Tantárgy: DIGITÁLIS ELEKTRONIKA
Tanár: Dr. Burány Nándor

4. félév

2+2 óra

1

III. RÉSZ
TERVEZÉS PROGRAMOZHATÓ LOGIKAI ÁRAMKÖRÖKKEL (*PLD*)
A *VERILOG* HARDVERNYELV

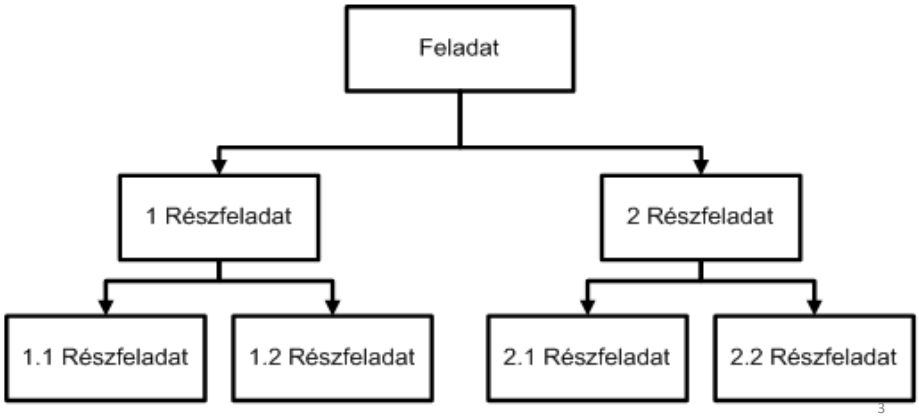
5. A tervezés menete
6. Alapfogalmak a Verilog HDL-ben
7. Modulok és port-ok
8. HDL leírás logikai kapukkal
9. Adatfolyam szintű HDL leírás
10. Viselkedési szintű HDL leírás

2

5.a A TERVEZÉS MENETE (IRÁNYA)

A digitális tervezésnél két fajta hozzáállás a jellemző:

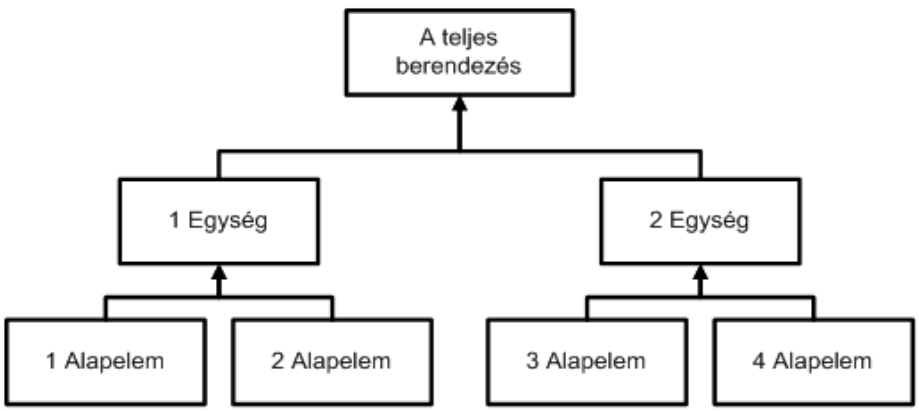
- Top-down (felülről lefelé) irány



3

5.b A TERVEZÉS MENETE (IRÁNYA)

- Bottom-up (alulról felfelé) irány

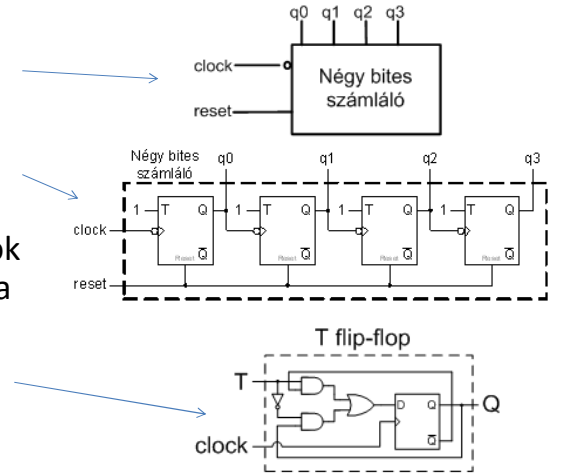


- A két módszert (irányt) gyakran ötvözzük.
- Mindkét esetben bizonyos **hierarchiáról** van szó.

4

5.1.a PÉLDA A FELÜLRŐL LEFELÉ TÖRTÉNŐ TERVEZÉSRE - NÉGY BITES SZÁMLÁLÓ MEGSZERKESZTÉSE

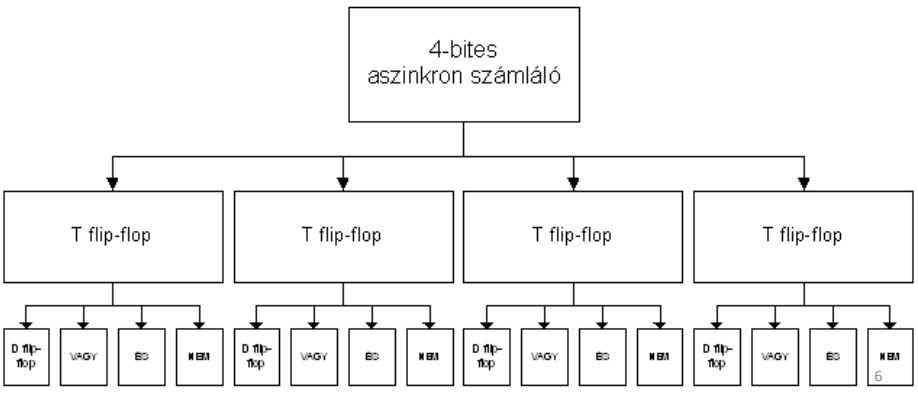
1. A számláló (a végcél)
2. A számláló felbontása T flip-flop-okra
3. A T flip-flop-ok megvalósítása D-flip-flop-ok és logikai kapuk segítségével.



5

5.1.b A NÉGY BITES SZÁMLÁLÓ HIERARCHIKUS SZERKEZETE

- A top-down tervezés eredményeként kapott szerkezet:



6

5.2.a MODULOK A VERILOG HDL-BEN

- A modulok hierarchikus tervezést tesznek lehetővé.
- A modul digitális funkcionális egységet definiál.
- A modult elég egyszer definiálni, utána több helyen/alkalommal is használható.
- A modul definíciója egy szöveges leírás.
- Az alapszerkezet:

```
module <a_modul_neve> (<port-ok_listája>);
 <a_modul_teste>
 endmodule
```

7

5.2.b MODULOK A VERILOG HDL-BEN

A szöveges leírásban a következő elvonatkoztatási (absztrakciós) szinteket használhatjuk:

1. Viselkedési- vagy algoritmus szint (a legmagasabb)
2. Adatfolyam szint
3. Logikai kapu szint
4. Kapcsoló szint (a legalacsonyabb)

- A négy szint szabadon kombinálható.
- A szöveges leírásból megfelelő szoftver végzi a tervezendő áramkör kialakítását (logikai szintézis).

8

5.3 A MODULOK ALKALMAZÁSA: INSTANCIÁK (ÁRAMKÖRI ELEMEK, KOMPONENSEK)

- Konkrét objektum létrehozása a minta (modul definíció) alapján – ez a példányosítás (instanciálás).
- Adott esetre megadjuk a csatlakozási pontokat.
- Egyedi nevet adunk a példányosított modulnak - instancianév, komponensnév.
- Példa modulok példányosítására:

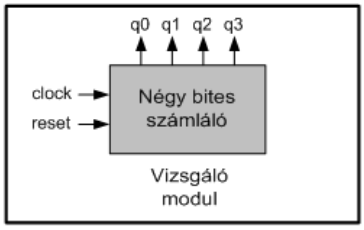
```
module szamlalo(q, clock, reset);
output [3:0] q;                    // a négy kimeneti bit deklarálása
input clock, reset;                // egy bites vezérlőjelek deklarálása
reg t=1'b1;
 T_FF tff0(q[0], t, clock, reset); // A T_FF modul példányosítása tff0 név alatt.
T_FF tff1(q[1], t, q[0], reset);   // A T_FF modul példányosítása tff1 név alatt.
T_FF tff2(q[2], t, q[1], reset);   // A T_FF modul példányosítása tff2 név alatt.
T_FF tff3(q[3], t, q[2], reset);   // A T_FF modul példányosítása tff3 név alatt.
 endmodule
```

9

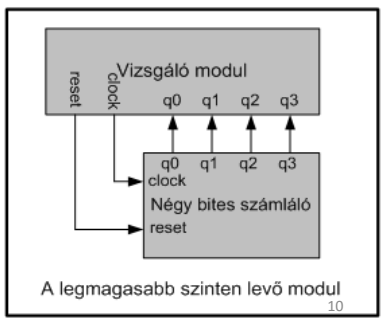
5.4 A HDL LEÍRÁSOK DIGITÁLIS SZIMULÁCIÓJA

- A működés kivizsgálása anélkül, hogy megépítenénk az áramkört (programoznánk a PLD-t).
- Nem 100%-os módszer, de nincs jobb.
- A vizsgálathoz úgyszintén HDL leírást használunk, mint a tervezéshez (leíráshoz) (de ez nem szintetizálható).

1. Vizsgáló modul példányosítja a tervezett modult



2. Egy üres modul példányosítja a vizsgáló modult és a tervezett modult is



10

6. ALAPFOGALMAK A VERILOG HDL-BEN

1. Nyelvi elemek és szabályok
2. Adatok és adathordozók típusai

11

6.1 A VERILOG HDL NYELV ELEMEI ÉS SZABÁLYAI

A nyelvi szerkezetek **jelölések** (token) **sorozatából** állnak (text file):

- üres hely (space),
- megjegyzés (kommentár, comment),
- műveleti jel (operator),
- szám (number),
- jelsor (string),
- azonosító (identifier),
- kulcsszó (keyword).

A Verilog szerkezeteket értelmező szoftver **megkülönbözteti a kis és a nagy betűket** (case sensitive).

12

6.1.1 ÜRES HELYEK

Nincs jelentőségük, azon kívül, hogy elválasztják az egyes jelöléseket.

A következő gombok lenyomásával kapunk üres helyet:

- space
- enter
- tab

Egyedül a jelsorok (string) értelmezésénél vesszük őket figyelembe.

13

6.1.2 MEGJEGYZÉSEK (KOMMENTÁROK)

Cél: a HDL leírás dokumentálása, az értelmezés megkönnyítése.

