



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Elektronikus Eszközök Tanszéke

Országos Tudományos Diákköri Konferencia

Pályamunka

ÚJELVŰ LOGIKAI ÁRAMKÖRCSALÁD KIFEJLESZTÉSE PROCESSZOROK MŰVELETVÉGZŐ EGYSÉGÉHEZ

KÉSZÍTETTE: BLUTMAN KRISTÓF VILLAMOSMÉRNÖK MSC V. ÉVF.

KONZULENS: DR. HOSSZÚ GÁBOR, EGYETEMI DOCENS, ELEKTRONIKUS ESZKÖZÖK TANSZÉKE

BUDAPEST, 2012

Tartalom

Tartalom.....	2
Bevezetés	3
Irodalmi áttekintés	5
1 Felhasznált elméletek, modellezési eszközök.....	6
1.1 Algoritmuselmélet	7
1.2 Elvonatkoztatási szintek	9
1.3 Absztrakt algebra	10
1.4 Boole algebra	12
2 A szakirodalomból ismert megoldások.....	16
2.1 Bináris összeadás	17
2.2 Bináris szorzás	23
2.3 A gyakorlatban használatos ALU architektúrák	29
2.4 Utasításkészlet	31
2.5 A sebesség vizsgálata.....	32
Az elért új eredmények	34
3 A Boole algebra új reprezentációja.....	35
3.1 A Boole modul.....	36
3.2 A Boole függvények reprezentációja.....	42
3.3 Új hardverleírási elv: Az ML-RTL leírás	46
4 Az IBZ kapu.....	49
4.1 A kétbites IBZ kapu	50
4.2 Az n-bites IBZ kapu.....	52
5 Az IBZ ALU	58
5.1 Az IBZ szelet	59
5.2 Az IBZ ALU műveletei	61
5.3 Az n-operandusú ALU	66
5.4 Utasításkészlet	67
6 Szimulációs eredmények összefoglalása és értékelésük	69
6.1 Az IBZ szelet jellemzői	70
6.2 VHDL implementáció.....	71
6.3 IBZ család alkalmazása mikroprocesszorban	72
Eredmények értékelése	74
Felhasznált irodalom.....	75
Melléklet.....	77
6.4 Az IBZ ALU VHDL implementációja és szimulációja.....	78
6.5 Az IBZ ALU alkalmazása AVR mikroprocesszorban.....	93
6.6 Az RTL és ML-RTL leírás összehasonlítása	95

Bevezetés

A processzorok sebességét nagyrészt az aritmetikai és logikai műveletek végrehajtása szabja meg. Ezt a feladatot az ún. aritmetikai-logikai egység (Arithmetic Logic Unit, ALU) látja el. Nagyfokú kihasználtsága miatt az ALU architektúrájának optimalizálásával jelentős eredményeket lehet elérni az utasítások ciklusszámának és ciklusidejének csökkentésében. Az elvégzett kutatás során a kifejlesztettem egy, a korábbiaknál elvileg gyorsabb működésű ALU-t.

Az ALU egy kombinációs logika, ami Boole függvényeket valósít meg. Kialakítottam a Boole algebra digitális áramköri megvalósításának egy, az eddigiektől eltérő modelljét és kidolgoztam egy új logikai áramkör családot (IBZ), amely a Boole algebra egy újfajta reprezentációján alapul. Az erre alapozott univerzális áramkör család (IBZ) alkalmas az összes Boole művelet végrehajtására, fő jellemzője pedig a különlegesen kis késleltetés. A dolgozat tartalmazza az IBZ áramkör család működését igazoló elméleti leírást, valamint a megvalósíthatósági szempontok vizsgálatát. Az áramkör család megvalósításához definiáltam egy sajátos hardverleírási szintet (ún. Moduláris Logikai-RTL), amely az áramkörökkel megvalósított Boole algebrai műveletek egymásutánosságát egy irányított gráf színezési problémájára vezeti vissza.

Az IBZ áramkör család alkalmazásaként egy új ALU architektúra is kidolgozásra került, ami az összes logikai műveletet, az aritmetikai műveleteket (összeadás, kivonás, szorzás és osztás), valamint a shift és rotate műveleteket optimális sebességgel hajtja végre. Az összetettebb rutinok végrehajtásához két speciális műveletet implementáltam, amelyek egyszerre adnak össze számokat és végeznek rajtuk logikai műveleteket. Az így elkészült ALU tartalmaz egy ehhez illeszkedő, új rendszerű, a szokásoshoz képest kibővített, utasításkészletet. Ez összetett, de egy órajelciklus alatt végrehajtható műveleteket tartalmaz. Az összeadás és szorzás műveletek megvalósításának különlegessége, hogy annak elvégzésére az irodalomban ismert gyors összeadó és szorzó algoritmusokat az IBZ kapuk segítségével valósítottam meg.

Az IBZ ALU implementálása VHDL nyelven, az általam kidolgozott Moduláris Logikai-RTL (ML-RTL) szinten történt. Alternatívaként egy RTL szintű, az irodalomban ismert ALU

architektúra is megvalósításra került. A két modell sebességét és helyfoglalását FPGA szintézer program segítségével vizsgálva megállapítható, hogy az alacsonyabb szintű, újelvű leírással jelentős sebességnövekedés érhető el. A kész IBZ ALU beépítésre került egy, az irodalomból ismert mikroprocesszor modellbe. A megoldás során az utasításdekóder kiegészítésre került, így egy, már létező processzor architektúra átkonfigurálhatóvá vált a dolgozatban korábban ismertetett újrendszerű megalkotott utasításkészlet használatára. A dolgozat tartalmazza az így kialakított mikroprocesszor alkalmazhatóságát.

Irodalmi áttekintés

Az alábbiakban a dolgozatom megértéséhez szükséges, közismert elméleti és újszerű gyakorlati modelleket, megoldásokat mutatom be. Bár az eredményeim eléréséhez szükséges volt a szakirodalomban széleskörű tanulmányozása, azok egy része jórészt független tőlük, és ezek az általam megfogalmazott, általános elvek nem kizárólag a jelen dolgozatban tárgyalt eszközökben hasznosíthatóak. Így ebben a fejezetben csak a nélkülözhetetlen megoldások kerülnek említésre.

Eredményeim halmaza két részre bontható – elméleti és alkalmazási. Előbbivel a célom egy matematikai alapokon nyugvó, pontosan definiált, logikusan felépített elmélet megalkotása volt. Éppen ezért a dolgozatban a matematika eszközei – definíciók, tételek, bizonyítások is szerepelnek. Az elméleti eredményeim erősen építkeznek olyan általános matematikai elméletekre, mint az algoritmuselmélet, absztrakt algebra és esetenként használatra kerül a matematikai logika és a kategóriaelmélet. A megértéshez szükséges minden fogalmat a dolgozat keretein belül pontosan definiáltam, és igyekeztem hiánymentesen felépíteni a matematikai gondolatmenetemet. Ennek ellenére a dolgozat helyenként matematikailag nehezebben érthető, tömör. Ezek a részek fokozott figyelmet követelnek az olvasótól, és több időt, türelmet.

Az alkalmazások részben az elméleti eredményeim implementálásai, így ott is vissza kell nyúlni a korábban kidolgozott matematikai modellhez, ugyanis a nélkül nagyon nehezen, csak heurisztikusan kezelhetőek. Mindazonáltal, minden alkalmazási eredmény ténylegesen is gyakorlati, hatásvizsgálatokat, szimulációkat is végeztem. Erre kiváló eszköznek bizonyult a VHDL nyelv, amely a magas szintű absztrakcióból kiindulva, egy standard módszer az elmélet gyakorlatba ültetésére. Az alkalmazási eredményeim tovább csoportosíthatóak. Egyrészt, vannak a teljesen általam kitalált eszközök, és másrészt, ezen eszközöket a széles körben használt áramkörüi funkciók, részfunkciók kiváltására, majd a két megoldás összehasonlítására használtam. Ezen utóbbi csoport miatt ebben a fejezetben bemutatom a jelenleg használt, legmodernebb áramköröket, architektúrákat is.

Bízom benne, hogy élvezetes, követhető olvasmánynak bizonyul a munkám, és sikerül átadnom a téma iránti lelkesedésemet az olvasónak.

1 Felhasznált elméletek, modellezési eszközök

Az első fejezetben a dolgozat témájának elméleti alapjait tárgyalom.

A legáltalánosabb modellezési szintről kiindulva, megállapítottam, hogy az összes processzor képes végrehajtani ugyanazokat az algoritmusokat, melyeket az algoritmuselméletben *számítható algoritmusoknak* neveznek. Ez a rész szolgál alapul az utasításkészletek absztrakt modellezésére is. Az alkalmazások által megkövetelt feltételek teljesítéséhez azonban az elektronikus eszközök alacsonyabb szintű leírása is szükséges. Számba vesszük a létező leírási szinteket, és kiválasztjuk az optimálist, amely elég komplex ahhoz, hogy kvalitatív paramétereket kezeljünk vele, de nem válik kezelhetetlenné bonyolult áramkörök esetében sem. A választás a logikai szintre esett. A logikai szint modellezésére a Boole algebra matematikai eszköztára nyújt megoldást, így részletes bemutatásra kerülnek a legfontosabb sajátosságai.

A Boole algebra új matematikai reprezentációjára alapozva alkottam meg dolgozatban később ismertetett, újból logikai áramköröket, és szintén erre alapozva került definiálásra egy új hardverleírási elv, az ML-RTL leírás.

1.1 Algoritmuselmélet

Definíció: Számítható algoritmus a Turing-gép által végrehajtható utasítások véges sorozata.

Definíció: A Turing-gép¹ egy rendezett hetes (7-tuple), melynek elemei:
 $T = (Q, \Sigma, \Gamma, q_0, b, F, \delta)$

- Q véges, nemüres halmaza az állapotoknak
- Γ véges, nemüres halmaza a szalagmemória szimbólumainak
- b az üres szimbólum
- $\Sigma \subseteq \Gamma \setminus \{b\}$ a bemeneti szimbólumok halmaza
- $q_0 \in Q$ a kezdeti állapot
- $F \subseteq Q$ az elfogadó állapotok halmaza
- $\delta: Q \setminus F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ az állapot átmeneti függvény, $\{L, R\}$ a léptető operátorok.

A Turing-gép segítségével pontosabban definiálható a számítható algoritmus.

Definíció: Szó egy H halmaz meghatározott elemeiből képzett egy rendezett n -es (n - tuple), ismétlést is megengedve.

Jelölés:² H^i a H halmaz, i hosszú szavainak halmaza.

Ekkor H^* a H halmaz elemeiből kompozícióval képzett összes lehetséges kifejezés halmaza:

$H^* = \bigcup_{i=0.. \infty} H^i$. Definiálható még a $H^+ = H^* \setminus \{\emptyset\} = \bigcup_{i=1.. \infty} H^i$.

Példa: $H = \{a, b\}$, $H^* = \{\emptyset, a, b, ab, aa, ba, bb, aaa, aab, \dots\}$ $H^+ = \{a, b, ab, \dots\}$

Definíció: Utasítás a δ halmaz minden eleme.

Definíció: Bemenet a Σ^* halmaz minden eleme.

Definíció: Számítható eljárás a tetszőleges $i \in \Sigma^*$ bemeneten végrehajtott $c \in \delta^*$ utasítássorozat.

Definíció: Számítható algoritmus az $\{i \in \Sigma^n, n \in \mathbb{N}, n < \infty\}$ bemeneten végrehajtott $\{c \in \delta^m, m \in \mathbb{N}, m < \infty\}$ utasítássorozat.

A Turing-gép szokásos megjelenítése egy állapotgép, amely egy szalagról olvas és ír be adatot, és lépked a szalagon.

A gép egy ciklusa a következő, megbonthatatlan lépéssorozatból áll:

1. Az automata q állapotban van. Beolvasunk egy ' a ' szimbólumot a szalagról.

¹ (HOPCROFT & ULLMAN, 1979) (FRIEDL, 2010)

² (FRIEDL, 2010)

2. A jelenlegi állapot, q és a beolvasott karakter, ' a ' (továbbiakban: $(q, 'a')$) meghatározza, milyen ' b ' karaktert írjunk vissza a szalagra a beolvasott ' a ' szimbólum helyére. Beírjuk ' b '-t.
3. $(q, 'a')$ szerint balra vagy jobbra lépünk. (néhány értelmezésben van helyben maradó utasítás is)
4. $(q, 'a')$ szerint a következő, q állapotba lépünk.

Tézis(TURING, 1936): Minden algoritmizálható probléma megoldható egy univerzális Turing-géppel (universal Turing machine, UTM).³

Azt a nyelvet, amely bármely Turing-gép által megvalósítható algoritmust képes leírni, *Turing-teljesnek*⁴ nevezzük. Egy Turing gép felfogható egy olyan processzorként is, aminek az utasításkészlete egy Turing-teljes nyelv. Két automata akkor *Turing-ekvivalens*,⁵ ha bármelyik képes szimulálni a másikat. Gyakorlatilag az összes mai processzor Turing-ekvivalens egymással. Az egyetlen lényeges különbség működés ideje. Bár egy Turing-géppel bármilyen processzor – elvben – szimulálható, de közel sem akkora sebességgel. A Turing-gép futási idejére csupán felső korlátot lehet adni, a pontos kiszámítására, akár becslésére nincs igazán jó módszer. Ez az ún. *megállási probléma (halting problem)*.⁶ A modellezési szinten változtatnunk kell, hiszen a Turing-féle modell nem ad számot – sok egyéb más mellett – a processzorok tényleges fizikai megvalósításáról és az időzítési viszonyokról.

³ Egy UTM tulajdonképpen egy *Turing-gép*, ami más Turing-gépeket képes szimulálni, a mi szempontunkból nincs nagy jelentősége a különbségnek.

⁴ (BACH, 2002)

⁵ (BACH, 2002)

⁶ (BACH, 2002)

1.2 Elvonatkoztatási szintek

A digitális áramkörök, azon belül a processzorok modellezését is úgynevezett *absztrakciós szintekre* szokás bontani. Ilyenkor azt kell eldöntenünk, hogy a valós fizikai világ információtartalmából melyek a fontos és melyek az elhanyagolható tényezők. Két szomszédos absztrakciós szint között felfelé haladva mindig valamilyen egyszerűsítés történik, a modell egyre kevesebb dolgot vesz figyelembe.

Absztrakciós szint	Matematikai leírás	Információtartalom ⁷
Algoritmikus	Algoritmuselmélet	1
RTL	Reguláris nyelvek és automaták	10
Logikai	Boole algebra	10^3
Áramkör	Tranzisztor modellek, hálózat egyenletek	10^5
Technology	Fizikai alaptörvények	$10^5 <$

1-1. táblázat: Absztrakciós szintek modelljei

A tervezés során ki kell választani a megfelelő szintet, amely alkalmas a probléma kezelésére, de nem túl bonyolult ahhoz, hogy számolni lehessen vele. Gyakran, azonban, ha alacsonyabb szintet választunk a leíráshoz, mint indokolt lenne, nagyobb szabadság kínálkozik az optimalizálásra.

⁷ (SZITTYA & JÁVOR, 1984)

1.3 Absztrakt algebra

A Boole algebra egy absztrakt algebra, így szükséges az utóbbi megfelelő ismerete az előbbi tanulmányozásához.

Alapfogalom: Egy H halmaz nem definiálható, ún. *alapfogalom*.

Definíció: Egy H halmaz *számossága* elemeinek a száma. Ha találunk egy olyan H' halmazt, hogy a H és H' között egyértelmű megfeleltetés létesíthető, akkor a két halmaz számossága megegyezik.

Jelölés: H halmaz *számossága* $|H|$.

Definíció:⁸ *Rendezett n -es (n -tuple)* egy n elemű G halmaz, amely elemeihez létezik $G \rightarrow \mathbb{N}$ izomorfizmus.

Definíció: *Magma* az a $\{H, \cdot\}$ rendezett kettes (2-tuple), amelyre teljesül (0).

Definíció: *Félcsoport* az a $\{H, \cdot\}$ rendezett kettes (2-tuple), amelyre teljesül (2).

Definíció: *Monoid* az a $\{H, \cdot, 1\}$ rendezett hármas (3-tuple), amelyre teljesül (2) és (6).

Definíció:⁹ *Csoport* az a $\{H, \cdot, ^{-1}, 1\}$ négyes (4-tuple), amelyre teljesül (2), (6) és (8).

Definíció:¹⁰ *Abel-csoport* az a csoport ((2), (6) és (8)), amelyre teljesül (4).

Definíció:¹¹ *Gyűrű* az a $R\{H, +, \cdot, -, 0\}$ ötös (5-tuple), amelyre $\{H, +, -, 0\}$ Abel-csoport((1), (3), (5), (7)), $\{H, \cdot\}$ félcsoport, és teljesül (9), (10).

Definíció:¹² *Kommutatív gyűrű* az a gyűrű ((1), (2), (3), (5), (7), (9), (10)), melyre teljesül (4).

Definíció:¹³ *Egységelemes gyűrű* az a $\{H, +, \cdot, -, 0, 1\}$ gyűrű ((1), (2), (3), (5), (7), (9), (10)), amelyre teljesül (6).

Definíció:¹⁴ *Test* az a $F\{H, +, \cdot, -, ^{-1}, 0, 1\}$ kommutatív (4), egységelemes (6) gyűrű ((1), (2), (3), (5), (7), (9), (10)), amelyre teljesül (8). $\{H, \cdot, ^{-1}, 1\}$ csoport.

- H nemüres halmaz,
- $+, \cdot : H \times H \rightarrow H$ bináris műveletek H felett, nevük összeadás és szorzás **(0)**
- $-, ^{-1} : H \rightarrow H$ inverzképző a $+, \cdot$ műveletekre nézve $(\)^{-1}$ csak *test* esetén létezik),

⁸ (GIVANT & HALMOS, 2009)

⁹ „

¹⁰ „

¹¹ „

¹² „

¹³ „

¹⁴ (LIDL & NIEDERREITER, 1997)

- $0, 1 \in H$ az egységelemek (1 csak egységelemes gyűrű esetén létezik).

Definíció:¹⁵ *Gyűrű homomorfizmus* egy művelettartó φ függvény két gyűrű, A és A' között: $(\varphi: A \rightarrow A')$, melyre teljesül $\varphi(a + b) = \varphi(a) + \varphi(b)$; $\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$; $\varphi(1) = 1$ minden $a, b \in A$ -ra.

Definíció:¹⁶ *Csoport homomorfizmus* egy művelettartó φ függvény két csoport, $(G, *)$ és $(G', \cdot)$ között: $(\varphi: G \rightarrow G')$, melyre teljesül $\varphi(a * b) = \varphi(a) \cdot \varphi(b)$.

Tulajdonság	Művelet ($p, q, r \in H$)			
	+		·	
Asszociativitás	$p + (q + r) = (p + q) + r$	(1)	$p \cdot (q \cdot r) = (p \cdot q) \cdot r$	(2)
Kommutativitás	$p + q = q + p$	(3)	$p \cdot q = q \cdot p$	(4)*
Egységelemek	$p + 0 = p$	(5)	$p \cdot 1 = p$	(6)**
Inverz	$p + (-p) = 0$	(7)	$p \cdot p^{-1} = 1$	(8)***
Disztributivitás			$p \cdot (q + r) = p \cdot q + p \cdot r$	(9)
			$(q + r) \cdot p = q \cdot p + r \cdot p$	(10)
Idempotencia	$p + p = 0$	(11)'	$p \cdot p = p$	(12)'

1-2. táblázat: Absztrakt algebrai axiómák

Definíció:¹⁷ *Modul* egy R gyűrű felett az az $\{M, \lambda\}$ kettes (2-tuple), ahol $M\{H, +, -, 0\}$ Abel-csoport, $\lambda: R \rightarrow (M \rightarrow M)$ *gyűrű homomorfizmus*. Ez azt jelenti, hogy $\forall a \in R$ -re $\exists \lambda(a): M \rightarrow M$, melyre, ha $r, s \in R$, $x, y \in M$, akkor $\lambda(r)(x + y) = \lambda(r)x + \lambda(r)y$, $\lambda(r + s)x = \lambda(r)x + \lambda(s)x$, $\lambda(r \cdot s)x = \lambda(r)(\lambda(s)x)$, $\lambda(1)x = x$. A legutolsó axióma akkor teljesül, ha R egységelemes gyűrű.

Definíció: *Vektortér* egy F test (skalár) felett az a $\{V, \cdot\}$ rendezett kettes, amelyben V Abel-csoport, $(\cdot): R \rightarrow (M \rightarrow M)$ gyűrű homomorfizmus (a skalárral való szorzás).

¹⁵ (HAZEWINKEL, GUBARENI, & KIRICHENKO, 2004)

¹⁶ (HAZEWINKEL, GUBARENI, & KIRICHENKO, 2004)

¹⁷ (ANDERSON & FULLER, 1992)

1.4 Boole algebra

A standard digitális biteket és a logikai kapuk által megvalósított funkciót – mint az Claude Shannon munkásságának¹⁸ köszönhetően kiderült – a Boole algebra matematikai apparátusával írhatjuk le talán legegyszerűbben.

Definíció: Legyen a *Boole skalár* a $B = \{0,1\}$ halmaz.

Definíció:¹⁹ *Boole gyűrű* az az *egységelemes* (6) gyűrű ((1), (2), (3), (5), (7), (9), (10)), amelyre teljesül (11)' (azaz $p = -p$), és (12)'. Mivel itt $(-)$ az identitás művelet, elhagyható: $\{H, +, \cdot, 0, 1\}$

Legegyszerűbb példa Boole gyűrűre a $\{B, +, \cdot, -, 0, 1\}$ gyűrű.

Definíció:²⁰ *Boole algebra* az a $\{H, +, \cdot, -, {}^{-1}, 0, 1\}$ test, amelyre $-$, ${}^{-1}$ unáris műveletek megegyeznek, és (bevezetjük a $\wedge \leftrightarrow \cdot$, $\vee \leftrightarrow +$, $\neg \leftrightarrow -$, $\neg \leftrightarrow {}^{-1}$ jelölést) teljesül a disztributivitás az összeadásra is: $p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$, továbbá az egy művelet inverze nem egyezik meg az egységelemmel, nem igaz (7) és (8):

$$p + (-p) = 0 \text{ helyett } p \vee \neg p = 1 \quad (7: 0 \rightarrow 1) \quad p \cdot p^{-1} = 1 \text{ helyett } p \wedge \neg p = 0 \quad (8: 1 \rightarrow 0)$$

Az érthetőség kedvéért az axiómák:

Ha $p, q, r \in H$, akkor

$p \wedge 1 = p$	$p \vee 0 = p$	(I)
$p \wedge \neg p = 0$	$p \vee \neg p = 1$	(II)
$p \wedge q = q \wedge p$	$p \vee q = q \vee p$	(III)
$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$	(IV)

1-1: A Boole algebra axiómái

Tétel (STONE, 1936): Egy *Boole algebra* átalakítható *Boole gyűrűvé* és viszont. A megfeleltetési szabályok a következők:

1. $p + q = (p \wedge \neg q) \vee (\neg p \wedge q)$; $p \cdot q = p \wedge q$ *Boole algebra* $\rightarrow$ *Boole gyűrű*
2. $p \vee q = p + q + p \cdot q$; $p \wedge q = p \cdot q$; $\neg p = p + 1$ *Boole gyűrű* $\rightarrow$ *Boole algebra*

¹⁸ (SHANNON, 1949)

¹⁹ (GIVANT & HALMOS, 2009)

²⁰ (GIVANT & HALMOS, 2009)

Vezessünk be új jelölést. Legyen a $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$ a „kizáró vagy” (XOR) művelet, $+ \leftrightarrow \oplus$, $\cdot \leftrightarrow \wedge$ megfeleltetéssel belátható, hogy a $\{B, \oplus, \wedge, 0, 1\}$ Boole-gyűrűt alkot: $\{B, \vee, \wedge, \neg, 0, 1\} \approx \{B, +, \cdot, 0, 1\} \approx \{B, \oplus, \wedge, 0, 1\}$.

Állítás: $(\{0,1\}, \wedge)$ és $(\{0,1\}, \vee)$ monoidok,²¹ így nem alkothatnak magasabb szintű algebrai struktúrát.

Állítás: $(\{0,1\}, \oplus)$ Abel-csoport.²²

A Boole algebra axiómái párokba rendezhetőek egy különleges szimmetria szerint, ahogy ezt tettük az (I), (II), (III), (IV) pontokban. Ennek a szimmetriának a megragadására vezessünk be új fogalmakat.

Definíció: A Boole kifejezések (Boolean polynomials) egy adott formális nyelv mondatai,²³ amit nevezzünk Boole-nyelvnek. Egy formális nyelv a megadható grammatikával:

A grammatikát egy négyes (4-tuple) határozza meg:²⁴ $G = (N, \Sigma, P, S)$ ahol

- N a grammatikai szimbólumok halmaza,
- Σ a nyelv karakterkészlete,
- P a levezetési szabályok összessége,
- S a mondatszimbólum.

A Boole kifejezések nyelvét a következő grammatika határozza meg: $N = \{A, C, D, V, \Lambda\}$, $\Sigma = \{a_i, 0, 1, (,), \wedge, \vee, \neg\}$, $i = 1..n$, $P =$

• $S \rightarrow D$	• $D \rightarrow (C)$	• $D \rightarrow \neg D$
• $C \rightarrow DVD$	• $C \rightarrow D\Lambda D$	• $D \rightarrow A$
• $V \rightarrow \vee$	• $\Lambda \rightarrow \wedge$	• $A \rightarrow a_i, 0, 1 \quad i = 1..n$

1-2: A Boole kifejezések levezetési szabályai

Adott egy levezetése P -nek, ami generálja egy mondatát a Boole-nyelvnek, egy E Boole-kifejezést. Az ilyen módon levezethető kifejezések halmazát S_P -vel jelölöm.

Definíció:²⁵ Egy $E \in S_P$ kifejezés levezetésében a következő szabályokat megváltoztatva kapjuk az E kifejezés variánsait:

²¹ (KATONA, RECSKI, & SZABÓ, 2006)

²² „

²³ (BACH, 2002)

²⁴ „

²⁵ (GIVANT & HALMOS, 2009)

Boole kifejezés	Jelölés	Szabálycsere
Önmaga	E	-
Komplementese	$'E$	a $S \rightarrow D$ helyett $S \rightarrow \neg D$
Kontraduálisa	E'	a $D \rightarrow A$ helyett $D \rightarrow \neg A$
Duálisa	E^*	$S \rightarrow D$ helyett $S \rightarrow \neg D$, $D \rightarrow A$ helyett $D \rightarrow \neg A$ ²⁶

1-3. táblázat: A Klein csoport operációinak eredménye

Definíció: a Boole függvények azok az $f(a_1, a_2, \dots, a_n), f: B^n \rightarrow B$ függvények, amelyekhez $\exists E \in S_p$ Boole kifejezés, hogy $f(a_1, a_2, \dots, a_n) = E(a_1, a_2, \dots, a_n)$.

Definíció: Az f Boole függvény komplementese/kontraduálisa/duálisa az az $'f/f'/f^*: B^n \rightarrow B$ függvény, amelyre, ha $f(a_1, a_2, \dots, a_n) = E(a_1, a_2, \dots, a_n)$ teljesül, akkor $'f/f'/f^*(a_1, a_2, \dots, a_n) = 'E/E'/E^*(a_1, a_2, \dots, a_n)$

Megmutatható, hogy a négy operátor, $I, ', \neg, *$: $(B^n \rightarrow B) \rightarrow (B^n \rightarrow B)$ csoportot képez a kompozícióra: ha $D = \{I, ', \neg, *\}$, akkor $K\{D, \circ, ^{-1}, I\}$. Ez az ún. Klein-féle négyescsoport.²⁷

Tulajdonságok:

- $I \circ I, ' \circ ' \neg \circ \neg * \circ * = I$, tehát minden elem inverze önmaga.
- Ha $\circ^n(a_1, a_2, \dots, a_n) = a_1 \circ a_2 \circ \dots \circ a_n$, akkor π_3 permutációra $\pi_3 \circ^3(' \neg \neg *) = I$
- A Klein csoport Abel-csoport, mert $\circ$ kommutatív. (4)

A Boole függvények megvalósíthatóak ún. logikai kapukkal, amik elektronikus áramkörök.²⁸

Például a $p \wedge q$ kifejezés Boole függvényét $\wedge(p, q)$ -val jelölve, ami a következő kapuval valósítható meg:



1-3. ábra: Az AND kapu

Ez az ún. „és” kapu (AND gate). Emellett számos más, eleminek vehető logikai áramkör létezik, melyeket építőelemként használva meg lehet konstruálni egy nagyobb funkcionális egységet.²⁹

²⁶ Ekvivalensen, a $V \rightarrow V$ helyett $V \rightarrow \wedge$, a $\wedge \rightarrow \wedge$ helyett $\wedge \rightarrow V$ is ugyanezt adja

²⁷ (GIVANT & HALMOS, 2009)

²⁸ (ARATÓ, 1984)

²⁹ (ARATÓ, 1984)

A Boole algebra nem tartalmazza a fizikai realizációkhoz rendelhető késleltetési időt. Ezért ezt nem tudjuk kezelni az eddig megszokott absztrakciós szinten, a logikai áramköröket konkrétan kell kezelnünk. Egy L logikai áramkörhöz létezik egy $f: L \mapsto \Delta t \in \mathbb{R}$ függvény, ahol Δt az ún. *késleltetési idő*. Egy általános áramkörstruktúra, amely tetszőleges n bementre definiálva van, az $L(n)$ n -bementű logikai áramkörnél szokás használni a $f(n) = \Delta t$ speciális esetet (például, n -bites összeadó, n -bites szorzó, stb.).

Jelölés: Legyenek $f, g: \mathbb{R} \rightarrow \mathbb{R}$. Azt mondjuk, $f(x) = O(g(x))$, ha $\exists M \in \mathbb{R}^+$ és $x_0 \in \mathbb{R}$, melyekre $x > x_0$ esetén $|f(x)| < M \cdot |g(x)|$. Például, egy $p(x)$ n -ed fokú polinomra $p(x) = O(x^n)$. Konvenció, hogy a logikai áramkörök késleltetésének minőségi összehasonlítása érdekében $f(n)$ helyett egy $O(g(n))$ függvényt adnak meg, ahol $g(n)$ logaritmikus, lineáris, négyzetes, n -ed fokú, exponenciális, stb. függvénye lehet n -nek.

2 A szakirodalomból ismert megoldások

Ebben a fejezetben azokat az eszközöket, megoldásokat mutatom be, amelyeket már széles körben alkalmaznak többek között processzorokban. Elsősorban a nagysebességű alkalmazásokon lesz a hangsúly, a jelen dolgozat céljának megfelelően. Bemutatásra kerülnek az ALU-k összes, aritmetikai és logikai és egyéb részegységei, és az erre kidolgozott, közel leggyorsabb megoldások. Az összeadás/kivonás bináris megvalósításához szükséges egy új számbábrázolási módszer, amellyel e két művelet elvégezhető ugyanazon az eszközön. A mai gyors összeadók architektúrája, és ezek közül négy egyedi megoldás is bemutatásra kerül. Mindezek egy helyen történő alkalmazásaként, a szokványos ALU architektúra legfontosabb elvei is ismertetve lesznek. A mindezt vezérlő utasításkészletek főbb alapelvei és időzítési vizsgálat zárja a fejezetet. Az általam kitalált új ALU architektúra és kiegészített utasításkészlet ezeken a megoldásokon alapul.

