



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Irányítástechnika és Informatika Tanszék

Funkcionális nyelven leírt algoritmusok alapján VHDL leírás automatikus generálása

DIPLOMATERV

Készítette

Suba Gergely

Konzulens

dr. Arató Péter

2011. december 16.

Tartalomjegyzék

Köszönetnyilvánítás	3
Kivonat	5
Bevezető	9
1. A fordítás ki- és bemenete	13
1.1. A Haskell nyelv és sajátosságai	13
1.1.1. Lambda-kalkulus	14
1.1.2. Típusrendszer	15
1.2. A Haskell GHC fordító főbb tulajdonságai	17
1.2.1. Köztes reprezentáció - a Core nyelv	17
1.2.2. Primitív típusok	19
1.2.3. Függvénybehelyettesítés	20
1.3. A VHDL nyelv és sajátosságai	22
2. A fordítási módszer alapelvei és megvalósítása	25
2.1. Az egyes modulok interfészei	27
2.1.1. Bemeneti forráskód	27
2.1.2. Elemi műveleti gráf	28
2.1.3. Kimeneti hardverleírás	30
2.1.4. Művelethalmaz	34
2.2. Forráskód fordítása elemi műveleti gráfra	36
2.2.1. Forráskód magasszintű feldolgozása	38
2.2.2. Elágazások átalakítása	38
2.2.3. Iterációk átalakítása	42
2.2.4. Alkalmazások átnevezése	43
2.2.5. Hívási fára történő átalakítás	43
2.2.6. Faegyszerűsítés	47
2.2.7. Elemi műveleti gráfra történő átalakítás	47
2.3. Elemi műveleti gráf fordítása hardverleíró nyelvre	48
2.4. Alap művelethalmaz definiálása	49
2.4.1. A művelethalmazok kötelező elemei	49
2.4.2. Egyszerű alapl műveletek	51

3. A módszer kiterjesztése többsebességű (multi-rate) problémákra	55
3.1. Többsebességű adatfolyamok a fordítás interfészein	56
3.1.1. Többsebességű elemi műveleti gráf	56
3.1.2. A bemeneti forráskód	59
3.1.3. Kimeneti hardverleírás	61
3.2. A fordítási algoritmus kiterjesztése	62
3.2.1. MapConversion transzformáció	63
3.2.2. A hierarchia automatikus felépítése	63
3.3. A művelethalmaz kivívítása	65
3.3.1. Stream műveletek	65
3.3.2. Tömb műveletek	65
4. A módszer gyakorlati alkalmazása mintafeladatokon	69
4.1. PID szabályozó	69
4.1.1. A fordítás lépései	69
4.1.2. Szimulációs eredmények	77
4.2. MP3 dekódoló algoritmus	78
4.2.1. A Haskell-EOG fordítás eredménye	79
4.2.2. Szimulációs eredmények	82
5. Integrált fejlesztői keretrendszer	85
5.1. Haskell szerkesztő	85
5.2. Elemi műveleti gráf szerkesztő	85
5.3. VHDL szerkesztő	88
5.4. A teljes fordítási folyamat	88
6. Továbbfejlesztési lehetőségek	89
6.1. Optimalizálási lépések	89
6.2. Egyebek	89
7. Összefoglalás	91
8. Kutatás szellemi tulajdon témakörben	93
8.1. Számítógéppel megvalósított találmányok	94
8.2. Keresés az adatbázisokban	94
8.3. Szabadalmak a diplomatervem témájában	98
8.4. A saját megoldás szabadalmazthatósági kérdései	102
Irodalomjegyzék	107
Függelék	109
F.1. Stream műveletek VHDL kódja	109
F.2. Tömb műveletek VHDL kódja	111
F.3. Az Haskell nyelven implementált szintézis program fordítói naplója	113

HALLGATÓI NYILATKOZAT

Alulírott *Suba Gergely*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2011. december 16.

Suba Gergely
hallgató

Köszönetnyilvánítás

Szeretnék köszönetet mondani konzulensemnek, Dr. Arató Péternek a diplomaterv készítése során nyújtott tanácsokért és azért, hogy kérdéseimmel bármikor fordulhattam hozzá.

Köszönöm Patai Gergelynek, hogy a Haskell nyelvhez kapcsolódó problémákban elakadás esetén mindig segített.

Ezúton szeretném megköszönni Dr. Szeberényi Imrének a teljes B.Sc. képzésemen átívelő és nagylelkű konzulensi munkáját, ill. azt, hogy az FPGA-ba szintetizált algoritmusok kutatási útján elindított.

Köszönöm Baji Dénesnek, hogy segítségével már általános iskolásként megismerhettem és megkedvelhettem az elektronika tárgykörét, ill. köszönöm Baksa Gézának a számítástechnikai ismereteim kezdeti fejlesztésében nyújtott segítségét.

A jelen diplomamunkámat és a korábbi tudományos diákköri dolgozatomat támogatta a 72611 sz. „Feladatfüggő többprocesszoros rendszerek tervezése” című OTKA kutatási téma és a TÁMOP IKT-P5-T3 projekt (Feladatorientált többprocesszoros rendszerek tervezési módszertanának kifejlesztése) az Irányítástechnika és Informatika Tanszéken.

Kivonat

Bár egyre nagyobb a hagyományos, processzoros rendszerek működési sebessége, kifejezetten időigényes feladatok, bonyolult biológiai, fizikai számítások szükségessé teszik, hogy az általános célú processzoroknál hatékonyabb hardverstruktúrákat, esetenként célhardvereket fejlesszünk a probléma megoldására. Ezeket a hardverstruktúrákat általában hardverleíró nyelveken (HDL) implementálják. A matematikusok, informatikusok előtt ezek a nyelvek általában kevésbé ismertek, ők az algoritmusokat általános programnyelveken fogalmazzák meg. A hardverleíró és általános programnyelvek nagyon eltérő módon használhatóak (a két módszert szinte szakadék választja el egymástól), és közöttük az átjárás lehetősége csekély. Az átjárás megoldására jelenleg is számos kutatás folyik, ezek a módszerek rendkívül lényeges részét képezik az ún. rendszerszintű szintézisnek.

Diplomatervemben olyan módszer kidolgozása és implementálása a célom, amelyek kiindulópontja a funkcionális nyelven leírt kód. A funkcionális nyelvek közül a Haskell nyelvre koncentrálok, ami egy igen lendületesen fejlődő, a kutatómunkát előnyösen támogató nyelv.

A kidolgozott eljárás fő előnye, hogy automatikusan generálja a bizonyos megkötésekkel rendelkező Haskell nyelvű kódból az FPGA-ba szintetizálható VHDL leírást. Köztes reprezentációként az elemi műveleti gráf (EOG) elnevezésű homogén szinkron adatfolyamgráfot (HSDF) használom. Dolgozatom során külön kitérek az egysebességű (single-rate) és a többsebességű (multi-rate) algoritmusok fordítási lépéseire.

A munkához felhasználok meglévő nyílt forráskódú szoftvereket, pl. a Glasgow Haskell fordítót (GHC), amelynek frontendje egy leegyszerűsített, az eredeti programkódnál kényelmesebben feldolgozható struktúrába tudja alakítani a bemeneti forráskódot. Ebből a struktúrából az általam fejlesztett program a VHDL kódot automatikusan generálja. A fordítási folyamatba beilleszthetünk akár egy olyan pipeline optimalizáló fokozatot, mint pl. az Irányítástechnika és Informatika tanszéken fejlesztett PIPE tervező programrendszert.

Ahhoz, hogy a fordítás felhasználóbarát módon mehessen végbe, egy integrált fejlesztői környezetre is szükség van. Ennek megvalósításához Eclipse alapú keretrendszert fejlesztettem, amely a forráskód szerkesztésétől a köztes gráfok megjelenítésén át a célkódok generálásáig minden egyes részfolyamatot előnyösen támogat.

Az eljárás működését gyakorlati példákon keresztül mutatom be, melyhez egy PID szabályozót és az MP3 dekódoló algoritmus egy részletét használom fel.

Abstract

Although the working speed of the conventional processors is growing continually, there are a lot of problems that are too time-consuming like biological or physical computing and that's why they have to be run in a more efficient hardware structure and sometimes in special hardware for the exact problem. Usually, these hardware structures are written in HDL (Hardware Description Language), that are not so well-known by the mathematicians and IT engineers, they usually write the algorithms in generic programming languages. Writing a problem in HDL or in generic programming language is very different. To overcome this gap, there are a plenty of researches that are very important parts of the so called System Level Synthesis.

The aim of my essay is to work up and implement an HLS method, where the source code is a functional language code describing the algorithm. I focus on the Haskell language, because it has a very powerful development and a large developer community, and it supports the research works well.

The advantage of my method is to generate the VHDL description of an FPGA from the restricted Haskell source code automatically. I use the elementary operation graph (EOG) as an intermediate representation, which is a homogenous synchronous dataflow graph (HSDF). In my essay I discuss the compiling steps of the single-rate and the multi-rate algorithms separately.

For this work I use existing open source software such as the Glasgow Haskell Compiler (GHC) program, because the frontend of this compiler generates a structure that can be processed easier than the original source code. The program I developed generate the VHDL code from this transformed structure automatically. In the compiler sequence we can insert a pipeline optimization stage, like the PIPE design system developed in the Department of Control Engineering and Information Technology.

To compile the code in user friendly way, an integrated development environment is needed. For this purpose I developed a framework based on Eclipse, so we can edit the source code, view the intermediate graphs and can generate the target code in the same CAD software in a beneficial way.

I demonstrate the functionality of the method using practical examples like a PID controller and a part of the MP3 decoding algorithm.

Bevezető

Diplomatervem készítése során olyan módszert dolgoztam ki, amely egy bizonyos megkötésekkel rendelkező funkcionális nyelven leírt algoritmust hardverleíró nyelvre fordít, ezáltal egy kereskedelmi forgalomban kapható FPGA-s fejlesztőkörnyezetet segítségül véve a magasszintű leírásból konkrét hardvermegvalósítást kaphatunk. Mindezt fordítóprogramként implementáltam, amellyel a módszer gyakorlati alkalmazhatóságát demonstrálok.

Sokszor előforduló igény, hogy egy adott alkalmazást hardverként implementáljunk. Bizonyos problémák esetén így sokkal hatékonyabb és nagyobb sebességű feladatvégrehajtást érhetünk el, egy PC fogyasztásának töredéke árán. Máskor konkrét cél lehet a hardverben történő futtatás, mert az algoritmus egy nagyobb hardverrendszer része, és nem szeretnénk PC-t csatlakoztatni a rendszerhez. A rendszerszintű szintézis és a hardver-szoftver együttes szintézis során szükséges, hogy legalább részfeladatokat hardverben implementáljunk.

Egy algoritmust általában sokkal könnyebb magasabb szintű, szoftveres nyelveken megfogalmazni, mint hardverleírásként. A matematikusok és informatikusok felfogásához is közelebb állnak a magasabb szintű nyelvek. A dolgozatom a szoftveres és hardveres nézőpont különbségeinek áthidalására ad segítséget; az eljárás magasszintű nyelvi leírásból automatikusan generálja a hardverhez közeli VHDL kódot.

A magasszintű nyelveken belül a funkcionális megközelítésre koncentrálok. Ezek a nyelvek (pl. Erlang, SML, Haskell) a deklaratív nyelvcsaládba tartoznak, és legfőképpen abban különböznek az imperatív nyelvektől (ilyen pl. a C, Java), hogy explicit vezérlésfolyamot nem definiálnak. Ez azt jelenti, hogy egy deklaratív nyelv azt írja le, mit kell végrehajtania az algoritmus adott lépésének, nem azt, hogy ezeket pontosan hogyan, milyen sorrendben kell tennie. Ez a tulajdonsága kényelmessé teszi ezeket a nyelveket arra, hogy hardvergenerálás bemeneteként használjuk, mert explicit vezérlésáramlás nélkül a párhuzamosan futtatható részek a leírásból egyszerűen adódnak.

A funkcionális nyelvek a hardverben történő implementálás szempontjából rendelkeznek egyéb fontos előnyökkel is. Kizárják pl. a globális változók definiálását, tehát nem hozhatunk létre olyan változót, aminek az értékét több függvény törzséből is megváltoztathatjuk. Még ennél is nagyobb előny, hogy minden változó csak egyszer kaphat értéket, amit életciklusa végéig megőriz. Az ilyen típusú változó könnyen reprezentálható bitenként egy flip-floppal, vagy nagyon egyszerű esetben bitenként egy vezetékkel.

Ezeket a nyelveket nagyon sokszor használják általános, szoftveres környezetben. Az előnyei szembetűnőek gráfok felépítések, bejárások, láncolt listák kezelésekor, de más algoritmusok esetében is van létjogosultságuk. A tiszta funkcionális programozást követve

nagyon jól újrahasználható, kevesebb hibalehetőséget magában foglaló programokat és modulokat fejleszthetünk.

A funkcionális nyelv és hardverleírás közötti fordítás kutatásának manapság megvan a kellő létjogosultsága. Míg imperatív nyelvről, pontosabban C-ről fordító *HLS* (magasszintű logikai szintézis) programok kereskedelmi forgalomban is kaphatóak ([1],[2],[3]), ugyan-ez nem mondható el a funkcionális nyelvek terén. Matematikusoknak kifejezetten előnyös lehet funkcionális nyelvet használni, mert a világképükhöz sokkal közelebb van ez a fajta programozási paradigma. Amennyiben hatékonysági okokból egy funkcionális nyelven megírt algoritmust hardverben szeretnének futtatni, manapság erre kevés lehetőség adódik, és azok is kísérleti, kiforratlan stádiumban vannak. A dolgozatom célkitűzése tehát egy funkcionális bemenetről történő automatizált hardverleírás-generálás, aminek során figyelembe kell venni a jelenleg alap kutatási szakaszban létező megoldásokat.

Diplomatervem készítése során bemeneti funkcionális nyelvként a Haskell egy megszo-
ritott változata mellett döntöttem, az indokaim a következők:

- ez egy nagyon dinamikusan fejlődő nyelv
- a Haskell-hez rendelkezésre áll egy, az elmúlt két évtizedben folyamatosan fejlesztett, viszonylag jól dokumentált fordítóprogram (GHC)
- a GHC fordító köztes reprezentációit könyvtári függvényekkel kézben lehet tartani, tehát saját fordító írásakor felhasználhatóak a GHC már meglévő algoritmusai is
- előtanulmányaim során megismerkedtem, ill. egy tanszéki kutatás keretében már foglalkoztam a nyelvvel, és ez alatt megptaszaltam az előnyeit más nyelvekhez (pl. Erlang és SML, ezeket előtanulmányaimból szintén ismerem) képest

Habár a konkrét megoldásom a Haskell nyelvhez kötődik, nem lenne nehéz áttérni más funkcionális nyelvekre, hiszen a többi funkcionális nyelvnek is a későbbiek során ismertetett lambda-kalkulus az alapja.

Ahogy azt még részletesen taglalni fogom, egy általános Haskell kód nem képzelhető el hardverszintézis bemeneteként, ezért ésszerű megkötések után a 2. fejezetben definiálni fogok egy fordítandó részhalmazt.

A következőkben áttekintem azokat a meglévő, szakirodalomból ismerhető fordítókat, amelyek funkcionális nyelvű forráskódból hardvert vagy hardverleíró nyelvet generálnak.

Lava:

A Lava [26] egy Haskell alapokon nyugvó hardverleíró nyelv. Definiál különféle hardverben közvetlenül alkalmazható típusokat és kombinátorokat, viszont egy általánosabb (lehet akár rekurziómentes) Haskell forráskódot nem tud lefordítani. A Lava hardverközeli nyelv, tehát a diplomatervem egyik fő célkitűzését nem teljesíti, mégpedig, hogy a hardveres világtól elszakadt, szoftveres fogalmakkal felépíthető forráskódot használhassunk a szintézis bemeneteként. (a Lava-ban pl. az elágazást is a hardveres világból ismert multiplexerrel kell megoldanunk, nem pedig a Haskell-ben megszokott *if* és *case* szerkezetekkel) A Lava

nyelvet a μFP [44] és a Ruby [32] nyelvek alapjain fejlesztették, az előbbi hardverspecifikus kombinátorokra épül, az utóbbi pedig egy relációs hardverleíró nyelv, ahol az áramköröket és komponenseiket a ki- és bemenetek közötti relációkkal adják meg. [4]

SASL:

A Cambridge egyetemen egy SASL (Statically-Allocated Stream Language) nevű funkcionális nyelvet és egy ahhoz tartozó hardverszintézist fejlesztettek ki, ami többek között egy PhD disszertáció [30] formájában látott napvilágot 2004-ben. A statikus itt arra utal, hogy fix, hardverben is megvalósítható struktúrákból áll a nyelv, a stream pedig a modulok ki- és bemeneti interfészére vonatkozik.

Az itt bemutatott szintézer képes CSP-t [33] és adatfolyamgráfot előállítani. Az adatfolyamgráf kérés és nyugtázás jeleket is magában foglal, tehát a dolgozatomban használt adatfolyamgráftól (EOG) ebben a lényeges kérdésben eltér.

C_{las}H:

A hollandiai Twente egyetemen 2009-ben két diplomaterv [23, 36] keretében elindult egy projekt **C_{las}H** néven, amely az első nagyobb nyilvánosságot 2010 nyarán kapta a Haskell hackageDB adatbázisában való megjelenéssel. A C_{las}H a CAES Language for Synchronous Hardware elnevezésből született mozaikszó. A fejlesztőknek itt is egy funkcionális nyelvű hardverleírás megalkotása volt a céljuk, amihez alapnak a Haskell nyelvet választották. (nekem hardverleírás megalkotása nem volt célom, sőt mi több, egy hardveres fogalmaktól mentes forráskód definiálását tűztem ki)

A C_{las}H a fordítás első lépéseként a GHC frontendet használja, mégpedig a szintaktikai édesítőszerek eltávolításáig. Ezen a ponton a C_{las}H-ben megvalósított normalizációs eljárások kapják bemenetként az itt kialakult Core reprezentációt, és hajtják végre a normalizációs lépéseket, egészen az elvárt normálformára való átalakításig. A fordítás végső lépése az elért normálforma hardverleírássá alakítása, a szerzők szerint ehhez a lépéshez már egyenes út vezet.

A C_{las}H alapjainak megismerése után nyilvánvalóvá vált, hogy a feladatkiírásom szempontjából ez a leginkább releváns, dokumentált fordító, így a működését részletesebben megismertem. Ezek alapján néhány szempont, amiben a diplomatervem során leírt fordító előnyre tesz szert a C_{las}H-sel szemben:

- explicit adatfolyamgráfot (EOG) használ köztes reprezentációként, így a pipeline optimalizálás a szakirodalomban [21] ismertetett módon elvégezhető
- a modularitást jobban előtérbe helyezi, így a fordítás egyes lépései (Haskell-EOG, ill. EOG-VHDL átalakító és ezek almoduljai) újrahasznosíthatóak
- a forráskódban használható művelethalmaz a fordítóprogram átírása nélkül bővíthető, csupán a művelethez tartozó VHDL modult kell definiálnunk
- a Core reprezentációt kiértékelő algoritmus determinisztikus (a C_{las}H normalizációs lépések nem konfluensek, a determinisztikusság pedig a dokumentáció alapján nem eldönthető)

- használja a GHC simplifier mechanizmusát, így annak nagyfokú optimalizáló tulajdonságát az egész fordítás előnyére tudjuk használni (pl. lefutnak a magasszintű átírási szabályok, a rewrite rule-ok is)

Az egész fordítórendszert modulárisan építem fel annak érdekében, hogy az egyes részfeladatok jól elkülöníthetők, a modulok közti interfészek egyszerűek és teljes mértékben definiáltak legyenek. Ezek alapján cserélhető modulokat kapunk, így a rendszer az egyes részfeladatok újrahasznosítását messzemenően támogatja.

A 1. fejezetben a be- és kimeneti nyelvek, azaz a Haskell és a VHDL sajátosságait taglalom. A 2. fejezetben ismertetem az általam kidolgozott, egysebességű (single-rate) adatfolyamokra alkalmazható fordítói módszert és fordítóprogramot, kitérve a Clash-sel való különbségekre és hasonlóságokra. A 3. fejezetben a módszert kiterjesztem többsebességű (multi-rate) adatfolyamokra, és ehhez kapcsolódóan elkészítem a szakirodalomból [21] ismert EOG modell többsebességű környezetben is alkalmazható változatát, az MREOG-t. A 4. fejezetben konkrét példákon ismertetem a módszer működését, amit az elkészült fordítóprogram köztes reprezentációival és VHDL szimulációval illusztrálok. Az 5. fejezetben a fordítást segítő, általam fejlesztett IDE alkalmazást mutatom be. A 6. fejezetben kitérek a továbbfejlesztési lehetőségekre, végül a 7. fejezetben összefoglalom a dolgozatomban elért eredményeket. A 8. fejezet a Szellemi Tulajdon Nemzeti Hivatala által kiírt hallgatói ösztöndíj pályázat keretében készült.

1. fejezet

A fordítás ki- és bemenete

1.1. A Haskell nyelv és sajátosságai

A Haskell nyelv egy szabványosított, általános felhasználású, tisztán funkcionális, lusta kiértékelésű, erősen típusos, polimorf típusokat támogató nyelv. Ebben a fejezetben a nyelvi sajátosságokat taglalom, amelyeket figyelembe kellett vennem a Haskell nyelvről történő fordítás során.

Az imént felsorolt fogalmak kicsit részletesebben kifejtve a következőket takarják:

- **szabványosított:** a nyelvnek két szabványa létezik, Haskell 98 [42] és Haskell 2010 [40] néven
- **általános felhasználású:** a nyelvet nem konkrét célfeladatra fejlesztették, széleskörűen használható, legyen szó tisztán algoritmikus feladatokról vagy grafikus interfésszel rendelkező programról
- **tisztán funkcionális (pure functional):** egy függvény kimenete csak a bemeneteitől függhet, tehát a függvények mellékhatásmentesek, aminek előnye a kódoláskor történő hibázások csökkenése és a modularitás nagy fokú támogatása, ugyanis a kódrészek egymástól teljesen függetlenül tesztelhetők ill. felhasználhatók
- **lusta kiértékelésű (lazy evaluation):** egy adott kifejezés csak akkor értékelődik ki, amikor a teljes program kiértékelése során erre mindenképpen szükség van (ez a call-by-need stratégia, lásd később)
- **erősen típusos:** egy változó típusa a futási időben nem változhat, vagyis különböző típusú változók között explicit konverzió szükséges (ellentétben pl. a php nyelvvel, ahol a változók típusa futásidőben változhat)
- **polimorf típusok:** paraméteres típusok, ahol a paraméterek szintén típusokat jelentenek (egy típus ezáltal specializáltabb típusok egy halmazát jelentheti - ez nagyban növeli a kód újrafelhasználhatóságát)

A funkcionális nyelvek a lambda-kalkulus-ra épülnek, ezért a következőkben a lambda-kalkulus alapjainak rövid leírása következik.

1.1.1. Lambda-kalkulus

A lambda-kalkulus megértésének jó kiindulópontja Csörnyei Zoltán Lambda-kalkulus - a funkcionális programozás alapjai című könyve [50]. A lambda-kalkulus a lambda-kifejezések vizsgálatával foglalkozik.

Egy **lambda-kifejezés** lehet

- egy egyszerű **változó**
- egy **lambda-absztrakció**, aminek a jelölése: $\lambda v.E$, ahol v egy változó és E egy lambda-kifejezés
- egy **alkalmazás**, aminek a jelölése: FE , ahol F és E egy-egy tetszőleges lambda-kifejezés (ha F egy lambda-absztrakció, akkor FE egy függvényalkalmazás)

A lambda-absztrakcióra gondolhatunk úgy, mint egy függvény megtestesítőjére, az alkalmazásra pedig úgy, mint egy függvény meghívására. (később látni fogjuk, hogy ez a felfogás sántít, de most egyszerűbb csak így gondolni rá) Tekintsük pl. a következő intuitív kifejezést: $(\lambda x.2 * x + 1)3$. Ez egy függvényalkalmazás, amelynek aktuális paramétere 3. Ha az absztrakció x változója helyére behelyettesítjük a 3-at, akkor már egyszerűen látszik, hogy ennek a kifejezésnek $(2 * 3 + 1) = 7$ az értéke.

A következőkben a lambda-kalkulus újabb fogalmait vezetjük be:

- **kötött/szabad változó**: Egy lambda-absztrakció kifejezésében azok a változók kötöttek, amelyek szerepelnek az absztrakció definíciójában, a többi változót szabadnak mondjuk. Tehát $\lambda x.x + y$ esetében x egy kötött, y egy szabad változó.
- **függvény**: Azok a lambda-absztrakciók tekinthetők függvénynek, amelyben nincs szabad változó. (tehát „kimeneti” értéke csak a „bemenettől” függ)
- **α -konverzió**: Alpha-konverziónak nevezzük a kötött változók átnevezését. Pl. $\lambda x.x$ alpha konverzió után kinézhet így is: $\lambda y.y$.
- **β -redukció**: A beta redukció a lambda-kalkulus legfontosabb konverziója, ez adja meg, hogy egy lambda-absztrakcióból és az aktuális paraméteréből álló alkalmazás milyen eredményt ad. A fenti példánál maradva $(\lambda x.2 * x + 1)3$ a beta redukció után $(2 * 3 + 1)$ alakot vesz fel.
- **η -konverzió**: Az eta-konverziót a következő egyszerű egyenlőséggel illusztráljuk: $\lambda x.Fx = F$, ahol x változó és F egy lambda-absztrakció. Ez azt jelenti, hogy egy lambda-absztrakció, ami egy Fx alkalmazásból áll, egyszerűen helyettesíthető az alkalmazás lambda-absztrakciójával, azaz F -fel.
- **lambda lifting**: Olyan módszer, amely eltávolítja a lambda-absztrakciókon belüli szabad változókat. Ezt oly módon hajtja végre, hogy a szabad változókat kötötté teszi egy újabb lambda-absztrakció bevezetésével, amin keresztül az addigi szabad változó értékének átadása biztosítható. Funkcionális nyelvű függvényekre ez a következőképp értelmezhető. Minden szabad változó esetében létrehozunk egy újabb

paramétert, majd a kizárólag kötött változókat tartalmazó lambda kifejezéseket függvényként a globális névtérbe helyezzük. (a függvényeknek persze egyedi névvel kell rendelkezniük)

Kiértékelési stratégiák

A kiértékelési stratégiák szabályok sorozatai, amelyeket végrehajtva egy programozási nyelv kifejezése kiértékelhető, vagyis a kifejezés értéke meghatározható. Lambda-kalkulusban ezeket általában redukciós stratégiáknak nevezik. A következő részben a leggyakrabban alkalmazott kiértékelési stratégiákat mutatom be:

- **call-by-value:** A mohó (más néven szigorú) kiértékelés családjába tartozó stratégia. A függvényhívás előtt kiértékeljük az összes argumentum-kifejezést, így a függvény hívásakor a már kiszámított értékeket adjuk át. Nagyon sok programozási nyelv használja ezt a módszert, nem is kell messzire mennünk, az egyik legáltalánosabban használt imperatív nyelv, a C pontosan így működik.
- **call-by-reference:** Szintén a mohó kiértékelés családjába tartozik. Az argumentumok kiértékelése itt is megtörténik a függvényhívás előtt, de a paraméterátadás a változóra mutató referencián keresztül bonyolódik, amely nagyobb méretű változó esetén jóval költséghatékonyabb lehet. A másik különbség, hogy az indirekció miatt a függvényen belül a külső változó értéke is megváltoztatható. A C++ alapértelmezetten call-by-value átadást valósít meg, de referenciát használva a call-by-reference stratégiát is támogatja.
- **call-by-name:** Lusta (nem szigorú) kiértékelésű. A függvényargumentum kifejezését nem értékeli ki a függvény hívása előtt, hanem csak egy hivatkozást (thunk) ad át a teljes argumentum-kifejezésre. Akkor értékelődik ki az argumentum, ha a függvényen belüli kiértékelés már mindenképpen szükségessé teszi ezt. Ez adja a megoldás legnagyobb előnyét, hiszen addig nem kell foglalkoznunk egy kifejezéssel, amíg biztosan nem válik annak tényleges felhasználása. Ez előny lehet akkor is, ha az argumentum kiértékelése önmagában végtelen ciklushoz vezetne, hiszen ha az argumentumot a függvényen belül nem használjuk, akkor nem állhat elő végtelen ciklus.
- **call-by-need** [22]: Lusta kiértékelésű szintén, a call-by-name stratégiától csak abban tér el, hogy az argumentumkifejezést legfeljebb egyszer értékeli ki, majd az értéket eltárolja egy változóban. Amikor a következőkben szükség van rá, az értéket a változóból veszi ki, tehát tényleges kiértékelés nem történik. A lusta kiértékelés leginkább használatos módja ez a stratégia, többek között a dolgozatomban szempontjából fontos Haskell nyelv is ezen alapszik.

1.1.2. Típusrendszer

A funkcionális nyelvek típusrendszerének alapja az algebrai adattípus (algebraic data type, ADT), amely Haskell esetében a következő definíciót jelenti ([42], 4.2.1 fejezet):

```
data T t1 t2 ... tn = A a1 a2 ... ak
                    | B b1 b2 ... bl
                    | ...
                    | X x1 x2 ... xm
```

T a típuskonstruktor, $t1..tn$ a paraméterei, $A,B..X$ az adatkonstruktorok, az $a1..xm$ közötti szimbólumok pedig az adott konstruktorhoz tartozó típusokat jelentik. Ez hazalmazelméleti nyelven megfogalmazva adatok Descartes-szorzatának összegét, vagyis megkülönböztetett unióját jelenti. Az összeg tagjait, vagyis a **típusalternatívákat** a $/$ (pipe) jelöléssel kell elválasztanunk. Egy $T\ t1\ \dots\ tn$ típusú változó bármelyik típusalternatíva szerinti értéket felvehet.

Az n -esek (hétköznapi nyelven a rekord típusok) a Descartes szorzatnak felelnek meg. Az n -eseket tekinthetjük az általános definíció egyik speciális esetének, amikor az összegnek csak egyetlen tagja van, és az adatkonstruktor $(,)$ (bármennyi vessző szerepelhet) alakban adott. A nyelvben a szokványos n -es jelölés, pl. (a,b,c,d) csak egy szintaktikai édesítőszer, tehát egy jobban olvasható leírása a $(,,,) a\ b$ struktúrának. Néhány példa algebrai adattípusok megadására [5]:

```
data Bool = False
          | True

data Maybe a = Nothing
             | Just a

data Either a b = Left a
                | Right b
```

A *Bool* a szokásos boolean típusnak felel meg, két egyszerű konstruktora van, *False* és *True* néven, az ilyen típusú kifejezés csak ezt a két értéket veheti fel. A *Maybe a* ettől annyiban tér el, hogy amikor a *Just* konstruktort használjuk, akkor meg kell adnunk egy tetszőleges típusú kifejezést is. (pl. *Just "szoveg"*). Az *Either a b* ennél is tovább megy, ez arra jó, hogy egy típusba foglaljon össze két egymással nem kompatibilis kifejezést. Pl. egy ilyen típusú kifejezés lehet *Left "szoveg"*, vagy *Right 67*.

A Haskell szabvány bevezeti a **típusosztály** fogalmát. Egy típusosztály függvényeket deklarál, amelyeket csak a típusosztályba tartozó típusokra lehet alkalmazni. Egy típusosztályt példányosítva konkrét típusok esetére megadhatjuk a függvények definícióját, így a különböző típusokra különböző függvényimplementáció adható. Egy egyszerű példa a *Num* típusosztály, amely algebrai függvényeket deklarál (pl. $(+)$, $(-)$, $(*)$). Az *Int* típus többek között a *Num* típusosztálynak is tagja, mert létezik egy *Num Int* példányosítás. Ebben a példányosításban az algebrai műveleteket az *Int* típus szerint implementálták, így pl. két *Int* típus összeadása esetén az itt definiált kód hajtódik végre.

1.2. A Haskell GHC fordító főbb tulajdonságai

Jelenleg a GHC számít „de facto standard” Haskell fordítónak, versenytársai funkcionalitásban csak messze lemaradva követik. A dolgozatomban szempontjából nagyon fontos volt megismernem a GHC fordító működését, ezért jelen fejezetben kitérek azokra a megismert tulajdonságokra, amelyekre a fejlesztés során nagy szükség volt.

1.2.1. Köztes reprezentáció - a Core nyelv

A GHC egy köztes nyelvet, a Core [47] struktúrát használja az éppen fordított kód egyszerűsített reprezentálására, ami tulajdonképpen egy lambdakalkulus elveit követő **AST** (absztrakt szintaxisfa) reprezentáció. Az optimalizációs lépések nagy része ezen a nyelven történik, ezekre a transzformációkra éppen ezért szoktak Core2Core néven is hivatkozni.

A Core lényegretörő definíciója:

```
data Expr b = Var Id
             | Lit Literal
             | App (Expr b) (Arg b)
             | Lam b (Expr b)
             | Let (Bind b) (Expr b)
             | Case (Expr b) b Type [Alt b]
             | Cast (Expr b) Coercion
             | Note Note (Expr b)
             | Type Type

type Alt b = (AltCon, [b], Expr b)

data Bind b = NonRec b (Expr b)
            | Rec [(b, Expr b)]
```

Az egyes kifejezések jelentése:

- **Var Id**: Egy változó használatát jelenti. Ez tulajdonképpen egy hivatkozás a legfelső szinten megadott globális, vagy a *Let* kifejezéssel leírt lokális változókötésre (más néven összerendelésre).
- **Lit Literal**: Konstanst jelöl. A konstansoknak egy szűk halmazát adhatjuk meg, az összes többi konstans ezekből származhat. (pl. az *App* kifejezéssel, adatkonstruktort alkalmazva) Néhány jellemző típus: *Int#*, *Int64#*, *Float*, *Double*, *FastString*
- **App (Expr b) (Arg b)**: Egy lambda-kalkulus szerinti alkalmazást jelöl, ezek alapján csak egyetlen argumentuma lehet. Több argumentumú függvényeket egymásba ágyazott alkalmazásokkal írhatunk le, így pl. az 1 és 2 számok összeadása a következőképp néz ki: *App (App (Var (+)) (Lit 1)) (Lit 2)*. A példából az is kiderül, hogy a *(+)* függvény itt egy változón keresztül jelenik meg, tehát egy előzőleg definiált függvény alkalmazására változóval hivatkozunk. A GHC az adatkonstrukciókat is alkalmazásként kezeli, tehát algebrai típusú konstansokat is csak az *App* segítségével írhatunk le.

