

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

Digitális rendszerek tervezése

FPGA áramkörökkel

Verilog RTL kódolás

Fehér Béla

Szántó Péter, Lazányi János, Raikovich Tamás

BME MIT

FPGA laboratórium



FPGA-k

FPGA: Field Programmable Gate Array

- programozható logikai áramkör

Jelentősebb gyártók:

- Xilinx, Altera, Actel, Quicklogic, Lattice

Jellemzők

- Komplexitás
 - 50000 – 8000000 kapu
 - 100 – 1000 I/O láb
 - 100 – 500 MHz működés (terv függő)

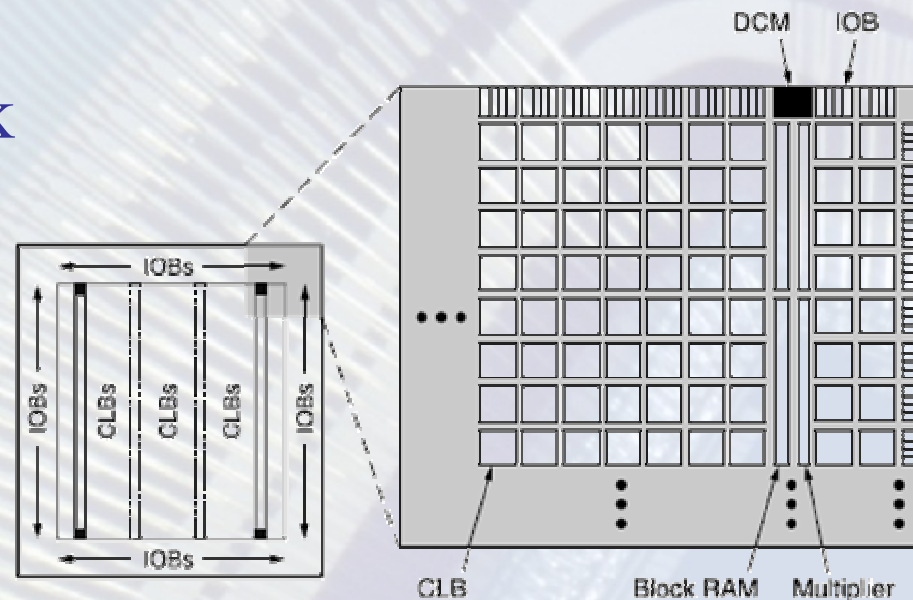
Xilinx FPGA-k

Több család

- Spartan: hatékony, optimalizált struktúra
- Virtex: speciális funkciók, gyorsabb, komplexebb, gazdagabb funkcionalitás

Felépítés:

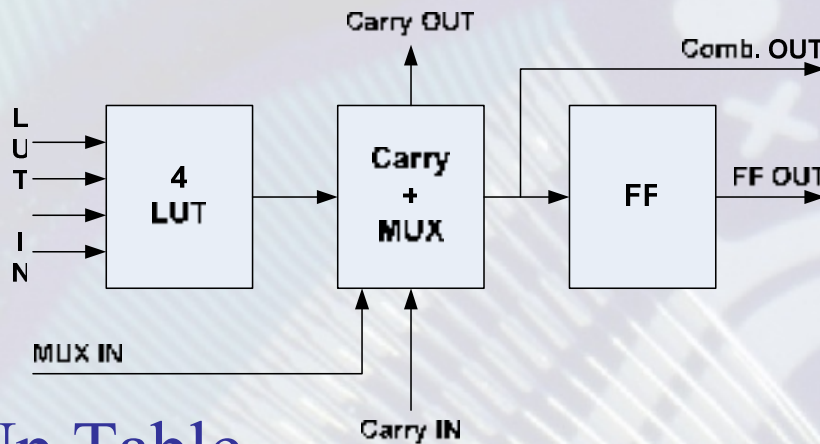
- CLB: logikai blokk
- IOB: I/O blokk
- BlokkRAM
- Szorzó
- Huzalozás



Xilinx FPGA: Alap logikai elem

Logikai elem (Logic Cell):

- 1 LUT4 + 1 FF + kiegészítő logika

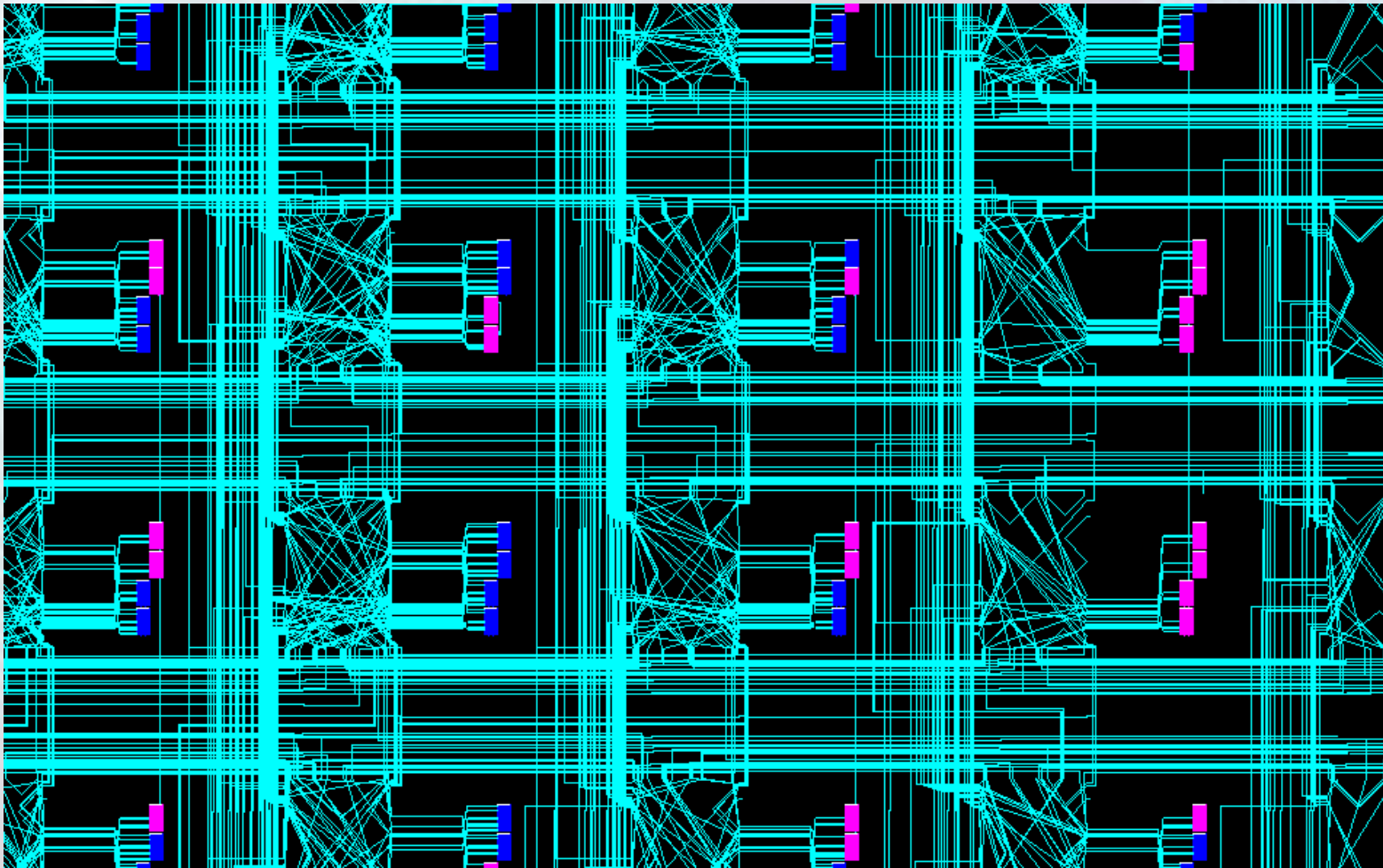


– LUT: Look-Up Table

- 16x1 bites memória (4 bemenet esetén)
- Cím: a logikai függvény bemeneti változói
- Tartalom: igazságtábla
- Bármilyen négy bemenetű, egy kimenetű logikai függvény megvalósítható

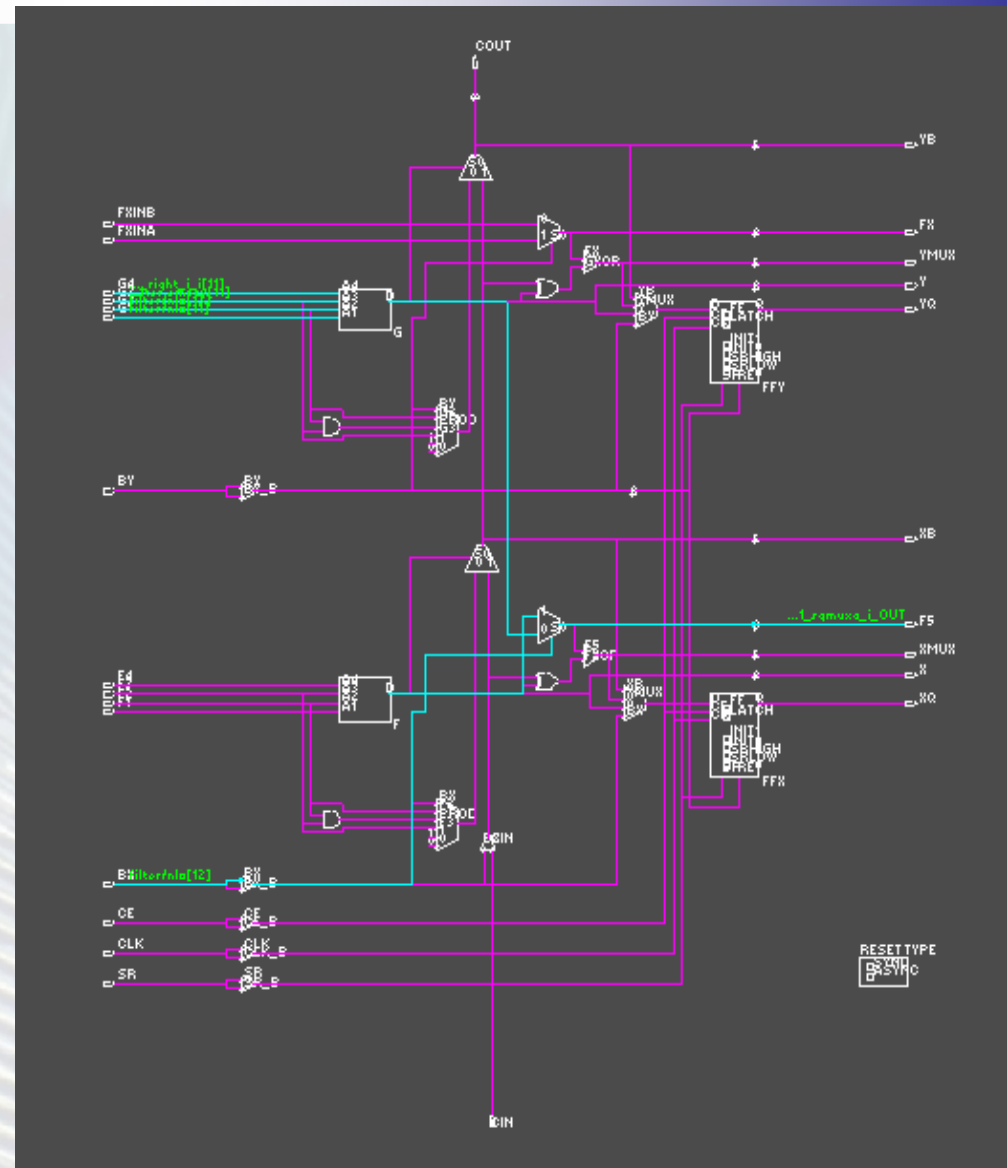
Xilinx FPGA-k

- Részlet egy kész tervből: logikai blokkok + huzalozás



Xilinx FPGA-k felépítése

- A CLB belső felépítése az FPGA Editor-ban nézve



Xilinx FPGA: konfiguráció

- **A konfigurációt (LUT tartalom, huzalozás, csatlakozások, egyéb paraméterek) SRAM tárolja**
- **Tápfeszültség kikapcsolásakor elveszíti a konfigurációt**
- **Bekapcsolás után konfiguráció szükséges**
 - EEPROM-ból, automatikusan
 - Fejlesztői kábel segítségével ún. JTAG porton keresztül

A HDL nyelvek

- **Verilog**

- 1984: Gateway Design Automation Inc.
- 1990: Cadence -> Open Verilog International
- 1995: IEEE szabványosítás
- 2001: Verilog 2001
- 2005: System Verilog

- **VHDL**

- 1983-85: IBM, Texas Instruments
- 1987: IEEE szabvány
- 1994: VHDL-1993

Egyéb megoldások

- **HDL fejlesztés a szoftver fejlesztéshez viszonyítva továbbra is időigényes**
- **Sok fejlesztő rendelkezik C/C++ ismerettel, viszonylag kevés HDL ismerettel**
- **Magasszintű hardver leíró nyelvek**
 - Celoxica Handel-C: C alapú, spec. kiegészítések
 - SystemC: szabványos, ma már (részben) szintetizálható, C++ alapú
 - Mentor Catapult-C: C++ kiegészítések nélkül
 - Impulse-C, Mitrion-C

HDL nyelvek célja

- **Hardver modellezés**
 - Mindkét nyelv jelentős része csak a hardver funkciók modellezésére ill. szimulációra használható
 - Szintetizálható részalmaz szintézer függő
- **Kapuszintű modulokból építkező, kapcsolási rajzon alapuló tervezési módszerek leváltása**
- **RTL (Register Transfer Level) szintű leírás**
 - Automatikus hardver szintézis a leírásból
 - Tervezői hatékonyság növelése

HDL nyelvek

- **Alapvetően moduláris felépítésű tervezést tesz lehetővé**
- **HDL modul**
 - Be-, kimenetek definiálása
 - Be-, kimenetek közötti logikai kapcsolatok és időzítések definiálása
- **NEM szoftver**
 - Alapvetően **időben párhuzamos, konkurrens** működést ír le

