

Paraméterezhető memória-vezérlő alrendszer megvalósítása FPGA-n tudományos számítások gyorsítására

TDK dolgozat

Raska István

Konzulens:

Dr. Nagy Zoltán

Füredi László



PÁZMÁNY PÉTER CATHOLIC UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

BUDAPEST

2012

Tartalomjegyzék

Bevezetés	2
Introduction	4
1. Felhasznált eszközök	6
1.1. A VHDL nyelv	6
1.1.1. Leíró megközelítés	6
1.1.2. Nyelvi elemek	9
1.1.3. Adattípusok és objektumok	9
1.1.4. Vezérlési szerkezetek	11
1.1.5. A leírás és a keletkező áramkörü elem összefüggése	11
1.2. Az FPGA fejlesztőrendszerek	12
1.2.1. Általában az FPGA-król	12
1.2.2. Xilinx Virtex-5 XC5VSX240T FPGA	18
1.2.3. Alpha-Data ADM-XRC-5T2 fejlesztőkártya	20
1.2.4. SDK	21
2. Elektromágneses tér szimuláció	26
2.1. Bevezetés	26
2.2. 3D Transmission-Line Modeling Módszer	26
2.2.1. Transmission-Line Modeling Módszer	26
2.2.2. Hibrid Symmetrical Condensed Node	27
2.3. Az FPGA-s implementáció	28
3. Az áramkör	33
3.1. FIFO	35
3.1.1. Beírás	36
3.1.2. Kiolvasás	37

3.2. Olvasó DMA	38
3.3. Író DMA	39
3.4. A változó-kezelő	40
3.5. A konstans-kezelő	41
3.6. Az eredmény-kezelő	41
Eredmények	43
Összefoglaló	44
Köszönetnyilvánítás	46
Irodalomjegyzék	48
Ábrajegyzék	50
Táblázatjegyzék	51
Rövidítések	52

Bevezetés

Számos, a gyakorlatban előforduló mérnöki és tudományos alkalmazásban szükséges komplex fizikai jelenségek szimulációja, melyeket nemlineáris parciális differenciálegyenletekkel írhatunk le. Az informatikai és villamosmérnöki gyakorlatban egyik legfontosabb feladat elektromágneses hullámok szimulációja, mely sok területen jelenik meg, pl. antennák tervezése, nagy sebességű nyomtatott áramkörü lapok huzalozása stb. Az elektromágneses hullámok viselkedését a Maxwell-egyenletek írják le, melyek digitális architektúrán történő szimulációja – diszkretizálás után – numerikus közelítéssel lehetséges. Nagyfrekvenciás nyomtatott áramkörü lapok szimulációja során a Maxwell-egyenletek numerikus megoldására a Transmission Line Matrix (TLM) módszer a leghatékonyabb, amely jól használható bonyolult geometriák, inhomogén és anizotrop közegek esetében is. A pontos szimulációhoz finom diszkretizálás szükséges, amely sok esetben igen nagy számítási teljesítményt igényel. A manapság megjelenő sok ezer magos architektúrák (GPU, FPGA) jó lehetőséget kínálnak numerikus algoritmusok gyorsítására. Az FPGA-n megvalósított TLM módszer hatékonyságát nagyban befolyásolja a szimulációra kialakított architektúra és megvalósítási platform jellemzői; ilyenek a működési frekvencia, a felületigény, és a rendelkezésre álló memória-sávszélesség.

A gyorsító-architektúra hatékony adatellátásához szükséges egy nagy sebességű memória interfész kialakítása, mely a memória alrendszer részleteit elfedi a feldolgozó-egységek előtt. A rendelkezésre álló fejlesztő eszközök eltérő architektúrája, elsősorban az FPGA-hoz csatlakozó (off-chip) memóriák típusa, mennyisége, sebessége és a memória chip-ek száma jelentős eltéréseket mutat. A gyorsító-áramkör fejlesztőkártyák közötti hordozhatóságának biztosításához a memóriavezérlő-áramkörnek konfigurálhatónak kell lennie. A kihívás abban áll, hogy az áramkörnek újrafelhasználhatónak kell lennie, akár egy másik aritmetikai egység kiszolgálásáról, akár egy másik típusú FPGA-n történő futtatásról legyen szó. Ennek érdekében generic típusokat kell alkalmazni, ahol lehetséges; ahol pedig nem az, ott könnyen alkalmazható megoldást kell találni az áramkör hordozhatóságának biztosítására. A dolgozat ennek az áramkörnek a tervezési lépéseit mutatja be.

Az ilyesfajta számításokhoz párhuzamos architektúrákon érdemes alkalmazni, amelyekkel a CPU-khoz képest nagyságrendekkel gyorsabban lehet elvégezni a szimulációkat, avagy több számítást lehet végrehajtani velük ugyanannyi idő alatt. A probléma megoldására én az FPGA architektúrát és a VHDL programozási nyelvet választottam, mert ezek állnak közelebb hozzám, továbbá sokféle alkalmazásban bizonyították már, hogy alkalmasak az ilyesfajta feladatok megoldására. A dokumentum első felében betekintést nyújtok a VHDL nyelv és az FPGA architektúra alapjaiba, ismertetem a jelenleg a PPKE-ITK-n és a SZTAKI-ban elérhető Alpha-Data fejlesztőrendszerek és a bennük helyet foglaló Xilinx Virtex-5-ös FPGA chippek főbb tulajdonságait. Megvizsgálom a fejlesztőkártyához tartozó SDK-ban található, már elkészített áramköröket azért, hogy melyek azok a modulok, amelyek fel tudok használni a probléma megoldására. Ezután matematikai levezetést adok az elektromágneses terek szimulációjához alkalmazható folytonos fizikai egyenletek diszkretizálásáról, a szimulációs térháló mibenlétéről, és az ezekből készíthető aritmetikai egységről. Végezetül felvázolom az elképzelt áramkör összetevőit és tulajdonságait, vagyis az SDK-ban található memóriavezérlő és arbitrációs egység, a már elkészített aritmetikai egység, és az általam elkészítendő, az előbbieket összeköttetését biztosító DMA vezérlők sematikus leírását.

Introduction

Many of the scientific and engineering applications require simulations of complex physical phenomena, which can be described by nonlinear partial differential equations. One of the most important task in the information technology and electrical engineering practice is the simulation of electromagnetic waves, which appear in many areas, for example antenna design, high-speed printed circuit board wiring, etc. The behaviour of electromagnetic waves described by Maxwell's equations, which equations can be solved on digital architectures - after discretization - with numerical approximation. Meanwhile the simulation on printed circuit boards, the most efficient way of the numerical solution of the Maxwell's equations is the Transmission Line Matrix (TLM) method, which is useful for complex geometries, inhomogeneous and anisotropic media as well. For the precise simulation fine discretization is needed, which require very high computational power in many cases. The nowadays published multi- and many-core architectures (GPUs, FPGAs) are a good way to speed up numerical algorithms. The efficiency of the FPGA implemented TLM method is influenced by the designed architecture for the simulation and the features of the implementation platform; such parameters are the operating frequency, the surface requirements, and the available memory bandwidth.

For the efficient data providing of the accelerator-architecture is needed to design a high-speed memory interface, which hides the details of the memory subsystem from the processing units. The different architectures of the available development tools, mainly the (off-chip) memory types, amounts, speeds and memory chip numbers connected to the FPGAs have a very significant variation. To ensure the portability of the accelerator circuit between the developer boards the memory-controller circuit should to be reconfigurable. The challenge is that the circuit have to be reusable, either to handle an another arithmetic circuit, or to running in an another FPGA. Therefore generic types should be used where possible; and where it is not, should be found an easily usable solution to ensure the portability of the circuit. This paper shows the design steps of that circuit.

For that computations worth using parallel architectures, which compared to CPUs

the simulations can be performed orders of magnitude faster, or more calculations can be carried out in the same amount of time. To solve this problem I chose the FPGA architecture and the VHDL programming language, because they are close to me, as well as in many applications has proven to suitable for such tasks. In the first half of the paper I provide insight into the basics of the VHDL language and of the FPGA architecture, I present the Alpha-Data development system currently available at PPCU-FIT (PPKE-ITK) and at CARI (SZTAKI) and the main properties of the Xilinx Virtex-5 FPGA chips therein. Examine the prepared circuits in the development card's SDK in order to select some of the modules that can be used to solve the main problem. Then I give a mathematical derivation of the discretization of the continuous physical equations used for the simulation of electromagnetic fields, the nature of the geometric mesh, and the arithmetic unit which from that can be made. Finally, I sketch the components and properties of the imagined circuit, namely the memory controller and arbitration unit located in the SDK, the already prepared arithmetic unit, and the schematic descriptions of the DMA controllers which ensure the connection of the previous two components, to be made by me.

1. fejezet

Felhasznált eszközök

1.1. A VHDL nyelv

A VHDL (vagyis a VHSIC Hardware Description Language) egy hardver-leíró nyelv, amely digitális és mixed-signal áramkörök tervezésére szolgál. A nyelv története az 1980-as évekre nyúlik vissza, amikor is az USA kormányának VHSIC (Very High Speed Integrated Circuits) programjának keretében különféle megoldásokat dolgoztak ki nagysebességű integrált áramkörök fejlesztéséhez. Világossá vált, hogy szükség van egy szabványos nyelvre az áramkörök leírásához, így született meg a VHDL, amelyet később az IEEE is szabványosított. Megalkotásakor törekedtek arra, hogy ne hozzanak létre ismét már létező koncepciókat, így sok elemet emeltek át az Ada és a Pascal nyelvből. [1]

Az egyik cél az volt, hogy hardver működését már a tervezési időszakban is szimulálni lehessen, megkönnyítve és meggyorsítva ezzel a fejlesztést. A másik pedig az, hogy a VHDL-ben leírt áramkört szintetizálni is lehessen, azaz egy eszköz segítségével az áramkör leírásából fizikai implementációt lehessen készíteni. A mai szintetizáló eszközök már mindenféle digitális és mixed-signal áramkörök fejlesztésére lehetőséget adnak, ráadásul ugyanabból a leírásból többféle típusú áramkör szintetizálható attól függően, hogy például az alacsony fogyasztást, a minimális elfoglalt chip-felületet vagy a maximális órajelet részesítjük előnyben. [2]

1.1.1. Leíró megközelítés

A VHDL-ben ugyanazt a hardvert többféleképpen is le lehet írni: struktúra vagy működés alapján. Strukturális esetben a leírás azt tartalmazza, hogy a hardverelemek – amelyek más entitások példányai – hogyan vannak összekötve egymással. Az összeköttetéseket minden esetben az entitások portjai (csatlakozói) közötti vezetékek alkotják.

Ugyanakkor nem mindig szerencsés a strukturális leírás, sok esetben az adott hardver-elem funkcióját kell leírni. Egy entitásnál feltételezzük, hogy a kimenetek a bemenetek függvényei, ezeket a függvényeket, vagyis a működést írjuk le ebben az esetben.

Nézzünk meg egy rövid példát: írjunk egy 2 bites számlálót (1.1. ábra). Elég hozzá egy bemenet, amelyen át az órajelet kapja, és két kimenet a számláló értékének. Természetesen minden entitáshoz tartozik egy késleltetés is, amelyet vagy generic-ként megadunk, vagy pedig hagyjuk, hogy automatikusan generálódjon a dizájnából.

```
entity count2 is
    port(
        clock : in bit;
        q1, q0 : out bit
    );
end count2;
```

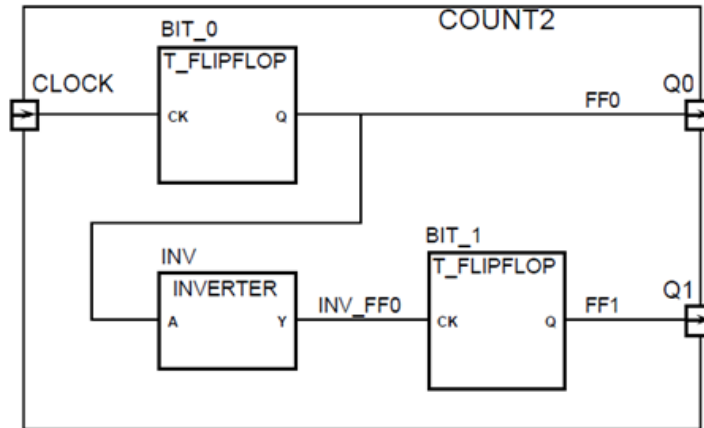
1.1. ábra. *VHDL entitás definíció*

Az interfész definíciója után az előbb leírtaknak megfelelően leírhatjuk a működést funkcionálisan, amelyet a 1.2. ábra mutat be. Ehhez a számlálóhoz elég egy **count_up** nevű proceszt létrehozni, amelynek az érzékenységi listájában az órajelet (**clock**) adjuk meg. Ez a procesz minden órajel magas szintjére a **count_value** értékét megnöveli eggyel és modulo 4-et vesz rajta (hiszen elég 0 és 3 között számlálnia), valamint ezzel egyidejűleg a **q0** és **q1** kimenetekre ráköti a **count_value** megfelelő bitjét.

```
architecture behaviour of count2 is
begin
    count_up : process(clock)
        variable count_value : natural := 0;
    begin
        if clock = '1' then
            count_value := (count_value + 1) mod 4;
            q0 <= bit'val(count_value mod 2);
            q1 <= bit'val(count_value / 2);
        end if;
    end process count_up;
endbehaviour;
```

1.2. ábra. *VHDL funkcionális leírás*

A másik lehetőség a strukturális leírás. Ugyanazt a működést, amelyet a fenti VHDL kód leír, a (1.3. ábrán) látható kapcsolással helyettesíthetjük.



1.3. ábra. 2-bites számláló blokkdiagram

Ezt a kapcsolást írja le a 1.4. ábrán látható VHDL kód. Kétféle entitásra lesz szükségünk: egy **t_flipflop**-ra, amely regiszterként funkcionál, és egy inverterre. Az egész áramkört szintén az órajel vezérli, azzal a különbséggel, hogy itt a végeredmény is az órajelből generálódik, nem pedig csak egy processzt vezérel.

```
architecture structure of count2 is
    component t_flipflop
        port( ck : in bit; q : out bit);
    end component;
    component inverter
        port( a : in bit; y : out bit);
    end component;
    signal ff0, ff1, inv_ff0 : bit;
begin
    bit_0 : t_flipflop
        port map(
            ck => clock,
            q => ff0);
    inv : inverter
        port map(
            a => ff0,
            y => inv_ff0);
    bit_1 : t_flipflop
        port map(
            ck => inv_ff0,
            q => ff1);

    q0 <= ff0;
    q1 <= ff1;
end structure;
```

1.4. ábra. VHDL struktúrális leírás

Kezdeti esetben mindkét regiszterben 0-k vannak. Amikor az órajel felmegy 1-re, akkor a **q0** kimeneten 1-es lesz, a **q1**-n 0 marad. Az órajel 0-ra esésekor a **q0** kimeneten 0 lesz, és a **q1** 1-esre vált. Az órajelnek a következő 1-re emelkedése után a **q0** kimenet 1-es lesz, de a **q1** 1-es marad az áramköri elemek késleltetése miatt. Az órajel újbóli változásakor az egész folyamat kezdődik előről.

