



Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Dinamikus rekonfigurálás Zynq eszközön

MSc Önálló laboratórium 2. zárójegyzőkönyv
2015/16. I. félév

Virovecz Ádám Péter (EUM0Y2)

V. évf, villamosmérnök szakos hallgató
MSc Beágyazott információs rendszerek főspecializáció

Konzulens:

dr. Fehér Béla docens
(Méréstechnika és Információs Rendszerek Tanszék)

Tartalomjegyzék

1. Bevezetés	3
2. Parciális rekonfiguráció	3
2.1. A parciális rekonfiguráció típusai	3
2.2. A parciális rekonfiguráció előnyei	4
2.3. A parciális rekonfiguráció támogatása Xilinx eszközökben	4
2.3.1. Modul alapú parciális rekonfiguráció	4
2.3.2. A parciális rekonfiguráció folyamata	6
2.3.3. A partíciók kijelölésének szabályai	8
2.3.4. Konfigurációs portok Xilinx eszközökön	9
2.4. Felmerülő problémák, kötöttségek	10
2.5. Lehetséges alkalmazások	10
3. Képfeldolgozás Zynq-en	11
3.1. A Zynq SoC és ZedBoard fejlesztői kártya	11
3.2. A megtervezett hardver részletes bemutatása	12
3.3. A képfeldolgozó modulok tervezése	15
3.3.1. Éldetektálás	16
3.3.2. Morfológiai operátorok	17
3.3.3. Zajszűrés	18
3.3.4. Thresholding	18
3.3.5. Élesítés	18
3.3.6. FAST corner detection	19
3.4. Partíciók hardveres leválasztása	19
3.5. Eredmények	21
3.6. Szoftveres megvalósítás	25
4. Összefoglalás	28
4.1. Értékelés	28
4.2. Továbbfejlesztési lehetőségek	28
5. Irodalomjegyzék	29

1. Bevezetés

A rekonfigurálható eszközök fejlődésével a gyártók olyan technológiákat fejlesztettek ki, amelyek elősegítik a flexibilis használatot, több lehetőséget biztosítanak az eszköz erőforrásainak kihasználására. A dinamikus rekonfiguráció egy ilyen technológia, amely lehetővé teszi, hogy egy FPGA egyes területeit úgy konfiguráljuk át, hogy közben a maradék területen lévő logika folyamatosan működik. A két legnagyobb FPGA gyártó, Xilinx és Altera, egyaránt támogatják a parciális rekonfigurációt. A dolgozatban a Xilinx parciális rekonfigurációs technológiáját ismertetem részletesen, a Zynq SoC-okban alkalmazható megoldásokat külön kiemelve.

A parciális rekonfiguráció előnyeinek szemléltetéséhez egy képfeldolgozási projektet készítettem. Ezt a dolgozatban részletesen ismertetem. A motivációt a CellProfiler nevű program adta, ami egy orvosi és biológiai képek feldolgozására kifejlesztett szoftver. A felhasználónak lehetősége van képfeldolgozó lánc kialakítására a rendelkezésre álló sokféle modul felhasználásával. Ennek alapján az önálló laboratórium munkámban célom volt, hogy egy futásidőben újrakonfigurálható, néhány modulból álló képfeldolgozó pipelint alakítsak ki FPGA-n.

A megvalósításhoz Zynq SoC-t (ZedBoard fejlesztőpanel) választottam, ami tartalmaz két ARM Cortex-A9 processzort (perifériákkal), és jelentős méretű programozható logikai erőforrásokat.

2. Parciális rekonfiguráció

A parciális rekonfiguráció^{[1][2][3][4][5][6]} lehetővé teszi, hogy egy már felkonfigurált FPGA-t módosítsunk. Az FPGA-k felkonfigurálása általánosságban rögtön a bekapcsolás után történik, ekkor a teljes FPGA felületet beállítjuk. A parciális rekonfiguráció azt biztosítja, hogy az első teljes konfiguráció után, az FPGA egyes területeit megváltoztassuk, oda más logikát töltünk be. A konfiguráció minden esetben egy bitfájl letöltését jelenti, azonban a parciális rekonfiguráció esetén ez egy kisebb, úgynevezett parciális bitfájl.

Parciális rekonfiguráció esetén az FPGA területét több részre osztjuk: *statikus logikának* nevezzük azokat a részeket, amiket nem módosítunk, rekonfigurálható *partíciónak* pedig azt a területet, amit szeretnénk rekonfigurálni.

2.1. A parciális rekonfiguráció típusai

A parciális rekonfigurációt több csoportba oszthatjuk a rekonfiguráció végrehajtásának módja szerint. Eszerint a rekonfiguráció lehet

1. *statikus rekonfiguráció*, ha a parciális rekonfiguráció végrehajtása alatt az eszköz nem aktív (például órajelek kikapcsolása),
2. *dinamikus rekonfiguráció*¹, ha a konfiguráció az eszköz aktív állapotában történik, azaz a statikus logika zavartalanul működhet a konfiguráció folyamata alatt.

¹A dolgozatban ezentúl a parciális rekonfigurációt a dinamikus rekonfiguráció szinonimájaként használom, mivel ez a dolgozat fő témája.

2.2. A parciális rekonfiguráció előnyei

A parciális rekonfiguráció alkalmazása számos előnnyel rendelkezik, és sok esetben kibővíti a lehetőségeinket egy FPGA terv fejlesztésénél.

- *Kevesebb erőforrás használat:* Egy adott projekt implementálásához kisebb méretű FPGA is elegendő lehet. A parciális rekonfigurációval lehetőség van a hardver modulok közötti időmultiplexálásra, azaz mindig azt a hardvermodult töltjük be, ami az aktuális feldolgozási fázisban szükséges. A kisebb FPGA alacsonyabb költséget és fogyasztást jelent.
- *Flexibilitás:* Parciális rekonfigurációval elérhetjük, hogy a rendszerünk sokféle feladat hardveres gyorsítására legyen alkalmas: ha futás közben másféle feladat gyorsítására lenne szükségünk, akkor a rendszerünk ezt adaptívan tudja követni, csak a megfelelő modul betöltésére van szükség. Így többféle feladat lehetséges gyorsítására készíthetjük fel a rendszerünket, és futásidőben azt töltjük be, amelyekre ténylegesen szükségünk van. A rekonfigurálható modulok későbbi frissítése is megoldható akár anélkül, hogy a teljes rendszert le kellene állítani.
- *Biztonság:* A parciális rekonfigurációnál felmerülhetnek biztonsági kérdések, a bitfájlok illetéktelen módosítása, vagy letöltése bizonyos feladatoknál nagy problémát okozhat. Emiatt a Xilinx eszközök támogatják a parciális bitfájlok titkosítását.
- *FPGA-k hibatűrésének növelése:* Az FPGA-k gyártása során alkalmazott technológia rohamos fejlődése miatt a felhasználói erőforrások és ennek következtében a konfigurációs erőforrások száma is nagymértékben nő. A modern FPGA-k konfigurációs bitfájlai akár 1 Gbit méretűek is lehetnek.[7] Emiatt az úgynevezett Single Event Upset (SEU) jellegű hibák esélye is magasabb. Erre megoldás lehet kritikus környezetben alkalmazott FPGA-k esetén a konfiguráció egy részének visszaolvasása és alkalmanként frissítése.
- *Gyorsabb rendszer feléledés:* Az előző pontban említett nagyméretű bitfájlok miatt a konfigurációs idő meglehetősen nagy, akár több száz milliszekundum hosszú is lehet. Ezt néhány alkalmazás nem tolerálja (például PCIe buszra kapcsolódó FPGA). Emiatt ebben az esetben az FPGA egy kis részét konfiguráljuk be kezdetben (pl. a PCIe busz interfészt), és utána konfiguráljuk a maradék nagyméretű részeket.

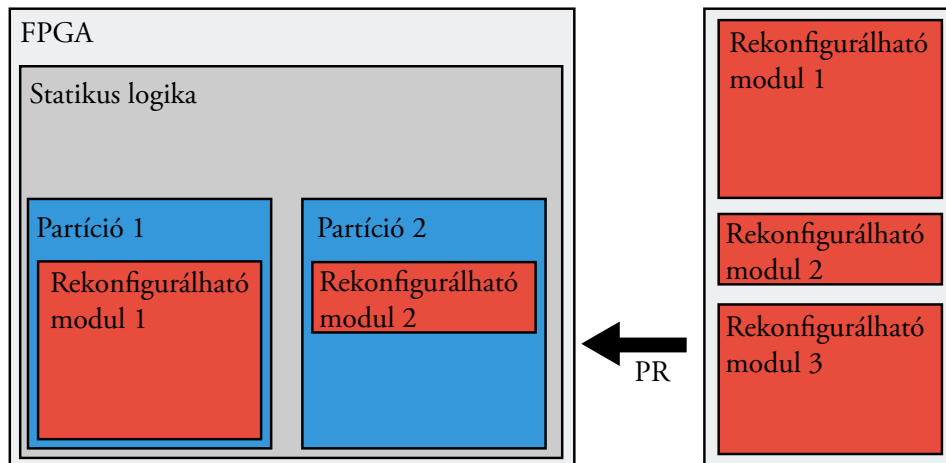
2.3. A parciális rekonfiguráció támogatása Xilinx eszközökben

A Xilinx eszközök a parciális rekonfigurációt mind hardveresen mind szoftveresen támogatják. Ebben a pontban a fejlesztés során használható eszközöket, lehetőségeket tekintem át.

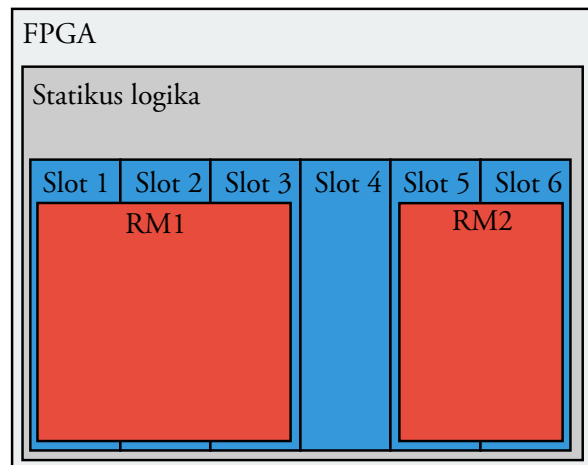
2.3.1. Modul alapú parciális rekonfiguráció

Az önálló laboratórium során csak a modul alapú rekonfigurációval foglalkoztam, a másik lehetőség a különbség alapú rekonfiguráció lenne. Ez azonban csak kisebb tervek esetén hasznos, ha nagyon hasonló modulok rekonfigurációját szeretnénk megoldani.

A modul alapú parciális rekonfiguráció [8] esetén a statikus és rekonfigurálható részek az FPGA felületen egyértelműen elválaszthatók (1. ábra). Az FPGA felületén kijelölhetünk olyan régiókat – partíciókat –, amelyek rekonfigurálhatók, és ezekbe a partíciókba tölthetjük be a modulokat. Azon modulokat, amelyeket ilyen partíciókba szeretnénk betölteni külön kell szintetizálni, és a partícióhoz való hozzárendelés után implementálni. Természetesen a partíciók méretének meghatározásánál a legnagyobb modul méretét kell figyelembe venni, és szükséges valamennyi plusz terület kijelölése, hogy a modul biztosan implementálható legyen.



