

Sistem Verilog

Osnove jezika

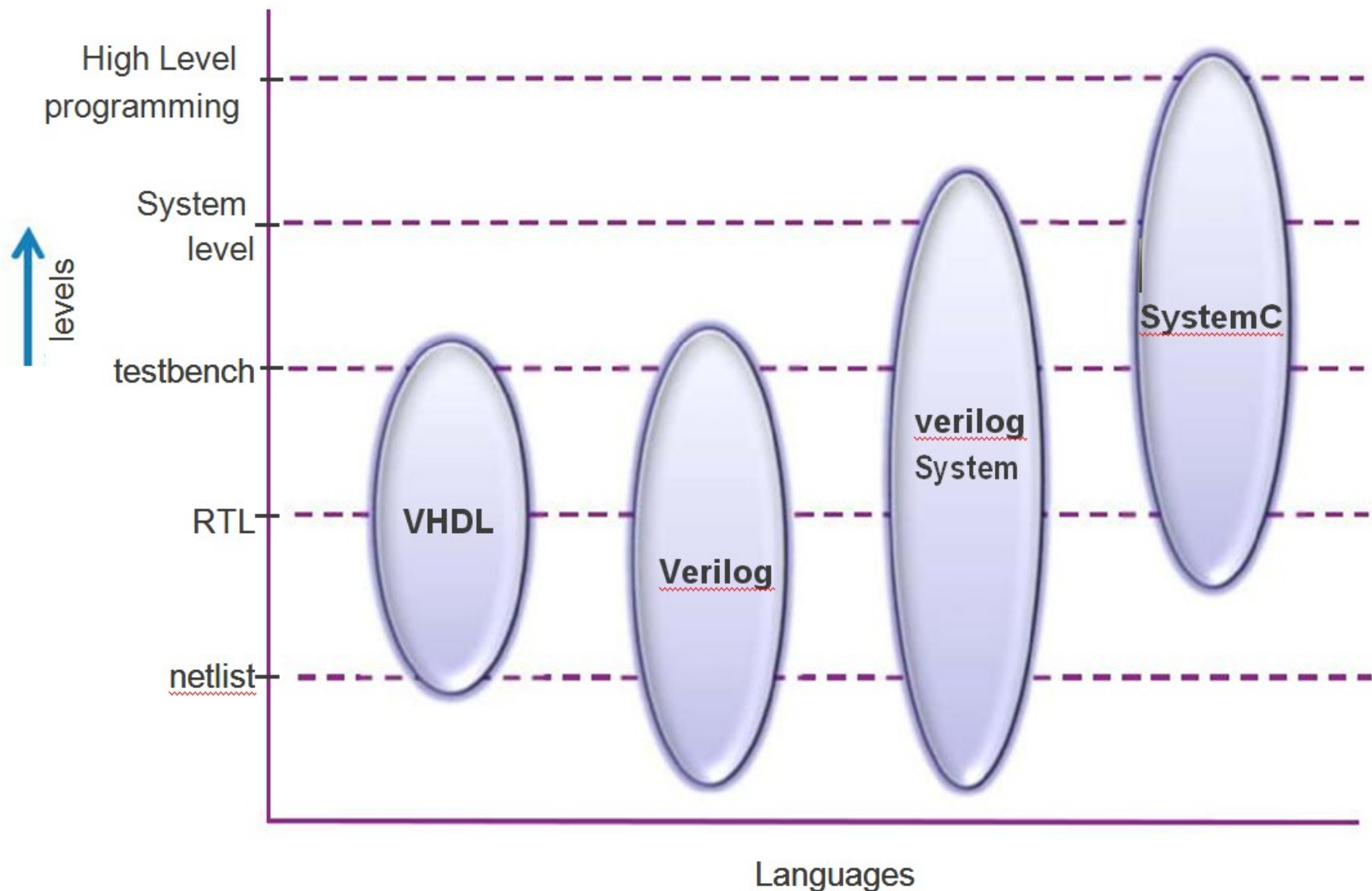
Problemi

- Uključuje složene Open verifikacione biblioteke
 - UVM, VMM, AVM...
- Puno toga mora da se uči
 - Veoma kompleksan sa mnogo raznolikih mogućnosti
- U mnogome oslonjen na složen SW svet
 - HW inženjeri ga ne vole puno

System Verilog je zapravo skup 5 jezika

- Sintetizibilni SV
- Assertions (specifične simulacione tehnike provere)
- Constraints (formiranje okvira unutar dizajna i/ili verifikacije)
- Coverage (pokrivenost)
- OOP (objektno orijentisano programiranje)

Nivoi apstrakcije HDL/HVL



HDL jezici zastupljenost u sintezi

RTL



Verification

Survey by Doulos (EU, USA, Asia)

SV kao programski jezik

- Verilog je proceduralni jezik i stoga nema dobru vremensku kontrolu
- Verilog mehanizmi kontrole assertion-a nisu jednostavni
- Proceduralna priroda Veriloga otežava testiranje paralelnih događaja u istom vremenskom priodu
- Verilog nema ugrađene mehanizme provere funkcionalne pokrivenosti
- SV je kreiran da modeluje event-based izvršavanje (modeluje izvršavanje događaja u vremenu)
- U svakom vremenskom periodu veliki broj događaja (events) se precizno raspoređuje (schedule)

Tipovi podataka

SV dodaje nove tipove podataka u odnosu na Verilog

Verilog podržava samo 4 osnovne logičke vrednosti (0,1,X,Z)

SV dodaje tip podatka sa 2-stanja

- bit 1bit, 2 stanja celobrojan (0,1), skalabilno na vektor
- byte 8bit, 2 stanja celobrojan (slično kao char u C-u)
- shortint 16 bit, 2 stanja celobrojan (slično short-u u C-u)
- int 32 bit, 2 stanja celobrojan(slično int-u u C-u)
- longint 64 bit, 2 stanja celobrojan (slično kao longlong u C-u)
- integer 32 bit, 4 stanja celobrojan

Specijalni tipovi za efikasno system-level modelovanje i verifikaciju

- void definiše funkcije koje ne vraćaju vrednost
- shortreal 32 bit floating point (sličan kao float u C-u)
- logic 1 bit, 4 stanja celobrojan, ne analizira snagu signala
- time 4 stanja, neoznačen, 64-bit celobrojan

Označeni i neoznačeni

- Celobrojni tipovi mogu biti označeni i neoznačeni
- Tipovi byte, shortint, int, integer i longint podrazumevano su označenog tipa.
- Tipovi bit, reg i logic podrazumevano su neoznačeni (isto važi za njihove nizove)
- Po potrebi korisnik eksplicitno deklariše vrstu označenost

int unsigned addr;

integer signed data;

shortint unsigned value;

Type Casting : unsigned to signed

if (signed'(datam) < 20) // datam is unsigned

Wire / Reg tip signala

Wire	Reg
<p>Wire elements are simple wires (or bus) in Verilog</p> <ol style="list-style-type: none">wire elements are used to connect input and output ports of a module instantiation together with some other element in your design.wire elements are used as inputs and outputs within an actual module declaration.wire elements must be driven by something, and cannot store a value without being driven.wire elements cannot be used as the left-hand side of an = or <= sign in an always@ block.wire elements are the only legal type on the left-hand side of an assign statement.wire elements are a stateless way of connecting two pieces in Verilog.wire elements can only be used to model combinational logic.	<p>reg are similar to wires, but can be used to store information like registers</p> <ol style="list-style-type: none">reg elements can be connected to the input port of a module instantiation.reg elements cannot be connected to the output port of a module instantiation.reg elements can be used as outputs within an actual module declaration.reg elements cannot be used as inputs within an actual module declaration.reg is the only legal type on the left-hand side of an always@ block = or <= sign.reg is the only legal type on the left-hand side of an initial block = sign (used in Test Benches).reg cannot be used on the left-hand side of an assign statement.reg can be used to create registers when used with always@ blocks.reg can be used to create both combinational and sequential logic.

