Verilog

OSNOVNI TIPOVI PODATAKA U VERILOGU

Podaci u verilogu mogu uzeti i četri vrednosti

0: logicka nula

1: logicka jedinica

x: nedefinisana vrednost (koristi se kod simulatora)

z: vrednost visoke impedanse

Vrednosti 0 i 1 su najcesce rezultat toga sto je signal povezan sa odgovarajucim izvorom niskog, odnosno visokog napona.

Vrednost x znači da signal nije definisan tj. Inicijalizovan i uglavnom se kotisti kod simulatora.

Vrednost z visoka impedansa označava otkačenu žicu tj. U pitanju je signal sa nekog isključenog kola.

Promenljive

U Verilogiu postoje dva tipa premenljivih promeljive tipa wire (žice) i registrarske promenljive reg.

Promenljiuva tipa wire služi da poveže izlaz iz nekog logičgok kola ili da se pomoću nje zada ulazna vrednost u neko logičko kolo. Svaka žica samim tim predstavlja tačku u kolu u kojoj se može izmeriti vrednost signala. Ono što je karakteristično za žice je da one same po sebi nemaju mogućnost da čuvaju neku vrednost, tj. nemaju memorijsku sposobnost.

Primer promenjivih tipa wire

```
 wire x;  // Deifni[emo jednobitni podatak tipa wire

wire [7:0]  y;  // 8-bitni podatak tipa wire
```

U nastavku se sa signalima može pristupati celovito x, ali i pojedinačnim bitovima y[5], kao i grupama bitova y[4:2] daje trobitni signal koji se sastoji iz bitova y[4], y[3] i y[2]).

Registarski tip podataka ima memorijsko svojstvo tj. U njega se može upisati vrednost.

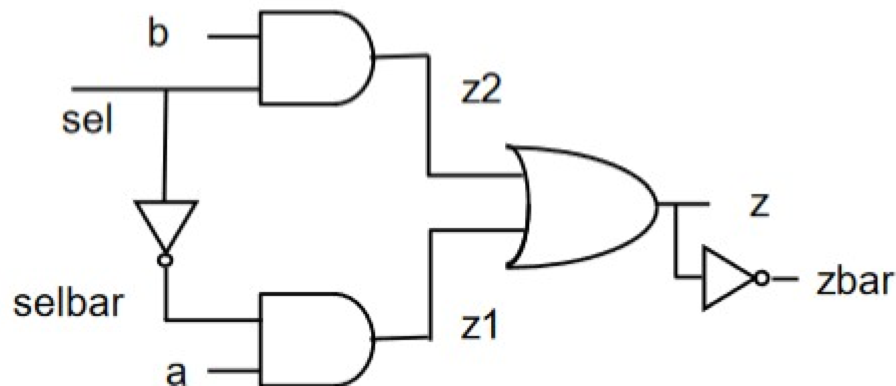
reg x; // deklariše jednobitni podatak tipa reg reg [7:0] y; // deklariše 8-bitni podatak tipa reg

x=1;

x=0;

y=8’b00000011; isto što i y=3;

Realizovati sledeće logičko kolo




```
module mux2(input a,b,sel, output z,zbar);
  assign z = sel ? b : a;
  assign zbar = ~z;
endmodule
```

Verilog takođe ima podršku za if, while, for , case strukture stim što treba voditi raluna da while petlja neće raditi na realnom hardveru jer nije unapred poznat broj iteracija.

primer if

```
if (a == 5)
  b = 15;
else
  b = 25;
```

primer case

```
reg [1:0] address;
case (address)
  2'b00 : statement1;
  2'b01, 2'b10 : statement2;
  default : statement3;
endcase
```

Petlje:

```
  for (index=0; index < 10; index = index + 2)
  mem[index] = index;
end

 while (t < 10)
    begin
     t = t + 1;
    end
```

