Elektronikus Eszközök Tanszéke

Nagy Gergely

Bevezetés a Verilog alapú digitális tervezésbe

Budapest, 2007

A Verilog nyelv alapjai

A nyelv története

A Verilog egy hardver leíró nyelv, azaz elektronikus rendszerek egy szöveges reprezentációja. Lehetőséget nyújt a tervek ellenőrzésére (*verifikáció*), szimulációjára, az időzítések beállítására, a tesztelésre és a digitális szintézisre.

A Verilog HDL (*hardware definition language*) az 1364-es IEEE szabvány, amelynek első verzióját 1995-ben készítették el, majd 2001-ben frissítették. A verilog nyelv hivatalos definícióját a *Verilog Language Reference Manual* (LRM) [1] tartalmazza. Az IEEE 1364 csoport 2004-ben feloszlott, jelenleg a IEEE P1800 csoport gondozza a nyelv szabványát.

A szabvány definiál egy programozói felületet is az egyéb programozási nyelvekhez való kapcsolódás elősegítésére – ez a *Programming Language Interface* (PLI).

A nyelv a 1980-as években keletkezett, amikor a Gateway Design Automation cég kifejlesztette a Verilog-XL nevű logikai szimulátorát, és vele együtt a Verilog nyelvet. A céget 1989-ben felvásárolta a Cadence Design Systems, amely 1990-ben nyilvánossá tette a Verilogot annak érdekében, hogy az egyfajta szabvánnyá váljon.

A nyelv ezt követő gyors elterjedése révén ún. *de facto* szabvánnyá vált, ami azt jelenti, hogy nem hivatalosan, széles körben szabványként kezelték. Végül az Accelera non-profit szervezet kezelésébe került és az vitte végig a szabványosítási eljáráson, így vált a Verilog mára *de iure* szabvánnyá.

Az Accelera a nyelv kiegészítéseivel foglalkozik, létrehozta a *SystemVerilog*ot, amely a rendszertervezést támogatja, valamint a *Verilog AMS*-t, amely a vegyes jelű (*mixed signal*), tehát analóg és digitális elemeket egyaránt tartalmazó rendszerek leírását adja.

Az RTL szintű tervezés

A hardver leíró nyelvek 1980-as évekbeli elterjedésének az oka az volt, hogy a digitális rendszerek olyan mértékben elbonyolodtak, hogy kézi módszerekkel, tranzisztor, vagy kapu szinten már nagyon nehezen, vagy egyáltalán nem voltak átláthatóak. Ezért volt szükség arra, hogy bevezessenek egy magasabb szintű leírást, amely lehetővé teszi a tervezők számára, hogy elvonatkoztassanak a technikai alapoktól, ugyanakkor azonban még nem annyira absztrakt, hogy egy algoritmussal ne lehetne tényleges hardverre „lefordítani".

Az a szint, ami eleget tesz a fenti követelményeknek, a *regiszter-transzfer szint* (*register transfer level*). Az RTL szintű tervezés során regisztereket és az azok közti kapcsolatot adhatjuk meg, tehát adattárolókat és adatutakat definiálunk.

A Verilog nyelv ezen túl tartalmaz magasabb szintű elemeket is, amelyek segítségével nagyon hatékonyan lehet vezérlési szerkezeteket leírni.

Lexikai elemek

Az alábbiakban röviden áttekintjük a Verilog nyelv alapjait, részletes programozási példák a későbbi fejezetekben találhatóak. Ez az áttekintés nagyban támaszkodik a [2] irodalomra.

A Verilog nyelvben a *megjegyzések* megegyeznek a C++ megjegyzéseivel:

- egy sornyi megjegyzés: `// sor végéig megjegyzés`
- blokkos, többsoros megjegyzés: `/* megjegyzés */`

Számkonstansok kettes, nyolcas, tizes és tizenhatos számrendszerben adhatóak meg, a negatív számokat kettes komplemensben kell leírni. A konstansok formátuma a következő: `<bitek száma>'<számrendszer><számkonstans>`, ahol a bitek száma azt adja meg, hogy a konstans hány biten legyen ábrázolva. Ez elhagyható, azonban legtöbbször, gyakorlati okokból fontos megadni. A számrendszert egy betűvel kell megadni:

- `b, B`: kettes
- `o, O`: nyolcas
- `d, D`: tizes
- `h, H`: tizenhatos

A kis és nagy betűk közt itt nincs különbség. A fentiek alapján tehát szabályos számkonstansok az alábbiak:

```
'd43, 'D43,  6'b101011, 'B101011, 'o53, 'O53, 'h2b, 6'H2b
```

A Verilog programban *azonosítók*kal láthatunk el vezetékeket, regisztereket, modulokat. Egy azonosító betűkből, számokból, dollár jelekből (`$`) és aláhúzás jelekből (`_`) állhat. Az első karaktere betű és aláhúzás jel lehet. Az azonosítók értelmezésénél a kis és nagybetűk meg vannak különböztetve. Ez igaz a nyelv foglalt azonosítóira is, amelyek kis betűvel íródnak.

Adattípusok

Értékkészlet

A változók értékkészlete némiképpen különbözik a hagyományos programnyelvekétől, ennek oka a hardverközeliség. Egy numerikus változó egy bitje négy értéket vehet fel:

- `0`: nulla, vagy hamis
- `1`: egy, vagy igaz
- `x`: nem ismert (*don't care*)
- `z`: nagy impedancia

Vezetékek

A vezetékek (`wire`) a struktúrális egységek közötti kapcsolatokat reprezentálják. A vezetékek nem tárolnak adatot, ezért folyamatosan meg kell őket hajtani, biztosítani kell, hogy állandóan értéket kapjanak. Ez úgy érhető el, hogy egy kapunak, modulnak a kimenetéhez csatlakoztatjuk őket, vagy egy folyamatos értékadásban (`assign` – ld. később) szerepeltetjük őket.

Az alábbiakban látható néhány példa a deklarálásukra:

```
wire out1, out2, bell, x;  // single-bit wires
wire [7:0] adr, dat;       // 8-bit buses
```

Az első sorban egybites vezetékek szerepelnek – ez az alapértelmezés. A másodikban két nyolcbites buszt adtuk meg, amelyeknél a legbaloldalibb bit a legmagasabb helyiértékű. Ha fordított értelemben szeretnénk megadni a bitek helyiértékét, azt a következőképpen tehetjük meg: `wire [0:7] adr`[1].

A vezetéknek hivatkozhatunk egy, vagy több bitjére a `[]` operátor segítségével: `adr[2]`, `dat[5:3]`.