1. ábra. Modul alapú parciális rekonfiguráció (szigetszerű elhelyezés).



2. ábra. Modul alapú parciális rekonfiguráció slot típusú elrendezéssel.

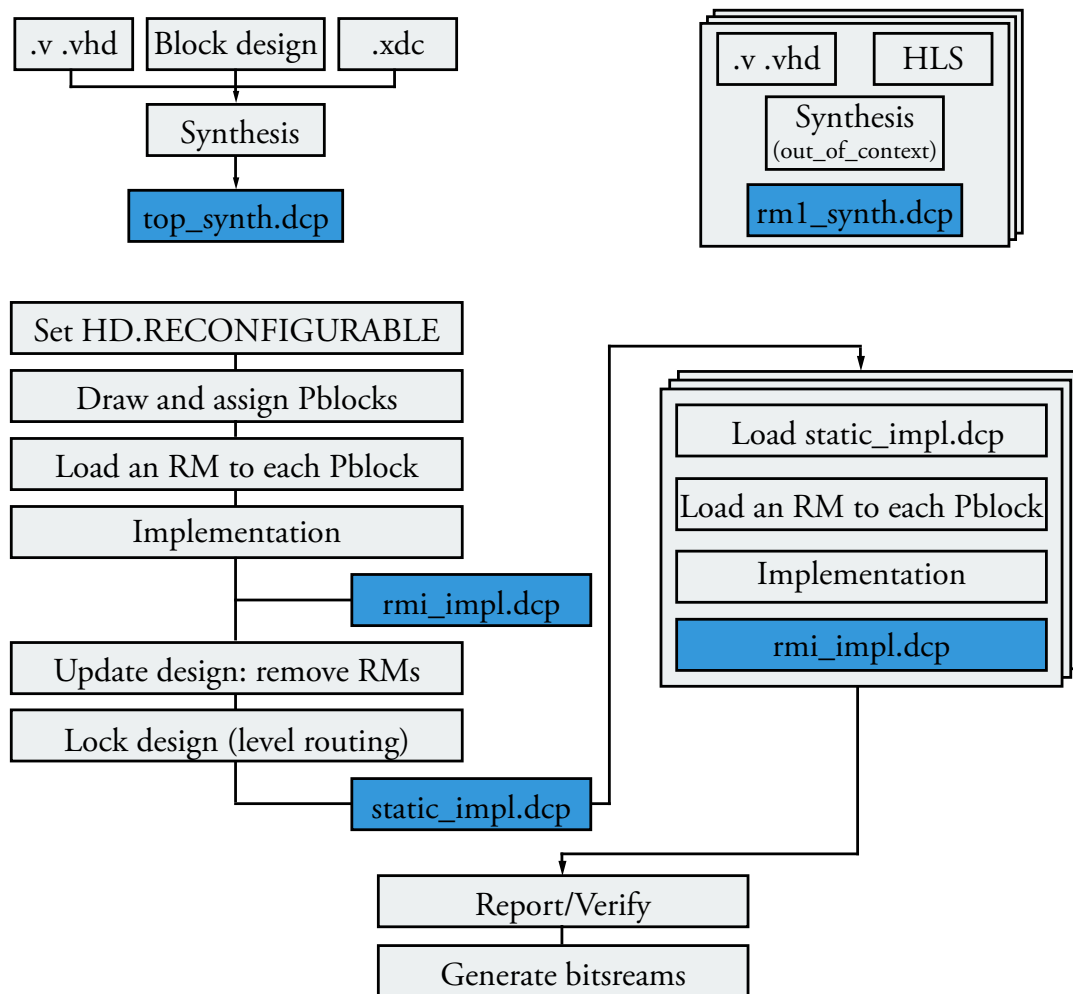
Felmerülhet annak a lehetősége, hogy néhány partíciónk van, és ezekbe szeretnénk többféle modult betölteni. Ha a hardver moduljaink erőforrás igénye jelentősen eltérő, akkor problémát jelenthet, hogy a partíciókat milyen méretűre válasszuk. Ez különösen igaz akkor, ha speciális erőforrásokat (BRAM, DSP modul) is szeretnénk használni, hiszen ezek kötött helyen oszlopokba rendezve vannak az FPGA felületen elhelyezve. Ha minden modult minden helyre be szeretnénk tölteni, akkor az előző bekezdésben említett szabály érvényes: minden partíció legyen a legnagyobb

modulnál valamennyivel nagyobb. Ez az úgynevezett szigetszerű elhelyezés, azaz egy modul mindig egy partícióba töltődik be. Jelenleg a fejlesztői eszközök ezt a lehetőséget támogatják. Elképzelhető lenne olyan megoldás, ahol egyforma méretű területeket kijelölünk az FPGA-n, és ezekből a modul méretének megfelelő számút foglalunk le (2. ábra). Ez a megoldás lényegesen növelné a flexibilitást, azonban rendkívül bonyolult lenne, és az optimális elhelyezés meghatározása nehéz feladat lenne.

2.3.2. A parciális rekonfiguráció folyamata

Ahhoz, hogy a tervünkben alkalmazhassunk parciális rekonfigurációt viszonylag bonyolult folyamatot kell követni. Szoftveresen mind a PlanAhead mind a Vivado Design Suite támogatja a parciális rekonfigurációt, azonban a Vivado használata esetén parancssoros a támogatás, projektszinten jelenleg nem támogatott. A következőkben a Vivado fejlesztői környezet parciális rekonfigurációs munkafolyamatát ismertetem, mivel a projektem elkészítésekor ezt használtam. A lépéseket a 3. ábrán is szemléltetem. A Vivado környezetben egy terv aktuális állapotát úgynevezett *checkpoint*-ba menthetjük, ami a betöltött terv összes adatát elmenti (lényegében egy fájlban tárolódnak a forrásfájlok, constraint-ek, generált fájlok stb.). A parciális rekonfigurációs folyamatban a checkpoint-ok mentése, betöltése kulcsszerepet játszik, hiszen így tudjuk a projekt különböző állapotait egyszerűen tárolni.

1. Az első lépés a statikus logika és a rekonfigurálható modulok elkészítése valamilyen hardvertervezési módszerrel (HDL, magas szintű szintézis). A statikus logikát egyszerűen szintetizálhatjuk, a rekonfigurálható modulokat egyesével *out of context* opcióval kell szintetizálni. Ez a paraméter meggátolja az I/O portok beillesztését a rekonfigurálható modulok esetében. Minden szintetizált modult és a szintetizált statikus logikát is egy checkpoint-ba mentünk.
2. A második lépésben kell definiálni a rekonfigurálható partíciókat és modulokat constraint-ek és paraméterek segítségével. Egy modult a HD.RECONFIGURABLE tulajdonság igazra állításával definiálhatunk rekonfigurálhatónak. Ez a paraméter a következő megkötéseket eredményezi a modulhoz rendelt partíción: megtiltja a modul határain lévő vezetékeken az optimalizálást, megtiltja statikus logika elhelyezését a partíción, és a modul teljes huzalozását a partíción belülre kényszeríti. Ekkor kell létrehozni a rekonfigurálható partíciókat is, azaz Pblock-okat. Ezek pontos elhelyezésére a következő pontban térek ki. A partíciókat és a modulokat össze kell rendelni.
3. A következő lépés az implementáció. Ehhez először a szintetizált statikus terv minden partíciójába be kell tölteni egy modult (ez általában a modulhoz tartozó checkpoint beolvasását jelenti a megfelelő cellába). Ezután a tervet implementáljuk a szokásos módon, és az implementált tervből készítünk egy checkpointot. Ezt követően a rekonfigurálható modulokat eltávolítjuk (azaz *black box*-ok lesznek), és a tervet rögzítjük huzalozás szintjén. Azaz ekkor a tervünk a statikus logikát és huzalozást tartalmazza (ebbe beleértve a statikus logika és rekonfigurálható partíciók közötti vezetékeket is). Az így kapott tervet is elmentjük



3. ábra. A parciális rekonfiguráció munkafolyamata Vivado-ban. (A kék kitöltésű téglalapok checkpoint-ok írását jelentik.)

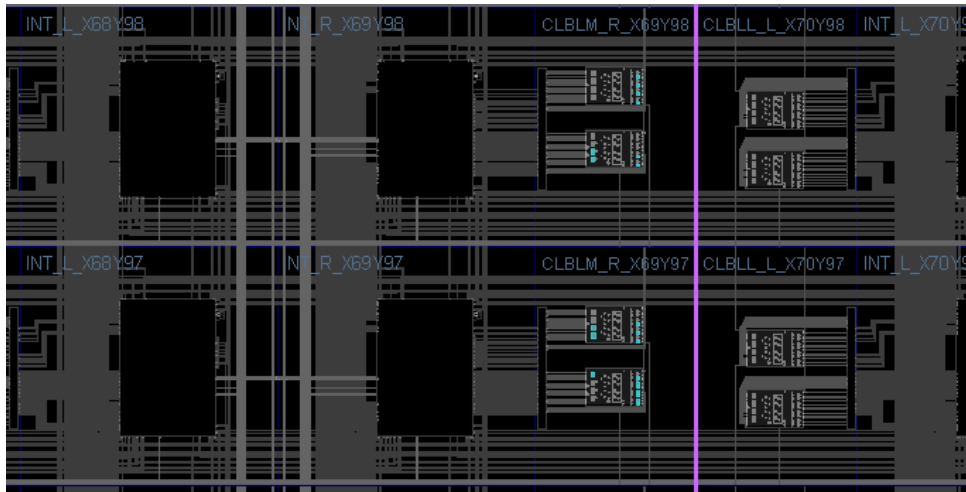
egy checkpoint-ba. A további implementációs lépésekben az előzőleg elmentett checkpoint-ba töltjük be a további modulokat, és futtatjuk le az előzőhöz hasonlóan az implementációt.

4. Következésként ajánlott lépés néhány jelentés generálása. Például megtekinthetjük a Pblock-ok foglaltságát, vagy az órajel hálózatok kihasználtságát. Fontos, hogy az implementáció után ellenőrizzük az időzítési jelentést, hiszen könnyen előfordulhat, hogy a megkötések nélkül implementált tervhez képest rosszabb időzítési tulajdonságokat kapunk. Ennek megfelelően szükséges lehet a partíciók áthelyezése, átméretezése, az időzítési constraint-ek vizsgálata.
5. A legfontosabb lépés az implementált verziók összehasonlító verifikálása. Ehhez a Vivado biztosít parancsot, amivel megadhatjuk az implementált tervek tartalmazó checkpoint-okat, és a verifikáló eszköz automatikusan összehasonlítja őket. Így ellenőrizhetjük, hogy az implementációs folyamatba nem csúszott-e hiba, például a statikus logika huzalozás szintjén ugyanolyan-e minden implementált változatban.
6. Utolsó lépés a bitfájlok generálásra. Ehhez az implementált változatokat tartalmazó

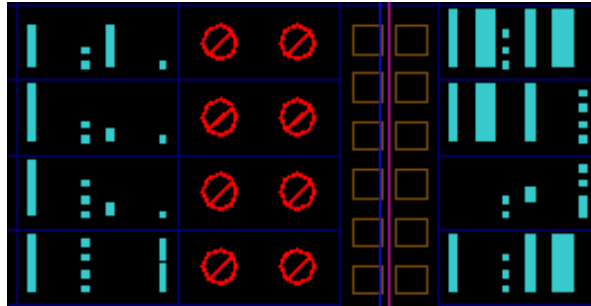
checkpoint-ok egyesével történő megnyitása és a bitfájl írási parancs kiadása szükséges. Ekkor kimenetként kapunk egy teljes bitfájlt, és a partícióknak megfelelő számú parciális bitfájlt.