Verilog szintaktika

- **Megjegyzések (mint C-ben)**

- // egy soros

- /* */ több soros

- **Konstansok**

- `<bitszám><'alap><érték>`

- `5'b00100:` 00100 decimális érték: 4, 5 bites

- `8'h4e:` 1001110 decimális érték: 78, 8 bites

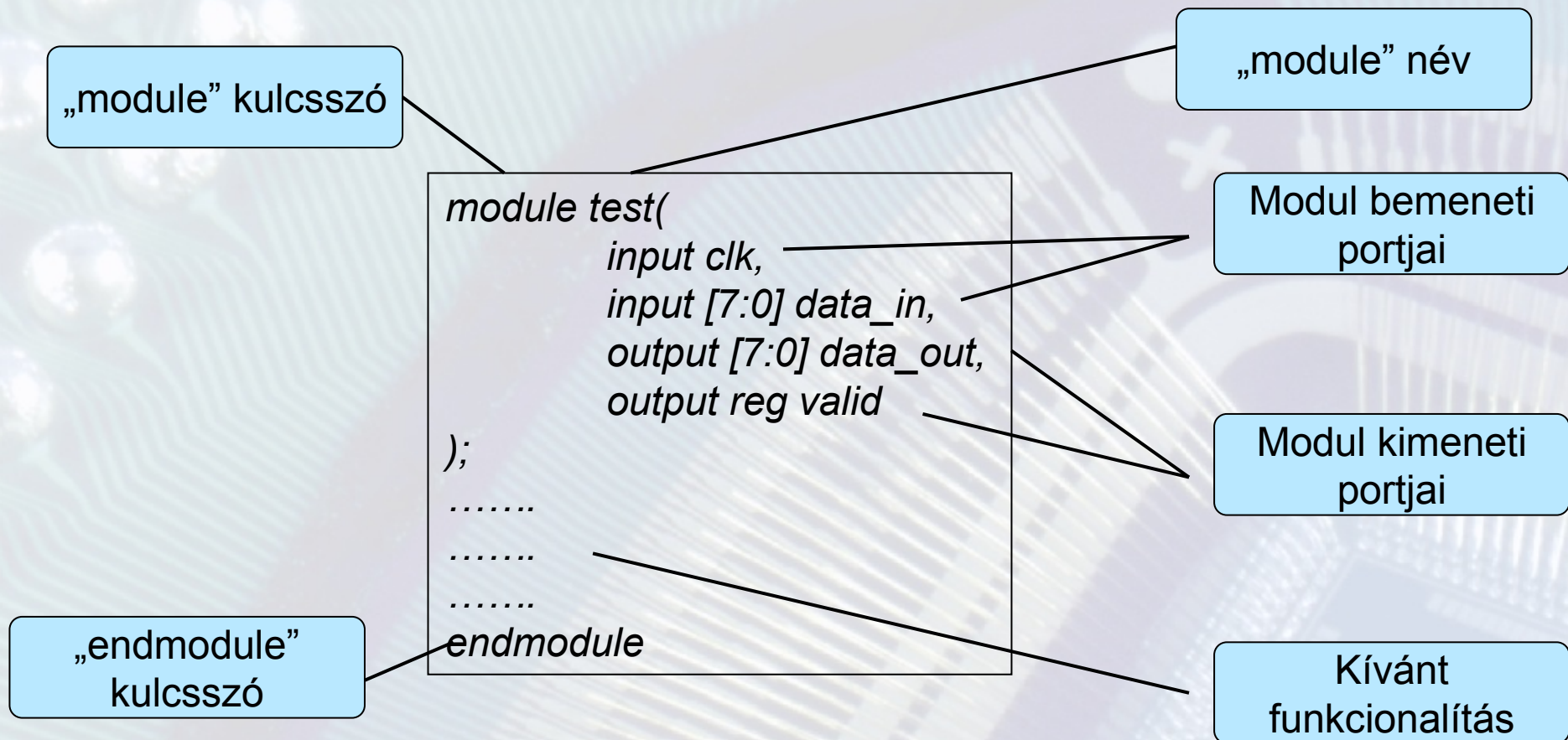
- `4'bZ:` ZZZZ nagy impedanciás állapot

Modulok

- **„Építőelem” komplex rendszerek létrehozására**
- **Hierarchikus leírás, feladat partícionálás**
- **Alkalmazható felülről lefelé, alulról felfelé tervezéskor**
- **Egy modul tetszőleges példányban beépíthető**
 - Nem szubrutin, mindegyik példány önálló valódi HW, saját erőforrásokkal
- **Adatkapcsolat az interfész leíráson keresztül**

Modul interfészlista

- Preferált a kompakt lista, kevesebb hiba



Hagyományos interfész lista

- **Dupla munka, kettős hibalehetőség**

„module” kulcsszó

„module” név

Port felsorolás
(típus nélkül)

```
module test(clk, data_in, data_out, valid);  
input clk;  
input [7:0] data_in;  
output [7:0] data_out;  
output reg valid;  
.....  
.....  
.....  
endmodule
```

Portok típusának
Megadása
Külön deklarációval

„endmodule”
kulcsszó

Bitműveletek

- **Logikai műveletek bitvektorokon (egy vagy több bites adatokon)**
 - \sim , $\&$, $|$, \wedge , (negálás, és, or, xor)
 - $\sim\&$, $\sim|$, $\sim\wedge$ (NAND, NOR, XNOR)
- **Vektorokon bitenkén, pl:**
 - $4'b1101 \& 4'b0110 = 4'b0100$
- **Ha a két operandus szélessége nem egyezik meg, a kisebbik az MSB biteken 0-val kiterjesztve**
 - $2'b11 \& 4'b1101 = 4'b0001$
- **A feltételes kifejezések logikai operátorai az igaz-hamis vizsgálatokhoz eltérőek:**
 - $!$, $\&\&$, $\|$ (negálás, és, vagy)

Bit redukciós operátorok

- Egy operandusú művelet, a bitvektor összes bitjét önálló egybites változóként értelmezve, eredménye is egy bites

- $\&$, $\sim\&$, $|$, $\sim|$, \wedge , $\sim\wedge$ (és, nem és, vagy, nem vagy)

– Példák:

- $\&4'b1101 = 1'b0$
- $|4'b1101 = 1'b1$

– Használat:

- Számláló kimenet végérték? `assign tc = &cnt;`
- ALU kimenet nulla? `assign z = ~|result;`

Komparátor operátorok

- **C-szintakszissal megegyező**
- **Egyenlőség**
 - `==`, `!=`
 - `===`: egyenlőség az „x, z” értékek figyelembevételével, azaz bizonyos bitek értéke tetszőleges
 - `!==`: nem egyenlő, „x, z” figyelembevételével
- **Nem egyenlőség**
 - `<`, `>`, `<=`, `>=`

Aritmetikai operátorok

- **C-szintakszissal megegyező**
- **Operátorok: +, -, *, /, %**
 - Nem mindegyik szintetizálható
 - Szintézer függő, de tipikusan / pl. csak akkor, ha az osztó kettő hatvány
 - Szorzásra választható implementációs stílus
 - Beépített blokk vagy LUT hálózat
 - Negatív számok kettes komplementes kódban

Egyéb operátorok

- **Konkatenálás (összefűzés): {}**, pl:
 - $\{4'b0101, 4'b1110\} = 8'b01011110$
 - $\{2\{3'b101\}, 2'b00\} = 8'b10110100$
- **Shift operátor**
 - \ll, \gg
 - $\ll\ll, \gg\gg$ Előjeles shift, MSB nem változik
- **Bit kiválasztás**
 - Kiválasztott rész konstans
 - $data[5:3]$

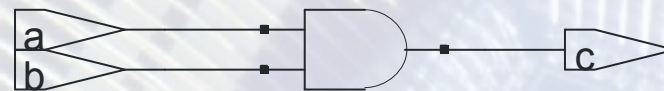
Adattípusok

- **A szintézis szempontjából kétfajta adat van**
- **A huzal típusú „wire”**
 - Nevének megfelelően viselkedik (vezeték)
 - Nincs saját állapota, az mindig örökli
 - Pl. 8 bites vezeték: `wire [7:0] data;`
- **A változó típusú „reg”**
 - Két értékadás között állapotát tartja
 - Értékadás történhet eseményvezérlésre , vagy órajelre
 - Szintézis utáni eredmény nem mindig regiszter
 - Vezeték
 - Latch
 - Flip-flop
 - Pl. 8 bites regiszter: `reg [7:0] data;`

Assign

- Tipikusan kombinációs logika leírására
- „assign”-val csak „wire” típusú változónak lehet értéket adni
- Konkurrens, folytonos értékadás
 - A bal oldali változó bármely változása a kifejezés kiértékelődését, új értékének meghatározását okozza

- Pl. `assign c = a & b;`



- Egy változó csak egy „assign” által kaphat értéket

Always blokk

- Magas szintű viselkedési leírás

- Szintakszis:

```
always @ (....)  
begin  
....  
....  
end
```

Érzékenységi lista

Blokkon belüli
műveletek

- Egy változó csak egy „always” blokkban kaphat értéket
- Always blokk nem lehet érzékeny a saját kimenetére
- Az „always” blokkok egymással párhuzamosan működnek

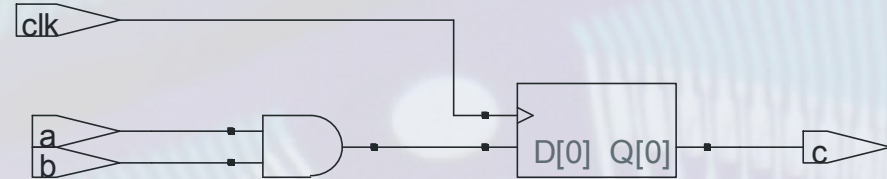
Always – értékadás

- **Eljáráson belül kétfajta értékadás**
- **Blokkoló értékadás: =**
 - Blokkolja az utána következő értékadásokat -> szekvenciális utasítás végrehajtás
- **Nem blokkoló értékadás: <=**
 - A nem blokkoló értékadások párhuzamosan hajtódnak végre, azaz a baloldali kifejezések kiértékelődnek az aktuális változó értékek szerint és az eredmény csak a fázis végén adódik át a bal oldali változónak
- **Blokkoló – nem blokkoló példa később**
 - A Verilog egyik népszerű témája

Always – Flip Flop

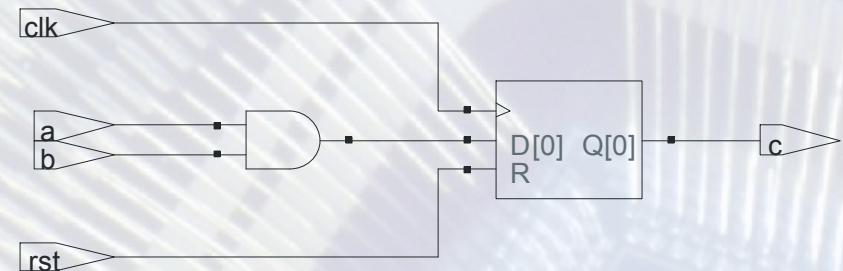
- **Flip Flop: érzékeny D tároló**

```
always @ (posedge clk)
    c <= a & b;
```



- **Szinkron reset**

```
always @ (posedge clk)
if (rst)
    c <= 1'b0;
else
    c <= a & b;
```



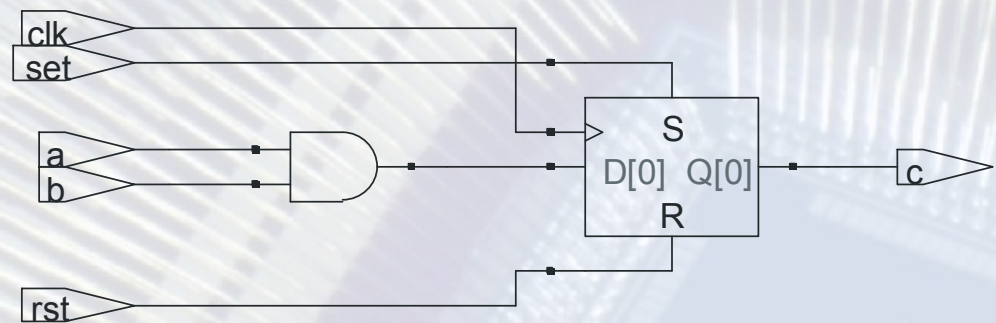
- **Aszinkron reset**

```
always @ (posedge clk, posedge rst)
if (rst)
    c <= 1'b0;
else
    c <= a & b;
```

Always – Flip Flop

- **Xilinx FPGA-kban a FF egy CLK bemenettel, két alaphelyzet beállító jellel és egy CE órajel engedélyező bemenettel rendelkezik.**
 - Szinkron vezérlés: Minden jel kiértékelése szinkron, ebben az esetben érvényesítés az órajel aktív élénél

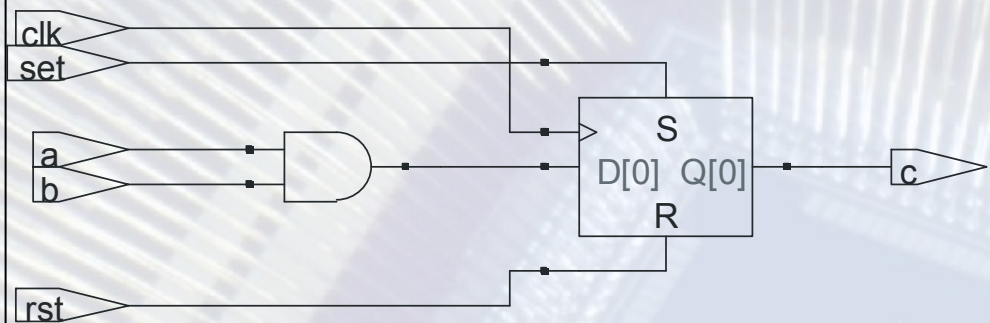
```
always @ (posedge clk)
if (rst)
    c <= 1'b0;
else if (set)
    c <= 1'b1;
else
    c <= a & b;
```



