



# Projektovanje digitalnih sistema

UDP



## ■ Verilog: korisnički definisane primitive (UDP)

### ■ User Defined Primitives

- Korisnik može definisati sopstvene primitive koje se koriste na isti način kao primitive ugrađene u verilog
- Pomoću UDP se mogu modelovati
  - Kombinatorna kola – izlaz je definisan isključivo trenutnom logičkom kombinacijom na ulazima (npr. multiplekser 4/1)
  - Sekvencijalna kola – koristi se trenutna vrijednost na ulazima i trenutno stanje na izlazu da se definiše sljedeće stanje na izlazu (npr. latch, flip flop, ...)
- UDP primitive **ne** instanciraju druge primitive ili module



## ■ Verilog: UDP – nastavak

- Deklaracija počinje sa `primitive` i slijedi je lista portova:  
`primitive <udp_naziv> (  
    <naziv_izlaznog_porta>, (dozvoljen je samo jedan)  
    <nazivi_ulaznih_portova>); (jedan ili više)`
- Potom slijedi deklaracija portova:  
`output <naziv_izlaznog_porta> ;  
input <nazivi_ulaznih_portova>;  
reg <naziv_izlaznog_porta>; (opciono – samo kod sekvencijalnog UDP)`
- Kod sekvencijalnih kola slijedi inicijalizacija (samo kod njih!)  
`initial <naziv_izlaznog_porta> = <vrijednost>;`
- UDP tabela stanja:  
`table  
    <sadržaj tabele stanja>  
end table`
- Završetak deklaracije:
  - `endprimitive`



## ■ Verilog: UDP – nastavak

- UDP tabela stanja je najvažniji dio – definiše kako će se izračunati izlaz na osnovu ulaza (i trenutnog stanja)
- Tabela se modeluje kao *lookup* tabela – redovi u tabeli predstavljaju redove u logičkoj tablici istinitosti
- Pravila prilikom definisanja primitive:
  - Isključivo skalarni ulazni portovi i može ih biti više
  - Isključivo skalarni izlazni port i može biti samo jedan – navodi se prvi u listi portova
  - Tabela stanja može sadržati vrijednosti 0, 1 i x – ako se primitivi proslijedi vrijednost z tretira se kao x
  - UDP se definiše na istom nivou kao modul – **ne** može se definisati **unutar** modula (može se samo instancirati unutar modula)
  - UDP ne podržava bidirekzione portove (inout)



## ■ Verilog: kombinatorni UDP

### ■ Primjer AND kolo:

```
primitive udp_and(out, a, b);  
  // deklaracije  
  output out; // ne smije biti reg kod kombinatornih kola  
  input a, b;  
  // definicija lookup tabele  
  table  
    // Ulazi moraju biti u istom redoslijedu kao u listi portova  
    // Izlaz na kraju iza dvotačke  
    // a  b : out;  
    0  0 : 0; // jedan ulaz u tabelu se završava sa ;  
    0  1 : 0;  
    1  0 : 0;  
    1  1 : 1;  
  endtable  
endprimitive
```

## ■ Verilog: kombinatorni UDP – nastavak

- Moraju se eksplicitno specificirati sve moguće kombinacije ulaznih signala koje daju poznat izlaz
- Ako se kombinacija sa ulaza ne nalazi u tabeli na izlazu će biti **x**
- Tabela u prethodnom primjeru ne tretira slučaj kada je neki od ulaza **x**, pa bi za npr. **a=x** i **b=0** primitiva **udp\_and** na izlazu dala **x** jer se ova kombinacija ulaza ne nalazi u tabeli. Međutim, izlaz bi trebalo da bude **0**
- Tabela bi trebalo da izgleda ovako:

```
table
// a b : out;
0 0 : 0;
0 1 : 0;
1 0 : 0;
1 1 : 1;
x 0 : 0;
0 x : 0;
endtable
```



## ■ Verilog: kombinacioni UDP – nastavak

### ■ Primjer OR kolo:

```
primitive udp_or(out, a, b);
```

```
output out;
```

```
input a, b;
```

```
table
```

```
// a b : out;
```

```
0 0 : 0;
```

```
0 1 : 1;
```

```
1 0 : 1;
```

```
1 1 : 1;
```

```
x 1 : 1;
```

```
1 x : 1;
```

```
endtable
```

```
endprimitive
```

```
// pokriveni su svi slučajevi gdje je poznata vrijednost izlaza (nije x)
```

## ■ Verilog: kombinatorni UDP – nastavak

- Kod OR kola, ako je jedan ulaz jednak jedinici, izlaz će biti jednak jedinici bez obzira na vrijednost drugog ulaza
- Tabela se može kraće zapisati koristeći simbol za “bilo što” (?):

```
primitive udp_or(out, a, b);  
output out;  
input a, b;  
table  
// a b : out  
0 0 : 0 ;  
1 ? : 1 ; // ? se proširuje na 0, 1 i x  
? 1 : 1 ; // ? se proširuje na 0, 1 i x  
0 x : x ;  
x 0 : x ;  
endtable  
endprimitive
```

- Istanciraju se na isti način kao i ugrađenje Verilog primitive



## ■ Verilog: primjer komb. UDP – multiplekser 2/1

```
primitive mux2_to_1 (y, a, b, sel);  
output y;  
input a, b, sel;  
    table  
        // a  b  sel : y  
        0  ?  0  : 0; // selektuje a;  b nije važno  
        1  ?  0  : 1; // selektuje a;  b nije važno  
        ?  0  1  : 0; // selektuje b;  a nije važno  
        ?  1  1  : 1; // selektuje b;  a nije važno  
    endtable  
endprimitive
```

