Verilog ismertető

Szántó Péter
BME Méréstechnika és Információs Rendszerek Tanszék, FPGA Labor
2011-09-01

Tartalomjegyzék

1. Bevezetés

A digitális rendszerek komplexitásának növekedésével a kapuszintű modulokból építkező, kapcsolási rajzon alapuló tervezési módszerek elégtelenné váltak. A tervezői hatékonyság növelése valamint az egyszerűbb verifikáció igénye vezetett a hardver leíró nyelvek (hardware description language - HDL) kialakulásához. Az első, s máig legelterjedtebb két nyelv (Verilog és VHDL) első verziója 1983-1984 környékén jelent meg, azóta ezek ipari szabvánnyá váltak, s természetesen néhány frissítést is megéltek.

Ezen nyelvek közös jellemzője, hogy elsősorban hardver funkciók modellezésére és szimulációjára fejlesztették ki őket, implementációra történő tervezés esetén a

nyelvnek csak egy részhalmaza használható; hogy ez pontosan mely nyelvi elemeket jelenti, azt az implementációt végző szoftver (az ún. szintézer) határozza meg.

A HDL leírásból két irányba indulhatunk. A szintézer előállítja a cél-eszköznek megfelelő, optimalizált huzalozási listát, ami az adott eszköz ún. primitívjeiből (alapvető építőelem, pl. flip-flop) és az ezek közti kapcsolatokat leíró összeköttetésekből áll. Ugyancsak a HDL leírás az alapja a megfelelő funkcionalitás ellenőrzésének, azaz a szimulációnak.

2. Verilog nyelvi elemek

2.1. Modulok definiálása

A Verilog nyelv hierarchikus, funkcionális egységeken alapuló tervezői megközelítést alkalmaz. A teljes terv több, kisebb modulból áll össze, melyek részfunkciókat valósítanak meg. Egy-egy modul komplexitásának meghatározása a tervezőn múlik. Az FPGA-t felépítő speciális alapelemek (pl. LUT, flip-flop, szorzó) elérhetőek előre definiált modulként, ezeket nevezzük primitívnek. Megjegyzendő ugyanakkor, hogy megfelelő HDL kódból a szintézerek képesek ezen primitívek legnagyobb részét automatikusan alkalmazni.

Egy modul definiálása a *module* kulcsszóval történik, s a modul fejlécében kell megadni a modul ki- és bemeneti portjait. Az alábbi példa egy egybites, két bemenetű kapu modul deklarációját mutatja:

```
module gate_1(in0, in1, out);
       input  in0;
       input  in1;
       output out;

//a modul funkcionalitásának leírása

endmodule
```

A *module* kulcsszót a modul neve követi, majd a portok felsorolása. Ezt követően kell megadni, hogy az egyes portok ki- vagy bemenetei a modulnak, illetve azt hogy ezek milyen szélesek (hány bitesek). A következő példa egy nyolcbites kapu modul deklarációját mutatja, ennek értelemszerűen mindkét bemenete és kimenete is nyolcbites. Azaz a két bemeneti vektoron bitenként végez műveletet, amit úgy is megfogalmazhatunk, hogy nyolc darab egybites, két bemenetű kaput tartalmaz egy elemként.

```
module gate_8(in0, in1, out);
       input  [7:0] in0;
       input  [7:0] in1;
       output [7:0] out;

//a modul funkcionalitásának leírása

endmodule
```

A Verilog-2001 lehetőséget nyújt egy alternatív moduldeklarálási szintaxisra is, mely áttekinthetőbb, illetve kevesebb gépelést igényel. A második példa esetére:

```
module gate_8(
        input  [7:0] in0,
        input  [7:0] in1,
        output [7:0] out
);

//a modul funkcionalitásának leírása

endmodule
```

A hierarchikus megközelítés alkalmazására a modulok egymásba ágyazásával nyílik lehetőség, erre a későbbiekben látunk példát.

Egy port lehet bemenet (input), kimenet (output) vagy kétirányú port (inout). Utóbbi számos külső periféria (pl. PS/2, memória) illesztése esetén szükséges, használata csak ezekben az esetekben indokolt, belső modulok esetén kerülendő.

A modulok a hardver tényleges működésének megfelelően konkurensek, azaz a bennük leírt funkcionalitás párhuzamosan hajtódik végre.

2.2. Operátorok

Szintaktikailag a Verilog a C nyelvben megismert operátorokra épít, ugyanakkor nem lehet eléggé hangsúlyozni: az operátorok hasonlósága ellenére hardver leíró nyelvről van szó, így a működés nem azonos a szoftverrel (és más tervezői megközelítést igényel).

Megjegyzések, konstansok

A megjegyzések elhelyezésére a C nyelvben megismert szintaxis alkalmazható: egy soros esetben a //, míg több soros esetben a /* */ használható.
Konstansok megadása a <bitszám>'<számrendszer><konstans> szintakszissal történik. Például:

5'b00100: 5 bites bináris konstans, decimális értéke 4
8'h4e: 8 bites hexadecimális konstans, decimális értéke 78

Az adott számrendszerben megszokott karakterek mellett az „x" a nem definiált (illetve don't care) értéket jelöli, míg „z" a nagyimpedanciás állapot leírását teszi lehetővé (utóbbira a későbbiekben látunk példát).

Bitműveletek

Operátorok: *~, &, |, ^, ~^:* negálás, és, vagy, xor, nxor
Ha az operandusok vektorok, a bitműveletek bitenként hajtódnak végre. Ha a két operandus nem azonos szélességű, a kisebbik a nagyobb helyiértékű biteken 0.
Például:

4'b1101 & 4'b0110 = 4'b0100
2'b11 & 4'b1101 = 4'b 0011 & 4'b1101 = 4'b0001

Bitredukciós operátorok

Operátorok: &, ~&, |, ~|, ^, ~^: és, nem-és, vagy, nem-vagy, xor, nem-xor
Ezen operátoroknak egyetlen operandusa van, az operátor ennek bitjein végzi el a kijelölt műveletet. A kimenet egyetlen bit.
Például:

&4'b1101 = 1'b0
|4'b1101 = 1'b1

Komparátor operátorok

Ugyanaz, mint C-ben, tehát == (egyenlő), != (nem egyenlő), < (kisebb), > (nagyobb), <= (kisebb-egyenlő), >= (nagyobb-egyenlő).

Aritmetikai operátorok

Szintén a C szintakszissal megegyező: +, -, *, /, %
Az osztás (/) és a maradékképzés (%) művelet tipikusan csak akkor szintetizálható, ha a jobb oldali operandus 2 valamely hatványa.
A negatív számok ábrázolása kettes komplemens kódban történik, előjeles változókat a „signed" kulcsszóval deklarálhatunk, például:

```
reg signed [7:0] op_0;
```

Egyéb operátorok

Több bites vektor-változók esetén bitek, vagy bit-részletek kiválasztására a [] ad lehetőséget. Pl. a 8 bites op_1 változóból kiválaszthatunk egy 4 bites részt: op_1[4:1]. A []-ben szereplő érték konstans.
A konkatenálás ({}) operátor az operandusait fűzi egymás mellé, például ha op_0="0011" és op_1="10", akkor {op_1, op_0}="100011".