Regiszterek

A regiszterek az adattároló elemek (*flipflop* vagy *latch*) absztrakciói. Egy regiszter valamilyen esemény hatására kap értéket és azt a következő esemény bekövetkeztéig őrzi. Tipikusan procedurális értékadás során (`always` blokk – ld. később) kap értéket. A feltétel nélküli `always` blokkban szereplő regiszterek automatikusan vezetékké redukálódnak. A regiszterek definíciója a vezetékekhez hasonlóan történik, de itt a `reg` kulcsszót kell használni.

```
reg out1, out2, bell, x;  // single-bit registers
reg [7:0] adr, dat;       // 8-bit buses
```

Paraméterek

A hagyományos programnyelvek általában biztosítanak valamilyen módszert arra, hogy különböző konstansokat adjunk meg, így téve áttekinthetőbbé és könnyebben javíthatóvá a programot. A Verilog nyelvben erre szolgálnak a paraméterek (`parameter`). Érdemes például egy busz szélességét paraméterként megadni, hogy amennyiben a későbbiekben ez változna, akkor csak egy helyen kelljen belejavítani a kódba, és ennek hatására az összes hivatkozott helyen megtörténjen a csere.

```
parameter width = 8;       // width of a data bus
parameter clockper = 50;   // clock period
```

Paramétert csak modulon belül definiálhatunk, azonban példányosításkor van lehetőség az érték megváltoztatására:

[1] Ezt az alakot nem minden fordító ismeri fel!

```
module parMod(a, q);
  parameter width = 4;
  input [width - 1 : 0] a;
  output q;

  assign q = &a;
endmodule

module test;
  reg [7:0] A;
  wire q;
  parMod pm(A, q);

  defparam pm.width = 8;
  ...
endmodule
```

Létezik egy másik szintaktika, amellyel példányosításkor is meg lehet adni a paraméter értékét:

```
parMod #(8) pm(A, q);
```

Több paraméter esetén az értékeket vesszővel elválasztva kell megadni a deklarálásuk sorrendjében.

Kifejezések

Egy kifejezés operátorok és operandusok sorozata. A Verilog nyelv operátorai a C nyelvével majdnem teljesen megegyeznek.

Aritmetikai operátorok

- *unáris operátorok* – előjel: `+`, `-`
- *bináris aritmetikai operátorok*: `+`, `-`, `*`, `/`
- *modulus (maradékképzés) operátor*: `%`

A Verilog a regisztereket előjel nélküli egészként kezeli!

Relációs operátorok

- `<`, `<=`, `>`, `>=`
- `==` (egyenlőség)
- `!=` (nem egyenlő)

Logikai operátorok

- `!` (tagadás)
- `&&` (és), `||` (vagy)

Bit-szintű operátorok

- `~` (invertálás)
- `&` (és), `|` (vagy)
- `^` (kizáró vagy – xor)
- `~^`, `^~` (kizáró nem-vagy – xnor)

Redukciós operátorok

- `&` (és), `|` (vagy), `^` (xor)
- `~&` (nem-és), `~|` (nem-vagy), `~^`, `^~` (xnor)

A redukciós operátorok segítségével egy vektort (több bites vezeték, regiszter) egy bitté redukálhatunk úgy, hogy a bitjeit operandusként szerepeltetve alkalmazunk egy operátort. Például így „ÉS-elhetjük" össze egy busz bitjeit:

```
wire [7:0] data;
wire a;

assign a = &data;
```

Léptető operátorok

- `<<` (balra léptetés)
- `>>` (jobbra léptetés)

Feltételes operátor

A C nyelvhez hasonlóan a Verilogban is létezik a feltételes, három operandusú operátor: `<kifejezés> ? <ha igaz> : <ha hamis>`

Bitösszefűző operátor

Több különálló vezetéket, vagy több busz egyes részeit néha hasznos egy új buszba összefűzni és egyben kezelni. Erre használható a bitösszefűzés:

```
...
wire [7:0] newbus;

assign newbus = {adr[5:0], data[2:1]};
```

Ha ugyanazt a vezetéket többször szeretnénk szerepeltetni a buszban, arra több lehetőség is van: `{k, k, k, k, k}`, illetve `{5{k}}`. Ez a két leírás ekvivalens, a második az első rövidített alakja.

Olyankor is hasznos a bitösszefűző operátor, ha egy regisztert szeretnénk úgy léptetni, hogy a bitjeit visszakötjük (például így készíthető Johnson számláló is). Egy egyszerű, körben léptetett shiftregiszter így valósítható meg:

```
reg [7:0] regiszter;
...
regiszter = {regiszter[6:0], regiszter[7]};
```

Értékadások

Egy értékadás során az értékadó operátor (`=`) jobb oldalán lévő kifejezés kiértékelődik, és a bal oldalon lévő változó megkapja az értéket. A Verilog nyelvben ez többféleképpen történhet, a végrehajtás módja függ attól, hogy vezetéknek, vagy regiszternek adunk értéket. Eszerint egy értékadás lehet *folyamatos*, illetve *procedurális*.

Folyamatos értékadás

A folyamatos értékadás (*continuous assignment*) segítségével vezetékekre kényszeríthetünk egy értéket. Azért hívják folyamatosnak az értékadást, mert mindig aktív – ha bármi változás történik a jobb oldalán, akkor rögtön frissíti a baloldalt. Ilyen módon lehet szimulálni egy kombinációs hálózatot anélkül, hogy kapuk kapcsolatait kéne megadnunk. Az alakja:

```
assign <vezeték típusú változó> = <kifejezés>
```

A kifejezésre nincs semmiféle megkötés, akár függvényt is hívhatunk benne.

Procedurális értékadás

Procedurális értékadással csak regisztereknek adhatunk értéket. Ilyen értékadások csak olyan utasítás-szerkezetekben szerepelhetnek, mint az `always` és az `initial`. Az értékadást mindig egy esemény bekövetkezte váltja ki, tehát nem folyamatos. A végrehajtása akkor történik, amikor a program az adott utasításra lép. A bekövetkeztét feltételes utasításokkal (`if`, `case`) befolyásolhatjuk.

Az értékadás bal oldala lehet egy egybites regiszter, vagy több bites busz, illetve utóbbinak bármely része. Ha egy busz egy vagy több bitjét választjuk ki, akkor azt konstans számokkal kell tennünk – változóval nem indexelhetünk (hiszen ez a hardver dinamikus megváltozását jelentené, ami lehetetlen)[2].

Az értékadás során kétféle végrehajtási mód között választhatunk. A *blokkoló értékadás* során a végrehajtás sorbaveszi az egymás alatt lévő értékadásokat, tehát abban a sorrendben hajtja végre őket, ahogy a kódban szerepelnek.

```
<regiszter típusú változó> = <kifejezés> // blokkoló értékadás
```

A *nem-blokkoló értékadás* nem akasztja meg a végrehajtást. Ily módon több egymás alá írt, nem-blokkoló értékadás egyszerre kerül végrehajtásra.

```
<regiszter típusú változó> <= <kifejezés> // nem-blokkoló értékadás
```

