VHDL oktatási segédlet

1.13. verzió

Tartalomjegyzék

Előszó

Ezen rövid leírás abból a célból jött létre, hogy segítse az Olvasót abban, hogy megismerje a VHDL hardverleíró nyelv alapjait. Ez a mű nem referencia, tehát nem arra szolgál, hogy a nyelv és a hardver programozás minden elemét kimerítően tárgyalja. Ha az Olvasó kedvet érez ahhoz, hogy mélyebb ismereteket szerezzen e témában, akkor javaslom, hogy egy megfelelő referenciamű tanulmányozásával helyettesítse ezen mű megismerését.

A leírás elkészítésekor szem előtt tartottam, hogy egyszerű, jól érthető módon adjam közre az ismereteket, de belátható, hogy 50-55 oldalban még csak kísérletet sem tehetek arra, hogy a digitális technika minden részterületére kitérjek, vagy az alapfogalmak magyarázásával vesztegessem az időt. Mivel az Olvasó – ha eddig a félévig eljutott – biztosan elvégezte a „digitális technika” nevű tárgyat, így elméletileg tisztában van a digitális technika alapjaival. Élek azzal a feltételezéssel, hogy Zsom Gyula tanár úr könyveiben leírtak nem ismeretlenek az Olvasó számára.

A jegyzet első részében a VHDL nyelv alapelemeit, lexikai komponenseit ismerhetjük meg, a második részben pedig egyszerű, a szimulátorban és az FPGA boardon is kipróbált mintaprogramokon keresztül mélyíthetjük el tudásunkat. Sajnos már-már kötelező, hogy a mintaprogramok ne működjenek elsőre. Ez ellen úgy védekezhetünk, hogy az Olvasó ténylegesen megérti azok működését, és ha a fejlesztői környezet idő közben megváltozik is, a megszerzett ismeretek alapján az Olvasó könnyen korrigálhatja a mintaprogramot.

A VHDL nyelv megismeréséhez sok sikert kíván:

a szerző

Zaventem, Belgium, 2008. szeptember 15.

1. A VHDL nyelvi elemei

Ebben a részben azzal foglalkozunk, hogy a a VHDL nyelv legalsó szintje milyen alapvető elemekből épül fel. Ezek az alapelemek a nyelv építőkövei, de olyan aprók, hogy önmagukban működésre képtelenek. Ennek ellenére szükséges megismerkedni velük, mert a következő fejezetek jelentős mértékben támaszkodnak az ebben a fejezetben elmondottakra: például később már nem tudom elmondani a bináris számok megadásának módját, hanem erre a fejezetre fogok hivatkozni.

A VHDL nyelv a többi programozási nyelvhez hasonlóan szigorúan megköveteli a szintaktikai szabályok követését. Ezen szabályok közül a legelső a felhasználható karakterkészlet meghatározása. Néhány kivételtől eltekintve (pl. Java) programozás során „hagyományos” ASCII karaktersorozatokat használunk arra, hogy utasításainkat a számítógép számára közöljük. Nincsen ez másként a VHDL esetében sem. Elvileg nem lehetetlen, hogy például UTF8-at használjunk a programozás során, de semmiképpen sem javaslom, hogy ezt a gyakorlatot folytassuk. A forrásfájl legyen mindig 7-bites ASCII formátumú. Ezzel sok kellemetlenségtől kíméljük meg magunkat.

1.1. Lefoglalt szavak

A VHDL kód elkészítése során utasításokat, kifejezéseket, operátorokat használunk. Ezek egyszerű szavak, melyek a fordító számára különleges jelentőséggel bírnak. Ezeket a „kifejezéseket” lefoglalt szavaknak nevezzük, melyek a következők:

```
abs             access          after           alias           all             and
architecture    array           assert          attribute       begin           block
body            buffer          bus             case            component
configuration   constant        disconnect      downto          else            elsif
end             entity          exit            file            for             function
generate        generic         group           guarded if      impure          in
inertial        inout           is              label           library         linkage
literal         loop            map             mod             nand            new
next            nor             not             null            of              on
open            or              others          out             package         port
postponed       procedure       process         protected       pure            range
record          register        reject          rem             report          return
rol             ror             select          severity        shared          signal
sla             sll             sra             srl             subtype         then
to              transport       type            unaffected      units           until
use             variable        wait            when            while           with
xnor            xor
```

A fenti kifejezések mellet használhatunk olyan olyan szavakat is, melyeket mi definiálunk. Ezek lehetnek például változók, számok, sőt megjegyzések is. A következő alfejezetek ezeket az általunk definiált kifejetéseket tárgyalják.

1.2. Megjegyzések

Ha a forráskódban megjegyzéseket, kommenteket szeretnénk elhelyezni, azt a következőképpen tehetjük meg:

```
… a line of VHDL description … -- a descriptive comment

vagy

-- The following code models
-- the control section of the system
… some VHDL code …
```

Általánosságban elmondható az, hogy a blokkok (fájl eleje, entity, architecture) elején célszerű összefoglalni, hogy az adott egység mit csinál, milyen bemenő és kimenő paraméterei vannak, mi a működési elve, stb. Ez megkönnyíti a későbbi felhasználást: néhány hét ... hónap ... év múlva már nem biztos, hogy szeretnénk az egész kódot átbogarászni, hogy vajon az a 200 sor vajon mivel mit csinál.

Ha az adott sorban valami nehezen érthető, nem szokványos dolog történik, mindenképpen érdemes egy rövid megjegyzések elmagyarázni az adott lépés működését. Ez azoknak lehet hasznos, akik sorról sorra meg akarják érteni kódunkat.

A megértést segíti, ha valamelyik világnyelv egyikén készítjük el a megjegyzést. Ma leginkább az angol számít általánosan elfogadottnak, így nem javaslom, hogy bárki is latin, ógörög, szanszkrit, burmai vagy páli nyelven kommentezze a kódját.

1.3. Azonosítók

Az azonosítókat akkor használjuk, amikor névvel látjuk el a kód bizonyos komponenseit, például az entitásokat, változókat, be-/kimeneti vezetékeket, stb.

Az azonosítók (nevek) képzésének szabályai a következők: Az azonosítók

- csak az ABC (értsd: angol ABC) betűit, számjegyeket és „_” jelet tartalmazhatják,
- mindig betűvel kezdődnek,
- nem végződhetnek „_” karakterrel
- nem tartalmazhatnak két egymást követő „_” karaktert,

- és nem nem lehetnek lefoglalt szavak (ld. fent).

A VHDL nyelv nem érzékeny a kis- és nagybetűkre (a Verilog nyelv érzékeny).

Mivel az Olvasó tanulta a PERL nyelvet, így érteni fogja a következő reguláris kifejezést is:

ID := [a-zA-Z]+(_?[a-zA-Z0-9]{1,})*

A `vim` editor ezt a formátumot támogatja (vegyük észre a kettő ekvivalenciáját):

ID := [a-zA-Z]\+\(_\?[a-zA-Z0-9]\{1,\}\)*

Azonosító lehet minden olyan karaktersorozat, melyre az előbbi reguláris kifejezések illeszkednek. Minden más azonosító hibás. Néhány példát mutat az azonosítók nevére a következő felsorolás:

```
A      X0     counter       Next_Value     generate_read_cycle   Maitreya
```

A következő felsorolás a hibás azonosítókat tartalmaz (rámutatva arra is, hogy miért hibásak):

```
last@value          -- contains an illegal character for an identifier
5bit_counter        -- starts with a nonalphabetic character
_A0                 -- starts with an underline
A0_                 -- ends with an underline
clock__pulse        -- two successive underlines
```

1.4. Számok

A VHDL referencia-leírás szerint a számok kétféle formát öltetnek: lehetnek egész literálok és valós számok. A kettő formailag abban különbözik, hogy a valós számok tartalmazhatnak „.”-ot (tizedespontot) is.

Egész számokra mutat példát a következő mintakód:

```
23              0               146
```

Valós számok tartalmazhatnak „.”-ot, de felírhatók normál alakban is:

```
23.1            0.0             3.14159
46E5            1E+12           19e00
1.234E09        98.6E+21        34.0e-08
```

1.5. Karakterek

Előfordulhat az is, hogy a VHDL kódban karakterkonstansokat is használni szeretnénk. A karakterek – csakúgy, mint PASCAL-ban – két aposztróf (') között kapnak helyet:

```
'A'             -- uppercase letter
'z'             -- lowercase letter
','             -- the punctuation character comma
'''             -- the punctuation character single quote
' '             -- the separator character space
```

1.6. Sztringek

A VHDL lehetővé teszi karaktersorozatok használatát is. Néhány szabályt szem előtt kell tartanunk, ha sztringeket szeretnénk használni:

- A karaktereket idézőjelek (”) közé kell tenni.
- Bármilyen karaktert tartalmazhatnak (beleértve a NULL karakteret is).
- Csak egy sorban helyezkedhetnek el a kódban.
- Ha egy sorban nem férnek el, akkor az összefűzés (&) operátort kell használni.

```
"A string"
"We can include any printing characters (e.g., &%@^*) in a string!!"
"00001111ZZZZ"
"" -- empty string
```

Összefűzésre mutat példát a következő mintakód:

```
"If a string will not fit on one line, "
& "then we can break it into parts on separate lines."
```

1.7. Bit-sztringek

Ha a kimenetek, alapértékek, stb. értékét nem decimálisan szeretnénk megadni, használhatunk bináris, oktális vagy hexadecimális formátumot is. Azért nevezik ezeket bit-sztringnek, mert minden számjegy közvetlenül egy vagy több bitet képvisel. A decimális formához képest annyi a különbség, hogy az érték „B”, „O” illetve „H” betűvel kezdődnek.

Lássunk néhány példát:

- Bináris forma:

```
B"0100011" B"10" b"1111_0010_0001" B""
```

- Oktális forma:

```
O"372"          -- equivalent to B"011_111_010"
o"00"           -- equivalent to B"000_000"
```

- Hexadecimális forma:

```
X"FA"           -- equivalent to B"1111_1010"
x"0d"           -- equivalent to B"0000_1101"
```

Ezzel befejeztük azon nyelvi elemek számbavételét, melyek szintaktikailag helyesek a VHDL kódban. Ebben a fejezetben megismertük azon kifejezéseket, melyekből felépíthetjük a működő kódunkat.

A következő részben azzal foglalkozunk, hogy a szintaktikai szabályoknak megfelelő alapelemekből hogyan készítsük el például a változóinkat, saját típusainkat, vagy az áramkörünk leírását.