2.3.3. A partíciók kijelölésének szabályai

A rekonfigurálható partíciók kijelölésekor számos szabálynak meg kell felelni. Mivel az önálló laboratórium munkám során Zynq eszközzel dolgoztam, itt a 7-es szériás FPGA-kra vonatkozó szabályokat ismertetem. Az UltraScale sorozat esetén bizonyos előírások megváltoztak. Az ismertetésnél nem törekszem a teljességre, csupán a legkritikusabb szabályokat emelem ki.[1]



(a) A 7-es szériás FPGA-k belső struktúrája. Látható, hogy a Pblock határa (függőleges lila vonal) két CLB oszlop között halad. Ha INT oszlop mellé raknánk a határt, akkor egy vagy kettő CLB oszlop le lenne tiltva.



(b) Órajel határt is magában foglaló Pblock hatása. Jól látható, hogy az órajel oszlop mellett egy CLB oszlop le lett tiltva. Így a huzalozó a statikus logikában fel tudja használni az órajel buffereket.

4. ábra. Néhány példa a Pblock elhelyezésnél alkalmazandó szabályokról.

A 7-es szériás FPGA-kban korlátozott a rekonfigurálható erőforrások halmaza. A következő FPGA erőforrások lehetnek rekonfigurálható modulokban: CLB, BRAM/FIFO, DSP48E1, PCI Express IP. Azaz például az órajel és I/O erőforrásoknak statikus logikában kell szerepelniük. Ha a partícióra szeretnénk beállítani a *reset after reconfig* tulajdonságot, akkor a Pblock-ot függőlegesen

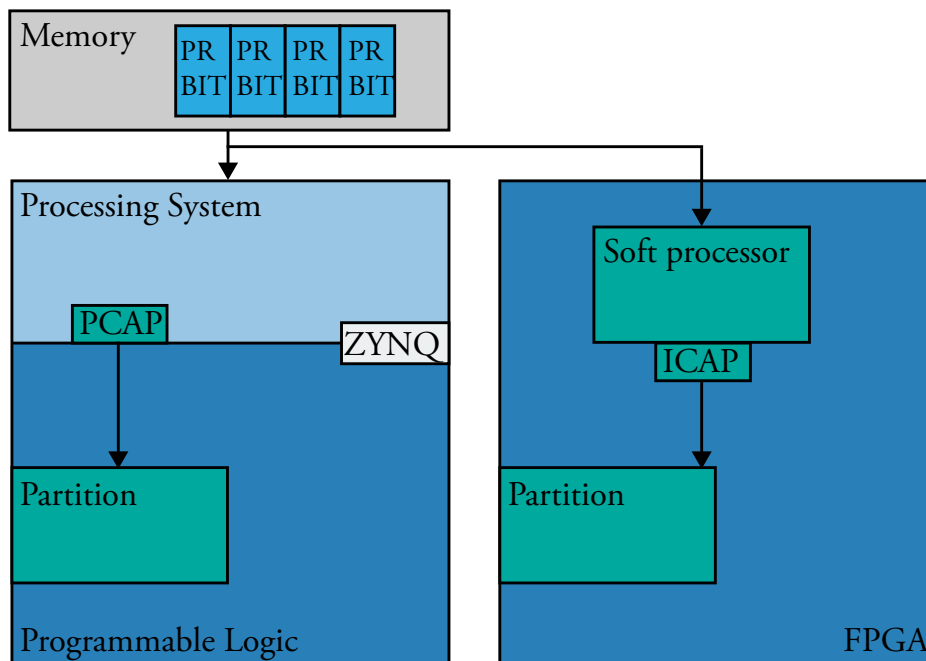
az órajel határokhoz kell igazítani. A Pblock bal és jobboldali határát két FPGA erőforrás oszlop közé kell helyezni, azaz CLB-CLB, CLB-BRAM vagy CLB-DSP határra, és nem interconnect elemek közé (INT-INT). Ha nem így járunk el, akkor bizonyos erőforrásoszlopok letiltódnak az adott határon. A Pblock-ok megfelelő igazítását a *snapping* paraméterrel automatizálhatjuk. Ha olyan területet is magába foglal a Pblock, ami nem rekonfigurálható (például órajel tartományok közötti bufferek, BSCAN, XADC stb.), akkor ezek az erőforrások a statikus logika számára nem lesznek elérhetők. Egy kivétel van, ez pedig az órajel bufferek, ugyanis ekkor az órajel oszlop melletti interconnect oszlopok és egy CLB oszlop is letiltódik a partíció számára, és a statikus logika számára használható marad. A Pblock-ok alakjára nincs megkötés, de az egyszerű huzalozás miatt erősen ajánlott a téglalap.

2.3.4. Konfigurációs portok Xilinx eszközökön

A parciális rekonfigurációt hardveresen korábban csak a felső kategóriás – Virtex sorozatú – Xilinx FPGA-k támogatták. A 7-es szériás FPGA-któl kezdve azonban minden eszköz támogatja ezt a technológiát, így az általam használt Zynq SoC-k is. A konfiguráció számos porton keresztül történhet. Ezeket röviden az alábbiakban mutatom be:

- *JTAG*: elsősorban fejlesztés során érdemes használni, támogatott a konfigurációs folyamat debuggolása is,
- *ICAP (Internal Configuration Access Port)*: belső konfigurációs port, ami részben az FPGA erőforrásokból kerül kialakításra. A konfigurációhoz egy processzoros rendszer vagy egy állapotgép szükséges, ami az ICAP portot meghajtja.
- *PCAP (Processor Configuration Access Port)*: speciálisan csak a Zynq eszközökben jelen lévő konfigurációs port. Működés szempontjából hasonló az ICAP-hez, de ez fix egységként a Zynq PS (Processing System) oldalán van beépítve, így nem igényel plusz FPGA erőforrást. Beépített DMA-t is tartalmaz, ami jelentősen gyorsítja a memóriában tárolt bitfájlok betöltését.
- *MCAP (Media Configuration Access Port)*: csak a legújabb UltraScale eszközökben megtalálható speciális konfigurációs port. A PCI Express alapú konfigurációt támogatja, az MCAP blokk az UltraScale FPGA-kban lévő hard PCIe IP-ben található.
- *Slave SelectMAP vagy Slave Serial*: elsősorban akkor ajánlott, ha az FPGA kezdeti konfigurációját és későbbi parciális rekonfigurációkat is ugyanarról a portról kívánjuk végrehajtani.

A konfigurációs sebesség tekintetében a leggyorsabbak a belső xCAP interfészek és a SelectMAP interfész, ezek maximum 100 MHz-es órajel mellett 3,2 Gbit-es sebességet kínálnak.



5. ábra. A PCAP és ICAP portok működésének szemléltetése.

2.4. Felmerülő problémák, kötöttségek

A parciális rekonfiguráció alkalmazása esetén bizonyos kötöttségekre figyelmet kell fordítani:

- *A rekonfigurálható erőforrások* az FPGA családtól függően eltérőek lehetnek. A 7-es szériás FPGA-k a CLB, BRAM, DSP és PCIe blokkok rekonfigurációját támogatják, az UltraScale eszközöknél ez jelentősen kibővült.
- *Kötött huzalozás a statikus logikához:* az egy partícióba helyezhető összes modul interfészének teljesen meg kell egyeznie.
- *Leválasztás:* a rekonfigurálás ideje alatt a statikus logikához kapcsolódó interfészeket megfelelően le kell választani. Ez különösen fontos lehet processzor buszra kapcsolódó modulok esetén. A leválasztás lehet hardveres, szoftveres vagy a kettő kombinációja.
- *Konfigurációs idő:* a parciális rekonfigurációhoz szükséges idő a partíció méretétől függ. A tervezés során figyelembe kell venni, az ehhez szükséges időt.

2.5. Lehetséges alkalmazások

A parciális rekonfigurációnak számos lehetséges alkalmazása van, a következőkben néhányat ismertetek röviden.

- *Hálózati eszközök:* előfordulhat, hogy az FPGA alapú hálózati eszköznek többféle interfész protokollt kell ismernie, és előre nem tudjuk melyik portján milyen protokoll beállítás lesz. A parciális rekonfigurációval dinamikusan betölthetjük az adott protokoll kezeléséhez szükséges modult.

- *SDR (Software Defined Radio)*: az előzőhöz teljesen hasonló módon, ebben az esetben például a modulációhoz/demodulációhoz szükséges megfelelő modult tudjuk betölteni, így hardveresen támogatva többféle modulációs eljárást.
- *Dinamikusan konfigurálható csomag-processzor*: a csomag-processzor a kapott csomagok típusának megfelelően átkonfigurálja egyes részeit. Akár az a megvalósítás is elképzelhető, hogy a csomagok hordozzák a feldolgozásukhoz szükséges hardver bitfájlját.
- *Kép- és videofeldolgozás*: lehetőség van konfigurálható képfeldolgozási pipeline kialakítására, ezt a dolgozatban később részletesen ismertetem. Továbbá a PR használatával például többféle videotömörítési eljárást támogató hardveres gyorsítót is tervezhetünk, ahol mindig az aktuális igénynek megfelelő hardveres tömörítőt töltjük be.

3. Képfeldolgozás Zynq-en

A parciális rekonfigurációban rejlő lehetőségek bemutatásához egy képfeldolgozási projektet készítettem. A megvalósítást Zynq[9][10][11] platformon végeztem. A projekt célja egy olyan képfeldolgozó pipeline kialakítása volt, ami futásidőben rekonfigurálható.

3.1. A Zynq SoC és ZedBoard fejlesztői kártya

A projekt felépítésének részletezése előtt a Zynq eszköz főbb tulajdonságait mutatom be – kiemelve a megalkotott rendszer szempontjából lényegeseket.

A Zynq-7000-es családban található eszközök egyben tartalmaznak (hard) processzormagokat és rekonfigurálható területet, ennek megfelelően két fő részre oszthatjuk őket: PS (Processing System) és PL (Programmable Logic).

A PS oldal legfontosabb egysége az APU (Application Processing Unit). Ebben található a két ARM Cortex-A9 processzormag (FPU és NEON motor SIMD feldolgozáshoz), magonként 32kB utasítás és adat gyorsítótár, timerek, OCM (On-Chip Memory, 256kB), interrupt rendszer, DMA, 512kB L2 cache. A PS oldal számos perifériát tartalmaz, amiket a kívánt lábakra a MIO (Multiplexed I/O) funkcióval tudunk kivezetni. A PS tartalmaz memória interfészeket is. A DDR memória vezérlőhöz DDR2/3 memóriákat csatlakoztathatunk, maximum 32 bit széles adatbuszon. Ezen kívül külön memóriavezérlővel támogatja az SRAM, NOR, NAND és QSPI típusú memóriákat az eszköz. Fontos megemlíteni a PS-ben található interconnect egységeket; ezek felelnek az egyes egységek gyors összekötéséért. Két fő interconnect modul található a PS-ben: a Central Interconnect az APU, a DDR memóriavezérlő, a perifériák és a PL között biztosít összeköttetést. A PL to Memory Interconnect nagysebességű kapcsolatot biztosít a DDR memóriavezérlő és a PL között.