2.3. Változók, értékadások

A Verilog alapvetően kétféle változótípust tartalmaz: a *wire*-t és a *reg*-et. Előbbi nevének megfelelően szintézis után (szinte mindig) vezetékként viselkedik, míg utóbbi eredménye a leírás módjától függ.

A *wire* típusú változók az *assign* utasítással kaphatnak értéket, s az így leírt kifejezés folytonosan kiértékelődik. Minden *wire* típusú változó csak egy *assign* által kaphat értéket, az értékadás operátora az =.

A *reg* típusú változók ún. *always* blokkban kaphatnak értéket. Az *always* blokk szintaxisa a következő:

```
always @ <érzékenységi lista>
   értékadások
```

Az ún. érzékenységi lista határozza meg, hogy a blokkon belüli értékadások mikor értékelődnek ki. Néhány tipikus példa:

- *always @ (posedge clk)*: élérzékeny *always* blokk, a *clk* jel felfutó élére hajtódik végre
- *always @ (posedge clk, posedge rst)*: élérzékeny *always* blokk, a *clk* vagy a *rst* jel felfutó élére hajtódik végre (pl. aszinkron reset megvalósításához)
- *always @ (op_0, op_1)*: szintérzékeny *always* blokk, *op_0* vagy *op_1* jelek bármelyikének változásakor végrehajtódik.
- *always @ (*)*: ugyancsak szintérzékeny *always* blokk (csak Verilog-2001-ben), az összes, a blokkban bemenetként használt jelre érzékeny (az értékadások jobb oldalán szereplő jelek), tehát bármelyik változásakor kiértékelődik

A *reg* típusú változókra is igaz, hogy csak egyetlen *always* blokkban kaphatnak értéket. Az *assign*-nal ellentétben *always* blokkon belül két, különböző értékadási módot különböztetünk meg. A blokkoló értékadás (**=**) mindaddig blokkolja az öt követő értékadások kiértékelését, míg ő maga ki nem értékelődött, azaz a végrehajtás ebben az esetben szekvenciális. Ezzel szemben a nem-blokkoló értékadások (**<=**) az érzékenységi lista igazzá válásakor párhuzamosan értékelődnek ki, így végrehajtásuk nem függ a leírás sorrendjétől. A hardver működését a nem-blokkoló értékadás szemlélteti jobban, így ennek használata javasolt (e két értékadással a 2.3.4. fejezet még foglalkozik).

Egy modulon belül lehetőség van több *assign* és *always* létrehozására, ezek egymással konkurens működésűek (azaz az összes modul összes *always* blokkja és *assign* értékadása egymással párhuzamosan működik).

Összegezve a legfontosabbakat:
- Egy változó csak egyetlen értékadásban (egyetlen assign által vagy egyetlen always blokkban) kaphat értéket.
- Egy változó értéke tetszőleges számú kifejezésben vizsgálható (olvasható).
- Wire típusú változók csak és kizárólag assign értékadással kaphatnak értéket.
- Reg típusú változókat csak és kizárólag always blokkban szabad írni.
- Assign értékadással tipikusan kombinációs logikákat írhatunk le.
- Always blokkal lehetőség van kombinációs logika, latch és FF leírására is.

A Verilog nyelvben a nem deklarált változók automatikusan deklarálódnak, mégpedig 1 bites wire típusként. Ez „remek” hibalehetőségeket rejt magában, így a változó deklarációkra érdemes nagyon odafigyelni – vagy alternatívaként a Verilog file legelső sorába elhelyezni az alábbi direktívát (ami kikapcsolja az automatikus deklarációt):
`default_nettype none

2.3.1. Kombinációs logikák leírása

Kombinációs logikák leírása mind *assign*, mind pedig *always* segítségével történhet. Előbbi talán szemléletesebb és tömörebb (és használata javasolt), ugyanakkor utóbbit indokolja, hogy egyes nyelvi elemek (*if…else*, illetve *case*) csak *always* blokkon belül használhatók.

Az *assign*-nal történő leírás esetén egyetlen kivételtől eltekintve (lásd következő fejezet) minden esetben kombinációs logikához jutunk, például egy 8 bites összeadót leírhatunk a következőképp (az összeadó két bemeneti jelét már deklaráltnak tekintjük):

```
wire [7:0] sum;
assign sum = op_0 + op_1;
```

A kombinációs hurkok kialakítása tilos, ami nyelvi szinten azt jelenti, hogy az *assign* bal oldalán álló változó nem szerepelhet az értékadás jobb oldalán is (az implementációt végző szintézer a kombinációs hurkokra figyelmeztet).

Az *always* blokkal történő leíráshoz induljunk ki a kombinációs logika definíciójából, miszerint a kimeneti jelek értékei csak a bemeneti jelek pillanatnyi értékétől függenek. Ebből következik, hogy a megfelelő funkcionalitást megvalósító *always* blokknak bármely bemenet megváltozásakor ki kell értékelnie a benne szereplő kifejezéseket – azaz az érzékenységi lista minden, a blokkban bemenetként használt jelre érzékeny. Például az előző összeadó leírása a következőképp történhet.

```
reg [7:0] sum;
always @ (op_0, op_1)
   sum <= op_0 + op_1;
```

Vagy ezzel ekvivalens (a ’*’ operátor csak a Verilog-2001-ben értelmezhető):

```
reg [7:0] sum;
always @ ( * )
   sum <= op_0 + op_1;
```

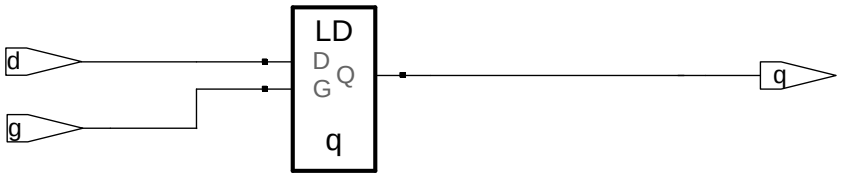
2.3.2. Latch leírása

A latch egy szintérzékeny tároló elem, mely a Gate bemenetének magas állapota alatt mintavételezi, illetve a kimenetén megjeleníti az adatbemenetén található értéket (transzparensen viselkedik), míg Gate jelének alacsony állapotában a mintavételezett értéket tartja. A működésnek megfelelő Verilog leírás mind *assign* mind pedig *always* felhasználásával elképzelhető:

```
wire q;
assign q = (g==1) ? d : q;
```

A fenti kód esetében a q jel megkapja a d értékét, amennyiben a g 1 értékű (→latch transzparens üzemmódja). Ellenkező esetben a q jel a q értékét kapja, azaz nem változik.
Ennél némileg szemléletesebb az always blokkal történő leírás.

```
reg q;
always @ ( * )
if (g)
   q <= d;
```



A latch-ek használata esetén problémát jelenthet a Gate jel esetleges hazárdossága, illetve az, hogy a Gate magas állapota alatt a kimenet követi a bemenet változásait (nem stabil). FPGA-ra történő fejlesztés esetén a latch használata kerülendő, a legtöbb szintézer figyelmeztet is latch észlelése esetén (Warning).