wire and reg can be used interchangeably:

- Both can appear on the right-hand side of assign statements and always@ block = or <= signs.
- Both can be connected to the input ports of module instantiations

Logika sa 4 stanja, Reg tip podatka

- Verilog tip podatka reg može biti sintetisan i kao čvor (net) i kao registar zavisno kako se koristi.
- SV uvodi novi tip podatka logic, zamenjuje reg i wire iz Veriloga (4 stanja 0,1,X,Z)
- Multiple driver na logic podatku nije dozvoljeno, wire može da se koristi za tu svrhu.

- SV podatak logic pokriva iz VHDL-a deo funkcionalnosti std_logic podatka.
- SV logic (0,1,X,Z)
- VDL std_logic (0,1,X,Z, U, H, L, W, -)

SV operatori 1/3

- SV operatori predstavljaju kombinaciju Verilog i C operatora.
- Verilog ne podržava +=, ++, -- operatore.
- SV operatori dodele su ekvivalentni bloking dodeli.

SV operatori 2/3

Operator token	Name	Operand data types
=	binary assignment operator	any
+= -= /= *=	binary arithmetic assignment operators	integral, real , shortreal
%=	binary arithmetic modulus assignment operator	integral
&= = ^=	binary bit-wise assignment operators	integral
>>= <<=	binary logical shift assignment operators	integral
>>>= <<<=	binary arithmetic shift assignment operators	integral
? :	conditional operator	any
+ -	unary arithmetic operators	integral, real , shortreal
!	unary logical negation operator	integral, real , shortreal
~ & ~& ~ ^ ~^ ^~	unary logical reduction operators	integral
+ - * / **	binary arithmetic operators	integral, real , shortreal
%	binary arithmetic modulus operator	integral
& ^ ^~ ~^	binary bit-wise operators	integral

SV operatori 3/3

Operator token	Name	Operand data types
>> <<	binary logical shift operators	integral
>>> <<<	binary arithmetic shift operators	integral
&& -> <->	binary logical operators	integral, real , shortreal
< <= > >=	binary relational operators	integral, real , shortreal
=== !=	binary case equality operators	any except real and shortreal
== !=	binary logical equality operators	any
==? !=?	binary wildcard equality operators	integral
++ --	unary increment, decrement operators	integral, real , shortreal
inside	binary set membership operator	singular for the left operand
dist ^a	binary distribution operator	integral
{ } { { } }	concatenation, replication operators	integral
{ << { } } { >> { } }	stream operators	integral

SV literali

- Literal predstavlja fiksnu vrednost predstavljenu u izvornom kodu jezika
 - Integer and logic literals
 - Real Literals
 - Time Literals
 - String Literals
 - Array Literals
 - Structure Literals

Celobrojni / logic leterali

- Verilog syntax ; **<size>'<value>**
- SV syntax ; **'<value>**
- Nistu jako tipski (not strongly typed).
Odsecanje može da se desi.
- Sizing of literals '0, '1, 'X, 'x, 'Z, 'z

Celobrojni/logic literali primeri

```
logic [4:0] mem_address ;
```

<code>mem_address = 5'hF ;</code>	<code>// fills the bit range width 5 with 01111</code>
<code>mem_address = 5'b1 ;</code>	<code>// fills the bit range width 5 with 00001</code>
<code>mem_address = 3'bz ;</code>	<code>// fills the bit range width 5 with 00zzz</code>
<code>mem_address = 'b110111;</code>	<code>// fills only the last 5 bits, 10111</code>
<code>mem_address = 'b1xz10zzx;</code>	<code>// fills only the last 5 bits, 10zzx</code>
<code>mem_address = '1;</code>	<code>// fills all the 5 bits, 11111</code>
<code>mem_address = '0;</code>	<code>// fills all the 5 bits, 00000</code>
<code>mem_address = 'h14;</code>	<code>// fills all the 5 bits, 10100</code>
<code>mem_address = 'h1;</code>	<code>// fills the 5 bits, 00001</code>
<code>mem_address = 'hz;</code>	<code>// fills all the 5 bits, zzzzz</code>

Vremenski literali, jedinice i preciznost

- Kao i Verilog, SV dozvoljava vremenske literale u okviru koda.
 - primer: **1ns**, **4.2ps**, **1step** (podrazumeva vreme koje je navedeno)
- Nema praznog prostora između vrednosti i jedinice literala.
 - syntax: **<value><time-literal-unit>**
- Vrednost može biti integer ili floating point.
- Vremenske jedinice (fs, ps, ns, us, ms, s, step)
- **step** size biva definisan vremenskom preciznošću (time-precision).
 - ``timescale 1ns/100ps`
 - `timeunit 1ns; timeprecision 100ps;` što je sample rate naše simulacije.
- Moguće je koristiti različite preciznosti među modulima.
- Ugnježdeni moduli/ top modul: time precision se preuzima sa top nivoa.
- Mora biti prva linija koda.

Hijerarhija vremenske rezolucije

- modelsim.ini //najprioritetniji
- TB
- Fajl // najniže rangiran

Vremenski literali primer

opredeljuje vremensko kašnjenje

znači nakon (nekog kašnjenja)

Nema praznog prostora između literala i vrednosti

- #1 znači nakon 1 time unit
- **\$display** (\$time);
- Vremenski literal biće zaokružen an aktuelnu simulacionu vremensku rezoluciju
- 1ns, 2ps, 3fs; NO SPACE

```

`timescale 1ns / 100ps
module top;
  logic [2:0] a, b, c, d, e, f, g;      logic clk, clk1 = 0 ;

  initial begin
    $display ("clk_b4_assignment = %h ", clk);
    #0ns clk = 1;
    $display ("clk_after_assignment = %h ", clk);
    #100ns clk = 0; #1.2ns b = 0;
    $display ("Displaying B value after 1.2ns ? = %h ", b);
    c = 'z; d = #1step c;
    $display ("Displaying D value after 1 step ? = %h ", d);
  end

  initial #1ns $display ("display initial block 1 ");
  initial #0ns $display ("display initial block 2 ");
  initial      $display ("display initial block 3 ");

  always @(posedge clk) begin
    #100ns f = 1;
    $display ("f value after 100ns = %h ", f);
    #0ns g = 1;
    $display ("g value after 0ns  = %h ", g);
    $display ("D value after 1 last step ??? = %h ", d);
  end
endmodule

```

clk_b4_assignment = x

display initial block 3

clk_after_assignment = 1

display initial block 2

display initial block 1

f value after 100ns = 1

g value after 0ns = 1

Displaying D value after 1 last step ??? = x

Displaying B value after 1.2ns ??? = 0

Displaying D value after 1 step ??? = z

String tip podatka

- Varijabilne dužine, dinamički alociran niz bajtova
 - string `example_string = "Hello world";`
- Slično kao u većini programskih jezika podržano je:
 - Komparacija, jednakost, nejednakost, replikacija, konkatencija
 - Dodatne funkcije za rad sa stringovima (`len`, `putc`, `getc`, `toupper`, `tolower`, `icmpare`, `compare`, `substr`, `atoi`, `atohex`,....)
 - Primer: `str.len()` vraća dužinu stringa (number of characters)