Always – Flip Flop

- **Xilinx FPGA-kban a FF egy CLK bemenettel, két alaphelyzet beállító jellel és egy CE órajel engedélyező bemenettel rendelkezik.**
 - Aszinkron vezérlés: A vezérlőjelek változása azonnal érvényre jut, prioritás a felírás sorrendjében

```
always @ (posedge clk, posedge rst,  
posedge set)  
if (rst)  
    c <= 1'b0;  
else if (set)  
    c <= 1'b1;  
else  
    c <= a & b;
```



Always – kombinációs logikához

- **Szemléletesen:**

- Az always blokk eseményvezérelt
- A bemenőjelek bármely változása ilyen esemény
- Ennek hatására az eljárás lefut, a kimenet kiértékelődik

```
always @ (a, b)  
c <= a & b;
```

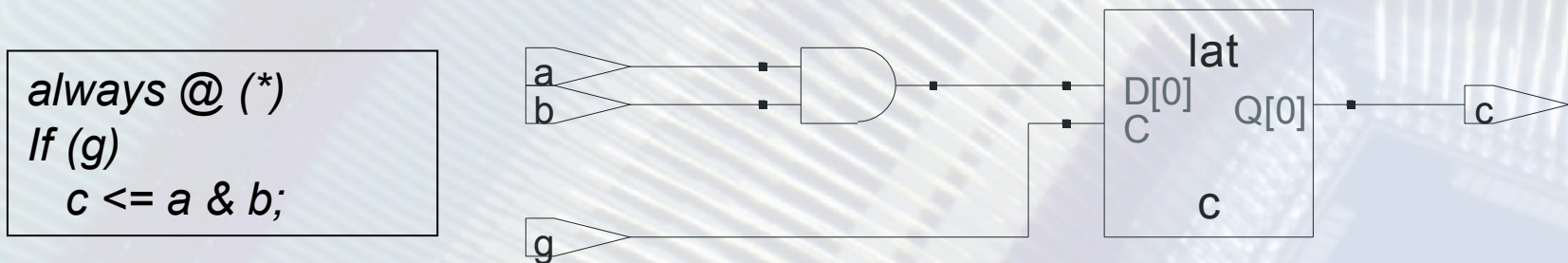
```
always @ (*)  
c <= a & b;
```



- Ha egy változó kimarad az érzékenységi listából, akkor véletlen latch keletkezhet

Always – latch

- **Latch tároló természetesen szándékosan is generálható:**
 - Az engedélyező „gate” bemenet magas értéke mellett a tároló transzparens, míg a „gate” bemenet alacsony értéke mellett zárt, tartja értékét.

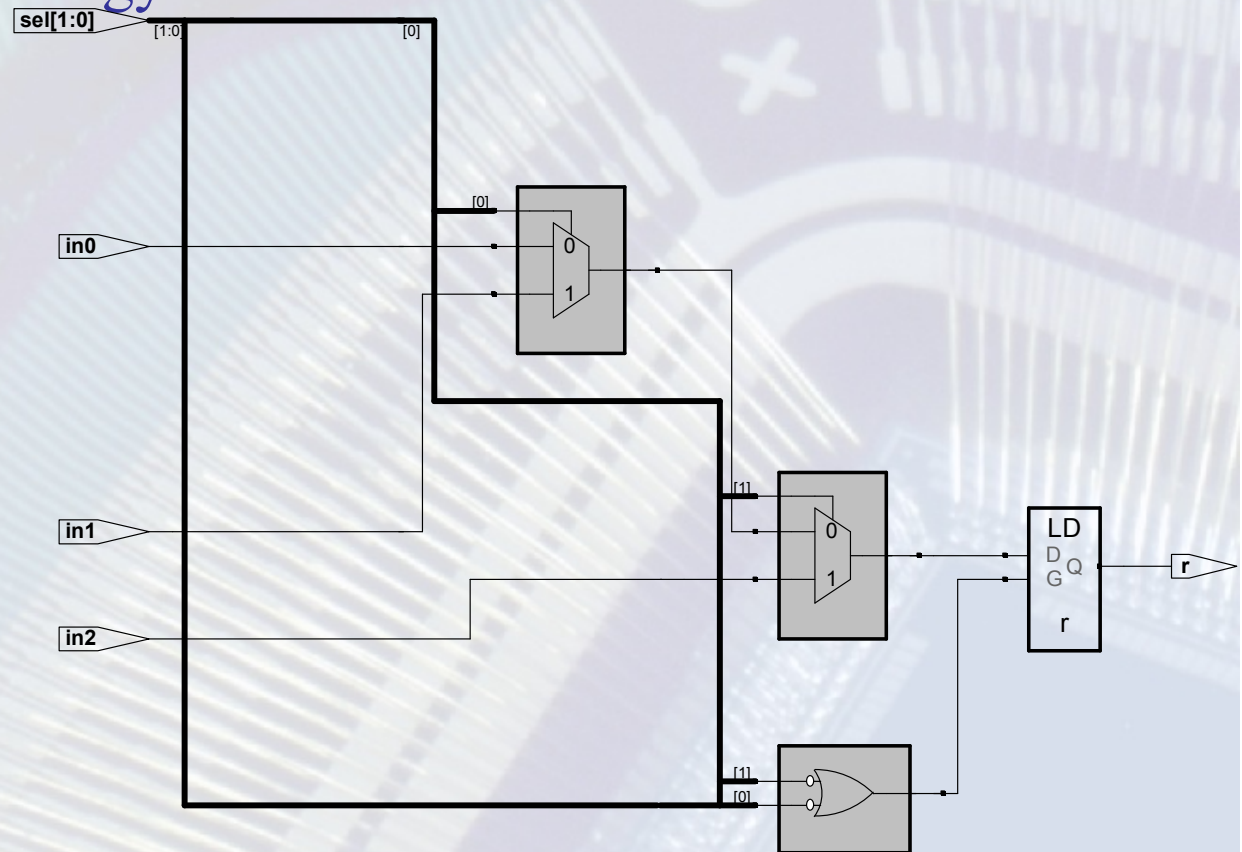


Always – latch hiba

- **A tipikus véletlen „Latch”**
 - Nem teljes “if” vagy „case” szerkezet
 - Szintézer általában figyelmeztet

```
always @ (*)  
case (sel)  
  2'b00: r <= in0;  
  2'b01: r <= in1;  
  2'b10: r <= in2;  
endcase
```

```
always @ (*)  
if (sel==0)  
  r <= in0;  
else if (sel==1)  
  r <= in1;  
else if (sel==2)  
  r <= in2;
```

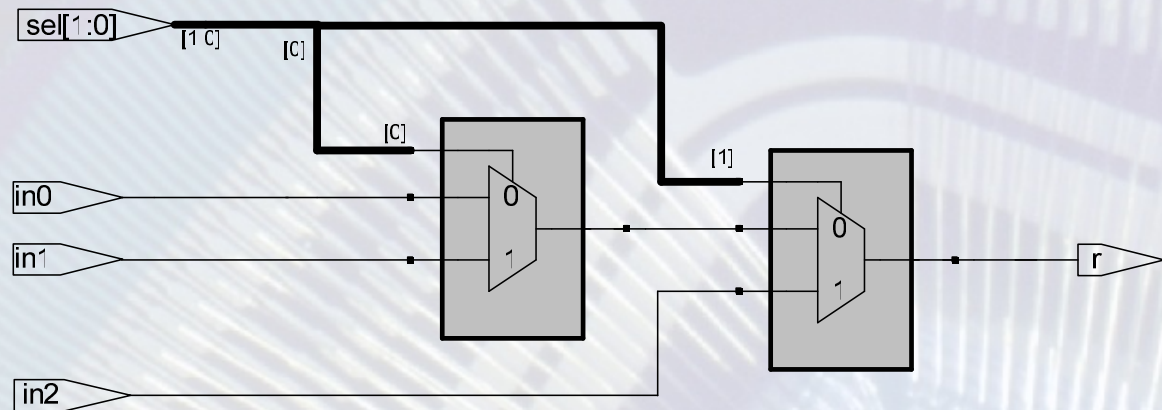


Always – helyes

- **Helyes kód**

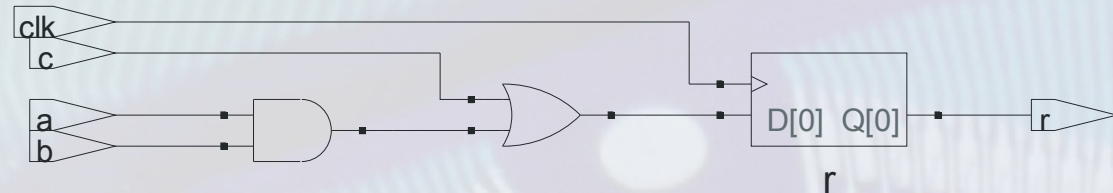
```
always @ (*)
case (sel)
  2'b00: r <= in0;
  2'b01: r <= in1;
  2'b10: r <= in2;
  default: r <= 'bx;
endcase
```

```
always @ (*)
if (sel==0)
  r <= in0;
else if (sel==1)
  r <= in1;
else
  r <= in2;
```

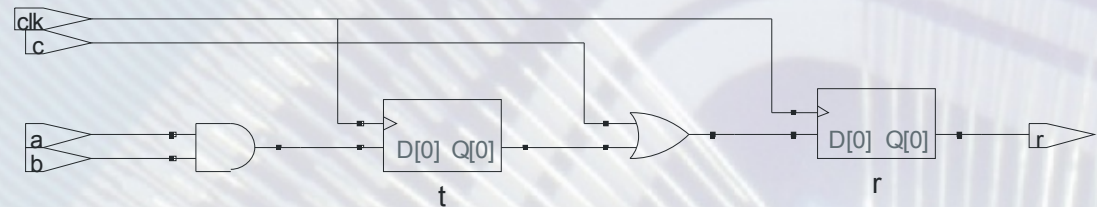


Blokkoló – nem blokkoló (1)

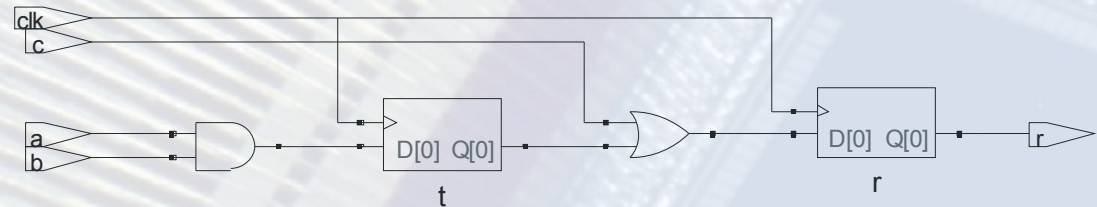
```
reg t, r;  
always @ (posedge clk)  
begin  
    t = a & b;  
    r = t | c;  
end
```



```
reg t, r;  
always @ (posedge clk)  
begin  
    t <= a & b;  
    r <= t | c;  
end
```

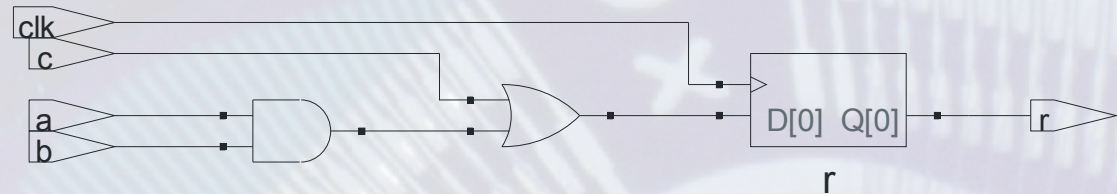


```
reg t, r;  
always @ (posedge clk)  
begin  
    r = t | c;  
    t = a & b;  
end
```

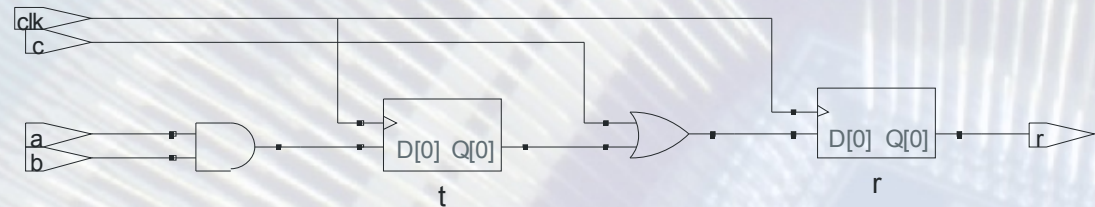


Blokkoló – nem blokkoló (2)

```
reg t, r;  
always @ (posedge clk)  
begin  
    t = a & b;  
    r <= t | c;  
end
```



```
reg t, r;  
always @ (posedge clk)  
begin  
    t <= a & b;  
    r = t | c;  
end
```



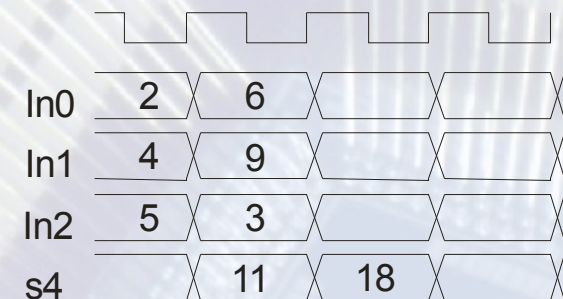
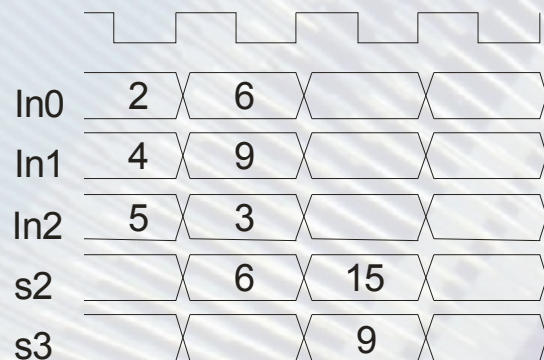
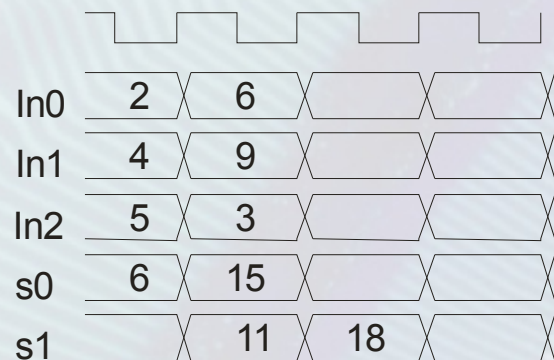
Blokkoló – nem blokkoló (3)