Változatok:

- egysoros (//)
- többsoros (/* */)

```
a = b + c;      // Ez egy egysoros kommentár.

/*Ez egy több soros
kommentár*/
```

14

6.1.3 MŰVELETI JELEK

- Felosztás: unáris, bináris, ternáris.
- Példák:

```
a = ~b;          //A ~ jel egy unáris műveleti jel, b a változó.
a = b && c;      // A && jel egy bináris műveleti jel, b és c
                 // a változók.
a = b ? c : d;   // A ?: jelkombináció egy ternáris
                 // műveleti jel, b, c és d a változók.
```

15

6.1.4.a SZÁMOK

Írásmódok:

- mérettel:

<méret>'<a számrendszer alapja><szám>

```
3'b101          // három bites bináris szám
16'hfa3e        // tizenhat bites hexadecimális szám
16'HFA3e        // ugyanaz, mint a fenti szám, csak egyes
                //számjegyeket nagy betűvel írtunk.
```

- méret nélkül:

<a számrendszer alapja><szám> vagy csak **<szám>**

```
'o7651          // Harminckét bites oktális szám.
'hAB3           // Harminckét bites hexadecimális szám.
4556            // Harminckét bites decimális szám.
```

16

6.1.4.b SZÁMOK

Különleges esetek a számok írásánál:

- **x** (ismeretlen) és **z** (nagyimpedanciás állapot) a számjegyek között (előfordul digitális rendszerek leírásánál)

```
12'h13x     // Tizenkét bites hexadecimális szám, a négy
            // legkisebb helyi értékű bit értéke ismeretlen.
6'hx        // Ismeretlen értékű hat bites szám.
32'bz       // Harminckét bites bináris szám. Minden bit
            // nagyimpedanciás állapotban van.
```

- **negatív** számok írásmódja (tárolás kettes komplemens formában)

```
-8'hA1      // Nyolc bites negatív szám.
8'h-A1      // A negatív számok téves írásmódja.
```

17

6.1.5 JELSOROK (STRING)

- Két idézőjel közé foglalt jelek összessége.
- Nem lehet egy sornál hosszabb.
- A szimulációk során használjuk - visszajelzések.
- Példa:

```
"Ez egy jelsor."
```

18

6.1.6 AZONOSÍTÓK

- Objektumok (modul, operandus...) nevei.
- Ezzel **azonosítjuk az objektumokat** esetleges hivatkozás során.
- Tartalmazhatnak: betűket, számokat, _ és $ jeleket.
- Nem kezdődhetnek számmal vagy $ jellel.

19

6.1.6 KULCSSZAVAK

- Az azonosítók egy különleges csoportja.
- **Nyelvi szerkezetek definiálására** szolgálnak.
- Mindig **kis betűvel** írandók.
- Példák:

```
reg szam;        // A reg kulcsszó; a szam egy adat azonosítója.
input Input;     // Az input  kulcsszó;  az Input viszont egy
                 // adathordozó azonosítója.
                 // A Verilog HDL-ben megkülönböztetjük
                 // a kis és a nagy betűket ezért az input nem
                 // ugyanazt jelenti mint az Input.
```

20

6.2 AZ ADATOK ÉS ADATHORDOZÓK TÍPUSAI

- net (wire, wand, wor, tri, triand, trior, trireg)
- reg
- vektor
- egész szám
- valós szám
- sor (array) (nem jelsor - string!)
- memória
- paraméter

A Verilog HDL-ben négy logikai állapot fordulhat elő:

Jelölés	Logikai állapot
0	logikai nulla
1	logikai egyes
x	ismeretlen állapot
z	nagyimpedanciás állapot

21

6.2.2 **net** TÍPUSÚ ADAT/ADATHORDOZÓ

- A deklarálásnál használható kulcsszavak: ***wire, wand, wor, tri, triand, trior, trireg***.
- Csomópontok, vezetékek logikai állapotát jelöli.
- Akkor kap értéket, ha logikai áramkör kimenetét hozzákötjük.
- Maga a *net* nem kulcsszó!

```
wire c;          // A csomópont (net) típusú c adathordozót
                 // deklaráló nyelvi  szerkezet
wire a, b;       // A csomópont (net) típusú a és b adathordozókat
                  // deklaráló nyelvi szerkezet
wire d = 1'b0;  // A csomópont (net) típusú d  adathordozót
                 // deklaráljuk és állandóan nulla logikai értéket
                 // rendelünk hozzá.
```

22

6.2.3 **reg** TÍPUSÚ ADATHORDOZÓ

- A deklarálásnál a ***reg*** kulcsszót használjuk.
- Értéket tud tárolni, mint a hardveres flip-flop.
- Az érték-**beírás** nem órajellel történik, hanem **hozzárendeléssel**.
- A hozzárendelések szinkronizálását később tárgyaljuk (10.3.2).
- Az első hozzárendelés előtt a tárolt érték **határozatlan** (x).

```
reg reset;        // A reset olyan adat, amely adott értéken marad.
initial           // Az initial  nyelvi szerkezetet később fogjuk
begin             // megmagyarázni.
        reset = 1'b1;   // A reset  nevű regiszternek logikai egyes
                        //értéket adunk.
        #100 reset = 1'b0;      // Száz időegység után a reset
                                //regiszter értékét
end                             // visszaállítjuk logikai nullára.
```

23

6.2.4 VEKTOROK

- **Több bites** adatok.
- **Nincs külön kulcsszó** a deklarálásra. Lehetnek **net** vagy ***reg*** típusúak.
- **Meg kell adni a bitek számát**, ellenkező esetben egy bites (skalár) adathordozót deklarálunk.

```
wire kimenet;              // A kimenet net típusú skalár adat.
wire [7:0] led_kijelzo;    // Egy 8 bites  net típusú adat vektor.
wire [15:0] ledA, ledB;    // Két 16 bites  net típusú adat vektor.
reg orajel;                // Regiszter típusú skalár adat.
reg [0:255] mem;           // Regiszter típusú 256 bites adat vektor.
```

- A teljes vektor mellett a kifejezésekben használhatók egyes bitek illetve bitcsoportok is.

```
ledB [7];          // A ledB adat 7-es számú bitje.
ledA[1:0];         // A ledB adat két legkisebb helyi értékű bitje.
```

24

6.2.5 **integer** TÍPUSÚ ADATOK (EGÉSZ SZÁMOK)

- Kulcsszó: ***integer***.
- Olyan tulajdonságokkal rendelkezik, mint a ***reg*** típusú adat.
- A számlálással kapcsolatos adathordozókat deklaráljuk így.
- **Lehet negatív** értéke is (regiszter csak pozitív számot tárolhat, vagy negatív számot kettes komplemens formában).

```
integer szamlalo;        // Az integer típusú, szamlalo elnevezésű adat
                         //deklarálása.
initial                  // Ezt a nyelvi szerkezetet később magyarázzuk.
        szamlalo = -1;   // A szamlalo elnevezésű adat kezdő értéke -1.
```

25

6.2.6 **real** TÍPUSÚ ADATOK (VALÓS SZÁMOK)

- Kulcsszó: ***real***.
- Olyan tulajdonságokkal rendelkezik, mint a ***reg*** típusú adat.
- **Valós** számértékeket lehet hozzá rendelni, **decimális vagy exponenciális** formában.

```
real alfa;          // Az alfa nevű, real  típusú adat deklarálása.
initial
begin
        alfa = 4e10;   // Az alfa adathordozóhoz exponenciális alakba
                       // felírt értéket rendelünk hozzá.
        alfa = 5.31;   // Új hozzárendelés, decimális alakban.
end
integer i;          // Az i adat integer típusú.
        i = alfa;   // Az i adat értéke 5 lesz a kerekítés következtében.
```

6.2.7 SOR (ARRAY)

- Az 1995-ös szabvány szerint csak ***reg*** és ***integer*** típusú adatokból képezhetők sorok és csak egydimenziósak. A 2001-es szabvány lehetővé teszi a többdimenziós sorokat, ugyanakkor sorok képezhetők ***net*** és ***real*** típusú adatokból is.
- **Nincs külön kulcsszó**, hanem megfelelő nyelvi szerkezettel definiáljuk: az **adathordozó neve után** írjuk a méretet.

```
integer szamok[0:7];  //Nyolc tagú, integer típusú adatokat
                      //tartalmazó sor.
reg mem[31:0];        // Harminckét egybites adatból képezett sor.
reg [3:0] port [0:7]; // Nyolc elemet tartalmazó sor, minden elem
                      // négy bites.
integer matrix [4:0] [4:0];     // Az 1995-ös Verilog szabvány nem
//támogatja többdimenziós sorok deklarálását, a 2001-es támogatja.
szamok[4]             // A szamok sor negyedik eleme.
port[5]               // A port sor ötös számú eleme.
```

27

6.2.8 MEMÓRIA

- Nincs külön kulcsszó, hanem megfelelő nyelvi szerkezettel definiáljuk: ***reg*** típusú adathordozó neve után írjuk a méretet (ugyanúgy, mint a vektoroknál).
- A szóhossz lehet egy vagy több bit.

```
reg mem1bit[0:1023];          // A mem1bit memória 1024 darab egy
                              // bites szóból alkotott sor.
reg [7:0] membyte [0:1023];   // A membyte memória 1024 darab
                              //nyolc bites szóból alkotott sor.
membyte[521]                  // A membyte név alatt deklarált
                              // memória 521-es számú eleme.
```

28

6.2.9.a PARAMÉTEREK

- A *Verilog HDL* lehetővé teszi konstansok definiálását a modulon belül a ***parameter*** kulcsszóval.
- A paraméterek használata áttekinthetőbbé teszi a hardvernyelvi leírásokat és megkönnyíti az esetleges változtatásokat (kisebb hibalehetőség).
- Különböző példányosítások során adott paraméternek különböző értékeket adhatunk.

```
parameter diodak_szama = 5; // A diodak_szama nevű adatot
        //konstansként deklaráltuk és 5-ös értéket adtunk neki.
```

29

6.2.9.b PARAMÉTEREK - A MEGADÁS ÉS AZ ALKALMAZÁS MÓDJA

- A lenti példa nyolc darab kizáró vagy kapu (XOR) példányosítását mutatja (ill. két nyocbites számon bitenként végzett kizáró vagy műveletet ír le), a kapukat 10 időegységnyi késés jellemzi.
- Ha változik a bitek száma ill. a késés, elegendő a paramétereket definiáló sort átalakítani (nem kell végigböngészni az egész modult).

```
module xorx
# (parameter width = 8, delay = 10)
(output [1:width] xout,
input [1:width] xin1, xin2);
assign #(delay) xout = xin1 ^ xin2;
endmodule
```

30

6.2.9.c PARAMÉTEREK - VÁLTOZTATÁS PÉLDÁNYOSÍTÁSKOR - SORREND SZERINT

- A lenti példában kétszer példányosítjuk az előző diakockán megadott *xorx* modult.
- Mindkét esetben nem a moduldefinícióban megadott bitszámot és késést alkalmazzuk, hanem új értékeket adunk meg (felülírjuk őket).
- Az új értékeket a paraméterek deklarálásának sorrendjében adjuk meg.

```
module overriddenParameters
(output [3:0] a1, a2);
reg [3:0] b1, c1, b2, c2;                    // a bemeneti értékeket regiszterek biztosítják
xorx #(4, 0) a(a1, b1, c1), b(a2, b2, c2);          // width=4, delay=0
endmodule
```