- *Lam b (Expr b)*: Egy lambda-absztrakciót definiál. A lambda-absztrakció tulajdonképpen nem más, mint egy kifejezés burkolva, aminek egyetlen kötött paramétere van (a kifejezés első, itt *b*-vel jelölt paramétere). Minden függvénydefiníció Core-ban történő ábrázolása lambda-absztrakcióval történik. Több argumentum esetén az *App*-hoz hasonlóan egymásba ágyazott *Lam* kifejezések szerepelnek, így pl. az összeadásnál: *Lam(a) (Lam(b) (...))*
- *Let (Bind b) (Expr b)*: Változó-kifejezés összerendelést (*Bind*, lásd később) és egy további kifejezést (*Expr*) definiál. A változó használata minden esetben a már bemutatott *Var* kifejezéssel írható le. A call-by-need kiértékelés miatt egy *Let* csomóponthoz érve csak az *Expr* kifejezéssel kell érdemben törődni, az összerendelésben található kifejezéssel annak használatáig nem kell foglalkoznunk.
- *Case (Expr b) b Type [Alt b]*: A Core nyelv kétségkívül legbonyolultabb szerkezete egy elágazást (*case*) reprezentál. A Haskell forráskód *case* és *if* kifejezései biztosan erre fordulnak, de emellett a változók illesztésére is ezt a formát használják. Az *Expr* itt az a kifejezés, ami alapján kiválasztásra kerül az alternatívák közül egyetlen ág. (a szakirodalomban erre a kifejezésre *scrutinee* néven hivatkoznak) A *b* egy változó kötés (hasonlóan a *Lam b*-jéhez), ez a teljes scrutinee-t köti az itt megadott változóra, ez azért fontos, hogy a továbbiakban hivatkozni lehessen a teljes kifejezésre. A *Type* a *Case* visszatérési értékének típusa, az *[Alt b]* tömb pedig az alternatívák leírása, amiről még szó lesz.
- *Cast (Expr b) Coercion*: Explicit típuskonverzió.
- *Note Note (Expr b)*: A Core fában lehetőség van megjegyzést elhelyezni, ezt a *Note* kifejezéssel tehetjük meg.
- *Type Type*: Típusdefiníció, a Core legfelső szintjén lehet megtalálni.
- *(AltCon, [b], Expr b)*: A *case* alternatívák ebben a formában jelennek meg. *AltCon* az alternatíva konstruktor, ami adatkonstruktor vagy literál lehet. A *[b]* tömb azokat a változókat tartalmazza, amiket az *AltCon* szerinti paraméterekhez kötünk, az *Expr b* pedig az adott alternatíva esetén kiértékelendő kifejezés.
- *NonRec b (Expr b)*: Nemrekurzív összerendelést definiál. A *b* változót és az *Expr* kifejezést rendeli össze, a kifejezésre ezek után a Core *Var* bejegyzésével hivatkozhatunk, megadva a kötött változó, azaz a *b* nevét. Az *Expr* kifejezésben csak a fában ezt a pontot megelőző összerendelésre hivatkozhatunk.
- *Rec [(b, Expr b)]*: Rekurzív összerendeléseket definiál. Egy-egy összerendelés a *b* paraméterben adott változót és az *Expr* paraméterben adott kifejezést rendeli össze. Az *Expr* kifejezésben a fában ezt a pontot megelőző összerendelésre, vagy a saját *Rec* kifejezésben felsorolt többi összerendelésére hivatkozhatunk. (ez a lényegi különbség a rekurzív és nemrekurzív összerendelés között) A kölcsönös hivatkozás kizárólag rekurzív összerendeléssel valósítható meg.

1.2.2. Primitív típusok

Primitív típusnak nevezzük a GHC beépített típusait, vagyis azokat a típusokat, amelyek alapértelmezetten nem szerepelhetnek a fordítandó kódban. A primitív típusok között megtaláljuk többek között a natív környezet típusait, a mai szokványos számítógépes környezetben 32 vagy 64 bites integert (*Int#*), a float (*Float#*) és a double (*Double#*) lebegőpontos típusokat. (a zárójeles nevek a típusok GHC szerinti nevei) A primitív típusok azonosítója konvencionálisan a kettőskereszt karakterrel zárul.

Egy primitív típus lehet **csomagolt** (boxed) vagy **csupasz** (unboxed) attól függően, hogy megvalósítása pointerrel (indirekt) vagy érték alapján történik. Az előbb felsorolt típusok mind csupasz típusok, de a *ByteArray#* pl. már csomagolt.

A primitív típusokat saját programjainkban is használhatjuk, ha a GHC.Prim modul importáljuk, és a kettőskereszt karakterek használatát fordítói opcióval engedélyezzük. ([46], 7.3.2 fejezet)

A primitív típusok azért fontosak számunkra, mert a GHC típusrendszer alapját képezik, tehát ezeket mindenképpen tudnunk kell ábrázolni hardverben. A legegyszerűbb típusok, így az *Int*, *Integer*, *Float* és *Double* is ezeknek a típusoknak a csomagolt változatai, ahogy az a GHC forrásfájlaiból ki is derül:

```
-- GHC.Types modul:
data Int      = I# Int#
data Float    = F# Float#
data Double   = D# Double#

-- GHC.Integer.Type modul:
data Integer = S# Int#           -- small integers
              | J# Int# ByteArray# -- large integers
```

Látható, hogy ezek az egyszerű, leggyakrabban használt típusok is ADT-k. Az *Int* például egy *I#* adatkonstruktorral és az *Int#* primitív típussal képzett típus. Az *Integer* korlátlan méretű egész számot tárolhat, így a megvalósítása is körülményesebb. Attól függően, hogy épp milyen nagyságrendű értéket tartalmaz, lehet az *Int*-hez hasonló (erre szolgál a *S#* konstruktor) és lehet egy kellően nagy tömb által megvalósított (erre szolgál a *J#* konstruktor).

A nyilvánvaló aritmetikai műveleteknek is mind-mind megvan a primitív megfelelője, így az összeadás a *(+#)*, a szorzás pedig a *(*#)* primitív műveletekkel valósítható meg. A GHC.Prim modulban található ezeknek a műveleteknek a „virtuális” definíciója, vagyis egy *let x = x in x* kifejezés, ami tényleges funkcióval nem rendelkezik, ugyanis a primitív műveleteknek már közvetlenül natív kódra kell fordulnia.

A *Num* típusosztályt megvalósítja többek között az *Int* és az *Integer* típus is, emiatt mindkét típushoz definiálni kellett a megfelelő aritmetikai műveleteket. Ez *Int*-ek összeadásánál annyit jelent, hogy a GHC kicsomagolja (**unboxing**) a bemeneteket *Int#* primitív típusra, majd ezeken végrehajtja a *(+#)* primitív műveletet, végül az eredményt újra becsomagolja (**boxing**). Pontosan ezen az elven működik a *GHC.Base.plusInt* függvény,

amit a *Num Int* példányosítás összeadás függvénye hív a *GHC.Num* modulban.

Az *Integer* esetében ennél sokkal bonyolultabb a helyzet, hiszen bármilyen nagyságú számokat is tartalmazhat (persze a számítógép memóriájának függvényében), amiket a véges, *Int#* típusú (*+#*) művelet – rosszabb esetben - egymás utáni végrehajtására kell leképeznie.

Esetünkben nagyon fontos kérdés, hogy az egész típusú literálok hogyan szerepelnek a Core reprezentációban. A *Literal* adattípus *MachInt* konstruktora *Int#* értéket szolgáltat, így a Core fában megjelenik a csomagoláshoz szükséges adatkonstruktor is:

```
app GHC.Types.I\#  
  lit 2
```

Egy 32 bitnél nagyobb *Integer*, történetesen a $7 * 10^9$ esetében a Core a következő, körülményes formát veszi fel:

```
app GHC.Integer.plusInteger  
  app GHC.Integer.smallInteger  
    lit 557549059  
  app GHC.Integer.timesInteger  
    app GHC.Integer.smallInteger  
      lit 3  
    app GHC.Integer.smallInteger  
      lit 2147483647
```

Ha belegondolunk, hogy a 32 bites architektúrán ez a szám egyetlen natív konstanssal nem lenne reprezentálható, akkor teljesen logikus, hogy már a Core reprezentációban is csak 32 biten ábrázolható konstansok és műveletek lehetnek. Ez a továbbiakat annyiban érinti, hogy egy nagyobb szám esetében az aritmetikai műveletek és a konstansok több primitív műveletet eredményeznek, holott egy hardverben tetszőleges bitszámú aritmetika kialakítható, így erre a „bonyolításra” nem lenne szükség.

1.2.3. Függvénybehelyettesítés

A **behelyettesítés** az a kódtranszformáció, ami egy függvény hívási pontjára a függvény törzsét bemásolja. (a szakirodalom az *inline* mellett *unfold* néven is hivatkozik erre a transzformációra, lásd pl. a GHC kézikönyvét [46]) Megjegyzendő, hogy a behelyettesítés lambda-kalkulus szerinti megfelelője a β -redukció.

A fordítás egyik kulcsfontosságú kérdéséhez értünk, ami mind processzoros, mind hardveres rendszer esetén nagyban kihat a futtatás hatékonyságára. Processzoron történő futtatás esetében a behelyettesítés gyorsabb futást eredményezhet (hisz a függvényhívásból eredő overhead eltűnik), viszont jó esély van arra, hogy a program bájtkódja ezáltal növekszik, mert adott esetben több helyre is bemásoljuk ugyanazt a kódot. Ezek alapján

az egyik alapvető szabály, hogy a kizárólag egyszer meghívott függvényekre mindenképpen végre kell hajtani az inlining-ot, ez ugyanis bájt kód növekedést nem okoz, ellenben gyorsabb futást eredményez(het).

A GHC egy összetett eljárással dönti el, hogy egy függvényt behelyettesít-e vagy sem [43]. Az összetettség ellenére vannak alapelvek [6], amik könnyen megérthetőek és ezeket ismerve egyszerűbben tudjuk befolyásolni a GHC behelyettesítési mechanizmusát, ezért most ezekre térek ki. A GHC simplifier programrésze akkor hajt végre automatikus behelyettesítést,

- ha a függvény mérete nagyon kicsi, vagy
- ha a modul export listájában nem szerepeltetjük a függvényt és a modulon belül csak egyszer hívjuk (általánosabban: ha a függvényt biztosan csak egyszer hívjuk meg)

További megjegyzések:

- Ha a modul export listájában szerepel a függvény, akkor a modul fordításánál még nem garantálható, hogy azt csak egy helyről fogjuk meghívni, mert hívhatjuk pl. másik modulból.
- Ha a modul fejlécét megadjuk, de az exportlistát elhagyjuk, az azt jelenti, hogy minden függvényt exportálunk.
- Ha modul fejlécet nem írunk, az eredmény a *module Main where* fejléccel ekvivalens lesz, tehát itt is minden függvényt exportálunk

Hardveres környezetben sokkal egyszerűbbé válhat a megvalósítás a behelyettesítő transzformációk elvégzése után. Ilyen esetben az adatfolyam egyszerű marad, egy adott feldolgozóegység az adatfolyamban csak egyszer fog szerepelni, így költséges multiplexerekre sincs szükség. Kizárólag nagyon költséges függvény esetén van értelme behelyettesítés nélküli megoldásban gondolkodni, tehát amikor a multiplexer beillesztésével jobban járunk, mint a műveletek megfelelő számú sokszorosításával. (ez persze a futási időt is megnövelheti, mert egy feldolgozóegység párhuzamosan több feladatot nem tud ellátni)

A behelyettesítésnek van még egy árnyoldala, ez pedig az információvesztés. Behelyettesítés esetén az inline függvény neve eltűnik, és az az információ is elvész, hogy az adatfolyamgráfban mely csomópontok halmazába alkotta az eredeti függvényt. Ezzel az információvesztéssel a teljes függvényt már nem tudjuk egy feldolgozóegységként kezelni, amivel esetleg későbbi optimalizálási lehetőségeket veszíthetünk.

Polimorf függvény implementálása hardverben túlságosan költséges és ésszerűtlen lenne, mivel akkor a változó típusát, egy metaadatot is nyilván kellene tartanunk. Ennek az a következménye, hogy egy polimorf függvény más-más típusparaméterrel történő hívása más-más tényleges függvény hívásának (nem ugyanazon függvény több helyről történő hívásának) számít, tehát ez nem zárhatja ki a behelyettesítést. A gyakorlatban ezt a speciális esetet úgy tudjuk megoldani, hogy első lépésben specializáljuk a függvényt (behelyettesítjük a típusparamétereket), majd csak egy második lépésben hajtjuk végre a behelyettesítési algoritmust.

1.3. A VHDL nyelv és sajátosságai

A VHDL az egyik legelterjedtebben használt hardverleíró nyelv, amely FPGA és ASIC áramkörök implementálására, szimulálására és dokumentálására szolgál. A VHDL nyelvet az IEEE 1076-os szabványa rögzíti, amelynek legelső verzióját 1987-ban, legfrissebb verzióját [35] pedig 2008-ban adták ki.

A jelenlegi szabvány nagyon terjedelmes (640 oldal), ebben nagyon sok nyelvi kifejezést és lehetőséget definiáltak, ám a kereskedelmi szintézer programok ezeknek csak egy részét támogatják. Ahhoz, hogy a dolgozatom során leírt fordítóprogram a lehető legtöbb szintézerrel együttműködjön, a továbbiakban csak az itt bemutatott egyszerű és általánosan használt kifejezéseket fogom használni.

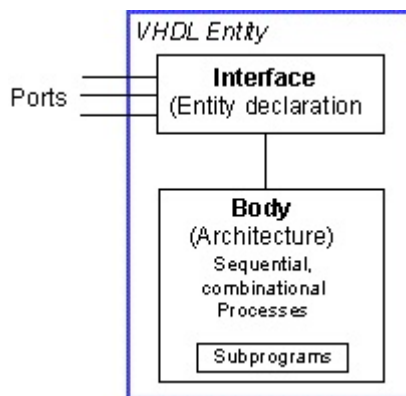
Egy hardverleírás és egy szoftveres programnyelv között a legnagyobb különbség, hogy a szoftver egy processzorban fut, ezért egymás után végrehajtandó utasításokra kell fordítani, míg a hardverleírás vezetékeket, logikai kapukat, regisztereket, stb. ír le statikus módon. Éppen ezért a deklaratív nyelvekhez hasonlóan a VHDL kifejezések sorrendje ugyancsak nem számít.

Egy VHDL modul az 1.1. ábrán vázolt felépítéssel rendelkezik. Minden modul két fontos részből áll. Az egyik az *entity* deklarációja, ami a modul interfésze, tehát a portokat és a generikus paramétereket írja le. A másik az *architecture*, ami a modul törzse, tehát a belső struktúrát és viselkedést írja le az 1.2. ábrán látható formában. (a -- jelek a VHDL kódban megjegyzések írására szolgálnak)

Ha használni szeretnénk egy másik modult, akkor azt példányosítani kell. Ehhez az *architecture*-ben deklaráljuk a modult a *component* kulcsszóval ill. a törzsben el kell végeznünk a konkrét példányosítást.

Az *architecture viselkedési* és *strukturális* model szerint írható le, vagy akár ezeket vegyesen is használhatjuk. A strukturális leírás (1.3. ábra) azt jelenti, hogy modulokat példányosítunk, és csupán az azok közti összeköttetéseket, azaz csak a struktúrát írjuk le. A viselkedési model szerint (1.4. ábra) konkrét logikát, áramköri elemeket használunk műveletek formájában.

A megvalósított fordítóprogramban csak a legszükségesebb típusokat használom. Az *std_logic* az egy bites vezeték, az *std_logic_vector* pedig több bites vezetékek deklarálására szolgál, és a legfelsőbb szintű modulban ezeken kívül nincs szükség más típusokra.



1.1. ábra. VHDL modulok felépítése [20]

```

entity NAME_OF_ENTITY is [ generic generic_declarations;]
    port (signal_names: mode type;
        —
        signal_names: mode type);

end [NAME_OF_ENTITY] ;

architecture architecture_name of NAME_OF_ENTITY is
    — Declarations
    — components, signals, constants,
    — functions, procedures, types

begin
    — Statements
end architecture_name;

```

1.2. ábra. VHDL modulok szöveges váza

```

architecture structural of BUZZER is

    component AND2
        port (in1, in2: in std_logic;
            out1: out std_logic);
    end component;
    — declaration of signals used to interconnect gates
    signal DOOR_NOT, SBELT_NOT, B1, B2: std_logic;

begin
    — Component instantiations statements
    U2: AND2 port map (IGNITION, DOOR_NOT, B1);
end structural;

```

1.3. ábra. VHDL strukturális leírás példa [20]

```

architecture behavioral of AND2 is
begin
    out1 <= in1 and in2;
end behavioral;

```

1.4. ábra. VHDL viselkedési leírás példa [20]

2. fejezet

A fordítási módszer alapelvei és megvalósítása

A most következő fejezetben az általam fejlesztett fordítási módszert mutatom be, amelyet Haskell nyelven implementáltam is a módszer tesztelhetőségét elősegítve. A módszer, vagyis a fordítóprogram követelményeinek elemzését, architektúráját, a köztes adatstruktúrákat és az egyes modulok algoritmusait részletezem a most következő oldalakon.

Ahogy a bevezetőben leírtam és indokoltam, az általam választott funkcionális nyelv a Haskell, ennek értelmében az elkészítendő eljárás ill. szoftver egy Haskell és VHDL nyelv közötti fordítóprogram. A fejlesztés ezen túlmenően további döntéseket igényel, így következő lépésként meghatároztam a fordítóprogrammal szemben támasztott követelményeket:

1. a Haskell forráskód és VHDL leírás közötti fordítás legyen teljesen automatikus
2. fordítás során a felhasználó kapjon hibaüzenetet, ha a forráskód szintaxisa nem megfelelő, vagy ha egy Haskell kifejezés a megkötések miatt nem használható
3. a program moduláris felépítésű legyen, hogy az egyes részek a jövőben minden további nélkül újrahasználhatóak legyenek
4. a jövőben könnyen, tehát a főbb modulok legfeljebb minimális módosítása után implementálhatóak legyenek optimalizációs algoritmusok
5. a forráskód megkötéseit ne kelljen teljes egészében a fejlesztési periódus elején meghatározni, vagyis a használható Haskell kifejezések lehetőleg a fordítóprogram utólagos fejlesztése nélkül, dinamikusan bővíthetők legyenek

A moduláris felépítés akkor előnyös, ha a modulok közötti interfészek egyszerűek és jól meghatározottak. Ennek szellemében interfészként adatfolyamgráfot vezetek be, ami így a Haskell és VHDL nyelvek közötti fordításnál köztes reprezentációként fog szolgálni. A választást az indokolja, hogy egy adatfolyamgráfból viszonylag egyszerűen generálható hardverleírás. Röviden összefoglalva úgy, hogy a gráf csomópontjait a hardverleírás operátoraira vagy moduljaira képezzük le, a gráf élei pedig egyszerűen vezetékek lesznek. A Haskell nyelvű forráskódból tehát ebben az esetben adatfolyamgráfot kell készítenünk, ami

az előbbinél komplikáltabb feladat, de erre a jelen fejezetben kidolgozott módszer megoldást ad.

Mivel a fordítás végső kimenete órajelvezérlést magában foglaló hardverleírás lesz, az adatfolyamgráfot érdemes **SDF**-ként (szinkron adatfolyamgráf) [38] felfogni. Első közelítésben az SDF egyszerűsített, homogén változat (HSDF) vizsgálom, ami csak olyan műveleteket enged meg, amelyek a bemenetről egyetlen adatot (token) mintavételezve egyetlen adatot (token) szolgáltatnak kimenetükön. Az ilyen adatfolyamokat **egysebességű** (single-rate) [25] adatfolyamoknak nevezzük. Később látni fogjuk, hogy az egysebességű adatfolyamok kizárólagossága nagyban korlátozza a fordító bemenetére adható algoritmusokat. Erre a problémára a módszer a 3. fejezetben leírt kiterjesztése fog megoldásként szolgálni.

HSDF-ből több féle konkrét modell [45] is létezik, de mivel ezek között számottevő különbség nincs, a választásom az **EOG** (elemi műveleti gráf) [21] megoldásra esett. Ehhez a modellhez PIPE néven rendelkezésre áll egy, az Irányítástechnika és Informatika Tanszéken fejlesztett pipeline optimalizációs módszer és programcsalád is, ezáltal a 4. követelmény szintén teljesül.

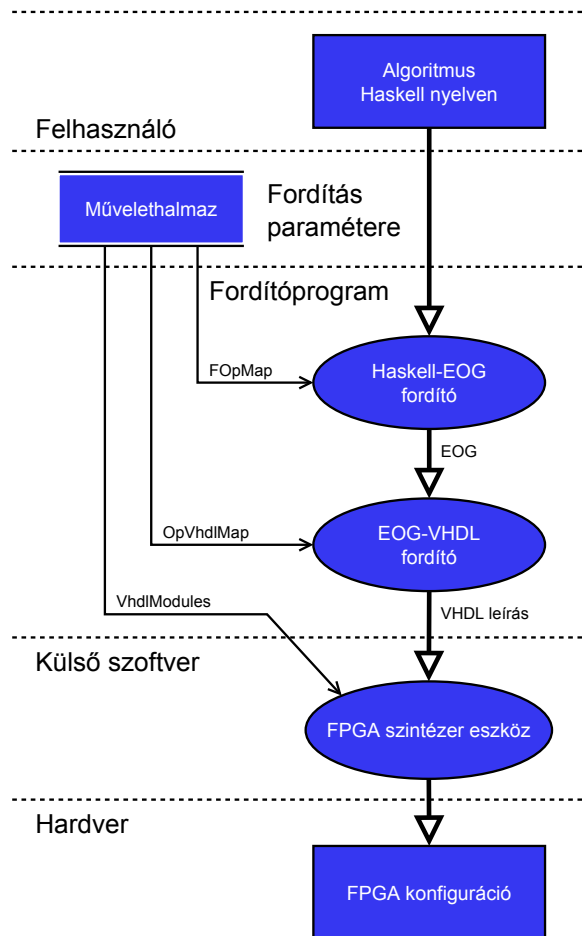
Az EOG tehát két részre, egy Haskell-EOG és egy EOG-VHDL fordítóra osztja a teljes problémát, és ezzel tk. az általános fordítóprogramokból már ismert frontend-backend struktúrát kapjuk.

A forráskódi megkötések dinamikus változtathatóságát (5. követelmény) a művelethalmaz fogalmának bevezetésével valósítottam meg. A **művelethalmaz** körülhatárolja azokat a műveleteket (függvényeket), amelyeket a forráskódban felhasználhatunk, és amelyeket használva a fordítóprogram helyes kimenetet generál. Ettől eltérő műveleteket használva a programnak hibajelzést kell adnia, mert az így leírt forráskód az adott művelethalmaz szerint nem fordítható. Ha a dinamikus változtatás célját el szeretnénk érni, akkor a művelethalmazt a fordítóprogram futási idejében szükséges kiértékelni.

Ezen megfontolásokat alapul véve kialakítottam a 2.11. ábrán látható, DFD (adatfolyam diagram) jelöléssel ábrázolt architektúrát.

A bemeneti adat tulajdonképpen a felhasználótól érkező algoritmus Haskell nyelvű forráskód formájában, amit a fordító Haskell-EOG modulja dolgoz fel első lépésként. A modul felhasználja a művelethalmaz függvény-művelet leképezését (FOpMap) mint másik fontos bemenő adatot, és a transzformációkat elvégezve kimenetként szolgáltat egy EOG-ben reprezentált algoritmusleírást.

Az EOG-VHDL fordító az EOG reprezentációból automatikusan generál egy VHDL modult, amely az algoritmus konkrét műveleteket nem tartalmazó hardveres váza, azaz strukturális leírása lesz. A kimeneten megjelenő VHDL modul példányosíthat egyéb, a művelethalmazban definiált VHDL modulokat, ezek tartalmazzák a műveletek hiányzó implementációit. Az így adódó VHDL fájlok együttesen adják az algoritmus hardveres leírását, amit átadva egy kereskedelembe kapható FPGA vagy ASIC szintézisnek, a kívánt hardvermegvalósítást kaphatjuk.



2.1. ábra. A fordító architektúrája

2.1. Az egyes modulok interfészei

Ebben a fejezetben sor kerül az imént felvázolt modulok ki- és bemeneti interfészeinek részletes ismertetésére, tehát a forráskód, az elemi műveleti gráf, a VHDL kimenet és a művelethalmaz csatolási felületeinek bemutatása következik.

2.1.1. Bemeneti forráskód

A művelethalmaz bevezetése miatt a konkrét, fordítandó függvényeket nem kellett a fejlesztés kezdetén meghatároznom, mert azokat csak az aktuálisan használt művelethalmaz fogja korlátozni. Ennek ellenére fontos kizárni a fordítóprogram által nem értelmezhető forráskódi kifejezéseket, és ezzel körülhatárolni a Haskell nyelv bemenetként használható részhalmazát.

Egy megkötések nélküli Haskell kód hardverként sajnos aligha lenne implementálható, gondoljunk pl. az adatrekurziót vagy függvénytípust tartalmazó ADT-re. Az adatrekurzió egyik egyszerű esete a lista, ami elméletben akár végtelen hosszú is lehet, így világos, hogy megfelelően alacsonyan tartott korlát nélkül hardverben nem implementálható. Ennek megfelelően minden adatrekurziót vagy paraméterként függvényt tartalmazó ADT-t ki kell zárunk a fordítható típusok közül. Ezeket a típusokat más, hardverkompatibilis

típusokkal kell a jövőben helyettesítenünk.

Az általános korlátozások után térjünk át a használható függvények kérdéskörére. A továbbiakban **elemi függvény** néven hivatkozom minden, a művelethalmazban leírt, közvetlenül hardverre fordítható függvényre.

A forráskódban használható elemi függvényeket a gyakorlatban úgy tudjuk a művelethalmaz elemeire korlátozni, hogy elkerüljük a Haskell szabványt implementáló GHC modul, vagyis a *Prelude* importálását. Ezt a következő nyelvi pragma segítségével érhetjük el:

```
{-# LANGUAGE NoImplicitPrelude #-}
```

A megoldás lényege az, hogy a művelethalmazban definiált elemi függvényeket a *Prelude* helyett az *InstructionSet.hs* importálásán keresztül érhetjük csak el, és ennek maradéktalanul teljesülnie kell minden fordítandó modulra.

Az elemi függvényeken kívül csak olyan függvényeket használhatunk a forráskódban, amiket mi magunk definiáltunk, és így igaz rájuk a fenti szabály, vagyis, hogy a *Prelude* helyett az *InstructionSet* modul műveleteit használják. Az ilyen, általunk megadott függvényekre a továbbiakban **összetett függvény** néven fogok hivatkozni. Egyetlen összetett függvényt minden forráskódnak konvencionálisan tartalmaznia kell, ez pedig a *hwmmain*, ami a szoftveres *main*-hez analóg módon hardveres topmodulként fog szolgálni.

A forráskód egy lehetséges példája:

```
{-# LANGUAGE NoImplicitPrelude #-}
module Adder(hwmain) where

import InstructionSet

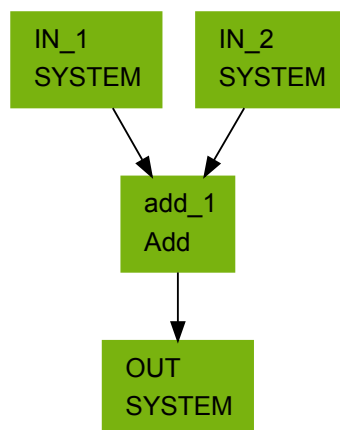
hwmain :: Int -> Int -> Int
hwmain a b = a+b
```

Ez a példa egy nagyon egyszerű összeadó áramkört valósít meg. Látható, hogy a *Prelude* helyett itt az *InstructionSet* modult importáljuk, és a kód egy összeadót valósít meg.

2.1.2. Elemi műveleti gráf

Az EOG a jelen rendszer egy nagyon fontos köztes reprezentációja, amelyet interfészként használva a teljes feladatot két jól elkülönülő részfeladatra osztjuk. Az EOG részletes leírása [21] helyett itt csak egy gyors áttekintés következik a lényegi kérdésekről. Az EOG egy adatfolyam hálózatok leírására szolgáló gráf, amelynek csomópontjai az adatfolyam hálózat **műveletei**, élei pedig a műveletek közötti **adatfolyamok**. Egy egyszerű összeadó EOG reprezentációja látható a 2.2. ábrán.

A csomópontokon a művelet azonosítóját, ill. alsó sorban annak típusát lehet látni. Az ábrán egyetlen valódi, add típusú művelet mellett szerepelnek még a ki- és bemenetek jelzésére szolgáló csomópontok, SYSTEM típusnévvel.



2.2. ábra. Egy összeadó elemi műveleti gráfja

A gráfot egy nagyon egyszerű formátumú fájl ([21], 13. fejezet) segítségével írhatjuk le. A fájl három féle sort tartalmazhat, definiálhatunk művelettípusokat, megadhatunk műveleti csomópontokat és beszúrhatunk megjegyzéseket.

A 2.2. ábra szerinti összeadó szöveges leírása pl. a következő:

```

## a művelettípusok következnek (ez egy megjegyzés)
# type      add      1
# type      SYSTEM   0

## a műveleti csomópontok leírása követezik (ez is megjegyzés)
IN_A        SYSTEM
IN_B        SYSTEM
add_1       add      IN_A      IN_B
OUT         SYSTEM   add_1

```

A **# type** kulcsszó segítségével definiáljuk az **add** és a **SYSTEM** művelettípusokat, és megadjuk azoknak órajelben kifejezett adatfeldolgozási idejét, vagyis azt az időt, ami a bemenet mintevételezése és az eredmény kimenetre való kiadása közötti órajelek száma, azaz **latency**-jét (lappangási idő). (a következő példákban helytakarékoság miatt a típust csak akkor fogom szerepeltetni, amikor a latency is fontos szempont)

A csomópontok megadásakor először az azonosító szerepel, ezt követi a típus, majd azoknak a műveleteknek az azonosítói következnek, amelyek az adott művelet bemeneteire csatlakoznak. Az egyes adatok között whitespace karaktereknek kell szerepelniük. A típusokat az eredeti leírástól eltérően a későbbiekben karakterfüzérként, idézőjelek között fogom szerepeltetni.

Az EOG szöveges leírásából látható, hogy egy műveletnek csak egyetlen kimenete lehet, viszont az az egy kimenet bármennyi művelet bemenetére csatlakozhat. Ebből az következik, hogyha minden művelethez hozzárendeljük a kimenetének adattípusát, akkor ezzel lényegében meghatároztuk az összes adatút típusát is.

minden művelethez saját sorrendi hálózatot társítva oldja meg a vezérlés problémáját. Választásom az elosztott vezérlésre esett, ez ugyanis teljes mértékben skálázható és az adatfolyam-szemlélettel sokkal inkább összeegyeztethető eljárás.

Ezek alapján minden egyes művelethez csatolunk egy DCC-t (elosztott vezérlő cella). Az egyszerű műveletek esetében ez egy nagyon egyszerű kapcsolás, tulajdonképpen csak annyi a feladata, hogy amint az összes bemenetén vezérlőjel érkezett, a kimeneti vezérlőbitet aktívra állítja, ezáltal az adatfolyamban következő műveletet értesíti a művelet befejeztéről. Ez a funkcionalitás egy egyszerű ÉS kapuval megvalósítható.

A vezérlőbitek tulajdonképpen azt jelzik, hogy a hozzá tartozó adatúton éppen érvényes adat van-e.

A konkrét megvalósítás a DCC-hez tartozó csomópont t lefutási idejétől függ. Ha egy olyan csomópontról van szó, amely csak egy kombinációs hálózatot tartalmaz ($t = 0$), akkor egyszerű ÉS kapuról beszélünk, ha viszont $t > 0$, akkor az ÉS kapu kimenetére egy t hosszú késleltetést is implementálnunk kell.

Műveletek kombinációs hálózatként

Legegyszerűbb esetben az elemi függvény funkcionalitásának megfelelő EOG művelet egy egyszerű kombinációs hálózattal megvalósítható. A kombinációs hálózat típusú műveletek VHDL nyelven történő leírását az eddig is használt példán, az összeadás műveleten mutatom be. A 2.3. ábrán látható a művelet VHDL kódja és az ebből Xilinx ISE által generált RTL kapcsolási rajz.

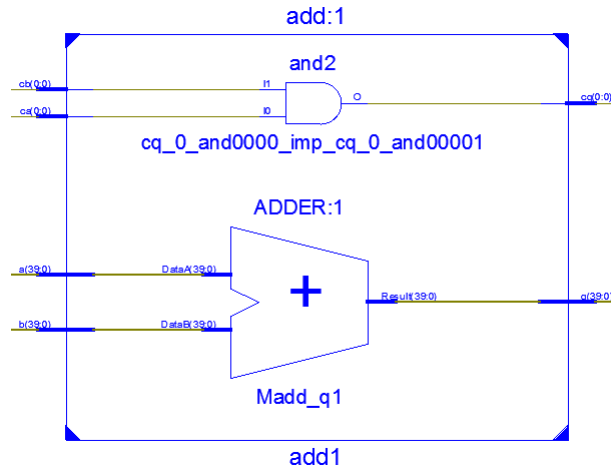
Az *entity* rész meghatározza a VHDL modul interfészét, az *architecture* pedig a belső viselkedést írja le. Látható, hogy bitvektorként két bemenetünk és egy kimenetünk van, a kimeneti érték pedig a két bemenet összege. Ugyancsak két bemenet és egy kimenet szolgál a vezérlés lebonyolítására, de ezek már egy bites portok. Mivel a modul egy egyszerű kombinációs hálózat, megjegyzésben fel kell tüntetnünk a 0 átfutási időt, erre szolgál a *latency = 0* sor.

