SZAKDOLGOZAT

Vitéz László

Debrecen
2010

Debreceni Egyetem
Informatikai kar
Informatikai Rendszerek és Hálózatok Tanszék

FPGA-VAL VEZÉRELT ROBOTKAR

Témavezető:
Dr. Végh János
Egyetemi tanár

Készítette:
Vitéz László
Mérnök informatikus

Debrecen
2010

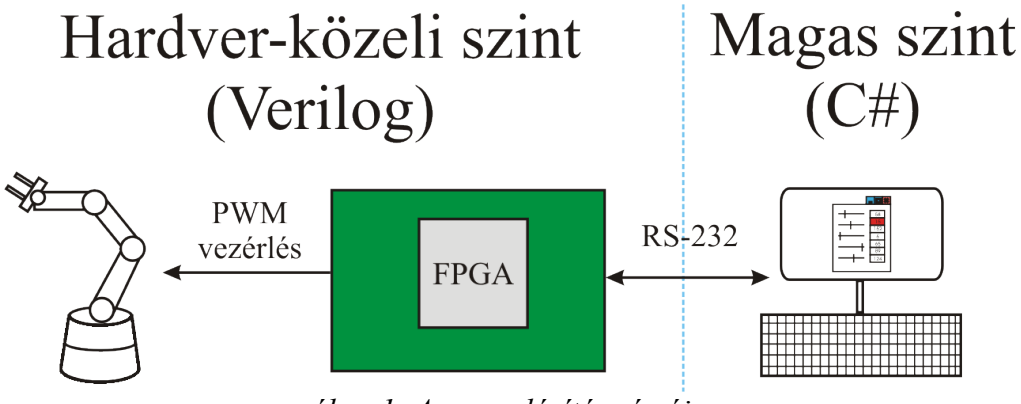
Tartalomjegyzék

Ábrajegyzék

1 Bevezetés

Az FPGA áramkörök sokoldalúságát az informatikusok napjainkban kezdik felismerni. Az áramkör nemcsak masszívan párhuzamosítható, hanem rendkívül rugalmasan alkalmazható, így egy ilyen chipbe az ABS vezérléstől kezdve a kijelzők meghajtására szolgáló áramkörökön át, szinte bármilyen funkciójú egység belefordítható. Az eszköz hardver közeli nyelvekkel programozható hatékonyan, ami azonban teljesen más szemléletet igényel, mint amit a magas szintű nyelveknél megszokhattunk, így a programozók többsége idegenkedik ettől a technológiától.

Munkám során egy szervomotorokkal szerelt robotkar vezérlését valósítottam meg FPGA áramkör segítségével. Célkitűzésem az volt, hogy egy olyan programegységet írjak, mely nagymértékben megkönnyíti és lerövidíti egy robotkar programozását; ne kelljen ahhoz mélyreható elektronikai és informatikai ismeret, hogy valaki egy robotkart mozgásra tudjon bírni, és vele bonyolult műveleteket, mozgássorozatokat hajtasson végre. Így a gyártósorba beállított robot különösebb szaktudás nélkül, könnyen és gyorsan beprogramozható. Az elkészült feladat sematikus váza az 1. ábrán látható.



ábra 1: A megvalósítás sémája

A felhasználó a robotkarral a vezérlőprogramon keresztül tartja a kapcsolatot, mely mindamellett, hogy átlátható és kényelmes felületet biztosít a kar programozására, biztonságossá is teszi azt, hiszen a kezelőnek nincsen beleszólása az alacsonyabb rétegekben folyó műveletekbe, így azok ellenőrzött körülmények között zajlanak.

A robotkar két állapotban lehet. Első amikor pozicionáljuk azt, második amikor automatikusan működik. Ez utóbbi esetben külső vezérlésre nincsen lehetőségünk (csupán a program futását szakíthatjuk meg), annak működése teljesen automatikus. Pozicionálási üzemmódban a vezérlő segítségével egyenként beállíthatjuk a motorok állását, majd ezt lementhetjük az eszköz memóriájába. Így létrehozunk egy lépést, mely után akármennyit beszúrhatunk a memória tárolókapacitásának függvényében. Természetesen több mozgásprogramot is képes az eszköz tárolni, melyek közül bármelyiket kiválaszthatjuk, majd elindíthatjuk annak futását. Ennek hatására a robot egymás után beáll az előre felvett pozíciókra. Az FPGA-n futó szoftver és a vezérlőprogram folyamatosan tájékoztatja a másikat arról, ha valamilyen állapotváltozás történt, így a két egység szinkronban működik egymással. Indításkor a vezérlőprogram pont ezen tény miatt elsőként lekérdezi az FPGA szoftverének állapotát, azaz mikor a felhasználó már irányítani képes a programot, az szinkronba került a hardverrel.

A feladat kidolgozása közben átfogó ismereteket szereztem az impulzus-szélesség modulációs vezérlés területén (a robotkar szervomotorjait ilyen technikával kell irányítani), adatszerkezetek kialakításáról, és azok leképezéséről a nyers memóriába (egy mozgásprogramot az FPGA-val összekapcsolt memóriachip tárol), magas szinten fejlesztett alkalmazásokról (C# nyelven fejlesztett kezelőprogram), valamint az RS–232-es protokollal történő kommunikáció használatáról (a kezelőprogram e protokoll szerint kommunikál az FPGA-val).

2 Leíró rész

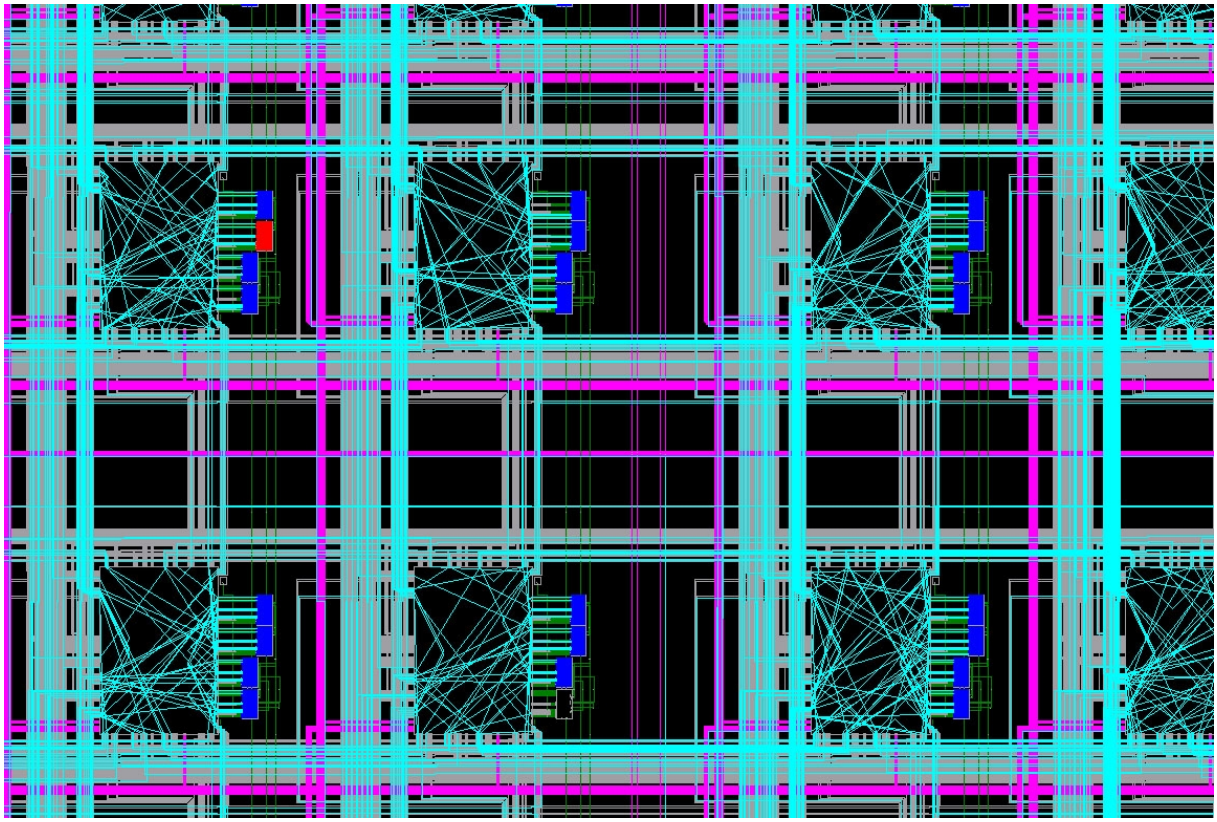
2.1 *Az FPGA (Field Programmable Gate Array)-ról [1],[2]*

Az FPGA áramköröket Ros Freeman a Xilinx egyik társalapítója találta fel 1984-ben. Felépítésüket tekintve három típust különböztetünk meg:

- OTP FPGA: Az ilyen típusú FPGA-kat egyszer lehet programozni (fuse technika), azaz szó szerint a programot egyszer és mindenkorra beleégetjük a chipbe. Előnye a kis méret, és a biztonság (a programunk nagyon kis valószínűséggel sérül), továbbá a kis fogyasztás.
- FLASH alapú FPGA-k: Ezen típusok megőrzik programjukat tápfeszültség nélkül is, azonban újraprogramozni őket csak az eszköz teljes törlése után lehet. Előnye, az újrakonfigurálhatóság, és a biztonság.
- SRAM alapú FPGA-k: Ezek a legelterjedtebb FPGA típusok. Programjukat csak tápfeszültség megléte mellet őrzik meg, így minden használat előtt újra kell őket konfigurálni. Előnyük, hogy mind-közül a leggyorsabbak, és bennük érhető el a legnagyobb komplexitás. További előnyük az olcsó gyártás (CMOS technológiával készülnek, ami ma a legelterjedtebb), valamint az, hogy bármikor – akár menet közben is – átprogramozhatóak. Hátrányuk viszont, hogy az SRAM cellát felépítő tranzisztorok nyugalmi áram-felvétele relatíve nagy, ezáltal az ilyen technológiával készült áramkörök fogyasztása is több a társaiénál (hordozható eszközökben nem célszerű alkalmazni őket). Meg kell azonban említeni, hogy a gyártók igyekeznek kiküszöbölni ezt a hibát, és kifejlesztettek alacsony fogyasztású SRAM FPGA-kat is, melyek akár 50%-kal kevesebbet fogyasztanak elődjeiknél [3]. További problémát okoznak az áramfelvételi csúcsok, melyek akkor következnek be, mikor sok logika akar egyszerre állapotot váltani. Ezen áramkörök kevésbé biztonságosak (állapotuk működés közben megváltozhat), mivel érzékenyebbek a kozmikus sugárzásra (SRAM cellák átbillenhetnek nagy-energiájú sugárzás hatására).

Ma két gyártó uralja a piacot, az egyik a már említett Xilinx, a másik az Altera. A továbbiakban az általam is használt Xilinx gyártmányú Spartan 3E típusú FPGA-król lesz szó.

Az ebbe a családba tartozó FPGA-k mindegyike SRAM alapú. Felépítését tekintve logikai blokkokat (CLB), be és kimeneti illesztőegységeket (IOB), blokk-ramokat (BRAM), szorzó (Multiplier)-, és órajel kezelő egységeket (DCM) tartalmaznak (15-ik ábra). Nem szabad azonban megfelejtkeznünk a huzalozásról sem, mely a szilíciumfelület jelentős részét elfoglalja. Ez drága (az elkészült eszköz költségének nagy százalékát a szilícium felülete teszi ki), azonban szükséges megoldás, ugyanis ezen huzalok segítségével tudjuk kialakítani az összeköttetéseket a blokkok között, biztosítva ezzel a megfelelő flexibilitást és komplexitást. A 2-ik ábra egy méretarányos kép, ahol a téglalapok a hasznos logikát, míg a vonalak a huzalokat szemléltetik. Az ábrát a XILINX FPGA Editora készítette, és az általam írt program reprezentációjának egy részletét tartalmazza. Programozás után az FPGA chipben pontosan ez az architektúra alakul ki.



ábra 2: A hasznos logika és a huzalozás aránya

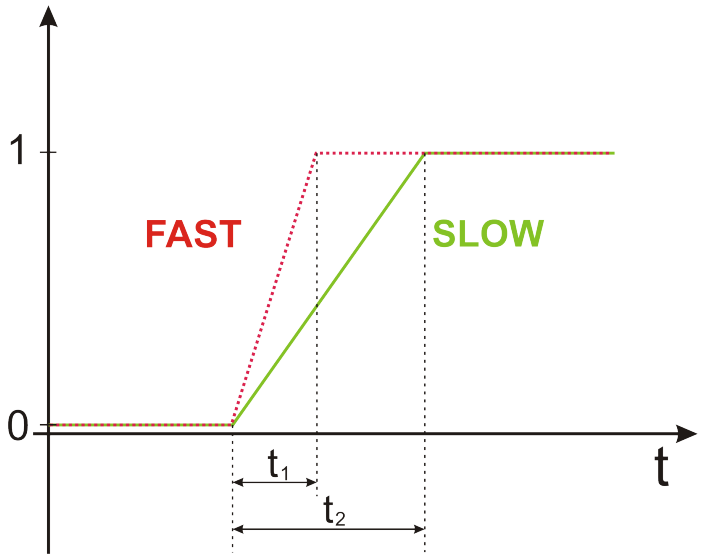
2.1.1 A CLB *(Configurable Logic Block)*-k

A konfigurálható logikai blokkok segítségével tudunk megvalósítani kombinációs és szekvenciális hálózatokat, de használhatjuk őket memóriaként is. A CLB-k képviselik az FPGA-k erőforrását (ez az előző fejezetben említett hasznos logika) így számuk szoros kapcsolatban áll az FPGA árával; minél több van benne, annál drágább az eszköz. Minden CLB négy darab SLICE-ot tartalmaz (16-ik ábra). A 2-ik ábrán a CLB-ket négy darab téglalapból álló csoport, míg a SLICE-okat maguk a téglalapok reprezentálják. Mindegyik SLICE tartalmaz két darab négy bemenetű LUT-ot (Look-Up Table), melyeket igen rugalmasan tudunk felhasználni; kialakíthatunk segítségükkel négybemenetű logikai függvényeket, 16×1 bites ramokat, vagy 16 bites shift regisztereket is, igaz ez utóbbi kettőre, csak M típusú SLICE esetén van lehetőségünk. A LUT-okon kívül a SLICE-ok rendelkeznek két darab regiszterrel, melyeket flip-flopként, vagy latchként használhatunk. Különbség a két eszköz között, hogy a flip-flop-ok rendelkeznek dedikált órajel bemenettel, míg a latchek, azaz tárolók nem. A fent említett erőforrásokon kívül mindegyik CLB rendelkezik két darab multiplexerrel, valamint átviteli logikával, ami még rugalmasabbá teszi felhasználhatóságukat. Ahogy fentebb utaltam rá, a SLICE-oknak két típusa van (L és M), melyek közül csak az M típusúban lévő LUT-ok használhatóak memóriaként, vagy shift regiszterként. Minden CLB-ben 2 darab M (bal oldali pár), és két darab L típusú foglal helyet (jobb oldali pár). Habár az M típusú SLICE-ok jóval több szolgáltatást nyújtanak, az L típusúak használata is indokolt, ugyanis kisebb méretük által csökken a CLB-k mérete, így több CLB integrálható ugyanakkora felületre.

2.1.2 AZ IOB *(Input/Output Blok)*

Az IOB-k teremtik meg a kapcsolatot a külvilág, és az FPGA belseje között, ennek megfelelően a szilícium lapka szélén helyezkednek el (15-ik ábra). Az IOB-k és az FPGA „lábai” között huzalozott kapcsolat áll fenn. Egy-egy láb funkcionálhat be-, ki-, illetve be- és kimenetként. A SPARTAN 3E típusú FPGA-k esetében vannak csak és kizárólag bemenetként használható lábak, azonban ezek száma az összes IOB számának maximum 25%-át teheti ki ([2] 10–ik oldal). Ebben az esetben a lábhoz tartozó IOB csak bemeneti egységgel rendelkezik, ellentétben azokkal, amelyek a fent említett három üzemmód mindegyikében

képesek működni. Az utóbbi esetben, a lábhoz tartozó IOB három részegységre bontható (bemeneti, kimeneti, be-, és kimeneti). Amennyiben az adott láb bemenetként funkcionál, a jel és 17-ik ábrának megfelelően programozható késleltetőkön és akár tárolókon keresztül is haladhat a logika felé (I, IQ1, IQ2). Az általam használt FPGA bemenetei 1.5, 1.8, 2.5, 3.3V kompatibilisek, ami azt jelenti, hogy ha ennél kisebb jellel akarunk dolgozni, azt az FPGA nem képes kezelni, ha pedig nagyobbal, a bemenetet védő ESD jelű diódák kinyitnak, védve ezzel az eszköz belsejét. Ezek a diódák sem bírnak el végtelen ideig nagy átfolyó áramot, így a nem megfelelő feszültség károsíthatja, vagy tönkre is teheti az áramkört. A diódák védik az FPGA belsejét a statikus feszültségektől is, azaz az eszközhöz kézzel is hozzáérhetünk, hiába nem vagyunk az eszközzel azonos potenciálszinten. A kimeneti és háromállapotú egység feladata jelet szállítani a logikától a csatlakozókhoz. Mindkét jelútba beépítésre került két-két tároló, valamint egy-egy DDR multiplexer. Ez utóbbiak azért nagyon hasznosak, mert a két tárolót inverz órajellel meghajtva, továbbá a multiplexerrel (DDR MUX) a megfelelőt a kimenetre választva dupla akkora sebesség érhető el, mint egyébként. A két bemeneti regiszter (IFF1, IFF2) használatával ez a funkció az inputokon is kiaknázható. Itt természetesen nincsen szükség multiplexerre, azonban két adatút az ára a dupla sávszélességnek(IQ1,IQ2). Mindegyik pin-hez (egy lába az FPGA-nak) tartozik egy tartó áramkör, ami biztosítja egy láb nagy impedanciás állapotát mikor az bemenetként funkcionál, illetve kimenetként megőrzi annak értékét amennyiben nincs rajta vezérlés. Az IOB-kben helyet kapott még egy le-, és egy felhúzó ellenállás is (pull-down, pull up), melyek bekapcsolása felülbírálja a tartó áramkör logikai értékét. Akárcsak a bemenetként, a kimenetként használt lábak is csak meghatározott feszültségszinteken tudnak működni, melyek definiálják a terhelő áram korlátját is (1. táblázat). Ezenkívül azt is beállíthatjuk, hogy a kimenet milyen gyorsan vegye fel az új állapotát. Ez az érték a slew-rate-nek nevezzük, ami lassúra, vagy pedig gyorsra állítható (3-ik ábra). Az IO lábakhoz tartozó alapértelmezett paraméterek: LVCMOS24, 12mA, slow slew-rate.



ábra 3: lassú és gyors slew rate

I/O sztenderd	Kimeneti áram (mA)					
	2	4	6	8	12	16
LVTTL	+	+	+	+	+	+
LVCMOS33	+	+	+	+	+	+
LVCMOS25	+	+	+	+	+	-
LVCOMS18	+	+	+	+	-	-
LVCMOS15	+	+	+	-	-	-
LVCMOS12	+	-	-	-	-	-

táblázat 1: A terhelő áram körlátai. Forrás: [2], 8-ik tábla

2.1.3 BlokkRAM

A Spartan 3E típusú FPGA-k 4-36 darab, egyenként 18kbites, dual portos dedikált blokk RAM-ot tartalmaznak. Az általam használt FPGA két oszlopba szervezve 12 darab ilyen blokkot tartalmaz (összesen 221.184 bit memória). Ezen egységeket használhatjuk kezdőértékek, vagy kimeneti regiszterek alapértelmezett értékeinek tárolására, továbbá függvénytáblázatként (pl. sinus, cosinus értékeinek letárolása), valamint beágyazott mikrovezérlő utasításainak, programjának tárolására, stb. A blokkramok elhelyezkedése a 18-ik ábrán látható. Az egységek egyik oldala CLB-kel, másik pedig a szórzóegységekkel

határolt. Ez utóbbiak 18×18 bites szorzóegységek, melyek felső 16 bitje közös a hozzájuk tartozó blokkrammal.

A blokk ramok szinkron[1], dual portos ramok, melyek adatmérete rugalmasan alakítható:

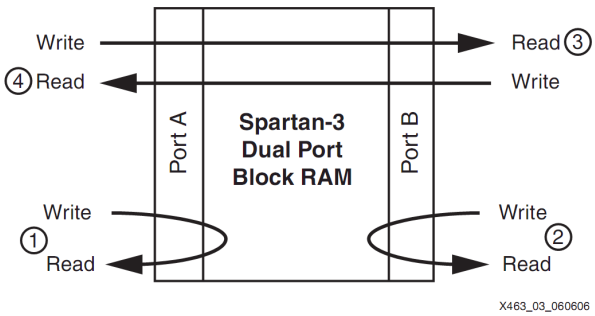
Adatút szélessége paritásbittel (bit)	Adat-busz szélessége (bit)	Paritásbit címbusz szélessége	Adat címzése	Memória kapcacitása
1	1	-	[13:0]	16.348
2	2	-	[12:0]	16.348
4	4	-	[11:0]	16.348
9	8	[0:0]	[10:0]	18.432
18	16	[1:0]	[9:0]	18.432
36	32	[3:0]	[8:0]	18.432

táblázat 2: Blokkramban kialakítható struktúrák. Forrás:[2], 22-ik tábla

Látható, hogy 1, 2, illetve 4 bites kialakítás esetén nincsen lehetőség paritásbit tárolására. Ez csak 8, 16, illetve 32 bites szervezés esetén lehetséges. A felhasználható memóriaterület ilyenkor is 16.348 bájt, azonban a paritásbitek tárolása miatt az összesen felhasznált tárterület 18 kbájt. Ezekből az adatokból egyértelműen következik, hogy a paritásbiteket az eszköz dedikált helyen tárolja.