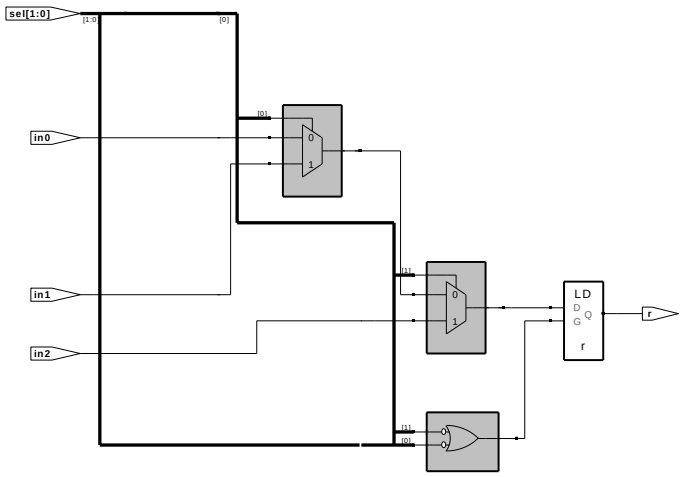
Tipikus hibák, melyek latch alkalmazásához vezetnek, miközben a cél kombinációs logika leírása:

- nem teljesen kifejtett case szerkezet
- if…else használata esetén hiányzik a feltétel nélküli else ág

Például egy 3:1 multiplexer implementálásakor könnyedén hibás leíráshoz juthatunk. Ahhoz, hogy a három bemenet közül választani lehessen 2 bites kiválasztó jelre van szükség – ez maga után vonja, hogy lesz egy olyan kiválasztó jel kombináció, amely a rendszer üzemszerű működése során nem fordulhat elő (az alábbi példában a 2'b11=3).

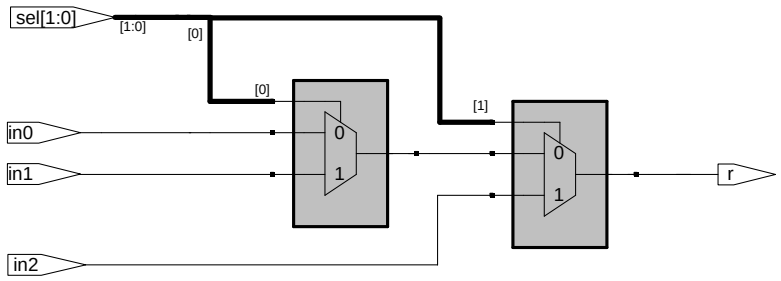
```
reg r;
always @ ( * )
if (sel==0)
   r <= in0;
else if (sel==1)
   r <= in1;
else if (sel==2)
   r <= in2;
```

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



Helyes Verilog leírás a case szerkezet használatával és az ebből keletkező hálózat:

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



Természetesen if-else szerkezet használatával is elkerülhető a latch implementációja, a case szerkezethez hasonlóan csak azt kell biztosítani, hogy minden bemeneti kombinációra legyen olyan ág, amely végrehajtódik. Triviális megoldás az utolsó ág feltételének elhagyása, azaz:

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

Megjegyzendő ugyanakkor, hogy ez a kód – bár jelen példában ugyanazon hardver szintetizálásához vezet – nem ekvivalens az előző, case szerkezetet használó leírással. Míg a case szerkezetes leírás esetében a nem használt bemeneti kombináció esetén a kimenet don't care, így a szintézer szabadon optimalizálhatja a generált kombinációs logikát, addig az if-else kódban megkötöttük, hogy a sel==3 esetén a kimenet legyen in2. Természetesen if-else szerkezet használatával is megadható a case szerkezettel ekvivelens kód az alábbi módon.

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

Természetesen if-else szerkezet használatával is elkerülhető a latch implementációja, a case szerkezethez hasonlóan csak azt kell biztosítani, hogy minden bemeneti kombinációra legyen olyan ág, amely végrehajtódik. Triviális megoldás az utolsó ág feltételének elhagyása, azaz:

A latch szándékos használata tehát kerülendő, a hibás (kombinációs logikának szánt) leírások esetén történő latch használatra pedig a szintézer figyelmeztet (Warning).

2.3.3. D Flip-Flop leírása

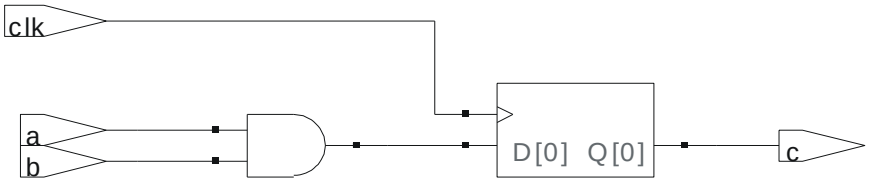
A D tároló egy élérzékeny tároló típus, amely az órajel bemenetének felfutó élére mintavételezi az adat bemenetét, s azt a következő órajel felfutó élig a kimeneten tartja. Ebből adódóan leírásakor élérzékeny *always* blokk használandó.

```
reg q;
always @ (posedge clk)
   q <= d;
```

A Verilog kódban természetesen „összevonható" a tároló elemet megelőző (annak bemeneti jelét generáló) esetleges kombinációs logikának, illetve magának a tárolónak a leírása.

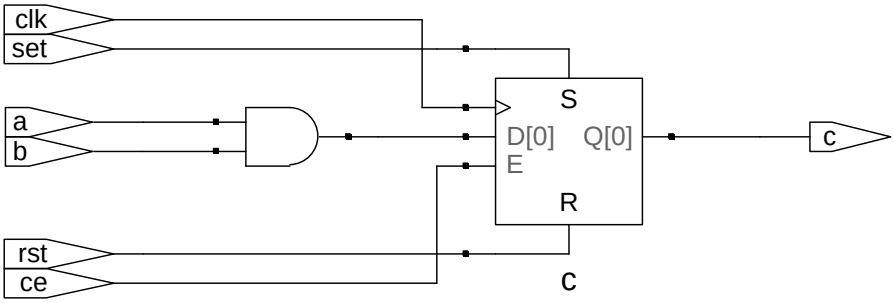
```
reg c;
always @(posedge clk)
   c <= a & b;
```

A fenti kódból az alábbi hálózat keletkezik:

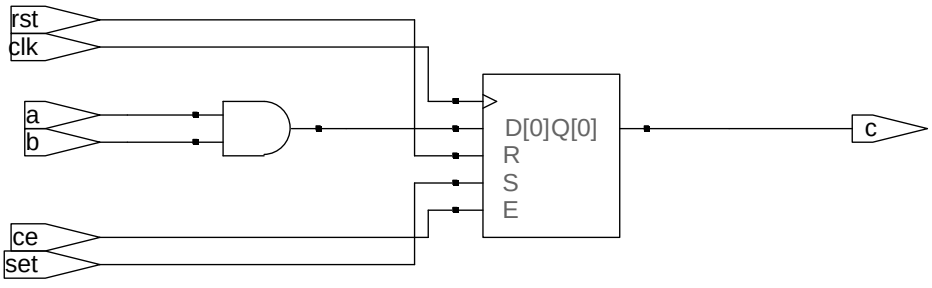


A Xilinx FPGA-kban található FF-ok az órajel és adat bemenetükön kívül rendelkeznek még egy reset (0-ba állító), egy set (1-be állító) valamint egy órajel engedélyező (ce) bemenettel is. Ezek közül a reset és a set lehet aszinkron és szinkron, utóbbi esetben a három bemenet prioritás sorrendje: reset, set, ce. Ennek megfelelően a FF teljes funkcionalitását az alábbi Verilog kódok nyújtják (előbbi aszinkron, utóbbi szinkron set és reset).

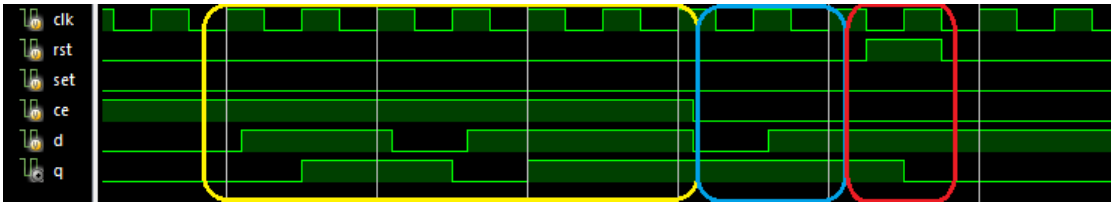
```
reg c;
always @ (posedge clk, posedge reset, posedge set)
if (reset)
   c <= 1'b0;
else if (set)
   c <= 1'b1;
else if (ce)
   c <= a & b;
```




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

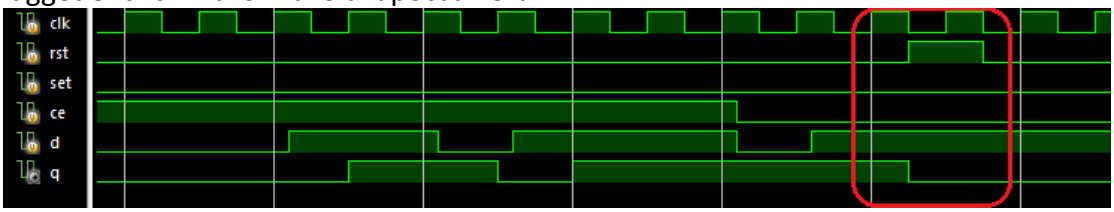


A következő ábra egy szinkron vezérlő jelekkel rendelkező D FF működését szemlélteti.



A hullámforma első néhány órajele alatt (sárgával jelölt rész) a D-FF alapvető működése látható: az órajel felfutó élekor mintavételezi a D adatbemenetet, s a

mintavételezett értéket a következő órajel felfutó élig tartja a Q kimeneten – azaz a bemenetet egy órajellel késlelteti. Az órajel engedélyező jel (ce) 0 állapota tiltja a mintavételezést, azaz a Q kimenet állapota nem változik a D adatbemenetnek megfelelően (kékkel jelölt rész). Az ábrán a reset (rst) bemenet szinkronitása is megfigyelhető: hiába vált a bemenet az órajel felfutó éle előtt 1-be, a FF csak az ezt követő órajel felfutó élkor kerül 0 állapotba (piros doboz).
Ezzel ellentétben aszinkron reset használata esetén az órajel bemenet állapotától függetlenül a FF azonnal 0 állapotba kerül.



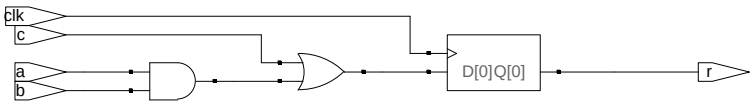
2.3.4. Blokkoló és nem-blokkoló értékadás

Mint az a fejezet elején már említésre került, *always* blokkon belül a Verilog lehetőséget nyújt mind blokkoló (**=**), mind pedig nem-blokkoló (**<=**) értékadás használatára. A figyelmes olvasónak feltűnhetett, hogy az előző fejezet összes példájában csak nem-blokkoló értékadások szerepeltek. Mielőtt ennek miértje röviden indoklásra kerülne: **a nagyon indokolt esetektől eltekintve (<1%) a blokkoló értékadások használata kerülendő**. Eme (igen erős) javaslat első oka szemléleti: egy hardver alapvetően párhuzamos működésű eszköz, amivel a szekvenciális végrehajtású blokkoló értékadás némileg ellentmondásban van. A második ok az, hogy a blokkoló értékadás a 2.3.3. fejezetben elmondottakat némileg bonyolítja, így a két értékadás kevert használata több hibalehetőséget rejt magában.

Nézzünk néhány példát a fentiek jobb megértésére.

```
reg [7:0] t, r;
always @ (posedge clk)
begin
   t = a & b;
   r = t | c;
end
```

A fenti HDL kód élérzékeny és blokkoló értékadásokat tartalmaz: a clk felfutó élénél kezdődik a kiértékelés, először végrehajtódik az első művelet (t új értéket kap), majd ezt követően értékelődik ki a második, t új értékét felhasználva (megj.: ha az első értékadás blokkoló, a második lehet akár nem-blokkoló is, a blokkoló értékadás mindenképpen blokkolja a végrehajtását). Ennek megfelelően a keletkező hardver blokkvázlata:

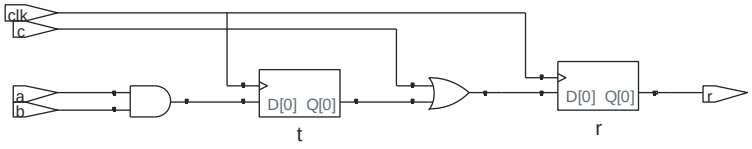


Figyeljük meg, hogy a fenti always blokkot tekintve ebben az esetben csak a második értékadás kimeneti változójából (r) keletkezett regiszter, t-ből nem.

