BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRÉSTECHNIKA ÉS INFORMÁCIÓS RENDSZEREK TANSZÉK

Bevezetés a Verilog hardverleíró nyelvbe

Fehér Béla, Raikovich Tamás

BME MIT

Hardverleíró nyelvek

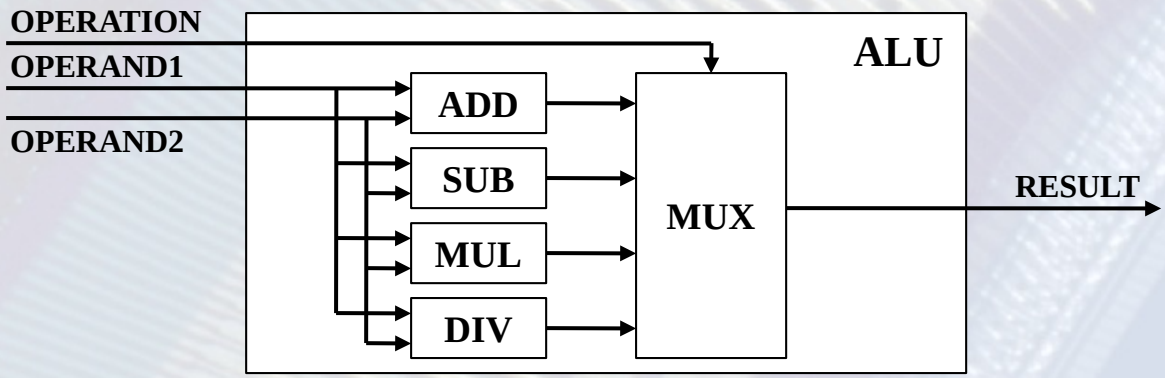
- **A hardverleíró nyelveket (HDL) digitális áramkörök modellezéséhez és szimulálásához fejlesztették ki**
- **A nyelvi elemeknek csak egy része használható a terv megvalósításához**
- **Fontos különbség a standard programnyelvek (C, C++) és a hardverleíró nyelvek között:**
  - Standard programnyelv: ***sorrendi*** végrehajtást ír le
  - HDL: ***párhuzamos*** és ***egyidejű*** viselkedést ír le
- **A két leggyakrabban használt hardverleíró nyelv:**
  - ***Verilog***
  - VHDL

Verilog HDL

- **A Verilog nyelv több tekintetben is hasonlít a C és a C++ programozási nyelvekre, például:**
  - A kis- és nagybetűket megkülönbözteti
  - Egysoros komment: //
  - Blokk komment: /* ……. */
  - Az operátorok nagy része ugyanaz
- **Azonban a Verilog forráskód nem szoftver!**
- **A továbbiakban csak azon nyelvi elemek kerülnek ismertetésre, melyek a hardver terv megvalósításához használhatók fel**
  - A verifikációhoz, szimulációhoz vannak további, csak erre a célra használható nyelvi elemek

Verilog HDL – Modulok

- **A Verilog nyelv hierarchikus, funkcionális egység alapú tervezési megközelítést használ:**
  - A teljes rendszer több kisebb ***modulból*** épül fel
  - Az egyes modulok komplexitását a tervező határozza meg
- **A Verilog modul részei:**
  - A bemeneti és a kimeneti portok leírása, melyeken keresztül a modul a „külvilághoz” kapcsolódik
  - A modul bemenetei és kimenetei között fennálló logikai kapcsolat leírása



Verilog HDL – Modulok
(Modul deklarálása)

- A legfelső (top-level) modul portjai a felhasznált hardver eszköz I/O lábaihoz kapcsolódnak
- A modul deklarálásának szintaxisa:

A modul neve — Port lista

```
module SomeFunction(op1, op2, result);

    input  wire [7:0] op1;
    input  wire [7:0] op2;
    output wire [7:0] result;

    assign result = op1 + op2;

endmodule
```

Opcionális — A portok deklarálása — A funkcionalitás leírása

Verilog HDL – Modulok

(Portok deklarálása)

- **A portok deklarálásának szintaxisa:**
  `<irány> <típus> <signed> <méret> <port_neve>;`
- **Irány:**
  - Bemeneti port: ***input***
  - Kimeneti port: ***output***
  - Kétirányú port: ***inout***
- **Típus: wire az alapértelmezett, ha nincs megadva**
  - ***wire*** (vezeték): a nevében benne van a viselkedése
  - ***reg*** (regiszter): nem mindig lesz belőle valódi regiszter
    - Lehet belőle vezeték, latch, illetve flip-flop
    - Portok esetén a ***reg*** típus csak kimeneti (***output***) porthoz rendelhető
- **Előjeles típus:**
  - A ***signed*** kulcsszó jelzi, ha a jel előjeles értéket reprezentál
  - A jelet ekkor kettes komplemens kódolásúnak kell tekinteni
- **Méret: [j : i]** → a port mérete $|j - i| + 1$ bit
  - A legnagyobb helyiértékű bit a ***j***-edik bit ($j \leq i$ is megengedett)
  - A legkisebb helyiértékű bit az ***i***-edik bit
  - Az alsó és a felső index felcserélése nincs hatással a bitsorrendre

Verilog HDL – Modulok
(Portok deklarálása)

- **Példák:**
  - 1 bites bemeneti port: `input wire sel;`
  - 16 bites kimeneti regiszter: `output reg [15:0] result;`
- **Alternatív modul leírás: a modul portjai a port listában is deklarálhatók**

A modul neve

```
module SomeFunction(
    input  wire [7:0] op1,
    input  wire [7:0] op2,
    output wire [7:0] result
);

assign result = op1 + op2;

endmodule
```

A portok deklarálása a modul port listájában

A funkcionalitás leírása

Verilog HDL – Modulok
(Modul példányosítása)

- A példányosítandó modul:

```
module SomeFunction(input A, input B, output C);
      ⋮
endmodule
```

- Ezt a következőképpen lehet példányosítani, azaz felhasználni egy másik modulban:

```
wire d, e, f;
SomeFunction Func1(.A(f), .B(e), .C(d));
```

Az *f* jel az *A* portra csatlakozik

A példányosítandó modul

A példány neve

Jelek hozzárendelése a portokhoz

- Egy modulból több példány is létrehozható, de a példányok neve eltérő kell, hogy legyen

Verilog HDL – Modulok
(Modul példányosítása – Példa)

- **Feladat: készítsünk 8 bites bináris számlálót két 4 bites bináris számláló kaszkádosításával**
- **A 4 bites számláló Verilog moduljának fejléce**
  - A ***tc*** jelbe az ***en*** jel nincs bekapuzva

```
module cnt_4bit(
    input  wire         clk, //Órajel bemenet
    input  wire         rst, //Reset bemenet
    input  wire         en,  //Engedélyező jel
    output reg  [3:0] q,     //A számláló kimenete
    output wire         tc   //Végállapot jelzés
);
```

- **A 8 bites számlálót megvalósító modulban két 4 bites számlálót kell példányosítani**

Verilog HDL – Modulok

(Modul példányosítása – Példa)

```
module cnt_8bit(
   input  wire       clk,  //Órajel bemenet
   input  wire       rst,  //Reset bemenet
   input  wire       en,   //Engedélyező jel
   output wire [7:0] q,    //A számláló kimenete
   output wire       tc    //Végállapot jelzés
);

wire tc1, tc2;             //A számlálók végállapot jelzései
wire en1 = en;             //Az első számláló engedélyező jele
wire en2 = en & tc1;       //A második számláló engedélyező jele

cnt_4bit cnt1(             //Az első 4 bites számláló példányosítása
   .clk(clk), .rst(rst), .en(en1), .q(q[3:0]), .tc(tc1)
);

cnt_4bit cnt2(             //A második 4 bites számláló példányosítása
   .clk(clk), .rst(rst), .en(en2), .q(q[7:4]), .tc(tc2)
);

assign tc = tc1 & tc2;     //A 8 bites számláló végállapot jelzése

endmodule
```

Verilog HDL – Modulok
(Belső jelek deklarálása)

- **A belső jelek deklarálásának szintaxisa:**
  `<típus> <signed> <méret> <jel_neve>;`
- **Hasonló a portok deklarálásához, de nincs irány és a típus megadása nem hagyható el**
- **Példák:**
  - 1 bites vezeték: `wire counter_enable;`
  - 16 bites regiszter: `reg [15:0] counter;`

Verilog HDL – Konstansok
(A jelek lehetséges értékei)

- **A Verilog nyelvben a jelek négyféle értéket vehetnek fel**
  - ***0***: logikai alacsony szint
  - ***1***: logikai magas szint
  - ***z***: nagyimpedanciás meghajtás
  - ***x***: ismeretlen, nem meghatározható, don't care
- **Modern hardver rendszertervezés esetén *z* értéket (nagyimpedanciás állapotot) csak az I/O interfészek megvalósításánál használunk**
- **Hardver megvalósítása esetén *x* érték (don't care) csak a leírás egyszerűsítésére használható (*casex* utasítás)**

Verilog HDL – Konstansok
(Numerikus konstansok)

- **A numerikus konstansok megadásának szintaxisa:**
  <–><*bitek_száma*> ’<s><*számrendszer*><*numerikus_konstans*>
- **Az előjeles konstansok kettes komplemens kódolásúak**
- **Negatív előjel: a konstans kettes komplemense képezhető vele**
- **Bitek száma: a konstans mérete bitekben**
  - Az alapértelmezett méret 32 bit, ha nincs megadva
- **Előjeles konstans: az s karakter jelzi**
  - Ha nincs megadva, akkor a konstans előjel nélküli
  - Az előjel bitet a megadott bitszám szerint kell értelmezni
  - Előjeles konstans esetén az előjel kiterjesztés automatikus
- **Számrendszer: decimális az alapértelmezett, ha nincs megadva**
  - Bináris: **b**, oktális: **o**, decimális: **d**, hexadecimális: **h**
- **A ’_’ karakter használható a számjegyek szeparálásához**
  - Jobban olvasható, áttekinthetőbb kódot eredményez