A dual portos működés két teljesen független (port A, port B) csatlakozást jelent ugyanahhoz a tárterülethez. A dual portos memória adatútjai a következőek lehetnek:

1. Írom, majd olvasom az A portot
2. Írom majd olvasom B portot
3. Beírok A portba, majd az adatot kiolvasom B-ből
4. Beírok B portból, majd kiolvasom A-ból



ábra 4: A dual portos működés vázlata

A dual portos működés kiváló lehetőséget nyújt különböző órajelen működő rendszerek szinkronizációjához, illetve adatcseréjéhez, mivel A, és B port teljesen független órajelbemenettel rendelkezik. Használatuknál azonban figyelmesnek kell lenni, ugyanis A és B között nincsen prioritás, így ha mindkét oldal felől ugyanazon cellát akarják írni, a cella

1: A szinkron működés annyit tesz, hogy az adatokat olvasni és írni az órajellel szinkronban tudom

tartalma bizonytalan lesz. Hasonló a helyzet abban az esetben is, mikor A és B órajele aszinkron. Ha az egyik oldalról írunk, a másikról pedig olvasunk akkor megjósolhatatlan az olvasás eredménye, mivel előfordulhat olyan eset, mikor a beírt adat még nem stabil az olvasás ideje alatt [1].

A és B port között nem csak az órajelben, hanem az adatvonalak számában is lehet különbség, ami azonban nem okoz problémát, mivel belső logika gondoskodik a megfeleltetésen. Így például ha az A port adatszervezése 32 bites szó alapú a B porté pedig byte, akkor a szó első nyolc bitjét és a hozzá tartozó paritásbitet a B porton a 0-s címről, a második bájtot és paritásbitjét az 1-es címről, és így tovább érem el. Ezen tulajdonság segítségével könnyen reprezentálható párhuzamos-soros, és mivel a funkció visszafelé is működik, soros-párhuzamos átalakító.

2.1.4 Szorzó egység

A dedikált szorzóegységek száma megegyezik a blokkramok számával, így 4-36-ig terjed. Egy szorzóegység két darab 18 bites kettes komplemens ábrázolású számot (-131.072...+131.071) vár a bemeneteire, és kimenetén szintén kettes komplemens ábrázolásban 36 bites eredményt (-17.179.869.184...+17.179.869.183) szolgáltat. Látható, hogy a kimeneti szám tartománya 35 bites, ennek oka, hogy a 35-ik bit redundáns, ugyanis a két bemenet 1-1 bitje előjelbit így valójában az eredmény 34 adat + 1 előjel biten is elfér. A kettes komplemens ábrázolás azért szerencsés, mert velük a műveletek előjelesen végezhetőek, továbbá gond nélkül összeszorozható két nem kettes komplemensben ábrázolt 17 bites szám is, úgy, hogy a 18-ik bit helyére 0-t írunk. A kettes komplemens ábrázolásáról bővebb információt a 6.1 pont tartalmaz. Az egységet használhatjuk 18 bitnél nagyobb számok összeszorzására is, ilyenkor három alternatíva közül válogathatunk. Első megoldás, hogy slice-okból kialakított szorzóegységekkel bővítjük ki a meglévőt, ami azonban rendkívül erőforrás-igényes. Második megoldás, hogy kaszkádosítjuk a dedikált egységeket, a harmadik pedig, hogy időosztásban használjuk azokat. Legutóbbi megoldással erőforrást spórolunk, viszont időben lassabban készül el az eredmény.

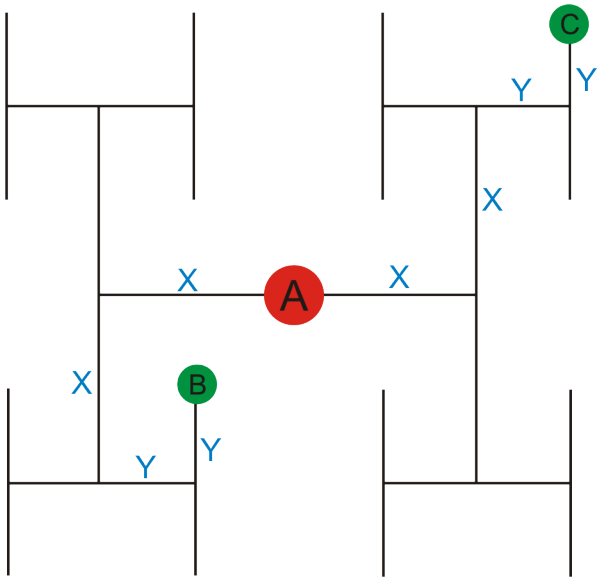
A szorzóegységek mindkét bemenete (A, B), valamint a kimenet (P) rendelkezik egy 18 bites be- és kikapcsolható regiszterrel. Ezek közös órajelbemenettel, azonban külön szinkron reset és engedélyező bemenettel rendelkeznek. Ezekben a regiszterekben tárolhatók szorzóegyüttható (pl. FIR szűrő), korábbi eredmények, stb. A kaszkádosításhoz a szorzóegységek B bemenete rendelkezik egy további be (BCIN), és kimenettel (BCOUT) (20-ik ábra). A BCIN bemenet párhuzamosított a B bemenettel, kaszkádosításnál ezt, míg egyéb esetben B bemenetet használjuk. BCOUT szolgáltatja minden időpillanatban a szorzó második bemenetén jelen lévő számot. Kaszkádosításnál a szomszédos egységeket tudjuk ezekkel a vonalakkal (BCIN, BCOUT) összekapcsolni (19-ik ábra). Ahogy az már a blokk-ramoknál említésre került, a szorzóegységek ezek mellett helyezkednek el. A blokk-ram A portjának a felső 16 bitje közös a szorzóegység A portjának 16 bitjével. Ez hasonlóan a B portnál. Habár ez előnyöket is kovácsol, hátránya, hogy a blokk-ramok 36×512 bites módban történő használatakor (2-ik táblázat utolsó sora) nem tudjuk elérni a mellette lévő szorzóegységet.

2.1.5 Órajel-hálózat és DCM

A Spartan típusú FPGA-k rendelkeznek kizárólag az órajel vezetésére kialakított huzalozással. Ezen huzalozás szempontjából a felület negyedekre osztható. Mindegyik negyed rendelkezik 8 külön órajelvonallal (A...H), miáltal 8 különböző órajellel tudjuk megtáplálni a negyedben elhelyezkedő szinkron működésű elemeket (IOB, CLB, DCM, BRAM, szorzóegység). Ezen vezetékek olyan sémájúak, hogy minél kisebb legyen a kapacitásuk, ami által nagyfrekvenciájú órajelek is használhatóak, valamint a s**k**ew rate értéke is minimális legyen, azaz, a lapka fizikailag két egymástól távol eső pontján az órajel azonos időpillanatban változzon. Az alacsony skew rate azért is fontos, mert így a set up[2], és a hold time[3] lerövidíthető, ami által gyorsabb áramkörök készíthetők. Ezen feltételeknek eleget tevő órajelvezetés például a H sémás órajelvezetés (5-ik ábra).

2: az az idő, amely alatt az adatnak nem szabad változnia az órajel esemény előtt

3: az az idő, mely alatt az adatnak nem szabad változni az órajel esemény után



ábra 5: "H" sémájú órajelellátás

Ha "A" pontban tápláljuk be az órajelet, akkor "C" és "B" egység teljesen szinkronban lesz, hiszen az órajel mindkét irányban 2X+2Y utat tesz meg. Ez a módszer persze a gyakorlatban nem alkalmazható gazdaságosan, mivel nagyon sok kihasználatlan hely maradna a szilíciumlapkán. Ehelyett a 21-ik ábrán látható módon alakítják ki a hálózatot, melynek megvan az az előnye, hogy rendkívül takarékosan használja ki a rendelkezésre álló felületet, és majdnem H struktúrával bír. A szinkron egységek azonban igénylik, hogy a skew rate minél kisebb legyen, többek között ezt a problémát is hivatott megoldani a DCM, azaz a Digital Clock Manager. Ezt úgy képes megtenni, hogy (ahogy az a 21 ábrán is látszik) méri a visszavezettet órajel, valamint a bejövő órajel közötti fáziskülönbséget, majd ennek megfelelően késlelteti azt. Ezen kívül a bejövő órajel frekvenciáját képes kétszerezni, illetve leosztani (1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6.0, 6.5, 7.0, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, és 16-od arányban). További funkciója még, hogy képes eltolni a bemenő órajelet 0°,90°,180°,270°-kal, valamint a bemeneti órajel felszorzott változatát 180°-kal.

2.1.6 Kapcsolómátrix és huzalozás

Ahogyan az már említésre került a 2.1 fejezetben, a huzalozás is fontos része az FPGA-knak. Ezen huzalozás teremti meg a kapcsolatot a CLB-k IOB-k és DCM-ek között. Mivel a huzalozás fix, a rugalmasság érdekében nem mindegy, hogy melyik vonalat melyik CLB-hez és hogyan kapcsoljuk. Ezen funkciót látja el minden CLB, IOB, és DCM előtt

elhelyezkedő kapcsolómátrix. A 2-ik ábrán ez a rész is jól látható a CLB-k előtt. (sok egymást keresztező vezeték egy téglalapban). Természetesen ugyanilyen kapcsolómátrixok (egyenként 4 darab) kapcsolják rá a blokkram, és szorzóegységek megfelelő vonalait a huzalokra. 4 fajta huzalozással rendelkezik az FPGA:

- Hosszú vonalak: Ezek a vezetékek végigfutnak az egész felületen vízszintes és hosszanti irányban, és csatlakoznak minden hatodik CLB-hez. Minden csatlakozási pontnál a 24 hosszú vonalból négy csatlakozik a kapcsolómátrixhoz. A hosszú vonalak kis kapacitású, jó minőségű kapcsolatot biztosítanak az egységek között, így alkalmasak nagy frekvenciájú jeltovábbításra, vagy akár órajelek továbbítására is, amennyiben az már nem fér el a dedikált hálózatán. (Ez fordítva nem igaz, az órajelhálózat csak órajelek továbbítására használható, adatéra nem.)
- Hatos vonalak: Ezek a vonalak horizontális és vertikális irányban kötik össze a szomszédos hatos csoportokat. Egy hatos csoport első, és harmadik eleme csatlakozik az őt követő csoport első elemével. Bármely csatlakozási pontnál összesen 32 ilyen vonalat használhatunk, melyből mindegyik nyolc vezetékkel rendelkezik.
- A dupla vonalak mind a négy irányban összekötik a csatlakozó egységet a vele szomszédos és az azt követő egységgel. A busz szélessége 8 bit.
- Direkt kapcsolódó vonalak: Ezek a vonalak kötik össze az egységet a vele szomszédos (vízszintesen, függőlegesen, és kereszt irányokban) egységekkel.

Mindezen vonalak mellett az eszköz rendelkezik még további két globális vezetékkel. Az egyik a GSR globális SET/RESET vezeték mely segítségével az összes flip-flopot kezdőállapotába hozhatjuk, a másik pedig a GTS vezeték mely segítségével az összes IO láb nagy-impedanciás állapotba állítható. Mindkét vezeték aszinkron működik, azaz abban a pillanatba kifejtik hatásukat mikor azok magas értékre váltanak; nem várnak órajeleseményre.

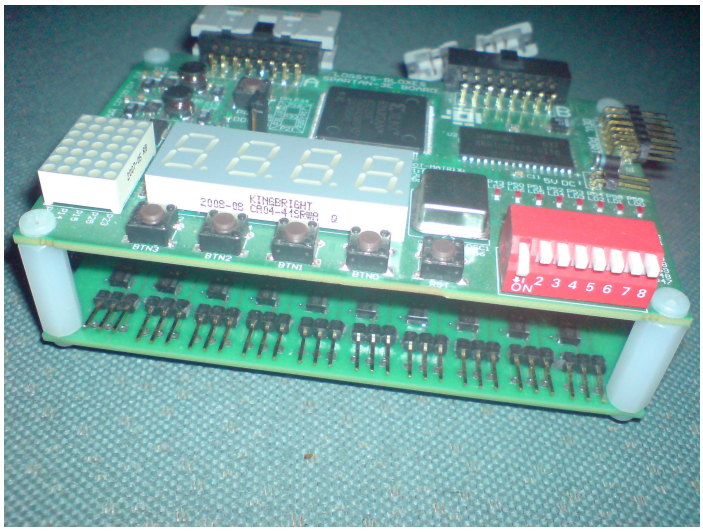
2.2 *Hardware elemek*

2.2.1 Az FPGA-t tartalmazó kártya

Az általam használt LOGSYS fejlesztői kártyát a Budapesti Műszaki Egyetemen fejlesztik, egy SPARTAN 3E 250 típusú FPGA-t tartalmaz. A panelen helyet kapott 5 darab mikrokapcsoló, egy 8 bites DIP kapcsoló, 4 darab hétszegmenses LED kijelző, egy 7×5-ös LED mátrix, 10 darab LED dióda, egy 128 kbájtos aszinkron SRAM, egy flash memória, és két darab általános célú 16 pólusú csatlakozó. A panelen található egységekből a LED mátrixot, és a hétszegmenses kijelzőt használom különböző információk megjelenítésére, a mikrokapcsolókat a megjelenítési üzemmódok közötti váltásra, és az aszinkron SRAM-ot a programok tárolására. A két általános célú csatlakozóval kapcsolom össze a panelt a PC-vel, valamint a robotkarral.

2.2.2 Robotkar

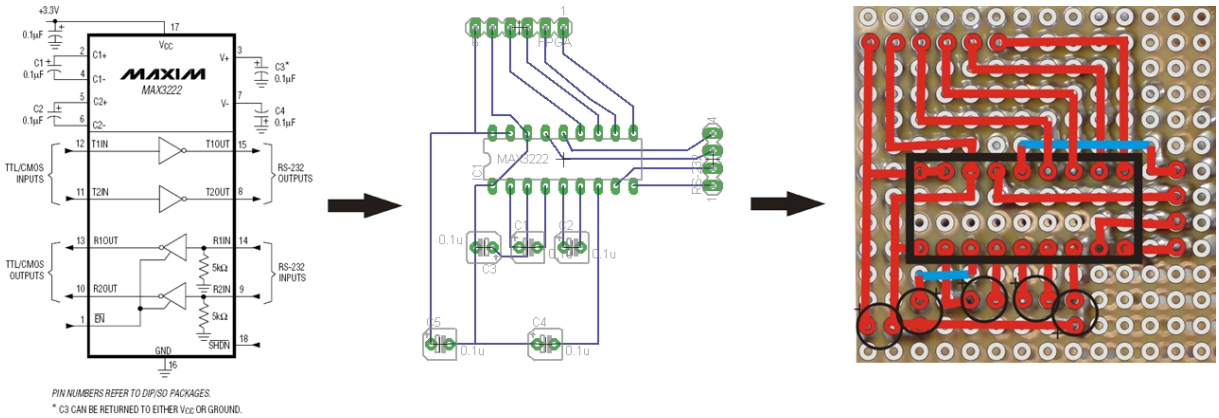
A robotkar hat darab szervomotorral szerelt, hobbi célokra alkalmas eszköz, melyet a NATIONAL INSTRUMENTS biztosított számomra. Eredetileg a motorok FUTABA gyártmányúak voltak, azonban ezek nyomatéka nem volt elegendő megmozdítani, sőt néha megtartani sem a robotot, így mindegyiket kicserélték TOWER PRO gyártmányúakra, amik már kellőképpen erősek voltak, és vezérlésük sem tért el az eredeti motorokétól. Az FPGA, és a robotkar közé biztonsági okokból beépítésre került egy plusz panel, ami 8 darab SHARP gyártmányú optocsatolót tartalmaz. Ezen egységek nagyon fontosak, ugyanis galvanikusan elválasztják a motorokat a LOGSYS paneltől, így a motorok meghibásodása, rövidzárlata esetén megóvják a náluknál jóval drágább FPGA kimeneteit. A LOGSYS panellel összeszerelt leválasztópanel a 6-ik ábrán látható. Az optocsatoló-áramkör egyik mellékhatása, hogy invertálja a bemenetére kerülő logikai jeleket, így ezt a programkódban kompenzálni kell.



ábra 6: A Logsys-panel, és alatta az optocsatoló egység

2.2.3 RS232-LVTTL konverter

Ahogyan azt már említettem az FPGA RS232-es protokollal kommunikál a vezérlőprogramjával. A protokoll azonban olyan feszültségszinteket rendel a logikai értékekhez, melyet az FPGA nem képes fogadni, így azt át kell alakítani úgy, hogy az feldolgozható legyen. Ezt a feszültségkonverziót végzi el az általam épített MAX3222 IC-vel szerelt segédpanel, melyet az IC használati útmutatójában található referencia kapcsolás alapján terveztem meg. Elsőként a kapcsolási rajzot bevittem az EAGLE nevű nyáktervező programba, majd generáltattam azzal egy egyréteges nyáktervet. Sajnos nem volt lehetőségem ilyen panelt legyártani, ezért helyette a vonalvezetést áttervezve egy tesztpanelen alakítottam ki a megfelelő áramköri rajzolatot (7-ik ábra).



ábra 7: Az Illesztőáramkör megvalósulásának fázisai

3 Megvalósítás

3.1 *Hardver-leíró nyelven implementált funkciók*

Az FPGA-ra fejlesztett kód VERILOG nyelven, modulokba szervezve került implementálásra. Fejlesztői környezetként a XILINX ISE WEBPACK 10.1 verziójú szoftverét használtam. A modul egy hardver blokkot reprezentál, jól definiált be- és kimenetekkel, valamint működéssel rendelkezik [5]. A modul a Verilogban az újrafelhasználás eszköze, egy megírt modult többször is példányosítani tudok. Használatuk egyszerűbbé, és könnyen olvashatóvá teszi a kódot, azonban nem szabad megfelejtkezni arról sem, hogy ahányszor csak példányosítok egy modult, a mögötte lévő áramkör annyiszor szintetizálódik, azaz fizikailag reprezentálódik az FPGA belsejében. Szakdolgozatomban az összes szekvenciális hálózatot tartalmazó modul szinkron működésű, melyet a kártyán lévő 16MHz-es rezgőkvarc lát el órajellel.

3.1.1 A kommunikációt megvalósító modul (rs232.v)

3.1.1.1 *Elméleti háttér*

A kommunikáció RS232-es szabvány szerint zajlik. Azért választottam ezt a protokollt, mert megvalósítása egyszerű, specifikációja pedig kielégíti az általam támasztott követelményeket (viszonylag érzéketlen az átviteli csatornára, megfelelő adatátviteli sebesség), továbbá a magas szintű nyelvek is támogatják, ugyanis a PC soros portja is ezt a szabványt használja.

3.1.1.1.1 Fizikai specifikáció[7]:

- A minimális oda-vissza kommunikációhoz három vezeték (GND, Rx, Tx) szükséges.
- Feszültségtartomány: -15V...+15V
- Logikai alacsony szint: +3V...+15V
- Logikai magas szint: -15V...-3V

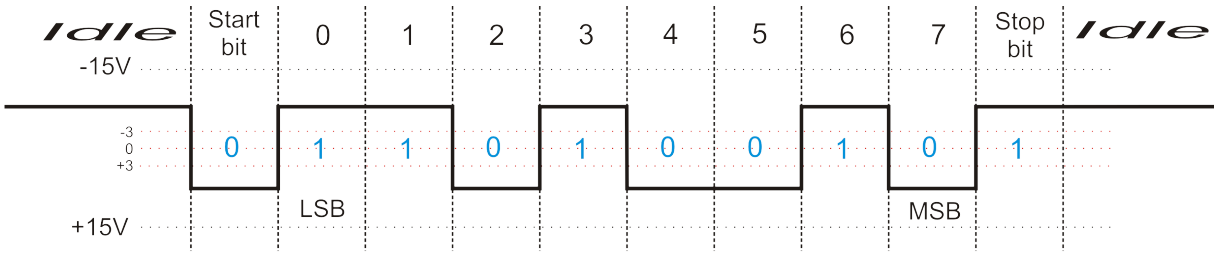
- Tiltott sáv: -3V...+3V
- Maximum sebesség: 115.200 bit/s (általam is használt)
- Maximum áthidalható távolság sebességfüggő; maximális sebességnél legfeljebb 4 m [8].

3.1.1.1.2 A kommunikáció menete[6]:

A kommunikáció aszinkron, keretekbe foglalt (start és stop bit) bájt szervezésű.

- Nyugalmi állapotban az adattovábbító vonalak logikai alacsony szinten állnak. Ez a +3V...+15V közötti feszültségnek felel meg
- Adatátvitel kezdeményezésekor az adó a Tx vonalát egy bit szélességig logikai 1-re húzza (start bit elküldése)
- Ezután nyolc adatbit következik. A bitáram LSB-MSB irányú, azaz a legalacsonyabb helyiértékű bit felől kezdjük a küldést.
- Majd opcionálisan paritásbit (ezt a lehetőségét a soros kommunikációnak nem használtam ki)
- Végül pedig egy bit szélességű logikai 0, azaz a stop-bit. Ezt a bitet természetesen azonnal követheti egy újabb csomag startbitje.

Látható, hogy egy bájt legkevesebb 10 bit átvitelével továbbítható. Az alábbi példa a 75_{10} szám átvitelét szemlélteti.



ábra 8: Péda egy bájt átvitelére RS232-es protokollal

Mivel egy bájt átvitelével igen csekély adat továbbítható, az információcsere a PC és az FPGA között csomagokba szervezve zajlik. Egy csomag, logikailag összetartozó bájt-csoport,

mely egy 8 bites parancsrésszel kezdődik. Ez a parancsrész pontosan definiálja azt, hogy az őt opcionálisan követő adatrész milyen hosszúságú, és annak milyen a struktúrája. Például: 8, 9, 10, 11-as tábla.

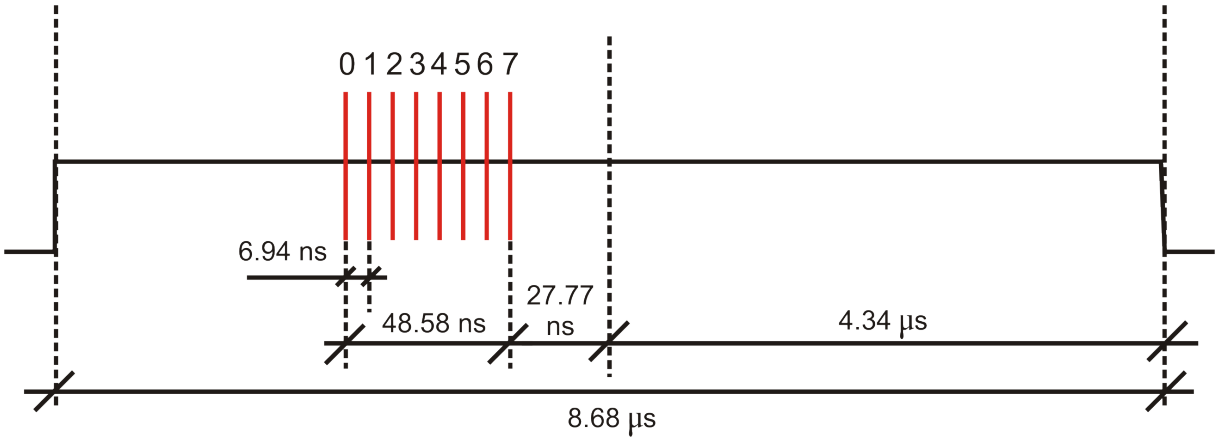
3.1.1.2 *Gyakorlati megvalósítás*

A kommunikációt végző modul mindaddig nyugalmi állapotban van, amíg az Rx vonalán startbitet nem fogad, esetleg felkérést nem kap arra, hogy a nyolc bites bemeneti buszán található adatot küldje el. Itt említeném meg az FPGA áramkörök egy hatalmas előnyét, a masszív párhuzamosíthatóságot ugyanis a fogadás és a küldés egymással párhuzamosan is képes működni még akkor is, ha ezek egymáshoz képest teljesen aszinkron zajlanak. Ez úgy valósítható meg, hogy a fogadással, és a küldéssel teljesen külön kódrész, így szintetizálás után külön-külön hardver foglalkozik.