Event tip podatka (opisuje događaj)

Event tip podatka omogućava komunikaciju i i
sinhronizaciju između procesa

event <variable_name> [= initial_value] ;

- Systemverilog events help to synchronize the object.
- - ex: **event** event_done ; **event** event_done = **null** ;
- event assigned with **null** becomes non-blocking

Primer upotrebe event tipa

```
module tb;

// Create an event variable that processes can use to trigger and wait
event event_a;

// Thread1: Triggers the event using "->" operator
initial begin
    #20 ->event_a;
    $display ("[%0t] Thread1: triggered event_a", $time);
end

// Thread2: Waits for the event using "@" operator
initial begin
    $display ("[%0t] Thread2: waiting for trigger ", $time);
    @(event_a);
    $display ("[%0t] Thread2: received event_a trigger ", $time);
end

// Thread3: Waits for the event using ".triggered"
initial begin
    $display ("[%0t] Thread3: waiting for trigger ", $time);
    wait(event_a.triggered);
    $display ("[%0t] Thread3: received event_a trigger", $time);
end
endmodule
```

Rezultat prethodnog primera

[0] Thread2: waiting for trigger

[0] Thread3: waiting for trigger

[20] Thread1: triggered event_a

[20] Thread2: received event_a trigger

[20] Thread3: received event_a trigger

Korisnički definisani tipovi podataka

- Korisnik može da napravi novi tip podatka
 - Class
 - Enumeration
 - Struct
 - Union
 - Typedef
- Prednost u njihovoj upotrebi su kraća imena, reusability, centralizacija

enum tip podatka

- **enum** tip definiše skup vrednosti koje podatak može da uzme
 - enum { red, blue, green } color1, color 2;
- svaki enum podatak je indeksiran.
 - enum { red =0, blue=1, green=2 } color1, color 2;
- SV enum tip je sličan C-ovskom.
- SV dozvoljava dodelu bilo koje enum varijable bilo kom enum podatku

enum metode

- first - prva vrednost
- last - poslednja vrednost
- next(N) - naredna
- prev(N) - prethodna
- num - broj vrednosti
- name – string koji vraća ime

enum primer

```
module enum_datatype;  
    //declaration  
    enum { red, green, blue, yellow, white, black } Colors;
```

```
//display members of Colors  
    initial begin  
        Colors = Colors.first;
```

```
    for(int i=0;i<6;i++) begin  
        $display("Colors :: Value of %0s \t is = %0d",Colors.name,Colors);  
        Colors = Colors.next;  
    end  
end  
endmodule
```

```
Colors :: Value of red is = 0  
Colors :: Value of green is = 1  
Colors :: Value of blue is = 2  
Colors :: Value of yellow is = 3  
Colors :: Value of white is = 4  
Colors :: Value of black is = 5
```

Strukture i unije

- SV strukture i unije su vrlo slične C-ovskim
- `struct { data_type variable ; } struct_name ;`
 struct { **bit** set ; **logic** [4:0]data; **logic** [7:8]addr; } instruct ;
 instruct.set = 1 ; *// assigns set to 1*
 typedef struct {**bit** set ;**logic** [4:0]data ;**logic** [7:8]addr;} inst_set ;
 inst_set instruct ; *// define a struct variable*
- Unije redukuju memorijske potrebe i podižu performanse.
 - **union** { **bit** set ; **logic** [4:0]data; **logic** [7:8]addr;} instruct ;
- Struktura instruct sadrži sve elemente set, data and addr. Svaki element je posebno alociran.
- Unija instruct sadrži samo jedan od navedenih elemenata set, data and addr u datom trenutku.

```
struct {  
    int val1;  
    byte val2;  
    bit [7:0] val3;  
} my_struct;
```

```
my_struct = '{32,24,9}; // my_struct.val1 = 32;
```


PACKED i UNPACKED objekti

- Packed objekti su popunjeni u memoriji kontinualno
- Mogu se kopirati na druge packed objekte
- Može im se pristupati parcijalno
- Packed objekti se mogu deklarirati samo za bitske tipove (bit, logic, reg, int)
- Unpacked objekti se popunjavaju u memoriju kako to simulatoru odgovara.
- Kopiranje, čitanje, upis unpacked objekata su pod kontrolom simulatora, memorijske lokacije nisu pod kontrolom korisnika.
- Unpacked objekti mogu biti bilo kog tipa uključujući i **real**.
- Packed unije treba da budu iste širine za sve članice.

```
struct packed { logic set; logic [7:0] addr; logic [31:0] data;} instruct ;
```



Packed i unpacked nizovi

- Ako dimenzija niza prethodi identifikatoru tako deklarišemo packed niz

`bit [7:0] c1; // packed array of scalar bit types`

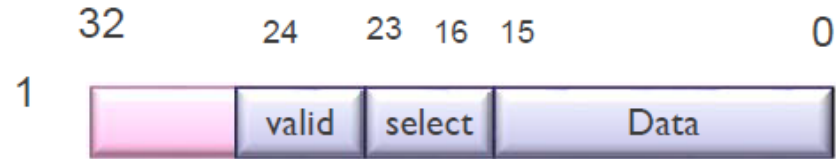
- Ako dimenzija sledi nakon identifikatora tako deklarišemo unpacked niz

`real u [7:0]; // unpacked array of real types`

Primer Packed structure

```
struct packed {  
    bit valid;  
    byte select;  
    bit [15:0] data;  
} dat_str;
```

Packed



```
top_select = dat_str.select;  
top_data = dat_str.data;  
top_valid = dat_str.valid;
```

Unpacked



Packed ili unpacked

- Podrazumevani tip je unpacked – ukoliko se ne navede

Packed structure

```
struct packed {  
    logic set;  
    logic [7:0] addr;  
    logic [31:0] data;  
} pack_s ;
```

Packed Union

```
union packed {  
    logic [31:0] set;  
    logic [31:0] addr;  
    logic [31:0] data;  
} pack_u ;
```

UnPacked structure

```
struct {  
    logic set;  
    logic [7:0] addr;  
    logic [31:0] data;  
} unpack_s ;
```

UnPacked Union

```
union {  
    logic set;  
    logic [7:0] addr;  
    logic [31:0] data;  
} unpack_u ;
```

Tagovana unija

- Strukture mogu da sadrže unije i unije mogu da sadrže strukture
- Ne tagovana unija može da se upiše kao jedan član, a da se pročita kao drugi.
- Tagovana unija čuva i vrednost člana i tag koji ga opredeljuje.
- Može se pročitati isključivo konzistentno onaj tip člana koji je opredeljen tagom.
- Tagovana unija je strogo tipizirana
- Ne može se upisati vrednost jednog člana a pročitati drugi član unije.