1.1.2. Nyelvi elemek

A VHDL tág teret enged az adatok leírására, éppen azért, hogy mindig a feladathoz legközelebb álló leírást tudjuk alkalmazni. A számokból egész és valós típusok vannak, de ezeket a 2-estől a 16-osig bármilyen számrendszerben leírhatjuk:

```
2#1100_0100#      16#C4#    4#301#E1      --196, integer number
2#1.1111_1111_111#E+11  16#F.FF#E2      --4095.0, real number
```

Szövegből megkülönböztetettek a karakter és string típusok, csakúgy, mint sok más nyelvben:

```
'A'      '*'      ''      ' ' --character types
"A_string_in_a_string:" "A_string"." --Strings
```

Ezekon kívül még bit sztringeket is létrehozhatunk, amelyek bit típusú tömb literálok:

```
B"1010110"      --length is 7
O"126"          --length is 9, equivalent to B"001_010_110"
X"56"           --length is 8, equivalent to B"0101_0110"
```

1.1.3. Adattípusok és objektumok

A nyelv sokféle beépített skalár és összetett típussal rendelkezik, és sokféle eszközt ad a fejlesztő kezébe, hogy összetett típusokat is létre lehessen hozni. A skalár típusok a számok, a fizikai mennyiségek és a felsoroló (enum) típusok. Összetett típusok a tömbök és a rekordok. Ezen kívül mutató (pointer) és fájl típusokkal is lehet dolgozni. Az egész típusok egész értékeket jelölhetnek egy megadott értéktartományon belül. Természetesen az értéktartományt is csak egész számokkal lehet megadni. Alapvető megközelítést jelöl a VHDL-ben a **to** és a **downto** kulcsszavak jelenléte; előbbi a növekvő, utóbbi a csökkenő sorrendre utal. Beépített egész típus az integer, értéktartománya -2147483647 és 2147483647 között van. Egész típusokat így tudunk deklarálni:

```
type byte_int is range 0 to 255;
type signed_word is range -32768 to 32767;
type bit_index is range 31 downto 0;
```

Fizikai típusként tudunk megadni olyan mennyiségeket, amelyek az adott áramköri elem fizikai tulajdonságaiként fognak kiértékelődni – később ezek adhatják az alapot az áramkör legyártásához. Ezek lehetnek késleltetés, ellenállás, és akár méret is. Nem kell SI-ben lennie, ugyanis a típusdeklarációnál átváltási számokkal együtt kell felsorolni az elnevezéseket. Beépített fizikai típus az idő. Lebegőpontos típusokat is létre tudunk hozni, hasonlóan az egész típusokhoz, azzal a különbséggel, hogy az értéktartománynál valós számokat kell megadnunk:

```
type signal_level is range -10.00 to +10.00;
type probability is range 0.0 to 1.0;
```

Sok esetben a felsorolási típusok is fontosak lehetnek. Hasonlóan más nyelvekhez, ez a típus karakterek vagy azonosítók rendezett sorozata; ezeknek egy felsoroláson belül különbözőknek kell lenniük, de más felsorolási típusokban újra felhasználhatóak.

```
type boolean is (false, true);
type logic_level is (unknown, low, undriven, high);
```

A VHDL-ben elemek indexelt sorozatát minden esetben tömb típusúnak tekintjük. Egy tömb lehet egy vagy több dimenziós, valamint a korlátos és nem korlátos attól függően, hogy a típusdeklarációban vagy az adott elem létrehozásakor adjuk meg a lehetséges értékkészletet.

```
type word is array (31 downto 0) of bit;
type vector is array (integer range <>) of real;
```

Létre lehet hozni altípusokat is, amelyekkel az eredeti típus értékkészletét tudjuk korlátozni.

```
subtype pin_count is integer range 0 to 400;
```

A megismert vagy elkészített típusokkal már objektumokat is létre tudunk hozni. A VHDL-ben egy objektum egy nevesített elem, amely egy specifikus típusból vesz fel egy értéket. Objektumok a konstansok, a vezetékek és a változók. Egy konstans tartja a megadott értékét, az sohasem változik, így nem lehet írni, csak olvasni. Ezzel szemben egy változó értéke bármikor változhat. Egy vezetékek értékeit továbbít az egyik hardver-elemtől a másikig, és emiatt az előbbiekkal szemben másféle megközelítést igényel.

1.1.4. Vezérlési szerkezetek

Más nyelvekhez hasonlóan, a VHDL-ben is megtalálhatóak a szokásos vezérlési szerkezetek: a feltételes elágazások (**if**, **case**) és a ciklusok (**loop**, **while**, **for** és **foreach**). Ezeken kívül speciálisnak számít a **null** és az **assert** megléte; előbbi arra szolgál, hogy bizonyos esetekben jelölni lehessen, hogy nincs szükség semmilyen utasítás végrehajtására, utóbbi pedig a teszteléshez és debug-oláshoz nyújt segítséget.

1.1.5. A leírás és a keletkező áramkörü elem összefüggése

Vannak bizonyos koncepciók a VHDL-ben arra vonatkozólag, hogy egy adott áramkörü elem létrehozásához milyen stílusban kell megírni a kódot. Az 1.5. ábra arra mutat példát, hogyan lehet egy latch-et létrehozni. A latch olyan egy bites memória, amely akkor frissül, amikor az engedélyező jelét magas jelszintre állítjuk. Engedélyező jel lehet egy külön vezeték vagy akár az órajel is.

```
q <= d when enable = '1' else q;  
  
process (d, enable)  
begin  
    if enable = '1' then  
        q <= d;  
    end if;  
end process;
```

1.5. ábra. *Latch leírása*

Hasonlóan lehet leírni egy D-flipflopot, azzal a különbséggel, hogy az **if**-nek van **else** ága is, és a **process** érzékenységi listájában az órajel mellett egy reset jel is szerepel. A VHDL leírása látható a 1.6. ábrán.

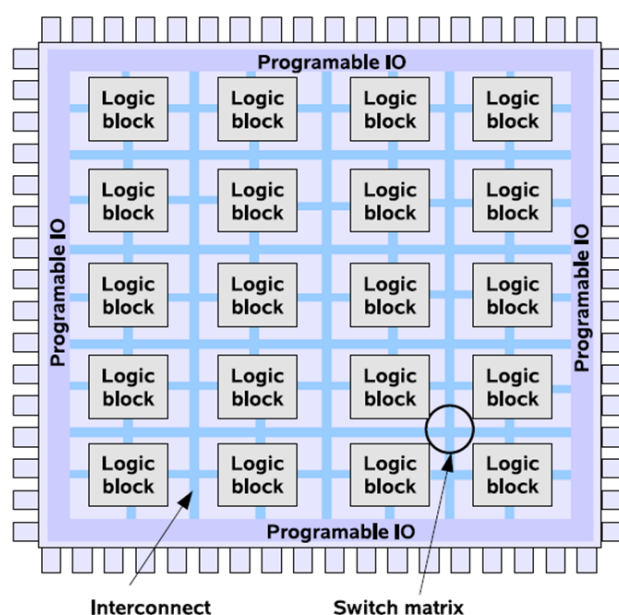
```
process (rst, clk)  
begin  
    if rst = '1' then  
        q <= '0';  
    elsif clk'event and clk = '1' then  
        q <= d;  
    end if;  
end process;
```

1.6. ábra. *D-flipflop leírása*

1.2. Az FPGA fejlesztőrendszerek

1.2.1. Általában az FPGA-król

A programozható áramkörök története nem az FPGA-kkal kezdődött; már korábban is léteztek próbálkozások és fejlett eszközök, mint a CPLD-k (Complex Programmable Logic Device), ugyanakkor mérföldkövet jelentett a megjelenésük a hardverfejlesztés területén. Az első FPGA-t 1985-ben mutatta be a Xilinx, amely három fő részből állt, ezek a programozható logikai cellák, más néven logikai blokkok, programozható összeköttetések és a programozható input/output cellák az eszköz külső részén (lásd 1.7. ábra). Az alapelv az, hogy az implementálandó funkciót kisebb modulokra bontják, minden modult egy-egy logikai blokk valósít meg, ezeket pedig a programozható összeköttetések segítségével kapcsolják össze, megvalósítva az eredeti funkciót. Az FPGA-k a felhasznált technológiától függően egyszer vagy akár számos alkalommal újraprogramozhatóak. [3]

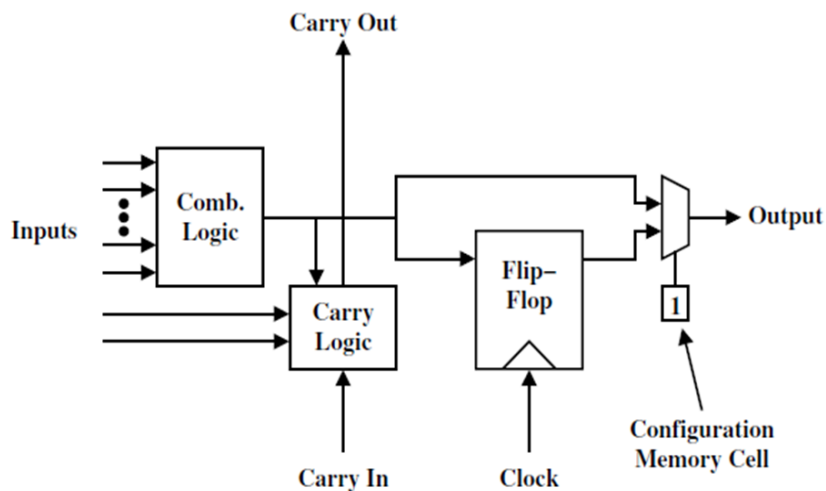


1.7. ábra. Általános FPGA elrendezés

A programozható logikai blokk

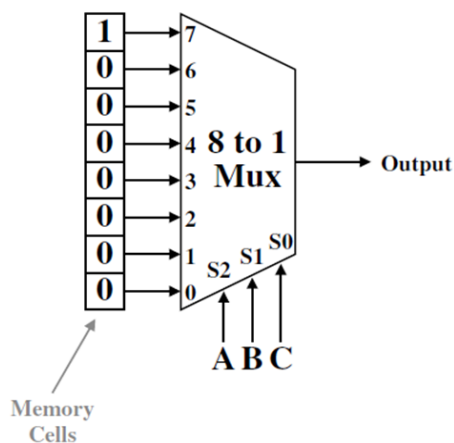
Az évek során különféle variációk láttak napvilágot a logikai blokkok tartalmáról, ugyanakkor általános kép is adható róluk. Mindegyik tartalmaz egy kombinációs logikát, néhány flip-flop-ot vagy latch-t; valamint egy blokkok közötti carry-bit átvivő logikát, amely a carry-bit kezeléséhez szükséges áramköri területet és gyorsaságot hivatott javítani. Ezen kívül multiplexerekkel és memóriákkal is kibővítik a blokkot, amelyek az

egyes részek és az egész blokk még specifikusabb funkcionalitását határozzák meg. Egy blokk lehetséges felépítését mutatja be a 1.8. ábra.



1.8. ábra. Konfigurálható logikai blokk

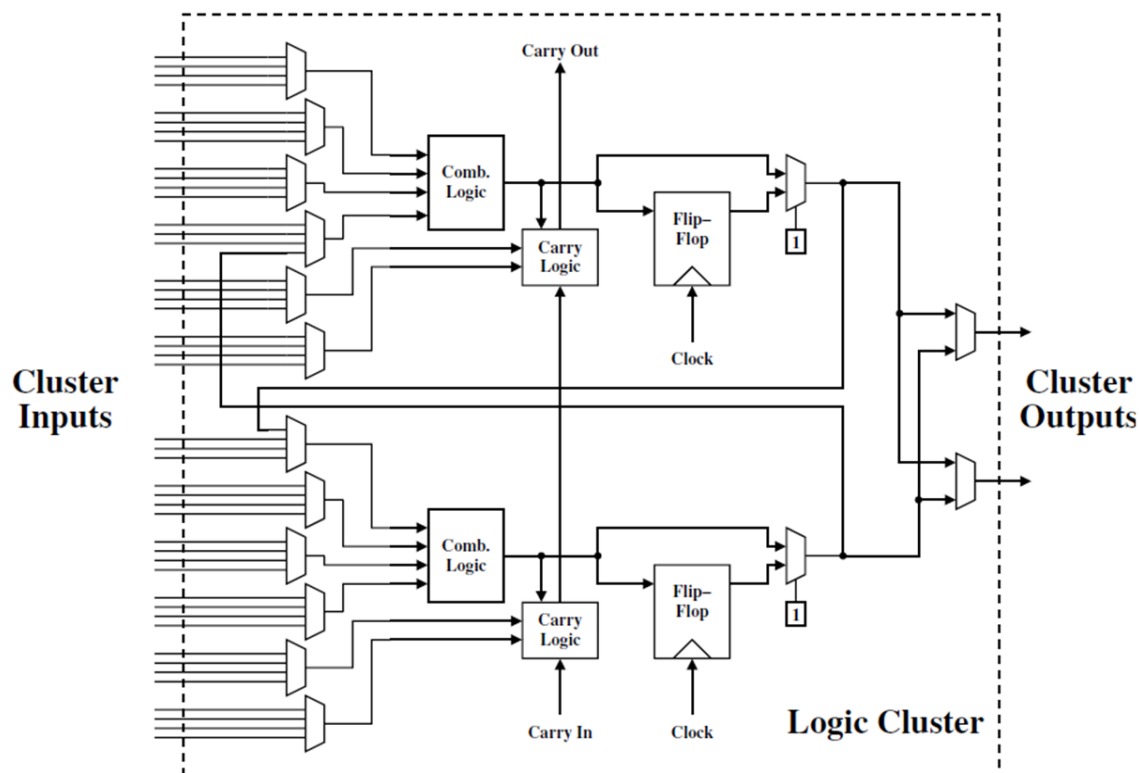
A gyártók igyekeznek az egyes részek flexibilitását a végletekig növelni. Például a flip-flop működhet latch-ként, kaphat aszinkron vagy szinkron reset-et, és akár az is meghatározható, hogy felfutó vagy lefutó él-vezérelt legyen. Hasonló módon, a carry-logika szorzóáramkörökben és full-adderekben is használható.



1.9. ábra. LUT

A kombinációs logikát is többféleképpen valósították már meg. A legelterjedtebb megoldás, hogy keresési táblaként (look-up table, LUT) implementálják, amely különféle függvények megvalósítását teszi lehetővé úgy, hogy csak az adott függvény lehetséges bemeneteit és az azokhoz tartozó kimeneteket tárolják el benne. Ezzel a módszerrel redukálható az áramkör késleltetése és a felület nagysága is. A LUT-okat leggyakrabban

multiplexerek és memóriák segítségével implementálják, a memóriákban tárolják el a lehetséges kimeneteket úgy, hogy mindegyik a MUX kiválasztó bitjei hatására kerül rá a kimenetre. A 1.9. ábrán egy 3 bites LUT található, amely a 3 bemenetnek megfelelően 8 féle kimenetet képes produkálni. Ennek megfelelően egy n bemenetű LUT 2^{2^n} féle függvényt képes megvalósítani. Az összekötő-mátrix bonyolultságának redukálása érdekében legtöbbször néhány logikai blokkot logikai fürtökbe (logic cluster) rendezik, amelyeken belül helyi, rövid és gyors összeköttetéseket alakítanak ki a blokkok között (lásd: 1.10 ábra).

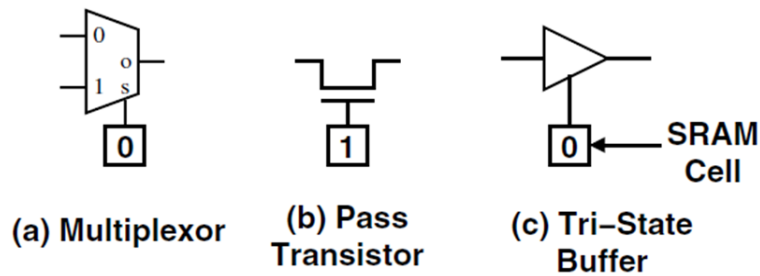


1.10. ábra. Slice cluster

A programozható összeköttetések

A logikai blokkok közötti összeköttetésekre is különféle megoldásokat dolgoztak ki az FPGA tervezők, de nem csak típusonként, hanem egy adott FPGA architektúrán belül is többféle megoldást alkalmaznak.