Ahhoz, hogy a soros vonalról érkező adatokat kezelni tudjuk le kell osztanunk a kártya 16 MHz-es órajelét, hiszen ez sokkal gyorsabb, mint a csatorna átviteli sebessége. Mihelyst valamilyen esemény történik (startbitet érzékelünk, vagy küldési kérés érkezik) a megfelelő órajelosztó rész megkezdi működését. Mivel $\frac{16.000.000}{115.200} = 138{,}88\dot{8} \notin \mathbb{Z}$ így pontosan nem tudunk szinkronba lenni a kommunikációs csatornával, azonban ez nem okoz problémát a kommunikáció asszinkron mivoltjából. Egy adatbitre 138 órajelütés jut, azonban a fenti képlet hányadosa közelebb áll a 139-hez, így az órajelosztáshoz is e konstans használata indokolt. Jó közelítéssel minden egyes adatbit közepéből mintát tudunk venni a 69-ik órajelütésnél. A csúszás minimalizálása érdekében a startbit alatt nem 139, hanem 138 órajelet várunk, ezáltal a csúszás a bit valódi közepe, és a minta-vett közép között 10^{-9}-en nagyságrendű, mely 10, sőt 100-ad része az egyéb esetekben kialakuló csúszásoknak. A mintavételt és az elcsúszást a 9-ik, és 22-ik ábra szemlélteti. Felhívnám a figyelmet, hogy a teljes bit hossza μs, az eltérés az elméleti középtől pedig ns nagyságrendű. A maximális eltérés 0,88%-a a teljes bithossznak, így ez az érték megfelelőnek tekinthető. A piros függőleges vonalak jelzik a soros vonalon egymást követő nyolc bit mintavételét. Annak érdekében, hogy folytonos bájt-továbbításkor minden bájt olvasásakor ez a késleltetés érvényesüljön, a stopbit fogadása alatt a modul alaphelyzetbe áll, azaz újra startbitre várva az egész folyamat kezdődik elölről. Ha ezt nem

így tenné, hosszú, folyamatos kommunikációnál a csúszások annyira felhalmozódnának, hogy nem a megfelelő helyről venné a mintát.

Egy adatbájt küldése hasonlóan zajlik. Miután elkezdtük küldeni a startbitet a program lementi egy regiszterbe az adatvonalán található értéket, így a küldendő adat az lesz ami a startbit pillanatában rendelkezésre állt a bemeneten, azaz továbbítás alatt változhat a bemenet. A modul rendelkezik két 8-8 bites busszal, egyik a küldendő, másikon pedig a fogadott adat továbbítására szolgál. Ezenkívül helyet kapott egy olyan vezeték, melyen keresztül a modul jelezheti, hogy megtörtént egy bájt fogadása, azaz a kimenő adatbuszán lévő információ helyes, továbbá két olyan vezeték, mely a küldést koordinálja. Egyikük logikai magas szintje jelzi, hogy adat továbbítása van folyamatban, másikukkal pedig kérvényezni tudjuk, hogy a modul továbbítsa a bemenő adatbuszán található adatot. Természetesen ez a modul kapcsolódik közvetlenül a B bővítősínre, ami pedig összeköttetésben áll a feszültségkonverteren keresztül a PC soros portjával. A küldő és fogadó vezetékeken kívül bekötésre került egy adat-áramlást szabályozó vezeték is. Ha ez a vezeték logikailag alacsony szintű, akkor az hardveresen megtiltja a PC-nek, hogy további adatot küldjön a soros porton. Ennek jelentősége később, az inicializálásnál és memóriakezelésnél lesz.



ábra 9: A nyolc mintavétel elcsúszása a bit közepéhez.

3.1.2 A kommunikációs portot megosztó modul (PORT_CONNECTOR.v)

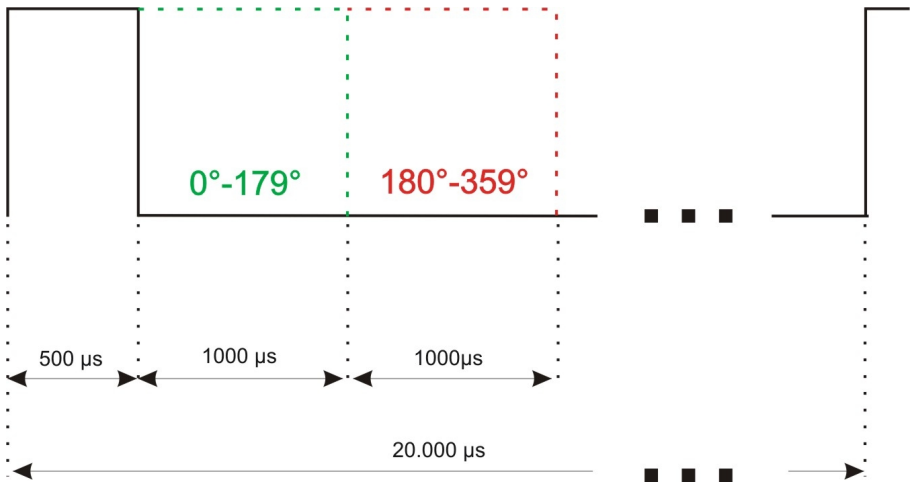
Egyetlen egy portot tudunk adattovábbításra használni, azonban két modulnak is szükséges az adatcsere a vezérlőprogrammal, ráadásul egyikük kommunikációs csatornához jutása kritikusabb, mint a párjáé. Jelen problémát oldja meg ez a modul úgy, hogy négy másikat prioritással csatlakoztat a soros porthoz. Ezen négy modul közül csak kettő került bekötésre, a fennmaradókat a későbbi fejlesztéshez lehet használni. A modul, egy kombinációs hálózatot kódol, működésére nincs szükség órajelre. Adat- és vezérlőkimenete közvetlenül csatlakozik az RS232-es modulra.

Az egység bemeneteihez (A, B, C, D) csatlakozó modulok adat be-, ki-, illetve kontrollvonalakkal kapcsolódnak. Ez utóbbi segítségével az adott egység kommunikációs kéréssel fordulhat a vezérlőhöz, illetve informálódhat arról, hogy a port foglalt-e. A bemenetek az ABC betűinek sorrendjének megfelelő prioritással rendelkeznek. A modul nem bonthatja meg az aktív kommunikációt, nincs kontrolláló szerepe, ha egyik bemenete épp adatcserét folytat. Miután ezzel felhagyott és visszaadta a portot, a vezérlő megvizsgálja, hogy a bemeneteire csatlakozó egységek melyikétől érkezett kérés, majd prioritásban a legmagasabbat összekapcsolja a soros portot kezelő modullal, és jelzi a többi modulnak, hogy a kommunikációs vonal használatban van. A kapcsolódó modulokban óvatosan kell kezelni a kommunikációt, ugyanis ha valamely magas prioritású modulban túl hanyagul programozunk és folyamatosan lefoglaljuk a soros portot, a prioritásban alatta lévő modulok soha, vagy csak nagyon ritkán tudnak kommunikálni. Szintén óvatosan kell kezelnünk az adatáramlást blokkoló vezetéket (flow_controll), ugyanis ezen vezetéket bármelyik modul aktiválja, azonnal blokkolja az adatcserét, mivel ezen vezetékekhez nincs prioritás rendelve; egymással vagy kapcsolatban állnak.

3.1.3 Motorvezérlő modul(PWM.v)

3.1.3.1 *Elméleti háttér*

A szervomotorokat PWM jellel kell vezérelni. Ez a betűszó a Pulse Width Modulation, azaz az impulzus-szélesség modulációt takarja. A vezérlés alapja egy négyszögjel, mely az információt a kitöltési tényezőjében[4] hordozza. Ezen szervomotorok esetében ez a négyszögjel 50 Hz-es alapfrekvenciával bír 500µs-os alapkitöltéssel. Ez a kitöltés felel meg a 0°-os pozíciónak, míg a 2500µs-os a 359°-os elfordulást kódolja (10-ik ábra). A kitöltést ezen intervallumhatárok között módosítva szabadon állíthatjuk a szervó elfordulását. Az 50Hz-es alapjel 20.000µs periódusidővel rendelkezik, azaz az információ egy periódus 20-ad részén kódolódik.



ábra 10: A szervók PWM vezrlése

3.1.3.2 *Gyakorlati megvalósítás*

A robotkart 6 darab szervomotor mozgatja, mely mindegyike külön szabályozható PWM vonalat igényel. A modul-felépítésnek hála elegendő volt egyszer megírni a PWM modult, majd ezt külön-külön példányosítani a megfelelő motorhoz[5]. A PWM jel a kimeneten az alapjelből, és a hozzáadott kitöltési jelből áll. Mivel az alapjel 50Hz-es, a kártya órajelét 320.000-el kell leosztani, azaz létre kell hozni egy számlálót, ami 0-tól 319.999-ig számlál. Kialakítva egy vezetéket ami csak akkor igaz amikor a számláló elérte a végállapotát, egy

4 : Egy periódusra vonatkoztatva az az arány amíg a jel logikai 1.

5 : A hat motor közül ötöt lehet azonos kitöltéssel vezérelni, a hatodik, mely a fogórészt mozgatja külön kitöltést igényel, ez azonban könnyen megvalósítható egy paraméter átírásával.

olyan vonalhoz jutunk amin 50 Hz-es aszimmetrikus órajel található. Ezen vezeték használható a PWM alapjelének előállításához, mégpedig úgy, hogy minden 16 Mhz-es órajelütésnél megvizsgáljuk annak értékét, és ha az 1, akkor elindítunk egy másik számlálót ami 0 tól 7999+X_d-ig számol. Amíg ez a számláló el nem érte a végállapotát, a PWM kimenet 1-es lesz, majd visszavált nullába, és az egész PWM-számláló áramkör deaktiválódik a következő 50 Hz-es órajelütés beérkeztéig. A PWM jel kitöltését egy nyolc bites szám reprezentálja, ahol a 0 a 0°-os, a 255 a 359°-os elfordulást jelenti. Ez a felbontás elég, mivel a szervomotorok ezt se tudják pontosan lekövetni (olcsó húsnak híg a leve), így fölöslegessé válik a nagyobb felbontású elfordulás-ábrázolás. Ezen nyolc bites értéket át kell kódolni tényleges kitöltési értékké. Ezt az átkódolást a bit2width.v modul végzi. A kitöltés 500-2500µs-ig változhat, ami azt jelenti, hogy 2000µs-nyi eltérést kell reprezentálni. Erre az időre pontosan 32.000 16MHz-es órajelütés jut, azaz ezt az intervallumot kell 8 biten lineárisan kódolni. Sajnos $\frac{32.000}{255} \approx 125{,}49 \notin \mathbb{Z}$ így ha csak 125, vagy 126 órajelet jelentene minden kitöltési érték, nagy elfordulásoknál pontatlan lenne a pozicionálása a szervoknak. Szerencsére az értékünk nagyon közel áll a 125,5-höz, így ha nyolc bites elfordulási érték felét 125, felét pedig 126 órajelütéssel helyettesítem, az össz-hiba minimalizálható. Teljes elfordulásnál csupán 3 órajelütésnyi túlcsúszási hiba van. Például: A bemeneti értékünk 79. A modul így 39*126 + 40*125-ös értéket, azaz a 9914-es számot adja kimenetül, míg a valós érték 9913,73. Természetesen teljes elfordulásnál nem 32.003, hanem 32.000 a modul kimenete.

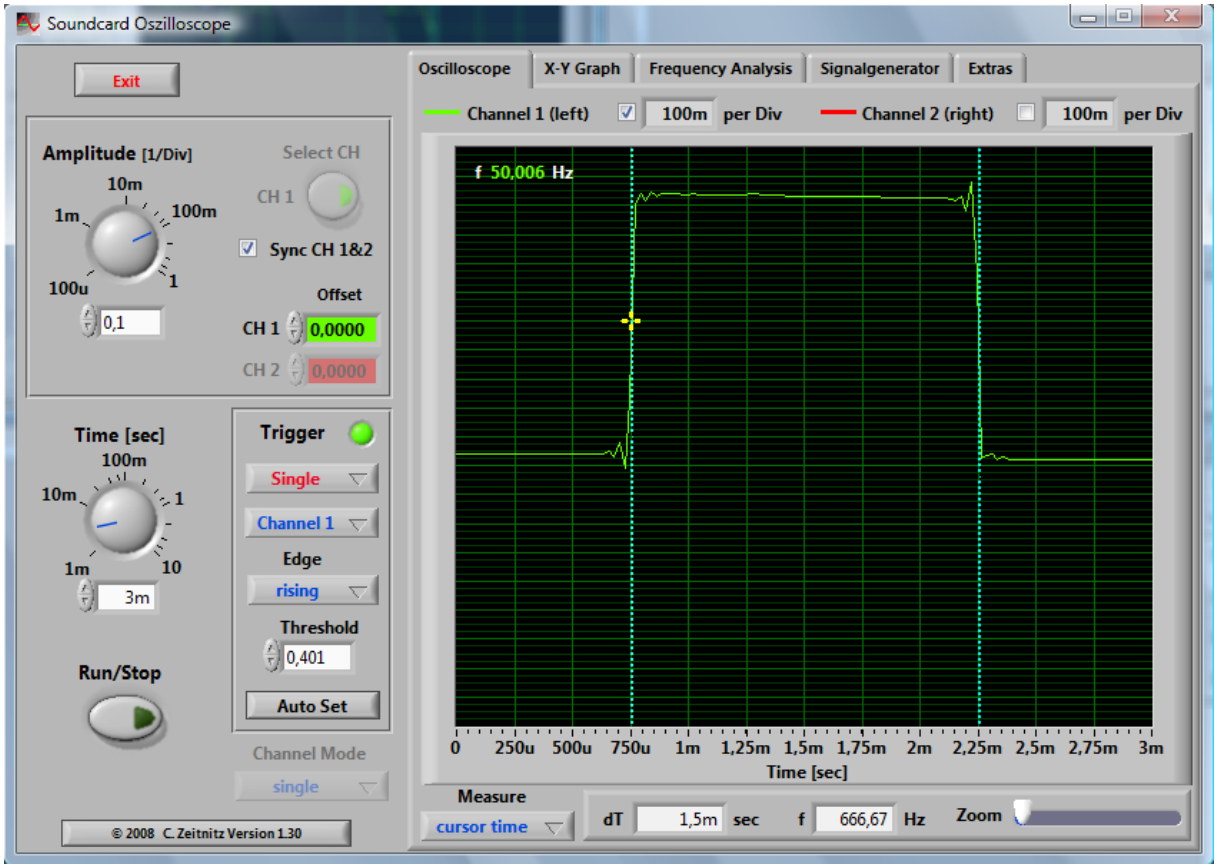
Visszatérve a PWM modulba, a bit2width modul bemenete a nyolc biten kódolt elfordulási érték, kimenete pedig 0...32.000-ig terjed (X_d értéke), mely hozzáadódik az alapkitöltéshez (7999). Így a PWM kimenet 8000...40.000-ig 16MHz-es órajelütésig lehet egyes, visszaszámolva:

$$\frac{8.000\,(ez\ az\ alap\ kitöltés) + X\,(0...32.000\ \ a\ bit2widt\ modul\ kimenete)}{16.000.000\,(ez\ a\ Logsys\ kártya\ órajele)} = 500\,\mu s + (0...2000)\,\mu s$$

Mivel a kar fogórészénél lévő szervó motor más típusú, így más PWM jelet is igényel. Információt sajnos nem sikerült találnom róla az interneten, így kísérletezéssel sikerült megállapítanom a hozzá tartozó PWM_k.v modul beállításait. Szerencsémre csak az

alapkitöltést kellett megváltoztatni 8000 ről 21.001-re, mely 1312 µs-nyi alapkitöltésnek felel meg. A végállástól végállásik történő vezérléshez a 2000µs-os lökettartomány elégnek bizonyult.

A PWM kimenet pontosságát mérésekkel is ellenőriztem, azokat egy 4N25GV típusú optocsatolóra kapcsoltam, majd ennek kimenetét a PC hangkártyájának bemenetével kötöttem össze. A hangkártyát a ZEINITZ programmal (http://www.zeitnitz.de/Christian/scope_en) úgy használhatom, mint egy kétcsatornás 20 Khz-es oszcilloszkópot, és mivel a PWM jelek 50Hz-es alapfrekvenciájúak, a jelalak megvizsgálható. Az alábbi ábrán látható az 1500µs kitöltésű PWM jel. A jel alapfrekvenciáját a képernyő bal felső sarkában olvashatjuk le, míg a kijelölt szakasz hosszát a dT adja meg a megfelelő mértékegységben.



ábra 11: 1500µs kitöltésű PWM jel az FPGA kimenetén

Az 500µs-os PWM jel a 23-ik ábrán, a 2500µs-os a 24-ik ábrán, egymást követő impulzusok a 25-ik ábrán, és a PWM jel spektruma a 26-ik ábrán figyelhető meg.

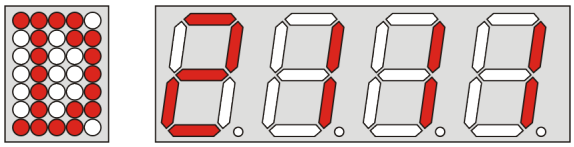
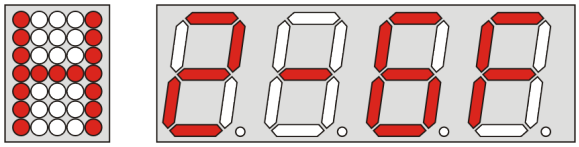
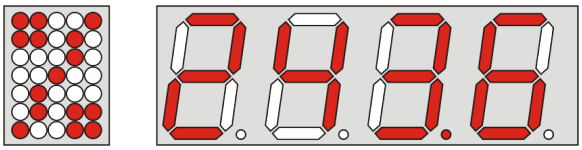
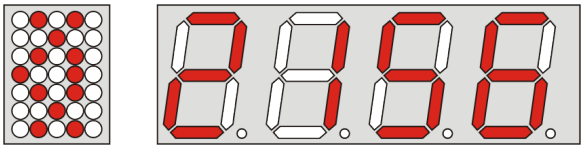
3.1.4 Parancsértelmező modul (COMMAND_HANDLER.v)

Ezen modul feladata kezelni, strukturálni a soros vonalról érkező adatokat, valamint annak függvényében értéket adni a PWM.v modulok kitöltési tényezőjét meghatározó nyolc bites értékeknek, hogy a működés milyen állapotban van. Bemenete közvetlenül kapcsolódik a soros portot kezelő modul kimenetére (azon nyolc bites kimenetre, amin a vett byte található), valamint annak érvényességét jelző vonalra, továbbá a memóriát kezelő modul azon kimeneteire, mely egy adott csatorna kitöltési tényezőjét adja meg, és arra mely beazonosítja azt, hogy ezen érték a 8 PWM csatorna közül melyikhez tartozik. Kimenete az aktuálisan fogadott parancs kódja, ennek érvényességét jelző vezeték, valamint a 8 PWM-csatornához rendelt, egyenként 8 bites kitöltési tényező.

A modul 3 alegységre bontható. Az első alegység megvizsgálja a futás állapotát, és ha az programfuttatási állapotban van, azaz a szervok az elfordulásuk értékét a memóriában tárolt adatokból nyerik, akkor az aktuális csatornához hozzá rendeli ezt az értéket. A második egység figyeli, hogy van-e érvényes soros porton fogadott adat. Ha igen, megvizsgálja, hogy az megfelel-e valamelyik parancskódnak. Ha így van, akkor kimenetére rakja ennek a parancsnak a kódját, és egy órajelciklusig érvényesként jelöli azt. Ezzel párhuzamosan a dekódolt parancsból pontosan kiderül, hogy őt hány adatbájt fogja követni. Amíg be nem érkezik ennyi adatbájt, a modul nem kezdi el a beérkező adatokat parancskódként értelmezni, hiszen tudatában van annak, hogy ezen adatok, a legutolsónak beérkezett parancshoz tartoznak. Mivel ezen modul tud csak értéket adni a PWM-modulok bemeneteinek, így itt kell lekezelni azt az esetet is, amikor egy olyan parancs érkezik, melyet az adott csatorna kitöltési tényezője követ. Minden olyan esetben amikor ilyen parancsot veszünk, és az üzemmód megengedi (nem programfuttatási üzemmódban vagyunk), értéket kap az adott csatorna abban a pillanatban, amikor beérkezett a hozzá tartozó adat. A harmadik alrésznek csupán annyi a dolga, hogy pontosan egy órajelciklusig legyen érvényes a parancskódot tartalmazó busz.