Viselkedési modellezés eszközei

Az `always` és `initial` blokkok

A sorrendi hálózatok viselkedését meghatározó eseményvezéreltséget két utasítás-szerkezettel modellezhetjük – ezek az `initial` és az `always`. Mindkettő a szimuláció kezdetekor aktivizálódik; az előbbi csak egyszer fut le, az utóbbi a szimuláció befejeződéséig mindannyiszor, ahányszor a fejrészében lévő esemény bekövetkezik. Alakjaik:

```
initial
  begin
    ...
  end

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

Utóbbi példában a blokkon belüli utasítások akkor kerülnek végrehajtásra, ha az `a` változó értékében pozitív (vagyis felfutó) élváltás történik. Amennyiben az `initial`, illetve `always` szerkezetek csak egy utasítást tartalmaznak, akkor azokat egyszerűen utánuk írhatjuk. Ha több utasítás is tartozik hozzájuk, akkor az egy *utasításblokk*, amit a `begin` és `end` kulcsszavak közé kell tenni. Ez igaz a későbbi szerkezetekre (`if`, `case`, stb.) is.

Létezik olyan `always` blokk is, amelynek nincsen érzékenységi listája – tehát ami után nem kerül feltétel. Egy ilyen blokk folyamatosan fut, tehát az utolsó utasítás végrehajtása után újból kezdődik az első sorától. Egy ilyen blokk nem szerepelhet olyan modulban, amiből hardvert szeretnénk szintetizálni, azonban a tesztkörnyezetek kialakításakor nagyon hasznos (ld. 15. old.).

Feltételes elágazások

Az feltételes elágazás legegyszerűbb megvalósítása az `if` szerkezet, amelynek összetetettebb feltételek esetére létezik `if-else if-else` alakja is.

```
if (<kifejezése>) <utasítás(blokk)>
else if (<kifejezése>) <utasítás(blokk)>
else <utasítás(blokk)>
```

Az `if` szerkezet kifejezések igaz/hamis (1/0) értéke alapján választ ki utasításokat végrehajtásra.

A `case` szerkezet egy adott kifejezés többféle – akármennyi – értéke alapján ágazik el. Ha a kifejezés egyik felsorolt értéket sem vette fel, akkor a `default` ág hajtódik végre, ennek hiányában egy utasítás sem kerül végrehajtásra.

[2] Ha ilyen funkcióra van szükségünk, akkor multiplexert kell alkalmaznunk (ld. 9. old.)

```
case (<kifejezés>)
<érték_1>: <utasítás(blokk)_1>;
<érték_2>: <utasítás(blokk)_2>;
<érték_3>: <utasítás(blokk)_3>;
  ...
<érték_n>: <utasítás(blokk)_n>;
default:  <alapértelmezés szerinti utasítás(blokk)>;
endcase
```

Ha szeretnénk don't care biteket is használni a feltételekben, akkor a `casez`, illetve a `casex` kulcsszavakat kell használnunk. A `casez` a `z` értékeket tudja feldolgozni, a `casex` mind a `z`, mind az `x` értékeket.

Feltételes utasítások csak szekvenciális utasításblokkokban (`initial`, `always`) szerepelhetnek.

Esemény-vizsgálat

Ahogy azt korábban említettük a procedurális értékadások vezetékek vagy regiszterek értékében történt változási események bekövetkeztével szinkronizáltan kerülnek végrehajtásra. Ennek eszköze a `@(<kifejezés>)` operátor. A végrehajtás szinkronizálható az adott változás irányához a `posedge`, illetve `negedge` kulcsszavak segítségével.

Késleltetett utasítás végrehajtás

Az utasítások végrehajtása a szimulációs belső órajellel késleltethető. Ez nem szintetizálható szerkezet, csak a teszteléskor használjuk:

```
# <konstans> <utasítás>
```

ahol a konstans azt jelzi, hogy hány órajelperódussal történik a késleltetés.

Lehet egy értékadást is késleltetni az alábbi alakban:

```
always @(posedge clk) B = #1 A;
```

Ez az alak bizonyos esetekben a szimulációnál szükséges lehet, ugyanis így lehet belevinni a rendszerbe azt a késleltetést, ami egy valóságos áramkörnél minden esetben jelen van. Természetesen ilyen esetben a késleltetésnek jóval kisebb értéknek kell lennie, mint a rendszert vezélrő órajel periódusideje.

A fenti alakkal azonban bánjunk nagyon óvatosan! Ahogy azt korábban említettük, a késleltetés nem szintetizálható, ezeket egy szintézer figyelmen kívül hagyja. Tehát késleltetést csak a tesztkörnyezetben szabad használni, illetve olyankor, ha a valós áramköri késleltetést szeretnénk szimulálni. Egy áramkör működését nem szabad ezekre a késleltetésekre alapozni, hiszen ezek nem kerülnek be a tényleges kapcsolásba.

Hierarchikus tervezés Verilogban

Egy bonyolult rendszer tervezése során érdemes azt felosztani kisebb, funkcionális egységekre és azok kapcsolatából építkezni. A hagyományos programnyelvekben ezt moduláris programozásnak hívták. A Verilog nyelvben pontosan ez a neve azoknak az egységeknek, amelyekbe egy kisebb funkció megvalósítását zárhatjuk. Egy modult úgy képzelhetünk el, mint egy diszkrét elemekből épített digitális hálózatban egy SSI/LSI IC-t, amely a bemenetein és kimenetein keresztül kapcsolódik az áramkör többi eleméhez. Ilyen IC-kből épül fel egy digitális hálózat és hasonlóan, modulok összeépítésével hozhatunk létre egy összetettebb rendszert Verilogban. A modulok más modulokat is tartalmazhatnak, így egyszerűbbekből bonyolultabbakat készíthetünk.

Egy modult definiálnunk kell, vagyis le kell írni a működését, majd később példányosítani kell. Ez utóbbi során építjük bele egy nagyobb egységbe úgy, hogy vezeték típusú változóval összekötjük a portjait más modulok portjaival.

A modul definiálása során meg kell adni a be-, és kimeneteit, valamint a működését a korábban leírt szerkezetek segítségével.

```
module name(p1, p2, p3, ... pn);
  input p1, p2;
  input [msb1 : lsb1] p3;
  output p4, p5;
  output [msb2 : lsb2] p6;
  ...
  reg p4, p5;
  reg [msb2 : lsb2] p6;
```

```
  ...
  // a modul leírása
  ...
endmodule
```

A modulban definiálhatunk belső vezetékeket, regisztereket, valamint példányosíthatunk korábban definiált modulokat. A működés definíciója általában egy (vagy több[3]) `always` blokkban van. Emellett szerepelhet még benne egy `initial` blokk is, ám ebben az esetben a blokk nem szintetizálható, így ezt csak tesztelési célból szokták alkalmazni.

Egy modul bemenetei szükségképpen vezetékek, amiket külső kapuk hajtanak meg. A bemeneteket külön vezetékként nem kell deklarálni, az `input` kulcsszó ezt egyértelműsíti. A kimenetek lehetnek folyamatos értékadással meghajtott vezetékek, vagy procedurális értékadással vezérelt regiszterek.

A modulokat példányosítani úgy kell, mint ahogy C++-ban létrehozunk egy statikus objektumot egy osztályból, csak a konstruktor paraméterei helyén itt a be-, és kimenetek kapcsolatai szerepelnek.