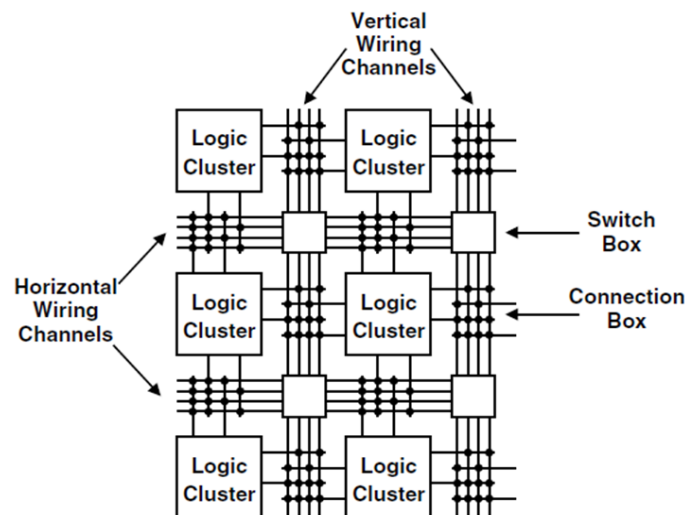
Az összeköttetések kialakításához háromféle kapcsolót alkalmaznak: ezek lehetnek multiplexer, áteresztő tranzisztor és tri-state buffer típusúak, ahogy azt a 1.11. ábra mutatja. Általában az első kettőt a logikai fürtökben használják a helyi vezetékezésnél, míg a globális összeköttetések kialakításánál mindhárom előfordul. Ezek közül is a mul-



1.11. ábra. Kapcsolási típusok

tiplererek a legelterjedtebbek, a hálózat bonyolultságától függően akár 8 bemenettel is rendelkezhetnek. A technológiából adódóan, ahányszor egy útvonalon egy tranzisztor található, annival több kapacitás rakódik a meghajtó-áramkörre, ezáltal lassulás következik be a jel terjedésében. Ezt azzal védik ki, hogy az összeköttetés-hálózatban aktív bufferelési eljárást használnak a terhelés függvényében.

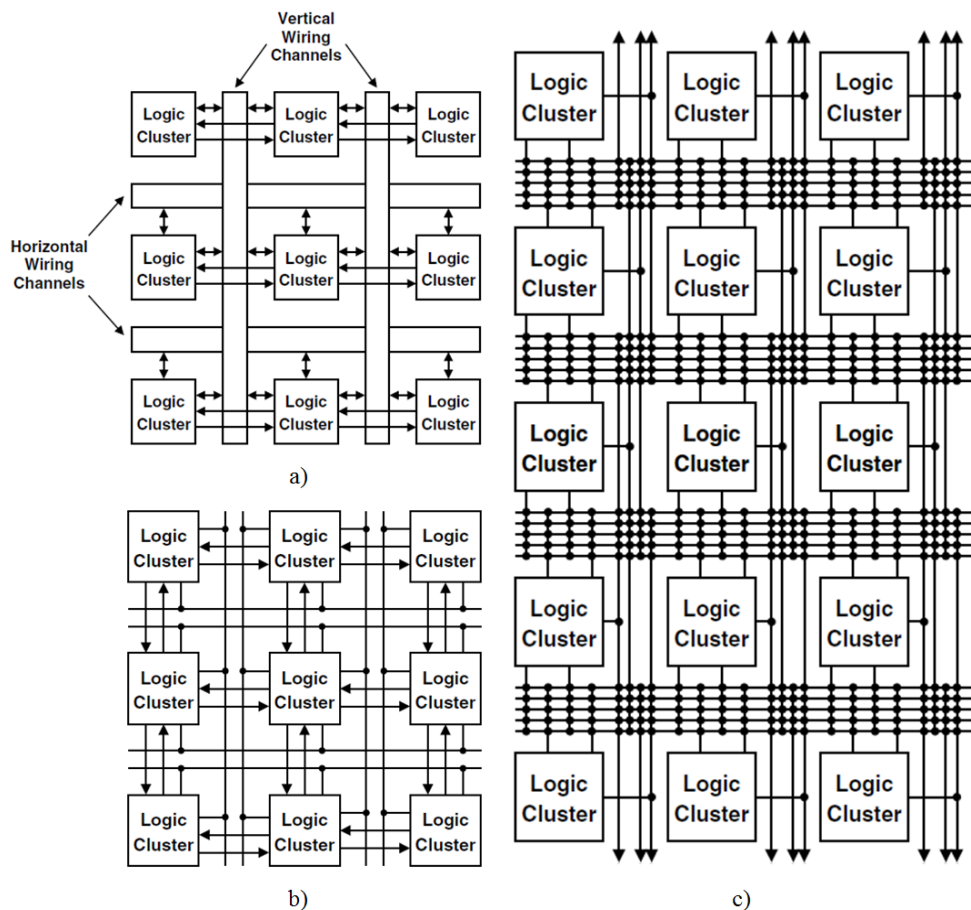
A globális összeköttetések elrendezésére is többféle variációt dolgoztak ki. Az egyik ezek közül a Xilinx által használt sziget-stílusú elrendezés (1.12. ábra). Ennél a vízszintes és függőleges vezetékek találkozásánál kapcsoló-mátrixokat (switch matrix) helyeznek el, ezekben találhatóak a programozható útválasztók. A fűrtök mind a vízszintes, mind a függőleges vezetékekhez hozzá vannak kötve. Ez azért előnyös megoldás, mert a vezetékek kisebb szakaszokra vannak bontva, így egyszerre szolgálhatnak globális és lokális összeköttetésként is.



1.12. ábra. Kapcsolási típusok

Egy másik megoldás az Altera által használt hosszú-vezetékes (long-line) megoldás. Itt

a fűrtök szintén össze vannak kötve a vízszintes és függőleges vezetékekkel, de ezek a vezetékek a chip teljes szélességéig elérnek, nincsenek megszakítva. A vízszintes és függőleges vezetékek közötti összeköttetéseket a logikai fűrtök belső összeköttetései segítségével lehet megoldani. Az újabb, több logikai blokkot tartalmazó FPGA-k megjelenésével ennél az architektúránál megnőtt a vezetékek terhelése is, így az újabb típusoknál már rövidebb vezetékszakaszokat kezdett el alkalmazni az Altera. A függőleges és vízszintes vezetékeken kívül még fűrtök közötti összeköttetéseket is kialakítottak sok Altera FPGA-ban. Ezt a struktúrát mutatja a 1.13. a) ábra. Egy harmadik típus az úgynevezett sejtes elrendezés, amely 1.13. b) ábrán figyelhető meg. Ennél igyekeznek a funkciókat úgy elrendezni, hogy minél több összeköttetést lokálisan, a logikai fűrtök között valósítsanak meg, és lehetőleg minél kevesebb vezetéket kelljen használni a globális összeköttetésekhez. Ezt azzal segítik elő, hogy a logikai blokkokat meglehetősen egyszerűre tervezik, így azok szimpla összeköttetesként is használhatóak. Nagy hátránya, hogy bonyolult, sok összetevőből álló áramköröknél jelentősen megnőhetnek a késleltetések.



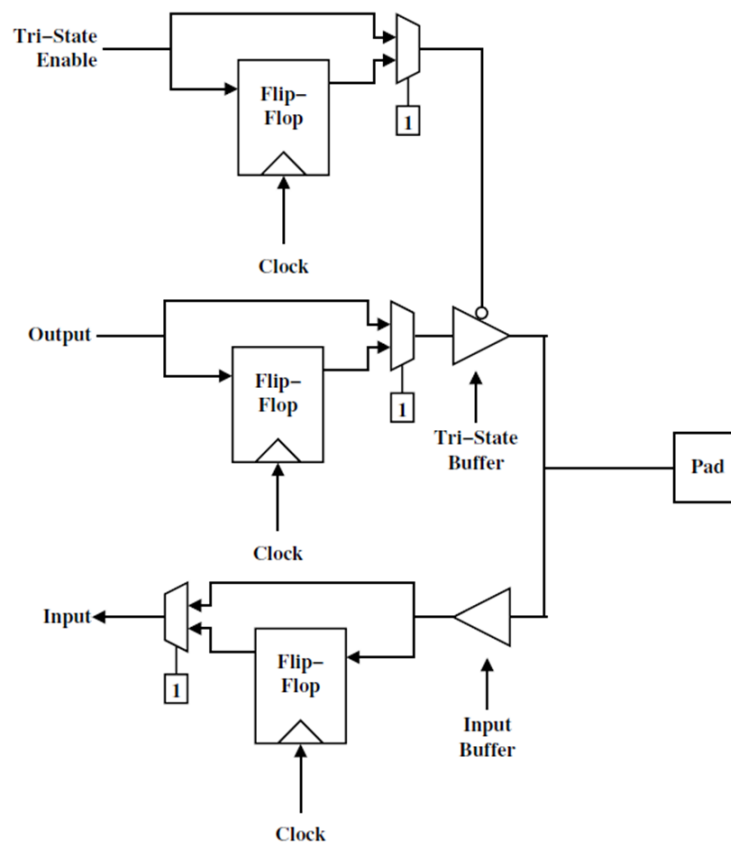
1.13. ábra. a) hosszú vezetékes, b) sejtes, c) soros FPGA cellaelrendezés

Végezetül még megemlítendő a soros architektúra (1.13. c) ábra), amely vízszintes

globális összeköttetéseket használ, ezekhez kapcsolódnak a fűrtök. Ilyen összeköttetéseket az egyszer programozható FPGA-kban használnak, újraprogramozható eszközökben ritkán látni ilyet.

A programozható I/O cellák

Az I/O lábak konfigurálhatóságára azért van szükség, hogy az FPGA még jobban testreszabhatóbb legyen, ne kelljen a tervezőknek megkötéseket követniük akkor, amikor az egyik áramkörhöz főleg bemenetekre van szükség, egy másikhoz pedig főleg kimenetekre. Ezek a blokkok a legtöbb FPGA-ban hasonló felépítésűek, nincs lényegi különbség (1.14. ábra). Programozástól függően viselkedhetnek bemenetként vagy kimenetként, konfigurálhatjuk regiszterrel vagy regiszter nélkül. A bufferelést a kimeneten tri-state, a bemeneten hagyományos buffer végzi. Ezeken kívül eszközönként sokféle szolgáltatást támogathatnak a megfelelő jelterjedés érdekében.



1.14. ábra. Programozható I/O cella

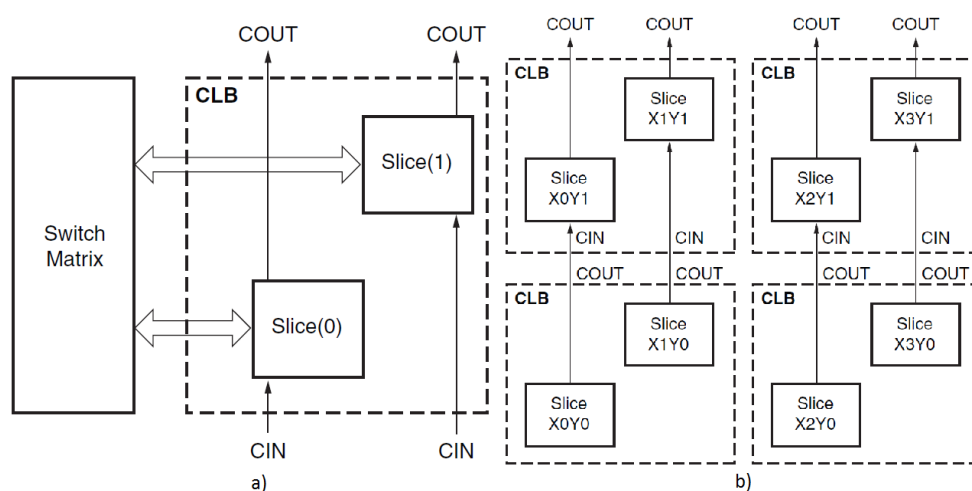
Egyéb blokkok

A fentiekén kívül előfordulnak specializált logikai blokkok is, amelyekkel az áramkörök hatékonyságát növelik. Ilyenek lehetnek a blokk memóriák, szorzó-összeadó cellák (amelyek kvázi DSP-k szerepét töltik be), soros transceiverek, ethernet MAC és PCI-Express interfészek, valamint komplett mikroprocesszorok.

1.2.2. Xilinx Virtex-5 XC5VSX240T FPGA

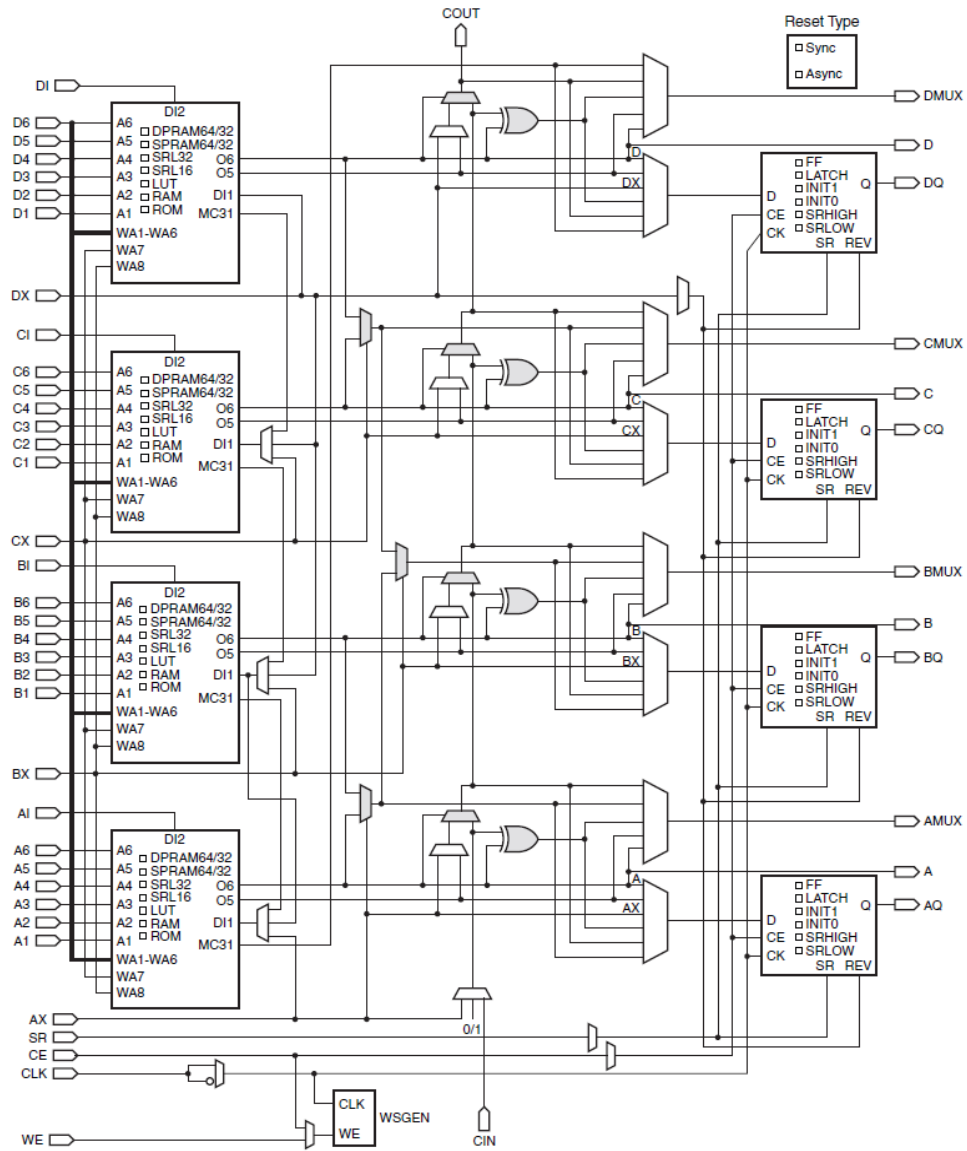
A Virtex-5-ös, 65 nm-es gyártástechnológiával készülő FPGA-családot 2009 elején vették be a piacra, felhasználási területtől függően többféle változatban kapható, ezek közül a legtöbb logikai blokkal, memóriával, DSP-procессzorral rendelkező az SX240T jelű, melyből a PPKE-ITK-n is található egy példány egy Alpha-Data ADM-XRC-5T2-es fejlesztőkártyán. Az eszköz egésze 550 MHz-es működési frekvenciára van hitelesítve, amely konfigurálható, többféle órajelet is alkalmazhatunk ugyanabban az áramkörben. Természetesen a működési frekvenciát befolyásolják az áramköri elemek késleltetései, valamint sok múlik az áramkör optimalizáltságán is. [4] [5]

Összesen 37 440 darab logikai blokkot (slice) tartalmaz, amelyek kettesével logikai fűtőkbe, úgynevezett CLB-kbe (Configurable Logic Blocks) vannak rendezve, így a chipen egy 240x78-as mátrixot alkotnak. A 1.15. a) ábrán látható a CLB-k és az összekötőmátrix közötti kapcsolat, a 1.15. b) ábra pedig a slice-ok egymáshoz viszonyított elhelyezkedését mutatja. Ahogy korábban volt róla szó, a Xilinx FPGA-in az összeköttetések tekintetében a sziget-stílusú elrendezést alkalmazza.