3.1.5 Kijelzést megvalósító modulok (szamlalo.v, kijelzo.v)

A LOGSYS panelen helyet kapott kijelzőket arra használtam fel, hogy a motorok pozíciójáról információt jelenítsenek meg. A pontmátrix kijelző informál minket arról, hogy a 7-szegmenses kijelzőn milyen típusú a megjelenítés, ez utóbbi első karaktere pedig arról, hogy a maradék három karakteren mely csatornához tartozó információt látjuk. Az RST feliratú gomb megnyomásával 4-féle kijelzési mód közül választhatunk. A kiválasztott csatorna számát a BTN3-as gomb lenyomásával csökkenteni-, míg a mellette lévő BTN2-es gombbal növelni tudjuk. Az alábbi ábra a kettes csatornához tartozó érték négyféle kijelzését mutatja be. A csatorna kitöltése 111_{10}

Megjelenő információ	Magyarázat
	Jelen esetben a kettes csatorna kitöltési értéke 111_{10}. Emlékeztetőül, ez az az érték, ami a kettes motorra rákötött PWM modul nyolcbites bemenetére kerül.
	A kettes csatorna kitöltési értéke 0x6F
	A kettes csatorna a teljes kitöltés 43,6%-ig 1-es értékű.
	A kettescsatorna jelenlegi kitöltése 156°-os elfordulást eredményez.

táblázat 3: Kijelzési módok

A LOGSYS kártya kapcsolási rajzáról látható ([10] 11-ik oldal), hogy a hétszegmenses kijelző szegmenseihez tartozó katódok közösek, továbbá a pontot kivéve ezek a vezetékek párhuzamosítva vannak az 5×7 pontmátrix kijelző 7 sorával. A karakterekhez, illetve oszlopokhoz tartozó anód minden esetben egyedi. Ez a megoldás kicsit kényelmetlenné teszi a kezelést, azonban rengeteg IO portot spórol meg, mivel ebben az esetben összesen 8+4+5, azaz 7 szegmens plusz pont, a karakterekhez tartozó 4-, illetve az oszlopokhoz tartozó 5 anód

vezeték. Ez összesen 17 IO portot foglalt el, ellentétben azzal az esettel amikor az anódok kerülnek közösítésre, és minden egyes szegmenshez, 8 vezetéket használunk. Így 8×4, azaz 32 IO portra lenne szükség csupán a 4 karakteres hétszegmenses kijelzőhöz.

A kijelzőket a kijelzo.v modul a felhasználói útmutatóban leírt módszer analógiájára idő-multiplexeléssel vezérli. A modul bemenete egy két bites busz, mely a kijelzés módját kódolja (0: decimális, 1: hexadecimális, 2: százalékos, 3: fok), és egy 16 bites (4×4 bit) vezeték, mely a kejelzendő információt hordozza. A kimenet a hét szegmenshez, plusz a tizedesponthoz csatlakozó busz, és a karakterekhez illetve oszlopvektorokhoz tartozó buszok. A modul 512Hz-es sebességgel váltogatja az aktív kijelzést, vagyis minden egyes oszlop, illetve karakter 1,9ms-ig jelez ki. Ez az idő elegendő ahhoz, hogy a LED-ek begyújtsanak. A teljes kijelző 56,8Hz-el frissül, ami a szem számára folyamatos. Váltáskor a modul megvizsgálja, hogy éppen melyik kijelző aktív. Amennyiben a pontmátrix, akkor a kijelzési módnak, és az aktív oszlopnak megfelelő 7 bites minta jelenik meg. A mintákat (D, H betű, százalék és fok-jel) a CorelDraw nevű programmal terveztem meg, és a terveknek megfelelő bitmintát írtam a Verilog kódba. Ha a 7 szegmenses kijelző az aktív, megnézi, hogy azon belül melyik karaktert kell kijelezni, és a 16 bites bemenet megfelelő 4 bitjét kódolja át úgy, hogy a kijelzőn ezen négy bit értékének megfelelő szegmensek gyulladjanak fel. Természetesen ennek a modulnak a feladata az is, hogy a tizedespontot kirakja akkor amikor szükséges, illetve, hogy hexadecimális kijelzési módnál kötőjelet rakjon a csatornaszám, és az értéke közé.

Az előbbi egység csupán a megfelelő módon jelenítette meg az információt, annak értékéről nem tud semmit. A PWM kitöltési értékek további feldolgozása szükséges, hogy általunk érthetőbben jelenjenek meg. Ezt a feladatot látja el a szamlalo.v modul, melynek kimenete szolgáltatja ez előbbiekben tárgyalt modul 16 bites bemenetét (a megjelenítendő értékek). Bemenete a PWM jelek kitöltési tényezői (egyenként 8 bites buszok), a megjeleníteni kívánt csatorna száma, valamint a megjelenítéshez használt üzemmód. A modul a bejövő PWM jelek közül annak értékét adja belső regiszterének, amelyik a kiválasztott csatornához tartozik. A későbbiekben a modul ezen regiszter tartalmával dolgozik. Ezután a kijelzési módot vizsgálja, és kapcsolja a megfelelő értéket a kimenetre. A legmagasabb helyiértékű négy bithez (a legbaloldalibb karakterhez tartozó bitnégyes) mindig a kiválasztott csatorna száma rendelődik hozzá. Mivel ez 0...7 között változhat, nem szükséges átalakítás (az ezen értéktartományba

eső számok tízes és tizenhatos számrendszerbeli ábrázolása megegyezik). Ha a kijelzési mód hexadecimális, a dolgunk egyszerű, ugyanis elég összekötni a kimenet utolsó két félbájtját a bemeneti regiszterrel. A fennmaradó 8...11 bit tartalma lényegtelen, ugyanis a kijelzo.v modul ilyen kijelzési módnál mindig egy kötőjelet rak erre a karakterre. Ha a kijelzési módunk decimális, akkor a bemeneti regiszter értékét át kell alakítani decimális ábrázolásúvá. Ezt a hex2dec.v modul segítségével lehet megtenni. Ez a modul egy kombinációs hálózatot realizál, ezáltal a bemenetére adott jel eredménye azonnal érvényes a modul kimenetén (természetesen itt is van egy idő, mire a jel megjelenik a kimeneten, azonban ez sokkal rövidebb, mint egy órajelciklus). A modul bemenetnek egy 20 bites hexa számot vár, amit 25 bites decimálissá alakít át (2^{20}=1.048.576, és mivel egy karaktert azaz 0...9 ig terjedő értéket legkevesebb 4 biten tudunk ábrázolni, így 6×5 lenne szükséges a szám ábrázolásához, azonban mivel 20 biten maximálisan a fenti szám mínusz egyet tudjuk ábrázolni, ahol legnagyobb helyiértékű számjegy csak 1 lehet, így ezt a karaktert elégséges egy biten tárolni, tehát az egész szám reprezentálásához 6×4+1 bit, azaz 25 bit szükséges).

A hex2dec modul bemenetére rákötjük a bemeneti regiszterünket (ez a regiszter tartalmazza az aktuálisan kijelzett csatorna PWM kitöltését). Értelemszerűen a maradék 12 bitre 0-t kötünk. A modul kimenetének utolsó 3×4 bitje lesz az ami decimális kijelzési módban a csatornaszám négy bitje mögött felsorakozik a szamlalo.v modul kimenetén.

Százalékos kijelzésnél a bemenet mind a nyolc bitjét megszorozzuk a hozzá tartozó konstanssal, majd összegezzük a számot. A módszer lényege, hogyha minden bit egyes, azaz a bemeneti értékünk 255, akkor a konstansok összege 100 legyen. Ha a legnagyobb helyiértékű bithez 50-et rendelek, és az alatta lévőhöz az őt megelőző konstans felét és így tovább, a konstansok összege értelemszerűen a 100-at adja ki. Mivel az FPGA alapértelmezetten nem tud lebegőpontos számokat ábrázolni, így ezzel a módszerrel, az LSB-hez közeledve a konstansok nagy pontatlanságot adnának ki. Mivel a hex2dec maximum 20 bites hexa bemenetet vár, használjuk ki, úgy, hogy szorozzuk fel a konstansok értékeit 1000-el, azaz a legnagyobb helyiértékű bithez tartozó konstans 50.000-et, az alatta lévő 25.000-et, és így tovább jelentsenek. Az n-edik bithez tartozó konstans értékének kiszámítási képlete a következő: $const_n = \left[\frac{100.000}{2^{8-n}} \right], ahol\ n = [7,0], n \in \mathbb{Z}$. A hex2dec modul bemenetére azon

konstansok összege kerül, amelyhez tartozó bit 1-es értékű, azaz $input = \sum_{n=0}^{7} bit_n * const_n$. A kimenet egy olyan szám lesz, mely 0 és 100.000 között a bemeneti értékkel arányosan változik, azaz a százalékos kitöltés ezerszerese. Mivel a szám decimális formátumban van, minden bitnégyes elhagyása 10-el való osztást jelent. A kijelző.v modul százalékos kijelzési módban a következő formában várja a bemenetet:

Bit sorszáma(MSB->LSB)	11	10	9	8	7	6	5	4	3	2	1	0
Bitnégyes helyiértéke	Tízes				Egyes				Tizedes			

A száz százalék kijelzését a 0xA00 érték átadásával tudom elérni, minden egyéb esetben a fenti táblázat a mérvadó. Ennek megfelelően a hex2dec modul kimenetének megfelelő csoportját adom át. Ez a 19-es helyiértékű bittől a 8-as helyiértékűig terjed.

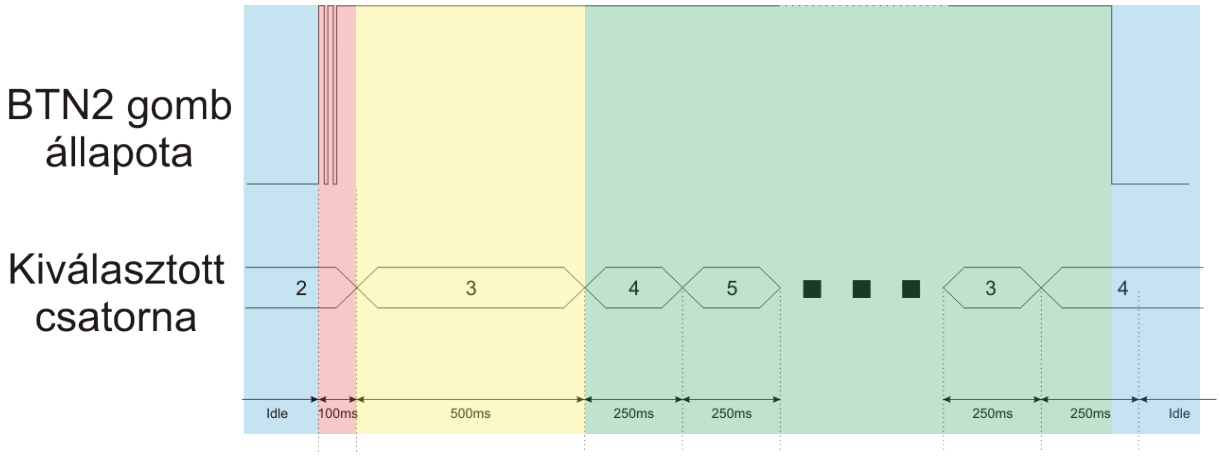
A fok kiszámítása némileg más módszerrel történik. Ebben az esetben kiszámoljuk az egy bit-hez tartozó fok értékét $\frac{360}{255} = 1{,}411$ majd, hogy pontos legyen a számolás megszorozzuk ezt 100-zal. Természetesen ezt a műveletet nem az FPGA végzi, hanem a kódban, egy konstans reprezentálja ezt az értéket (141). Ezt a számot, és az aktuális kitöltést összeszorozva átadjuk szintén egy hex2dec modulnak, ami elvégzi a konverziót. Végül pedig visszaosztva a kimeneten található számot 100-zal (jobbra shiftelem 8 bittel) megkapom az elfordulás értékét fokban. Ez az érték jelenik meg a kijelzőn.

3.1.6 Gombokat kezelő modul (CONTROL.v)

A kijelzőkön csekély információ jeleníthető meg egyszerre, tehát az azon megjelenő információt ki kell választanunk. Ezt a megfelelő gombok megnyomásával érhetjük el. A gombnyomási eseményt jelen modul kezeli le szakszerűen. Bemenete a gombokhoz vezető busz, kimenete a kijelzési mód, valamint a kijelezendő csatorna száma. Az öt gomb közül csak három használt, funkciói a 3.1.5 -ös pont elején részletesen ismertetésre kerültek. A modul szándékosan úgy lett megvalósítva, hogy csak akkor reagáljon, ha egyszerre egy gombot nyomtunk le. Ha ez bekövetkezik, a modul elindítja a pergésmentesítés-t (12-es ábra, piros sáv), azaz 100 msot kivár, és vizsgálja, hogy lenyomva tartottuk-e addig a gombot.

Ennek egyik oka a zavarszűrés, azaz, hogy a gomb eseménye csak akkor következik be, ha ténylegesen lenyomtuk az adott gombot, másik a mikrokapcsolók fizikai tulajdonságai miatt lényeges, ugyanis a kapcsoló úgy működik, hogy két homorú lemez nyomás hatására egymáshoz pattan, és így zárja az áramkört. Az összepattanáskor azonban a lemezek visszaverődnek egymásról, akár annyira is, hogy a másodperc tört része alatt nyissák, majd újra zárják az áramkört. Ezt a jelenséget hívjuk pergésnek. 100 ms elteltével a lamellák beállnak nyugalmi helyzetükbe, így ezzel a megoldással kiküszöböltük azt is, hogy a pergés miatt többször is lefusson a gombnyomáshoz rendelt kód.

Miután a pergésmentesítés megtörtént megvizsgáljuk, hogy melyik gombot nyomtuk le, és annak megfelelően módosítjuk a kijelezendő csatorna számát, vagy a kijelzési módot (12-es ábra sárga rész). Ha a gombot tovább tartottuk lenyomva, mint fél másodperc, akkor egy másik üzemmódba lépünk át, ami negyed másodpercenként újra, és újra aktiválja a gomb eseményéhez rendelt áramkört, mindaddig amíg a gombot lenyomva tartjuk (12-es ábra, zöld rész). A megértéshez az alábbi ábra nyújt segítséget.



ábra 12: Példa a BTN2-es gomb megnyomására, és annak állapotaira

3.1.7 Az inicializáló, és esemény kezelő modul (STATE_HANDLER.v)

Ennek a modulnak a feladata az, hogy folyamatosan informálja a PC-n futó vezérlőprogramot az FPGA-ban bekövetkezett állapotváltozásokról. Az inicializálási folyamatot is ez a modul kezeli le úgy, hogy az összes figyelt eseményt bekövetkezettnek tekinti, így mindenről informálja a vezérlőprogramot. A modul a csatornák értékét, a megjelenített csatornát, és a megjelenítés módját figyeli. Ha bármelyikben változás következik be, elküldi új értékét a vezérlőprogramnak.

A modul bemenetei a megfigyelt vonalak, (nyolc darab 8 bites PWM jel, a kijelzett csatorna száma, és a megjelenítés módja) kimenete pedig a 3.1.2 pontban tárgyalt megosztómodul B portjához csatlakozik. A modul csak akkor működik, ha a soros portot senki sem használja, mivel hiába történt is változás, ha a port foglalt nem tudjuk a változást elküldeni, így fölösleges bármit is vizsgálni. Ha a port szabad, akkor megvizsgáljuk, hogy történt-e valamelyik figyelt értékben módosulás. Minden bemenetnek van egy vele megegyező méretű regisztere, melyben az aktuális órajel előtti értéke került letárolásra. Ezen regiszterek valamelyikével kerül összehasonlításra a megfelelő bemenet. Mivel összesen 11 értéket kell megvizsgálni (a valóságban 12-t, azonban a legutolsó értékhez nincsen még kód implementálva) megnehezítené a kezelést, ha minden bemenetet a megfelelő regiszterhez hasonlítanánk. Ehelyett egy 12 bites regiszter, bitjei reprezentálják az eltérést, azaz, ha az adott helyiértéken a bit 1-es, akkor a hozzárendelt bemenet, és a regisztere között eltérés van, ha 0, akkor nincs. Ezen vektor elemeit vagy-kapcsolatba hozva könnyen megállapítható, hogy a 12 figyelt bemenet közül melyik változott meg (bármelyik bit egyes, a vagy-kapcsolat eredménye is 1 lesz, egyébként 0). Ezzel a vagy-kapcsolattal áll összeköttetésben a soros portot elkérő vonal is. Ha az igaz, azaz változás történt, és megkaptam a soros portot, megvizsgálom a vektort a Verilog casez utasításával. Ezen nyelvi eszköz segítségével szét tudom bontani a bekövetkezett eseményeket, függetlenül attól, hogy abból hány következett be egyszerre. Megértéséhez vizsgáljuk meg a casez utasítás igazságtábláját:

Casez	0	1	X	Z(?)
0	1	0	0	1
1	0	1	0	1
X	0	0	1	1
Z(?)	1	1	1	1

táblázat 4: Casez utasítás igazságtáblája. Forrás: [5]

Az X értéknek csak a szimulációknál van szerepe, így azt hagyjuk figyelmen kívül. Látható, hogy a Z érték a 0-val, és 1-el is igaz eredményt ad, így a minta illesztésénél ott ahol a Z érték szerepel (ez a kódban ?-ként jelenik meg, mivel így jobban szemléltet), mindig igaz lesz a kifejezés eredménye. Ezzel elértük azt, hogy a mintában csak azon biteket vizsgáljuk, melyek számunkra lényegesek. Egy másik fontos tulajdonsága az utasításnak, hogy ha a felsorolt esetek közül több minta is illeszkedik, mindig a felsorolás sorrendjében első egyezés kódja fut le, az alatta lévők nem, azaz a casez prioritás dekóderként is működik. Mindezen hasznos tulajdonság kiaknázásra került a kódban is, úgy, hogy az eseteknél a 12 bites mintákban egy bit értéke definit, a többi érték helyén kérdőjel szerepel. Így a 12 bites mintát 12 esettel kezelhetem le. A vizsgált bit az MSB-től LSB irányába vándorol, azaz ha az összes esemény be is következik, mindig meghatározott sorrendben kerülnek azok kiértékelésre.

Miután kiválasztottuk, hogy melyik eseményhez rendelt kód fusson le, első lépésben elküldjük a továbbítani kívánt információhoz tartozó parancskódot, majd utána a megfelelő adatokat. Az utolsó adatbájt továbbításakor elmentjük az új értéker annak shiftregiszterébe, és a figyelt értékhez tartozó bitet 0-ra állítjuk a változást indikáló vektorban.

Előfordulhat olyan eset amikor ugyan történik állapotváltozás, azonban azt mégsem szeretnénk elküldeni a vezérlőprogramnak. Ebben az esetben csak értékül adjuk a megfelelő tárolóregisztereknek az aktuális bemeneti értéket.

Ha a soros port szabad, és nem történt semmiféle változás az indikáló vektorunk szerint, akkor frissítjük abban a státuszbiteket (megvizsgáljuk, hogy a bemenethez tartozó regiszter, és a bemenet értéke megegyezik-e).

3.1.8 Memóriát és automatikus futtatást kezelő modul (MEMORY_CONTOLLER.v)

Szakdolgozatomnak ez a legterjedelmesebb és legbonyolultabb modulja, melynek feladata kialakítani a megfelelő struktúrát a nyers memóriában, majd azután kezelni azt. A memória tartalmazza a mozgásprogramokat (továbbiakban programok). A modul kapcsolatban áll a memóriával, a soros portot elosztó-, a kijelző-, és a parancsértelmező modullal.

A LOGSYS kártyán helyett kapott egy 128 kbájtos statikus[6] ram. A memória hozzáférési ideje 10ns ([12]), így az időzítések nem igényelnek külön figyelmet. (a kártya 16MHz-en megy, melynek periódusideje 62.5ns). A RAM 17 bites cím-, valamint 8 bites adatbusszal rendelkezik, így $2^{17}=131.072$ bájt információ tárolható az egységben. Ezeken, és a tápellátást biztosító lábakon kívül rendelkezik még három aktív alacsony szintű vezérlőlábbal $\overline{WE}$ $\overline{CS}$ és $\overline{OE}$. A $\overline{CS}$ lábbal tudjuk a memóriát aktív állapotba hozni, amíg ez a láb logikai 1 értékű, a chipet nem tudjuk használni. Ennek a lábnak akkor lenne jelentősége, ha több memóriachipet szeretnénk egyszerre használni, és általa kontrollálnánk azt, hogy melyik chipet akarjuk írni, vagy olvasni. A memória használata a következő:

- Beállítjuk a vezérlővonalakat
- A címsínre helyezzük azon bájtnak a címét amelyiket írni, vagy olvasni szeretnénk
- Az adatsínen az adott címen lévő adat jelenik meg, esetleg az azon lévő adat kerül rögzítésre

Ennek megfelelően az adatsín két irányú (tri-state vonalak), így vezérelni szükséges az adatáramlás irányát. Erre a $\overline{WE}$ és $\overline{OE}$ lábakkal van lehetőségünk. A lábak állapotának megfelelően a memória az alábbi üzemmódokban lehet:

6 :A statikus ramokban az adattároló elemeket bistabil latchek alkotják, ezáltal a beírt adat nem igényel frissítést, ellentétben a DRAM-okkal. Az SRAM-ok csak tápfeszültség megléte mellett őrzik meg az adatokat, viszont léteznek olyan speciális SRAM-ok is, melyek tápfeszültség nélkül is tárolják azt, azonban a kártyán nem ilyen kapott helyet.[11]

$\overline{WE}$	$\overline{OE}$	Üzemmód
0	0	Ír
0	1	Ír
1	0	Olvas
1	1	Z

táblázat 5: A memória működési üzemmódjai. Forrás: 28-ik ábra

A fenti táblázat forrásául szolgáló sematikus ábrából látható, hogy ha mindkét vezérlő bemenet 1-es állapotban van, akkor az adatvonal nincsen összekapcsolva sem a kimentet meghajtó fokozattal, sem a bemenettel. Természetesen ez az állapot nem egyezik meg azzal a nagyimpedanciás állapottal amikor az eszközt írni szeretnénk. Látható továbbá az is, hogy hardveres a védelem a kimenet és a bemenet szembefordítása ellen, azaz, mikor mindkét vezérlővonal logikai 0, az írás üzemmód aktiválódik.

3.1.8.1 *A memóriában kialakított struktúra*

Mivel a memória bájt szervezésű, így több bájtot szükséges összefogni az információk tárolásához. Ezen összefogott bájtokat a továbbiakban blokkokként említem. A blokkok 12 bájt méretűek, és attól függően, hogy milyen funkciót töltenek be három típusba sorolhatóak. A tárolt blokkok első két bájtjának a struktúrája megegyezik, magyarázatot lásd később. A memória logikailag két részre osztott; blokk részlegre, ahol a blokkokat, azaz a tényleges információkat tároljuk, és a BAT részlegre, azaz a BlockAllocatonTable, egy a FAT mintájára kialakított tartalomjegyzék-részlegre, ahol menedzseljük azt, hogy a felhasználható blokkok közül melyek szabadok, illetve melyek nem. A blokk részleg a memória 000.000 (0x00000) kezdőcímétől a 129.719 (0x1FAB7) címig tart. Ez összesen 10.810 blokk tárolására ad lehetőséget. A BAT részleg a 129.720-as (0x1FAB8) címtől kezdődően a memória végéig tart (29-ik ábra). A BAT részlegben minden egyes bit a hozzá tartozó blokk foglaltságát jelzi,így egy címen nyolc blokkot tarthatunk nyilván. Ez azt jelenti, hogy összesen 10.816 blokkot tudnánk megcímezni, nekünk azonban csak 10.810 blokk fér el a memóriában, így az egyik címen a nyolc bitből csak 2-t használunk.

