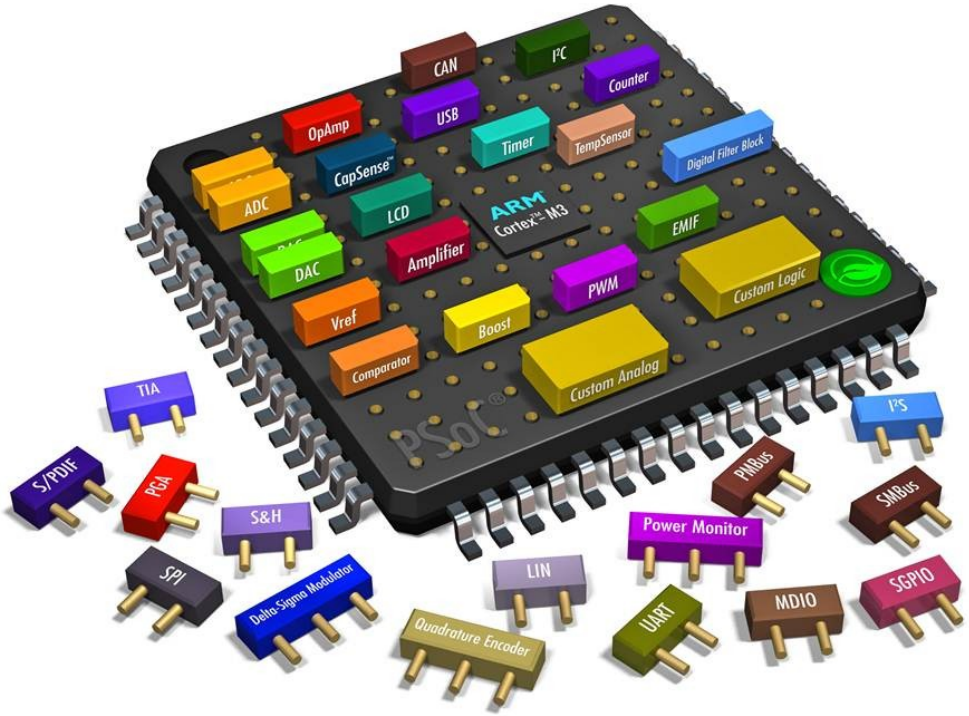
Újrakonfigurálható eszközök



3. Verilog blokkok és struktúrák

Végh János: Bevezetés a Verilog hardverleíró nyelvbe c. jegyzete nyomán

Tartalom

- Rendszertaszkok és -függvények
- Formátumozott kiíratás a naplófájlba ($monitor)
- Az idő kezelése a szimuláció során
- Az **always** blokk és az eseményvezérlés
- Szekvenciális áramkörök tervezési szempontjai
- Blokkoló és nemblokkoló értékadások
- **if - else** vezérlő utasítás
- Viselkedés-alapú tervezés

Rendszertaszkok és -függvények

- A **$monitor** és a **$display** rendszertaszkok a szimulátor naplófájlba írnak. A különbség annyi, hogy a **$display** csak egyszer ír a naplófájlba, nem végez folyamatos monitorozás.
- **Formátumozott kiíratás:** egy formátumvezérlő szöveglánc mondja meg, hogy a paramétereket milyen formátumban kell kiírni. Pl. `$monitor("t=%3d x=%d,y=%d,z=%d \n",$time,x,y,z);`
  - **%d** decimális kiíratás
  - **%4b** bináris kiíratás 4 számjeggyel
  - **%h** hexadecimális kiíratás
- Az idő formátumának megadása: `$timeformat(-9, 1, "ns", -7);`
  -9: időegység (ns), 1: számjegyek száma, "ns": string, -7: mezőszélesség
- **$stop** – töréspont
- **$finish** – szimuláció vége

Az idő kezelése

- Az előző előadás példáiban az idő dimenziótlan mennyiség volt.
- Verilogban az idő mértékegységét a `timescale` direktívával adhatjuk meg. Például:

```
timescale 1ns/100ps  //időegység/pontosság
```