Verilog HDL – Konstansok

(Numerikus konstansok – Példák)

Példák konstansok megadására:

- **8'b0000_0100: 8 bites bináris konstans, értéke 4**
- **6'h1f: 6 bites hexadecimális konstans, értéke 31**
  - Binárisan: **6'b01_1111**
- **128: 32 bites decimális konstans**
  - Binárisan: **32'b00000000_00000000_00000000_10000000**
- **– 4'sd15: 4 bites decimális konstans, értéke 1**
  - **4'sd15** binárisan **4'sb1111**, azaz **–1** (előjeles!)
  - A negatív előjel ennek veszi a kettes komplemensét → **4'sb0001**

Példák az előjel kiterjesztésre:

```
wire [5:0] a =  4'd9;    //a=6'b00_1001 (9)
wire [5:0] b =  4'sd5;   //b=6'b00_0101 (5)
wire [5:0] c =  4'sd9;   //c=6'b11_1001 (-7)
wire [5:0] d = -4'd6;    //d=6'b11_1010 (-6)
```

Verilog HDL – Konstansok
(String konstansok)

- A string (szöveg) konstansok megadásának szintaxisa:
  "a_string_karakterei"
- A stringben lévő karakterek a 8 bites ASCII kódjaikra képződnek le, ezért a string konstans bitszáma a karakterek számának nyolcszorosa
- A legfelső nyolc bit veszi fel a string első karakteréhez tartozó értéket
- Példa:

```
wire [23:0] str = "HDL";
//str[7:0]   = 8'b0100_1100 ('L')
//str[15:8]  = 8'b0100_0100 ('D')
//str[23:16] = 8'b0100_1000 ('H')
```

Verilog HDL – Paraméterek
(Paraméterek definiálása)

- A paraméterek definiálásának szintaxisa:
  `parameter <név> = <konstans>;`
  `localparam <név> = <konstans>;`
- A paraméter neve konstansként használható abban a modulban, amelyben a paraméter definiálva lett
- A modul példányosításakor a ***normál paraméterek*** értéke megváltoztatható
- A ***lokális paraméterek*** értéke nem változtatható meg
- Példa:

```
parameter WIDTH = 8;

wire [WIDTH-1:0] data;
```

Verilog HDL – Paraméterek
(Paraméterek definiálása)

- Alternatív modul leírás: a normál paraméterek a modul fejlécében is definiálhatók

A modul neve

```
module SomeFunction #(
    parameter WIDTH = 8,
    parameter OTHER_PARAM = 2
) (
    input  wire [WIDTH-1:0] op1,
    input  wire [WIDTH-1:0] op2,
    output wire [WIDTH-1:0] result
);

assign result = op1 + op2;

endmodule
```

A paraméterek definiálása a modul fejlécében

A portok deklarálása a port listában

A funkcionalitás leírása

Verilog HDL – Paraméterek
(Paraméterrel rendelkező modul példányosítása)

- **A paraméterekkel rendelkező modul:**

```
module SomeFunction(input A, input B, output C);
paramerer P1 = 8;
parameter P2 = 16;
      ⋮
endmodule
```

- **A fenti modul példányosítása:**

```
wire d, e, f;
SomeFunction #(
    .P1(3), .P2(20)
) Func2 (
    .A(f), .B(e), .C(d)
);
```

Új értékek hozzárendelése a paraméterekhez (opcionális)

Jelek hozzárendelése a portokhoz

Verilog HDL - Operátorok

- A Verilog operátoroknak 1, 2 vagy 3 operandusuk lehet
- A kifejezések jobboldalán vegyesen szerepelhetnek ***wire*** típusú, ***reg*** típusú és konstans operandusok
- Ha egy művelet azonos méretű operandusokat igényel, akkor a kisebb méretű operandus általában nullákkal lesz kiterjesztve a nagyobb operandus méretére
- A kifejezések kiértékelése a normál precedencia szabályok szerint történik (a precedencia zárójelekkel befolyásolható)

Operátor	Precedencia
Unáris +, -, !, ~	1. (legnagyobb)
*, /, %	2.
Bináris +, -	3.
<<, >>, <<<, >>>	4.
<, <=, >, >=	5.
==, !=, ===, !==	6.

Operátor	Precedencia
&, ~&	7.
^, ~^	8.
|, ~|	9.
&&	10.
||	11.
? : (feltételes op.)	12. (legkisebb)

Verilog HDL – Operátorok
(Konkatenálás operátor)

- **Konkatenálás operátor: { }**
  - Több operandus összefűzése
    `{5'b10110, 2'b10, 1'b0, 1'b1} = 9'b1_0110_1001`
  - Ugyanazon operandus többszöri összefűzése
    `{4{3'b101}} = 12'b101_101_101_101`
- **Fontos felhasználási esetek:**
  - Előjel kiterjesztés: az előjel bitet a felső bitekbe kell másolni

```
wire [3:0] s_4bit; //4 bites előjeles
wire [7:0] s_8bit; //8 bites előjeles
assign s_8bit = {{4{s_4bit[3]}}, s_4bit};
```

  - Vektor maszkolása egyetlen bittel: az 1 bites kisebb operandus nullákkal lenne kiterjesztve a nagyobb operandus méretére

```
wire [3:0] data;
wire [3:0] mdata;
wire       enable;
assign mdata = data & enable;      //Rossz!!!
assign mdata = data & {4{enable}}; //Helyes
```

Verilog HDL – Operátorok
(Feltételes és indexelő operátorok)

- **Feltételes operátor: ? :**
  `<feltételes_kifejezés> ? <kifejezés1> : <kifejezés2>`
  - Az egyetlen 3 operandusú operátor
  - Először a ***feltételes_kifejezés*** értékelődik ki
    - Ha az eredmény nem 0: a ***kifejezés1*** értékelődik ki
    - Ha az eredmény 0: a ***kifejezés2*** értékelődik ki
- **Vektor egy részének kiválasztása:**
  `vektor_nev[i], vektor_nev[j:i]`
  - [i] kiválasztja a vektor ***i***-edik bitjét
  - [j:i] kiválasztja a vektor **j**-edik és **i**-edik bitje közötti részét (a határokat is beleértve)

Verilog HDL – Operátorok

(Bitenkénti és logikai operátorok)

- **Bitenkénti operátorok: ~ (NOT), & (AND), | (OR), ^ (XOR)**
  - Operandusok száma: NOT: 1 / AND, OR, XOR: 2
  - Vektor operandusok esetén a művelet bitenként hajtódik végre
  - **Ha az operandusok mérete eltérő, akkor előjeltől függetlenül a kisebb operandus nullákkal lesz kiterjesztve a nagyobb operandus méretére**
    - Ha nem ezt szeretnénk, akkor használjuk a konkatenálás operátort
  - Példák:
    - `4'b0100 | 4'b1001 = 4'b1101`
    - `~8'b0110_1100 = 8'b1001_0011`
- **Logikai operátorok: ! (NOT), && (AND), || (OR)**
  - Operandusok száma: NOT: 1 / AND, OR: 2
  - Az eredmény mindig egybites: 0 vagy 1
  - Példák:
    - `4'b0000 || 4'b0111 = 0 || 1 = 1`
    - `4'b0000 && 4'b0111 = 0 && 1 = 0`
    - `!4'b0000 = !0 = 1`

Verilog HDL – Operátorok
(Bit redukciós operátorok)

- **Bit redukciós operátorok:**
  & (AND), ~& (NAND), | (OR), ~| (NOR), ^ (XOR), ~^ (XNOR)
  - Operandusok száma: 1
  - Egyetlen vektoron hajtanak végre bitenkénti műveletet
  - Az eredmény mindig egybites: 0 vagy 1
  - Példák:
    - `&4'b0101 = 0 & 1 & 0 & 1 = 0`
    - `|4'b0101 = 0 | 1 | 0 | 1 = 1`
- **Fontos felhasználási esetek:**
  - Nulla érték tesztelése: a vektor bitjeinek NOR kapcsolata

```
wire [11:0] data;
wire all_zeros = ~|data;
```

  - 2^N-1 érték tesztelése: a vektor bitjeinek AND kapcsolata

```
wire all_ones = &data;
```

  - Számláló végállapotának jelzése:

```
wire tc = (dir) ? (&cnt) : (~|cnt);
```

Verilog HDL – Operátorok

(Aritmetikai operátorok)

- **Aritmetikai operátorok: + (összeadás), - (kivonás), * (szorzás), / (osztás), % (modulus)**
  - Operandusok száma: 2
  - Ha az FPGA eszköz nem tartalmaz szorzót, akkor a szorzás operátor csak akkor szintetizálható, ha az egyik operandus kettő hatvány értékű konstans
  - Az osztás és a modulus operátorok csak akkor szintetizálhatók, ha a jobboldali operandus kettő hatvány értékű konstans
  - **Összeadásnál, kivonásnál és szorzásnál a kisebb méretű előjeles operandus esetén előjel kiterjesztés történik a nagyobb operandus méretére**
  - Átvitel bitek használhatósága az összeadásnál és kivonásnál Xilinx FPGA eszközök esetén:

	Összeadó	Kivonó	Összeadó/kivonó
Bemenő átvitel bit (C_{in})	van	van	nincs
Kimenő átvitel bit (C_{out})	van	nincs	nincs

Verilog HDL – Operátorok
(Shift operátorok)