31

6.2.9.d PARAMÉTEREK - VÁLTOZTATÁS PÉLDÁNYOSÍTÁSKOR - NÉV SZERINT

- Ebben a példában is kétszer példányosítjuk a korábbi diakockán megadott *xorx* modult.
- Mindkét esetben nem a moduldefinícióban megadott bitszámot és késést alkalmazzuk, hanem **új értékeket adunk meg** (felülírjuk őket).
- Az új-, csak itt érvényes értékeket a paraméterek neve mellé írjuk, zárójelben.
- Ha nem adunk meg új értéket, marad a moduldefinícióban megadott érték.

```
module overriddenParameters
(output [3:0] a1, a2);
reg [3:0] b1, c1, b2, c2;                    // a bemeneti értékeket regiszterek biztosítják
xorx #(.width(4), .delay(0) ) a(a1, b1, c1), b(a2, b2, c2);
endmodule
```

32

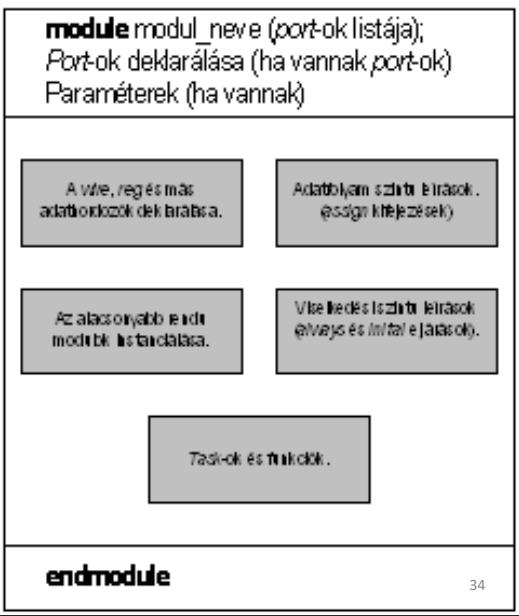
7. MODULOK ÉS PORT-OK

1. Modulok belső **szerkezete**:
   - név
   - deklarációk
   - példányosítások
   - adatfolyam szintű leírások
   - viselkedési szintű leírások
2. Modulok **csatlakoztatás**ának szabályai
   - port lista
   - port-ok deklarálása
   - port-ok összekötése belső adathordozókkal
   - portok összekötése a külső jelekkel

33

7.1.a MODULOK BELSŐ SZERKEZETE

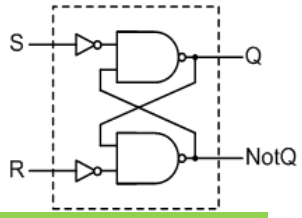
- ***module*** - kulcsszó, minden modul ezzel kezdődik.
- ***endmodule*** - kulcsszó, minden modul ezzel végződik.
- **a modul teste** - a megvalósítandó funkciót írja le.



34

7.1.b MODULOK BELSŐ SZERKEZETE

- Példa: RS latch-et leíró modul.
- Nem kell tartalmazni minden lehetséges elemet.




```
//A modul neve és a port-ok elnevezései
module RS_Latch(Q, NotQ, R, S);
 // A port-ok deklarálása
output Q, NotQ;
input R, S;
 // Alacsonyabb rendű modulok példányosítása.
// Ez esetben két NEM-ÉS kaput
// példányosítunk, amelyek alapelemek (angolul: primitive)
//a Verilog HDL-ben.
nand n1(Q, ~S, NotQ);
nand n2(NotQ, ~R, Q);
 // A modult kötelezően a következő kulcsszóval zárjuk le:
endmodule
```

35

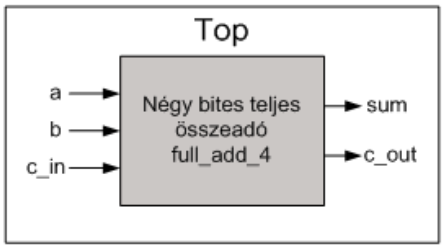
7.2 PORT-OK

- A port-ok olyanok, mint az integrált áramkörök kivezetései: ezeken keresztül valósul meg a **belső szerkezet kapcsolata a külső áramkörökkel** (más modulokkal).
- A modulba más úton nem tudunk belelátni, csak a port-okon keresztül.
- A belső szerkezetet valójában nem láthatjuk, csak a funkcionalitást tapasztaljuk.
- A belső felépítést (elnevezések, műveletek...) szabadon változtathatjuk.

36

7.2.1 PORT LISTA

- A modul definiálásakor **felsoroljuk a port-ok neveit**.
- Megeshet, hogy a modulnak nincs kapcsolata a külvilággal - ilyenkor a port lista elmarad (csak szimulációs modulokra jellemző).




```
module full_add_4(sum, c_out, a, b, c_in);
// Port listával rendelkező modul.
module Top;
// Port lista nélküli modul.
```

37

7.2.2.a PORT-OK DEKLARÁLÁSA - 1995-ÖS SZABVÁNY SZERINT

- A port deklarálás során meg kell adni az irányt: ***input*** (bemenet), ***output*** (kimenet), ***inout*** (kétirányú).

```
module full_add_4(sum, c_out, a, b, c_in);
 // A port-ok deklarálásának kezdete
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
// A port-ok deklarálásának vége.
 //---------------------------------
// A modul teste.
//---------------------------------
endmodule
```

38

7.2.2.b PORT-OK DEKLARÁLÁSA - 1995-ÖS SZABVÁNY SZERINT

- A port deklarálásával definiálunk egy azonos nevű belső adathordozót is. Alapértelmezés szerint az adathordozó típusa **wire**.
- Ha nem felel meg a ***wire*** típus, meg kell változtatni (adathordozó deklarálással).

```
module DFF(q, d, clock, reset);
output q;   // A q port a kimenetként történő deklarálás folytán
            // wire típusú lesz.
reg q;  // Mivel a kimeneti értéket memorizálni kell, a port-hoz
        // tartozó adat típusát új deklarációval meg kell
        // változtatni reg típusúra.
input d, clock, reset;
        // A modul teste.
endmodule
```

39

7.2.2.c PORT-OK DEKLARÁLÁSA - 2001-ES SZABVÁNY SZERINT

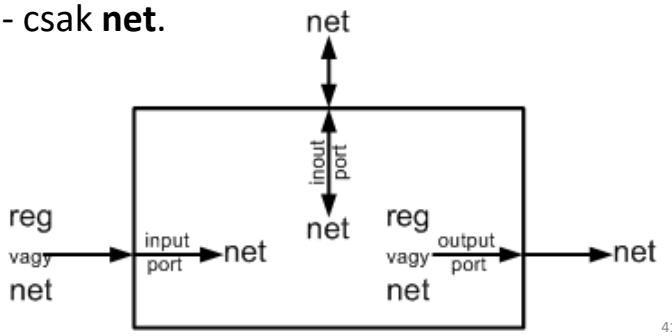
- A portok deklarálása elvégezhető a **port listában**.
- **Több deklarációt** is megadhatunk egyidejűleg (egy sorban).
- Az előző diakockán szereplő példa a következő módon írható (egyszerűbben):

```
module DFF(output reg q, input d, clock, reset);
        // Az adathordozók típusait a port listában adtuk meg.
        // Három külön deklaráció feleslegessé vált.
        // Itt következik a modul teste.
endmodule
```

40

7.2.3.a A PORT-OK ÖSSZEKÖTÉSÉNEK SZABÁLYAI

- A modul belső adathordozó típusának deklarálása összhangban kell, hogy legyen a port irányának deklarálásával:
  - **input** - csak **net**,
  - **output** - **net** vagy **reg**,
  - **inout** - csak **net**.



41

7.2.3.b A PORT-OK ÖSSZEKÖTÉSÉNEK SZABÁLYAI

- Az esetek többségében a külső és a belső vektorok bitjeinek a száma azonos.
- Különbség lehetséges, de az értelmező szoftver figyelmeztet.
- Maradhatnak össze nem kötött port-ok.

42

7.2.4.a PORT-OK ÖSSZEKÖTÉSE A KÜLSŐ JELEKKEL RENDEZETT LISTA (ORDERED LIST) ALAPJÁN

- A sorrend számít, a nevek lehetnek különbözők.

```
module full_add_4(sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
        // A modul teste.
endmodule

module Top;
 reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
        full_add_4 fa0(SUM, C_OUT, A, B, C_IN);
        // A vizsgáló rész hardver nyelvi leírása.
endmodule
```

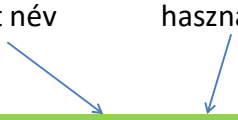
43

7.2.4.b PORT-OK ÖSSZEKÖTÉSE A KÜLSŐ JELEKKEL A PORT-OK NEVEI ALAPJÁN

- **Nagy számú port** esetén célszerű
- Nem lesz keveredés.
- A felesleges port-ok **kihagyhatók**.

a **példányosított** modul leírásában deklarált név

a ***példányosító*** modulban használt név

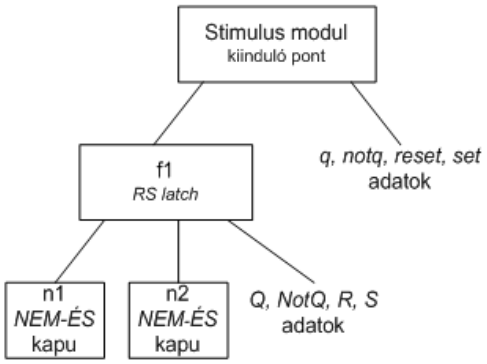


```
full_add_4 fa0(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A));
```

44

7.3 HIERARCHIKUS ELNEVEZÉSEK

- Több szintű tervezés - bonyolult struktúra
- Az egyértelmű eligazodás érdekében az instanciák (komponensek) és az adathordozók neveit kiegészítjük az elérési útvonallal.



ADATHORDOZÓK PÉLDÁNYOSÍTOTT MODULOK (INSTANCIÁK)

Stimulus.q	**Stimulus.f1.Q**	
Stimulus.notq	**Stimulus.f1.NotQ**	**Stimulus.f1**
Stimulus.reset	**Stimulus.f1.R**	**Stimulus.f1.n1**
Stimulus.set	**Stimulus.f1.S**	**Stimulus.f1.n2**

45

8. HDL LEÍRÁS LOGIKAI KAPUKKAL (GATE-LEVEL MODELING)

- A **második** elvonatkoztatási szint - gate level modeling.
- Az **első szint**et (modellezés kapcsolókkal) rendszerint csak az alkatrészgyártók alkalmazzák.
- **Régi, bevált áramköri megoldások** átmentésére alkalmas a PLD technológiába.

46

8.1 ÁLTALÁNOS TUDNIVALÓK A VERILOG LOGIKAI KAPUKRÓL

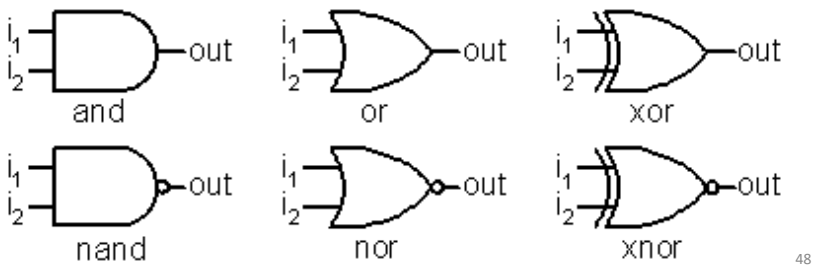
- A kapuk **előre (gyárilag) definiált modulok** a Verilog nyelvben.
- **Használatuk példányosítással** történik, mint bármely más modul esetében.
- A kapuk **modulnevei adottak**, mindig ugyanazon a módon kell őket írni, hogy az értelmező szoftver azonosítani tudja őket.
- Különböző **ÉS kapuk, VAGY kapuk és illesztő áramkörök** állnak rendelkezésre.
- Rendszerint sok más, **összetettebb áramkör** (funkcionális egység) HDL leírása is példányosítható, de ez szoftverfüggő.