Minden egyes aszinkron művelethez egy-egy ehhez hasonló VHDL modul tartozik, és ezeket egy-egy külön fájlban tároljuk.

A modulok rendelkezhetnek generikus paraméterekkel, ami azt jelenti, hogy a VHDL kód egy részét a példányosítás helyén megadott adatokkal cseréli ki a szintézer. Ez egy fontos tulajdonság, ugyanis ezt a lehetőséget kihasználva elég pl. egy általános összeadót definiálnunk, aminek a bitszám paraméterét példányosításkor adjuk meg, így nem kell a különböző bitszámú műveletekhez különböző VHDL modult létrehoznunk a művelethal-mazban. A CLasH egyik hátránya, hogy a VHDL kódban nem használ generikus paramétereket.

Műveletek szinkron hálózatként

Léteznek olyan műveletek, amelyek csak szinkron hálózatként valósíthatóak meg, ilyen pl. egy állapotgép, vagy egy blokk-RAM olvasó és író művelete. Ezen kívül minden kombinációs hálózattal megadott művelet leírható szinkron hálózattal is. Egy aszinkron hálózat helyett bevezetett szinkron hálózatnak többek között akkor van létjogosultsága, ha a



```

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

-- latency = 0
entity add is
    generic ( p1: integer );
    port ( a : in  std_logic_vector ((p1-1) downto 0);
          b : in  std_logic_vector ((p1-1) downto 0);
          q : out std_logic_vector ((p1-1) downto 0);
          ca : in  std_logic;
          cb : in  std_logic;
          cq : out std_logic
    );
end add;

architecture Behavioral of add is
begin
    q <= a+b;
    cq <= ca and cb;
end Behavioral;

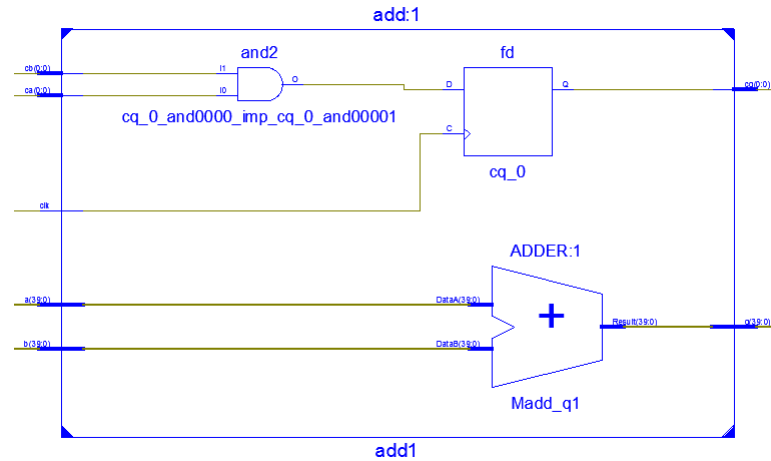
```

2.3. ábra. Aszinkron összeadó

kombinációs hálózatként megvalósított műveletek nagy száma miatt a flip-flopok közötti vezetékek nagyon hosszúak lennének. Ekkor ugyanis a szintézernek az egész alkalmazás órajelét alacsonyan kellene tartania, ami negatív hatással lehet egy másik részegység teljesítményére. Ebben az esetben az szinkron hálózatra való áttérés javíthat a maximális órajelsebességen, így a teljes rendszer teljesítményén.

Az előbbi összeadó szinkronhálózattá alakított változata a 2.4. ábrán követhető nyomon. Az előzőekhez képest az RTL ábrán egyetlen szembeűnő változás található, ez pedig a vezérlés kimeneténél megjelenő D flip-flop a vezérlő ág kimenetén. A kódban megjelenik a *clk* órajel port, és a *process* szekció az órajelre történő vezérléshez. A *latency* itt 1, mivel a művelet egy órajel alatt hajtódik végre.

Időigényesebb műveleteknél (pl. osztás) sokszor előfordul, hogy a futási idő $t > 1$, ilyen



```

-- latency = 1
entity add is
  generic ( p1: integer );
  port ( a : in  std_logic_vector ((p1-1) downto 0);
        b : in  std_logic_vector ((p1-1) downto 0);
        q : out std_logic_vector ((p1-1) downto 0);
        ca : in  std_logic;
        cb : in  std_logic;
        cq : out std_logic;
        clk: in  std_logic
      );
end add;

architecture Behavioral of add is
begin
  q <= a+b;

  process1: process (clk)
  begin
    if (clk'event and clk = '1') then
      cq <= ca and cb;
    end if;
  end process;
end Behavioral;

```

2.4. ábra. Szinkron összeadó

esetben a flip-flop helyett több órajel hosszú időzítő egységet kell beépítenünk, vagy a vezérlés kimenetét közvetlenül a belső állapotgépre kell kötnünk.

Legfelső szintű modul

A legfelső szintű modul az egyetlen, fordítás során generált VHDL modul. Ez tulajdonképpen az EOG gráfnak megfeleltethető hardverleírás, ami kizárólag modul példányosításokat és a modulok közt futó vezetékeket (bitvektorokat) tartalmaz. A topmodulban minden EOG művelet példányosításként, és minden EOG adatfolyam bitvektorként jelenik meg.

```

entity main is
  port ( in1  : in   std_logic_vector (39 downto 0);
        in2  : in   std_logic_vector (39 downto 0);
        in3  : in   std_logic_vector (39 downto 0);
        out1 : out  std_logic_vector (39 downto 0)
  );
end main;

architecture Behavioral of main is
  component add is
    generic ( p1: integer );
    port ( a : in   std_logic_vector ((p1-1) downto 0);
          b : in   std_logic_vector ((p1-1) downto 0);
          q : out  std_logic_vector ((p1-1) downto 0)
    );
  end component;

  signal x: std_logic_vector (39 downto 0);
  signal cx: boolean;

  — osszes tobbi koztes adatfolyam deklaracioja

begin
  — add peldanyositasa add1 neven
  add1: add
    generic map (p1 => 40)
    port map(in1, in2, x);

  — add peldanyositasa add2 neven
  add2: add
    generic map (p1 => 40)
    port map(x, in3, out1);

  — osszes tobbi peldanyositas
end Behavioral;

```

2.5. ábra. VHDL topmodul példa

A topmodul egy lehetséges példáját a 2.5. ábra mutatja. Az *entity* rész a teljes alkalmazás ki- és bemeneteit sorolja fel. Az *architecture* szakaszban először deklarálnunk kell minden később használt modult (*component* kifejezés), majd az adatfolyamoknak megfelelő vezetékeket deklaráljuk. Minden adatfolyamhoz két deklaráció tartozik, egyik az adatvezetékeket, másik pedig a vezérlő vezetéket hozza létre.

Az *architecture* törzsében a műveleteknek megfelelő modulpéldányosítások szerepelnek. A modulok portjaiként egyrészt az imént deklarált vezetékek, másrészt az alkalmazás ki- és bemenetei szerepelhetnek.

2.1.4. Művelethalmaz

A művelethalmaz egy olyan interfész, ami felsorolja a forráskódban legálisan felhasználható függvényeket és típusokat, ill. ezekhez konkrét hardveres megvalósítást társít VHDL leírás

formájában.

A művelethalmazt egyaránt felhasználja a Haskell-EOG és az EOG-VHDL fordítómodul is. Az előbbinek ez alapján kell eldöntenie, hogy a használt műveletek definiáltak-e, vagyis a megadott hibátlan szintaktikájú forráskód ténylegesen fordítható-e. A VHDL-t generáló fordító a művelethalmazban definiált VHDL modulokat példányosítja a kimeneti leírásban.

A művelethalmaz először is magában foglalja a már bevezetett, művelethalmazra jellemző függvényeket exportáló modult (*InstructionSet.hs*), amit a felhasználónak a *Prelude* helyett importálnia kell saját forráskódjában. A használható elemi függvények listáját, vagyis a FOpMap-ot ezen kívül a Haskell-EOG fordító is megkapja a következő példával szemléltetett fájlformátumban:

GHC.Num.(+)	Add
GHC.Classes.()	Or
GHC.Num.abs	Abs

Ez a leképezés kettős célt szolgál. Egyrészt a Haskell-EOG fordító ennek segítségével el tudja dönteni, hogy egy adott elemi függvény használata jogos-e, másrészt az EOG leírásban nem engedélyezett karaktersorozatok helyett helyettesítő neveket vezet be. (a példában a *(+)* nevű helyett bevezetésre került az *Add* függvény)

Egy másik fontos leképezés az OpVhdlMap, ami az adott esetben polimorf módon leírt elemi függvényeket EOG műveleti nevekre képezi le. (ezek a nevek már megegyeznek a műveletnek megfelelő VHDL modul nevével) A polimorf leírás azt jelenti, hogy a függvény neve után generikus paraméterként (*<>*) szerepelnek a függvény típusargumentumai is. A leképezés egy lehetséges példája:

Add<Int>	add<32>
Add<Int8>	add<8>
Add<Float>	add_float
Or	or
Abs<Int>	abs<32>

Ebből a példából az látható, hogy a 8 és 32 bites *Int*-re vonatkozó összeadás egyaránt elvégezhető az *add* nevű VHDL modullal, egy esetleges *Float* típusargumentummal meghívott összeadást viszont már egy másik VHDL modulnak kell hardveresen implementálnia, mert a lebegőpontos értékek összeadása az egész típusokéhoz képest túlságosan eltér.

A művelethalmaz az iménti leképezéseken kívül definiálja az egyes elemi függvények viselkedését leíró VHDL modulokat a 2.3. és a 2.4. ábrákon már bemutatott formában.

A modulok interfésze (entity) kötött sorrendben tartalmazza a portokat, egy két bemenetű EOG művelet esetében pl. az ábrákon látható formában kell azokat megadni. A modul portjainak első fele az adatokat, a második fele a vezérlést szolgálja. Az *a* és a *b* a modul bemeneti paraméterei, a *q* a kimenet, a *c* prefixszel ellátott változatok pedig az ugyanezen jelekhez tartozó vezérlőbitek. Ahogy az EOG interfész leírásánál már írtam, a

műveletek csak egyetlen kimenettel rendelkezhetnek, ez azt jelenti, hogy a modul portjai közül pontosan kettő, egy adat és egy vezérlő port lesz kimeneti irányú.

Az eddigiek összefoglalásaképpen négy pontban felsoroltam a művelethalmaz elemeit:

- *InstructionSet.hs* - az egyedüli, használható, „beépített” könyvtár
- *fop.map* - FOpMap leképezés
- *opvhdl.map* - OpVhdl leképezés
- műveletenként egy-egy *.vhdl* fájl - a művelet hardverleírására

2.2. Forráskód fordítása elemi műveleti gráfra

A 2.1.2 fejezetben bemutatott elemi műveleti gráf a fordítás során egy közbülső, egyszerű absztrakt leírását adja a funkcionális nyelven megfogalmazott algoritmusnak. A most következő fejezetben ismertetem a Haskell forráskód és az elemi műveleti gráf közötti átalakítási lépéseket, és az ezekhez szükséges algoritmusokat.

A forráskódok feldolgozását szerencsére nem kell a legelső lépéssel, vagyis a lexikális elemzéssel kezdeni, ugyanis ehhez a forrásnyelvhez létezik már szabad forráskódú fordító-program. A Haskell forráskódok PC-s tárgykódra fordítására a jelenleg leginkább elterjedt szoftver a Glasgow Haskell Compiler (GHC) [7]. Ez a fordító is, mint általában a fordítók többsége két jól elkülöníthető fordítási fázist különböztet meg, egyik a forráskódot egy köztes, leegyszerűsített nyelvre fordítja (**frontend** fokozat), a másik pedig a köztes nyelvi leírásból végül tárgykódot generál (**backend** fokozat).

A GHC frontend végzi a lexikális és szintaktikai elemzést, a szintaktikai édesítőszereket feloldja, elkészíti az absztrakt szintaxis fát (AST) és alapvető kódoptimalizálást végez. Ezek a funkciók a jelen rendszer számára is nélkülözhetetlenek, ezért célszerű ezt a – már meglévő – programrészt felhasználni, elkerülve ezzel ugyanilyen funkciók saját kóddal történő újbóli implementálását. A ClasH a legelső, optimalizáció előtti Core reprezentációt veszi alapul, az én megoldásom pedig az egyszerűsítések utánit. Ez a megoldás azért előnyös, mert kihasználja a GHC nagyfokú optimalizálását, így többek között a magasszintű átírási szabályok (rewrite rules) is lefutnak.

A GHC backendje úgy készült, hogy a fordítás végén processzoros tárgykódot kapjunk, így ez a rész számunkra nem újrafelhasználható.

A frontend és backend között az 1.2.1. fejezetben már ismertetett Core nyelv szolgál köztes interfészként. A Haskell forráskód helyett az EOG fordításhoz ezt a reprezentációt tekintem kiindulási alapnak. Ha jövőben Haskell helyett más funkcionális nyelvet kívánunk hardveres környezetre fordítani, akkor egy jó megoldás lehet Core alakra hozni azt; így a dolgozatom során elkészült modulok teljes egészében felhasználhatóak lesznek, csupán a GHC frontend rész helyett kell egy adott nyelvre specializált frontendet implementálni.

A GHC alapjában egy több platformra is lefordított parancssori program, de emellett a konkrét fordítást végző részeket kiexportálták könyvtári függvényekként is, vagyis egy Haskell nyelven írt programmal könnyedén tudunk Haskell kódot fordítani, sőt, a fordítás lépéseit és azok paramétereit mélyrehatóan tudjuk szabályozni.

A következő alfejezetekben bemutatom az egyes lépések algoritmusait, amik alapján implementáltam az elkészült Haskell-EOG modult. Ebben a részben szerepelni fognak Core nyelvi részletek, így a használt szintaktikáról szót kell ejtenem. A Core faszerkezet szülő-gyermek viszonyát behúzással fogom jelölni, ahol minden azonos szülőhöz tartozó gyermek azonos oszlopban kezdődik.

Példán szemléltetve:

```
a =
  lam t u
    app (+)
      var t
      var u
```

Az *app* tehát a *lam* egyedüli gyereke, a két *var* szülője pedig az *app*. Az *a =* egy azonosító-kifejezés összerendelést (a Core szerint ez a *Bind*) jelent, ahol a kifejezést gyerekként ábrázoljuk (ez lesz a *lam*).

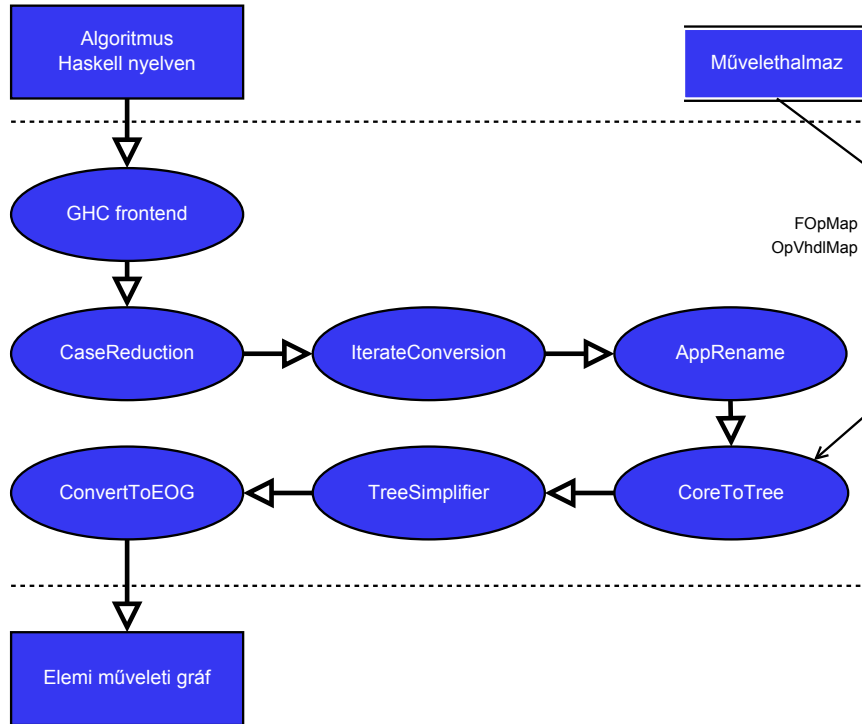
A *var*-nak és a *lit*-nek nincs gyereke, a *lam*-nak és a *let*-nek egy gyereke van. Az *app*-nak annyi gyermeke van, amennyi paraméterrel rendelkezik, ami eltérés az eredeti Core reprezentációhoz képest, hiszen a lambda-kalkulusban csak egy paraméterű alkalmazások találhatók. Ezt a reprezentációt az egyszerűség kedvéért vezetem be, de a gyakorlati megvalósításnál ugyancsak előkerül ez az átalakítás. Szerencsére a bőbeszédű Core *App* csomópontjait függvéynévre és paraméterlistára átalakító műveletet egy GHC könyvtári függvény (*collectArgs*) meghívásával könnyedén el tudjuk végezni.

A *case* csomópont sorrendben első gyermekeként a saját kifejezését, többi gyermekként pedig az elágazó alternatívákat tartalmazza.

A 2.6. ábrán a Haskell-EOG fordító architektúrája látható, ahol az egyes algoritmusok címszavakban:

1. GHC frontend: forráskód magasszintű feldolgozása
2. CaseReduction: elágazások átalakítása
3. IterateConversion: iterációk átalakítása
4. AppRename: alkalmazások átnevezése
5. CoreToTree: hívási fára történő átalakítás
6. TreeSimplifier: faegyszerűsítés
7. ConvertToEOG: elemi műveleti gráfra történő átalakítás

A következő alfejezetekben részletesen ismertetem a fordítás egymás-utáni fázisait.



2.6. ábra. A Haskell-EOG modul architektúrája

2.2.1. Forráskód magasszintű feldolgozása

Ahogy már szó volt róla, a Haskell forráskód Core reprezentációra alakítását a GHC könyvtári függvényeivel el lehet végezni. A ClasH megoldásával ellentétben nem a kezdeti Core kimenetet használom fel, hanem kihasználom a GHC optimalizációs algoritmusait, és az algoritmusok lefutása után keletkező Core állapotot használom a következő átalakítási lépés bemeneteként.

A GHC három optimalizációs szintet különböztet meg, 0 a nem optimalizált, 1 a jó minőségű, de nem túl hosszú fordítással elérhető, a 2 pedig a maximálisan optimalizált kód. Mivel a hardver szempontjából nagyon fontos az erőforrásigény minimálisra szorítása, ezért a fordítóprogramban a maximálisan optimalizált kódot használom.

2.2.2. Elágazások átalakítása

A Core reprezentáció kétségkívül legnagyobb bonyolultságú elemei a *Case* csomópontok, amik egyaránt használatosak **elágazások** és **mintaillesztések** leírására. A fordítás során előbb vagy utóbb a Core ezen részeit is EOG elemekre (műveletekre és adatfolyamokra) kell fordítanunk.

Azt a megoldást választottam, hogy a *Case* csomópontokat már a fordítás elején *Let* és *App* csomópontokra alakítom, ugyanis ezek EOG-ra fordítása triviálisabb, és a később sorra kerülő CoreToTree algoritmus ezt minden további erőfeszítés nélkül meg is fogja tenni. Ennek a lépésnek a mielőbbi elvégzése azért fontos, mert ezáltal a bonyolult *Case* szerkezettel a továbbiakban nem kell foglalkoznom.

A [36] 4.3.5 fejezetében ismertetett Case normalization eljárások nagyon hasonló felada-

tot látnak el, mint az itt leírt **CaseReduction** transzformáció. A különbség az, hogy az én megoldásomban minden *Case* csomópontot azonnal megszüntetnek, ezzel egyszerűsítve a további átalakítási lépéseket.

01: case of b	01:
02: E	02: let scrutinee_1 = E
03: .	03: b = scrutinee_1
04: .	04: app mux
05: alt C1 c11 c12	05: let c11 = app select<C1,0>
06: .	06: var scrutinee_1
07: .	07: c12 = app select<C2,1>
08: .	08: var scrutinee_1
09: .	09: app ifcon<C1>
10: .	10: var scrutinee_1
11: E1	11: E1
12: alt C2 c21 c22	12: let c21 = app select<C2,0>
13: .	13: var scrutinee_1
14: .	14: c22 = app select<C3,1>
15: .	15: var scrutinee_1
16: .	16: app ifcon<C2>
17: .	17: var scrutinee_1
18: E2	18: E2

2.7. ábra. *CaseReduction* általános képlet (a pont-ok csak az átláthatóságot segítik)

Az itt ismertetett algoritmus az, amely bemenetként kapja a közvetlenül a GHC könyvtárakból származó Core kódot, majd a 2.7. ábrán általánosságban ismertetett átalakítást elvégzi.

Első lépésben bevezetünk egy új változót, *scrutinee_1* néven, amihez a fenti kódban *E*-nek jelölt *scrutinee*-t kötjük. (ez analóg a ClasH Scrutinee simplification normalizációjával) A *Case* szintaktikája megengedi egy új változó bevezetését (a fenti kódban ezt *b*-vel jelöltem), ilyen esetben ehhez hozzárendeljük a *scrutinee_1*-et, tehát végeredményben mindkét változó a scrutinee kifejezésre fog mutatni. A ClasH ezt az utóbbi változó bevezetést eliminálja azáltal, hogy a kódban az összes ilyen változót lecseréli az új *scrutinee_1* változóra. Ezt a megoldást nem tartottam szerencsésnek, ugyanis az alkifejezések újabb bejárása miatt plusz költséggel jár.

Az előző eljárással létrehozott *Let* törzsébe a *mux* nevű, ebben a fázisban létrehozott alkalmazás kerül. A *mux* egy olyan függvény, amely több *Maybe a* típusú bemenetből egyetlen *Maybe a* típusú adattal tér vissza. Azzal a bemenettel térünk vissza változtatás nélkül, amelyik *Just* konstruktorral rendelkezik. (ha több ilyen is van, akkor a paraméterek sorrendjében legelsővel térünk vissza)

Egy három bemenetű *mux* viselkedését tekintve a következő, gyakorlatban nem realizált Haskell kóddal lenne ekvivalens:

```

mux3 a b c =
  if (isJust(a))
    then a
    else
  if (isJust(b))
    then b
    else c

```

Fontos megjegyezni, hogy ez a Haskell kód nem kerül lefordításra, mert ha ez megtörténne, azzal az egész átalakítás értelmét vesztené, ugyanis ez a kód szintén tartalmaz *Case* szerkezetet (*if* formájában), tehát ezzel a *Case* teljes eliminálása nem valósult volna meg. A megoldás az lesz, hogy ezt a speciális függvényt elemi függvényként használom, tehát közvetlenül hardverben fogom implementálni.

Térjünk át a *Case* alternatíváira, az *alt* csomópontokra. Minden ilyen bejegyzés egy-egy *Let* csomópontra fordul, amiben az *alt*-ban bevezetett összes változót kötni fogjuk. (hasnolón a ClasH Case Normalization megoldásához) A kötés változója az *alt*-ban megjelenő változó, az ahhoz kötött kifejezés pedig a *select* alkalmazás lesz.

Példaként tekintsük meg, hogy a *select<C1,0>* milyen képzeletbeli viselkedést takar (a változó megnevezések a 2.7. ábrából származnak):

```

select_C1_0 x = case x of {
                  C1 c11 _ -> c11
                }

```

Ezzel tulajdonképpen az illesztést valósítottuk meg oly módon, hogy a *select* függvény egy adott ADT kifejezésből az adott konstruktorhoz tartozó egyik argumentumértéket adja vissza. Az iménti példakód a *C1* konstruktor utáni legelső paramétert, azaz a *c11*-et adja vissza. A *mux*-hoz hasonlóan ez a kód is kizárólag hardverben kerül implementálásra.

A *Let* kifejezés törzséről még nem esett szó. A törzs egy *ifcon* nevű függvény lesz, ami azért felel, hogy csak a megfelelő konstruktor esetében értékeljük ki az adott kifejezést, és adjuk vissza annak eredményét.

Az előbbiekhöz hasonlóan az *ifcon* is leírható képzeletbeli Haskell kóddal:

```

ifcon_C1 x e = case x of {
                C1 _ _ -> Just e;
                _ -> Nothing
              }

```

Ebben az esetben tehát akkor adjuk vissza az *e* kifejezést (ami az *e* kiértékelését fogja maga után vonni), ha az *x* argumentum a *C1* konstruktorral kapott értéket. A bevezetett *mux*, *select* és *ifcon* alkalmazások tehát az imént ismertetett szemantikákkal helyettesíteni tudnak egy *Case* csomópontot, így ez utóbbira a továbbiakban nem lesz szükség. Példaként tekintsünk meg egy illesztést (2.8. ábra), és egy elágazást (2.9. ábra) megvalósító Core fát, és azoknak CaseReduction átalakítás utáni változatait!

01: case (bind wild)	01:
02: app foo	02: let scrutinee_1 = app foo
03: lit 3	03: lit 3
04:	04: app mux
05: alt (,,) a b c	05: let a = app select<(,,),0>
06:	06: var scrutinee_1
07:	07: b = app select<(,,),1>
08:	08: var scrutinee_1
09:	09: c = app select<(,,),2>
10:	10: var scrutinee_1
11: app bar	11: app bar
12: var a	12: var a
13: var b	13: var b

2.8. ábra. *CaseReduction* illesztés példa

01: case	01:
02: app foo	02: let scrutinee_2 = app foo
03: lit 3	03: lit 3
04:	04: app mux
05: alt Left a	05: let a = app select<Left, 0>
06:	06: var scrutinee_2
07:	07: app ifcon<Left>
08:	08: var scrutinee_2
09: app bar1	09: app bar1
10: var a	10: var a
11: alt Right a	11: let a = app select<Right, 0>
12:	12: var scrutinee_2
13:	13: app ifcon<Right>
14:	14: var scrutinee_2
15: app bar2	15: app bar2
16: var a	16: var a

2.9. ábra. *CaseReduction* elágazás példa

2.2.3. Iterációk átalakítása

A Core *Rec {}* kifejezéseit a fordítóprogram nem dolgozza fel, ez ugyanis rekurzív függvényeket vagy változóhivatkozásokat eredményezne. Rekurzió nélkül viszont algoritmusok nagy számát zárnánk ki a fordítás alól, ami nem engedhető meg. A probléma megoldására az *iterate* Haskell függvény használatát vezetem be, amelynek eredeti definíciója a GHC.List modulban található:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Az *iterate* első paramétere egy egyparaméteres *f* függvény, ami minden iterációban meghívódik. Az *f* az első iterációs lépésnél az *iterate* második paramétereként megadott kezdeti értéket, a következő iterációk során pedig az azt megelőző iterációs lépés kimenetét kapja.

Az *iterate* közvetlenül nem fordítható EOG-ra, mivel függvényparamétert tartalmaz. A megoldás az **IterateConversion** Core2Core transzformáció bevezetése, amely a fordító következő lépései számára feldolgozható formába alakítja az iterációt.

A CaseReduction lépésnél megismert szintaktika szerint elkészítettem az IterateConversion transzformáció általános képletét, amit a 2.10. ábrán ismertetek.

01: app iterate_old	01: let x =
02: lam x	02: app iterate_new
03: E	03: var it_out
04: E2	04: E2
05:	05: let it_out =
06:	06: E
07:	07: var it_out

2.10. ábra. *IterateConversion* általános képlet

A *lam* csomópont helyett egy *let* csomópontot vezetek be, ami a *lam*-ban szereplő *x* paraméterhez egy újonnan létrehozott kifejezést rendel. Egy újabb *let* csomópont szintén bevezetésre kerül, ami az *it_out* azonosítóhoz az eredeti belső függvény *lam* nélküli törzsét, azaz az *E*-t rendeli. Ez utóbbira azért van szükség, hogy az *E* kifejezésre a továbbiakban két helyről is hivatkozhatunk.

Az *x*-hez rendelt új kifejezés a függvényparamétertől mentes *iterate_new* alkalmazás, aminek első paramétere az imént bevezetett *it_out*-ra mutató változó, második paramétere pedig az *E2*, vagyis a kezdeti érték kifejezése.

Az átalakítást megvizsgálva észrevehető, hogy a *var it_out* bevezetése rekurziót eredményez, ugyanis az egy később bevezetett azonosítóra hivatkozik. (ehhez még azt kell látni, hogy az *E* kifejezésgráf valahol tartalmaz egy *var x* csomópontot). Ahhoz, hogy a későbbi CoreToTree algoritmus a behelyettesítésekkel ne kerüljön végtelenciklusba, a *var it_out*-ot egy olyan megjegyzéssel kell ellátnunk, ami az *inline*-t ezen a ponton

megakadályozza.

2.2.4. Alkalmazások átnevezése

Minden elemi függvény alkalmazásából EOG művelet lesz, ezért ezeknek az alkalmazásoknak egyedi névvel kell rendelkezniük. Az itt ismertetett, alkalmazásoknak egyedi nevet adó algoritmus a továbbiakban **AppRename** néven fog szerepelni.

A változók egyedi nevének tárolására a GHC egyszerű lehetőséget biztosít a változók *unique* mezőivel. A fordító implementálása közbeni saját tapasztalatom, hogy a GHC által biztosított *unique* mezők nem mindig egyediek, pl. a *Case* csomópont egyes alternatívái tartalmaznak azonos *unique* értéket. (a *Case* csomópontokat az előzőekben elimináltuk, de az átalakítás után a nevek az új típusú csomópontokban tovább léteznek)

Megoldásomban az alkalmazások változóinak *unique* mezői *oi* formájú értéket kapnak, ahol *i* az alkalmazás egyedi sorszáma. A sorszámot a Core fa inorder bejárása adja, ami azt jelenti, hogy ebben a sorrendben kerülnek az egyes indexek az alkalmazások változóinak *unique* mezőibe.

2.2.5. Hívási fára történő átalakítás

A Core reprezentáció kiértékelésére és egy új, operációkat és hivatkozásokat tartalmazó faszerkezet, az **OpTree** létrehozására a **CoreToTree** algoritmus szolgál. Új faszerkezet létrehozása azért volt szükséges, mert a Core reprezentáció túlságosan lambda-kalkulus közeli, így pl. a több bemenetű műveletek leírása nagyon körülményes.

A C_{las}H megoldásban normalizációs eljárások sokaságát alkalmazzák, hogy végül egy hardverközeli Core szülessen. A megoldás hátránya, hogy az egyes normalizációs eljárások futtatási sorrendje nem kötött, egy adott állapotban az az átírási szabály indítható, amelynek az előfeltételei teljesülnek. Egy adott állapotban több normalizációs lépés is végrehajtható, és a lépések sorrendjétől függően más-és-más végeredmény születhet, tehát az algoritmus nem konfluens. (a szerző, véleményem szerint hibásan, a „nem-determinisztikus” szót használja ennek kifejezésére ([36] 4.4.2. fejezet))

Az én megoldásomban normalizálás helyett **supercompiling** közeli technikát alkalmazok. A supercompiling-ot először Turchin mutatta be 1986-ban [48], és azóta is nagy számú cikk foglalkozott a témával, az egyik legfrissebb pl. Neil Mitchell 2010-es publikációja [41].

A supercompiling tulajdonképpen a program fordításidejű végrehajtása, tehát pontosan olyan sorrendben járjuk be a Core szerkezetet, ahogyan azt a processzor a programunk futása közben tenné. Mivel a Haskell a lusta kiértékelésen alapul, ezért a bejárásnak is eszerint, vagyis call-by-need stratégiával kell történnie.

Ahhoz, hogy a bejárás minden pontján a fordítóprogram tisztában legyen az ott használható azonosítók halmazával, bevezetek egy környezetet reprezentáló típust. A környezet tulajdonképpen egy sor azonosító-kifejezés párt foglal magában, aminek az értelme az, hogy a későbbiekben az azonosítóval hivatkozhatunk az adott kifejezésre.

Fontos észrevenni, hogy a környezetet csupán két típusú Core csomópont képes bővíteni, a *Let*, ami új lokális változót hoz létre és a *Lam*, ami új paraméterváltozót hoz létre. (a

Case szintén bővítené a környezetet, de az előző algoritmusok után ez a típusú csomópont már nem jelenlhet meg)

A bejárás legbonyolultabb pontja az, amikor egy változó kiértékelésekor a változó által hivatkozott kifejezést kell behelyettesítenünk. Ez a fázis nagyban hasonlít az 1.2.3. fejezetben ismertetett GHC által megvalósított inlining-ra.