## ■ Verilog: primjer komb. UDP – MUX 4/1(vježba)

```
primitive mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
```

```
    output out;
```

```
    input i0, i1, i2, i3, s1, s0;
```

```
table
```

```
// i0 i1 i2 i3 s1 s0 out
```

```
    1 ? ? ? 0 0 : 1;
```

```
    0 ? ? ? 0 0 : 0;
```

```
    ? 1 ? ? 0 1 : 1;
```

```
    ? 0 ? ? 0 1 : 0;
```

```
    ? ? 1 ? 1 0 : 1;
```

```
    ? ? 0 ? 1 0 : 0;
```

```
    ? ? ? 1 1 1 : 1;
```

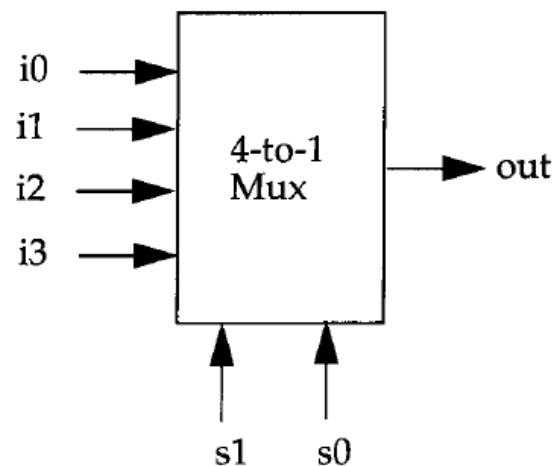
```
    ? ? ? 0 1 1 : 0;
```

```
    ? ? ? ? x ? : x;
```

```
    ? ? ? ? ? x : x;
```

```
endtable
```

```
endprimitive
```



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



- Verilog: primjer komb. UDP – multiplekser 4/1
- UDP je pogodan za implementiranje funkcija kojima je poznata tabela istinitosti: ne mora se tražiti logička šema
- Tabela stanja veoma brzo raste sa povećanjem broja ulaznih portova
- Memorijski zahtjevi kod simulacije UDP eksponencijalno rastu sa povećanjem broja ulaznih portova

## ■ Verilog: primjer komb. UDP – multiplekser 4/1

```
module stimulus;
reg IN0, IN1, IN2, IN3, S1, S0;
wire OUTPUT;
// instanciranje mux-a
mux4_to_1 moj_mux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
initial
begin
IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
#1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);
$monitor( $time, " S1 = %b, S0 = %b, OUTPUT = %b", S1, S0, OUTPUT);
S1 = 0; S0 = 0; // izaberi IN0
#1 S1 = 0; S0 = 1; // izaberi IN1
#1 S1 = 1; S0 = 0; // izaberi IN2
#1 S1 = 1; S0 = 1; // izaberi IN3
#1 $finish;
end
endmodule
```



## ■ Verilog: sekvencijalni UDP

- Izlazni port se uvijek deklariše kao **reg**
- Može se izvršiti inicijalizacija izlaza (**initial**)
- Format tabele stanja se blago razlikuje:  
`<ulaz_1> <ulaz_2> ... <ulaz_n> : <trenutno_stanje> : <sljedeće_stanje>;`
- `<trenutno_stanje>` je trenutna vrijednost izlazne promjenljive (porta)
- `<sljedeće_stanje>` se računa na osnovu trenutnog stanja i ulaznih portova
- Moraju se navesti sve kombinacije ulaznih signala da bi se izbjegle nepoznate vrijednosti na izlazu
- Specifikacija ulaznih portova u tabeli stanja se može izvršiti u smislu:
  - Nivoa ulaznih signala (*level – sensitive*)
  - Tranzicija ulaznih signala (*edge - sensitive*)

## ■ Verilog: sekvencijalni UDP – *level sensitive*

■ Najpoznatiji primjer *level-sensitive* sekvencijalnog kola je *latch*

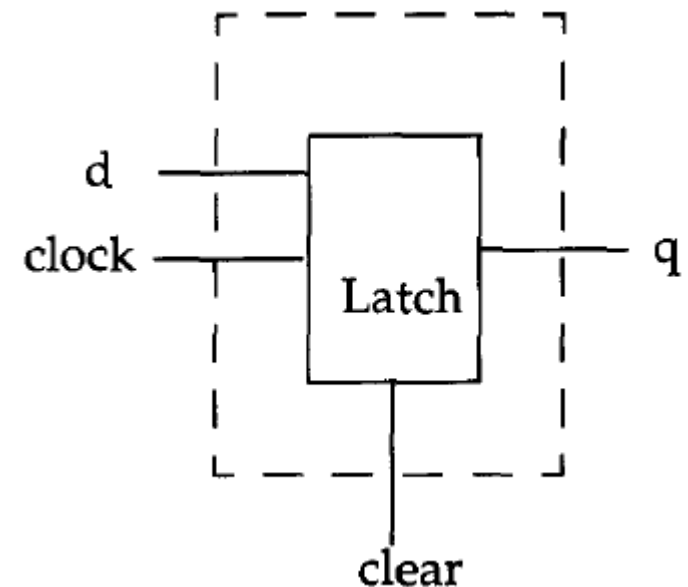
### ■ Primjer: D *latch*, sa *clear* signalom

■ Ako je *clear*=1 izlaz *q* je uvijek 0

■ Ako je *clear*=0 onda:

□  $q=d$ , kada je  $clock=1$

□  $q$  ne mijenja vrijednost, kada je  $clock=0$



■ Simbolom '—' (crtica) se označava da nema promjene vrijednosti izlaza



## ■ Verilog: primjer *level sensitive* sekv. UDP – D *latch*

```
primitive d_latch(q, d, clock, clear);  
output q;  
reg q; // sekvencijalno kolo!!!  
input d, clock, clear;  
initial // dozvoljen je samo jedan initial izraz  
    q = 0; // inicijalizacija na 0  
table  
    // d clock clear : q : q+          q+ je sledeće stanje  
    ? ? 1 : ? : 0; // resetovanje  
    1 1 0 : ? : 1; // q=d=1  
    0 1 0 : ? : 0; // q=d=0  
    ? 0 0 : ? : -; // q se ne mijenja  
endtable  
endprimitive
```

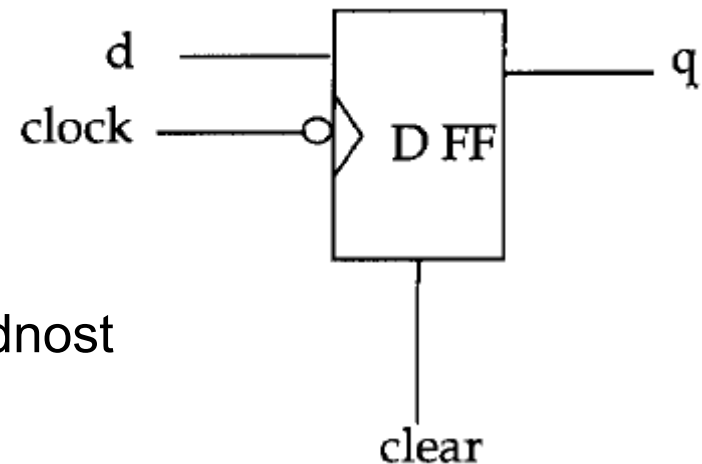
## ■ Verilog: primjer *level sensitive* sekv. UDP – D *latch*

```
module stimulus;
reg D, CLK, CLEAR;
wire Q;
d_latch (Q, D, CLK, CLEAR); // instanciranje latch-a
initial
begin
D = 0; CLK = 0; CLEAR = 1;
#1 $display("D= %b, CLK= %b, CLEAR= %b, Q= %b\n",D,CLK,CLEAR,Q);
$monitor( $time," D=%b, CLK=%b, CLEAR=%b, Q=%b", D,CLK,CLEAR,Q);
CLEAR = 0; CLK = 1;
#1 D = 1; #1 D = 0;
#1 CLK = 0;
#1 D = 1;
#1 CLK = 1;
#1 CLEAR = 1;
#1 $finish;
end
endmodule
```

	D = 0, CLK= 0, CLEAR= 1, Q= 0
1	D = 0, CLK = 1, CLEAR = 0, Q = 0
2	D = 1, CLK = 1, CLEAR = 0, Q = 1
3	D = 0, CLK = 1, CLEAR = 0, Q = 0
4	D = 0, CLK = 0, CLEAR = 0, Q = 0
5	D = 1, CLK = 0, CLEAR = 0, Q = 0
6	D = 1, CLK = 1, CLEAR = 0, Q = 1
7	D = 1, CLK = 1, CLEAR = 1, Q = 0

## ■ Verilog: sekvencijalni UDP – *edge sensitive*

- Mijenjaju stanje na ivicu (promjenu nivoa) i/ili nivo ulaznog signala
- Najpoznatiji primjer *edge-sensitive* sekvencijalnog kola je flip flop (*edge-triggered*)
- **Primjer: D flip flop koji reaguje na silaznu ivicu, sa *clear* signalom**
- Ako je *clear=1* izlaz *q* je uvijek 0
- Ako je *clear=0* onda normalno funkcioniše:
  - na silaznu ivicu signala *clock*:  $q=d$
  - u ostalim slučajevima *q* ne mijenja vrijednost





## ■ Verilog: sekvencijalni UDP – *edge sensitive*

- U tabeli stanja se tranzicije označavaju na sljedeći način:
  - (10) označava silaznu ivicu – tranziciju sa logičke 1 na logičku 0
  - (1x) označava tranziciju sa logičke 1 na x
  - (0?) označava tranziciju sa logičke 0 na 0, 1 ili x (moguća uzlazna ivica)
  - (x1) označava tranziciju sa x na 1 (moguća uzlazna ivica)
  - (??) označava tranziciju sa 0, 1 ili x na 0, 1 ili x

## ■ Verilog: primjer *edge sensitive* sekv. UDP – D *flip flop*

```
primitive edge_d_ff(q, d, clock, clear);
```

```
output q;
```

```
reg q;
```

```
input d, clock, clear;
```

```
initial
```

```
q = 0;
```

```
table
```

```
// d clock clear : q : q+ ;
```

```
  ? ? 1 : ? : 0; // q = 0 ako je clear = 1
```

```
  ? ? (10) : ? : -; // ignorisati silaznu ivicu signala clear
```

```
  1 (10) 0 : ? : 1; // q=d na silaznu ivicu
```

```
  0 (10) 0 : ? : 0; // q=d na silaznu ivicu
```

```
  ? (1x) 0 : ? : -; // zadrži q kod nepoznate tranzicije
```

```
  ? (0?) 0 : ? : -; // ignorisati uzlaznu ivicu signala clock
```

```
  ? (x1) 0 : ? : -; // ignorisati uzlaznu ivicu signala clock
```

```
  (??) ? 0 : ? : -; // ignorisati promjene signala d kad je clock stabilan
```

```
endtable
```

```
endprimitive
```