Pl. 3 bemenetű összeadó

```
reg s0, s1;  
always @ (posedge clk)  
begin  
    s0 = in0 + in1;  
    s1 = s0 + in2;  
end
```

```
reg s2, s3;  
always @ (posedge clk)  
begin  
    s2 <= in0 + in1;  
    s3 <= s2 + in2;  
end
```

```
reg s4;  
always @ (posedge clk)  
begin  
    s4 <= in0 + in1 + in2;  
end
```

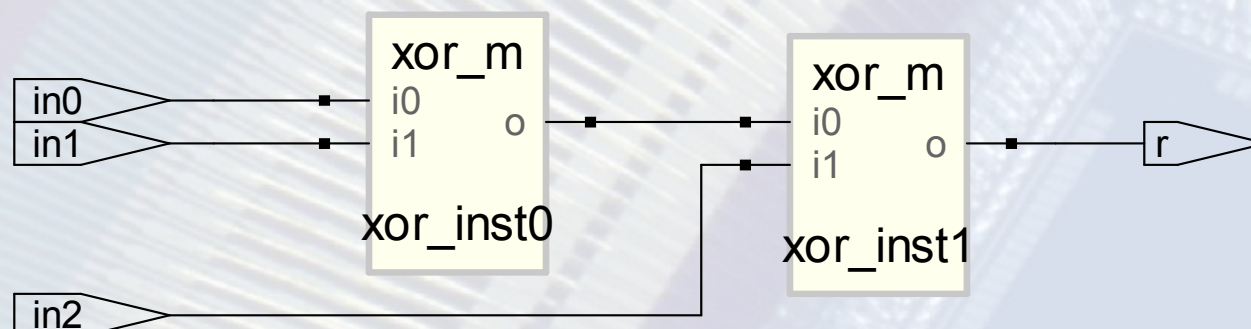


Strukturális leírás

- **Hierarchia felépítése: modulok összekapcsolása**

```
module top_level (input in0, in1, in2, output r);  
  wire xor0;  
  xor_m xor_inst0(.i0(in0), .i1(in1), .o(xor0));  
  xor_m xor_inst1(.i0(xor0), .i1(in2), .o(r));  
endmodule
```

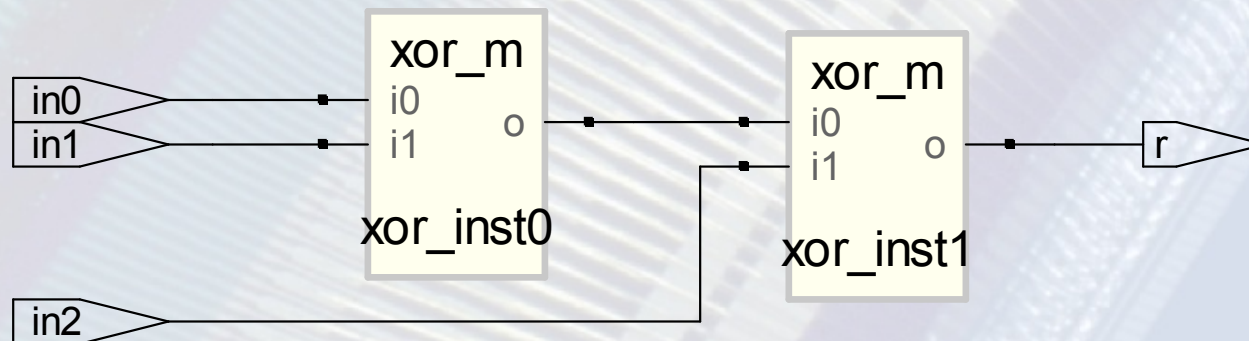
```
module top_level (input in0, in1, in2, output r);  
  wire xor0;  
  xor_m xor_inst0(in0, in1, xor0);  
  xor_m xor_inst1(xor0, in2, r);  
endmodule
```



Strukturális leírás - generate

- Hierarchia felépítése: modulok összekapcsolása**

```
wire [2:0] in_bus0; wire [1:0] in_bus1;  
assign in_bus0[0] = in0;  
assign in_bus1 = {in2, in1};  
generate  
  genvar k;  
  for (k=0; k < 2; k++)  
    begin: inst  
      xor_m(.i0(in_bus0[k]), .i1(in_bus1[k]), .o(in_bus0[k+1]));  
    end  
endgenerate
```



Példa – MUX (1.)

- **Különböző leírási stílusok a 2:1 multiplexerre**

```
module mux_21 (input in0, in1, sel, output r);  
  assign r = (sel==1'b1) ? in1 : in0;  
endmodule
```

```
module mux_21 (input in0, in1, sel, output reg r);  
  always @ (*)  
  if (sel==1'b1) r <= in1;  
  else          r <= in0;  
endmodule
```

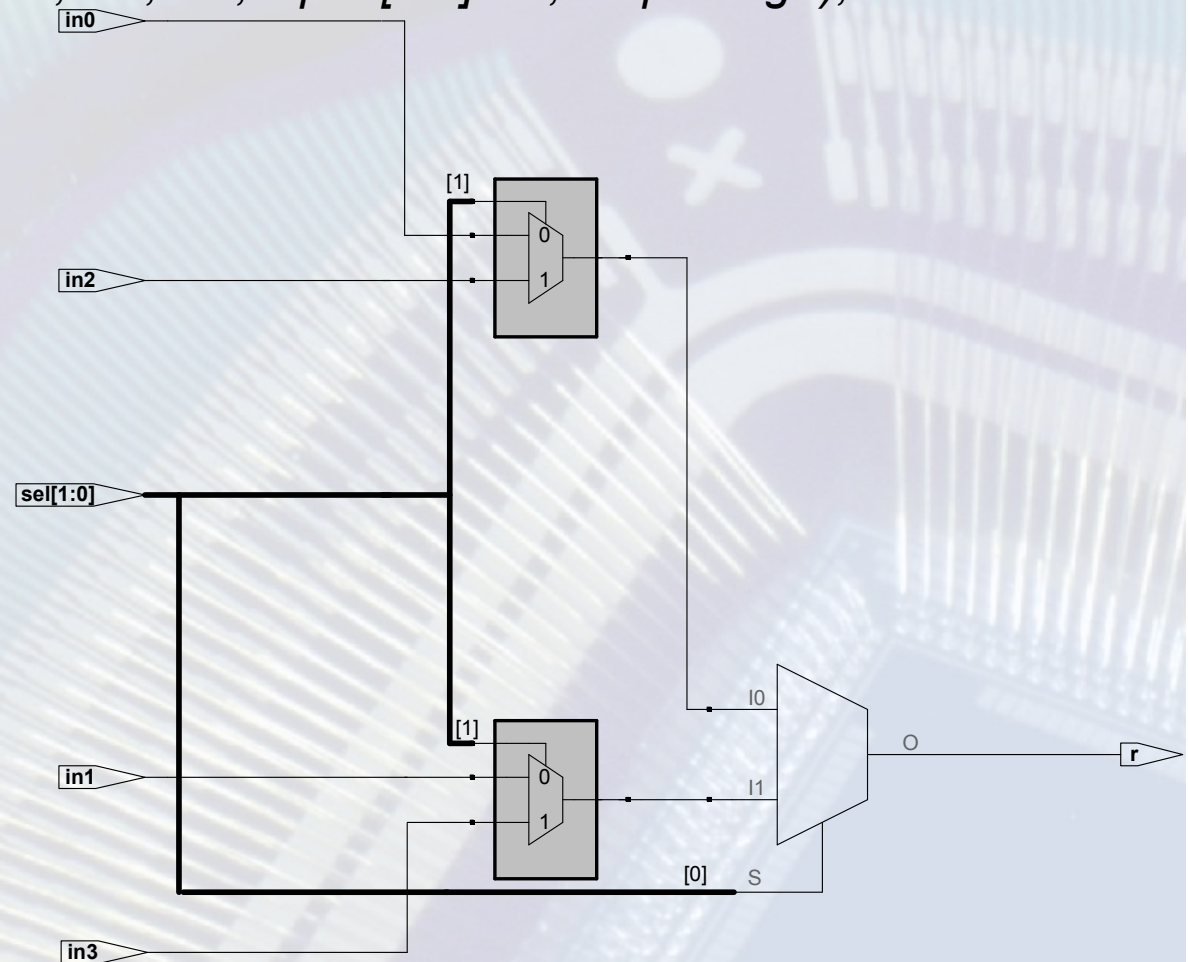
```
module mux_21 (input in0, in1, sel, output reg r);  
  always @ (*)  
  case(sel)  
    1'b0: r <= in0;  
    1'b1: r <= in1;  
  endmodule
```



Példa – MUX (2.)

- **4:1 multiplexer**

```
module mux_41 (input in0, in1, in2, in3, input [1:0] sel, output reg r);  
always @ (*)  
case(sel)  
    2'b00: r <= in0;  
    2'b01: r <= in1;  
    2'b10: r <= in2;  
    2'b11: r <= in3;  
endcase  
endmodule
```

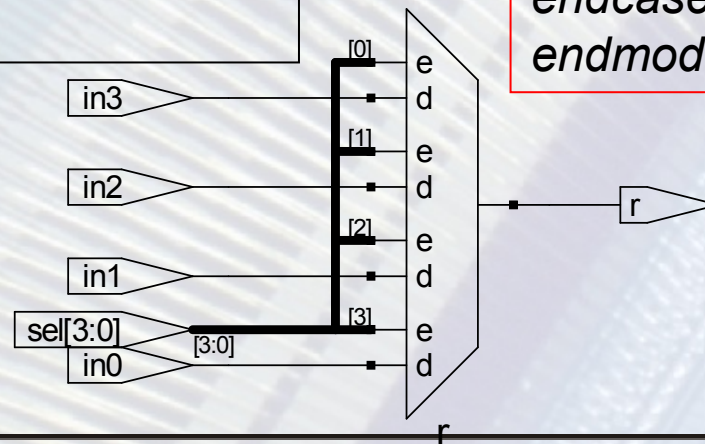


Példa – MUX (3.)

- 4:1 multiplexer, 4 bites dekódolt kiválasztó jelek

```
always @ (*)
casez(sel) /*synthesis parallel_case*/
  4'b1???: r <= in0;
  4'b?1???: r <= in1;
  4'b??1?: r <= in2;
  4'b???1: r <= in3;
  default: r <= 'bx;
endcase
endmodule
```

```
always @ (*)
case(sel)
  4'b1000: r <= in0;
  4'b0100: r <= in1;
  4'b0010: r <= in2;
  4'b0001: r <= in3;
  default: r <= 'bx;
endcase
endmodule
```



Példa – 1 bites összeadó

```
module add1_full (input a, b, cin, output cout, s);  
xor3_m xor(.i0(a), .i1(b), .i2(cin), .o(s));  
wire a0, a1, a2;  
and2_m and0(.i0(a), .i1(b), .o(a0));  
and2_m and1(.i0(a), .i1(cin), .o(a1));  
and2_m and2(.i0(b), .i1(cin), .o(a2));  
or3_m or(.i0(a0), .i1(a1), .i2(a2), .o(cout))  
endmodule
```

```
module add1_full (input a, b, cin, output cout, s);  
    assign s = a ^ b ^ cin;  
    assign cout = (a & b) | (a & cin) | (b & cin);  
endmodule
```

```
module add1_full (input a, b, cin, output cout, s);  
    assign {cout, s} = a + b + cin;  
endmodule
```