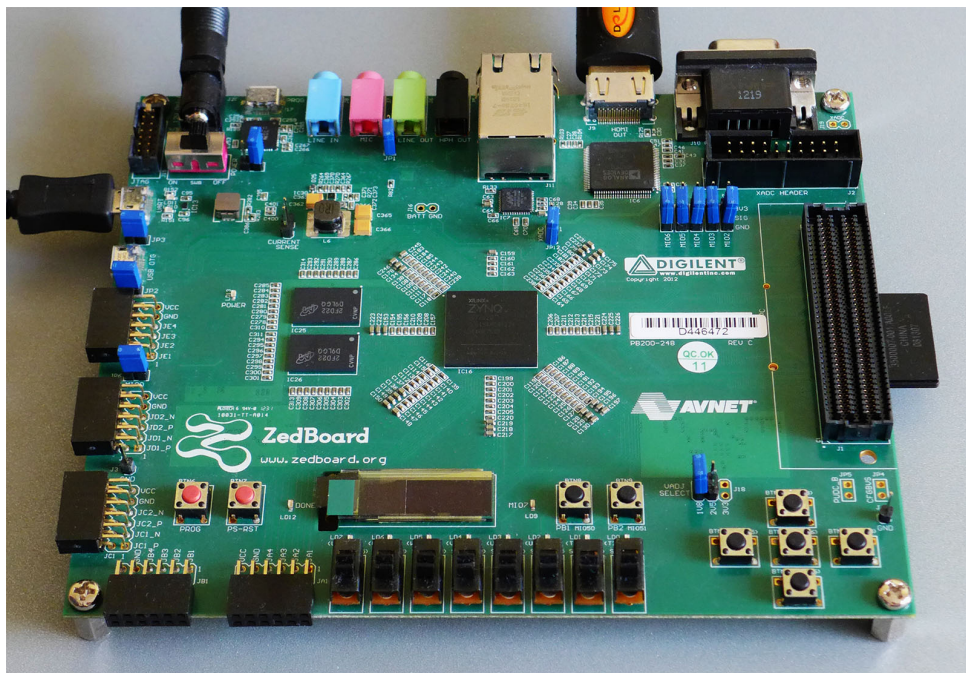
A PL oldal az eszköz típusától függően Artix 7 vagy Kintex 7 technológiájú FPGA-t tartalmaz. A szokásos FPGA erőforrások állnak rendelkezésre: CLB, BRAM, DSP, CMT (MMCM vagy PLL), konfigurálható IO, transceiver-ek (bizonyos modellekben), XADC.

Természetesen nagyon fontos a PL és PS oldal közötti gyors kommunikációs csatorna biztosítása. A két részegység között AXI átvezetések, EMIO interfész, megszakítás jelek, DMA jelek, órajelek

és debug jelek vannak kialakítva. A konfiguráció a PCAP porton keresztül történhet. Az AXI interfészekből három típus lett kialakítva, eltérő tulajdonságokkal. Az AXI_GP interfészek általános célú, 32 bites portok (2 slave és 2 master). A 4 darab AXI_HP interfész 32 vagy 64 bites átvitelt tesznek lehetővé. Ezek a portok nagysebességű kapcsolatot biztosítanak, szorosan kapcsolódnak a memóriavezérlőhöz és nagyméretű burst-öket tesznek lehetővé. A harmadik típus az AXI_ACP port, ami egy 64 bites cache-koherens port, ami az SCU-hoz (Snoop Control Unit) kapcsolódik, így alacsony késleltetésű átvitelt tesz lehetővé az APU és a PL között.

A ZedBoard egy Zynq XC7Z020-as típusú SoC-t tartalmazó fejlesztői panel. A kártyán többek között van 512 MB DDR3 memória, 256 Mb QSPI Flash, USB-JTAG, USB-UART interfész, Ethernet, HDMI kimenet, gombok, kapcsolók, LED-ek, kijelző, audió be- és kimenetek és SD kártya foglalat. A jó felszereltsége miatt sokféle projektben alkalmazható fejlesztői platformként. Az XC7Z020 PL része egy közepes méretű FPGA-t tartalmaz. Főbb paraméterei:

- 53200 LUT és 106400 FF,
- 140 db. 36 Kb-es BRAM,
- 220 db. DSP modul,
- 200 I/O interfész.



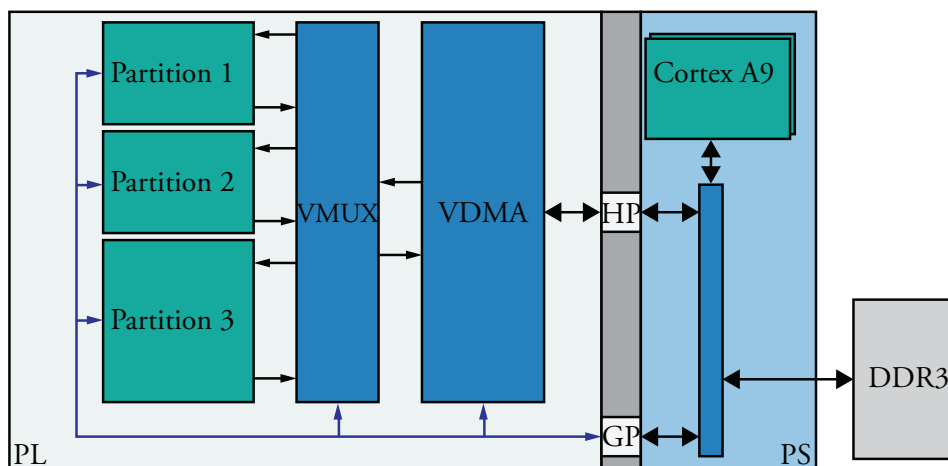
6. ábra. A ZedBoard fejlesztői panel.

3.2. A megtervezett hardver részletes bemutatása

A FPGA-n implementált hardvert a Vivado Design Suite felhasználásával fejlesztettem. Ez a fejlesztői rendszer lehetővé teszi, hogy a tervünket vizuálisan, blokkvázlat szerűen készítsük el IP

magok felhasználásával. Ezek lehetnek gyárilag hozzáférhető magok, vagy készíthetünk saját IP-t is.

A kitűzött cél az volt, hogy néhány rekonfigurálható képfeldolgozó blokkot egymás után kapcsolva legyen kialakítva egy képfeldolgozó pipeline. A nagyobb rugalmasság érdekében a tervben egy videó stream multiplexert is elhelyeztem, amivel (bizonyos korlátozásokkal) változtatható a feldolgozó blokkok bekötése. Ennek a rendszernek a vázlatos felépítése a 7. ábrán látható.



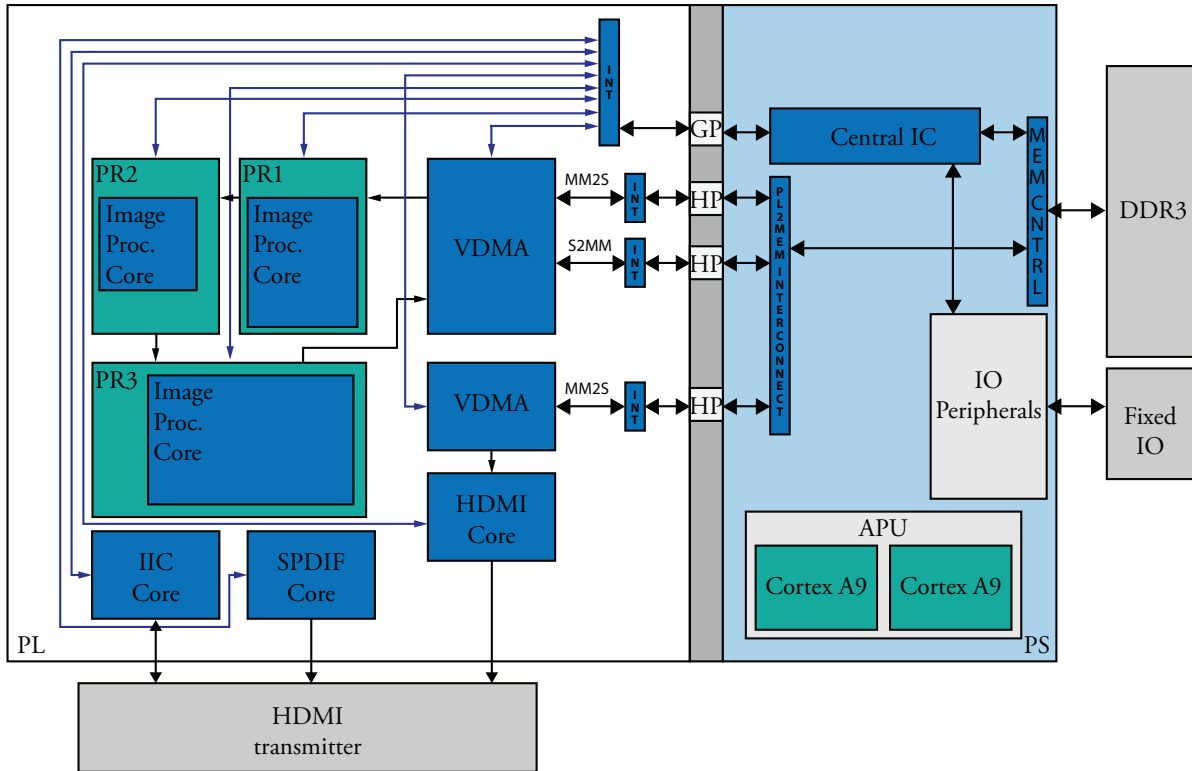
7. ábra. Képfeldolgozó pipeline terv.

A jelenleg megvalósított tervben a VMUX még nem szerepel, mivel beillesztésével időzíti problémák léptek fel. Emiatt a három blokkból felépülő sorba kapcsolt modulokból álló képfeldolgozó pipeline-t mutatom be részletesen. Ennek részletes blokkvázlata az 8. ábrán látható.

A blokkvázlat részletes ismertetését a kimeneti megjelenítő egységgel kezdem. Ahhoz, hogy a képfeldolgozási művelet eredménye egyszerűen látható legyen, a ZedBoard-on elérhető HDMI kimenetet használtam fel. Így valósidejű videokimenet megjelenítésére van lehetőség. A ZedBoard-on található egy HDMI transzmitter IC (ADV7511), ennek következtében az FPGA-val nem közvetlenül HDMI jeleket kell generálni, hanem párhuzamos videojelet. A HDMI transzmitter meghajtását az ábrán HDMI Core, SPDIF Core és IIC Core nevű IP-k végzik.