```
<modulnév> <egyedi azonosító>(<port kapcsolati-lista>);
```

A port kapcsolati-listában a modul megfelelő be-, illetve kimeneteihez vezetékeket, illetve regisztereket kell illeszteni. A modul bemeneteire olyan változót kell kapcsolni, ami képes azt meghajtani, tipikusan ez regisztereket jelent. A kimenetek maguk regiszterek, így azokhoz vezetékeket kapcsolunk.

Amennyiben a modul példányosításakor meg szeretnénk változtatni a modulban érvényes paramétereket, akkor a következő szintaktikát kell alkalmazni:

```
<modulnév> <parameter-specifikáció> <egyedi azonosító> (<port kapcsolati lista>);
```

Létezik egy a fentitől eltérő szintaktika is, amellyel meg lehet adni a portkapcsolatokat. Ebben az esetben nem számít a pontos sorrend, ehelyett névszerint lehet összerendelni a portot a vezetékekkel illetve regiszterekkel:

```
<modulnév> <egyedi azonosító> (.<portnév>(<kapcsolódó vezeték / regiszter>))
```

Az alábbiakban lássunk egy-egy példát mindkettőre:

```
module counter(clk, rst, out);

...

module testbench;
begin
  reg clock;
  reg reset;
  wire output;

  counter cnt1(clock, reset, output);
  counter cnt2(.out(output), .clk(clock), .rst(reset));
...
```

A fentiekben áttekintettük a Verilog nyelv elemeinek egy részhalmazát. A megismert elemek segítségével elkezdhetjük a tervezési módszerekkel való ismerkedést, azonban az igényes és kifinomult tervezéshez szükséges a nyelv teljesebb ismerete. A további elmélyedéshez a [2] és [3] irodalmakat, valamint az Internetet ajánljuk.

A következőkben a Verilog nyelv segítségével megvalósítható tervezési módszerekbe adunk betekintést. Külön fejezetekben tárgyaljuk a kombinációs, illetve a sorrendi hálózatok tervezési elveit.

Kombinációs logikák szintézise

Logikai függvények megvalosítása

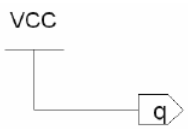
Folytonos értékadás használata

Kombinációs logikákat a legegyszerűbben folytonos értékadással (`assign`) hozhatunk létre. Ebben az esetben a megvalósítandó logikai függvényt kell megadnunk. Az értékadás jobb oldalán szereplő jelek nem fordulhatnak elő a bal oldalon – visszacsatolás nem megengedett [3].

[3] bizonyos fordítók modulonként csak egy `always` blokkot engednek meg

A szintézerek elvégzik a logikai függvények egyszerűsítését, ne várjuk, hogy pontosan a megadott kifejezés logikai kapukra való leképezését fogjuk kapni. Az alábbi példa kódjából például az 1. ábrán látható kapcsolás lett, ugyanis a függvény értéke minden bemeneti kombináció esetén logikai 1.

```
module Combinational1(a,b,c,q);
  input a;
  input b;
  input c;
  output q;
  assign q = ~(a & b & c) | ~(a & ~c);
endmodule
```



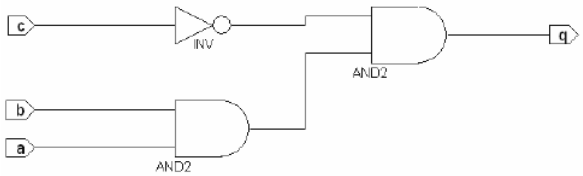
1. ábra *Szintetizált kombinációs hálózat*

Kombinációs logikák megvalósítására ez a módszer a legtisztább és legegyértelműbb, ugyanakkor egy bonyolult függvény esetén nehezen áttekinthető.

Eseményvezérlő alkalmazása kombinációs logikákban

Az `always` eseményvezérlő szerkezetet általában élvezérelt sorrendi logikák leírására használjuk, azonban kialakíthatóak a segítségével kombinációs kapcsolások is [3]. A következő kódból a fordító tisztán logikai kapukból felépülő hálózatot szintetizál – ez látható is a 2. ábrán.

```
module Combinational2(a, b, c, q);
  input a;
  input b;
  input c;
  output q;
  reg q;

  always @(a or b or c)
    begin
      if (b == 1'b1) q = a & ~c;
      else q = 1'b0;
    end
endmodule
```



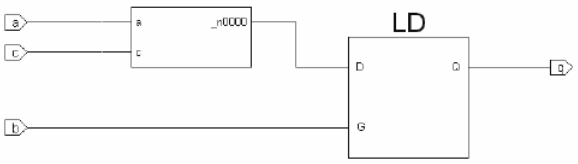
2. ábra *Egy `always` blokkból szintetizált kombinációs hálózat*

Ahhoz azonban, hogy egy `always` blokkból kombinációs hálózatot kapjunk, a kódnak szigorú követelményeknek kell megfelelnie. Egy kombinációs hálózatot megvalósító `always` blokkban csak `reg` típusú változónak adhatunk értéket. Ez azonban láthatóan nem eredményez feltétlenül tároló elemet a szintetizált hálózatban [3].

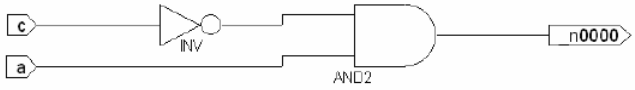
A tisztán kombinációs hálózat létrejöttének feltétele, hogy az `always` blokk érzékenységi listájában (a fejrésze) ne él-, hanem szintérzékenyek legyenek a jelek (ne szerepeljenek a `posedge`, illetve `negedge` kulcsszavak), és minden, a blokk értékadásainak jobb oldalán felsorolt jel szerepeljen [3].

Ezentúl az `if`, `case` és `?:` szerkezeteket teljesen ki kell fejteni. Ez azt jelenti, hogy az `if`-nek kell, hogy legyen `else` ága is, a `case`-ben fel kell sorolni minden lehetséges kombinációt, a `?:` kifejezés alkalmazása esetén pedig a feltétel teljesüléséhez, és nem teljesüléséhez tartozó ágakat is ki kell tölteni [3].

Ha például a fenti kódból elhagyjuk az `else` ágat, a 3. illetve a 4. ábrán látható kapcsolást kapjuk. A felső (3.) ábrán egy kombinációs hálózati blokk és egy data-gate tároló látható, az alsón (4. ábra) pedig a kombinációs blokk kifejtése.



3. ábra *Hibás kód miatt tárolót tartalmazó kapcsolás*



4. ábra *A 3. ábrán szereplő kombinációs blokk belső szerkezete*

Multiplexerek megvalósítása

A multiplexer olyan digitális áramköri elem, amely a bementére érkező adatjelek közül a kiválasztó jel alapján egyet a kimenetére kapcsol.

Multiplexerek megadása teljes `case` szerkezettel

A multiplexerek egyszerű kombinációs logikák. Megfelelő kód esetén a szintézer felismeri, hogy multiplexert kell megvalósítania. Az alábbiakban ugyanazt a 4-ből 1 multiplexert valósítjuk majd meg eltérő szerkezetekkel.

Egy teljesen kifejtett `case` utasítás tulajdonképpen az igazi multiplexert leképező nyelvi elem. A `case`-t `always` blokkba kell zárni, az érzékenységi listába pedig a multiplexer kiválasztó jele és az adatbemenet kerül [3].