Always struktura

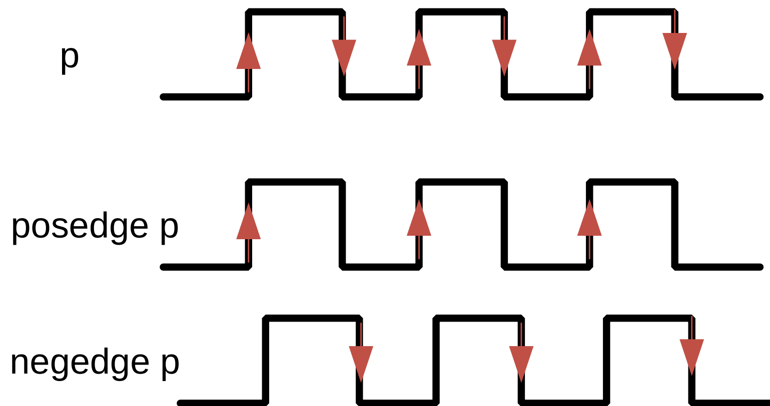
Blok oblika

```
always @(p)
begin
end
//se aktivira na svaku promenu p
always @(*)
begin
end
//Uvek se izvrsava
```

Always struktura

Postoji više varijanti

1. always @(p) // na svaku promenu p
2. always @(posedge p) // na uzlaznu ivicu od p
3. always @(negedge edge p) // na silaznu ivicu od p



Nakon always naredbe sledi blok begin end ukoliko imamo više od jedne naredbe

Always struktura

Razlika između operatora = i <=

```
always @(posedge p or negedge q)
begin
 p <= q;
q <= p;
end
```

Kod operatora <= se najpre izračunava vrednost izraza na desnoj strani a sama dodela se vrši tek na kraju bloka kada se dođe do end. Tako da gornji primer razmenjuje vrednost između p i q kad bilo koji od njih promeni vrednost.

Always struktura

Kada bi se koristio operator obične dodele =

Primer bi morao da izgleda ovako

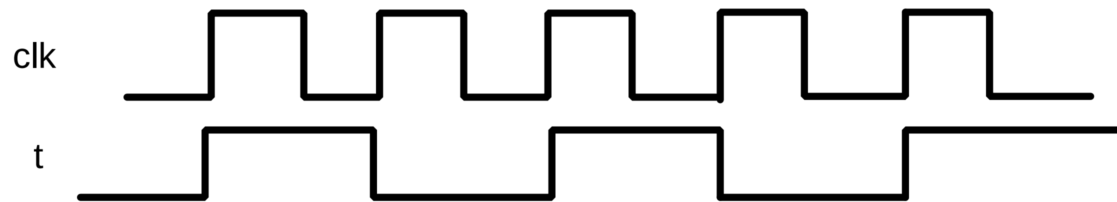
```
always @(posedge p or negedge q)
begin
 t = p;
p= q;
q=t;
end
```

Kod operatora = se najpre izračunava vrednost izraza na desnoj strani nakon čega se ta vrednost dodeli operandu na desnojh strani Tako da gornji primer razmenjuje vrednost između p i q kad bilo koji od njih promeni vrednost stim što je neophondo korišćenje i pomoćne promenljive t .

Deljenje clok-a

Ukoliko imamo signal clk koji se menja u vremena prema slici

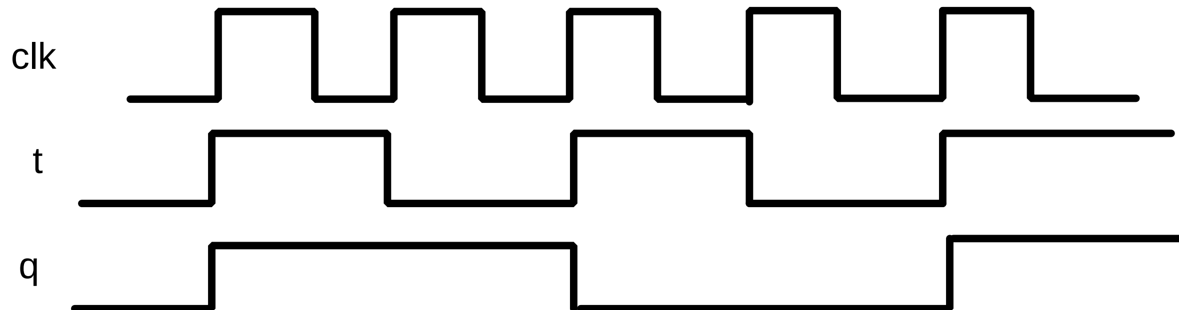
```
always @( posedge clk)
begin
t<=~t;
end
```