A HDMI Core az Analog Devices által ingyenes hozzáférhetővé tett IP mag. Ez generálja a videojelet: a fejlesztői kártyán 16 video adatvezeték található, így 4:2:2 módon mintavételezett YCbCr formátumú képeket kell létrehozni. A képfeldolgozó modulok 24 bites RGB színformátumot használnak, így ennek a magnak át kell alakítani ezt a jelet 16 bites YCbCr jellé. Ezen kívül a videó időzítéséhez szükséges jeleket is ez a mag állítja elő. A HDMI Core egy AXI4-Stream bemeneten keresztül kapja a videó adatfolyamot. Továbbá rendelkezik egy konfigurációt biztosító AXI4-Lite porttal. A buszójátelek mellett az aktuális képfelbontásnak és frissítési frekvenciának megfelelő pixel órajelet is biztosítani kell a magnak (ez a projektben alkalmazott maximális 1920x1080 pixeles felbontásnál 148,5 MHz). Ezt az órajelet egy programozható MMCM blokk állítja elő, így futás közben is lehetőség van felbontás váltásra, csupán az órajelet generáló blokkot kell megfelelően átprogramozni.

Az SPDIF Core egy Analog Devices IP, ami lehetőséget biztosít HDMI-n keresztül hang



8. ábra. A fejlesztett projekt részletes blokkvázlata.

továbbítására. Mint a blokkvázlaton látható, ez nem rendelkezik külön stream jellegű bemenettel, hanem a Zynq GP portjáról kapja a hangot is. Az ábrán nem tüntettem fel, de a Zynq egyik DMA request vonalára is csatlakozik ez az IP, így a hangátvitel a PS-ben található beépített DMA-val is megvalósítható. Ehhez az IP-hez elegendő a Zynq GP portjának használata, hiszen a hangátvitel lényegesen kevesebb sávszélességet igényel, mint egy videóstream.

Az IIC Core egy Xilinx I²C mag, ami a HDMI transzmitter felprogramozásához szükséges I²C interfészt biztosítja.

A képfeldolgozó magok egy-egy rekonfigurálható tartományban találhatók. A videó adatfolyam átvitele AXI4-Stream buszon történik. Minden képfeldolgozó mag rendelkezik egy AXI4-Lite konfigurációs porttal, ahol egyszerű inicializáló beállításokat lehet elvégezni (pl. képméret, szűrő paraméterek állítása).

A videojelek memóriából történő kiolvasásáért és memóriába írásáért a VDMA modulok felelnek. Ezek olyan speciális DMA-k, amik kifejezetten képi jelek átvitelére lettek kifejlesztve. Átvitelt biztosítanak a memória alapú busz (AXI4) és az adatfolyam jellegű busz (AXI4-Stream) között. A PS-hez a HP (High Performance) porton keresztül kapcsolódnak, így nagy sebességű adatátvitel lehetséges a memória és a DMA között (kikerüli a Central Interconnect-et, ami némileg lassabb átvitelt tesz lehetővé). A fejlesztés során egy érdekes hibába ütköztem a VDMA-k használata közben: ha a képfeldolgozó modult kiszolgáló VDMA két csatornáját (MM2S és S2MM) a PS ugyanazon HP portjára kötöttem (természetesen egy interconnect-en keresztül), akkor rendellenesen működtek, "lefagytak" a DMA csatornái, csak a teljes reset segített. Ha a

két csatornát külön HP portra kötöttem, akkor viszont normálisan működtek. A hiba okára még nem sikerült rájönnöm, de az első esetben is megfelelően kellene működniük, hiszen elegendő sáv szélesség áll rendelkezésre egy HP porton keresztül is. (Egy Xilinx-féle mintaprojekt is az első megoldást használta.) A HDMI-t kiszolgáló VDMA csupán egy csatornát tartalmaz, hiszen a megjelenítéshez csak memória olvasási művelet szükséges.

A videojelek szinkronizációját a HDMI Core által generált vertikális szinkronjellel biztosítottam. A két MM2S VDMA csatornát erre a jelre szinkronizáltam. A memóriában kétszer három framebuffert alakítottam ki. Az első három framebuffer a bemenő képeket tartalmazza, ezt olvassa a képfeldolgozó modult meghajtó VDMA. A feldolgozott képet a felső VDMA kiírja a második három framebufferbe, és ezt olvassa a HDMI kimeneti VDMA. Ez utóbbi esetben biztosítani kellett, hogy azonos képet egyszerre ne írjon és olvasson egy DMA, mivel úgy képtörés lehet. Ezt a VDMA-kba épített GenLock funkció segíti, ami lehetővé teszi, hogy két VDMA csatorna egy külön interfészen keresztül elküldje egymásnak, hogy éppen hányadik framebufferen dolgozik, így a másik azt kihagyja, vagy várakozik.

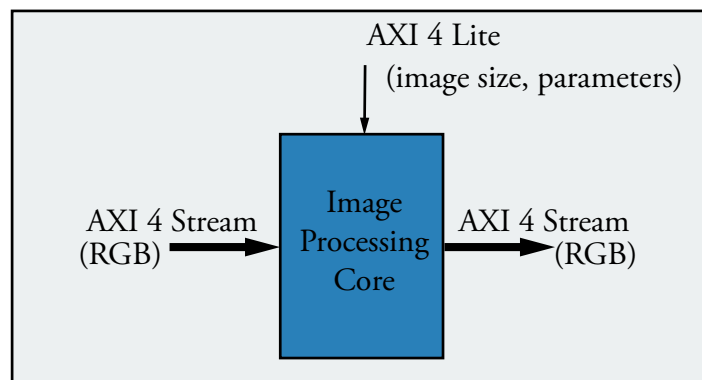
A megvalósított rendszer több órajelet használ: az AXI4-Lite busz 100 MHz-es órajelet használ, a videó adatfolyamot szállító AXI4-Stream busz pedig 150 MHz-es órajelről jár. A 150 MHz-es órajelre a megfelelően nagy felbontású video tartalmak átvitele miatt van szükség. A 100 MHz-es órajelet a PS szolgáltatja és egy MMCM modul ebből állítja elő a 150 MHz-es órajelet. A HDMI modulnak szüksége van speciális pixel órajelekre is, amit egy külön MMCM modul állít elő. Ez a modul – ahogy már korábban említettem – programozható, a programozást AXI4-Lite interfészen keresztül lehetséges. A hangátvitelhez szükséges 12,288 MHz-es órajelet is egy MMCM blokk állítja elő.

3.3. A képfeldolgozó modulok tervezése

A képfeldolgozó modulok tervezése során célom volt, hogy viszonylag gyorsan, többféle modult tudjak fejleszteni. Emiatt nem HDL nyelven végeztem a fejlesztést, hanem magas szintű szintézist (Vivado HLS[12][13]) használva.

Jelfeldolgozási, adatfolyamokkal dolgozó hardver modulok fejlesztésére gyors megoldást kínál a Vivado HLS technológia. Ekkor a hardver működését C, C++ vagy SystemC nyelven írhatjuk le. Természetesen a C vagy C++ nyelv önmagában nem alkalmas hardver leírására, hiszen programozási nyelvek, így a szintetizálendő hardver működését pragmak segítségével befolyásolhatjuk. Így például megadhatunk interfészeket, az egyes műveletek hardveres leképezését, ciklusok pipeline-ba szervezését vagy kifejtését (unroll).

Minden képfeldolgozó modult teljesen azonos interfésszel terveztem, hogy a dinamikus rekonfigurálásuk megvalósítható legyen: egy AXI4-Lite interfész a paraméterek beállítására, egy-egy AXI4-Stream interfész a bemenő és kimenő videó adatfolyamnak. Ez a 9. ábrán látható.



9. ábra. Egy képfeldolgozó modul interfészei.

A képfeldolgozó modulok típusaihoz az ötletet a CellProfiler-ben is jelen lévő komponensek adták. Így a cél a következő típusú modulok megvalósítása volt:

- éldetektálás,
- morfológiai operátorok,
- zajsűrés,
- élesítés,
- thresholding,
- corner detection.

Az elkészült szűrőket a következőkben részletesen is bemutatom.

3.3.1. Éldetektálás

Sokféle éldetektálási algoritmus létezik, amelyek eltérő tulajdonságokkal rendelkeznek. A legismertebbek a Sobel, Roberts, Prewitt, Canny, Laplace of Gaussian operátorok.[14] Az első három algoritmus viszonylag egyszerű, csupán két 3x3 kernellel kell a képet szűrni (a függőleges és vízszintes irányú élek detektálása érdekében), majd az eredmény a két szűrés négyzetösszege. Ezen operátorok előnye az egyszerűségük, hátrányuk, hogy eléggé érzékenyek a zajra, valamint nem pontosak. A Laplace of Gaussian operátor a nevéből láthatóan Laplace operátor alkalmazását jelenti egy Gauss-féle szűrővel szűrt képre. A Gauss szűrés lényeges, mivel a Laplace szűrő nagyon érzékeny a zajra a kétszeres derivált miatt. Ez a szűrő pontosan detektálja az élek helyét, de nagyobb kernel szükséges jellemzően (az optimális kernelméret függ a Gauss szűrésnél alkalmazott σ szórástól), viszont a sarkoknál hibázhat. A Canny szűrő a legbonyolultabb, de a legpontosabb is, különösen zajos környezetben. Egyetlen hátránya, hogy elég erőforrás-igényes.

A fentiek közül eddig a Prewitt, Roberts és Sobel szűrőket valósítottam meg. Kihívást jelentett az optimális hardver generálása a kódból, azaz a megfelelő pragmak használata. További problémát jelentett az időzítési paraméterek teljesítése, a generált hardver kezdetben nem teljesítette a

150 MHz-es frekvenciához szükséges időzítési megkötéseket. Ezt némi átalakítással, és megfelelő pragmak használatával sikerült elérni, de kiemelném, hogy az alkalmazott FPGA határai nagyjából ennél a 150 MHz-es frekvenciánál vannak, ilyen bonyolultságú IP-k használatánál. (Például a Xilinx VDMA felhasználói útmutatójából kiderül, hogy a maximális AXI4-Stream órajel 150 MHz lehet Artix 7 sorozatú -1 sebességsztályú FPGA esetén[15].)

3.3.2. Morfológiai operátorok

A morfológiai operátorok a képek képtérben történő geometriai manipulációját teszik lehetővé. Kezdetben bináris képekre voltak értelmezve, később kiterjesztették szürkeárnyaltos képekre is az operátorokat. A leggyakoribb operátorok a dilatació, erózió, nyitás és zárás. A morfológiai műveletekkel a matematika egy külön területe foglalkozik, az operátorok részletes matematikai tulajdonságait itt nem ismertetem[16].

A fenti operátorok közül az eróziót, dilataciót és ezek kombinációjából képzett morfológiai gradienst és top-hat transzformációt valósítottam meg. A morfológiai operátoroknál meg kell határozni, hogy mekkora és milyen alakú strukturáló elemet használunk. A fenti operátorok megvalósításánál 3×3 méretű négyzetes strukturáló elemet használtam. Ezen kívül készítettem egy olyan erodáló/dilatáló modult is, ami képes maximum 5×5 méretű és többféle alakú (négyzet, gyémánt, különböző irányú vonal) strukturáló elemmel dolgozni. Ebben az esetben futás közben lehet megadni, hogy erodálni vagy dilatálni szeretnénk, és milyen típusú strukturáló elemet használjunk.