Deklaracije podataka

- SV podržava literale, parametre, konstante, variable, čvorove (nets) i attribute
- Verilog 2001,
 - Variable su za proceduralne dodele, a čvorovi (nets) za kontinualne dodele
- SV proširuje funkcionalnost varijabli i koristi ih i u proceduralnim i kontinulanim dodelama. (slično kao wire iz Veriloga)
- konstante
 - localparam, specparam, const
- Doseg i životni vek
 - Podaci deklarisan kao static (ili) van modula, interface-a, task-a, funkcije ima globalni doseg i statički životni vek.
 - Podaci deklarisan unutar modula, interface-a, task-a, funkcije imaju lokalni doseg.
 - Podaci deklarisan kao **automatic** imaju lokalni doseg
- Čvorovi (nets), regs i logic
 - Varijable se mogu ažurirati kontinualno i proceduralno
- alias
 - alias W[31:24] = MSB; // korisnički definisana imena

Tipovi podataka na portovima

- Verilog ima restrikcije po pitanju tipova podataka na portovima
 - Samo čvorovi (nets) su dozvoljeni kao ulazi
 - Čvorovi (nets), registri ili integeri mogu biti izlazni
- SV nema restrikcije po pitanju tipova na portovima
 - Bilo koji tip podatka može biti i na ulazima i izlazima
 - Real, Array, Structures su takođe dozvoljene

Kastovanje

- Verilog je slabo tipski jezik poput C-a, dozvoljava promet podataka različitog tipa
- Nema provere tipa tokom kompajliranja
- SV je po tom pitanju striktniji
- Koristi se castcast(`) operator za statičko kastovanje
 - <casting_type>'(<expression>)
 - <size>'(<expression>)
 - • <sign>'(<expression>)
- \$cast, za dinamičko kastovanje
 - • \$cast (dest_var, source_exp)

- Dinamičko kastovanje se koristi pri prebacivanju objekata jedne klase u drugu. Sve dok se pri izvršenju programa ne dođe do te linije nije poznato koje sve tipove objekata sadrže date varijable sadrže. Podržava OOP.
- Statičko kastovanje, realizuje se tokom kompajliranja.

Primeri kastovanja

initial begin

```
$display(unsigned'(shortint'(252423))); // unsigned of shortint of value
```

```
$display( 8'(444)); // 8bit value of 444
```

```
$display(signed'(4'(44))); // signed of 4 bit value of 44
```

```
if ($cast( val, 1 + 2 )) // if success $cast returns 1 else 0
```

```
$display ("Dynamic casting works !");
```

end

55815

-68

-4

Dynamic casting works !

assign dodela vrednosti

- Dodeljuje vrednost čvorovima (nets) i varijablama
- Dve vrste
 - Kontinualna dodela (continuous assign – čvorovima)
 - Proceduralna dodela (dodela varijablama)

Continuous Assignments

assign out_put = ina + inb + inc;

Similar to VHDL “when” statements

Procedural Assignments

always@ (ina or inb or carry_in)
out_put = ina + inb + inc;

Similar to VHDL process with
sensitivity lists

Kontrolni izrazi u SV

- Sekvencijalna kontrola
- Kontrola je sekvencijalna ako je navedena unutar sekvencijalnog koda
 - if-else Statement
 - case Statement
 - repeat loop
 - for loop
 - while loop
 - do-while
 - foreach
 - Loop Control
 - randcase Statements

- if-else izraz
 - if-else je selekcionirani izraz
- Case izraz
 - Omogućava višestruko grananje
- repeat loop
 - Ponavlja izvršenje izraza ili bloka izraza fiksni broj puta.
- for loop
 - Kreira petlju
- while loop
 - Petlja se izvršava sve dok je uslov ispunjen
- do-while
 - Uslov se proverava, nakon prolaska kroz petlju
- Foreach
 - Iterativno se prolazi kroz elemente jednoimenzionog niza ili reda (Queue)
- Loop Control
 - break i continue izrazi unutar petlje

If ... else

```
if( x == 20)
    $display(" X is equal");
else
    $display(" X is not equal");
```

Ako se selektuje samo jedan iskaz, ne trebaju begin ... end

If ... else

```
if ( x == 20)
  begin
    y = 22;
    $display(" X is equal");
  end
else
  begin
    y = 23;
    $display(" X is not equal");
  end
end
```

Ako u selekciji imamo više iskaza, trebaju begin ... End
Slično kao u Verilogu

case, casez, casex

```
1  case (case_expression) // case statement header
2      case_item_1 : begin
3          case_statement_1a;
4          case_statement_1b;
5      end
6      case_item_2 : case_statement_2;
7      default      : case_statement_default;
8  endcase
```

- case konstrukcija se sastoji od:
 - ključne reči case, casez ili casex
 - case_expression zraz u zagradi nakon ključne reči, vrednost izraza može biti konstanta ili vektor
 - case_item je vrednost sa kojom se poredi case_expression
 - case_statement su izrazi koji se izvršavaju u slučaju jednakosti case_expression i case_item-a, ukoliko ih je više postavljamo ih između begin i end
 - default je opciona mogućnost za postavljanje podrazumevanih case_statement_default

- case zahteva **potpunu jednakost** case_expression i case_item-a
- Međutim casez dozvoljava “džokere”- wildcard ? i Z koji se tretiraju kao dont care.
- primer

```
1  always @(irq) begin
2      {int2, int1, int0} = 3'b000;
3      casez (irq)
4          3'b1?? : int2 = 1'b1;
5          3'b?1? : int1 = 1'b1;
6          3'b??1 : int0 = 1'b1;
7          default: {int2, int1, int0} = 3'b000;
8      endcase
9  end
```

Tumačenje primera

- Predstavlja prioritetni dekodler
 - Ukoliko zbog džokera imamo više zadovoljenih case_item-a, prvi ima prioritet.
 - Ako se pojavi `irq=3'b111` izvršava se `3'b1?? : int2 = 1'b1;`
 - A šta biva sa `int1` i `int0`?
 - Sećamo se da smo njima dodelili podrazumevane vrednosti na početku kombinacionog procesa, dakle biće `int1=0 int0=0`

caseX

- Pored ? i Z, kao džokere tumači i X vrednost.
- Pri njegovoj primeni treba biti vrlo oprezan.

unique i priority modifikatori

- Modifikuju kontrolne izraze if, case, casez, casex
- primer unique modifikacije casez:

```
1  always @(irq) begin
2      {int2, int1, int0} = 3'b000;
3      unique casez (irq)
4          3'b1?? : int2 = 1'b1;
5          3'b?1? : int1 = 1'b1;
6          3'b??1 : int0 = 1'b1;
7      endcase
8  end
```

Tumačenje primera

- Unique pre selekcije opredeljuje da tačno jedan uslov može biti ispunjen.
 - Ukoliko je istovremeno ispunjeno više uslova
 - Ukoliko nije ispunjen ni jedan uslov (tada svakako nema podrazumevane default linije)
- u oba navedena slučaja alat će generisati upozorenje.