A blokkoknak három típusa: start, step és stop. A blokkok struktúrálisan három részre bonthatóak: blokk típusát meghatározó rész, mutató rész, és adat rész. Egy

mozgásprogramnak minimálisan tartalmaznia kell mind a három blokktípust. Így a program keretbe záródik, és értelmetlen programot sem tudunk létrehozni. Egy program lényegében nem más, mint blokkok cirkuláris láncolása. A láncot a start típusú blokkon keresztül tudom megfogni, mely a lista fejének tekinthető. A további elemek a lista adattagjai (step típus), és végezetül egy speciális step típus, a stop típus, mely ugyancsak tartalmaz adatot, azonban mutatórészlege visszamutat az első step blokkra. Ezzel a megoldással lehetőség nyílik, hogy egy programot többször is lefuttathassunk egymás után. Nyilvánvaló, hogy egy programban csak egy start, és egy stop blokk lehet, míg step blokkból annyi, ahány lépésből áll a program, mínusz egy (a stop blokk miatt). A start, valamint a step, és stop blokkok felépítését az alábbi két táblázat szemlélteti.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
2'b01 (START_INF)		Következő utasítás relatív címe [13:8]						1. byte
Következő utasítás relatív címe [7:0]								2. byte
Program-szám(program_number[11:4])								3. byte
Program-szám(program_number[3:0])				ismétlésre vonatkozó inf. [3:0]				4. byte
Z (a 12 bytos blokkméret miatt szükses ez a nyolc byte. tartalma érdektelen)								5. byte 6. byte 7. byte 8. byte 9. byte 10. byte 11. byte 12. byte

táblázat 6: A start blokk felépítése

2'b10 (STEP_INF)/ 2'b11 (STOP_INF)	Következő/első utasítás relatív címe [13:8]	1. byte
Következő/első utasítás relatív címe [7:0]		2. byte
A blokkban tárolt információ alkalmazása előtti késleltetés **felső** nyolc bitje[15:8]		3. byte
A blokkban tárolt információ alkalmazása előtti késleltetés **alsó** nyolc bitje[7:0]		4. byte
1-es csatorna adata [7:0]		5. byte
2-es csatorna adata [7:0]		6. byte
3-as csatorna adata [7:0]		7. byte
4-es csatorna adata [7:0]		8. byte
5-ös csatorna adata [7:0]		9. byte
6-os csatorna adata [7:0]		10. byte
7-es csatorna adata [7:0]		11. byte
8-as csatorna adata [7:0]		12. byte

táblázat 7: A step, és a stop blokk felépítése

Látható, hogy mind a három típusnál a letárolt 1. bájt legmagasabb helyiértékű két bitje mutatja meg, hogy az aktuális blokk milyen típusú, azaz az milyen információt milyen struktúrában tárol. A start blokknak a 0x0, a step-nek a 0x1, a stop blokknak pedig a 0x2-es szám felel meg. Ezt követve 14 bájton tárolódik a végrehajtásban következő blokk relatív címe, azaz, a blokk sorszáma (a 10.810 blokkból az adott blokk hanyadik). A relatív címzés megvalósítása nehezebbé tette a kódolást, azonban előnye kézzel fogható. A 10.810-es számot 14 biten tudjuk ábrázolni, így egy blokk relatív címe ennyi biten tárolható ellentétben az abszolút címzésnél, ahol 17 bit lenne ehhez szükséges. Természetesen minden relatív cím a használata előtt egy kombinációs hálózat segítségével visszaalakításra kerül abszolút címmé. A 12 bájtos egységes blokkméret itt válik fontossá, mivel így sokkal könnyebben alakítható vissza a relatív címből az abszolút, mintha a blokk méretek különböznének. A relatív címzés használatának további előnye, hogy a step és stop blokkon megspóroltunk virtuálisan 3 bitet, ez azonban 8 lesz a valóságban, mivel a blokk struktúrája megbomlik (step és stop blokknál nem férne bele 12 bájtba az adat, csak 13-ba).

A start, és a step/stop blokkok struktúrája a cím-rész után eltér. Ahogyan az már említésre került egy programnak tartalmaznia kell mind a három blokktípust. Ez azt jelenti, hogy 10.810 blokkból legtöbb 3603 program alakítható ki. Mivel a memória nem csak egy, hanem sok különálló programot tartalmazhat, valahogyan azonosítanunk kell azokat. Erre szolgál a

start blokk soron következő 12 bitje. Mérete azért ennyi, hogy ábrázolni tudjuk a 3603-as számot is, azaz lehető legnagyobb programszámot. A 6-ik táblázatból láthatjuk, hogy a programszám alsó bitjeit a memóriában úgy tudtuk letárolni, hogy 4 bit szabadon marad. Ezen négy bit tárolja azt az információt, hogy a program magja (a step blokkok és a stop blokk) egymás után hányszor fusson le[7]. A startblokk soron következő 8 bájtja használaton kívül van, az egységes blokkméret miatt azonban ezek megléte szükséges. A tárterület vesztesége maximum 28Kbájt, azonban a későbbi fejlesztésekre ezen tárterület felhasználásával úgy van lehetőség, hogy nem kell módosítani a relatív-abszolút, abszolút-relatív címfordításon, továbbá a blokkok struktúráján.

A stop és step blokkok között a különbség a blokkazonosítón kívül a relatív címzésnél van, ugyanis míg a step blokkok a soron következő step/stop blokkot mutatják, addig a stop blokk visszamutat a legelső step blokkra. A megkülönböztetett szerepe a legutolsó lépésnek azonban lényeges, mivel a futtatórendszer tudtára adja, hogy az aktuálisan futtatott program elért a végére, miáltal számolni tudjuk azt, hogy az adott program hányszor futott le. A következő két bájtban a késleltetési információ kerül letárolásra, majd ezt követően 8 darab 8 biten ábrázolt PWM érték. Ennél a robotnál természetesen a nyolc csatornából csak hatot, azaz hat bájtot használunk. Minden egyes csatorna/bájt egy-egy motor elfordulását tárolja. A késleltetési információ adja meg azt, hogy a futtatórendszer hány ms-ot várjon mielőtt a blokkban tárolt PWM információkat értékül adná a motoroknak. Ha nem várnánk az értékek alkalmazása előtt, akkor a motoroknak nem lenne idejük beállni az adott pozícióra. A késleltetés 0...65535ms között változhat, azaz egy blokkon a végrehajtás maximum egy percig állhat. Ez természetesen nem azt jelenti, hogy a robotkar csak egy percig tud a pozíciójában maradni, ugyanis felvehetünk egymás után több egy perces késleltetésű, azonos tartalmú blokkot is.

A memóriában a blokkok meghatározott struktúra szerint helyezkednek el, mégpedig úgy, hogy a startblokkok a memória elejétől kezdődően **folytonosan** töltik fel azt, míg a stop és step blokkok a blokk részleg végétől kezdődően visszafelé tárolódnak. A folytonos start-blokk feltöltés maga után vonja azt is, hogy egyetlen egy step és stop blokk sem előzheti meg a start blokkok bármelyikét. Ez a megkötés azonban nem vonatkozik a step és stop blokkokra, ők a memóriában akárhol, és akárhogy elhelyezkedhetnek. A startblokkokra a konvenció azért szükséges, hogy mindig közvetlenül elérhetőek legyenek. A program keresését és indítását is

7 : ha ez az érték 0xF, akkor a program végtelenítve fut

megkönnyíti ez a stuktúra, mivel elég csak a keresett program számát olvasni, és ha az nem egyezik, a címet 12 bájttal arrébb tolva, a következő startblokk programszámát olvashatom. Ahogyan azt feljebb említettem a BAT-részleg 10.806 blokkot tud megcímezni, ez azonban 6-al több, mint amennyi ténylegesen elfér a memóriában. A nem használt plusz hat bit a BAT tábla utolsó bájtjának 0...6-ik bitjéig terjed (30-ik ábra).

3.1.8.2 *A kód működése*

A modulban létrehoztam speciális, a vezérlést és a működést megkönnyítő dedikált vezetékeket, regisztereket. Ilyen a free_spaces, mely a szabad blokkok számát mondja meg, a program_number, mint programszám a start blokkok fejébe kerül, majd inkrementálódik, a start_block_absolute_end, és a SAD_block_absolute_end, melyek a start, illetve a stop, és step típusú blokkok végét mutatják a BAT részlegben. Természetesen ezen két mutató egy-egy a BAT részlegre mutató címet tartalmaz, azonban ne felejtkezzünk meg arról, hogy egy címen nyolc blokkról tárolunk információt, így szükséges még mindkét mutatóhoz egy-egy segédmutató, mely megmondja, hogy az adott címen melyik bitre hivatkozunk. Ezen két segédmutató a start_shift, és a SAD_shift. További vezetékek segítik azt, hogy a BAT részleg egy címén lévő bitjéhez milyen abszolút, illetve relatív cím tartozik. A címek kiszámolásakor tekintettel kellett lenni arra is, hogy a BAT részleg utolsó bájtjából csak két bit használatos. A modul kilenc parancsot tud értelmezni, melyeket a parancsértelmező modul dekódol (3.1.4), és rak rá a belső buszra, melyhez ezen modul is csatlakozik. Minden egyes parancs végrehajtása előtt a modul megvizsgálja, hogy épp nincs-e valamely korábbi parancs végrehajtás alatt.

3.1.8.2.1 Reset parancs (0x0C)

A parancs hatására a modul a segédváltozókat alapállapotba állítja (a szabad blokkok számlálója a maximum, azaz 10.810-es értéket veszi fel, a programszámláló a 0-s kezdőértékre áll be, és a mutatók, valamint a hozzájuk tartozó segédmutató is a kezdeti értékét veszi fel. A startblokk mutató a BAT elejét, a stop és step pedig a BAT végét címzi a megfelelő eltolással.). Következő lépésben a memória vezérlővonalát írási módra állítja be, azaz $\overline{WE}$=0, $\overline{OE}$=1, majd a BAT részleg összes címén végighaladva, feltölti a memória ezen részét nullákkal, elérve ezzel azt, hogy a blokk-részleg szabadnak látszik. Itt máris

nyilvánvalóvá válik a BAT előnye, ugyanis csupán 1352 címen kell végighaladni 131.072 helyett, ami 84,5µs-ig tart a 8192µs helyett. Így a memória törlése közel 100-ad annyi időt vesz igénybe, mint egyébként. A memória törlése előtt a modul a soros porton felfüggeszti a kommunikációt, hiszen hiába érkezik be bármilyen parancs is, azt nem tudja lekezelni, mivel épp törli a memóriát. Miután a törlés befejeződött, a rutin feloldja a blokkolást, és jelzi, hogy elkészült feladatával; a memória alapállapotba került.

3.1.8.2.2 Inicializáló parancs (0x00)

Előfordulhat az az eset, hogy a vezérlőprogram egy olyan eszközhöz kapcsolódik, melynek memóriája már tartalmaz programokat. Ebben az esetben nem szerencsés kitörölni azokat induláskor. Mivel a vezérlőprogram, és az FPGA teljes szinkronban van, a vezérlőprogramot értesíteni kell a memória tartalmáról. A 0x00-s parancs hatására ez történik. Ez a parancs ugyanaz, mint amit a 3.1.7 -pontban tárgyalt STANDLER_HANDLER modul kezel. Elsőként a memóriában tárolt programok szinkronizációja, majd ezt követően a csatornák állása, és a kijelzési információk frissülnek.

Az inicializálás menete, hogy a startblokkokon, és azok programjain végighaladunk, és elküldjük az azokban tárolt információt a vezérlőprogramnak. Első lépésként a memória címét beállítjuk a BAT részleg kezdőcímére, ugyanis ha vannak startblokkok, akkor azok biztosan itt helyezkednek el, továbbá a memóriát olvasásra állítjuk ($\overline{WE}$=1, $\overline{OE}$=0), és blokkoljuk a kommunikációt a soros porton. Ez az inicializálási állapot. Ha ez a rutin lefutott, áttérhetünk a programok küldésére. A modul ezen része két egymásba-ágyazott állapotgépet rejt magában. Az első egy két állapotú, melynek első állapotában a BAT részleget vizsgáljuk és kódoljuk vissza a start blokkok abszolút címét, második állapotában pedig a blokkrészleget olvassuk. BAT módban a memória aktuális címén lévő adatokat vagy-kapcsolatba hozva eldönthető, hogy van-e az adott címen foglalt start blokk (emiatt fontos, hogy a startblokkok egymás után folytonosan helyezkedjenek el). Ha a vagy-kapcsolat értéke hamis, akkor nincsen több start-blokk, azaz az inicializálás befejeződött, oldjuk fel a soros port blokkolását, és adjuk vissza azt, valamint jelezzük, a STATE_HANDLER modulnak, hogy a memória inicializálása befejeződött[8]. Ellenkező esetben meg kell vizsgálni a cím minden bitjét. Ha a soron következő nulla, akkor a fent említett eljárást kell újból végrehajtani. Ha nem, akkor az adott bitet

8 : A STATE_HANDLER modul csak ezen jelzés megléte után kezdi el a már tárgyalt inicializálási rutinját.

abszolút címre dekódolva[9] a rutin átáll blokk üzemmódra. A program e helyen nézi azt is, hogy a BAT adott címén melyik bitet vizsgálta, ha ez az utolsó volt, akkor átállítja a memória BAT mutatóját a következő címre, ezáltal amikor megint ebbe az üzemmódba kerül át, új adatokkal dolgozhat.

Blokk részlegbe kerüléskor rendelkezünk egy olyan abszolút címmel, ami a BAT részleg vizsgált bitjének megfelelő blokk kezdőcímét adja. Ha ezen a kezdőcímen nem startblokk szerepel, akkor az inicializálás szintén befejeződött, ugyanis ez az eset az, amikor a memória tele van, és a startszekció mellett közvetlenül már nem start típusú blokkok tárolódnak, a modul pedig már az összes tárolt programot végigvizsgálta. Ellenkező esetben a második állapotkép kezdi meg működését, melynek állapotai a blokk aktuálisan beolvasott bájtjainak felelnek meg, így 12 állapottal rendelkezik. Minden egyes állapot végén a modul lépteti az olvasni kívánt memória címét.

1. Az első állapotban elkérjük a soros portot, és ha start blokkot olvastunk, a küldendő adat vonalára 0x15-ös számot rakjuk, mely arról informálja a vezérlőprogramot, hogy ezt a parancsot a program száma, és ismétlési információk fogják követni a 9-ik táblázatnak megfelelő struktúrában. Ellenkező esetben az előbbi lépés kimarad és egyből a 0x16-os parancs kerül elküldésre, amit 2 bájt késleltetési, és 8×1 bájt csatornaadat követ (8-ik tábla).

   Ebben az állapotban olvassuk be továbbá a következő blokk relatív címének első hat bitjét is.

2. Ebben az állapotban fejezzük be a következő blokk relatív címének beolvasását. Itt jegyezném meg, hogy nem kell külön lekezelni azt, hogy milyen blokkot olvasunk be, mert az összes blokk első két bájtjának struktúrája - pont ezen megfontolásból - azonos.

3. A harmadik bájttól kezdődően a blokktípusokat külön kell kezelni. Startblokk esetén elkezdjük beolvasni a program számát, míg stop és step blokkoknál megvárjuk amíg a megfelelő modul elküldi a parancsot a vezérlőprogramnak (Az állapotgép idáig 3 órajelciklus alatt jutott el, azonban a parancs elküldése legjobb esetben is több mint 1300 órajelciklusba telik). Amíg ez be nem következik, ebben az állapotban

9 : Ezen cím lesz a bit által reprezentált blokk kezdőcíme.

maradunk. Ha a parancs továbbításra került, küldendő adatként a memória tartalmát adjuk meg, azaz a késleltetési információ első bájtját.

4. Ebben az állapotban is megkülönböztetjük a blokkokat típusuk szerint. Startnál beolvassuk a programszám maradékát és az ismétlési információkat. Mindaddig ebben az állapotában marad a rutin amíg elküldésre nem került a korábbi adat. Ha ez megtörtént startblokk esetében elküldjük a programszám felső nyolc bitjét, stop és step blokknál pedig a késleltetési információ utolsó bájtját.
5. Ebben az állapotban szabad port esetén startblokkoknál elküldjük a programszám maradékát, és az ismétlési információt, majd nullázzuk az állapotgépet. A memória címét a beolvasott relatív címből kapott abszolút címre, azaz a láncban következő blokk abszolút címére állítjuk át. Step és stop blokkok esetében elküldésre kerül az első csatorna kitöltési tényezője.
6. A 6-tól a 11-ik állapotig csak akkor juthatunk el, ha step, vagy stop blokkot olvastunk. Az összes állapotban ugyanazt csináljuk; szabad portra várunk, és elküldjük a megfelelő csatorna kitöltési értékét.
7. 12 ik állapotban elválik egymástól a step és a stop blokk, mivel ha ez utóbbit olvastuk be, akkor elértük a program végét. Ilyenkor elküldjük az utolsó csatorna kitöltési értékét és nullázzuk az állapotgépet. Memóriacímként a BAT mutatót állítjuk be, és az üzemmódot ennek megfelelően BAT olvasásra váltjuk.

   Step blokknál szintén elküldjük a csatornaadatot, azonban memóriacímként - akárcsak start blokknál - a következő blokk abszolút címét adjuk meg, majd nullázzuk az állapotgépet, mindezt úgy, hogy blokkolvasási módban maradunk.

3.1.8.2.3 Új program létrehozása a memóriába (0x0D)

Akárcsak a legtöbb parancs műveleteit, ezt is megelőzi egy inicializálási fázis. Ebben az állapotban a rutin megvizsgálja, hogy van-e a memóriában elég szabad hely az új program létrehozásához (ez 3 szabad blokkot jelent). Ha nincs, arról értesíti a vezérlőprogramot, ellenkező esetben olvasásra állítja a memóriát, címnek pedig a BAT részleg elejét adjuk meg.

Az új program létrehozása a létrehozott blokkoknak megfelelően 3 fázisból áll, mely sorrendben start, step, és stop fázis. Minden fázis két részre osztható. Első részben a rutin meghatározza a BAT-ban az írandó blokkot (a fejezet elején tárgyalt mutatók, és segédmutatók segítségével), és annak bitjét egyesre cseréli, majd a memória címét ezen bit által reprezentált blokkra állítja. A mutató, és segédmutatójának értéke a műveletek végeztével aktualizálódik. Ez a rész minden fázisban hasonló, csupán a blokkoknak megfelelő mutatókat kell használni.

A második rész blokkonként némileg eltér. Az első két bit tartalma attól függ, hogy éppen milyen blokktípussal dolgozik a rutin. A következő hat bit a láncban következő blokk relatív címének első hat bitje. Ezt az információt a SAD_block_relative_end vezeték tartalmazza, mely folyamatosan biztosítja a SAD_block_absolute_end mutatóból és ennek eltolásából a megfelelő relatív címet. Természetesen stop fázisban nem használható ez a cím, mivel rossz helyre mutat, helyette a start fázisban (a start és stop blokkok mutatói mindig ugyanarra a blokkra, az első step blokkra mutatnak) lementésre került első step blokk címét használjuk. A második bájt tartalma a láncban következő blokk címének második része. A harmadik-negyedik bájt tartalma start blokk esetén a program száma, melyet a fejezet elején tárgyalt regiszterből nyer (majd utána inkrementálja azt), és alapértelmezetten az ismétlési érték (0). Start fázis esetén a rutin be is fejezi működését, és átlép step fázisra (fölösleges a startblokk hátramaradt nyolc bájtját feltölteni, mert azok tartalma irreleváns). Stop és step blokk esetén ebbe a két bájtba (3-ik, és 4-ik) kerül beírásra az alapértelmezett késleltetési érték (2000ms, azaz 0x07D0). Az 5-ik...12-ik bájt tartalmazza az alapértelmezett kitöltési tényezőket, mely 50%-os kitöltöttséget, azaz 0x80 értéket jelent. A kódban az alapértelmezett értékek, mint paraméterek szerepelnek, így elég megváltoztatni ezek értékét egy helyen; újrafordítás után máris az új értékek kerülnek alkalmazásra.

A harmadik, stop fázis befejeztével jelezzük a munka végeztét, és feloldjuk a blokkolást a soros portról.

3.1.8.2.4 Lépés beszúrása (0x11)

Lehetőség van bármely lépés elé, illetve mögé beszúrni újabb lépéseket. Mivel láncolt listáról van szó, ez alapesetben a blokkokban tárolt címek átírásával jár. Ha viszont utolsó elemet szeretnénk beszúrni, a blokkok típusát is módosítanunk kell. A beszúrás helyének függvényében az alábbi három eset áll fent:

1. Az első step blokk elé szeretnénk beszúrni egy új elemet. Teendők:
    - menedzselni a BAT-ban az új elemet
    - beállítani a blokk típusát step-re, valamint feltölteni az alapértelmezett értékekkel
    - start blokk mutatóját átállítani az új elemre
    - az új elem mutatóját beállítani a régi első elem címére
2. Step blokkok közé szeretnénk elemet beszúrni. Teendők:
    - menedzselni a BAT-ban az új elemet
    - beállítani a blokk típusát step-re, valamint feltölteni az alapértelmezett értékekkel
    - beállítani az új elemet megelőző step blokk mutatóját az új elemre
    - beállítani az új elem mutatóját az elemet követő blokk címére
3. Utolsó elemnek szeretnénk beszúrni. Teendők:
    - menedzselni a BAT-ban az új elemet
    - beállítani a blokk típusát stop-ra, valamint feltölteni az alapértelmezett értékekkel
    - beállítani a stop blokk típusát stepre, és a mutatóját átállítani az új elemre
    - beállítani az új elem mutatóját a start blokkra

Az új elem helyileg, mindig a blokk részleg végére kerül allokálásra. A beszúrási művelet elvégzéséhez szükségünk van azon program számára amibe be szeretnénk szúrni az új elemet, valamint az új elem pozíciójára. Ezt az információt a vezérlőprogram küldi el az eszköz számára 10-ik táblázatnak megfelelő struktúrában. Az értékek nem kerülnek elküldésre,

ugyanis az új elem mindig az alapértelmezett késleltetési, valamint csatornaértékekkel rendelkezik.