2.1 Bináris összeadás

Ha $n = k + l$ kettes komplementes³⁰ számokat szeretnénk összeadni. Legyenek ezek $a' = \overline{a_{k-1}a_{k-2} \dots a_0 a_{-1} \dots a_{1-l} a_{-l}}$ és $b' = \overline{b_{k-1}b_{k-2} \dots b_0 b_{-1} \dots b_{1-l} b_{-l}}$. Vezessük be az a_i, b_i párokat egy-egy félösszeadóba, amit HA_i -vel jelölök. Ezekből n darab lesz. Ennek a kimenetén két szám lesz jelen: az p_i kimenetekből képzett $\overline{p_{k-1}p_{k-2} \dots p_0 p_{-1} \dots p_{1-l} p_{-l}}$ szám, és a g_i kimenetek $\overline{g_{k-1}g_{k-2} \dots g_0 g_{-1} \dots g_{1-l} g_{-l} C}$. A C egy extra bemenet lesz az utolsó számjegy beviteléhez. Ennek a két számnak az összege lesz az operandusok összege, $\overline{s_{k-1}s_{k-2} \dots s_0 s_{-1} \dots s_{1-l} s_{-l}}$. Hogy az összeadás megtörténjen, a félösszeadókat cseréljük le teljes összeadókra! Ekkor minden FA_i képes még egy extra egybités számot hozzáadni az eredményhez. Ez lesz c_{i-1}^{out} , az FA_{i-1} carry kimenete. Az összekötési megfeleltetés:

$$\overline{c_k^{in} c_{k-1}^{in} c_{k-2}^{in} \dots c_0^{in} c_{-1}^{in} \dots c_{1-l}^{in} c_{-l}^{in}} = \overline{C_{out} c_{k-2}^{out} c_{k-3}^{out} \dots c_0^{out} c_{-1}^{out} \dots c_{-l}^{out} C_{in}}.$$

			$\overline{0a_{k-1}a_{k-2} \dots a_0 a_{-1} \dots a_{1-l} a_{-l}}$
			$\overline{0b_{k-1}b_{k-2} \dots b_0 b_{-1} \dots b_{1-l} b_{-l}}$
$\oplus$	$\overline{0b_{k-1}b_{k-2} \dots b_0 b_{-1} \dots b_{1-l} b_{-l}}$	$\wedge$	$\overline{g_{k-1}g_{k-2} \dots g_0 g_{-1} \dots g_{1-l} g_{-l} C_{in}}$
$=$	$\overline{0p_{k-1}p_{k-2} \dots p_0 p_{-1} \dots p_{1-l} p_{-l}}$	$\rightarrow \odot$	$\overline{0p_{k-1}p_{k-2} \dots p_0 p_{-1} \dots p_{1-l} p_{-l}}$
			$= \overline{c_{k-1}^{out} c_{k-2}^{out} c_{k-3}^{out} \dots c_0^{out} c_{-1}^{out} \dots c_{-l}^{out} C_{in}}$
$\oplus$	$\overline{C_{out} c_{k-1}^{in} c_{k-2}^{in} \dots c_0^{in} c_{-1}^{in} \dots c_{1-l}^{in} c_{-l}^{in}}$	$\leftarrow$	$= \overline{C_{out} c_{k-1}^{in} c_{k-2}^{in} \dots c_0^{in} c_{-1}^{in} \dots c_{1-l}^{in} c_{-l}^{in}}$
$=$	$\overline{C_{out} s_{k-1} s_{k-2} \dots s_0 s_{-1} \dots s_{1-l} s_{-l}}$		

2-1. táblázat: Az összeadás folyamatának szemléltetése

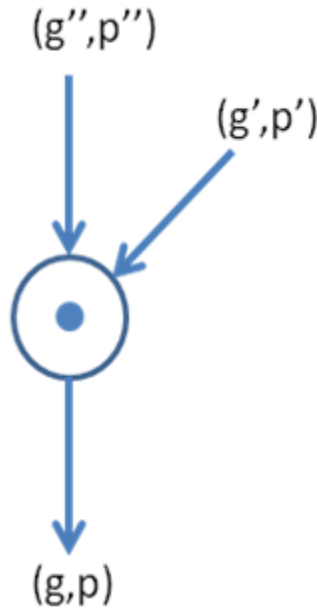
Figyeljük meg, hogy minden c_i^{out} függ az összes előző, $\{c_j^{out}, j < i\}$ kimenettől. Ezért az összeadó az eredményt csak azután adja ki, ha a carry az összes FA_i -n végighaladt. Ennek gyorsítása érdekében találtak ki más megoldásokat a carry kezelésére. A táblázatban szerepel egy bizonyos $\odot$ művelet, amely a carry előállítása p és g segítségével. A teljes összeadóban ez a $c_i^{out} = g_i \vee (p_i \wedge c_i^{in})$ műveletként hajtódik végre. A $\odot$ operátor azonban a hiányzó c_i^{in} -t a korábbi $g_j, p_j, j < i$ számjegyekből számolja. A naiv összeadóban minden FA_i -n végig kell haladnia a jelnek, hogy a kimenet érvényes legyen. Ezekből n darab van, így $\Delta t = O(n)$.

³⁰ (HOLDSWORTH & WOODS, 2002)

2.1.1 A carry-lookahead³¹ összeadó

A $\odot$ függvényt megvalósító áramkört *lookahead-carry-unit*nek nevezem. A FA_i teljes összeadóinkat úgy kell módosítani, hogy c_i^{out} helyett a g_i, p_i jeleket adja ki a kimenetén, ezeket a módosított egységeke fogom ezentúl FA_i -vel jelölni. Ezután a $\overline{g_{k-1}g_{k-2} \dots g_0g_{-1} \dots g_{1-l}g_{-l}C_{in}}$ és $\overline{0p_{k-1}p_{k-2} \dots p_0p_{-1} \dots p_{1-l}p_{-l}}$ számokat bevezetjük az ún. *lookahead-carry-unit*ba, ami előállítja a kívánt $\overline{c_{k-1}^{out}c_{k-2}^{out}c_{k-3}^{out} \dots c_0^{out}c_{-1}^{out} \dots c_{-l}^{out}C_{in}}$ számot. Ennek biteit visszavezetjük a FA_i c_i^{in} bemeneteire, amelynek eredményeként előáll a $\overline{C_{out}S_{k-1}S_{k-2} \dots S_0S_{-1} \dots S_{1-l}S_{-l}}$ kívánt összeg. Az egyetlen kérdés, hogy a *lookahead-carry-unit*ot hogyan valósítsuk meg. Állítsuk elő a $c_{i+1}^{in} = c_i^{out}$ -t! Definiáljuk rekurzívan a $G_i = g_i \vee (p_i \wedge G_{i-1})$ és $P_i = p_i \wedge P_{i-1}$ jeleket úgy, hogy $G_{-l} = g_{-l}$ és $P_{-l} = p_{-l}$. Ezek lesznek a több-bites propagate és generate jelek.

Állítás: Ekkor $c_i^{out} = G_i \vee (P_i \wedge C_{in})$. Ha $g_{-(l+1)} = C_{in}$, $G_{-l} = g_{-l} \vee (p_{-l} \wedge C_{in})$, akkor $c_i^{out} = G_i$. Legyen a $\odot$ művelet³² a következő: $(g_i, p_i) \odot (g_j, p_j) = (g_i \vee (p_i \wedge g_j), p_i \wedge p_j)$. Jelöljük $x_i = (g_i, p_i)$, $X_i = (G_i, P_i)$.



2-2. ábra³³ A prefix fa alapeleme

Az ilyen fát *prefix fának* nevezik.

³¹ (PARHAMI, 2000)

³² (LADNER & FISCHER, 1980)

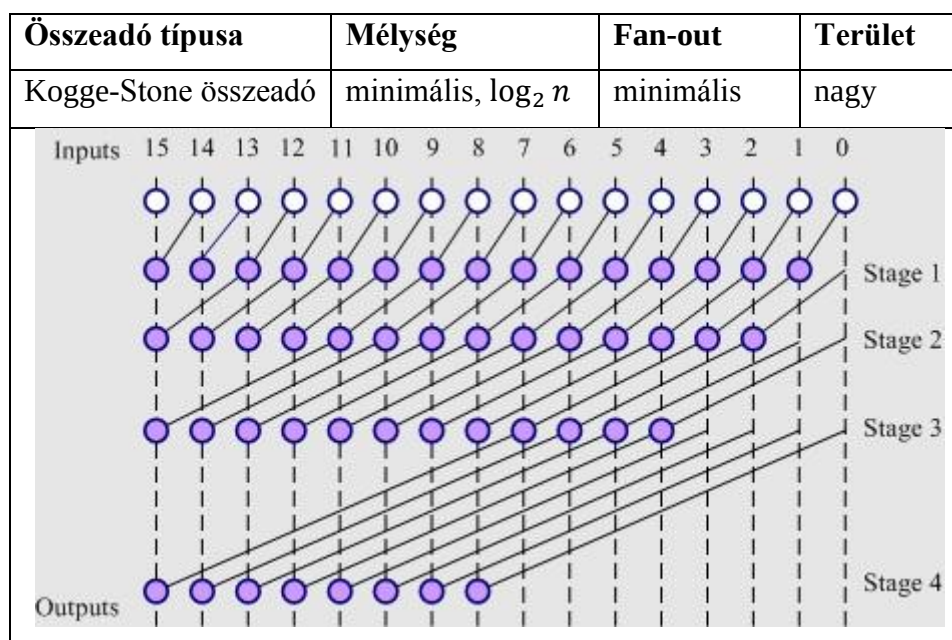
³³ (PARHAMI, 2000)

Állítás: $X_i = \odot x_{j=-l..i}$, azaz a korábban megismert $\odot$ műveletre igaz:

$\odot: (x_{k-1}, \dots, x_{-l}) \mapsto ((x_{k-1} \odot \dots \odot x_{-l}), (x_{k-2} \odot \dots \odot x_{-l}), \dots, (x_{1-l} \odot x_{-l}), x_{-l}) = (X_{k-1}, \dots, X_{-l})$. A $\odot$ művelet minimális késleltetéssel történő elvégzését *parallel prefix problémának*³⁴ hívják. A gyakorlatban fa struktúrában végzik el a műveletet. Az alapvető építőelem két, (g_i, p_i) páron elvégzett művelet, ami a 2-2 ábrán látható.

2.1.2 A leggyorsabb összeadók

Teljes körű elméleti és gyakorlati vizsgálatok megállapították, hogy a Carry-lookahead összeadók leggyorsabbak és a legkisebb helyfoglalásúak.³⁵ Az irodalomban számtalan, közel az elméleti minimum késleltetéshez közeli variáns született a prefix fa összeadókra. Ilyen megoldások közül talán a legoptimálisabbak a *Kogge-Stone összeadó*, a *Ladner-Fischer összeadó*, a *Brent-Kung összeadó* és a *Hans-Carlson összeadó*. Láthatjuk, hogy ez a fa teljesíti a parallel prefix feltételeket, ugyanis minden kimenete csatlakozik az összes kisebb sorszámú bemenethez - $X_i = (x_i \odot \dots \odot x_{-l})$ teljesül minden esetben. Hasonlítsuk össze a prefix fákat!³⁶



2-3.:A Kogge-Stone prefix fa³⁷

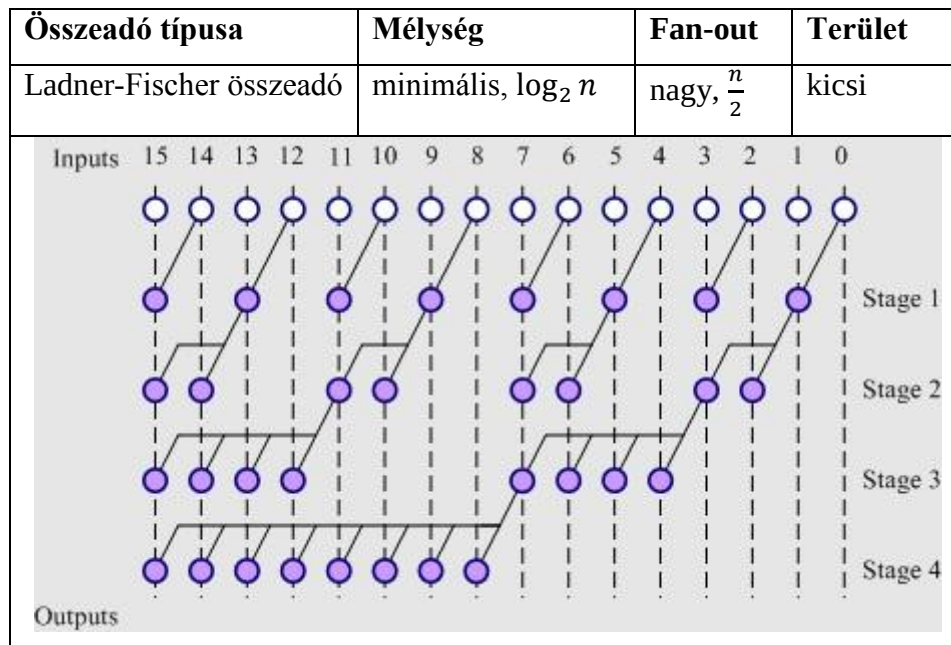
³⁴ (ZHU, CHENG, & GRAHAM, On the Construction of Zero-Deficiency Parallel Prefix Circuits with Minimum Depth, 2006)

³⁵ (CALLWAY & SWARTZLANDER, 1992)

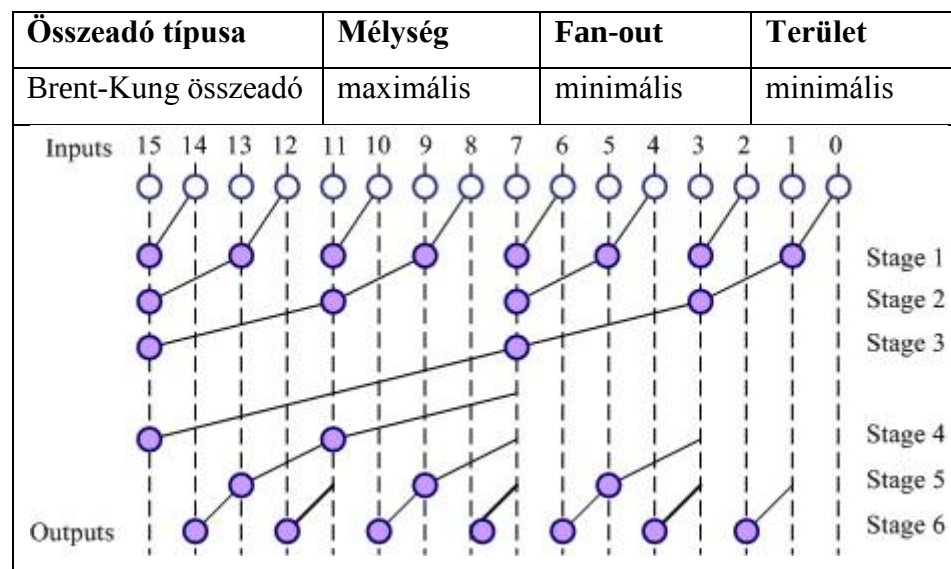
³⁶ <http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html>

³⁷ (KOGGE, 1973)

A 2-3 - 2-6 ábrákból kiderül, hogy miért hívják fának ezt a gráfelméleti szempontból fának nem minősülő³⁸ architektúrát; minden X_i kimenethez hozzárendelhető egy tényleges fa, amely az összes $x_i, \dots, x_{-i}$ bemenettől vezető olyan utat tartalmazza, amely más bemenetet nem érint.



2-4: A Ladner-Fischer prefix fa³⁹

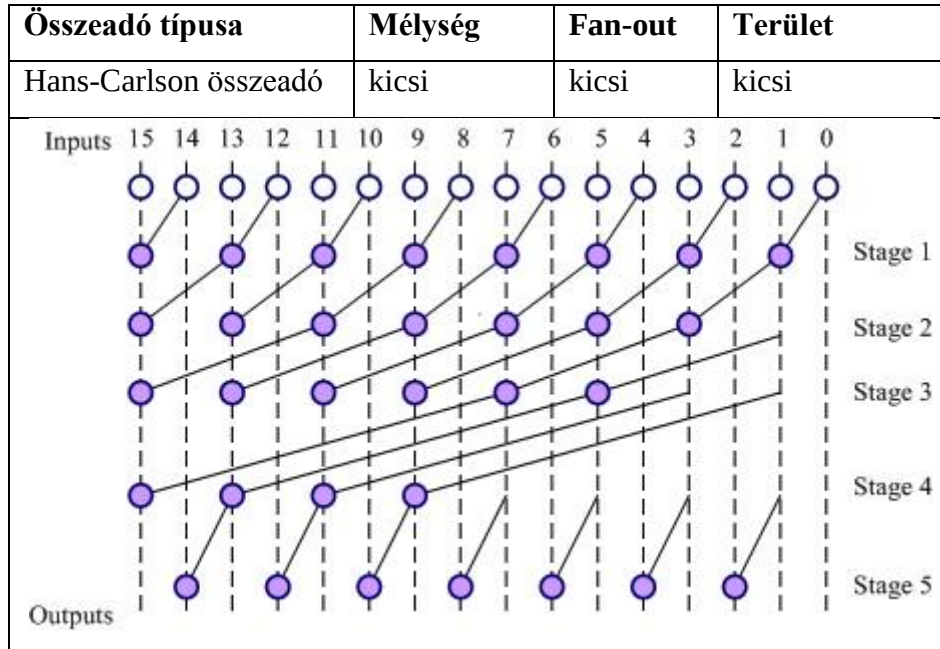


2-5: A Brent-Kung prefix fa⁴⁰

³⁸ található benne kör, bővebben lásd (KATONA, RECSKI, & SZABÓ, 2006)

³⁹ (LADNER & FISCHER, 1980)

⁴⁰ (BRENT & KUNG, 1982)



2-6.: A Hans-Carlson prefix fa⁴¹

Egy másik prefix fa az ún. Sklansky fa.⁴² Az optimális mélysége a prefix fának $\log_2 n$,⁴³ így a teljes késleltetés a prefix fa és az FA_i egységeknek összesen $\Delta t = O(\log_2 n)$, mivel minden FA_i késleltetése konstans. Ez az $O(n)$ -hez képest rendkívül jelentős javulás. Az összeadók architektúrájánál egyéb elvek is figyelembe vehetőek. Az ún. Ling összeadó⁴⁴ korábban már ismertetésre került. Egy másik megoldás, amivel a ma létező leggyorsabb összeadókat logikailag tervezik, hogy a hagyományos CLA-ban használt AND és NOT kapukat NAND kapukkal helyettesítik.⁴⁵ Ez a következő egyszerű átalakítással lehetséges: a $\odot$ művelet megvalósításánál a köztes generate jelek kiszámolására használt $g' = g_i \vee (p_i \wedge g_j)$ alak helyett a $g_i \vee (p_i \wedge g_j) = \neg(\neg g_i \wedge \neg(p_i \wedge g_j))$ de Morgan azonosságot használják. A $\neg(p_i \wedge g_j)$ egy NAND kapu függvénye a p_i és g_j jelekre, a $\neg(\neg g_i \wedge (...))$ pedig a $\neg g_i$ és a másik NAND kapu kimenetére nézve NAND kapu függvénye. A cellaszintű tervezés még eredményesebb dolgokra képes. Ma már egyetlen elemi cellában képesek megvalósítani egy több-bites összeadót. Egy 32 bites összeadó kritikus útja az 1 ns körüli késleltetési

⁴¹ (HAN & CARLSON, 1987)

⁴² (SKLANSKY, 1960)

⁴³ (ZHU, CHENG, & GRAHAM, 2005)

⁴⁴ (LING, 1966) (LING, High speed binary adder, 1981)

⁴⁵ (PAI & CHEN, 2004)

tartományban is lehetséges.^{46 47} Ez utóbbi változtatások vagy csak konstans idővel kisebb késletetéssel bírnak, vagy egy 1-nél nem sokkal kisebb, elhanyagolható szorzóval, ezért az $O()$ jelölés szempontjából nincsen változás.

⁴⁶ (WANG, JULIEN, MILLER, WANG, & BIZZAN, 1997)

⁴⁷ (BEWICK, SONG, MICHELI, & FLYNN, 1988)

2.2 Bináris szorzás

A szorzás műveletét digitálisan megvalósítani jóval komplexebb feladat, mint az összeadásé. A probléma mindenesetre elég általános, számos szorzótípus ismert. A jelen dolgozat célja a maximális sebességű megvalósítások továbbfejlesztése, így ebben a fejezetben a fa struktúrájú szorzókról lesz szó. Bár a szorzás kommutatív művelet, a következőkben különbség lesz a szorzó és szorzandó között. A bináris szorzók tipikusan az írásbeli szorzás algoritmusát valósítják meg (papír-ceruza módszer, paper and pencil method). A szorzó minden jegyével külön-külön megszorozzák a szorzandót, és i -szer balra csúsztatják (i -times shifting) annak helyiértékének megfelelően, ami a 2^i -nel történő szorzásnak felel meg.

Legyen $a' = \overline{a_{k-1}a_{k-2} \dots a_0 a_{-1} \dots a_{1-l}a_{-l}} = \sum_{i=k-1 \dots -l} a_i \cdot r^i$ a szorzandó, $b' = \overline{b_{k-1}b_{k-2} \dots b_0 b_{-1} \dots b_{1-l}b_{-l}} = \sum_{i=k-1 \dots -l} b_i \cdot r^i$ a szorzó. $a' \cdot b' = a' \cdot (\sum_{i=k-1 \dots -l} b_i \cdot r^i) = \sum_{i=k-1 \dots -l} b_i \cdot a' \cdot r^i = \sum_{i=k-1 \dots -l} b_i \cdot (\sum_{j=k-1 \dots -l} a_j \cdot r^{j+i})$. Látható, hogy a $+i$ tag a radix kitevőjében képviseli az eltolást. A szorzás eredménye, ha $k + l = n$, akkor $2 \cdot n$ jegyű lesz.

	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	0	·	b_{k-1}
+	0	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	·	b_{k-2}
+	0	0	0	0	...	...	...	...	...	0	0	0	·	...
+	0	0	0	0	0	0	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	·	b_{1-l}
+	0	0	0	0	0	0	0	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	·	b_{-l}
=	c_{2k-2}	c_{2k-3}	c_{2k-4}	c_{2k-5}	...	c_0	c_{-1}	...	c_{3-l}	c_{2-l}	c_{1-l}	c_{-l}		

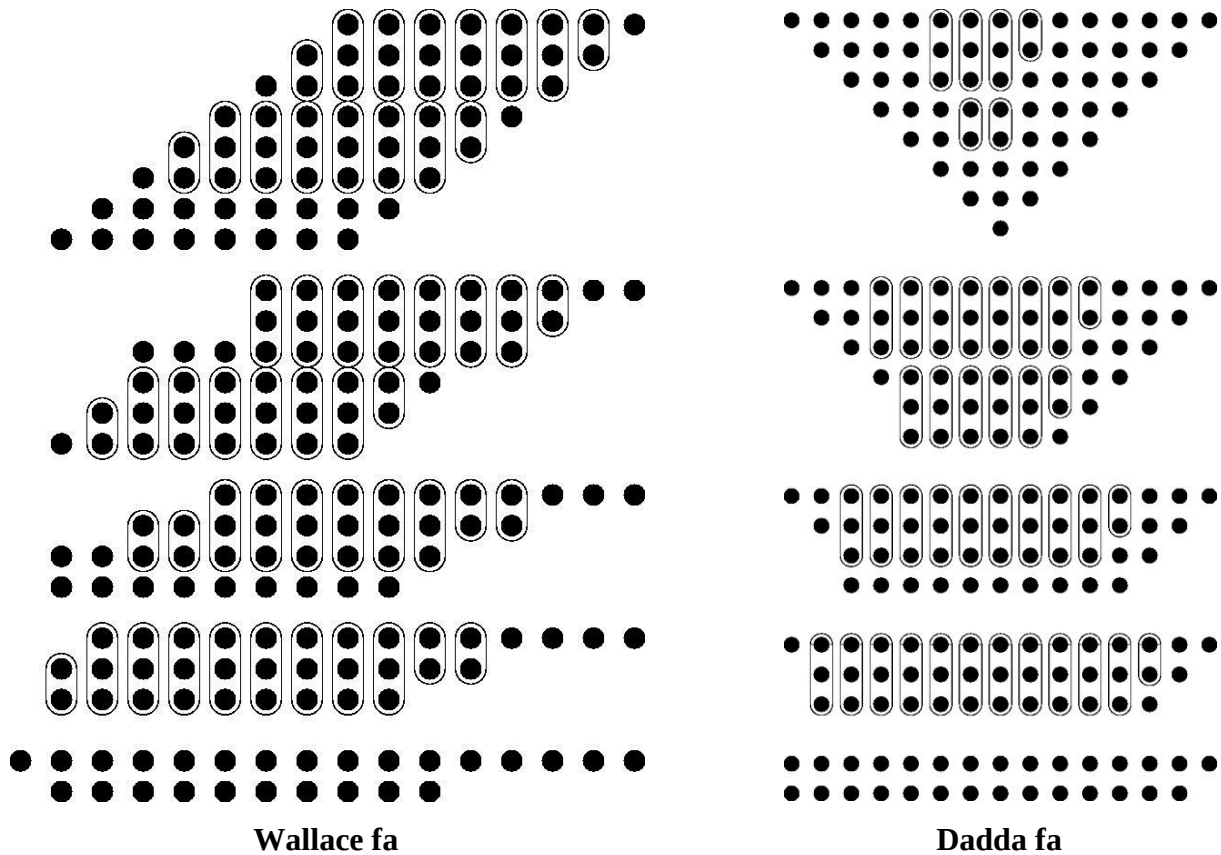
2-7. táblázat: A szorzás művelete, 1. szint

A tömb struktúra minden bitnyi helyébe képzeljünk el vezetékeket. Az egybites szorzást AND kapukkal szokták megvalósítani, aminek egyik bemenete a_i szorzandó számjegy, a másik pedig egy b_j szorzó számjegy. Így kialakul a következő, n darab, $2n$ hosszú szám:

$b_{k-1} \cdot a_{k-1}$	$b_{k-1} \cdot a_{k-2}$	...	$b_{k-1} \cdot a_{1-l}$	$b_{k-1} \cdot a_{-l}$	0	0
0	...	...	...	...	...	0
0	0	$b_{-l} \cdot a_{k-1}$	$b_{-l} \cdot a_{k-2}$	...	$b_{-l} \cdot a_{1-l}$	$b_{-l} \cdot a_{-l}$

2-8. táblázat: A szorzás művelete, 2. szint

Természetesen, ahol $b_j = 0$, ott az egész szám 0 lesz. Ezt az n darab számot gyorsan összeadni az ún. Wallace fa⁴⁸ vagy Dadda fa⁴⁹ struktúrával lehet. Ezekben az ún. carry-save adder (CSA) egységekből álló tömb végzi az összeadást. A CSA olyan összeadó, amely három darab számot két számmá alakít úgy, hogy azok összege a három szám összege legyen. Ennek megvalósítása a legegyszerűbb n darab teljes összeadóból álló tömbbel, ahol a carry in bemenet lesz a harmadik operandus: $a' + b' + c^{in'} = s' + 2 \cdot c^{out'}$, ahol $a' = \sum_{i=k-1 \dots -l} a_i \cdot r^i$, $b' = \sum_{i=k-1 \dots -l} b_i \cdot r^i$. A Wallace/Dadda fa mindig három darab részösszegből csinál kettőt, amíg már csak két darab összeadandó szám marad. Ezeket pedig egy standard bináris összeadó alakítja át a végeredménnyé.



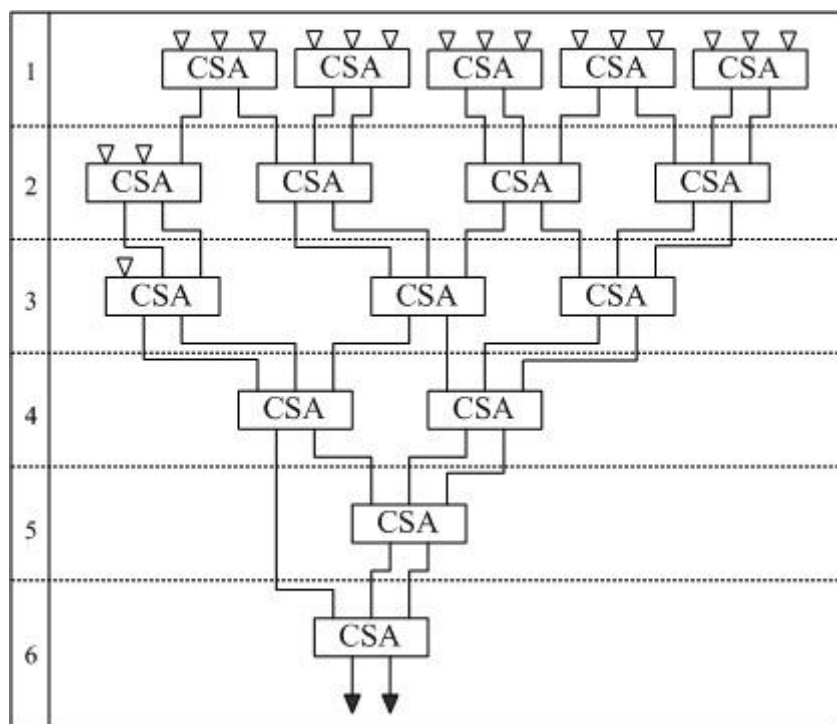
2-9. ábra:⁵⁰ CSA fák

⁴⁸ (WALLACE, 1964)

⁴⁹ (DADDA, 1965)

⁵⁰ http://upload.wikimedia.org/wikipedia/commons/6/65/Wallace_tree_8x8.svg

http://upload.wikimedia.org/wikipedia/commons/3/3e/Dadda_tree_8x8.svg



2-10. ábra: Egy Wallace fa megvalósítása

Előjeles, azaz kettes komplementes ábrázolás esetén elvileg lehetséges lenne az a módszer is, hogy megfigyeljük a negatív tényezőket, átalakítjuk pozitív számmá őket, majd az eredményt visszaalakítjuk az eredeti előjeleknek megfelelően. Ez kettes komplementes esetén túlságosan is lassú művelet.⁵¹ Erre a problémára a *Baugh-Wooley algoritmust*⁵² használják legelterjedtebben, ami a következő: Először vizsgáljuk meg azt az esetet, amikor a szorzandó negatív és a szorzó pozitív. A szorzandóból képzett részösszegeket át tudjuk írni n -kettes komplementes alakból $2n$ -kettes komplementes alakba, ami a 2-7. általánosításának vehető:

	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	0	·	b_{k-1}
+	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	·	b_{k-2}
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	...	...	...	...	...	0	0	0	·	...
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	·	b_{1-l}
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}		·	b_{-l}
=	c_{2k-2}	c_{2k-3}	c_{2k-4}	c_{2k-5}	...	c_0	c_{-1}	...	c_{3-l}	c_{2-l}	c_{1-l}	c_{-l}		

2-11. táblázat: Bináris szorzás negatív szorzandó esetén

⁵¹ (PARHAMI, 2000)

⁵² (BAUGH & WOOLEY, 1973)

A másik eset, amikor a b' szorzó negatív. Ekkor $a' \cdot b'$ szorzat helyett az $a' \cdot (2^n - b') = 2^n \cdot a' - b' \cdot a'$ lesz a végeredmény. Nyilván ki kellene vonni $2^n \cdot a'$ -t belőle. Ezt úgy szokták megoldani, hogy az utolsó részösszeget, azaz a $\overline{a_{k-1}a_{k-2}\dots a_0a_{-1}\dots a_{1-l}a_{-l}} \cdot b_{k-1} \cdot 2^{n-1}$ részösszeget a végén nem hozzáadják az eredményhez, hanem kivonják belőle. Így az pontosan $2 \cdot a' \cdot b_{k-1} \cdot 2^{n-1}$ -et csökken, azaz $2^n \cdot a' \cdot b_{k-1}$ -et. Maga a kivonás kettes komplementben úgy a legcélszerűbb, hogy nem képezzük egyből az $a' \cdot b_{k-1} \cdot 2^{n-1}$ kettes komplementjét, csupán invertáljuk minden számjegyét. Ekkor pontosan 1-gyel lett kevesebb a részösszegünk a kívátnál, de a végső összeadónál beállítjuk a C_{in} carry bemenetet 1-re. Ez minden esetben jó eredményt fog adni.

- ha a szorzó tényleg negatív volt, akkor $b_{k-1} = 1$ miatt $2^n \cdot a' \cdot 1 + 1 = 2^n \cdot a' + 1$ kivonásra kerül a részösszegeből, de ezt a $C_{in} = 1$ bemenettel kompenzálni tudjuk.
- ha a szorzó pozitív volt, akkor $b_{k-1} = 0$ miatt $2^n \cdot a' \cdot 0 + 1 = 1$ kerül kivonásra a részösszegeből, de a végső összeadó $C_{in} = 1$ miatt ezt kompenzálja.

-	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	0	· b_{k-1}
+	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	0	0	0	0	0	· b_{k-2}
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	...	...	...	...	...	0	0	0	· ...
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	0	· b_{1-l}
+	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-1}	a_{k-2}	...	a_{1-l}	a_{-l}	· b_{-l}
=	c_{2k-2}	c_{2k-3}	c_{2k-4}	c_{2k-5}	...	c_0	c_{-1}	...	c_{3-l}	c_{2-l}	c_{1-l}	c_{-l}	

2-12. táblázat: Bináris szorzás negatív számok esetén

Kijelenthető, hogy a kettes komplement szorzás minimális módosítással végrehajtható az előjel nélküli szorzáshoz képest.