2. Típusok a VHDL-ben

Ebben a részben azzal ismerkedünk meg, hogy az előző fejezetben említett lefoglalt szavakat hogyan rendeljük össze, hogyan használjuk a számokat, egyáltalán milyen típusú számok léteznek a VHDL nyelvben, hogyan tudunk vezérlési szerkezeteket használni, és a legfontosabb: hogyan írhatjuk le az elkészítendő áramkörünk felépítését, működését.

Ebben a részben már nem a nyelv legalsó szintjét tárgyaljuk, ez a szint leginkább a felhasználás szabályaival, a szemantikai elvekkel foglalkozik.

2.1. Skalár típusok

Érdekes definíciót ad a skalár típusok meghatározására a VHDL referenciakönyv: a skalár típus értékei oszthatatlanok. Talán egyszerűbb megmondani azt, hogy mi nem skalár, mint azt, hogy mi az: nem skalár a tömb és a sztring.

2.1.1. Egész számok és altípusai

Az egész szám (**integer**) a VHDL beépített típusa, -2147483648 és +2147483647 között vehet fel értéket. Látszik, hogy egy 32 bites, előjeles típusról van szó.

Altípusai a következők:

- **natural**: természetes számok, melyek a 0 és a legnagyobb egész közötti értékek lehetnek,
- **positive**: pozitív számok, melyek az 1 és a legnagyobb egész közötti értékek lehetnek,

Az egész számokra a következő operátorok (műveletek) értelmezettek:

:=	értékadás
+	összeadás
-	kivonás[1], vagy előjelváltás[2]
*****	szorzás[3]
/	(egész)osztás[4]

1 Mint kétoperandusú, bináris művelet.

2 Mint egyoperandusú, unáris művelet.

3 Nem minden esetben tud a fordító megfelelő kódot generálni.

4 Nem minden esetben tud a fordító megfelelő kódot generálni.

mod maradékképzés, a végeredmény előjele az osztóéval egyezik meg

rem maradékképzés, a végeredmény előjele az osztandóéval egyezik meg

abs abszolút érték

****** hatványozás (jobb oldali operandus nem lehet negatív)

Ha nem kívánjuk a teljes számábrázolási tartományt használni, célszerű (tényleg!) altípust létrehozni. Ezt a következőképpen tehetjük meg:

```
subtype small_int is integer range -128 to 127;
```

Ezzel létrehoztunk egy 8 bites előjeles típust a beépített *integer* típusból. Ha változót[5] szeretnénk deklarálni, így tehetjük meg a legegyszerűbben:

```
variable adjustment : integer;
variable deviation : small_int;
```

A következő mintakód az operátorok használatára mutat példát:

```
deviation := deviation + adjustment;
```

Látható, hogy az utasításokat „;”-val kell lezárni. Ha egy olyan változót szeretnénk, deklarálni, melynek értéke konstans, így tehetjük meg:

```
constant number_of_bytes : integer := 4;
```

Ez utóbbi akkor lehet hasznos, ha az áramkörünk valamilyen paraméterét nem szándékozunk „beledrótozni” a kódba.

5 A változó ugyanarra szolgál, mint a többi programnyelvben: adatokat tudunk úgy tárolni, hogy a tárol adat később bármikor lekérdezhető, megódosítható.

2.1.2. Lebegőpontos számok használata

A lebegőpontos számok (`real`) arra szolgálnak, hogy valós számokat képviseljenek a VHDL nyelvben. A lebegőpontos számokat a karakterisztika és mantissza értékükkel adhatjuk meg. A VHDL legtöbb implementációja az IEEE 64-bites, dupla pontosságú megvalósítását alkalmazza.

A lebegőpontos számok körében a következő műveletek értelmezettek: összeadás („+”), kivonás és előjelváltás („-”), szorzás („*”), osztás („/”), abszolút érték (abs) és hatványozás („**”).

Kétoperandusú műveletek esetén mindkét operandus típusa ugyanaz. Ez alól csak a hatványozás jelent kivételt, mert itt a kitevő mindig egy egész szám.

A lebegőpontos számok használatára mutat példát a következő kódrészlet:

```
constant e : real := 2.718281828;
variable sum, average, largest : real;
largest := e ** 2;
average := (largest + sum) / 2;
```

A lebegőpontos számok használatára elég ritkán szokott szükség lenni, ezért erre ne is fordítsunk több figyelmet.

2.1.3. Időzítési értékek

A VHDL lehetővé teszi, hogy a szimuláció során bizonyos események adott késleltetéssel kövessék egymást. Ezen késleltetési értékek megadására szolgál a `time` típus. Az időzítési értékek két részből állnak: a mérőszámból és a mértékegységből. A szóköz karakter használata kötelező a kettő között. Pl.:

```
5 ns          22 us          471.3 msec
```

A mértékegységek a következők lehetnek:

```
fs     ps     ns     us     ms     sec    min    hr
```

2.1.4. Felsorolás típus

Ezt a típust akkor használjuk, amikor felsorolásszerűen megmondjuk, hogy a változó milyen értékeket vehet fel. Például az oktális számjegyek csak a 0...7 számjegy--tartományból kerülhetnek ki:

```
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
```

Egy összetettebb példát mutat a következő kódrészlet:

```
type alu_function is (disable, pass, add, subtract, multiply, divide);

variable alu_op : alu_function;

alu_op := subtract;
```

A felsorolás típust igen gyakran használjuk például logikai szintek definiálására, lehetséges válaszok, hibaértékek megadására, stb.

2.1.5. Karakter típus

A karakter (`character`) egy olyan felsorolás típus, amely az ISO 8859-1 kódlap 8 bites karaktereit tartalmazza: ezek lehetnek betűk, számok, vezérlő karakterek[6], stb. A karakter definíciója a következő:

```
type character is (
       nul, soh, stx, etx, eot, enq, ack, bel,
       bs, ht, lf, vt, ff, cr, so, si,
       dle, dc1, dc2, dc3, dc4, nak, syn, etb,
       can, em, sub, esc, fsp, gsp, rsp, usp,
       ' ', '!', '"', '#', '$', '%', '&', ''',
       '(', ')', '*', '+', ',', '-', '.', '/',
       '0', '1', '2', '3', '4', '5', '6', '7',
       '8', '9', ':', ';', '<', '=', '>', '?',
       '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
       'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
       'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
       'X', 'Y', 'Z', '[', '\', ']', '^', '_',
       '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
       'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
```

6 Soremelés, lapdobás, sípolás, stb.

```
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', del,
        c128, c129, c130, c131, c132, c133, c134, c135,
        c136, c137, c138, c139, c140, c141, c142, c143,
        c144, c145, c146, c147, c148, c149, c150, c151,
        c152, c153, c154, c155, c156, c157, c158, c159,
        ' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
        '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
        '°', '±', '²', '³', '´', 'µ', '¶', '·',
        '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
        'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
        'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
        'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
        'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
        'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
        'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
        'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
        'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ'
);
```

A karakter típust így tudjuk a VHDL forrásban használni:

```
variable cmd_char, terminator : character;

cmd_char := 'P';
terminator := cr;
```

2.1.6. Logikai típus (boolean)

A logikai műveletek (összehasonlítás, ÉS, VAGY, negálás) eredményét logikai típusú változóban tárolhatjuk. Az ilyen típusú eredményeket általában vezérlési célokra szoktuk használni.

A logikai típus definíciója rendkívül egyszerű, mivel az értékkészlete mindössze két elemet tartalmaz: igaz (true) és hamis (false).

```
type boolean is (false, true);
```

```
Az eredmény true:
       123 = 123, 'A' = 'A', 7 ns = 7 ns

Az eredmény false:
       123 = 456, 'A' = 'z', 7 ns = 2 us
```

Az összehasonlító operátorok, melyek eredménye logikai típusú, a következők:

<	kisebb
<=	kisebb vagy egyenlő
>	nagyobb
>=	nagyobb vagy egyenlő
=	egyenlő
/=	nem egyenlő

A logikai műveletek operandusai logikai típusúak, és a műveletek eredménye is logikai típusú:

and	logikai ÉS	a > 12 **and** b = bits
or	logikai VAGY	a > 12 **or** b = bits
nand	logikai NEM-ÉS	a > 12 **nand** b = bits
nor	logikai NEM-VAGY	a > 12 **nor** b = bits
xor	logikai kizáró-VAGY	a > 12 **xor** b = bits
xnor	logikai ekvivalencia	a > 12 **xnor** b = bits
not	logikai negálás	**not** b = bits

Természetesen a (már előző félévekben tanult) De Morgan szabályok segítségével az előbb felsorolt operátorok helyettesíthetők egymással.

2.1.7. Bitek

Mivel a VHDL nyelv digitális rendszerek leírására használatos, célszerű, hogy legyen olyan adattípusa, amely bit-értékeket reprezentál. Erre a célra az előre definiált `bit` típust használhatjuk:

```
type bit is ('0', '1');
```

Az előző alfejezetben említett logikai operátorok a `bit` típussal is használhatók, ebben az esetben az operandusok és az eredmény is `bit` típusú.

A `logikai típus` és a `bit` között az a különbség, hogy az előbbi elsősorban absztrakt értékekkel dolgozik, míg a `bit` hardver szintekkel.

2.1.8. Standard Logic

Az IEEE szabványosított egy `std_logic_1164` nevű csomagot. Ez a csomag tartalmaz egy `std_ulogic` nevű felsorolt típust, mely lefedi a CPLD-k, FPGA-k jelvezetékeinek logikai állapotát. Az említett típust így definiálták:

```
type std_ulogic is (
    'U',       -- Uninitialized
    'X',       -- Forcing unknown
    '0',       -- Forcing zero
    '1',       -- Forcing one
    'Z',       -- High impedance
    'W',       -- Weak unknown
    'L',       -- Weak zero
    'H',       -- Weak one
    '-'        -- Don't care
);
```

Látható, hogy egy kimenet értéke nem csak logikai „0” vagy „1” lehet, hanem ismeretlen, nem inicializált, határozott „0”, határozott „1”, nagyimpedanciás, gyenge ismeretlen, gyenge „0”, gyenge „1”, és don't care értéket is felvehet.

Ez azt jelenti, hogy definiálhatunk olyan jelvezetékeket is, melyet aktív áramkör hajt meg, vagy amit felhúzó / lehúzó ellenállás állít be valamilyen értékre.

Mivel a `std_ulogic` típus nem része a VHDL nyelvnek, hanem egy külső csomagban kapott helyet, ezért használat előtt be építeni (`include`) a kódunkba:

```
library ieee;
use ieee.std_logic_1164.all;
```

Ha ezt megtesszük, pontosan ugyanúgy használhatjuk az `std_ulogic` típust, mint bármelyik másikat (beleértve a bit típust is).