Mikor megérkezett a beszúrási parancs, a modul elvégzi az inicializálási funkciókat (állapotgépek változóinak kinullázása, memóriamódot olvasásra állítja, memóriacímet a BAT-részleg stop/step szekciójának végére állítja). Ebben a fázisban vizsgáljuk meg azt is, hogy van-e szabad hely a blokk beszúrására. Ha minden rendben van, egy olyan állapotba kerülünk, ahol várjuk, hogy megérkezzen a programszám. Mivel ez 12 bájtos információ, csak két soros átviteli ciklusban lehet továbbítani. Ha megérkezett, akkor a BAT részlegben lefoglaljuk az új blokknak a helyét, és a memória címét átállítjuk ezen blokk kezdőcímének kettővel növelt értékére. Így a mutatónk az új blokk késleltetési részére mutat. Ezután annak késleltetési és adat részét feltöltjük az alapértelmezett értékekkel. Egy segédregiszterbe elmentjük a blokk relatív címét is, hogy azt később csak bemásolhassuk a megfelelő blokk mutató-részlegébe. Ezután, a memória módját átállítjuk olvasásra, és a startblokkokon végighaladva megkeressük azon programot, melybe be szeretnék szúrni az új elemet. Ezt a startszekció elejétől a végéig haladva tesszük meg úgy, hogy a startblokkban tárolt programszám felső nyolc bitjét (startblokk 3-ik bájtja) összehasonlítjuk a fogadott programszám felső nyolc bitjével. Ha ezek megegyeznek, akkor beolvassuk a programszám maradék részét is, máskülönben léptetjük 12-vel a memória címét. (egy blokk 12 bájt, ezáltal pontosan a következő startblokk programszámát fogjuk olvasni). Ha programszám maradék része nem egyezik, akkor 11-gyel kell léptetni a memória címét, hogy az a következő blokk programszámjára mutasson. Azzal a módszerrel, hogy nem az egész programszámot hasonlítjuk össze, hanem annak csak egy részét, jelentősen lerövidíthetjük a keresést. Így nem kell minden egyes blokknál 2 bájtot beolvasni, hanem sok esetben elég egyet is, hogy eldöntsük nem a keresett program fejét olvassuk. A keresés legrosszabb esetben is alig haladja meg, a beszúrás pozíciójának megérkeztét. Ennek részletes magyarázatát lásd a 6.2 fejezetben.

Miután megtaláltuk a keresett programot, egy másik állapotba térünk át, melynek beindulási feltétele, hogy fogadtuk már a beszúrandó elem pozícióját. (Itt jegyezném meg, hogy a blokk kezdőértékekkel történő feltöltése, és a program keresése a pozíció fogadásával teljesen párhuzamosan zajlik, továbbá, hogy csak és kizárólag olyan programszámot kereshetünk, amely létezik is. Ellenkező esetben a modul működése kiszámíthatatlan, azonban erre

normális körülmények között nincsen lehetőség, ugyanis a program számozása a felhasználó elől rejtve történik, továbbá az FPGA-n futó szoftver, és a vezérlőprogram tökéletes szinkronban van egymással, vagyis a felhasználó a vezérlőprogram felületén nem tud olyan programra hivatkozni, mely nem létezik.)

Miután a modul fogadta a beszúrás pozícióját, (ez az érték 14 bites, mivel a 10.810 lépés csak ilyen bitszélességen fér el) megkeresi az azt megelőző elemet. Ezt úgy éri el, hogy az aktuális blokk (első esetben ez a start blokk) címrészlegét beolvassa majd átugrik a következő elemre, miközben egy segédszámláló értékét növeli. Ezt a folyamatot mindaddig ismétli, amíg a segédszámláló értéke el nem éri a beszúrás pozícióját. Így minden esetben a láncban a beszúrandó elem előtti blokknál állunk meg. Ezzel a megoldással a keresést nem kell külön bontanunk annak függvényében, hogy hova történik a beszúrás, mert ha az első elem elé szeretnénk bővíteni, az aktív blokk a start-, ha az utolsó mögé, akkor pedig a stopblokk lesz. Mivel a blokkok első két bájtja azonos, így a beszúrásnál csak az utolsó elemként történőt kell megkülönböztetni, és azt is csupán a blokk típus indikálása végett.

Miután tehát megtaláltuk az elem pozícióját kimentjük az aktuális blokkban tárol relatív címet, és ezzel egyidőben lecseréljük azt a beszúrandó blokk címére. Ha az aktuális blokk stop típusú, lecseréljük annak típusát step-re, mivel a beszúrást követően nem ez, hanem az új blokk lesz az utolsó elem a láncban. Következő lépésben átugrunk a beszúrandó blokk kezdőcímére, és beállítjuk annak címrészlegét a korábban kimentett címre, azaz az új blokk a lánc további részére fog mutatni. Ezzel egy időben beállításra kerül a blokk típusa step-re vagy stop-ra annak megfelelően, hogy a lánc melyik elemeként reprezentálódik. Mivel az új blokk késleltetési és adatrésze már korábban beállításra került, a soros port visszaadásával, és a kommunikáció blokkolásának feloldásával be is fejeződött az új elem beszúrása.

3.1.8.2.5 Programindítási parancs (0x19), és a program futása

Ahhoz, hogy egy adott program futása elkezdődjön ismernünk kell a program számát, és a futtatás kezdőpozícióját. Ezen információt a vezérlőprogram küldi a 0x19-es parancsot követően, a 10-ik táblának megfelelő felépítésben. A program és a megfelelő blokk megkeresésének metódusa azonos a beszúrási parancsnál tárgyaltakéval. Miután megtaláltuk a megfelelő blokkot, a memória címét úgy módosítjuk, hogy az a blokk késleltetésére

mutasson (a kezdőcímhez hozzáadunk 2-t), majd a modul programfuttatási állapotba tér át. Ebben az állapotban a modul működésébe kívülről csak a futtatás leállítása (0x1A) paranccsal tudunk beleszólni.

Futtatási üzemmódban 3 szál működik kvázi párhuzamosan. Első szál a késleltetési értéket számolja, a második a blokk tartalmát olvassa, a harmadik pedig a soros porton kommunikál. Elsőként a második szál kezdi meg a működését beolvasva a késleltetési információkat. Ha ez megtörtént, az első szál is elkezd működni, azaz kivárja azt az értéket, amit a blokk tárolt. Ezzel párhuzamosan a második szál beolvassa a blokkban tárolt 8 csatorna kitöltési tényezőjét. A modulok közötti kapcsolatok redukálása érdekében nem 8 darab 8 bites buszt használtam a kitöltések továbbítására, hanem egy 8 és egy három biteset, melyek közül az első a kitöltési értéket, második pedig azon csatorna számát adja meg, melyhez az adott kitöltés tartozik (multiplexelés). Miután az időzítés letelt, a szál elkéri a soros portot, majd a beolvasott kitöltési tényezők közül a megfelelőt értékül adja az előzőekben tárgyalt kimenetnek. Ezzel egyidőben a kiválasztott csatorna száma is megjelenik. Az aktuális értékek, és a PWM vonalak összerendelését a 3.1.4 -es pontban tárgyalt modul végzi. A harmadik szál akkor működhet, ha a modul megkapta a soros portot és azon nem folyik kommunikáció. Így explicit az első szál portkérése váltja ki a harmadik működését. Ha a port rögtön szabad, akkor a csatornák értékadása és a kommunikáció párhuzamosan zajlik. A harmadik szál az új adatokat a vezérlőprogram számára a 9-ik, majd a 8-ik táblának meglelő struktúrában küldi el. Nyilvánvaló, hogy a késleltetési idő alatt a vezérlőprogram még a régi értéketeket mutatja.

Mikor a második szál elérte a blokk végét, azaz az utolsó kitöltési tényezőt is beolvasta, a memóriacímet visszaállítja a blokk kezdőcímére, hogy a blokk típusát, és a következő blokk címét beolvashassa. Miután ezt megtette, bevárja míg a harmadik szál elküldi a vezérlőprogramnak az utolsó csatorna kitöltését is (szinkronizációs pont). Ha ez nem így lenne, a második szál idő előtt léptetné a program futásának számlálóját (azaz mikor elértük a stop blokkot), és a harmadik szál nem az aktuális adatokkal dolgozna, ami téves működéshez vezetne. A harmadik szál ebben az állapotban vizsgálja meg, hogy beérkezett-e futás közben stop parancs, avagy elértük-e megfelelő számúszor a program végét. Ha valamelyik feltétel igaz, akkor leállítjuk a program futását, azaz visszaállítjuk a modult normál módba, és a vezérlőprogramot informáljuk arról, hogy a programfutás megszakadt. Ellenkező esetben a

szál visszaáll alapállapotába, és visszaadja a soros portot, ezzel blokkolva saját maga további működését. A második szál utolsó állapotában lévő kód csak akkor fut le, amikor a harmadik szál már visszakerült alapállapotába (tehát nem történt olyan esemény amely megszakította a program futását). Ebben az esetben a memória címe a láncban következő blokk kezdőcímének kettővel növelt értéke, azaz a következő blokk késleltetési részlege lesz, majd a második szál is alaphelyzetbe kerül, és folytatódik a működés a következő blokkal.

3.1.8.2.6 Programfutást megszakító parancs (0x1A)

Programfutás alatt csak ez a parancs vált ki a modulban változást. Beérkeztekor egy regiszter értéket vált, azonban hatását a parancs csak akkor fejti ki, mikor a harmadik szál megvizsgálja ezt a regisztert (lásd előző pontban). Ezzel a megoldással egy megkezdődött folyamatot nem lehet félbeszakítani, azaz ha a parancs egy hosszú késleltetés alatt érkezett, a beolvasott blokk már mindenképpen kifejti hatását. Ez rendkívül hasznos, ugyanis így elkerülhető az aszinkron működés a vezérlőprogram, és az FPGA-n futó program között.

3.1.8.2.7 Adott blokk olvasása (0x0E)

A parancs struktúráját a 10-ik tábla tartalmazza. A programot a memóriában a 3.1.8.2.4 -es pontban tárgyalt módszer szerint keressük meg. Mikor ez megtörtént lementjük a program számát, és a startblokkban tárolt ismétlési adatokat. Miután beérkezett a keresett blokk pozíciója is, elküldjük a programszámot, és ismétlési adatokat a 9-ik táblázatban szereplő parancsfejjel, és struktúrával. Ezalatt a már említett módon megindul a beérkezett pozíció keresése. Ha ezt megleltük, elküldjük a blokkban tárolt adatokat az 8-ik táblának megfelelően. Az utolsó bájt átvitele után jelezi a kódrészlet, hogy befejezte a parancs feldolgozását, és visszaadja a soros portot.

3.1.8.2.8 Adott blokk írása (0x18)

Ezen parancs hatására a startblokk ismétlési adata, és az adott blokk késleltetése, valamint csatornaadatai íródnak felül. A parancs a 11-ik táblának megfelelő struktúrában érkezik meg. A keresési módszerek azonosak az előzőekben tárgyaltakkal. Miután megtaláltuk a megfelelő blokkot, és beérkezett a késleltetési adat is, lecseréljük azt. A

csatorna adatai is frissülnek, mégpedig a nyolc csatorna aktuális pozíciójára. Ezeket az adatokat nem szükséges elküldenie a vezérlőprogramnak, hiszen azok mindig naprakészek az eszközben (bármilyen változás történik a vezérlőprogramban, az azonnal elküldi az új adatokat az FPGA-nak). A modulok közötti kapcsolatok számának redukálása érdekében a bemeneti csatornaadat is multiplexeléssel férhető hozzá, azaz egy nyolc bit széles vezeték reprezentálja azon csatorna értékét, melyet a csatorna választó vezetékkel megjelöltünk. Az összerendelést a main.v modul végzi. A csatorna-választó vonallal végigpásztázva az összes csatornán, azok kitöltési értékei kerülnek rögzítésre a blokk megfelelő címére. Mikor ez megtörtént, a memóriát visszaállítjuk olvasási módba, nehogy véletlenül más adatot is felülírjunk a következő címmódosításkor, végül visszaadjuk a kommunikációs portot.

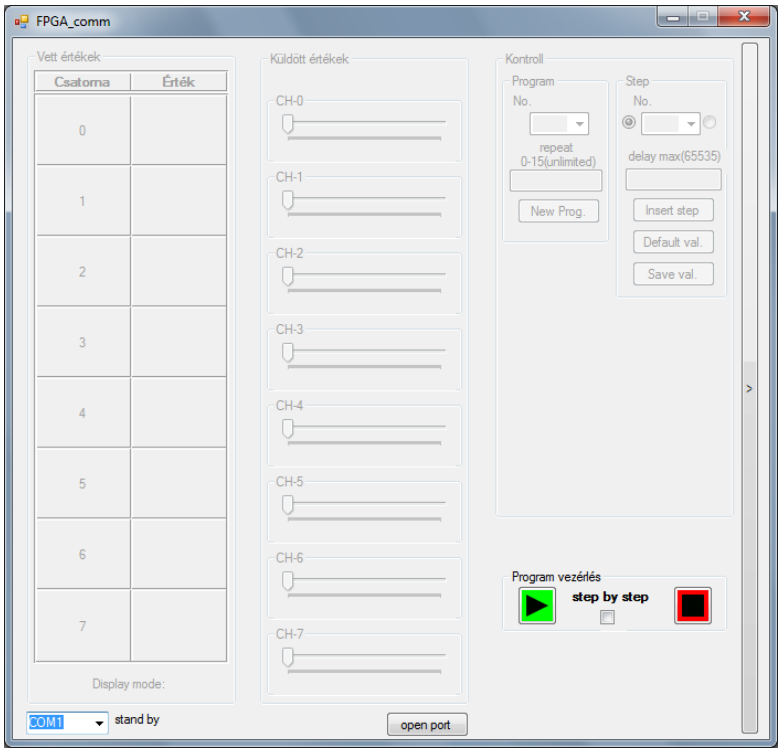
3.1.8.2.9 Direkt címet olvasó parancs (0xFE)

Ez a parancs a debugolás miatt került beépítésre, és rendkívül hasznosnak bizonyult a blokkszerkezet és a blokkok közötti kapcsolatok kialakításakor. A parancsot 3 bájt követi ami az olvasandó címet tartalmazza. Hatására az FPGA visszaküldi a címen található adatot. (Csak az adatot, parancs nélkül.)

3.2 *A vezérlőprogram*

A LOGSYS panel szűkölködik kezelő és kijelző eszközökben, ezért egy vezérlőprogram fejlesztése mellett döntöttem, melyen a legfontosabb információk egyszerre jelennek meg, és a kezelést is megkönnyíti. A program C# nyelven íródott Sharp Develop fejlesztői környezet segítségével. Ez egy ingyenesen letölthető, nyílt forráskódú szoftver. A vezérlőprogram írásakor a nyelv eszközrendszerét, mintsem az objektumorientált programozás előnyeit használtam fel. Mivel maga a vezérlőprogram nem kapcsolódik szorosan a szakdolgozat témájához, csupán a működése kerül ismertetésre.

A program futtatásához .NET 2.0-s keretrendszer szükséges. Induláskor a szoftver lekérdezi a soros portok listáját, és ha ez üres, arról egy hibaablakkal értesít minket, és a felület inaktívan töltődik be. Ellenkező esetben a következő képernyő fogad minket:



ábra 13: A vezérlőprogram inaktiv állapotban

A program 4+1 csoportra került felosztásra. A *Vett érték* csoportban láthatom azt, hogy az adott csatornához milyen értékek érkeztek legutoljára az FPGA-tól. Ezek az értékek mindig decimális formátumban kerülnek megjelenítésre, függetlenül attól, hogy az FPGA milyen kijelzési módban van. Működés közben a *Csatorna* oszlopban azon cella háttere vált vörösre, mely épp megjelenítésre kerül a panelen. Ezen csoport *Display mode:* utáni értéke informál minket arról is, hogy a panelen milyen kijelzési módot használunk.

A *Küldött érték* csoportban tudom beállítani azt, hogy a csatornákhoz milyen kitöltési tényező tartozzon, így ezen kontrollok értéke 0...255 között változhat. A csúszkák mozgatásával tudom pozicionálni a robotkar motorjait, így ezen kontrollok változtatásával tudjuk beállítani a robotot egy adott pozícióba.

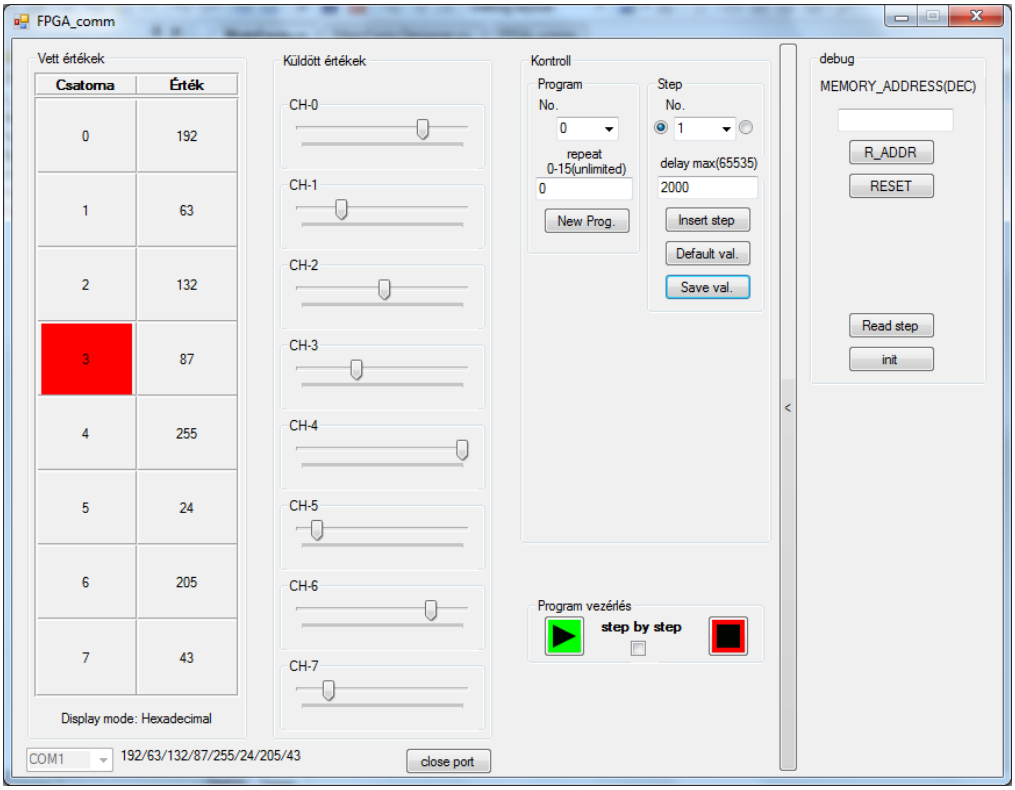
A *Kontroll* csoport elemeivel tudom a kart programozni. A *Program* alcsoportban a programszintű műveletekkel tudok operálni, úgymint aktív program kiválasztása, annak ismétlési információjának megjelenítése, és módosítása, valamint új program létrehozása. Ennek megfelelően a *Step* alcsoportban a lépésszintű műveletek érhetőek el. A Lépés számát

kiválasztó menü két oldalán található rádió-gombokkal tudom beállítani azt, hogy az *Insert step* feliratú gomb lenyomásakor az új lépés az aktuálisan kiválasztott lépés elé, vagy pedig mögé szúródjon be. Miután beállítottuk a motorok pozícióját a *Küldött érték* kontrolljai segítségével, a *Save val.* feliratú gomb lenyomásával kerülnek rögzítésre a kiválasztott program adott lépésébe/blokkjába. Mindeddig a módosítások csupán virtuálisak. Mivel a vezérlőprogram lekérdezi minden egyes lépés-váltáskor az újonnan kiválasztott lépés adatait, így ha a *Save val.* gomb lenyomása nélkül váltunk pozíciót, a módosítások elvesznek. Ebben az alcsoportban kerül megjelenítésre a blokkhoz tartozó késleltetési információ is. Ez az érték is módosítható, mely szintén csak akkor kerül letárolásra, ha az előzőekben említett gomb megnyomásra került. A *Default val.* gomb a felületen lévő kontrollokat állítja alaphelyzetükbe. Ezen alapértékek megegyeznek az FPGA-ban tároltakkal, azaz kitöltési tényezőknél 128, ismétlésnél 0, késleltetésnél 2000.

A programablak bal alsó sarkában elhelyezett legördülő menüből tudjuk kiválasztani, hogy melyik soros porthoz szeretnénk kapcsolódni az *open port* gomb lenyomásakor. Csatlakozáskor feltételezi a szoftver, hogy megfelelő portot választottunk ki, ha mégsem tettünk volna így, a vezérlőprogram működése leáll. A csatlakozás és szinkronizálás első fázisában a szoftver megkérdezi, hogy törölje-e a memória tartalmát. Ha erre a kérdésre igennel válaszolunk, akkor az inicializálási parancs elküldése előtt, egy memóriatörlési parancs kerül továbbításra. Erre a kérdésre két dolog miatt is szükség van. Az első, hogy egy olyan eszközhöz csatlakozunk, mely még nem kapcsolódott a vezérlőprogramhoz. Ebben az esetben az SRAM nem lett kinullázva, és mivel a tápfeszültséget is valószínűleg nemrég kapcsoltuk be, annak tartalma nem meghatározott, ami teljesen összezavarhatja az FPGA működését. A második eset, hogy egy olyan egységhez csatlakozunk, melynek memóriájában már található program. Ilyenkor a nem gombra kattintva megkezdődik az inicializáció. Befejeztével a kontrollok blokkolása feloldódik, és megkezdhetjük a robotkar programozását.

A *Programvezérlés* csoport gombjaival tudunk egy a *Kontroll* csoportban definiált programot az adott pontjától futtatni. Futás közben csak a stop-gomb engedélyezett (szinkronban a 3.1.8.2.5 -ös fejezetben tárgyaltakkal), mellyel le tudjuk állítani a program futását. A vezérlőprogram jobb oldalán helyet kapott egy hosszú gomb, mely aktiválja, illetve deaktiválja a *debug* felületet. Ezen a felületen tudunk lekérdezni adott memóriacím-tartalmat,

itt tudjuk direkt módon alaphelyzetbe állítani a memóriát, illetve az inicializálási funkciót meghívni. Az alábbi ábrán láthatjuk a vezérlőprogramot működés közben:



ábra 14: A vezérlőprogarm működés közben debug felülettel

4 Összefoglalás

Szakdolgozatom egy szervó motorokkal szerelt hobbi célokra alkalmas robotkar automatizált mozgatásáról, és programozásáról szól. A robotkart egy FPGA áramkör segítségével vezérlem, mely minden olyan feladatot ellát, ami hardver közeli szemléletet igényel. A vezérlés, és a programozás egy PC-n futó alkalmazás segítségével történik.