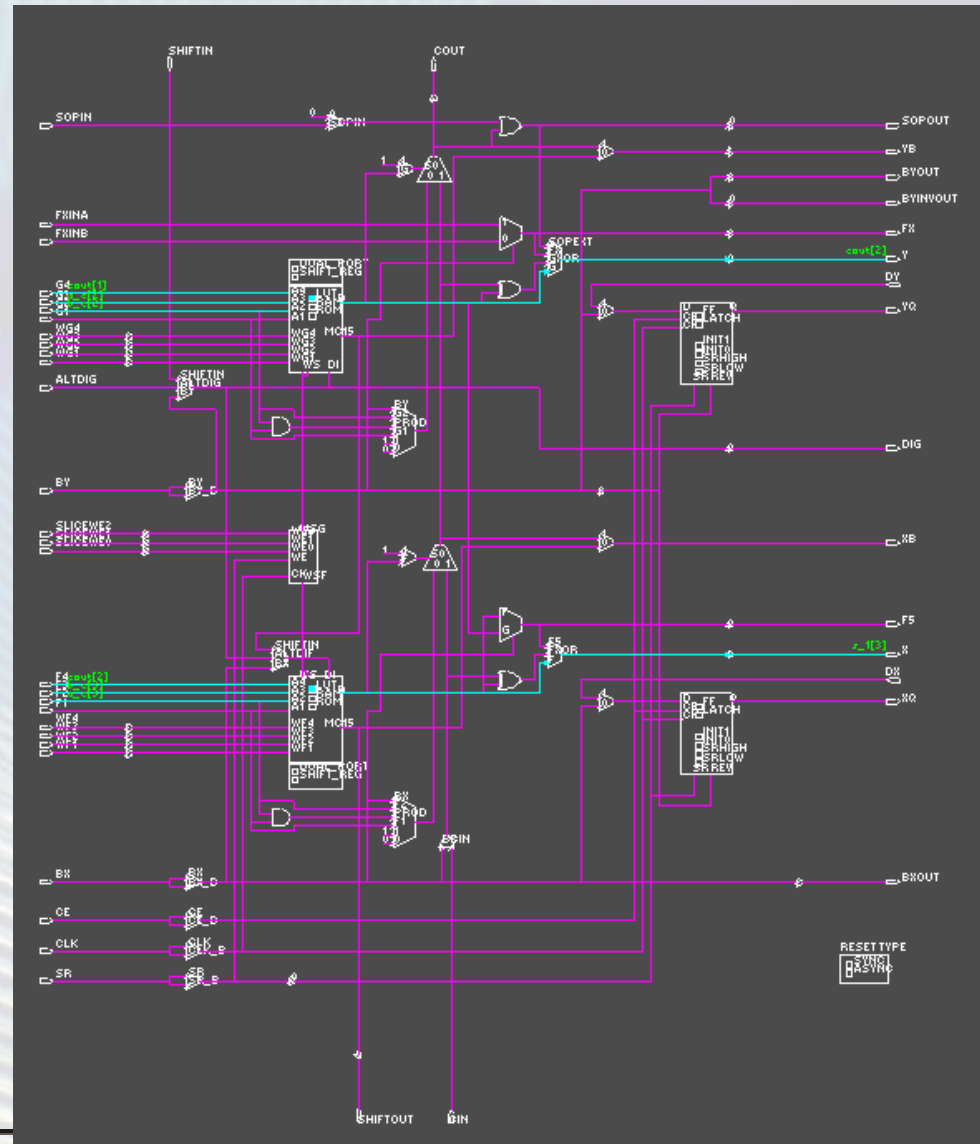
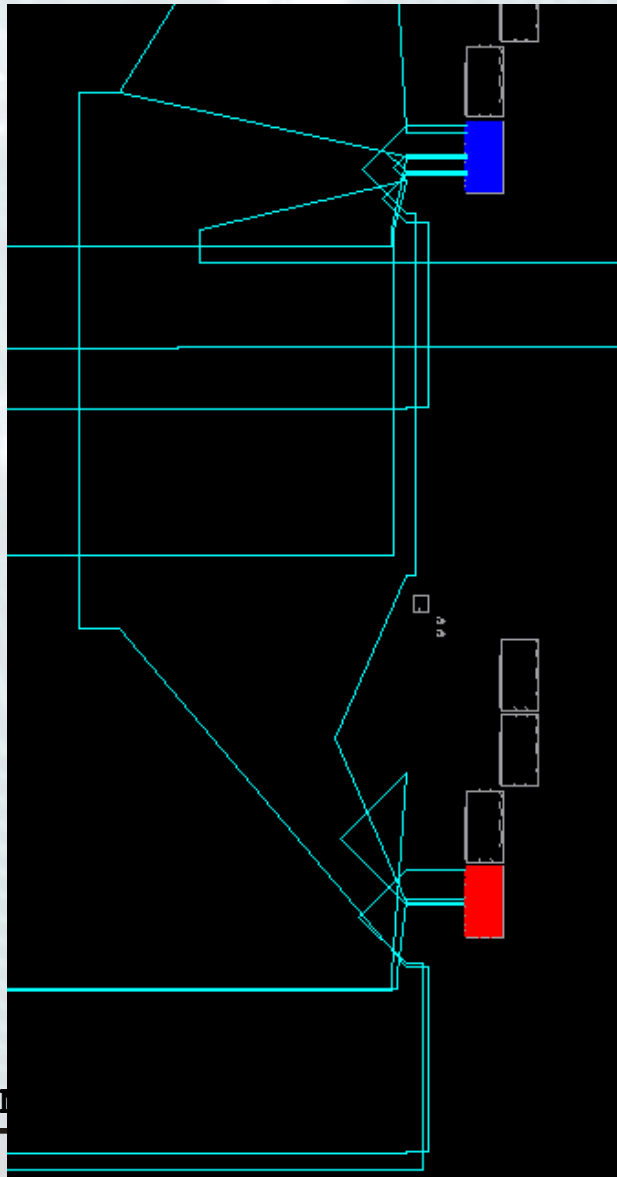


Példa – 4 bites összeadó

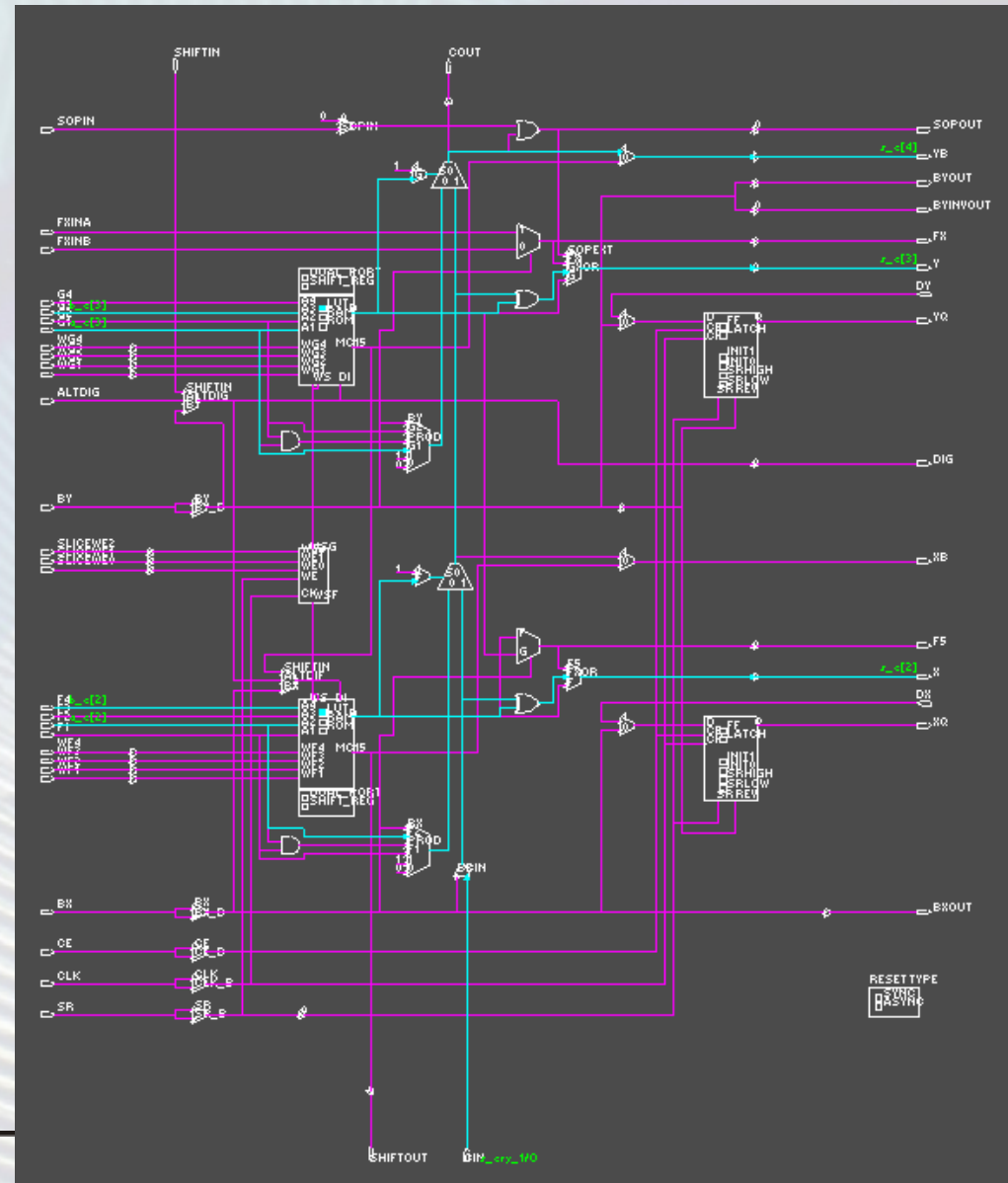
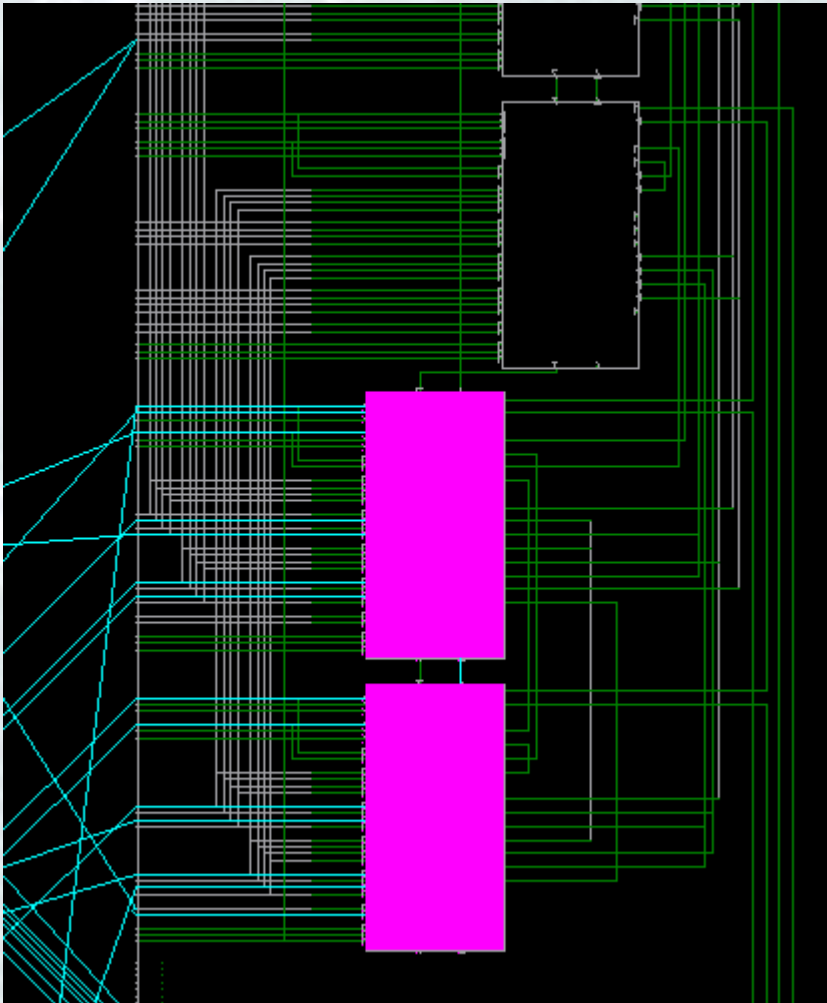
```
module add4 (input [3:0] a, b, output [4:0] s);  
  wire [3:0] cout;  
  add1_full add0(.a(a[0]), .b(b[0]), .cin(1'b0), .cout(cout[0]), .s(s[0]));  
  add1_full add1(.a(a[1]), .b(b[1]), .cin(cout[0]), .cout(cout[1]), .s(s[1]));  
  add1_full add2(.a(a[2]), .b(b[2]), .cin(cout[1]), .cout(cout[2]), .s(s[2]));  
  add1_full add3(.a(a[3]), .b(b[3]), .cin(cout[2]), .cout(s[4]), .s(s[3]));  
endmodule
```

```
module add4 (input [3:0] a, b, output [4:0] s);  
  assign s = a + b;  
endmodule
```

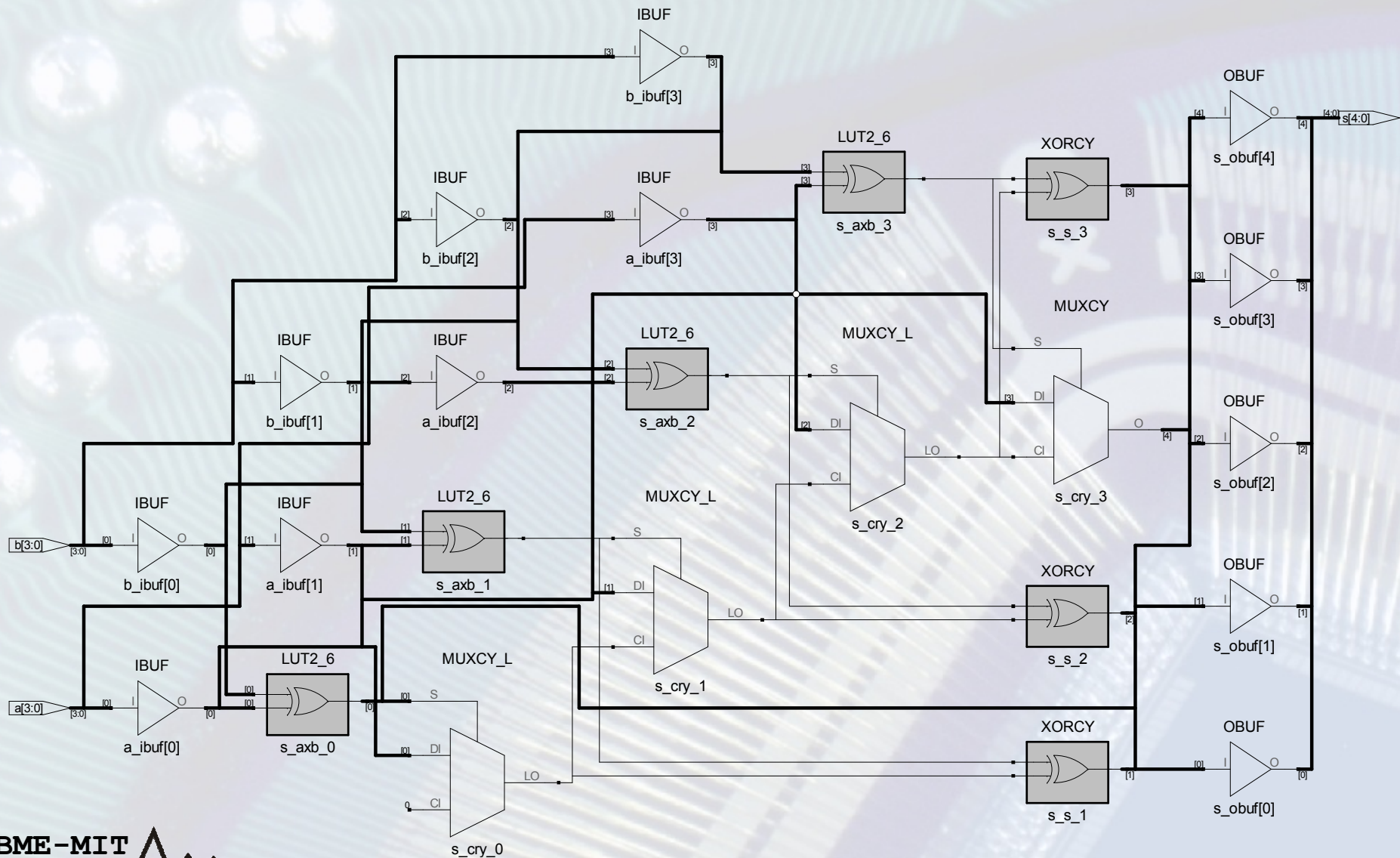
Példa – 4 bites összeadó, logikai op.