```
module MultiplexerCase(q, data, select);
  output q;
  input [3:0] data;
  input [1:0] select;

  reg q;

  always @(select or data)
    begin
      case (select)
        2'b00: q = data[0];
        2'b01: q = data[1];
        2'b10: q = data[2];
        2'b11: q = data[3];
      endcase
    end
endmodule
```

Multiplexerek szintézise `?:` szerkezettel

Az alábbiakban a `?:` operátor alkalmazását mutatjuk be [3] multiplexerek szintetizálására[4].

```
module Multiplexer2(q, data, select);
  output q;
  input [3:0] data;
  input [1:0] select;

  assign q =
    (select == 2'b00) ? data[0] :
    (select == 2'b01) ? data[1] :
    (select == 2'b10) ? data[2] :
    (select == 2'b11) ? data[3] : 1'bx;
endmodule
```

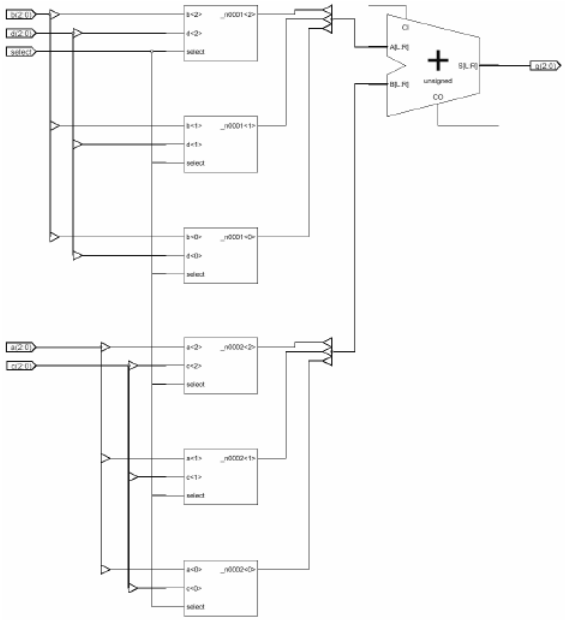
[4]Multiplexerek megvalósítására a `case` szerkezetet ajánljuk, ezt az alakot csak az érdekesség kedvéért szerepeltetjük itt

Vegyük észre, hogy az értékadás jobb oldalán tulajdonképpen `?:` szerkezetek vannak egymásbaágyazva. A feltétel nem teljesüléséhez tartozó ágba mindig egy új feltétel kerül. Az utolsó sorba kell valamit írni – hiszen különben tároló kerülne a kapcsolásba –, ezért ebben az esetben a kimenetekhez don't care értéket rendelünk. Mivel azonban felsoroltuk az összes bemeneti kombináció, ez az értékadás sosem következik be.

Multiplexer megvalósítható még `if-else if` egymásbaágyazással is, ez azonban meglehetősen átláthatatlan kódot eredményez.

Multiplexelt adatutak kialakítása

A multiplexerek egy nagyon fontos alkalmazása az adatutak kialakítása, ugyanis segítségükkel bonyolult műveletvégző elemek takaríthatóak meg. A következő példában két teljesen különböző tagokból álló összeadást végeztetünk el ugyanazon összeadó segítségével úgy, hogy a bemenetére multiplexer segítségével adjuk rá egyik, vagy másik operandust [3].



5. ábra *Multiplexelt adatutak kialakítása*

```
module Muxadd(q, select, a, b, c, d);
  output [2:0] q;
  input select;
  input [2:0] a;
  input [2:0] b;
  input [2:0] c;
  input [2:0] d;

  assign q = (select == 0) ? a + b : c + d;
```

A fenti egyszerű kódból az 5. ábrán látható, első ránézésre bonyolult kapcsolást kapjuk. Azonban valójában jelentős a megtakarítás, ugyanis a hálózat közepén található alhálózatokat reprezentáló dobozok mindössze egyszerű, egybites multiplexereket rejtenek, és így csak egy összeadó egységre van szükség.

A don't care értékek feldolgozása

A Verilog fordítók figyelembe tudják venni a don't care értékeket az optimalizálásnál [3]. A `casex` utasítás használata esetén az elágazási feltételekben megadhatunk `x` értékeket, amelyek azt jelentik, hogy azon a biten tetszőleges érték állhat. Az alábbi multiplexer példában két bites a kiválasztó jel, azonban csak két adatvezeték között kell választania, így a blokk belseje olyan, mintha csak egy kiválasztó jelvezeték lenne benne. A fordító ugyanakkor figyelmeztet, hogy egy bemeneti bit nem került felhaszálásra. Ez jelen esetben szándékos volt, azonban sokszor kódolási hibákra hívja fel a figyelmet ezzel az üzenettel!

```
module CaseXTest(q, select, data);
```

```
  output q;
  input [1:0] select;
  input [1:0] data;
  reg q;

  always @(select)
    begin
      casex (select)
        2'bx0: q = data[0];
        2'bx1: q = data[1];
      endcase
    end
endmodule
```

Demultiplexerek megvalósítása

Egy demultiplexernek egy darab egybites adatbemenete van és egy címbemenete, a kimenete pedig egy adatbusz. A bemenetre érkező adatjelet a cím által kiválasztott kimeneti bitre másolja át, a kimenet többi bitje nulla. A kimenetet egy több-bites számként felfogva a fenti feladat tulajdonképpen azt jelenti, hogy amennyiben a bemeneten egyes érték van, akkor azt annyi bittel kell eltolni balra, amennyi a bemenet értéke. Ha nulla van a bemeneten, akkor a teljes kimenet nulla értékű lesz, ez azonban nem különbözik a fenti esettől, hiszen a nulla tetszőleges eltolásával is nullát kapunk. Mindezen megfontolások alapján a demultiplexer Verilogos megfogalmazás az alábbi egyszerű alakban adható meg. Ez a forma jól szintetizálható.

```
module demultiplexer(in,select,out);
  input in;
  input [2:0] select;
  output [7:0] out;

  assign out = in << select;
endmodule
```

Dekóderek és enkóderek megvalósítása

A fentiekben bemutatott programozási technikákkal egyszerűen megvalósíthatóak multiplexerek, és a felhasznált szerkezetekkel hasonló módon alakíthatóak ki dekóderek és enkóderek. A *dekóder*ek bemenete egy n bites bináris szám, kimenetük pedig 2^n darab jelvezeték, amelyek közül a bemeneten lévő számnak megfelelőre logikai egyest, a többire logikai nullát adnak. Az *enkóder*ek ennek pontosan a fordítottjára képesek, vagyis 2^n bemenetük van, a kimenetük pedig egy n bites bináris szám.

Dekóderek megvalósítása

A dekóderek a Verilogbeli megfogalmazásukat tekintve nagyon hasonlítanak a multiplexerekre. Leghatékonyabban őket is `case` szerkezetekkel lehet megvalósítani. Mivel az elágazási feltételek közt fel kell sorolnunk az összes bemeneti kombinációt, a `default` sor elhagyható (ugyanazt a kapcsolást kapjuk akár jelen van, akár nincs – ezért került megjegyzésbe [3]). Ha nem adjuk meg az összes bemeneti kombinációt, akkor nem szabad elhagyni a `default` sort, hiszen különben sorrendi hálózatot kapunk!