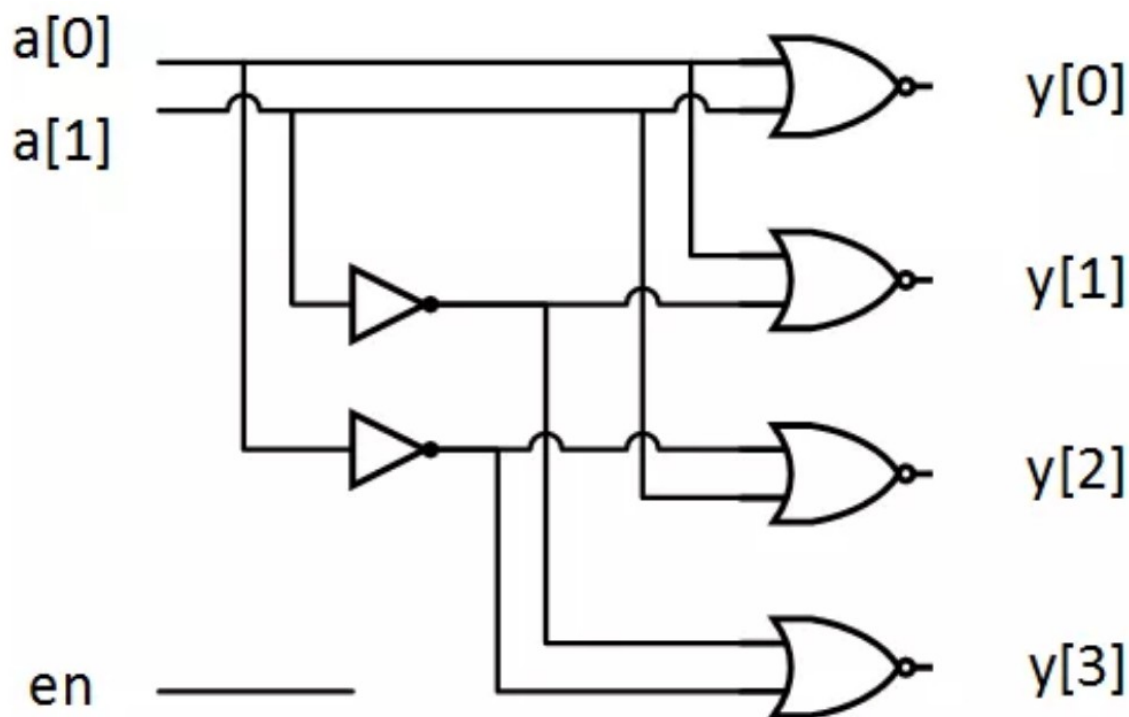
priority modifikator

- Modifikuje case izraz, tako da se generiše upozorenje ako ni jedna selekcija nije ispunjena i nema default linije.
- Primer:

```
1 logic [3:0] y;  
2 logic [1:0] a;  
3 logic      en;  
4  
5 always_comb begin  
6     y = '0;  
7     priority case ({en,a})  
8         3'b100: y[a] = 1'b1;  
9         3'b101: y[a] = 1'b1;  
10        3'b110: y[a] = 1'b1;  
11        3'b111: y[a] = 1'b1;  
12    endcase  
13 end
```

Tumačenje primera

- Priority modifikator u primeru opredeljuje da sve ne navedene case vrednosti postaju dont care i mogu se optimizovati.



SV proceduralni izrazi

- *initial // izrazi se izvrše jednom, na početku simulacije*
final // izrazi se izvrše jednom, na kraju simulacije
- *always, always_comb, always_latch, always_ff // predstavljaju beskonačnu petlju*
- *task // izrazi unutar taska se izvršavaju pozivom taska*
- *function // izrazi unutar funkcije se izvršavaju pozivom funkcije i vraće sračunatu vrednost*
- Blokirajuće (Blocking) i ne blokirajuće (Non-Blocking) dodele
- Blocking (=) Sračunavanje desne strane izraza i dodela u jednom koraku
- Non-Blocking (<=) Desna strana izraza se sračunava trenutno, ali dodela vrednosti levoj strani se odlaže dok se ne sračunaju sve desne strani sličnih izraza u datom vremenskom koraku simulacije

Blocking / Non blocking

```
module top1 ;  
  
    int x,y,z;  
    bit clk;  
    initial for (int i =0; i <5; i++) clk = #1 ~clk;  
  
    initial begin  
        x = 99;           // blocking  
        y = x;            // blocking  
        z = y;            // blocking  
        $display ("Z is ", z, "at time ", $time);  
    end  
endmodule
```

Blocking / Non blocking

Z is 99 at time 0

Blocking / Non blocking

```
module top1 ;  
    int x,y,z;  
    bit clk;  
    initial for (int i =0; i <5; i++) clk = #1 ~clk;  
    initial begin  
        x = 99;           // blocking  
        y <= x;           // Non-blocking  
        z = y;           // blocking  
        // # 1 z = y ;      // blocking  
        $display ("Z is ", z, "at time ", $time);  
    end  
endmodule
```

Blocking / Non blocking

Z is 0 at time 0

Ukoliko se dodela `z=y`; zakomentariše,

a

#1 `z=y`; aktivira tada

Z is 99 at time 1

Initial i Final

- Initial (begin) ... (end)
- Izvršava se
 - Kada simulacija počne
 - Može se koristiti više initial izraza paralelno
 - Koristi se za inicijalizaciju brojnika akcija u okviru simulacije

Initial primer

```
initial begin
```

```
    $display ("clk_b4_assignment = %h ", clk);
```

```
    #0ns clk = 1;
```

```
    $display ("clk_after_assignment = %h ", clk);
```

```
end
```

```
initial #1ns $display ("display initial block 1 ");
```

```
initial #0ns $display ("display initial block 2 ");
```

```
initial      $display ("display initial block 3 ");
```

- Initial primer resultat

```
clk_b4_assignment = x  
display initial block 3  
clk_after_assignment = 1  
display initial block 2  
display initial block 1
```


Final

- final (begin) ... (end)
- Izvršava se
 - Nakon završetka simulacije
 - Može se koristiti više final izraza paralelno
 - Koristi se za markiranje završetka simulacije
 - Kontrolisano kašnjenje unutar final bloka nije dozvoljeno #

Final primer

```
final begin
```

```
    $display ("Final value of Data = %h ", data);
```

```
    data = 1; // even assignments can be done
```

```
    $display (" Final value of Data = %h ", data);
```

```
end
```

```
final $display ("display final block 1 ");
```

```
final $display ("display final block 2 ");
```

```
final $display ("display final block 3 ");
```

- Final primer resultat

Final value of Data = 00000000

Final value of Data = 00000001

display final block 1

display final block 2

display final block 3

Loop izrazi

- for loop
 - loop varijable se mogu deklarirati unutar petlje
 - Moguće je više iniciranja, inkrementiranja, dekrementiranja
 - Lokalno deklarirane varijable su samo lokalno vidljive

Loop primer

```
int i, j, k ;  
i = 0; j = j * i ;  
for (int i = 0, j = 0; i <= 3 ; j--, i++ )  
$display ("Loop Variables i and j", i, j);
```

Loop primer resultat

Loop Variables i and j	0	0
Loop Variables i and j	1	-1
Loop Variables i and j	2	-2
Loop Variables i and j	3	-3

Do ... while loop

- Kao C-ovski do while loop
- Izrazi se izvršavaju bar jednom, pre provere uslova

Do while primer

```
module while_loop ();  
    byte a = 0;  
    initial begin  
        do begin  
            $display ("Current value of a = %g", a);  
            a ++;  
        end while (a < 10);  
  
        #1 $finish;  
    end  
endmodule
```


Do while primer rezultat

- Current value of a = 0
Current value of a = 1
Current value of a = 2
Current value of a = 3
Current value of a = 4
Current value of a = 5
Current value of a = 6
Current value of a = 7
Current value of a = 8
Current value of a = 9

Foreach loop

- Iterativno prolazi kroz članove niza
- Dlično kao do loop, ali koristi granice niza
- Loop variable moraju pokriti dimenzije niza
- Loop variable su automatske, read-only i vidljive lokalno u petlji