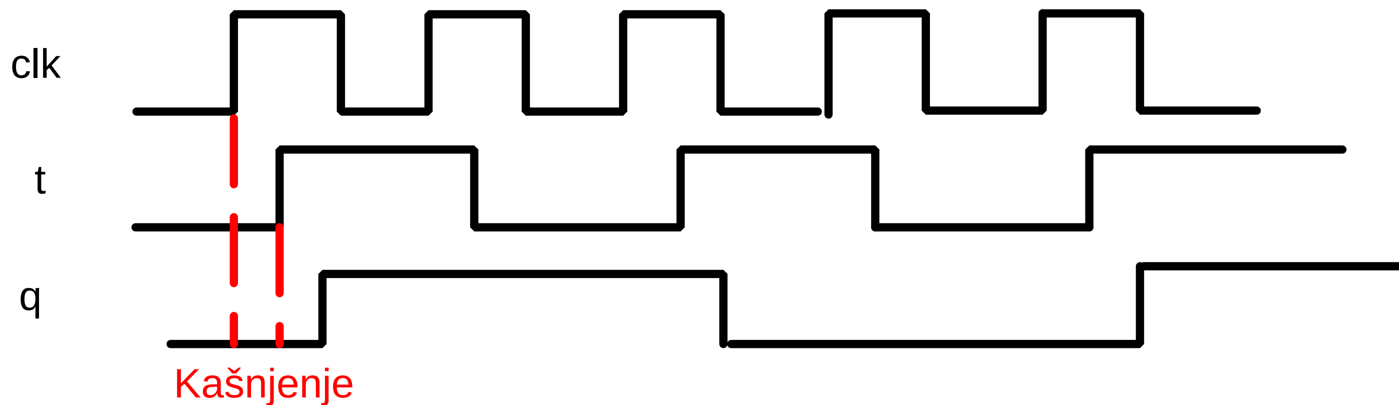
Deljenje clok-a

Ukoliko imamo signal clk koji se menja u vremena prema slici možemo dobiti signale koji su frekvencija ½, ¼. ...$1/2^n$

```
always @( posedge  clk)
t<=~t;
always @( posedge  t)
q<=~q;
```



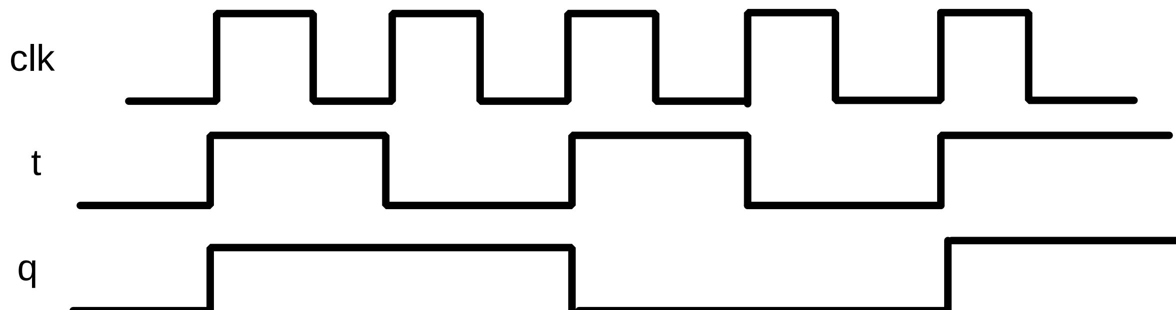
Porblem koji se ovde javlja kod realnih kola naročito ako imamo veći broj always blokova u nizu je sinhronizaicja tj kašnejnje



Iz tog razloga je bolje koristiti sledeće

```
reg [1:0] count;
always@(posedge clk)
count <= count + 1;   // counts 0..15
wire t = (count[0] == 1’b1);
wire q = (count[1] == 1’b1);
```