```
module Dekoder(data, code);
  output [3:0] data;
  input [1:0] code;

  reg [3:0] data;

  always @(code)
    begin
      case (code)
        2'b00: data = 4'b0001;
        2'b01: data = 4'b0010;
        2'b10: data = 4'b0100;
        2'b11: data = 4'b1000;
```

```
        // default: data = 4'bxxxx;
      endcase
    end
endmodule
```

Enkóderek megvalósítása

Az enkóderek 2^n darab bementén egyszerre egy egyes lehet, aminek hatására az egyest tartalmazó láb sorrendjének megfelelő bináris szám jelenik meg a kimeneten. Az eddigi áramköreinkben sosem kellett törődnünk azzal, hogy a kapott bemenetek egyáltalán értelmesek-e. Erre a problémára két megoldás létezik.

Az első megoldás az, ha létrehozunk egy egybites kimenetet, ami azt jelzi, hogy a bemenet érvényes-e. A gond az, hogy az érvényességet ellenőrző logikai függvény szimmetrikus, tehát nem egyszerűsíthető. A gyakorlatban jellemzően nem alakítanak ki ilyen áramkört, hanem egy másik megoldást alkalmaznak: a *priorításos enkóder*eket. Hibás kód esetére a kimeneti értékkészlet elemeit valamilyen prioritás szerint adjuk meg. Így sokkal kisebb hálózatot kapunk, ugyanis a gyakorlatban ez don't care-ek bevezetését jelenti.

```
module PriorityEncoder(code, valid, data);
  output [1:0] code;
  input [3:0] data;
  reg [1:0] code;

  always @(data)
    begin
      casex (data)
        4'b1xxx: code = 2'b11;
        4'b01xx: code = 2'b10;
        4'b001x: code = 2'b01;
        4'b0001: code = 2'b00;
        default: code = 2'bxx;
      endcase
    end
endmodule
```

Ez az enkóder úgy működik, hogy ha a legmagasabb helyiértékű biten egyes van, akkor úgy tekinti, mintha a többin nulla lenne, és hozzárendeli a legnagyobb bináris számot a kimeneti értékkészletből, jelen esetben a 4-et. Ez lesz tehát a legnagyobb prioritású bemenet, hiszen bármelyik másikat is szerettük volna kiválasztani, ha tévesen logikai egyes került erre a bitre, akkor ő fog érvényre jutni. Az eggyel alacsonyabb helyiértéken lévő bit eggyel alacsonyabb proritású is, és így tovább. Ebben a `casex` szerkezetben az alapértelmezés szerinti érték megadása tulajdonképpen felesleges, hiszen a don't care-ek segítségével minden bemeneti kombinációt „lefogtunk".

Enkóder kialakítható `if-else` szerkezetekkel is, de a `case` alkalmazása áttekinthetőbb kódot eredményez.

Összeadók szintézise

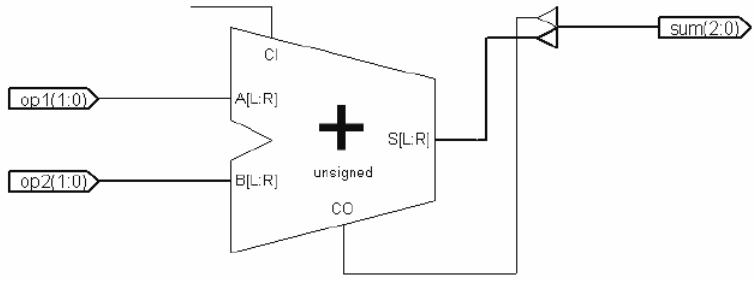
Összeadókat nem kell magunknak terveznünk, mivel minden fordító biztosít ilyen egységeket könyvtári elemként. Ahogy a multiplexelt adatutakat bemutató példában már láthattuk, elég leírnunk az összeadó operátort. Ha saját magunk, kapunkból definiálunk egy összeadót, azt a fordító nem ismeri fel összeadóként [3].

Érdemes megfigyelni az említett kapcsoláson (5. ábra), hogy az összeadó átvitel kimenete (*carry out*) sehová sem lett bekötve, ugyanis az összeg pont annyi bites volt, amennyi bitesek az operandusok. Ha azonban az összeget eggyel több bitesre állítjuk, akkor ezt felismeri a fordító[5], és az átvitel kimenetet beköti a legmagasabb helyiértékű bitre. Ez látható az alábbiakban.

```
module Adder(sum, op1, op2);
  output [2:0] sum;
  input [1:0] op1;
  input [1:0] op2;

  assign sum = op1 + op2;
endmodule
```

Ha a bemenetek mérete nem egyezik, akkor jóval bonyolultabb hálózatot kapunk.

[5] Nem minden szintézer ismeri ezt fel



6. ábra *Összeadó*

Sorrendi hálózatok tervezése

Bevezetés

Az olyan logikai hálózatokat nevezzük *sorrendi*eknek, amelyek tárolt állapotokkal rendelkeznek. Minden időpillanatban egy adott állapotban vannak. Egy másik állapotba a bemeneteik változásának hatására kerülhetnek. Azt, hogy milyen állapotváltás játszódik le, és mi kerül a hálózat kimenetére, a bemenete és az aktuális állapota dönti el. Egy sorrendi hálózat *szinkron*, amennyiben állapot-, és kimenetváltozás csak órajelre történhet. A Verilog HDL alapvetően kombinációs, és szinkron sorrendi hálózatok leírásában hatékony.

Egy szinkron sorrendi cella váza

A Verilog HDL-ben modulárisan tervezünk – a feladatot részfeladatokra bontjuk fel, és ezeket különálló egységekben valósítjuk meg. Bonyolultabb funkciókat az alapegységek összekapcsolásával nyerhetünk. Verilogban az önálló funkcionális egység nyelvi eleme a `module`.

Amennyiben szinkron sorrendi működést szeretnénk megvalósítani, akkor teljesítenünk kell a definíció által előírt követelményeket:

1. Állapotokkal kell rendelkeznie. Ez a gyakorlatban egy belső memóriát igényel, amely tárolja az aktuális állapotot, amelynek értékét a modul a kimenetének meghatározásához felhasználja.
2. Órajelre kell állapotot és kimenetet váltania, tehát minden modulnak kell, hogy legyen egy órajel bemenete, és a változásokat ehhez kell szinkronizálni.

Az alábbi kód egy egyszerű szinkron sorrendi modul kódjának vázát tartalmazza. Látható, hogy az adat bemenet (`in`) mellett tartalmaz egy órajel bemenetet is (`clk`). A kimenete (`q`) egyben egy regiszter is. A legegyszerűbb esetben itt tároljuk a hálózat állapotait is, így egy *Moore-automatát* valósítva meg.