A robot az automatikus működéshez szükséges pozícióadatokat a LOGSYS kártyán lévő SRAM-ban tárolja. A memóriában 12 bájtos csoportok összefogásával blokkokat alakítottam ki. Ezen blokkok cirkulárisan láncolt listája egy vezérlőprogram. A mozgásprogram maga a listán történő végighaladás. A listaelemek tartalmaznak egy pozíciót (összesen nyolc motorállást), így amikor időzítve végighaladok azokon, a robotkar az elemekben tárolt pozíciókat felvéve automatikusan működik. Az elemek tartalmát a vezérlőprogram segítségével alakíthatom ki. A vezérlőprogram és az FPGA-n futó szoftver teljes szinkronban működik. Miután befejeztem a mozgásprogram kialakítását, és annak futását elindítottam, az a vezérlőprogram nélkül, standalone is elvégzi a feladatát, azaz a controllprogramnak csupán kezelőfelületi funkciója van, azonban ha azt használom, mindig a hardver állapotát tükrözi.

Munkám egy nagyobb projekt része, mely a NATIONAL INSTRUMENTS felkérésére készül. Jelenleg a szakdolgozatom tárgyát képező rész továbbá a konzolos vezérlés valósult meg, mellyel Tóthfalusi Tamás évfolyamtársam foglalkozott. Ha a projekt eléri a megfelelő fejlettségi szintet, valószínűsíthető, hogy az már nem csak a hobby-célú robotokon, hanem a gyakorlatban is alkalmazásra kerül. Úgy érzem sikerült elérni a kitűzött célomat, hiszen a kar teljesen automatizáltan is működni képes, annak programozása lényegesen egyszerűbbé vált. Habár mind az FPGA-n futó szoftveren, mind pedig a vezérlőprogramon van még mit javítani, a nyomdokaimba lépőknek megfelelő alapokat biztosít a további fejlesztésekhez, melyek között szerepel egy mozgásprogram egészének törlése, egy mozgáselem törlése, valamint program-program átjárhatóság kifejlesztése (egyik programból meg lehessen hívni egy másik programot). További terv a mozgásprogramok startblokkjának rendezése, hogy a programkeresés lerövidülhessen, valamint a mozgásprogramok lementése merevlemezre, vagy a LOGSYS kártyán található flash memóriába (ezen utóbbi eszköz megőrizni az adatot tápfeszültség nélkül is), hogy azok később visszatölthetőek legyenek.

5 Irodalomjegyzék

[1] Fehér Béla: Digitális rendszerek tervezése FPGA áramkörökkel, Budapest, Budapest Műszaki és Gazdaságtudományi Egyetem (2009, slides)

[2] Spartan 3E FPGA Family: Data Sheet, United States, Xilinx Inc. (2009, v3.8)

[3] Jameel Husein: Xilinx Total System Power Wbcast 09101202, https://admin.acrobat.com/_a700655680/xilinx09101202/ (2010.02.24)

[4] http://only-vlsi.blogspot.com/2008/04/setup-and-hold-time.html (2010.03.18)

[5] Dr. Végh János: Bevezetés a Verilog hardverleíró nyelvbe,Debrecen, egyetemi jegyzet (2008/2009 I félév)

[6] http://www.fpga4fun.com/SerialInterface1.html (2010.03.23)

[7] http://en.wikipedia.org/wiki/RS-232 (2010.03.23)

[8] Summary of RS232 and RS422 standards, http://www.evertz.com/resources/quartz/RS232-RS422-standards-summary.pdf (2010.03.23)

[9] http://www.taltech.com/TALtech_web/resources/intro-sc.html#parity (2010.03.23)

[10] http://logsys.mit.bme.hu/sites/default/files/page/2009/09/LOGSYS_SP3E_FPGA_Board.pdf

[11] http://en.wikipedia.org/wiki/Static_random_access_memory

[12] K6R1008V1D CMOS SRAM adatlap v1.0 2001 December 18.

6 Függelék

6.1 *Kettes komplemens ábrázolás:*

Egy n bites szám kettes komplemensben ábrázolva:

MSB előjelbit, 0, ha a szám pozitív, 1, ha negatív. A maradék n-1 biten ábrázoljuk a számot a szokásos módon. Ha a számunk pozitív, akkor készen vagyunk, ha nem, akkor bitenként negáljuk az n-1 bites vektort, és adjunk hozzá 1-et. Pl.: ábrázoljuk az 5-ös, valamint a -7-es számot és adjuk össze őket.

1. Az 5_{10} pozitív, így az előjelbit marad nulla. A maradék három biten ábrázolva a számot, így: $5_{10}=0101_2$
2. A -7 negatív, így az előjel bit 1. Azután ábrázoljuk a 7-et, ez 111. Ezt bitenként negáljuk (000), majd adjunk hozzá 1-et. Így a $-7_{10}=1001_2$
3. Adjuk össze:
   0101
   + 1001
   1110
4. Az eredmény a várakozásnak megfelelően mínusz maradt. Ennek megfelelően a maradék 3 bites számból kivonunk egyet (101) majd invertáljuk azt (010), így a végeredmény 1010, azaz -2, ellenőrizve (-7)+5=(-2)

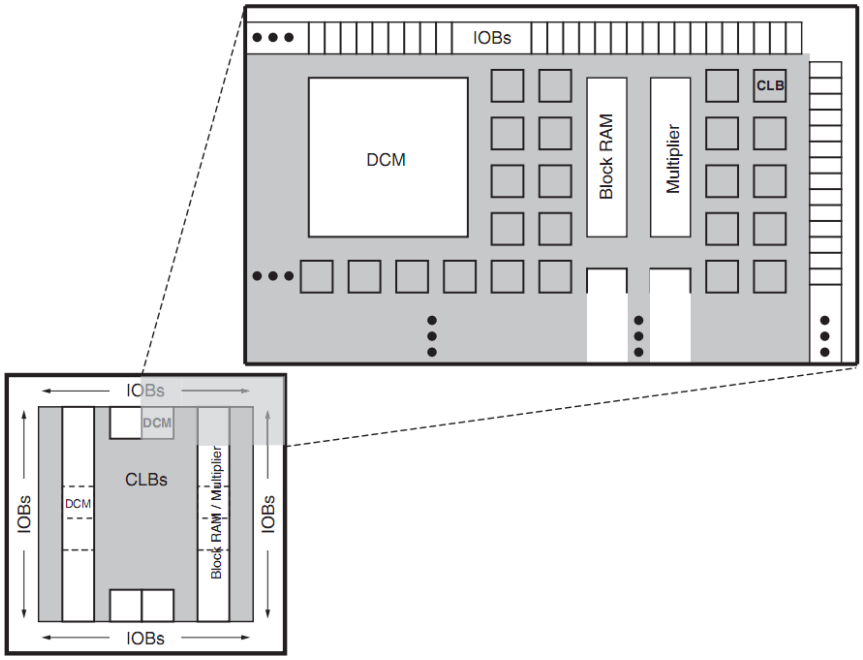
Negatív szám visszaalakításánál a kivonás-invertálás lépés ekvivalens az invertálás-összeadással. Ahogyan azt a példa is bizonyítja az ily módon ábrázolt számokkal végzett összeadás (és hasonlóan a többi alapművelet is) előjeles művelet.

6.2 *A legrosszabb keresési idő magyarázata*

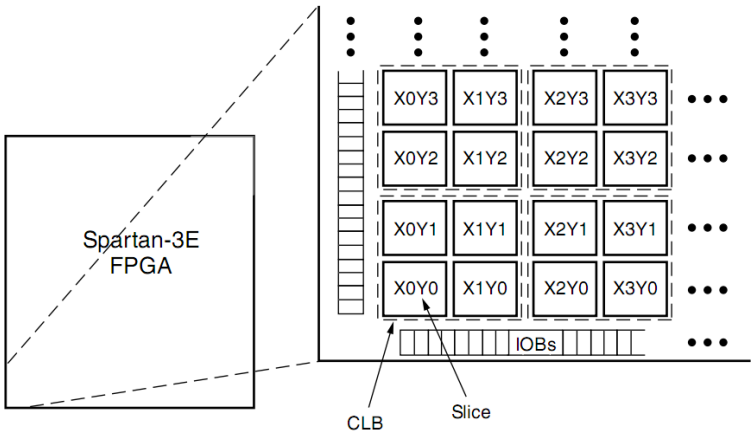
3603 program esetén marad egy szabad blokkunk amit beszúrhatunk, és ha az utolsó program végére szeretnénk ezt megtenni, 3687+32 összehasonlítást kell végezni a programszámokra (mivel a programszámok első nyolc bitje csak az utolsó 16 esetben egyezik, az előtte lévő startblokkokból csak a felső nyolc bit szükséges a vizsgálathoz. A maradék 16 esetben mind a két bájtot beolvassuk, és ha az csak az utolsó esetben egyezik meg a keresett programszámmal, akkor pontosan 3587+32 órajel szükséges a vizsgálathoz). Ehhez

hozzáadódik még az inicializáláshoz szükséges 13 órajelciklus. Ehhez 16Mhz-es órajellel számolva $\frac{3587+32+13}{16.000.000}=227\mu s$ szükséges. A lépésszám 3 bájtban kerül továbbításra, melyből egy már átvitelre került, így a lépésszám átvitele $\frac{2*10(startbit+8\text{db}\,adatbit+stopbit)}{115.200}=173{,}9\,\mu s$ -ig tart. Látható, hogy a keresés a legrosszabb esetben 1,3-szor annyi ideig tart, mint az átvitel maga. Ez azonban a legrosszabb, legkiforgatottabb lehetőség, a valóságban az esetek döntő többségében a keresés és inicializálás jóval hamarabb lefut, minthogy a pozíció adata beérkezne.

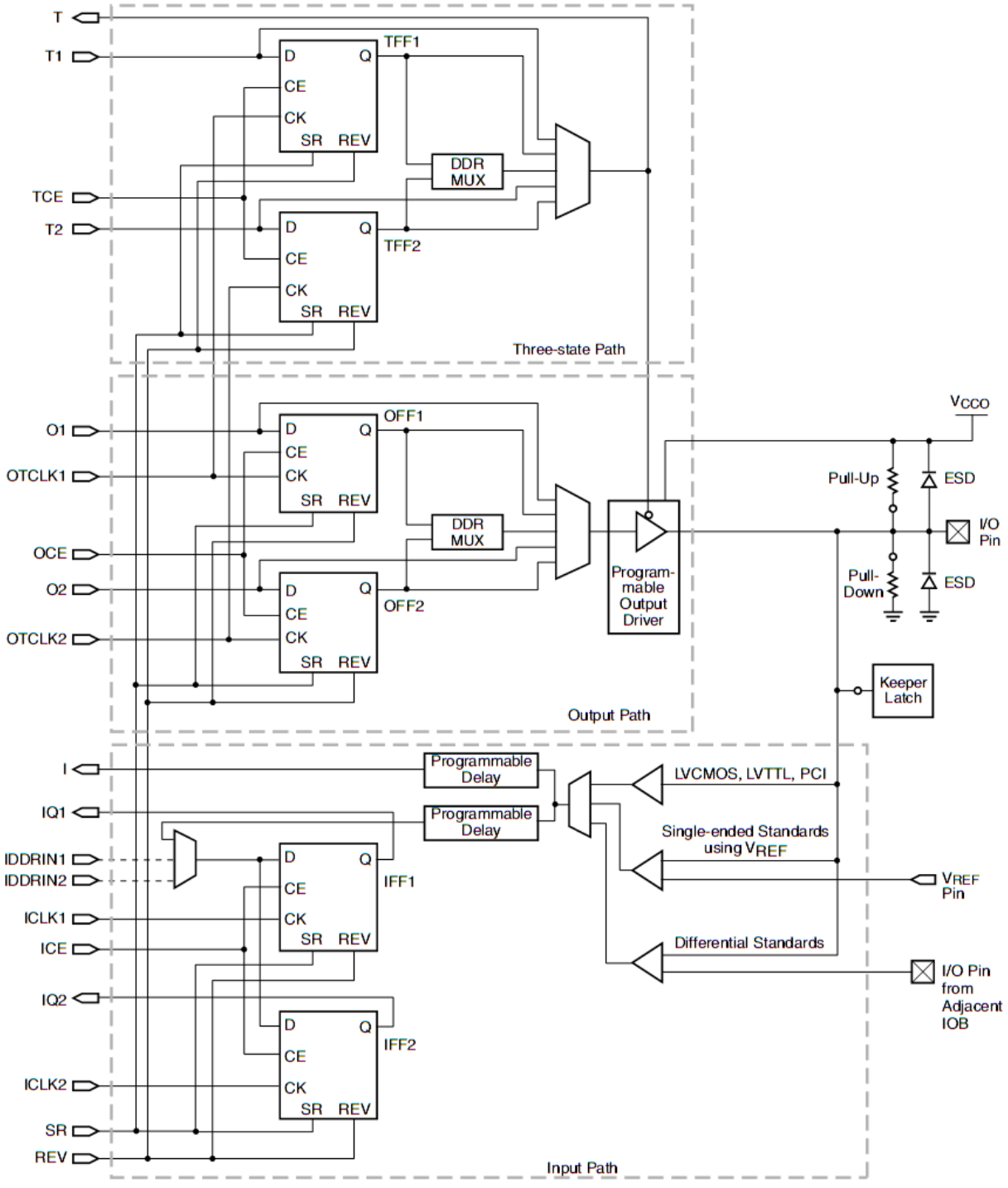
6.3 *Ábrák és táblázatok*



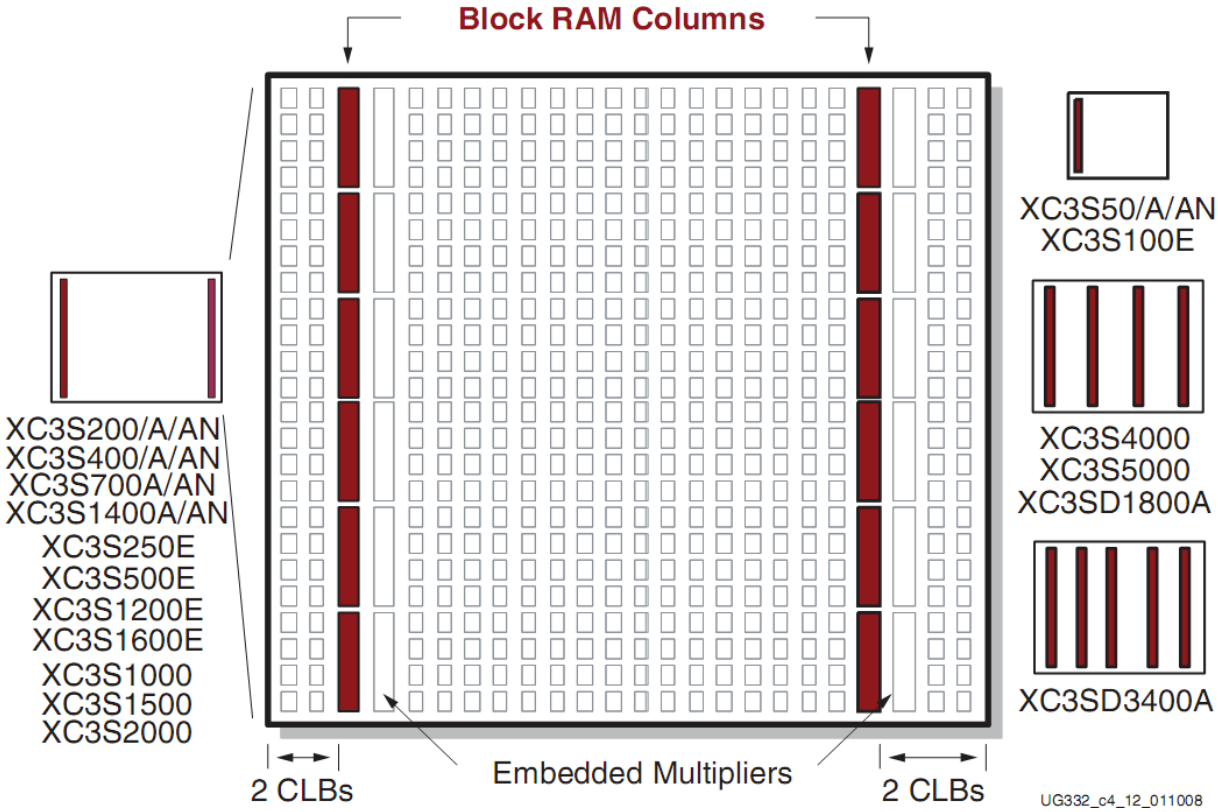
ábra 15: A Xilinx Spartan FPGA-k sematikus felépítése



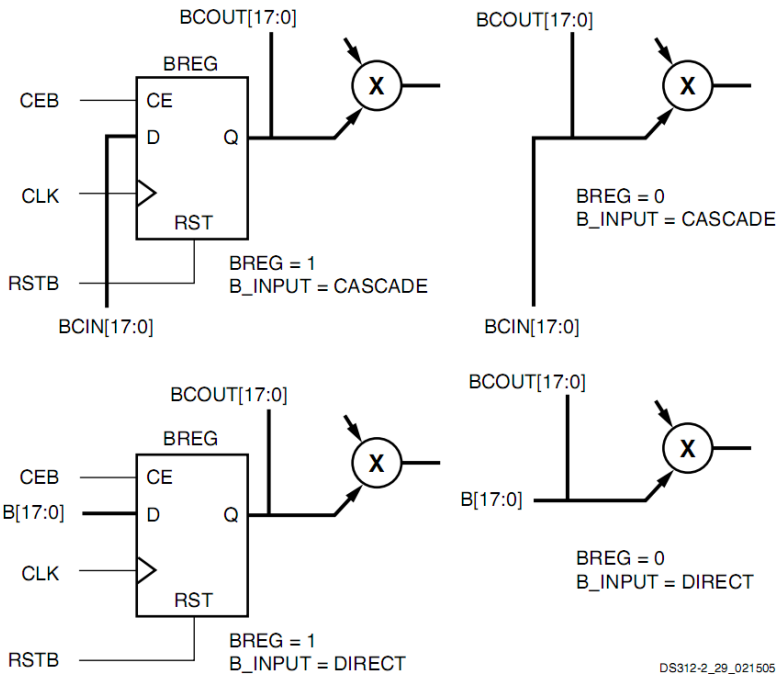
ábra 16: CLB-k és Slice-ok



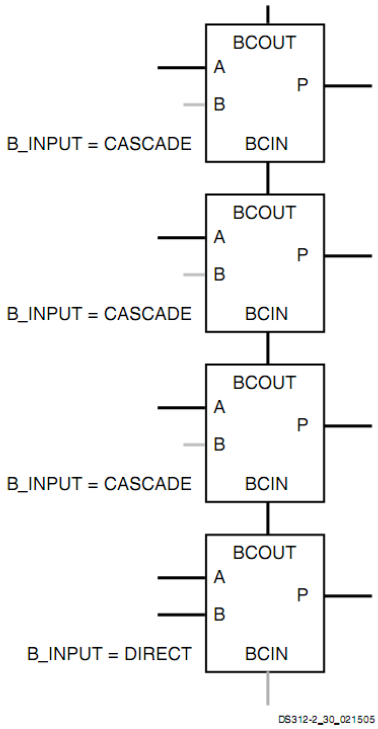
ábra 17: A 3 állapotú IOB-k felépítése



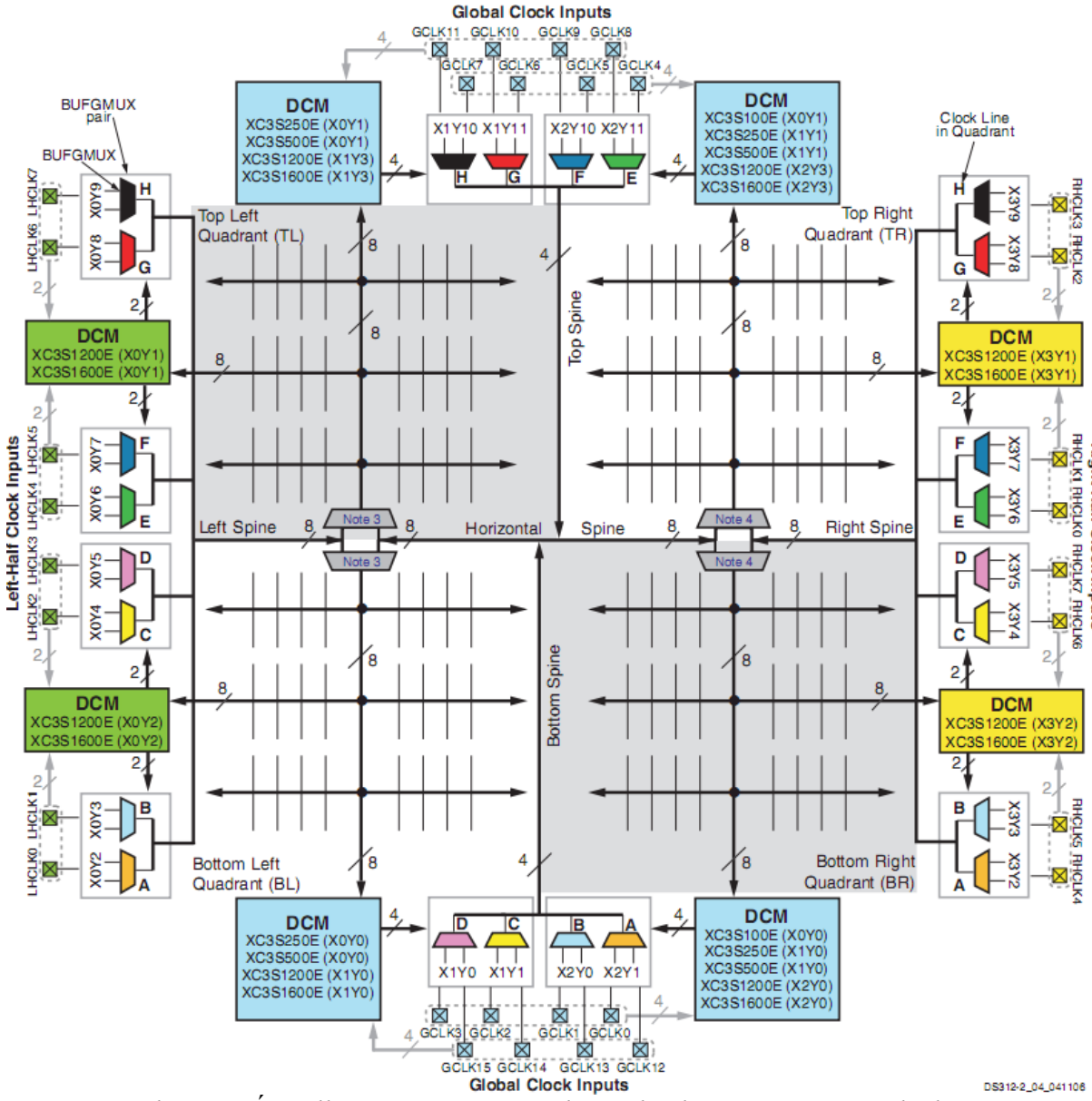
ábra 18: Blokk-ramok, és szorzóegységek elhelyezkedése