Ugyanezt nem blokkoló értékadással:

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

Ebben az esetben clk felfutó élére mind t, mind pedig r párhuzamosan kiértékelődik, felhasználva a, b, c és t aktuális értékét. Utóbbi (t) aktuális értéke az, amelyet az előző órajel (clk) felfutó élnél kapott. Az ennek megfelelő hardver blokkvázlat:

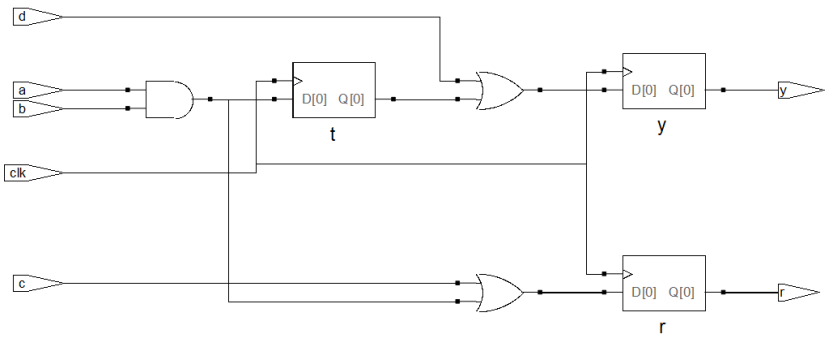


Ebben az esetben a fentebb leírtak teljesülnek, tehát a kimeneti változókból regiszter lesz.

Tekintsük az alábbi, két always blokkból álló példát.

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

reg y;
always @ (posedge clk)
begin
   y = d | t;
end
```

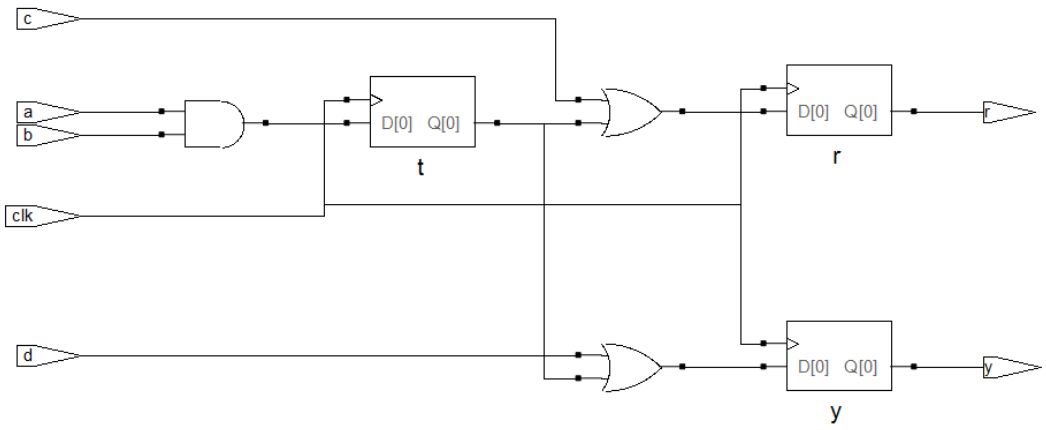
Az ebből generált hardver blokkvázlata az alábbi.



Látható, hogy a fentebb leírtaknak megfelelően az első always blokkból egy sorba kapcsolt ÉS + VAGY kapu keletkezik, utóbbi kimenete egy D-FF adatbemenetére kerül

(ez megfelel az előbbi blokkoló értékadás blokkvázlatban látottaknak). A második always blokkból létrejövő hardver – amely ugyancsak felhasználja az első blokkban írt „t” értéket – ugyanakkor nem közvetlenül az ÉS kapu kimenetét használja, hanem annak regiszterezett értékét. Azaz adott időpillanatban a két always blokk más-más „t” értékekkel kerül kiértékelésre; a blokkoló értékadás ugyanis csak annak az always blokknak a végrehajtását blokkolja, amelyben szerepel.
Mint az alábbi kód (ás blokkvázlat) mutatja, nem-blokkoló értékadást használva nem áll fenn ez az eset, hiszen ekkor minden, élérzékeny always blokkban írt változóból ténylegesen D-FF realizálódik.

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

reg y;
always @ (posedge clk)
begin
   y <= d | t;
end
```



Blokkoló értékadások használatára az esetek nagy részében semmi szükség, így a nem-blokkoló értékadások használata kötelező!

2.4. Strukturális leírás

Hierarchikus, kisebb modulokból történő építkezés esetén természetesen szükség van a modulok összekapcsolására, a hierarchia felépítésére. Az erre szolgáló szintaxis:

<modul_típus> <modul_példánynév>(port hozzárendelés);

Tekintsünk egy 2 bemenetű, 1 bites XOR kaput:

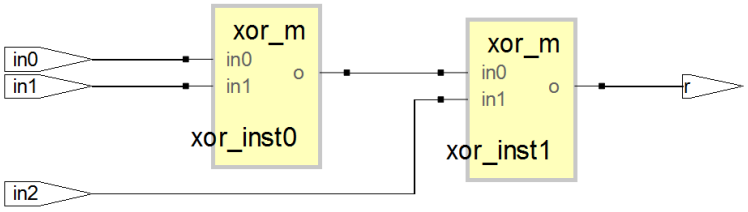
```
module xor_m(in0, in1, o);
       input  in0;
       input  in1;
```

```
        output o;

assign o = in0 ^  in1;

endmodule
```

Ennek felhasználásával egyszerűen létrehozható egy három bemenetű XOR kapu:

```
module xor_3 (input in0, in1, in2, output r);

wire xor0;
xor_m xor_inst0(.in0(in0), .in1(in1), .o(xor0));
xor_m xor_inst1(.in0(xor0), .in1(in2), .o(r));

endmodule
```

Mint azt az alábbi blokkvázlat is mutatja, az első XOR kapunk (xor_inst0) bemeneteire a három bemenetű XOR kapunk első két bemenetét kötjük, míg a második XOR kapunkra az első kapu kiemenetét (xor0), valamint a harmadik bemenetet kötjük. A két darab két bemenetű XOR kapu összekötéséhez szükség volt egy vezetékre (xor0).



A beillesztett modul bemeneteire közvetlenül csatlakoztatható akár „wire”, akár „reg” típusú jel is, a kimenetek azonban csak „wire” típusú jelet hajthatnak meg.

2.5. Szimuláció esetén használható nyelvi elemek

Mint azt a bevezetőben már említettük, a Verilog kifejlesztésének elsődleges célja a rendszermodellezés volt, így a nyelv számos, implementáció során nem használható konstrukciót tartalmaz. Ezek a nyelvi elemek ugyanakkor lehetővé teszik a szimulációhoz szükséges gerjesztések viszonylag egyszerű generálását. Szimuláció során a tesztelendő modult egy ún. *test fixture*-be illesztjük be, ami önmaga is egy verilog modul. A *test fixture*-ben pedig generáljuk a szimuláció során szükséges bemeneteket, illetve esetlegesen összehasonlítjuk a kapott kimenetet az általunk előre ismert, elvárt eredménnyel.