2.2. Tömb

Ha azonos típusú adatok halmazát szeretnénk használni, akkor célszerű tömböt definiálni, melyet így hozhatunk létre:

```
array ( natural to|downto natural ) of element_subtype_indication;
```

Például egy adatbuszt, címbuszt, stb. lehetőség szerint nem különálló vezetékekből építünk meg, hanem „azonos típusú adatokat”, azaz biteket fogunk össze, és az eredményt **busz**nak nevezzük. A busz definíciója ilyen formát ölthet:

```
type bus8 is array (31 downto 0) of bit;
```

A `bit` helyett természetesen az `std_logic` is használható. A buszt nem csak vezetékezésre használhatjuk, hanem értékek tárolására is. Ekkor visszajutunk a „hagyományos” változókhoz, melyek ugyancsak bitekből épülnek fel.

Vezetékezésre mindig buszt használjunk, különben az eredmény számos meglepetést okozhat.

Az előbb egy tömb típust hoztunk létre, melyből úgy lesz változónk (azaz tényleges buszunk), ha példányosítjuk azt:

```
variable buffer_register, data_bus : bus8;
```

A tömb elemeinek így adhatunk értéket:

```
data_bus(0) := '1';
```

A bitvektorokat (vagyis bitek tömbjét) olyan gyakran használjuk, hogy a VHDL nyelv kitalálói külön típust hoztak létre:

```
type bit_vector is array (natural range <>) of bit;
```

```
type std_ulogic_vector is array ( natural range <> ) of std_ulogic;
```

Azért nagyon kényelmes ez a megoldás, mert egy áramkör kimenetét-bemenetét definiálhatjuk `std_ulogic_vector`-nak, vagyis olyan busznak, melyen keresztül egyszerre több bitnyi információ cserél gazdát.

Lássunk egy példát (működésének pontos magyarázatára még kissé várnia kell az Olvasónak):

```
entity ADDER is
    generic(n: natural :=2 );
    port(
        A:      in std_logic_vector(n-1 downto 0);
        B:      in std_logic_vector(n-1 downto 0);
        carry:  out std_logic;
        sum:    out std_logic_vector(n-1 downto 0)
    );
end ADDER;
```

Az előbbi példa egy n = 2 bites (a `generic` olyan, mint a C nyelvben a `#define`, vagyis szimbólumkonstanst hozhatunk létre) összeadó kimeneteinek és bemeneteinek definícióját mutatja: az `A` és a `B` `n` bites a bemenet, a `sum` pedig `n` bites kimenet. Ezen kívül megadtunk egy `carry` kimenet is, amely egy „közönséges” bit.

Ebben a részen áttekintettük a számunkra feltétlenül szükséges típusokat. A következő részben a vezérlési szerkezetekkel foglalkozunk.

3. Vezérlési szerkezetek

Ebben a fejezetben arról lesz szó, hogy hogyan tudjuk a kód futását a processzen[7] belül befolyásolni, hogyan tudunk feltételes végrehajtást definiálni, és hogyan tudunk ciklusokat létrehozni.

Az eddig bemutatott példákban csak az értékadás operátort használtuk („:=”), amely feltétel nélkül végrehajtódott. Ha egy utasítást vagy egy utasításcsoportot csak bizonyos feltételek teljesülése esetén akarunk végrehajtatni, akkor – csakúgy, mint minden más programnyelvben – az `if-then-else` szerkezetet használjuk.

Ha egy paraméter értékétől függően más-más utasítássorozatot kell végrehajtani, akkor a `case-when` szerkezetet célszerű használni.

Ha egy feladatot több, mint egyszer szeretnénk elvégeztetni, akkor célszerű azt ciklusba szervezni. Hogy a bemutatásra kerülő három ciklus közül mikor melyiket érdemes használni, arra vonatkozóan az Olvasó a ciklusutasítás részletes leírásánál talál majd útmutatást.

3.1. Feltételes végrehajtás

Ha egy utasítássorozatot csak bizonyos feltétel(ek) teljesülése esetén szeretnénk végrehajtatni, akkor feltételes utasításokat kell használnunk. A feltételes végrehajtás kétféleképpen történhet: az első megoldás csak két lehetséges „útvonalat” tesz lehetővé (ha … akkor … egyébként …), míg a másik megoldás több lehetőség közül enged választani (ha ez, akkor …, ha az, akkor …, ha amaz, akkor pedig …).

3.1.1. If – elágazás

Ha az utasítássorozat végrehajtását logikai feltételhez kívánjuk kötni, az `if-then-else` szerkezetet használjuk. Szintaktikája a következő:

```
if boolean_expression then
    { sequential_statement }
{ elsif boolean_expression then
    { sequential_statement } }
```

7 Lásd: később. A processz egy szekvenciálisan végrehajtott utasításlista, hasonlít a „klasszikus” programnyelvek függvényeihez, eljárásaihoz, csak a VHDL processzei egymással teljesen párhuzamosan futnak.

```
[ else
    { sequential_statement } ]
end if;
```

Mivel az előbbi forrás alapján elég nehéz megérteni az utasítás szintaktikáját, nézzünk néhány egyszerű kódrészletet:

```
if en = '1' then
    stored_value := data_in;
end if;
```

Vagyis ha az `en` értéke '1', akkor a `stored_value` legyen egyenlő `data_in`-nel.

```
if sel = 0 then
    result <= input_0;        -- executed if sel = 0
else
    result <= input_1;        -- executed if sel /= 0
end if;
```

Ha `sel` értéke 0, akkor a `result` értéke legyen `input_0` értékével megegyező, egyébként `result` legyen egyenlő `input_1` értékével.

Figyeljük meg, hogy ebben a példában a szokásos „**:=**” operátor helyett a „**<=**” operátort használtuk. A kettő között az eltérés mindössze annyi, hogy az előbbivel változónak adunk értéket (a változó nincsen közvetlenül kivezetve az FPGA egyik lábára sem), míg az utóbbival jelvezetéknek (ami esetleg ki van vezetve az FPGA valamelyik lábára, vagy egy másik modul bemenetére „van kötve”).

Egy bonyolultabb példát mutat a következő kódrészlet. Ebben a mintakódban bemutatásra kerül a (már előzőleg említett) logikai operátorok használata is:

```
if mode = immediate then
    operand := immed_operand;
elsif opcode = load or opcode = add or opcode = subtract then
    operand := memory_operand;
else
    operand := address_operand;
end if;
```

Ettől bonyolultabb feltételekkel nem fogunk találkozni a VHDL nyelv megismerése során.

3.1.2. Case – többirányú elágazás

Ezt a megoldást akkor használjuk, amikor egy változó aktuális értéke alapján választjuk ki, hogy a lehetséges utasítássorozatok (általában több, mint 2 lehetőség) közül melyiket kell végrehajtani. Szintaktikája a következő:

```
case expression is
    ( when choices => { sequential_statement } )
    { … }
end case;
```

Nézzünk példát erre is:

```
type alu_func is (pass1, pass2, add, subtract);

…

case func is
    when pass1 =>
       result := operand1;
    when pass2 =>
       result := operand2;
    when add =>
       result := operand1 + operand2;
    when subtract =>
       result := operand1 - operand2;
    when others =>
       result := -1;
end case;
```

Az előző példa egy nagyon egyszerű ALU-t valósít meg: ha a `func` paraméter értéke arra ad utasítást, hogy az áramkör engedje át valamelyik operandus értékét (`pass1`, `pass2`), akkor az eredmény (`result`) a megfelelő operandus értékét veszi fel. Ha a `func` paraméterrel az összeadást választjuk ki (`add`), akkor az eredmény a két operandus összege (`operand1 + operand2`) lesz, míg kivonás (`subtract`) esetén az eredmény a két operandus különbségét (`operand1 - operand2`) veszi fel.

Minden más esetben (`when others`) az eredmény -1.

3.2. Ciklusok

Ha egy feladatot többször kell végrehajtani, ciklust célszerű használni. Ebben a részben arról lesz szó, hogy a háromféle megoldás közül az adott feladathoz melyiket válasszuk ki.

3.2.1. Loop ciklus

Ez a VHDL legegyszerűbb ciklusa. A ciklustörzset feltétel nélkül, folyamatosan, vég nélkül ismétli. Mivel ez tulajdonképpen egy végtelen ciklus, szinte biztos, hogy létezik egy megfelelő utasítás, amely segítségével ki lehet lépni belőle.

A `loop` ciklus szintaktikája a következő:

```
loop
    { sequential_statement }
end loop;
```

A `sequential_statement` jelen esetben a ciklustörzset jelenti.

Nézzük most az `exit`-tel kiegészített `loop` szintaktikáját:

```
loop
    { sequential_statement }
    [ label : ] exit [ loop_label ] [ when boolean_expression ] ;
end loop;
```

Mivel az előbbi szintaktikai definíciók nehezen érthetők, nézzük meg, hogy a gyakorlatban hogyan használhatjuk a ciklusokat. Természetesen van mód arra, hogy megszakítsuk a ciklus futását. Erre szolgál az `exit` utasítás, melynek végrehajtását feltételhez is köthetjük:

```
loop
    count_value := (count_value + 1) mod 16;
    count <= count_value;
    exit when reset = '1';
end loop;
```

```
if condition then
    exit;
end if;
```

3.2.2. While ciklus

Az előző megoldásnak egy kifinomultabb változatát valósítja meg a `while` ciklus. Szintaktikáját a következő kódrészlet mutatja:

```
while boolean_expression loop
    { sequential_statement }
end loop;
```

A CPLD/FPGA addig hajtja végre a ciklustörzset, amíg a `boolean_expression` logikai kifejezés igaz értékű. Látható, hogy ez egy elöltesztelő ciklus.

Ennek a szintaktikája könnyebben érthető, mégis úgy gondolom, hogy érdemes néhány példát nézni a `while` ciklus használatára is:

$$\cos\phi = 1 - \frac{\phi^2}{2!} + \frac{\phi^4}{4!} - \frac{\phi^6}{6!} \ldots$$

```
variable sum, term : real;
variable n : natural;

...

sum := 1.0;
term := 1.0;
n := 0;

while abs term > abs (sum / 1.0E6) loop
    n := n + 2;
    term := (-term) * phi**2 / real(((n-1) * n));
    sum := sum + term;
end loop;

result <= sum;
```

Ezt a ciklust előszeretettel használjuk, ezért megismerését mindenképpen ajánlom az Olvasó számára.