```
module alap_cella(q, clk, in);
  input clk;
  input in;
  output q;

  reg q;

  always @(posedge clk)
  begin
    //a funkció leírása
  end
endmodule
```

Látható, hogy a funkciót az `always` blokkon belül valósítjuk meg, amely akkor futtatja a benne lévő utasításokat, amikor a fejében lévő feltétel teljesül: jelen esetben amikor az órajelnek felfutó éle van. Minden tisztán szinkron sorrendi hálózatnak a fenti váznak megfelelően kell felépülnie.

Egy egyszerű hálózat megvalósítása

Nézzünk meg egy egyszerű példát! Az alábbi modul kimenete egy négybites regiszter, a bemenete egybites. Ha az órajel felfutásakor a bemenet nulla, akkor nullázódik a kimenet is. Ha pedig egyes értékű, akkor a kimenet értéke eggyel növekszik. Ha a bemetetet folyamatosan egyes értéken tartjuk, akkor a hálózat tulajdonképpen négy biten megszámlálja az órajeleket.

```
module valami(q, clk, in);
  input clk;
  input in;
  output [3:0] q;

  reg [3:0] q;

  always @(posedge clk)
  begin
    case (in)
      1'b0 : q = 0;
      1'b1 : q = q + 1;
    endcase
  end
endmodule
```

Egy számláló

Valósítsunk meg egy valódi számlálót! A fenti hálózaton módosítani kell. A bemenet tulajdonképpen két vezérlő funkciót mos össze: az inicializálást (reset funkció), és a számlálás engedélyezését. Valósítsunk meg tehát egy számlálót, amely a reset bemenete segítségével kinullázható, és amely, ha engedélyezve van a számlálás, akkor minden órajelre növeli az értékét, ha nincs, akkor tartja azt!

```
module szamlalo(q, clk, reset, enable);
  input clk;
  input reset;
  input enable;
  output [3:0] q;

  reg [3:0] q;

  always @(posedge clk)
  begin
    if (reset == 1'b1) q = 4'b0;
    else
      if (enable) q = q + 1;
      else q = q;
  end
endmodule
```

A szinkron működés szabályait egy esetben szokták áthágni: a kinullázást gyakran az órajeltől függetlenül is meg lehet tenni. Ezt *aszinkron reset*nek hívják, és azt jelenti, hogy ha a `reset` bemenet bármikor felfut, a kimenet kinullázódik függetlenül attól, hogy az órajel épp milyen értéken van.

Ez úgy oldható meg, hogy az `always` blokkot ,,érzékennyé tesszük" a `reset` jelre is:

```
module aszinkronreset_szamlalo(q, clk, reset, enable);
  input clk;
  input reset;
  input enable;
  output [3:0] q;

  reg [3:0] q;

  always @(posedge clk or posedge reset)
  begin
    if (reset == 1'b1) q = 4'b0;
    else
      if (enable) q = q + 1;
      else q = q;
  end
endmodule
```

Még okosabbá tehetjük a számlálónkat, ha képessé tesszük arra, hogy mindkét irányban tudjon számlálni. Ennek érdekében az `enable` bemenetet alakítsuk át üzemmód bemenetté:

```
module fel_le_szamlalo(q, clk, reset, mode);
  input clk;
  input reset;
  input [1:0] mode;
  output [3:0] q;

  reg [3:0] q;

  always @(posedge clk or posedge reset)
  begin
    if (reset == 1'b1) q = 4'b0;
    else
      case (mode)
        2'b00: q = q;
        2'b01: q = q + 1;
        2'b10: q = q - 1;
        2'b11: q = q; //azért, hogy teljes legyen a case
      endcase
  end
endmodule
```

Tesztkörnyezet építése szinkron sorrendi hálózatokhoz

A Verilog HDL-ben a tesztkörnyezet egy olyan modul, amelynek nincsenek portjai (vagyis nincs sem bemenete, sem kimenete). Egy ilyen modul nem szintetizálható, csupán arra jó, hogy leellenőrizzük a programunkat.

A környezet kialakításához három dolgot kell megtennünk:

1. Regisztereket kell lértehoznunk, amelyek az áramkörök számára bemenetként fognak szolgálni, és amelyek segítségével „gerjeszthetjük" a moduljainkat.
2. Elő kell állítani az órajelet.
3. Példányosítani kell a moduljainkat, és megfelelően össze kell kötni őket.

Az alábbi tesztkörnyezet-váz megvalósítja a fentieket[6]:

```
module testbench;
  reg input_1, input_2 ...
  reg clk;

  initial
  begin
    clk = 0;
    //tesztjelek
  end

  always
  begin
   #1 clk = ~clk;
  end
endmodule
```

Az `initial` blokk csupán egyszer fut le. Ebben állíthatjuk be a vezérlőjeleket. Lehetőség van a futtatás késleltetésére – ezt `#n` alakban tehetjük meg, ahol n a szimulátor oszthatatlan időegységeiben való késleltetést jelent. Ezt úgy érdemes felhasználni, hogy nulla késleltetéssel beállítjuk a vezérlőjelek kezdeti értékeit, majd egyenként változtatjuk az értékeiket közben mindig beiktatva néhány egységnyi késleltetést.

[6]Látható, hogy a kódban a modulnév után nem szerepel zárójel – ezt így elfogadja az értelmező, de használható a C nyelvhez közelebbi szintaktika is: `module testbench();`

Az órajel egy érzékenységi feltétellel nem rendelkező, tehát feltétel nélkül, örökké futó `always` blokkban van megvalósítva, ahol egy késleltetéssel mindig invertáljuk az egybites `clk` regiszter értékét, amit az `initial` blokkban inicializáltunk. Így egy két időegységnyi periódussal rendelkező órajelet hoztunk létre. Ez a lehető leggyorsabb, amit meg tudunk valósítani. Sokszor nem érdemes ilyen rövidre állítani az órajel periódus idejét – ez gondot okozhat a szintetizált áramkör szimulációjánál, illetve így elvesztjük azt a korábban említett lehetőséget, hogy az áramkör valós késleltetéseit rövid – az órajel periódus idejénél lényegesen rövidebb – késleltetési idők segítségével vegyük figyelembe. Így hasznos lehet, ha az órajel periódusidejét 10 és 100 időegység közötti értékekre állítjuk.

Mivel az `always` blokk örökké fut, a szimuláció leállításáról gondoskodni kell. Bizonyos szimulátorokban erre szolgál a `$finish` utasítás, azonban sok szimulátorban ma már a környezetben állíthatjuk be a szimuláció hosszát, így a kódba ezt nem kell beleírni.

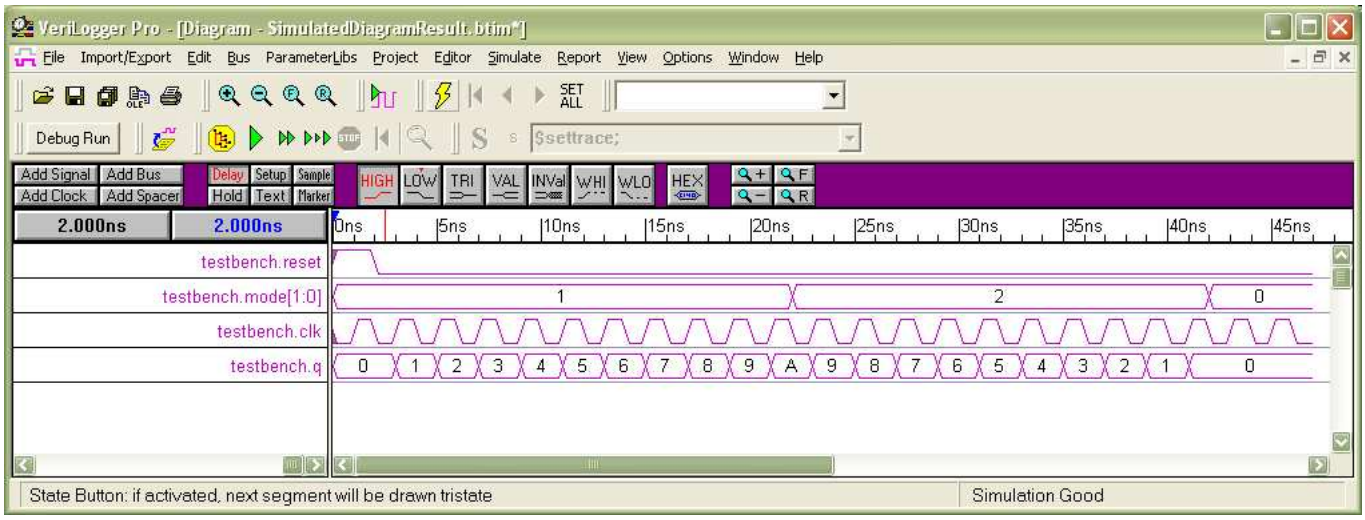
A számláló tesztkörnyezete

A bemeneteket regiszterként kell deklarálnunk, hiszen azok értékét be szeretnénk állítani, és azt szeretnénk, hogy azok folyamatosan tartsák ezeket az értékeket. A kimenetek vezetékek, vagyis `wire` típusúak. Az egybites vezetékeket nem kötelező deklarálni, azok automatikusan „létrejönnek” az első használatkor. Ha azonban a kimenet több bites, mint jelen esetben is, akkor explicite deklarálni kell, ahogy ez látható is az alábbi kódban. Általános azt lehet tanácsolni, hogy inkább az egybites vezetékeket is deklaráljuk, mert átláthatóbbá teszi a kódot.