- **Logikai shift operátorok: << (balra), >> (jobbra)**
  - Operandusok száma: 2
  - Példák:
    - **8'b0011_1100 >> 2 = 8'b0000_1111**
    - **8'b0011_1100 << 2 = 8'b1111_0000**
- **Aritmetikai shift operátorok: <<< (balra), >>> (jobbra)**
  - Operandusok száma: 2
  - A balra történő aritmetikai shiftelés és előjel nélküli operandus esetén a jobbra történő aritmetikai shiftelés megegyezik az adott irányú logikai shifteléssel
  - Előjeles operandus esetén a jobbra történő aritmetikai shiftelés megtartja az előjel bitet
  - Példák:
    - **8'b1001_1100 >>> 1 = 8'b0100_1110**
    - **8'sb1001_1100 >>> 1 = 8'b1100_1110**

Verilog HDL – Operátorok
(Relációs operátorok)

- **Relációs operátorok:**
  == (egyenlő), != (nem egyenlő), < (kisebb), > (nagyobb),
  <= (kisebb vagy egyenlő), >= (nagyobb vagy egyenlő)
  - Operandusok száma: 2
  - Az eredmény mindig egybites: 0 vagy 1
  - Az egyenlő és a nem egyenlő reláció kapus logikára, a kisebb és a nagyobb reláció jellemzően aritmetikai funkcióra képződik le
  - **A kisebb méretű előjeles operandus esetén előjel kiterjesztés történik a nagyobb operandus méretére**
  - Példák:
    - **(4'b1011 < 4'b0111) = 0**
    - **(4'b1011 != 4'b0111) = 1**

Verilog HDL - Értékadás

- **Logikai kapcsolat megadása *wire* típusú jelek esetén:**
  `assign <wire_jel> = <kifejezés>;`
  - A bal oldali ***wire_jel*** által reprezentált értéket a jobboldali ***kifejezés*** minden pillanatban meghatározza (kombinációs logika)
  - Példa:

```
wire [15:0] a, b, c;
assign c = a & b;
```

  - A deklarálásnál is megadható a logikai kapcsolat:

```
wire [15:0] a, b;
wire [15:0] c = a & b;
```

- **Érték hozzárendelése *reg* típusú jelekhez az *always* blokkokban lehetséges a blokkoló (=) vagy a nem blokkoló (<=) értékadás operátor segítségével**

Verilog HDL – Értékadás
(Always blokk)

- **Always blokk:**
  `always @(érzékenységi lista)`
  `hozzárendelések`
  - Az érzékenységi lista határozza meg az eseményeket, melyek hatására a hozzárendelések kiértékelődnek:
    - **`always @(a, b, c)`**: a hozzárendelések akkor értékelődnek ki, ha az ***a***, ***b*** vagy ***c*** jel értéke megváltozik
    - **`always @(*)`**: a hozzárendelések akkor értékelődnek ki, ha az ***always*** blokk egyik bemenetének értéke megváltozik
    - **`always @(posedge clk)`**: a hozzárendelések a ***clk*** jel felfutó élének hatására értékelődnek ki
    - **`always @(negedge clk, posedge rst)`**: a hozzárendelések a ***clk*** jel lefutó élének hatására vagy az ***rst*** jel 1-be állításának hatására értékelődnek ki

Verilog HDL – Értékadás
(Always blokk)

- ***Always* blokk:**
  - Egy adott ***reg*** típusú jelhez az érték hozzárendelése csak egy ***always*** blokkban megengedett szintézis esetén
  - Az ***if…else***, a ***case*** és a ***for*** utasítások az ***always*** blokkokon belül használhatók
  - Ha az ***always*** blokkban több utasítás van, akkor azokat a ***begin*** és az ***end*** kulcsszavak közé kell csoportosítani
  - Példa:

```
wire a;
reg  b, c, d;

always @(a, b)
begin
   c <= a & b;
   d <= a | b;
end
```

```
//Tiltott 2. értékadás
always @(*)
begin
   c <= b ^ d;
end
```

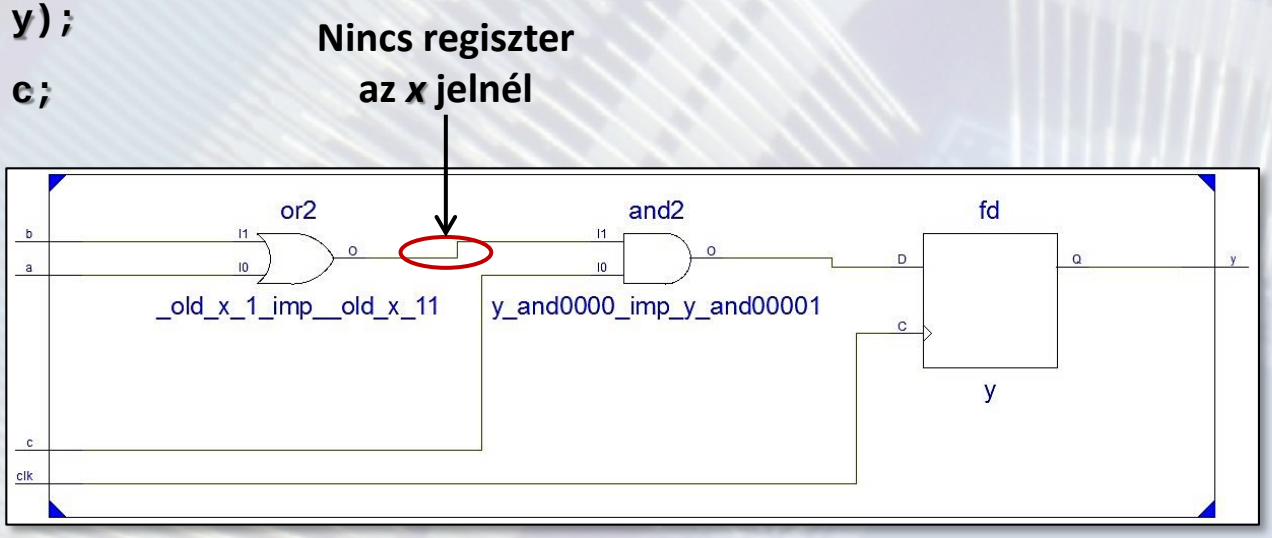
Verilog HDL – Értékadás

(Always blokk – Blokkoló értékadás)

Blokkoló értékadás

- A blokkoló értékadás operátor (=) mindaddig blokkolja a következő utasítás végrehajtását, amíg ki nem értékelődik a hozzárendelés
- Ezért, ha egy blokkoló értékadás eredményét egy későbbi értékadó utasítás felhasználja ugyanazon ***always*** blokkon belül, akkor az adott blokkoló értékadáshoz nem lesz tároló beépítve a sorrendi hálózatba (lásd a lenti példában az ***x*** jelet)

```
module M(clk, a, b, c, y);

input  wire clk, a, b, c;
output reg  y;

reg x;

always @(posedge clk)
begin
   x = a | b;
   y = x & c;
end

endmodule
```



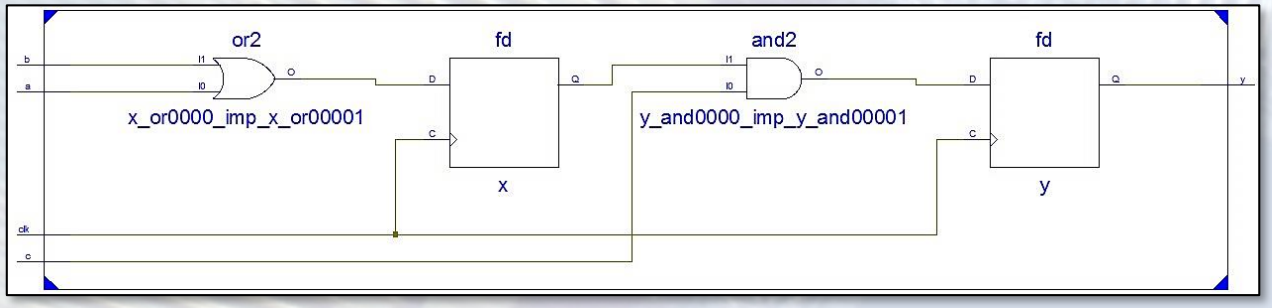
Verilog HDL – Értékadás

(Always blokk – Nem blokkoló értékadás)

Nem blokkoló értékadás

- A nem blokkoló értékadás operátorok (<=) a következő blokkoló értékadás utasításig egymással párhuzamosan értékelődnek ki
- Ezért minden egyes nem blokkoló értékadás esetén tároló kerül beépítésre a sorrendi hálózatba
- Ahol lehet, használjunk nem blokkoló értékadást, mivel ez közelebb áll a hardveres szemlélethez

```
module M(clk, a, b, c, y);

input  wire clk, a, b, c;
output reg  y;

reg x;

always @(posedge clk)
begin
   x <= a | b;
   y <= x & c;
end

endmodule
```



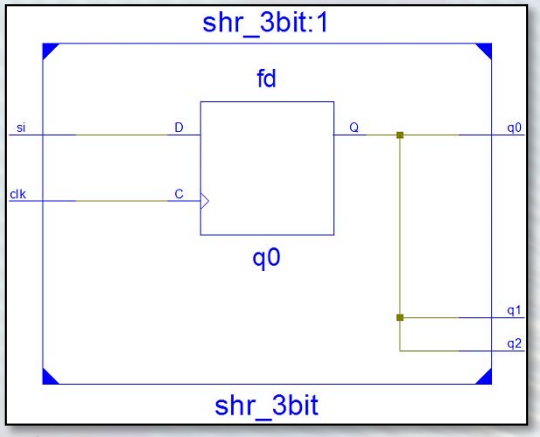
Verilog HDL – Értékadás
(Példa)