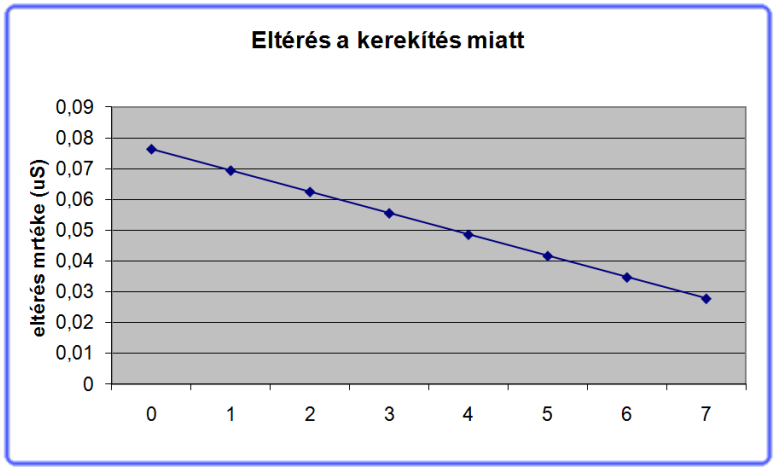
ábra 20: BC in és OUT vezetékek csatlakozási lehetőségei. Forrás: [2],38-ik ábra



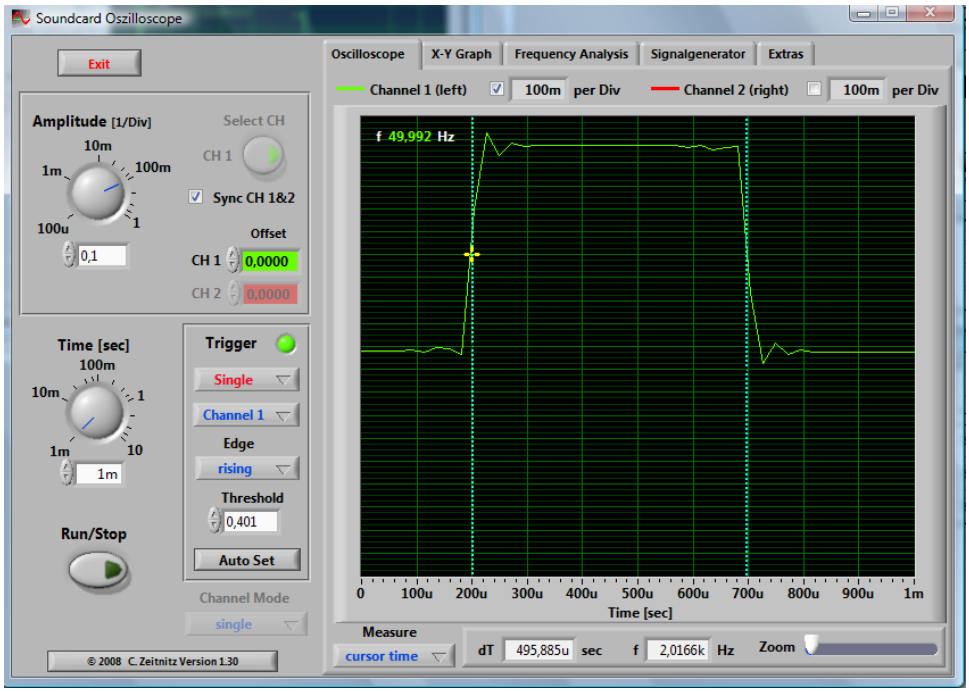
ábra 19: Szorzóegységek kaszkádosításának módszere. Forrás:[2], 39 ábra



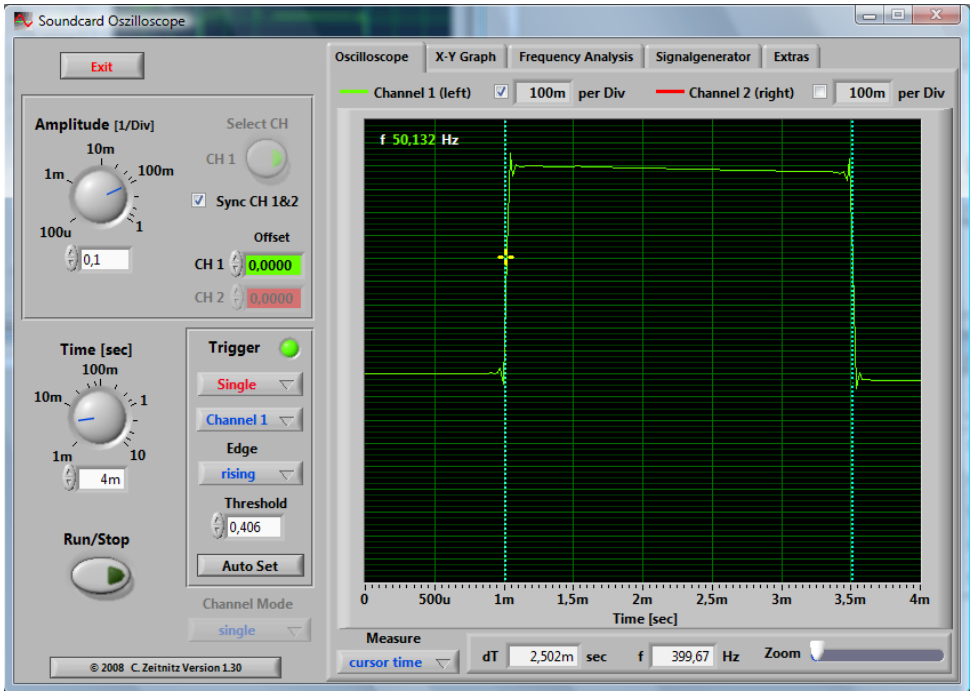
ábra 21: Órajelházózat a SPARTAN kártyáknál. Forrás: [2], 45-ik ábra



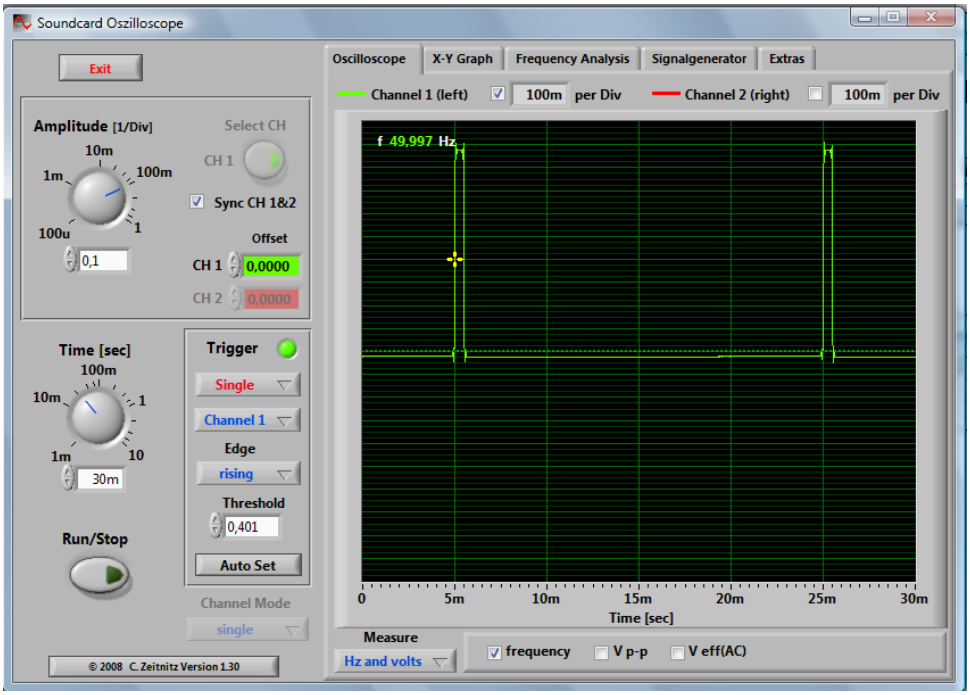
ábra 22: Mintavételezési hiba



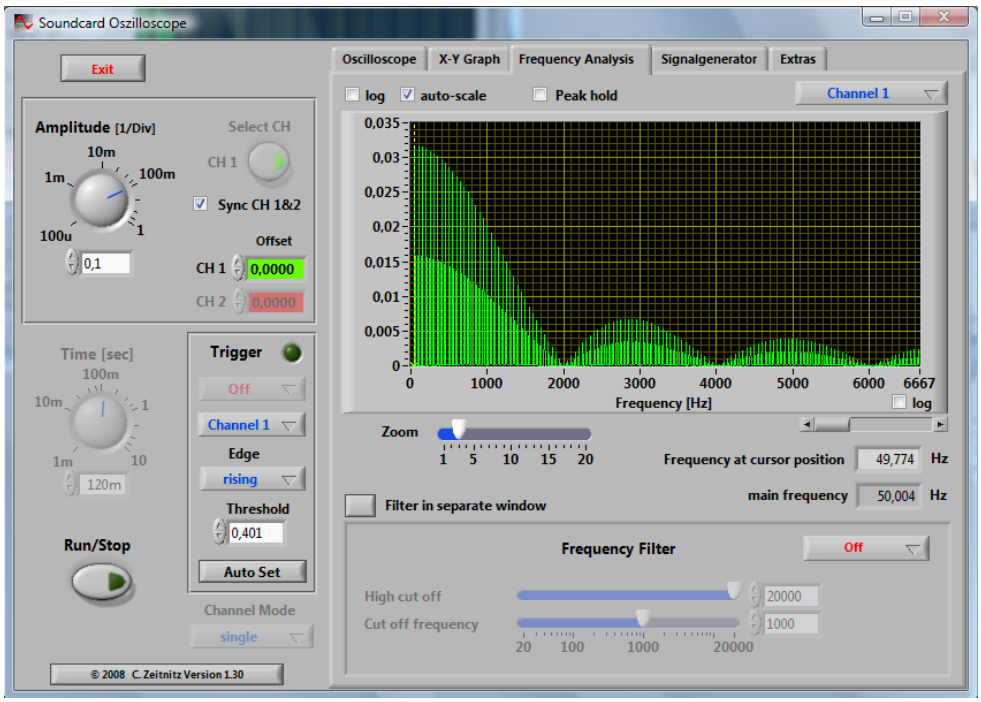
ábra 23: 500µs alapkitöltésű PWM jel



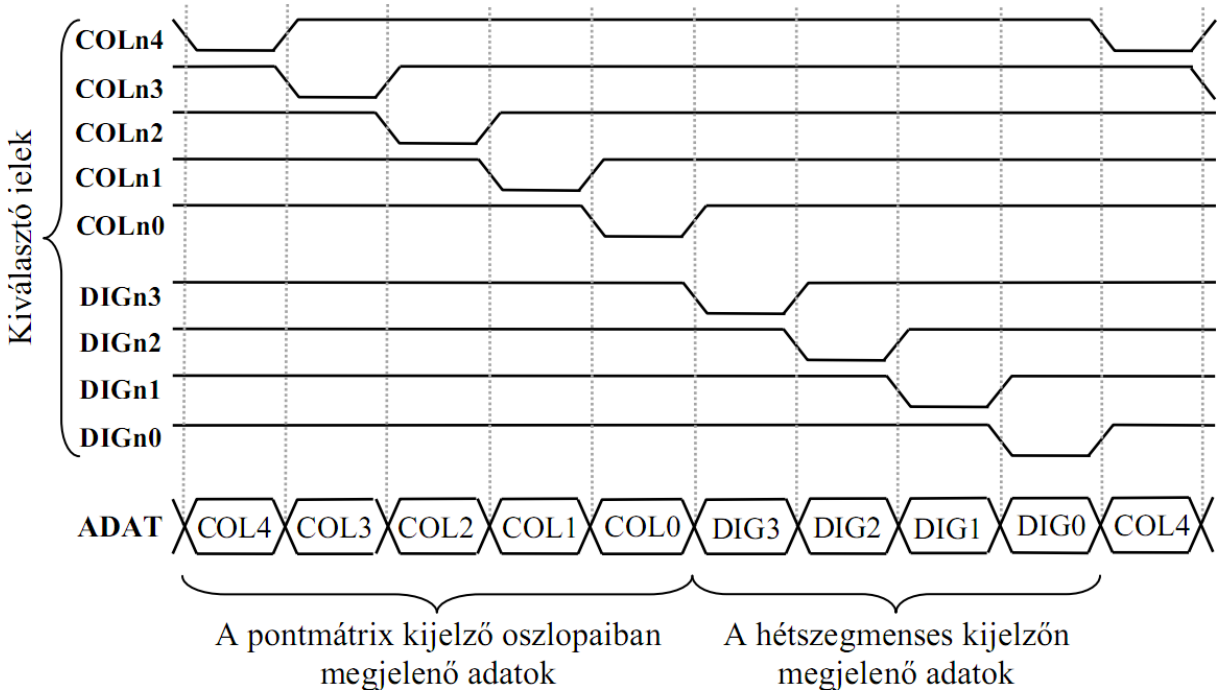
ábra 24: 2500µs-os teljes kitöltöttségű PWM jel



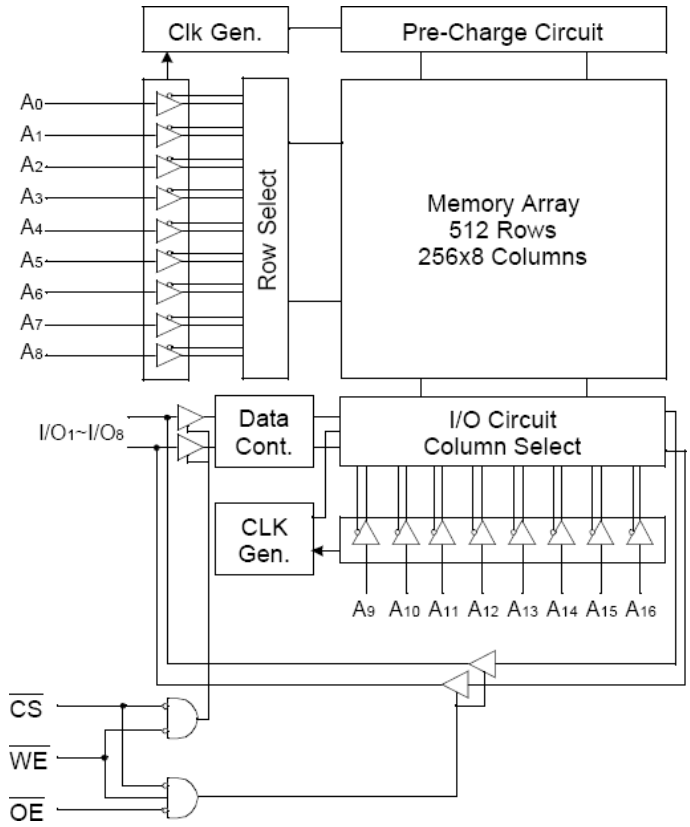
ábra 25: Egymás 500µs-os PWM jelek a kimeneten



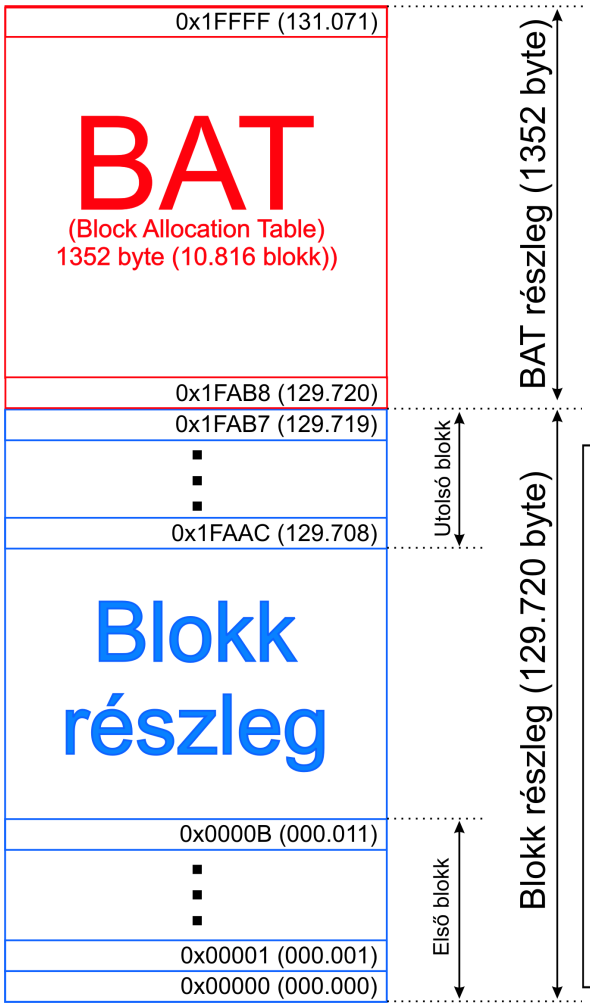
ábra 26: A PWM jel Fourier spektruma



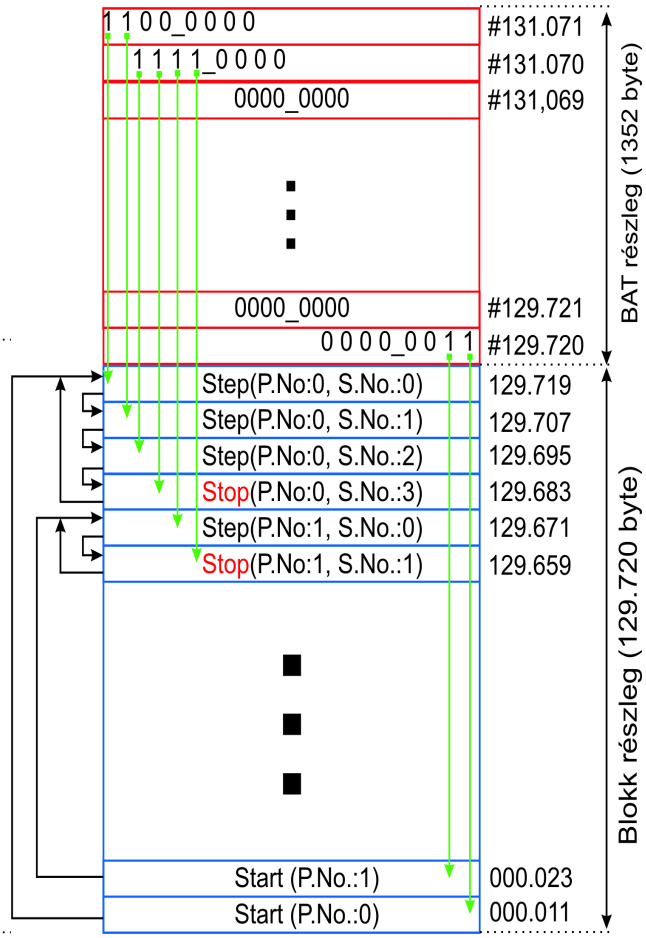
ábra 27: A LED kijelzők időmultiplexelt vezérlése. Forrás: [10] 3-3 ábra.



ábra 28: Az SRAM felépítése Forrás: [12]



ábra 29: Nyers memóriastruktúra



ábra 30: Blokkolt memóriastruktúra

7	6	5	4	3	2	1	0	
DAD (0x16)								1. byte
A blokkban tárolt információ alkalmazása előtti késleltetés **felső** nyolc bitje[15:8]								2.byte
A blokkban tárolt információ alkalmazása előtti késleltetés **alsó** nyolc bitje[7:0]								3. byte
1-es csatorna adata [7:0]								4. byte
2-es csatorna adata [7:0]								5. byte
3-as csatorna adata [7:0]								6. byte
4-es csatorna adata [7:0]								7. byte
5-ös csatorna adata [7:0]								8. byte
6-os csatorna adata [7:0]								9. byte
7-es csatorna adata [7:0]								10. byte
8-as csatorna adata [7:0]								11. byte

táblázat 8: késleltetést és adatot tartalmazó csomag felépítése

7	6	5	4	3	2	1	0	
PNUM_REP (0x15)								1. byte
Programszám [11:4]								2.byte
Programszám[3:0]				Ismétlés[13:10]				3. byte

táblázat 9: Program számot és ismétlési információt tartalmazó csomag

7	6	5	4	3	2	1	0	
INSERT (0x11)/READ_STEP(0x0E)/START_PROGRAM(0x19)								1. byte
Programszám (received_program_number)[11:4]								2.byte
Programszám[3:0]				Lépsszám(received_step_number)[13:10]				3. byte
Lépésszám[9:2]								4. byte
Lépésszám[1:0]		XXXXXX						5. byte

táblázat 10: Új blokk, blokk olvasás, program indítás parancsának csomagja

7	6	5	4	3	2	1	0	
WRITE_STEP(0x18)								1. byte
Programszám (received_program_number)[11:4]								2.byte
Programszám [3:0]				Lépsszám(received_step_number)[13:10]				3. byte
Lépésszám[9:2]								4. byte
Lépésszám [1:0]		Ismétlési adatok [3:0]				XX		5. byte
A blokkban tárolt információ alkalmazása előtti késleltetés **felső** nyolc bitje[15:8]								6. byte
A blokkban tárolt információ alkalmazása előtti késleltetés **alsó** nyolc bitje[7:0]								7. byte

táblázat 11: Adott lépést író parancs struktúrája

7 Köszönetnyilvánítás

Szakdolgozatom nem jöhetet volna létre Dr. Végh János, Tóth László, Póser István és Nagy Gábor szakmai segítségnyújtása nélkül, akiknek ezúton szeretném megköszönni támogatásukat.

További köszönetet mondanék Borbély Dianának aki sokat segített abban, hogy a dolgozat elnyerje végső formáját.