## ■ Verilog: primjer *edge sensitive* sekv. UDP – D *flip flop*

```
primitive edge_d_ff(q, d, clock, clear);
module stimulus;
reg D, CLK, CLEAR;
wire Q;
edge_d_ff (Q, D, CLK, CLEAR);
initial
begin
D = 0; CLK = 0; CLEAR = 1;
#1 $display("D= %b, CLK= %b, CLEAR= %b, Q= %b\n",D,CLK,CLEAR,Q);
$monitor( $time, " D=%b, CLK=%b, CLEAR=%b, Q=%b", D,CLK,CLEAR,Q);
CLEAR = 0; CLK = 1;
#1 D = 1; #1 D = 0;
#1 CLK = 0;
#1 D = 1;
#1 CLK = 1; #1 CLK = 0;
#1 CLEAR = 1;
#1 $finish;
end
endmodule
```

D = 0, CLK = 0, CLEAR = 1, Q = 0  
1 D = 0, CLK = 1, CLEAR = 0, Q = 0  
2 D = 1, CLK = 1, CLEAR = 0, Q = 0  
3 D = 0, CLK = 1, CLEAR = 0, Q = 0  
4 D = 0, CLK = 0, CLEAR = 0, Q = 0  
5 D = 1, CLK = 0, CLEAR = 0, Q = 0  
6 D = 1, CLK = 1, CLEAR = 0, Q = 0  
7 D = 1, CLK = 0, CLEAR = 0, Q = 1  
8 D = 1, CLK = 0, CLEAR = 1, Q = 0



## ■ Verilog: sekvencijalni UDP – *edge sensitive*

- Važno je u potpunosti specificirati UDP uzimajući u obzir sve moguće kombinacije tranzicija i nivoa ulaznih signala za koje kolo ima poznati izlaz
- U suprotnom, neka kombinacija može rezultirati nepoznatom vrijednošću
- U jednoj tabeli je moguće specificirati jednu tranziciju samo jedanput
- U jednom redu tabele je dozvoljeno specificirati samo jednu tranziciju

table

...

(01) (10) 0 ? : 1 ; // nedozvoljeno: dvije tranzicije u istom redu

...

endtable

## ■ Verilog: kraći zapis nivoa i tranzicija kod UDP

- U tabeli je dat pregled simbola koji se mogu koristiti u cilju kraćeg zapisivanja nivoa signala i njihovih tranzicija

Simbol	Interpretacija	Opis
?	0 ili 1 ili X	? znači da promjenljiva može biti 0 ili 1 ili x – ne može se specificirati u izlaznom polju
b	0 ili 1	Isto kao ?, samo što x nije uključen – ne može se specificirati u izlaznom polju
f	(10)	Silazna ivica na ulazu
r	(01)	Uzlazna ivica na ulazu
p	(01) ili (0x) ili (x1) ili (1z) ili (z1)	Uzlazna ivica uključujući x i z
n	(10) ili (1x) ili (x0) ili (0z) ili (z0)	Silazna ivica uključujući x i z
*	(??)	Sve tranzicije
-	bez promjene	Nema promjene stanja – može se specificirati samo u izlaznom polju sekvencijalnog UDP

## ■ Verilog: primjer D flip flop UDP u kraćem zapisu

- Koristeći tabelu skraćenih zapisa, primjer d flip flop primitive (tabela stanja) se može drugačije napisati na sledeći način:

```
primitive edge_d_ff(q, d, clock, clear);
```

```
...
```

```
table
```

```
// d clock clear : q : q+ ;
```

```
? ? 1 : ? : 0; // q = 0 ako je clear = 1
```

```
? ? f : ? : -; // ignorisati silaznu ivicu signala clear
```

```
1 f 0 : ? : 1; // q=d na silaznu ivicu
```

```
0 f 0 : ? : 0; // q=d na silaznu ivicu
```

```
? (1x) 0 : ? : -; // zadrži q kod nepoznate tranzicije
```

```
? p 0 : ? : -; // ignorisati uzlaznu ivicu signala clock
```

```
* ? 0 : ? : -; // ignorisati promjene signala d kad je clock stabilan
```

```
endtable
```

```
endprimitive
```



## ■ Verilog: napomene kod UDP dizajna

- Prilikom dizajna funkcionalnog bloka treba odlučiti da li ga projektovati kao modul ili kao UDP
- Kod tog izbora treba obratiti pažnju na sljedeće:
  - UDP modeluje **samo funkcionalnost**; ne modeluje tajming niti tehnologiju implementacije (CMOS, TTL, ECL, ...); modul se koristi uvijek kad treba modelovati kompletan blok sa tajmingom i tehnologijom
  - Blok se može modelovati kao UDP samo ako ima tačno **jedan** izlazni port; ako ih ima više mora se modelovati kao modul
  - **Ograničenje** u broju ulaznih portova kod UDP zavisi od konkretnog Verilog simulatora; od njih se zahtijeva da podržavaju najmanje 9 ulaza za sekvencijalni UDP i 10 ulaza za kombinacioni UDP
  - UDP se implementira kao *lookup* tabela u memoriji; sa povećanjem broja ulaza broj redova u tabeli (memorijski zahtjevi) eksponencijalno rastu – **ne treba koristiti UDP kad je veliki broj ulaza**