Az algoritmus bejárja a teljes Core fát, mégpedig úgy, hogy első körben kizárólag a *hmain* globális összerendeléshez tartozó kifejezést értékeli ki. Nézzük, hogy az egyes csomópontokhoz érve milyen lépéseket kell végrehajtanunk:

- *Lit literal*: A legegyszerűbb eset, egyetlen literált definiál. A *Lit* a Core fában egy levél, ezért csak annyi a dolgunk, hogy az OpTree fa adott pontjához ezt a levelet hozzáadjuk.
- *Let bind expr*: A call-by-need stratégia értelmében a fához nem adunk hozzá egyetlen elemet sem, viszonyt a környezetet a *bind*-ben megadott kötéssel bővítjük. (azaz létrehozunk egy thunk-öt [34]) A bővítés után a törzset (*expr*) az új környezetet átadva kiértékeljük, így biztosítva, hogy a továbbiakban az itt definiált azonosítóra hivatkozhasunk.
- *App var [params]*: Ez az alkalmazások azon formája, amely már több paramétert is tartalmazhat, mert a lambda-kalkulus egyparaméterű alkalmazásait már egyszerűen átalakítottuk többparaméterű függvényhívásra. Itt két eset lehetséges:
 - Elemi függvény esetén a fában új gyereket hozunk létre, ezeket ugyanis EOG műveletként kell megvalósítani.
 - Összetett függvény esetén behelyettesítést kell végeznünk. A kiértékelés következő lépése tehát a függvény definíciója lesz, amit a teljes eddigi, görgetett környezettel kell végrehajtanunk. Azért szükségesek a környezet legutóbb felvett elemei is (tehát a lokális vagy argumentum azonosítók), mert a bejárando lambda-absztrakció a paraméterein keresztül ezeket elérheti. A függvény definíciójának kiértékelésekor szükségesek az argumentumok is, ezért a [params] tömb elemeit átadjuk, mégpedig a lusta kiértékelés miatt feldolgozás nélkül.
- *Lam b expr*: Egy függvény definíciójának kiértékelése ezzel a csomóponttal kezdődik. Egy kivétellel minden esetben egy *App* csomópont előzte meg közvetlenül ennek a kiértékelését, ez a kivétel pedig a legfelső szintű *Lam*, amely paraméterein keresztül kommunikál a külvilággal. A csomópont kiértékelésénél első lépésként a paraméter-ill. argumentumlistából képzett azonosító-kifejezés párokat a környezethez hozzá kell adnunk, arra az esetre, ha a függvényen belüli csomópontokban hivatkozunk a paraméterekre. Ezt a környezetbővítést azért ilyen későn tesszük, mert a lokális névtérben szereplő paramétereket a *Lam* hozza létre, tehát a környezethez szükséges azonosító-nevek csak ettől kezdve állnak rendelkezésre.
- *Var id*: a csomópontnál az általa hivatkozott kifejezés értékét kell mögöttes jelentésként elképzelnünk. Ha a hivatkozott kifejezést még nem értékeltük ki (tehát az csak

egy thunk), akkor erre a pontra érve ezt meg kell tennünk, tehát az adott kifejezés-törzset ide kell másolnunk (behelyettesítés). Ha a kiértékelés már megtörtént, akkor az OpTree fát csak egy referenciával kell kibővítenünk, ami a már kiértékelt kifejezés-re (az OpTree már létező algráfjára) mutat. Háromfajta azonosítót különböztetünk meg:

- **globális azonosító:** Amit bármelyik függvény törzse elérhet.
- **lokális azonosító:** Egy *Let* hozta létre az azonosítót, amit a fa ez alatti elemei érnek el. Ha ilyen változóhoz érünk, akkor az azonosítóhoz rendelt kifejezés kiértékelésénél a környezet nem változik, hiszen egy lokális összerendelés nem mutat ki a lambda-absztrakcióból.
- **argumentum azonosító:** Egy *Lam* hozta létre, szintén a fa ez alatti elemei érik el. Ha ilyen típusú változó kerül kiértékelésre, akkor a környezet legutolsó szintjét eltávolítjuk (hiszen a *Lam*-on kívülről a *Lam* által hozzáadott azonosítók nem elérhetőek) és így folytatjuk a fabejárást.

Az algoritmus működését a 2.1. táblázattal demonstrálom.

<pre> a = lam t u app (+) var t var u b = lam aa xx yy app aa var xx var yy c = lam x let i = lit 1 let y = var i app b var a var x var y </pre>	<pre> 01. - 02. node:(lam ...) 03. node:(op +) 04. node:(var t) go:09 05. node:(var u) go:10 06. - 07. node:(lam ...) 08. node:(app aa) go:18 09. node:(var xx) go:19 10. node:(var yy) go:20 11. - 12. node:(lam x) 13. - go:15 14. node:(lit 1) finish 15. - go:17 16. node:(var i) go:14 17. node:(app b) go:07 18. node:(var a) go:02 19. node:(lnk IN) back:05 20. node:(var y) go:16 </pre>	<pre> - + [u=10,t=09] [yy..] [y..] [c..] - átad: ([yy..] [y..] [c..]) átad: ([yy..] [y..] [c..]) - + [yy=20,xx=19,aa=18] [y..] [c..] átad: ([yy..] [y..] [c..]) átad: ([y,i,x] [c,b,a]) átad: ([y,i,x] [c,b,a]) + [c=12,b=07,a=02] + [x=IN] [c,b,a] + [i=14,x] [c,b,a] - + [y=16,i,x] [c,b,a] átad: ([i] [c,b,a]) - átad: ([yy..] [y..] [c..]) - átad: ([i,x] [c,b,a]) </pre>
--	---	---

2.1. táblázat. *CoreToTree* algoritmus példán keresztül

Az első oszlopban az algoritmus bemeneteként kapott Core reprezentáció látható. A második oszlop az egyes Core elemekhez sorszámot rendel és a Core elemhez kötődő cselekvést tartalmazza. A *go:n* azt jelenti, hogy következő lépésben az n. sort kell kiértékelnünk, a *node:(nodeexpr)* pedig azt, hogy a kimeneti OpTree fában egy *nodeexpr* gyereket kell létrehoznunk. A harmadik oszlop a környezet mindenkorai állapotát tükrözi.

A *human* függvény szerepét helytakarékosági okokból itt a *c* nevű függvény tölti be. Ezek alapján kiindulópontunk a *c*-hez kötött kifejezés, vagyis a 12. sorszámmal jelölt Core elem. A kezdeti környezetben a globális változók (*[c, b, a]*) találhatóak, a program pedig soronként halad, kivéve, ha a *go* utasítás másképp nem rendelkezik.

A 14. és 16. sort átugorjuk, hiszen a call-by-need miatt ezeket nem értékelhetjük ki addig, amíg a végső érték meghatározása miatt ez elkerülhetetlenné nem válik. A 17. sorban az alkalmazás kiértékelése *app b* következik. Ez azt jelenti, hogy a következő OpTree bejegyzések a függvény törzséből (7. sor) fognak származni. Láthatjuk, hogy a 7. sorban a környezet az *yy*, *xx* és *aa* argumentumváltozókkal bővül. Az egyes azonosítókhoz rendelt értékek a jobb átláthatóság végett a hivatkozott kifejezés sorszámát mutatják. A 8. sorban az *aa* nevű paraméterben kapott függvény alkalmazása követhető látszik. A paraméter a 18. sorban kap értéket, de ez is csak egy hivatkozás a 2. sorban található *a* nevű függvény törzsére. A 3. sorban egy elemi függvényt találunk, így abból egy tényleges műveleti bejegyzés (*op +*) keletkezik az OpTree fában.

A *(+)* függvény első bemeneti paramétere többszörös indirekción (4. és 9. sorok) keresztül elér a 19. sorban található *lnk IN* csomópontig, ami végül egy levél elem lesz az OpTree fában. Ezt követően a *(+)* függvény második bemenete kerül kiértékelésre, ami szintén többszörös indirekción (5., 10., 20., 16. sorok) keresztül a 14. sor *lit 1* kifejezésénél ér véget (ebből szintén OpTree levél lesz), és ezzel a teljes Core bemenetet bejárta az algoritmus.

A kiértékelés eredménye, azaz a kimeneti OpTree fa végül a következő formát veszi fel:

```
lam x
  app b
    lam aa xx yy
      app aa
        var a
          lam t u
            op add
              var t
                var xx
                  var x
                    lnk IN
              var u
                var yy
                  var y
                    var i
                      lit 1
```

Az OpTree fa a Core kiértékelt változata. Szerepel benne a Core *Let* kifejezésein kívül az összes többi, így láthatjuk pl., hogy az összeadás művelet operandusai három ill. négy indirekción keresztül jutnak el az összeadáshoz (*t-xx-x* és *u-yy-y-i*). Az *op +* csomópont a CoreToTree kimenetén *op add* formában jelenik meg, mert a művelethalmaz FOpMap leképzése a *+* jelet *add*-ra módosítja. Az azonosítók egyszerűség kedvéért nem tartalmazzák a *unique* mezőt, így pl. az *add_o8_i4* helyett csak *add*, az *IN_1_a0* helyett pedig csak *IN* szerepel.

A *lam*, *app* és *var* elemek az EOG gráfban nem jelennek meg, ennek ellenére a hibakeresésnél nagyon fontos, hogy ezen a szinten még megmaradjanak.

2.2.6. Faegyszerűsítés

Az előző pontban kialakult fa több olyan csomópontot is tartalmaz, amelyeket a későbbiekben nem használunk, így ezeket meg kell szüntetnünk, ezt az egyszerűsítést a **TreeSimplifier** algoritmus végzi el.

Az algoritmus első lépése, hogy minden *App*, *Lam* és *Var* csomópontot eltüntetünk oly módon, hogy ezeket a gyerekekkel helyettesítjük. Ezt mindig meg tudjuk tenni, mert ezeknek a csomópontoknak kötelezően egyetlen gyereke kell, hogy legyen. Az algoritmus második lépése az, hogy minden egyes *Lit* csomópontot *const* típusú *Op* csomóponttá, azaz konstans műveletté alakít.

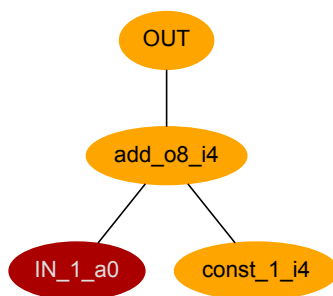
Elképzelhető, hogy egy konstans többször is felhasználásra kerül, pl. a *lit 0* adott esetben sok helyen szerepel. Lehetőség lenne arra, hogy a konstans csomópont csak egyszer szerepeljen, és minden azt felhasználó műveletnek ide huzalozzuk be a bemenetét, de ez fölösleges és adott esetben a hardver vezetékeinek számát növeli. Emiatt minden konstant csak egy művelet használ, ezért a TreeSimplifier az összes konstansból egyedi nevű *Op* csomópontot generál.

A következő kódrészleten jól látszik, hogy a CoreToTree algoritmusnál bemutatott bonyolult, magasabbrendű függvényeket is tartalmazó Core leírásunk a TreeSimplifier lefutása után jóval egyszerűbb formát vesz fel:

```
lam x
  add_o8_i4
    lnk IN_1_a0
    op const_1_i4
```

2.2.7. Elemi műveleti gráfra történő átalakítás

A kialakult OpTree fát egy inorder bejárással könnyedén EOG fájlformátumra hozhatjuk. Az előbbi példának megfelelő gráf a 2.11. ábrán látható.



2.11. ábra. OpTree

Az EOG első soraiban felsoroljuk az összes bemeneti, az utolsóban pedig a kimeneti

műveletet. A gráf inorder bejárása adja a be- és kimenetek közt az adatfolyamban található műveletek listáját.

Az inorder bejárás szerinti első pont az *IN_1_a0*, viszont ez csak egy link a bemenetre (ezt jelzi a piros szín), ezért ezzel nem kell semmit tenni. Következő lépés a *const_1_i4* levél, így ez kerül a bemenet utáni első sorba. Ezt követi az *add_o8_i4* csúcs, ami így a 2. sorba kerül, a sor végén pedig fel kell tüntetni bemenetként az *IN_1_a0* és *const_1_i4* gráfcsúcsokat.

Ezeket követve a Haskell-EOG modul kimeneteként a következő EOG leírást kapjuk:

IN_1_a0	"SYSTEM"	
const_1_i4	"const<1>"	
add_o8_i4	"add"	IN_1_a0 const_1_i4
OUT	"SYSTEM"	add_o8_i4

2.3. Elemi műveleti gráf fordítása hardverleíró nyelvre

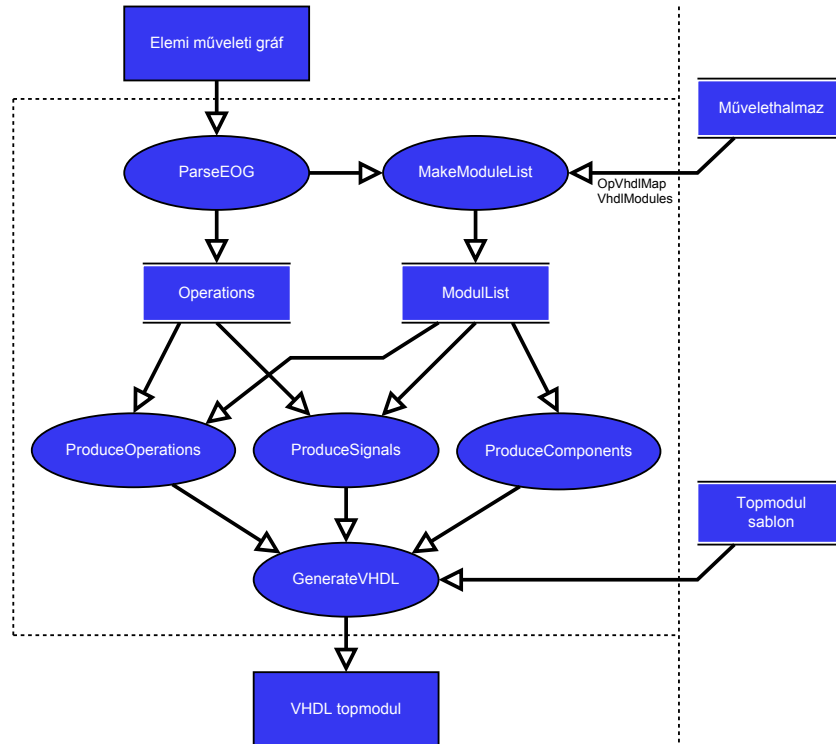
A fejezet során olyan módszer kerül bemutatásra, amely a Haskell-EOG modul kimenetéből, azaz az EOG formátumú adatfolyam leírásból és a művelethalmazból előállítja a VHDL leírást. Ezt a leírást a piacon kapható FPGA tervező szoftverek fel tudják használni konkrét FPGA szintetizálására, aminek eredményeképp a magas szinten megfogalmazott alkalmazást végül ténylegesen hardverként implementálhatjuk.

A fordító egyetlen VHDL modult szolgáltat kimenetként, ami a művelethalmazban definiált és a konkrét algoritmusban felhasznált műveleteket példányosítja, a közöttük szükséges adatutakat pedig kialakítja.

Az EOG-VHDL fordítómodul belső architektúráját a 2.12. ábrán vázoltam.

Az egyes almodulok jelentése:

- **ParseEOG**: feldolgozza az elemi műveleti gráfot, és egy belső struktúrában (*Operations*) eltárolja.
- **MakeModuleList**: összegyűjti az összes, EOG-ben megjelenő műveletet, és a hozzájuk tartozó VHDL modulokat feldolgozza. A feldolgozás eredményeképp létrehozza a *ModuleList* strukturát, ami az egyes műveletekhez tartozó legfontosabb paramétereket tárolja. (latency, port és generic deklarációk)
- **ProduceComponents**: a ModuleList-ből VHDL *component* szekciókat generál. Ezek a szekciók a topmodulban deklarálják a felhasználni kívánt modulokat, hogy azokat a főmodul törzsében példányosítani lehessen.
- **ProduceSignals**: minden művelethez létrehozza a hozzá tartozó kimeneti vezetékeket, egy adatvezeték és egy vezérlő vezetéket. Az előbbi a művelet kimeneti adatcsatornáját, az utóbbi pedig az adott csomóponthoz tartozó kimeneti vezérlőbitet fogja reprezentálni. Az adatvezeték bitszámát a művelethez tartozó, ModuleList-ben eltárolt deklarációk (port és generic) és a példányosításnál megadott sablonparaméterek határozzák meg.



2.12. ábra. Az EOG-VHDL modul architektúrája

- **ProduceOperations:** minden művelethez létrehozza az adott típusnak megfelelő modulpéldányosítást. Példányosításkor az EOG típusnevekben szereplő sablonparamétereket használja a *generic map*-ben, ill. a ProduceComponents által létrehozott köztes vezetékeket és a ki- bemeneti portokat használja a port map-ben.
- **GenerateVHDL:** a 2.13. ábrán bemutatott topmodul sablonfájl alapján létrehozza a végső, kimeneti topmodult, azaz a sablonba beilleszti a hiányzó VHDL szekciókat. (a kódban a hiányzó szekciókat `%input%`, `%output%`, `%components%`, `%signals%` és `%operations%` sablonparaméterként jelzem).

2.4. Alap művelethalmaz definiálása

A Haskell-VHDL fordítóprogram az előző fejezetekben leírt algoritmusok alapján elkészült. A megengedett forrásnyelvi függvények és típusok listáját eddig a pontig nem definiáltam, tettem ezt azért, mert a változtatható művelethalmaz bevezetésével a fordítóprogram és a fordítható elemi függvények listája szétválk egymástól. A most következő fejezetben bepótolom a hiányosságot, és teljeskörűen definiálok egy alap művelethalmazt.

2.4.1. A művelethalmazok kötelező elemei

A Haskell-EOG fordító algoritmusai bevezetnek olyan műveleteket, amelyek nem a forráskódban megadott elemi függvények megfelelői, hanem a Haskell forráskód bizonyos kifejezései miatt kerülnek az EOG interfészre. Ilyen kifejezés a *case* szerkezet, amelyet a CaseReduction algoritmus három különböző műveletre alakít (*select*, *ifcon*, *mux*),

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.defs.all;

entity benchmark is
  port (restart, clk: in std_logic;
        %input% : in std_logic_vector (31 downto 0);
        Output   : out std_logic_vector (95 downto 0);
        c%input% : in std_logic;
        cOutput  : out std_logic
  );
end benchmark;

architecture Behavioral of benchmark is
  %components%
  %signals%

begin
  %operations%
  Output <= %output%;
  cOutput <= c%output%;
end Behavioral;

```

2.13. ábra. VHDL topmodul sablon

ezenkívül a konstansok (*const*) és a Core alkalmazásként megvalósuló adatkonstruktorok (*dcon*). Az *IterationConversion* algoritmus az *iterate* függvénnyel operál, így ez is része a kötelező elemeknek. Ezeknek a műveleteknek szükségképpen minden művelethalmaznak elemei kell, hogy legyenek, hiszen bármelyik Haskell forráskód tartalmazhatja ezeket a kifejezéseket. A Haskell-EOG fordító öt különböző műveletet vezet be, ezek listáját a 2.2. ábrán soroltam fel.

<i>const</i>	Const<width,value>	konstans
<i>dcon1</i>	DCon1<cw,c,v1>	1 param. ADT konstruálás
<i>dcon2</i>	DCon2<cw,c,v1,v2>	2 param. ADT konstruálás
<i>dcon3</i>	DCon3<cw,c,v1,v2,v3>	3 param. ADT konstruálás
<i>select</i>	Select<inw,from,outw>	ADT adatleválasztás
<i>ifcon</i>	Ifcon<width,cw,c>	case alternatíva
<i>mux2</i>	Mux2<width>	case multiplex (2 irány)
<i>mux3</i>	Mux3<width>	case multiplex (3 irány)
<i>iterate</i>	Iterate<width>	iteráció

2.2. táblázat. Kötelező művelethalmaz elemek (az oszlopok rendre: bevezetett függvénytípus, VHDL modulnév sablonparaméterekkel és a művelet leírása)

Az egyes műveletek és paramétereik jelentése a következő:

- *const*: olyan művelet, amelynek nincs bemeneti paramétere, így a kimenetén mindig egy konstans értéket közvetít
 - *width*: a konstans bitszélessége
 - *value*: a konstans értéke
- *dcon1..dcon3*: létrehoz egy ADT adatot megadott konstruktorral és paraméterekkel (data constructor)
 - *cw*: a konstruktor bitszélessége
 - *c*: a konstruktor értéke
 - *v1..v3*: a konstruktor utáni paraméterek (számuk attól függ, hogy az adott konstruktornak hány paramétere van)
- *select*: egy ADT adatútról leválasztja az egyik konstruktor utáni értéket
 - *inw*: a bemeneti bitszélesség
 - *from*: az első olyan bit sorszáma, amely szerepelni fog a kimeneten
 - *outw*: a kimeneti bitszélesség
- *ifcon*: megvizsgálja, hogy egy ADT adatúton a megfelelő konstruktorral érkezik-e az adat, ha a válasz igen, adattovábbítás történik
 - *width*: a bemeneti bitszélesség
 - *cw*: a konstruktor bitszélessége
 - *c*: a konstruktor értéke
- *mux2..mux3*: az elágazások utáni egyesítésért felel (az elágazások száma szerint különböző megvalósítás szükséges)
 - *width*: a ki- és bemeneti bitszélesség
- *iterate*: iterációt megvalósító művelet, ami a Haskell *iterate* függvényből, átalakítással jön létre.
 - *width*: a ki- és bemeneti bitszélesség

2.4.2. Egyszerű alpműveletek

Megvizsgáltam a GHC 6.12.2 verziójának *Prelude* alapkönyvtárát [5], és ezt alapul véve meghatároztam azokat a műveleteket, amelyek a számtípusokon értelmesek és segítségükkel matematikai függvények széles köre írható le.

A számtípusok közül kizárom a *Rational* és az *Integer* használatát, ezek ugyanis rekurzív struktúrák, így azok implementálása a végtelen számok ábrázolásának elkerülése

érdekében nem valósulhat meg. A lebegőpontos típusok implementálása megvalósítható lenne létező, VHDL formában adott hardveres IP-kkel. Ezek erőforrásigénye kifejezetten nagy, viszont lényegében nem változtatják a nyelv kifejezőerejét, ugyanis kisebb pontosságot megengedve a műveletek fixpontos ábrázolással (így tehát egész típusokkal) is megvalósíthatóak. Az alap művelethalmazban ezért lebegőpontos műveleteket nem implementáltam.

A művelethalmazban a 32 bites *Int* számtípust fogom definiálni, ennek analógiájára bármilyen más fixpontos típus megvalósítható. A számtípus mellett nélkülözhetetlen a logikai típus, így a *Bool* ugyancsak bekerül a művelethalmaz típusai közé. Azok az ADT típusok, amelyek csak ezeket a típusokat tartalmazzák paraméterként, minden további nélkül használhatóak. (ilyenek pl. a *Maybe a* és *Either a b* paraméteres típusok)

A típusok meghatározása után fontos körülhatárolni az azokkal végezhető műveleteket. Fontos, hogy egy olyan műveletalmazt adjunk meg, amellyel minden értelmes művelet megvalósítható. A *Bool* típus szokásos műveletei a logikai és, vagy, nem, az *Int* típus aritmetikai műveleteit pedig az *Eq*, *Ord*, *Num* és *Integral* típusosztályok definiálják.

A művelethalmaz összes, kötelező műveleteken (2.4.1. fejezet) kívüli elemeit összefoglaltam a 2.3. táblázatban.

A művelethalmaz egyes elemeinek megvalósítása a következőképpen alakul. A *Num* típusosztály összes művelete a 2.1.3. fejezetben implementált (+) művelethez hasonlóan egyszerű kombinációs hálózattal triviálisan megvalósítható.

Az *Eq* és *Ord* típusosztályok műveletei sem okoznak problémát, ezek annyiban különböznek az előző csoporttól, hogy a kimenetük (*Min* és *Max* kivételével) *Bool* típusú lesz. (Az *Eq* típusosztálynál a *compare* műveletet nem szerepeltettem, mert ez a többi művelettel kiváltható)

A *Bool* típushoz tartozó ~~es~~, *//* és *not* műveletek két-két ill. egy db. bitből állítanak elő ugyancsak egybites adatkimenetet, ez szintén megvalósítható kombinációs hálózatként.

Külön bánásmódot érdemelnek az *Integral* típusosztály *quot*, *rem*, *div* és *mod* műveletei, ezeket ugyanis algoritmikus költségüknel fogva nem éri meg kombinációs hálózattal megvalósítani, ezeket ezért több órajelciklust igénylő VHDL modulokkal valósítom meg.

Bool műveletek:

$(\&\&) :: Bool \rightarrow Bool \rightarrow Bool$	<i>And</i>	logikai és
$() :: Bool \rightarrow Bool \rightarrow Bool$	<i>Or</i>	logikai vagy
$not :: Bool \rightarrow Bool$	<i>Not</i>	logikai nem

GHC.Classes.Eq osztály műveletei:

$(==) :: a \rightarrow a \rightarrow Bool$	<i>Eq<width></i>	egyenlő
$(/=) :: a \rightarrow a \rightarrow Bool$	<i>Ne<width></i>	nem egyenlő

GHC.Classes.Ord osztály műveletei:

$(<) :: a \rightarrow a \rightarrow Bool$	<i>Lt<width></i>	kisebb
$(<=) :: a \rightarrow a \rightarrow Bool$	<i>Le<width></i>	kisebb v. egyenlő
$(>) :: a \rightarrow a \rightarrow Bool$	<i>Gt<width></i>	nagyobb
$(>=) :: a \rightarrow a \rightarrow Bool$	<i>Ge<width></i>	nagyobb v. egyenlő
$min :: a \rightarrow a \rightarrow a$	<i>Min<width></i>	minimum
$max :: a \rightarrow a \rightarrow a$	<i>Max<width></i>	maximum

GHC.Classes.Num osztály műveletei:

$(+) :: a \rightarrow a \rightarrow a$	<i>Add<width></i>	összeadás
$(*) :: a \rightarrow a \rightarrow a$	<i>Mul<width></i>	szorzás
$(-) :: a \rightarrow a \rightarrow a$	<i>Sub<width></i>	kivonás
$negate :: a \rightarrow a$	<i>Neg<width></i>	negáció
$abs :: a \rightarrow a$	<i>Abs<width></i>	abszolút érték
$signum :: a \rightarrow a$	<i>Signum<width></i>	szignum

GHC.Real.Integral osztály műveletei:

$quot :: a \rightarrow a \rightarrow a$	<i>Quot<width></i>	osztás
$rem :: a \rightarrow a \rightarrow a$	<i>Rem<width></i>	maradék
$div :: a \rightarrow a \rightarrow a$	<i>Div<width></i>	osztás
$mod :: a \rightarrow a \rightarrow a$	<i>Mod<width></i>	maradék

2.3. táblázat. A művelethalmaz összes eleme (az oszlopok rendre: Haskell függvényfejléc, VHDL modul-név sablonparaméterrel, ill. a művelet szövegesen)

3. fejezet

A módszer kiterjesztése többsebességű (multi-rate) problémákra

A 2. fejezetben bemutatott módszer egyszerű, kizárólag egysebességű adatfolyamok Haskell nyelvű implementálására, és annak VHDL-be történő konvertálására jól használható, azonban **többsebességű** (multi-rate) [39, 27, 25] adatfolyamokat is tartalmazó Haskell forráskódokra nem alkalmazható.

A következő szakaszban egy egyszerű számítási képlet példáján keresztül megvilágítom, miért nem hagyhatjuk figyelmen kívül a többsebességű adatfolyamokat is tartalmazó problémákat. Ezt követően a használt adatfolyam modellt (EOG) és a fordítói módszert olyan kiterjesztésekkel látom el, amelyekkel a rendszer alkalmas lesz többsebességű adatfolyam-gráfok kezelésére is.

Adott a következő egyszerű feladat, miszerint ki kell számolnunk a bemenetként kapott x értékre a következő képletet:

$$g(x) + \sum_{i=0}^k f(x + i) \quad (3.1)$$

Itt f és g egy-egy általános, a 2.4. fejezetben definiált alpműveletekkel leírható függvény. (fontos megjegyezni, hogy ezek bármilyen függvények lehetnek, így pl. nem csak összeadás vagy szorzás, amely esetben a fenti képletből eltüntethető lenne a szumma)

Tekintsük úgy, hogy f és g egy-egy olyan művelet, amit az EOG reprezentációban a gráf csomópontjaként felhasználhatunk, és legyen $k = 1023$. A probléma fő forrása az, hogy a g függvényt egy adott x bemenetre egyszer kell kiértékelnünk, míg az f függvényt adott x bemenetre 1024-szer alkalmaznunk kell. Ez a képlet az eddig ismertetett EOG leírással csak úgy realizálható, ha az egyetlen g csomópont mellett létrehozunk 1024 db. f , és az azokhoz tartozó 1023 db. összeadó csomópontot. Egyértelműen látszik, hogy ez az eljárás megfelelően nagy k esetén erőforrásproblémák miatt hardverben kivitelezhetetlen, még ha a köztes reprezentációként szolgáló elemi műveleti gráf esetleg kezelhetetlen (de legalábbis

átláthatatlan) nagyságáról nem is ejtünk szót.

A megoldás az, hogy az f függvény fizikailag egyszer (esetleg kis számú egészszor, ezzel az esettel továbbfejlesztési lehetőségként foglalkozom) kerül megvalósításra, és egy x bemenet beérkezése esetén egymás után többször (k -szor) kerül alkalmazásra. Ezt az eddig használt EOG modell nem tudja reprezentálni, mert a gráf különböző műveleteire különböző újraindítási időkkel kellene számolnunk, így az alapfeltevések és az algoritmusok [21] nem használhatóak. A megoldás az EOG modell kiterjesztése olyan formára, ami már képes a többsebességű adatfolyamokat is kezelni.

A következőkben megvizsgálom a 2. fejezetben ismertetett interfészeket a többsebességű adatfolyamok megjelenése szempontjából, majd az interfészeket úgy módosítom, hogy az új feltételek is kielégíthetőek legyenek.

3.1. Többsebességű adatfolyamok a fordítás interfészein

A többsebességű adatfolyamok problémája a fordítási módszer mindhárom interfészén, a Haskell nyelvű forráskódban, a köztes adatfolyam reprezentációban és a VHDL hardverleírásban is megjelennek. A 3.1.1. fejezetben kiterjesztem az EOG modellt olyan formára, hogy az alkalmas legyen többsebességű adatfolyamgráfok leírására, a 3.1.2. fejezetben azt vizsgálom, hogy a Haskell nyelvű forráskódból miként következtethető ki minden műveletre vonatkozóan a kimeneti adatfolyamok sebességeinek eltérése, majd a 3.1.3. fejezetben a kimeneti VHDL leírás szükséges változtatásait taglalom.

3.1.1. Többsebességű elemi műveleti gráf

A 2.1.2. fejezetben röviden ismertetett EOG reprezentáció egy egysebességű adatfolyamgráf, és a modell többsebességű adatfolyamok esetén nem alkalmazható. A most következő fejezetben kiterjesztem az EOG modellt, ami ennek köszönhetően alkalmas lesz többsebességű adatfolyamok leírására is. A kiterjesztésre a továbbiakban **MREOG** (multi-rate EOG, azaz többsebességű elemi műveleti gráf) néven fogok hivatkozni. Ahol az egyértelműség miatt fontos, ott az egysebességű EOG-t SREOG (single-rate EOG) néven fogom használni.

Fontos cél, hogy az EOG modellhez kapcsolódó szabályok az MREOG-n is fennáljanak, így a hozzá kapcsolódó nagy számú algoritmusok az új modellen is alkalmazhatóak lesznek.

Az MREOG egy olyan adatfolyamgráf, amelyből egyértelműen megtudható, hogy melyek az azonos frekvenciájú jeleket továbbító adatfolyamok. (a frekvencia itt a megszokott értelmű, azaz az adatfolyamon érkező értékek egy másodpercre vetített darabszáma)

Az MREOG **blokk** egy olyan rekurzív struktúra, amely további blokkokat és műveleteket tartalmaz. Az utóbbiakra a blokk **saját műveletei** néven hivatkozunk. Egy blokk saját műveletei közül bármelyik kettő között kizárólag azonos frekvenciájú jeleket továbbító adatfolyamok szerepelhetnek és erre az azonos frekvenciára a **blokk sajátfrekvenciája** néven is hivatkozhatunk.

Az MREOG az előbbi definíciót használva egy egyetlen blokkból álló rekurzív struktúra, amely így az összes műveletet tartalmazza a köztük felépített adatfolyamokkal együtt. (az

egyetlen blokk persze tartalmazhat újabb blokkokat, ahogy az a definícióból kiderül)

Az így definiált blokkok hierarchiát alkotnak, ez pedig leírható egy fával, aminek gyökereleme a külső blokk, azaz amit már egyik blokk sem tartalmaz. (ez tulajdonképpen az az egyetlen blokk, ami a teljes MREOG-t jelenti) A fa levelei a műveletek, köztes csomópontjai pedig az egyes blokkok. Blokk nem lehet levél, hiszen ez azt jelentené, hogy a blokk nem tartalmaz saját műveletet és további blokkokat sem. A fa csomópontjai tehát az MREOG blokkjai és műveletei, a köztük futó éleket pedig a következő szabályokkal határozhatjuk meg.

A fában a B_i bloknak megfelelő csomópont a B_j bloknak megfelelő csomópont gyereke lesz, ha B_j közvetlenül tartalmazza B_i -t, azaz a feltétel halmazműveletekkel leírva (a halmaz elemei maguk a műveletek):

$$B_i \subset B_j \wedge \nexists k, k \neq i, k \neq j, B_i \subset B_k \subset B_j$$

A fában az o_i műveletnek megfelelő csomópont a B_j bloknak megfelelő csomópont gyereke lesz, ha o_i a B_j saját művelete, azaz a feltétel:

$$o_i \in B_j \wedge \nexists k, k \neq j, B_k \subset B_j, o_i \in B_k$$

Az így definiált hierarchia fát a továbbiakban segédgráfként alkalmazom, ami a műveletek vagy a blokkok bizonyos paramétereinek számítására lesz alkalmas. Mivel a fa az eredeti gráf adatfolyamait nem tartalmazza (élei nem adatfolyamok, hanem a hierarchia szinteket választják el egymástól), ezért VHDL generálásra nem is lenne alkalmas.

Fentiekből következik, hogy egy SREOG egyetlen blokkot tartalmaz, amelynek saját-frekvenciája megegyezik a bemenet (és ezzel együtt a kimenet) frekvenciájával, azaz minden egyes bemeneti jelre az összes művelet egyszer hajtódik végre. Egy MREOG esetében a modell több, egymásba ágyazott blokkot is tartalmaz. A gyökér blokk sajátműveleteire itt is igaz, hogy minden bemenetre egyszer hajtódnak végre, ezzel szemben a gyerek blokkok sajátműveletei bemenetenként n -szer hajtódnak végre, ahol $n > 1, n \in \mathbb{N}$.

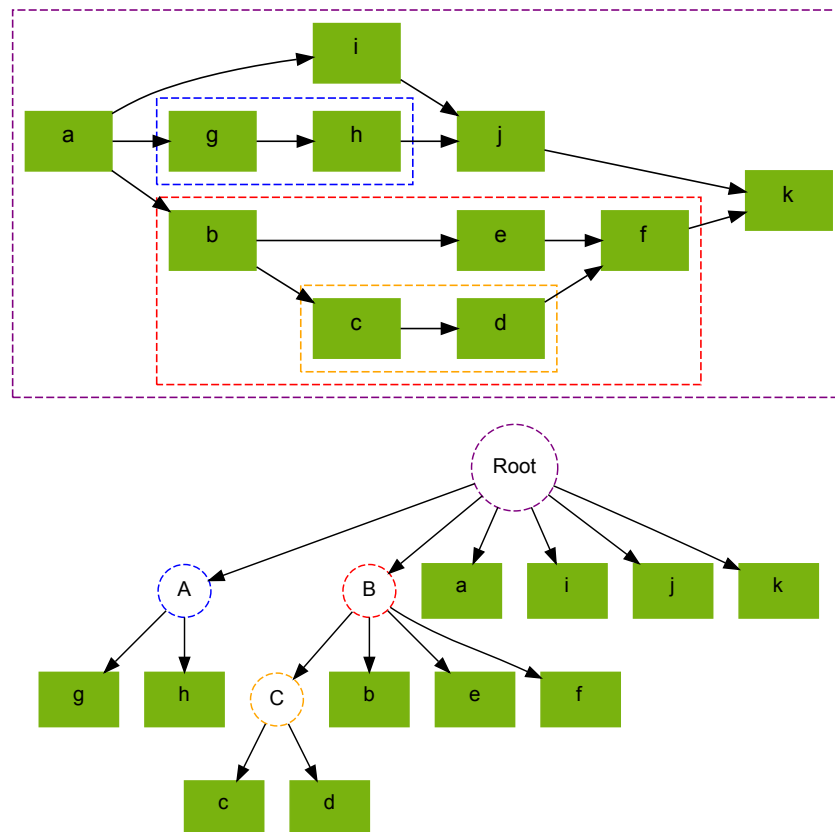
Minden blokk egyedi azonosítóval rendelkezik, és a gyökér blokk neve definíció szerint Root. Egy művelet hierarchiaszintje a gyökérelem és a művelet közötti úton található csomópontok száma lesz. (a gyökérelem és a művelet csomópontok nem számítanak, így egy két él hosszú út esetében a hierarchiaszint 1)

Az EOG-hez képest a grafikus megjelenés úgy módosul, hogy a hierarchia fát tekintve minden blokk esetében keretezéssel látjuk el a blokk gyerekeit, így tudjuk elválasztani azokat a többi művelettől és blokktól.

Az EOG szöveges leírása úgy módosul, hogy a típusmegadás utáni oszlopban listaalakban felsoroljuk a gyökértől az adott művelethez vezető úton található blokkok azonosítóját. A típus template paraméterének első értéke a hierarchiaszint lesz (ami nem más, mint az előbbi lista elemszáma egyel csökkentve). Egy egyszerű MREOG szöveges és grafikus ábrázolása és a hozzá tartozó hierarchia fa a 3.1. ábrán követhető nyomon.

Az MREOG definiálása után rátérek az EOG két fontos paraméterének számítására a

a	"a<0, 32>"	["Root"]	
b	"b<1, 32>"	["Root", "B"]	a
c	"c<2, 32>"	["Root", "B", "C"]	b
d	"d<2, 32>"	["Root", "B", "C"]	c
e	"e<1, 32>"	["Root", "B"]	b
f	"f<1, 32>"	["Root", "B"]	d e
g	"g<1, 32>"	["Root", "A"]	a
h	"h<1, 32>"	["Root", "A"]	g
i	"i<0, 32>"	["Root"]	a
j	"j<0, 32>"	["Root"]	i h
k	"k<0, 32>"	["Root"]	j f



3.1. ábra. Egyszerű MREOG szöveges és grafikus leírása, és a hozzá tartozó hierarchia gráf (az MREOG-ban és a hierarchia gráfban az egymásnak megfeleltethető blokkokat azonos színnel és szaggatott vonallal jelöltem)

többsebességű esetre vonatkozóan.

Újrarendelési- és lappangási idő számítása

SREOG esetében alapvető fontosságú az újrarendelési- és lappangási idők számítása [21]. Az **újrarendelési idő** (R) azt adja meg, hogy az adatfolyamgráf bemenetére milyen időközönként érkezik újabb feldolgozandó adat. A **lappangási idő** (L) azt adja meg, hogy a bemeneti mintavételezéshez képest mennyi idő múlva jelenik meg a kiszámított eredmény az adatfolyamgráf kimenetén. Fontos, hogy ezek számítása MREOG esetére is megoldható

legyen.

Egy MREOG csupán a hierarchia eltüntetésével („lapítás”) nem lesz alkalmas az R és L meghatározására, mert az eltérő adatfolyam-frekvenciák miatt a műveletek egyenként vett futási ideje (t_i) nem összemérhető.

Az MREOG azon B_i blokkjára, amely csak leveleket (vagyis csak műveleteket) tartalmaz, az R és L számítás a szokásos módon [21] elvégezhető, hiszen ez csak egysebességű adatfolyamokat tartalmaz. A B_i blokkra megkapott R_i és L_i értékekből kiszámítható a B_i blokkra, mint virtuális műveletre vett futási idő, azaz a t_i :

$$t_i = (n_i - 1) * R_i + L_i \quad (3.2)$$

A képletben szereplő n_i azt jelenti, hogy az B_i műveleteit n_i -szer futtatjuk le minden olyan bemenetre, ami a B_i -n kívülről jön, azaz közvetlen szülőjétől. Az n_i -t a B_i blokk rátájának nevezzük, ami azt mondja meg, hogy mialatt B_i szülőjében egyszer alkalmazzuk a sajátműveleteket, az alatt mennyiszor alkalmazzuk B_i sajátműveleteit. (informálisan: mennyivel „gyorsabb” a belső blokk, mint a külső)

A képlet azt jelenti, hogy az eredő t_i futási idő (azaz, hogy meddig foglalt a virtuális művelet) nem más, mint a B_i -re vett lappangási idő (triviális), hozzáadva ehhez azt az időt, ameddig a bemenetre az egymás utáni adatok érkeznek. Mivel a bemenetre R_i időnként érkezhet adat, és n_i az adatok darabszáma, ez az idő $(n_i - 1) * R_i$ lesz.

Ha t_i -t kiszámoltuk, akkor a B_i blokkot egyetlen virtuális művelettel helyettesíthetjük, amelynek futási ideje t_i lesz. Azért fontos a virtuális művelet fogalmát bevezetni, mert ez a művelet az eredeti adatfolyamgráfban nem található meg, ezt az algoritmus hozza létre. A helyettesítés után minden olyan adatfolyam megmarad, amely a B_i blokkon belüli és azon kívüli műveleteket köt össze, viszont a B_i blokkon belüli művelet helyett (annak hiányában) az új virtuális művelet lesz a végződése. (azaz az él egyik csúcsa az új virtuális művelet lesz)

Az iménti algoritmust iteratíván addig futtatjuk, míg végül a gyökér blokk is egyetlen virtuális műveletre redukálódik. Ez az a pont, ahol már minden blokkra megkaptuk annak újraindítási- és lappangási idejét. A gyökér blokkra vett R és L értékek lesznek az MREOG R és L értékei.

3.1.2. A bemeneti forráskód

A 3.1. képletet sokféleképpen leírhatjuk Haskell nyelven, három egyszerű megvalósítást a következő kódrészlet szemléltet.