1.15. ábra. a) slice cluster, b) slice-ok relatív elrendezése

A Virtex-5-ös sorozatban kétféle típusú slice-ról beszélhetünk: SLICEM és SLICEL. Minden slice-ban 4 darab 6-bemenetű LUT, 4 darab tároló elem, konfigurálható multiplexerek és carry logika található; a SLICEL-ekben nincs más elem. A SLICEM-ekben ezeken kívül a LUT-ok memóriaként is használhatóak, többféle konfigurációban, például 64x1 bites 1 portos, 32x1 bites duál portos; továbbá ezek 32-bit mély shift-regiszterként is működhetnek (1.16. ábra). Ez a módszer arra szolgál, hogy kis méretű memóriákat is hatékonyan lehessen megvalósítani. Így az FPGA-n összesen 149 760 darab LUT, 4 200 kbit elosztott memória, 2 100 kbit shift regiszter és 149 760 darab flip-flop kapott helyet.



1.16. ábra. Virtex 5 slicem

Látható, hogy a LUT-okat úgy implementálták, hogy azok többféle felhasználást tegyenek lehetővé. Ezeken bármely 6 bemenetű, vagy két 5 bemenetű logikai függvény

megvalósítható; utóbbi esetben csak akkor, ha a bemenetek valamennyire összefüggenek. A slice-okon belül dedikált multiplexerek is vannak, amelyek arra szolgálnak, hogy nagyobb méretű logikai függvények is megvalósíthatóak legyenek, például két 6-bemenetű LUT-ot egy 7-bemenetűvé lehet összefűzni; ilyenkor a 7. bemenet a multiplexerhez megy, amely az egyik vagy másik LUT-ot választja ki (lásd például 1.16. ábra **CX** bemenet).

A chipen blokkmemóriák is helyet kaptak, összesen 516 darab 36 kbit-es egységről van szó, amelyek mindegyike két darab 18 kbit-es memóriaként is konfigurálható, tehát maximum 1032 darab 18 kbit blokkmemória áll rendelkezésünkre. Az FPGA-k fontos felhasználási területe a digitális jelfeldolgozás, amelyhez a legfontosabb művelet a szorzás és az összeadás, így ezen az eszközön is találhatóak úgynevezett DSP slice-ok (Digital Signal Processing), szám szerint 1056 darab, amelyekben egy 18x25 bites előjeles szorzó és egy 48 bites akkumulátor található.

1.2.3. Alpha-Data ADM-XRC-5T2 fejlesztőkártya

Ez egy PMC (PCI Mezzanine Card) szabványú kártya, amely egy hordozókártyán keresztül a PCI-Express buszhoz csatlakozik. A hordozóra két ilyen kártya tehető, egy teljes szett található jelenleg a PPKE-ITK-n, a korábban részletezett Virtex-5-ös FPGA-val felvértezve. A 1.17. ábra képen látható egy ilyen kártya, csak másik típusú FPGA-val felszerelve. [6] [7]



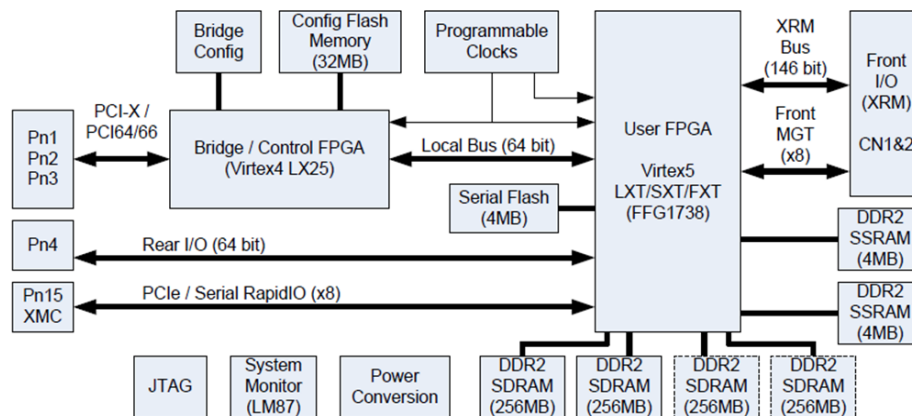
1.17. ábra. ADM-XRC-5T2-es kártya

Integrált memóriával rendelkezik, összesen 1 GB DDR-II memória van benne 4 lapkára elosztva, amelyek 666 MHz effektív frekvencián működnek. A fő FPGA-n kívül egy kisebb, Virtex-4-es FPGA is található a kártyán, amely a fő FPGA vezérlését és az interfészekkel való kapcsolattartást segíti elő.

Nagy számú csatolófelületet támogat a kártya, ilyenek a saját 146 darab LVTTTL/LVDS

(Low Voltage Transistor-Transistor Logic / Low Voltage Differential Signaling) I/O szabványú konnektorok, és a PMC hordozókártyán keresztüli 64 darab csatlakozás. Mind-egyik jelszintje konfigurálható 1,8 V, 2,5 V vagy 3,3 V-os feszültségi szintek között.

Háromféleképpen is konfigurálható a kártya, a PCI buszon keresztül, a beépített flash memóriából és a külső JTAG csatlakozóról. A kártya blokk-diagramját mutatja a 1.18. ábra.



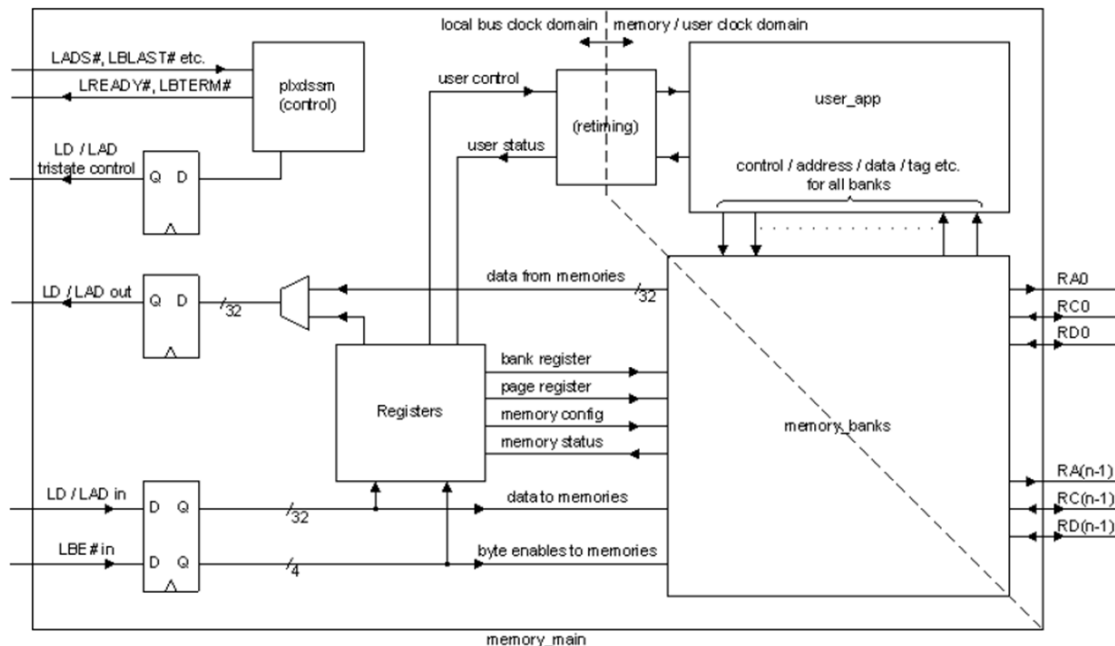
1.18. ábra. ADM-XRC-5T2-es kártya blokk diagramja

1.2.4. SDK

Az Alpha-Data a fejlesztőkártyáihoz SDK-t (Software Development Kit) is biztosít, amelyben teljes VHDL és Verilog áramköri leírások, valamint az ezek működtetéséhez szükséges CPU-n futó programok is találhatóak. Ezek: DMA, I/O vezérlő, megszakítás-kezelés, regiszterek elérése a PCI-Express buszon keresztül, órajel-konfiguráció, valamint memóriavezérlő. Az ebben a dokumentumban részletezett problémához a memóriavezérlő példaáramkör áll a legközelebb, ezt fogom most bemutatni. [8]

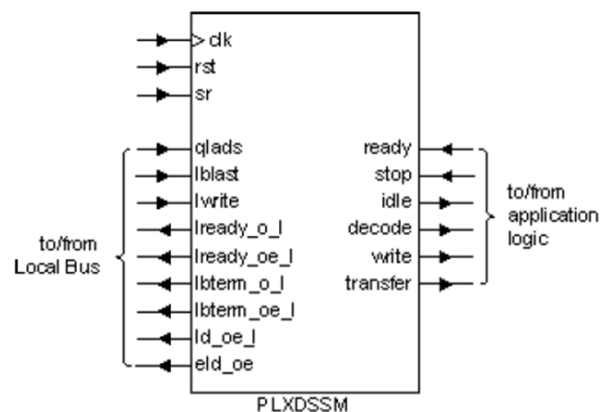
A memóriavezérlő három nagy logikai egységre bontható. Ezek a Local Bus Interface, amely a PCI-Express buszon keresztül összeköttetést biztosít a CPU és FPGA-s memória között; az User Application, amelyben gyárilag egy memóriateszt foglal helyet, de kicserélhető saját kóddal is; és a Memory Ports, amely összeköttetést biztosít a Local Bus Interface, az User Application és a fizikai memóriák között. A 1.19 ábrán is elkülöníthetők ezek a részek, de jobban látszódnak az összeköttetések. Annak érdekében, hogy az áramkör működése le legyen választva a PCI-Express busz órajeléről, kétféle órajelen üzemel az áramkör, melyek fázisban és frekvenciában is eltérhetnek egymástól. Maga a vezérlő úgy lett kialakítva, hogy a lehető legkevésbé függjön a fizikai memória típusától;

ugyanaz az áramkör kis változtatásokkal képes működni DDR-II-es, DDR-III-as, vagy akár régebbi memóriákkal is.



1.19. ábra. A memóriavezérlő blokkdiagramja

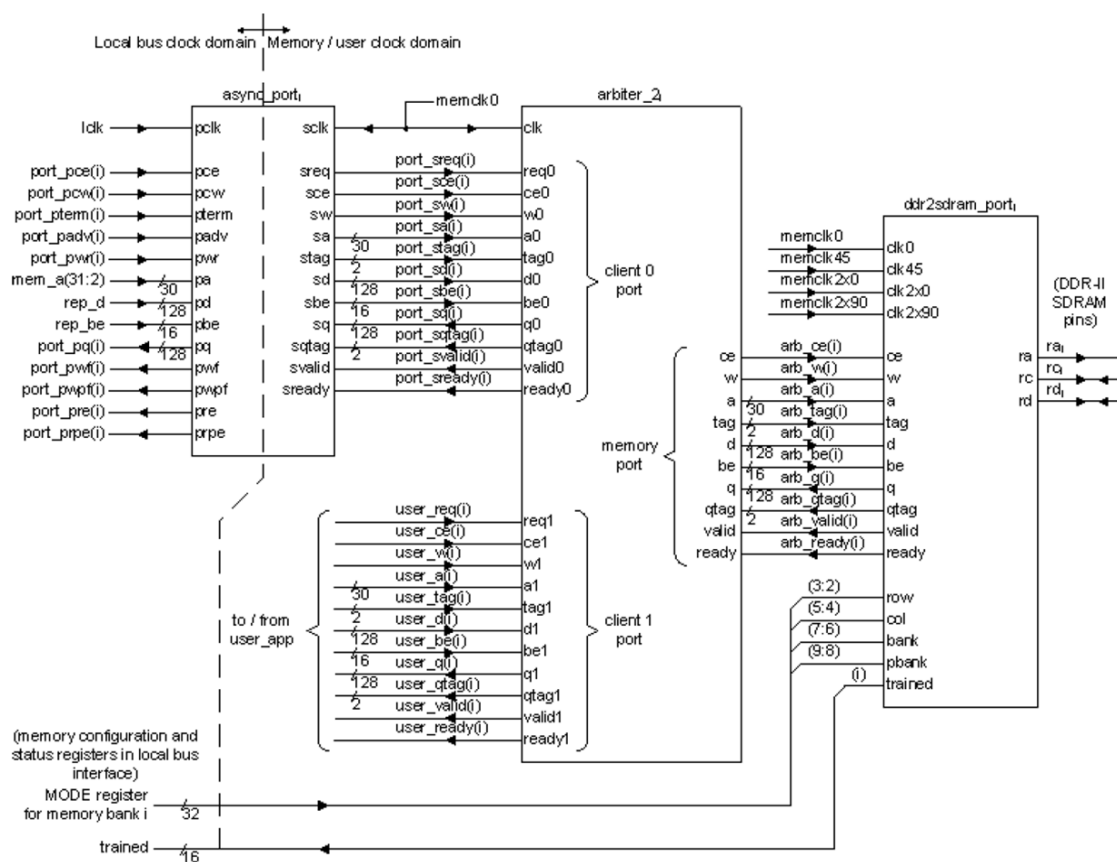
Mivel az FPGA-hoz rendelhető PCI címtartomány mérete korlátozott, így a memóriavezérlő két részre osztja a teljes rendszermemóriát: egy alsó, 2 MB-os részre, amely regiszterek formájában van jelen az FPGA-n, és egy 2 MB fölötti részre, amely a külső memóriát jelenti. Ez egyrészt arra szolgál, hogy a 2 MB alatti regiszterekkel közvetlen kapcsolatot tudunk tartani az áramkörünkkel, másrészt a címtartomány folyamatos, így ugyanazzal a címmel tudjuk a regisztereket és a memóriát is elérni a PCI-Express interfész felől.