```
module testbench;
  reg reset;
  reg [1:0] mode;
  reg clk;

  wire [3:0] q;

  fel_le_szamlalo szam_1(q, clk, reset, mode);

  initial
  begin
    clk = 0;
    mode = 1;
    reset = 1;

    #2
    reset = 0;

    #20
    mode = 2;

    #20
    mode = 0;

  end

  always
  begin
   #1 clk = ~clk;
  end
endmodule
```

Jól látható az `initial` blokk alapján, hogy egy sorrendi hálózat tesztelésekor az a cél, hogy minden állapotát kipróbáljuk. A folyamatosan pulzáló órajel mellett először kiadjuk a `reset` jelet, majd a számláló mindhárom üzemmódját beállítjuk, és megfigyeljük a kimenet változását egy szimulátor segítségével (lásd a 7. ábrán) – jelen esetben ez a *VeriLogger*[7].

[7] A VeriLogger szimulátorban szükség van a `$finish` kulcsszóra a szimuláció leállításához. Ezt írjuk az initial blokk végére megfelelő késleltetés után.



7. ábra *A számláló szimulációja a* VeriLogger *segítségével*

A Verilog-2001 új lehetőségei

A Verilog nyelv szabványát 2001-ben megújították, ekkor sok új elemmel bővült[8]. Ezek közül csak néhányat említünk meg az alábbiakban.

Egyesített port és adattípus deklaráció

A sorrendi hálózatok kimenetei jellemzően regiszterek. Ezeket a régi szabvány szerint két lépésben tudtuk deklarálni: meg kellett adni azt, hogy az adott port kimenet, majd egy külön sorban, hogy egyben egy regiszter is.

```
module sequential(out, in, clk);
  input clk;
  intput in;
  output out;
  reg out;
  ...
endmodule
```

Az új szabvány szerint erre van lehetőség egyesített alakban is:

```
module sequential(out, in, clk);
  input clk;
  intput in;
  output reg out;
  ...
endmodule
```

Ha a kimenet nem egybites, akkor a következő szintaktikát alkalmazhatjuk: `output reg [7:0] out`.
Lehetőség van egy további egyszerűsítésre: meg lehet adni a portokat a C függvényekhez hasonló módon:

```
module sequential(output reg out, input wire in, input wire clk);
  ...
endmodule
```

A paraméterek értékének megadása példányosításkor

A Verilog-2001-ben a paraméterek példányosításkori megadásának van egy módosított alakja is, ami hasonló az argumentumok megadásához:

```
parMod #(.param1(2), .param2(1)) pm(a, b, c, q);
```

ahol a paraméternevek: `param1`, `param2`.

[8] Még nem minden fordító és szimulátor ismeri az új szabványt – fontos, hogy ezt ellenőrizzük a munka kezdetén!

Skálázható modulok – hardvergenerálás

Felmerülhet olyan probléma, hogy egy adott egységből nagyon sokat kéne példányosítani. Ez nagyon sok gépelést jelentene, ami rengeteg hibalehetőséget rejt magában. A Verilog-2001 lehetőséget biztosít modulok példányosítására kódból. Ezt az eszközt fel lehet használni arra is, hogy valamilyen paramétertől függően eltérő hardvert példányosítsunk. Az alábbi példában nyolc darabot hozunk létre egy adott modulból:

```
module myMod(input a, input b, output q);
...

module generator(input [7:0] a, input [7:0] b, output [7:0] q);

  genvar i;

  generate
    for (i = 0; i < 8; i=i+1)
    begin : bigMod
      myMod m(a[i], b[i], q[i]);
    end
  endgenerate

endmodule
```

A hardver generálása a `generate` blokkban történik. Ezen belül egy `for` ciklus segítségével hozzuk létre az egyes példányokat. A ciklusváltozó speciális, a típusa `genvar` és a `generate` blokkon kívül deklaráljuk. A `for` ciklus `begin` utasítása után meg kell adni az elkészülő, összetett modul nevét (a rá való hivatkozást lásd lejjebb). A cikluson belül használhatjuk a `genvar` változót, mint paramétert. A fenti példában például a bemeneti buszok egy-egy bitjét választjuk ki a segítségével, hiszen a sokszorosított modul egybites bemeneteket tud fogadni.

Amennyiben hivatkozni szeretnénk az egyes sokszorosított példányokra, azt a következő módon tehetjük meg: `bigMod[n].m`, ahol az `n` helyére egy adott szám kerül (jelen esetben 0 és 7 között).

Hivatkozások

[1] *IEEE 1364 csoport:* OVI Verilog HDL LRM

[2] *Gärtner Péter:* Verilog for synthesis – Primer and Introduction, http:\\www.eet.bme.hu\∼ gaertner

[3] *Fehér Béla:* A Verilog hardver leíró nyelv alkalmazása a logikai szintézisben, Budapesti Műszaki és Gazdaságtudományi Egyetem, Méréstechnikai és Információs Rendszerek Tanszék

[4] *Stuart Sutherland:* The IEEE 1364-2001 Verilog Standard, What's New and Why You Need It, HDLCon, March 2000

[5] *Doulos cég honlapja:* http:\\www.doulos.com

Tartalomjegyzék