2.5.1. Timescale

A szintetizálható modulokkal ellentétben szimuláció esetén létezik a szimulációs idő, amellyel a *test fixture* eseményei ütemezhetők (egy implementációra készülő Verilog kód esetén erre nyilvánvalóan nincs lehetőség). Azt, hogy a megadott időadatok milyen mértékegységben értendők, illetve hogy mekkora a szimulációs lépésköz az ún. timescale direktívával adhatjuk meg, amely a *test fixture* modul deklarációja előtt szerepel.

```
'timescale 1ns/1ps
module test_fixture ();
```

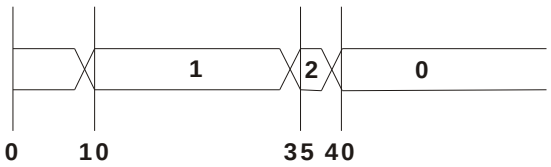
A fenti példában a szimulációs lépésköz 1ps, míg a megadott időadatok ns-ban értendők.

2.5.2. Initial blokk

Az *Initial* blokk az *always* blokkal ellentétben csak egyszer „fut le”, indulása a szimuláció indulásának pillanata. Csakúgy, mint a többi esetben, az *Initial* blokkok egymással, az *always* blokkokkal és az *assign* utasításokkal párhuzamosan működnek. Az *Initial* blokkokra az a megkötés sem vonatkozik, hogy egy változó csak egy blokkban kaphat értéket, benne azonban csak *reg* típusú változók használhatók. Az időzítési adatokat a **#** operátor után adhatjuk meg, s az egy *Initial* blokkon belüli értékek összeadódnak. Nézzünk egy példát egy jelgenerálására.

```
'timescale 1ns/1ps
module test_fixture ();

reg [7:0] test;

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

A létrejövő hullámforma:



2.5.3. Always blokk szimulációban

Szimuláció során az *always* blokk sem csak külső eseményre reagálhat, hanem itt is lehetőség van időzítésre. Például szinkron rendszer tesztelése esetén szükségünk van egy órajelre, amit pl. az alábbi módon generálhatunk.

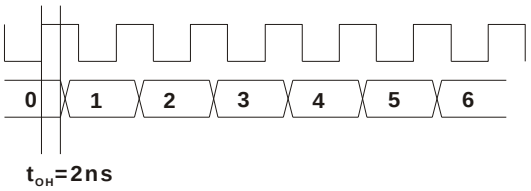
```
'timescale 1ns/1ps
module test_fixture ();

reg clk;
initial clk <= 0;

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

Ez a kódrészlet a szimuláció indításakor 0-ba állítja a clk változót, majd minden 5. ns-kor invertálja, létrehozva egy 10 ns periódusidejű órajelet.

A megfelelő tesztvektorok a valós hardver működését modellezik (hiszen a tesztelt modul az implementálást követően más hardverelemekkel fog kommunikálni), így a szimuláció során az órajelre generált bemeneteknél a megfelelő előkészítési és tartási időket (setup és hold time) biztosítanunk kell. Az alábbi példa egy számlálót szemléltet, amelynek értéke megfelelő időben kerül a tesztelt modulra (2 ns-mal az órajel felfutó éle után, modellezve a test fixture-ben létrehozott számláló hold time-ját).

```
'timescale 1ns/1ps
module test_fixture ();

reg clk;
initial clk <= 0;

always #5 clk <= ~clk;

reg [7:0] cntr;
initial cntr <= 0;
always @ (posedge clk)
   #2 cntr <= cntr + 1;
```

A generált hullámforma a megfelelő tartási idővel:



3. Példák

3.1. Multiplexer

Egy N:1 multiplexer olyan elem, amely az N darab bemenete közül (in0….inN-1) a kiválasztó jelnek (sel) megfelelőt adja a kimenetre. 2:1 MUX esetén a kiválasztó jel 1 bites, a HDL leírás pedig történhet *assign*-nal vagy *always*-zel is.

```
module mux_21 (input in0, in1, sel, output r);

assign r = (sel) ? in1 : in0;

endmodule
```

```
module mux_21 (input in0, in1, sel, output reg r);

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

endmodule
```

Több bemenetű multiplexer esetén célszerű a case szerkezet használata:

```
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
```

3.2. Shift regiszter

Következő példánk legyen egy szinkron tölthető, engedélyezhető jobbra/balra shiftelő 16 bites shift regiszter. A regiszter csak akkor működik, ha „*ce*” bemenete 1 értékű. Ekkor az „*ld*” jel hatására a 16 bites din bemenet töltődik a regiszterbe, egyébként pedig a „*dir*” 1 értékére a regiszter jobbra, 0 értékére balra shiftel. A shiftelés során belépő értéket az sdin 1 bites adatbemenet határozza meg. Az órajel a clk.

```
module shift_reg(
       input  clk, ce, ld, dir, sdin,
       input  [15:0] din,
       output [15:0] dout
);

reg [15:0] shr;

always @(posedge clk)
if (ce==1)
       if (ld==1)
              shr <= din;
       else if (dir==1)
              shr <= {sdin, shr[15:1]};
       else
              shr <= {shr[14:0], sdin};

assign dout = shr;

endmodule
```

3.3. Shift-regiszter tömb

Az előző példában egy bites jelet késleltettünk 16 órajellel, azonban számtalanszor szükségessé válik több bites (vektor) értékek késleltetése. Szerencsére a Verilog erre is kedvező konstrukciót nyújt. Az alábbi példában 8 bites értéket késleltetünk tetszőleges számú (de legfeljebb 16) órajellel.

```
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
```

Az első, eddig nem tárgyalt nyelvi konstrukció a shift regisztert megvalósító tömb deklarációja. Az első, szögletes zárójelben szereplő érték – az eddigieknek megfelelően – a változó szélességét definiálja, míg a második, a változó mögé írt érték a tömb elemszámát adja meg. A [15:0] ebben az esetben NEM azt jelenti, hogy a 16 biten ábrázolható 65.536 elemet fog tartalmazni a tömb, hanem csak 16-t.

Magát a shiftelés műveletet egy always blokkon belüli for ciklussal valósítjuk meg. A tömb 0-ik helyére az adatbemenetet írjuk, míg a tömb további elemeit a megelőző tömb elem értékével frissítjük. A fenti kód fontos konklúziója, hogy Verilogban a ciklusok NEM időbeli ciklikusságot adnak meg, hanem ugyanolyan funkcionális elemek kényelmes többszörözését teszik lehetővé – jelen példában ez egyszerűen 15 darab 8 bites regiszter létrehozását jelenti.

A kód utolsó értékadása nem más, mint egy multiplexer leírása. A 16 elemű tömbből a 4 bites addr bemenet használatával minden időpillanatban egy-egy tetszőleges elem kiválasztható, azaz 8 bites adatvonalakkal rendelkező 16:1 multiplexert valósít meg a kód.