```
megoldas1 x = g(x) + segedf 1023
    where segedf 0 = f x
           segedf i = f (x+i) + segedf (i-1)

megoldas2 x = g(x) + fst $ iterate segedf (0,0) !! 1024
    where segedf (s,i) = (s+f (x+i), i+1)

megoldas3 x = g(x) + sum $ map (\i -> f(x+i)) [0..1023]
```


Az első megoldás rekurzív függvényt használ, a második és a harmadik megoldás a rekurziót magasabb rendű függvények segítségével elrejt, és egyúttal bevezet egy-egy listát és az ahhoz szükséges műveleteket. (*sum*, *map*, (!!)) Mindhárom megoldásnál egyértelmű, hogy az x adott értékére az f függvény többször kerül alkalmazásra. Első esetben a rekurzióból következtethetünk erre, második és harmadik esetben amiatt, mert az *iterate*, ill. a *map* beágyazott függvényében használjuk f -et.

A rekurzív függvények közvetlen használatát kizártuk, ezért az első megoldást nem engedhetjük meg. A harmadik megoldásnak a másodikkal szemben a leírás egyszerűsége és az eredeti képlethez nagyon hasonló leírásmód az előnye, így a következő vizsgálatoknál ezzel az esettel foglalkozom tovább.

A $[0..1023]$ listamegadás egy szintaktikai édesítőszer, amely az *enumFromTo*(0, 1023) függvényhívásra képződik le. Ezek alapján a példamegoldásban négy függvényünk van, amelyek pontosan a négy lehetséges ki- és bemeneti típuskombinációval rendelkeznek:

- $(+)$: skalár bemenet és skalár kimenet, további példák lehetnének a 2.4. fejezetben bemutatott függvények (kivételez *iterate*)
- *enumFromTo* a b : skalár bemenet és lista kimenet
- *map* f xs : lista bemenet és lista kimenet, és a lista minden elemére lefut az f függvény
- *sum* xs : lista bemenet és skalár kimenet, további példák lehetnének a *product*, *maximum* és a *foldl*, ez utóbbi egy általános lista és skalár közötti konverter

A fejezet bevezetőjében már szó volt róla, hogy az f függvénynek megfelelő műveletet egyszer hozzuk létre és egymás után többször alkalmazzuk, azaz a bemeneti adatokat egyszerével, időben elválasztva adjuk át. Ezt a funkciót a Haskell megoldásban a *map* biztosítja, ami a bemeneti lista elemeivel egymás után hívja f -et. Ez azt jelenti, hogy a listát felfoghatjuk **stream**-ként, a *map*-nek megfelelő művelet pedig egészen egyszerűen a stream-en egymás után érkező elemeket egymás-után adja át f -nek.

Ehhez a gondolatmenethez tökéletesen kapcsolódik a Stream Fusion [28] eljárás, amelyet azonban más céllal hoztak létre. Az volt a cél, hogy az egymás utáni listaműveleteket (*map*, *zip*, *foldr*, stb.) a GHC a fordítás során optimalizálja, és egyetlen átalakított függvénybe tömörítse. Ezzel az egymás utáni listareprezentációk számát csökkentik, adott esetben akár a listák teljesen el is tűnnek az optimalizálás során, ami a processzorra fordítás esetében is nagyon kedvező hatású. Alkalmazva a Stream Fusion eljárást alapjait, fogalmi szinten áttérhetünk listákról stream-ek használatára.

A skalár bemenetű és lista kimenetű függvények **gyorsító** függvények, hiszen egyetlen értékből egy listát (több értéket, azaz stream-et) generálnak. A lista bemenetű és skalár kimenetű függvények **lassító** függvények, mert listát (több értéket, azaz stream-et) kapnak és ebből egyetlen értéket adnak vissza.

A példamegoldásból a sebességszintek nagyon egyszerűen kivehetőek. Mivel a $g(x)$ a *map*-en kívül van, ezért az minden bemenetre egyszer hajtódik végre, az $f(x+i)$ függvényt

pedig a *map*-en belülre írtuk, így azt meg kell hívni a `[0..1023]` lista minden elemére, azaz a sebessége egy hierarchiaszinttel gyorsabb a *g*-nél. (az általánosan felírt *map f xs* függvényhívás esetében az *f* számít belülnek, hiszen ez a függvényparaméter)

Sajnos általános esetben a helyzet nem ilyen egyszerű, ennek demonstrálására szolgál a következő kódrészlet.

```
maptest :: Int -> Int
maptest x =
  let belso j =
    let a = sum $ map (\i -> i+5) [0..4] :: Int
        b = sum $ map (\i -> i+j+5) [0..5] :: Int
        c = sum $ map (\i -> i+x+5) [0..6] :: Int
        d = sum $ map (\i -> i+j+x+5) [0..7] :: Int
    in (a,b,c,d)
  in sum $ map belso [0..3]
```

A példában egy 0 – 3 intervallumban mozgó külső ciklus (ciklusváltozója: *j*), és az azon belül helyet foglaló négy darab belső ciklus (ciklusváltozók: *i*) található. Látható, hogy bár az *a* változó kiszámítása a külső cikluson belül található, értékét elég a program futása során egyszer kiszámítani, ugyanis nem függ a bemenettől. A *c* változó már függ a bemenettől, viszont a külső ciklus változójától, *j*-től nem, így elég minden *x* bemenetre egyszer kiszámítani értékét. Adott esetben *a* vagy *c* kiszámítása nagyon hosszantartó művelet is lehet, így kifejezetten hátrányos, ha ezeket a műveleteket *j*-szer hajtjuk végre az elégséges egyszeri végrehajtás helyett.

Az *a* vagy a *c* értéke paraméterként a külső cikluson kívülről is megkapható lenne, és ilyen esetben a fordító az inlining-nak köszönhetően szintén a *belso* függvényen belülre helyezné ezt a kifejezést. Ebből sajnos az következik, hogy nem elég azt vizsgálni, hogy az egymásba ágyazott *map* függvényekben milyen mélyen található egy művelet.

Egy másik megoldás lehet, ha minden műveletnél megvizsgáljuk, hogy operandusai milyen változóktól függnek. Ezek a változók a ciklusváltozók és az egyetlen bemeneti változó. Egy kizárólag konstans operandusokkal rendelkező összeadás művelet eredménye pl. konstans, ellenben ha az egyik operandus az *x* bemeneti változótól függ, akkor az összeadás eredménye is *x*-től fog függni, és így tovább. Ha egy művelet függ a belső *j* ciklusváltozótól, és a *j* függ a külső *i* ciklusváltozótól, akkor a művelet mindkét ciklusváltozótól függni fog, tehát a függés tranzitív.

Ez utóbbi számítás már megfelel elvárásainknak, mert csak akkor végez el egy műveletet, ha a művelet bemenetei változtak, és így a műveletek felesleges használatát kiszűrtük. Ismét megjegyzem, ez az inlining miatt egy kulcsfontosságú kérdés.

3.1.3. Kimeneti hardverleírás

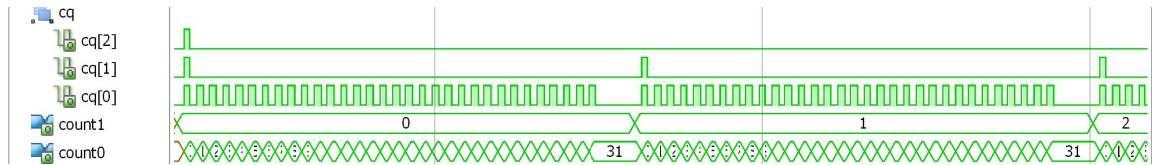
A VHDL leírásban az egyes műveleteket megtestesítő modulok a control bitek hatására lépnek működésbe. MREOG esetében ez az egyszerű vezérlés nem elegendő, ehhez pedig igazolásként elég csak a *sum* művelet működésére gondolni. A *sum* függvény Haskell-ben a lista elemeinek összegét számolja ki, azaz az MREOG *sum* művelete a stream elemeire végzi

ezt. A *sum* VHDL modul a bemenetére érkező értékeket összegzi, de ezt csak addig szabad tennie, amíg egy adott stream-en belüli értékek érkeznek. Amint új stream kezdődik, a szummázást ismét nulláról kell kezdenünk, hiszen a *sum* a stream elemeit adja össze, nem a végtelenségig érkező adatokat.

Célszerű tehát bevezetni egy második control bitet, ami nem az adott skalár érték mintavételezhetőségét jelzi, hanem a skalárokat tartalmazó stream mintavételezhetőségét, azaz a stream kezdetét. Ez a control bit pontosan akkor lesz aktív, amikor a stream első elemének (skalárjának) control bitje aktív.

Egy olyan stream-nél, amelynek minden eleme skalárokat tartalmazó stream, be kell vezetnünk egy harmadik control bitet, hiszen egy *sum* modul összegzi a belső stream-eket, de ennek eredménye skalárokat tartalmazó stream, amit egyébként ismét szummázhatunk. Mindebből tehát az következik, hogy minden MREOG műveleti csomóponthoz a hierarchiaszintjével azonos bitszélességű vektort kell rendelnünk vezérlőjel gyanánt.

A 3.2. ábra egy kétszintű stream vezérlőjeleinek hullámformáját mutatja. A *cq[2]* jel a bemenet mintavételezése, a *cq[1]* magas szintje az egyes bemenetre előálló streamen érkező adatok mintavételezése, de mivel a stream-en belüli adatok szintén stream-ek, ezért megjelenik egy harmadik szint is. A *cq[0]* jel a belső streamen érkező adatok mintavételezését jelenti. Általánosan elmondható, hogy a vezérlő vektor 0. bitje mindig a leggyorsabb mintavételezést fogja jelenteni, ugyanis így biztosítható egyszerűen, hogy a skalár-skalár típusú műveletek mindig a legbelső stream adatain operáljanak.



3.2. ábra. Vezérlő bitek stream esetén

Mivel a vezérlőbit helyett a vezérléshez bitvektort használunk, és a bitvektor szélessége a hierarchiaszinttől függ, ezért azt a VHDL modulok generikus paramétereként célszerű átadnunk. A generikus paramétereket a 2. fejezetben leírt módszer is támogatja, így ez nem okoz plusz munkát. Innentől kezdve minden VHDL modul első generikus paramétere a hierarchiában elfoglalt szintet fogja jelezni.

3.2. A fordítási algoritmus kiterjesztése

Az interfészek vizsgálata és módosítása után rátérek a fordító azon módosításaira, amelyek lehetővé teszik a heterogén sebességű műveletek használatát. A Haskell-EOG fordítóban ehhez két új algoritmust kellett bevezetnem, egyik a 3.2.1. fejezetben bemutatásra kerülő MapConversion, a másikat pedig a 3.2.2. fejezetben mutatom be, és a műveletek hierarchiában elfoglalt helyét határozza meg. Az EOG-VHDL fordító csak minimális módosításra szorul, itt a vezérlő *signal*-ok *std_logic*-ról (egy bites jel) *std_logic_vector*-ra (több bites vektor) való átírása elegendő a sikerhez, így ezt a későbbiekben nem fejtem ki bővebben.

3.2.1. MapConversion transzformáció

A *map* magasabbrendű függvény definíciója:

```
map :: (a -> b) -> [a] -> [b]
map - []          = []
map f (x:xs) = f x : map f xs
```

A *map* függvény első paramétere egy egyparámeteres *f* függvény, amely a második paraméterben megadott lista összes elemére meghívódik. Az *f* visszatérési értéke a második paraméterrel megegyező elemszámú lista azonos helyére kerül.

A **MapConversion** transzformációs lépés nagyon hasonló a 2.2.3. fejezetben bemutatott IterateConversion-höz, ez is egy egyszerű Core2Core transzformáció, amit a 3.3. ábrán a már megszokott formalizmussal ismertettek.

<pre>01: app map 02: lam x 03: E 04: E2</pre>	<pre>01: let x = 02: E2 03: note "lam x" 04: E</pre>
---	--

3.3. ábra. MapConversion általános képlet

Mint látható, a transzformáció nagyon egyszerű, és tulajdonképpen a *map* függvény eltűntetéséről szól. A *map* kimenetét az *E* kifejezés, azaz a belső lambda absztrakció adja. Azokra a pontokra, ahol a belső lambda absztrakció paraméterét (*x*-et) felhasználjuk az *E2* kifejezés kerül, vagyis a második paraméterként megadott lista. (ami az MREOG nyelvezet szerint már stream) Ezt a működést az újonnan bevezetett *let x =* csomópont valósítja meg. Fontos, hogy az *E* kifejezés gyökerét egy *note* csomóponttal megjelöljük, mert az OpTree fában ez a csomópont fogja jelezni az eltűntetett *map* belső lambda absztrakciójának kezdetét.

A Haskell nyelv szempontjából a Core a transzformáció után nem marad típuskonzisztens, ugyanis az *E2*-nek a típusa lista, az *E*-n belül felhasznált *x* pedig listaelem típusú, viszont *x* itt magát a listát, azaz *E2*-t kapja meg. Az ilyen értelemben vett típusinkonzisztencia nem okoz problémát, hiszen a gráfból MREOG reprezentáció készül, ahol a stream és a skalár csak a hierarchiában elfoglalt szint vonatkozásában tér el egymástól, egyébként a kettő típusazonos.

A MapConversion transzformációt a 2.6. ábrán vázolt Haskell-EOG pipeline IterateConversion és AppRename transzformációi közé célszerű beékelnünk.

3.2.2. A hierarchia automatikus felépítése

A többsebességű adatfolyamokat is támogató fordítás leglényegesebb pontjához érkeztünk. Most következik annak az algoritmusnak a leírása, amely minden műveletre, azaz az OpTree minden csomópontjára meghatározza az MREOG hierarchiában elfoglalt helyét, azaz felépíti magát az MREOG-t. Az algoritmust a 2.6. ábrán látható fordítói pipeline

CoreToTree és TreeSimplifier átalakításai közé kell beékelnünk, mert ez az a pont, ahol a Core-ból már OpTree reprezentáció lett, és a szükséges segéd csomópontok (mint pl. a *note "lam x"*) még nem tűntek el.

Az algoritmus négy lépésből áll, és mindegyik az OpTree fát járja be $O(n)$ költséggel. A lépések a következők:

1. Ciklusváltozók sorszámozása: az OpTree preorder bejárása során a MapConversion által létrehozott *Note lam* csomópontok esetében a kapcsolódó változó (a MapConversion transzformációnál látott példában ez *x* volt) neve elé egy sorszámot írunk, ami az egymásba ágyazás során egyre magasabb értéket vesz fel. Erre azért van szükség, hogy a blokk azonosítók sorrendhelyesen (először a gyökérelem, majd annak gyereke, stb.) kerülhessenek felsorolásra.
2. Műveletek változófüggésének meghatározása: az OpTree postorder bejárásánál minden node-ra megállapítjuk, hogy az mely változóktól függ, azaz a node alatti részfák mely változók értékeit használják fel. A postorder bejárás biztosítja, hogy először a levelekre értékelődjön ki a függőségvizsgálat, így minden node számításakor csak a közvetlenül alatta lévő node-ok függőségeit kell figyelembe venni.
3. Változófüggőségek tranzitív kiterjesztése: ha egy node az *i* ciklusváltozótól függ, és az *i* ciklusváltozó pedig a *j* ciklusváltozótól, akkor a node is függ *j*-től. Az OpTree-t bejárjuk preorder módon, és minden *Note lam* csomópontnál a hozzá tartozó változót és függőségeit feljegyezzük, majd a gyerekeinek a bejárásakor ezt átadjuk. Ha egy művelet *i*-től függ, akkor a függőségi halmazhoz hozzávesszük az *i* változóhoz rendelt függőségeket is. Ezt könnyen megtehetjük, mert a *Note lam i* csomópontnál *i* függőségeit feljegyeztük, és paraméterként átadtuk a gyerekeknek.
4. Hierarchiaszintek meghatározása: preorder bejárásnál minden node-ra megszámláljuk az előbbieken már feltöltött függőségi lista elemeinek számát, ez lesz maga a hierarchiaszint. Az üres függőségi lista azt jelenti, hogy az adott művelet eredménye konstans. Ilyen esetben a konstans szinkronizálni kell az azt felhasználó nem konstans művelettel, azaz a művelet függőségi listáját a konstansra átmásoljuk. Ez azért nagyon fontos, mert találkozhatunk stream konstansokkal is, amelyek a gyakorlatban nem konstansok, hiszen az adatokat egymás után felsorolva küldik. Ilyen esetben a felsorolást végző műveletet (rendszerint *EnumFromTo*) a blokkjának megfelelő be-menettel triggerelni kell. Ha ez nem történik meg, az *EnumFromTo* műveletnek nem adjuk át az információt, hogy mikor kell elkezdenie a felsorolást, így az egymás után többször semmiképp nem tud végrehajtódni. (ami pedig követelmény lenne)

Az algoritmus lefutása után minden node-hoz ismert lesz a hozzá tartozó hierarchiaszint és a tartalmazó blokkok gyökérellemmel kezdődő listája, az azonosítóikkal megadva.

Az EOG szöveges reprezentációjának elkészítésekor a művelet első generikus paramétere a hierarchiaszint lesz, a harmadik oszlopba pedig a blokk azonosítók kerülnek sorrendhelyesen. (a ciklusváltozók sorszámozása miatt ez alfanumerikus sorrendet jelent)

3.3. A művelethalmaz kivívítése

A fejezet eddigi részében a fordító módszerének kibővítésével foglalkoztam, hogy a rendszer alkalmas legyen többsebességű adatfolyamok kezelésére is. A módszert implementáltam, a fordítóprogram most már képes az MREOG-ben reprezentálható algoritmusok Haskell-ből VHDL-be történő fordítására. Egy dolog maradt hátra, ez pedig a konkrét műveletek definiálása és VHDL-ben történő leírása, hogy a fordító új képességeit ténylegesen ki tudjuk használni Haskell nyelven írt forráskódokkal. A következő két alfejezet során a művelethalmazt kibővíttem a stream-, ill. a tömbkezelő műveletekkel.

3.3.1. Stream műveletek

A stream műveletek a Haskell lista kezelő függvényeiből adódnak. A továbbiakban a stream-ek fogalmánál maradok, ezért fontos megjegyezni, hogy a forráskód szintjén ezek még listaként szerepelnek. (ezt a műveletek tárgyalásánál érdemes mindi észben tartani)

A 3.1.2. alfejezetben láttuk, hogy a 3.1. képlet számítása három listaműveleten alapszik: *enumFromTo*, *map* és *sum*. Ez a három művelet már elégséges az MREOG létrejöttéhez, mert tartalmaz gyorsító (*enumFromTo*) és lassító (*sum*) műveletet is. Mivel minden lassító művelet alapjaiban megegyezik, ezért most példaként csak a *sum*-mal foglalkozom. Ennek analógiájára nagyon könnyen áttérhetnénk a *product* vagy a *maximum*, vagy egyéb lassító függvényre is. A gyorsító függvényeknél ugyan ez a helyzet.

A *Map*-et a MapConversion transzformáció eliminálta, így csak az *EnumFromTo*, és *Sum* modulok VHDL leírásáról kell szót ejtenem.

Az *EnumFromTo* tulajdonképpen úgy viselkedik, mint egy for ciklus. A bemenetén érkező intervallumhatárokon belül lépteti a kimenetére kötött számláló értékét, a generikus paraméterként megadott késleltetési időközönként. Ez az időköz a blokkra alkalmazott, R újraindítási idő kiszámítása során adódik. Ha a B_i blokk újraindítási ideje R_i , akkor a B_i -ben található EnumFromTo késleltetési időköze pontosan az R_i érték lesz. Ez egyszerűen abból adódik, hogy az EnumFromTo-nak két érték között ki kell várnia az utána következő műveletek minimális újraindítási idejét, hogy azok a műveletek rendben le tudjanak futni.

A *Sum* a bemenetére érkező stream értékeit összegzi. A stream kezdetét jelző második szintű vezérlőbit a modulban található *accum* regisztert nullázza, így minden stream kezdetén a számlálás nulláról indul.

A modulok egy-egy lehetséges VHDL kódja a függelékben megtalálható.

3.3.2. Tömb műveletek

A stream-ekkel és műveleteikkel kapcsolatban egy fontos dologról még nem esett szó. Haskell-ben könnyen leírhatunk olyan algoritmust, ami először előállít egy listát, majd ezen a listán többször végigmegy. Egy egyszerű példa, ha a lista maximumát hozzá kell adnunk a lista összes eleméhez. Nyilvánvaló, hogy először ki kell választanunk a maximális elemet (első bejárás), és csak ezután kezdődhet az összeadás (második bejárás).

Az előbb felvázolt lista a fordítás során stream-ként kerül létrehozásra, ami azt jelenti, hogy a lista elemei egymás után, és csak egyszer jelennek meg a stream-et megtestesítő

adatfolyamon. A stream elemeit tehát nem soroljuk fel kétszer, így a fent vázolt algoritmus nem tud működni. A problémára két főbb megoldás létezhet. A stream előállításáért felelős műveleteket esetleg rá lehet venni, hogy ismételten kiszámolják a stream elemeit, így az újra felsorolható. Második esetben az egyszer kiadott stream-et memóriában eltároljuk, és legközelebb a memóriából olvassuk vissza stream-ként, így ameddig a memóriában fellelhető az elmentett stream, azon bármennyiszer végig tudunk menni. Az első eset az eddigi műveletekkel megoldható, de egyes esetekben előfordulhat, hogy a stream újragenerálása csak nagyon költségesen lenne kivitelezhető, és így csak a második megoldás kerülhet előtérbe.

Ebben az alfejezetben a második megoldással foglalkozom, vagyis azzal, hogy hogyan tudunk egy stream-et elmenteni, majd újra felolvasni. Ezt kidolgozva más előnyökhöz is jutunk, mégpedig, hogy a letárolt lista adott indexű eleméhez a memóriában való tárolás miatt $O(1)$ idő alatt hozzá tudunk férni, szemben a lista vagy a stream esetében, ahol ez az idő $O(n)$.

Haskell-ben létezik tömböt megvalósító adatstruktúra, amelynek definíciója a következőképp néz ki:

```
data (Ix a) => Array a b = MkArray (a,a) (a -> b) deriving ()
```

Egy tömböt ezek szerint megadhatunk egy index intervallummal, és egy olyan függvénnyel, ami az indexből értékre képez le. A működésbe ennél részletesebben nem kell belemélyednünk, helyette az *Array*-el végezhető alapl műveletekre érdemes koncentrálni:

- *listArray*: Array létrehozás, és az elemek feltöltése kezdeti értékkel
- *(//)*: adott indexű elem módosítása
- *(!)*: adott indexű elem lekérése

Ezzel a három művelettel a felvázolt probléma már megoldható. A *listArray*-t nem tekintem műveletnek, mert a létrehozás fordítási időben, statikusan működhet. (az elosztottan használt memóriák és a memóriaallokálás kérdése túlmutat jelen dolgozat határain) Az elemek kezdeti értékkel való feltöltését pedig szintén statikusan fordítási időben, vagy dinamikusan a *(//)* művelet használatával meg tudjuk valósítani.

Minden *listArray*-hez tehát egy-egy block RAM-ot kell létrehoznunk fordítási időben, amit utána az olvasási és írási műveletekkel futási időben is kezelhetünk.

A *(!)* operátor, vagyis a VHDL szinten *nth*-nak hívott modul a bemeneti értékét a block RAM címvonalára küldi, a block RAM által a kimeneti adatvonalon szolgáltatott érték pedig az *nth* kimenetére fog kerülni egy órajel eltolással. (a block RAM szintén a rendszer órajelére van kötve, ez szinkron hálózat esetében indokolt is)

A *(//)* operátor listát vár, vagyis VHDL megfelelője, a *wr* nevű modul stream-et. A stream-en érkező adatok párosával érkeznek (2-es tuple-ként), aminek egyik eleme az indexet jelenti, másik eleme pedig az index által mutatott helyre beírandó értéket. A művelet megvalósítása szintén egyszerű, a *wr* modul az indexet a block RAM címvonalára,

az értéket pedig a bemeneti adatvonalára köti, így a következő órajelre az érték bekerül a block RAM index által meghatározott címére.

A modulok egy-egy lehetséges VHDL kódja a függelékben megtalálható.

4. fejezet

A módszer gyakorlati alkalmazása mintafeladatokon

A 2. és a 3. fejezetekben általam kidolgozott módszert Haskell nyelven írt fordítóprogramként implementáltam, melynek a fordítás során keletkezett naplója a függelékben megtekinthető. Az elkészült fordítóprogramot egy PID szabályozó és egy MP3 dekódoló algoritmusrészleten teszteltem, jelen fejezetben a teszt eredményeit mutatom be.

4.1. PID szabályozó

Az elkészült fordítóprogramot Csák Bence Ph.D. értekezésében [24] példaként használt PID szabályozó algoritmussal teszteltem.

Az értekezés szerinti C nyelven írt kód a 4.1. ábrán olvasható. Ennek Haskell nyelvi változatát elkészítettem, amit a 4.2. ábrán mutatok be. A *plant_algo* és *pid_algo* függvények mindkét megoldásban ugyan azt a belső működést valósítják meg. A legfőbb eltérés az, hogy a Haskell tiszta funkcionális nyelv, tehát a C verzió statikus változóit (*esum* és *e*) a függvény paramétereként és a visszatérésnél is szerepeltetni kell. A C példa for ciklusának belsejét a Haskell kód *iteration* függvényeként implementáltam, a for ciklus, tehát a teljes algoritmus pedig az *algo_main* függvénybe került.

Ahhoz, hogy a hardveres osztás minél egyszerűbben megvalósítható legyen, a 8-al való osztáshoz bevezettem egy *quot8* nevű függvényt, amit ezért a művelethalmazban is definiálnom kellett.

4.1.1. A fordítás lépései

A fordítóprogram első lépésként a Haskell forráskódból a GHC könyvtári függvényei segítségével előállítja a 4.3. ábrán látható Core reprezentációt. A CaseReduction algoritmus ebből a reprezentációból eliminál minden *case* kifejezést, az így kapott kódot pedig az IterateConversion alakítja át, ennek eredménye látható a 4.4. ábrán. A még mindig Core formájában létező leírás az AppRename és CoreToTree fázisokon keresztülhaladva a 4.5. ábrán szövegesen bemutatott gráfot eredményezi, aminek a TreeSimplifier algoritmus utáni, egyszerűsített változatát grafikusán a 4.6. ábrán szemléltetem. A ConvertToEOG al-

```

#include <stdio.h>
char plant_algo(char c, char y)
{
    y = (7*c + y) / 8;
}
char pid_algo(char x, char y)
{
    static const char cp=2;
    static const char ci=1;
    static const char cd=2;
    static char esum = 0;
    static char e = 0;
    char p,i,d, eold, ediff;

    eold = e;
    e = x - y;
    ediff = e - eold;
    esum += e;
    p = cp * e / 8;

    i = ci * esum / 8;
    d = cd * ediff / 8;

    return p + i + d; // c
}

int main()
{
    char c, y, i, x=8;
    y = 0;
    for (i=0; i<30; i++)
    {
        c = pid_algo(x, y);
        y = plant_algo(c, y);
        printf("%d\n", y);
    }

    return 0;
}

```

4.1. ábra. PID algoritmus C forráskódja

```

{-# LANGUAGE NoImplicitPrelude #-}
module Pid(hwmain) where
import InstructionSet

plant_algo :: Int -> Int -> Int
plant_algo c y = quot8 (7*c + y);

pid_algo :: Int -> Int -> (Int,Int) -> (Int, (Int,Int))
pid_algo x y (e,esum) =
    let eold = e
        e2 = x - y
        ediff = e2 - eold
        esum2 = esum + e2
        p = quot8 $ 2 * e2
        i = quot8 $ 1 * esum2
        d = quot8 $ 2 * ediff
    in (p+i+d, (e2,esum2))

iteration :: (Int,Int,Int) -> Int -> (Int,Int,Int)
iteration (e,esum,y) x =
    let (c, (e2,esum2)) = pid_algo x y (e,esum)
        y2 = plant_algo c y
    in (e2,esum2,y2)

algo_main :: Int -> [(Int,Int,Int)]
algo_main input = iterate (\x -> iteration x input) (0,0,0)

hwmain = algo_main

```

4.2. ábra. PID algoritmus Haskell forráskódja

goritmus ennek a gráfnak az preorder bejárásával adja a Haskell-EOG fordítómodul ki-
menetét, azaz a a 4.7. ábrán szövegesen, majd a 4.8. ábrán vizuálisan megjelenített elemi
műveleti gráfot.

Az EOG-VHDL fordítómodul az EOG-ből már közvetlenül VHDL leírást tud generálni,
az eredményt a 4.9. ábrán szemléltetem.