Na ovaj način se postiže bolja sinhronizacija

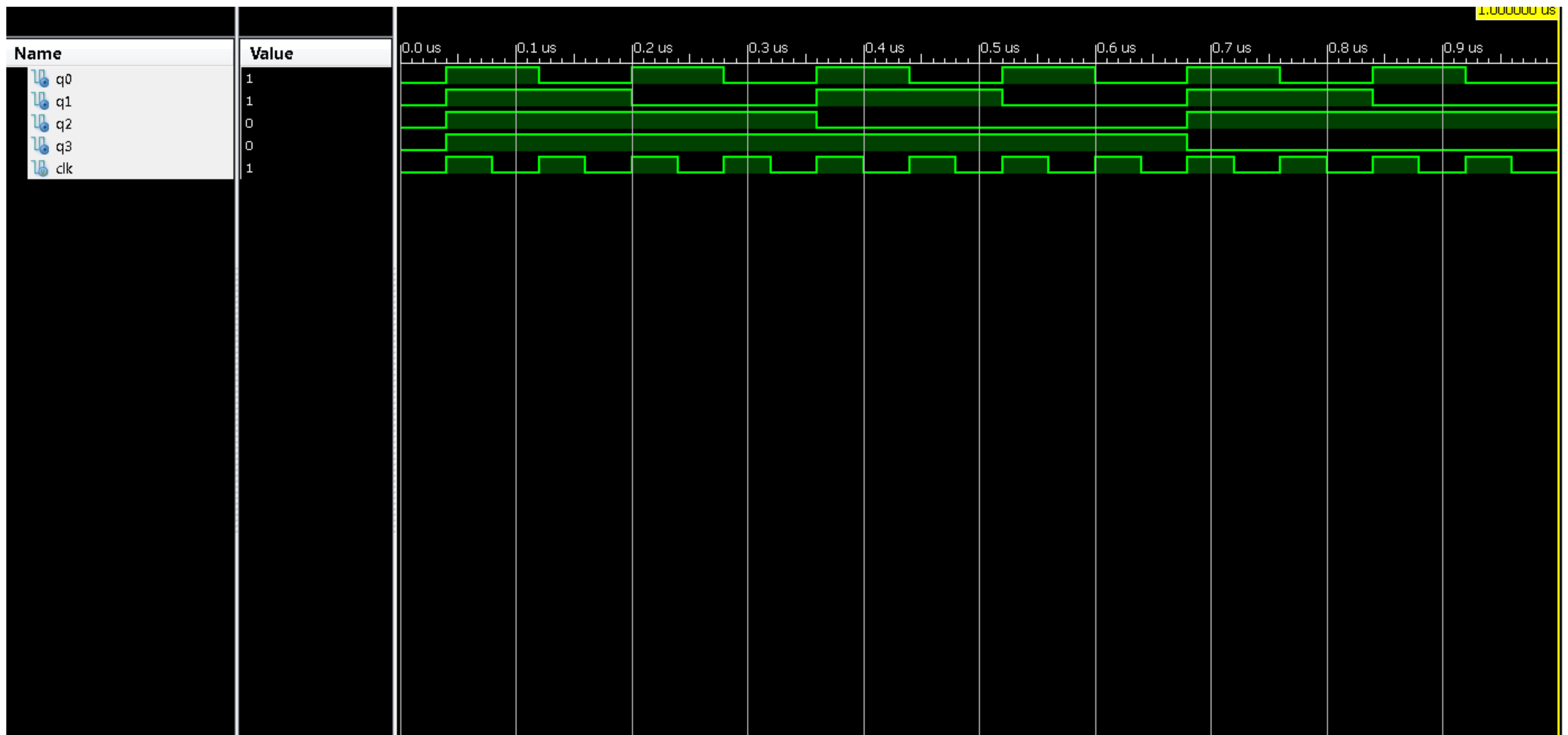


Primer

Realizovati 4-bitni brojač.

```
module counter(input clk, output q);
reg q=0;
    always @(posedge clk)
    q<=~q;
endmodule

module counter_top(input clk, output q0,q1,q2,a3
   );
counter  c1 (.clk(clk),  .q(q0));
counter  c2 ( .clk(q0),  .q(q1));
counter  c3 ( .clk(q1),  .q(q2));
counter  c4 ( .clk(q2),  .q(q3));
endmodule
```

```
module isim;
  // Inputs
  reg clk;
  // Outputs
  wire q0;
  wire q1;
  wire q2;
  wire q3;

  // Instantiate the Unit Under Test (UUT)
  counter_top uut (
  .clk(clk),
  .q0(q0),
  .q1(q1),
  .q2(q2),
  .q3(q3)
  );
  initial begin
  // Initialize Inputs
  clk = 0;
  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  end
  always @(*)
  begin
  #40
  clk<=~clk;
  end
endmodule
```