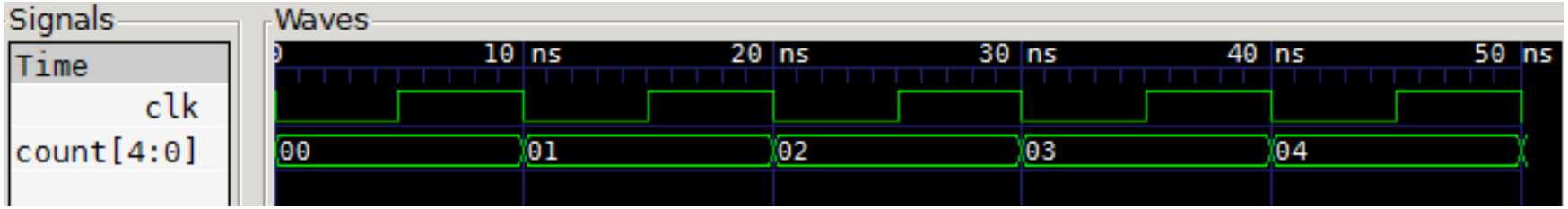
- Minden késleltetés a `timescale` direktívával megadott mértékegység-ben értendő. Például a fenti időegységgel számolva a **#10** érték egy 10 ns-os procedurális késleltetést jelent.
- Minden időérték a megadott pontosság egész számú többszörösére lesz kerekítve.
- Ha több modulban eltérő időskálát adunk meg, akkor a fordítási sorrendtől függ, hogy melyik időskála érvényesül (többnyire az utolsónak megadott érvényesül).
- Azokban a modulokban, amelyekben nem adunk meg időskálát, a saját időskálájukat az előzőleg fordított modulokból származtathatják.

Az always blokk

- A procedurális blokkok másik formája az **always** blokk, ami abban különbözik az **initial** blokktól, hogy nem csak egyszer hajtódik végre, hanem végtelen ciklust definiál.
- Aminek egy **always** blokkban értéket adunk, azt regiszterként kell definiálni.

```
module ClockGen(clk);
  output clk;     // a kimenet
  reg clk;
  initial
    clk = 0;
  always
    #5 clk = ~clk ;
endmodule
```

```
module counter(count);
  output [4:0] count;
  reg [4:0] count;
  initial
    count = 0;
  always
    #10 count = count + 1;
endmodule
```



Az always blokk

- Az előző oldali példákat az alábbi próbapaddal vizsgáltuk (**always_tb.v**) :

```
`timescale 1ns / 100ps
module test;
  reg clk;
  reg [4:0] count;
  initial begin
     $dumpfile("test.vcd");
     $dumpvars(0,test);
     clk = 0;
     count = 0;
     #50 $finish;
  end
  always #5 clk = !clk;
  always #10 count = count + 1;
  initial
     $monitor("time = %t, count = %4b (%0d)", $time, count, count);
endmodule
```

Megjegyzés:
Az előző oldali **ClockGen** és **counter** modulokat lustaságból beledolgoztuk a **test** modulba...

Futtatás:

```
> iverilog -o test always_tb
> vvp test
time =                    0, count = 00000 (0)
time =                  100, count = 00001 (1)
time =                  200, count = 00010 (2)
time =                  300, count = 00011 (3)
time =                  400, count = 00100 (4)
time =                  500, count = 00101 (5)
>gtkwave test.vcd
```

Kombinációs logikai eseményvezérlés

- Az **always** utasítás után szerepelhet egy @ és egy ún. érzékenységi lista, amelyben felsorolt események hatására lefut az **always** blokk.
- Az alábbi példában a komparátor két bemenőjele bármelyikének megváltozása az az esemény, melynek hatására a kiértékelési procedúrát újra le kell futtatni.
- Az **always** blokk csak egy utasítást tartalmazhat. Ha több tennivaló van, azokat össze kell fognunk a **begin ... end** párral.

```
module comparator(x,y,z);
  input wire x;
  input wire y;
  output reg z;
  reg p,q;

  always @(x,y)
    begin
      p = (~x & ~y);
      q =  x & y;
      z = p |q;
    end
endmodule
```

comparator.v

Megjegyzés:

Az alábbi kétféle írásmód ekvivalens

```
always @( a or b or c)

always @( a, b, c)
```

Forrás: referencedesigner.com/tutorials/verilog/verilog_16.php

Kombinációs logikai eseményvezérlés

- Az előző oldali komparátor modult az alábbi próbapaddal ellenőrizhetjük.

comparator_tb.v

```
module test;
  reg x;
  reg y;
  wire z;
  comparator uut (x,y,z);
  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,test);
       x = 0;
       y = 0;
    #20 x = 1;
    #20 y = 1;
    #20 x = 0;
    #20 y = 0;
    #40 $finish;
  end

  initial begin
    $monitor("x=%d,y=%d,z=%d \n",x,y,z,);
  end
endmodule
```

```
> iverilog -o test comparator.v comparator_tb.v

> vvp test
VCD info: dumpfile test.vcd opened for output.

x=0,y=0,z=1

x=1,y=0,z=0

x=1,y=1,z=1

x=0,y=1,z=0

x=0,y=0,z=1
```

Forrás: referencedesigner.com/tutorials/verilog/verilog_16.php

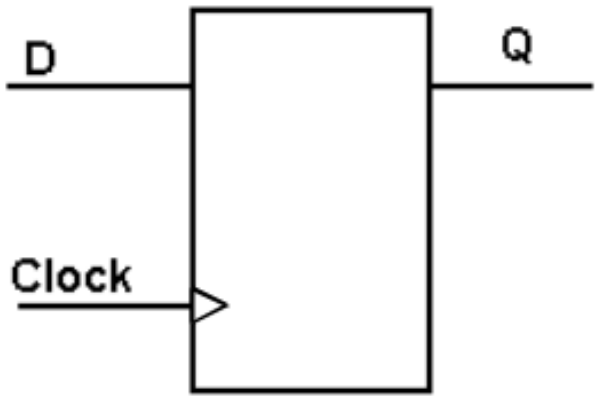
Eseményvezérlés sorrendi áramköröknél

- A sorrendi áramkörök véges állapothalmazzal rendelkeznek, s a kimenetek állapota a bemenetek pillanatnyi állapotán kívül az előzményektől (az előző állapottól) is függ.
- A szekvenciális áramköröknél általában van egy globális órajel, és az áramkörben az állapotváltások az óra felfutó vagy lefutó élénél történnek.
- **D flip-flop:** az órajel pozitív élénél a D bemenet pillanatnyi állapota rögzül a Q kimeneten.

dff.v

```
module dff (
    input wire Clock,
    input wire D,
    output reg Q
);

  always @(posedge Clock)
      Q = D;
endmodule
```



Forrás: referencedesigner.com/tutorials/verilog/verilog_31.php

Eseményvezérlés sorrendi áramköröknél

```
`timescale 1ns / 1ns
module test;
  reg Clock = 0;
  reg D = 0;
  wire Q;
  dff d1 (Clock,D, Q);
  integer i;
  initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,test);
    #8  D= 1;
    #10 D= 0;
    #10 D= 0;
    #10 D =1;
    #10 D =0;
    #10 D = 1;
    #40 $finish;
  end
  always #10 Clock = ~Clock;

  initial begin
    $monitor("Clock=%d,D=%d,Q=
%d\n",Clock,D,Q);
  end
Endmodule
```