2.2.1 A leggyorsabb szorzók

A szorzóáramkörök sebességét az őket alkotó nagysebességű összeadók sebessége határozza meg.⁵³ A szorzóalgoritmusok közül „kis számok” esetén a Wallace és Dadda fák, valamint az arra épülő Booth kódolás⁵⁴ a leggyorsabb. A **Booth algoritmus** röviden a következő. A szorzót előjelesen ábrázoljuk, értéke egybites helyett kétbites lesz. A $b' = \sum_{i=k-1\dots-l} b_i \cdot r^i$ helyett $b' = b_{k-1} \cdot r^k - \sum_{i=k-1\dots-l} (b_i - b_{i-1}) \cdot r^i = \sum_{j=k\dots-l} B_j \cdot r^j$, ahol $B_j =$

⁵³ (BEWICK, 1994)

⁵⁴ (BOOTH, 1951)

$(b_j - b_{j-1}), b_k \stackrel{\text{def}}{=} 2 \cdot b_{k-1}$. A $B_j = \{-2, -1, 0, 1\}$ kétbites szám kettes komplementben ábrázolva. $B_j = \{-2, 0\}$ esetén a megfelelő részösszeget nem adjuk hozzá az összeghez, $B_j = 1$ esetén összeadás történik, $B_j = -1$ esetén kivonás. Az algoritmus minden, egymással szomszédos helyiértéken lévő, 1-es számjegyekből képzett tömböt átalakít két számjegy kivételével nulla tömbbé. Ha $q \leq i \leq p$ indexekre $b_i = 1$, emellett $b_{p+1} = b_{q-1} = 0$, akkor $\sum_{i=p+1 \dots q} (b_i - b_{i-1}) \cdot r^i = (b_{p+1} - b_p) \cdot r^{p+1} + \sum_{j=p \dots q+1} (b_j - b_{j-1}) \cdot r^j + (b_q - b_{q-1}) \cdot r^q = (b_{p+1} - b_p) \cdot r^{p+1} + (b_q - b_{q-1}) \cdot r^q$. Másképpen kifejezve, a számjegy-jelöléssel:

$$\overline{b_{k-1} \dots b_{p+2} 0 1 1 \dots 1 1 0 b_{q-2} \dots b_{-l}} \rightarrow \overline{B_k B_{k-1} \dots B_{p+2} (01)(00)(00) \dots (00)(11) B_{q-1} \dots B_{-l}}.$$

Végül vizsgáljuk meg a késleltetési viszonyokat. Az eddigi szorzóknál megfigyelhető, hogy az elvégzendő elemi szorzások száma két n -bites szorzó és szorzandó esetén $O(n^2)$, ugyanis a szorzó minden számjegye (számuk n) hat a szorzandóra (amely számjegyeinek száma ugyancsak n). Nagyon nagy n esetén ez igen jelentős növekedés még a naiv összeadás $O(n)$ erőforrásigényéhez képest is. Ennek a tényleges csökkentésére lehet használatos a **Karatsuba algoritmus**,⁵⁵ amely az operandusok kettéválasztásával egy $n \cdot n$ szorzás helyett három darab, $\frac{n}{2} \cdot \frac{n}{2}$ szorzást végez el néhány összeadás hozzávételével $O(n^{\log_2 3})$ alatt. Ennek az általánosítása a **Toom–Cook algoritmus**,⁵⁶ vagy algoritmus család, amely nem ketté, hanem tetszőleges m részre osztja az operandusokat, és egy polinom együtthatóinak kiszámolására vezeti vissza a problémát. A legoptimálisabb az $O(n \cdot \log_2 n \cdot 2^{(2 \cdot \log_2 n)^{0.5}})$.⁵⁷ Ezeknél az algoritmusoknál is gyorsabb Fourier transzformációra épülő **Schönhage–Strassen algoritmus**,⁵⁸ amely 35 éven keresztül egyeduralkodó volt a gyors szorzóalgoritmusok terén, a maga $O(n \cdot \log_2 n \cdot \log_2 \log_2 n)$ komplexitásával, mígnem a közelmúltban közzétett **Fürer algoritmus**⁵⁹ nem előzte meg ebben, amely jelenleg is a leggyorsabb ismert szorzóalgoritmus, $n \cdot \log_2 n \cdot 2^{O(\log_2^* n)}$ -val ($\log_2^* n$ itt a $\min\{i \geq 0, \log_2^{(i)} n \leq 1\}$) közelítve az alsó elméleti határ $O(n \cdot \log(n))$ -hoz.⁶⁰ Ennek segítségével 4.7 ns késleltetésű, 32-bites szorzók is készíthetők.⁶¹

⁵⁵ (KARATSUBA & OFMAN, 1960)

⁵⁶ (COOK, 1966)

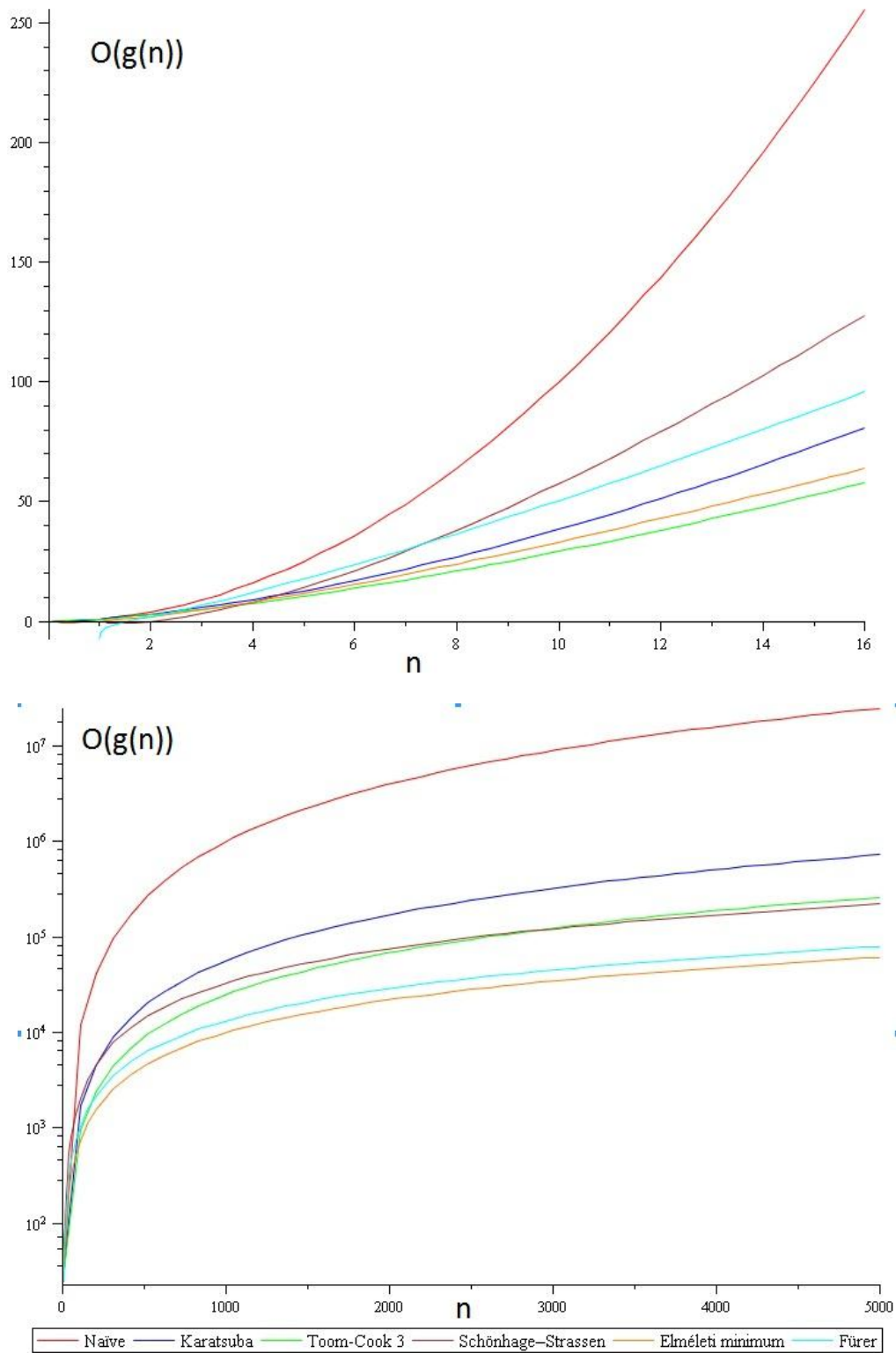
⁵⁷ (KNUTH, 1997)

⁵⁸ (SCHÖNHAGE & STRASSEN, 1971)

⁵⁹ (FÜRER, 2007)

⁶⁰ (SCHÖNHAGE & STRASSEN, 1971)

⁶¹ (GOTO, és mtsai., 1997)

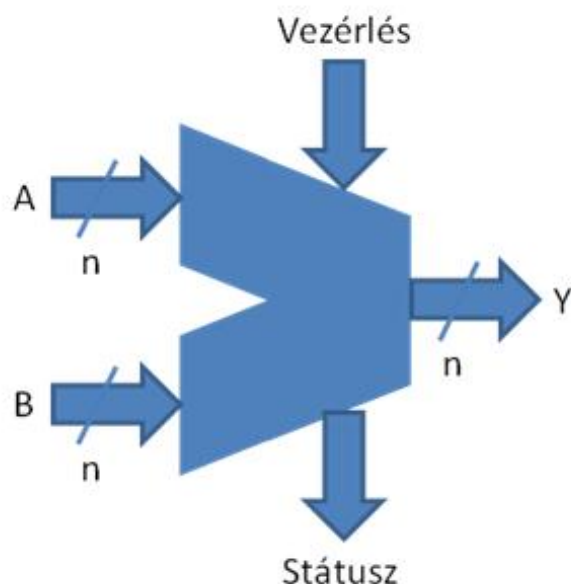


2-13. ábra: A gyors szorzók komplexitás-bemenetszám összehasonlítása

Az ábrán a gyors szorzó algoritmusok elméleti komplexitása láthatóak. A gyakorlatban azonban nehezen meghatározható, hogy milyen bitszélesség után éri meg alkalmazni egy másik algoritmust, a naiv szorzás alacsony bitszám esetén gyorsabb.

2.3 A gyakorlatban használatos ALU architektúrák

Az ALU a processzor legfontosabb műveletvégző egysége. A feladata, hogy két darab, n -bites bemenetből előállítson egy n -bites kimenetet. A kimenet egy, az ALU vezérlése által kiválasztott művelet eredménye kell, hogy legyen. A műveletről szükséges információt a *státusz szó*⁶² tartalmazza (pl. összeadásnál van carry, nullával történő osztás, stb.).



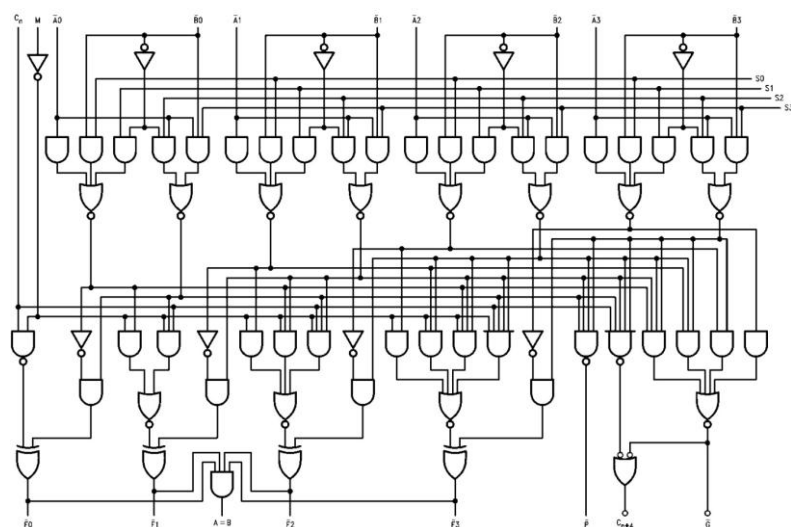
2-14. ábra: Egy ALU ki- és bemenetei

Az ALU-k általában egybites ALU-k tömbjéből, az ún. *bit szeletekből*⁶³ állnak. Ezek párhuzamosan tartalmazzák az összes ALU műveletet, amelyek közül multiplexer választja ki, mely művelet eredménye kerül a kimenetre. Röviden nézzük végig az, ALU-k által megvalósított legfontosabb műveleteket. Ezeknek számos variánsa létezik (pl. nem $A + B$, hanem $A + B + 1$). **Logikai műveletek:** általában a bitenkénti $\wedge, \vee, \oplus, \neg$ Boole függvények kerülnek megvalósításra. Ezeket logikai AND, OR, XOR, NOT kapukkal valósítják meg. **Aritmetikai műveletek:** $+, -, *, /$ a szokásos, egyszerűbb ALU-k nem tartalmazzák a jóval bonyolultabb szorzást és osztást. **Shift/rotate műveletek:** $(x_n, x_{n-1}, \dots, x_2, x_1) \mapsto (V, x_n, x_{n-1}, \dots, x_2)$ és a $(x_n, x_{n-1}, \dots, x_2, x_1) \mapsto (x_{n-1}, \dots, x_2, x_1, W)$ Boole függvényekről van szó. Shift esetén V és W konstans (0 vagy 1). A rotate műveletre $V = x_1$ és $W = x_n$. Szemléltetés gyanánt egyszerű példa egy alacsony bitszámú ALU, ahol még elég a heurisztikus logikai tervezés. Míg a processzorokban ritka, hogy minden logikai műveletet

⁶² (SHIVA, 2000)

⁶³ (TANENBAUM, 2006)

megvalósítsanak, addig a külön IC-ként kapható ALU-k az aritmetikai és logikai műveletek széles tartományát fedik le.⁶⁴



2-15. ábra: Egyszerű 4-bites ALU⁶⁵

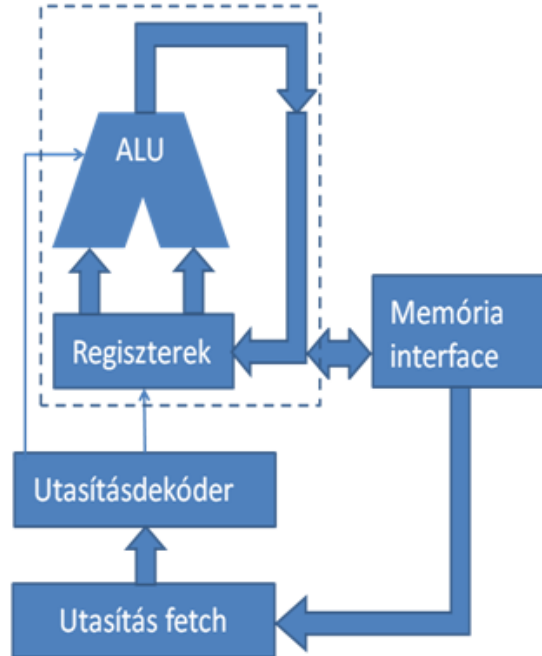
⁶⁴ Egy példa a (Fairchild, 2000)

⁶⁵ (Fairchild, 2000)

2.4 Utasításkészlet

Egy processzor utasításkészlet architektúrája (instruction set architecture, ISA) azon elemi utasítások összessége és végrehajtásuk algoritmusai, amihez a programozó hozzáférhet.

Egy utasítás végrehajtásának vázlatos mikroarchitektúrája látható a 2-16. ábrán.



2-16. ábra:⁶⁶ Az utasításvégrehajtás elemei

A kijelölt részt nevezik ún. *adatútnak* (*data path*⁶⁷). Az ALU-t, valamint be- és kimeneteit tartalmazza. Ennek időzíteni viszonyai szabják meg döntően a processzor sebességét.⁶⁸ Vegyük sorra az utasítás végrehajtás legfőbb lépéseit⁶⁹ (ezek további részlépésekre bonthatóak):

1. Az utasítás felhozatala a memóriából, dekódolása (instruction fetch, decode).
2. Az operandusok felhozatala a memóriából egy regiszterbe (operand fetch).
3. Utasítás végrehajtása.
4. Eredmény kiírása (akkumulátorba, regiszterbe, stb.)

A 3. és 4. lépést nevezik *adatút ciklusnak* (*data path cycle*⁷⁰). A processzor sebességét nagyrészt az adatút ciklus végrehajtási ideje szabja meg.

⁶⁶ (TANENBAUM, 2006)

⁶⁷ „

⁶⁸ „

⁶⁹ „

⁷⁰ „

2.5 A sebesség vizsgálata

Modellezzük az utasításkészletre leforduló, magasabb szintű programok lefutását. Nevezzük *parancsnak* (*command*) a magas szintű utasítást, és *utasításnak* (*instruction*) a gépi szintű utasítást. Minden parancs meghatározott gépi utasításokra fordítódik le, amiknek adott *ciklusidő* szükséges.⁷¹

Mennyiség	Jelölés
A processzor <i>ciklusideje</i>	Δt_{cycle}
Az <i>i</i> -edik <i>parancs j</i> -edik (gépi) <i>utasításának</i> ciklusszáma (Cycles Per Instruction, CPI). ⁷²	$CPI_{i,j}$
Az <i>i</i> -edik <i>parancs</i> (gépi) <i>utasításainak</i> száma	$N_{instruction_i}$
A program <i>parancsainak</i> száma	$N_{command}$
A program <i>futási ideje</i>	$\Delta t_{program}$

2-17. táblázat: A sebesség tényezői

A következő egyszerű, közelítő alakra jutunk: $\Delta t_{program} \cong N_{command} \cdot N_{instruction_{AV}} \cdot CPI_{program_{AV}} \cdot \Delta t_{cycle}$. Tehát a *futási idő* négy fő tényezőtől függ. Vizsgáljuk meg, hogyan befolyásolhatjuk a paramétereket!

Tényező	Fő befolyásoló eszköz		
	Mikroarchitektúra	Utasításkészlet (ISA) ⁷³	Operációs rendszer (OSM) ⁷⁴
Δt_{cycle}	•		
$CPI_{program_{AV}}$	•	•	
$N_{instruction_{AV}}$		•	
$N_{command}$			•

2-18. ábra: Az időzítést meghatározó tényezők és az architektúra elemeinek kapcsolata

Vizsgáljuk meg alaposabban, mi határozza meg a processzor *ciklusidejét*! Mint korábban említésre került, a ciklusidőt az ALU műveletek végrehajtása szabja meg. Ezek a következő komponensekből állnak:⁷⁵

⁷¹ (TANENBAUM, 2006)

⁷² „

⁷³ Instruction Set Architecture

⁷⁴ Operating System machine

⁷⁵ (TANENBAUM, 2006)

1. Vezérlőjelek megjelenése - Δw
2. Egy kiválasztott regiszter előállítja az ALU bemenetét - Δx
3. ALU művelet elvégzése az operandusokon - Δy
4. Az eredmény visszaírása a regiszterekbe – Δz

A ciklusidő ezek összegéből tevődik össze: $\Delta t_{cycle} = \Delta w + \Delta x + \Delta y + \Delta z$

Az elért új eredmények

A matematikai alapok és a gyakorlatban használt logikai áramkörüi funkciók ismertetése után lássuk a jelen dolgozatban tárgyalt, elért eredményeimet:

- Kidolgoztam a Boole algebra **Boole modul** reprezentációját. A modul a vektortér fogalom általánosítása, hiszen vektortér csak algebrai test felett létezhet, míg modul algebrai gyűrű felett van definiálva. Az új, általam felfedezett reprezentációt úgy kezeltem, mint egy saját matematikai keretrendszert, amely segítségével a dolgozatban előforduló elméleti és gyakorlati problémák egyaránt kezelhetők. Mivel ez az én saját elméletem, a legtöbb tételt, definíciót nem szakirodalomból veszem, hanem saját magam alkotom meg annak megfelelően, hogy az így felépített matematikai modell illeszkedjen a felvetett problémákra (áramkör-minimalizálás, stb.). Mivel a tételeket magam alkotom meg, a bizonyítások is az én munkám részét képezik. Ezt az új modellem megértésének az elősegítése, és az elméletem matematikai korrektsége végett teszem.
- Definiáltam egy hardverleírási elvet, a **Dualitási szabályt**, hozzá egy hibrid absztrakciós szintet, az **ML-RTL** szintet.
- Megterveztem funkcionálisan egy logikai áramkör családot, az **IBZ családot**.
- Megalkottam egy új ALU architektúrát, az **IBZ ALU**-t.
- Leírtam az IBZ ALU-hoz illeszkedő új utasításkészletet, az **IBZ ISA**-t.

Szimulációs eredmények

- Az ML-RTL leírás VHDL nyelven történő analízise, összehasonlítás a magasabb szintű leírásokkal
- FPGA szintézis és szimulációs módszerekkel megállapítottam, hogy az ML-RTL szinten leírt IBZ ALU helyfoglalása sebessége az RTL szinten leírt standard ALU architektúráéhoz képest kisebb.
- Az IBZ ISA implementációja egy AVR mikroprocesszorba.

3 A Boole algebra új reprezentációja

Az előzőekben tárgyalásra került a Boole algebra, mint az elsődleges matematikai modell a logikai áramkörök működésére. Mint minden algebrának, a Boole algebrának is sok megjelenítése (*representation*) létezik. Azt a speciális esetet, amikor egy absztrakt algebra elemeinek reprezentációja modulokkal vagy vektorterekkel (speciális modulok), műveleteinek reprezentációja pedig a nehezen kezelhető absztrakt algebra helyett a lineáris algebrával (mátrixokkal) történik, reprezentáció elméletnek (*representation theory*)⁷⁶ nevezik. Az alábbiakban bemutatásra kerül a Boole algebra egy, általam kidolgozott reprezentációja, amelynek elméleti igazolásával megalkotásra került egy újelvű logikai áramkörcsalád, az IBZ. A modell segítségével definiáltam egy új absztrakciós szintet, emellett az elméleti alapot nyújt a dolgozat további részének kifejtéséhez.

⁷⁶ (FULTON & HARRY, 1991)

3.1 A Boole modul

A $B\{\{0,1\}, \vee, \wedge, \neg, 0, 1\}$ Boole algebrával szokás modellezni a digitális biteket, mint egy $b \in B = \{0,1\}$ Boole skalár. Több bitet együtt kezelve, a reprezentáció bit tömbbé módosul: $b \in B^n, b_i \in B, b = (b_1, \dots, b_n)$. Az a probléma ezzel a megjelenítéssel, hogy nehézkes a műveletek definiálása. Definiáljunk a Boole gyűrű és a Boole test feletti komplexebb struktúrákat!

Definíció: A Boole modul egy $\{\mathcal{B}, \lambda\}$ modul a $B\{\{0,1\}, +_B, \cdot, 0_B, 1_B\}$ Boole gyűrű felett, ahol $\mathcal{B}\{\{\beta_1, \beta_0\}, +_{\mathcal{B}}, -, \beta_0\}$ Abel-csoport, β_0 és β_1 rendre a '0' és '1' bitek reprezentációi. A gyűrű homomorfizmust lehet skalárral való szorzás műveletként is definiálni: $\lambda \equiv (\cdot): B \times \mathcal{B} \rightarrow \mathcal{B}, a \cdot \beta_i \mapsto \beta_j; a, b \in B; \beta_i, \beta_j \in \mathcal{B}$. Így a Boole modul a következő: $\{\mathcal{B}, \cdot\}$. A szorzat bevezetésével új alakra lehet hozni a modul-axiómákat a $\lambda(a) \rightarrow a \cdot$ megfeleltetés segítségével: $a \cdot (\beta_i +_{\mathcal{B}} \beta_j) = a \cdot \beta_i +_{\mathcal{B}} a \cdot \beta_j; (a +_B b) \cdot \beta_i = a \cdot \beta_i +_{\mathcal{B}} b \cdot \beta_i; (a \cdot b) \cdot \beta_i = a \cdot (b \cdot \beta_i); 1_B \cdot \beta_i = \beta_i$. Fontos különbséget tenni a $+_B$ és a $+_{\mathcal{B}}$ műveletek között. Az előbbi a B Boole gyűrű művelete, az utóbbi a $\mathcal{B}$ Abel-csoporté. A továbbiakban ezeket külön nem jelölöm, értelemszerűen az összeadandók szabják meg, melyikről van szó.

Definíció: A Boole vektortér vagy „bittér” egy $\{\mathcal{B}, \cdot\}$ vektortér a $B\{\{0,1\}, \vee, \wedge, \neg, 0, 1\}$ Boole test felett, a Boole modul minden tulajdonsága vonatkozik rá.

A Boole modul ábrázolható a $B \times B$ Boole skalárok Descartes szorzataként is, amelyben a két bit: $\beta_0 = 1 \times 0$ a '0' bit és $\beta_1 = 0 \times 1$ az '1' bit a konvenció szerint, rendezett kettessel (2-tuple) kifejezve $\beta_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \beta_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

A biteken végzett művelet így reprezentálható $B \times B \rightarrow B \times B$ mátrixként, és a mátrixvektor szorzással számítható a $\{B, +, \cdot, 0, 1\}$ Boole gyűrű felett. Vizsgáljuk meg, miben alkalmasabb a Boole modul a Boole algebránál a logikai áramkörök leírására! Az egybites Boole műveletek összevont igazságtáblája:

Bemenet	Identitás	Negálás	Kontradikció	Tautológia
0	0	1	0	1
1	1	0	0	1

3-1. táblázat: Az egybites Boole függvények igazságtáblája

Keressük meg azokat az M mátrixot, amelyre, ha $b \in \mathcal{B}$ jelöl egy bitet, és az ezen alkalmazott Boole függvény, $F: \mathcal{B} \rightarrow \mathcal{B}, b \mapsto c$ esetén $M \cdot b = c$. A feltételeink a következők:

- Minden egybites művelet mátrixa 2×2 kell, hogy legyen, hiszen egy bitet egy másik bitbe képez: $M: B \times B \rightarrow B \times B$
- Egy bitet reprezentáló vektorból egy másik, bitet reprezentáló érvényes vektort csinál.
- Mivel csak $B = \{0,1\}$ Boole skalárból álló elemeik vannak, ezek a mátrixok ún. *Boole mátrixok* vagy *logikai mátrixok*.⁷⁷

Az *identitás*, amely minden bitet önmagába képez, triviális módon az egységmátrixszal írható le, ez egy tetszőleges $B \times B$ vektort érintetlenül hagy. A *negálás* művelete a következő

tulajdonsággal rendelkezik: $\neg: \beta_0 \mapsto \beta_1, \beta_1 \mapsto \beta_0, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mapsto \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \mapsto \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

Nem nehéz belátni, hogy ezt a feltételt csak a $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ mátrix teljesíti. A kontradikció és a tautológia, melyek tetszőleges bitet β_0 -ba illetve β_1 -be képeznek, mátrixaikra pedig belátható,

hogy rendre $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ és $\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$ lesznek. Foglalkozzunk össze az egybites Boole műveletekkel:

Művelet	Mátrix	Művelet	Mátrix
Kontradikció	$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$	$\neg A$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
A	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	Tautológia	$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$

3-2. táblázat: Az egybites Boole függvények mátrixai

Ahhoz, hogy két bitet összefűzzünk egy kétbites busszá, a Descartes szorzat ($\times$) helyett használjuk a *tenzorszorzás*⁷⁸($\otimes$) műveletét.

Definíció: A *skalárszorzat* a $\langle \mid \rangle: B^n \times B^n \rightarrow B, a \times b \rightarrow \begin{cases} 1, & \text{ha } a = b \\ 0, & \text{ha } a \neq b \end{cases}$ művelet.

Definíció: Egy $a \in B^n$ szó *duálisa*⁷⁹ (vigyázat: nem De Morgan duálisa!) az az $\mathbf{a} \in B^n \rightarrow B$, melyre $\mathbf{a}(a) = \langle a|a \rangle$.

Definíció: Egy B Boole modul duális modulja az a B^* a *duális szavakból* képzett modul.

Definíció:⁸⁰ Legyen R gyűrű, M_1, M_2 modulok R felett. M_1 és M_2 *tenzorszorzata* R felett az a $\otimes_R: M_1 \times M_2 \rightarrow L = (M_1 \otimes_R M_2)$ bilineáris művelet, amelyre L modul R felett, és

⁷⁷ (KIM, 1982)

⁷⁸ Néha szokás a tenzorszorzatot mátrixok esetén Kronecker szorzatnak, vektor és transzponált vektor szorzása esetén külső vagy diadikus szorzatnak is nevezni.

⁷⁹ (HALMOS, 1974)

tetszőleges N R feletti modulra és $\varphi: M_1 \times M_2 \rightarrow N$ bilineáris leképezésre létezik egy speciális $\psi: N \rightarrow L$ leképezés, melyre $\psi \circ \otimes_R = \varphi$.

Definíció:⁸¹ Azt mondjuk, hogy egy $T \in (\otimes^n M^*) \otimes (\otimes^m M)$ egy (n, m) súlyú *tenzor* M modul felett.

Jelölés: $\otimes: \mathcal{B}^n \times \mathcal{B}^m \rightarrow \mathcal{B}^{\otimes(n+m)}, \mathcal{B}^{\otimes k} \stackrel{\text{def}}{=} \mathcal{B}^k$.

Jelölés: $\otimes_n: \mathcal{B}_1 \times \mathcal{B}_2 \times \dots \times \mathcal{B}_n \rightarrow \mathcal{B}^{\otimes n} \stackrel{\text{def}}{=} \mathcal{B}^n, a_1 \times a_2 \times \dots \times a_n \mapsto a_1 \otimes a_2 \otimes \dots \otimes a_n$.

Speciális esetként tegyük fel az alábbiakat: $\otimes_1 = I: \mathcal{B} \rightarrow \mathcal{B}^{\otimes_1} \stackrel{\text{def}}{=} \mathcal{B}^1 = \mathcal{B}, a \mapsto a$, $\otimes_0: \mathcal{B} \rightarrow \mathcal{B}^{\otimes_0} \stackrel{\text{def}}{=} \mathcal{B}^0 = B = \{0,1\}$. A $\mathcal{B}^0 = B = \{0,1\}$ nem más, mint a Boole modul skalármezője. A tenzorszorzatok koordinátakomponensenként a következő módon számolhatók. Ha $v = v_0 \cdot \beta_0 + v_1 \cdot \beta_1$, és $w = w_0 \cdot \beta_0 + w_1 \cdot \beta_1$, akkor $v \otimes w \stackrel{\text{def}}{=} v_0 \cdot w_0 \cdot \beta_0 \otimes \beta_0 + v_0 \cdot w_1 \cdot \beta_0 \otimes \beta_1 + v_1 \cdot w_0 \cdot \beta_1 \otimes \beta_0 + v_1 \cdot w_1 \cdot \beta_1 \otimes \beta_1$.

Ez ezzel ekvivalens: $v = \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}$ és $w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ bitek, akkor $v \otimes w \stackrel{\text{def}}{=} \begin{bmatrix} v_0 \cdot w_0 \\ v_0 \cdot w_1 \\ v_1 \cdot w_0 \\ v_1 \cdot w_1 \end{bmatrix}$.

A '0' bit jele β_0 , az '1' bit jele a β_1 . ha ezek duálisait rendre β_0 és β_1 jelölöm, akkor látható, hogy a $\otimes_0: a \mapsto \beta_1(a) = \langle \beta_1 | a \rangle$. Szemléletesen, egy bit „vektorának” duálisa megfelel a „transzponált vektornak”:

β_0	β_1	β_0	β_1
$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \end{bmatrix}$

3-3. táblázat: A bitek és duálisai

A tenzorszorzást duális függvényekre a következőképpen definiálhatjuk:

Egy $a, b, c, d \in \mathcal{B}^n$, $\mathbf{a}, \mathbf{b} \in \mathcal{B}^n \rightarrow \mathcal{B}^0$ az a és b duálisai. $a \otimes \mathbf{a}: \mathcal{B}^n \rightarrow \mathcal{B}^n$, $b \mapsto a \cdot \mathbf{a}(b) = a \cdot \langle \mathbf{a} | b \rangle$, $\mathbf{a} \otimes \mathbf{b}: \mathcal{B}^{2n} \rightarrow B$, $c \otimes d \mapsto \mathbf{a}(c) \cdot \mathbf{b}(d) = \langle \mathbf{a} | c \rangle \cdot \langle \mathbf{b} | d \rangle$. A logikai függvények mátrixát algebrai alakban is ki lehet fejezni a β_0 és β_1 bitekkel, és azok β_0 és β_1 duálisának segítségével.⁸² Ennek oka következő izomorfia: $(\mathcal{B}^* \otimes \mathcal{B}^*) \otimes \mathcal{B} \approx \mathcal{B}^2 \rightarrow \mathcal{B}$. Néhány példa⁸³: Egybites Boole függvények (itt T a tautológia, K a kontradikció):

$$I = \beta_0 \otimes \beta_0 + \beta_1 \otimes \beta_1$$

⁸⁰

⁸¹

⁸² (MIZRAJI, 1992)

⁸³ (MIZRAJI, 1992)

$$\neg = \beta_0 \otimes \beta_0 + \beta_1 \otimes \beta_1$$

$$T = \beta_1 \otimes \beta_0 + \beta_1 \otimes \beta_1$$

$$K = \beta_0 \otimes \beta_0 + \beta_0 \otimes \beta_1$$

Néhány kétbites Boole függvény:

$$\wedge = \beta_0 \otimes (\beta_0 \otimes \beta_0 + \beta_0 \otimes \beta_1 + \beta_1 \otimes \beta_0) + \beta_1 \otimes (\beta_1 \otimes \beta_1)$$

$$\neg \wedge = \beta_1 \otimes (\beta_0 \otimes \beta_0 + \beta_0 \otimes \beta_1 + \beta_1 \otimes \beta_0) + \beta_0 \otimes (\beta_1 \otimes \beta_1)$$

$$\vee = \beta_0 \otimes (\beta_0 \otimes \beta_0) + \beta_1 \otimes (\beta_0 \otimes \beta_1 + \beta_1 \otimes \beta_0 + \beta_1 \otimes \beta_1)$$

$$\neg \vee = \beta_1 \otimes (\beta_0 \otimes \beta_0) + \beta_0 \otimes (\beta_0 \otimes \beta_1 + \beta_1 \otimes \beta_0 + \beta_1 \otimes \beta_1)$$

Ezekről belátható, hogy valóban a kívánt logikai függvényt alkotják. A jelölés kevésbé átlátható, ezért az általam javasolt egyszerűsített jelölés: $I = [\beta_0, \beta_1], \neg = [\beta_1, \beta_0], T = [\beta_1, \beta_1], K = [\beta_0, \beta_0], \wedge = [\beta_0, \beta_0, \beta_0, \beta_1], \neg \wedge = [\beta_1, \beta_1, \beta_1, \beta_0], \vee = [\beta_0, \beta_1, \beta_1, \beta_1], \neg \vee = [\beta_1, \beta_0, \beta_0, \beta_0]$. Ebből a jelölésből azonnal kiolvasható a mátrix alak.