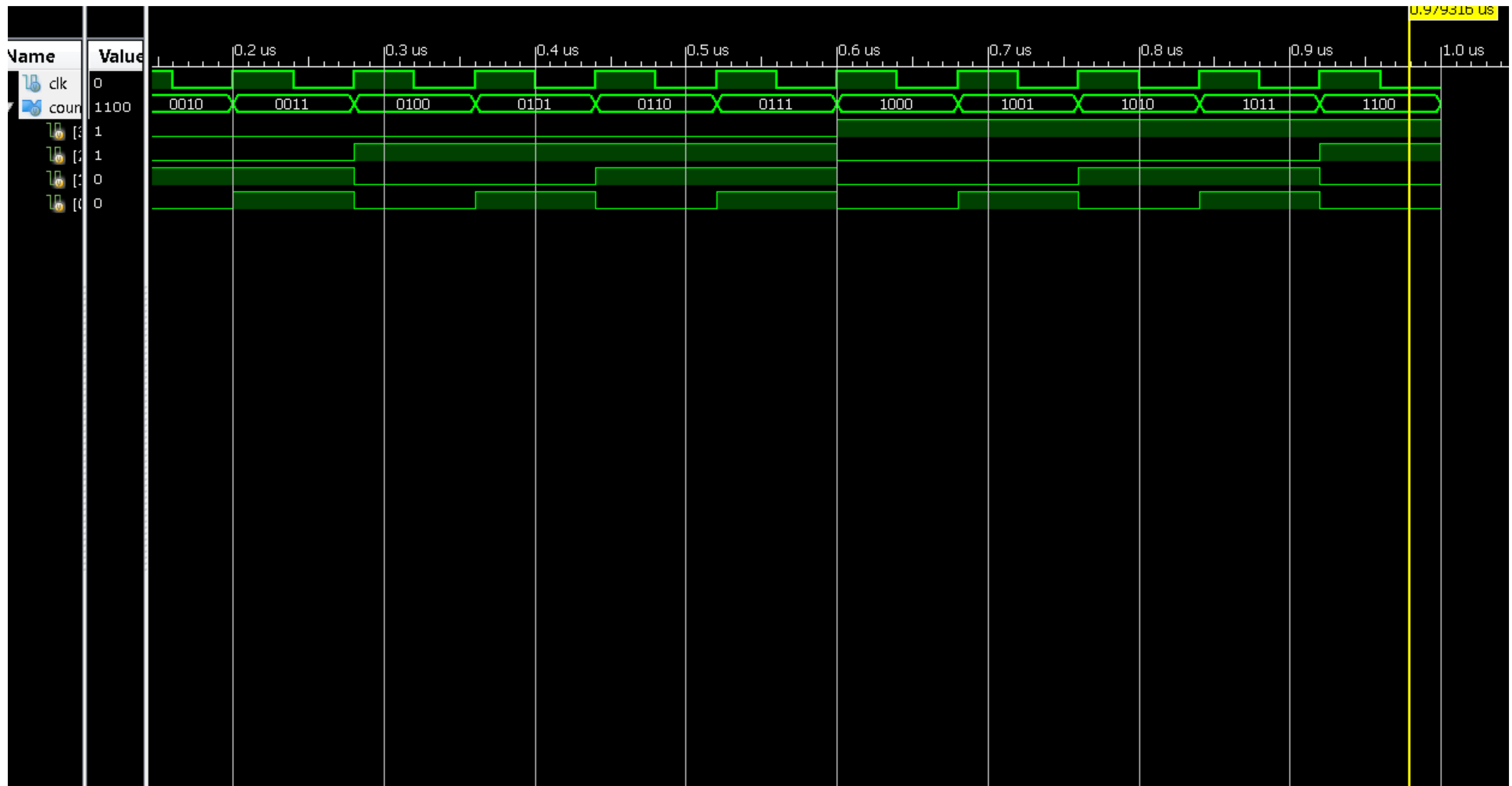


1.000000 us
Name
Value
q0
1
q1
1
q2
0
q3
0
clk
1
0.0 us
0.1 us
0.2 us
0.3 us
0.4 us
0.5 us
0.6 us
0.7 us
0.8 us
0.9 us

Drugi način

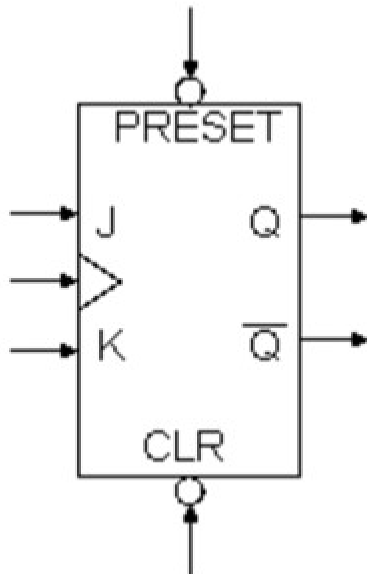
```
module isim;
	// Inputs
	reg clk;
	// Outputs
reg [3:0] count;
	// Instantiate the Unit Under Test (UUT)
	initial begin
		// Initialize Inputs
		clk = 0;
  count=4'b0000;
		// Wait 100 ns for global reset to finish
		#100;
		// Add stimulus here
	end
	always @(*)
	begin
	#40
	clk<=~clk;
	end
		always @(posedge clk)
	begin
	count<=count+1;
	end
endmodule
```