dff_tb.v

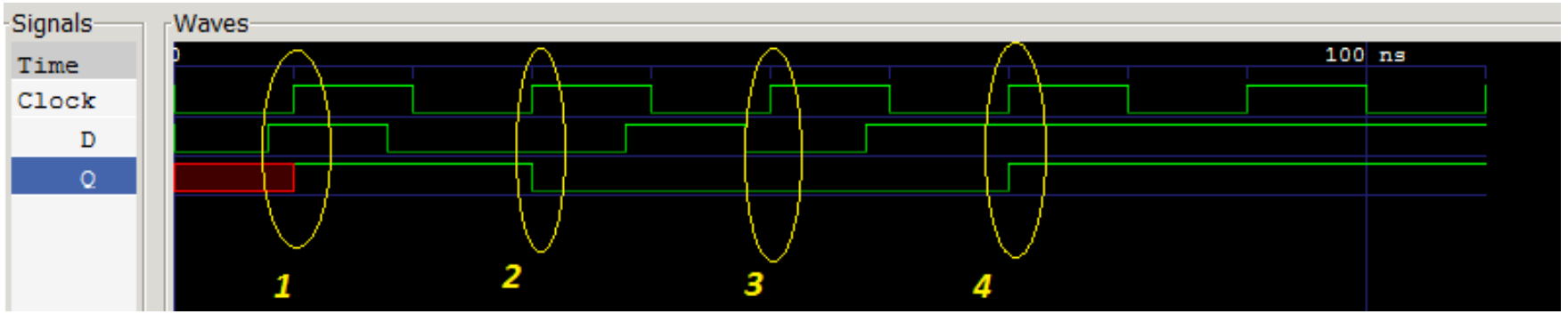
Futtatás:

```
> iverilog -o test dff_tb.v dff.v

> vvp test

> gtkwave test.vcd
```

Forrás: referencedesigner.com/tutorials/verilog/verilog_31.php



Szimulációs versenyhelyzetek elkerülése

```
// Egyszerre írjuk/olvassuk b-t
always @(posedge clock)
  b = a;

always @(posedge clock)
  c = b;
```

= blokkoló értékadás

- A Verilog nem-determinisztikus, így nem tudható, hogy `c` értéke `b` lesz, vagy `a`.
- A szimulációs versenyhelyzet elkerülésére használjunk nem-blokkoló értékadásokat! Ezeknél az értékadás csak az összes jobboldali kifejezés kiértékelése után történik.

```
// Nem-blokkoló értékadások
always @(posedge clock)
  b <= a;

always @(posedge clock)
  c <= b;
```

<= nem-blokkoló értékadás

Nem-blokkoló értékadások

```
x <= y + z;
```

- A közönséges nem-blokkoló értékadás végrehajtási sorrendje a következő:

1) Kiértékelésre kerül a jobboldali kifejezés, az eredmény eltárolódik
2) Végrehajtásra kerül minden más utasítás (kivéve a nem-blokkoló értékadásokat)
3) Végrehajtásra kerül az értékadás

Értékadáson belüli késleltetések

```
x <= #5 y + z;
```

- A fenti értékadás végrehajtási sorrendje a következő:

1) Kiértékelésre kerül a jobboldali kifejezés, az eredmény eltárolódik
2) Eltárolódik az értékadás időpontja (jelen esetben: t + 5)
3) Az előírt időpontban végrehajtásra kerül minden más utasítás (kivéve a nem-blokkoló értékadásokat)
4) Végrehajtásra kerül az értékadás az eltárolt adattal

Értékadáson belüli eseménykezelés

```
x <= @(posedge clk) y + z;
```

- A fenti értékadás végrehajtási sorrendje a következő:

1) Kiértékelésre kerül a jobboldali kifejezés, az eredmény eltárolódik
2) Előjegyzésre kerül az értékadás, hogy a következő eseménykor hajtódjon végre
3) Az előírt időpontban (a várt eseménykor) végrehajtásra kerül minden más utasítás (kivéve a nem-blokkoló értékadásokat)
4) Végrehajtásra kerül az értékadás az eltárolt adattal

Blokkoló és nem blokkoló értékadások

- Mi lesz az eredménye az alábbi értékadásoknak?

```
module blocking_nonblocking();

reg a,b,c,d;

initial begin
  $dumpfile("test_bnb.vcd");
  $dumpvars(0,blocking_nonblocking);
  #10 a = 0;
  #11 a = 1;
  #12 a = 0;
  #13 a = 1;
end

initial begin
  #10 b <= 0;
  #11 b <= 1;
  #12 b <= 0;
  #13 b <= 1;
end
```