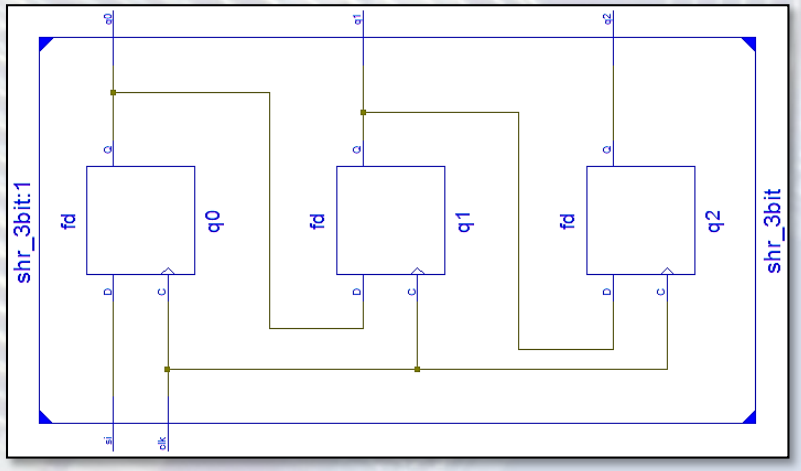
Példa: 3 bites shiftregiszter

- Figyeljük meg a két megvalósítás közötti különbséget
- Csak a nem blokkoló értékadás használata esetén kapunk helyes eredményt

```
module shr_3bit(clk, si, q0, q1, q2);

input  wire clk, si;
output reg  q0, q1, q2;

always @(posedge clk)
begin
   q0 = si;
   q1 = q0;
   q2 = q1;
end

endmodule
```

```
module shr_3bit(clk, si, q0, q1, q2);

input  wire clk, si;
output reg  q0, q1, q2;

always @(posedge clk)
begin
   q0 <= si;
   q1 <= q0;
   q2 <= q1;
end

endmodule
```





Verilog HDL – IF utasítás

- **Az *if* utasítás szintaxisa:**
  `if (kifejezés) utasítás1; [else utasítás2;]`
  - Először a megadott ***kifejezés*** kerül kiértékelésre:
    - Ha értéke nem 0: az ***utasítás1*** hajtódik végre
    - Ha értéke 0: az ***utasítás2*** hajtódik végre
  - Az ***else*** ág opcionális, elhagyható
  - Több utasítás esetén azokat a ***begin*** és az ***end*** kulcsszavak közé kell csoportosítani
  - Az egymásba ágyazott ***if*** utasítások hierarchikus, sorrendi kiértékelést jelentenek
    - Ez a tipikus megvalósítása a funkcionális egységek vezérlő jeleinek

Verilog HDL – CASE utasítás

- A ***case*** utasítás szintaxisa:

```
case (kifejezés)
    alternatíva1: utasítás1;
    alternatíva2: utasítás2;
        ⋮                ⋮
    default     : default_utasítás;
endcase
```

  - A ***kifejezés*** értéke összehasonlításra kerül az ***alternatívákkal*** a megadásuk sorrendjében (a sorrendjük prioritást jelenthet!)
  - A legelső, a kifejezés értékével egyező alternatívához tartozó utasítás kerül végrehajtásra
  - Ha nincs egyező alternatíva, akkor a ***default*** kulcsszó után lévő ***default_utasítás*** kerül végrehajtásra (opcionális)
  - Több utasítás esetén azokat a ***begin*** és az ***end*** kulcsszavak közé kell csoportosítani
  - A ***casex*** utasítás esetén az alternatívák tartalmazhatnak ***x*** (don't care) értéket is, ez néha egyszerűbb leírást tesz lehetővé

Verilog HDL – FOR utasítás

- **A *for* utasítás szintaxisa:**
```
for ([inicializálás]; [feltétel]; [művelet])
   utasítás;
```
- **A for ciklus működése a következő:**
  1. Az ***inicializáló rész*** beállítja a ciklusváltozó kezdeti értékét
  2. Kiértékelődik a ***feltétel***, ha hamis, akkor kilépünk a ciklusból
  3. Végrehajtódik a megadott ***utasítás***
  4. Végrehajtódik a megadott ***művelet***, majd ugrás a 2. pontra
- **Több utasítás esetén azokat a *begin* és az *end* kulcsszavak közé kell csoportosítani, a *begin* kulcsszót pedig egyedi címkével kell ellátni (*begin*: *címke*)**
- **Hardver megvalósítása esetén a *for* ciklus az *always* blokkban csak statikus módon, a leírás egyszerűsítéséhez használható (például indexelés vagy érték vizsgálat)**
  - A programozási nyelvekben használt általános ***for*** ciklus a hardverben vezérlővel és adatstruktúrával helyettesíthető
- **A ciklusváltozót *integer* típusúnak kell deklarálni**

Verilog HDL – FOR utasítás

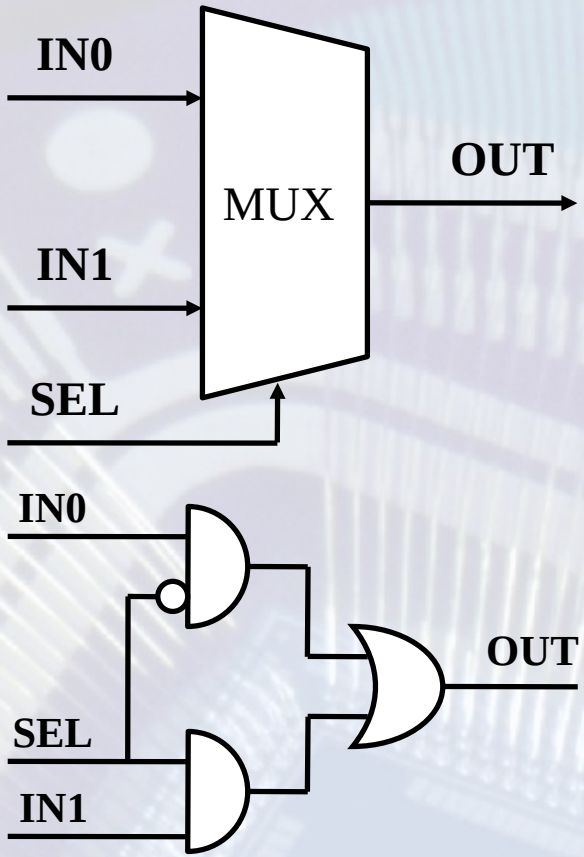
- **Első példa: bitsorrend felcserélése**
  - A ***for*** ciklust indexelésre használjuk
  - Ciklus nélkül 32 darab értékadással lehetne megvalósítani
- **Második példa: 4 x 8 bites regisztertömb**
  - A ***for*** ciklust indexelésre és egyenlőség vizsgálatra használjuk
  - Ciklus nélkül 4 darab feltételes értékadással lehetne megvalósítani

```
module BitReverse(din, dout);

input  wire [31:0] din;
output reg  [31:0] dout;

integer i;   //Ciklusváltozó

always @(*)
   for (i=0; i<32; i=i+1)
   begin: reverse_loop
      dout[i] <= din[31-i];
   end

endmodule
```

```
module RegFile(clk,addr,we,din,r0,r1,r2,r3);

input  wire       clk, we;
input  wire [1:0] addr;
input  wire [7:0] din;
output wire [7:0] r0, r1, r2, r3;

reg [7:0] r [3:0];   //4 x 8 bites reg. tömb
integer i;           //Ciklusváltozó

always @(posedge clk)
   for (i=0; i<4; i=i+1)
      if (we && (addr == i))
         r[i] <= din;

assign {r3,r2,r1,r0} = {r[3],r[2],r[1],r[0]};

endmodule
```

Verilog HDL - Példák

- **Példa: 1 bites 2/1-es multiplexer**
  - A bemenetei közül kiválaszt egyet és ennek értékét adja ki a kimenetére
  - Portok:
    - Adat bemenetek: IN0, IN1
    - Bemenet kiválasztó jel: SEL
    - Adat kimenet: OUT
  - Művelet:
    - Ha SEL=0: IN0 kapcsolódik a kimenetre
    - Ha SEL=1: IN1 kapcsolódik a kimenetre
  - A multiplexer egy kombinációs hálózat




```
wire in0, in1, sel, out;
assign out = (in0 & ~sel) | (in1 & sel);
```

Verilog HDL - Példák

- **Példa: 8 bites 2/1-es multiplexer**
  - 1. megoldás: bitenkénti operátorok használata
  - ***Ebben az esetben a bitenkénti operátorok használata nem célszerű, mert a leírt funkció nem állapítható meg könnyen a forráskód alapján!***

```
module Mux_2to1_8bit(
   input  wire [7:0] in0,
   input  wire [7:0] in1,
   input  wire       sel,
   output wire [7:0] out
);

assign out = (in0 & {8{~sel}}) | (in1 & {8{sel}});

endmodule
```

Verilog HDL - Példák

- **Példa: 8 bites 2/1-es multiplexer**
  - 2. megoldás: feltételes operátor használata

```
module Mux_2to1_8bit(
    input  wire [7:0] in0,
    input  wire [7:0] in1,
    input  wire       sel,
    output wire [7:0] out
);

assign out = (sel) ? in1 : in0;

endmodule
```

Verilog HDL - Példák

- **Példa: 8 bites 2/1-es multiplexer**
  - 3. megoldás: IF utasítás használata

```
module Mux_2to1_8bit(
  input  wire [7:0] in0,
  input  wire [7:0] in1,
  input  wire       sel,
  output reg  [7:0] out
);

always @(*)          //vagy always @(in0, in1, sel)
   if (sel == 0)
      out <= in0;
   else
      out <= in1;

endmodule
```