```

nrec quot8_reEp =
  app InstructionSet.quot8_re4n<Int> [1]
  var GHC.Real.$fIntegralInt_re0b = GHC.Real.Integral GHC.Types.Int

nrec quot81_reEt =
  app InstructionSet.quot8_re4n<Int> [1]
  var GHC.Real.$fIntegralInt_re0b = GHC.Real.Integral GHC.Types.Int

nrec Pid.hwmain_redC =
  lam input_aeBj
  app GHC.List.iterate_re4o<(Int, Int, Int)> [2]
  lam x_aeBk
  case
  var x_aeBk = (GHC.Types.Int, GHC.Types.Int, GHC.Types.Int)
  alt (,,) e esum y
  let nrec e2_seE1 =
    app GHC.Num._01F<Int> [3]
    var input_aeBj = GHC.Types.Int
    var y_aeBd = GHC.Types.Int
  let nrec esum2_seEk =
    app GHC.Num._rdZC<Int> [3]
    var esum_aeBc = GHC.Types.Int
    var e2_seE1 = GHC.Types.Int
  app GHC.Tuple.(,,)_77<Int, Int, Int> [3]
  var e2_seE1 = GHC.Types.Int
  var esum2_seEk = GHC.Types.Int
  app quot81_reEt◇ [1]
  app GHC.Num._rdZC<Int> [3]
  app GHC.Num._rdZD<Int> [3]
  app GHC.Types.I#_6d◇ [1]
  lit 7
  app GHC.Num._rdZC<Int> [3]
  app GHC.Num._rdZC<Int> [3]
  app quot8_reEp◇ [1]
  app GHC.Num._rdZD<Int> [3]
  app GHC.Types.I#_6d◇ [1]
  lit 2
  var e2_seE1 = GHC.Types.Int
  app quot8_reEp◇ [1]
  app GHC.Num._rdZD<Int> [3]
  app GHC.Types.I#_6d◇ [1]
  lit 1
  var esum2_seEk = GHC.Types.Int
  app quot8_reEp◇ [1]
  app GHC.Num._rdZD<Int> [3]
  app GHC.Types.I#_6d◇ [1]
  lit 2
  app GHC.Num._01F<Int> [3]
  var e2_seE1 = GHC.Types.Int
  var e_aeBb = GHC.Types.Int
  var y_aeBd = GHC.Types.Int
  app GHC.Tuple.(,,)_77<Int, Int, Int> [3]
  app GHC.Types.I#_6d◇ [1]
  lit 0
  app GHC.Types.I#_6d◇ [1]
  lit 0
  app GHC.Types.I#_6d◇ [1]
  lit 0

```

4.3. ábra. GHC által szolgáltatott Core kód

```

nrec Pid.hwmain_va =
  lam input_vb
    let nrec x_vc =
      app iterate_od◇ [2]
      var C_ou - #
      app GHC.Tuple.(, ,)_oe<Int, Int, Int> [3]
      app GHC.Types.I#_of◇ [1]
      lit 0
      app GHC.Types.I#_og◇ [1]
      lit 0
      app GHC.Types.I#_oh◇ [1]
      lit 0
    let nrec it_out_0_vi =
      let nrec scrutinee_0_vj =
        var x_vc - (GHC.Types.Int, GHC.Types.Int, GHC.Types.Int)
      let nrec e_vk =
        app select <(, ,), 0>_ol◇ [1]
        var scrutinee_0_vj - #
      let nrec esum_vm =
        app select <(, ,), 1>_on◇ [1]
        var scrutinee_0_vj - #
      let nrec y_vo =
        app select <(, ,), 2>_op◇ [1]
        var scrutinee_0_vj - #
      let nrec e2_vq =
        app GHC.Num._or<Int> [3]
        var input_vb - GHC.Types.Int
        var y_vo - GHC.Types.Int
      let nrec esum2_vs =
        app GHC.Num._ot<Int> [3]
        var esum_vm - GHC.Types.Int
        var e2_vq - GHC.Types.Int
      app GHC.Tuple.(, ,)_ou<Int, Int, Int> [3]
      var e2_vq - GHC.Types.Int
      var esum2_vs - GHC.Types.Int
      app quot81_v8◇ [1]
      app GHC.Num._ow<Int> [3]
      app GHC.Num._ox<Int> [3]
      app GHC.Types.I#_oy◇ [1]
      lit 7
      app GHC.Num._oz<Int> [3]
      app GHC.Num._oA<Int> [3]
      app quot8_v3◇ [1]
      app GHC.Num._oC<Int> [3]
      app GHC.Types.I#_oD◇ [1]
      lit 2
      var e2_vq - GHC.Types.Int
      app quot8_v3◇ [1]
      app GHC.Num._oF<Int> [3]
      app GHC.Types.I#_oG◇ [1]
      lit 1
      var esum2_vs - GHC.Types.Int
      app quot8_v3◇ [1]
      app GHC.Num._oI<Int> [3]
      app GHC.Types.I#_oJ◇ [1]
      lit 2
      app GHC.Num._oK<Int> [3]
      var e2_vq - GHC.Types.Int
      var e_vk - GHC.Types.Int
      var y_vo - GHC.Types.Int
    var it_out_0_vi - #

```

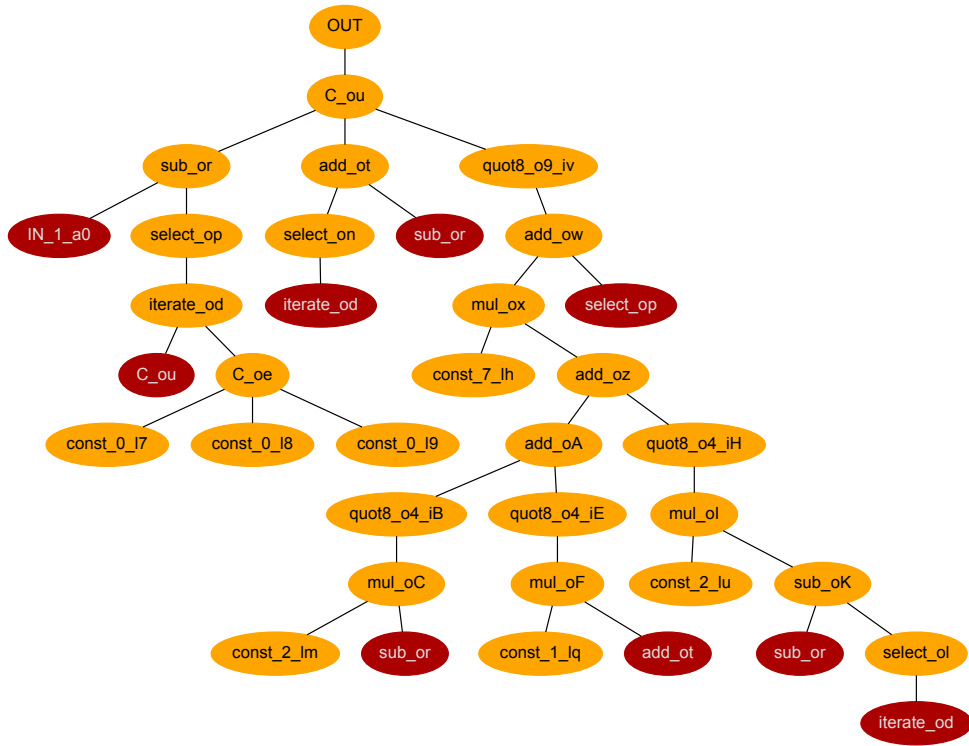
4.4. ábra. CaseReduction utáni Core kód

```

lam [] input {
  var [it_out_0_vi] let-var {
    op [C<GHC.Tuple.(, ), Int, Int, Int>_ou] {
      var [e2_vq] let-var {
        op [sub<Int>_or] {
          var [input_vb] lam-var {
            vv [IN_1_a0] }
          var [y_vo] let-var {
            op [select <(, ), 2><>_op] {
              var [scrutinee_0_vj] let-var {
                var [x_vc] let-var {
                  op [iterate <>_od] {
                    vv [C_ou]
                    op [C<GHC.Tuple.(, ), Int, Int, Int>_oe] {
                      op [C<GHC.Types.I#>_of] { lit [const<Int, 0>_0_] }
                      op [C<GHC.Types.I#>_og] { lit [const<Int, 0>_0_] }
                      op [C<GHC.Types.I#>_oh] { lit [const<Int, 0>_0_] } } } } } } } } }
                }
              }
            }
          }
        }
      }
    }
  }
  var [esum2_vs] let-var {
    op [add<Int>_ot] {
      var [esum_vm] let-var {
        op [select <(, ), 1><>_on] {
          lnk [iterate <>_od] } }
      lnk [sub<Int>_or] } }
  }
  capp [quot81] {
    var [quot81_v8] ? {
      lam [eta_kamu] 1 {
        op [quot8<Int>_o9_iv] {
          op [add<Int>_ow] {
            op [mul<Int>_ox] {
              op [C<GHC.Types.I#>_oy] {
                lit [const<Int, 7>_7_] }
              }
            }
          }
        }
        op [add<Int>_oz] {
          op [add<Int>_oA] {
            capp [quot8] {
              var [quot8_v3] ? {
                lam [eta_kamu] 1 {
                  op [quot8<Int>_o4_iB] {
                    op [mul<Int>_oC] {
                      op [C<GHC.Types.I#>_oD] { lit [const<Int, 2>_2_] }
                      lnk [sub<Int>_or] } } } } }
                  }
                }
              }
            }
          }
        }
        op [add<Int>_ot] } } } } }
      }
    }
  }
  capp [quot8] {
    var [quot8_v3] ? {
      lam [eta_kamu] 1 {
        op [quot8<Int>_o4_iE] {
          op [mul<Int>_oF] {
            op [C<GHC.Types.I#>_oG] { lit [const<Int, 1>_1_] }
            lnk [add<Int>_ot] } } } } }
          }
        }
      }
    }
  }
  capp [quot8] {
    var [quot8_v3] ? {
      lam [eta_kamu] 1 {
        op [quot8<Int>_o4_iH] {
          op [mul<Int>_oI] {
            op [C<GHC.Types.I#>_oJ] {
              lit [const<Int, 2>_2_] }
            }
          }
        }
        op [sub<Int>_oK] {
          lnk [sub<Int>_or]
          var [e_vk] let-var {
            op [select <(, ), 0><>_ol] {
              lnk [iterate <>_od] } } } } } } } }
          }
        }
      }
    }
  }
  lnk [select <(, ), 2><>_op] } } } } } } } }

```

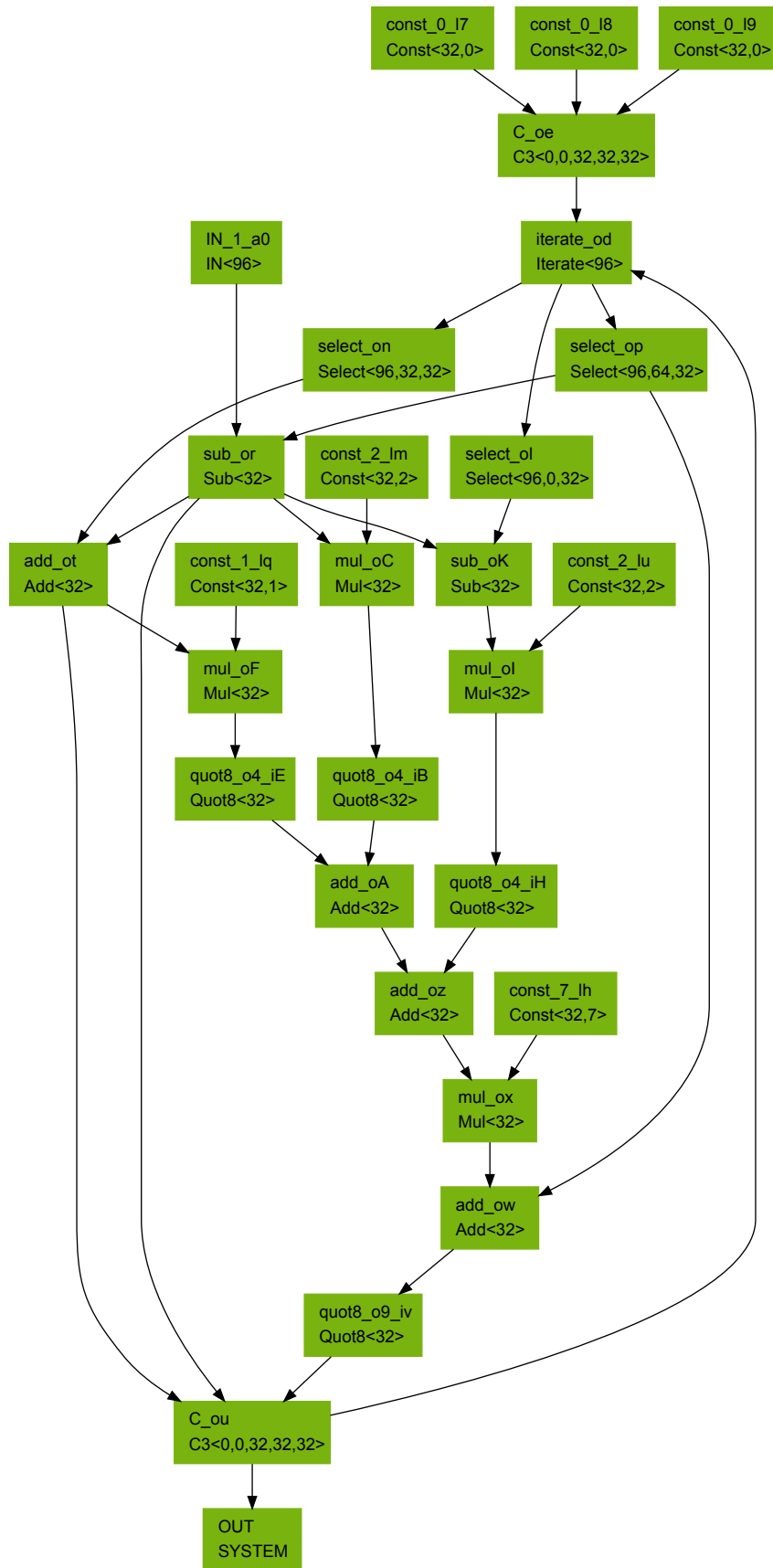
4.5. ábra. A CoreToTree algoritmus utáni gráf



4.6. ábra. A TreeSimplifier algoritmus utáni gráf

IN_1_a0	"IN<96>"	
const_0_17	"Const<32,0>"	const_0_17 const_0_18 const_0_19
const_0_18	"Const<32,0>"	C_ou C_oe
const_0_19	"Const<32,0>"	iterate_od
C_oe	"C3<0,0,32,32,32>"	IN_1_a0 select_op
iterate_od	"Iterate<96>"	iterate_od
select_op	"Select<96,64,32>"	select_on sub_or
sub_or	"Sub<32>"	
select_on	"Select<96,32,32>"	
add_ot	"Add<32>"	
const_7_lh	"Const<32,7>"	
const_2_lm	"Const<32,2>"	const_2_lm sub_or
mul_oC	"Mul<32>"	mul_oC
quot8_o4_iB	"Quot8<32>"	
const_1_lq	"Const<32,1>"	
mul_oF	"Mul<32>"	const_1_lq add_ot
quot8_o4_iE	"Quot8<32>"	mul_oF
add_oA	"Add<32>"	quot8_o4_iB quot8_o4_iE
const_2_lu	"Const<32,2>"	
select_ol	"Select<96,0,32>"	iterate_od
sub_oK	"Sub<32>"	sub_or select_ol
mul_oI	"Mul<32>"	const_2_lu sub_oK
quot8_o4_iH	"Quot8<32>"	mul_oI
add_oz	"Add<32>"	add_oA quot8_o4_iH
mul_ox	"Mul<32>"	const_7_lh add_oz
add_ow	"Add<32>"	mul_ox select_op
quot8_o9_iv	"Quot8<32>"	add_ow
C_ou	"C3<0,0,32,32,32>"	sub_or add_ot quot8_o9_iv
OUT	"OUT<96>"	C_ou

4.7. ábra. A Haskell-EOG modul kimenete



4.8. ábra. A Haskell-EOG modul kimenete grafikusán

```

library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; use work.defs.all;

entity benchmark is
    Port ( restart, clk : in STD_LOGIC;
          IN_1_a0 : in STD_LOGIC_VECTOR (31 downto 0);
          Output : out STD_LOGIC_VECTOR (95 downto 0);
          cIN_1_a0: in std_logic;
          cOutput: out std_logic );
end benchmark;

architecture Behavioral of benchmark is
    component const_source is
        generic(width:integer;value:integer);
        port(out1:out std_logic_vector((width-1)downto 0):=(others=>'0'));
        cq:out std_logic;
        clk:in std_logic);
    end component;
    component c3_a is
        generic(cwidth:integer;cvalue:integer;w1:integer;w2:integer;w3:integer);
        port(in1:in std_logic_vector((w1-1)downto 0);
              in2:in std_logic_vector((w2-1)downto 0);
              in3:in std_logic_vector((w3-1)downto 0);
              q:out std_logic_vector((w1+w2+w3-1)downto 0);
              c1:in std_logic; c2:in std_logic; c3:in std_logic;
              cq:out std_logic;
              clk:in std_logic);
    end component;
    component iterate is
        generic(p1:integer);
        port(feedback:in std_logic_vector((p1-1)downto 0);
              firstvalue:in std_logic_vector((p1-1)downto 0);
              q:out std_logic_vector((p1-1)downto 0);
              cfeedback:in std_logic; cfirstvalue:in std_logic;
              cq:out std_logic;
              clk:in std_logic);
    end component;

    signal const_0_l7: STD_LOGIC_VECTOR ((32-1)downto 0);
    signal cconst_0_l7: std_logic;
    signal C_oe: STD_LOGIC_VECTOR ((32+32+32-1)downto 0);
    signal cC_oe: std_logic;
    signal iterate_od: STD_LOGIC_VECTOR ((96-1)downto 0);
    signal cite_rate_od: std_logic;
    -- tovabbi signal-ok

    begin
    const_0_l7_o: const_source
        generic map (32,0)
        port map(const_0_l7, cconst_0_l7, clk);
    C_oe_o: c3_a
        generic map (0,0,32,32,32)
        port map(const_0_l7, const_0_l8, const_0_l9, C_oe,
                  cconst_0_l7, cconst_0_l8, cconst_0_l9, cC_oe, clk);
    iterate_od_o: iterate
        generic map (96)
        port map(C_ou, C_oe, iterate_od, cC_ou, cC_oe, cite_rate_od, clk);
    -- tovabbi peldanyositasok

    Output <= C_ou;
    cOutput <= cC_ou;
end Behavioral;

```

4.9. ábra. A generált VHDL kód (részlet)

4.1.2. Szimulációs eredmények

Az elkészült VHDL leírást a Xilinx ISE 11.5 verziójú programjával teszteltem, és ugyanezen programcsomag ISim alkalmazásával szimuláltam.

A szimulálást a Ph.D értekezés szerinti bemenettel végeztem, azaz a bemeneti jel a 0 értékről állandósult 8-ra változik, ez az ugrás a 4.11. ábrán bemutatott idődiagramon 50 ns időpillanatban következik be. A PID szabályzó a *cin_1_a0* bemenet felfutó élére min-tavételez, ami 16 cikluson keresztül folytatódik. Az *output y* jele szolgáltatja a kimenetet, a 16 ciklus lefutása után ez 7-nél állandósul.

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	194	9,312	2%	
Number of 4 input LUTs	773	9,312	8%	
Number of occupied Slices	430	4,656	9%	
Number of Slices containing only related logic	430	430	100%	
Number of Slices containing unrelated logic	0	430	0%	
Total Number of 4 input LUTs	815	9,312	8%	
Number used as logic	773			
Number used as a route-thru	42			

4.10. ábra. Az FPGA szintézis eredménye



4.11. ábra. A szimuláció eredménye

4.2. MP3 dekódoló algoritmus

Második tesztetként az MP3 dekódoló algoritmus egy részletét választottam, hogy ezzel is megvilágítsam a módszer gyakorlati használhatóságát. Fontos cél volt, hogy a tesztet a többsébségű kiterjesztés funkcióit is használja, az előbbieken bemutatott PID algoritmus ugyanis egysebességű adatfolyamgráfot ad, így a modell kiterjesztésének vizsgálata nem végezhető el segítségével.

A választott MP3 algoritmusrészlet a dekódolás végfokozatában található **SFB** (Synthesis Filter Bank) modul, amely a frekvenciatartományban kódolt jelsorozatból előállítja az időtartománybeli alakot, azaz a kódolás során mintavételezethez hasonló hanghullámot.

Az MP3 dekódoló algoritmusnak létezik Haskell forráskódú változata [8], ami viszont az SFB-t külső, C nyelvű modulként tartalmazza (PC-n futtatva így gyorsabb kódot kaptak a fejlesztők). A C kód Haskell-es megfelelőjét elkészítettem, amivel az automatikus szintézis tesztelése már elvégezhető.

Lebegőpontról áttértem fixpontos számábrázolásra, és az így módosított algoritmust az eredeti, Haskell alapú mp3 dekódoló programba beillesztve teszteltem. A fixpontos számábrázolást kipróbáltam különböző bitszélességekkel. 7 bites pontosságnál zajos hanghatást tapasztaltam, de 10 bitnél ez számomra már nem volt hallható, így jókora ráhagyással a választásom 32 bites számábrázolásra, és ezen belül 16 bites törtszámábrázolásra esett. Az így módosított Synthesis Filter Bank algoritmust a 4.12. ábrán látható kódrészlettel írtam le.

Az algoritmus szekvenciálisan négy részre osztható. Az 1. fokozat a bemenetből előállítja az *slArray* segéd tömböt, amiből az *ssum* függvény (2. fokozat) az állapot tömb frissítéséhez szükséges értékeket szolgáltatja. Az új állapot tömbből az *uconv* függvény (3. fokozat) elkészíti az *ulArray* segéd tömböt, és ebből az *usum* függvény (4. fokozat) előállítja a soron következő kimeneti, időtartománybeli mintavételi értékeket. A fokozatok összekapcsolásáért a *synthH1* függvény felel, ez tekinthető a Synthesis Filter Bank algoritmust implementáló függvénynek.

Az *ssum* függvény 32 db. frekvenciatartománybeli adatot kap, amiből 64 db. új értéket állít elő az állapot tömb (*stateArray*) számára. A teljes állapot tömb 1024 elemet tartalmaz, így az újonnan kiszámolt 64 elem mellett az előző 15 futás során keletkezett 64-64 értéket is tárolja. Az *uconv* fokozat az állapot tömb és a *synthArray* lookup table alapján az 512 elemű *ulArray* tömb feltöltéséért felelős. A tömb 16-16 elemét (nem egymást követő elemekről van szó) az *usum* összegzi, az egyes összegek pedig a soron következő kimeneti mintáknak felelnek meg.

```

{-# LANGUAGE NoImplicitPrelude #-}
module MP3(hwmain) where

import InstructionSet
import SFBConstants

hwmain x = synthH1 inputArray (stateArray,[]) x

uconv newstate start k =
  let (i,j) = divMod64 k
      convli = if (j<32)
                then (i*128 + j)
                else (i*128 + (j-32) + 96)
      conv1 = newstate ! (mod1024(convli+start))
      conv2 = synthArray ! k
  in (k,conv1 'imul' conv2)

usum :: IntArray -> [Int]
usum uL = map sumn [0..31]
  where sumn i = sum $ map (func i) [0..15]
        func i j = uL ! (j*32 + i)

ssum :: Int -> IntArray -> [(Int,Int)]
ssum start sL = map (\i -> (i+start,sumn i)) [0..63]
  where sumn i = sum $ map (func i) [0..31]
        func i j = ((sL ! j) 'imul' (lookupArray ! (i*(32::Int) + j)))

synthH1 :: IntArray -> (IntArray,[Int]) -> Int -> (IntArray,[Int])
synthH1 input (state,-) s =
  let sL = map (\i -> (i,input ! (i*18 + s))) [0..31]
      s2L = sLArray // sL
      start = mod1024(2048-(s+1)*64)
      newstate = state // (ssum start s2L)
      uL = map (uconv newstate start) [0..511]
      u2L = uLArray // uL
  in (newstate,usum u2L)