47

8.1.1.a HDL *ÉS* KAPUK ÉS *VAGY* KAPUK

- Egybites (skalár) kimenet, kettő vagy több egybites bemenet.
- A port listában az első elem a kimenet, a többiek a bemenetek.

Modulnév	Funkció
and	ÉS
nand	NEM-ÉS
or	VAGY
nor	NEM-VAGY
xor	kizáró VAGY
xnor	kizáró NEM VAGY



48

8.1.1.b HDL *ÉS* KAPUK ÉS *VAGY* KAPUK

Példák kapuk példányosítására:

```
wire OUT, IN1, IN2, IN3;
// Kétbemenetű logikai kapuk példányosítása komponens névvel.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
// Három bemenetű NEM-ÉS kapu példányosítása.
nand na1_3inp(OUT, IN1, IN2, IN3);
// ÉS kapu példányosítása komponens név nélkül
// (engedélyezett módszer).
and (OUT, IN1, IN2);
```

49

8.1.1.c HDL *ÉS* KAPUK ÉS *VAGY* KAPUK

- A kimeneti értékek számítása a mellékelt táblázatok szerint történik.
- Figyelembe vesszük a **határozatlan és a nagy-impedanciás bemenetek**et is.
- A HDL logikai kapuk mindig vagy alacsony vagy magas logikai szintet hoznak létre a kimenetükön (nincs köztes állapot), csak megtörténik, hogy **nem tudni, melyik logikai szint fog jelentkezni** - ezért vannak a táblázatban **x**-ek.

and	i1			
	0	1	x	z
i2 0	0	0	0	0
i2 1	0	1	x	x
i2 x	0	x	x	x
i2 z	0	x	x	x

nand	i1			
	0	1	x	z
i2 0	1	1	1	1
i2 1	1	0	x	x
i2 x	1	x	x	x
i2 z	1	x	x	x

or	i1			
	0	1	x	z
i2 0	0	1	x	x
i2 1	1	1	1	1
i2 x	x	1	x	x
i2 z	x	1	x	x

nor	i1			
	0	1	x	z
i2 0	1	0	x	x
i2 1	0	0	0	0
i2 x	x	0	x	x
i2 z	x	0	x	x

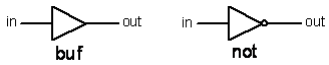
xor	i1			
	0	1	x	z
i2 0	0	1	x	x
i2 1	1	0	x	x
i2 x	x	x	x	x
i2 z	x	x	x	x

xnor	i1			
	0	1	x	z
i2 0	1	0	x	x
i2 1	0	1	x	x
i2 x	x	x	x	x
i2 z	x	x	x	x

50

8.1.2 EGYSZERŰ ILLESZTŐ ÁRAMKÖRÖK

- Egybites bemenet.
- Egy vagy több bites kimenet (elágazás)
- Invertáló vagy neminvertáló illesztő.



buf	
in	out
0	0
1	1
x	x
z	x

not	
in	out
0	1
1	0
x	x
z	x

```
// Egy kimenetű neminvertáló és invertáló elemek példányosítása.
buf b1(OUT, IN);
not n1(OUT, IN);
 // Több kimenetű elem példányosítása.
buf b1_3out(OUT1, OUT2, OUT3, IN)
// Invertáló illesztő példányosítása komponens név nélkül
//(engedélyezett módszer).
not(OUT1, OUT2, IN);
```

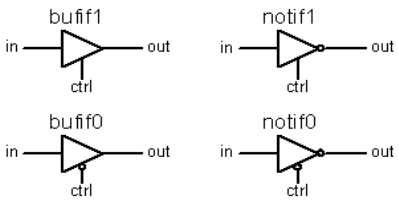
51

8.1.3 HÁROM ÁLLAPOTÚ ILLESZTŐK

- Egybites **adatbemenet**, plusz egy **vezérlőbemenet**.
- Egy vagy több bites kimenet (elágazás)
- Invertáló vagy neminvertáló funkció.

bufif1	ctrl			
	0	1	x	z
in 0	z	0	x	x
in 1	z	1	x	x
in x	z	x	x	x
in z	z	x	x	x

notif1	ctrl			
	0	1	x	z
in 0	z	1	x	x
in 1	z	0	x	x
in x	z	x	x	x
in z	z	x	x	x



bufif0	ctrl			
	0	1	x	z
in 0	0	z	x	x
in 1	1	z	x	x
in x	x	z	x	x
in z	x	z	x	x

notif0	ctrl			
	0	1	x	z
in 0	1	z	x	x
in 1	0	z	x	x
in x	x	z	x	x
in z	x	z	x	x

```
// Neminvertáló háromállapotú illesztők példányosítása.
bufif1 b1(out, in, control);          //komponens névvel
bufif0 (out, in, control);            // komponens név nélkül
 // Invertáló háromállapotú illesztők példányosítása.
notif1 n1(out, in, control);          // komponens névvel
notif0 (out, in, control);            // komponens név nélkül
```

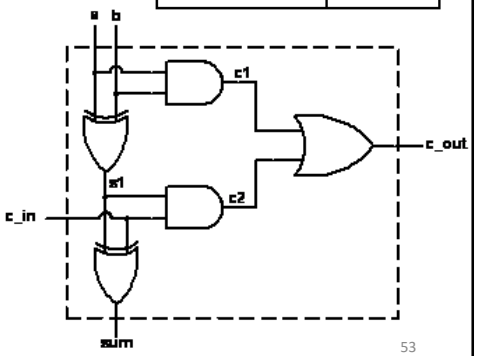
52

8.2.a PÉLDA KAPU SZINTŰ TERVEZÉSRE

- Négy bites teljes összeadó HDL leírása a cél.
- Először egybites összeadónak megfelelő modult definiálunk.
  $sum = (a \oplus b \oplus c\_in)$
  $c\_out = a \cdot b + c\_in \cdot (a \oplus b)$

c_in	a	b	sum	c_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
module FullAdder1bit(sum, c_out, a, b, c_in);
 // A port-ok deklarálása.
output sum, c_out;
input a, b, c_in;
 // A belső csomópontok deklarálása.
wire s1, c1, c2;
 // A logikai kapuk példányosítása (név
nélkül).
xor (s1, a, b);
xor (sum, s1, c_in);
 and (c1, a, b);
and (c2, s1, c_in);
 or (c_out, c1, c2);
 endmodule
```

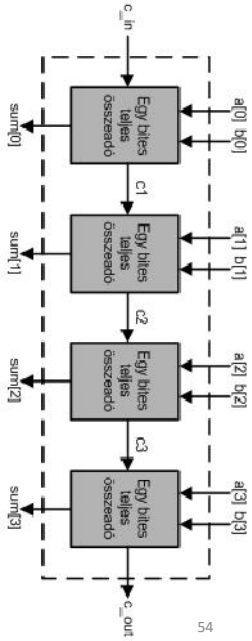


53

8.2.b PÉLDA KAPU SZINTŰ TERVEZÉSRE

A négy egybites teljes összeadó összekapcsolása.

```
module FullAdder4_bit(sum, c_out, a, b, c_in);
 // A port-ok deklarálása.
output [3:0] sum;
output c_out;
 input [3:0] a, b;
input c_in;
 // A belső csomópontok deklarálása.
wire c1, c2, c3;
 // A négy darab egy bites teljes összeadó
példányosítása.
FullAdder1bit fa0(sum[0], c1, a[0], b[0], c_in);
FullAdder1bit fa1(sum[1], c2, a[1], b[1], c1);
FullAdder1bit fa2(sum[2], c3, a[2], b[2], c2);
FullAdder1bit fa3(sum[3], c_out, a[3], b[3], c3);
 endmodule
```



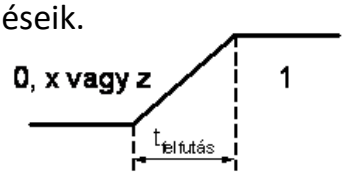
54

8.3.a A HDL KAPUK KÉSÉSEI

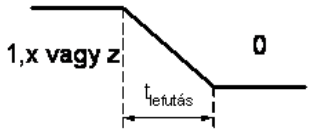
- A hardveres logikai kapuk késéseinek modellezése céljából a HDL kapuknak is lehetnek késéseik.

A késések esetei:

- **felfutási** késés



- **lefutási** késés



- **kikapcsolási** késés - átmenet a nagyimpedanciás állapotba.
- **határozott értékről határozatlanra** (x) váltás - nem adjuk meg külön, hanem az előző három idő közül a minimálisat veszi a szoftver.
- Rendszerint **relatív időket** adunk meg (nem konkrét mértékegység, pl. ns).

55

8.3.b A HDL KAPUK KÉSÉSEI - A NYELVI SZERKEZETEK

A lehetséges nyelvi szerkezetek:

```
// Minden késési idő egyenlő a delay_time paraméter értékével.
and #(delay_time) a1(out1, in1, in2);
// A felfutási és a lefutási időre különböző értékeket adunk meg.
or #(rise_value, fall_value) o1(out2, in1, in2);
// A felfutási-  a lefutási- és a kikapcsolási időre is különböző
//értékeket adunk meg.
bufif1 #(rise_val, fall_value, turn_off_value) b1(out3, in, control);
```

Konkrét esetek:

```
and #(5) a1(out1, in1, in2);      // Minden átmenet  5
                                  //időegységet vesz igénybe.
and #(4,6) a2(out2, in1, in2);  // felfutási idő = 4, lefutási idő = 6.
bufif0 #(3,4,5) b1(out3, in1, in2);      // felfutási idő = 3, lefutási
                                         // idő = 4, kikapcsolási idő = 5
```

8.3.1.a MINIMÁLIS, TIPIKUS ÉS MAXIMÁLIS ÉRTÉKEK AZ EGYES KÉSÉSEKRE

- A hardveresen megvalósított kapuk késései között különbségek vannak, még adott gyártónak ugyanazon IC-jén belül is.
- A HDL leírásban adott késésre (pl. felfutás) három értéket adunk meg.
- A szimuláció során utasítjuk a szoftvert, hogy melyik értékkel dolgozzon (pl. minden kapunál vegye a minimális értéket).
- Ha egy késésre csak egy idő van megadva, az a tipikus érték.

57

8.3.1.b MINIMÁLIS, TIPIKUS ÉS MAXIMÁLIS ÉRTÉKEK AZ EGYES KÉSÉSEKRE - PÉLDÁK

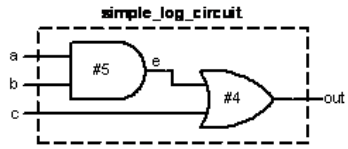
```
and #(4:5:6) a1(out1, i1, i2);
// A  fenti esetben a felfutási-, lefutási- és kikapcsolási idők egyenlők:
// Min = 4, Tip = 5, Max = 6.
 and #(3:4:5, 5:6:7) a2(out2, in1, in2);
// A felfutási idők:                Min = 3, Tip = 4, Max = 5
// A lefutási idők:                 Min = 5, Tip = 6, Max = 7
// A kikapcsolási idők              Min = 3, Tip = 4, Max = 5
// Emlékeztetünk, hogy kikapcsolási időként ez esetben a felfutási és
// a lefutási idők közül a kisebbet használjuk,
// mivel maga a kikapcsolási idő nincs megadva
 and #(2:3:4, 3:4:5, 4:5:6) a3(out3, in1, in2);
// Felfutási idők:                  Min = 2, Tip = 3, Max = 4
// Lefutási idők:                   Min = 3, Tip = 4, Max = 5
// Kikapcsolási idők:               Min = 4, Tip = 5, Max = 6
```

58

8.3.2.a A KÉSÉSEK HATÁSÁNAK SZIMULÁLÁSA

Egyszerű kombinációs hálózat:

A hálózat viselkedését leíró modul (késéseket tartalmaz):