3.2.3. For ciklus

Ha előre tudjuk, hogy a ciklustörzset véges sokszor kell végrehajtani, akkor `for` ciklust használunk. Szintaktikája a következő:

```
for identifier in discrete_range loop
    { sequential_statement }
end loop;
```

Az `identifier` paraméter egy olyan egész szám szerepét tölti be, amely minden iterációban eggyel nő vagy csökken a `discrete_range`-től függően.

A `discrete_range`-t a következőképpen értelmezzük:

```
simple_expression ( to | downto ) simple_expression
```

A példaprogramban az FPGA 128-szor hajtja végre a ciklustörzset:

```
for count_value in 0 to 127 loop
    count_out <= count_value;
    wait for 5 ns;
end loop;
```

A fenti kódrészlet hatására az FPGA 0 és 127 között számol, egyesével növelne a `counter_value` értékét, és minden iterációban elvégez egy értékadást és egy 5 ns-os késleltetést (csak a szimulátorban van szerepe a késleltetésnek).

Lehetőség van arra is, hogy a ciklusváltozó értéke ne felfelé, hanem lefelé változzon. Ekkor a `to` helyett a `downto` lefoglalt szót kell alkalmazni, és a tartomány két végpontját fel kell cserélni.

3.3. Assert

Egy érdekes feltételes elágazást tesz lehetővé az `assert` utasítás. Ha a paraméterként kapott boolean (logikai) kifejezés nem igaz, akkor végrehajtja a törzset, ami gyakran egy kiíró művelet szokott lenni. De hova íródik ki szöveg? - kérdezheti bárki. Nos, ez a lehetőség akkor hasznos, ha a szimulátor segítségével teszteljük a kódunkat.

Az `assert` szintaktikája a következő:

```
assert boolean_expression
[ report expression ] [ severity expression ] ;
```

A mintaprogram ellenőrzést végez, és a hibát jelzi azt a szimulátor segítségével:

```
assert initial_value <= max_value
    report "initial value too large";
```

Vagyis ha az `initial_value` értéke nagyobb, mint a `max_value`, akkor a szimulátorban egy hibaüzenet jelenik meg. Ezt a lehetőséget azért tette lehetővé a VHDL megalkotója, hogy megkönnyítse a nyomkövetést-hibafelderítést.

Ebben a fejezetben megismertük a VHDL nyelv vezérlési szerkezeteit, és azok felhasználását. A következő fejezetben azt nézzük meg, hogy hogyan tudjuk az áramkörünk működését leírni, felépítését meghatározni.

4. A hardver leírása

Ha egy áramkört vagy áramkörrészt szeretnénk leírni VHDL nyelven, először meg kell határoznunk a kimeneti-bemeneti összeállítását[8], majd le kell írnunk az áramkör működését[9]. Ha ezt megtesszük, még nem jön létre automatikusan semmilyen áramkör, az előbbi lépésekkel csak egy „tervrajzot” készítettünk. Ahhoz, hogy egy ténylegesen működő, fizikailag megvalósított áramkört kapjunk, **példányosítani** kell az előbb létrehozott tervrajzot. Természetesen egy terv (**entity** + **architecture**) alapján több fizikai áramkör is létrehozható.

Ha leírtuk az áramkörünket, és példányosítottuk is azt, akkor még egyáltalán nem vagyunk készen. Valahogy le is kell tesztelnünk a kész rendszert: az egyik lehetőség, hogy lefordítjuk a forrást, az eredményt beletöltjük az CPLD-be, FPGA-ba, és különböző kapcsolókkal, LED-ekkel végigpróbálgatjuk az áramkör összes állapotát. Ezt egyszer még csakcsak megcsinálja az ember, de ha hibásan működik az áramkör, és javítani kell a forráson, akkor másodszorra már nem szívesen csinálja meg senki.

Ezért találták ki a VHDL megalkotói az „automatikus” tesztelést: létrehozunk egy olyan áramkört is, amely teszteli az elkészíteni kívánt fekete dobozunkat. Ez dupla munkának tűnik, de ha elkezdünk dolgozni, akkor elég hamar kiderül, hogy enélkül nem nagyon megyünk semmire. Sőt, a második esetben (automatizált tesztelés) az ellenőrzést elvégezhetjük a szimulátorban, ahol használhatjuk az `assert` utasítást, így nagy biztonsággal végigkövetkeztetjük az áramkörünk működését.

A szimulátor természetesen lehetővé teszi, hogy a kimenetek-menetek állapotát minden időpillanatban lássuk, és akár el is mentsük későbbi felhasználásra.

4.1. Entity – a bemenetek és kimenetek definiálása

A bevezetőben már említettem, hogy ez a rész arra szolgál, hogy definiáljuk az áramkör interfészét, vagyis azt, hogy az eszközünk milyen vezetékeken keresztül kapcsolódik a „külvilághoz”. A külvilág ebben az esetben nem jelent fizikai kivezetést az FPGA lábaira.

Az `entity` definiálása a következő szintaktika szerint történik:

8 Entity: tulajdonképpen ez az áramkör (mint „fekete doboz”) kapcsainak definiálása.

9 Architecture: megadjuk, hogy pontosan mit is csináljon az a bizonyos fekete doboz.

```
entity identifier is
    [ port ( port_interface_list ) ; ]
end [ entity ] [ identifier ] ;

interface_list ←
    ( identifier { , … } : [ mode ] subtype_indication ) { ; … }

mode ← in | out | inout
```

Lássuk egy konkrét példán keresztül, hogy miként lehet az áramkörrészlet kivezetéseit definiálni: a mintaprogram egy összeadó leírását mutatja, melynek 2 bemenete (**in**: `a` és `b`) és egy kimenete (**out**: `sum`) van:

```
entity adder is
    port (
        a, b : in word;
        sum  : out word
    );
end entity adder;
```

Látható, hogy az entitást el kell nevezni, hogy később hivatkozni tudjuk rá (például amikor leírjuk a működését, vagy amikor példányosítjuk). Az előbbi példában előre meg kell mondanunk, hogy mit értünk `word` alatt, erre a saját típusok definiálásánál már láttunk példát.

Az **inout** azt jelenti, hogy az adott jelvezeték lehet akár bemenet, akár kimenet, a felhasználás módjától függően.

Lássunk még egy példát: tekintsünk egy olyan D-tárolót, mely egy adatbemenettel (`D`), egy órajelbemenettel (`clk`), egy aszinkron törlő bemenettel (`clr`) és egy adatkimenettel (`Q`) rendelkezik:

```
entity edge_triggered_Dff is
    port (
        D : in bit;
        clk : in bit;
        clr : in bit;
        Q : out bit
    );
end entity edge_triggered_Dff;
```

Azt hiszem, hogy ennyi magyarázat elegendő a megértéshez. A következő részben arról lesz szó, hogy hogyan definiálhatjuk az áramkör működését.

4.2. Architecture – a működés leírása

Amint az előző fejezetben már olvashattuk, ebben a részben arról lesz szó, hogy hogyan határozhatjuk meg az áramkör működését.

Mire idáig eljut az Olvasó, elvileg tisztában van a VHDL nyelv alapelemeivel (A VHDL nyelvi elemei), a használható skalár és nem skalár típusokkal (Típusok a VHDL-ben), ismeri a változók deklarálásának módját és a vezérlési szerkezeteket (Vezérlési szerkezetek). Mindez szükséges ahhoz, hogy pontosan leírjuk az áramkörünk működését:

```
architecture identifier of entity_name is
    { block_declarative_item }
begin
    { concurrent_statement }
end [ architecture ] [ identifier ] ;
```

Meglehetősen nehéz az előbbi leírásból kitalálni, hogy miként is tudunk működő modellt előállítani. Hogy jobban megértsük a VHDL logikáját, nézzünk egy egyszerű példaprogramot:

```
library ieee;
use ieee.std_logic_1164.all;

entity Driver is
    port(
        x: in std_logic;
        F: out std_logic
    );
end Driver;

architecture behav of Driver is
begin
    F <= x;
end behav;
```

A mintaprogram alapján az FPGA az `x` bemenet értékét az `F` kimenetre másolja (esetleg teljesítményerősítést végezve, de ez hardverfüggő!). Az elkészült áramkört drivernek, azaz meghajtónak hívjuk.

Ettől bonyolultabb példát most nem nézünk, ha az Olvasó összetettebb mintaprogramokkal szeretne megismerkedni, akkor ajánlom a Mintafeladatok című fejezet áttanulmányozását.

4.2.1. Processzek

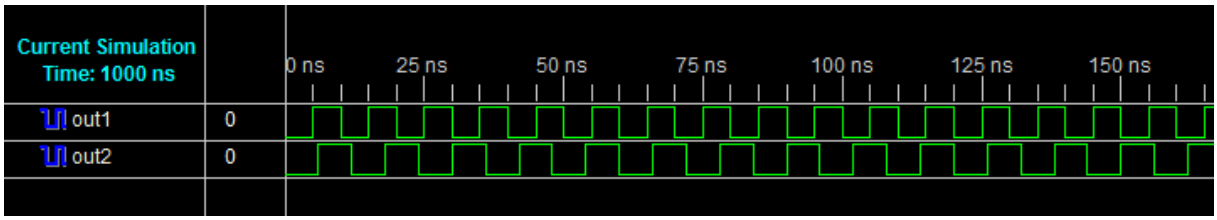
Az FPGA-t, CPLD-t úgy nem úgy kell elképzelni, mint egy mikroprocesszort, amely sorban hajtja végre az utasításokat. Az FPGA a kitűzött feladatokat párhuzamosan végzi el. Ezek a feladatok vagy nagyon egyszerűek, kombinációs hálózat jellegűek (mint az előző példában, amely a bemenet értékét egyszerűen a kimenetre „másolta”), vagy szekvenciális, sorrendi folyamatok.

A szekvenciális folyamatokat processzeknek nevezzük. A processzek ugyan „programok”, de egymással párhuzamosan működnek. Ez a hardver fizikai felépítéséből adódik.

Lássunk egy példát két, egymással párhuzamos folyamatra:

```
library ieee;
use ieee.std_logic_1164.all;

entity Blinky is
    port(
        out1, out2: out std_logic
    );
end Blinky;

architecture behav of Blinky is
begin

    Clock0 : process is
    begin
        out1 <= '0';
        wait for 5 ns;
        out1 <= '1';
        wait for 5 ns;
    end process;

    Clock1 : process is
    begin
        out2 <= '0';
        wait for 6 ns;
        out2 <= '1';
        wait for 6 ns;
    end process;

end behav;
```

Ebben a példában két processz dolgozik párhuzamosan: az áramköri elemünk egy-egy lábán szolgáltatnak előbb logikai 0-t, majd logikai 1-et. A késleltetés értékét is beállíthatjuk, de ez az érték csak a szimulátorban fog pontos értéket jelenteni.