Példa: $\wedge = [\beta_0, \beta_0, \beta_0, \beta_1] = \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

3.1.1 A Boole függvények kifejezése mátrixokkal

Első lépésben a két bemenetű logikai kapuk $F: B^2 \rightarrow B$ függvényosztályát szeretnénk mátrixokkal reprezentálni. Ezek két bitből képeznek egyet, azaz $M: (B \times B) \otimes (B \times B) \rightarrow B \times B$, következésképpen egy 2×4 -es mátrixnak kell reprezentálnia a műveleteket.

Egy 2×4 -es mátrix általánosan felírható az $\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \end{bmatrix}, \forall m \in B$ alakban.

Egy ilyen mátrixot alkalmazva egy kétbites szóra:

$$\begin{matrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} \\ * \\ \begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} & m_{1,4} \\ m_{2,1} & m_{2,2} & m_{2,3} & m_{2,4} \end{bmatrix} \end{matrix} \begin{bmatrix} u_0 \cdot m_{1,1} + u_1 \cdot m_{1,1} + u_2 \cdot m_{1,1} + u_3 \cdot m_{1,1} \\ u_0 \cdot m_{2,1} + u_1 \cdot m_{2,1} + u_2 \cdot m_{2,1} + u_3 \cdot m_{2,1} \end{bmatrix}$$

Tudjuk, hogy az u koordinátaelemei közül csak egy darab lesz 1-es. Legyen ez a k indexű komponens. Ekkor az eredmény a következő lesz:

$$\begin{bmatrix} u_k \cdot m_{1,k+1} \\ u_k \cdot m_{2,k+1} \end{bmatrix} = \begin{bmatrix} m_{1,k+1} \\ m_{2,k+1} \end{bmatrix}.$$

Az így kapott „vektornak” egy bitet kell reprezentálnia, ezért csak az egyik eleme lehet 1, a másiknak 0-nak kell lennie. Ezzel meg is kaptuk a feltételt a kapuk mátrixára: olyan 2×4 -es mátrixok, amelyek oszlopaiban kizárólag egy 1-es szerepelhet, másik elem 0. Ez a feltétel megfelel egy bit reprezentálásának. Így egy kapumátrix felírható a $(\beta_i, \beta_j, \beta_k, \beta_l)$ alakban, ahol $i, j, k, l = \{0, 1\}$. Ezek szerint négy darab, szabadon megválasztható m paraméterünk van, mert oszloponként egy független eleme van a mátrixnak. Ez 2^4 különböző mátrixot határoz meg, tehát lefedtük az összes kétbites műveletet. A továbbiakban használok a következő jelölést: $m_{1,k+1} \equiv m_k$, ahol k számot binárisan ábrázolom (azaz $0 \rightarrow 00$, $1 \rightarrow 01$, $2 \rightarrow 10$, $3 \rightarrow 11$)

$$\begin{bmatrix} v_0 \cdot w_0 \\ v_0 \cdot w_1 \\ v_1 \cdot w_0 \\ v_1 \cdot w_1 \end{bmatrix} \cdot \begin{bmatrix} m_{00} + 1 & m_{01} + 1 & m_{10} + 1 & m_{11} + 1 \\ m_{00} & m_{01} & m_{10} & m_{11} \end{bmatrix} = \begin{bmatrix} (m_{00} v_0 w_0 + m_{01} v_0 w_1 + m_{10} v_1 w_0 + m_{11} v_1 w_1) + 1 \\ m_{00} v_0 w_0 + m_{01} v_0 w_1 + m_{10} v_1 w_0 + m_{11} v_1 w_1 \end{bmatrix}$$

3-4. ábra: A logikai művelet mint mátrix-vektor szorzás

Az összes kétbites Boole művelet mátrixa:

Művelet	Mátrix	Művelet	Mátrix
Kontradikció	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$	$\neg(A \vee B)$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$
$A \wedge B$	$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\neg(A \oplus B)$	$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$
$\neg(A \Rightarrow B)$	$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\neg B$	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$
A	$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$	$A \Leftarrow B$	$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$
$\neg(A \Leftarrow B)$	$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$	$\neg A$	$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$
B	$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$	$A \Rightarrow B$	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$
$A \oplus B$	$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$	$\neg(A \wedge B)$	$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$
$A \vee B$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$	Tautológia	$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$

3-5. ábra: A kétbites Boole függvények mátrixai

3.2 A Boole függvények reprezentációja

Tekintsük az $F: \mathcal{B}^n \rightarrow \mathcal{B}$ függvényeket. A teljesen általános eset, azaz az $G: \mathcal{B}^n \rightarrow \mathcal{B}^m$ eset visszavezethető az előbbire úgy, hogy m darab F függvényt használunk. Ha bevezetjük az $X_{n,m}: \mathcal{B}^n \rightarrow \mathcal{B}^{n \cdot m}, a \mapsto a \otimes a \otimes a \otimes \dots \otimes a = \bigotimes_{i=1..m} a, a \in \mathcal{B}^n$ műveletet, akkor $G: \mathcal{B}^n \rightarrow \mathcal{B}^m, G = (F_1 \otimes F_2 \otimes F_3 \otimes \dots \otimes F_m) \circ X_{n,m}$.

3.2.1 A redukált tenzorszorzat

Kettőnél több bit esetén felmerül a kérdés, hogy van-e értelme bizonyos műveleteknek. Az a négybites művelet például, amelyik csak kettő bemenettől függ, például azok AND kapcsolata, nem használja ki a négybites műveletvégzés lehetőségeit: $F(a, b, c, d) \rightarrow a \wedge b; a, b, c, d \in \mathcal{B}$. Ezek a műveletek előállíthatóak *tenzorszorzatként*, de nem a szokványos értelemben, hiszen több, egyértékű függvény *tenzorszorzata* többértékű: $F: \mathcal{B}^p \rightarrow \mathcal{B}, G: \mathcal{B}^r \rightarrow \mathcal{B}, F \otimes G: \mathcal{B}^{p+r} \rightarrow \mathcal{B}^2$. Ezért be kell vezetni az ún. redukciós operátort, ami semmi mást nem csinál, mint egy több-bites szóból egy vezetékét kiválaszt. Ez nem lesz más, mint az általunk korábban vizsgált „A” és „B” Boole függvények, amelyek mátrixai a 3-2. táblázatban találhatóak:

Definíció: *Kétféleképp redukciós operátor* a $A: \mathcal{B}^2 \rightarrow \mathcal{B}, (a, b) \mapsto a; a, b \in \mathcal{B}, B: \mathcal{B}^2 \rightarrow \mathcal{B}, (a, b) \mapsto b; a, b \in \mathcal{B}$.

Az „A” és „B” Boole függvények általánosíthatóak n bitre is.

Definíció: *N-bites redukciós operátor* a $A_k: \mathcal{B}^n \rightarrow \mathcal{B}, (a_1, a_2, \dots, a_l, \dots, a_n) \mapsto a_k; a_l \in \mathcal{B}, 1 \leq k \leq n$.

Definíció: α *redukciós operátor*, melyre ha $\alpha \subseteq (a_1, a_2, \dots, a_l, \dots, a_n)$, akkor $A_\alpha: \mathcal{B}^n \rightarrow \mathcal{B}, (a_1, a_2, \dots, a_l, \dots, a_n) \mapsto \alpha$.

Definíció: A műveletek α -*redukált tenzorszorzata*: $F: \mathcal{B}^p \rightarrow \mathcal{B}^q, G: \mathcal{B}^r \rightarrow \mathcal{B}^s, F \otimes G: \mathcal{B}^{p+r} \rightarrow \mathcal{B}^{q+s}, A_\alpha: \mathcal{B}^{q+s} \rightarrow \mathcal{B}^{|\alpha|}, F \otimes_\alpha G = A_\alpha \circ (F \otimes G): \mathcal{B}^{p+r} \rightarrow \mathcal{B}^{|\alpha|}$.

Tipikusan az $|\alpha| = 1$ esetet vizsgálom. Vegyük figyelembe a *tenzorszorzás* alábbi szabályát:

Ha $F: \mathcal{B}^p \rightarrow \mathcal{B}^q, G: \mathcal{B}^r \rightarrow \mathcal{B}^s, u \in \mathcal{B}^p, v \in \mathcal{B}^r$, akkor $(F \otimes G)(u \otimes v) = F(u) \otimes G(v)$.

Jelölje f az $(F \otimes G)$ F -re vonatkozó kimeneteit, míg g a G -re vonatkozóakat. Ekkor

$(F \otimes_f G)(u \otimes v) = F(u)$, és $(F \otimes_g G)(u \otimes v) = G(v)$. Most vegyük elő az eredeti

példánkat: $F(a, b, c, d) \rightarrow a \wedge b; a, b, c, d \in \mathcal{B}$. Ha bevezetjük az $AND(a, b,) \rightarrow a \wedge b; a, b \in \mathcal{B}$ jelölést, akkor $F = AND \otimes_1 I$, ahol I lehet tetszőleges kétféleképp redukált művelet, itt az

identitást jelenti. Ezzel a példával az került szemléltetésre, hogy azok a műveletek, amik

kihasználják egy függvény összes bemenetét, azok nem írhatóak fel az $F \otimes_\alpha G$ alakban, míg minden más igen. A *redukált tenzorszorzat* általánosítására egy mód adódik.

Definíció: A H -szorzat a következő: $F: \mathcal{B}^p \rightarrow \mathcal{B}^q, G: \mathcal{B}^r \rightarrow \mathcal{B}^s, F \otimes G: \mathcal{B}^{p+r} \rightarrow \mathcal{B}^{q+s}, H: \mathcal{B}^{q+s} \rightarrow \mathcal{B}^t, F \otimes_H G = H \circ (F \otimes G): \mathcal{B}^{p+r} \rightarrow \mathcal{B}^t$.

3.2.2 Összetett logikai függvények generálása

$$\Lambda_n: \mathcal{B}^n \rightarrow \mathcal{B}, a \mapsto a_1 \wedge a_2 \wedge \dots \wedge a_n \doteq \bigwedge_{j=1..n} a_j \doteq \Lambda_n a, a \in \mathcal{B}^n$$

$$\vee_n: \mathcal{B}^n \rightarrow \mathcal{B}, a \mapsto a_1 \vee a_2 \vee \dots \vee a_n \doteq \bigvee_{j=1..n} a_j \doteq \vee_n a, a \in \mathcal{B}^n$$

Speciális esetek:

$$n = 1: \Lambda_1 = I: \mathcal{B} \rightarrow \mathcal{B}, a \mapsto a, \vee_1 = I: \mathcal{B} \rightarrow \mathcal{B}, a \mapsto a$$

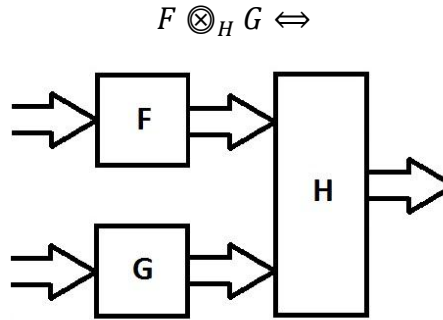
$$n = 0: \Lambda_0: \mathcal{B} \rightarrow \mathcal{B}, \{0,1\} \mapsto \beta_1, \vee_0: \mathcal{B} \rightarrow \mathcal{B}, \{0,1\} \mapsto \beta_0$$

További jelölések az $\Lambda_n a$ -ra az $\inf a$, az $\vee_n a$ -ra a $\sup a$.⁸⁴ Az ilyen műveletek leírásához az ún. H -szorzathoz kell fordulnunk:

Ha $H: \mathcal{B}^2 \rightarrow \mathcal{B}, a \mapsto a_1 \wedge a_2, a \in \mathcal{B}^2$, akkor $\Lambda_n = I \otimes_H I \otimes_H \dots \otimes_H I = I \otimes_\wedge I \otimes_\wedge \dots \otimes_\wedge I$.

ha $H: \mathcal{B}^2 \rightarrow \mathcal{B}, a \mapsto a_1 \vee a_2, a \in \mathcal{B}^2$, akkor $\vee_n = I \otimes_H I \otimes_H \dots \otimes_H I = I \otimes_\vee I \otimes_\vee \dots \otimes_\vee I$.

A zárójelezés tetszőleges ezen műveletek kommutativitása miatt. Vegyük észre, hogy a H -szorzat kombinációs hálózattal az alábbi módon szemléltethető:



3-6. ábra: A H -szorzat szemléltetése áramkörökkel

Ez arrafele mutat, hogy a H -szorzat algebrával részben leírhatóvá válnak a kombinációs hálózatok. Ellentétben a szokásos Boole modullal, itt nem $a \in \mathcal{B}$ elemek és az ezeken végrehajtható $M: \mathcal{B}^2 \rightarrow \mathcal{B}$ binér műveleteket vesszük alapul, hanem a $M: \mathcal{B} \rightarrow \mathcal{B}$ elemeket és az ezeken végrehajtható $\otimes_H: (\mathcal{B}^n \rightarrow \mathcal{B})^2 \rightarrow (\mathcal{B}^{2n} \rightarrow \mathcal{B})$. Az $(M, \otimes_H)$ rendezett kettesből képzett algebrai kifejezések alkalmasak logikai áramkör leírására. Segítségükkel az egy- és

⁸⁴ (VLADIMIROV, 2002)

kétbites Boole műveletek mátrixaiból egyszerűen generálhatóak az újak. Emlékezzünk vissza a kapumátrixok megalkotására. Ott csak heurisztikusan tudtunk következtetni a mátrixok alakjára. Ez most már automatikussá vált. Az általánosságban elmondható, hogy a bemenet n -bites, a kimenet egybites lesz, tehát a mátrix 2×2^n -es lesz. Az is világos, hogy továbbra is csak egy darab '1' kerülhet minden oszlopba. Így az ilyen mátrixok száma 2^{2^n} lesz, és mind különböző, így a korábbi bizonyítási módszerekkel belátható, hogy valóban az n -bites Boole függvényeket reprezentálják.

3.2.3 A több-bites műveletek felhasználhatósága

Kérdésünk a következő: Az $F: \mathcal{B}^n \rightarrow \mathcal{B}$ függvények közül mennyi nem állítható elő az $F \otimes_\alpha G$ alakban, azaz mennyi függ az összes bemenettől, mekkora a nem *elfajult* függvények száma? Ha egy halmaz számosságát $|A|$ jelöljük, akkor $|F: \mathcal{B}^n \rightarrow \mathcal{B}| = 2^{2^n}$. Ez egyszerű leszámplálási problémaként kapható meg, az n darab bemenettel 2^n különböző bemeneti kombináció lehetséges, és ezekhez kell egyenként a $\{0,1\}$ halmazból rendelni egy-egy elemet. Azaz a 2^n hosszú bitszavak számát keressük, ez pedig a fent említett eredmény. Ebből még le kell vonni a csak $k < n$ bittől függő műveleteket. Az $F: \mathcal{B}^n \rightarrow \mathcal{B}$ halmazban $\binom{n}{k} \cdot 2^{2^k}$ azoknak a száma, amelyek *maximum* k darab bemenettől függenek (azaz $\exists G: \mathcal{B}^k \rightarrow \mathcal{B}$, melyre $F = G \otimes_g I$). Ez is könnyen belátható, az n bemenetből $\binom{n}{k}$ féleképpen választhatunk ki k darabot, és ezeken a biteken a korábbiak miatt 2^{2^k} művelet lehetséges. Jelöljük $\Phi(n)$ -nel a keresett számokat.

Nyilvánvaló, hogy $\Phi(n) = \binom{n}{n} \cdot 2^{2^n} - \sum_{k=0}^{n-1} \binom{n}{k} \cdot \Phi$, az egyes részösszegek pedig a következők:

$$\Phi(0) = 2^{2^0} = 2,$$

$$\Phi(1) = 2^{2^1} - \Phi(0) = 2, \quad \Phi(2) = 2^{2^2} - \binom{2}{1}\Phi(1) - \binom{2}{0}\Phi(0) = 10,$$

$$\Phi(3) = 2^{2^3} - \binom{3}{2}\Phi(2) - \binom{3}{1}\Phi(1) - \binom{3}{0}\Phi(0) = 256 - 30 - 6 - 2 = 218,$$

$$\Phi(4) = 2^{2^4} - \binom{4}{3}\Phi(3) - \binom{4}{2}\Phi(2) - \binom{4}{1}\Phi(1) - \binom{4}{0}\Phi(0) = 65536 - 4 \cdot 256 - 6 \cdot 10 - 4 \cdot 2 - 1 \cdot 2 = 64502.$$

Láthatjuk, hogy míg $n = 1$ -nél $\Phi(1) = 0.5000 \cdot |F: \mathcal{B}^1 \rightarrow \mathcal{B}|$, addig $n = 4$ esetén $\Phi(4) \cong 0.9842 \cdot |F: \mathcal{B}^4 \rightarrow \mathcal{B}|$, és ez az arány a bitszám növekedésével meredeken növekszik. A számok ugyanakkor óvva intenek attól, hogy osztályzni kezdjük a többbites műveleteket, mint ahogy tettük azt a kétbiteseknél: csak a 4-bitesek között már 2^{16} darab van! Vegyük mit sem sejtve a 8-bites műveletek számát. Ez $2^{2^8} = 2^{256} \cong 3.2 \cdot (10^3)^{25} \cong 10^{75.5}$, nagyobb, mint az

Avogadro-szám a harmadik hatványra emelve, de még a naptömeg grammban kifejezett értékének a négyzeténél is. Ennek azonban jó része nem *elfajult* művelet. Egy alsó becslés adható a $\Phi(8)$ -ra: $\Phi(8) \geq 2^{2^8} - \binom{8}{7} \cdot 2^{2^7} = 2^{256} - 2^{131} = (1 - 10^{-37.5}) \cdot 2^{256}$. A számítások azt mutatják, hogy a bitszám növekedésével növekszenek a teljes bemenetet kihasználó műveletek száma. Legyen π az n -elemű halmazok egy permutációja,⁸⁵ tehát a mi esetünkben felírható egy olyan $2^n \times 2^n$ -es mátrixként, amelynek minden oszlopában és sorában pontosan egy darab szerepelhet:

$\pi \in \Pi: \mathcal{B}^n \rightarrow \mathcal{B}^n, (a_1, a_2, \dots, a_l, \dots, a_n) \mapsto \pi(a_1, a_2, \dots, a_l, \dots, a_n); a_l \in \mathcal{B}$. Szimmetrikus műveletnek azt az $F: \mathcal{B}^n \rightarrow \mathcal{B}$ -t nevezzük, amelyre $\forall \pi \in \Pi: \mathcal{B}^n \rightarrow \mathcal{B}^n$ -re $F \circ \pi = F$, azaz bármely bemenetét is cseréljük ki F -nek, az nem változtatja értékét. Nem nehéz belátni, hogy az ilyen $F: \mathcal{B}^n \rightarrow \mathcal{B}$ függvényekből összesen 2^n létezik; a definícióból adódóan az ilyen függvények kimenete kizárólag a bemeneten lévő '0' és '1' bitek számától függ. Ennek folytán n bemenet esetén lehetséges 0 darab, 1 darab ... n darab 1-es a bemeneten. Látható tehát, hogy bár az n -bites műveletek nagy arányban teljesen kihasználtak, de csak elenyésző részük szimmetrikus. A 8-bites példánál maradva, az arány $\frac{2^{2^8}}{2^8} = \frac{2^{256}}{2^8} = 2^{248}$! Így látható, hogy az n -bites műveletvégzők kifejezetten az aszimmetrikus műveletekre használhatóak.

⁸⁵ (KATONA, RECSKI, & SZABÓ, 2006)

A Boole algebra Boole modul reprezentációja könnyebben kezelhetővé tesz olyan ismert Boole algebrai azonosságokat, mint a De Morgan azonosság. Ezen tulajdonsága lehetővé teszi többek között a Klein csoport alaposabb vizsgálatára, amelyből fontos következtetéseket vonhatunk le a Boole függvények „összekötéséből” (kompozíciójából) képzett hálózatok gráfelméleti leírására. Erre vonatkozóan jelen fejezetben megállapításra kerül egy szabály amit a hardverleírásban lehet alkalmazni. A hagyományos, tranzisztor alapú digitális áramkörökben a negált kimenetű Boole függvények megvalósítása a legegyszerűbb. Ennek az oka, hogy a tranzisztorokból megvalósított kapcsolás akkor nyit (azaz, mint vezérelt kapcsoló, zár), ha a bemenetre beadott $a, b \in \mathcal{B}^n$ -re a kapcsolás $f: \mathcal{B}^n \rightarrow \mathcal{B}$ Boole függvényét hajtva $f(a) = 1$. Ezen feltétel fennállása azonban, mivel az f kapcsolás a földponton köti össze a kimenetet (CMOS logikában: Pull down network, PDN), a kimeneten pontosan az ellenkező értékű 0 jelenik meg. Vizsgáljuk meg egy CMOS elemi cella példáját. Ha $f(b) = 1$, akkor a tápfeszültséggel kapcsolatot teremtő kapcsolás (CMOS logikában: Pull-up network, PUN) aktivizálódik. Ennek a mi lesz a $g: \mathcal{B}^n \rightarrow \mathcal{B}$ függvénye? Vegyük elő a korábban ismertetett Klein csoportot. A $D = \{I, ', _ , _ * \}$ Boole függvényekre ható operátorok csoportot alkottak a kompozícióra, mégpedig Abel- (kommutatív-) csoportot. Továbbá igaz a $_ * _ ' _ ' _ * = I$ összefüggés. Hogyan kaphatjuk meg a g -t az f -en végrehajtott operátorokkal? A PUN nMOS helyett pMOS tranzisztorokból áll, ez megfelel a $f \rightarrow f'$ transzformációnak. A földpont helyett a tápfeszültséggel köti össze a bemenetet, ez megfelel a $f \rightarrow 'f$ transzformációnak. Végül, a PUN kapcsolás az ún. duális kapcsolása⁸⁶ a PDN kapcsolásnak. Ez a $f \rightarrow f^*$ transzformáció. Ez pontosan a $_ * _ ' _ ' _ * = I$ transzformáció, tehát pontosan ugyanazt a függvényt valósítja meg g , mint f . Ha nem vesszük figyelembe, hogy mivel kötik össze a kimenetet, akkor a $f \rightarrow 'f$ transzformációt kihagyva $h = f^{**}$ lesz a PDN új függvénye. A De Morgan azonosság alapján $h = f^{**} = 'f$, tehát a PDN pontosan akkor lesz nyitva, amikor a PUN zárva, és fordítva. f a PDN kapcsolás függvénye, de ehelyett $h = 'f$ lett megvalósítva, azaz az eredeti Boole függvény komplemente.

Tétel: Az f Boole függvény variánsaira (komplement/kontraduális/duális) $f' = \neg \circ f$,
 $f' = f \circ \neg$, $f^* = \neg \circ f \circ \neg$.

Definíció: A *vezetékköteg* egy szó fizikai realizációja.

Definíció: Egy vezetékköteg *ponált logikájú*, ha minden $a \in \mathcal{B}^n$ szó a -ként van jelen rajta.

⁸⁶ (KATONA, RECSKI, & SZABÓ, 2006)

Definíció: Egy vezetékköteg *negált logikájú*, ha minden $b \in \mathcal{B}^n$ szó $\neg b$ -ként van jelen rajta.

Most vizsgáljuk meg, mi történik, ha két Boole függvényt egymás után végrehajtunk egy vezetékkötegen! Ezeket jelöljük f -fel és g -vel, de ne keverjük össze őket az előző PUN és PDN függvényekkel. **Ponált logika:** Olyan megvalósítás kell, ami egy ponált vezetékköteget kap bementként, és a kimenetén ponált vezetékköteget állít elő: $a \rightarrow (g \circ f)(a)$. Ha két Boole függvényt akarunk egymás szekvenciájaként megvalósítani, akkor ezt matematikailag a következő két módon is megtehetjük: $g \circ f = 'g^* \circ 'f$. **Bizonyítás:** $'g^* \circ 'f = g' \circ 'f = (g \circ \neg) \circ (\neg \circ f) = g \circ (\neg \circ \neg) \circ f = g \circ f$. Alternatív jelöléssel: $G \circ F(a) = \overline{G^*} \circ \overline{F}(a)$. Itt, a ponált logika használatával, a $\overline{G^*}$ és $\overline{F}$ függvények megvalósítható digitális áramköri függvények: $C_1 = \overline{G^*}, C_2 = \overline{F}$. A bemenet és a kimenet ponált. **Negált logika:** Olyan megvalósítás kell, ami egy negált vezetékköteget kap bementként, és a kimenetén negált vezetékköteget állít elő: $\neg a \rightarrow \neg(g \circ f)(a)$. A negált logika miatt, $('g \circ f)(a) = (\neg \circ g \circ \neg \circ f^* \circ \neg)a = ('g \circ 'f^*)(\neg a)$ Alternatív jelöléssel: $\overline{G \circ F(a)} = \overline{G} \circ \overline{F^*}(\bar{a})$. A negált logikát használva $\overline{G}$ és $\overline{F^*}$ ugyanúgy megvalósítható digitális logikai függvények: $C_1 = \overline{G}, C_2 = \overline{F^*}$. A bemenet negált és a kimenet is negált. Ezek alapján kijelenthető, hogy két darab digitális áramkör kompozíciójával tartható a vezetékkötegek ponáltsága, ezért az optimális megoldás mindig *páros darabszámú* logikai áramkör szekvenciáját jelenti. Általánosabban, a problémát modellezni lehet egy irányított gráffal. Ennek legyen a neve *reprezentációs gráf*. A következő táblázat tartalmazza a megfeleltetéseket.

$G = \{E, V\}$	Gráf	$\leftrightarrow$	Logikai áramkörök
$P \in V$	Csúcs	$\leftrightarrow$	$f: \mathcal{B}^n \rightarrow \mathcal{B}^m$ Boole függvény
$e \in E$	Él, élek adott halmaza	$\leftrightarrow$	$a \in \mathcal{B}^k$ Vezetékköteg
Él irányítása	P csúcs felé	$\leftrightarrow$	$a \in \mathcal{B}^n$ $P \leftrightarrow f$ Függvény bemenete
	P csúctól ki	$\leftrightarrow$	$f(a) \in \mathcal{B}^m$ $P \leftrightarrow f$ Függvény kimenete
Él színezése	kék/stb.	$\leftrightarrow$	Ponált vezetékköteg
	zöld/stb.	$\leftrightarrow$	Negált vezetékköteg

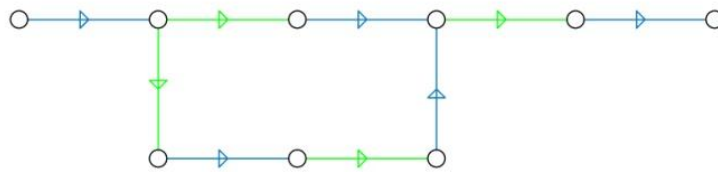
3-7. táblázat: A Boole gráf megfeleltetései a logikai áramkörökkel

Definíció:⁸⁷ Gráf az a $G = \{E, V\}$ rendezett kettes, ahol E az élek halmaza, V a csúcsok halmaza, ahol létezik egy $E \rightarrow V \times V$ függvény (minden élnek két csúcsa lehetséges).

⁸⁷ (KATONA, RECSKI, & SZABÓ, 2006)

Definíció: A *Boole gráf* az a gráf, amelyben léteznek $E \rightarrow \mathcal{B}^k$ és $V \rightarrow \{f: \mathcal{B}^n \rightarrow \mathcal{B}^m\}$, $k, m, n \in \mathbb{N}$ függvények.

A Boole gráf színezésének az a szabálya, hogy egy csúcsba belépő és kilépő élek különböző színűek kell, hogy legyenek. Ez fejezi ki a digitálisan megvalósított Boole függvények ponáltság fordulását. **Dualitási szabály:** *A reprezentációs gráfban minden irányított út egymás után következő élei szín szerint váltakoznak.* Ennek a szemléltetésére nézzünk egy példát:



3-8. ábra: Logikai áramkör gráfszínezési példa

A példánkban jelölje a kék színű él a ponált, a zöld a negált logikát! Láthatjuk, hogy a be- és kimeneten mindenhol azonos a ponáltság, és a *dualitási szabályt* is betartottuk. A logikai áramkörök összekötésének matematikai modellezésére alapozva alkossunk meg egy olyan hardware leíró szintet, amely adatút késleltetés szempontjából optimális megoldások pontos megadására képes. A leíráshoz létrehozásához szükséges algoritmus a következő:

1. A rendszerünk kritikus egységeit bontsuk elemi logikai áramkörökre az optimális sebesség figyelembevételével!
2. Az így kapott hálózatból a korábban leírt módon alkossunk irányított gráfot, és próbáljuk a *dualitási szabály* szerint kiszínezni! Ha sikerült, készen van a leírás.
3. Ha ez lehetetlen, gondoljuk át, hogyan lehetne másképp részegységekre bontani a rendszerünket, kísérletezzünk új felbontással!

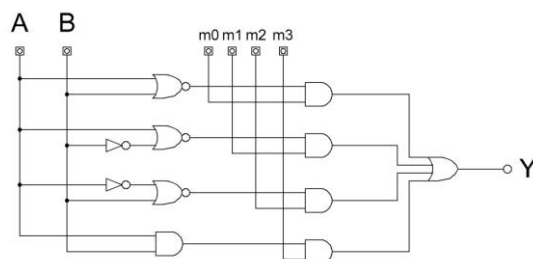
A leírás hátránya, hogy kis változtatás fel tudja az egészet borítani, és nagyon időigényes – akár kapusinten is tervezni kell. Egyszerűbb alegységek tervezése során azonban, például olyan fontosságú esetben, mint az ALU, megéri alkalmazni.

4 Az IBZ kapu

A 3. fejezetben részletezett elméleti modell alkalmazásaként ebben a fejezetben egy újelvű logikai áramkör családról lesz szó, amely kiválthat sok, a 2. fejezetben ismerttetett áramkör részfunkcióját. Ezek közül a leglényegesebbek, *logikai kapuk*, a *teljes- és fél-összeadók*, a $\odot$ *operátor* megvalósítása a prefix fákban, az *egy bites szorzóegységek*, *shifterek* és az *ALU szeletek*.