1.20. ábra. PLXDSSM

Fontos része az áramkörnek a PLXDSSM nevű állapotgép, amely direct-slave átvite-
leket képes kezelni a PCI-Express busz felé. Ennek következtében minden áramkörben
csak egy darab lehet belőle. Önmagában csak a vezérlést biztosítja, az adatútvonalakat
külön kell kezelni. A blokk diagramját mutatja a 1.20. ábra.

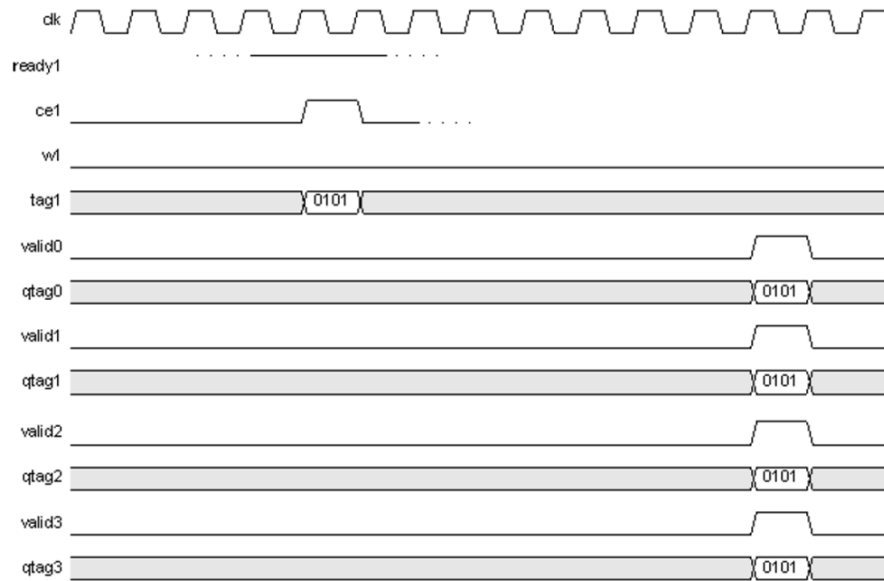
Mivel a fizikai memóriákat egy időben használhatja több csatornán is a saját áram-
körünk, valamint a PCI-Express busz felől is folyamatosan elérhetőnek kell lennie, így
arbitrációra van szükség. Minden fizikai memóriamodulhoz szükség van egy-egy arbit-
rációs egységre, az ezekhez szükséges összeköttetéseket mutatja a 1.21. ábra, amelyen
ugyan egy 2 interfésszel rendelkező arbiter látható, de az SDK-ban 3 és 4 interfészes
is rendelkezésre áll, amelyek mindegyike fizikailag a 4 interfésszel rendelkezőből vannak
származtatva. Egy arbiter úgy működik, hogy a memóriaportokat osztja el több felé úgy,
hogy az interfészei felől úgy tűnik, mintha közvetlenül a memória-interfészhez csatlakoz-
nának a modulok, de az arbiter egy előre definiált algoritmus segítségével időszelleteket
oszt ki a memória-interfész használatára. Természetesen a hatékonyság növelése érdeké-
ben csak az kaphatja meg a memória-interfészt, amelyik kérte is.



1.21. ábra. Az arbitrációs egység blokkdiagramja

A memória-interfész, így az arbiter portjai is a következőképpen működnek:

- A **ce** jel jelzi a memória-interfésznek, hogy hozzáférést akarnak a memóriához. Minden órajel-ciklusban, amíg a **ce** jel magas, egy parancs adódik ki a memóriának.
- A **w** jel a **ce** jellel együtt adja meg, hogy a parancs az írást (1) vagy olvasást (0) jelöl.
- Az **a** jel szintén a **ce** jellel együtt értelmes, és egy memóriacímet tartalmaz.
- A **be** jel kiértékelése a **ce** és a **w** jelek logikai ÉS kapcsolatán múlik, és az aktív magas byte engedélyező jel szerepét tölti be egy írás során. Amikor a **be** i. bitje 1, az i. byte lesz írva; ha pedig 0, akkor az i. byte nem lesz írva.
- A **d** jel szintén a **ce** és a **w** jelek logikai ÉS kapcsolatának eredményeként kerül kiértékelésre, így egy írási utasításkor lesz figyelembe véve.
- A **tag** jel a **w** jel negáltjának és a **ce** jelnek a logikai ÉS kapcsolatával kerül feldolgozásra, így egy olvasási utasításkor van értelmezve. Miután a memória feldolgozza az olvasási kérést, a kívánt adat mellett a **qtag** jelen jön vissza a kiadott tag, megjelölve, hogy az adat melyik olvasásnak lett az eredménye. A **tag** jelölhet egy eszközt, de olvasási parancsonként is változhat, ezt az áramkör tervezője határozza meg.
- A **q** jelen egy olvasási parancs eredménye jelenik meg a memória felől, és a **valid** jel mondja meg, hogy valódi adat van-e a rajta vagy sem.
- A **qtag** jel az olvasási parancskor kiadott **tag** jel értékét hordozza, és a **valid** jel adja meg az érvényességét.
- A **valid** jel azt adja meg, hogy egy olvasási parancs után az eredmény rajta van-e a **q** és a **qtag** jeleken vagy sem.
- A **ready** jel azt jelzi a memória-interfész felől, hogy kiadhatunk-e parancsot vagy sem. Amennyiben a **ready** 0, a **ce** jelnek nem szabad rákötve lennie az interfészre.



1.22. ábra. A *valid* és a *qtag* jelek összefüggése

Fontos tény, hogy amikor kiadunk egy parancsot és a hozzá tartozó tag-et, akkor a végrehajtás után az összes **qtag** vonalon megjelenik az előzőleg kiadott tag. Tehát egy olvasási művelet eredményét nem feltétlenül kell ugyanannak az eszköznek értelmeznie, mint amelyik elindította azt. Természetesen ilyenkor a **valid** jelek szintje is magas lesz, ahogyan azt a 1.22. ábrán láthatjuk.

Ezekon kívül még másféle portok is előfordulhatnak a memória típusától függően. DDR-II esetén ezek a **row**, **col**, **bank** és **pbank** jelek, amelyek a memória konfigurálásához és vezérléséhez szükségesek.

2. fejezet

Elektromágneses tér szimuláció

2.1. Bevezetés

Az elektromágneses tér szimuláció során a Maxwell-egyenletek megoldása szükséges (2.1-2.6), mivel a Maxwell-egyenletek megoldása nagy számítási teljesítményt igényel, ezért az egyenletek megoldása helyett mindig valamilyen közelítő modellt alkalmazunk. Jelen esetben közelítő modellként a Transmission Line Modelling módszert (TLM) használjuk fel, hogy adott pontokban meg tudjuk határozni az elektromos- illetve mágneses térerősség értékeket.

$$\epsilon_0 \epsilon_x \frac{\partial e_x}{\partial t} = \frac{\partial h_z}{\partial y} - \frac{\partial h_y}{\partial z} - \sigma_{ex} e_x \quad (2.1)$$

$$\epsilon_0 \epsilon_y \frac{\partial e_y}{\partial t} = \frac{\partial h_x}{\partial z} - \frac{\partial h_z}{\partial x} - \sigma_{ey} e_y \quad (2.2)$$

$$\epsilon_0 \epsilon_z \frac{\partial e_z}{\partial t} = \frac{\partial h_y}{\partial x} - \frac{\partial h_x}{\partial y} - \sigma_{ez} e_z \quad (2.3)$$

$$\mu_0 \mu_x \frac{\partial h_x}{\partial t} = \frac{\partial e_y}{\partial z} - \frac{\partial e_z}{\partial y} - \sigma_{mx} h_x \quad (2.4)$$

$$\mu_0 \mu_y \frac{\partial h_y}{\partial t} = \frac{\partial e_z}{\partial x} - \frac{\partial e_x}{\partial z} - \sigma_{my} h_y \quad (2.5)$$

$$\mu_0 \mu_z \frac{\partial h_z}{\partial t} = \frac{\partial e_x}{\partial y} - \frac{\partial e_y}{\partial x} - \sigma_{mz} h_z \quad (2.6)$$

2.2. 3D Transmission-Line Modeling Módszer

2.2.1. Transmission-Line Modeling Módszer

A Transmission-line modelling módszer alapjait 1971-ben [9] publikálták először, azóta hivatkozhatunk rá a TLM rövidítéssel. A modellt Johns kiterjesztette három dimenzióra is 1987-ben ([10]). A TLM diszkrét modellje az elektromágneses tereknek, amely

alkalmazható 1- 2- és 3- dimenziós esetben, idő és frekvenciafüggő alkalmazásokban. Ez egy explicit numerikus módszer, amelynek alapja, hogy a hullámterjedést egy elektromos vezetéken terjedő jelként szimuláljuk. Ezen típusú hullámforma terjedés szimulációja miatt nemcsak elektromos rendszerekben történő jelterjedésre lehet felhasználni a módszert, hanem mondjuk hangterjedés szimulációjára is alkalmazható.

A TLM modell vezeték darabokból, ellenállás részekből és forrásokból van felépítve. Az induktivitások és kapacitások helyettesíthetők soros vagy párhuzamos vezeték darabokkal. Az ellenállás modellezhető egy végtelen hosszú vezetékcsonkkal. Az eljárás lényege, hogy a folytonos rendszert egy koncentrált elemekből álló rendszerrel helyettesítjük, azaz a megoldáshalmazt egy négyzet alakú hálóra („mesh”) bontjuk, és ezekkel reprezentáljuk a távvezetéseket. A modell a teret diszkrét pontokra osztja („node”) és az egyes pontokat a tér három irányában távvezeték modulokkal köti össze.

Kihasználva a távvezetésekre vonatkozó egyenletek és az elektromágneses tér alapösszefüggései közötti analógiát, a modell térszámításra is használható. Így számos, de hasonló alakú differenciálegyenletet kell megoldani, ám azok megoldása algoritmizálhatóbb, s így számítógépes szimuláció segítségével a tér jellemzői közelítőleg meghatározhatók.

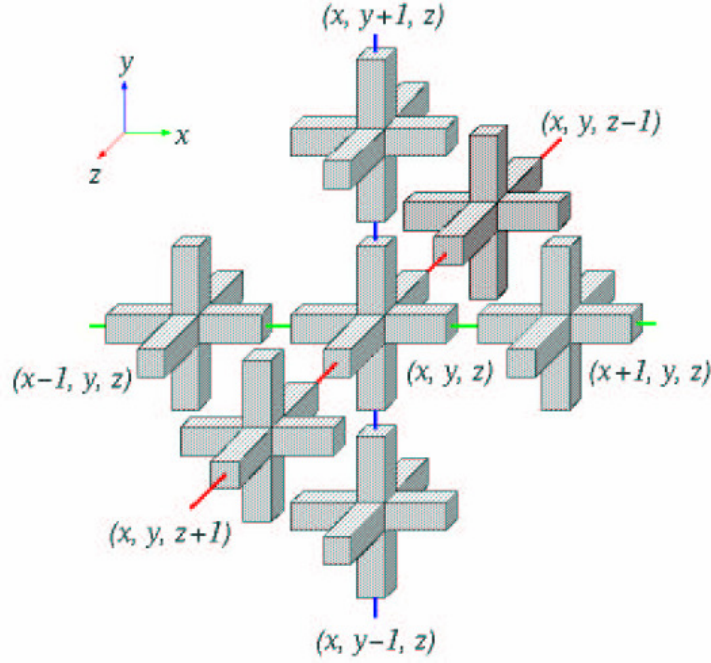
A rendszerre vonatkozó alapegyenletek a Kirchhoff-törvényekből számíthatók illetve egyszerűsíthetők. A kétdimenziós modell az ún. „négykapu” realizációt alkalmazza, vagyis a tér egyes pontjai négyzetrácsot alkotnak, és egy-egy pontban „négykapukat” helyezve, azok karakterisztikáját meghatározva tudjuk kiszámolni az adott pont térjellemzőit.

Annak érdekében, hogy a kétdimenziós TLM módszert kiterjesszük három dimenzióra, az eredeti négyzetrácsot ki kell terjeszteni egy kocka alakú rácsszerkezetté, ahol az egyes „node”-ok mind a 6 közvetlen szomszédjukkal össze van kötve. A TLM módszer esetén a háromdimenziós problémát „tizenkét-kapukból” álló rendszer modellezi. Ez azt jelenti, hogy jelen esetben az egyes „node”-oknak nem 4, hanem 6 szomszédja lesz, méghozzá térbeli elrendezésben. (2.1. ábra)

2.2.2. Hibrid Symmetrical Condensed Node

A három dimenziós "csomópontok"-nak több fajtája van, a legegyszerűbb felírás a symmetrical condensed node (SCN)[10] és a legkomplexebb a general symmetrical condensed node (GSCN)[11].

A symmetrical condensed node egy 12 portos megvalósítás, amely megadja egy "kis"



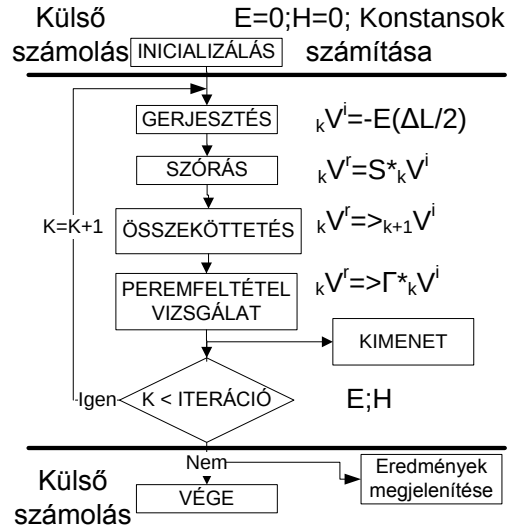
2.1. ábra. 3D TLM SCN input voltage polarization

térész elektromágneses térerősségét. A general symmetrical condensed node, viszont modellezni képes a különböző közegeket elektromos, mágneses és veszteséges szempontjából [11].

A hybrid symmetrical condensed node (HSCN)[12] modell komplexitásban a symmetrical condensed node és a general symmetrical condensed node módszerek között található. Ez a módszer a legjobban alkalmazható a párhuzamos architektúrákon való futtatásra, ezért is esett erre a módszerre a választás.

2.3. Az FPGA-s implementáció

A TLM modell számításának struktúrája az 2.2 ábrán látható. Bizonyos adatokat már offline is ki lehet számolni illetve meghatározni ezzel csökkentve a későbbi memória sávszélességet. A memória interfész felé a modell minden egyes időpillanatban 7 időtől függő paramétert ($E_x, E_y, E_z, H_x, H_y, H_z, Y$) fog közvetíteni 32 vagy 64 biten. Az FPGA belső memóriájába a három dimenziós struktúrából már 2 lapot mindig betöltünk, ezzel elérhető az hogy egy csomópont számolásához mindig elég egy csomópontot a rendszernek átadni. Az időfüggő paramétereken kívül – amik eddig 56 vagy 28 byte-ot tesznek ki attól függően, hogy 32 vagy 64 bits értékeket használunk – a rendszerbe be kell vinni egy a konstans paramétereket illetve értékeket kiválasztó byte-ot is. A paramétereket



2.2. ábra. A megvalósított architektúra

előre felöltjük az FPGA belső memóriájába és mindig a számunkra megfelelő értékeket címezzük meg. Ezek a lépések mind a külső számításokban elvégzendő feladatok. A memória interfész ekkor kapja az egyik nagy terhelést hiszen a fent leírtak szerint ilyenkor kell felölteni az FGPA belső memóriáját, hogy a számításokat elkezdhessük.