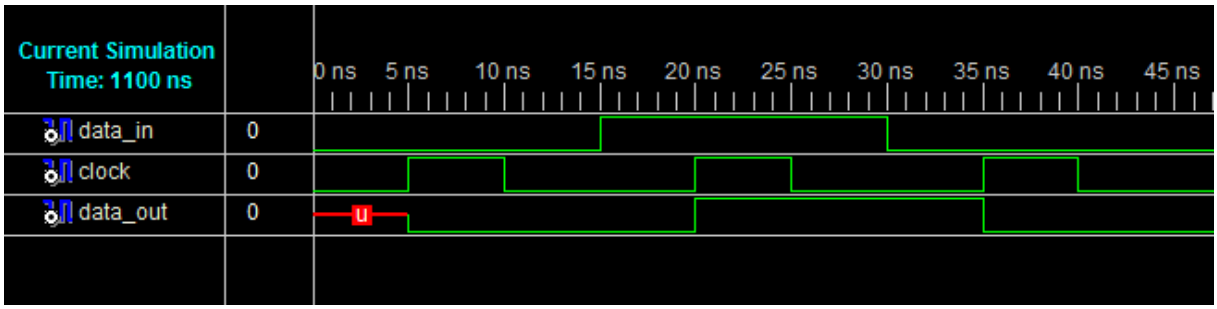
A valóságban mindig arra van szükség, hogy az áramkör a bemeneten megjelenő változásra reagáljon, ne pedig „saját kedve” szerint generáljon kimenő jelet. Ezt úgy oldjuk meg, hogy definiáljuk a bemenő jeleket (mint ahogyan azt már megismertük), és megadjuk, hogy ezen jelek **megváltozásakor** az FPGA kezdje el végrehajtani a processz törzsét. Ez leginkább a mikroprocesszorok megszakításaira hasonlít.

Van lehetőség arra is, hogy egy processz több jel változásakor aktivizálódjon. Ugyanígy egy jel több processzt is aktívvá tehet. Az aktívvá válás azt jelenti, hogy a processz definiálásakor megadott utasítások egymás után, sorban végrehajtásra kerülnek, majd a processz újra várakozó állapotba kerül, amíg a megfelelő jel újra meg nem változik.

Vegyünk példaként egy szinkron D-tárolót. Definíció szerint amikor az órajelbemenetére felfutó él érkezik, az adatbemenet értékét mintavételezi, és azt a kimenetre másolja. Tulajdonképpen az 4.2. fejezetben elkészített aszinkron meghajtót tettük szinkronná:

```
library ieee ;
use ieee.std_logic_1164.all;
use work.all;

-----------------------------------------------

entity dff is
port(
    data_in:   in  std_logic;
    clock:     in  std_logic;
    data_out:  out std_logic
);
end dff;

-----------------------------------------------

architecture behav of dff is
begin
    process(clock)
    begin
        -- clock rising edge
        if (clock = '1') then
            data_out <= data_in;
        end if;
    end process;
end behav;
```

A processznek adott paraméter (az úgynevezett szenzitív lista) azt mutatja, hogy mely jelek változása aktiválja a processzt. Ebben az esetben ez az órajel (`clock`) lesz az egyetlen ilyen jelvezeték. Ha élváltozás történik, és az órajel '1' értékű (vagyis az él „fel-felé”, 0-ról 1-re változott), akkor a bemenetet a kimenetre kell másolni.



A piros „U” betű azt jelenti, hogy még nincsen inicializálva a kimenet.

Felmerülhet a kérdés: mi történik akkor, ha olyan számlálót csinálunk, melynek két bemenete van: egy felfelé számláló és egy lefelé számláló. Mindkét bemenet felfutó élre érzékeny, és órajel jellegű. Mi történik, ha két processzt definiálok, és mindkettő ugyanazt a kimeneti értéket akarja módosítani. Elvileg a processzek egymással párhuzamosan működnek, tehát egyszerre férnek hozzá a kimenetekhez, bemenetekhez, és egyszerre küldhetnek ellentétes tartalmú parancsot a kimenetre. Nos, a megoldás egyszerűbb, mint gondolnánk: a fordító ezt nem hagyja. Ilyenkor olyan processzt (egyet) kell definiálni, melynek a szenzitív listájában (vesszővel elválasztva) mindkét bemenő jel szerepel. Azt természetesen meg kell vizsgálni, hogy melyik bemenet változott meg. Mivel a processz szekvenciálisan fut le, így nem lehetséges, hogy egyszerre két különböző érték kerüljön a kimenetre. Az persze előfordulhat, hogy néhány nanoszekundum különbséggel váltja egymást az első és a második érték. Ehhez hasonló problémára mutat majd megoldást a 5. fejezet.

A szenzitív listán kívül létezik még egy módszer, melynek segítségével a processzünk futását megállíthatjuk, és jelváltozásra várakozhatunk: ez pedig a `wait` utasítás:

```
wait [ on signal_name { , … } ]
    [ until boolean_expression ]
    [ for time_expression ] ;
```

Nézzünk példát a használatára:

```
half_add : process is
begin
    sum <= a xor b after T_pd;
    carry <= a and b after T_pd;
    wait on a, b;
end process half_add;
```

Ebben az esetben megakad a végrehajtás a `wait` utasításnál, egészen addig, amíg `a` vagy `b` meg nem változik. Ha ez megtörténne, frissül a `sum` és a `carry` értéke, majd ismét várakozás következik. Ez a `wait` legfontosabb felhasználása.

Használhatjuk a `wait`-et arra is, hogy felfüggesszük a végrehajtást, amíg egy feltétel **teljesül**:

```
wait until clk = '0';
```

A `wait` önmagában is állhat, ekkor a futás hátralevő idejére megállítja a processz végrehajtását, úgy is mondhatnánk, hogy a processz lefagy (jó értelemben véve).

4.3. Példányosítás

Az előzőekben két fontos fogalommal ismerkedtünk meg: az entitással és architektúrával. Az előbbi arra szolgál, hogy definiáljuk: egy „ fekete doboznak” (pl. egy NAND kapunak vagy egy számlálónak) milyen kivezetései vannak, az architektúra segítségével pedig a működését írjuk le. Ez még kevés ahhoz, hogy ténylegesen létrejöjjön bármiféle működő áramkör.

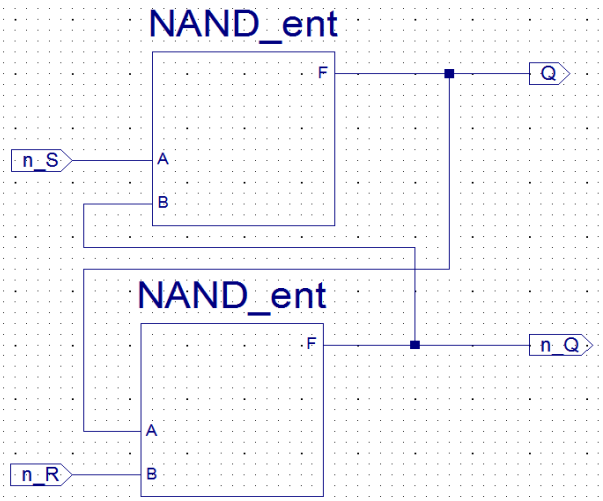
Egy elvi kapcsolási rajz még nem jelenti az áramkör megépítését, de egy elvi rajz alapján meg tudjuk építeni az áramkört. Sőt, akár többet is.

A példányosítás arra szolgál, hogy az entitással és architektúrával jellemzett VHDL elemből fizikailag megvalósított áramkörrészt készítsünk. Ha definiáljuk a NAND kaput az entitással és az architektúrával, akkor két NAND kapu segítségével építhetünk egy inverz RS tárolót:

```
library ieee;
use ieee.std_logic_1164.all;

----------------------------------------
```

```
entity NAND_ent is
port(
    A: in std_logic;
    B: in std_logic;
    F: out std_logic
);
end NAND_ent;

----------------------------------------

architecture behv of NAND_ent is
begin
    F <= A nand B;
end behv;

----------------------------------------
```

Az ábrán (pusztán emlékeztetőül) az inverz RS tároló elvi rajza látható:



Ha az előbb definiált NAND kapu segítségével inverz RS tárolót szeretnénk építeni, így tehetjük meg azt:

```
library ieee;
use ieee.std_logic_1164.all;

entity inv_rs is
    port (
        n_R : in    std_logic;
        n_S : in    std_logic;
```

```
        n_Q : out   std_logic;
        Q   : out   std_logic
    );
end inv_rs;
```

Ezzel leírtuk a NAND kaput, mint áramköri elemet. Lássuk, hogyan tudunk NAND kapuból építkezni:

```
architecture BEHAVIORAL of inv_rs is
    component NAND_ent
        port (
            F : out   std_logic;
            A : in    std_logic;
            B : in    std_logic
        );
    end component;

begin
    NAND1 : NAND_ent
        port map (
            A => n_S,
            B => n_Q,
            F => Q
        );

    NAND2 : NAND_ent
        port map (
              A => Q,
              B => n_R,
              F => n_Q
        );
end BEHAVIORAL;
```

Az inverz RS tárolónk is egy újabb fekete doboz lesz. Vagyis azt mondhatjuk, hogy mostantól kezdve az `inv_RS` egy újabb komponens, amit ugyanúgy használhatunk, mint bármilyen, eddig definiált elemet. A tárolónak is vannak bemenetei, kimenetei, ezek most egyszerű vezetékek, de lehetnének buszok (bitek tömbje, pl. `Q: out std_logic_vector(n-1 downto 0)`) is.

Mint minden VHDL doboznak, ennek is meg kell határoznunk a viselkedését. Először kijelentjük, hogy használni szeretnénk a `NAND_ent` **komponenst** (amit egy másik VHDL fájlban már megírtunk, vagy könyvtárból beemeltünk), melynek vannak bemenetei, kimenetei...

A viselkedési modellben szükségünk van két `NAND_ent` típusú áramköri elemre (komponensre) (`NAND1`, `NAND2`), melyeket úgy huzalozunk össze (`port map`), hogy `NAND1.A` legyen a tároló `S` bemenete, `NAND1.B` legyen a `Q` kimenet (ami egyúttal a `NAND2.F` is),

NAND2.A legyen a tároló Q kimenete (ami egyúttal a NAND2.F is), NAND1.B legyen a tároló R bemenete. Pontosan úgy, ahogy a rajzon is látható.