3.4. Számláló

Számlálónk egy aszinkron reset-elhető, engedélyezhetően tölthető, fel/le számláló modul.

```
module counter(
       input  clk, rst, ce, ld, dir,
       input  [15:0] din,
       output [15:0] dout
);

reg [15:0] cntr;

always @(posedge clk, posedge rst)
if (rst==1)
       cntr <= 16'b0;
else if (ce==1)
       if (ld==1)
              cntr <= din;
       else if (dir==1)
              cntr <= cntr + 1;
       else
              cntr <= cntr - 1;

assign dout = cntr;
endmodule
```

3.5. Másodperc számláló

Képzeljünk el egy olyan feladatot, melyben a másodperceket kell számlálnunk, mégpedig egy 50 MHz-es órajellel rendelkező FPGA segítségével. Tehát egy olyan számlálóra van szükségünk, mely másodpercenként csak egyszer számol, nem pedig az 50MHz-es órajel minden felfutó élére. Ezt kétféleképpen érhetjük el:

- Generálunk egy 1 Hz-es órajelet, s a számláló D-FF-jainak órajel bemenetére ezt kötjük.
- Generálunk egy 1 Hz frekvenciájú engedélyező jelet. A rendszer összes FF-ja továbbra is az 50 MHz-es órajelről jár, de a másodperc számlálót megvalósító FF-k működését ezzel a jellel engedélyezzük (felhasználva a ce bemenetet).

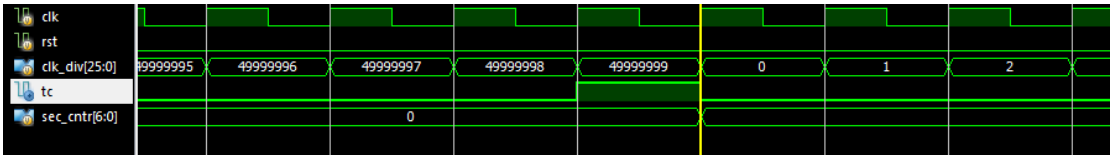
Itt nem részletezett okokból kifolyólag a második megoldás a preferált. Olyan jelre van tehát szükségünk, amely 50 millió órajelenként egyszer (→1 Hz), egyetlen órajel periódusig egy értékű, egyébként pedig 0. Az egymást követő 50 millió órajel megkülönböztetésére egy számlálót fogunk használni, amely ciklikusan 0-tól 49.999.999-ig számol. Ennek egy tetszőlegesen kiválasztott értékére történő komparálás szolgáltatja az engedélyező jelet.

```
module sec (input clk, rst, output [6:0] dout);

reg [25:0] clk_div;
wire tc;
always @ (posedge clk)
if (rst)
   clk_div <= 0;
else
   if (tc)
      clk_div <= 0;
   else
      clk_div <= clk_div + 1;

assign tc = (clk_div == 49999999);

reg [6:0] sec_cntr;
always @ (posedge clk)
if (rst)
    sec_cntr <= 0;
else if (tc)
    if (sec_cntr==59)
       sec_cntr <= 0;
    else
       sec_cntr <= sec_cntr + 1;

assign dout = sec_cntr;

endmodule
```

Az első *always* blokk az 1 másodperc alatt körbeforduló számlálót valósítja meg (0-tól 49.999.999-ig számlál). A blokk alatti *assign* egy kombinációs komparátor, amelynek kimenete egyetlen órajel ütemig magas értékű, mégpedig akkor, amikor a számláló regisztere a 49.999.999 értéket tartalmazza. Ez az egy órajel hosszúságú

impulzus engedélyezi a tényleges másodperc számláló működését, amely így másodpercenként egyszer változik.

A működést az alábbi időzítési diagram szemlélteti. Amint az engedélyező jelet generáló számláló eléri a 49.999.999 értéket, a kombinációs komparátor kimenete azonnal (tc) 1-be vált. Ennek hatására a következő órajel felfutó élkor a számláló a 0 értéket veszi fel (minek hatására a tc jel azonnal 0-ba vált), míg a másodperc számláló engedélyeződik, s felfelé számol.

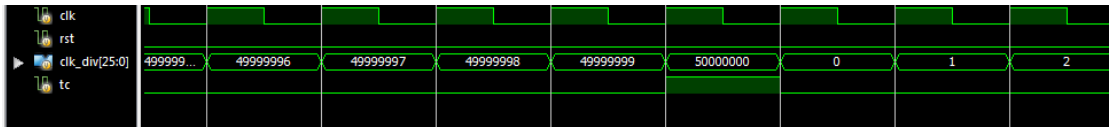


A fenti konstrukcióban az engedélyező jelet egy kombinációs logika generálja (komparátor). Ha az ezt a jelet felhasználó hardver-rész komplex (azaz sok FF clock enable bementére csatlakozik ez az engedélyező jel), akkor implementáció szempontjából célszerűbb, ha kombinációs logika helyett inkább egy FF kimenetéről származik a jel. Amennyiben a fenti kódban csak regiszterezzük az engedélyező jelet, az alábbi megoldáshoz jutunk:

```
module sec (input clk, rst, output [6:0] dout);

reg [25:0] clk_div;
reg tc;
always @ (posedge clk)
if (rst)
        clk_div <= 0;
else
        if (tc)
                clk_div <= 0;
        else
                clk_div <= clk_div + 1;

always @ (posedge clk)
   tc <= (clk_div == 49999999);

reg [6:0] sec_cntr;
always @ (posedge clk)
if (rst)
        sec_cntr <= 0;
else if (tc)
        if (sec_cntr==59)
                sec_cntr <= 0;
        else
                sec_cntr <= sec_cntr + 1;

assign dout = sec_cntr;

endmodule
```

A szimuláció eredménye előtt gondoljuk végig a kód működését. A továbbra is meglévő komparátor kimenetén helyeztünk el egy D FF-t, ami a bemeneti jelét (azaz a komparátor kimenetét) egy órajellel késlelteti. A komparátor továbbra is a számláló 49.999.999 állapota alatt ad ’1’-t, amit a FF késleltet, azaz a FF kimenete a számláló 50.000.000 értéke alatt lesz ’1’. Ennek hatására a számláló egy órajellel

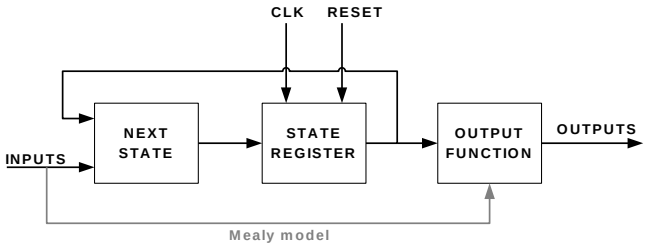
később kapja meg az őt resetelő tc jelet, azaz 49.999.999 helyett 50.000.000-ig fog számolni.



Ahhoz tehát, hogy az eredeti funkcionalitással megegyező hardvert kapjunk, nem 49.999.999-cel, hanem 49.999.998-cal kell komparálni. Ez által a komparátor kimenete egy órajellel előbb vált magas állapotba, kompenzálva a FF egy órajeles késleltetését.