4.1 A kétbites IBZ kapu

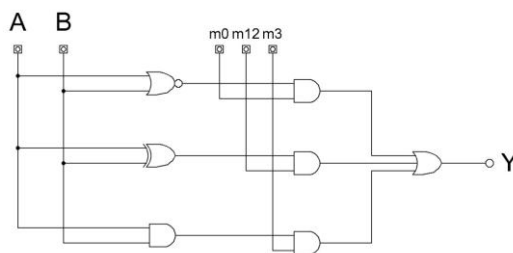
Emlékezzünk vissza a logikai kapumátrixokra. Ezekről bebizonyosodott, hogy minden kétbites Boole műveletet reprezentálni tudnak. Ha sikerül egy ilyen mátrixot a valóságban implementálni, akkor ugyanez funkció áramköri realizációként már alkalmazható. A logikai kapukat modellezni tudtuk egy 2×4 -es mátrixszal, amelynek 4 darab, szabadon megválasztható paramétere volt. Konstruáljunk egy „mátrixszorzó” áramkört! A vezérlő bitek megadják, mi szerepeljen a mátrixban, és a bemeneti biteken végrehajtjuk a mátrixszorzást. Egy kétbites Boole mátrix kimenetére alkalmazva β_1 -et a $v_0 \cdot w_0 \cdot m_{00} + v_0 \cdot w_1 \cdot m_{01} + v_1 \cdot w_0 \cdot m_{10} + v_1 \cdot w_1 \cdot m_{11}$ alakot kapjuk, amit továbbra is egy Boole gyűrű feletti algebrai kifejezés. Valósítsuk meg logikai áramkörrel! Helyettesítsük be a $+$ és $\cdot$ Boole-gyűrűbeli műveletek helyére a Boole algebrai műveleteket, azok helyére pedig a logikai kapukat. A $v_a \cdot w_b$ értékeket elő lehet állítani negálással és identitással a bementekből. Ebből a $v_a \cdot w_b$ kifejezésekben $p \cdot q = p \wedge q$ megfeleltetésnek megfelelően AND kapukat alkalmazunk. A $m_{ab} \cdot (v_a \cdot w_b)$ -t ugyancsak AND kapu valósítja meg. A két bemenet $(v_a \cdot w_b)$ és m_{ab} . Végül, az előállt $m_{ab} \cdot v_a \cdot w_b$ négytagú összegét kell venni. Erre a $p + q = (p \wedge \neg q) \vee (\neg p \wedge q) = p \oplus q$ szolgált. Vegyük azonban figyelembe, hogy a $m_{ab} \cdot v_a \cdot w_b$ összeadandók közül maximum egy tag lesz 1, a többi 0. Ha $p, q \in B$ közül maximum csak az egyik 1, akkor $p \oplus q = p \vee q$. Így vagy a $0 + 0 + 0 + 1$, vagy a $0 + 0 + 0 + 0$ kifejezések egyenlőek lesznek $0 \vee 0 \vee 0 \vee 1$ és $0 \vee 0 \vee 0 \vee 0$ kifejezésekkel. Ez egy négybemenetű OR kapu funkciója. Az első két lépés egy dekóder funkció megvalósítása, amit most „vektorképzőnek” nevezek. Néhány NOT és AND kapu összevonásra került NOT és NOR kapukká, a dualitási feltételeknek megfelelően. A harmadik és negyedik lépés maga a műveletvégzés, ezek a *mátrixszorzó* nevet kapják. Ebből a két alapegységből tevődik össze a vezérelhető kapu.



4-1. ábra: IBZ4⁸⁸ vezérelhető kapu

⁸⁸ Az IBZ család tagjait bemenet szerint IBZ3-nak és IBZ4-nek nevezem.

Ezt az IBZ kapu naiv, kapusintű megvalósítása.⁸⁹ Ha nem akarjuk az összes Boole műveletet megvalósítani, kínálkozik a lehetőség, hogy csökkentsük az állapotok számát, valamelyiket összevonjunk egy másik állapottal. Ez annyit jelent, hogy két állapotbitet formálisan összekötünk egy OR kapuval, és a kimenet lesz az összevont állapot. Ezeket a lehetőségeket járjuk most körbe. Figyeljük meg, hogy a középső két állapot összevonása egy XOR kaput eredményez. Ellenőrizve, valóban, a két állapot közül akkor „1” valamelyik, ha a két bement különböző. Ezzel a lépéssel csak szimmetrikus műveleteket tudunk majd elvégezni.



4-2. ábra: IBZ3 vezérelhető kapu

Az IBZ3 kapu a AND, NAND, OR, NOR, XOR, XNOR, ONE és ZERO kapukat képes megvalósítani. Az alábbi gondolatmenettel azt szerettem volna megmutatni, hogyan lehet eljutni az IBZ kapuig a Boole modul reprezentációból.

⁸⁹ A következőkben adok egy sokkal praktikusabb megvalósítást

4.2 Az n-bites IBZ kapu

A multiplexerek használata megnyitja az utat a kettőnél több operandusú IBZ kapuk felé. A 3. fejezetben kimerítően tárgyalásra kerültek a több-bites Boole műveletek, mint a kétbites Boole műveletek általánosításai. Ennek alkalmazásaként vizsgáljuk meg az áramköri megvalósítást! Használjunk n bitszámú multiplexert!

Itt egy multiplexer *bitszáma* a kiválasztó bemenetek számát jelenti. Ez megegyezik az operandusok számával.

Állítás: Egy n bitszámú multiplexer képes megvalósítani minden $F: \mathcal{B}^n \rightarrow \mathcal{B}$ Boole műveletet.

Bizonyítás: Egy n bitszámú multiplexer adatbemeneteinek száma 2^n . Ezek a bemenetek így összesen 2^{2^n} különböző műveletet valósít meg. Másrészt, egy multiplexer garantáltan más függvényt valósít meg, ha az adat bemeneteire más bemenetet kapcsolunk, így 2^{2^n} különböző Boole műveletünk van. A 3.2.3 fejezetben beláttuk, hogy pontosan ennyi n -bites Boole művelet létezik, így két halmaz egyenlő.

Definíció: *IBZ család* azon $\{f_{IBZ}, \Gamma\}$ rendezett kettes, ahol $f_{IBZ}: \mathcal{B}^k \rightarrow (\mathcal{B}^l \rightarrow \mathcal{B}^m)$ az ún. *IBZ függvény*, $\Gamma = \mathcal{B}^k$ az ún. *vezérlő szavak* halmaza úgy, hogy $\forall g: \mathcal{B}^l \rightarrow \mathcal{B}^m$ Boole függvényhez $\exists \gamma \in \Gamma$, hogy $f_{IBZ}(\gamma) = g$.

Lemma: Ha $\gamma \in \Gamma = \mathcal{B}^n$, $g: \mathcal{B}^n \rightarrow \mathcal{B}$, és $\mathcal{M} \subseteq \left\{ M: \mathcal{B}^n \rightarrow \mathcal{B}^n, M = M_1 \otimes M_2 \otimes \dots \otimes \right.$

$M_n \doteq \otimes_{j=1..n} M_j, M_j = \left\{ \begin{matrix} I \\ \neg \end{matrix} \right\} \right\}$, akkor $g \Leftrightarrow \mathcal{M} \Leftrightarrow \gamma$.

Definíció: Az n -bites IBZ kapu az *IBZ család* azon eleme, amelyre $f_{IBZ}: \mathcal{B}^k \rightarrow (\mathcal{B}^l \rightarrow \mathcal{B}^m)$ *IBZ függvény*, $\Gamma = \mathcal{B}^k$ *vezérlő szavak* esetén $k = 2^n, l = n, m = 1$, azaz tetszőleges $g: \mathcal{B}^n \rightarrow \mathcal{B}$.

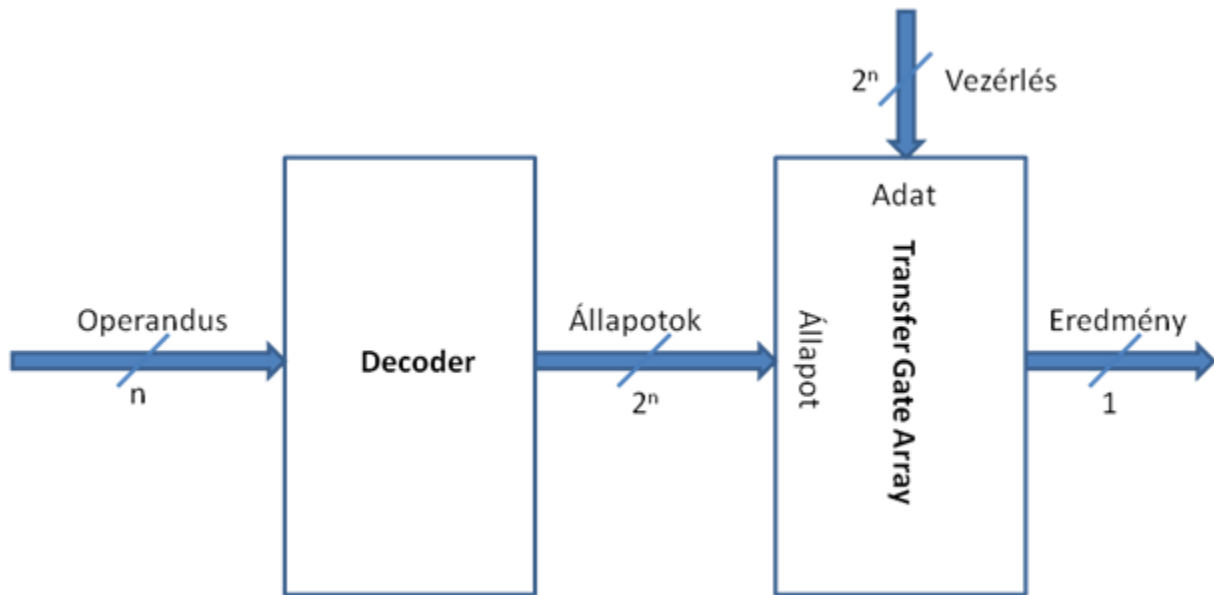
Lemma: Egy n -bites IBZ kapu Boole függvénye azonos egy n -bites multiplexerével.

A Boole függvények előállítása lehetséges az ún. Look-up Table-lel (LUT).^{90, 91} Fontos különbség azonban, hogy az IBZ kapunál nem egy lassú memóriarekesszel vagy egyéb tárolóval oldjuk meg a bejegyzések jelenlétét, hanem azok vezérlőjelként valósítják meg a Boole függvényt. A Boole függvények ezen megvalósítása valójában a Boole modul mátrixszorzás műveletén alapul.

A multiplexer vázlatos felépítése:

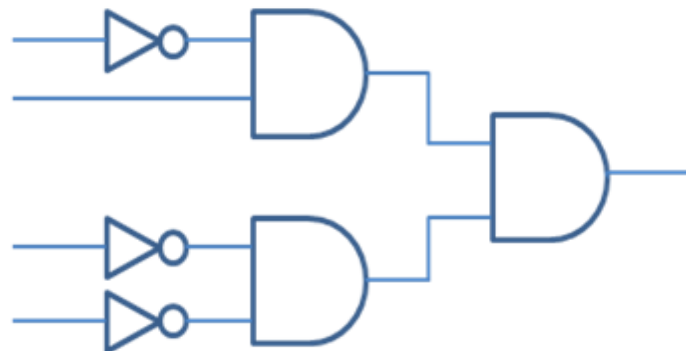
⁹⁰ (ROTH & KINNEY, 2010)

⁹¹ (HOLDSWORTH & WOODS, 2002)



4-3. ábra: Az IBZ kapu belső szerkezete

Az n növelésével azonban óvatosan kell bánnunk. A multiplexerben az adat bemenetek száma 2^n , ami azt jelenti, hogy ilyen széles szót kell elvezetni a később megvalósítandó ALU minden alapkapujához. A másik probléma a multiplexerben található dekóder bonyolultsága. Egy négybites állapot előállítása átlagosan két invertert és három logikai kaput igényel.

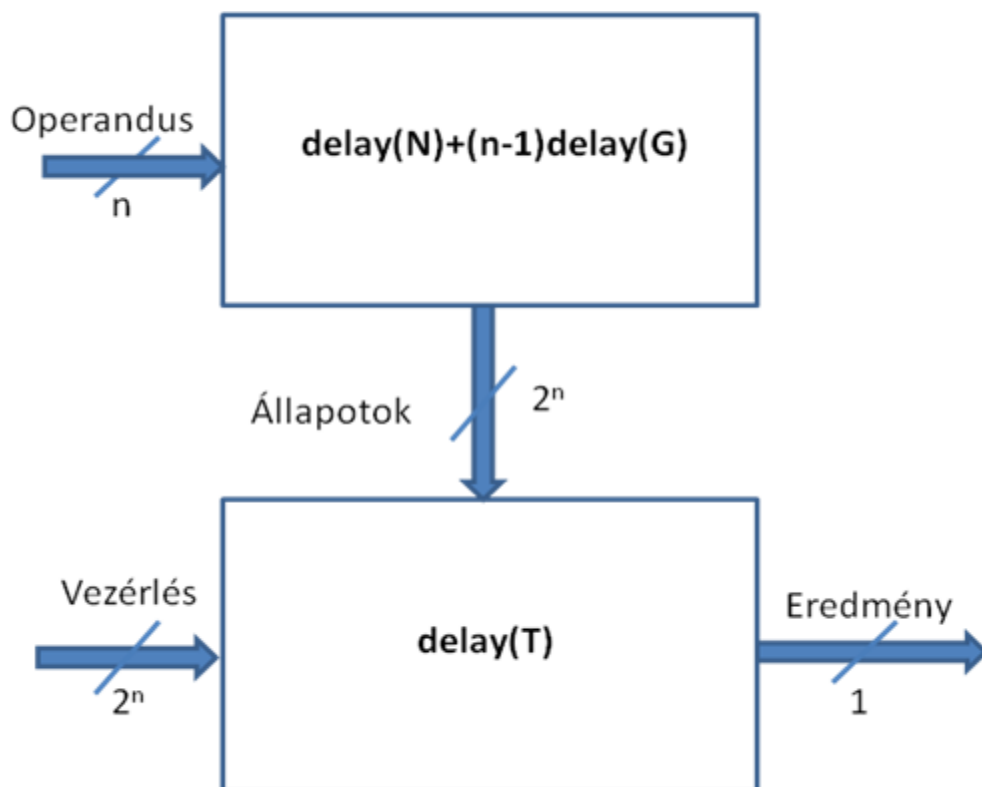


4-4. ábra: Az $\bar{a}b\bar{c}d$ állapot

Ha a bemenetek száma $n = 2^k$, akkor egy állapot előállításához bináris fa⁹² használható. Ha csak AND kapukat használunk, akkor az inverterek elhelyezésével tudjuk az állapotot megszabni. Egy állapot felírható $\bigwedge_{j=1..n} M_j a_j = \bigwedge_n (M_1 a_1 \otimes M_2 a_2 \otimes \dots \otimes M_n a_n) = \bigwedge_n (\otimes_{j=1..n} M_j a_j) = \bigwedge_n M a$ alakban, ahol $\bigwedge_n$ fejezi ki az n -bites konjunkciót, azaz az AND kapuk bináris fáját, és M pedig az inverterek elhelyezését.

⁹² (KATONA, RECSKI, & SZABÓ, 2006)

Azt már korábban beláttuk, hogy $\Lambda_n = I \otimes_{\wedge} I \otimes_{\wedge} \dots \otimes_{\wedge} I$, és $M = M_1 \otimes M_2 \otimes \dots \otimes M_n \doteq \otimes_{j=1..n} M_j$, $M_j = \left\{ \begin{smallmatrix} I \\ \neg \end{smallmatrix} \right\}$, így $\Lambda_n M = M_1 \otimes_{\wedge} M_2 \otimes_{\wedge} \dots \otimes_{\wedge} M_n$, ami tetszőlegesen zárójelezhető. Így először kettesével, majd négyesével, stb. zárójelezve egy bináris AND kapu fa algebrai leírásához jutunk. Ebben – mindenféle áramkör minimalizálás nélkül – $\sum_{i=0..k-1} 2^i = 2^k - 1 = n - 1$ darab kapura lesz szükség, ami lineáris növekedést mutat. Az állapotok száma azonban 2^n , ami exponenciális, tehát $2^n \cdot (n - 1)$ a növekedés. Arról nem is beszélve, hogy az ehhez szükséges vezérlőbemenetek száma is 2^n . Összefoglalva, a multiplexer kiválasztó bemenetei, azaz az operandusok $n - 1$ kapukésleltetést és max. 1 inverter késleltetést szenvednek, amíg a transzfer kapukig elérnek. A multiplexer adat bemenetei, azaz a vezérlőbitek csupán a 1 transzfer kapunyi késleltetés után megjelennek a kimeneten. A késleltetés tehát, ha $delay(N)$ az inverter (azaz NOT kapu) késleltetése, $delay(G)$ egy két bemenetű kapu (Gate) késleltetése és $delay(T)$ a transzfer kapu késleltetése, akkor a teljes késleltetés $delay(N) + (n - 1) \cdot delay(G) + delay(T)$. Ezen késleltetések között fennáll a $delay(G) > delay(N) > delay(T)$ összefüggés, de pontos értékük technológiafüggő.



4-5. ábra: Az n-bites multiplexer késleltetési viszonyai

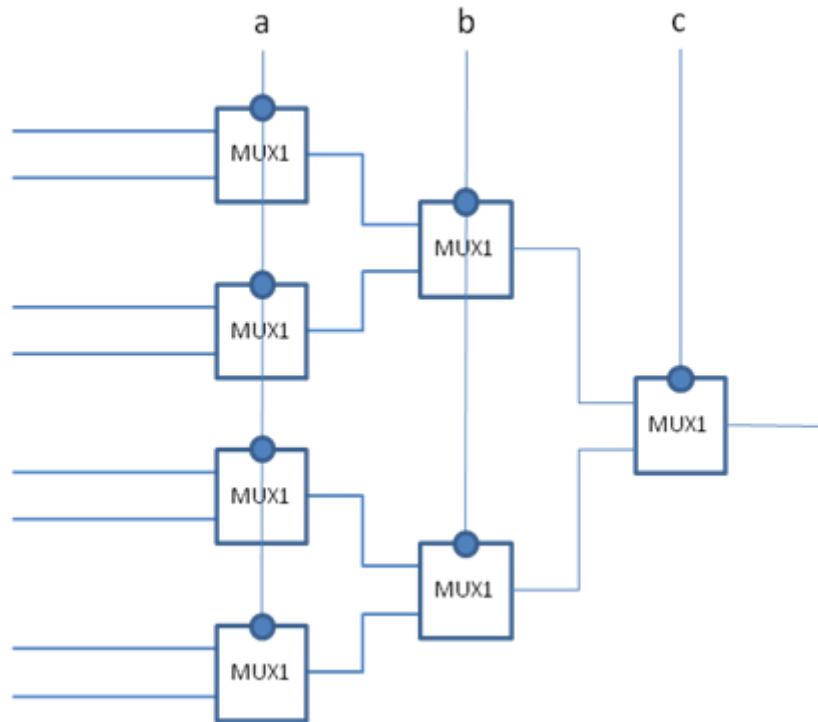
Az ábra egy n -bites multiplexer késleltetési viszonyait mutatja be. A dekóderben (felső egység) 2^n darab, $(n - 1)$ kaput tartalmazó fa struktúra biztosítja a működést. A fa struktúra transzfer kapukra történő átszervezésével azonban jelentős sebességnövekedést érhetünk el. Használjunk egy darab n -bites multiplexer helyett sok 1-bites multiplexert! Egy 1-bites multiplexer Boole függvénye a kétbites k -redukált tenzorszorzat lesz (3.2.1) - $A_k: \mathcal{B}^n \rightarrow \mathcal{B}, (a_1, a_2) \mapsto a_k; a_k \in (a_1, a_2), I \otimes_k I = A_k \circ (I \otimes I)$, ahol a_k az egyetlen kiválasztó bemenet függvényében a_1 vagy a_2 lesz. A kiválasztó bemenetek *igazságtartalmát*, jelöljük $o_1, o_2, \dots, o_n$, és $k = o_l$ lesz a multiplexer függvénye, ha az l -edik kiválasztó bemenet van rákötve. Egy n -bites multiplexer függvénye $A_k: (a_1, a_2, \dots, a_l, \dots, a_{2^n}) \mapsto a_k; 1 \leq k \leq 2^n$, ahol $k = \sum_{l=1..n} 2^{l-1} o_l$. Ez azt fejezi ki, hogy az összes kiválasztó bemenet egy fixpontos bináris számként vehető, amely megadja a kiválasztandó a_k sorszámát.

Állítás: $A_k = \left(\left((I_1 \otimes_{o_1} I_2) \otimes_{o_2} (I_3 \otimes_{o_1} I_4) \right) \otimes_{o_3} \left((I_5 \otimes_{o_1} I_6) \otimes_{o_2} (I_7 \otimes_{o_1} I_8) \right) \right) \otimes_{o_4} \dots$

Bizonyítás: A bizonyítást nem algebrai, hanem pusztán egyszerű logikai következtetésekkel érjük el. Az n -bites multiplexernél a $k = \sum_{l=1..n} 2^{l-1} o_l$ számot nemcsak egyben, hanem számjegyenként is elvégezhetjük. Tegyük fel, hogy k adott, és az a_k kimenetet fogja kiválasztani a multiplexer. Ekkor az állításban látott kifejezésben a k -adik identitás művelethez (I_k) mindig adható pontosan egy olyan $(o_1, o_2, \dots, o_n)$ szám n -es, hogy a művelet a k -adik bemenetet válassza. Belátható, hogy ez a szám n -es a $k = \sum_{l=1..n} 2^{l-1} o_l$ szám 2 hatványaival történő osztásának a maradéka lesz (0 vagy 1), így az $\overline{o_n \dots o_2 o_1}$ számjegyekből álló számra igaz lesz, hogy $\overline{o_n \dots o_2 o_1} = k = \sum_{l=1..n} 2^{l-1} o_l$. $\square$ Az állítás bizonyítható a Boole függvények ún. *bináris döntési fa*-ként történő reprezentációjából.⁹³ Így az IBZ kapu ötletéhez el lehet jutni ezen úton is.⁹⁴

⁹³ (SHANNON C. , 1949)

⁹⁴ (SCHAFFER & PERKOWSKI, 1993)



4-6. ábra: A_k megvalósítása. $n = 3$, ($a \mapsto o_1, b \mapsto o_2, c \mapsto o_3$ a bemenetek)^{95 96}

A globális állapotdekóder, az egybites multiplexerek lokális állapotdekódereinek tömbjéből áll. Egy egybites multiplexer állapotdekódere egy inverterből és egy vezetékből áll:

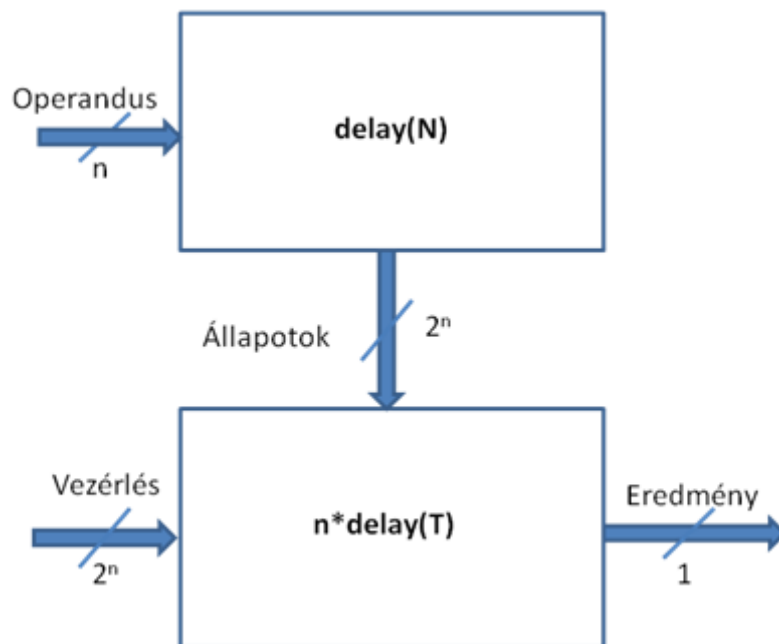


4-7. ábra: Egybites multiplexer dekódere

Mivel a kapcsolásban nem használtunk kétbites kaput, ezért a teljes késleltetés az állapotdekódolás oldaláról $\text{delay}(N)$ -nel egyenlő. A fa struktúra sajátossága miatt a jelnek $\log_2 2^n = n$ transzfer kapun kell áthaladnia. A teljes késleltetés $\text{delay}(N) + n \cdot \text{delay}(T)$.

⁹⁵ (HOLDSWORTH & WOODS, 2002)

⁹⁶ (HERNANDEZ-AGUIRRE, BUCKLES, & COELLO-COELLO, 2000)



4-8. ábra: Az egybites multiplexer fa késleltetési viszonyai

Az új multiplexer struktúrával elérhető, hogy az IBZ kapu késleltetése rendkívül kicsi legyen. Legvégül vizsgáljuk meg a technikai megvalósítás részleteit, alkalmazva a 3.3 fejezet eredményeit! Az elemi cellákkal az ún. invertáló multiplexer valósítható meg legkönnyebben. Ha ezek közül páros számút rakunk egymás után, akkor – a dualitási tétel értelmében – ugyanúgy ponált kimenetet kapunk. Így az n paritásától függ, hogy a Dualitási szabály teljesül-e. Egyértelmű, hogy a gyakorlatban törekedni kell a páros bemenetű kapukra.

$$n_{opt} = 2 \cdot m, m \in \mathbb{N} \setminus \{0\}.$$

5 Az IBZ ALU

Szó esett arról, hogy az IBZ kapu kiválthat sok, klasszikus logikai funkciót. Erre alapozva létrehoztam egy saját ALU architektúrát (IBZ ALU), amelyre az alábbi alapelvek vonatkoznak:

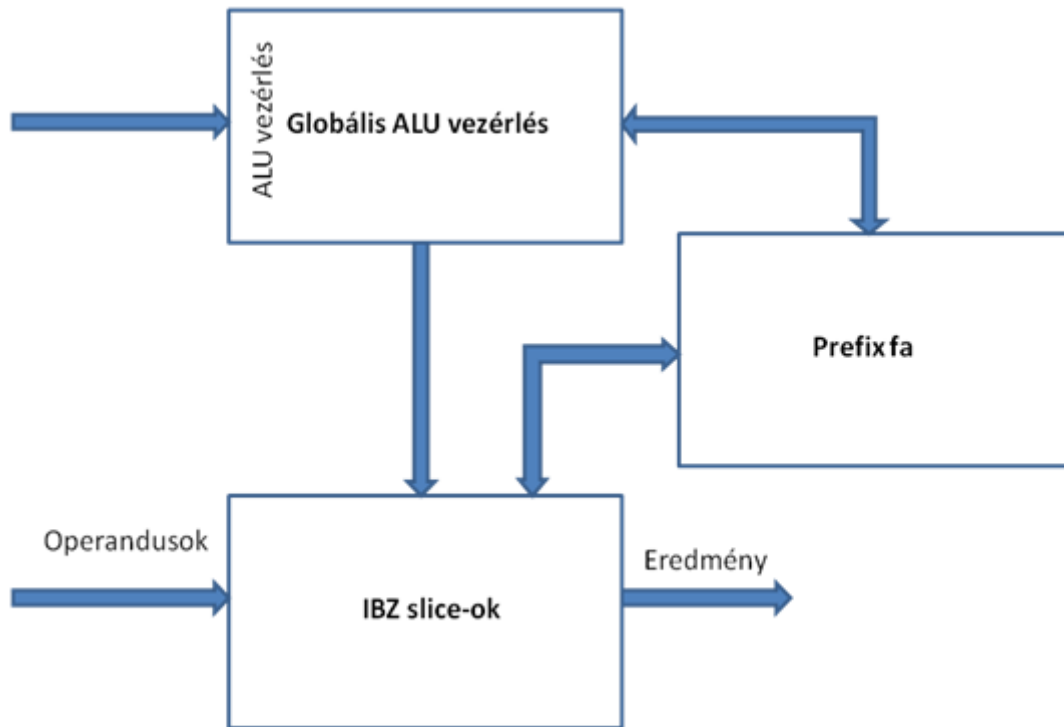
- Az adatút a lehető legkevesebb kapun halad keresztül, és a vezetékek hossza nem túl nagy. Így a **ciklusidő** csökkenthető.
- A másik általam javítani kívánt dolog az utasítások számának csökkentése, azok összevonása, a **ciklusszám**ukat továbbra sem növelve (nem RISC→CISC átalakítás!).

Ezt a feladatot az IBZ család alkalmazásával valósítottam meg. A logikai és aritmetikai műveletek együtt történő kezelése például az FPGA-k logikai blokkjaiban (Programmable Logic Block, PLB)⁹⁷ történik. Én ezt az ötletet átültettem az ALU-k architektúrájára, új elvi alapokat megfogalmazva.

⁹⁷ (LEIJTEN-NOWAK, 2004)

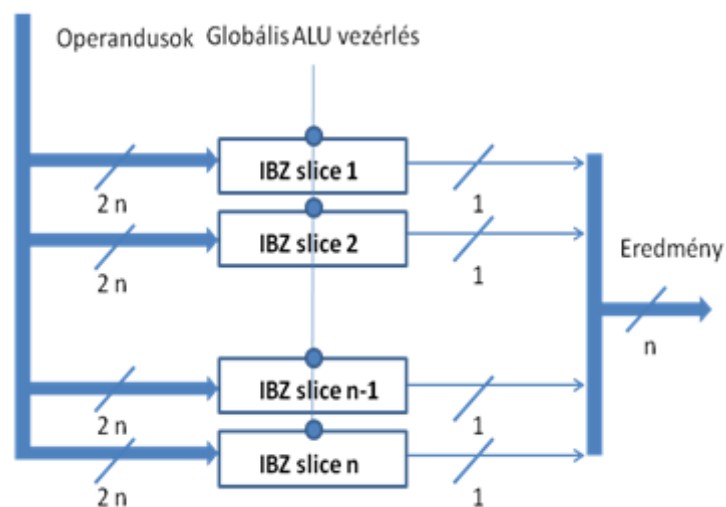
5.1 Az IBZ szelet

Az IBZ ALU architektúrájában az ALU szeletek szerepét az ún. IBZ szelet veszi át. Az IBZ szelet mellett az IBZ ALU tartalmaz egyéb részegységeket (például prefix fa az összeadásoz, és globális vezérlést).



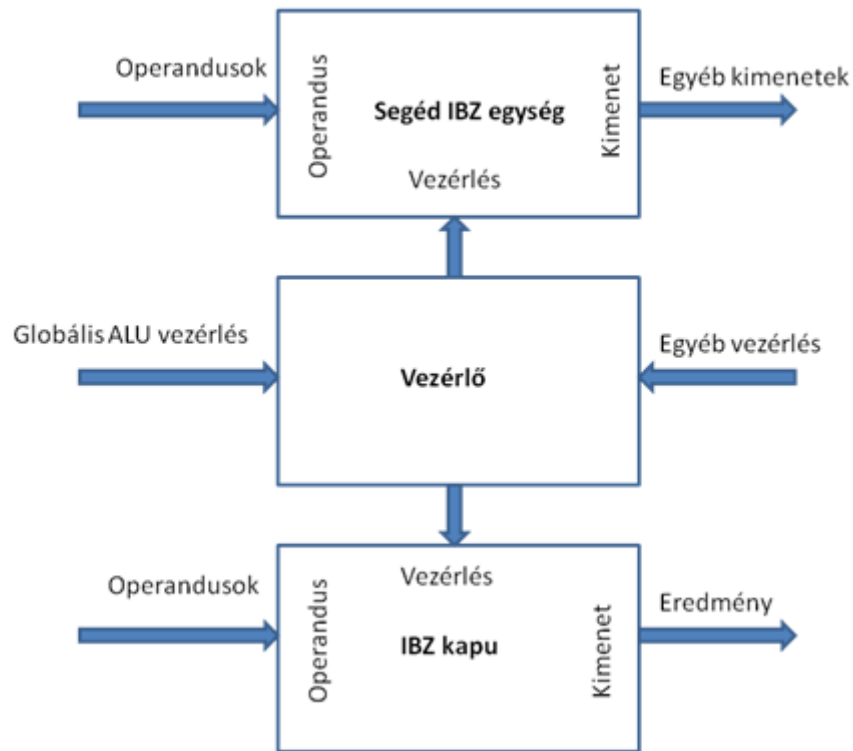
5-1. ábra: Az IBZ ALU blokkvázlata

Az IBZ szeletek egy n tagú tömböt alkotnak, és mindegyik egy-egy bemeneti operandus bitet kap a bemenetén.



5-2. ábra: Az IBZ tömb

Egy IBZ szelet egy vezérlőegységből és két számítási egységből áll. Az egyik számítási egység egy IBZ kapu, a másik a segédműveleteket hajtja végre (például, összeadásnál a propagate és generate jelek előállítását).



5-3. ábra: Egy IBZ szelet belső szerkezete

5.2 Az IBZ ALU műveletei

Az alábbiakban az IBZ szeletben megvalósított ALU műveleteket ismertetem.