```
module simple_log_circuit(out, a, b, c);
output out;
input a, b, c;
wire e;
and #(5) a1(e, a, b);    // 5 időegységnyi késés az ÉS logikai kapunál.
or #(4) o1(out, e, c);   // 4 időegységnyi késés a VAGY logikai kapunál.
endmodule
```

59

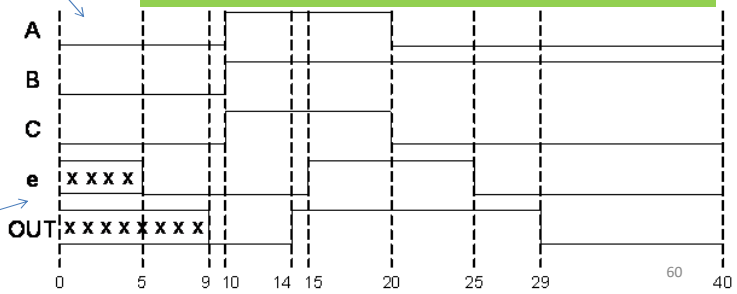
8.3.2.b A KÉSÉSEK HATÁSÁNAK SZIMULÁLÁSA

A gerjesztő modul:

A bemenő és kimenő jelek:

```
module stimulus;
reg A, B, C;
wire OUT;
simple_log_circuit slc1(OUT, A, B, C);
initial
begin
                A = 1'b0; B = 1'b0; C = 1'b0;
        #10     A = 1'b1; B = 1'b1; C = 1'b1;
        #10     A = 1'b0; B = 1'b1; C = 1'b0;
        #20     $finish;
end
endmodule
```

A késések letelte előtt a egyes logikai áramkörök kimenetei határozatlanok (x).



60

9. ADATFOLYAM SZINTŰ HDL LEÍRÁS (DATAFLOW MODELING)

- Magasabb (harmadik) szint.
- A tervező az adatokon elvégzendő műveletekre összpontosíthat (hatékonyabb).
- A végén itt is logikai áramkört kell kapni, de a szintézist szoftver végzi, nem ember.
- A logikai szintézisre ma hatékony szoftverek állnak rendelkezésre a PLD gyártóktól (rendszerint ingyenes) és független szoftverházaktól (fizetéses).
- A harmadik és a negyedik (viselkedési) szintű tervezés összefoglaló elnevezése: *RTL - register transfer level* tervezés.

61

9.1.a FOLYAMATOS HOZZÁRENDELÉS (CONTINUOUS ASSIGNMENT)

- A *net* típusú adathordozóknak adunk értéket.
- Hasonló a helyzet, mint a logikai kapuknál, csak könnyebb leírni a bonyolult műveleteket (magasabb szintű absztrakció).
- Minden folyamatos hozzárendelés az ***assign*** kulcsszóval kezdődik:

```
assign <késés> <hozzárendelés>;
```

- Késés megadása nem kötelező.

62

9.1.b A HOZZÁRENDELÉSEK TULAJDONSÁGAI

- A hozzárendelést egy kifejezéssel írjuk le.
- A kifejezés tartalmaz egy egyenlőségjelet.
- Az egyenlőségjel bal oldalára *net* típusú (skalár vagy vektor) adathordozó nevét írjuk, amelynek értéket akarunk adni (nem lehet ***reg***!).
- Az egyenlőségjel jobb oldalán *net* és/vagy ***reg*** típusú adathordozók és műveleti jelek kombinációjával adjuk meg, hogy mit kell hozzárendelni a bal oldalhoz.

```
// Folyamatos hozzárendelés.
//Az out, in1 és in2 adathordozók net típusúak.
assign out = in1 & in2;
// Folyamatos hozzárendelés net típusú adathordozó vektorokhoz.
// Az addr1_bits és addr2_bits adathordozók
//16 bites reg típusú vektorok.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
```

63

9.1.1. IMPLICIT FOLYAMATOS HOZZÁRENDELÉS

- Egy különleges (rövidített) mód a hozzárendelések írására.
- A hozzárendelést az adathordozó deklarálásakor végezzük.
- Nem használjuk az ***assign*** kulcsszót ebben a szerkezetben.

```
// A folyamatos hozzárendelés írásának szabályos módja
// (először deklaráljuk az adathordozót)
wire out;
assign out = in1 + in2;

// Ugyanazt az eredményt érjük el implicit folyamatos
hozzárendeléssel.
wire out = in1 + in2;
```

64

9.2 KÉSÉSEK A HOZZÁRENDELÉSEKBEN

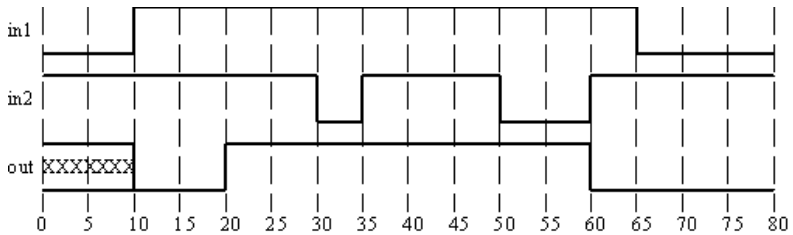
Három módszer a késések megadására:

1. A szabályos hozzárendelésekben (***assign*** kulcsszó mellett)
2. Az implicit hozzárendelésekben.
3. A *net* típusú adathordozók deklarálásakor.

65

9.2.1 KÉSÉSEK A SZABÁLYOS HOZZÁRENDELÉSEKBEN

Példa: `assign #10 out = in1 & in2;`



Mit tapasztalunk?

- A szimuláció kezdetén, a késés idejére a kimenet határozatlan.
- A bemeneti változások a késés letelte után fejtik ki a hatásukat a kimeneten.
- A késésnél rövidebb ideig tartó megcsuklások (glitch) a bemeneti jeleken nincsennek hatással a kimenetre.

9.2.1 KÉSÉSEK AZ IMPLICIT HOZZÁRENDELÉSEKBEN

- Az adathordozó deklarálásakor adjuk meg a késést:

```
wire #10 out = in1 & in2;
 // A fenti kifejezés hatása ugyanaz, mintha
// külön deklarálnánk az adathordozókat és
// külön definiálnánk a folyamatos
// hozzárendelést a megfelelő késéssel.
wire out;
assign #10 out = in1 & in2;
```

67

9.2.3 KÉSÉSEK MEGADÁSA A net TÍPUSÚ ADATHORDOZÓK DEKLARÁLÁSAKOR

- A *net* típusú (***wire*** stb.) adathordozó deklarálásakor adjuk meg a késést.
- A hozzárendelés alkalmazásakor kiértékelt új logikai állapot nem azonnal kerül az adathordozóra, hanem a korábbi deklarációban megadott késés elmúltával.

```
wire #10 out;
assign out = in1 & in2;

// A fenti leírás értelme megegyezik a lentivel:
wire out;
assign #10 out = in1 & in2;
```

68

9.3 KIFEJEZÉSEK, MŰVELETI JELEK ÉS OPERANDUSOK

- Az adatfolyam szintű tervezésnél a tervezendő digitális áramkört matematikai kifejezésekkel írjuk le.
- A kifejezések műveleti jelekkel összekapcsolt operandusokból állnak.
- Három különböző kifejezés:

```
a ^ b
addr1[7:4] + addr2[7:4]
in1 | in2
```

69

9.3.1 OPERANDUSOK

- Az operandusok valamilyen adatok/ adathordozók (*net, **reg, integer, real***), konstansok és függvényértékek lehetnek (Verilog függvényekkel itt nem foglalkozunk).
- Vektorok esetében a vektor egy része is lehet operandus (nem csak az egész).

```
integer count, final_count;
final_count = count + 1;        // A count adathordozó integer típusú
real a, b, c;
c = a – b;                      // Az a és b adathordozók real típusúak.
reg [7:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1 [3:0] ^ reg2[3:0];
// A reg1[3:0] és reg2[3:0] operandusok a reg1 és a
//reg2 vektor jellegű változók részeit képezik.
```

70

9.3.2 MŰVELETI JELEK

- A műveletei jelekkel definiáljuk, hogy milyen műveletet kell elvégezni az operandusokon (adatokon).
- Példa:

```
d1 && d2          // A && egy bináris műveleti jel
                  // d1 és d2 operandusokkal.
!a[0]     // A ! egy unáris műveleti jel a[0] operandussal.
C >> 2    // A >> egy bináris műveleti jel C és 2 operandusokkal.
```

71

9.4 A MÜVELETEK/MŰVELETI JELEK FAJTÁI

A Verilog kifejezésekben előforduló műveleteket a következő osztályokba soroljuk:

- aritmetikai,
- logikai,
- viszonyító,
- egyenlőségi,
- bitre vonatkozó,
- redukáló,
- eltoló,
- összecsatoló és többszöröző,
- feltételes.

72

9.4.1 ARITMETIKAI MŰVELETEK/MŰVELETI JELEK

- A táblázatban minden művelet két operandusra vonatkozik (bináris).
- A + és a - vonatkozhat csak egy operandusra is (unáris).

MŰVELET	JEL	OP. SZ.
szorzás	*	2
osztás	/	2
összeadás	+	2
kivonás	-	2
modul sz. osztás	%	2

```
A = 4'b0011; B = 4'b0100;      // A és B reg típusú vektorok.
D = 6; E = 4;                  // D és E integer típusú adathordozók.
 A * B                 // A és B szorzása. Az eredmény  4'b1100.
D / E                  // D osztása E-vel. A hányados 1. A maradékot figyelmen kívül hagyjuk.
A + B                  // A és B összeadása.  Az összeg 4'b0111.
B – A                  // A kivonása B-ből.  A különbség 4'b0001.
 in1 = 4'b101x;
in2 = 4'b1011;
 sum = in1 + in2;      // Az összeg  4'bx lesz.
 // A modul szerinti osztási művelet eredménye a maradék.
13 % 3      // Az eredmény 1.
16 % 4      // Az eredmény 0.
```

73

9.4.2 LOGIKAI MŰVELETEK/MŰVELETI JELEK

- A bitek számától függetlenül az operandust logikai értéknek tekintjük (egy bit).
- Az eredmény mindig egy bit (0,1).
- Kifejezések is használhatók operandusként.

MŰVELET	JEL	OP. SZ.
ÉS	&&	2
VAGY	||	2
NEM	!	1