Példa – 4 bites összeadó, + operátor



Példa – 4 bites összeadó, +



Példa: Shift regiszter

- **16 bites shift regiszter,**
 - A LUT4 SRL16 soros shiftregiszter kihasználására

```
module shr (input clk, sh, din, output dout);
```

```
reg [15:0] shr;
```

```
always @ (posedge clk)
```

```
if (sh)
```

```
shr <= {shr[14:0], din};
```

```
assign dout = shr[15];
```

```
endmodule
```

Példa: Számláló

- Számláló minta leírás

- Szinkron, 8 bites
- Szinkron RESET
- Tölthető
- Engedélyezhető
- fel/le számláló

- Megj:

- A CE nagyobb prioritású, mint a töltés, ez nem tipikus

```
module m_cntr (input      clk, rst, ce, load, dir,
               input [7:0] din,
               output [7:0] dout);

    reg [7:0] cntr_reg;
    always @ (posedge clk)
        if (rst)
            cntr_reg <= 0;
        else if (ce)
            if (load)
                cntr_reg <= din;
            else if (dir)
                cntr_reg <= cntr_reg - 1;
            else
                cntr_reg <= cntr_reg + 1;

    assign dout = cntr_reg;

endmodule
```

Háromállapotú vonalak

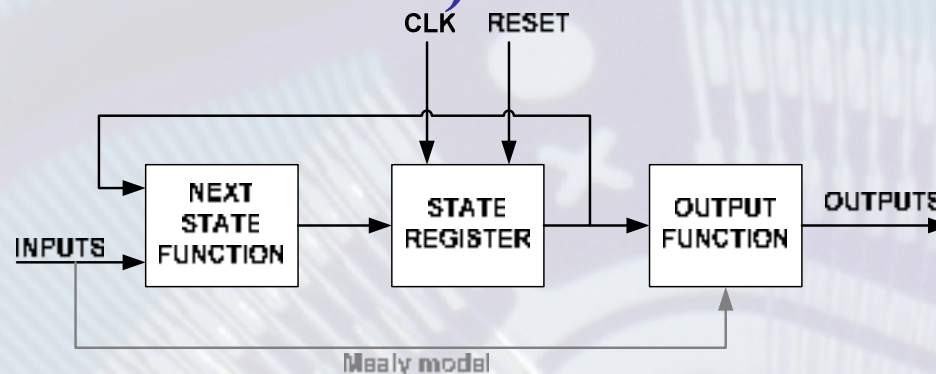
- **Kétirányú kommunikációs vonalak**

- Mai FPGA-kban belül nincs HiZ buffer!
- I/O lábak mind kétirányúak
 - Külső memóriák adatbusza
 - Processzoros busz

```
module tri_state (input clk, inout [7:0] data_io);  
  
wire [7:0] data_in, data_out;  
wire bus_drv;  
  
assign data_in = data_io;  
assign data_io = (bus_drv) ? data_out : 8'bz;  
  
endmodule
```

FSM – Finite State Machine

- **Állapotgép – vezérlési szerkezetek kialakítása**
- **Általános struktúra (Moore modell)**

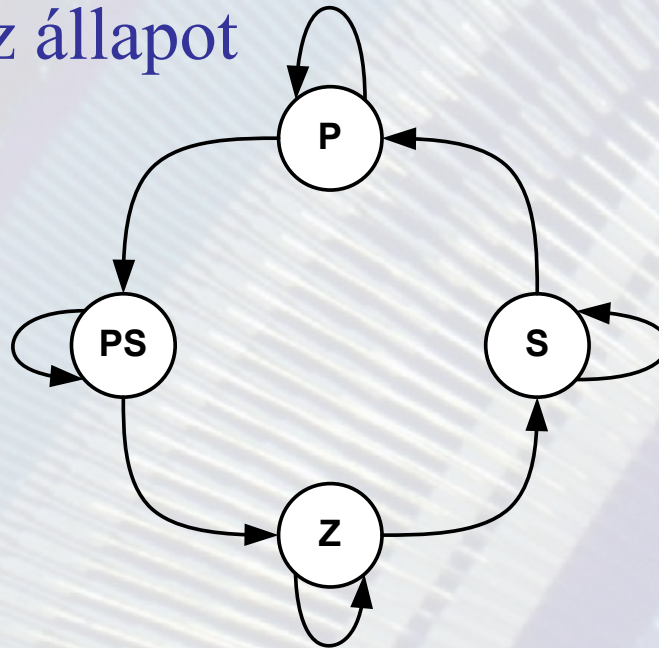


- State register: állapotváltozó
- Next state function: következő állapotot dekódoló logika
- Output function: kimeneti jeleket előállító logika
 - Moore: állapotváltozó alapján
 - Mealy: állapotváltozó + bemenetek alapján

FSM példa

- **Közlekedési lámpa**

- Állapotok: piros, sárga, zöld, piros-sárga (a villogó sárga nem implementált)
- Bemeneti változók: időzítő az egyes állapotokhoz
- Kimenet: az állapot



FSM példa – Verilog (1)

```
module lampa(  
    input      clk, rst,  
    output reg [2:0] led);  
  
parameter PIROS   = 2'b00;  
parameter PS     = 2'b01;  
parameter ZOLD   = 2'b10;  
parameter SARGA  = 2'b11;  
  
reg [15:0] timer;  
reg [1:0] state_reg;  
reg [1:0] next_state;  
  
always @ (posedge clk)  
if (rst)  
    state_reg <= PIROS;  
else  
    state_reg <= next_state;
```

```
always @ (*)  
case(state_reg)  
    PIROS: begin  
        if (timer == 0)  
            next_state <= PS;  
        else  
            next_state <= PIROS;  
        end  
    PS: begin  
        if (timer == 0)  
            next_state <= ZOLD;  
        else  
            next_state <= PS;  
        end  
    SARGA: begin  
        if (timer == 0)  
            next_state <= PIROS;  
        else  
            next_state <= SARGA;  
        end  
    ZOLD: begin  
        if (timer == 0)  
            next_state <= SARGA;  
        else  
            next_state <= ZOLD;  
        end  
    default:  
        next_state <= 3'bxxx;  
endcase
```

FSM példa – Verilog (2)

```
always @ (posedge clk)
case(state_reg)
  PIROS: begin
    if (timer == 0)
      timer <= 500; //next_state <= PS;
    else
      timer <= timer - 1;
    end
  PS: begin
    if (timer == 0)
      timer <= 4000; //next_state <= ZOLD;
    else
      timer <= timer - 1;
    end
  SARGA: begin
    if (timer == 0)
      timer <= 4500; //next_state <= PIROS;
    else
      timer <= timer - 1;
    end
  ZOLD: begin
    if (timer == 0)
      timer <= 500; //next_state <= SARGA;
    else
      timer <= timer - 1;
    end
endcase
```

Időzítő

- Állapotváltáskor egy állapotfüggő kezdőértéket tölt be
- Lefelé számol
- == 0 : állapotváltás

```
always @ (*)
case (state_reg)
  PS: led <= 3'b110;
  PIROS: led <= 3'b100;
  SARGA: led <= 3'b010;
  ZOLD: led <= 3'b001;
endcase

endmodule
```

Paraméterezett modulok

- Paraméterezhető szélességű összeadó

```
module add(a, b, s);  
    parameter width = 8;  
    input [width-1:0] a, b;  
    output [width:0] s;  
  
    assign s = a + b;  
  
endmodule
```

- Paraméterezhető modul felhasználása

```
wire [15:0] op0, op1;  
wire [16:0] res;  
  
add #(  
    .width(16)  
)  
add_16(  
    .a(op0),  
    .b(op1),  
    .s(res)  
);
```

Szimuláció

- **Testbench: ISE két lehetőséget biztosít**
 - Testbench Waveform
 - Grafikus felületen megadható bemenetek
 - Verilog Test Fixture
 - Verilog kódban megírt stimulus
- **Szimulátor**
 - ISE szimulátor
 - Modelsim (MXE)

Verilog Test Fixture

- **Test Fixture**

- A Test Fixture egy Verilog modul, ez elsz mindig a legfelső szintű modul
- A tesztelendő modul almodulként van beillesztve
- Minden, a szintézisnél használt nyelvi elem felhasználható
- Nem szintetizálható nyelvi elemek is

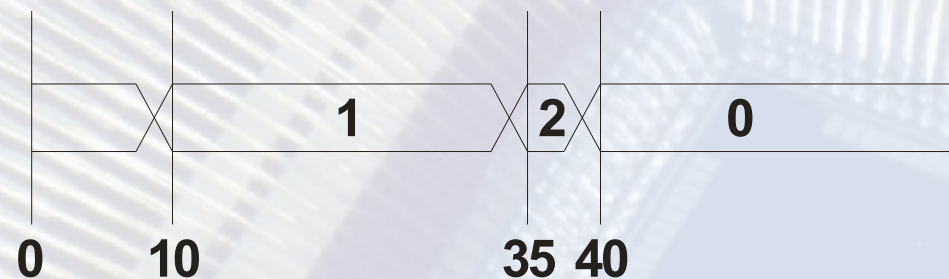
Időalap

- ‘timescale 1ns/1ps
 - Megadott idők ns-ban értendők
 - Szimulációs lépésköz: 1 ps

Test Fixture - initial

- „initial” blokk
 - 0. időpillanatban kezdődik a végrehajtása
 - Egyszer fut le, belső időzítések akumulálódnak
 - Az „initial” blokkok egymással, és az „always” blokkokkal párhuzamosan működnek

```
initial
begin
  a <= 0;
  #10 a <= 1;
  #25 a <= 2;
  #5  a <= 0;
end
```



Test Fixture - always

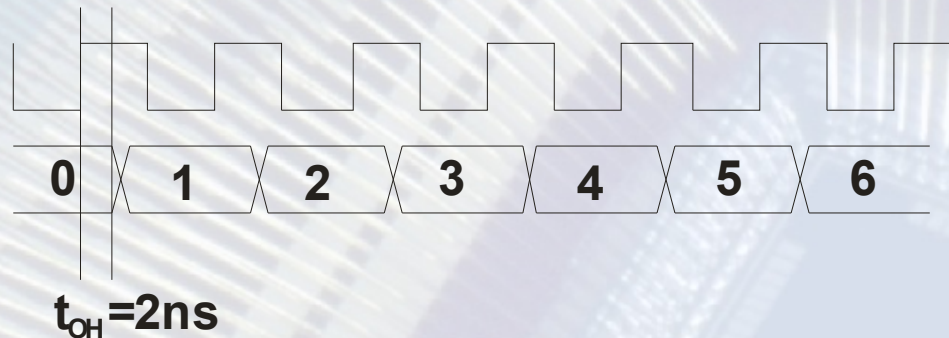
- Tipikus feladatok
- Órajel generálás

```
initial
  clk <= 1;

always #5
  clk <= ~clk;
```

- Órajelre működő bemenetek (pl. számláló)

```
initial cntr <= 0;
always @ (posedge clk)
  #2 cntr <= cntr + 1;
```



Task

- **Deklaráció:**
 - Abban a modulban, amelyik használja
 - Külön file-ban (több modulban is használható)
- **Tetszőleges számú be- és kimenet**
- **Tartalmazhat időzítést**
- **A task-ban deklarált változók lokálisak**
- **A globális változók használhatók a task-ban**
- **Task meghívhat másik task-ot**



Example - Task

- Aszinkron írás ciklus szimulációja



- Verilog kód

```
task bus_w(input [15:0] addr, input [7:0] data);  
begin  
    xaddr <= addr;  
    #5 xdata <= data;  
    #3 xwe <= 0;  
    #10 xwe <= 1;  
    while (xack != 1) wait;  
    #4 xdata <= 8'bz;  
    xaddr <= 0;  
end  
endtask;
```

Példa - Task

- „bus_w” a „tasks.v” file-ban deklarálva
- x* globális változók a test fixture-ben deklarálva
- Task felhasználása
 - 3 írás ciklus
 - 10 ns szünet a ciklusok között

```
`include "tasks.v"  
  
initial  
begin  
        bus_w(16'h0, 8'h4);  
        #10 bus_w(16'h1, 8'h65);  
        #10 bus_w(16'h2, 8'h42);  
end
```

Filekezelés

- **File-ból olvasás tömbbe**

```
reg [9:0] input_data[255:0];  
initial  
    $readmemh("input.txt", input_data);
```

- **Adatok kiírása file-ba**

```
integer file_out;  
wire res_valid;  
wire [16:0] res;  
  
initial  
    file_out = $fopen("output.txt");  
  
always @ (posedge clk)  
    if (res_valid)  
        $fwrite(file_out, "%d \n", res);
```

FPGA primitívek

- **Minden FPGA erőforrás közvetlenül is beépíthető**
- **CLB**
 - LUT 16x1 bit ROM, RAM
 - LUT 16x1 bit shift regiszter
 - MUXFi, MUXCY, XORCY
 - Flip-flop primitívek
- **BlokkRAM**
 - 16384 bit dual port memória + paritás
 - Paraméterezhető adatszélesség
- **Hardver szorzó (aszinkron, szinkron)**

LUT ROM

- ROM (aszinkron!)
 - HDL kód

```
module rom16 (input [3:0] address, output reg [7:0] data);  
always @ (*)  
case(address)  
    4'b0000: data <= KONSTANS0;  
    4'b0001: data <= KONSTANS1;  
    .....  
    4'b1111: data <= KONSTANS15;  
endcase  
endmodule
```

- Xilinx primitívek
 - ROM16X1, ROM32x1,.....