Verilog HDL - Példák

- **Példa: 8 bites 2/1-es multiplexer**
  - 4. megoldás: CASE utasítás használata

```
module Mux_2to1_8bit(
  input  wire [7:0] in0,
  input  wire [7:0] in1,
  input  wire       sel,
  output reg  [7:0] out
);

always @(*)          //vagy always @(in0, in1, sel)
   case (sel)
      1'b0: out <= in0;
      1'b1: out <= in1;
   endcase

endmodule
```

Verilog HDL - Példák

- **Példa: 8 bites 4/1-es multiplexer**

```
module Mux_4to1_8bit(in0, in1, in2, in3, sel, out);

input  wire [7:0] in0, in1, in2, in3;
input  wire [1:0] sel;
output reg  [7:0] out;

always @(*)  //vagy always @(in0, in1, in2, in3, sel)
   case (sel)
      2'b00: out <= in0;
      2'b01: out <= in1;
      2'b10: out <= in2;
      2'b11: out <= in3;
   endcase

endmodule
```

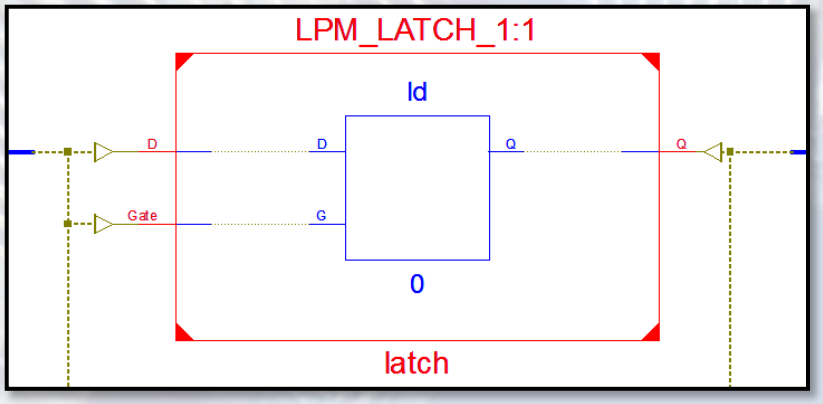
Kombinációs hálózat leírása

- **A *wire* típusú jelekkel csak kombinációs hálózat valósítható meg**
- **A *reg* típusú jelek megvalósíthatnak kombinációs és sorrendi hálózatot is**
- ***Kombinációs hálózat* leírása *reg* típusú jelekkel**
  - A hozzárendeléseket akkor kell kiértékelni, ha valamelyik bemenet értéke megváltozik:
    - Az ***always*** blokk érzékenységi listájának tartalmaznia kell az összes bemeneti jelet vagy a * karaktert
    - A ***posedge*** vagy a ***negedge*** kulcsszó nem szerepelhet

Kombinációs hálózat leírása

- ***Kombinációs hálózat*** **leírása** ***reg*** **típusú jelekkel**
  - Az ***always*** blokk csak teljesen specifikált ***if*** és ***case*** utasításokat tartalmazhat
  - Ha az ***if*** és ***case*** utasítások nem teljesen specifikáltak, akkor ***latch*** (aszinkron flip-flop) kerül az áramkörbe
    - A ***reg*** típusú jel állapotát a ***latch*** megőrzi, ha nem történik hozzárendelés az always blokkban

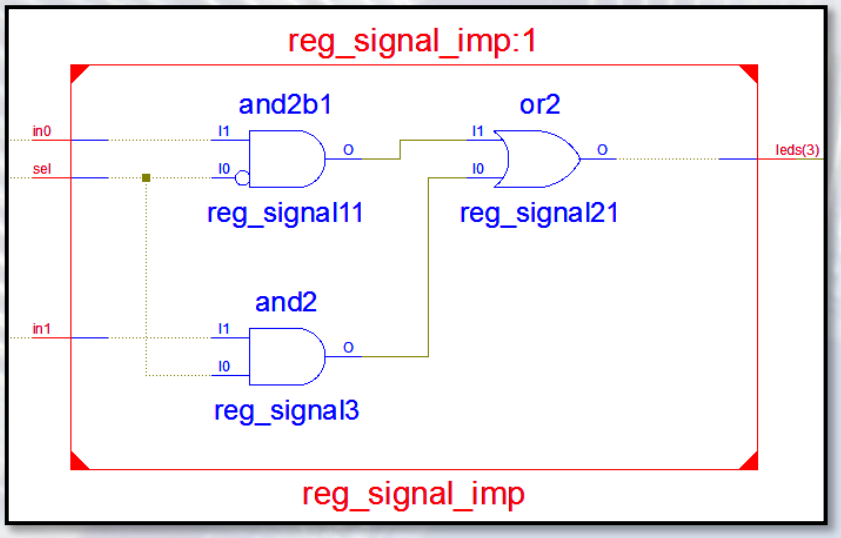
```
reg reg_signal;

always @(*)
   if (sel)
      reg_signal <= in0;
```



Kombinációs hálózat leírása

- **A latch-ek nem kívánatosak, nem kombinációs logikát, hanem aszinkron sorrendi logikát valósítanak meg**
- **Ha az *if* és a *case* utasítások teljesen specifikáltak (*if*: minden *else* ág megtalálható, *case*: minden lehetséges alternatíva fel van sorolva vagy van *default* kulcsszó):**
  - Az eredmény kombinációs hálózat lesz (példa: MUX)

```
reg reg_signal;

always @(*)
   if (sel)
      reg_signal <= in1;
   else
      reg_signal <= in0;
```



Sorrendi hálózat leírása

- **Sorrendi logika csak *reg* típusú jelekkel írható le**
  - Aszinkron: latch (**kerülendő!**)
  - Szinkron: flip-flop, regiszter
- **A regiszterek az órajel felfutó vagy lefutó élének hatására változtatják meg az állapotukat**
  - Az ***always*** blokk érzékenységi listájának tartalmaznia kell az órajelet, amely előtt a ***posedge*** (felfutó él) vagy a ***negedge*** (lefutó él) kulcsszó áll
    - A két kulcsszó egyszerre nem szerepelhet az órajel előtt
  - Az órajel az ***always*** blokkban lévő belső kifejezésekben explicit módon, mint jel nem szerepelhet
- **Az *if* és a *case* utasítás lehet nem teljesen specifikált**
  - A flip-flop órajel engedélyező bemenete használható az állapotváltozás elkerülésére, ha nem történik értékadás az ***always*** blokkban

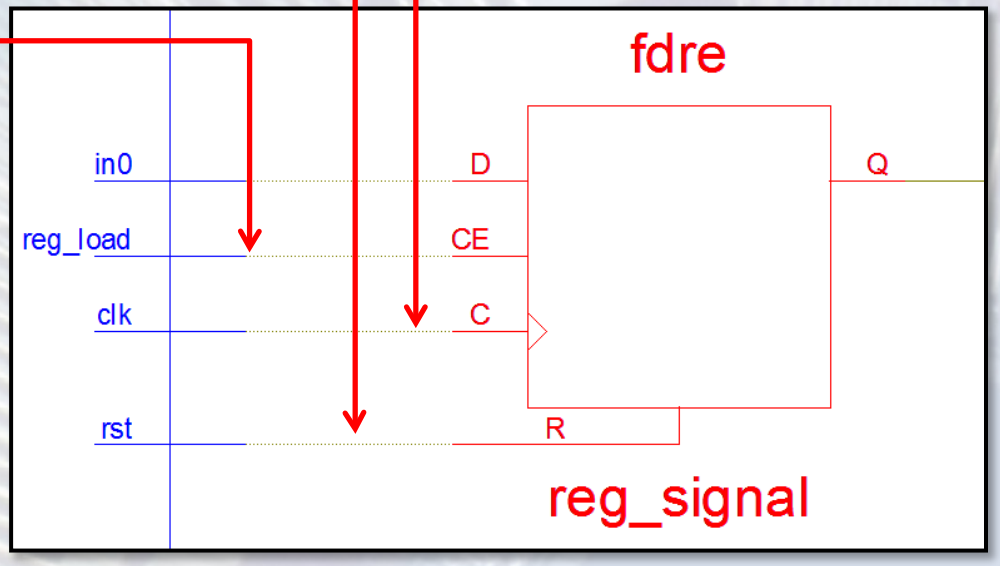
Sorrendi hálózat leírása

```
reg  reg_signal;

always @(posedge clk)
  if (rst)
    reg_signal <= 1'b0;
  else
    if (reg_load)
      reg_signal <= in0;
```

órajel

reset jel

órajel engedélyező jel



Adatút komponensek

- **Összeadó:**
  - Használjuk a + (összeadás) operátort
  - Az eredmény lehet 1 bittel szélesebb: az MSb a kimeneti átvitel bit

```
wire [15:0] a, b, sum1, sum2;
wire        cin, cout;
assign sum1 = a + b + cin;
assign {cout, sum2} = a + b + cin;
```

- **Kivonó:**
  - Használjuk a – (kivonás) operátort
  - Nincs átvitel kimenet: használjunk helyette 1 bittel szélesebb kivonót

```
wire [15:0] a, b, diff;
wire        cin;
assign diff = a – b - cin;
```

- **Összeadó/kivonó:**
  - Nincs sem átvitel bemenet, sem átvitel kimenet

```
wire [15:0] a, b;
wire [15:0] result
wire        sel;
assign result = (sel) ? (a – b) : (a + b);
```