```
A = 3; B = 0;
A && B                         // Az eredmény: ( 1 && 0 ) = 0.
A || B                         // Az eredmény : (1 || 0 ) = 1.
!A                             // Az eredmény : NEM( 1 ) = 0.
!B                             // Az eredmény : NEM( 0 ) = 1.
 // Határozatlan operandusok alkalmazása.
A = 2'b0x; B = 2'b10;
A && B                         // Az eredmény: ( x && 1 ) = x.
 // Kifejezések használata operandusként.
(a == 1 ) && ( b == 3 )  // Az eredmény 1 ha mindkét kifejezés logikai értéke 1.
// Az erdemény 0, ha legalább az egyik kifejezés logikai értéke 0.
```

74

9.4.3 VISZONYÍTÓ MŰVELETEK/MŰVELETI JELEK

- Számértékeket hasonlítunk össze.
- Az eredmény egy bit (0, 1, x).
- Ha az operandusoknak akár egy bitje is x vagy z, az eredmény x.

MŰVELET	JEL	OP. SZ.
nagyobb	>	2
kisebb	<	2
nagyobb v. egyenlő	>=	2
kisebb v. egyenlő	<=	2

```
A = 4; B = 2;
X = 4'b1010; Y = 4'b1011; Z = 4'b1xxx;

A <= B; // Az eredmény 0.
A > B;  // Az eredmény 1.
Y <= X; // Az eredmény 0.
Y < Z;  // Az eredmény x.
```

75

9.4.4 EGYENLŐSÉGI MŰVELETEK/MŰVELETI JELEK

- Számértékeket hasonlítunk össze.
- Az eredmény egy bit (0, 1, x).
- Ha az operandusnak akár egy bitje is x vagy z, az eredmény x.
- a *case* egyenlőség/egyenlőtlenség esetében az eredmény nem lehet x.

MŰVELET	JEL	OP. SZ.
egyenlő	==	2
nem egyenlő (különböző)	!=	2
case egyenlő	===	2
case nem egyenlő	!==	2

```
A = 4; B = 3;
X = 4'B1010; Y = 4'b1101;
Z = 4'b1xxz; M = 4'b1xxz; N = 4'b1xxx;

A == B    // Az eredmény 0.
X != Y    // Az eredmény 1.
X == Z    // Az eredmény x.
Z === M   // Az eredmény 1 mert minden bit megegyezik, még az x és z értékűek is.
Z === N   // Az eredmény 0 mert a legkisebb helyi értékű bitek különböznek.
M !== N   // Az eredmény 1.
```

76

9.4.5 BITENKÉNT VÉGZETT LOGIKAI MŰVELETEK/MŰVELETI JELEK

- Lényeges eltérés a (közönséges) logikai műveletektől!
- A bit tagadásnál az operandus bitjeit egyenként negáljuk.
- A többi műveletnél a két operandus megfelelő bitjein páronként végezzük az előírt műveletet.

MŰVELET	JEL	OP. SZ.
bit tagadás	~	1
bit ÉS	&	2
bit VAGY	|	2
bit kizáró VAGY	^	2
bit kizáró NEM-VAGY	^~ vagy ~^	2

```
X = 4'b1010;   Y = 4'b1101    ;          Z = 4'b10x1;
 ~X       // Bitenkénti NEM művelet. Az eredmény 4'b0101.
X & Y     // Bitenkénti ÉS művelet. Az eredmény 4'b1000.
X | Y     // Bitenkénti VAGY művelet. Az eredmény 4'b1111.
X ^ Y     // Bitenkénti kizáró VAGY művelet. Az eredmény 4'b0111.
X ^~ Y    // Bitenkénti kizáró nem VAGY  művelet.
          // Az  eredmény 4'b1000.
X & Z     // Bitenkénti ÉS művelet. Az eredmény 4'b10x0.
```

77

9.4.6 REDUKÁLÓ MŰVELETEK/MŰVELETI JELEK

- Egyetlen operandus (unáris műveletek).
- Az operandus egymást követő bitjeire alkalmazzuk az előírt műveletet.
- A legnagyobb helyi értéktől indulunk ki.

MŰVELET	JEL	OP. SZ.
redukáló ÉS	&	1
redukáló NEM-ÉS	~&	1
redukáló VAGY	|	1
redukáló NEM-VAGY	~|	1
redukáló kizáró VAGY	^	1
redukáló kizáró NEM-VAGY	^~ vagy ~^	1

```
X = 4'b1010;
&X      // A kiértékelés a következő egyenlet szerint történik: 1 & 0 & 1 & 0 = 1'b0.
|X      // A kiértékelés a következő egyenlet szerint történik: 1 | 0 | 1 | 0 = 1'b1.
^X      // A kiértékelés a következő egyenlet szerint történik: 1 ^ 0 ^ 1 ^ 0 = 1'b0.
 // A kizáró VAGY és  a kizáró NEM-VAGY műveletek felhasználhatók adott vektor
// párossági vagy páratlansági bitjének képzésére.
```

78

9.4.7 ELTOLÓ MŰVELETEK/MŰVELETI JELEK

- Egy vektor bitjeit toljuk jobbra vagy balra a megfelelő számú hellyel.
- Az első operandus a vektor, a második az eltolás mértéke.
- A megüresedett helyekre 0 íródik, a kitolt bitek elvesznek.

MŰVELET	JEL	OP. SZ.
jobbra tolás	>>	2
balra tolás	<<	2

```
X = 4'b1100;
Y = X >> 1;  // Y = 4'b0110. Az X vektor tartalma egy hellyel
// jobbra tolódik és a legnagyobb helyi értékű helyre nulla íródik be.
Y = X << 1;  // Y = 4'b1000. Az X vektor tartalma egy hellyel balra tolódik és
// a legkisebb helyi értékű helyre nulla íródik be.
Y = X << 2;  // Y = 4'b0000. Az X vektor tartalma két hellyel balra tolódik és
// a két legkisebb helyi értékű helyre nulla íródik be.
```

79

9.4.8 ÖSSZECSATOLÓ MŰVELET/MŰVELETI JEL

- Két vagy több operandust (vektort vagy skalárt) egyesít.
- Az operandusként használt vektorok hossza előre deklarálva van.
- Vektorok részei is lehetnek operandusok.

MŰVELET	JEL	OP. SZ.
összecsatolás	{}	tetszőleges számú

```
A = 1'b1;        B = 2'b00;        C = 2'b10;        D = 3'b110;

Y = { B, C };                      // Y = 4'b0010
Y = { A, B, C, D, 3'b001 };        // Y = 11'b10010110001
Y = { A, B[0], C[1] };             // Y = 3'b101
```

80

9.4.9 TÖBBSZÖRÖZŐ MŰVELET/MŰVELETI JEL

- Az első operandus egy konstans (szám). Ez határozza meg, hogy a második operandust hányszor kell egymás után írni.
- A többszörözés kombinálható az összecsatolással.

MŰVELET	JEL	OP. SZ.
többszörözés	{{}}	2

```
reg A;
reg [1:0] B, C;
A = 1'b1;        B = 2'b00;        C = 2'b10;

Y = { 4{A} };                      // Y = 4'b1111
Y = { 4{A}, 2{B} };                // Y = 8'b11110000
Y = { 4{A}, 2{B}, C };             // Y = 10'b1111000010
```

81

9.4.10. FELTÉTELES MŰVELET/MŰVELETI JEL

- Három operandus.
- A szintaxis:

MŰVELET	JEL	OP. SZ.
feltételes	?:	3

feltétel ? igaz feltételhez kötött kifejezés : hamis feltételhez kötött kifejezés;

- Ha a feltétel logikai értéke 1, az igaz feltételhez kötött kifejezés értékelődik ki, ellenkező esetben a hamis.
- Ha a feltétel határozatlan (x), a két kifejezés megegyező bitjei-, ellenkező esetben x-ek kerülnek a megfelelő helyekre.
- Úgy működik, mint egy 2/1-es multiplexer, de alkalmas háromállapotú illesztő megvalósítására is.

```
// 2/1-es multiplexer megvalósítása feltételes utasítással.
assign out = control ? in1 : in0;
 // Három állapotú illesztő megvalósítása feltételes utasítással.
assign addr_bus = drive_enable ? addr_out : 36'bz;
reg [1:0] A, B;  // A és B két bites reg típusú adathordozók.
 assign out = A[0] ? ( B[0] ? 1'b1 : 1'b0 ) : ( B[0] ? 1'b0 : 1'b1);
// Rekurzív módon alkalmaztuk a feltételes utasítást.
```

82

9.4.11 A MŰVELETEK HIERARCHIÁJA

- Tanácsos zárójelekkel rögzíteni a műveletek elvégzésének sorrendjét.
- Egyébként a táblázatban megadott sorrend az érvényes:

MŰVELETEK TÍPUSA	MŰVELETI JELEK	PRIORITÁS
unáris műveletek, szorzás, osztás, modul szerinti osztás	+, -, !, ~, *, /, %	Legmagasabb
összeadás, kivonás, eltolás	+, -, <<, >>	
viszonyító műveletek egyenlőségi műveletek	< ,<=, >, >= ==, !=, ===, !==	
redukáló műveletek logikai műveletek	&, ~& ^, ~^ |, ~| &&, ||	
feltételes műveletek	?:	Legalacsonyabb

83

10. VISELKEDÉSI SZINTŰ *HDL* LEÍRÁS (BEHAVIORAL MODELING)

- A legmagasabb (negyedik) elvonatkoztatási szint.
- Algoritmus jellegű, az áramkör viselkedését írja le, nem mond semmit a megvalósítás módjáról.
- A logikai szintézist (áramköri megoldás kialakítása) nem a tervező végzi, hanem megfelelő szoftver.
- A szimulációt végző (gerjesztő) modulokat általában viselkedési szinten írjuk le.

84

10.1. SZERKESZTETT ELJÁRÁSOK (STRUCTURED PROCEDURES)

Két fajta szerkesztett eljárás keretében írhatók a viselkedési szintű leírások. A megfelelő kulcsszavak alapján ezeket:

1. ***initial*** és
2. ***always***

eljárásoknak nevezzük.

- Egy modul tartalmazhat több szerkesztett eljárást is.
- Nem kezdődhet új eljárás leírása, amíg az előzőt be nem fejeztük.
- A szimuláció során és a megvalósított áramkörben az eljárások mind t=0 időben indulnak és konkurrens módon (párhuzamosan) hajtódnak végre.

85

10.1.1.a AZ ***initial*** ELJÁRÁS

Nyelvi szerkezet:

initial kulcsszó + hozzárendelés(eke)t tartalmazó blokk.

- Egy modul tartalmazhat több ***initial*** eljárást is.
- Mindegyik eljárás *t=0*-ban indul és egymástól függetlenül hajtódik végre és fejeződik be.
- Szerep: egyszeri beállítások a digitális áramkörökben.
- Leginkább szimulációs modulok leírásában használjuk (nem áramkör fejlesztése).
- Egy hozzárendelés: közvetlenül a kulcsszó után írandó.
- Több hozzárendelés: ***begin*** és ***end*** kulcsszavak közé írandó.

86

10.1.1.b AZ *initial* ELJÁRÁS - PÉLDÁK

- Több ***initial*** eljárást tartalmazó modul:

```
module Stimulus;
reg a, b, m;
initial
        m=1'b0;  // Egy hozzárendelés esetén nincs szükség a begin és end  kulcsszavakra.
initial
begin
        #5 a=1'b1;          // Két hozzárendelés esetén már szükséges a begin és end
        #25 b=1'b0;         // kulcsszavak használata. A késésekkel később foglalkozunk.
end
initial
        #50 $finish;        // A szimuláció végét definiáló initial eljárás
                            // (egy utasítást tartalmaz).
endmodule
```

- Minhárom eljárás t=0-ban indul. A hozzárendelések tényleges elvégzési idejei a táblázatban láthatók:

IDŐ	HOZZÁRENDELÉS
0	m=1'b0
5	a=1'b1
30	b=1'b0
50	$finish

87

10.1. 2 AZ *always* ELJÁRÁS

Nyelvi szerkezet:
always kulcsszó + hozzárendelés(eke)t tartalmazó blokk.

- A megadott hozzárendelések ciklikusan ismétlődnek.
- Itt is használatosak a ***begin*** és az ***end*** kulcsszavak.
- Példa: órajel definiálása ***always*** eljárással (szimulációt leíró modul).