## ■ Verilog: napomene kod UDP dizajna – nastavak

- Nekad je **jednostavnije** modelovati neki blok kao modul – kada ima veliki broj redova u tabeli stanja (npr. multipleksor 8/1 je mnogo lakše modelovati opisom ponašanja ili tokom podataka)
- UDP tabela stanja treba biti specificirana što je moguće kompletnije: treba staviti **sve moguće kombinacije ulaznih signala** koje daju poznati izlaz; ako neka ulazna kombinacija nije specificirana izlaz će za nju imati vrijednost X
- Poželjno je **koristiti kraće zapise** (simbole) kad god je to moguće: čitljivije i konciznije
- Level-sensitive redovi u tabeli stanja imaju **prioritet** u odnosu na edge-sensitive; ukoliko se oni *sudaraju* na istom ulazu, izlaz je određen level-sensitive uslovom



# Projektovanje digitalnih sistema

*Dataflow* modelovanje



## ■ Verilog: *dataflow*

- U kompleksnijem dizajnu broj logičkih kapija može biti jako veliki i tada nije pogodno koristiti modelovanje na nivou kapija (*gate-level*)
- Implementira se **funkcija** na nivou apstrakcije višem od nivoa logičkih kola
- Uređaj se dizajnira na osnovu opisa **toka** podataka između registara ili na osnovu opisa načina kako se podaci **obrađuju**, umjesto instanciranja logičkih kola
- Automatizovani alati kreiraju kolo na nivou logičkih kapija, na osnovu opisa dizajna: **logička sinteza**
- Često se koristi kombinacija: nivo logičkih kapija, *dataflow* i *behavioral* dizajn
- Termin **RTL** (*Register Transfer Level*) se koristi za kombinaciju *dataflow* i *behavioral* dizajna



## ■ Verilog: ***continuous assignment***

- Najosnovniji izraz u *dataflow* modelovanju: koristi se za postavljanje određene vrijednosti na *net*
- Zamjenjuje logičke kapije u opisu kola – opisuje ga na višem nivou apstrakcije
- Počinje sa ključnom riječi **assign**:

*assign* <jačina\_signala> <kašnjenje> <lista\_pridruživanja>;

- <jačina\_signala> je opcionalna; *default* je *strong1/ strong0*
- <kašnjenje> je takođe opcionalno i koristi se na isti način kao kod log. kapija
- Primjer (zanemariti operacije – fokus je na specifikaciji *assign*):

*assign* izlaz = ul1 & ul2; // izlaz je tipa *net*

*assign* addr[15:0] = addr1[15:0] ^ addr2[15:0]; // vektori tipa *net*



## ■ Verilog: *continuous assignment* – karakteristike

- Sa lijeve strane u listi pridruživanja mora biti *net* (skalar ili vektor); ne može biti registarska promjenljiva
- Izrazi u listi se proračunavaju čim dođe do promjene nekog od operandi sa desne strane znaka jednakosti i dobijena vrijednost se dodjeljuje promjenljivoj sa lijeve strane znaka jednakosti
- Operandi sa desne strane znaka jednakosti mogu biti
  - Registarske promjenljive (skalari ili vektori)
  - Net promjenljive (skalari ili vektori)
  - Pozivi funkcija
- Kašnjenje se specificira u vremenskim jedinicama i izražava vrijeme u kojem se izračunata vrijednost dodjeljuje promjenljivoj
- Kašnjenje se koristi da modeluje propagaciju u realnim kolima



- Verilog: *continuous assignment* – implicitni zapis

- Može se specificirati za vrijeme deklaracije

// Standardni zapis

```
wire izlaz;
```

```
assign izlaz = ulaz1 & ulaz2;
```

// Implicitni zapis

```
wire izlaz = ulaz1 & ulaz2;
```



## ■ Verilog: *continuous assignment* – kašnjenje

- Može se specificirati na tri načina: regularno uvođenje kašnjenja, implicitno uvođenje kašnjenja i prilikom deklaracije
- Regularno uvođenje kašnjenja

*assign #10 izlaz = ulaz1 & ulaz2;*

- Kašnjenje se navodi nakon *assign*
- Kad se promijeni *ulaz1* ili *ulaz2* proći će 10 vremenskih jedinica prije nego se izvrši računanje i rezultat operacije pridruži *izlazu*
- Ako *ulaz1* ili *ulaz2* ponovo promijene vrijednosti prije isteka 10 vremenskih jedinica, uzeće se vrijednosti u trenutku računanja – *inertial delay*
- => ulazni impuls koji je kraći od vremena kašnjenja neće se propagirati na izlaz



## ■ Verilog: *continuous assignment* – kašnjenje

- Primjer za assign #10 izlaz = ulaz1 & ulaz2;

```
`timescale 1ns / 1ps
```

```
module kasnjenje_assign;
```

```
reg ulaz1, ulaz2;
```

```
wire izlaz;
```

```
assign #10 izlaz = ulaz1 & ulaz2;
```

```
initial
```

```
begin
```

```
    $monitor("ulaz1=%b ulaz2=%b izlaz=%b", ulaz1, ulaz2, izlaz);
```

```
    ulaz1=0; ulaz2=0;
```

```
    #20 ulaz1=1; ulaz2=1;
```

```
    #40 ulaz1=0;
```

```
    #20 ulaz1=1;
```

```
    #5 ulaz1=0;
```