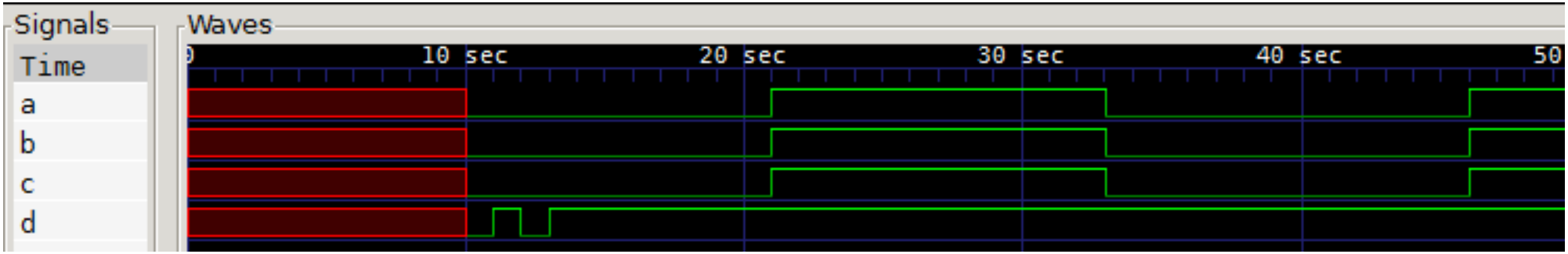
```
initial begin
   c = #10 0;
   c = #11 1;
   c = #12 0;
   c = #13 1;
end

initial begin
   d <= #10 0;
   d <= #11 1;
   d <= #12 0;
   d <= #13 1;
end

initial begin
  $monitor("TIME = %g A = %b B = %b C = %b D
= %b",$time, a, b, c, d);
  #50 $finish;
end
endmodule
```

Blokkoló és nem blokkoló értékadások

- Az utolsó eset kivételével az utasítások blokkolják a következőt: a és b esetben a külső késleltetés miatt, a c esetben pedig a blokkoló értékadás miatt. A d esetben azonban a nem-blokkoló értékadás miatt egyszerre mindegyik értékadás előjegyzésre kerül




```
#10 a = 0;
#11 a = 1;
#12 a = 0;
#13 a = 1;
```

```
#10 b <= 0;
#11 b <= 1;
#12 b <= 0;
#13 b <= 1
```

```
c = #10 0;
c = #11 1;
c = #12 0;
c = #13 1;
```

```
d <= #10 0;
d <= #11 1;
d <= #12 0;
d <= #13 1;
```

Blokkoló és nem blokkoló értékadások

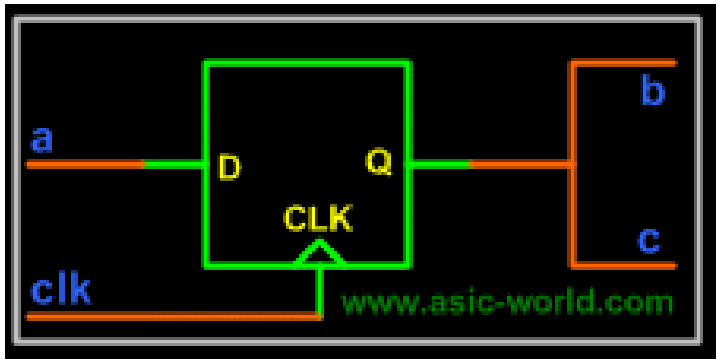
- Hogyan szintetizálódik a blokkoló és a nem-blokkoló értékadás?

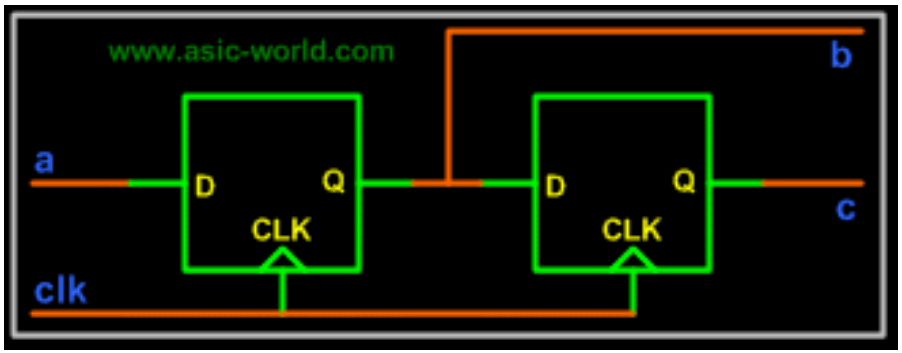
```
module blocking (clk,a,c);
  input wire clk;
  input wire a;
  output reg c;
  reg b;

  always @ (posedge clk )
    begin
     b = a;
     c = b;
    end
endmodule
```

```
module nonblocking (clk,a,c);
  input wire clk;
  input wire a;
  output reg c;
  reg b;

  always @ (posedge clk )
    begin
     b <= a;
     c <= b;
    end
endmodule
```