A dilatació operátor minden pixelnél a strukturáló elem által kijelölt pixelek maximuma:

$$[f \oplus b](x, y) = \max_{(s, t) \in b} \{f(x - s, y - t)\},$$

ahol f a kép és b a strukturáló elem. Az erózió operátor egy kép tetszőleges pixelére a strukturáló elem által kijelölt pixelek minimuma:

$$[f \ominus b](x, y) = \min_{(s, t) \in b} \{f(x - s, y - t)\}.$$

Dilatáló és erodáló blokkok kombinációjával elkészíthetjük a képek nyitását (opening):

$$f \circ b = (f \ominus b) \oplus b,$$

és zárását (closing):

$$f \bullet b = (f \oplus b) \ominus b.$$

Egy f kép g morfológiai gradiense

$$g = (f \oplus b) - (f \ominus b)$$

alakban írható fel, ahol b a strukturáló elem. A top-hat transzformációt a

$$T_{hat}(f) = f - (f \circ b)$$

formulával írhatjuk le.

Az egyszerű 3×3 méretű strukturáló elemmel dolgozó operátorokat OpenCV függvényekkel valósítottam meg, a bonyolultabb maximum 5×5 méretű erodáló/dilatáló modult pedig teljesen én kódoltam. Az OpenCV függvényeknél is megadható a strukturáló elem, viszont a hardveres kötöttségek miatt csak sablon paraméterként. Emiatt változtatható strukturáló elemmel működő szűrő esetén nem használhatók.

3.3.3. Zajsztűrés

A képek zajsztűrése általában bonyolult feladat, mert nagyon sokféle zaj előfordulhat, és ennek megfelelően eltérő zajsztűrő algoritmusok szükségesek. Lehetséges zajsztűrő megoldások lehetnek az egyszerű átlagolástól, mediánsztűrőtől kezdve a bonyolult transzformált tartományban (pl. wavelet) működő szűrők.

A félév során egy mediánsztűrőt készítettem el HLS-ben, ami elsősorban az impulzuszajok szűrésére alkalmas. Nehézséget az jelentett, hogy a mediánsztűrő működése szürkeárnyaltos képek esetén definiált. Színes képek esetén többféle megvalósítás képzelhető el. Egyik lehetőség a pixelek világosságtartalmának mediánjának felhasználása, ami viszonylag egyszerű megvalósítást tesz lehetővé, és elfogadható eredményt ad. A színcsatornánként alkalmazott medián nem lenne jó megoldás mivel jelentős színzajt eredményezne a szűrt képen.

3.3.4. Thresholding

A thresholding során a képen használt színek számát redukáljuk, akár egészen két színre. Egy bináris thresholdingot végző modult implementáltam, aminek a felhasználó tudja beállítani, hogy mekkora legyen a küszöb. Elképzelhető lenne olyan fejlesztés, hogy a modul "kitalálja" a megfelelő küszöb értéket (például Otsu-thresholding). Ebben az esetben a teljes képet analizálni kell először és utána alkalmazni a thresholding-ot, ami a stream jellegű megvalósítás miatt nehezen kivitelezhető. Két megoldás képzelhető el: többször kerül kiolvasásra ugyanaz a kép a memóriából, így teljesen szabályosan megoldható a thresholding, vagy mindig az előző képkocka alapján alkalmazunk thresholding-ot az aktuális képkockára. Természetesen ez utóbbi megoldás lényegesen kevesebb erőforrást igényel, de videó esetén a pontossága nem lesz tökéletes.

3.3.5. Élesítés

A kép élesítéshez az Unsharp Mask metódust használtam, 3×3 kernel mérettel. Ez a szűrés úgy működik, hogy az eredeti képből kivonja a Gauss-sztűrével szűrt képet, és az így kapott kép élesebb lesz. A neve onnan származik, hogy az Gauss-sztűrt, "unsharp" képet használjuk fel a kép élesítéséhez. Ez a szűrő a legtöbb fotószerkesztő programban elérhető, én az Adobe Photoshop-ban található szűrőhöz nagyon hasonló unsharp szűrőt implementáltam (természetesen a pontos

forrás nem áll rendelkezésre). A szűréshez használt kernel kiszámítása:

$$U = I + 2\frac{\beta}{100}(I - G),$$

ahol $I = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$, G egy 3×3 Gauss-kernel $\sigma = 0.6$ paraméterrel, β pedig az élesítés mértéke. Az élesítés mértéke futás közben szabadon módosítható, a modul újraszámolja minden képkocka után a kernelt.

3.3.6. FAST corner detection

A FAST (Feautres from Accelerated Segment Test) egy sarokpont érzékelő (corner detection) algoritmus, amivel a képet jellemző pontokat nyerhetjük ki (feature points). Az így kapott pontokkal például objektumokat követhetünk. Ez a módszer gyorsabb, mint sok más sarokpont detektáló algoritmus, erre utal a nevének rövidítése is. A FAST algoritmust röviden az alábbiakban foglalom össze.[17] Az algoritmus szürkeárnyaltos képen dolgozik, kimenete a sarokpontok helye.

1. Válasszunk ki egy p pixelt, aminek az intenzitása I_p . Legyen a threshold értéke t !
2. Vizsgáljunk meg a pixel körül egy olyan körvonalat, ami 16 pixelből áll (ez egy 3 sugarú kör).
3. A p pixel sarokpont, ha a körvonalon van n hosszú folytonos vonal, ami világosabb, mint $I_p + t$ vagy sötétebb, mint $I_p - t$. Például egy lehetséges n érték a 12 ($n \leq 16$).

Lehetséges az algoritmust tovább gyorsítani, illetve gépi tanulás eszközével pontosítani a működését.

HLS-ben az algoritmust OpenCV segítségével implementáltam. A *FASTX* függvény azonban csak a sarokpontok maszkját adja meg (vagy a koordinátákat), a megfelelő megjelenítés érdekében ezt még rá kellett rajzolni az eredeti képre. Ahhoz, hogy jól látható legyen a pontok helye, először dilatációt alkalmaztam, majd utána adtam hozzá a maszkot az eredeti képhez. Az implementációnál fontos volt figyelembe venni, hogy a FAST algoritmus és a dilatáció lefutásáig az eredeti kép néhány sorát el kell tárolni. Ez összesen 10 sort jelentett, emiatt ennek a modulnak különösen nagy volt a Block RAM igénye.

3.4. Partíciók hardveres leválasztása

A partíciók leválasztásának szükségességét már a parciális rekonfigurációt részletező pontban említettem. Most részletesen is kitérnék rá, ugyanis a terv fontos része volt a leválasztás megfelelő megvalósítása.

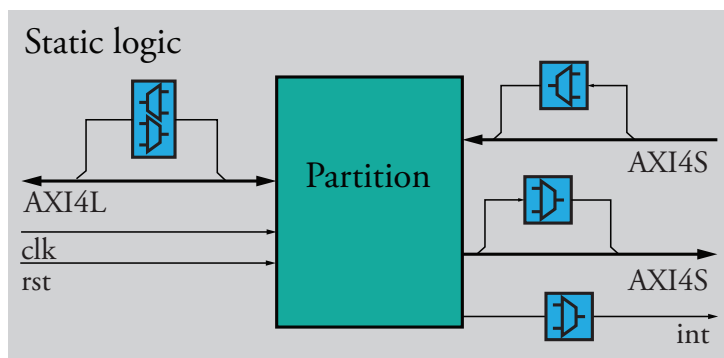
A rekonfigurálható modulok és a statikus logika közötti vezetékek leválasztása fontos, ugyanis a rekonfiguráció folyamata alatt a partíció logikai erőforrásai nem definiált állapotba kerülnek, így a vezetékeken tetszőleges jel megjelenhet. Ez különösen egy processzoros rendszernél lehet veszélyes, hiszen a megjelenő jelek, akár a teljes buszrendszert hibás állapotba hozhatják. A hardveres leválasztás egyszerűen megoldható egy multiplexer segítségével, ami leválasztott állapotban

valamilyen előre meghatározott értéken tartja a vezetéket. A leválasztást végző multiplexerek egy közös vezérlőjelről kapcsolhatók.

A korábban ismertetett képfeldolgozó modulok két AXI4-Stream, egy AXI4-Lite busszal, órajel és reset bemenettel, valamint interrupt kimenettel rendelkeznek. A buszok esetében a teljes leválasztás akár több mint 100 vezeték leválasztását jelentené, viszont szerencsére a buszvezérlő jelek leválasztása is elegendő. Ez az AXI4-Lite esetén 10 jelet jelent, az AXI4-Stream-nél pedig buszonként kettőt. Azaz viszonylag kevés erőforrással megvalósítható a leválasztás. A buszokon kívül fontos az interrupt jel leválasztása is.

A leválasztás egyszerűsítése érdekében a Vivado 2015.3 verziójától kezdve tartalmaz egy Partial Reconfiguration Decoupler IP-t. Ez kényelmessé teszi a gyakran használt busztípusok leválasztását, ugyanis a megfelelő buszt definiálva az IP beállításáiban, csak a szükséges jeleket választja le. Vezérlése történhet egy vezérlőjellel, vagy akár AXI4-Lite buszon is. Azt is támogatja, ha a vezérlőjel és a leválasztandó jel eltérő órajeltartományban van; ilyenkor a vezérlőjelet néhány D tárolóval szinkronizálja.

A projektben én is felhasználtam az előző bekezdésben részletezett IP-t. Azonba az AXI4-Lite busz esetén nem működött megfelelően, valamiért néhány vezeték elhagyott az implementáció során. Mivel a hiba okára nem sikerült rájönnöm, saját leválasztást terveztem az AXI4-Lite buszhoz. Ez a Xilinx ajánlásának megfelelő vezetékeket választja le egy-egy multiplexer segítségével. A kialakított leválasztó struktúra vázlatos felépítése a 10. ábrán látható.