Drugi način



Primer

Realizovati J-K flip-flop



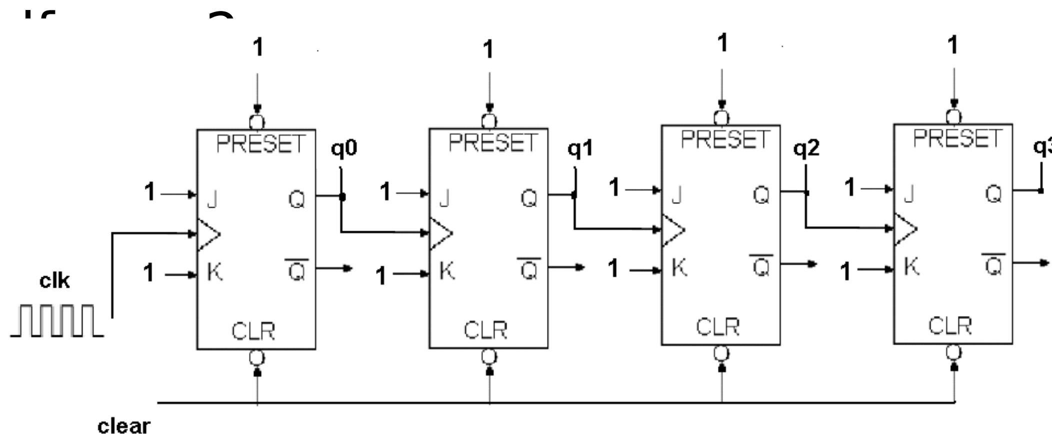
Inputs					Outputs	
preset	clear	clk	J	K	Q	$\overline{Q}$
0	X	X	X	X	1	0
1	0	X	X	X	0	1
1	1	no edge	X	X	Q	$\overline{Q}$
1	1	$\uparrow$	0	0	Q	$\overline{Q}$
1	1	$\uparrow$	1	0	1	0
1	1	$\uparrow$	0	1	0	1
1	1	$\uparrow$	1	1	toggle	