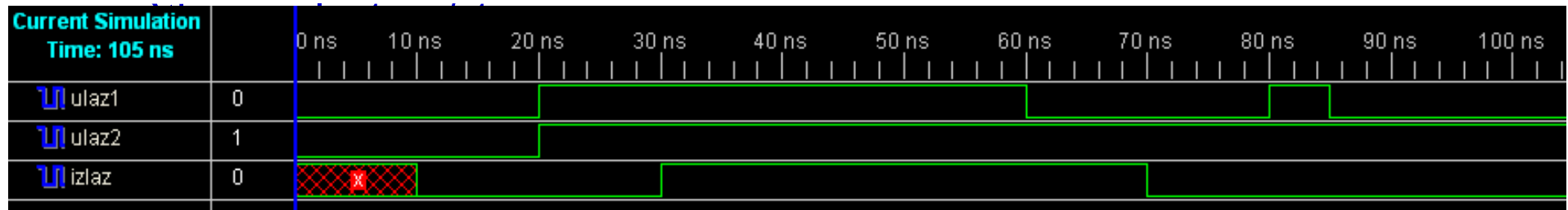
```
    #20 $finish;
```

```
end
```

```
endmodule
```

## ■ Verilog: *continuous assignment* – kašnjenje

- Primjer za `assign #10 izlaz = ulaz1 & ulaz2;`



```
assign #10 izlaz = ulaz1 & ulaz2;
initial
begin
    $monitor("ulaz1=%b ulaz2=%b izlaz=%b", ulaz1, ulaz2, izlaz);
    ulaz1=0; ulaz2=0;
    #20 ulaz1=1; ulaz2=1;
    #40 ulaz1=0;
    #20 ulaz1=1;
    #5 ulaz1=0;
    #20 $finish;
end
endmodule
```



- Verilog: *continuous assignment* – kašnjenje


- Implicitno uvođenje kašnjenja (prilikom deklaracije):

```
wire #10 izlaz = ulaz1 & ulaz2;
```

- Isto je što i:

```
wire izlaz;
```

```
assign #10 izlaz = ulaz1 & ulaz2;
```

- 
- Verilog: **kašnjenje** prilikom deklaracije *net*-a
  - Kašnjenje se može specificirati na promjenljivoj tipa *net* prilikom deklaracije, bez stavljanja *continuous assignment*-a
  - U tom slučaju svaka promjena na promjenljivoj će biti adekvatno odložena

// Net kašnjenje

wire # 10 izlaz;

assign izlaz = ulaz1 & ulaz2;

// Gornji izraz ima isti efekat kao i sledeće:

wire izlaz;

assign #10 izlaz = ulaz1 & ulaz2;



## ■ Verilog: izrazi, operatori i operandi

- *Dataflow* modelovanje opisuje dizajn preko izraza umjesto preko logičkih kapija
- Izrazi, operatori i operandi čine osnovu *dataflow* modelovanja
- **Izrazi** su konstrukcije koje kombinuju operande i operatore da proizvedu rezultat

// Primjeri izraza – kombinuju operande i operatore

$a \wedge b$

$\text{addr1}[20:17] + \text{addr2}[20:17]$

$\text{ulaz1} \mid \text{ulaz2}$

- **Operandi** mogu biti konstante, cijeli brojevi, realni brojevi, net, reg, time, bitovi (jedan ili više bitova vektor net-a ili vektor reg-a), memorije ili funkcijski pozivi (o funkcijama kasnije)



## ■ Verilog: izrazi, operatori i **operandi**

### ■ Primjeri:

```
integer brojac, ukupni_brojac;  
ukupni_brojac = brojac + 1; // brojac je cjelobrojni operand
```

```
real a, b , c;  
c = a - b; // a i b su realni operandi
```

```
reg [15:0] reg1, reg2;  
reg [3:0] reg_3;  
reg_3 = reg1[3:0] ^ reg2[3:0]; // reg1[3:0] i reg2[3:0] su reg operandi
```

```
reg povratna_vrijednost;  
povratna_vrijednost = izracunati_parnost(A, B);  
// izracunati_parnost je operand funkcijskog tipa
```



## ■ Verilog: izrazi, **operatori** i operandi

- Operatori djeluju na operande u cilju dobijanja potrebnog rezultata

- Primjeri:

d1 && d2 // && je operator nad operandima d1 i d2

!a[0] // ! je operator nad operandom a[0]

b >> 1 // >> je operator nad operandima b i 1

- Tipovi operatora:

- Aritmetički
- Logički
- Relacioni
- Jednakosti
- Nad bitovima
- Redukcije
- Pomjeranja
- Konkatencije – replikacije
- Uslovni operator

## ■ Verilog: aritmetički operatori

- Aritmetički operatori su binarni
- + i – mogu biti unarni (znak broja)

```
module aritmeticki_operatori();  
initial begin  
    $display (" 5 + 10 = %d", 5 + 10);  
    $display (" 5 - 10 = %d", 5 - 10);  
    $display (" 10 - 5 = %d", 10 - 5);  
    $display (" 10 * 5 = %d", 10 * 5);  
    $display (" 10 / 5 = %d", 10 / 5);  
    $display (" 10 / -5 = %d", 10 / -5);  
    $display (" 10 %s 3 = %d","%", 10 % 3);  
    $display (" +5      = %d", +5);  
    $display (" -5      = %d", -5);  
    #10 $finish;  
end  
endmodule
```

Operator	Opis
*	množenje
/	dijeljenje
+	sabiranje
–	oduzimanje
%	moduo

Rezultat:

```
5 + 10 = 15  
5 - 10 = -5  
10 - 5 = 5  
10 * 5 = 50  
10 / 5 = 2  
10 / -5 = -2  
10 % 3 = 1  
+5      = 5  
-5      = -5
```