5.2.1 Logikai műveletek

A kétbites IBZ kapu (továbbiakban: IBZ kapu) a logikai műveleteket a vezérlő bemenetén kapott négybites szó segítségével állítja elő. Az IBZ kapu függvénye a következőképpen alakul: $f_{IBZ}: \mathcal{B}^4 \rightarrow (\mathcal{B}^2 \rightarrow \mathcal{B})$ az IBZ működését leíró IBZ függvény, $\gamma \in \mathcal{B}^4$ a vezérlő szó, $\gamma = \beta_i \otimes \beta_j \otimes \beta_k \otimes \beta_l$, $f_{IBZ}(\gamma): \mathcal{B}^2 \rightarrow \mathcal{B}$, $f_{IBZ}(\beta_i, \beta_j, \beta_k, \beta_l) = \beta_i \otimes \beta_0 \otimes \beta_0 + \beta_j \otimes \beta_0 \otimes \beta_1 + \beta_k \otimes \beta_1 \otimes \beta_0 + \beta_l \otimes (\beta_1 \otimes \beta_1) = [\beta_i, \beta_j, \beta_k, \beta_l]$. A $[\beta_i, \beta_j, \beta_k, \beta_l]$ mindig mátrixot fog jelölni. Ez pontosan 2^4 műveletet eredményez, tehát $f_{IBZ}(\gamma)$ bármely kétbites Boole művelet lehet.

5.2.2 Aritmetikai műveletek

Az ismert prefix fás összeadók megvalósításához a következő megfeleltetések tehetők:

Teljes összeadó	$\leftrightarrow$ IBZ kapu
Propagate és generate	$\leftrightarrow$ Segéd IBZ kapu
Lookahead Carry Unit ($\odot$ függvény)	$\leftrightarrow$ IBZ Prefix fa
$\odot$ művelet	$\leftrightarrow$ Segéd IBZ kapu

5-1. táblázat: Az IBZ kapuk felhasználása standard funkciókhoz

A p, g propagate és generate jelek helyettesítik a C_{out} kimeneti carry-t így csak a $s = c^{in} \oplus (a \oplus b)$ kimenetet kell az IBZ kapunak előállítania. Ez felfogható egy $O(c^{in})(a \oplus b) = O(c^{in}) \circ \oplus (a, b)$ függvénynek is, ahol $O: \mathcal{B} \rightarrow (\mathcal{B} \rightarrow \mathcal{B})$, $O(c)(f) = c \oplus f$, $\oplus_2: \mathcal{B}^2 \rightarrow \mathcal{B}$, $\oplus_2(a, b) = a \oplus b$. Az IBZ kapu a $O(c^{in}) \circ \oplus_2$ függvényt kell, hogy előállítsa. $c^{in} = \beta_0$ esetén $O(c^{in}) \circ \oplus_2 = I \circ \oplus_2 = \oplus_2$, míg $c^{in} = \beta_1$ esetén $O(c^{in}) \circ \oplus_2 = \neg \circ \oplus_2$. $\oplus_2 = [\beta_0, \beta_1, \beta_1, \beta_0]$, ezért $f_{IBZ}(\gamma) = [c^{in}, \neg c^{in}, \neg c^{in}, c^{in}]$. Ez azt jelenti, hogy γ vezérlő szóra teljesül $\gamma = c^{in} \otimes \neg c^{in} \otimes \neg c^{in} \otimes c^{in}$. Tehát a 00 és 11 bementekre c^{in} -t kell bevezetni, míg a 01 és 10 bemenetekre $\neg c^{in}$ -t, azaz c^{in} egy NOT kapun átvezetve. Ez a módszer az ún. Shannon-dekompozíció.⁹⁸ A p, g propagate és generate jelek előállítása $p = a \vee b$, $g = a \wedge b$ miatt az operandusokból egyszerű XOR és AND kapukkal történik. Az IBZ prefix fa a Segéd IBZ logikával helyettesített $\odot$ műveletből áll. $(g_i, p_i) \odot (g_j, p_j) = (g_i \vee (p_i \wedge g_j), p_i \wedge p_j)$ alapján a $p_i \wedge p_j$ ismét egyszerű AND kapukkal képezhető. A $g_i \vee (p_i \wedge g_j)$ függvényt

⁹⁸ (SHANNON C. , 1949)

megint IBZ kapuval helyettesítjük. Itt is definiálható O operátor, hogy $g_i \vee (p_i \wedge g_j) = O(g_i) \circ \wedge_2 (a, b)$. $g_i = \beta_0$ esetén $O(g_i) \circ \wedge_2 = I \circ \wedge_2 = \wedge_2$, míg $g_i = \beta_1$ esetén $O(g_i) \circ \wedge_2 = T$ (a tautológia, minden bemenetre β_1 a kimenete). $\wedge_2 = [\beta_0, \beta_0, \beta_0, \beta_1]$, ezért $f_{IBZ}(\gamma) = [g_i, g_i, g_i, \beta_1]$. $\gamma = g_i \otimes g_i \otimes g_i \otimes \beta_1$. A g_i -t a 00, 01, 10 bemenetekre kell bekötni, és érdekes módon az 11 láb konstans '1' bit lesz. A kivonás műveletét szeretnénk hasonlóan elvégezni, mint az összeadást, hogy ne kelljen egy extra részegységet beépíteni, ami csak lassítaná az adatutatót. Legyen a koncepciónk a következő: a kettes komplementes bemenetek közül a negatív bemeneteknek csak az egyes komplementjét képezzük (bitenként negáljuk). Ez a kívánt negatív számnál 1-gyel kisebb számot eredményez. Ezt a hiányt az ALU C_{in} bemenetével kompenzáljuk majd. Ezek az $(\neg a, b)$, $(\gamma)(a, \neg b)$ és $(\neg a, \neg b)$ eseteknek felelnek meg. Általánosítsuk a problémát, és az általánosítás speciális eseteként meg fogjuk kapni a kivonás vezérlőszavait.

5.2.3 Vegyes logikai-aritmetikai műveletek

Legyenek $q, r: \mathcal{B} \rightarrow \mathcal{B}$ függvények, ekkor a bemenetek $(q(a), r(b))$. Célunk a $q(a) + r(b)$ előállítás. Vizsgáljuk meg, hogyan tudjuk az $s = c^{in} \oplus (q(a) \oplus r(b))$ kimenetet előállítani. Ekkor $O(c^{in})(q(a) \oplus r(b)) = O(c^{in}) \circ \oplus_2 \circ (q \otimes r)(a, b)$. $O(\beta_0) \circ \oplus_2 = I \circ \oplus_2 = \oplus_2$ miatt $O(\beta_0) \circ \oplus_2 \circ (q \otimes r) = \oplus_2 \circ (q \otimes r) = q \otimes_{\oplus} r$, $O(\beta_1) \circ \oplus_2 = \neg \circ \oplus_2$ miatt $O(\beta_1) \circ \oplus_2 \circ (q \otimes r) = \neg \circ \oplus_2 \circ (q \otimes r) = q \otimes_{-\oplus} r$. Ha $q = [\beta_i, \beta_j]$, $r = [\beta_k, \beta_l]$, akkor $q \otimes_{\oplus} r = [\beta_i \oplus \beta_k, \beta_i \oplus \beta_l, \beta_j \oplus \beta_k, \beta_j \oplus \beta_l]$, és $q \otimes_{-\oplus} r$ ennek a komplemente (negáltja). Vezessünk be új jelölést: $[\beta_i \oplus \beta_k, \beta_i \oplus \beta_l, \beta_j \oplus \beta_k, \beta_j \oplus \beta_l] = [\beta_a, \beta_b, \beta_c, \beta_d]$. A teljes IBZ függvény ekkor $f_{IBZ}(\gamma) = O(c^{in}) \circ \oplus_2 \circ (q \otimes r) = c^{in} \oplus [\beta_a, \beta_b, \beta_c, \beta_d] = [(c^{in} \oplus \beta_a), (c^{in} \oplus \beta_b), (c^{in} \oplus \beta_c), (c^{in} \oplus \beta_d)]$. Tehát a vezérlőszó $\gamma = (c^{in} \oplus \beta_a) \otimes (c^{in} \oplus \beta_b) \otimes (c^{in} \oplus \beta_c) \otimes (c^{in} \oplus \beta_d)$. A p, g propagate és generate jelek előállítása $p = q(a) \vee r(b)$, $g = q(a) \wedge r(b)$ alakban történik. Ezek a műveletek felírhatóak $q \otimes_{\vee} r$ és $q \otimes_{\wedge} r$. $\otimes_{\vee} = [\beta_i \vee \beta_k, \beta_i \vee \beta_l, \beta_j \vee \beta_k, \beta_j \vee \beta_l]$, és $\otimes_{\wedge} = [\beta_i \wedge \beta_k, \beta_i \wedge \beta_l, \beta_j \wedge \beta_k, \beta_j \wedge \beta_l]$. A két γ vezérlőszó ezekből képezhető a már ismert módon. Az egyenletek azt mutatják, hogy ebben az esetben két darab IBZ kaput kell elhelyezni a segéd IBZ egységben. A kivonás művelete a $(q, r) = \{(I, \neg), (\neg, I), (\neg, \neg)\}$ esetek. Nagyon fontos ilyenkor figyelni a C_{in} és C_{out} globális carry jelekre, mert ezek is transzformálódnak. Ez a transzformáció technikai jellegű, ezért jelen sorokban nem részletezzük.⁹⁹

⁹⁹Bővebben lásd: (LEIJTEN-NOWAK, 2004)

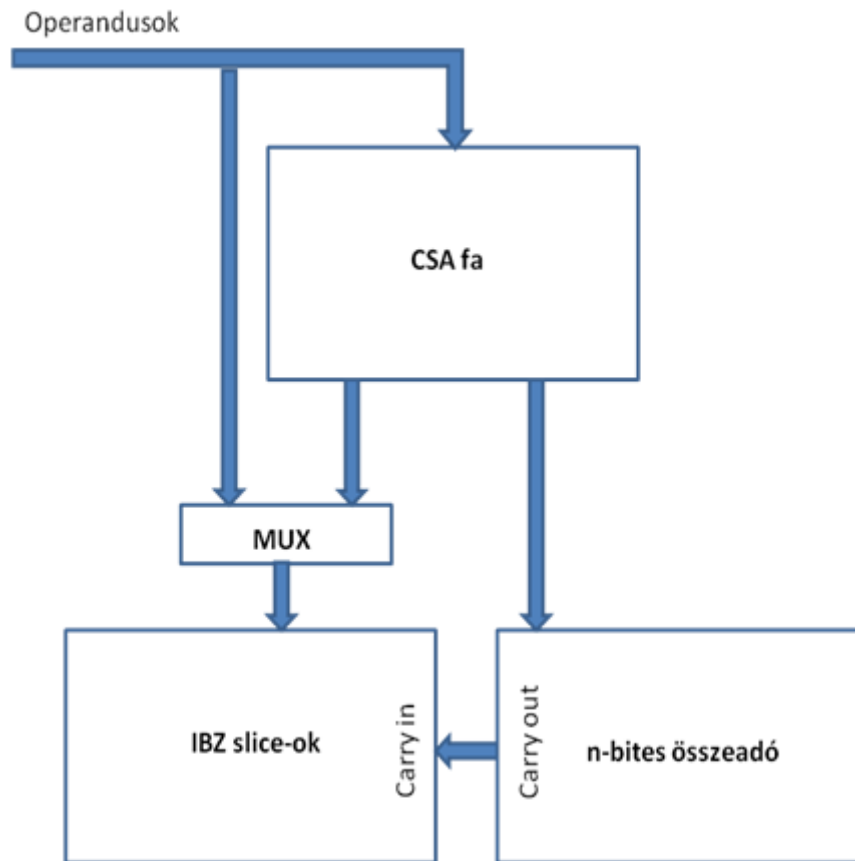
Legyen $t: \mathcal{B}^2 \rightarrow \mathcal{B}$ függvények, ekkor a bemenetek $(a, t(a, b))$. Célunk a $a + t(a, b)$ előállítás. Vizsgáljuk meg, hogyan tudjuk az $s = c^{in} \oplus (a \oplus t(a, b))$ kimenetet előállítani. Ekkor $O(c^{in})(a \oplus t(a, b)) = O(c^{in}) \circ \oplus_2 \circ \tau(a, b)$, ahol $\tau: (a \otimes b) \rightarrow (a \otimes t(a, b))$. $O(\beta_0) \circ \oplus_2 = I \circ \oplus_2 = \oplus_2$ miatt $O(\beta_0) \circ \oplus_2 \circ \tau = \oplus_2 \circ \tau$, $O(\beta_1) \circ \oplus_2 = \neg \circ \oplus_2$ miatt $O(\beta_1) \circ \oplus_2 \circ \tau = \neg \circ \oplus_2 \circ \tau$. Ha $t = [\beta_i, \beta_j, \beta_k, \beta_l]$, akkor $\tau = ([\beta_0, \beta_0, \beta_1, \beta_1], [\beta_i, \beta_j, \beta_k, \beta_l])$, $\oplus_2 \circ \tau = [\beta_0 \oplus \beta_i, \beta_0 \oplus \beta_j, \beta_1 \oplus \beta_k, \beta_1 \oplus \beta_l] = [\beta_i, \beta_j, \neg\beta_k, \neg\beta_l]$. A teljes IBZ függvény ekkor $f_{IBZ}(\gamma) = O(c^{in}) \circ \oplus_2 \circ (q \otimes r) = c^{in} \oplus [\beta_i, \beta_j, \neg\beta_k, \neg\beta_l] = [(c^{in} \oplus \beta_i), (c^{in} \oplus \beta_j), (c^{in} \oplus \neg\beta_k), (c^{in} \oplus \neg\beta_l)]$. Tehát a vezérlőszó $\gamma = (c^{in} \oplus \beta_i) \otimes (c^{in} \oplus \beta_j), (c^{in} \oplus \neg\beta_k) \otimes (c^{in} \oplus \neg\beta_l)$. A p, g propagate és generate jelek előállítása $p = a \vee t(a, b)$, $g = a \wedge t(a, b)$ alakban történik. $\vee_2 \circ \tau = [\beta_0 \vee \beta_i, \beta_0 \vee \beta_j, \beta_1 \vee \beta_k, \beta_1 \vee \beta_l] = [\beta_i, \beta_j, \beta_1, \beta_1]$, és $\wedge_2 \circ \tau = [\beta_0 \wedge \beta_i, \beta_0 \wedge \beta_j, \beta_1 \wedge \beta_k, \beta_1 \wedge \beta_l] = [\beta_0, \beta_0, \beta_k, \beta_l]$. A γ vezérlőszavak $\beta_i \otimes \beta_j \otimes \beta_1 \otimes \beta_1$ és $\beta_0 \otimes \beta_0 \otimes \beta_k \otimes \beta_l$ lesznek. Ebben az esetben is két darab IBZ kaput kell elhelyezni a segéd IBZ egységben. A kivonás művelete itt is előállítható a $t(a, b) = \neg b$ választással, de csak az egyik operandusra.

5.2.4 A shift és rotate műveletek

A k -szoros shift és rotate műveletek képezhetőek, ha minden S_i IBZ slicere $f_{IBZ}(\gamma)(a_i, b_i) = a_{i \pm k}$, azaz a vezérlőszó $\gamma = a_{i \pm k} \otimes a_{i \pm k} \otimes a_{i \pm k} \otimes a_{i \pm k}$. $+k$ esetén balra, $-k$ esetén jobbra shiftelés/rotate művelet lép életbe. $i \pm k$ ciklikusan értendő, azaz, ha n bit széles az ALU, akkor a modulo n maradékosztály csoportjaként.

5.2.5 A szorzás művelete

Az ismertetett szorzók két nagy részből állnak; a Carry-Save összeadók (CSA) fájából és a végső összeadóból. Az utóbbi helyére egyelőre építsünk be egy, az irodalomból ismert gyors összeadót, és koncentráljunk a CSA fára! Ha $r = 2$ radix szorzót készítünk akkor egybites CSA egységek alkotják a CSA fát. Ezek megvalósíthatóak egy teljes összeadóval. Egy teljes összeadó, pedig, megvalósítható IBZ kapukkal. Az $s = c^{in} \oplus (a \oplus b)$ -re a γ vezérlőszó megegyezik a korábban tárgyaltakkal: $\gamma = c^{in} \otimes \neg c^{in} \otimes \neg c^{in} \otimes c^{in}$. A $c^{out} = g \vee (p \wedge c^{in})$ generálása pedig megegyezik a korábban ismertetett $\odot$ művelet $\gamma = g \otimes g \otimes g \otimes \beta_1$ vezérszavával. Így két IBZ kapuval megvalósítható egy teljes összeadó.



5-4. ábra: A szorzás egy lehetséges megvalósítása

A szorzó tömb beépítése az IBZ ALU-ba nehézkes, mert a kimenet dupla széles, $2n$. Két megoldás alkalmazható. Lehetséges teljesen különválasztani, egy új egységbe, valamilyen segéd ALU-ba beépíteni a szorzót. A másik lehetőség, hogy az IBZ ALU-ban az operandusokat rávezetjük egy CSA fára, ahonnan multiplexer segítségével rávezethető az IBZ szelet tömbre, ami a végső összeadást elvégzi. Ha nem akarunk emiatt $2n$ széles IBZ szelet tömböt csinálni, akkor elég csak a felső n bitet bevezetni, míg az alsó n bitet egy különálló összeadóval összeadni, a carry-t bevezetni az IBZ szeletbe. Ez a megoldás a multiplexerek használata miatt a többi művelet rovására megy.

5.2.6 Az osztás művelete

Az osztás műveletére nem dolgoztam ki IBZ családdal megvalósított architektúrát, így egy standard osztó használható. Ez mérete, komplexitása miatt megfontolandó, hogy egy külön segéd ALU-ban helyezzük-e el.

5.2.7 Adatmozgató műveletek

Ha egyik regiszter tartalmát szeretnénk beírni egy másik regiszterbe, akkor az IBZ szelet bemenetén a regiszter busz bemenetet kell átengedni. Legyen az akkumulátorban tárolt

operandus a , és b a regiszter busz bemenete. Ekkor az $f_{IBZ}(\gamma)(a, b) = b$ tárolódik a másik regiszterben. Ezt a $\gamma = \beta_0 \otimes \beta_1 \otimes \beta_0 \otimes \beta_1$ vezérlőszóval valósítható meg. Ellenőrzés: $f_{IBZ}(\gamma) = \beta_0 \otimes \beta_0 \otimes \beta_0 + \beta_1 \otimes \beta_0 \otimes \beta_1 + \beta_0 \otimes \beta_1 \otimes \beta_0 + \beta_1 \otimes \beta_1 \otimes \beta_1$. Vegyük észre, hogy minden tag $\beta_i \otimes \beta_j \otimes \beta_i$, tehát a kimenet csak a „második” bementtől függ:

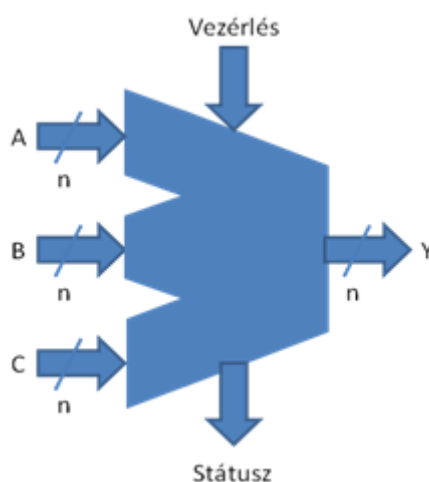
$$\text{Ha } b = \beta_0, \text{ akkor } \beta_i \cdot \beta_j(a) \cdot \beta_i(\beta_0) = \begin{cases} \beta_0, & \text{ha } \beta_i = \beta_0 \\ \beta_0, & \text{ha } \beta_i = \beta_1 \end{cases},$$

$$\text{és ha } b = \beta_1, \text{ akkor } \beta_i \cdot \beta_j(a) \cdot \beta_i(\beta_1) = \begin{cases} \beta_0, & \text{ha } \beta_i = \beta_0, a \text{ tetsz.} \\ \beta_1, & \text{ha } \beta_i = \beta_1, \beta_j(a) = 1. \end{cases}$$

A $b = \beta_0$ esetben $f_{IBZ}(\beta_0, \beta_1, \beta_0, \beta_1)(\beta_0) = \beta_0$, a $b = \beta_1$ esetben $f_{IBZ}(\beta_0, \beta_1, \beta_0, \beta_1)(\beta_1) = \beta_1$.

5.3 Az n -operandusú ALU

A kétoperandusú ALU fogalma általánosítható az n -bites IBZ kapu fogalma segítségével. Ez egy 2^n hosszú vezérlőszóhoz rendel hozzá egy $g: \mathcal{B}^n \rightarrow \mathcal{B}$ n -bites Boole függvényt az $f_{IBZ}: \mathcal{B}^{2^n} \rightarrow (\mathcal{B}^n \rightarrow \mathcal{B})$ IBZ függvény alapján. Ennek a motivációja abban keresendő, hogy egy ALU két bemenetei közül általában csak az egyik csatlakozik a regiszterbuszhoz, a másik egy speciális regiszterhez, az ún. *akkumulátor*ba vezet, amely egyben az ALU művelet eredményét tároló regiszter. Így, ha két regiszter tartalmán szeretnénk műveletet elvégezni, az egyiket először be kell tölteni az akkumulátorba, majd a következő órajelciklusban (!) elvégezni a műveletet a másik regiszterben tároltakkal. Erre két megoldás kínálkozik. Az egyik, hogy az ALU mindkét bementét hozzákötjük egy latch-en keresztül egy új regiszterbuszhoz,¹⁰⁰ a másik, hogy az akkumulátort megtartva, háromoperandusúvá tesszük az ALU-t. Ez utóbbi megoldást vizsgáljuk meg, azaz az $n = 3$ esetet. A háromoperandusú ALU-val két regiszteren elvégzett műveletet az *akkumulátor* tartalma is képes befolyásolni. Így az előzőleg elvégzett ALU műveletnek nemcsak a *flagek* módosításával van lehetősége befolyásolni a következő műveletet, hanem az akkumulátorban történő tárolással a teljes eredmény hatással lehet rá.



5-5. ábra: A háromoperandusú ALU

A háromoperandusú ALU-ban az összeadás továbbra is csak két műveleten végezhető el, a harmadik művelet általában logikai vagy adatmozgató. A Segéd IBZ egység megvalósítására számtalan módosítás lehetséges, amik közül az összes tárgyalása jelen szűkös kereteink között nem lehetséges.

¹⁰⁰ (TANENBAUM, 2006)

5.4 Utasításkészlet

Az új, háromoperandusú ALU-ra építve egy kibővített utasításkészlet készíthető. bevezetőként nézzünk egy szélsőséges példát!

Példa 1: Általában ennél kevésbé radikális utasításszám-csökkenés érhető el, az alábbiakkal csak azt szeretném szemléltetni, hogy néhány időigényes szubrutin hogyan tehető sokkal egyszerűbbé. Két regiszterben tárolt szavakon végrehajtott műveletet maszkolni szeretnénk az akkumulátor tartalmával. Kétooperandusú ALU esetén a következő négy utasítás hajtódik végre:

1.	STORE A $\rightarrow$ reg3	Akkumulátor tartalmának elmentése a 3-as regiszterbe.
2.	LOAD reg1 $\rightarrow$ A	Az 1-es regiszter tartalmának betöltése az akkumulátorba.
3.	OPERATION(reg2, A)	Művelet elvégzése a 2-es regiszter és az akkumulátor között
4.	MASK(reg3, A)	A 3-as regiszterrel maszkoljuk az akkumulátort.

5-2. táblázat: Kétooperandusú ALU utasítások

Háromoperandusú ALU-ban ez egy utasítást jelent: MASK(OPERATION(reg1, reg2), A), azaz $f_{IBZ}(\gamma) = \text{MASK} \circ (I \otimes \text{OPERATION}) \in \mathcal{B}^3 \rightarrow \mathcal{B}$.

Állítás: Egy γ IBZ vezérlőszó rendelhető egy (γ_0, c) rendezett ketteshez, ahol γ_0 az őszvezérszó, c pedig az IBZ szelet vezérlése. Egy példa: az összeadás műveletnél $s = c^{in} \oplus (a \oplus b)$ összeg, ahol $\gamma = c^{in} \otimes \neg c^{in} \otimes \neg c^{in} \otimes c^{in} = c^{in} \oplus (\beta_0 \otimes \beta_1 \otimes \beta_1 \otimes \beta_0)$. Ekkor $\gamma_0 = (\beta_0 \otimes \beta_1 \otimes \beta_1 \otimes \beta_0)$ az őszvezérszó, egy XOR művelet, és $c = c^{in}$, a carry in lesz a vezérlés.

Definíció: IBZ Utasítás az az $U_{IBZ}(\gamma_0, \mathcal{C}, \mathcal{A})$ rendezett hármas, amelyhez rendelhető egy $M = f_{IBZ}(\gamma(\gamma_0, c))$ Boole műveletet megvalósító IBZ függvény. Itt $(\gamma_0, \mathcal{C})$ az ALU globális vezérlését jelenti, ahol $\mathcal{C}$ minden egyéb vezérlést jelent a γ_0 őszvezérszó mellett. $\mathcal{A}$ az ALU-t nem, csak a számítógép-architektúra egyéb elemeit érintő vezérlés.

Definíció: IBZ Utasításkészlet az $\mathcal{U}_{IBZ} = \{U | U_{IBZ}\}$ halmaz.

Definíció: *IBZ Program* a $\mathcal{P} = \{\mathcal{U}_{IBZ}, \circ\}^*$ mondatok halmaza,¹⁰¹ amelyhez létezik $\Delta: \{\mathcal{U}_{IBZ}, \circ\}^* \rightarrow \{\delta, \circ\}^*$ leképezés az $\mathcal{U}_{IBZ}$ IBZ utasításkészletből a δ Turing-utasítások mondatainak halmazába. A $\circ$ binér művelet két utasítás kompozíciója, egymás után történő végrehajtása.

Általános séma két utasítás egyidejű elvégzésére, ha a hozzájuk rendelhető $(U_1 \rightarrow M_1$ és $U_2 \rightarrow M_2)$ M_1 és M_2 kétbites műveleteket az $M' = M_1 \circ (I \otimes M_2)$ művelettel helyettesíthetjük, ekkor $U' \rightarrow M'$. Azokat az utasításokat, amiket el lehet végezni ilyen módon, **IBZ-összevonható** utasításoknak hívom.

Definíció: IBZ-összevonás a $\Omega: U \times U \rightarrow U_{IBZ}$ leképezés, ahol U tetszőleges utasításhalmaz, U_{IBZ} az U utasításhalmaz feletti IBZ utasításhalmaz.

Egy tetszőleges utasításkészlet módosítható úgy, hogy tartalmazza az összes, U_1 és U_2 IBZ-összevonható utasításokból képzett háromoperandusú U' utasítást. Egy ilyen processzor egy programjában, ha egymás után két darab IBZ-összevonható utasítás áll, azaz $p \in \mathcal{P}$, $p = (\dots \circ U_1 \circ U_2 \circ \dots)$, akkor egy két elemet tartalmazó zárójelezés matematikai feladatát megoldva, $p' = (\dots \circ U' \circ \dots)$ program konstruálható, ahol $U' = \Omega(U_1, U_2)$ az IBZ-összevont utasítás.

¹⁰¹ H halmaz, ekkor H^* az összes konstruálható *mondat* halmaza, lásd az Algoritmuselmélet részt.

6 Szimulációs eredmények összefoglalása és értékelésük

Az elméleti elgondolások önmagukban nem nyújtanak tényleges eredményt egy, a gyakorlati alkalmazásoknak szánt eszközről, ezért elengedhetetlen, hogy a matematikai leírást a valóságban is igazoljuk. A mai modern FPGA-k megfelelő anyagi háttér és technológiai ismeret (ezek szükségesek egy ASIC, alkalmazás-specifikus IC megvalósításához) nélkül is biztosítanak a tervező számára önellenőrzést. A hardverleíró nyelvek, köztük a VHDL nyelv sajátossága, hogy több elvonatkoztatási szinten is leírhatjuk az eszközünket, így szabadon dönthetünk, hogy az elméletet milyen szélesen terjesztjük ki, mielőtt azt implementálnánk az FPGA-ra. Az általam kidolgozott ML-RTL leírási szint jóval részletesebb, mint pusztán egy RTL leírás, mert kisebb részegységekre bontjuk az áramkörünket. A tesztelés teljes tartományát így fel tudom használni az eredményeim ellenőrzésére.



6-1. ábra: A tesztelés folyamata

Szimulációs eredmények

- Az ML-RTL leírás VHDL nyelven történő analízise, összehasonlítás a magasabb szintű leírásokkal
- FPGA szintézis és szimulációs módszerekkel megállapítottam, hogy az ML-RTL szinten leírt IBZ ALU helyfoglalása és sebessége kedvezőbb az RTL szinten leírt standard ALU architektúráéhoz képest kisebb.
- Az IBZ ISA implementációja egy AVR mikroprocesszorba.

Felhasznált eszközök:

- Az általam kifejlesztett egységeket VHDL¹⁰² nyelven implementáltam.
- Szimulációhoz a ModelSim¹⁰³ környezet hallgatói változatát használtam.
- Az időzítési analízist az Altera Quartus¹⁰⁴ környezetben végeztem.
- A C programok fordítását az AVR Studio 4¹⁰⁵ környezetben végeztem.

¹⁰² <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=893288>

¹⁰³ ModelSIM PE Student Edition 10.0a; Revision: 2011.02; Date: Feb. 20 2011. Copyright 1991-2011 Mentor Graphics Corporation

¹⁰⁴ Quartus II Version 10.1 Build 197 01/19/2011 SJ Web Edition, Service Pack Installed: 1; Copyright © 1991-2011 Altera Corporation.

6.1 Az IBZ szelet jellemzői

Az implementáció során az IBZ szelet a legfontosabb elem, hiszen ennek a késleltetési viszonyai minden műveletre hatással vannak. Ezért először végezzünk egy rövid vizsgálatot, összehasonlítást egy standard ALU szelethez képest! Legyen m a vezérlőbitek száma, és korábban használt n az architektúra bitszélessége. Egy elemi egybites műveletvégző blokkban elhelyezkedő műveletvégző egységek száma pedig p .

	standard ALU szelet	IBZ ALU szelet
Kapubemenetek száma	$O(2n \cdot 2^m)$	$O(2n \cdot m)$
Kapuk száma egy blokkban	$O(2^m)$	$O(m)$
Kaputípusok száma	$O(2^m)$	$O(m)$
Buszvezetékek száma	$3n + m$	$3n + m$
Megvalósítható műveletek	m	2^m

6-1. táblázat: A standard és IBZ szeletek összehasonlítása

Jól látható, hogy helyfoglalás szempontjából az IBZ ALU előnyösebb a műveletek száma és buszszélesség növelésénél. Így az előzőekben megvalósított $m = 2^n$ helyett m -et kisebbnek választva (nem lefedve az összes műveletet), optimalizálható az utasítások hossza a vezérlőszón keresztül. Az IBZ kapu analízisének láttuk, hogy a bemenetek számának k -val történő növelése az IBZ kaput alkotó multiplexerben k darab transzfer kapu késleltetéssel növeli az ALU működési idejét (korábban: Δy). Ez növeli az órajelciklus lehetséges legkisebb idejét, de az utasítások számát jelentősen lecsökkentettük. Ez az elméleti eredmények gyakorlati oldala. A szimuláció során ezzel kell majd összevetni a számított és valós eredményeket.

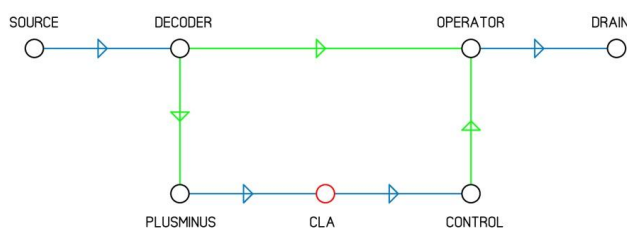
¹⁰⁵ ATMEL AVR Studio 4, Version 4.18, Build 684; Copyright © 1999-2009 Atmel Corporation. All rights reserved.

6.2 VHDL implementáció

Az IBZ szelet működésének részletes vizsgálatához annak funkcióját VHDL nyelven implementáltam. Egy magas szintű, algoritmikus leírás, és egy részegységekre bontott, vegyesen RTL és logikai szintű leírás keverékét vizsgáltam sebesség szempontjából. Az IBZ szelet entitásnak kétféle építményt adtam meg.

- Az egyik magasabb szintű RTL leírás, amely a VHDL beépített aritmetikai és logikai operátorait használja. Jóval rövidebb és kényelmesebb elkészíteni ezt, de a szintézist kevésbé tudjuk megszabni. Ez a leírás minden ALU vezérlésre megadja az elvégzendő műveletet, amit az IEEE könyvtárak tartalmaznak (az IEEE könyvtárak alatt jelen esetben az `ieee.std_logic_1164` és `ieee.std_logic_arith` értem).
- A másik az általam kidolgozott szint, az ML-RTL leírás. Az IBZ szeleten belül az IBZ kapukat és a CLA-t egyéb segédegységek mellett külön definiáltam benne. Ügyeltem arra, hogy olyan részletességig bontsam részegységekre az IBZ szeletet, hogy azok egyenként biztosan minimális késleltetésű áramkörökként szintetizálódjanak. Ezt a leírást azért készítettem el, hogy demonstráljam az IBZ kapuk tulajdonságát: sokféle funkciót meg lehet velük valósítani alacsony késleltetés mellett.

