;
```

even



Jedna od glavnih upotreba VHDL-a je simulacija, koja se koristi u cilju analize operacije koju obavlja kolo, ili u cilju provjere korektnosti dizajna. Simuliranje je slično eksperimentu sa fizičkim kolom, gdje se na ulaz kola dovode odgovarajući signali (analogno signal generatoru) i posmatra se odziv (analogno logic analyzer-u). Simulacija VHDL opisa je kao virtueleni eksperiment, gdje je fizičko kolo predstavljeno VHDL opisom. Moguće je razviti rutine koje oponašaju funkciju signal generatora i koje prihvataju i upoređuju odgovarajuće odzive – **testbench**.

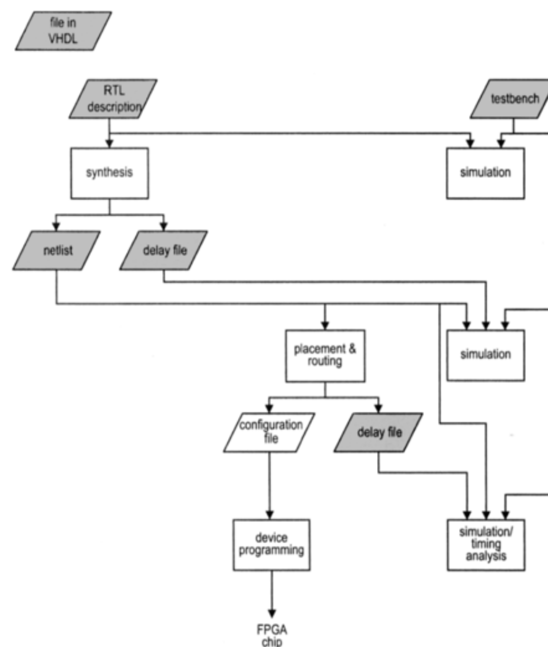
Prikazani testbench sadrži generator test vektora koji generiše pobudu sistema, i verifier koji provjerava korektnost odziva. Testbench se sastoji od entiteta i arhitekture. Naravno, nema portove.
Instanciranjem odgovarajućih komponenti u okviru testbencha ukazuje se na njihovu upotrebu u procesu testiranja. Pinovi komponenti se povezuju za generator testnih signala i verifier.

Konfiguracija

```
configuration demo_config of even_detector_testbench is
    for tb_arch
        for uut: even_detector
            use entity work.even_detector(arc_even_detector);
        end for;
    end for;
end demo_config;
```

VHDL u principu razdvaja entitet i arhitekturu u dvije nezavisne jedinice. Više arhitektura se može vezati za jedan entitet. U trenutku simulacije, može se izvršiti odabir arhitekture koja se želi simulirati, odnosno koju je potrebno povezati sa entitetom.
VHDL podržava mehanizam **configuration** pomoću koga se ostavruje povezivanje entiteta i odgovarajuće arhitekture.
Ovaj postupak nije uvijek potreban. Ukoliko kofiguracijom nije drugačije zadato, entitetu se u okviru testbencha pridružuje posljednja kompajlirana arhitetktura posmatranog entiteta.

Razvojni tok. Kod za sintezu



Dizajn kompleksnog sistema počinje opisom na visokom nivou apstrakcije, koji sadrži opis odgovarajućeg ponašanja sistema, kao i testbench koji generiše test vektore za testiranje funkcionalnosti sistema. Međutim, treba voditi računa da je opis (kod) orjentisan ka sintezi. Kod bi trebalo da bude RT-level opis sistema i da predstavlja neku vrstu skice za hardver, a sve u cilju što efikasnije implementacije. Arhitekture pogodne za sintezu bile bi one date na slajdovima 7 i 10, dok je primjer loše arhitekture sa aspekta sinteze dat na slajdu 11.

Dobro zadat opis sistema sa aspekta sinteze najprije treba verifikovati, potom izvršiti sintezu. Rezultat je gate-level netlist, predstavljen kroz VHDL strukturalni opis. Uz pomoć testbench-a provjerava se korektnost sinteze, a potom se analizira timing sistema. Netlist opis se dalje prosljeđuje na placement and routing. Generisani konfiguracioni fajlovi nisu VHDL kod. Ipak, dodatne informacije o timing-u sistema će biti dostupne. Slijedi još jedna verifikacija kroz testbench.

VHDL se koristi da modeluje sve aspekte digitalnog hardvera i da olakša cjelokupan proces dizajniranja. Nakon razvoja, kod se može izvršiti kroz simulator ili synthesizer. Ova dva procesa su veoma različiti.

Kada je u pitanju simulacija, dizajn je relizovan u virtuelnom okruženju: softver simulator. Host računar koristi set instrukcija kako bi oponašao operaciju kola. Kako računar sadrži processor, simulacija kola se obavlja sekvencijalno, po principu

vremenskog multipleksiranja, dijeleći zajdeničke resurse. Sa druge strane, kod sinteze svi operatori dati kroz VHDL kod su hardverski mapirani. Posmatrajmo primjer 10 operacija sabiranja. Kod simulacije, broj operacija sabiranja, zadat u VHDL-u, ne igra značajnu ulogu, jer se samo jedno sabiranje može izvršavati u datom trenutku.U slučaju sinteze, svaki operator sabiranja je hardverski mapiran u sabirač, koji je prilično složen, i zato je poželjno da se hardver "dijeli" i da se smanji broj operatora sabiranja u VHDL opisu. Slično, sofisticirane kontrolne strukture, kao što su petlje ili naredbe uslovnog grananja, mogu se jednostavno simulirati na host računaru,dok se teže mogu efikasno hardverski implementirati.
Za sintezu se može koristiti samo podskup VHDL instrukcija. Iako je set instrukcija pogodnih za sintezu sužen, ipak je dovoljno fleksibilan. Kolo se može opisati na veliki broj načina. Čak i ako je svaki od njih sintetizabilan, ne znači da je u pitanju efikasna implementacija. Softver za sintezu može izvršiti loklanu optimizaciju, ali finalno kolo najviše zavisi od inicijalnog opisa. Neadekvatna deskripcija troši neopravdano resurse i ima loše vremenske performanse.