## ■ Verilog: aritmetički operatori – nastavak

### ■ Primjeri:

`A = 4'b0011; B = 4'b0100; // A i B su vektori reg`

`D = 6; E = 4; // D i E su cijeli brojevi`

`A * B // Proizvod A i B. Rezultat je 4'b1100`

`D / E // Dijeljenje D sa E. Rezultat je 1. Odbacuje se decimalni dio.`

`A + B // Sabiranje A i B. Rezultat je 4'b0111`

`B - A // Oduzimanje A od B. Rezultat je 4'b0001`

`13 % 3 // Dijeljenje po modulu. Rezultat je 1`

`16 % 4 // Dijeljenje po modulu. Rezultat je 0`

`-7 % 2 // Rezultat je -1 => uzima znak prvog operanda`

`7 % -2 // Rezultat je +1 => uzima znak prvog operanda`

- Ako je vrijednost nekog bita kod operandada jednak **x**, čitav izraz ima vrijednost **x**:

`ulaz1 = 4'b101x;`

`ulaz2 = 4'b1010;`

`zbir = ulaz1 + ulaz2; // zbir će imati vrijednost 4'bx`

## ■ Verilog: aritmetički operatori – nastavak

- Unarni aritmetički operatori imaju viši prioritet od binarnih
- Negativni brojevi se reprezentuju u dvojnomo komplementu i zato ih nije preporučljivo koristiti u zapisu sa bazom brojnog sistema
- Koriste se kod cjelobrojnih i realnih promjenljivih

`-10 / 5 // Rezultat je -2`

`// ne koristiti brojeve tipa <sss> ‘<osnova> <nnn>`

`-’d10 / 5 // je ekvivalentno sa (dvojni komplement od 10)/5 =`

`// =  $(2^{32} - 10) / 5$`

`// gdje je 32 širina riječi. Ovo vodi do neočekivanog rezultata.`



## ■ Verilog: logički operatori

Operator	Opis
!	Logička negacija
&&	Logičko I (AND)
	Logičko ILI (OR)

- Daju uvijek jednobitni rezultat:  
1 (tačno), 0 (netačno), x (neodređeno)
- Ako operand nije jednak nuli, ekvivalentan je logičkoj 1 (tačno)
- Ako je operand jednak nuli, ekvivalentan je logičkoj 0 (netačno)
- Ako je bilo koji bit operanda jednak **x** ili **z**, operand je ekvivalentan **x**
- Operandi logičkih operatora mogu biti promjenljive ili izrazi

## ■ Verilog: logički operatori – primjer

```
module logicki_operatori;
initial begin
    // Logičko AND
    $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
    $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
    $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
    // Logičko OR
    $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
    $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
    $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
    // Logičko NE
    $display ("! 1'b1      = %b", (! 1'b1));
    $display ("! 1'b0      = %b", (! 1'b0));
    #10 $finish;
end
endmodule
```

Rezultat:

```
1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1
1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
! 1'b1      = 0
! 1'b0      = 1
```



## ■ Verilog: logički operatori – primjer

A = 3; B = 0;

A && B // rezultat je 0 (ekvivalentno sa log.1 && log.0)

A || B // rezultat je 1 (ekvivalentno sa log.1 || log.0)

!A // rezultat je 0

!B // rezultat je 1

A = 2'b0x; B = 2'b10;

A && B // rezultat je x (ekvivalentno sa x && log.1)

// izrazi

(a == 2) && (b == 3) // rezultat je 1 ako su oba izraza istinita

// rezultat je 0 ako je bilo koji izraz netačan



## ■ Verilog: relacioni operatori

Operator	Opis
$a < b$	a manje od b
$a > b$	a veće od b
$a \leq b$	a manje ili jednako b
$a \geq b$	a veće ili jednako b

- Izraz daje vrijednost logičke 1 ako je tačan, a logičke 0 ako je netačan
- Ako postoji bar jedan **x** ili **z** bit u operandima, izraz daje vrijednost **x**

## ■ Verilog: relaciji operatori – primjer

```
module relaciji_operatori;
integer A = 4, B = 3;
reg [3:0] X=4'b1010, Y = 4'b1101, Z = 4'b0xxx;
initial begin
    $display (" 5    <=  10 = %b", 5 <= 10);
    $display (" 5    >=  10 = %b", 5 >= 10);
    $display (" 1'bx  <=  10 = %b", 1'bx <= 10);
    $display (" 1'bz  <=  10 = %b", 1'bz <= 10);
    $display (" A <= B (%0d <= %0d) = %b", A, B);
    $display (" A > B (%0d > %0d) = %b", A, B);
    $display (" Y >= X (%b >= %b) = %b", Y, X);
    $display (" Y < Z (%b < %b) = %b", Y, Z, Y);
    #10 $finish;
end
endmodule
```

Rezultat:

```
5    <=  10 = 1
5    >=  10 = 0
1'bx  <=  10 = x
1'bz  <=  10 = x
A <= B (4 <= 3) = 0
A > B (4 > 3) = 1
Y >= X (1101 > 1010) = 1
Y < Z (1101 < 0xxx) = 0
```

## ■ Verilog: operatori jednakosti

Operator	Opis
<code>a === b</code>	a jednako b, uključujući x i z (potpuna jednakost)
<code>a !== b</code>	a nije jednako b, uključujući x i z (nije potpuna jednakost)
<code>a == b</code>	a jednako b, rezultat može biti nepoznat (logička jednakost)
<code>a != b</code>	a nije jednako b, rezultat može biti nepoznat (logička nejednakost)