A számítások menete a következő lépésekben zajlik a számítási struktúra szerint (2.2. ábra):

GERJESZTÉS

Ebben a számítási lépésben a főmemória felől (első időlépésben) vagy az FPGA belső memóriája felől (többi időlépésben) érkező elektromos- és mágneses térerősségeket átalakítjuk feszültség értékekké. Összesen 12 feszültségértéket fogunk képezni. Ezek lesznek az adott csomópont előreterjedő feszültségértékei.

SZÓRÁS

Ez a leg számításigényesebb feladat. Itt elő kell állítanunk a szórás mátrixot, amelyek szórás mátrix megadja a jelterjedésre vonatkozó paramétereket. A szórás mátrix elemei következő egyenletekkel írhatók le (2.7-2.20):

$$a_{pp} = -\frac{Y_0 + G_s + 2(Y_l - Y_t)}{2[Y_0 + G_s + 2(Y_l + Y_t)]} \quad (2.7)$$

$$b_{pq} = -\frac{2Y_t}{Y_0 + G_s + 2(Y_l + Y_t)} \quad (2.8)$$

$$c_{pp} = -[\frac{Y_0 + G_s + 2(Y_l - Y_t)}{2[Y_0 + G_s + 2(Y_l + Y_t)]} + \frac{R_t Y_t}{2(R_t Y_t + 4)}] \quad (2.9)$$

$$d_{pq} = \frac{2}{R_t Y_t + 4} \quad (2.10)$$

$$g_{pq} = \frac{2Y_0}{Y_0 + G_s + 2(Y_l + Y_t)} \quad (2.11)$$

$$h_{pq} = \frac{Y_0 - G_s - 2(Y_l - Y_t)}{Y_0 + G_s + 2(Y_l + Y_t)} = g_{ij} - 1 \quad (2.12)$$

$$k_{pq} = \frac{1}{Y_0 + G_s + 2(Y_l + Y_t)} \quad (2.13)$$

$$V_{16} = j_x Z_0 \Delta z \Delta y \quad V_{17} = j_y Z_0 \Delta x \Delta z \quad V_{18} = j_z Z_0 \Delta x \Delta y \quad (2.14)$$

$$G_{S_p} = \sigma_{ep} \frac{\Delta q \Delta r}{\Delta p} [S] \quad (2.15)$$

$$R_{mp} = \sigma_{mp} \frac{\Delta q \Delta r}{\Delta p} [\Omega] \quad (2.16)$$

$$Y_{pq} = \frac{1}{Z_{pq}} = \sqrt{\frac{C_{pq}}{L_{pq}}} [S] \quad Y_{O_q} = \frac{2C_O^i}{\Delta t} [S] \quad (2.17)$$

$$\Delta t = \Delta p \sqrt{C_{pq} L_{pq}} [m/s] \quad (2.18)$$

$$C_{rp} \Delta r + C_{qp} \Delta q + C_O^p = \varepsilon_0 \varepsilon_p \frac{\Delta q \Delta r}{\Delta p} [F] \quad (2.19)$$

$$L_{rp} \Delta r + L_{qp} \Delta q = \mu_0 \mu_p \frac{\Delta q \Delta r}{\Delta p} [H] \quad (2.20)$$

ahol R_t – a mágneses veszteség $[\Omega]$ $t = x, y$ vagy z irányokban;

G_s – az elektromos veszteség $[S]$ $s = x, y$ vagy z irányokban;

C_{pq} – a 6 vonal kapacitása $[F/m]$;

L_{pq} – a 6 vonal induktivitása $[H/m]$;

C_{O_p} – a 3 nyitott csomópont kapacitása $[F]$;

p, q, r -ek helyettesítendőek az x, y, z irányokkal.

A szórásnézet elemeinek számítása után, hogy megkapjuk a megfelelő irányok szórási paramétereit, össze kell szoroznunk a szórásmátrixot (2.1) az előző lépésben kiszámolt feszültségek vektorával.

2.1. táblázat. A szórásmátrix

		Y_l	x	x	x	x	y	y	y	y	z	z	z	z	Capacitors			Sources		
		Y_t	z	z	y	y	z	z	x	x	y	y	x	x						
		Y_s	y	y	z	z	x	x	z	z	x	x	y	y	x	y	z	x	y	z
R_t	G_s		1y	2y	3z	4z	5x	6x	7z	8z	9x	10x	11y	12y	13	14	15	16	17	18
y	x	9x			d_{zx}	$-d_{zx}$	b_{zx}	b_{zx}			a_{zx}	c_{zx}			g			k		
y	x	10x			$-d_{zx}$	d_{zx}	b_{zx}	b_{zx}			c_{zx}	a_{zx}			g			k		
	x	13x					b	b			b	b			h			k		
z	x	5x	d_{yx}	$-d_{yx}$			a_{yx}	c_{yx}			b_{yx}	b_{yx}			g			k		
z	x	6x	$-d_{yx}$	d_{yx}			c_{yx}	a_{yx}			b_{yx}	b_{yx}			g			k		
z	y	1y	a_{xz}	c_{xy}			d_{xy}	$-d_{xy}$					b_{xy}	b_{xy}	g			k		
z	y	2y	c_{xy}	a_{xy}			$-d_{xy}$	d_{xy}					b_{xy}	b_{xy}	g			k		
	y	14y	b	b									b	b	h			k		
x	y	11y	b_{zy}	b_{zy}					d_{zy}	$-d_{zy}$			a_{zy}	c_{zy}	g			k		
x	y	12y	b_{zy}	b_{zy}					$-d_{zy}$	d_{zy}			c_{zy}	a_{zy}	g			k		
y	z	3z			a_{xz}	c_{xz}			b_{xz}	b_{xz}	d_{xz}	$-d_{xz}$					g			k
y	z	4z			c_{xz}	a_{xz}			b_{xz}	b_{xz}	$-d_{xz}$	d_{xz}					g			k
	z	15z			b	b			b	b							h			k
x	z	7z			b_{yz}	b_{yz}			a_{yz}	c_{yz}			d_{yz}	$-d_{yz}$			g			k
x	z	8z			b_{yz}	b_{yz}			c_{yz}	a_{yz}			$-d_{yz}$	d_{yz}			g			k

ÖSSZEKÖTTETÉS

Ebben a lépésben az előzőleg megkapott szórt értékeket kell hozzárendelni a következő tér illetve időlépésekben számolandó node-ok megfelelő bemeneteihez.

PEREMFELTÉTEL VIZSGÁLAT

A peremfeltétel vizsgálat lépésben ellenőrizni kell hogy az adott kiszámolt pont nincsen e közel a szimulálandó tér pereméhez, ha esetleg igen akkor alkalmazni kell a megfelelő peremfeltételt, amelyet a felhasználó a szimulálandó feladat szerint előre megválaszt.

KIMENET

Ebben a lépésben válik ismét szükségessé a memória interfész használata, hiszen a kiszámolt értékeket az FPGA belső memóriájából a fő memóriába kell visszatöltenünk.

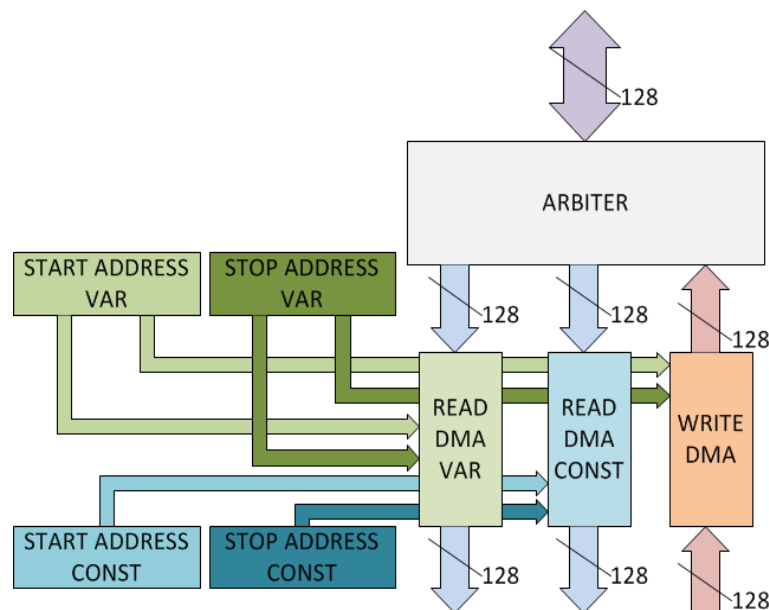
EGYÉB KÜLSŐ SZÁMÍTÁSOK

A memória interfész által a fő memóriába küldött adatokat használjuk később be-mentként a megjelenítéshez, amely megjelenítő segítségével láthatóvá válik 2 illetve 3 dimenzióban is az áramkörben változó elektromos- és mágneses terek mértéke.

3. fejezet

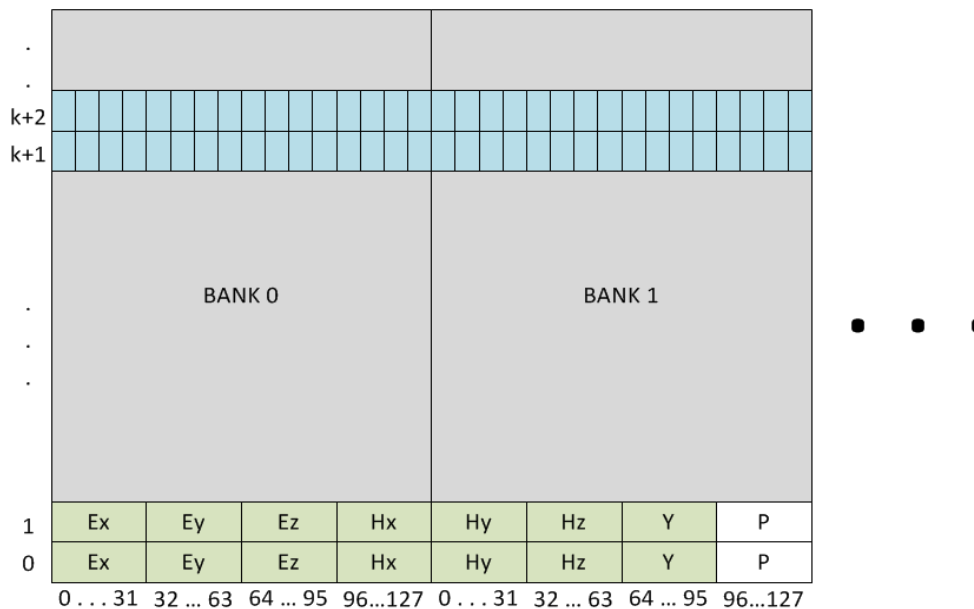
Az áramkör

Ahogy korábban már rámutattam, az Alpha-Data SDK memória-vezérlő példaáramkörét vettem alapul a probléma megoldásához. Ez azért logikus lépés, mert egy memória-vezérlő megírása nagyon komplex, az eredeti szimulációs problémán túlmutató feladat lenne. Továbbá, az aritmetikai egység is rendelkezésemre áll, így a kettő közötti kapcsolatot, vagyis egy DMA-vezérlőt kell létrehoznom. Itt valójában három darab DMA-vezérlőről van szó, két olvasóról és egy íróról. Az előbbieket feladata az adatok beolvasása a memóriából az aritmetikai egység felé, míg utóbbié az eredmények visszaírása a memóriába. Mivel ez a három DMA egység konkurens egymással, így ezeket egy arbitrációs egységen keresztül kell csatlakoztatni a memória-interfészhez. Az elképzelt struktúra vázlata látható a 3.1. ábrán.



3.1. ábra. A DMA-k elrendezése

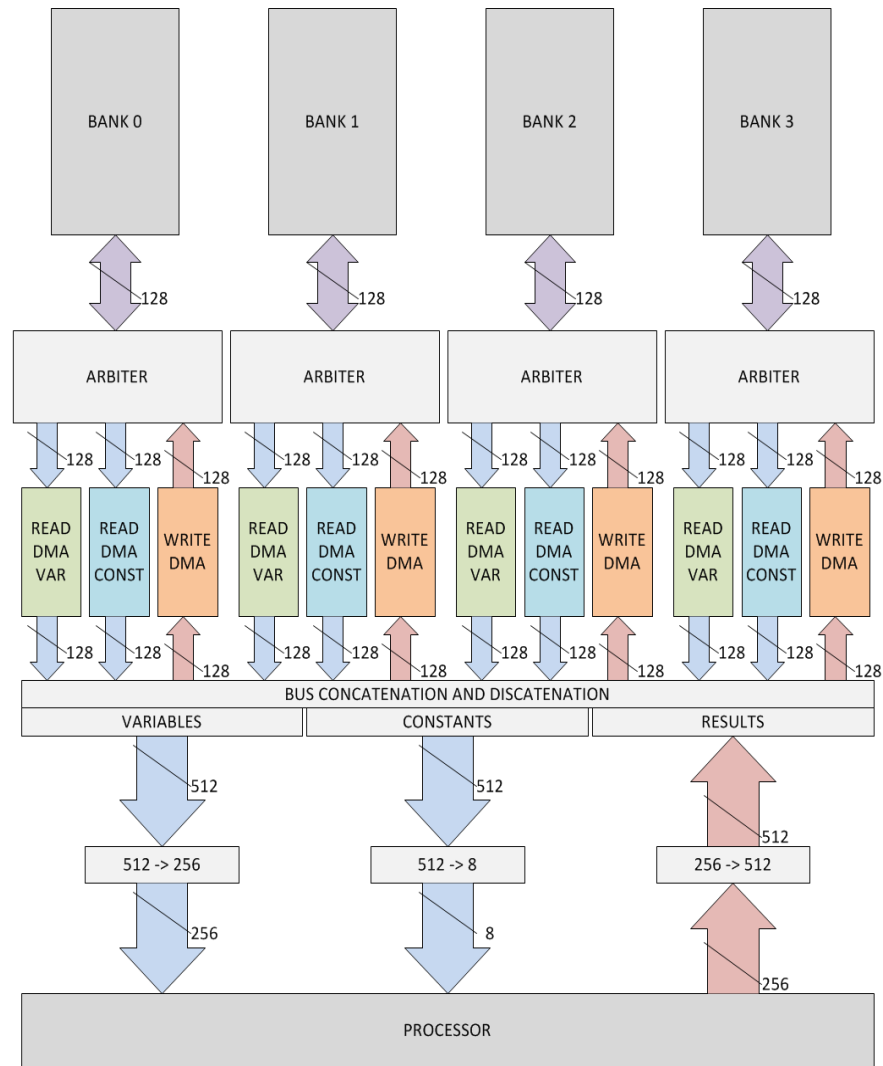
A memória-interfész 128 bit széles, ezt osztja el az arbiter négyfelé. Az egyik olvasó DMA a memória egyik feléből, a másik a másik feléből olvassa be az adatokat; ezért van szükség a két DMA-nak két különböző **START** és **STOP** címre. Az író DMA címtartománya megegyezhet az alsó memóriaterületet olvasó DMA címtartományával, ugyanis a kiolvasott adatokat általában ugyanoda kell visszaírni a számítás után. Ez megfelel az adott cella állapotfrissítésének. Minden DMA-nak vannak engedélyező bemenetei, amelyeken keresztül az aritmetikai egység jelzi számára, hogy szüksége van-e adatra, avagy elkészült-e a számítással és tovább akarja adni az eredményt. Természetesen ehhez adatútvonalak is szükségesek, amelyek át megkapja az adatokat, illetve visszaadja a végeredményt. Nyilván ezeken kívül további jelek is szükségesek, amelyekkel a számítást el lehet indítani, meg lehet állítani, az aritmetikai egység visszajelelhet, hogy végzett, illetve kontrollálható az, hogy hány iterációt hajtson végre az áramkör az eseménytéren.