3.6. Állapotgép leírása

Bonyolultabb vezérlési szerkezetek kialakítására használhatunk állapotgépet. Az általános állapotgép struktúrát mutatja az alábbi ábra.



Az éppen aktuális állapotot az állapotregiszter tárolja. A következő állapot meghatározását egy kombinációs hálózat végzi az aktuális bemeneteknek és az aktuális állapotnak megfelelően. A kimeneti értékeket egy újabb kombinációs vagy szinkron hálózat-rész határozza meg – Moore modell esetén csak az állapotregiszter felhasználásával, míg Mealy modell esetén az állapotregiszter és az aktuális bemenetek felhasználásával.

Implementáció során fontos kérdés a megfelelő állapotkódolás kiválasztása (bár a szintézerek többsége optimalizálja a talált állapotgépeket). Az elterjedtebb állapotkódolási lehetőségek (példát az alábbi táblázat tartalmaz):

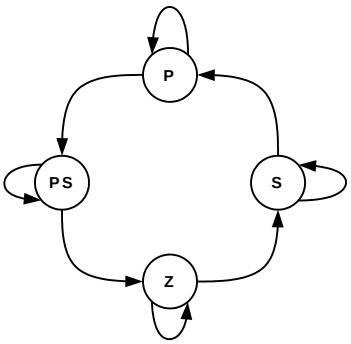
- Szekvenciális: az állapotokat (valamilyen szisztéma szerint) beszámozzuk, s a sorszám lesz az állapotok kódja. Tipikusan ez a megoldás igényli a legkevesebb állapotregiszter bitet, a járulékos kombinációs logika nagysága ugyanakkor erőteljesen függ az állapotok számától és a kimeneti függvény bonyolultságától. Előbbi oka, hogy az állapotváltásokhoz az összes állapotot dekódolni kell (egyszerűbben fogalmazva: állapot számnyi egyenlőség komparátorra van szükség).
- 1-az-N-ből (one-hot): az állapotregiszternek olyan széles, mint ahány állapot van. Minden pillanatban csak egyetlen bit aktív (azaz tartalmaz ’1’ értéket), ennek helye az állapotregiszterben azonosítja az állapotot. Éppen ezért nagyon jól használható olyan esetekben, amikor az állapotokon egymás után lépkedünk végig, és a kimeneti változóink állapotfüggő vezérlőjelek.
- Gray kód: jellemzője, hogy az egymás után következő bináris értékek között csak egyetlen bit változik (ugyanez nem jelenthető ki biztosan az egymást követő állapotokról, hiszen azok nem feltétlenül „sorban” követik egymást).

- Kimenet szerinti kódolás: az előző három csoportba nem feltétlenül sorolható állapotkódolási eljárás, mely esetén az állapotregiszter bitjei megfelelnek az állapotgép kimeneteinek (tehát nincs szükség kimeneti logikára).

Példa a fenti kódolásokra:

Decimális érték	Szekvenciális	One-hot	Gray
0	000	00000001	000
1	001	00000010	001
2	010	00000100	011
3	011	00001000	010
4	100	00010000	110
5	101	00100000	111
6	110	01000000	101
7	111	10000000	100

Nézzük meg egy egyszerű példán mindezt a gyakorlatban is. Az alábbi állapotdiagram egy egyszerűsített közlekedési lámpa állapotdiagramját mutatja (nincs villogó sárga mód).



A négy állapot:

- P: piros
- PS: piros és sárga
- Z: zöld
- S: sárga

A lámpa minden állapotban adott, de állapotonként különböző ideig tartózkodik. Az állapotkódolás szekvenciális, a Verilog kódban látható.

A működést leíró Verilog kód:

```
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;

// allapotregiszter leirasa
always @ (posedge clk)
if (rst)
       state_reg <= PIROS;
else
       state_reg <= next_state;

// allapotvaltasok (kombinációs logika)
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 <= PIROS;
endcase



// az egyes állapotok időzítése
always @ (posedge clk)
if (rst)
   timer <= 4500;
else
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



// kimeneti dekoder, kombinációs logika
always @ ( * )
case (state_reg)
      PS:    led <= 3'b110;
      PIROS: led <= 3'b100;
      SARGA: led <= 3'b010;
      ZOLD:  led <= 3'b001;
endcase

endmodule
```

Az első *always* blokk az állapotregiszter leírása: reset hatására a legbiztonságosabb PIROS módba kerül, majd minden órajelben betölti a következő állapotot előállító kombinációs logika kimenetét.

A következő *always* blokk az állapotváltás leírása; amennyiben az időzítő lejárt (elérte a 0-t), a következő állapotba lép, amíg ez nem történik meg, addig az aktuális állapotban marad.

A harmadik blokk az időzítő leírása. Az állapotváltáskor (tehát az időzítő 0 értéke esetén) a következő állapotnak megfelelő idő-értéket tölti be, majd a következő órajel ciklusok alatt ezt dekrementálja (az áttekinthetőség kedvéért megjegyzésként a következő állapot is látható).

Az utolsó (kombinációs) *always* blokk felelős a kimenet dekódolásáért: mindhárom izzóhoz egy-egy bit tartozik, amely '1' értékű, ha az izzó ég.

Természetesen a fenti megoldás nem kötelező érvényű: az állapotkódolás tetszőlegesen megválasztható, vagy akár a szintézerre is bízható. Az időzítőt is sokféleképpen meg lehet valósítani: használhatnánk akár állapotonként külön számlálót, vagy felfelé számlálót (ekkor mindig 0-t töltenénk be, de a állapottól függően a számláló más-más értékénél).

3.7. Nagyimpedanciás vonalak kezelése

Tételezzük fel, hogy egy nagyon egyszerű memóriához szeretnénk csatlakozni, melynek adatbusza kétirányú: írás esetén a memóriavezérlő hajtja meg, míg olvasás esetén a memória. A kétirányú vonalat vezérlő egységünk feladatai:

- előállítani megfelelő vezérlőjelet, amely engedélyezi a vonalak meghajtását (az alábbi példában: write)
- engedélyezett meghajtás esetén meghajtani a vonalakat
- egyébként nagyimpedanciás állapotba helyezni a vonalakat, így a memória képes azokat meghajtani

Az alábbi példában a „write" jelet (az itt nem részletezett) memória vezérlő generálja, s ugyancsak ettől az egységtől származik a kiírandó adat (data_out) is. A

memória felől érkező adat (data_in) ugyancsak a vezérlőbe fut be, míg a data_io a memória 8 bites adatbusza.

```
module tri_state (inout [7:0] data_io);

wire [7:0] data_in, data_out;
wire write;

assign data_in = data_io;
assign data_io = (write) ? data_out : 8'bz;
……
……
……
……
endmodule
```

(Megjegyzés: írás alatt nyilvánvalóan a data_in vonalon is a kimenő adat jelenik meg, mivel azonban a memóriavezérlő tudja, hogy éppen egy írási folyamat zajlik, ez nem jelent problémát).