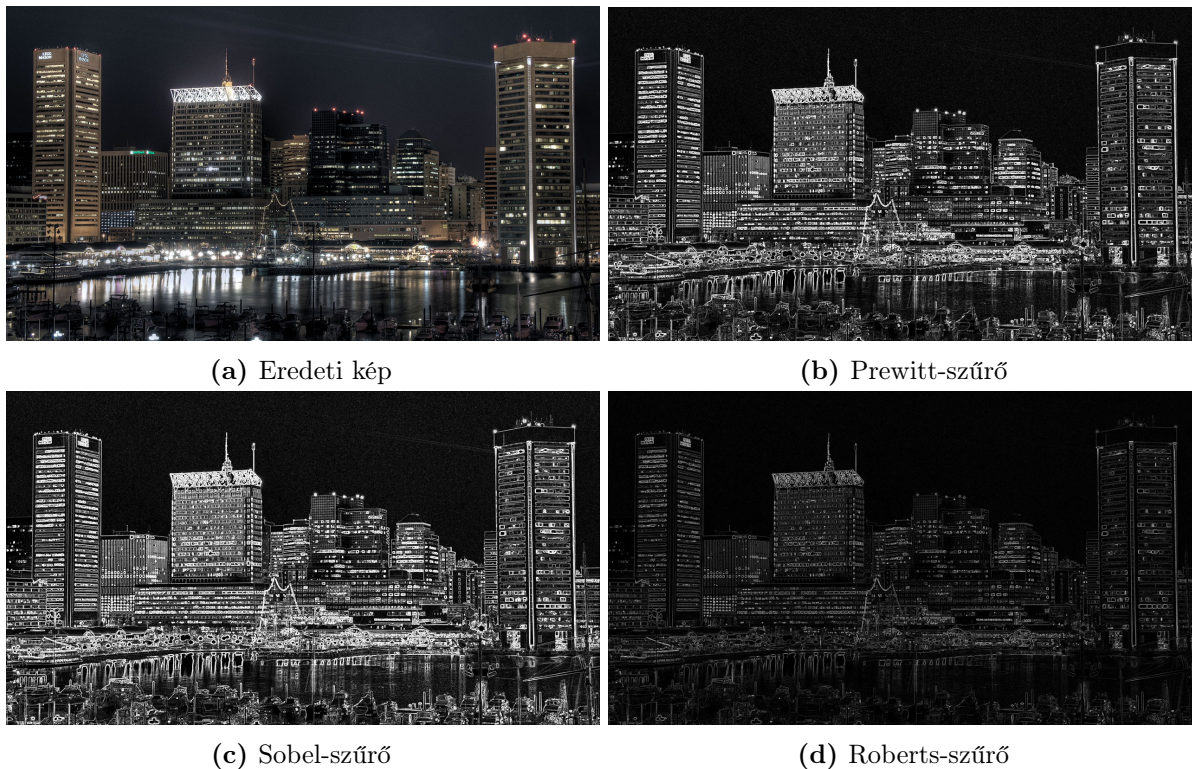


10. ábra. A hardveres leválasztás.

3.5. Eredmények

A szűrők mindegyikét HLS-ben valósítottam meg, és IP magként exportáltam, így fel lehetett használni a szűrőket a Vivado-ban történő tervezéskor. Az elkészült szűrők hatásait a képeken a következőkben láthatjuk.

A 11. ábrán a különböző éldetektáló szűrők alkalmazását láthatjuk. A Sobel-szűrő eredménye a legvilágosabb élek, de ez kissé több zajt is eredményez. A Prewitt-szűrőnél is jól kivehetők az élek, de a nagyméretű felületeknél kisebb a zaj, mint a Sobel-szűrőnél. A Roberts-szűrő kevésbé világos éleket eredményez, de a zajérzékenysége ennek a legjobb.



11. ábra. Éldetektálás különböző szűrőkkel.

A morfológiai operátorok eredménye a 12. ábrán látható. Az erodált képen a sötét régiók, a dilatált képen pedig a világos részek kiemelése látható. A kettő különbségéből képzett morfológiai gradiens pedig lényegében egyfajta élkeresés.

A mediánszűrés eredménye a 13. ábrán látható. Látható, hogy a bal oldali színes impulzuszajjal terhelt képet meglehetősen jó eredménnyel szűri a mediánszűrő. A kapott eredményen csak néhány helyen fedezhetők fel zajos pixelek.



(a) Eredeti kép



(b) Erodált kép

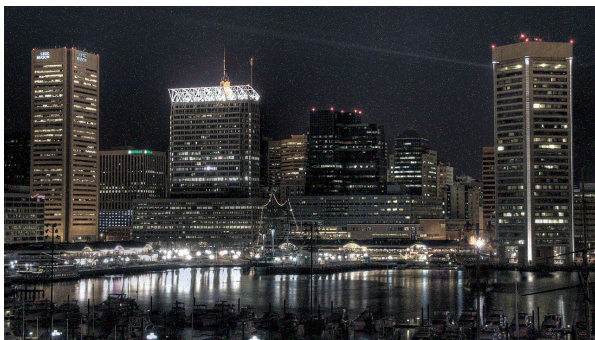


(c) Dilatált kép

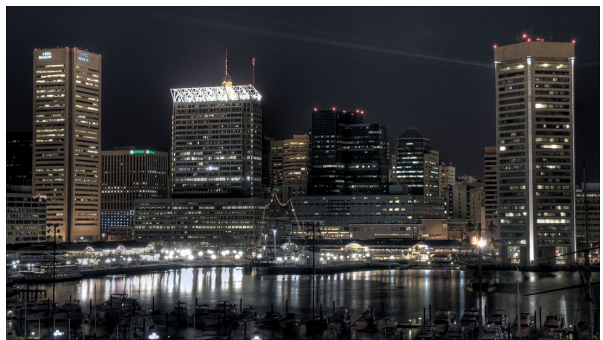


(d) Morfológiai gradiens

12. ábra. A morfológiai operátorok hatása.



(a) Zajos kép



(b) Mediánszűrővel szűrt kép

13. ábra. A mediánszűrő hatása.

A 14. ábrán az unsharp maszk hatása látható. Itt nem az egész képet szűrtam be, hanem csak egy részletét, hogy jobban láthatók legyenek az élesítés hatásai.

Végül a 15. ábrán látható a FAST corner detection algoritmus futásának eredményei. Itt szintén csak képrészletek vannak beszúrva. A bal oldali kép egy tesztkép, itt látható, hogy többé-kevésbé sikeresen megtalálja a sarkokat. A jobb oldalon pedig egy épület képe látható. Itt megfigyelhetjük, hogy közel pontosan detektálta a sarkokat, és a homogén felületeken (például a háttéren) nem volt hibás detektálás.

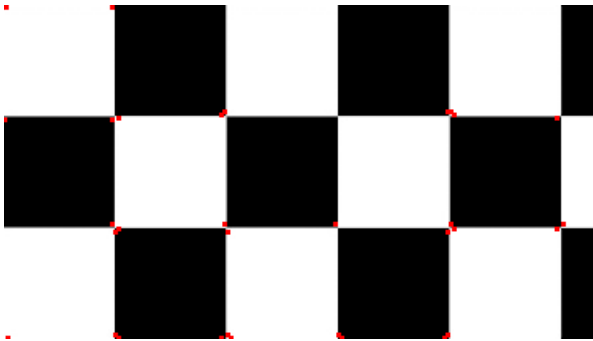


(a) Eredeti kép



(b) Élesített kép

14. ábra. Az unsharp maszk hatása. Láthatóan az élek környezetében nőtt a kontraszt, viszont ezzel együtt a zajszint is emelkedett.



(a) Sakktábla tesztkép



(b) Egy épület sarokpontjai

15. ábra. A FAST sarokpont keresés hatása egy sakktábla tesztképre, és a fenti képekről kivágott egy épületre. Pirossal kiemelve láthatók a megtalált sarokpontok.

A szűrők megvalósítása után azt is megvizsgáltam, hogy várhatóan mennyi FPGA erőforrásra lesz szükségük. Ezt a Vivado HLS az IP exportálás után generált jelentésből lehet kinyerni. A várható erőforrásigények alapján határoztam meg a rekonfigurálható partíciók helyzetét és méretét.

A fejlesztés első állomásaként egy partíciót használtam, és a kisebb méretű szűrőket teszteltem. Ez 900 slice-t, 10 RAMB36-ot és 40 DSP48E1-t tartalmazott. A második változatnál már három partíciót definiáltam, két kisebb méretűt és egy nagyobbat. A két kisebb partíció mérete megegyezett. Ebben a változatban a nagyobb méretű szűrők, csak a nagy partícióba fértek bele, viszont a kisebb szűrők természetesen bárhova elhelyezhetők. A partíciók méretét az 1. táblázat, a szűrők implementáció utáni erőforrásigényét pedig a 2. táblázat foglalja össze.

1. táblázat. A partíciók mérete.

	Slice	LUT	FF	RAMB36	DSP48E1
PB_S1	600	2400	4800	10	20
PB_S2	600	2400	4800	10	20
PB_BIG	2400	19200	9600	30	60

2. táblázat. Az implementált szűrők erőforrásigénye.

	PB	Slice	LUT	FF	BRAM Tile	DSP48E1
Threshold	S	260 (43%)	707 (29%)	874 (18%)	0 (0%)	3 (30%)
Roberts	S	280 (47%)	692 (29%)	750 (16%)	1,5 (15%)	1 (0%)
Sobel	S	358 (60%)	892 (37%)	1086 (23%)	1 (10%)	2 (10%)
Prewitt	S	329 (55%)	801 (33%)	1049 (22%)	1,5 (15%)	2 (10%)
Medián	S	350 (58%)	971 (41%)	1242 (26%)	1 (10%)	2 (10%)
Dilate	S	475 (79%)	1209 (50%)	1809 (38%)	4,5 (45%)	0 (0%)
Erode	S	487 (81%)	1243 (52%)	1809 (38%)	4,5 (45%)	0 (0%)
Morph. Gradient	B	901 (38%)	2286 (24%)	3398 (18%)	4,5 (15%)	0 (0%)
Morph. 5x5	B	2281 (95%)	6873 (72%)	9228 (48%)	8 (27%)	2 (3%)
Top-hat transform	B	1033 (43%)	2429 (25%)	3527 (18%)	21 (70%)	0 (0%)
Unsharp	B	528 (22%)	1127 (12%)	1684 (9%)	4,5 (15%)	34 (56%)
FAST	B	1846 (77%)	4887 (51%)	4901 (26%)	29,5 (98%)	3 (5%)

A táblázat második oszlopában az S betű jelöli a kis partíciót, B a nagy partíciót, a százalékos foglaltságot ezekhez kell viszonyítani. A BRAM oszlopban a használt BRAM mezőket tüntettem fel, a konkrét erőforrást nem részleteztem (ez sorbufferek esetén RAMB18, bizonyos szűrőknél pedig FIFO).

Látható, hogy a thresholding és az éldetektáló szűrők igénylik a legkevesebb erőforrást. A thresholding megvalósításához még sorbuffer sem kell. Az éldetektáló szűrők pedig lineáris szűrést valósítanak meg nagyon egyszerű 3×3 méretű kernellel. A medián, az erózió és a dilatáció valamennyivel több általános erőforrást igényel, ugyanis ezekben az esetekben sorba rendezési feladat van. Az Unsharp maszk az általános logikai erőforrások szempontjából bőven beleférne a kisebb partícióban, azonban a törtszámmal való szorzások miatt itt a szűréshez sok DSP blokk kell. Lényegesen több logikai erőforrást igényel a top-hat transzformáció és a morfológiai gradiens, mivel ezekhez dilatálás és erózió is kell. A top-hat transzformációnál mutatkozó nagy BRAM igény a dilatáció és erózió egymás utáni elvégzése miatt kell, valamint a szűrés alatt az eredeti kép néhány sorát is tárolni kell. Az 5×5 méretű általános morfológiai szűrő (erózió/dilatáció)

is jóval több erőforrást igényel, hiszen itt 25 értéket kell sorba rendezni, ráadásul a strukturáló elem is konfigurálható. A FAST corner detection algoritmus is meglehetősen erőforrás igényes, különösen BRAM tekintetében.