Ha olyan bonyolult a működési modellünk, hogy vezetékeket kell használnunk, azt is megtehetjük (ezt az eredeti inv_rs-en keresztül mutatom be, eredetileg ezt generálta a Xilinx ISE WebPack):

```
library ieee;
use ieee.std_logic_1164.ALL;

entity sch is
    port (
        n_R : in    std_logic;
        n_S : in    std_logic;
        n_Q : out   std_logic;
        Q   : out   std_logic
    );
end sch;

architecture BEHAVIORAL of sch is
    signal Q_DUMMY   : std_logic;
    signal n_Q_DUMMY : std_logic;
    component NAND_ent
        port (
            F : out   std_logic;
            A : in    std_logic;
            B : in    std_logic
        );
    end component;

begin
    n_Q <= n_Q_DUMMY;
    Q <= Q_DUMMY;

    NAND1 : NAND_ent
        port map (
            A => n_S,
            B => n_Q_DUMMY,
            F => Q_DUMMY
        );

    NAND2 : NAND_ent
        port map (
            A => Q_DUMMY,
            B => n_R,
            F => n_Q_DUMMY
        );

end BEHAVIORAL;
```

Ebben a példában két vezetéket (signal) használtunk, melyek ugyanúgy használhatók, mint a változók. Valóban akkor lehet rá szükségünk, ha az áramkörünk bonyolult felépítésű.

4.4. Test bench

Az előző fejezetek alapján már pontosan tudjuk definiálni, hogy hogyan működjön az áramkörünk, sőt, összetettebb áramköri részeket is össze tudunk rakni egyszerűbb komponensekből, de még mindig nem lehetünk teljesen biztosak abban, hogy az elkészített kódunk valóban jól működik.

Hogy megbizonyosodjunk az VHDL kódunk működőképességéről, egy olyan áramkört kellene létrehozni, amely le tudja tesztelni a kódunkat. Ezt a teszt áramkört (amit ugyanúgy entitással-architektúrával kell definiálni) test bench-nek nevezzük. Ez általában úgy működik, hogy tipikus jelkombinációkat (pontosabban variációt, mert számít a sorrend) kapcsol a tesztelendő áramkörünk bemenetére, mi pedig a szimulátor kimenetén keresztül ellenőrizzük, hogy helyes-e a kimenet értéke.

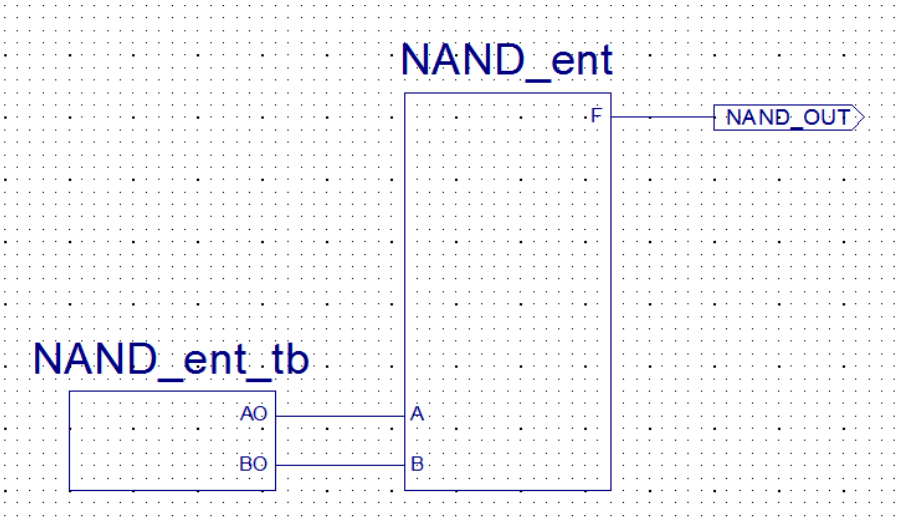
Ha ismert bemenetre az áramkör helyes kimenettel reagál, akkor élhetünk azzal a naiv feltételezéssel, hogy helyesen működik.

A NAND kapunk test bench-e az igazságtáblázat alapján végigpróbálhatná az összes bemeneti kombinációt. Természetesen némi késleltetést azért bele kell programoznunk, hogy értékelhető eredményt kapjunk:

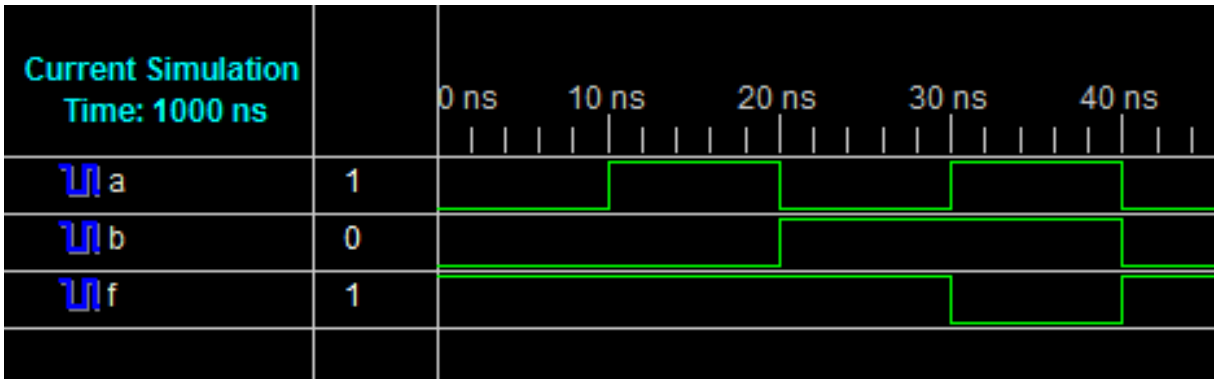
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity NAND_ent_tb is
    port(
        AO: out std_logic;
        BO: out std_logic
    );
end entity;

architecture behv of NAND_ent_tb is
begin
    process begin
        AO <= '0';
        BO <= '0';

        wait for 10 ns;

        AO <= '1';
        BO <= '0';

        wait for 10 ns;

        AO <= '0';
        BO <= '1';

        wait for 10 ns;

        AO <= '1';
        BO <= '1';
```

```
        wait for 10 ns;
    end process;
end behv;
```

A test bench és a tesztelendő áramkör összehuzalozását (sőt, minden, magas szinten végzett műveletet, például az áramkörök hozzárendelését az FPGA kimeneteihez) mindenképpen a schematic editorban (elvi kapcsolási rajz szerkesztőben) javaslom, mert az ember vizuális lény, és könnyebben végez ilyen jellegű műveleteket vizuálisan. A működés leírása természetesen mindig VHDL-ben történjen.



Ha elvégezzük a szimulációt, az alábbi eredményt kapjuk:



Láthatjuk, hogy a kapu kimenete akkor és csak akkor nulla értékű, ha mindkét bemenet logikai 1 szinten van. Tehát a NAND kapunk valóban jól működik.

Az inverz RS tároló hasonlóan tesztelhető, ennek elvégzését az Olvasóra bízom. A D-flipflop automatikusan generált testbench-e a következőképpen néz ki:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY dff_tb IS
END dff_tb;

ARCHITECTURE behavior OF dff_tb IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT dff
    PORT(
        data_in : IN  std_logic;
        clock : IN  std_logic;
        data_out : OUT  std_logic
    );
    END COMPONENT;

    --Inputs
    signal data_in : std_logic := '0';
    signal clock : std_logic := '0';

    --Outputs
    signal data_out : std_logic;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: dff PORT MAP (
        data_in => data_in,
        clock => clock,
        data_out => data_out
    );

    -- Stimulus process
    stim_proc: process
    begin
        data_in <= '0';
        clock <= '0';
        wait for 5 ns;

        data_in <= '0';
        clock <= '1';
        wait for 5 ns;

        data_in <= '0';
        clock <= '0';
        wait for 5 ns;

        data_in <= '1';
        clock <= '0';
        wait for 5 ns;

        data_in <= '1';
        clock <= '1';
        wait for 5 ns;
```

```
        data_in <= '1';
        clock <= '0';
        wait for 5 ns;


        data_in <= '0';
        clock <= '0';
        wait for 5 ns;

        data_in <= '0';
        clock <= '1';
        wait for 5 ns;

        data_in <= '0';
        clock <= '0';
        wait for 5 ns;

        wait;
    end process;
end;
```

4.5. Az FPGA kimenetei

Valójában egy lépés választ el minket attól, hogy fizikai, azaz látható-hallható áramkört építsünk: még meg kell tanulnunk az elkészített és letesztelt áramkör kimeneteit az FPGA lábaihoz rendelni. Ez egy alapvetően egyszerű folyamat: vagy a fejlesztőkörnyezet beépített programját használjuk a kimenet-bemenet → FPGA láb összerendeléshez, vagy készítünk egy ún. constraints fájlt.

Linux alatt számos problémát tapasztaltam, mert a Xilinx programja magától képtelen constraints fájlt készíteni, ezért „kézzel” kell helyette ezt megtennünk:

```
NET "A"  LOC = "G4";
NET "B"  LOC = "H3";
NET "F"  LOC = "E1";
```

Ezzel a NAND kapu 3 lábát kivezettük az FPGA lábaira (BGA tokozásnál sorok [A...P] és és oszlopok vannak [1..16]). Természetesen az előbbi inverz RS tároló lábait is hasonlóan kellene hozzárendelni. A test bench-eket nem vezetjük ki, sőt, a végső áramkörben nem is kapnak helyet.

4.6. Könyvtárak, library-k

Azt senki sem gondolhatja komolyan, hogy egy processzor tervezését a NAND kapunál kell kezdeni. Számtalan, előre elkészített alapelemet kapunk a CPLD/FPGA fejlesztői környezethez. Ezeket típustól függően könyvtárakba szervezik, hogy megkönnyítsék felhasználásukat. Az alapelemek pontosan ugyanúgy használandók, mint a NAND kapunk, vagy mint az inverz RS tárolónk.

5. Mintafeladatok

Ebben a részben felelevenítjük a már elmondottakat, olyan (működő!) mintaprogramokat veszünk szemügyre, melyek valóban előfordulnak a mindennapi gyakorlatban. Az egyszerűbbtől az összetettebb felé haladva megismerjük a szükséges forrásokat, és a teszteléshez szükséges környezetet.

5.1. Kombinációs hálózatok

Kombinációs hálózatoknak azokat a digitális hálózatokat nevezzük, melyek állapota csak az AKTUÁLIS bemenetek állapotától függ, tehát a kombinációs hálózat nem rendelkezik emlékező (memória) áramkörrel. Természetesen lehetséges, hogy kombinációs hálózatokból emlékező áramköröket építsünk (lásd: a NAND kapuval megvalósított inverz RS tároló példáját), de ettől most eltekintünk.