Célom ezzel kettős volt; egyrészt, teszteltem, hogy az IBZ szelet késleltetési viszonyai igazolják-e az elméleti eredményeket, másrészt, hogy a kidolgozott ML-RTL leírás valóban gyorsabb áramkörre szintetizálódik-e.

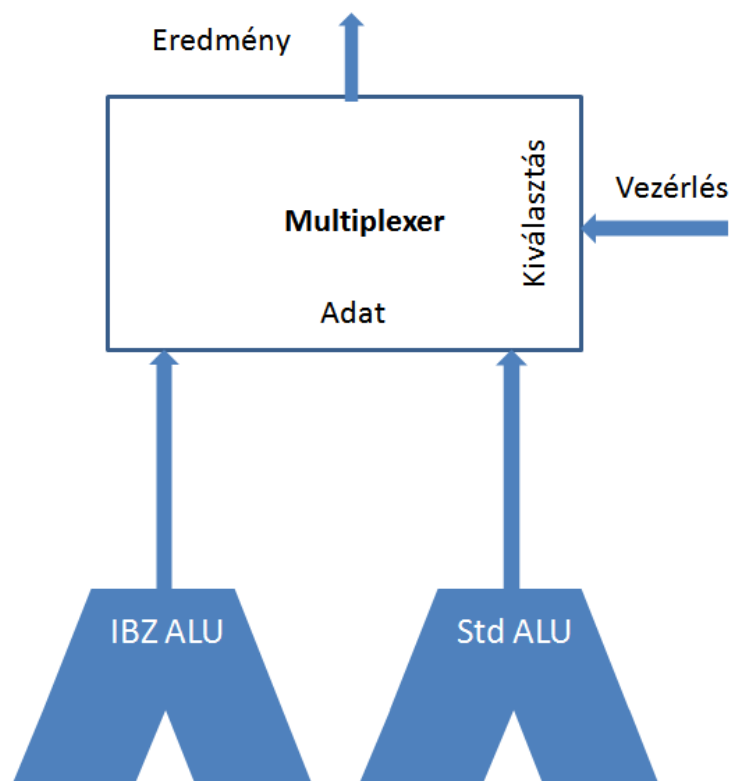


6-2. ábra: Az IBZ szelet ML-RTL vizsgálata

6.3 IBZ család alkalmazása mikroprocesszorban

Az IBZ ALU-hoz kerestem egy alkalmas mikroprocesszor VHDL modellt. A választásom az Atmel AVR Attiny X61 mikroprocesszorokra¹⁰⁶ esett, hiszen az AVR mikroprocesszorok RISC utasításkészletűek, így az IBZ ALU- és IBZ ISA-kompatibilisak. Az IBZ ALU beültetése a következő módon zajlott:

1. Első lépésben elhelyeztem az ALU-t minden összekötés nélkül a modellben.
2. Az eredeti ALU bemeneteit rákötöttem az IBZ ALU bemeneteire
3. A kimenetet rákötöttem egy multiplexerre, ami választani tud az eredeti és a beültetett ALU között.
4. A data fetch ciklus során beállítom a multiplexer vezérlését és az IBZ ALU control szavát. Minden utasításnál kiválasztom, melyik ALU-t kívánom használni. A műveletvégzés alatt mindössze kiolvasom multiplexer kimenetét.



6-3. ábra: Az ALU multiplexelés

A két ALU sebességét és helyfoglalását az Altera Quartus környezetben végeztem. Kiderült, hogy az IBZ ALU esetén a szükséges logikai elemek száma kevesebb, mint egyharmada a standard ALU által igényeltnek. A maximális működési frekvencia ellenben körülbelül 1.5-szerese.

¹⁰⁶ (HILVARSSON, 2008)

	standard ALU	IBZ ALU	std/IBZ	IBZ/std
Logikai elemek száma [db]	128	40	3.200	
Maximális frekvencia [MHz]	78.02	118.71		1.522

6-2. táblázat¹⁰⁷

A processzorra C programot fordítottam AVR Studio segítségével, aminek végrehajtását szimuláltam, és vizsgáltam, mely utasítások hajtódnak végre az AVR utasításkészletből.¹⁰⁸ Az IBZ ALU egy EOR és egy Decrement AVR assembly utasítás végrehajtásánál lett tesztelve. A működés funkcionálisan egyezett az eredeti processzormodellnél tapasztaltakkal.

¹⁰⁷ Az eredmények részletes ismertetését lásd a mellékletben.

¹⁰⁸ (ATMEL, 2010)

Eredmények értékelése

A dolgozatban részleteztem a vizsgálataimat és azok eredményeit. Lássuk, hogyan értékelhetők ezek!

- A Boole algebra **Boole modul** reprezentációja különösen hasznosnak bizonyult a hagyományos, algebrai alakon alapuló reprezentációhoz képest. Segítségével vált lehetővé az IBZ család megalkotása, az ML-RTL szint korrekt matematikai kezelése. Az egész jelen dolgozat stabil alapját képezi.
- Az **ML-RTL** szint hasznossága nyilvánvalóvá vált az FPGA szintézis eljárások többszöri lefuttatása után. Az összehasonlítás során kiszűrtem azt a lehetséges okot, hogy az IBZ család alacsony késleltetése lenne ezért a felelős – egy IBZ ALU-t implementáltam az RTL és ML-RTL leírások szintjén.
- Az **IBZ ALU**-ban történő alkalmazása mutatta ki igazán, hogy az **IBZ kapu** mire is képes. Segítségével alacsony késleltetésű, vezérelhető műveletvégzés valósítható meg. Az egészhez nem kell más, mint egy multiplexer – csak a szokásoshoz képest felcserélve a bemeneteit.
- Az **IBZ ALU** a CPU-kban található, kevés logikai műveletre képes, ámde nagysebességű ALU-k és ezek moduláris chipként kapható, sok Boole műveletre képes, de lassú párjaik előnyös tulajdonságát ötvözi – megkoronázva azzal a tulajdonságával, hogy minimális áldozatokkal lehet többoperandusú variánsát megépíteni.
- Az **IBZ ALU** előnyös tulajdonságait az **IBZ ISA**, azaz utasításkészlet használja ki. Teljesítőképességét egy C programmal vizsgáltam, és megállapítottam, hogy két egymás utáni utasítás összevonásával jelentősen gyorsabban futnak le a programok.

Felhasznált irodalom

- [1] ANDERSON, F., & FULLER, K. (1992). *Rings and Categories of Modules*. New York: Springer-Verlag.
- [2] ARATÓ, P. D. (1984). *Logikai rendszerek tervezése*. Budapest: Műegyetemi tankönyvkiadó.
- [3] ATMEL, I. (2010). Letöltés dátuma: 2011. May, forrás: [www.atmel.com: http://www.atmel.com/dyn/products/documents.asp?category_id=163&family_id=607&subfamily_id=791](http://www.atmel.com/dyn/products/documents.asp?category_id=163&family_id=607&subfamily_id=791)
- [4] BACH, I. (2002). *Formális nyelvek*. Budapest: TYPOTEX Kiadó.
- [5] BAUGH, C., & WOOLEY, B. (1973). A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Transactions on Computers* , 1045 - 1047 .
- [6] BEWICK, G. W. (1994. február). Fast multiplication: Algorithms and implementation. Palo Alto, California, United States of America.
- [7] BEWICK, G., SONG, P., MICHELI, G. D., & FLYNN, M. J. (1988). Approaching a Nanosecond : A 32-Bit Adder. *Proceedings of the 1988 IEEE International Conference on Computer* (old.: 221–226). Rye Brook, NY , USA: IEEE.
- [8] BOOTH, A. D. (1951). A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics* , 236-240.
- [9] BRENT, R., & KUNG, H. (1982). A Regular Layout for Parallel Adders. *Computers, IEEE Transactions on* , 260.
- [10] CALLWAY, T. K., & SWARTZLANDER, E. E. (1992). Optimizing arithmetic elements. *VLSI Signal Processing* , 91–100.
- [11] COOK, S. A. (1966). *On the minimum computation time of functions*, Ph.D. thesis. Boston, Massachusetts, USA: Harvard University.
- [12] DADDA, L. (1965). Some schemes for parallel multipliers. *Alta Frequenza* , 349–356.
- [13] Fairchild, S. (2000. April). DM74LS181 4-Bit Arithmetic Logic Unit.
- [14] FRIEDEN, B. R. (2004). *Science from Fisher Information: A Unification*, 2nd Ed. Cambridge, United Kingdom: Cambridge University Press.
- [15] FRIEDL, K. d. (2010). *Nyelvek és Automaták - órai jegyzet*. Budapest.
- [16] FULTON, W., & HARRY, J. (1991). *Representation theory: a first course*. New York: Springer-Verlag.
- [17] FÜRER, M. (2007). Faster Integer Multiplication. *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing* , 57-66.
- [18] GIVANT, S. R., & HALMOS, P. R. (2009). *Introduction to Boolean algebras*. Springer.
- [19] GOTO, G., INOUE, A., OHE, R., KASHIWAKURA, S., MITARAI, S., TSURU, T., és mtsai. (1997). A 4.1-ns Compact 54x54-b Multiplier. *IEEE JOURNAL OF SOLID-STATE CIRCUITS* , 1676-1682.
- [20] HALMOS, P. R. (1974). *Finite-dimensional vector spaces*. New York: Springer.
- [21] HAN, T., & CARLSON, D. A. (1987). Fast Area-Efficient VLSI Adders. *8th symposium on Computer Arithmetic*, (old.: 49-56).
- [22] HAZEWINDEL, M., GUBARENI, N. M., & KIRICHENKO, V. V. (2004). *Algebras, rings and modules*. Dordrecht, The Netherlands: Kluwer Academic Publishers.
- [23] HERNANDEZ-AGUIRRE, A., BUCKLES, B., & COELLO-COELLO, C. (2000). Gate-level synthesis of Boolean functions using binary multiplexers and genetic programming. *Proceedings of the 2000 Congress on Evolutionary Computation* , 675 - 682.
- [24] HILVARSSON, A. (2008. Sep 14). AVRtinyX61core. Sweden.
- [25] HOLDSWORTH, B., & WOODS, C. (2002). *Digital Logic Design, Fourth Edition*. Woburn, Massachusetts, United States of America: Newnes.
- [26] HOPCROFT, J., & ULLMAN, J. (1979). *Introduction to automata theory, languages and computation (1st ed.)*. ADDISON-WESLEY, READING MASS.
- [27] KARATSUBA, A., & OFMAN, Y. (1960). Multiplication of Many-Digital Numbers by Automatic Computers. *Proceedings of the USSR Academy of Sciences*, (old.: 293–294). Moscow, Moscow Oblast, Russia.
- [28] KATONA, G., RECSKI, A., & SZABÓ, A. (2006). *A számítástudomány alapjai*. Budapest: Typotex.
- [29] KIM, K. H. (1982). *Boolean Matrix Theory and Applications*. Dekker .
- [30] KNUTH, D. (1997). *The Art of Computer Programming, Volume 2. Third Edition*. Boston, Massachusetts, United States of America: Addison-Wesley.
- [31] KOGGE, P. M. (1973. Aug.). A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *Computers, IEEE Transactions on* , 786.

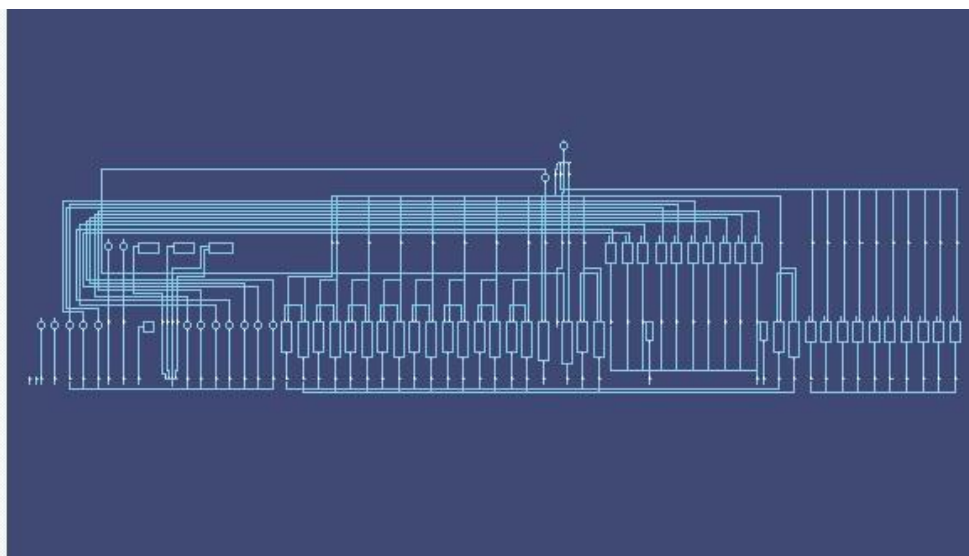
- [32] LADNER, R. E., & FISCHER, M. J. (1980). Parallel Prefix Computation. *Journal of the Association for Computing Machinery* , 831-838.
- [33] LANTOS, B. (2001). *Fuzzy systems and generic algorithms*. Budapest: Műegyetemi kiadó.
- [34] LEIJTEN-NOWAK, K. (2004). *Szabadalom száma: US7251672*. United States of America.
- [35] LIDL, R., & NIEDERREITER, H. (1997). *Finite Fields*. Cambridge, United Kingdom: Cambridge University Press.
- [36] LING, H. (1981). High speed binary adder. *IBM Journal of Research and Development* , 156-166.
- [37] LING, H. (1966). High speed binary parallel adder. *IEEE Transactions on Computers* , 799-802.
- [38] MIZRAJI, E. (1992). Vector logics: the matrix-vector representation of logical calculus. *Fuzzy Sets and Systems* , 179-185.
- [39] PAI, Y.-T., & CHEN, Y.-K. (2004). The fastest carry lookahead adder. *Second IEEE International Workshop on Electronic Design, Test and Applications* , 434 - 436 .
- [40] PARHAMI, B. (2000). *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford: Oxford University Press.
- [41] ROTH, C. J., & KINNEY, L. L. (2010). *Fundamentals of Logic Design*. Stamford, CT, USA: CENGAGE Learning.
- [42] SCHAFER, I., & PERKOWSKI, M. (1993). Synthesis of multilevel multiplexer circuits for incompletely specified multioutput Boolean functions with mapping to multiplexer based FPGA's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* , 1655 - 1664 .
- [43] SCHÖNHAGE, A., & STRASSEN, V. (1971). Schnelle Multiplikation großer Zahlen. *Computing* 7 , 281-292.
- [44] SHANNON, C. (1949). The Synthesis of Two-Terminal Switching Circuits. *Bell System Technical Journal* , 28: 59–98.
- [45] SHIVA, S. G. (2000). *Computer Design and Architecture, third edition, revised and expanded*. New York, NY, United States of America: Marcel Dekker.
- [46] SKLANSKY, J. (1960). Conditional-Sum Addition Logic. *IRE Transactions on Electronic Computers* , 226-231.
- [47] STONE, M. (1936). The Theory of Representations for Boolean Algebras. *Transactions of the American Mathematical Society* , Vol. 40, No. 1) 40 (1): 37–111.
- [48] SZITTYA, O., & JÁVOR, A. (1984). Logikai rendszerek. In *Mikroelektronikai berendezés-orientált áramkörök tervezése* (old.: 430). Budapest: EDUSYSTEM Oktatásfejlesztési Pjt.
- [49] TANENBAUM, A. S. (2006). *Structured Computer Organization - 5th edition*. Englewood Cliffs, New Jersey, USA: Prentice-Hall.
- [50] TURING, A. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, (old.: 42 (2)). London.
- [51] VLADIMIROV, D. (2002). *Boolean algebras in analysis*. Dordrecht, The Netherlands: Kluwer Academic Publishers.
- [52] WALLACE, C. S. (1964). A suggestion for a fast multiplier . *IEEE Transactions on Electronic Comp.* , 14-17.
- [53] WANG, Z., JULIEN, G. A., MILLER, W. C., WANG, J., & BIZZAN, S. S. (1997). Fast Adders Using Enhanced Multiple-Output Domino Logic. *IEEE Journal of Solid-State Circuits* , 206-214.
- [54] ZHU, H., CHENG, C.-K., & GRAHAM, R. (2005). *Constructing Zero-deficiency Parallel Prefix Adder of Minimum Depth*. La Jolla, California: IEEE.
- [55] ZHU, H., CHENG, C.-K., & GRAHAM, R. (2006). On the Construction of Zero-Deficiency Parallel Prefix Circuits with Minimum Depth. *ACM Transactions on Design Automation of Electronic Systems* , 387–409.

Melléklet

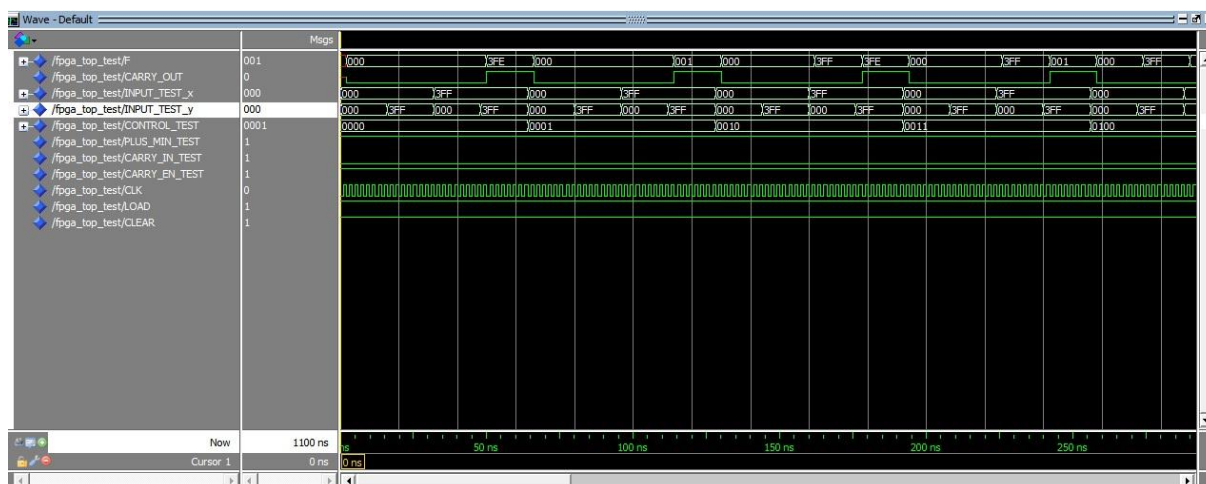
6.4 Az IBZ ALU VHDL implementációja és szimulációja

Name	Status	Type	Order
ALU.vhd	✓	VHDL	0
ALU_test.vhd	✓	VHDL	1
CLA.vhd	✓	VHDL	2
FPGA_TOP.vhd	✓	VHDL	3
FPGA_TOP_test.vhd	✓	VHDL	4
IBZ4_stateout.vhd	✓	VHDL	5
IBZ4_stateout_test.vhd	✓	VHDL	6
OP.vhd	✓	VHDL	7
Plus_minus_mux.vhd	✓	VHDL	8

6-4: A szimulációhoz használt VHDL projekt



6-5: A logikai szintézis eredménye



6-6: A szimuláció, logikai működés helyességének vizsgálata

6.4.1 Az IBZ kapu

```
Ln#
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity State_decoder_ent is
5  port(
6      x: in std_logic;
7      y: in std_logic;
8      state: out std_logic_vector(3 downto 0)
9  );
10 end State_decoder_ent;
11
12 --DATAFLOW DESCRIPTION
13
14 architecture State_decoder_dtf of State_decoder_ent is
15 begin
16     state <= (3 => (x nor y), 2 => (x nor (not y)), 1 => ((not x) nor y), 0 => (x and y));
17 end State_decoder_dtf;
18
19 -----
20 library ieee;
21 use ieee.std_logic_1164.all;
22
23 entity State_encoder_ent is
24 port(
25     state: in std_logic_vector(3 downto 0);
26     CONTROL: in std_logic_vector(3 downto 0);
27     F: out std_logic
28 );
29 end State_encoder_ent;
30
31 --DATAFLOW DESCRIPTION
32
33 architecture State_encoder_dtf of State_encoder_ent is
34 begin
35     F <= '0' when (state and CONTROL) = "0000" else
36         '1';
37
38 end State_encoder_dtf;
```

6-7: Az IBZ kapu entitás

```
Ln#
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity State_decoder_ent is
5  port(
6      x: in std_logic;
7      y: in std_logic;
8      state: out std_logic_vector(3 downto 0)
9  );
10 end State_decoder_ent;
11
12 --DATAFLOW DESCRIPTION
13
14 architecture State_decoder_dtf of State_decoder_ent is
15 begin
16     state <= (3 => (x nor y), 2 => (x nor (not y)), 1 => ((not x) nor y), 0 => (x and y));
17 end State_decoder_dtf;
18
19 -----
20 library ieee;
21 use ieee.std_logic_1164.all;
22
23 entity State_encoder_ent is
24 port(
25     state: in std_logic_vector(3 downto 0);
26     CONTROL: in std_logic_vector(3 downto 0);
27     F: out std_logic
28 );
29 end State_encoder_ent;
30
31 --DATAFLOW DESCRIPTION
32
33 architecture State_encoder_dtf of State_encoder_ent is
34 begin
35     F <= '0' when (state and CONTROL) = "0000" else
36         '1';
37
38 end State_encoder_dtf;
```

6-8: Az állapotdekódér

Ln#	
29	end State_encoder_ent;
30	
31	--DATAFLOW DESCRIPTION
32	
33	architecture State_encoder_dtf of State_encoder_ent is
34	begin
35	F <= '0' when (state and CONTROL) = "0000" else
36	'1';
37	
38	end State_encoder_dtf;
39	
40	-----
41	
42	
43	library ieee;
44	use ieee.std_logic_1164.all;
45	
46	entity IBZ4_stateout_ent is
47	port(
48	x: in std_logic;
49	y: in std_logic;
50	CONTROL: in std_logic_vector(3 downto 0);
51	F: out std_logic;
52	STATE: out std_logic_vector(3 downto 0)
53	);
54	end IBZ4_stateout_ent;
55	
56	

6-9: A „mátrixszorzó” funkcionális blokk

Ln#	
55	
56	
57	
58	--ARCHITECTURAL DESCRIPTION
59	
60	architecture IBZ4_stateout_arch of IBZ4_stateout_ent is
61	
62	component State_decoder_ent is
63	port(
64	x: in std_logic;
65	y: in std_logic;
66	state: out std_logic_vector(3 downto 0)
67	);
68	end component;
69	
70	component State_encoder_ent is
71	port(
72	state: in std_logic_vector(3 downto 0);
73	CONTROL: in std_logic_vector(3 downto 0);
74	F: out std_logic
75	);
76	end component;
77	
78	
79	SIGNAL state_inner : std_logic_vector(3 downto 0);
80	
81	begin
82	State_decoder_inst : State_decoder_ent
83	PORT MAP (x => x, y => y, state => state_inner);
84	
85	State_encoder_inst : State_encoder_ent
86	PORT MAP (state => state_inner, CONTROL => CONTROL, F => F);
87	
88	STATE <= state_inner;
89	
90	end IBZ4_stateout_arch;
91	
92	

6-10: Az egységek huzalozása

```

92
93 --DATAFLOW DESCRIPTION
94
95 architecture IBZ4_stateout_dtf of IBZ4_stateout_ent is
96
97     SIGNAL state_inner : std_logic_vector(3 downto 0);
98
99     begin
100
101         state_inner<= (3 => (x nor y), 2 => (x nor (not y)), 1 => ((not x) nor y), 0 => (x and y));
102         F <= '0' when (state_inner and CONTROL) = "0000" else
103             '1';
104         STATE <= state_inner;
105
106     end IBZ4_stateout_dtf;
107

```

6-11: Az IBZ kapu állapotkivezetései

Ln#	
1	library ieee;
2	use ieee.std_logic_1164.all;
3	use ieee.std_logic_arith.all;
4	use ieee.std_logic_unsigned.all;
5	
6	
7	entity IBZ4_stateout_test is
8	PORT (
9	F: BUFFER std_logic;
10	STATE: BUFFER std_logic_vector(3 downto 0)
11	);
12	end;
13	
14	architecture only of IBZ4_stateout_test is
15	
16	COMPONENT IBZ4_stateout_ent port (
17	x: in std_logic;
18	y: in std_logic;
19	CONTROL: in std_logic_vector(3 downto 0);
20	F: BUFFER std_logic;
21	STATE: BUFFER std_logic_vector(3 downto 0)
22	);
23	END COMPONENT ;
24	
25	SIGNAL CONTROL_TEST: std_logic_vector(3 downto 0) := "0000";
26	SIGNAL INPUT_TEST: std_logic_vector(1 downto 0) := "00";

6-12: Az IBZ kapu tesztelése

```

28 begin
29
30 mapping : IBZ4_stateout_ent
31     PORT MAP (
32         x => INPUT_TEST(0),
33         y => INPUT_TEST(1),
34         CONTROL => CONTROL_TEST(3 downto 0),
35         F => F,
36         STATE => STATE
37     );
38
39 a0 : PROCESS
40     begin
41         wait for 1 ns; INPUT_TEST <= INPUT_TEST + 1;
42     end PROCESS a0;
43
44
45 a2 : PROCESS
46     begin
47         wait for 4 ns; CONTROL_TEST <= CONTROL_TEST + 1;
48     end PROCESS a2;
49
50
51 end only;
52

```

6-13: A teszteléshez használt jelek

6.4.2 Az IBZ szelet többi eleme: a Segéd IBZ egység és a vezérlő

Ln#	
1	library ieee;
2	use ieee.std_logic_1164.all;
3	
4	entity Plus_minus_mux_ent is
5	port (
6	state: in std_logic_vector(3 downto 0);
7	plus_min: in std_logic;
8	carry_generate: out std_logic;
9	carry_propagate_neg: out std_logic
10	);
11	end Plus_minus_mux_ent;
12	
13	--BEHAVIORAL DESCRIPTION
14	
15	architecture Plus_minus_mux_beh of Plus_minus_mux_ent is
16	begin
17	with plus_min select
18	carry_generate <= state(0) when '1',
19	state(1) when '0',
20	'X' when others;
21	
22	with plus_min select
23	carry_propagate_neg <= state(3) when '1',
24	state(2) when '0',
25	'X' when others;
26	end Plus_minus_mux_beh;
27	
28	
29	
30	

6-14: A Segéd IBZ egység

Ln#	
1	library ieee;
2	use ieee.std_logic_1164.all;
3	
4	entity OP_ent is
5	port (
6	carry_neg: in std_logic;
7	control: in std_logic_vector(3 downto 0);
8	F: out std_logic_vector(3 downto 0)
9	);
10	end OP_ent;
11	
12	--BEHAVIORAL DESCRIPTION
13	
14	architecture OP_beh of OP_ent is
15	begin
16	F <= control when carry_neg = '1' else (not control);
17	end OP_beh;
18	

6-15: Az IBZ szelet vezérlése

6.4.3 A CLA fa

Ln#	
1	LIBRARY ieee;
2	USE ieee.std_logic_1164.ALL;
3	
4	ENTITY CLA_ent IS
5	generic(n: positive :=8);
6	PORT
7	(
8	carry_generate : IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
9	carry_propagate_neg : IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
10	carry_en: in std_logic;
11	carry_in : IN STD_LOGIC;
12	carry_control_neg : OUT STD_LOGIC_VECTOR(n-1 DOWNT0 0);
13	carry_out_neg : OUT STD_LOGIC
14	);
15	END CLA_ent;
16	
17	--BEHAVIORAL DESCRIPTION
18	ARCHITECTURE CLA_beh OF CLA_ent IS
19	
20	SIGNAL carry_internal_neg : STD_LOGIC_VECTOR(n DOWNT0 0);
21	
22	BEGIN
23	
24	carry_internal_neg(0) <= not carry_in;
25	
26	carry: for i in 0 to n-1 generate
27	carry_internal_neg(i+1) <= carry_generate(i) NOR (carry_propagate_neg(i) NOR carry_internal_neg(i));
28	end generate carry;
29	
30	with carry_en select
31	carry_control_neg <= carry_internal_neg(n-1 DOWNT0 0) when '1',
32	(others => '1') when others;
33	
34	carry_out_neg <= carry_internal_neg(n);
35	
36	END CLA_beh;
37	
38	

6-16: A Carry-lookahead prefix fa

6.4.4 Az IBZ ALU

Ln#	
1	library ieee;
2	use ieee.std_logic_1164.all;
3	use ieee.std_logic_unsigned.all;
4	use ieee.std_logic_arith.all;
5	
6	
7	entity ALU_ent is
8	generic(n: positive :=8);
9	port (
10	x: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
11	y: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
12	CONTROL: IN STD_LOGIC_VECTOR(3 downto 0);
13	PLUS_MIN: IN STD_LOGIC;
14	CARRY_EN: IN STD_LOGIC;
15	CARRY_IN: IN STD_LOGIC;
16	CARRY_OUT: OUT STD_LOGIC;
17	F: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0)
18	);
19	end ALU_ent;
20	
21	

6-17: Az IBZ ALU egyed

Ln#	
21	
22	--BEHAVIOURAL DESCRIPTION
23	
24	architecture ALU_beh of ALU_ent is
25	
26	begin
27	
28	F <= x+y+CARRY_IN when (PLUS_MIN and CARRY_EN) = '1' else
29	x-y+CARRY_IN-'1' when ((not PLUS_MIN) and CARRY_EN) = '1' else
30	(others => '0')
31	x and y when CONTROL = "0000" else
32	x and (not y) when CONTROL = "0001" else
33	x when CONTROL = "0010" else
34	(not x) and y when CONTROL = "0011" else
35	y when CONTROL = "0100" else
36	x xor y when CONTROL = "0101" else
37	x or y when CONTROL = "0110" else
38	x nor y when CONTROL = "0111" else
39	x xnor y when CONTROL = "1000" else
40	not y when CONTROL = "1001" else
41	x nor (not y) when CONTROL = "1010" else
42	not x when CONTROL = "1011" else
43	(not x) nor y when CONTROL = "1100" else
44	x nand y when CONTROL = "1101" else
45	(others => '1')
46	(others => 'X');
47	
48	end ALU_beh;
49	
50	

6-18: Az IBZ ALU RTL szintű leírása

Ln#	
51	
52	--ARCHITECTURAL DESCRIPTION
53	
54	architecture ALU_arch of ALU_ent is
55	
56	component CLA_ent IS
57	generic(n: positive);
58	PORT
59	{
60	carry_generate : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
61	carry_propagate_neg : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
62	carry_en: in std_logic;
63	carry_in : IN STD_LOGIC;
64	carry_control_neg : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
65	carry_out_neg : OUT STD_LOGIC
66	};
67	end component;
68	
69	component IB24_stateout_ent is
70	port
71	{
72	x: in std_logic;
73	y: in std_logic;
74	CONTROL: in std_logic_vector(3 downto 0);
75	F: out std_logic;
76	STATE: out std_logic_vector(3 downto 0)
77	};
78	end component;
79	
80	component OP_ent is
81	port
82	{
83	carry_neg : in std_logic;
84	control: in std_logic_vector(3 downto 0);
85	F: out std_logic_vector(3 downto 0)
86	};
87	end component;
88	

6-19: Az IBZ ALU ML-RTL szintű leírása – modulokra bontás

Ln#	
89	component Plus_minus_mux_ent is
90	port (
91	state: in std_logic_vector(3 downto 0);
92	plus_min: in std_logic;
93	carry_generate: out std_logic;
94	carry_propagate_neg: out std_logic
95	);
96	end component;
97	
98	SIGNAL CARRY_BUS_NEG : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
99	SIGNAL STATE_BUS : STD_LOGIC_VECTOR(4*n-1 DOWNTO 0);
100	SIGNAL PROPAGATE_BUS_NEG : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
101	SIGNAL GENERATE_BUS : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
102	SIGNAL CONTROL_BUS : STD_LOGIC_VECTOR(4*n-1 DOWNTO 0);
103	SIGNAL CARRY_OUT_MINUS : STD_LOGIC;
104	
105	begin
106	
107	CARRY_OUT <= CARRY_OUT_MINUS when PLUS_MIN = '0' else
108	not CARRY_OUT_MINUS when PLUS_MIN = '1' else
109	'X';
110	
111	CLA_inst : CLA_ent
112	GENERIC MAP (n=>n)
113	PORT MAP
114	{
115	carry_generate =>GENERATE_BUS,
116	carry_propagate_neg =>PROPAGATE_BUS_NEG,
117	carry_en =>CARRY_EN,
118	carry_in =>CARRY_IN,
119	carry_control_neg =>CARRY_BUS_NEG(n-1 DOWNTO 0),
120	carry_out_neg =>CARRY_OUT_MINUS
121	};