Foreach loop primer

```
module tb;
  int array[5] = '{1, 2, 3, 4, 5}';
  int sum;

  initial begin
    // Here, "i" is the iterator and can be named as anything you like
    // Iterate through each element from index 0 to end using a foreach
    // loop.
    foreach (array[i])
      $display ("array[%0d] = %0d", i, array[i]);

    // Multiple statements in foreach loop requires begin end
    // Here, we are calculating the sum of all numbers in the array
    // And because there are 2 statements within foreach there should
    // be a begin-end
    foreach (array[l_index]) begin
      sum += array[l_index];
      $display ("array[%0d] = %0d, sum = %0d", l_index, array[l_index], sum);
    end
  end
endmodule
```

Foreach loop rezultat

array[0] = 1

array[1] = 2

array[2] = 3

array[3] = 4

array[4] = 5

array[0] = 1, sum = 1

array[1] = 2, sum = 3

array[2] = 3, sum = 6

array[3] = 4, sum = 10

array[4] = 5, sum = 15

Jump izrazi

- SV uključuje C-ovske jump izraze **break, continue & return**
- Break – iskače iz petlje
- Continue – skače na kraj petlje
- return expr – exit iz funkcije
- return – exit iz taska ili void funkcije
- continue i break koriste se samo u petlji
- continue i break **se ne koriste u** fork .. join bloku
- funkcija sa return vrednošću mora vratiti izraz korektnog zahtevanog tipa
- disable – slično kao break i continue; za napuštanje bloka

IFF (*IF AND IF ONLY*) EVENT CONTROL

- U Verilogu, promena neke varijable, ili čvora (net) se detektuje sa @ kontolom
- Promena bilo kog bita datog vektora će trigerovati event
- SV dodaje “iff” ključnu reč, kao kvalifikator event kontrole
- Event kontrola se trigeruje ako i samo ako “iff” uslov bude ispunjen
- “iff” ima prioritet nad “or”
- Efikasan u simulaciji (proverava uslov pre ulaska u blok)

primer

```
always @( abc iff enable == 1 )  
    reg_1 <= reg2 ;
```

Tumačenje primera

```
always @(abc)  
    if (enable)  
        reg_1 <= reg2 ;
```


SV hijerarhija

- Verilog ima jednostavnu organizaciju
- SV dodaje mnoga unapređenja za hijerarhijske dizajne
 - Packages (data, types, class, subroutines)
 - Implicit port connections, .*
 - Interfaces
 - Compilation-unit scope visibility
- Hijerarhijsko ima može da specificira objekt
- SV, predaje bilo koji tip podataka kroz module ports
 - To mogu biti nets, variables, arrays, structures, etc..
 - Verilog dozvoljava samo net, reg, integer and time, kroz module ports
- Packages dozvoljavju sharing:
 - parametara, data, type, subroutines, etc..

Packages

- Packages daju mehanizam deljenja informacija
- Mogu biti u istom ili različitim datotekama
- Deklaracije unutar packages se lako dele
- Koriste se *.svh ekstenzije za heder fajlove,
- *.sv za implementaciju

packages

```
package pack1;  
  integer global_count ;  
  task print_task(int value);  
    ... ..  
  endtask  
  function get_value (input int val);  
    ... ..  
  endfunction  
endpackage : pack1
```


Referenciranje package-a

- Referenciranje pomoću operatora ::
- `package_name::data2 = package_name::data1 ;`
- Druga opcija pomoću import ključne reči
- `import package_name::* ;` //importuje celokupan sadržaj package-a
- `import package_name::specific_type ;` // importuje eksplicitan deo package-a

Package import primer

```
package pack;  
  integer global_counter ;  
  task incr ;  
    ... ..  
  endtask  
  function get_value (input int val);  
    ... ..  
  endfunction  
endpackage : pack
```

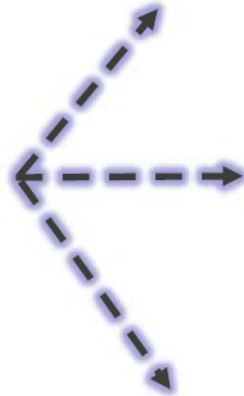
```
module mod1;  
  import pack::* ;  
  
  initial begin  
    global_counter = 567 ;  
    $display (pack::global_counter); // 567  
    incr(); // use as if it is locally declared  
  end  
endmodule : mod1
```



Compile first
package

```
module mod1;  
  import pack::* ;  
  initial begin  
    global_counter = 567 ;  
    incr(); // use as if it is locally declared  
  end  
endmodule : mod1
```

```
package pack;  
  integer global_counter ;  
  task incr ;  
    ... ..  
  endtask  
endpackage : pack
```



```
module mod1;  
  initial begin  
    pack::global_counter = 567 ;  
    pack::incr(); // use it from package  
  end  
endmodule : mod1
```

```
module mod1;  
  import pack::global_counter ;  
  initial begin  
    global_counter = 567 ; // local use  
    pack::incr(); // use it from package  
  end  
endmodule : mod1
```

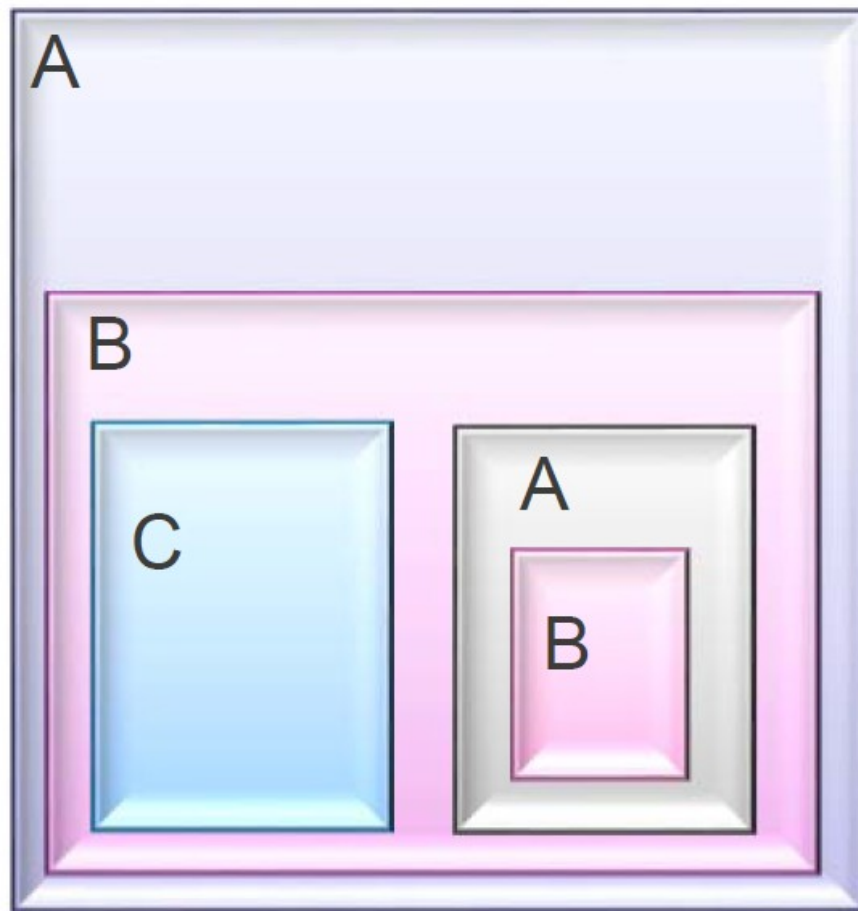
Ugnježdeni moduli

```
module nested();  
  int global_var; // Global variable available in all modules  
  
  module m1;  
    ... ..  
    initial global_var = 1;  
  endmodule  
  
  module m2;  
    int local_var; // local variable available only in m2  
    ... ..  
  endmodule  
  
  module m3;  
    ... ..  
  endmodule  
  
  initial local_var = 1; // ERROR // No Visibility !  
  
  m1 ins1();  
  m2 ins2();  
  m3 ins3();  
  
endmodule
```