Feltételes operátor: ?

- A C nyelvhez hasonlóan Verilogban is használhatjuk a feltételes operátort. Formája: `<feltétel> ? <1_kifejezés> : <2_kifejezés>`

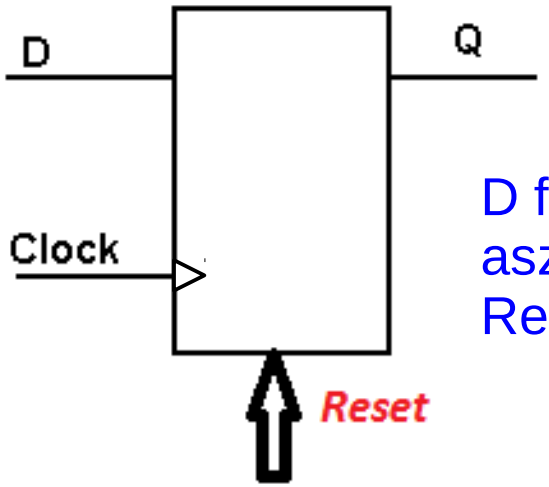
```
1  module conditional_operator();
2
3  wire out;
4  reg enable,data;
5  // Tri state buffer
6  assign out = (enable) ? data : 1'bz;
7
8  initial  begin
9     $display  ("time\t enable data out");
10    $monitor  ("%g\t %b     %b    %b",$time,enable,data,out);
11    enable = 0;
12    data = 0;
13    #1   data = 1;
14    #1   data = 0;
15    #1   enable = 1;
16    #1   data = 1;
17    #1   data = 0;
18    #1   enable = 0;
19    #10   $finish;
20 end
21
22 endmodule
```

Ha a feltétel teljesül, akkor az első kifejezés, ellenkező esetben pedig a második kifejezés lesz a visszatérési érték.

time	enable	data	out
0	0	0	z
1	0	1	z
2	0	0	z
3	1	0	0
4	1	1	1
5	1	0	0
6	0	0	z

If – else utasítás

- Az **if – else** utasítás a C nyelvhez hasonlóan vezérli az egyes ágakban elhelyezett utasítások végrehajtását.
- Több utasítás esetén **begin – end** közé kell foglalni azokat.
- Az **if – else** utasítások egymásba ágyazhatóak. Bizonyos esetekben az **else** ág „hovatartozásának” egyértelműsítése miatt **begin – end** közé kell zárni az összefogni kívánt ágakat.



D flip-flop aszinkron Reset-tel

```
module dff(
    input wire Clock,
    input wire Reset
    input wire D,
    output reg Q
);
always @(posedge Clock or Reset)
    if (Reset)
      Q = 0;
    else
      Q = D;
endmodule
```

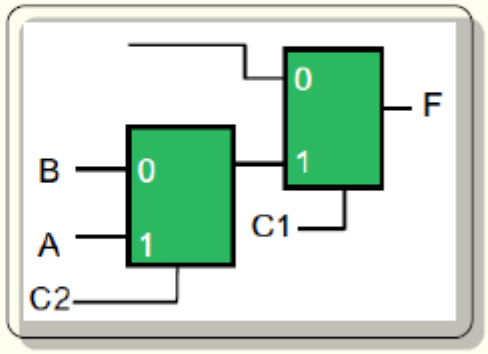
Aszinkron Reset esetén először mindig a Reset jelet kell vizsgálni!