LUT RAM

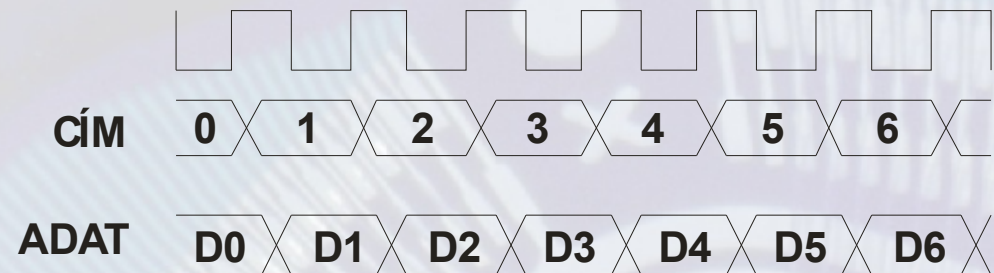
- **RAM: szinkron írás, aszinkron olvasás**
 - HDL kód

```
module ram16 (input clk, we, input [3:0] addr, input [7:0] din,  
              output [7:0] dout);  
    reg [7:0] mem[15:0];  
    always @ (posedge clk)  
    if (we)  
        mem[addr] <= din;  
  
    assign dout = mem[addr];  
endmodule
```

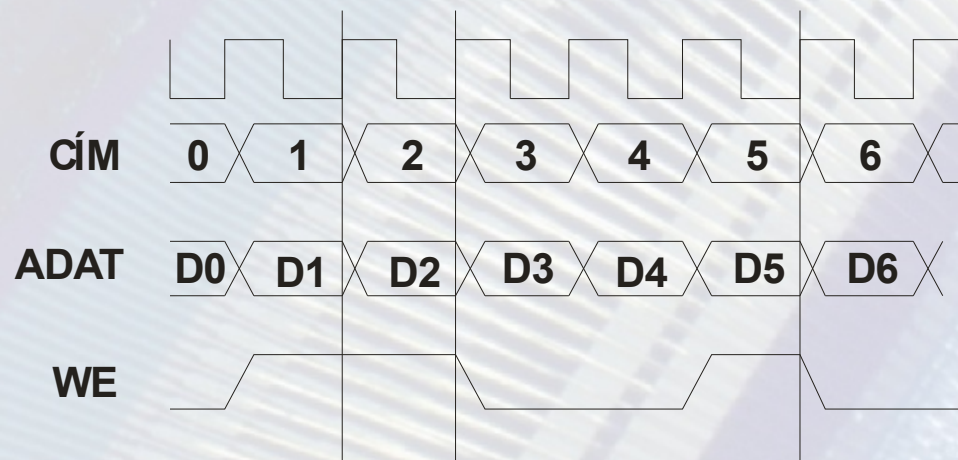
- Xilinx primitívek
 - Single port: RAM16X1S,
 - Dual port: RAM16X1D,

LUT RAM időzítés

- **Olvasás: aszinkron**
 - Számlálóval generált címzés



- **Írás: szinkron**
 - Írás történik a bejelölt órajel felfutó élénél



Shift regiszter

- LUT shift regiszter

- HDL kód

```
module shr_16x1 (input clk, sh, din, input [3:0] addr,
                 output dout);
    reg [15:0] shr;
    always @ (posedge clk)
    if (sh)
        shr <= {shr[14:0], din};

    assign dout = shr[addr];

endmodule
```

- NINCS reset bemenet

- Xilinx primitívek

- SRLC16, SRLC16E

Shift regiszter tömb

- **Maximum 16 mély 8 bit széles késleltető regiszter**

```
module shr_16x8 (input clk, sh, input [3:0] addr, input [7:0]
    din,
                    output [7:0] dout);
    reg [7:0] shr[15:0];
    integer i;
    always @ (posedge clk)
    if (sh)
    begin
        shr[0] <= din;
        for (i=15; i>0, i=i-1) begin
            shr[i] <= shr[i-1];
        end
    end
    assign dout = shr[addr];
endmodule
```

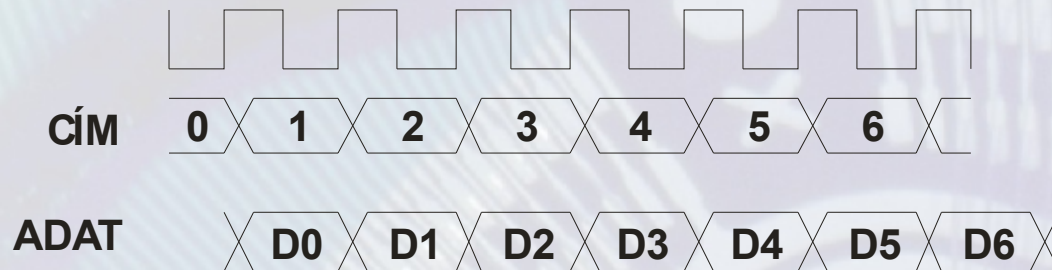
BlokkRAM

- **Szinkron, dual port memória**
 - Méret: 16384 + 2048 (paritás)
 - Adatszélesség: 1, 2, 4, 9, 18, 36 bit
 - Portok:
 - CLK, WE, EN, SSR (órajel, írás engedélyezés, engedélyezés, reset)
 - ADDR, DI, DO (cím, adat be-, kimenet)
 - Minden bemenet mintavételezett
 - Kimenet az órajel felfutó élt követően kb. 2-3 ns
 - Xilinx primitívek
 - Single port: RAMB16_S1...RAMB16_S36
 - Dual port: RAMB16_S1_S1...RAMB16_S36_S36

BlokkRAM időzítés

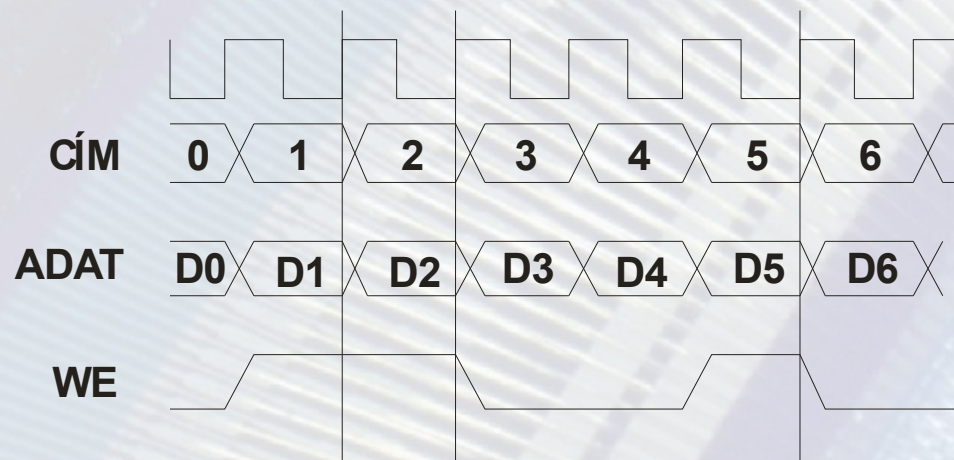
- **Olvasás: szinkron**

- Számlálóval generált címzés



- **Írás: szinkron**

- Írás történik a bejelölt órajel felfutó éleknél



Írás-olvasás ütközés

- **Írás alatt a BlokkRAM adatkimenete (írási port)**
 - Nem változik (NO_CHANGE)
 - A régi adat kerül a kimenetre (READ_FIRST)
 - Az éppen beírt adat kerül a kimenetre (WRITE_FIRST)
- **Dual-port konfiguráció esetében ha a két port címe megegyezik, és az egyikén írás történik, a másik adatkimenete érvénytelen (kivéve READ_FIRST mód)**
- **Mindkét porton azonos címre történő írás érvénytelenné teszi a beírt adatot**

BlokkRAM primitív

```
RAMB16_S9 #(
    .INIT(9'h000),           // Value of output RAM registers at startup
    .SRVAL(9'h000),        // Output value upon SSR assertion
    .WRITE_MODE("WRITE_FIRST")
) RAMB16_S9_inst (
    .DO(DO),                // 8-bit Data Output
    .DOP(DOP),              // 1-bit parity Output
    .ADDR(ADDR),            // 11-bit Address Input
    .CLK(CLK),              // Clock
    .DI(DI),                // 8-bit Data Input
    .DIP(DIP),              // 1-bit parity Input
    .EN(EN),                // RAM Enable Input
    .SSR(SSR),              // Synchronous Set/Reset Input
    .WE(WE)                 // Write Enable Input
);
```

SP BlokkRAM – Read First

```
module sp_ram(input clk, input we, input en, input [10:0] addr,  
             input [7:0] din, output [7:0] dout);  
  
reg [7:0] memory[2047:0];  
reg [7:0] dout_reg;  
  
always @ (posedge clk)  
if (en)  
begin  
    if (we)  
        memory[addr] <= din;  
        dout_reg <= memory[addr];  
end  
  
assign dout = dout_reg;  
  
endmodule
```

SP BlokkRAM – Write First

```
module sp_ram(input clk, input we, input en, input [10:0] addr,  
             input [7:0] din, output [7:0] dout);  
  
reg [7:0] memory[2047:0];  
reg [7:0] dout_reg;  
  
always @ (posedge clk)  
if (en)  
begin  
    if (we)  
        memory[addr] = din;  
        dout_reg = memory[addr];  
end  
  
assign dout = dout_reg;  
  
endmodule
```


SP BlokkRAM – No Change

```
module sp_ram(input clk, input we, input en, input [10:0] addr,  
              input [ 7:0] din, output [7:0] dout);  
  
reg [7:0] memory[2047:0];  
reg [7:0] dout_reg;  
  
always @ (posedge clk)  
if (en)  
begin  
    if (we)  
        memory[addr] <= din;  
    else  
        dout_reg <= memory[addr];  
end  
  
assign dout = dout_reg;  
  
endmodule
```

DP BlokkRAM

```
module dp_ram(input clk_a, we_a, en_a, clk_b, we_b, en_b,  
             input [10:0] addr_a, addr_b,  
             input [ 7:0] din_a, din_b, output [7:0] dout_a, dout_b);  
  
    reg [7:0] memory[2047:0];  
    reg [7:0] dout_reg_a, dout_reg_b;  
  
    always @ (posedge clk_a)  
    if (en_a)  
    begin  
        if (we_a)  
            memory[addr_a] <= din_a;  
            dout_reg_a <= memory[addr_a];  
        end  
    assign dout_a = dout_reg_a;  
  
    always @ (posedge clk_b)  
    if (en_b)  
    begin  
        if (we_b)  
            memory[addr_b] <= din_b;  
            dout_reg_b <= memory[addr_b];  
        end  
    assign dout_b = dout_reg_b;  
endmodule
```

FIFO16

- **Virtex-4/5 primitív**
 - nem adható meg HDL leírással

```
FIFO18 #(
    .ALMOST_FULL_OFFSET(12'h080),
    .ALMOST_EMPTY_OFFSET(12'h080),
    .DATA_WIDTH(4),
    .DO_REG(1),
    .EN_SYN("FALSE"),
    .FIRST_WORD_FALL_THROUGH("FALSE")
) FIFO18_inst (
    .ALMOSTEMPTY(ALMOSTEMPTY),
    .ALMOSTFULL(ALMOSTFULL),
    .DO(DO),
    .DOP(DOP),
    .EMPTY(EMPTY),
    .FULL(FULL),
    .RDCOUNT(RDCOUNT),
    .RDERR(RDERR),
    .WRCOUNT(WRCOUNT),
    .WRERR(WRERR),
    .DI(DI),
    .DIP(DIP),
    .RDCLK(RDCLK),
    .RDEN(RDEN),
    .RST(RST),
    .WRCLK(WRCLK),
    .WREN(WREN));
```

Szorzó: 18x18, előjeles

- **HDL**

- Kombinációs

```
module mul_c (input signed [17:0] a, b, output signed [35:0] p);  
    assign p = a*b;  
endmodule
```

- Szinkron

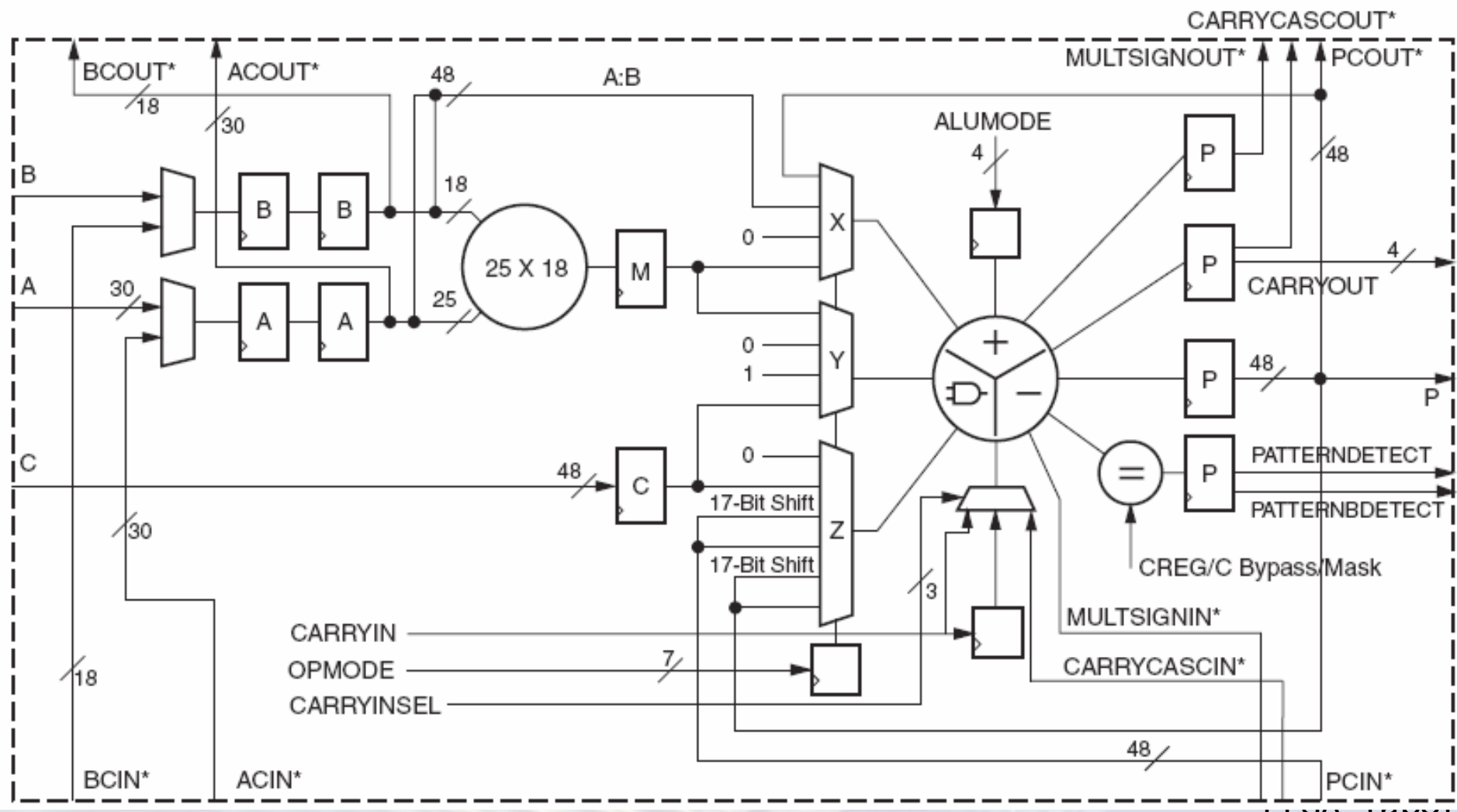
```
module mul_s (input clk, en, input signed [17:0] a, b, output reg signed [35:0] p);  
    always @ (posedge clk)  
        if (en)  
            p <= a*b;  
endmodule
```