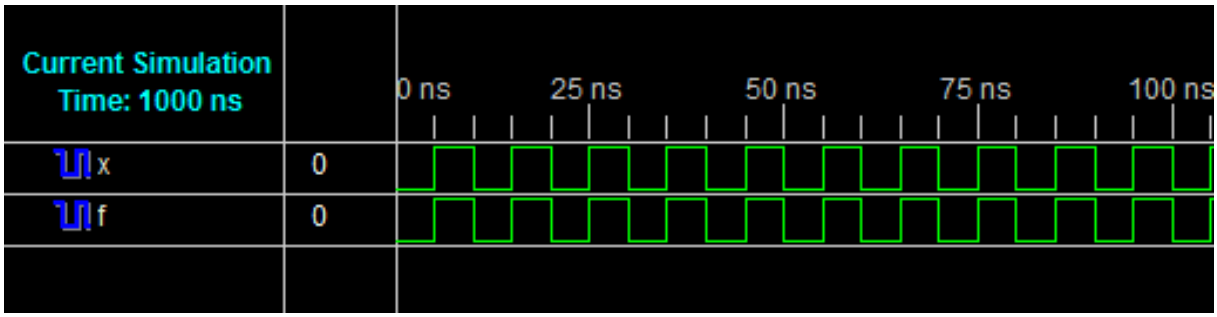
Ebben az alfejezetben a ténylegesen kombinációs alapokon működő áramkörökkel foglalkozunk.

5.1.1. Aszinkron D-tároló, inverter

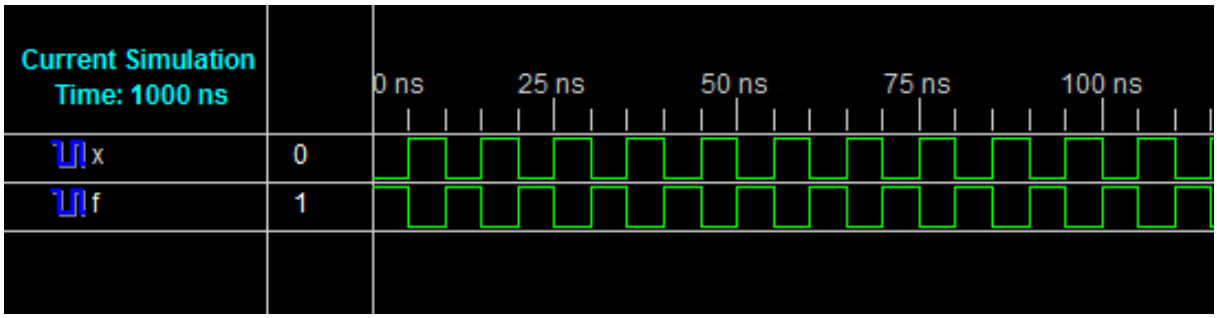
Maga a név némi tréfát rejt magában. Egy olyan D-tárolóról van ugyanis szó, amelynek működése nincsen az órajelhez kötve, vagyis a bemenet értékét a kimenetre „másolja” anélkül, hogy az órajel ezt bármilyen módon befolyásolná. Nem másról van szó, mint egy „kellően vastag rézdrótról”.

A megfelelő VHDL kód az alábbi módon írható fel:

```
library ieee;
use ieee.std_logic_1164.all;

entity Driver is
    port(
        x: in std_logic;
        F: out std_logic
    );
end Driver;

architecture behv2 of Driver is
begin
    F <= x;
end behv2;
```



Az előbbi kódból elég egyszerű kialakítani az invertert: csak cseréljük ki a hozzárendelést egy másikra. A helyes operátor megtalálását az Olvasóra bízom.



5.1.2. 1 bites digitális komparátor (==)

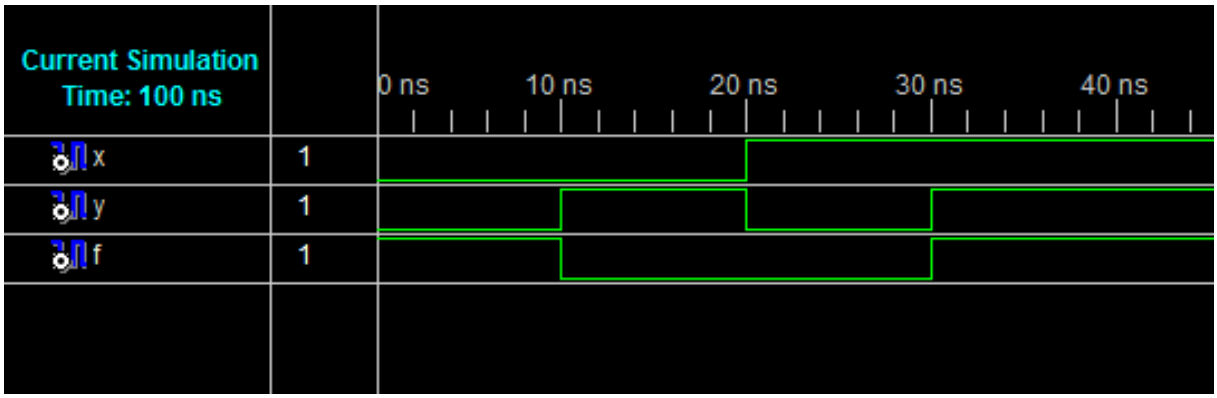
Ez az áramkör két bemenettel és egy kimenettel rendelkezik: meg tudja állapítani, hogy a bemenetek értéke megegyezik-e:

```
library ieee;
use ieee.std_logic_1164.all;

entity XOR_ent is
    port(
        x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end XOR_ent;

architecture behav of XOR_ent is
begin
    F <= x xnor y;
end behav;
```

A hozzá tartozó testbench a következő:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY digit_comp_1bit_tb IS
END digit_comp_1bit_tb;

ARCHITECTURE behavior OF digit_comp_1bit_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT XOR_ent
    PORT(
        x : IN  std_logic;
        y : IN  std_logic;
        F : OUT  std_logic
    );
    END COMPONENT;

    --Inputs
    signal x : std_logic := '0';
    signal y : std_logic := '0';

    --Outputs
    signal F : std_logic;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: XOR_ent PORT MAP (
        x => x,
        y => y,
        F => F
    );

    -- Stimulus process
    stim_proc: process
    begin
        x <= '0'; y <= '0'; wait for 10 ns;
        x <= '0'; y <= '1'; wait for 10 ns;
        x <= '1'; y <= '0'; wait for 10 ns;
        x <= '1'; y <= '1'; wait for 10 ns;
        wait;
    end process;

END;
```

A szimuláció eredményét a következő kép mutatja. Látható, hogy a kimenet (`f`) értéke akkor „1”, ha a bemenet (`x` és `y`) értéke megegyezik.



A következő részben egy olyan 1 bites komparátorral ismerkedtünk meg, amely azt is el tudja dönteni, hogy a két beérkező „szám” közül melyik a nagyobb, illetve egyenlő-e a kettő.

5.1.3. Bonyolultabb digitális komparátor (==, <, >)

Ez az áramkör két bemenettel és 3 kimenettel rendelkezik. A két bemeneten 1-1 bit érkezik, és a kimenő 3 biten azt jelzi az áramkör, hogy az `x` bemenet értéke nagyobb-e (`L = 1`) vagy kisebb-e, mint az `y` bemenet értéke (`G = 1`), vagy esetleg a két szám egyenlő (`E = 1`).

```
library ieee;
use ieee.std_logic_1164.all;

entity XOR_ent is
    port(
        x: in std_logic;
        y: in std_logic;
        E: out std_logic;
        G: out std_logic;
        L: out std_logic
    );
end XOR_ent;

architecture behav of XOR_ent is
begin
    process (x, y)
    begin
        if x = '0' and y = '0'
        then
            E <= '1'; G <= '0'; L <= '0';
        end if;

        if x = '0' and y = '1'
        then
            E <= '0'; G <= '0'; L <= '1';
```

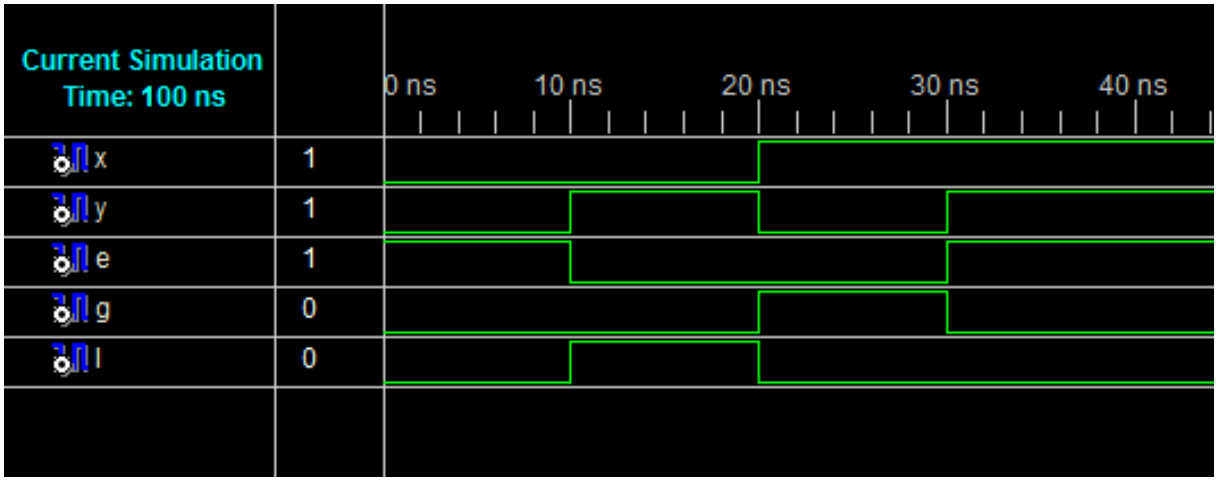
```
        end if;

        if x = '1' and y = '0'
        then
            E <= '0'; G <= '1'; L <= '0';
        end if;

        if x = '1' and y = '1'
        then
            E <= '1'; G <= '0'; L <= '0';
        end if;
    end process;
end behav;
```

Természetesen ehhez a forráskódhoz is mellékelem a testbenchet, mert enélkül rendkívül nehéz a tesztelés:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY digit_comp_1bit_tb IS
END digit_comp_1bit_tb;

ARCHITECTURE behavior OF digit_comp_1bit_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT XOR_ent
    PORT(
         x : IN  std_logic;
         y : IN  std_logic;
         E : OUT  std_logic;
         G : OUT  std_logic;
         L : OUT  std_logic
    );
    END COMPONENT;

    --Inputs
    signal x : std_logic := '0';
    signal y : std_logic := '0';

    --Outputs
    signal E : std_logic;
    signal G : std_logic;
    signal L : std_logic;

BEGIN

    -- Instantiate the Unit Under Test (UUT)
    uut: XOR_ent PORT MAP (
        x => x,
        y => y,
```

```
        E => E,
        G => G,
        L => L
    );


    -- Stimulus process
    stim_proc: process
    begin
        x <= '0'; y <= '0'; wait for 10 ns;
        x <= '0'; y <= '1'; wait for 10 ns;
        x <= '1'; y <= '0'; wait for 10 ns;
        x <= '1'; y <= '1'; wait for 10 ns;
        wait;
    end process;
END;
```

A szimuláció eredményét a következő ábra mutatja. A bemenetre mind a 4 kombinációt rákapcsolja a test bench, a szimulátorral pedig megfigyelhetjük a bemenetekhez tartozó kimenetek értékét.