If – else utasítás

- Egymásba ágyazott **if – else** hatása

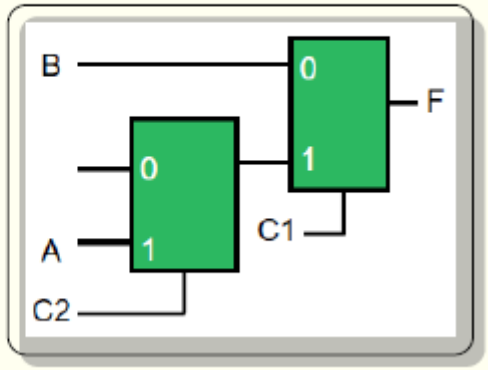
begin – end nélkül

```
// Akár így írjuk:
if (C1)
  if(C2)
    F = A;
  else
    F = B;
// Akár így:
if (C1)
  if(C2)
    F = A;
else
  F = B;
```



begin – end használatával

```
// Ez viszont mást jelent:
if (C1)
begin
  if(C2)
    F = A;
end
else
  F = B;
```



If – else utasítás

- Négybites, újraindítható, le - föl számláló megvalósítása:

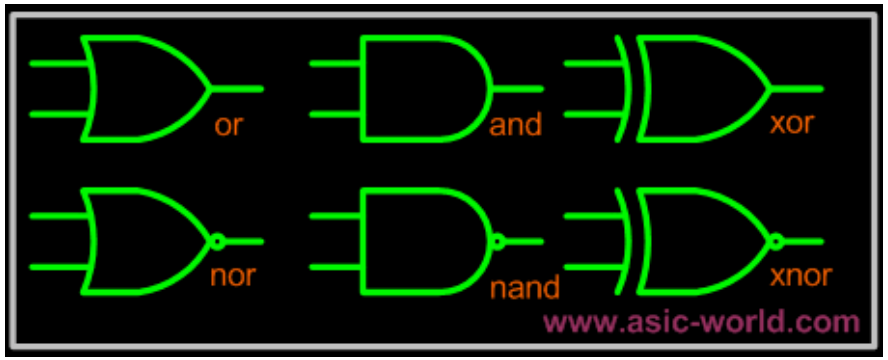
  - Elsőként az aszinkron reset jelet vizsgáljuk
  - Ha a számlálás engedélyezett (enable == 1) és felfelé számlálunk (up_en ==1), akkor növeljük a számlálót.
  - Ha a számlálás engedélyezett (enable == 1) és lefelé számlálunk (down_en ==1), akkor csökkentjük a számlálót.

```
1  module parallel_if();
2
3  reg [3:0] counter;
4  wire clk,reset,enable, up_en, down_en;
5
6  always @ (posedge clk)
7  // If reset is asserted
8  if (reset == 1'b0) begin
9     counter <= 4'b0000;
10 end else begin
11   // If counter is enable and up count is mode
12   if (enable == 1'b1 && up_en == 1'b1) begin
13     counter <= counter + 1'b1;
14   end
15   // If counter is enable and down count is mode
16   if (enable == 1'b1 && down_en == 1'b1) begin
17     counter <= counter - 1'b1;
18   end
19 end
20
21 endmodule
```

- Feltételeztük, hogy **up_en** és **down_en** egyidejűleg nem aktív!

Viselkedés-alapú tervezés

- A **kapuszintű modellezéssel** az előző előadásban már találkoztunk: a logikai kapuknak a Verilog beépített primitív moduljai felelnek meg (buf, not, and, nand, or, nor, xor, xnor stb.).