Adatút komponensek

- **Shifter:**
  - Használjuk a { } (konkatenálás) operátort a shift operátorok helyett
  - A konstansok méretét meg kell adni

```
wire [7:0] din;
wire [7:0] lshift = {din[6:0], 1'b0}; //bal shift
wire [7:0] rshift = {1'b0, din[7:1]}; //jobb shift
```

- **Komparátor:**
  - Használjuk a relációs operátorokat

```
wire [15:0] a, b;
wire        a_lt_b = (a < b);   //Kisebb komparátor
wire        a_eq_b = (a == b);  //Egyenlőség komp.
wire        a_gt_b = (a > b);   //Nagyobb komparátor
```

- **Szorzó:**
  - Használjuk a * (szorzás) operátort
  - A szorzat mérete az operandusok méretének összege
  - Csak akkor szintetizálható, ha az FPGA tartalmaz szorzót

```
wire [15:0] a, b;
wire [31:0] prod = a * b;
```

Adatút komponensek

- **Shiftregiszter (példa):**
  - Szinkron reset és töltés
  - Kétirányú: balra és jobbra is tud léptetni

```
reg  [7:0] shr;
wire [7:0] din;
wire       rst, load, dir, serin;

always @(posedge clk)
  if (rst)
    shr <= 8'd0;                          //reset
  else
    if (load)
      shr <= din;                         //tölt
    else
      if (dir)
        shr <= {serin, shr[7:1]};         //jobbra léptet
      else
        shr <= {shr[6:0], serin};         //balra léptet
```

Adatút komponensek

- **Számláló (példa):**
  - Szinkron reset és töltés
  - Kétirányú: felfele és lefele is tud számlálni

```
reg  [8:0] cnt;
wire [8:0] din;
wire       rst, load, dir;
wire       tc = (dir) ? (cnt==9'd0) : (cnt==9'd511);

always @(posedge clk)
  if (rst)
    cnt <= 9'd0;                    //reset
  else
    if (load)
      cnt <= din;                   //tölt
    else
      if (dir)
        cnt <= cnt - 9'd1;          //lefele számlál
      else
        cnt <= cnt + 9'd1;          //felfele számlál
```

A vezérlő jelek prioritása

A vezérlő bemenetek értéke abban a sorrendben kerül vizsgálatra, ahogyan azok az *always* blokkon belül fel vannak sorolva

```
always @(posedge clk)
  if (rst)
    cnt <= 9'd0;
  else
    if (load)
      cnt <= data_in;
    else
      if (en)
        cnt <= cnt + 9'd1;
```

rst	load	en	Művelet
1	x	x	Reset
0	1	x	Tölt
0	0	1	Számlál
0	0	0	Tart

```
always @(posedge clk)
  if (rst)
    cnt <= 9'd0;
  else
    if (en)
      if (load)
        cnt <= data_in;
      else
        cnt <= cnt + 9'd1;
```

rst	en	load	Művelet
1	x	x	Reset
0	1	1	Tölt
0	1	0	Számlál
0	0	x	Tart

```
always @(posedge clk)
  if (en)
    if (clr)
      cnt <= 9'd0;
    else
      if (load)
        cnt <= data_in;
      else
        cnt <= cnt + 9'd1;
```

en	clr	load	Művelet
0	x	x	Tart
1	1	x	Törlés
1	0	1	Tölt
1	0	0	Számlál

Szinkron és aszinkron vezérlő jelek

- ***Szinkron*** **vezérlő jelek:**
  - Hatásuk csak az órajel esemény bekövetkezése után érvényesül
  - Az érzékenységi lista nem tartalmazza a szinkron vezérlő jeleket

```
//Aktív magas szinkron reset
always @(posedge clk)
  if (rst)
    some_reg <= 1'b0;
  else
    some_reg <= data_in;
```

```
//Aktív alacsony szinkron reset
always @(posedge clk)
  if (rst == 0)
    some_reg <= 1'b0;
  else
    some_reg <= data_in
```

- ***Aszinkron*** **vezérlő jelek:**
  - Hatásuk azonnal érvényesül
  - Az érzékenységi listának tartalmaznia kell az aszinkron vezérlő jeleket, melyek előtt a ***posedge*** vagy a ***negedge*** kulcsszó áll

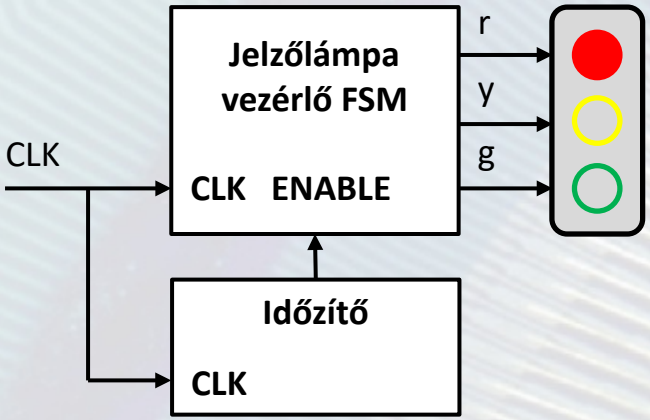
```
//Aktív magas aszinkron reset
always @(posedge clk, posedge rst)
  if (rst)
    some_reg <= 1'b0;
  else
    some_reg <= data_in;
```

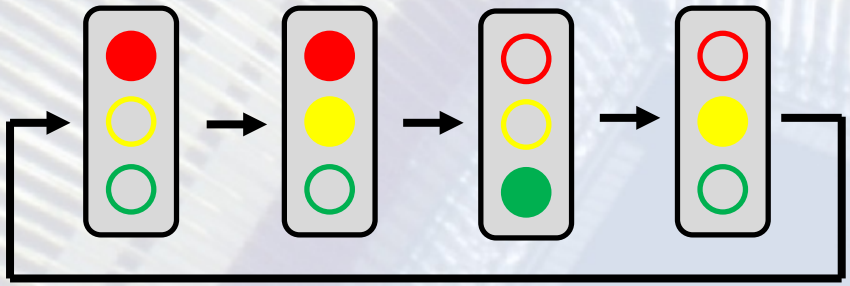
```
//Aktív alacsony aszinkron reset
always @(posedge clk, negedge rst)
  if (rst == 0)
    some_reg <= 1'b0;
  else
    some_reg <= data_in
```

Állapotgépek (FSM)

- **Lokális paraméterek használhatók az állapotok definiálásához**
- **Egy regiszter szükséges az aktuális állapot tárolására**
- **A *case* utasítás használható az aktuális állapot kiválasztására**
  - Minden alternatíva esetén az ***if*** vagy a ***case*** utasítással vizsgálható a bemenetek értéke és végrehajtható a megfelelő állapotátmenet

- **Példa: közúti jelzőlámpa vezérlő**
  - 4 állapot: piros, piros-sárga, zöld, sárga
  - Egy külső időzítő generálja az egy órajelpulzusnyi engedélyező jelet





Állapotgépek (FSM)

Első megvalósítás:

- Az állapotregiszter és a következő állapot logika azonos blokkban van
- Egyedi állapotkódolás (a szintézer optimalizálhatja)

```
localparam STATE_R  = 2'd0;
localparam STATE_RY = 2'd1;
localparam STATE_G  = 2'd2;
localparam STATE_Y  = 2'd3;

reg [1:0] state;

//Az állapotregiszter és a köv. áll. logika
always @(posedge clk)
begin
  if (rst)
    state <= STATE_R;
  else
    case (state)
      STATE_R : if (enable)
                  state <= STATE_RY;
                else
                  state <= STATE_R;
      STATE_RY: if (enable)
                  state <= STATE_G;
                else
                  state <= STATE_RY;
      STATE_G : if (enable)
                  state <= STATE_Y;
                else
                  state <= STATE_G;
      STATE_Y : if (enable)
                  state <= STATE_R;
                else
                  state <= STATE_Y;
    endcase
end

//A kimenetek meghajtása
assign r = (state==STATE_R) | (state==STATE_RY);
assign y = (state==STATE_Y) | (state==STATE_RY);
assign g = (state==STATE_G);
```

Állapotgépek (FSM)

Második megvalósítás:

- **Az állapotregiszter és a következő állapot logika külön blokkban van**
- **Egyedi állapotkódolás (a szintézer optimalizálhatja)**

```
localparam STATE_R  = 2'd0;
localparam STATE_RY = 2'd1;
localparam STATE_G  = 2'd2;
localparam STATE_Y  = 2'd3;

reg [1:0] state;
reg [1:0] next_state;

//Állapotregiszter (sorrendi hálózat)
always @(posedge clk)
  if (rst)
    state <= STATE_R;
  else
    if (enable)
      state <= next_state;
```

```
//Köv. állapot logika (kombinációs hálózat)
always @(*)
  case (state)
    STATE_R : next_state <= STATE_RY;
    STATE_RY: next_state <= STATE_G;
    STATE_G : next_state <= STATE_Y;
    STATE_Y : next_state <= STATE_R;
  endcase

//A kimenetek meghajtása
assign r = (state == STATE_R) |
           (state == STATE_RY);
assign y = (state == STATE_Y) |
           (state == STATE_RY);
assign g = (state == STATE_G);
```

Állapotgépek (FSM)

Harmadik megvalósítás:

- **Az állapotregiszter és a következő állapot logika külön blokkban van**
- **Kimeneti kódolás: az *(* fsm_encoding = "user" *)* Xilinx specifikus Verilog direktíva tiltja az állapotkódolás optimalizálását az adott regiszterre**

```
localparam STATE_R  = 3'b100;
localparam STATE_RY = 3'b110;
localparam STATE_G  = 3'b001;
localparam STATE_Y  = 3'b010;

(* fsm_encoding = "user" *)
reg [2:0] state;
reg [2:0] next_state;

//Állapotregiszter (sorrendi hálózat)
always @(posedge clk)
  if (rst)
    state <= STATE_R;
  else
    if (enable)
      state <= next_state;
```

```
//Köv. állapot logika (kombinációs hálózat)
always @(*)
  case (state)
    STATE_R : next_state <= STATE_RY;
    STATE_RY: next_state <= STATE_G;
    STATE_G : next_state <= STATE_Y;
    STATE_Y : next_state <= STATE_R;
  endcase

//A kimenetek meghajtása
assign r = state[2];
assign y = state[1];
assign g = state[0];
```