3.2. ábra. A memória felosztása

A memória két részre lesz felosztva (3.2 ábra): az alsó részén tároljuk el a cellákhoz tartozó állapotokat, jelen példában az Ex, Ey, Ez, Hx, Hy, Hz és Y értékeket, a felső részén pedig minden cellának egy konstans paraméterét, például 1, ha nincs fal és 0, ha van, tehát kell-e számolni az adott cellában vagy sem, továbbá azt, hogy a számolandó cellában milyen anyaggal dolgozunk; ez a feldolgozó egységtől függ. Mivel a memóriába sorfolytonosan töltjük be az adatokat, így biztosítani kell, hogy a feldolgozó egységnek minden lépésben rendelkezésre álljon egy cellányi változó és a hozzá tartozó konstans érték. A feltételezés az, hogy 32 bites változóink vannak, amelyeket egy 256 bites tömbben

tárolunk, a hozzájuk tartozó konstansok pedig 8 bitesek. Azért, hogy az adatok helyes sorrendben kerüljenek a feldolgozóegységhez, a DMA-kból érkező 4 darab 128 bites buszt egy darab 512 bit széles busszá fogjuk össze, melyet aszimmetrikus FIFO-k segítségével 256 illetve 8 bites buszokká redukáljuk le. A teljes áramkör blokkdiagramját mutatja a 3.3 ábra.

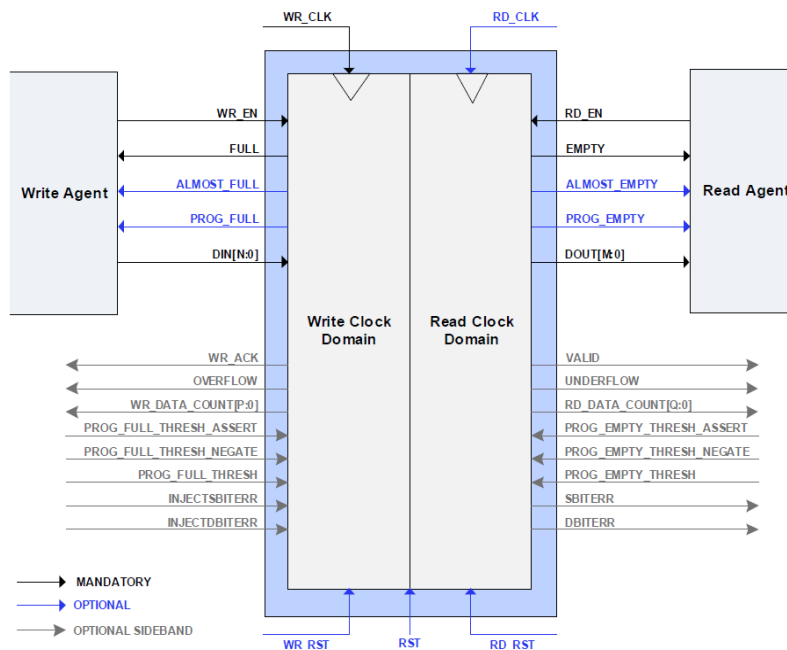


3.3. ábra. A teljes áramkör blokkdiagramja

3.1. FIFO

Az áramkörben használt konfigurálható FIFO-k többféle ki- és bemenettel rendelkeznek, melyek közül 6 darab bemenet (**Din**, **wr_en**, **wr_clk**, **rd_en**, **rd_clk**, **rst**), és 7 darab kimenet (**Dout**, **full**, **almost_full**, **empty**, **almost_empty**, **prog_full**, **prog_empty**) fontos a mi szempontunkból. A **Din** és **Dout** buszok szélessége az alkalmazástól függ, de ezt a fejlesztőeszköz sajátosságai miatt a Core Generator-ban kell

megadni, tehát nem állíthatóak dinamikusan. A többi jel 1 bites, jelzésre és vezérlésre szolgálnak. A **prog_full** és a **prog_empty** jeleket szintén a Core Generator-ban kell beállítanunk, és arra szolgálnak, hogy jelezzék, a megadott számú adat benne van-e a FIFO-ban vagy sem. A **wr_clk** és az **rd_clk** órajelek és arra szolgálnak, hogy az írást és a kiolvasást eltérő ütemben lehessen végrehajtani, pl. olyankor, amikor a memória-interfész és az aritmetikai egység eltérő frekvencián üzemel. Az áramkör jelenlegi állapotában mindkettő a közös **clk** órajelre van kötve. Az **rst** jellel lehet a FIFO-t reset állapotba hozni. [13] Az áramkör 5-féle FIFO-t használ, melyek csak az írási-olvasási bitszélességükben különböznek: az író- és olvasó DMA-kban egy-egy 128 és egy-egy 1 bites található, a változókat kezelő rész egy 512-ről 256 bitre redukálót, a konstansokat kezelő egy 512-ről 64 bitre és egy 64-ről 8 bitre redukálót tartalmaz (utóbbiban azért van kettő, mert a beépített block ramok nem képesek egy lépésben kezelni a busz-szélesség váltását), az eredményeket kezelő pedig egy 256 bitről 512 bitre alakítót tartalmaz. A busz-szélesség változást úgy érik el, hogy a k bit széles kimeneten először a bemeneti n bit széles busz alsó k bitjét, majd a következő kéréskor a felső k bitjét adják ki. Ennek a jelentősége a dolgozat későbbi részében táruul majd fel.

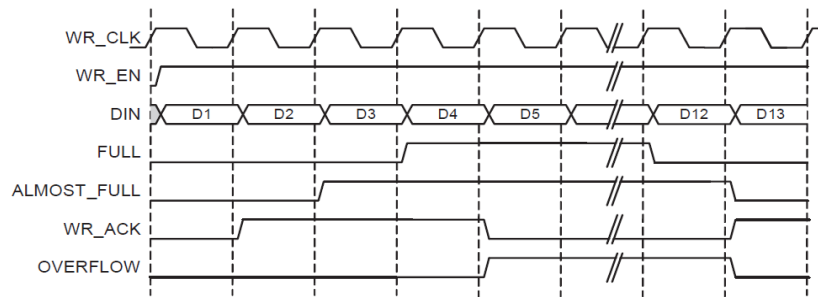


3.4. ábra. FIFO

3.1.1. Beírás

Beíráskor a **Din** buszt, **wr_en**, a **full**, az **almost_full** és a **prog_full** jelek számítanak. Amikor a **full** jel magas jelszinten van, akkor a FIFO-ban nincs több hely,

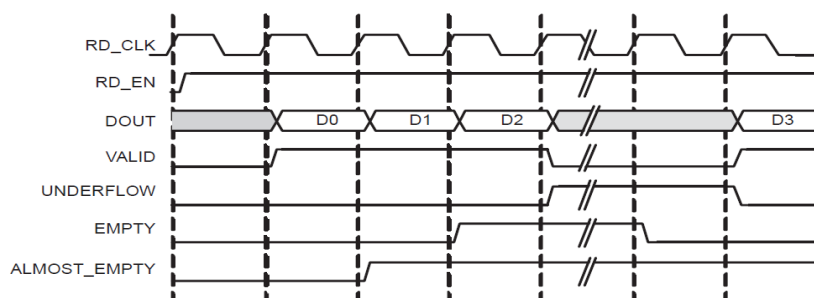
nem tudunk beleírni. Az **almost_full** jel esetén 1 darab hely még van benne, tehát a következő órajelre még ki tudunk adni egy írási parancsot. A **prog_full** jelet akkor kell néznünk, amikor egyszerre több adatot kell beírunk, pl. a memória-interfészen keresztül burst-ökben érkezik az adat és az adatáramlást az elindítás után nem tudjuk közben megállítani. Amennyiben minden feltétel teljesül, a Din buszon a beírandó adat meglétével egy időben kiadva a **wr_en** jelet az új adat bekerül a FIFO-ba.



3.5. ábra. FIFO beírási időzítési diagram

3.1.2. Kiolvasás

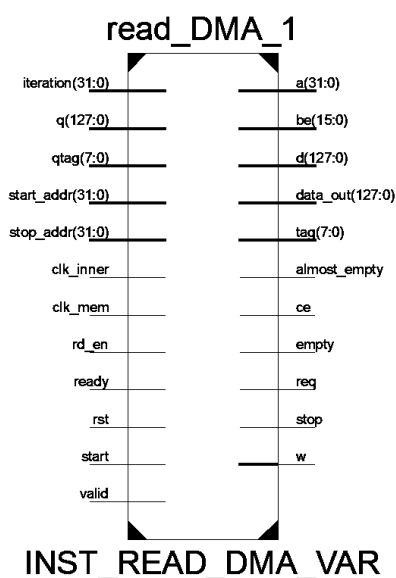
Kiolvasáskor a **Dout** buszt, a **rd_en**, az **empty**, az **almost_empty** és a **prog_empty** jeleket kell figyelembe venni. Amennyiben az **empty** jel magas jelszinten van, úgy a FIFO üres, a kiolvasás hamis eredménnyel járna. Az **almost_empty** jel jelzi, hogy még 1 darab adat van benne, tehát egy olvasási parancsot még ki tudunk adni. A **prog_empty** jelnél az előre beállított számú adat benne van a FIFO-ban, tehát ennyit még ki tudunk olvasni belőle. Ennek akkor van jelentősége, amikor egyszerre sok adatot kell átvinnünk a FIFO-ból pl. a memóriába, és nincs lehetőség az adatfolyam megszakítására. Amennyiben a feltételek teljesülnek, a **rd_en** jel kiadására egy új adat jelenik meg a **Dout** kimeneten.



3.6. ábra. FIFO kiolvasási időzítési diagram

3.2. Olvasó DMA

Az olvasó DMA a szokványos buszokkal kapcsolódik a memória-interfészhez, melyek közül a **d**, a **w** és a **be** jeleket figyelmen kívül hagyjuk, ugyanis ezek a jelek az írási funkcióhoz szükségesek, de ez a DMA csak olvasásra van létrehozva. A **tag_set** generikus attribútummal azt adjuk meg, hogy az adott DMA-nak mi legyen a tag-je; ezen tag segítségével tudjuk meghatározni, hogy a memória-interfész felől érkező adat ennek az eszköznek szól-e vagy egy másiknak. Áramkörileg azt adja meg, hogy a **tag** busz hányadik bitje legyen 1-es, míg a többi 0 (ez a fajta tag-megadás azért jobb, mintha egy bináris számot küldenénk ki a **tag** buszon, mert akkor egy dekódoló-áramkörre lenne szükség ahhoz, hogy megtudjuk, a **qtag** buszon át érkező tag a miénk-e vagy sem). A **start_addr** és az **end_addr** a már korábban említett címbuszok, a **start** és **stop** jelek pedig vezérlőjelek.

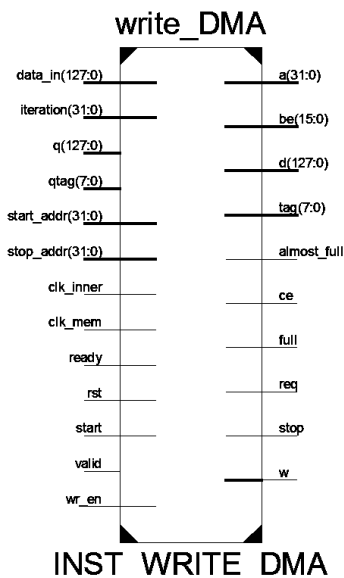


3.7. ábra. Olvasó DMA bemenetek és kimenetek

A **start** jel kiadásakor beállítódnak a start- és stop címek, a rendszer működő állapotba kerül. A **req** jel jelzi a memória-interfész felé, hogy kéri a buszt, amelyre – a memóriabusz állapotától függően valamennyi idő elteltével – a **ready** jelzi, hogy megkaptuk a buszt. A következő órajelre a **req** jelet 0-ra állítjuk, és még egy órajellel később annyi ideig tartjuk 1-ben a **ce** jelet, amennyi ideig a **ready** 1-es marad (tehát 2 órajelnyi késleltetés van a rendszerben – ez a memória-interfésztől függ). Minden **ce** jelnél kiadunk egy címet az a buszon keresztül, jelezve, hogy a megadott címekről adatot szeretnénk olvasni. A címek növeléséről egy számláló gondoskodik, amely a stop címig számol, és ezt

megadott iterációnyi alkalommal ismétli. Természetesen még egy ellenőrzés közbe van iktatva, hogy csak akkor adjunk ki **req**, illetve **ce** jelet (és ezzel együtt címet), amikor biztosan van és lesz hely a FIFO-ban, ne következzen be adatvesztés. Azért, hogy ne csak a FIFO aktuális állapotát, hanem a már kiadott címeket, azaz a memória-interfész felől visszatérő adatok mennyiségét is tudjuk, egy „dummy” FIFO-t alkalmazunk. Ebbe minden egyes kiadott **ce** utasításra egy 1-es kerül, valamint minden egyes, a „valódi” FIFO-ból kiolvasott adatra egy adat szintén kiolvasódik; és mivel a két FIFO mélysége ugyanakkora, így a dummy FIFO-ban lévő elemek száma pontosan megegyezik a normál FIFO-ban lévő elemek számával + a memória-interfész felől biztosan visszatérő adatok számával. A **data_out** buszon keresztül lehet kiolvasni a DMA-ban lévő FIFO-ból az adatokat az **empty**, **almost_empty** és **rd_en** vezérlőjelek segítségével, melyeket a FIFO-ról szóló részben már tárgyaltunk.

3.3. Író DMA



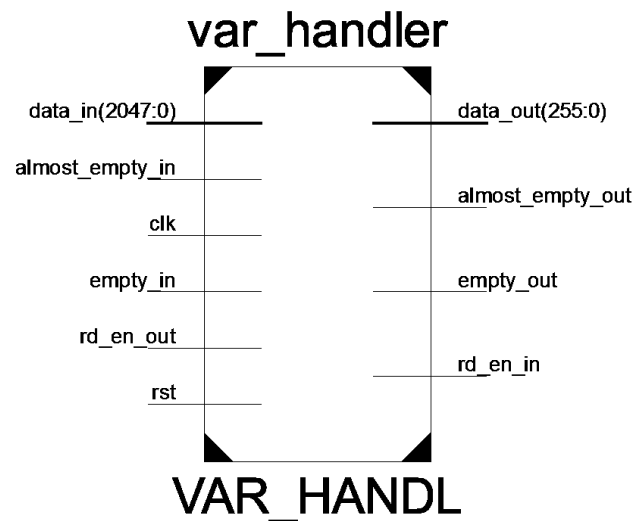
3.8. ábra. Író DMA bemenetek és kimenetek

Az író DMA szintén a szokványos buszokkal kapcsolódik a memória-interfészhez, de a **q** és **qtag** buszokat figyelmen kívül hagyja. A **tag** busznak csak formális jelentősége van, mert ugyan be van állítva a tag a **tag_set** szerint, de nem várunk vissza adatot. Mivel ezzel a DMA-val csak írunk és sohasem olvasunk, így a **w** jelet folyamatosan 1-esben tarthatjuk.