```
module clock_gen;
reg clock;
initial
        clock=1'b0;      // Induljon az órajel alacsony logikai szintről.
always
        #10 clock=~clock;        // Minden 10 időegység után  invertáljuk
                                 // clock logikai értékét.
initial
        #1000 $finish;   // Az órajel impulzusok végét definiáló initial eljárás.
endmodule
```

88

10.2 AZ ***initial*** és ***always*** ELJÁRÁSOKBAN SZEREPLŐ HOZZÁRENDELÉSEK

- A viselkedési szintű hozzárendelésekben új értékeket rendelünk a ***reg***, ***integer*** és ***real*** típusú adathordozókhoz.
- Az adat nem változik, amíg nem alkalmazunk újabb hozzárendelést (memória jellegű viselkedés) - lényegi eltérés az adatfolyam szintű leírásokban szereplő hozzárendelésektől.
- A viselkedési szintű hozzárendelést végző nyelvi szerkezet:

<adathordozó neve> <hozzárendelés jele><kifejezés>

- Két típus: **blokkoló** és **nem blokkoló** hozzárendelések.

89

10.2.1.a BLOKKOLÓ HOZZÁRENDELÉSEK

- A blokkoló hozzárendelés jele: **=** (egyenlőség jel).
- A hozzárendelések abban a sorrendben végződnek el, ahogyan fel vannak tüntetve az adott (***initial*** ill. ***always***) eljárásban - a soron levő hozzárendelés blokkolja az utána következőket.
- A külön (***initial*** ill. ***always***) eljárásokban szereplő hozzárendelések nem blokkolják egymást, egymástól függetlenül kerülnek elvégzésre.

90

10.2.1.b BLOKKOLÓ HOZZÁRENDELÉSEK

- Példa: blokkoló hozzárendeléseket tartalmazó modul-részlet:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
        x=1'b0; y=1'b1; z=1'b1;
        count=0;
        reg_a=16'b0; reg_b=reg_a;
        #15 reg_a[2]=1'b1;
        #10 reg_b[15:13]={x,y,z};
        count=count+1;
end
```

SORREND	ELVÉGZÉS IDEJE	HOZZÁRENDELÉS
1	0	x=1'b0
2	0	y=1'b1
3	0	z=1'b1
4	0	count=0
5	0	reg_a=1'b0
6	0	reg_b=reg_a
7	15	reg_a[2]=1'b1
8	25	reg_b[15:13]={x,y,z}
9	25	count=count+1

91

10.2.2.a NEM BLOKKOLÓ HOZZÁRENDELÉSEK

- A nem blokkoló hozzárendelés jele: **<=** (kisebb jel és egyenlőség jel egymás mellett, ugyanaz mint a viszonyító jel).
- A soron levő hozzárendelés nem blokkolja a rákövetkező hozárendeléseket, egymástól függetlenül kerülnek elvégzésre.
- Ha nincs bejelölve késés, az adott hozzárendelés *t=0*-ban végződik el.
- A késések nem összegződnek.

92

10.2.2.b NEM BLOKKOLÓ HOZZÁRENDELÉSEK

- Példa blokkoló és nem blokkoló hozzárendelések vegyes alkalmazására:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial
begin
        x=1'b0; y=1'b; x=1'b1;
        count=0;
        reg_a=16'b0; reg_b=reg_a;
        reg_a[2]<= #15 1'b1;
        reg_b[15:13]<= #10 {x,y,z};
        count<=count+1;
end
```

SORREND	ELVÉGZÉS IDEJE	HOZZÁRENDELÉS
1	0	x=1'b0
2	0	y=1'b1
3	0	z=1'b1
4	0	count=0
5	0	reg_a=1'b0
6	0	reg_b=reg_a
7	15	reg_a[2]<=1'b1
8	10	reg_b[15:13]<={x,y,z}
9	0	count<=count+1

93

10.2.2.c NEM BLOKKOLÓ HOZZÁRENDELÉSEK

- A nem blokkoló hozzárendelések meg tudják akadályozni a **versenyfutási jelenségeket** (ez történik, ha az adatok értéke a műveletek sorrendjétől függ, ugyanakkor a sorrend bizonytalan).
- A módszer: első lépésként ideiglenes (köztes) tárolás történik, második lépésben a tárolt értéket használja fel az áramkör.
- Ez a leírás hasonlít legjobban a digitális áramköröknél alkalmazott élvezérlésre.
- Példa:

```
always @ (posedge clock)
begin
        reg1<=#1 in1;
        reg2<=@(negedge clock) in2^in3;
        reg3<=#1 reg1;
end
```

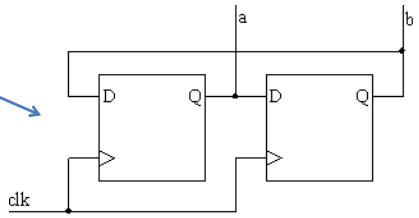
- A kifejezések jobb oldalain szereplő változók értékei átmenetileg megjegyződnek, a kifejezések kiértékelődnek.
- A hozzárendelések vagy mindjárt ez után történnek vagy amikor eljön az ideje (késés, órajel éle).
- A regiszterekbe való beírás sorrendje lényegtelen, mert nem a pillanatnyi, hanem a korábban memorizált érték íródik be.

94

10.2.2.d KÜLÖNBSÉG A BLOKKOLÓ ÉS A NEM BLOKKOLÓ HOZZÁRENDELÉSEK KÖZÖTT

- Nem blokkoló hozzárendelésekkel az adathordozók értékei kicserélődnek az órajel minden felfutó élénél.

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



- Blokkoló hozzárendelésekkel mindkét regiszterbe ugyanaz az érték kerül (nem tudni, melyik).

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

95

10.3 A HOZZÁRENDELÉSEK IDŐZÍTÉSE A VISELKEDÉSI SZINTŰ LEÍRÁSOKBAN

- Ha nincs időzítés, a szimuláció során nincs előrehaladás az időskálán.
- A valósághű leírás megköveteli az időzítéseket.
- Az időzítés eszközei:
  1. késések,
  2. élvezérlés,
  3. szintvezérlés.

96

10.3.1.a IDŐZÍTÉS KÉSÉSEK MEGADÁSÁVAL

- Késés: idő a hozzárendelés sorrakerülése és végrehajtása között.
- A késést megadó nyelvi szerkezet:

  # késés,

  ahol a késés írható számérték, azonosító vagy min/typ/max kifejezés formájában.
- Példa a késések írásának **szabályos módjára**:

```
initial
begin
        #10 y=1'b1;
        #latency z=1'b0;
        #(4,5,6) q=1'b0;
end
```

97

10.3.1.b IDŐZÍTÉS KÉSÉSEK MEGADÁSÁVAL

Késés megadása **hozzárendelésen belül**:

- A késés értéke a hozzárendelési jel jobb oldalára íródik.
- A jobb oldalon szereplő kifejezés a hozzárendelés sorrakerülésének pillanatában kiértékelődik (a szabályosan írott késésnél a kiértékelés is késik).
- A hozzárendelés akkor következik be, amikor letelik a késés.

Késés megadása a hozzárendelésen belül. →

Ugyanaz az eredmény köztes tárolással és szabályosan írott késéssel. →

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

initial
begin
        temp_xz=x+z
        #5 y=temp_xz
end
```

98

10.3.2.a IDŐZÍTÉS ÉLVEZÉRLÉSSEL ÉS SZINTVEZÉRLÉSSEL

- A mai digitális berendezések többsége szinkron hálózat (órajelet igényel).
- A szinkronizálás történhet a vezérlőjel megfelelő logikai szintjével vagy a megfelelő élével.
- Az élvezérlés az elterjedtebb.
- Módszerek:
  1. szabályos élvezérlés,
  2. nevezett eseményre való várakozás,
  3. több jel egyikére való várakozás,
  4. várakozás megfelelő logikai szintre.

FIGYELEM: A HARDVERNYELVI LEÍRÁSOKBAN NINCS KÉSZ ÓRAJEL FORRÁS, A TERVEZŐ KELL, HOGY LÉTREHOZZA AKÁR HARDVERT LEÍRÓ MODULRÓL-, AKÁR SZIMULÁCIÓS MODULRÓL VAN SZÓ.

99

10.3.2.b SZABÁLYOS ÉLVEZÉRLÉS LEÍRÁSA

- Az **@** kulcsszóval, valamint a ***posedge*** és ***negedge*** kulcsszavakkal definiáljuk.
- Magas logikai szintnél végzi a hozzárendelést.
- Felfutó élre reagál.
- Lefutó élre reagál.
- Memorizáljuk *d* értékét a sorrakerüléskor, beírás az órajel felfutó élénél.

```
always @ (clock)
        q=d;
always @ (posedge clock)
        q=d;
always @ (negedge clock)
        q=d;
always
        q=@(posedge clock) d;
```

100

10.3.2.c IDŐZÍTÉS NEVEZETT ESEMÉNYRE VALÓ VÁRAKOZÁSSAL

1. Deklarálunk egy eseményt (***event*** kulcsszóval).
2. Definiáljuk az eseményt.
3. Elvégezzük az eseményre várakozó hozzárendelést.

```
event received_data;

always @(posedge clock)
begin
        if (last_data_packet)
                ->received_data;
end

always @ (received_data)
data_buf={data_pkt[0], data_pkt[1],
data_pkt[2], data_pkt[3]};
```

101

10.3.2.d TÖBB JEL EGYIKÉRE VALÓ VÁRAKOZÁS - 1995-ÖS SZABVÁNY SZERINT

- A szabályos módszerrel definiált élvezérlés kiterjesztése.
- Több jel közül az egyik megváltozása (éle!) indítja a hozzárendelést.
- Az egyes jelek nevei közé ***or*** kulcsszót írunk.

```
always @(reset or clock or d)
begin
        if (reset)
                q=1'b0;
        else    if(clock)
                q=d;
end
```

- A zárójeles részt érzékenységi listának nevezzük.
- Ez a leírás alapján szintvezérelt áramkör (D-latch) szintetizálódik.
- A **szintvezérelt** áramköröknél is a vezérlőjelek **él**ét követően történnek a változások (mert ekkor alakul ki megfelelő logikai szint), mégsem beszélünk élvezérlésről.