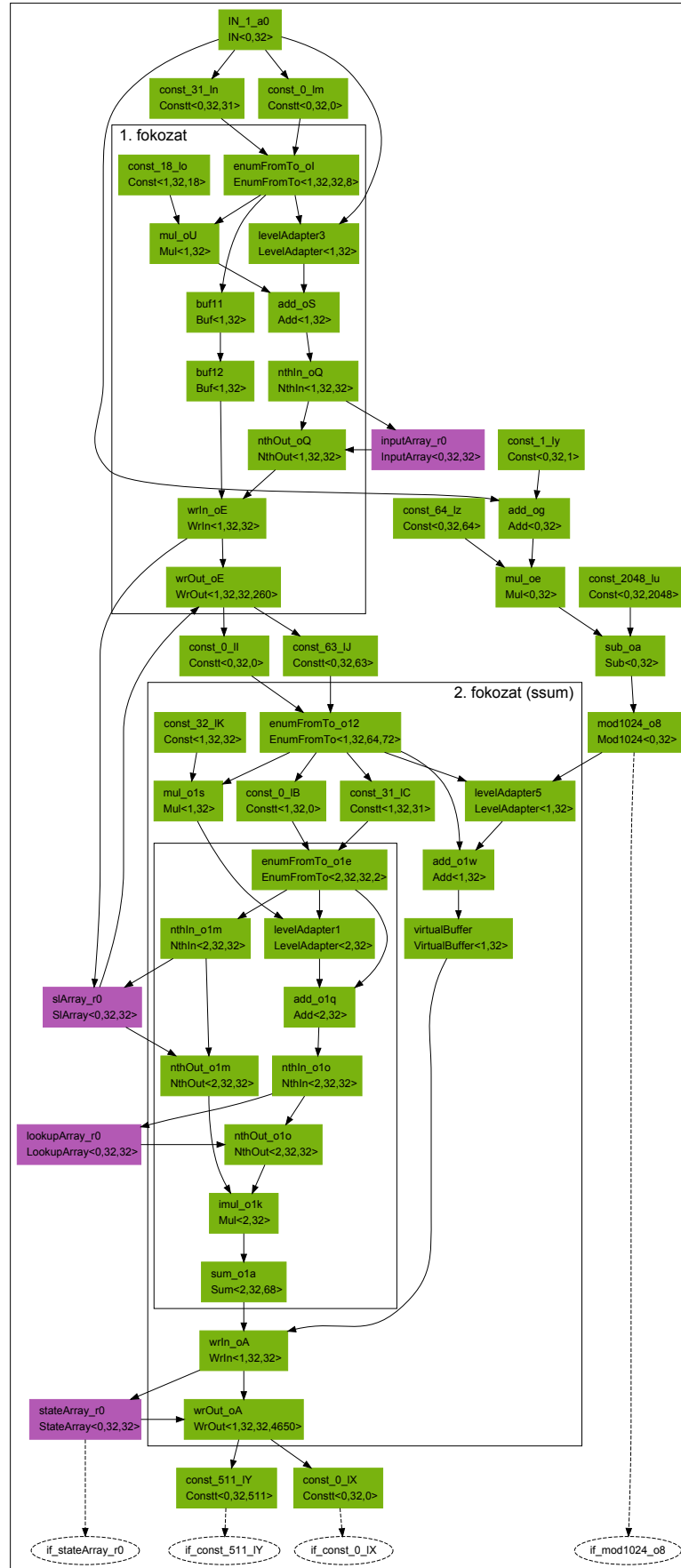
```

4.12. ábra. MP3 Synthesis Filter Bank

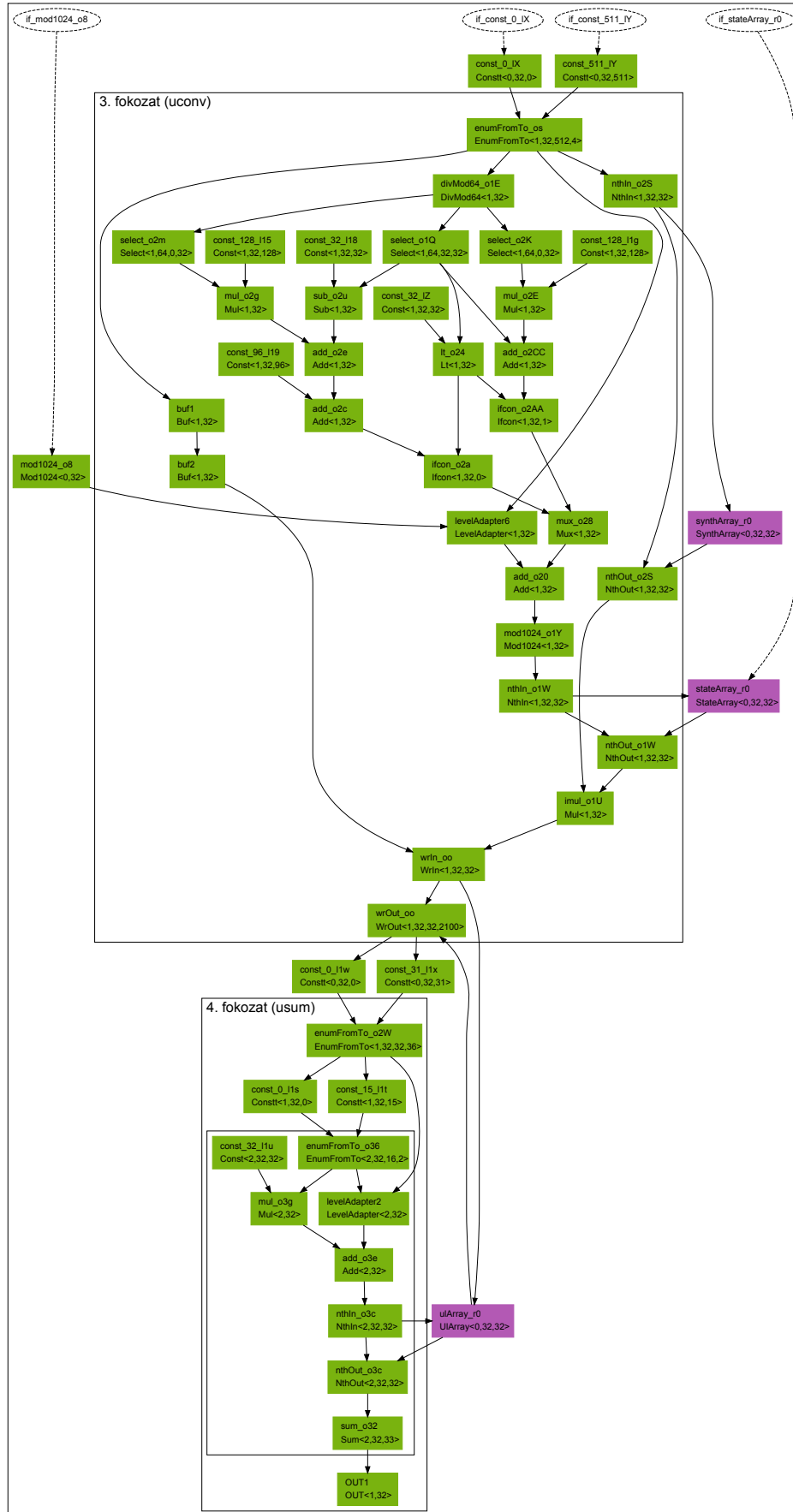
4.2.1. A Haskell-EOG fordítás eredménye

A *synthH1* függvényből generált EOG-t az átláthatóság érdekében két részletben, a 4.13. és a 4.14. ábrán mutatom be. A 4.13. ábra az 1. és a 2. fokozatot, a 4.14. ábra pedig rendre a 3. és 4. fokozatot ábrázolja. Mindkét adatfolyamgráfban megtaláljuk a következő műveleteket: *stateArray_r0*, *const_511_lY*, *const_0_lX*, *mod1024_o8*. Ezek mindkét ábrán azonos műveleteket jelölnek, tehát az ábrák ezek segítségével összekapcsolhatóak. A négy szóban forgó műveletet a grafikus képen négy darab, szaggatott vonallal jelzett ellipszis alakú pont teszi láthatóvá.

A tömböket megvalósító műveleteket lilával jelöltem. Jól látható, hogy az 1. és a 2. fokozat között az *sLArray* tömb biztosítja a kapcsolatot, a 2. és 3. fokozat között a *stateArray*, míg a 3. és 4. fokozat között az *uLArray* tölti be ezt a szerepet. A *lookupArray* és a *synthArray* tömböket csak olvasásra használjuk, ezek tulajdonképpen lookup table funkciót látnak el. Az *inputArray*-t szintén csak olvassuk, ez ugyanis a bemeneti, frekvenciatartománybeli értékeket tartalmazza.



4.13. ábra. MP3 SFB algoritmusból generált EOG (1. és 2. fokozat)

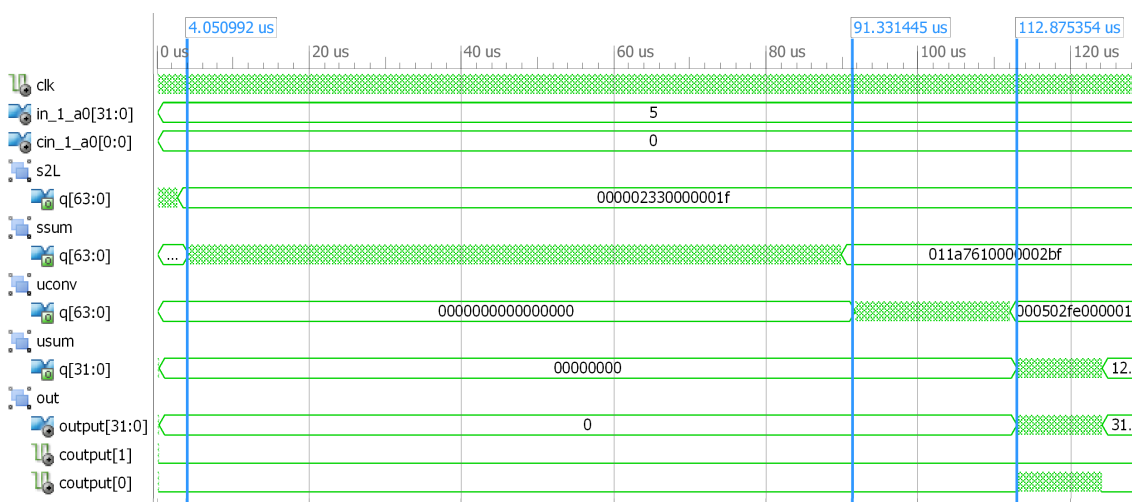


4.14. ábra. MP3 SFB algoritmusból generált EOG (3. és 4. fokozat)

4.2.2. Szimulációs eredmények

Az MP3 SFB algoritmusból automatikusan generált VHDL leírást Xilinx ISim 13.2 környezetben szimuláltam, az így kapott hullámformákat a következő néhány oldalon mutatom be.

A 4.15. ábra az algoritmus egyetlen bemenetre adott jelalakváltozásait mutatja. Ezen az ábrán konkrét számértékek a felbontás nagysága miatt nem láthatóak, az egyes fokozatok és a teljes algoritmus időigénye viszont nagyon szépen leolvasható. Az 1. fokozat a bemenet hatására kb. 4 us után, a 2. fokozat kb. 91 us után, a 3. fokozat pedig közel 113 us után fejezi be működését. A teljes algoritmus 125 us alatt fut le, ami az 50 Mhz-es órajelnek megfelelően 6250 órajelciklusnak felel meg. Jól látható, hogy a négy közül a 2. fokozatnak, vagyis az *ssum* függvénynek van a legnagyobb lappangási ideje. Ez érthető is, hiszen az itt található két egymásba ágyazott ciklus $32 * 64 = 2048$ iterációt hajt végre, szemben az *uconv* és az *usum* 512 iterációjával.

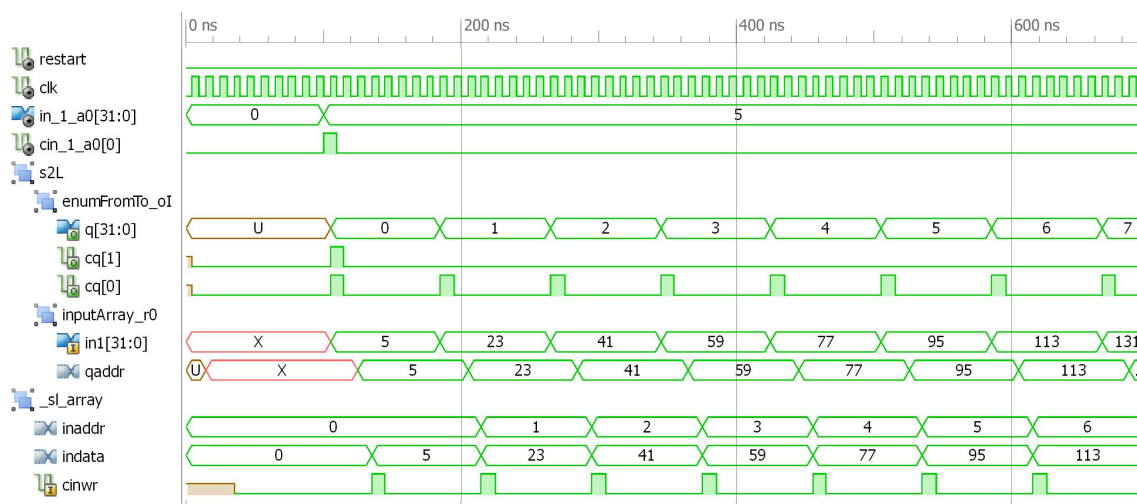


4.15. ábra. SFB modul full

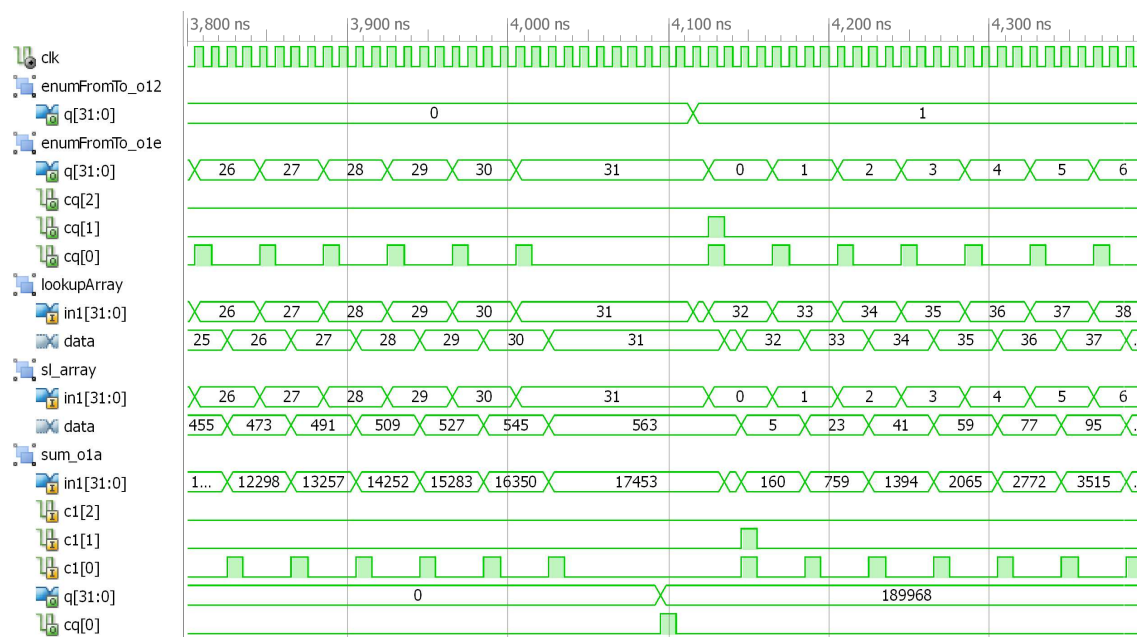
Az első, azaz az *slArray* tömböt előállító fokozat szimulációs eredményét a 4.16. ábrán mutatom be. Egyszerűség kedvéért az *inputArray* egy olyan tömb, melynek minden *i*-edik pozíciójában *i* érték szerepel (identikus leképzés), ezért érthető, hogy az 5-ös indexet lekérdezve 5-ös számot, 23-ast lekérdezve 23-as számot kapunk. Végül az így adódó számok sorban beíródnak az *slArray* tömb egymás utáni elemeire.

Az *ssum* rész (2. fokozat) nagyfelbontású szimulációját a 4.17., kiselbontású szimulációját pedig a 4.18. ábra mutatja. A nagyfelbontású ábrán a külső ciklus éppen a 0 és az 1 értékek között vált, a belső ciklusból pedig 13 iteráció látszik. A *lookupArray* (ami szintén identikus leképzés) és *slArray* tömbök lekérdezése belső iterációnként egyszer történik meg. A *sum_o1a* művelet a bemenetére érkező értékeket összegzi (a külső ciklus 0 értékénél vett összeg pl. 189968).

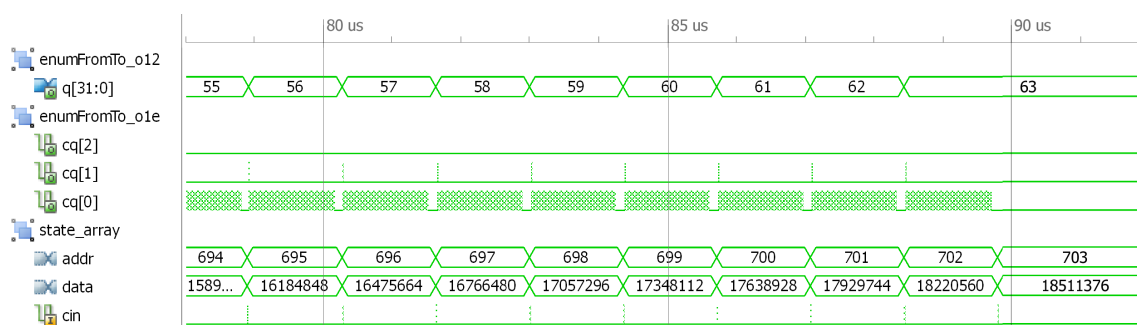
A kiselbontású képen már csak a külső ciklus értéke olvasható le. Látható, hogy a ciklus 63-ig megy, és minden lépésben a *stateArray* tömb egymás utáni helyeire a *sum_o1a* kimenetén megjelenő értéket írjuk.



4.16. ábra. SFB szimuláció, 1. fokozat

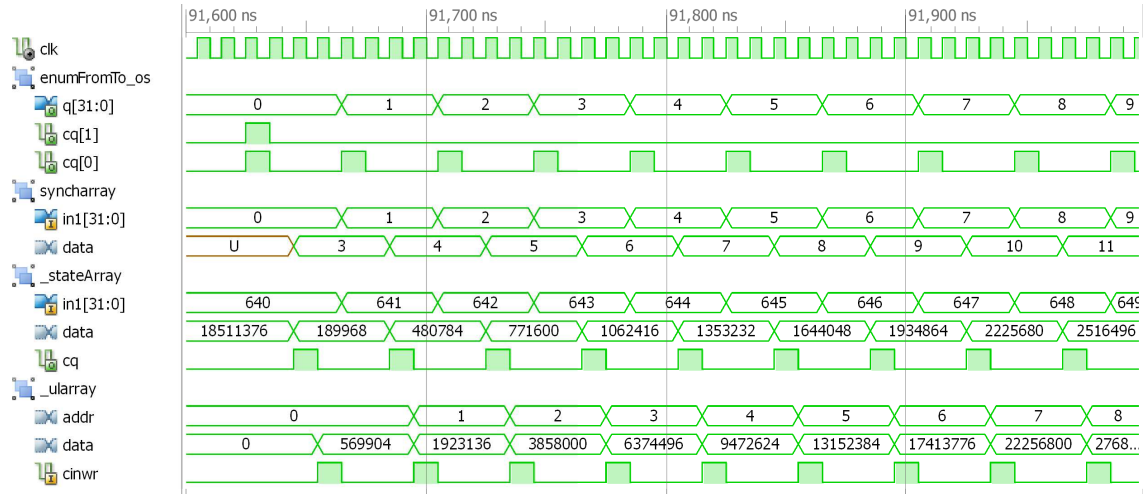


4.17. ábra. SFB szimuláció, ssom fokozat (nagyfelbontás)



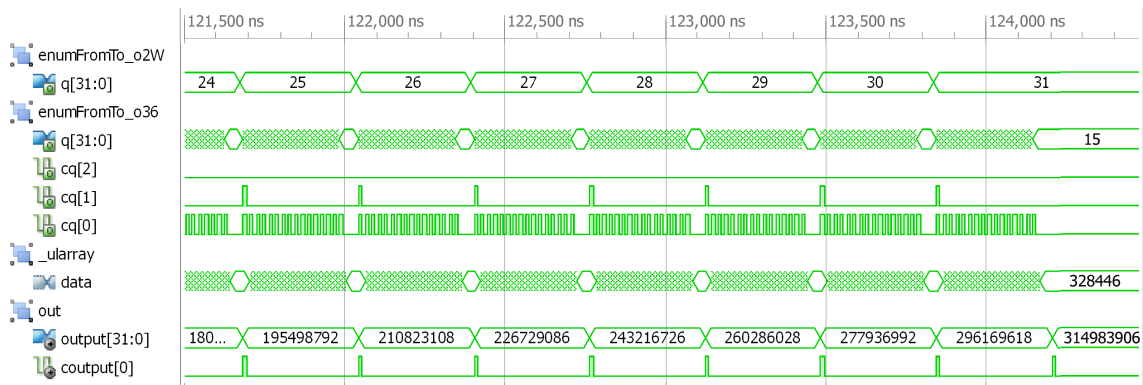
4.18. ábra. SFB szimuláció, ssom fokozat (kisfelbontás)

Következő fokozatunk a 4.19. ábrán szereplő *uconv*. A ciklus itt 0-511 között lépked, ebből az ábrán a 0-8 közti szimulációs eredmény látható. Iterációként egyszer kérdezzük le a *synthArray* lookup table értékét. Ez esetben a tömböt úgy adtam meg, hogy az *i*-edik helyen *i* + 3 szerepel. A *stateArray* megfelelő elemének lekérdezése után az *ulArray* tömb egymás utáni elemei kerülnek feltöltésre. (a 0. helyre pl. az 569904 kerül)



4.19. ábra. SFB szimuláció, *uconv* fokozat

Az SFB végső lépésének (*usum* modul) szimulációja a 4.20. ábrán látható. Ez esetben is két egymásba ágyazott ciklusunk van, a külső ciklusváltozó a 24-31 közötti, végső tartományban látszik. Az ábra jól mutatja a belső ciklus stream vezérlését is; cq[1] a külső, cq[0] pedig a belső ciklusváltozó növelésekor aktív. Az *ulArray*-t minden belső iterációban lekérdezzük (csak az utolsó lekérdezett érték, a 328466 látható a felbontás miatt). Ezeket az értékeket a külső ciklusváltozó minden egyes értékére szummázzuk, így adódik az ábra alján látható kimeneti stream, azaz az elvárt, időtartománybeli jelek sora.



4.20. ábra. SFB szimuláció, *usum* fokozat

Az FPGA-s algoritmussal 125 us alatt 32 db. kimeneti minta állítható elő. 44100 Hz-es mintavételnél a 32 db. mintát 726 us alatt kell előállítanunk valósidejű lejátszáshoz. Ez azt jelenti, hogy monó hanganyagnál a hardver több, mint ötször gyorsabban végez az elvárt-nál, sztereó hanganyagnál pedig több, mint kétszer gyorsabban szolgáltat eredményeket. A szintézis eredménye tehát real-time lejátszásra is alkalmas.

5. fejezet

Integrált fejlesztői keretrendszer

Ebben a fejezetben bemutatom az általam összeállított integrált fejlesztői környezetet, amely képes az egész fordítási folyamatot kézben tartani, ezáltal a fejlesztést egyszerűbbé és felhasználóbaráttá teszi. A megoldásomat az Eclipse nyílt fejlesztőrendszerre építettem, mert ezzel kevés erőbefektetéssel nagy tudású környezetet lehet létrehozni. A világban nagyon sok szoftverfejlesztési környezettel foglalkozó projekt épül az Eclipse-re, egységes eljárást biztosítva a szerkesztéshez, fordításhoz, célkód generáláshoz és hibakereséshez.

Beágyazott környezetben szintén használják a keretrendszert, többek között ilyen a Xilinx FPGA-s softcore CPU-k fejlesztését támogató Software Development Kit [9].

Ebben a fejezetben bemutatom azokat az Eclipse plugin-okat amivel a köztes leírásokat (Haskell kód, EOG, VHDL) megjeleníthetjük és szerkeszthetjük. Ezeknek egy része saját fejlesztés, a többi plugin pedig az internetről szabadon letölthető.

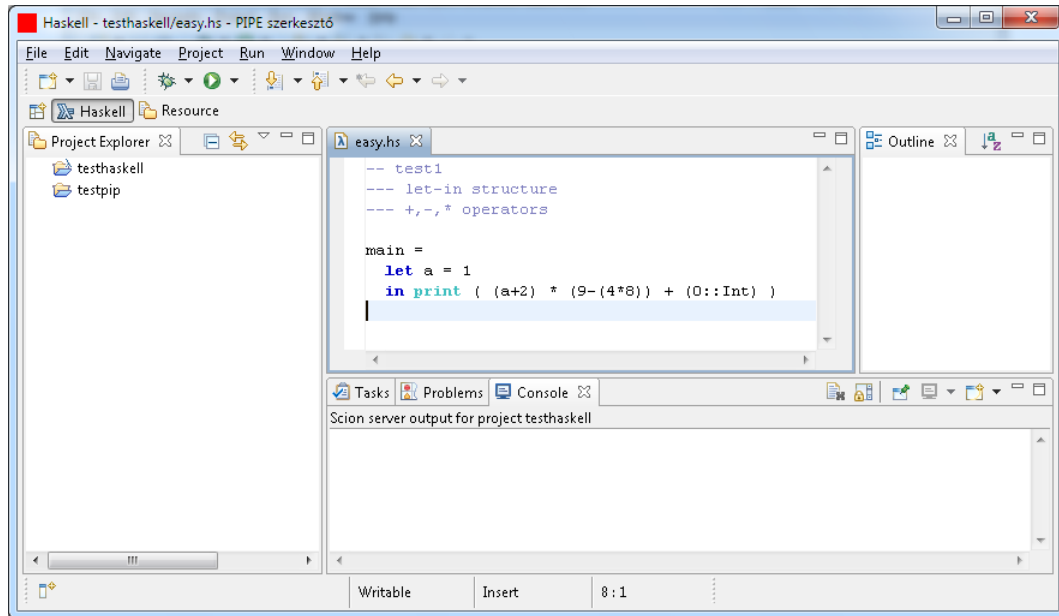
5.1. Haskell szerkesztő

Szerencsére létezik az Eclipsenek egy kiterjesztése, amely funkcionális nyelveket, és ezeken belül is legfőképpen a Haskell fejlesztését támogatja. Az EclipseFP [10] plugin az Eclipse keretrendszert képessé teszi arra, hogy Haskell programokat fejlesszünk és ennek során az összes szükséges lépés a kezünkben legyen a szerkesztéstől egészen a futtatható tárgykód elkészítéséig. A fordítás a GHC-n keresztül valósul meg, tehát a kimenet processzoros tárgykód.

Az 5.1. ábrán a Haskell szerkesztőt és az éppen megnyitott, nagyon egyszerű forráskódot láthatjuk működés közben. Ennek az Eclipse pluginnak a haszna triviális, olyan kényelmi szolgáltatásokat kapunk, mint pl. a kód highlight, automatikus formázás, fordítói warning, error üzenetek jelzése és a GHCi saját konzolban történő futtatása. (ezáltal interaktív fordítói felülethez jutunk)

5.2. Elemi műveleti gráf szerkesztő

A fordítás Haskell-EOG modulja a kódot elemi műveleti gráfra fordítja. Ezt a köztes leírást nagyon hasznos megjeleníteni (és adott esetben szerkeszteni), ez ugyanis a fordító döntéseiről, optimalizációjáról ad képet a felhasználónak.



5.1. ábra. Haskell forráskód szerkesztése Eclipse-ben

Az EOG fájlformátum nem szabványos, tehát az ehhez a formátumhoz illeszkedő szerkesztő plugint csak saját fejlesztés árán tudjuk megvalósítani.

Az EOG fájl tulajdonképpen egy DSL (domain specifikus nyelv), ezért ehhez a fejlesztéshez felhasználtam az Xtext plugint [11], ami DSL-ek definiálására és kezelésére szakosodott. Először az 5.2. ábrán látható kóddal definiáltam az EOG nyelvet **BNF**-ben (Backus Normal Form), amiből az Xtext már automatikusan generálta a szerkesztőfelületek java kódjait. Az elkészült plugint működés közben az 5.3. ábrán láthatjuk.

A háttérben az EMF (Eclipse Modeling Framework, [12]) futtatásával automatikusan generálódott egy Eclipse Ecore [13] modell, ennek köszönhetően az elemi műveleti gráfhoz megfelelően absztrakt szinten hozzá tudunk férni a plugin más pontjairól is, ami a következő lépés miatt fontos is lesz.

Az EOG fájl egy nagyon egyszerű és fontos köztes reprezentáció, de gyakorlati haszna csak akkor szembetűnő, ha a fájl szövegének megtekintése vagy módosítása grafikus szerkesztőben, grafikus elemeket használva történik. Mivel az EOG belső modelljét az EMF generálta, adott volt a lehetőség az Eclipse egy másik bővítő modulját, a GMF-et (Graphical Modeling Framework, [14]) használva grafikus felület fejlesztéséhez.

A grafikus szerkesztőt a fenti pluginokat felhasználva elkészítettem, így az EOG fájlból automatikusan grafikus nézetet generálhat a felhasználó, ill. grafikus nézetből létrehozhatja a hozzá tartozó szöveges állományt. A grafikus felület működés közben az 5.4. ábrán tekinthető meg.

Ahogy látható, minden kis téglalap egy EOG csomópont, a köztük lévő nyilak pedig a két csomópont közötti élek, vagyis adatfolyamok.

```

grammar hu.progtar.Eog with org.eclipse.xtext.common.Terminals

generate eog "http://www.progtar.hu/Eog"

Model hidden(WS, ML_COMMENT, SL_COMMENT):
    (types+=Type)*
    (operations+=Operation)*;

terminal SL_COMMENT: '## ' !('\n'|\r')* ('\r'? '\n')?;
terminal WS: (' '|\t'|\r')+ ;
terminal sorveg: '\n'*;

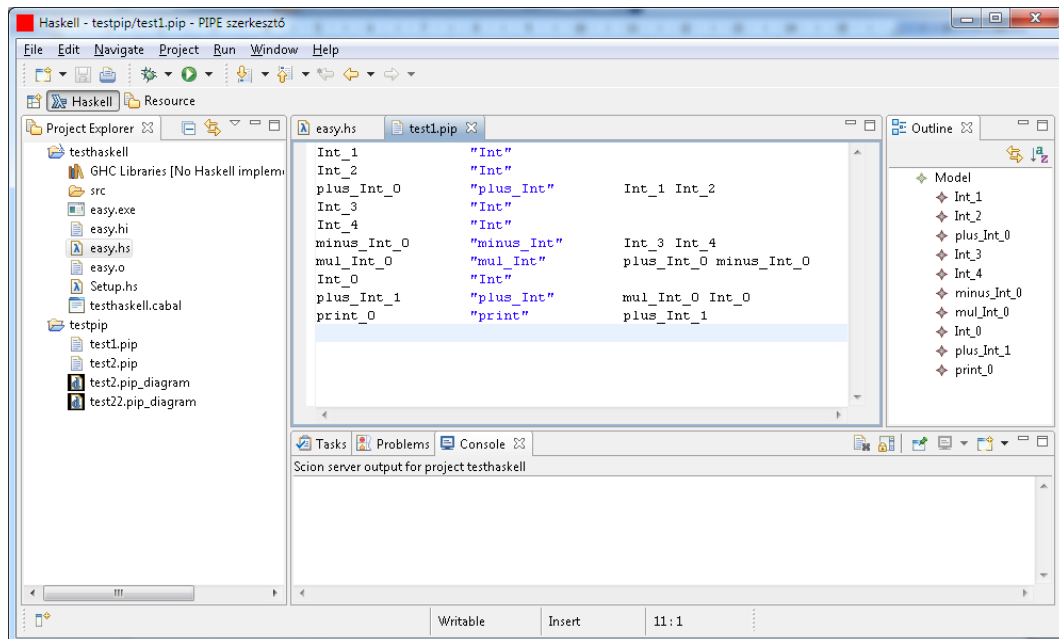
Type:
    '# type' name=ID delay=INT '\n';

Operation returns Operation:
    name=ID type=STRING (inputs+=Pipe)* '\n';