Látható, hogyha mindkét bemenet értéke megegyezik, akkor az `E` kimenet 1 értékű (0 - 10 ns és 30 - 40 ns). Ha az `x` bemenet 1 értékű és az `y` értéke 0 (20 - 30 ns), akkor a `G` értéke 1 lesz. Ha az `y` értéke nagyobb, mint az `x` értéke (10 - 20 ns), akkor az `L` kimenet lesz logikai igaz értékű, míg a többi hamis értéket vesz fel.

5.2. Szekvenciális hálózatok

Definíció szerint a szekvenciális hálózatok olyan digitális áramkörök, melyek állapota nem csak a bemeneteinek állapotától függ, hanem a belső „emlékező” áramköreinek (memória) az állapotától is.

Ebben a fejezetben azzal ismerkedünk meg, hogy hogyan lehet VHDL nyelven szekvenciális hálózatot definiálni.

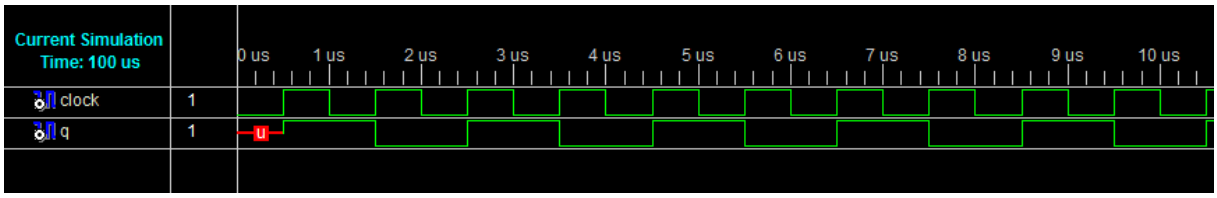
5.2.1. Szinkron T-tároló

A T-tároló egy rendkívül egyszerű szekvenciális hálózat: csak 1 bemenete van, az órajel (`clock`). A kimenet értékét (`q`) az órajel felfutó (vagy éppen lefutó) élének hatására meginvertálja. Vagyis tudnia kell azt, hogy mi a kimenet aktuális értéke, mert annak inverzét kell a felfutó él hatására a kimenetre juttatnia. A forráskódban a memória-áramkört a `signal previous_state : bit := '0'` képviseli.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity t_flipflop is
    Port (
        clock : in  STD_LOGIC;
        q : out  STD_LOGIC
    );
end t_flipflop;

architecture Behavioral of t_flipflop is
    signal previous_state : bit := '0';
begin
    process (clock)
    begin
        if clock = '1'
        then
            if previous_state = '0'
            then
                previous_state <= '1';
                q <= '1';
            else
                previous_state <= '0';
                q <= '0';
            end if;
        end if;
    end process;
end Behavioral;
```

A test bench nem csinál mást, mint 0,5 µs-onként meginvertálja az órajel értékét. Ennek hatására a T-tároló kimenete 1 µs-onként fog változni.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY t_flipflop_tb IS
END t_flipflop_tb;

ARCHITECTURE behavior OF t_flipflop_tb IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT t_flipflop
    PORT(
        clock : IN  std_logic;
        q : OUT  std_logic
    );
    END COMPONENT;


    --Inputs
    signal clock : std_logic := '0';

    --Outputs
    signal q : std_logic;

    -- Clock period definitions
    constant clock_period : time := 1us;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: t_flipflop PORT MAP (
        clock => clock,
        q => q
    );

    -- Clock process definitions
    clock_process :process
    begin
        clock <= '0';
        wait for clock_period/2;
        clock <= '1';
        wait for clock_period/2;
    end process;
END;
```



A szimuláció teljes mértékben alátámasztja a fent leírtakat. Az órajel periódusideje 0,5 µs, míg a `Q` kimeneten 1 µs periódusidejű négyszögjel jelenik meg.

5.2.2. 8-bites számláló

Ebben a részben egy bonyolultabb szekvenciális hálózatot valósítunk meg. Az órajel-bemenetre érkező jel egy 8 bites számláló értékét növeli. Mivel ez egy 8 bites számláló, ezért a kimeneten megjelenő értékek egy véges Galois-mező ($GF(2^8)$, $\mathbf{Z}/256\mathbf{Z}$) elemei lesznek.

Az áramkörnek két bemenete van, a `clear` bemenet 1 értéke törli a számláló értékét, és tiltja a számláló működését. Ha a `clear` bemenet 0 értékű, és felfutó él érkezik az órajelbemenetre (`clock`), akkor a számláló értéke 1-gyel növekszik, és ez az érték kikerül a `Q` kimenetre.

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------

entity counter is
    port(
        clock: in std_logic;
        clear: in std_logic;
        Q:     out std_logic_vector(7 downto 0)
    );
end counter;

-------------------------------------------------------

architecture behavior of counter is
    signal value: std_logic_vector(7 downto 0);

begin
    process(clock, clear)
    begin
        if clear = '1' then
            value(0) <= '0';
            value(1) <= '0';
            value(2) <= '0';
            value(3) <= '0';
```

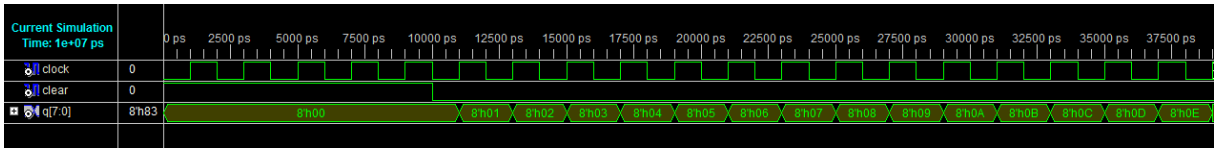
```
            value(4) <= '0';
            value(5) <= '0';
            value(6) <= '0';
            value(7) <= '0';
        elsif (clock='1' and clock'event) then
            value <= value + 1;
        end if;
    end process;

    Q <= value;

end behavior;
```

A test bench az előző példához hasonlóan működik, csak az órajel kimenetét változtatja adott időközönként, ezzel gerjesztve a 8 bites számlálót.

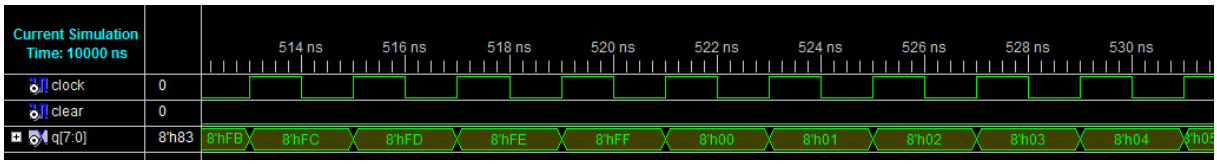
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

    COMPONENT counter
    PORT(
        clock : IN  std_logic;
        clear : IN  std_logic;
        Q : OUT  std_logic_vector(7 downto 0)
    );
    END COMPONENT;

    signal clock : std_logic;
    signal clear : std_logic := '0';
    signal Q : std_logic_vector(7 downto 0);

BEGIN
    uut: counter PORT MAP (
        clock => clock,
        clear => clear,
        Q => Q
    );

    process
    begin
        clear <= '1'; wait for 10 ns;
        clear <= '0'; wait;
    end process;

    process
    begin
        clock <= '0'; wait for 1 ns;
```

```
        clock <= '1'; wait for 1 ns;
    end process;

END;
```

Látható, hogy a `clear` bemenet teljesen tiltja a számlálást. Ha a `clear` bemenet 0 értékű, akkor a kimenet értéke szépen növekedik...



... egészen 255-ig (FF_{hex}-ig), amikor túlcsordul a számláló, és újra 00_{hex} értéket vesz fel (512 ns) a kimenet.



Ezzel megnéztük néhány alapvető, és nagyon egyszerű áramkör leírását VHDL nyelven. Az Olvasó ennek alapján már el tud indulni a VHDL programozás rögös útján.

6. Összefoglalás, végkövetkeztetés

A jegyzet első részében a VHDL nyelv alapelemeit, lexikai komponenseit ismertük meg, áttekintettük például az azonosítók, számok, sztringek definiálását, használatát.

A következő fejezetben a nyelv vezérlési szerkezeteivel ismerkedtünk meg, többek között a feltételes elágazással, a többszörös elágazással, a cilkusszervező utasításokkal.

Az ötödik részben a hardver leírásával foglalkoztunk. Megtanultuk, hogy hogyan lehet új komponens definiálni és azt alkatrészként felhasználni.

Végül a megszerzett ismeretek segítségével számos működő áramkört alkottunk.

Látható, hogy a VHDL nyelv nem teljesen alkalmatlan digitális áramkörök működésének leírására.

7. Felhasznált szoftverek

- Ubuntu Linux 8.10 (Linux kernel 2.6.27-9-generic #1 SMP)
- OpenOffice.org 2.4.1
- The GIMP 2.4.7
- Xilinx ISE WebPACK 10.4

8. Felhasznált irodalom

- Peter J. Ashenden: VHDL Tutorial (http://www.tutground.net/Files/VHDL_TUTORIAL.pdf)
- Weijun Zhang: VHDL Tutorial: Learn by Example (http://esd.cs.ucr.edu/labs/tutorial/)
- VHDL MINI-REFERENCE (http://www.eng.auburn.edu/department/ee/mgc/vhdl.html)