- **Xilinx primitívek**

- MUL18X18, MUL18X18S



Virtex-5 DSP blokk

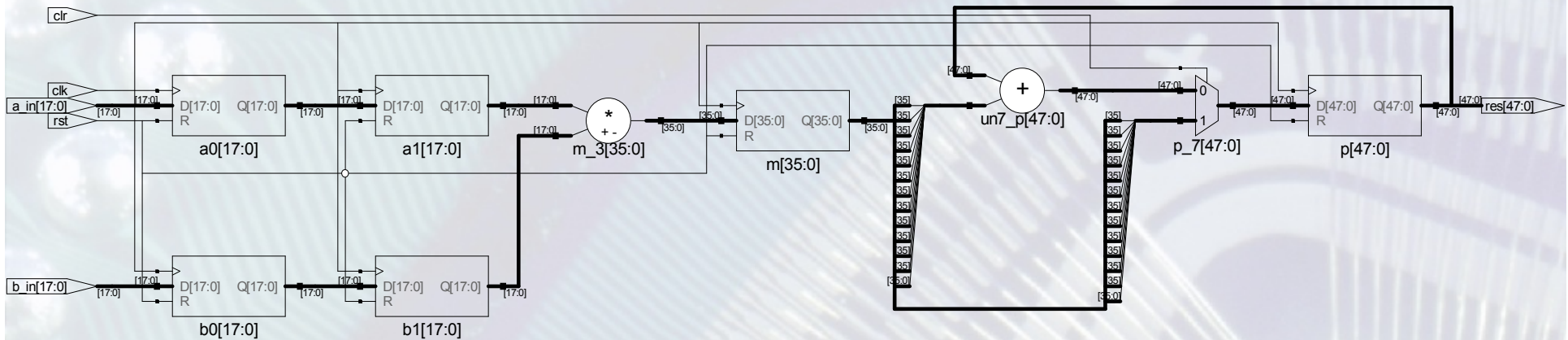


MAC

```
module mac( input      clk, rst, clr, input [17:0] a_in, b_in, output [47:0] res );  
  
  reg signed [17:0] a0, a1, b0, b1; reg signed [35:0] m; reg signed [47:0] p;  
  
  always @ (posedge clk)  
  if (rst)  
  begin  
    a0 <= 0; b0 <= 0; a1 <= 0; b1 <= 0; m <= 0; p <= 0;  
  end  
  else  
  begin  
    a0 <= a_in;  
    b0 <= b_in;  
    a1 <= a0;  
    b1 <= b0;  
    m <= a1*b1;  
    if (clr)  
      p <= m;  
    else  
      p <= p+m;  
  end  
  
  assign res = p;  
endmodule
```

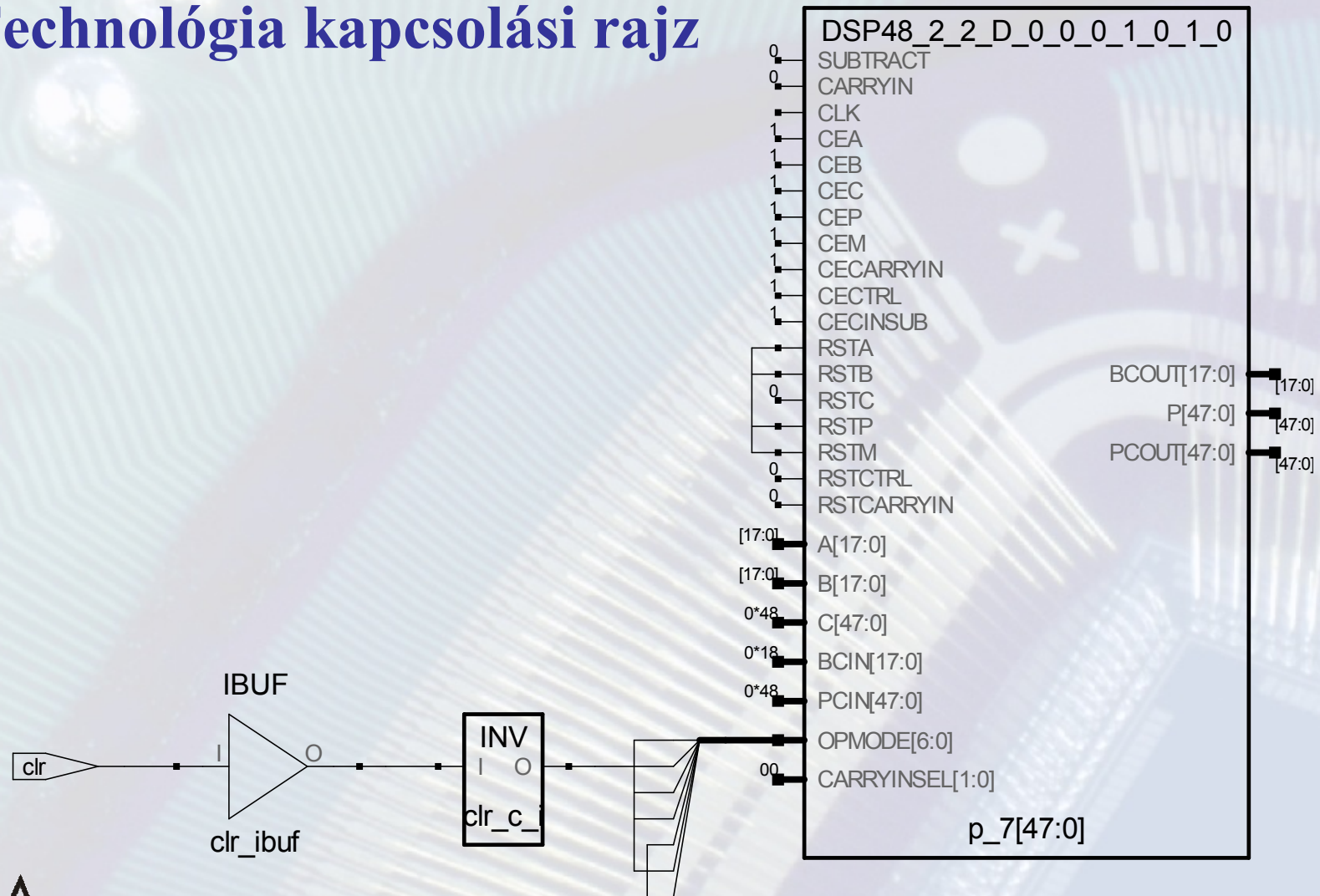
MAC – implementáció (1)

- HDL kapcsolási rajz



MAC – implementáció (2)

- Technológia kapcsolási rajz



Órajel kezelés – bufferek

- **Dedikált órajel terjesztő hálózat**
 - Külső forrás: IBUFG, IBUFGDS (automatikus, ha GCLK láb a forrás)
 - Belső jel: BUFG
- **Az órajel terjesztő hálózat (huzalozás, bufferek) száma véges -> lehető legkevesebb órajel használata**
- **BUFG használata**

```
BUFG BUFG_inst (  
  .O(O),    // Clock buffer output  
  .I(I)     // Clock buffer input  
);
```

Órajel kezelés – DCM

- **Digital Clock Manager**

- Fázis tolás
- Órajel osztás,
szorzás (törttel)

```
DCM DCM_inst (  
  .CLK0(CLK0), // 0 degree DCM CLK output  
  .CLK180(CLK180), // 180 degree DCM CLK output  
  .CLK270(CLK270), // 270 degree DCM CLK output  
  .CLK2X(CLK2X), // 2X DCM CLK output  
  .CLK2X180(CLK2X180), // 2X, 180 degree DCM CLK out  
  .CLK90(CLK90), // 90 degree DCM CLK output  
  .CLKDV(CLKDV), // Divided DCM CLK out (CLKDV_DIVIDE)  
  .CLKFX(CLKFX), // DCM CLK synthesis out (M/D)  
  .CLKFX180(CLKFX180), // 180 degree CLK synthesis out  
  .LOCKED(LOCKED), // DCM LOCK status output  
  .PSDONE(PSDONE), // Dynamic phase adjust done output  
  .STATUS(STATUS), // 8-bit DCM status bits output  
  .CLKFB(CLKFB), // DCM clock feedback  
  .CLKIN(CLKIN), // Clock input (from IBUFG, BUFG or DCM)  
  .PSCLK(PSCLK), // Dynamic phase adjust clock input  
  .PSEN(PSEN), // Dynamic phase adjust enable input  
  .PSINCDEC(PSINCDEC),  
  .RST(RST) // DCM asynchronous reset input  
);
```

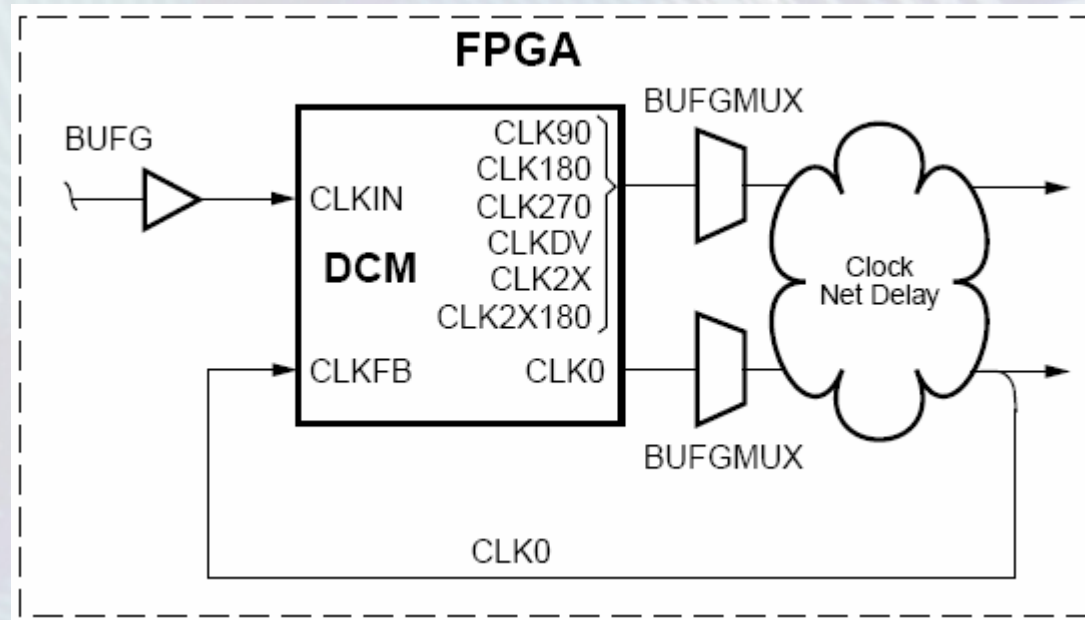
Órajel kezelés – DCM paraméterek

- **DCM paraméterezzhető modul**

```
DCM #(
    .CLKDV_DIVIDE(2.0), // Divide by: 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5
                          // 7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 or 16.0
    .CLKFX_DIVIDE(1), // Can be any integer from 1 to 32
    .CLKFX_MULTIPLY(4), // Can be any integer from 2 to 32
    .CLKIN_DIVIDE_BY_2("FALSE"), // TRUE/FALSE to enable CLKIN divide by two feature
    .CLKIN_PERIOD(0.0), // Specify period of input clock
    .CLKOUT_PHASE_SHIFT("NONE"), // Specify phase shift of NONE, FIXED or VARIABLE
    .CLK_FEEDBACK("1X"), // Specify clock feedback of NONE, 1X or 2X
    .DESKEW_ADJUST("SYSTEM_SYNCHRONOUS"), // SOURCE_SYNCHRONOUS,
    .DFS_FREQUENCY_MODE("LOW"),
    .DLL_FREQUENCY_MODE("LOW"), // HIGH or LOW frequency mode for DLL
    .DUTY_CYCLE_CORRECTION("TRUE"), // Duty cycle correction, TRUE or FALSE
    .FACTORY_JF(16'hC080), // FACTORY JF values
    .PHASE_SHIFT(0),
    .STARTUP_WAIT("FALSE")
) DCM_inst (
    .CLK0(CLK0), // 0 degree DCM CLK output
    .CLK180(CLK180), // 180 degree DCM CLK output
    .....
);
```

Órajel kezelés – DCM felhasználása

- **DCM felhasználása**
 - Bemeneti és kimeneti órajelek terjesztő hálózaton
 - CLK0 kimenet CLKFB-re visszacsatolva



Összefoglalás

- **Az FPGA erőforrások használata a HDL nyelvekből különböző módokon is elérhető**
- **Érdemes a magasabb szintű leírást használni**
 - Kevesebb munka, tömör leírás
 - Egyértelmű tervezői szándék specifikáció
 - Eszköz független leírás, könnyebben migrálható más eszközökre
- **A speciális funkciók, egyedi beállítások szükségessé tehetik az alacsonyszintű technológiai primitívek használatát**