Global Variable assigned locally

Nested modules have the variables encapsulated

Pristup modulu kroz top level instancu



- \$root je referenca top level instance
 - \$root . **top_level** . sub_level1 . sub_level2
 - \$root . **A** . B
 - Modul C ako pokuša da referencira A.B može biti i lokalno i na top nivou
 - \$root.A.B referencira direktno top level hijerarhije

Instanciranje modula

- Povezivanje portova (port connections)
- Pozicione konekcije
- Konekcije po imenu
- implicitne .name konekcije
- implicitne .* konekcije

Poziciono konektovanje

```
module alu (  
    output reg [7:0] alu_out,  
    output reg zero,  
    input [7:0] ain, bin,  
    input [2:0] opcode);  
// RTL code for the alu module  
endmodule  
  
// u top modulu instanciramo alu modul ovako:  
    //izlistavamo signale redom  
alu alu_inst (top_alu_out, , top_ain, top_bin, top_opcode);  
    // zero output is unconnected
```

Konekcije po imenu

```
module_name instance_name ( .port_name(expression),  
    .port_name(expression) );
```

port_name predstavlja naziv porta, kako je definisan pri deklaraciji modula
expression predstavlja naziv signala na koji se dati port iz modula povezuje tu
gde je instanciran

```
// u top modulu instanciramo alu modul ovako:
```

```
    //izlistavamo signale redom
```

```
alu alu_inst (.alu_out(top_alu_out), .zero(), .ain(top_ain, .bin(top_bin),  
    .opcode(top_opcode));
```

```
    // zero output is unconnected
```

Implicitne .name konekcije

```
alu alu_inst (.alu_out, .zero(), .ain, .bin, .opcode);
```

Neophodno je da svi portovi imaju istoimene signale u hijerarhiji u kojoj se instancira dati modul.

ukoliko neki port, na primer **zero** treba da ostane ne spojen, on ne treba da ima istoimeni signal u hijerarhiji, već se navodi .zero().

Ukoliko neki port nema istoimeni signal u hijerarhiji, alat će prijaviti grešku.

Implicitne .* konekcije

- `alu alu_inst (.*, .zero());`

ukoliko je potrebno držati zero port otvoren, navodi se ime porta i prazna zagrada `.zero()`.

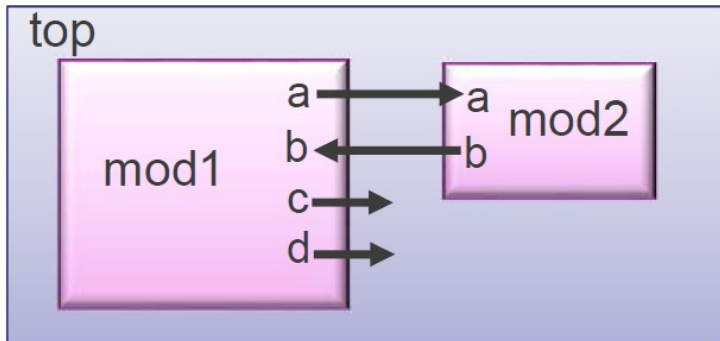
ukoliko je neki port potrebno povezati na signal u hijerarhiji različitog imena, njega treba povezati po imenu ili poziciono.

Ukoliko imamo kombinaciju, nekoliko otvorenih portova i nekoliko portova na signalima različitog imena, a ostalo se povezuje implicitno, alat prepoznaje da sve portove koji nisu opredeljeni povezuje na implicitne konekcije.

extern moduli

- Moguće je instancirati module čije deklaracije su postavljene drugde
- U datom nivou hijerarhije gde se oni instanciraju navodi se ključna reč **extern module**
- Tada se ne mora voditi računa o redosledu kompajliranja

eksterni moduli primer



File1.sv

```
module mod1 (a,b,c,d);
    input [7:0] b;           // port definitions here
    output [8:0] a,c,d;     // port definitions here
endmodule

module mod2 (a,b);
    input [8:0] a;
    output [7:0] b;
endmodule
```

File2.sv

```
extern module mod1 (a,b,c,d);    // port a,b,c,d size not specified here !!
extern module mod2
#(parameter size=8, parameter type TP = logic [7:0]) // generic parameters passed
(input [size:0] a, output TP b); // port directions with size specified here.

module top;
    wire [8:0] a,c,d;
    logic [7:0] b;
    mod1 mod1 (.*);              // implicit port mapping
    mod2 mod2 (.*);              // implicit port mapping
endmodule
```

Kontrola procesa

- SV omogućava mehanizme koji dozvoljavaju jednom procesu da prekine ili pak čeka na završenje drugog procesa.
- **wait fork** mehanizam čeka završetak procesa.
- **disable** mehanizam zaustavlja sve aktivnosti unutar bloka ili taska, bez obzira na odnos predak-naslednik
- **disable fork** mehanizam zaustavlja izvršenje procesa pod hijerarhijom predak-naslednik

ALWAYS_COMB

- Blok za modelovanje kombinacione logike u sintezi
- Nije dozvoljeno iz više always_comb blokova goniti isti signal (multiple drivers)
- always_comb je osetljiv na bilo koju promenu izvorišnih signala unutar bloka.
- ne može da sadrži timing ili event kontrole jer je sintetizibilan
- always_comb sa if ali bez else može da generiše leč, praćen upozorenjem - warning

always_comb primer

```
initial begin
```

```
b = 0 ; c = 0 ;
```

```
b = #10ns 1 ; c = #10ns 1 ;
```

```
b = #20ns 0 ; c = #20ns 0 ;
```

```
end
```

```
always_comb
```

```
begin
```

```
    a <= b & c ;
```

```
    d  = b & c ;
```

```
    $display ("A and D value is",a,d);
```

```
end
```

primer tumačenje

A and D value is x 0

A and D value is 0 0

A and D value is 0 1

A and D value is 1 0

A and D value is 0 0

ALWAYS_LATCH

- Blok za modelovanje logike u lečeve.
- Lečevi nisu gradivni elementi FPGA, ali za ASIC se mogu regularno koristiti.
- Poželjno je alatu eksplicitno definisati da željenu logiku implementira kao leč.
- Lista osetljivosti je podrazumevana, ne navodi se.
- Signali ažurirani unutar always_latch ne mogu se ažurirati i sa druge strane
- Procedura se izvršava jednom u trenutku nula i tad proverava konzistentnost izlaza i ulaza.
- *Alat može da prijavi upozorenje ukoliko ovaj blok ne sintetiše leč.*

always_latch primer

Generiše leč

```
always_latch  
begin  
  if (enable)  
    q <= 0 ;  
  else if (execute)  
    q <= d ;  
end
```

ne generiše leč- upozorenje

```
always_latch  
begin  
  if (enable)  
    q <= 0 ;  
  else  
    q <= d ;  
end
```