102

10.3.2.e TÖBB JEL EGYIKÉRE VALÓ VÁRAKOZÁS - 2001-ES SZABVÁNY SZERINT

- A zárójelben megadott érzékenységi lista (mikor kell elvégezni az **always** blokkban felírt műveleteket), írható egyszerű felsorolásként, **vesszőkkel elválasztva**.
- Ha kombinációs hálózatot szeretnénk szintetizálni **always** eljárás segítségével, akkor használható az **@(*)** jelölés.
- Jelentése: ha bármely bemenő változó megváltozik, végezd el a műveleteket.

```
always @(reset, clock, d)
begin
        if (reset)
                q=1'b0;
        else    if(clock)
                q=d;
end
```

```
always @(*)
begin
eSeg = 1;
if(~A & D) eSeg = 0;
if(~A & B & ~C) eSeg = 0;
if(~B & ~C & D) eSeg = 0;
end
```

103

10.3.2.f VÁRAKOZÁS MEGFELELŐ LOGIKAI SZINTRE (SZINTVEZÉRLÉS)

- A ***wait*** kulcsszóval definiáljuk.
- A hozzárendelés akkor történik meg, amikor a feltétel magas logikai szinten van.
- A példában minden 20 időegységnyi késést követően történik egy inkrementálás, feltéve, hogy a *count_enable* magas logikai szintjével engedélyeztük.

```
always
        wait (count_enable) #20 count=count+1;
```

104

10.4.a FELTÉTELHEZ KÖTÖTT HOZZÁRENDELÉSEK

- Az ***initial*** és ***always*** eljárásokban a hozzárendelés végrehajtását feltételhez lehet kötni.
- A feltételhez kötést az ***if*** és az ***else*** kulcsszavakkal végezzük.
- Három eset:
  1. Csak ***if*** kulcsszót használunk.
  2. Az ***if*** és az ***else*** kulcsszavakat használjuk.
  3. Az ***if – else if – else*** láncolatot használjuk.

105

10.4.b FELTÉTELHEZ KÖTÖTT HOZZÁRENDELÉSEK - CSAK ***if*** KULCSSZÓT HASZNÁLUNK

- Csak akkor történik hozzárendelés, ha igaz bizonyos feltétel.

```
if (!clock)        buffer=data;
```

106

10.4.c FELTÉTELHEZ KÖTÖTT HOZZÁRENDELÉSEK - AZ *if* ÉS AZ *else* KULCSSZAVAKAT HASZNÁLJUK

- Ezek a hozzárendelések akkor hajtódnak végre, ha igaz az ***if*** kulcsszó melletti feltétel.

```
if (number_queued<MAX_Q_DEPTH)
begin
        data_queued=data;
        number_queued= number_queued+1;
end
else
        $display ("Queue full, Try again");
```

- Az ***else*** kulcsszót követő hozzárendelést/ parancsot akkor végzi el, ha a feltétel nem igaz.

107

10.4.d FELTÉTELHEZ KÖTÖTT HOZZÁRENDELÉSEK - AZ *if – else if – else* LÁNCOLATOT HASZNÁLJUK

- A feltételnek **több értéke** lehet.
- A feltétel különböző értékeire **különböző műveleteket** kell elvégezni.

```
if (alu_control==0)
        y=x+z;
else if (alu_control==1)
        y=x-z;
else if (alu_control==2)
        y=x*z;
else
        $diplay ("Invalid alu control signal")
```

108

10.5 TÖBBFELÉ TÖRTÉNŐ ELÁGAZÁSOK

- Az ***if-else*** szerkezetek nem célszerűek, ha a feltételnek sok különböző értéke lehet (nem áttekinthető a leírás).
- Helyette ***case*** szerkezeteket használunk. Az itt használatos kulcsszavak: ***case, casex, casez, endcase, default.***

109

10.5.1 A *case* SZERKEZETEK

- Három felé lehet elágazni.
- Ha egyik feltétel sem teljesül. A ***default*** csak egyszer írható egy szerkezetben!
- 4/1-es multiplexer megvalósítása ***case*** szerkezettel.

```
reg [1:0] alu_control;
 case (alu_control)
        2'd0: y=x+z;
        2'd1: y=x-z;
        2'd2: y=x*z;
        default: $display ("Invalid alu control signal");
endcase
```

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
        2'd0: out=i0;
        2'd1: out=i1;
        2'd2: out=i2;
        2'd3: out=i3;
        default: $display ("Invalid control signals");
endcase
endmodule
```

110

10.5.2 A *casex* ÉS A *casez* KULCSSZAVAK HASZNÁLATA

- A ***case*** szerkezetek rövidítése céljából használjuk a ***casex*** és ***casez*** kulcsszavakat.
- ***casez*** használata esetén a feltétel *z*-vel jelölt bitje felvehet bármilyen értéket, nem lesz kihatással az elágazásra.
- ***casex*** használatakor a feltétel x-szel jelölt helyén állhat akár x, akár z, nem vesszük figyelembe a hozzárendelés végrehajtásakor.

```
reg[3:0] encoding;
integer state;
casex (encoding)
        4'b1xxx: state=3;
        4'bx1xx: state=2;
        4'bxx1x: state=1;
        4'bxxx1: state=0;
        default:  state=0;
endcase
```

111

10.6 HURKOK A VISELKEDÉSI SZINTŰ LEÍRÁSOKBAN

- Az ***always*** eljárásokban a hozzárendelések állandóan ismétlődnek.
- Hurkok definiálásával a hozzárendelések ismétlése **feltételhez köthető**.
- A hurkok definiálására használt kulcsszavak: ***while, for, repeat, forever***.

112

10.6.1 A *while* TÍPUSÚ HUROK

- A nyelvi szerkezet:
  while (kifejezés)
- A hurokban szereplő hozzárendelések **addig ismétlődnek, amíg a zárójelben levő kifejezés (feltétel) igaz**.

```
integer count;
initial
begin
        count=0;
        while (count < 127)
        begin
                $display ("count=%d", count);
                count=count+1;
        end
end
```

113

10.6.2 A *for* TÍPUSÚ HUROK

- A nyelvi szerkezet: ***for*** (kifejezés)
- A zárójelben levő kifejezés **három dolgot** tartalmaz:
  1. kezdeti feltétel (érték),
  2. a befejezési feltétel vizsgálata,
  3. a befejezési feltételben vizsgált érték módosítása.
- **Elsősorban számláló** leírására alkalmas, de ismétlődő kombinációs hálózati elemekhez is jó.

```
integer count;
initial
        for (count=0; count<128; count=count+1)
                $display ("count=%d", count);
```

114

10.6.3 A *repeat* TÍPUSÚ HUROK

- A nyelvi szerkezet: ***repeat*** (számérték)
- Akkor alkalmazzuk, ha **előre tudjuk, hányszor kell elvégezni** a hurokban szereplő hozzárendelést.

```
integer count;
initial
begin
        count=0;
        repeat (128)
        begin
                $display (“count=%d”, count);
                count=count+1;
        end
end
```

115

10.6.4 A *forever* TÍPUSÚ HUROK

- A nyelvi szerkezet: ***forever***
- Nincs feltétel.
- Elvileg **végtelenül ismétlődik** ($finish-sel állítható le).
- Alkalmas pl. órajel generálására.

```
initial
begin
        clock=1’b0;
        forever #10 clock=~clock;
end
```

116

10.7 SOROS ÉS PÁRHUZAMOS BLOKKOK

- A blokkok **hozzárendelések csoportosítására** szolgálnak.
- **Soros** blokk: ***begin, end*** kulcsszavakkal definiáljuk. A hozzárendelések a felírás sorrendjében hajtódnak végre.
- **Párhuzamos** blokk: ***fork, join*** kulcsszavakkal definiáljuk. A hozzárendelések párhuzamosan (egy időben) kerülnek sorra. A végrehajtás időpontja késésekkel befolyásolható.

117

10.7.1 SOROS BLOKKOK

- A hozzárendelések a **felírás sorrendjében** kerülnek sorra.
- Nem indul új hozzárendelés, amíg az előzőt végre nem hajtottuk.
- Az egyes hozzárendeléseknél megadott **késések relatívak** (összegződnek), a megelőző hozzárendelés végrehajtásától számoljuk.

Késés nélkül

```
initial
begin
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
end
```

Késéssel

```
reg x,y;
reg [1:0] z, w;
initial
begin
        x=1'b0;
        #5 y=1'b1;
        #10 z={x,y};
        #20 w={y,x};
end
```

118

10.7.2 PÁRHUZAMOS BLOKKOK

- A hozzárendelések **egyidőben** kerülnek sorra, mint a nem blokkoló hozzárendeléseknél, de **előzőleg nem memorizálódnak az értékek** a blokkba való belépés pillanatában.
- A feltüntetett késések abszolút értékek (nem adódnak össze).
- Versenyfutási jelenség lép fel, ha ugyanabban a blokkban módosítjuk is az adatot és használjuk is más kifejezésben.

Késéssel

```
reg x,y;
reg [1:0] z, w;
initial
fork
        x=1'b0;
        #5 y=1'b1;
        #10 z={x,y};
        #20 w={y,x};
join
```

Késés nélkül - versenyfutási jelenség lép fel

```
reg x,y;
reg [1:0] z, w;
initial
fork
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
join
```

119

10.7.3 KOMBINÁLT BLOKKOK

- A soros és a párhuzamos blokkok kombinálhatók.
- A példában a soros blokk a fő blokk. Ennek a hozzárendelései sorjában hajtódnak végre.
- A párhuzamos blokk a soros blokkon belüli sorrakerüléskor hajtódik végre.

```
initial
begin
        x=1'b0;
        fork
                #5 y=1'b1;
                #10 z={x,y};
        join
        #20 w={y,x};
end
```

120

10.7.4 NEVEZETT BLOKKOK

- A blokkoknak nevet adhatunk.
- A nevezett blokkoknál:
  1. Helyi adathordozókat deklarálhatunk.
  2. Az egyes adathordozókat hierarchikus nevek segítségével elérhetjük.
  3. A végrehajtást megszakíthatjuk.

```
module top;

initial
begin: block1
integer i;
.....
end

initial
fork: block2
reg i;
.....
join
endmodule
```

Nem ugyanaz

121

10.7.4 NEVEZETT BLOKK MEGSZAKÍTÁSA

- A ***disable*** kulcsszóval a nevezett blokk végrehajtása megszakítható.
- A példában kilépünk a ***while*** hurokból, ha a *flag* vektor soron levő bitje egyes.

```
initial
begin
        flag=16'b0010_0000_0000_0000;
        i=0;
        begin: block1
        while (i<16)
                begin
                        if (flag[i])
                        begin
                                $display ("True bit at element number %d", i);
                                disable block1;
                        end
                        i=i+1;
                end
        end
```

122

Vége a III. résznek

Tervezés programozható logikai áramkörökkel (*PLD*) a *Verilog* hardvernyelv

123