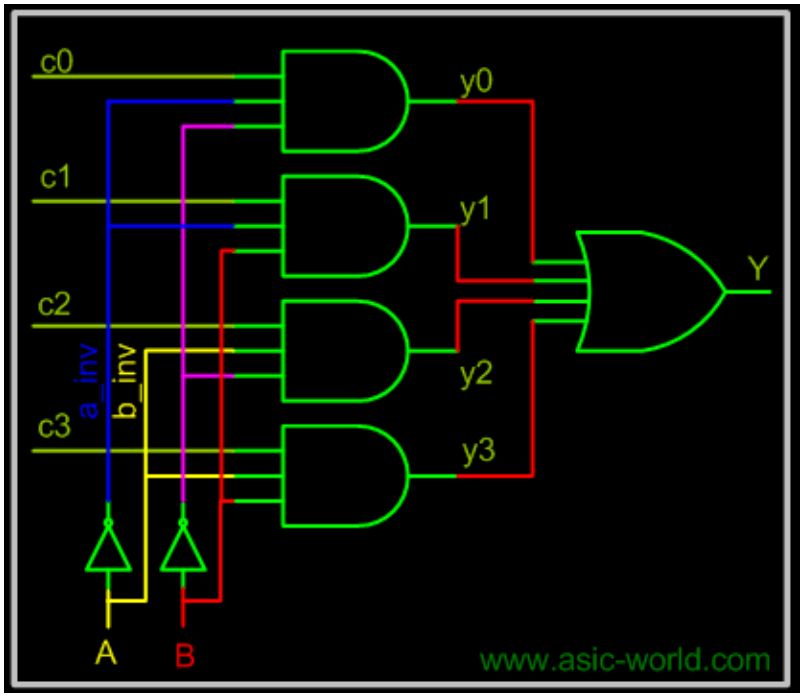


- A tervezést azonban végezhetjük úgy is, hogy a moduljainkat nem elemi kapuáramkörünkből rakjuk össze, hanem az fogalmazzuk meg, hogy milyen viselkedést várunk el. Ez az úgynevezett **viselkedési modellezés** (behavioral modeling), vagy **viselkedés-alapú tervezés.**
- Az előző előadás **counter** példája, vagy a mai előadás legtöbb példája ilyen **viselkedési modellezés** volt.

Kapu szintű vagy viselkedési modellezés

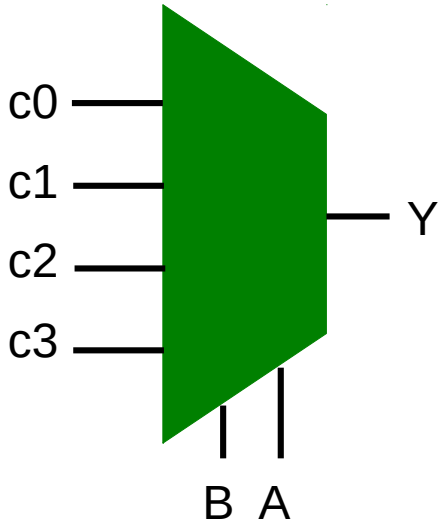
```
module mux_from_gates (c0,c1,c2,c3,A,B,Y);
  input wire c0,c1,c2,c3,A,B;
  output wire Y;
  wire a_inv, b_inv, y0, y1, y2, y3;
// Választójelek invertálása
  not (a_inv, A);
  not (b_inv, B);
// 3-input AND gate
  and (y0,c0,a_inv,b_inv);
  and (y1,c1,a_inv,B);
  and (y2,c2,A,b_inv);
  and (y3,c3,A,B);
  or (Y, y0,y1,y2,y3); // 4-input OR gate
endmodule
```

Kapu szintű modell




```
module MUX4 (c0,c1,c2,c3,A,B,Y);
  input wire c0,c1,c2,c3,A,B;
  output reg Y;
  always @ *  //Minden esemény aktiválja
    case ({B,A})
      0: Y = c0; 1: Y = c1; 2: Y = c2; 3: Y = c3;
    endcase
endmodule
```

Viselkedési modell



Felhasznált irodalom és segédanyagok

- Icarus Verilog Simulator: http://iverilog.icarus.com/
- GtkWave wave viever: http://gtkwave.sourceforge.net/
- Verilog online tutorial: vol.verilog.com/VOL/main.htm
- Verilog tutorial for beginners: www.referencedesigner.com/tutorials/verilog/verilog_01.php
- ASIC world – Verilog tutorial: asic-world.com/verilog/veritut.html
- Végh János: Bevezetés a **Verilog** hardverleíró nyelvbe
- Végh János: Segédeszközök az **Altera DE2** tanulói készlethez
- Végh János: Bevezetés a **Quartus II V13** fejlesztő rendszerbe