ALWAYS_FF

- Blok za modelovanje logike u flip-flop
- Modeluje sintetizibilni sekvencijalni registar
- Mora imati listu osetljivosti
- Može da modeluje sinhronu ili asinhronu logiku.
- alat može da prijavi upozorenje, ukoliko blok ne sintetiše *flip-flop*.

always_ff primer

```
always_ff @ (posedge clk or negedge rst)
  if (!rst)
    q <= 0 ;
  else
    q <= d ;
```

Šta se podrazumeva kao posedge / negedge

Šta se podrazumeva kao posedge / negedge

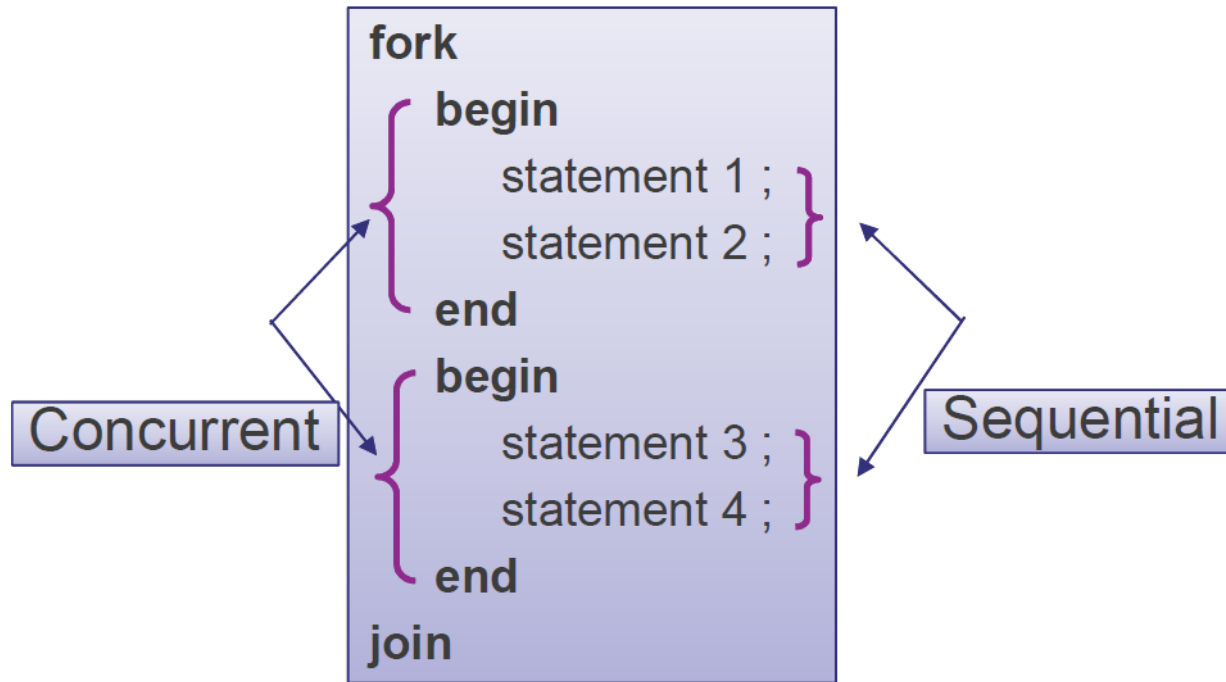
From	To			
	0	1	x	z
0	No edge	posedge	posedge	posedge
1	negedge	No edge	negedge	negedge
x	negedge	posedge	No edge	No edge
z	negedge	posedge	No edge	No edge

fork join mehanizam

- fork...join konstrukcija omogućava kreiranje konkurentnih procesa
- Verilog 2001 ima osnovnu fork...join konstrukciju
- Svi tako kreirani procesi se pokreću istovremeno.
- SV ima dve dodatne varijante fork...join konstrukcije u odnosu na Verilog:

fork ... join_any

fork ... join_none



fork...join varijacije

- fork...join :
 - Svi procesi se odvijaju konkurentno (paralelno)
 - Čeka se završetak svih procesa
- fork...join_any :
 - Čeka se završetak najbržeg procesa.
 - Svi ostali sporiji procesi nastavljaju da se odvijaju
- fork...join_none :
 - Konstrukcija se završava trenutno
 - Svi procesi nastavljaju da se odvijaju

FORK...JOIN nasuprot BEGIN...END

- begin...end izvršava sve interne procese sekvencijalno jedan za drugim
- fork...join izvršava sve interne procese konkurentno
- return izraz unutar fork...join konstrukcije nije dozvoljen

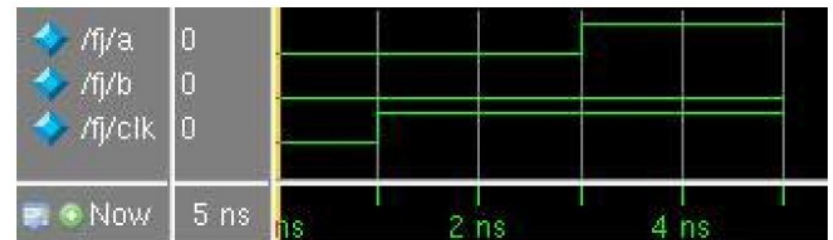
FORK...JOIN nasuprot

```
bit a, b, clk ;  
initial #1 clk = 1;  
always@(posedge clk)  
begin  
#2 a = 1;  
#1 b = a;  
#4 ;  
end  
endmodule
```



BEGIN...END

```
bit a, b, clk ;  
initial #1 clk = 1;  
always@(posedge clk)  
fork  
#2 a = 1;  
#1 b = a;  
#4 ;  
join  
endmodule
```



WAIT FORK

- wait fork blokira pozivajuće procese dok se svi pod procesi ne završe
- wait fork zaustavlja dalje izvršenje dok se svi forked blokovi ne završe
- To podrazumeva da se svi forked blokovi završe pre počeka drugih procesa

wait fork

```
task test;  
  fork  
    proc1();           // proc1 and proc2 starts at the same time (concurrently)  
    proc2();           // proc1 or proc2, any process may finish first, other keeps running  
  join_any  
  fork                 // this forked process starts after either proc1 or proc2 is completed  
    proc3();           // proc3 and proc4 runs concurrently along with unfinished process -  
    proc4();           // of the previous fork ..join_any  
  join_none  
  wait fork;          // wait for all the process to finish and give away the process control  
endtask
```

fork .. disable

- disable fork izraz prekida sve preostale aktivne fork blokove
- bolje je koristiti ga nakon join_any da bi se osiguralno da se bar jedan fork proces završi prirodno
- uz join_any, join_none i wait_fork može se realizovati kompleksan odnos paralelnih i serijskih procesa.
- Verilog disable završava sve procese u datom bloku
- SV disable fork završava proces iz pozivajućeg bloka

fork disable primer

- Dva procesa
uredjaj, čim
preostali pr

```
task test;  
  fork  
    poll_proc1(  
    poll_proc2(  
  join_any  
  disable fork;  
endtask
```

```
task test;  
  fork  
    poll_proc1();  
    poll_proc2();  
  join_any  
  disable fork;  
endtask
```

uzmu isti
đaj i

fork możliwości