Memóriák (RAM, ROM)

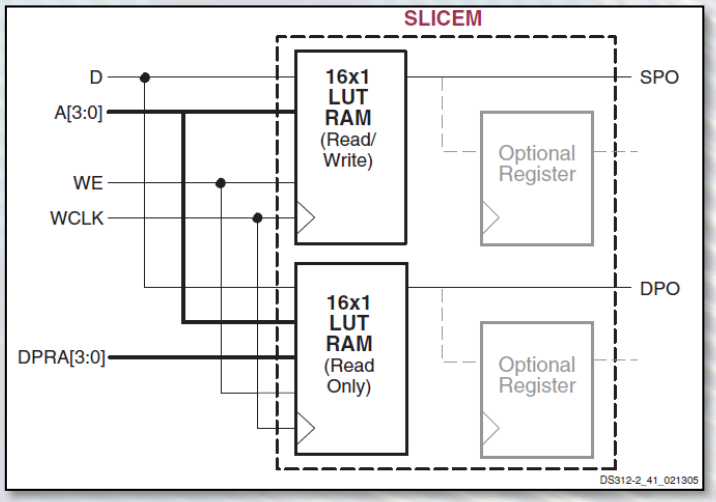
- **Memóriák leírása Verilog nyelven:**
  - A memória tekinthető egy egydimenziós tömbnek
    - WIDTH: egy szóban lévő bitek száma
    - WORDS: a memóriában lévő szavak száma, melynek **kettő hatványnak** kell lennie
  - A memória adatot tárol (állapottal rendelkezik), ezért regiszter típusúnak kell deklarálni

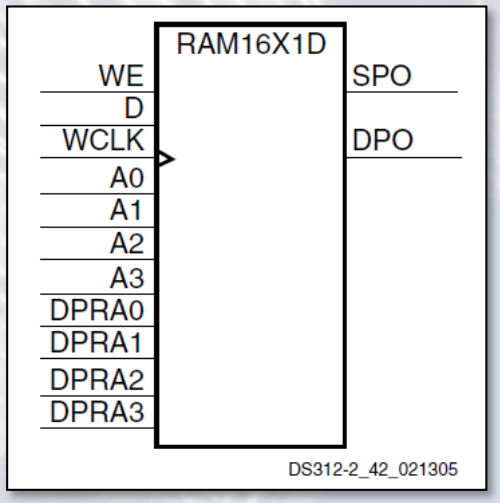
```
reg [WIDTH-1:0] mem [WORDS-1:0];
```

- **A Xilinx FPGA-k kétféle típusú memóriát tartalmaznak**
  - Elosztott RAM (distributed RAM)
  - Blokk RAM (block RAM)
- **Mindkét memória külön adat bemenettel és kimenettel rendelkezik**

Memóriák (RAM, ROM)

- **Elosztott RAM (Xilinx Spartan-3E FPGA):**
  - Kismennyiségű adat hatékony tárolásához (pl. regisztertömb)
  - 1 írási porttal és 1 vagy 2 olvasási porttal rendelkezik
    - A cím megosztott az írási és az első olvasási port között (***A***)
    - A második olvasási port külön cím bemenettel rendelkezik (***DPRA***)
  - Az írási művelet szinkron
    - Az órajel felfutó (vagy lefutó) élére történik, ha engedélyezve van (***WE=1***)
  - Az olvasási művelet aszinkron
    - A megcímzett adat „azonnal” megjelenik az adatkimeneten (***SPO, DPO***)





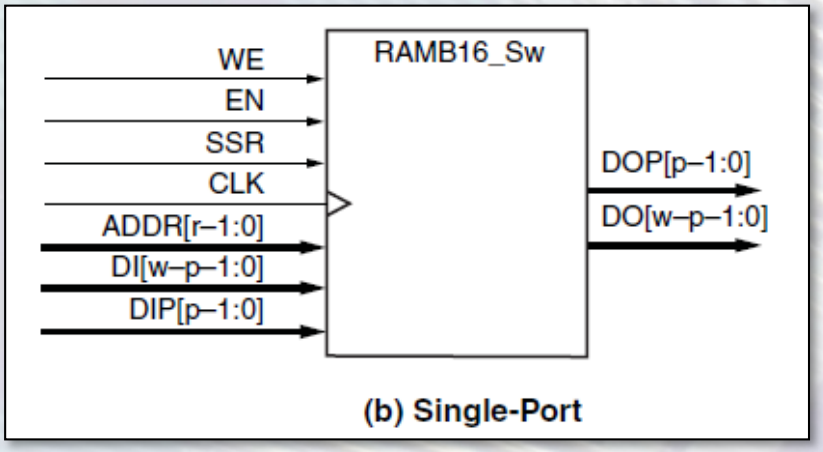
Memóriák (RAM, ROM)

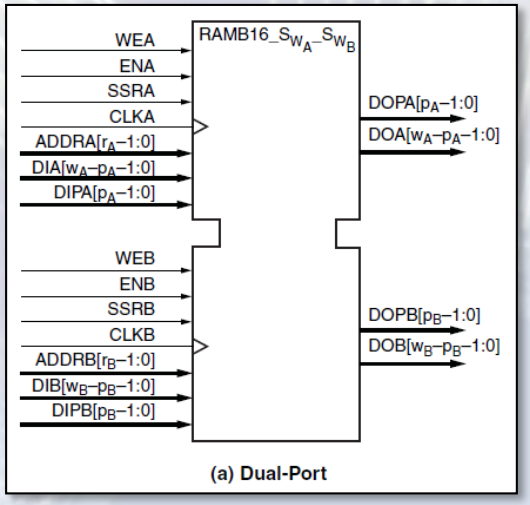
- **Az elosztott RAM Verilog leírása:**
  - Példa: 32 x 4 bites RAM 1 írási és 2 olvasási porttal
  - A ***(* ram_style = "distributed" *)*** Xilinx specifikus Verilog direktíva utasítja a szintézert, hogy elosztott RAM-ot használjon a memória megvalósításához

```
(* ram_style = "distributed" *)
reg  [3:0] mem [31:0];
wire [4:0] addr_a;       //Cím az írási és az 1. olvasási porthoz
wire [4:0] addr_b;       //Cím a 2. olvasási porthoz
wire [3:0] din;          //A beírandó adat
wire       write_en;     //Írás engedélyező jel

//Írási port (szinkron)
always @(posedge clk)
  if (write_en)
    mem[addr_a] <= din;

//Olvasási portok (aszinkron)
wire [3:0] dout_a = mem[addr_a];
wire [3:0] dout_b = mem[addr_b];
```

Memóriák (RAM, ROM)

- **Blokk RAM (Xilinx Spartan-3E FPGA):**
  - 18 kbit kapacitás RAM blokkonként, különféle konfigurációk
    - 16k x 1 bit, 8k x 2 bit, 4k x 4 bit, 2k x 9 bit, 1k x 18 bit és 512 x 36 bit
  - Két független írási/olvasási porttal rendelkezik (x = A,B)
    - Órajel (***CLKx***), adatkimenet reset (***SSRx***), engedélyezés (***ENx***), írás eng. (***WEx***),
    - Cím (***ADDRx***), adatbemenet (***DIx, DIPx***), adatkimenet (***DOx, DOPx***)
  - A parancsot a memória az órajel felfutó (v. lefutó) élére mintavételezi
    - Az írás az órajel felfutó (vagy lefutó) élére történik, ha ***ENx***=1 és ***WEx***=1
    - Az olvasás az órajel felfutó (vagy lefutó) élére történik, ha ***ENx***=1
    - ***Így a szinkron olvasás 1 órajelnyi késleltetést jelent!***





Memóriák (RAM, ROM)

- **A blokk RAM Verilog leírása:**
  - Példa: 2k x 16 bites RAM 1 írási és 1 olvasási porttal
  - A ***(* ram_style = "block" *)*** Xilinx specifikus Verilog direktíva utasítja a szintézert a blokk RAM használatára
  - Ha nincs megadva engedélyező jel, akkor ***ENx*** konstans 1 értékű lesz

```
(* ram_style = "block" *)
reg  [15:0] mem [2047:0];
wire [10:0] wr_addr;      //Írási cím
wire [10:0] rd_addr;      //Olvasási cím
wire [15:0] din;          //A beírandó adat
reg  [15:0] dout;         //Adatkimenet (szinkron olvasás -> reg)
wire        write_en;     //Írás engedélyező jel

//Írási port (szinkron)
always @(posedge clk)
  if (write_en)
    mem[wr_addr] <= din;

//Olvasási port (szinkron), lehetne az írási
//portot megvalósító always blokkban is
always @(posedge clk)
  dout <= mem[rd_addr];
```

Memóriák (RAM, ROM)

- **A memória tartalmának inicializálása külső adatfájlból**
  - Az adatfájl soronként 1 bináris vagy hexadecimális stringet tartalmaz
  - A fájlban lévő sorok számának azonosnak kell lennie a memóriában lévő szavak számával
  - Az inicializáláshoz használjuk a ***$readmemb*** (bin. adatfájl) vagy a ***$readmemh*** (hex. adatfájl) Verilog függvényeket ***initial*** blokkon belül

```
$readmemb("adatfájl", ram_név, kezdőcím, végcím);
$readmemh("adatfájl", ram_név, kezdőcím, végcím);
```