A fentieket összefoglalva megállapíthatjuk, hogy a szűrőknek meglehetősen változatos erőforrás igénye van, emiatt a partíciók kijelölésénél figyelni kell, hogy a megfelelő erőforrások rendelkezésre álljanak. A partíciók méretének meghatározásánál azt is figyelembe kell venni, hogy a logika elemek megfelelően elhelyezhetők és huzalozhatók legyenek. A projektem fejlesztése során többször előfordult, hogy a statikus tervben tökéletesen implementálható szűrő Pblock-ban már nem teljesítette az időzíteni megkötéseket. A másik problémám a nagy méretű partíció elhelyezéséből adódott. A 2.3.3. pontban már ismertettem a partíciók kijelölésének szabályait, azonban ezeket nagyobb blokkok esetén nem egyszerű betartani. A megfelelő időzíteni eredmények elérése érdekében célszerű törekedni a tökéletes elhelyezésre.

A teljes terv erőforrásigénye a 3. táblázatban látható. Megállapíthatjuk, hogy nagyjából az FPGA erőforrások negyede van kihasználva. A 16. ábrán pedig a teljes implementált tervet láthatjuk a partíciókkal együtt.

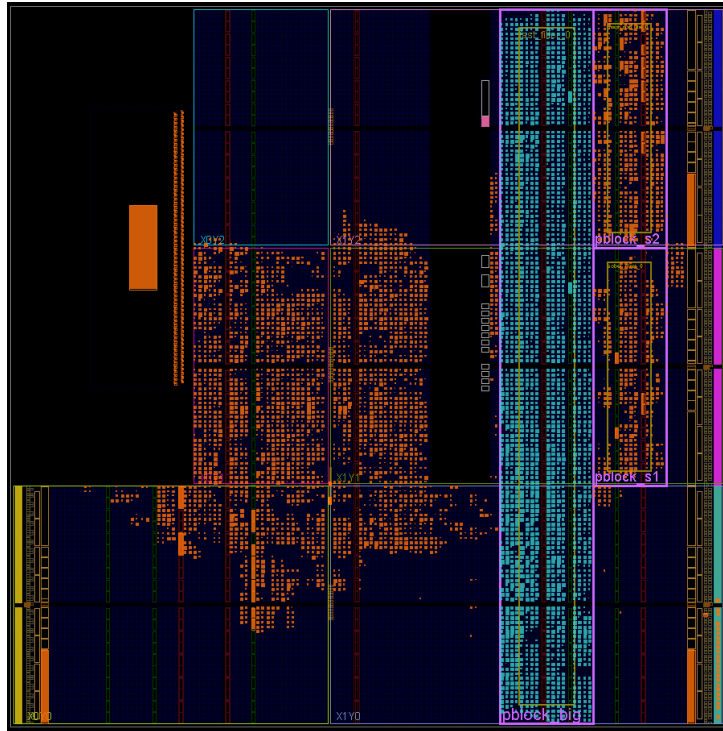
3. táblázat. A teljes implementált terv erőforrásigénye (Morph5x5, Sobel és Erode szűrőkkel).

	Felhasznált	Rendelkezésre áll	Kihasználtság (%)
Slice	6054	13300	46%
LUT logika	14080	53200	26%
LUT memória	1219	17400	7%
FF	21223	106400	20%
Block RAM Tile	25	140	18%
RAMB36/FIFO	11	140	8%
RAMB18	28	280	10%
DSP48E1	13	220	6%
IOB	23	200	12%
MMCE2	3	4	75%

3.6. Szoftveres megvalósítás

A képfeldolgozó lánc menedzseléséhez természetesen megfelelő szoftver elkészítése is szükséges volt. Az alapvető funkciók, amiket a szoftvernek el kellett látni: bemenő képek szolgáltatása SD kártyáról vagy Ethernet portról (valós idejű video bemenettel nem rendelkezik a ZedBoard fejlesztői kártya), a hardverelemek inicializálása, a HDMI transzmitter, a parciális rekonfiguráció és a felhasználói interakciók kezelése.

A szoftveres megvalósításban két lehetőség volt: közvetlenül az ARM processzorra írt (ún. bare metal) alkalmazás fejlesztése, vagy operációs rendszer, Linux használata, és Linuxra történő alkalmazásfejlesztés. Először egy bare metal szoftvert készítettem el, ugyanis így a hardveres modulok felélesztése és a lehetséges hibák megtalálása, kijavítása könnyebbnek tűnt. Továbbfejlesztési



16. ábra. A teljes implementált terv a Vivado-ban (Device nézet, Morph5x5, Sobel és Erode szűrők).

célként azonban a Linux használatát tűztem ki, mivel néhány feladat elvégzése (pl. fájlok kezelése) sokkal egyszerűbb lenne.

Jelenleg a szoftver a következő feladatokat végzi el.

1. Inicializációs lépések, a HDMI Core és a HDMI transzmitter felkonfigurálása.
2. A bemenő képek beolvasása SD kártyáról, vagy tesztábrára generálás. Szintén az SD kártyáról történik a parciális bináris fájlok beolvasása. Ezeket előre eltároljuk a DDR memóriában, hogy konfiguráció esetén gyorsan elérhetők legyenek.
3. A HDMI kapcsolat inicializálása: EDID információ kiolvasása a rákapcsolt kijelzőből, és ez alapján a felbontás beállítása. A felhasználó módosíthatja a felbontást soros porton küldött parancsokkal. (A pixel órajelet egy programozható MMCM blokk állítja elő, így futásidőben változtatható a kimeneti frekvenciája.)
4. A VDMA-k és a képfeldolgozó modul inicializálása, a megfelelő felbontás beállítása. A VDMA-k esetében a működési mód, a framebufferek memóriacímének beállítása.
5. Végtelen ciklusban bizonyos időnként a HDMI transzmitter működésének ellenőrzése. Az UART porton érkező adatok figyelése.

A szoftver a fenti lépések sikerességét vagy hibáit soros porton keresztül kiírja. Ha a felhasználó rekonfigurálási parancsot ad ki, akkor a következő lépések játszódnak le:

1. A képfeldolgozó pipelinehoz tartozó VDMA és a szűrők paramétereinek ideiglenes elmentése. A VDMA írási és olvasási csatornájának leállítása.
2. A hardveres leválasztó egység bekapcsolása, az adatcache kiürítése (flush).
3. A konfiguráció indítása. Ehhez meg kell adni a bináris fájl helyét a memóriában és a hosszát. A konfigurációt az *XDevConfig* driver végzi, ezt korábban parciális rekonfigurációra inicializáltuk.
4. A konfiguráció sikerességéről kapunk egy státuszjelet. Ha hiba történt, kiírunk egy üzenetet a terminálra és kilépünk.
5. A hardveres leválasztó egység kikapcsolása.
6. Az első lépésben elmentett paraméterekkel a VDMA és a szűrők inicializálása, elindítása.

A fenti műveletsor végrehajtása kisebb méretű partíció esetén nagyjából 6-7 ms időt vesz igénybe. Ebből a rekonfiguráció ideje (a processzor globális timer-ével mérve) 2,4 ms. Ezt egy 302 kB méretű bitfájlnál mértem, azaz a rekonfiguráció sebessége nagyjából 121 MB/s. A fenti időadatok alapján a rekonfiguráció néhány videóframe alatt megtörténik 60 Hz-es képfrissítés esetén. Természetesen nagyobb bitfájl esetén arányosan nő a rekonfigurálási idő is.

4. Összefoglalás

4.1. Értékelés

Az önálló laboratórium munkám során megismerkedtem a parciális rekonfiguráció technológiájával és egy ennek a technológiának előnyeit kihasználó projekt fejlesztését is elvégeztem. Úgy gondolom, ez a képfeldolgozási projekt bizonyítja, hogy a parciális rekonfiguráció bizonyos esetekben hasznos eszköz, hiszen ennek alkalmazása nélkül nem tudtam volna ennyire sokféle képfeldolgozó modult implementálni.

A projekt jelenlegi állapotában a HDMI kimenet és a képfeldolgozó modulok működnek. A parciális rekonfiguráció egy modullal és néhány egyszerűbb szűrővel tökéletesen működött. A három partíciós megoldásnál néhány nagyobb szűrő implementálása problémát okozott, emiatt ez a projekt még nem funkcionál tökéletesen.

4.2. Továbbfejlesztési lehetőségek

Természetesen a projekt jelenlegi állapotában nem tekinthető teljesen késznek, számos fejlesztési lehetőség adott. A legfontosabb fejlesztés a Linux használata lenne, mivel így jó néhány feladat kezelését leegyszerűsítene az operációs rendszer használata (pl. SD kártya vagy hálózati kapcsolatok). További fejlesztési feladat a több partíciót alkalmazó képfeldolgozó pipeline kiegészítése a VMUX modullal, és az ehhez történő szoftver kifejlesztése.

A ZedBoard-on nincs valósidejű video bemenet, de érdemes lenne továbbfejleszteni a rendszert úgy, hogy valamilyen forrásból (USB, Ethernet vagy SD kártya) video bemenetet is tudjon kezelni. A képfeldolgozó modulokat is lehetne bővíteni, beleértve olyan modulok elkészítését, ami a Zynq-ben található ARM magokat is kihasználja, hiszen azok jelenleg nagyrészt kihasználatlanok.

A parciális rekonfigurációt tekintve felmerült az az ötlet, hogy a szkriptelt megoldást egy felhasználói felülettel egészítsem ki. Ebben lehetőség lenne a statikus terv és a rekonfigurálandó modulok importálására, és néhány kattintással elvégezhető lenne a teljes PR implementációs folyamat. Mivel jelenleg a Vivado-ban csak parancsokkal lehet egy PR tervet implementálni, úgy gondolom egy ilyen eszköz sokat gyorsítana a fejlesztésen.

5. Irodalomjegyzék

- [1] Xilinx. *UG909 Vivado Design Suite User Guide - Partial Reconfiguration v2015.1*.
- [2] Xilinx. *UG947 Vivado Design Suite Tutorial: Partial Reconfiguration v2015.1*.
- [3] Xilinx. *XAPP1159 Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices*.
- [4] Partial Reconfiguration on FPGAs. Dirk koch.
- [5] Surbhi Singhal. A survey of dynamically reconfigurable fpga devices.
- [6] M. Al Kadi, P. Rudolph, D. Gohringer, and M. Hubner. Dynamic and partial reconfiguration of zynq 7000 under linux. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pages 1–5, Dec 2013.
- [7] Xilinx. *UltraScale Architecture Configuration v1.3*.
- [8] Naresh Ganesh Nayak. Accelerated computation using runtime partial reconfiguration. Master’s thesis, Universität Stuttgart, 2013.
- [9] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, 2014.
- [10] Xilinx. *UG585 Zynq-7000 All Programmable SoC Technical Reference Manual*.
- [11] Xilinx. *UG821 Zynq-7000 All Programmable Soc Software Developers Guide v10.0*.
- [12] Alexander Wold. High level synthesis.
- [13] Xilinx. *UG902 Vivado Design Suite User Guide - High-Level Synthesis*.
- [14] Dr. Himanshu Aggarwal Raman Maini. Study and comparison of various image edge detection techniques. *International Journal of Image Processing*, 3(1).
- [15] Xilinx. *PG020 AXI Video Direct Memory Access v6.2 LogiCORE IP Product Guide*.
- [16] Gleb V. Tcheslavski. Morphological image processing: Gray-scale morphology.
- [17] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, pages 1508–1515 Vol. 2, Oct 2005.