Primer

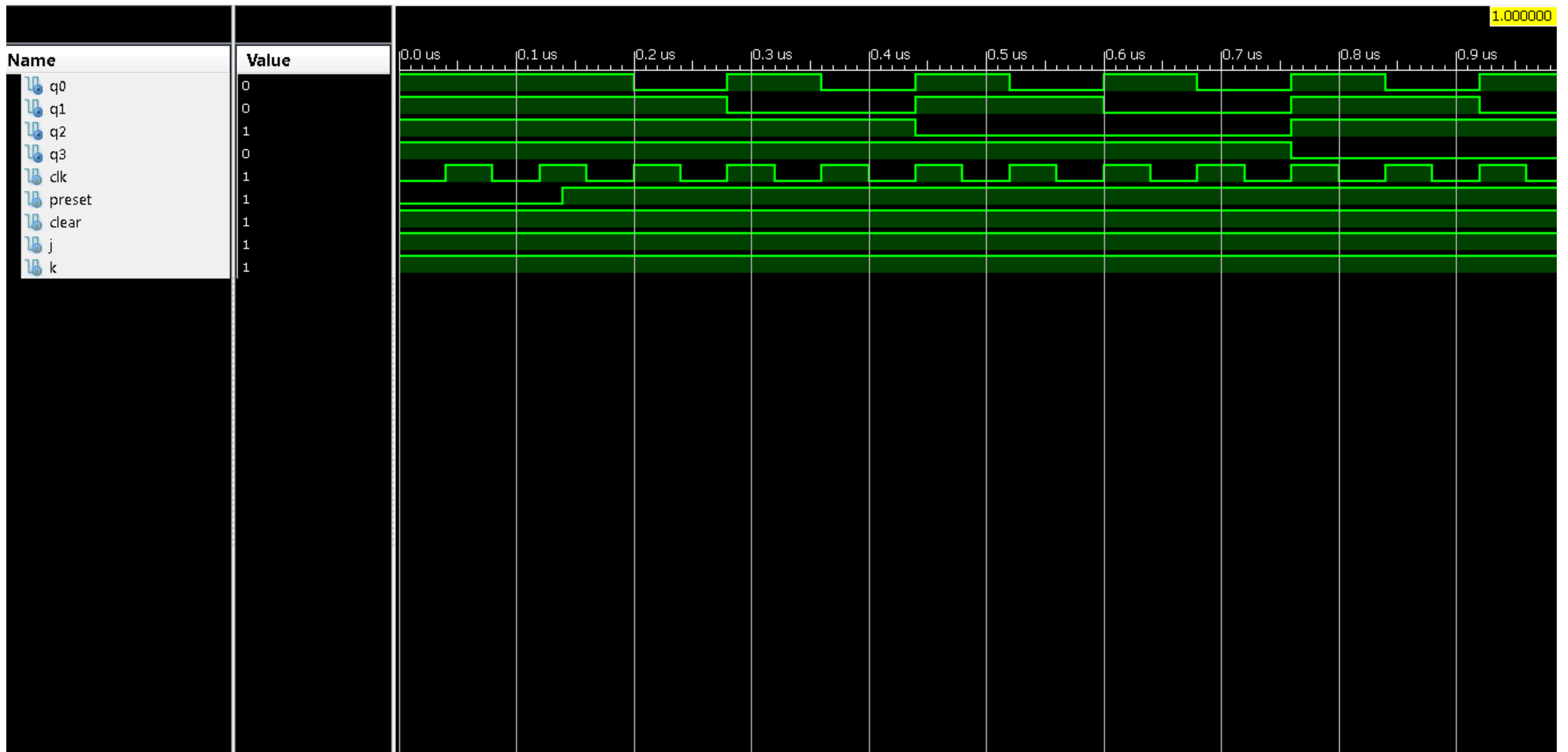
```
module jk_flip_lfop(input clk, preset,clear,j,k, output q);
 reg q=0;
 always @(posedge clk or negedge preset or negedge clear)
 begin
  if(preset==0)
       q<=1'b1;
    else
    if (clear==0)
    q<=1'b0;
        else
    case ({j,k})
     2'b00 : q <= q; //ovaj red moze i da se izbrise
     2'b01 : q <= 1'b0;
     2'b10 : q <= 1'b1;
     2'b11 : q <= ~q;
    endcase
  end
endmodule
```

Realizovati 4-bitni brojal uz pomoč J-K flip-flopova

```
module counter (input clk,preset,clear,j,k, output
q0,q1,q2,q3);
  jk_flip_lfop qq0
(.clk(clk), .preset(preset), .clear(clear), .j(j),.k(k), .q(q0));
  jk_flip_lfop qq1
(.clk(q0), .preset(preset), .clear(clear), .j(j),.k(k),  .q(q1));
  jk_flip
(.clk(q1                                                q(q2));
  jk_flip
(.clk(q2                                                q(q3));
Endmo
```