- ***Initial*** **blokk:**

```
initial
    hozzárendelések
```

  - Nem használható hardver komponensek megvalósításához
  - Az ***initial*** blokkon belüli hozzárendelések a szintézis vagy a szimuláció kezdetén értékelődnek ki
  - Az ***if…else***, a ***case*** és a ***for*** utasítások használhatók az ***initial*** blokkban
  - Több utasítás esetén a ***begin*** és az ***end*** kulcsszavakat kell használni

Memóriák (RAM, ROM)

- **Példa: a 2k x 16 bites RAM tartalmának inicializálása**

```
(* ram_style = "block" *)
reg  [15:0] mem [2047:0];
wire [10:0] wr_addr;      //Írási cím
wire [10:0] rd_addr;      //Olvasási cím
wire [15:0] din;          //A beírandó adat
reg  [15:0] dout;         //Adatkimenet (szinkron olvasás -> reg)
wire        write_en;     //Írás engedélyező jel

//A RAM tartalmának inicializálása
initial
   $readmemh("mem_data_hex.txt", mem, 0, 2047);

//Írási és olvasási portok (szinkron)
always @(posedge clk)
begin
  if (write_en)
    mem[wr_addr] <= din;

  dout <= mem[rd_addr];
end
```

A fájl tartalma:

0x000:	01a5
0x001:	d2f8
0x002:	1342
0x003:	6a18
0x004:	4209
0x005:	ffff
0x006:	89ab
0x007:	5566
⋮	⋮
0x7fe:	99aa
0x7ff:	abcd

Memóriák (RAM, ROM)

- Kisméretű ROM-ok leírhatók a *case* utasítás segítségével:

```
wire [2:0] rom_addr;  //8 x 8 bites ROM
reg  [7:0] rom_dout;

always @(*)
   case (rom_addr)
      3'd0: rom_dout <= 8'b1010_1010;
      3'd1: rom_dout <= 8'b1111_1000;
      3'd2: rom_dout <= 8'b0010_0000;
      3'd3: rom_dout <= 8'b1110_0011;
      3'd4: rom_dout <= 8'b0000_0000;
      3'd5: rom_dout <= 8'b0010_1110;
      3'd6: rom_dout <= 8'b1011_1011;
      3'd7: rom_dout <= 8'b1111_1011;
   endcase
```

- Írási port nélkül az elosztott RAM és a blokk RAM használható ROM-ként is
  - A memória tartalmát inicializálni kell!

Memóriák (RAM, ROM)

- **Az elosztott ROM Verilog leírása:**
  - Példa: 32 x 4 bites ROM
  - A ***(* rom_style = "distributed" *)*** Xilinx specifikus Verilog direktíva utasítja a szintézert, hogy elosztott ROM-ot használjon a memória megvalósításához

```
(* rom_style = "distributed" *)
reg  [3:0] mem [31:0];
wire [4:0] rd_addr;     //Olvasási cím
wire [3:0] dout;        //Adatkimenet

//A ROM tartalmának inicializálása (kötelező!)
initial
  $readmemh("rom_data_hex.txt", mem, 0, 31);

//Olvasási port (aszinkron)
assign dout = mem[rd_addr];
```

Memóriák (RAM, ROM)

- **A blokk ROM Verilog leírása:**
  - Példa: 2k x 16 bites ROM 2 olvasási porttal
  - A ***(* rom_style = "block" *)*** Xilinx specifikus Verilog direktíva utasítja a szintézert, hogy blokk ROM-ot használjon a memória megvalósításához

```
(* rom_style = "block" *)
reg  [15:0] mem [2047:0];
wire [10:0] rd_addr1;     //Cím az 1. olvasási porthoz
wire [10:0] rd_addr2;     //Cím a 2. olvasási porthoz
reg  [15:0] dout1;        //Adatkimenet (szinkron olvasás -> reg)
reg  [15:0] dout2;        //Adatkimenet (szinkron olvasás -> reg)

//A ROM tartalmának inicializálása (kötelező!)
initial
  $readmemh("rom_data_hex.txt", mem, 0, 2047);

//Olvasási portok (szinkron)
always @(posedge clk)
begin
  dout1 <= mem[rd_addr1];
  dout2 <= mem[rd_addr2];
end
```

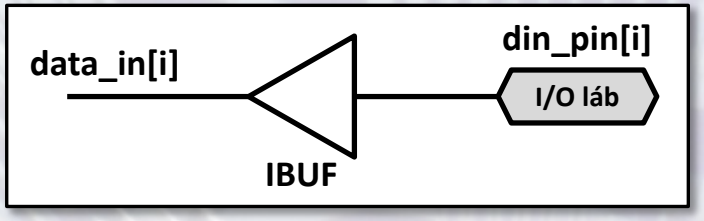
Nagyimpedanciás jelek

- Modern hardver rendszertervezés esetén ***z*** értéket (nagyimpedanciás állapotot) csak az I/O interfészek megvalósításánál használunk
- A mai modern FPGA eszközök az I/O blokkon kívül nem tartalmaznak belső háromállapotú meghajtókat, mivel azok nem megfelelő vezérlése rövidzárlatot okozhat
- A Xilinx XST szintézer a Verilog leírásban lévő belső háromállapotú meghajtókat logikával helyettesíti oly módon, hogy a ***z*** értéket logikai magas szintűnek (1) veszi (mintha felhúzó ellenállás lenne kötve a jelre)

I/O interfész megvalósítása

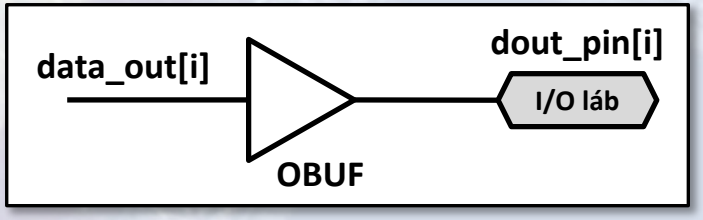
- Csak bemenetként használt I/O láb:

```
input wire [7:0] din_pin;    //Bemeneti port

wire [7:0] data_in;          //Bejövő adat

assign data_in = din_pin;
```



- Csak kimenetként használt I/O láb:

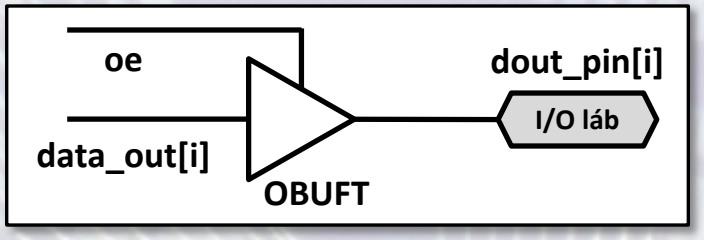
```
output wire [7:0] dout_pin; //Kimeneti port

wire [7:0] data_out;        //Kimenő adat

assign dout_pin = data_out;
```



I/O interfész megvalósítása

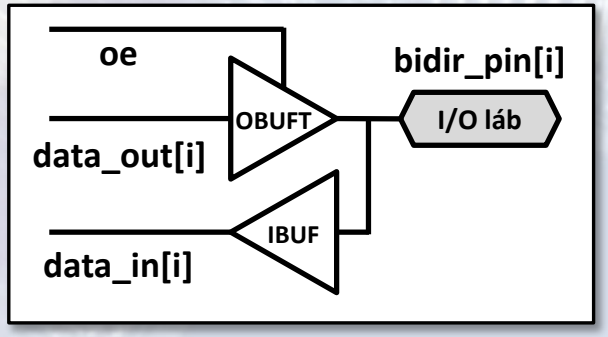
- **Háromállapotú kimenetként használt I/O láb:**

```
output wire [7:0] dout_pin; //Kimeneti port

wire [7:0] data_out;        //Kimenő adat
wire       oe;              //Közös engedélyező jel

assign dout_pin = (oe) ? data_out : 8'bzzzz_zzzz;
```



- **Kétirányú I/O láb:**

```
inout wire [7:0] bidir_pin; //Kétirányú port

wire [7:0] data_out;        //Kimenő adat
wire [7:0] data_in;         //Bejövő adat
wire       oe;              //Közös eng. jel

assign bidir_pin = (oe) ? data_out : 8'hzz;
assign data_in   = bidir_pin;
```



I/O interfész megvalósítása

- A GPIO perifériák esetén minden bithez külön kimeneti meghajtó engedélyező jel is tartozik, ekkor célszerű a ***generate*** blokk használata
- ***Generate*** blokk:

```
generate
   utasítások;
endgenerate
```

  - Kódrészlet paraméterektől függő feltételes példányosításához használható
  - Az ***if…else***, a ***case*** és a ***for*** utasítások használhatók a ***generate*** blokkban
  - A ***for*** utasítás ciklusváltozóját ***genvar*** típusúnak kell deklarálni
  - Minden feltételesen példányosítandó kódrészletet ***begin*** és ***end*** közé kell írni, a ***begin*** kulcsszavakat pedig egyedi címkével kell ellátni

- Példa:

```
output wire [31:0] dout_pin; //32 bites kimeneti port

wire   [31:0] data_out;      //Kimenő adat
wire   [31:0] oe;            //Egyedi engedélyező jelek minden bithez
genvar        i;

generate                     //Ebből a generate blokkból 32 db assign utasítás
   for (i=0; i<32; i=i+1)    //keletkezik (minden kimeneti bithez egy-egy)
   begin: dout_loop
      assign dout_pin[i] = (oe[i]) ? data_out[i] : 1'bz;
   end
endgenerate
```