6-20: Az IBZ ALU ML-RTL leírása – belső carry busz belső huzalozása

```

122
123     ibz_gen: for i in 0 to n-1 generate
124
125     OP_inst : OP_ent
126     PORT MAP
127     (
128         carry_neg      => CARRY_BUS_NEG(i),
129         control        => CONTROL,
130         F              => CONTROL_BUS(4*(i+1)-1 DOWNT0 4*i)
131     );
132
133     Plus_minus_mux_inst : Plus_minus_mux_ent
134     PORT MAP
135     (
136         state => STATE_BUS(4*(i+1)-1 DOWNT0 4*i),
137         plus_min => PLUS_MIN,
138         carry_generate => GENERATE_BUS(i),
139         carry_propagate_neg => PROPAGATE_BUS_NEG(i)
140     );
141
142     IBZ4_inst : IBZ4_stateout_ent
143     PORT MAP
144     (
145         x      => x(i),
146         y      => y(i),
147         CONTROL      => CONTROL_BUS(4*(i+1)-1 DOWNT0 4*i),
148         STATE        => STATE_BUS(4*(i+1)-1 DOWNT0 4*i),
149         F            => F(i)
150     );
151
152
153     end generate ibz_gen;
154
155
156 end ALU_arch;
157

```

6-21: Az IBZ szelet elemei – az IBZ kapu, a Segéd IBZ egység és a vezérlés

6.4.5 Az FPGA szintézishez használt egyed

Ln#	
1	library ieee ;
2	use ieee.std_logic_1164.all;
3	use ieee.std_logic_unsigned.all;
4	
5	-----
6	
7	entity reg_ent is
8	
9	generic(n: positive :=8; edge: std_logic :='1');
10	port(
11	I: in std_logic_vector(n-1 downto 0);
12	clock: in std_logic;
13	load: in std_logic;
14	clear: in std_logic;
15	Q: out std_logic_vector(n-1 downto 0)
16	);
17	end reg_ent;
18	
19	-----
20	

6-22: A ki-és bemeneti regiszterekhez használt regiszter egyed

```

--
19  -----
20
21  architecture reg_arch of reg_ent is
22
23      signal Q_tmp: std_logic_vector(n-1 downto 0);
24
25  begin
26
27      process(I, clock, load, clear)
28      begin
29
30          if clear = '0' then
31              -- use 'range in signal assignment
32              Q_tmp <= (Q_tmp'range => '0');
33          elsif (clock=edge and clock'event) then
34              if load = '1' then
35                  Q_tmp <= I;
36              end if;
37          end if;
38
39          end process;
40
41          -- concurrent statement
42          Q <= Q_tmp;
43
44  end reg_arch;
45
46  -----
47  -----
48

```

6-23: A regiszter szabályos leírása

```

45
46  -----
47  -----
48
49  library ieee ;
50  use ieee.std_logic_1164.all;
51  use ieee.std_logic_unsigned.all;
52
53  -----
54
55  entity FPGA_TOP_ent is
56
57  generic(n: positive :=8);
58  port (
59      CLK:      IN    STD_LOGIC;
60      LOAD:     IN    STD_LOGIC;
61      CLEAR:    IN    STD_LOGIC;
62      x:        IN    STD_LOGIC_VECTOR(n-1 DOWNT0 0);
63      y:        IN    STD_LOGIC_VECTOR(n-1 DOWNT0 0);
64      CONTROL:  IN    STD_LOGIC_VECTOR(3 downto 0);
65      PLUS_MIN: IN    STD_LOGIC;
66      CARRY_EN: IN    STD_LOGIC;
67      CARRY_IN: IN    STD_LOGIC;
68      CARRY_OUT:OUT STD_LOGIC;
69      F:        OUT   STD_LOGIC_VECTOR(n-1 DOWNT0 0)
70  );
71  end FPGA_TOP_ent;
72
73  -----
74

```

6-24: Az FPGA szintézishez használt egyed

```

73 -----
74
75 architecture FPGA_TOP_arch of FPGA_TOP_ent is
76
77 component ALU_ent IS
78     generic(n: positive);
79     port(
80         x:      IN    STD_LOGIC_VECTOR(n-1 DOWNTO 0);
81         y:      IN    STD_LOGIC_VECTOR(n-1 DOWNTO 0);
82         CONTROL: IN    STD_LOGIC_VECTOR(3 downto 0);
83         PLUS_MIN: IN    STD_LOGIC;
84         CARRY_EN: IN    STD_LOGIC;
85         CARRY_IN: IN    STD_LOGIC;
86         CARRY_OUT: OUT  STD_LOGIC;
87         F:      OUT    STD_LOGIC_VECTOR(n-1 DOWNTO 0)
88     );
89 end component;
90
91 component reg_ent IS
92     generic(n: positive; edge: std_logic);
93     port(
94         I: in std_logic_vector(n-1 downto 0);
95         clock: in std_logic;
96         load: in std_logic;
97         clear: in std_logic;
98         Q: out std_logic_vector(n-1 downto 0)
99     );
100 end component;
101
102 SIGNAL X_bus : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
103 SIGNAL Y_bus : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
104 SIGNAL CONTROL_bus : STD_LOGIC_VECTOR(3 DOWNTO 0);
105 SIGNAL PLUS_MIN_bus : STD_LOGIC;
106 SIGNAL CARRY_EN_bus : STD_LOGIC;
107 SIGNAL CARRY_IN_bus : STD_LOGIC;
108 SIGNAL CARRY_OUT_bus : STD_LOGIC;
109 SIGNAL F_bus : STD_LOGIC_VECTOR(n-1 DOWNTO 0);
110

```

6-25: Az IBZ ALU és regiszterek definiálása

Ln#	
111	begin
112	
113	ALU_inst : ALU_ent
114	GENERIC MAP (n=>n)
115	PORT MAP
116	(
117	x => X_bus,
118	y => Y_bus,
119	CONTROL => CONTROL_bus,
120	PLUS_MIN => PLUS_MIN_bus,
121	CARRY_EN => CARRY_EN_bus,
122	CARRY_IN => CARRY_IN_bus,
123	CARRY_OUT => CARRY_OUT_bus,
124	F => F_bus
125	);
126	
127	reg_in_x : reg_ent
128	GENERIC MAP (n=>n, edge=>'1')
129	PORT MAP
130	(
131	I => x,
132	clock => CLK,
133	load => LOAD,
134	clear => CLEAR,
135	Q => X_bus
136	);
137	
138	reg_in_y : reg_ent
139	GENERIC MAP (n=>n, edge=>'1')
140	PORT MAP
141	(
142	I => y,
143	clock => CLK,
144	load => LOAD,
145	clear => CLEAR,
146	Q => Y_bus
147	);
...	

6-26: Az IBZ ALU összekötése a bemeneti regiszterekkel

Ln#	
148	
149	reg_in : reg_ent
150	GENERIC MAP (n=>7, edge=>'1')
151	PORT MAP
152	(
153	I(3 downto 0) => CONTROL,
154	I(4) => PLUS_MIN,
155	I(5) => CARRY_EN,
156	I(6) => CARRY_IN,
157	clock => CLK,
158	load => LOAD,
159	clear => CLEAR,
160	Q(3 downto 0) => CONTROL_bus,
161	Q(4) => PLUS_MIN_bus,
162	Q(5) => CARRY_EN_bus,
163	Q(6) => CARRY_IN_bus
164	);
165	
166	reg_out_F : reg_ent
167	GENERIC MAP (n=>n, edge=>'0')
168	PORT MAP
169	(
170	I => F_bus,
171	clock => CLK,
172	load => LOAD,
173	clear => CLEAR,
174	Q => F
175	);
176	
176	
177	reg_out_carry : reg_ent
178	GENERIC MAP (n=>1, edge=>'0')
179	PORT MAP
180	(
181	I(0) => CARRY_OUT_bus,
182	clock => CLK,
183	load => LOAD,
184	clear => CLEAR,
185	Q(0) => CARRY_OUT
186	);
187	
188	
189	end FPGA_TOP_arch;
190	

6-27: A kimenet és a vezérlés regiszterei

Ln#	
1	library ieee;
2	use ieee.std_logic_1164.all;
3	use ieee.std_logic_unsigned.all;
4	use ieee.std_logic_arith.all;
5	
6	
7	entity FPGA_TOP_test is
8	generic(n: positive :=10; delay: time:=64 ns; delay_short: time:=16 ns; delay_clk: time:=1 ns);
9	PORT
10	(
11	F: BUFFER STD_LOGIC_VECTOR(n-1 DOWNT0 0);
12	CARRY_OUT:BUFFER STD_LOGIC
13	);
14	end FPGA_TOP_test;
15	
16	architecture only of FPGA_TOP_test is
17	
18	COMPONENT FPGA_TOP_ent is
19	generic(n: positive);
20	port(
21	CLK: IN STD_LOGIC;
22	LOAD: IN STD_LOGIC;
23	CLEAR: IN STD_LOGIC;
24	x: IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
25	y: IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
26	CONTROL: IN STD_LOGIC_VECTOR(3 downto 0);
27	PLUS_MIN: IN STD_LOGIC;
28	CARRY_EN: IN STD_LOGIC;
29	CARRY_IN: IN STD_LOGIC;
30	CARRY_OUT: OUT STD_LOGIC;
31	F: OUT STD_LOGIC_VECTOR(n-1 DOWNT0 0)
32	);
33	END COMPONENT ;
34	..

6-28: Az FPGA-ra szintetizálendő egyed tesztelése

6.4.6 Az FPGA egyed tesztelése

```

34
35  SIGNAL INPUT_TEST_x: STD_LOGIC_VECTOR(n-1 DOWNT0 0);
36  SIGNAL INPUT_TEST_y: STD_LOGIC_VECTOR(n-1 DOWNT0 0);
37  SIGNAL CONTROL_TEST: STD_LOGIC_VECTOR(3 DOWNT0 0) := "0000";
38  SIGNAL PLUS_MIN_TEST: STD_LOGIC := '1';
39  SIGNAL CARRY_IN_TEST: STD_LOGIC := '0';
40  SIGNAL CARRY_EN_TEST: STD_LOGIC := '0';
41  SIGNAL CLK: STD_LOGIC := '0';
42  SIGNAL LOAD: STD_LOGIC := '1';
43  SIGNAL CLEAR: STD_LOGIC := '1';
44
45  begin
46
47  FPGA_TOP_inst : FPGA_TOP_ent
48    GENERIC MAP(n => n)
49    PORT MAP (
50      CLK => CLK,
51      LOAD => LOAD,
52      CLEAR => CLEAR,
53      x => INPUT_TEST_x,
54      y => INPUT_TEST_y,
55      CONTROL => CONTROL_TEST,
56      PLUS_MIN => PLUS_MIN_TEST,
57      CARRY_EN => CARRY_EN_TEST,
58      CARRY_IN => CARRY_IN_TEST,
59      CARRY_OUT => CARRY_OUT,
60      F => F
61    );
62
63  clock : PROCESS
64    begin
65      wait for delay_clk;
66      CLK <= not CLK;
67    end PROCESS clock;
68
69

```

6-29: A jelvezetékek definiálása, FPGA egyed beültetése

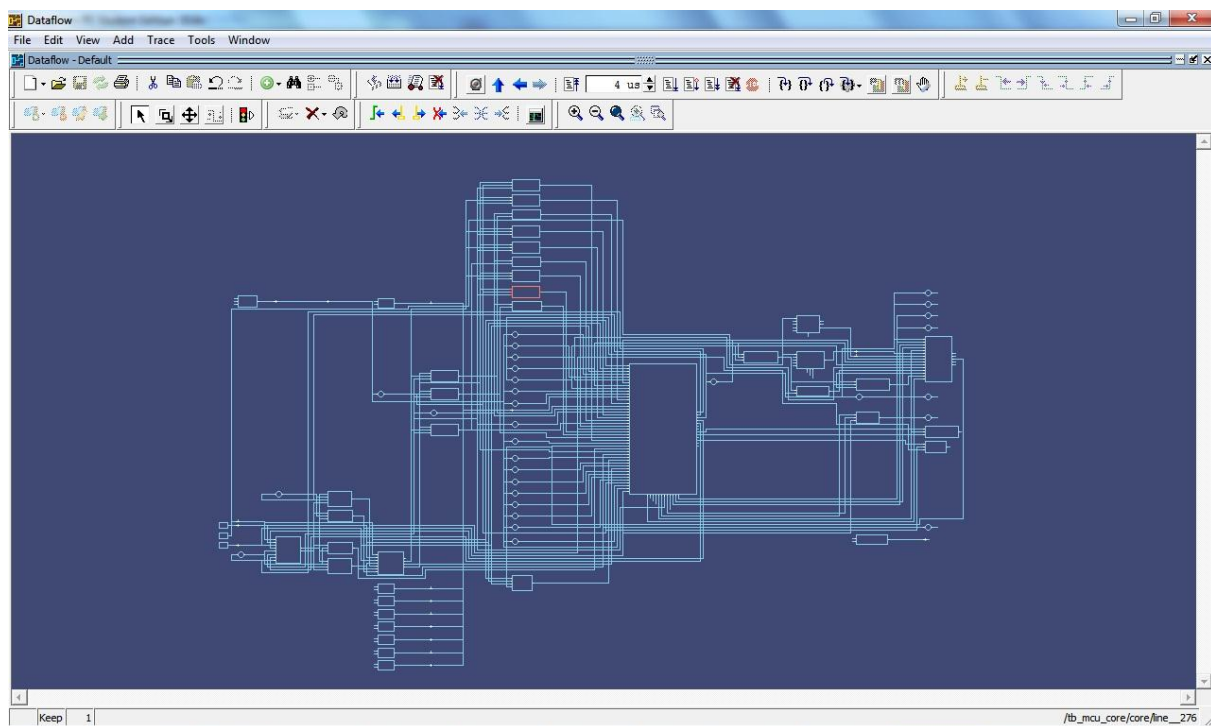
Ln#	
69	
70	logic : PROCESS
71	begin
72	wait for delay;
73	CONTROL_TEST <= CONTROL_TEST + 1;
74	end PROCESS logic;
75	
76	x : PROCESS
77	begin
78	INPUT_TEST_x <= (INPUT_TEST_x'range => '0');
79	wait for delay_short;
80	wait for delay_short;
81	INPUT_TEST_x <= (INPUT_TEST_x'range => '1');
82	wait for delay_short;
83	wait for delay_short;
84	end PROCESS x;
85	
86	y : PROCESS
87	begin
88	INPUT_TEST_y <= (INPUT_TEST_x'range => '0');
89	wait for delay_short;
90	INPUT_TEST_y <= (INPUT_TEST_x'range => '1');
91	wait for delay_short;
92	end PROCESS y;
93	
94	carry : PROCESS
95	begin
96	CARRY_EN_TEST <= not CARRY_EN_TEST;
97	wait for 16*delay;
98	CARRY_IN_TEST <= not CARRY_IN_TEST;
99	wait for 16*delay;
100	CARRY_IN_TEST <= not CARRY_IN_TEST;
101	end PROCESS carry;
102	
103	plus_min : PROCESS
104	begin
105	wait for 64*delay;
106	PLUS_MIN_TEST <= not PLUS_MIN_TEST;
107	end PROCESS plus_min;
108	
109	
110	
111	
112	end only;
113	

6-30: A tesztjelek definiálása

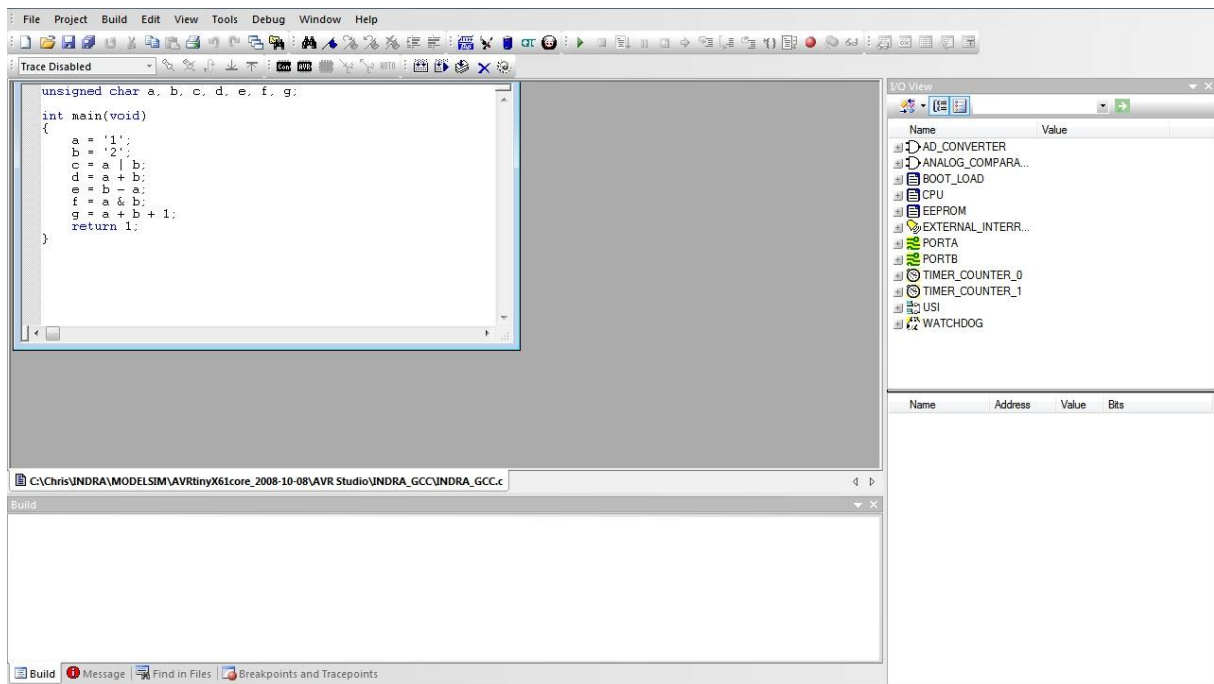
6.5 Az IBZ ALU alkalmazása AVR mikroprocesszorban

Name	Status	Type	Order
ALU.vhd	✓	VHDL	3
CLA.vhd	✓	VHDL	4
IBZ4_stateout.vhd	✓	VHDL	5
mcu_core.vhd	✓	VHDL	0
OP.vhd	✓	VHDL	6
Plus_minus_mux.vhd	✓	VHDL	7
tb_mcu_core.vhd	✓	VHDL	1
tb_pm_hex.vhd	✓	VHDL	2

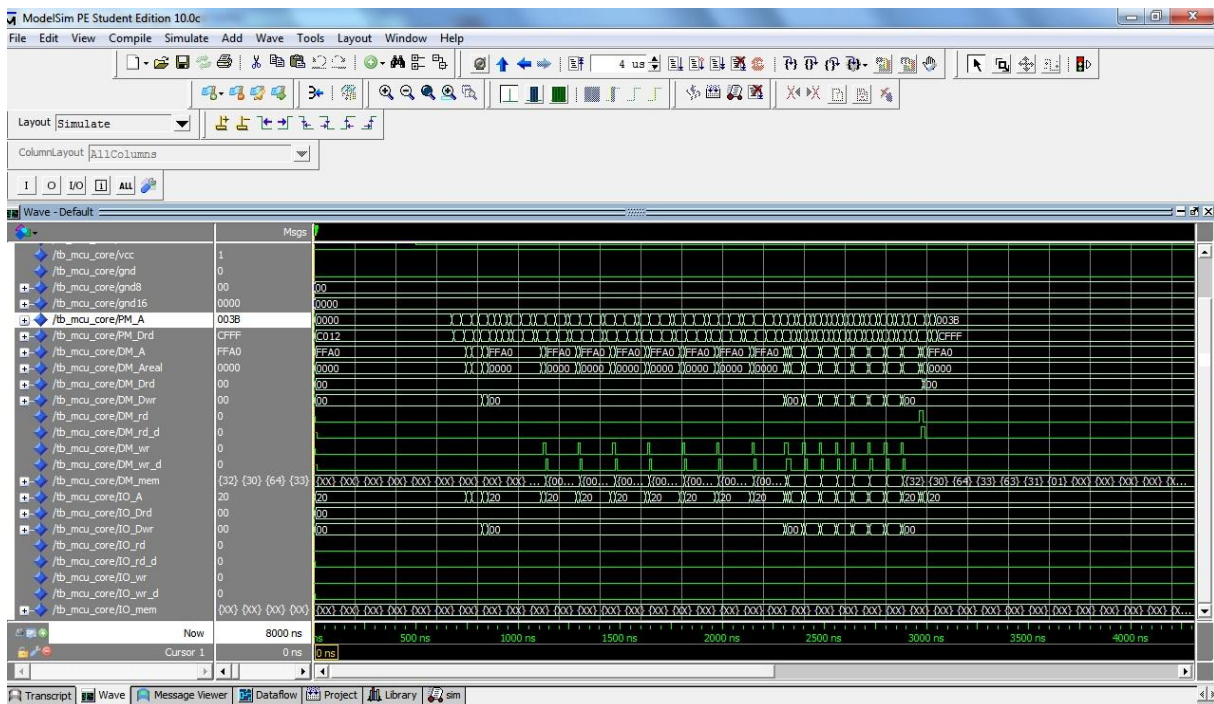
6-31: Az IBZ-AVR projekt



6-32: A szintézis eredménye



6-33: A lefordítandó C kód AVR Studio GCC Toolchain segítségével



6-34: A lefordult C kód szimulációja ModelSimben

6.6 Az RTL és ML-RTL leírás összehasonlítása

Flow Summary	
Flow Status	Successful - Fri Oct 28 09:55:23 2011
Quartus II Version	10.1 Build 197 01/19/2011 SP 1 SJ Web Edition
Revision Name	FPGA_TOP_ent
Top-level Entity Name	FPGA_TOP_ent
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	40 / 33,216 (< 1 %)
Total combinational functions	37 / 33,216 (< 1 %)
Dedicated logic registers	32 / 33,216 (< 1 %)
Total registers	32
Total pins	35 / 475 (7 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

6-35: ML-RTL ALU - A számított helyfoglalás

Flow Summary	
Flow Status	Successful - Fri Oct 28 10:12:07 2011
Quartus II Version	10.1 Build 197 01/19/2011 SP 1 SJ Web Edition
Revision Name	FPGA_TOP_ent
Top-level Entity Name	FPGA_TOP_ent
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	128 / 33,216 (< 1 %)
Total combinational functions	128 / 33,216 (< 1 %)
Dedicated logic registers	31 / 33,216 (< 1 %)
Total registers	31
Total pins	35 / 475 (7 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

6-36: Standard RTL ALU – A számított helyfoglalás

Resource	Usage
1 Estimated Total logic elements	60
2	
3 Total combinational functions	37
4 Logic element usage by number of LUT inputs	
5 -- 4 input functions	26
6 -- 3 input functions	9
7 -- <=2 input functions	2
8	
9 Logic elements by mode	
10 -- normal mode	37
11 -- arithmetic mode	0
12	
13 Total registers	32
14 -- Dedicated logic registers	32
15 -- I/O registers	0
16	
17 I/O pins	35
18 Maximum fan-out node	CLK
19 Maximum fan-out	32
20 Total fan-out	272
21 Average fan-out	2.62

6-37: ML-RTL ALU Erőforrás használata

Resource	Usage
1 Estimated Total logic elements	151
2	
3 Total combinational functions	128
4 Logic element usage by number of LUT inputs	
5 -- 4 input functions	80
6 -- 3 input functions	34
7 -- <=2 input functions	14
8	
9 Logic elements by mode	
10 -- normal mode	105
11 -- arithmetic mode	23
12	
13 Total registers	31
14 -- Dedicated logic registers	31
15 -- I/O registers	0
16	
17 I/O pins	35
18 Maximum fan-out node	reg_entreg_in[2]
19 Maximum fan-out	45
20 Total fan-out	581
21 Average fan-out	2.99

6-38: Standard RTL ALU Erőforrás használata

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	118.71 MHz	118.71 MHz	CLK	

6-39: Megengedett maximális frekvencia ML-RTL leírásra

Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	78.02 MHz	78.02 MHz	CLK	

6-40: Megengedett maximális frekvencia standard RTL leírásra

Setup Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	CARRY_EN	CLK	4.749	4.749	Rise	CLK
2	CARRY_IN	CLK	4.855	4.855	Rise	CLK
3	▲ CONTROL[*]	CLK	4.943	4.943	Rise	CLK
4	CONTROL[0]	CLK	4.943	4.943	Rise	CLK
5	CONTROL[1]	CLK	4.880	4.880	Rise	CLK
6	CONTROL[2]	CLK	4.612	4.612	Rise	CLK
7	CONTROL[3]	CLK	4.597	4.597	Rise	CLK
8	LOAD	CLK	5.277	5.277	Rise	CLK
9	PLUS_MIN	CLK	4.673	4.673	Rise	CLK
10	▲ x[*]	CLK	3.793	3.793	Rise	CLK
11	x[0]	CLK	3.605	3.605	Rise	CLK
12	x[1]	CLK	3.259	3.259	Rise	CLK
13	x[2]	CLK	3.793	3.793	Rise	CLK
14	x[3]	CLK	3.571	3.571	Rise	CLK
15	x[4]	CLK	3.569	3.569	Rise	CLK
16	x[5]	CLK	3.563	3.563	Rise	CLK
17	x[6]	CLK	3.522	3.522	Rise	CLK
18	x[7]	CLK	3.248	3.248	Rise	CLK
19	▲ y[*]	CLK	3.552	3.552	Rise	CLK
20	y[0]	CLK	3.545	3.545	Rise	CLK
21	y[1]	CLK	3.254	3.254	Rise	CLK
22	y[2]	CLK	3.535	3.535	Rise	CLK
23	y[3]	CLK	3.534	3.534	Rise	CLK
24	y[4]	CLK	3.533	3.533	Rise	CLK
25	y[5]	CLK	3.402	3.402	Rise	CLK
26	y[6]	CLK	3.257	3.257	Rise	CLK
27	y[7]	CLK	3.552	3.552	Rise	CLK
28	LOAD	CLK	5.281	5.281	Fall	CLK

6-41: Setup time ML-RTL szinten leírt ALU-ra

Setup Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	CARRY_EN	CLK	4.485	4.485	Rise	CLK
2	CARRY_IN	CLK	4.479	4.479	Rise	CLK
3	▲ CONTROL[*]	CLK	3.823	3.823	Rise	CLK
4	CONTROL[0]	CLK	3.823	3.823	Rise	CLK
5	CONTROL[1]	CLK	3.752	3.752	Rise	CLK
6	CONTROL[2]	CLK	3.805	3.805	Rise	CLK
7	CONTROL[3]	CLK	3.789	3.789	Rise	CLK
8	LOAD	CLK	4.359	4.359	Rise	CLK
9	PLUS_MIN	CLK	4.428	4.428	Rise	CLK
10	▲ x[*]	CLK	4.370	4.370	Rise	CLK
11	x[0]	CLK	4.370	4.370	Rise	CLK
12	x[1]	CLK	4.020	4.020	Rise	CLK
13	x[2]	CLK	4.142	4.142	Rise	CLK
14	x[3]	CLK	4.077	4.077	Rise	CLK
15	x[4]	CLK	4.364	4.364	Rise	CLK
16	x[5]	CLK	4.083	4.083	Rise	CLK
17	x[6]	CLK	4.111	4.111	Rise	CLK
18	x[7]	CLK	4.105	4.105	Rise	CLK
19	▲ y[*]	CLK	4.412	4.412	Rise	CLK
20	y[0]	CLK	4.143	4.143	Rise	CLK
21	y[1]	CLK	4.412	4.412	Rise	CLK
22	y[2]	CLK	4.112	4.112	Rise	CLK
23	y[3]	CLK	4.165	4.165	Rise	CLK
24	y[4]	CLK	4.371	4.371	Rise	CLK
25	y[5]	CLK	4.109	4.109	Rise	CLK
26	y[6]	CLK	4.114	4.114	Rise	CLK
27	y[7]	CLK	4.123	4.123	Rise	CLK
28	LOAD	CLK	4.336	4.336	Fall	CLK

6-42: Setup time standard RTL szinten leírt ALU-ra

Hold Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	CARRY_EN	CLK	-4.519	-4.519	Rise	CLK
2	CARRY_IN	CLK	-4.625	-4.625	Rise	CLK
3	▲ CONTROL[*]	CLK	-4.367	-4.367	Rise	CLK
4	CONTROL[0]	CLK	-4.713	-4.713	Rise	CLK
5	CONTROL[1]	CLK	-4.650	-4.650	Rise	CLK
6	CONTROL[2]	CLK	-4.382	-4.382	Rise	CLK
7	CONTROL[3]	CLK	-4.367	-4.367	Rise	CLK
8	LOAD	CLK	-4.990	-4.990	Rise	CLK
9	PLUS_MIN	CLK	-4.443	-4.443	Rise	CLK
10	▲ x[*]	CLK	-3.018	-3.018	Rise	CLK
11	x[0]	CLK	-3.375	-3.375	Rise	CLK
12	x[1]	CLK	-3.029	-3.029	Rise	CLK
13	x[2]	CLK	-3.563	-3.563	Rise	CLK
14	x[3]	CLK	-3.341	-3.341	Rise	CLK
15	x[4]	CLK	-3.339	-3.339	Rise	CLK
16	x[5]	CLK	-3.333	-3.333	Rise	CLK
17	x[6]	CLK	-3.292	-3.292	Rise	CLK
18	x[7]	CLK	-3.018	-3.018	Rise	CLK
19	▲ y[*]	CLK	-3.024	-3.024	Rise	CLK
20	y[0]	CLK	-3.315	-3.315	Rise	CLK
21	y[1]	CLK	-3.024	-3.024	Rise	CLK
22	y[2]	CLK	-3.305	-3.305	Rise	CLK
23	y[3]	CLK	-3.304	-3.304	Rise	CLK
24	y[4]	CLK	-3.303	-3.303	Rise	CLK
25	y[5]	CLK	-3.172	-3.172	Rise	CLK
26	y[6]	CLK	-3.027	-3.027	Rise	CLK
27	y[7]	CLK	-3.322	-3.322	Rise	CLK
28	LOAD	CLK	-4.987	-4.987	Fall	CLK

6-43: Hold time ML-RTL ALU-ra

Hold Times						
	Data Port	Clock Port	Rise	Fall	Clock Edge	Clock Reference
1	CARRY_EN	CLK	-4.255	-4.255	Rise	CLK
2	CARRY_IN	CLK	-4.249	-4.249	Rise	CLK
3	▲ CONTROL[*]	CLK	-3.522	-3.522	Rise	CLK
4	CONTROL[0]	CLK	-3.593	-3.593	Rise	CLK
5	CONTROL[1]	CLK	-3.522	-3.522	Rise	CLK
6	CONTROL[2]	CLK	-3.575	-3.575	Rise	CLK
7	CONTROL[3]	CLK	-3.559	-3.559	Rise	CLK
8	LOAD	CLK	-3.895	-3.895	Rise	CLK
9	PLUS_MIN	CLK	-4.198	-4.198	Rise	CLK
10	▲ x[*]	CLK	-3.790	-3.790	Rise	CLK
11	x[0]	CLK	-4.140	-4.140	Rise	CLK
12	x[1]	CLK	-3.790	-3.790	Rise	CLK
13	x[2]	CLK	-3.912	-3.912	Rise	CLK
14	x[3]	CLK	-3.847	-3.847	Rise	CLK
15	x[4]	CLK	-4.134	-4.134	Rise	CLK
16	x[5]	CLK	-3.853	-3.853	Rise	CLK
17	x[6]	CLK	-3.881	-3.881	Rise	CLK
18	x[7]	CLK	-3.875	-3.875	Rise	CLK
19	▲ y[*]	CLK	-3.879	-3.879	Rise	CLK
20	y[0]	CLK	-3.913	-3.913	Rise	CLK
21	y[1]	CLK	-4.182	-4.182	Rise	CLK
22	y[2]	CLK	-3.882	-3.882	Rise	CLK
23	y[3]	CLK	-3.935	-3.935	Rise	CLK
24	y[4]	CLK	-4.141	-4.141	Rise	CLK
25	y[5]	CLK	-3.879	-3.879	Rise	CLK
26	y[6]	CLK	-3.884	-3.884	Rise	CLK
27	y[7]	CLK	-3.893	-3.893	Rise	CLK
28	LOAD	CLK	-3.868	-3.868	Fall	CLK

6-44: Hold time standard RTL ALU-ra