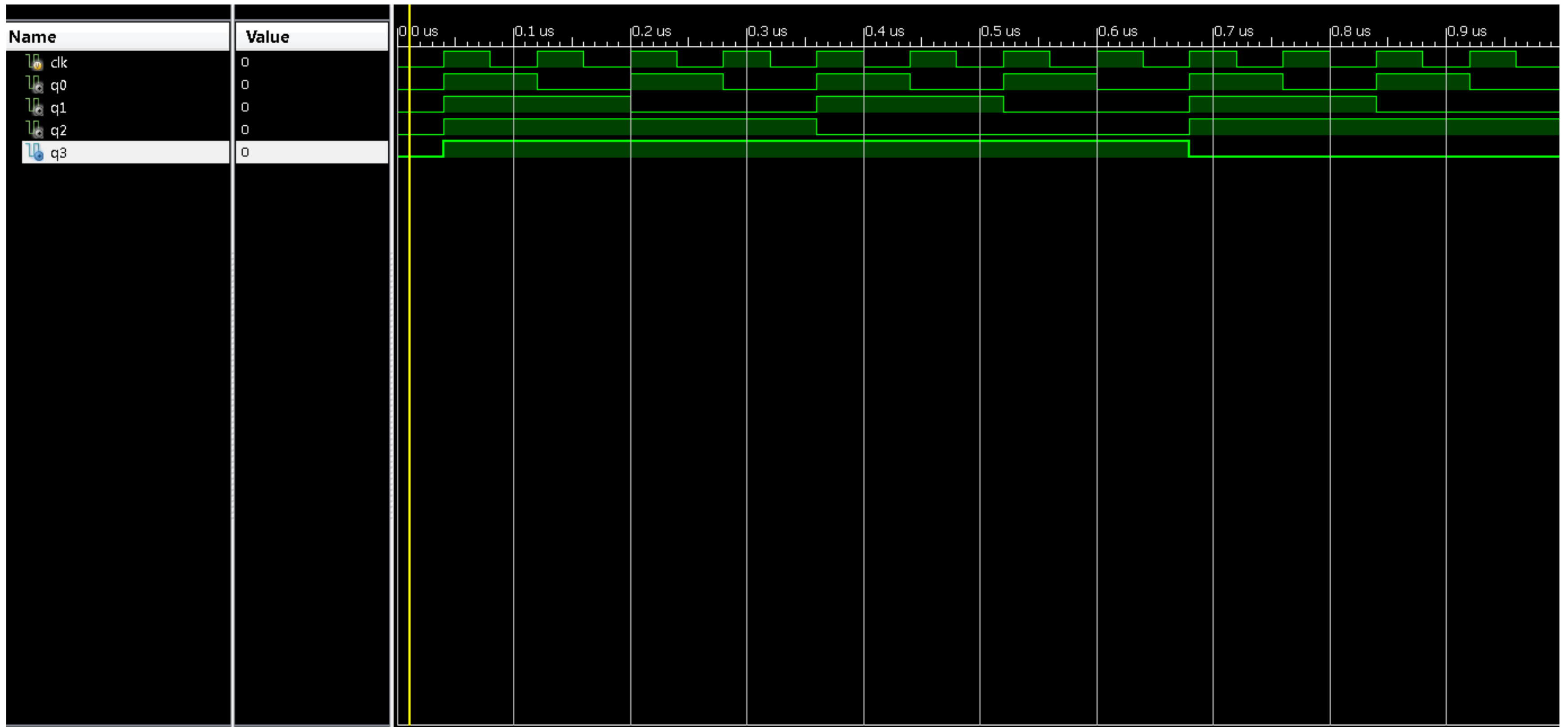


```
module sim_counter;
   // Inputs
   reg clk;
   reg preset;
   reg clear;
   reg j;
   reg k;
   // Outputs
   wire q0;
   wire q1;
  wire q2;
   wire q3;

   // Instantiate the Unit Under Test (UUT)

   counter uut (
   .clk(clk),
   .preset(preset),
   .clear(clear),
   .j(j),
   .k(k),
   .q0(q0),
   .q1(q1),
     .q2(q2),
     .q3(q3)
   );

   initial begin
   // Initialize Inputs
   clk = 0;
   preset=0;
   clear=1;
   j=1;
   k=1;
   // Wait 100 ns for global reset to finish
   #100;
   // Add stimulus here
   end
always @(*)
begin
#40;
clk<=~clk;
end
always @(posedge clk)
begin
#100;
preset<=1;
end
endmodule
```


1.000000
Name
Value
q0
q1
q2
q3
clk
preset
clear
j
k
0
0
1
0
1
1
1
1
1
0.0 us
0.1 us
0.2 us
0.3 us
0.4 us
0.5 us
0.6 us
0.7 us
0.8 us
0.9 us

```
module counter (input clk, output
q0,q1,q2,q3);
  jk_flip_lfop qq0 (.clk(clk),.q(q0));
  jk_flip_lfop qq1 (.clk(q0),.q(q1));
  jk_flip_lfop qq2 (.clk(q1),.q(q2));
  jk_flip_lfop qq3 (.clk(q2),.q(q3));
Endmodule
```

```
module jk_flip_lfop(input clk,output q);
 reg q=0;
 always @(posedge clk)
 begin
q<=~q;
/*
  if(preset==0)
    q<=1'b1;
    else
    if (clear==0)
    q<=1'b0;
     else
    case ({j,k})
     2'b00 : q <= q; //ovaj red moze i da se izbrise
     2'b01 : q <= 1'b0;
     2'b10 : q <= 1'b1;
     2'b11 : q <= ~q;
    endcase     */
  end
endmodule
```



Name
Value
clk
q0
q1
q2
q3
0
0
0
0
0
0 us
0.1 us
0.2 us
0.3 us
0.4 us
0.5 us
0.6 us
0.7 us
0.8 us
0.9 us