- Upoređuju dva operanda bit po bit
- Ako su operandi nejednake dužine, kraći se dopunjava nulama
- Vraćaju logičku 1 ako je izraz tačan, a logičku 0 ako je netačan
- Operatori logičke (ne)jednakosti (`==` i `!=`) daju rezultat **x**, ako u nekom operandu ima bitova **x** i/ili **z**
- Operatori `===` i `!==` (*case equality*) daju kao rezultat logičku 1 ili logičku 0, zavisno da li se na svim mjestima nalaze iste vrijednosti, uključujući **x** i **z**
- Oni nikada ne daju rezultat **x**

## ■ Verilog: operatori jednakosti – primjer

```
module operatori_jednakosti();
```

```
initial begin
```

```
// Case Equality
```

```
$display (" 4'bx001 === 4'bx001 = %b", (4'bx001 === 4'bx001));
```

```
$display (" 4'bx0x1 === 4'bx001 = %b", (4'bx0x1 === 4'bx001));
```

```
$display (" 4'bz0x1 === 4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
```

```
$display (" 4'bz0x1 === 4'bz001 = %b", (4
```

```
// Case Inequality
```

```
$display (" 4'bx0x1 !== 4'bx001 = %b", (4'
```

```
$display (" 4'bz0x1 !== 4'bz001 = %b", (4'
```

```
// Logical Equality
```

```
$display (" 5      == 10      = %b", (5 == 10));
```

```
$display (" 5      == 5       = %b", (5 == 5));
```

```
$display (" 4'b0001 == 4'bx001 = %b", (4'
```

```
$display (" 4'bx001 == 4'bx001 = %b", (4'
```

```
// Logical Inequality
```

```
$display (" 5      != 5       = %b", (5 != 5));
```

```
$display (" 5      != 6       = %b", (5 != 6));
```

```
#10 $finish;
```

```
end
```

```
endmodule
```

Rezultat:

```
4'bx001 === 4'bx001 = 1
```

```
4'bx0x1 === 4'bx001 = 0
```

```
4'bz0x1 === 4'bz0x1 = 1
```

```
4'bz0x1 === 4'bz001 = 0
```

```
4'bx0x1 !== 4'bx001 = 1
```

```
4'bz0x1 !== 4'bz001 = 1
```

```
5      == 10      = 0
```

```
5      == 5       = 1
```

```
4'b0001 == 4'bx001 = x
```

```
4'bx001 == 4'bx001 = x
```

```
5      != 5       = 0
```

```
5      != 6       = 1
```

## ■ Verilog: operatori nad bitovima (*bitwise*)

Operator	Opis
~	negacija (NOT)
&	I (AND)
	ILI (OR)
^	ekskluzivno ILI
^~ ili ~^	ekskluzivno NILI

- Bit po bit: uzimaju jedan bit operanda i izvršavaju operaciju sa odgovarajućim bitom drugog operanda
- Ako operandi nisu iste dužine, kraći se dopunjava nulama
- **z** se tretira kao **x**
- Negacija je unarna operacija – nema drugog operanda

## ■ Verilog: operatori nad bitovima (*bitwise*) – primjer

```
module bitwise_operatori;  
initial begin
```

```
    // Bit Wise NOT
```

```
    $display (" ~4'b0001      = %b", (~4'b0001));
```

```
    $display (" ~4'bx001      = %b", (~4'bx001));
```

```
    $display (" ~4'bz001      = %b", (~4'bz001));
```

```
    // Bit Wise AND
```

```
    $display (" 4'b0001 & 4'b1001 = %b", (
```

```
    $display (" 4'b1001 & 4'bx001 = %b", (
```

```
    $display (" 4'b1001 & 4'bz001 = %b", (
```

```
    // Bit Wise OR
```

```
    $display (" 4'b0001 | 4'b1001 = %b", (
```

```
    $display (" 4'b0001 | 4'bx001 = %b", (
```

```
    $display (" 4'b0001 | 4'bz001 = %b", (
```

```
    #10 $finish;
```

```
end
```

```
endmodule
```

Rezultat:

```
~4'b0001      = 1110
```

```
~4'bx001      = x110
```

```
~4'bz001      = x110
```

```
4'b0001 & 4'b1001 = 0001
```

```
4'b1001 & 4'bx001 = x001
```

```
4'b1001 & 4'bz001 = x001
```

```
4'b0001 | 4'b1001 = 1001
```

```
4'b0001 | 4'bx001 = x001
```

```
4'b0001 | 4'bz001 = x001
```

## ■ Verilog: operatori nad bitovima (*bitwise*) – primjer

```
module bitwise_operatori;
```

```
initial begin
```

```
// Bit Wise XOR
```

```
$display (" 4'b0001 ^ 4'b1001 = %b", (4'b0001 ^ 4'b1001));
```

```
$display (" 4'b0001 ^ 4'bx001 = %b", (4'b0001 ^ 4'bx001));
```

```
$display (" 4'b0001 ^ 4'bz001 = %b", (4'b0001 ^ 4'bz001));
```

```
// Bit Wise XNOR
```

```
$display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
```

```
$display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
```

```
$display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
```

```
#10 $finish;
```

```
end
```

```
endmodule
```

Rezultat:

4'b0001 ^ 4'b1001 = 1000

4'b0001 ^ 4'bx001 = x000

4'b0001 ^ 4'bz001 = x000

4'b0001 ~^ 4'b1001 = 0111

4'b0001 ~^ 4'bx001 = x111

4'b0001 ~^ 4'bz001 = x111



- Verilog: operatori nad bitovima (*bitwise*)

- Razlikovati od logičkih operatora:

// X = 4'b1010, Y = 4'b0000

X I Y // bitwise operacija: rezultat je 4'b1010

X II Y // logička operacija: ekvivalentno sa 1 II 0, rezultat je 1 (tačno)