Szintén található benne egy dummy FIFO a valódi FIFO mellett, ugyanazon okból,

mint az olvasó DMA esetében. Amennyiben van adat a FIFO-ban, a **req** jellel kérjük a buszt, és a **ready** jel után két órajel-ciklussal elkezdjük kiküldeni az adatokat a memória-interfész felé úgy, hogy minden **ce** jellel egy időben érvényes adat van a **d** adat- és **a** címbuszokon. A címek kezeléséről szintén egy számláló gondoskodik, csakúgy, mint az olvasó DMA esetében. Új adat a DMA-felé való továbbításában a **data_in**, a **wr_en**, a **full** és az **almost_full** jelek játszanak szerepet. A **data_in** buszon érkezik az új adat a FIFO-ba, a másik három jel pedig a FIFO vezérlésében játszik szerepet.

3.4. A változó-kezelő



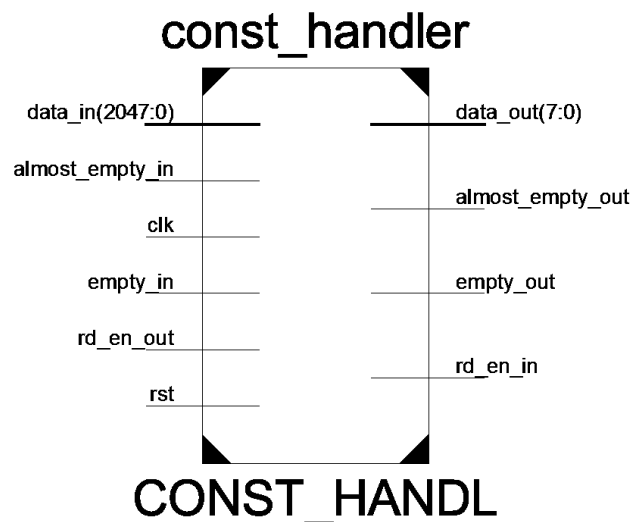
3.9. ábra. A változó-kezelő

Ez a modul fogja össze a változókat olvasó DMA-kból érkező 128 bites buszokat egy nagy 512 bit széles busszá, és alakítja át úgy, hogy egy darab 256 bit széles busz távozzon belőle. Mivel az eredeti memória-tesztelő áramkörben **max_num_bank** darab memória-bankot tud kezelni, de valójában csak 4 darabról van szó, így áramkörileg **max_num_bank*128** bit széles buszt fogad, amelyből csak az alsó 512 bitet veszi figyelembe. A modul egy darab FIFO-t tartalmaz, melyet 512 biten lehet írni és 256 biten lehet olvasni. Ez a FIFO-k leírásánál olvasható elven darabolja fel a bemenő adatokat. A modul úgy lett kialakítva, mintha egy külső FIFO-t vezérelne az **empty_in**, **almost_empty_in** és **rd_en_in** jeleken keresztül; valójában egyszerre 4 darab FIFO-t vezérel, tehát ezeknek a jelek ÉS kapukon keresztül kell bejutniuk ebbe a modulba azért, hogy mind a 4 FIFO-ból egyszerre olvashassa ki a 4x128 bitnyi adatot a saját

FIFO-jába. Természetesen itt azt is figyelembe veszi, hogy a FIFO-ba be lehet-e írni az új adatot. A mobuldból az adatok kiolvasása ugyanolyan vezérlőjelek segítségével történik, mint ahogyan ez a modul kiolvassa a DMA-kból az adatokat (**empty__out**, **almost__empty__out**, **rd_en__out**).

3.5. A konstans-kezelő

Ez a modul hasonlóan működik, mint a változó-kezelő; ugyanúgy 4x1218 bit adatot olvas a konstansokat olvasó DMA-k felől a saját FIFO-jába. A különbség annyi, hogy a Xilinx FPGA-kban található block ramok nem tudják egy lépésben átkonvertálni az 512 bit széles buszt 8 bitessé, ezért két darab FIFO-t kell alkalmazni: az egyik 512-ről 64 bitre, a második 64-ről 8 bitre csökkenti a busz szélességét. A kettő közötti kapcsolathoz azt kell megvizsgálni, hogy az első FIFO-ból lehet-e olvasni legalább 1 adatot és a második FIFO-ban van-e legalább 1 hely; ha mindkét feltétel teljesül, akkor át lehet írni az adatot, ellenben nem történik semmi. A kiolvasás természetesen a második FIFO-ból történik, ugyanolyan vezérlőjelek segítségével, mint ahogyan ez a modul kiolvassa a DMA-kból az adatokat (**empty__out**, **almost__empty__out**, **rd_en__out**).

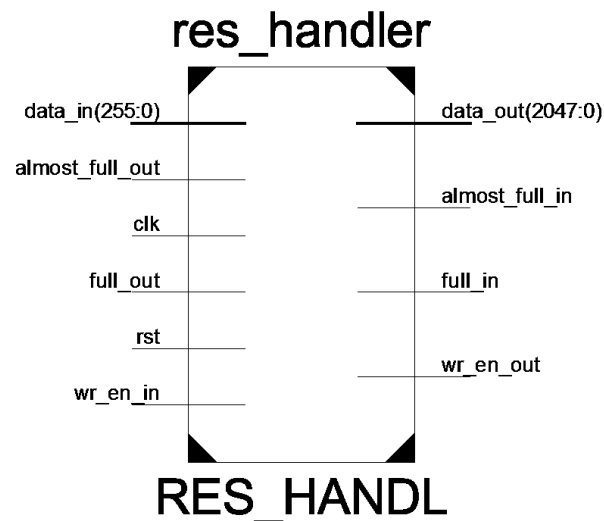


3.10. ábra. A konstans-kezelő

3.6. Az eredmény-kezelő

Ez a modul felelős azért, hogy az aritmetikai egységtől érkező eredmények eljussanak az író DMA-kig. Egy 256 bit széles buszt vár, amelyen keresztül a FIFO-jába íródhatnak

be az adatok a szokásos FIFO-vezérlő jelek segítségével (**full_in**, **almost_full_in**, **wr_en_in**). Szintén egy aszimmetrikus FIFO-ról van szó, a bemenete 256 bites, míg a kimenete 512 bites. Ez utóbbi busz darabolódik fel 4x128 bitre. A FIFO-ból az író DMA-khoz úgy jut el a jel, hogy egy köztes procesz figyel ezen modul **empty_out** és **almost_empty_out** jeleit, és amennyiben lehet olvasni a FIFO-ból (és lehet írni mindegyik író DMA-ba), úgy kiadódik az **rd_en_out** jel és az átírás megtörténik.



3.11. ábra. A eredmény-kezelő

Eredmények

Az előzőekben leírt áramkör teljesíti a vele szemben támasztott követelményeket, rendelkezik azokkal a ki- és bemenetekkel, amelyeken át biztosított a vezérlése és az aritmetikai egység adatellátása. A következő tervezési lépés az áramkör működésének szimulálása, valamint az FPGA-ra való implementálás és tesztelés. Az első eredmények biztatóak, írt és olvasott a fizikai DDR memóriákból és a várt eredményeket hozta. Ezek a tesztek a dolgozat véglegesítésénél is még folyamatban vannak, mert csak a sikeres szimulációk után kapcsolható össze az aritmetikai egységgel, miután megbizonyosodtunk az áramkör helyes és hatékony működéséről.

Összefoglaló

A bevezetésben felvázoltam, hogy milyen feladatot is kell megoldanom és azt milyen eszközökkel szeretném megoldani; feladatom elektromágneses szimulációk gyorsítása, amelyet FPGA-n, VHDL nyelv segítségével fogok megvalósítani. A beszámoló elején bemutattam a VHDL nyelvet, annak történetét és néhány összetevőjét, mint a típusok, vezérlési szerkezetek, objektum-deklarációk, a leírás és az áramkör kapcsolata, valamint adtam ezekre néhány szemléletes példát.

Ezután rátértem az FPGA fejlesztőrendszerek bemutatására, kezdtem az általános FPGA architektúra ismertetésével, majd ezek alapján áttértem a Xilinx Virtex-5-ös chippek specifikus bemutatására. Megvizsgáltam a slice-ok felépítését, az ezekből kialakított CLB-k és az őket összekötő hálózat felépítését; ezek mind szükségesek a lehető legoptimálisabb áramkör leírásához. Innentől következett a rendelkezésre álló Alpha-Data fejlesztőrendszerek és a hozzájuk szükséges SDK bemutatása. A feladat szempontjából kiemelten fontos, hogy a tervezendő áramkör hogyan kap adatokat a memóriából, ezért egy más előre elkészített memória-vezérlő áramköri leírást vettem alapul. Bemutattam hogyan, milyen komponensek segítségével teremtünk összeköttetést a PCI-Express busz, a fizikai memória és a tervezendő áramkör között; ilyen komponensek a PLXDSSM direct-slave átvitel-vezérlő állapotgép, és az arbitrációs egység is.

Következett a probléma matematikai leírása és elemzése. Az elektromágneses szimulációk alapja folytonos fizikai egyenletekben rejlik, amelyeket a digitális áramkörök miatt diszkrétizálni kell. Ezt el kell végezni mind időben, mind térben, hiszen az eseményteret egy térhálóval tudjuk megfelelően leírni, a folyamatot pedig időlépésenkénti számításokkal. A diszkrétizálás után olyan egyenleteket kaptunk, amelyek megfelelően leírják az áramlások mibenlétét, továbbá ezeket már könnyen egy aritmetikai egységbe tudjuk foglalni, VHDL leírással.

Végezetül következett a konkrét feladatom felvázolása: DMA vezérlőket kell írni, amelyek biztosítják a folyamatos adatáramlást az aritmetikai egység és a memória között. Ehhez a számítási cellák miatt két olvasó DMA-ra, és egy író DMA-ra van szükség;

előbbiek az állapotváltozókat és a cellák konstans paramétereit olvassák be az aritmetikai egység felé, utóbbi pedig az eredményeket írja vissza a memóriába. Továbbá, mivel előbbiek memória-bankonként jönnek létre, így változó-, konstans- és eredmény-kezelő áramkörökre is szükség van, amelyek összeköttetést létesítenek az aritmetikai egység és a DMA-k között. A feladat nem triviális, ugyanis sok múlik azon, hogy igazodni kell a memória-vezérlő protokolljához, gyorsnak kell lennie az áramkörnek a késleltetések minimalizálása miatt, továbbá menedzselhetőnek is kell lennie a szimuláció megfelelő irányítása érdekében.

Köszönetnyilvánítás

Köszönettel tartozom témavezetőmnek és konzulensemnek

Dr. Nagy Zoltánnak és Füredi Lászlónak,

akik mindenben hozzájárult ahhoz, hogy ezen dokumentum elkészülhessen, folyamatos felügyelték és segítették a munkámat az elmúlt időszak során.

Irodalomjegyzék

- [1] P. Ashenden and University of Adelaide. Dept. of Computer Science, *The VHDL Cookbook*. Department of Computer Science, University of Adelaide, 1991. [Online]. Available: http://books.google.hu/books?id=__5SMNAAACAAJ
- [2] L. A. et al. (2012, Nov.) History of vhdl | the web's where you study in! [Online]. Available: <http://www.ustudy.in/node/3403>
- [3] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*, 1st ed. Springer Publishing Company, Incorporated, 2007.
- [4] Xilinx Inc. (2012, Nov.) Xilinx. [Online]. Available: <http://www.xilinx.com/>
- [5] —. (2012, Nov.) Xilinx virtex-5. [Online]. Available: <http://www.xilinx.com/support/index.htm#nav=sd-nav-link-19179&tab=tab-sd>
- [6] Alpha-Data Parallel Systems Ltd. (2012, Nov.) Alpha-data. [Online]. Available: <http://www.alpha-data.com/>
- [7] —. (2012, Nov.) Alpha-data adm-xrc-5t2. [Online]. Available: <http://www.alpha-data.com/products.php?product=adm-xrc-5t2>
- [8] —. (2012, Nov.) Alpha-data sdk. [Online]. Available: <http://www.alpha-data.com/applications.php?app=sdk3>
- [9] P. B. Johns and R. L. Beurle, „Numerical solution of two-dimensional scattering problems using a transmission-line matrix,” *Iet Software/iee Proceedings - Software*, 1971.
- [10] P. B. Johns, „A symmetrical condensed node for the tlm method,” *Microwave Theory and Techniques IEEE Transactions on*, vol. 35, no. 4, p. 370–377, 1987.

- [11] V. Trenkic, C. Christopoulos, and T. Benson, „Theory of the symmetrical super-condensed node for the tlm method,” *Microwave Theory and Techniques, IEEE Transactions on*, vol. 43, no. 6, pp. 1342 –1348, jun 1995.
- [12] R. Scaramuzza and A. Lowery, „Hybrid symmetrical condensed node for the tlm method,” *Electronics Letters*, vol. 26, no. 23, pp. 1947 –1949, nov. 1990.
- [13] Xilinx Inc., *LogiCORE IP FIFO Generator v9.3*, Nov. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_3/pg057-fifo-generator.pdf

Ábrák jegyzéke

1.1. VHDL entitás definíció	7
1.2. VHDL funkcionális leírás	7
1.3. 2-bites számláló blokkdiagram	8
1.4. VHDL struktúrális leírás	8
1.5. Latch leírása	11
1.6. D-flipflop leírása	11
1.7. Általános FPGA elrendezés	12
1.8. Konfigurálható logikai blokk	13
1.9. LUT	13
1.10. Slice cluster	14
1.11. Kapcsolási típusok	15
1.12. Kapcsolási típusok	15
1.13. a) hosszú vezetékes, b) sejtes, c) soros FPGA cellaelrendezés	16
1.14. Programozható I/O cella	17
1.15. a) slice cluster, b) slice-ok relatív elrendezése	18
1.16. Virtex 5 slicem	19
1.17. ADM-XRC-5T2-es kártya	20
1.18. ADM-XRC-5T2-es kártya blokk diagramja	21
1.19. A memóriavezérlő blokkdiagramja	22
1.20. PLXDSSM	22
1.21. Az arbitrációs egység blokkdiagramja	23
1.22. A valid és a qtag jelek összefüggése	25
2.1. 3D TLM SCN input voltage polarization	28
2.2. A megvalósított architektúra	29
3.1. A DMA-k elrendezése	33
3.2. A memória felosztása	34

3.3. A teljes áramkör blokkdiagramja	35
3.4. FIFO	36
3.5. FIFO beírási időzítési diagram	37
3.6. FIFO kiolvasási időzítési diagram	37
3.7. Olvasó DMA bemenetek és kimenetek	38
3.8. Író DMA bemenetek és kimenetek	39
3.9. A változó-kezelő	40
3.10. A konstans-kezelő	41
3.11. A eredmény-kezelő	42

Táblázatok jegyzéke

2.1. A szórásmátrix	31
-------------------------------	----

Rövidítések

CLB	Configurable logic block
CPLD	Complex programmable logic device
CPU	Central processing unit
DDR	Double data rate
DMA	Direct memory access
DSP	Digital signal processor
FIFO	First in, first out
FPGA	Field-programmable gate array
GPU	Graphics processing unit
IEEE	Institute of Electrical and Electronics Engineers
JTAG	Joint test action group
LUT	Look-up table
LVDS	Low voltage differential signaling
LVTTL	Low voltage transistor-transistor logic
MB	Megabyte
MUX	Multiplexer
PCI	Peripheral component interconnect
PMC	PCI mezzanine card
PPKE-ITK	Pázmány Péter Katolikus Egyetem Információs Technológiai Kar
SDK	Software development kit
SZTAKI	Számítástechnikai és Automatizálási Kutatóintézet
TLM	Transmission line matrix
VHDL	VHSIC hardware description language
VHSIC	Very-high-speed integrated circuits
SCN	Symmetrical Condensed Node
GSCN	General Symmetrical Condensed Node
HSCN	Hibrid Symmetrical Condensed Node