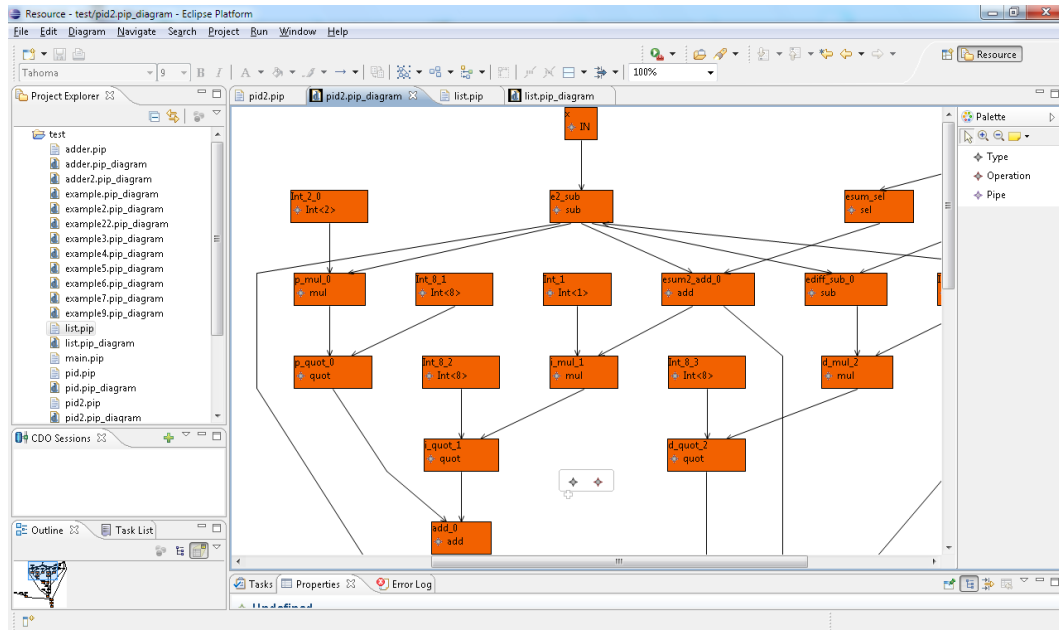
Pipe:
    source=[Operation|ID];

```

5.2. ábra. Az EOG fájl szintaktikájának BNF alakja



5.3. ábra. Az EOG leírás szerkesztése Eclipse-ben



5.4. ábra. EOG grafikus nézete Eclipse-ben

5.3. VHDL szerkesztő

A teljes fordítási folyamat végén egy VHDL kimeneti leírás generálódik, amit majd FPGA-ba szintetizálhatunk külső szoftverekkel.

A VHDL egy sokat használt nyelv, valószínűleg ennek köszönheti, hogy szerkesztésére szintén létezik meglévő Eclipse plugin, Verilog editor [15] néven. (a Verilog egy másik hardverleíró nyelv, de ez a plugin a VHDL-t is képes kezelni)

5.4. A teljes fordítási folyamat

A fordítást külső programmodulokkal valósítjuk meg, így ha a fejlesztői keretrendszerünket képessé kívánjuk tenni a fordítás elvégzésére is, akkor az FP pluginhoz hasonlóan a külső fordítóprogramokat az Eclipse automatikusan meg tudja hívni.

A következő lista az Eclipse-el történő fejlesztési folyamat lépéseit írja le.

A felhasználó

1. létrehozza a forráskódot a Haskell szerkesztőben
2. az elkészült Haskell kódból EOG-t generál a 2.2. fejezetben leírt modult futtatva
3. meg tudja nézni az így készült EOG eredményt, ha szükséges visszaléphet az 1. pontra, ill. kisebb módosításokat akár a grafikus szerkesztőn keresztül is végrehajthat az adatfolyam gráfon
4. az EOG-t a 2.3. fejezetben leírt modul segítségével lefordítja VHDL kódra
5. az eredményt megtekintheti és szerkesztheti a VHDL szerkesztőben
6. majd a végleges kódot FPGA-ba szintetizálhatja külső szoftvereket (pl. Xilinx ISE konzolos alkalmazásai) indítva

6. fejezet

Továbbfejlesztési lehetőségek

Az elkészült program képes lefordítani a definiált művelethalmaznak megfelelő, leírt megkötésekkel rendelkező Haskell kódot. A módszer és a fejlesztett program számtalan továbbfejlesztési lehetőség elé néz, amelyekből néhányat a most kezdődő fejezetben ismertetek.

6.1. Optimalizálási lépések

A fordítórendszer támogatja optimalizációs programok integrálását. Az IIT Tanszéken fejlesztett PIPE optimalizációs tervezőrendszer EOG bemenetet vár, így az általam fejlesztett fordítóprogram Haskell-EOG és EOG-VHDL moduljai közé beékelhető lenne.

Adott esetben az algoritmust EOG szinten optimalizálni tudjuk azzal a céllal, hogy pipeline működés során nagyobb áteresztőképességet, vagyis az újraindítási idő csökkenését érjünk el. Ennek a gyakorlati haszna akkor szembetűnő, ha az EOG-val meghatározott műveletsort egymás után sok adaton kell elvégezni, ekkor ugyanis a pipeline működés hatalmas hatékonyságbeli növekedést jelenthet.

A PIPE rendszer az újraindítási időt képes redukálni, a műveletek bemeneti csatornáit szinkronizálja, a feladat végrehajtó egységek típusának és számának ismeretében biztosítja az allokációt, majd meghatározza az egyes műveletek kezdési idejét.

6.2. Egyebek

- A bemenetként használható Haskell kódban jelenleg nem szerepelhet függvényrekurzió, és ezzel algoritmusleírások egy részét kizárjuk a fordítható kódok közül. Megoldás lehet pl., ha előre meghatározott maximális számú rekurzív hívást megengedünk, mert így a hardveres megvalósítás realizálhatóvá válna. A rekurzió kezelésében ezen kívül is sok nyitott kérdés rejtőzik, így a jövőben érdemes lehet ezzel behatóan foglalkozni. Kiindulópont lehet két releváns cikk [29, 49] is.
- Egy érdekes és időszerű kutatási téma lehet az LLVM [37] alapján történő automatikus HDL generálás, erre a köztes reprezentációra ugyanis egyre több nyelvről lehet fordítani, többek között létezik egy Haskell-LLVM frontend is.
- A kimeneti VHDL leírás egyszerűsítésén érdemes lehet még dolgozni. Jelenleg minden

EOG művelet modulpéldányosításra fordul, így pl. a VHDL-ben egyszerűen ábrázolható konstansok is, amik emiatt nagyobb és átláthatóság szempontjából rosszabb kimeneti fájlt eredményeznek.

- VHDL-ről a szintézerek általában EDIF (Electronic Design Interchange Format) formátumra fordítanak, így vizsgálat tárgyát képezheti az is, hogy a fordító kimenete esetleg a VHDL-től alacsonyabb szintű, FPGA közeli kódot jelentsen.
- Elképzelhető lenne a listák olyan kezelése, hogy a lista elmein végrehajtandó műveleteket előre megadott n számú feldolgozóegység párhuzamosan végezze, így a sebesség és az erőforráshasználat között az n értékének finomhangolásával tudnánk szabályozni a szintézist.
- Tervezem egy MicroBlaze lágymagos processzor alapú tesztrendszer elkészítését, amellyel a fordítóprogram kimeneteként létrejött VHDL kód valódi FPGA-s környezetben is egyszerűen, akár automatikusan tesztelhető lenne.

7. fejezet

Összefoglalás

Dolgozatom során egy olyan módszer alapelveit és megvalósítását dolgoztam ki, amely a Haskell nyelven bizonyos megkötésekkel leírt algoritmusokat VHDL leírássá alakítja. Az eljárás minden lépésére (moduljára) kidolgoztam algoritmusokat és azokat PC-n futó fordítóprogramként implementáltam is.

A kidolgozandó módszer és fordítóprogram elé a következő követelményeket támasztottam:

1. a Haskell forráskód és VHDL leírás közötti fordítás legyen teljesen automatikus
2. fordítás során a felhasználó kapjon hibaüzenetet, ha a forráskód szintaxisa nem megfelelő, vagy ha egy Haskell kifejezés a megkötések miatt nem használható
3. a program moduláris felépítésű legyen, hogy az egyes részek a jövőben minden további nélkül újrahasználhatóak legyenek
4. a jövőben könnyen, tehát a főbb modulok legfeljebb minimális módosítása után implementálhatóak legyenek optimalizációs algoritmusok
5. a forráskód megkötéseit ne kelljen teljes egészében a fejlesztési periódus elején meghatározni, vagyis a használható Haskell kifejezések lehetőleg a fordítóprogram utólagos fejlesztése nélkül, dinamikusan bővíthetők legyenek

Az 1. és 3. követelmény a kidolgozott módszer és az ennek alapján elkészült moduláris felépítésű fordítóprogram eredményeként teljesül. A szintaktikai hibákat a módszerbe beépített GHC frontend jelzi, így a felhasználó azonnal értesül a forráskódi elírásokról vagy a hibás kódolásról. Amennyiben a szintaktikailag helyes forráskódban olyan függvényt használunk, ami a fordítóprogram aktuális művelethalmazában nincs benne, szintén hibajelzést kapunk, ezzel tehát a 2. követelmény szintén teljesül. Köztes reprezentációként EOG-t használok, ezáltal optimalizációs algoritmusok beépítését a módszer messzemenően támogatja, ezzel teljesül a 4. követelmény. Az 5. követelmény a művelethalmaz fogalmának bevezetésével teljesül.

A rendszert moduláris elven építettem fel, hogy az egyes részek külön-külön is minél általánosabban felhasználhatóak legyenek. A nyelvi vagy technológiai peremfeltételek módosítása esetén csak az eljárás egy-egy részét, az annak megfelelő modult kell módosítanunk.

Amennyiben Haskell forrásnyelv helyett más funkcionális nyelvből szeretnénk kiindulni, akkor a GHC fokozat helyett egy arra specializált fordítóprogramot kell készítenünk, ami a GHC Core nyelvű köztes leírást generálja. Ebből a leírásból az általam elkészített rendszer képes VHDL-t generálni. FPGA helyett ASIC integrált áramkörök gyártására is van lehetőség, a kimenetként kapott VHDL ugyanis ASIC gyártás alapja lehet.

A rendszer alkalmas optimalizáló fokozat beépítésére, így pl. az IIT Tanszéken fejlesztett PIPE optimalizáló rendszert a fordítás részévé lehet tenni, így pipeline algoritmusok esetén a hardver hatékonysága nagymértékben javulhat.

A rendszer rugalmasságát nagyban növeli, hogy a művelethalmaz dinamikusan bővíthető a fordítóprogram fejlesztése nélkül is. A bővítéshez csupán arra van szükség, hogy egy adott Haskell függvényhez létrehozzunk egy VHDL modult, és a művelethalmaz megfelelő fájljaiban ezt a függvényt regisztráljuk. Egy megépített mp3 lejátszó VHDL modulját importálva pl. egyetlen haskell függvényhívással akár egy mp3 lejátszás is indítható.

A rendszer támogatja a hierarchikus tervezést, tehát a magasszintű nyelven kódoló felhasználó (pl. matematikus vagy informatikus) és a hardvertervező egymástól függetlenül dolgozhat az adott problémán. A matematikus leírja a szoftvert, és amennyiben nem használ egyedi függvényt, a kódja automatikusan fordítható hardverre. Ha az algoritmusához különleges hardverre lenne szükség (pl. optimalizáció miatt), akkor a hardvertervező ezzel párhuzamosan megírhatja a szükséges VHDL modult, és ennek végeztével a matematikus által leírt függvény ezt a kódot fogja példányosítani.

A módszert kiterjesztettem többsebességű adatfolyamok esetére. Ennek keretében kidolgoztam az EOG modell többsebességű változatát, amire az MREOG nevet vezettem be. Algoritmust adtam az újraindítási- és lappangási idők kiszámítására, hogy azok az új modellben is meghatározhatóak legyenek.

Egy PID szabályzó algoritmuson és az MP3 dekódoló algoritmus részletén keresztül bemutattam a fordítás lépéseit és a lépések közötti köztes reprezentációkat, a fordítói csővezeték végén megkapott VHDL kódot pedig a Xilinx ISE FPGA fejlesztőrendszerrel szimuláltam.

A teljes fordítási folyamat irányítására egy Eclipse alapú keretrendszert javaslok, amelyhez a szükséges beépülő modulokat implementáltam.

8. fejezet

Kutatás szellemi tulajdon témakörben

A most következő fejezet egy, a Szellemi Tulajdon Nemzeti Hivatala által kiírt pályázatra készült. Egy rövid bevezető után a 8.1. fejezetben összefoglalom a számítógéppel megvalósított találmányokról szóló irányelveket, a 8.2. fejezetben ismertetem az általam szabadalmi adatbázisokban használt keresőfeltételeket és értékelem az azokra kapott eredményeket, a 8.3. fejezetben bemutatom a diplomamunkám szempontjából legfontosabb szabadalmi bejelentéseket, végül a 8.4. fejezetben a munkám szabadalmi vonatkozásait elemzem.

Magyarországon a polgári jog szabályozza a szellemi tulajdonvédelem jogkörét, ami szerzői jog és iparjogvédelem részekből áll. A szerzői jog irodalmi, művészeti vagy tudományos alkotásokat (pl. matematikai számítás, algoritmus, számítógépi program és dokumentáció) véd külön bejelentés nélkül. Ennek értelmében a leírt saját algoritmusok, saját programrészletek és a teljes írás szerzői joga engem, mint a jelen diplomaterv íróját megillet. (a más szerzőtől idézett részek persze kivételt képeznek) Amennyiben ezeket bárki a beleegyezésem nélkül felhasználja, szerzői jog sértést követ el, ami megegyezés hiányában bírósági úton érvényesíthető. (mivel a dolgozatot egyetemi diplomatervként írom, az egyetem bizonyos jogokat elvár, ezeknek az elfogadásáról szól az aláírt nyilatkozat a tartalomjegyzéket követő oldalon)

A szellemi tulajdonvédelem másik fontos területe az iparjogvédelem, ami tk. egy gyűjtőfogalom a szabadalom, a használati minta, a formatervezési minta, a védjegy, a növényfajtalom, a földrajzi árujelző és az eredetmegjelölés oltalmi formákra vonatkozóan.

A felsoroltak közül jelen dolgozat szempontjából az 1995. évi XXXIII. törvényben [19] leírt szabadalmi oltalom, azaz a találmányok védelmi formája releváns. A törvény szerint „szabadalmazható minden új, feltalálói tevékenységen alapuló, iparilag alkalmazható találmány a technika bármely területén”, amely a következőket jelenti:

- új: nem tartozik a technika állásához, tehát nem volt még bárki számára hozzáférhető, bárhol a világon
- feltalálói tevékenységen alapuló: szakember számára nem nyilvánvaló

- iparilag alkalmazható: ha az ipar valamely ágában előállítható vagy használható

„A Magyar Köztársaságban érvényes szabadalmat vagy nemzeti- vagy európai- vagy a Szabadalmi Együtműködési Szerződés (PCT) keretein belül indított szabadalmi bejelentéssel lehet szerezni” [16], ami érvényességét tekintve a következőket jelenti:

- nemzeti: kizárólag Magyarországon érvényes
- európai: a bejelentés megtehető az Európai Szabadalmi Hivatalnál (EPO) és a Szellemi Tulajdon Nemzeti Hivatalánál és elfogadás esetén érvényes az Európai Szabadalmi Egyezmény (EPC) azon tagországaiban, amelyek a bejelentésben szerepelnek
- PCT (Patent Cooperation Treaty) - a világ legtöbb országára [17] kérhető oltalom, az ügyben a WIPO (World Intellectual Property Organization) jár el

A következő fejezetben a számítógéppel megvalósított találmányok speciális vonatkozásait taglalom.

8.1. Számítógéppel megvalósított találmányok

A számítógépes program, azaz a szoftver, a szerzői jog tárgyköre, ki van zárva a szabadalmaztathatóság alól, ha csupán „szoftver, mint olyan” minőségében jelentik azt be. Ennek ellenére van lehetőség szoftvert szabadalmaztatni, de csak akkor, ha annak műszaki vonatkozásai is vannak, pl. ha az egy - amúgy szabadalmaztatható - találmány vagy eljárás része, vagy létezik valamilyen műszaki hatásuk, eredményük [18].

A diplomamunkám során egy fordítóprogramot fejlesztettem. Ha úgy vesszük, hogy a fordítóprogram egy bemeneti adathalmazt (szoftveres forráskód) egy kimeneti adathalmazzá (hardverleírás) konvertál, ebben a formájában csak adatkonverciónak, egy algoritmusnak tekinthető, következésképp nem szabadalmaztatható. Ha viszont egy olyan műszaki megoldásként (vagy eljárásként) tekintünk a fordítóprogramra, aminek segítségével hardvert fejleszthetünk, akkor ez a szoftverrel támogatott eljárás, legalábbis ebben a tekintetben, nincs kizárva a szabadalmaztathatóság alól.

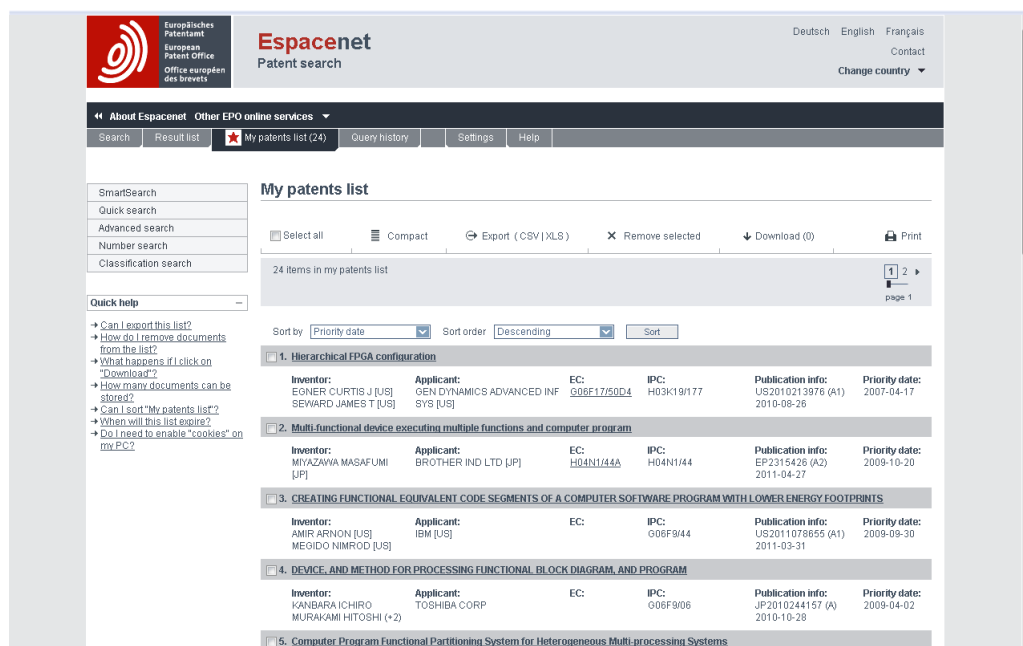
A műszaki jelleg könnyen belátható, képzeljük csak el, hogy a fordítóprogram segítségével fizikusok egyszerű módon implementálhatnak pl. részecskekutatót segítő feladatokat FPGA-s áramkörök formájában.

Mindez azt jelenti, hogy a diplomamunkám témájában előfordulhatnak elfogadott szabadalmak, amit a következő fejezetben alá is támasztok.

8.2. Keresés az adatbázisokban

A jelenleg használatos magyar iparjogvédelmi online adatbázis a PIPACS (Publikus IParjogvédelmi AdatbázisCSalád), igaz, már csak addig, amíg a jelenleg fejlesztés alatt álló új keresőrendszer ezt fel nem váltja.

Az első keresés során szabadalmak szövegében kerestem a diplomamunkámra vonatkozó egyszerű szavakat. A hardver szóra 39, a szoftver szóra 84, a számítógép szóra pedig 685



8.1. ábra. Európai online adatbázis - <http://ep.espacenet.com>

találat érkezett, ami nagyon csekély számnak tekinthető, főleg, ha figyelembe vesszük, hogy ezek közül a ténylegesen elfogadott találmányok száma a fenti számoknak csak töredéke.

A 39 hardveres találat közül logikai szintézis eljárásokat, vagy ahhoz közeli találmányokat kerestem. Ennek a feltételnek csupán egy találat felelt meg, a P0203904 ügyszámmal rendelkező „Eljárás algoritmikus funkcionális egység megvalósítására” című találmány, de ez sem kapott szabadalmi oltalmat. Összességében elmondható, hogy a témában országos szinten beadott szabadalmat kvázi nem lehet találni.

Következő lépésben a 8.1. ábrán demonstrált európai online adatbázisban kerestem, ami - véleményem szerint - felhasználói felületét tekintve sokkal ergonomikusabb, ezáltal a keresés is hatékonyabban végezhető. A portál két hasznos funkcióját ki is emelném, az egyik az a lehetőség, hogy a megtalált szabadalmi bejegyzést egyszerűen elmenthetjük egy listába, ha a jövőben még szükségünk lenne rá, a másik pedig a legutóbbi keresések eltárolhatósága.

Először itt is a diplomamunkám szempontjából fontos kulcsszavakra kerestem rá egyenként, aminek a találati eredménye:

- fpga: 1080
- hardware: 11777
- compiler: 1718
- synthesis: 67810
- pipeline: 25000

Az itt látható találati számok a magyar keresésnél tapasztaltakhoz képest nagyságrendekkel magasabbak, így a szűkítéshez szükséges a kulcsszavak egymással való kombinálása

is:

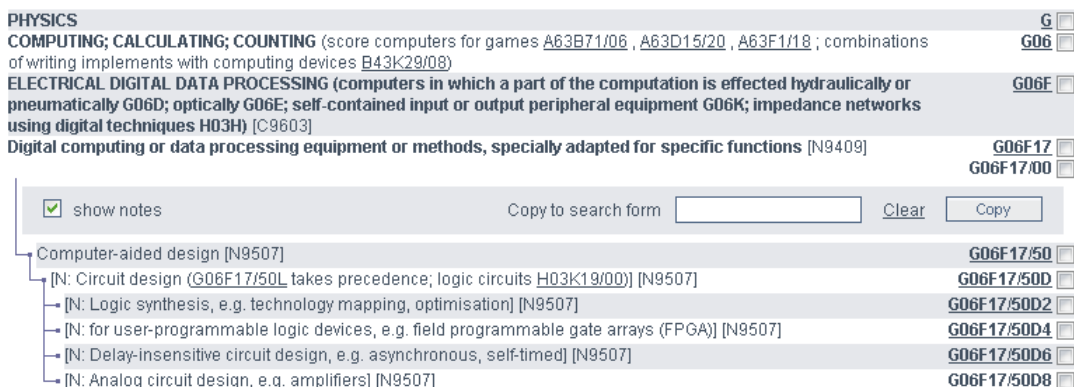
- hardware compiler: 20
- hardware synthesis: 38
- logic synthesis: 192, tovább szűkítve: logic hardware synthesis: 2
- high-level synthesis: 86, tovább szűkítve: high-level hardware synthesis: 4
- hardware software design: 34
- hardware compiler: 20

Ezután egy új keresési stratégiát alkalmaztam, ahol az ismert FPGA gyártók, ill. a magasszintű szintézis szoftvereket fejlesztő cégek találmányaira kerestem rá az applicant mezőt kitöltve:

- Xilinx (FPGA gyártó): 3137
- Altera (FPGA gyártó): 2953
- Lattice (FPGA gyártó): 1452
- Cadence (szoftverfejlesztő): 1485
- Mentor Graphics (szoftverfejlesztő): 647
- Altium (szoftverfejlesztő): 1

Az itt látható nagy számokból az derül ki, hogy az Altium kivételével a nagy világ-cégek jelentős energiát fordítanak a technológiájuk szabadalmaztatására. Ezek a cégek egymással is versengenek a szabadalmak terén, mert egy-egy jól megválasztott téma és kidolgozás esetén a konkurens cég eladásait nagy mértékben korlátozhatják. A két nagy FPGA gyártónál (Xilinx, Altera) különösen látható a fej-fej mellett haladás, a szabadalmi bejelentéseik száma csupán 6%-al tér el. Ahhoz, hogy a gyártók szerinti keresés eredményei is használhatóak legyenek, újabb keresési szavak megadásával szűkítettem a találatokat:

- Altera, synthesis: 14
- Xilinx, synthesis: 15
- Mentor Graphics, synthesis: 19
- Cadence, synthesis: 25
- Altera, compiler: 7
- Xilinx, compiler: 0
- Mentor Graphics, compiler: 0



8.2. ábra. G06F17/50D2 osztály

- Cadence, compiler: 0

A hardver szóra itt nem kerestem rá, hiszen mindegyik cég hardveres tervezőszoftve-
rekkel foglalkozik. Érdekes észrevétel, hogy a fordító (compiler) szót egyedül az Altera
használja, a többi cégnél ezt a funkciót a szintézis szó látja el.

A szabadalmi bejegyzéseket témájuk alapján osztályokba sorolják, így harmadik keresési
stratégiaként a releváns osztályok feltérképezését tűztem ki célul. A logikai szintézis és az
FPGA-s találmányok külön európai osztállyal (G06F17/50D2, ill. G06F17/50D4) rendel-
keznek, az előbbinek az osztályhierarchiát a 8.2. ábra szemlélteti. Ezek után a magasszintű
nyelvi fordítók osztályának (G06F9/45) vizsgálatára tértem rá.

Az osztályok és az azokon belüli szűkítések alapján történő keresés eredménye:

- G06F17/50D2 osztály (logikai szintézis): 1957
 - Xilinx: 23
 - Altera: 27
 - Mentor Graphics: 12
 - Cadence: 70
 - compiler: 10
 - synthesis: 168, tovább szűkítve: G06F17/50D2, high level synthesis: 17
 - software: 7
 - hardware: 29
- G06F17/50D4 osztály (FPGA-s találmányok): 1762
 - Xilinx: 343
 - Altera: 205
 - Mentor Graphics: 7
 - Cadence: 1
 - compiler: 6
 - synthesis: 14
 - software: 17
 - hardware: 107

- G06F9/45 osztály (magas szintű fordítók): 835
 - Xilinx: 0
 - Altera: 0
 - Mentor Graphics: 0
 - Cadence: 1
 - synthesis: 0
 - hardware: 2

Mindhárom keresési stratégiánál, ahol az adódó találati számok az 1-40 nagyságrendbe estek, ott a listák bejegyzéseit egyenként megvizsgáltam, és ebből kiválasztottam a diplomamunkámhoz legközelebb álló találatokat, amit a következő fejezetben kicsit részletesebben is ismertetek.

8.3. Szabadalmak a diplomatervem témájában

A keresés eredményeiből kiválasztottam az öt legrelevánsabb szabadalmi bejelentést, ezeket a most következő fejezetben röviden ismertetem is.

FPGA szoftveres programozása

- *eredeti cím*: System and method for programming the hardware of field programmable gate arrays (FPGAs) and related reconfiguration resources as if they were software by creating hardware objects
- *bejelentő*: Xilinx Inc.
- *elsőbbség napja*: 1998. január 26.
- *egyesült államokbeli oltalom*: US6230307B1

A témában általam talált, időrendben legelső szabadalmat az egyik legnagyobb FPGA gyártó, a Xilinx jelentette be.

A szabadalmazás igénypontjai nem magasszintű szintézis eljárásról szólnak, így a továbbiakban a beadvány bevezetőjének diplomamunkám szempontjából fontos részeire térek csak ki.

A szabadalmat bejelentő szerint a fejlesztők három nagy kihívással néznek szembe: igény növekvő számítási teljesítményre, gyorsabb kivitelezés szükségessége és a flexibilitás nagyfokú igénye. Az algoritmusok implementálása hardver és szoftver alapon történhet. A szoftver magas fokú rugalmasságot biztosít a fejlesztés során, ellenben fix processzoros architektúrán fut, így annak általános utasításai a teljesítményt visszafogják. A hardveres megoldásokat dedikáltan a feladatra tervezik, ami a teljesítmény szempontjából nagyon előnyös, ellenben sokkal alacsonyabb rugalmassággal és hosszabb fejlesztési idővel jár.

A bevezető ezen része nagyon szépen összefoglalja a diplomamunkám egyik motívációját, a szoftveres és hardveres megoldások közötti átjárási lehetőség igényét.

Eljárás magasszintű nyelvek hardverre fordítására

- *eredeti cím*: Means and method for compiling high level software languages into algorithmically equivalent hardware representations
- *bejelentő*: Xilinx Inc.
- *elsőbbség napja*: 2000. február 10.
- *egyesült államokbeli oltalom*: US6625797B1
- *PCT közzétételi szám*: WO0159593A2

Az itt leírt szabadalom bejelentője ugyancsak a Xilinx. Ahogy a bevezetőben olvasható, a jelenlegi hardveres fejlesztés hardverleíró nyelvekkel történik (pl. Verilog, VHDL). Ezek a nyelvek hatékonyak hardveres áramkörök leírására, de magasszintű absztrakciót nem biztosítanak, amivel a komplexitás menedzselhető lenne. Ezzel szemben a szoftveres nyelvek és azon belül az OOP nyelvek (pl. Java, C++) robosztus magasszintű konstrukciókat biztosítanak, ezek alapján nagyon hatékonyak a komplexitás kézbentartása tekintetében és a fejlesztés minőségében.

Az itt felsorolt kihívások irányították a figyelmet egyrészt a HDL nyelvek magasabb szintű konstrukciókkal, másrészt a magasszintű szoftveres nyelvek hardver-specifikus konstrukciókkal való kiterjesztésére, viszont egyik sem produkált elfogadható eredményeket. Az utóbbinál pl. az a probléma, hogy az absztrakciós szint a kiterjesztéssel könnyen eltűnhet és a nyelv lényegében egy új HDL nyelvként fog funkcionálni, ami nem volt cél.

A találmány a magasszintű szoftveres leírás hatékony, ekvivalens digitális hardverre történő fordításáról szól. Első lépésként adat- és vezérlésfolyam gráfokra fordítja a magasszintű nyelvet, ami közbülső formátumként optimalizációra, transzformációra ad lehetőséget. Az így adódó gráf egyaránt fordítható RTL vagy netlist szintű hardveres implementációra.

A leírás tartalmaz gráf optimalizálást, thread kezelést és azok szinkronizációját, és a kivételeket is kezelni tudó objektum-orientált kiindulóponti megoldást.

A lényegesebb igénypontok közül az első egy eljárást ír le, ez forráskódú specifikációból generál digitális logikai leírást, ami a következő lépéseket jelenti: forráskód fordítása irányított gráfokra, a gráfok összekapcsolása, aminek eredményeképp egy új gráf jön létre, az új gráf ellátása annotációkkal, ami a sorrendi hardverelemek beillesztési helyét jelzi, végül az eredménygráf hardveres reprezentációra történő fordítása.

Egy másik lényeges igénypontban az eljárás a forráskódból először bájtkódra fordít az általánosan ismert módokon, majd a bájtkódból generál CDFG (Control and Data Flow Graph) gráfot, innentől a lépések az előző igénypontnál analóg módon zajlanak.

A találmányt PCT-n keresztül bejelentették, viszont Magyarországot nem jelölték meg (a 81. és 84. sorszámú bejegyzésben a HU országcód nem szerepel)

Mivel az én eljárásomban is található irányított gráfra történő fordítás, ezért ez a szabadalom mindenképpen egy fontos kiindulópontja lehet egy újdonságkutatásnak.

Szoftver-hardver fordító

- *eredeti cím*: software-to-hardware compiler
- *bejelentő*: Altera Corporation
- *elsőbbség napja*: 2000. augusztus 7.
- szabadalomcsalád (részlet):
 - *PCT közzétételi szám*: WO0213004A3
 - *európai közzétételi szám*: EP1742159A3
 - *egyesült államokbeli közzétételi szám*: US2007169033A1
 - *egyesült államokbeli szabadalomszám*: US7257780B2

A diplomamunkám témájához az itt leírt szabadalom is nagyon közeli. A bejelentő az előző szabadalom bejelentőjének legnagyobb riválisa, azaz a másik nagy FPGA gyártó, név szerint az Altera Corporation. A bejelentésekor megjelölték Magyarországot, de az oltalom nem terjed ki hazánkra, mert a nemzeti szakaszt nem indították el. (a szabadalom családja nem tartalmaz magyar szabadalomszámot)

A bejelentő megpróbálta nagyon tág határokkal védeni alkotását: olyan találmányt jelentett be, amely közvetlenül magasszintű szoftveres kódból hardverkonfigurációt generál, amivel programozható logikai egységet (pl. FPGA) lehet konfigurálni.

Az 1-37. közötti igénypontok a dokumentum szerint törölve lettek, így az első említésre méltó a 38. sorszámmal ellátott igénypont, amely egyrészt a magasszintű szoftveres kód hardver szempontjából transzparens parsolásáról és az ezt követő fordításról szól. A szabadalommal többek közt olyan fordítási eljárást kívántak védeni, amely során biztosan végbement egy változóról vezetékhalmazra történő leképzés. Itt a halmaz egyik vezetéke jelzi, hogy a változó kiszámításra került, és a további vezetékek a változót reprezentálják.

Az 50. igénypontban előkerül a vezérlésfolyam. Feltételként az áll, hogy a futás-idejű döntés legkevesebb részben a vezérlésfolyam alapján kell történni.

Az 59. igénypontban a kifejezések újbóli használatának optimalizálása kerül előtérbe. Feltétel az, hogy létezik legalább egy ilyen kifejezés, amely egynél többször szerepel a program forráskódjában. Ezt követően az eljárás optimalizálja a hardver

konfigurációs adatot, hogy csak egyszer allokálja a magasszinten többször használt kifejezést. Az egyszer allokált erőforrások hardver példányoknak tekinthetők.

Az itt szereplő találmány egyik alapkoncepciója, hogy a változók vezetékekkel kerüljenek reprezentálásra. Ez véleményem szerint triviális lépés, de a szabadalom érvényes, így az hatással van a saját megoldásom során bemutatott ezzel analóg, változó-vezeték leképzésre.

A szabadalmi leíráshoz ábrák sokaságát fűzték hozzá, a lényegesebbeket címükkel együtt feljegyeztem:

- általános felépítés (1. ábra)
- konstansok reprezentálása műveleti blokként (5. ábra)
- egy összeadó reprezentálása műveleti blokként (10. ábra)
- hardveres blokkok és azok konkatenálása (12-13. ábra)
- elágazások kezelése (14. ábra)
- a vezérlésáramlás megjelenése (17. ábra)
- osztott RAM (24-25. ábra)
- pointerok leképzése (26-27. ábra)
- elágazás kezelése (28-32. ábra)

Fordító objektum-orientált nyelvről hardverre

- *eredeti cím*: a compiler for object-oriented hardware circuit design descriptions
- *bejelentő*: Sharp Kabushiki Kaisha (Japan)
- *elsőbbség napja*: 2003. június 18.
- *brit közzétételi szám*: GB2403032A

A szabadalmi bejelentés olyan eljárásról szól, amely egy magasszintű logikai szintézis rendszerben objektum-orientált kód alapján fordít, ami akár virtuális függvényhívást is tartalmazhat. A szabadalmi leírásban többek között a téma egyik alap szakirodalmára [31] is hivatkoznak.

A 21 igénypont közül a lényegesebbek leírásával folytatom a találmány ismertetését.

Az egyik lényeges igénypont magasszintű szintézis eljárás során rendszer-szintű objektum-orientált program fordításáról szól. Az eljárás metódusonként legalább egy paramétert hozzáad egy metódushoz a következők szerint:

- ha a metódus hozzáfér vagy frissít egy tagváltozót, akkor a paraméter típusa az adott tagváltozó típusa
- ha a metódus egy virtuális metódushívást tartalmaz, akkor a paraméter típusa a megfelelő osztály típusa
- ha az OOP program egy objektum memóriaterületére mutató referenciát tartalmaz, akkor a paraméter a megfelelő objektum memóriaterületet reprezentálja

Az 5. igénypont fogalmazza meg az eljárást, ami az előző igényekben leírt objektum-orientált programot RTL szintre fordítja.

Magas szintű nyelvi leírás a hardverleíró szintézis alkalmazásokban

- *eredeti cím*: Using high-level language functions in HDL synthesis tools
- *bejelentő*: Xilinx Inc.
- *elsőbbség napja*: 2006. június 9.
- *egyesült államokbeli oltalom*: US7574688B1

Eljárás, ami egyszerre biztosítja magasszintű nyelvi leírás (HLL, High-level Language) és hardverleírás (HDL, Hardware Description Language) használatát a magasszintű szintézis során.

Ennél a szabadalomnál csak az első igénypont leglényegesebb részeit fejtem ki, mert a többi a diplomamunkám szempontjából nem releváns. Ezek a részek olyan eljárást mutatnak be, ami HDL reprezentációt fordít, használva egy HLL függvényt. A HDL modul egyik attribútuma értékét egy HLL függvény kimenete adja, amit meghatározhatunk pl. úgy, hogy natív környezetben futtatjuk a HLL függvényt.

8.4. A saját megoldás szabadalmazhatósági kérdései

A diplomatervem során leírt magasszintű szintézis eljárás néhány részéhez találtam nagyon közeli szabadalmakat, így ezekre biztosan nem adható oltalom:

- változó reprezentációja vezetékként, ahol egy bit szolgál az adat érvényességének jelölésére, a többi a változó értéke (Altera, US7257780B2 közzétételi szám)
- adatfolyam-gráf reprezentáció használata a szintézis során (Xilinx, US6625797B1 közzétételi szám)

Az első egy érvényben lévő szabadalom, a második pedig csak bejelentés formájú eljárást, így ezekre biztosan nem igényelhető szabadalom.

Néhány részlet a saját módszereim közül, amiket az eddigi keresések során nem találtam meg szabadalmaztatott eljárásként, és ezek alapján a jövőben esetlegesen szabadalmaztatható:

- ADT struktúrák hardveres reprezentációja
- primitív függvények és EOG műveletek megfeleltetése
- a tömbök szintézisénel az adatutak szisztematikus átírása blokk-RAM kezelő EOG műveletekre

Irodalomjegyzék

- [1] <http://www.mentor.com/esl/catapult/overview>.
- [2] <http://www.impulseaccelerated.com/>.
- [3] <http://www.c-to-verilog.com/index.html>.
- [4] <http://www.comlab.ox.ac.uk/geraint.jones/ruby/>.
- [5] <http://haskell.org/ghc/docs/6.12.2/html/libraries/base-4.2.0.1/Prelude.html>.
- [6] <http://www.haskell.org/haskellwiki/Performance/GHC>.
- [7] <http://www.haskell.org/>.
- [8] <http://blog.bjrn.se/2008/10/lets-build-mp3-decoder.html>.
- [9] <http://www.xilinx.com/tools/sdk.htm>.
- [10] <http://eclipsefp.sourceforge.net/>.
- [11] <http://www.eclipse.org/Xtext/>.
- [12] <http://www.eclipse.org/modeling/emf/>.
- [13] <http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xtext.doc/help/metamodelInference.html>.
- [14] <http://www.eclipse.org/modeling/gmp/>.
- [15] <http://veditor.sourceforge.net/>.
- [16] <http://www.sztnh.gov.hu/szabadalom/>.
- [17] http://www.wipo.int/pct/en/list_states.pdf.
- [18] <http://www.sztnh.gov.hu/kiadv/ipsz/200406/01-hamori.html>.
- [19] 1995. évi xxxiii. törvény a találmányok szabadalmi oltalmáról. http://www.sztnh.gov.hu/jogforras/1995_XXXIII_Szt.pdf.

- [20] Vhdl tutorial. http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html.
- [21] Peter Arato, Visegrady Tamas, and Istvan Jankovits. *High Level Synthesis of Pipelined Datapaths*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [22] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7:265–301, May 1997.
- [23] C. Baaij. *Clash : from haskell to hardware*, December 2009.
- [24] Csák Bence. *Közvetlen hardver generálás magasszintű nyelvi leírásból*. PhD thesis, Budapest University of Technology and Economics, 2009.
- [25] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Static scheduling of multi-rate and cyclo-static dsp-applications. In *VLSI Signal Processing, VII, 1994., [Workshop on]*, pages 137 –146, 1994.
- [26] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34:174–184, September 1998.
- [27] Joseph Buck and Edward A. Lee. The token flow model, 1992.
- [28] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, ICFP '07*, pages 315–326, New York, NY, USA, 2007. ACM.
- [29] Stephen A. Edwards and Jia Zeng. Static elaboration of recursion for concurrent software. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '08*, pages 71–80, New York, NY, USA, 2008. ACM.
- [30] Simon Frankau. Hardware synthesis from a stream-processing functional language, 2004.
- [31] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
- [32] Shaori Guo and Wayne Luk. Compiling ruby into fpgas. In *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL '95*, pages 188–197, London, UK, 1995. Springer-Verlag.
- [33] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

- [34] P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM*, 4:55–58, January 1961.
- [35] ISO/IEC. Ieee 1076-2008: Ieee standard vhdl language reference manual. Technical report, Institute of Electrical and Electronics Engineers, Computer Society, 26. January 2009.
- [36] M. Kooijman. Haskell as a higher order structural hardware description language, December 2009.
- [37] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. See <http://llvm.cs.uiuc.edu>.
- [38] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.
- [39] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987.
- [40] Simon Marlow. Haskell 2010 language report, 2010.
- [41] Neil Mitchell. Rethinking supercompilation. *SIGPLAN Not.*, 45:309–320, September 2010.
- [42] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.
- [43] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12:393–434, July 2002.
- [44] Mary Sheeran. ufp, an algebraic vlsi design language - phd thesis, 1983.
- [45] S. Sriram and S.S. Bhattacharyya. *Embedded multiprocessors: scheduling and synchronization*. Signal processing and communications. CRC Press, 2009.
- [46] The GHC Team. The glorious glasgow haskell compilation system user’s guide, version 6.12.2.
- [47] Andrew Tolmach. An external representation for the ghc core language, 2009.
- [48] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8:292–325, June 1986.
- [49] Valery and Sklyarov. Fpga-based implementation of recursive algorithms. *Microprocessors and Microsystems*, 28(5-6):197 – 211, 2004.
- [50] Csörnyei Zoltán. *Lambda-kalkulus*. Typotex, 2007.

Függelék

F.1. Stream műveletek VHDL kódja

```
entity sum is
  generic(cl: integer; width: integer; delay: integer);
  port (in1: in  std_logic_vector ((width - 1) downto 0);
        q  : out std_logic_vector ((width - 1) downto 0);
        cl : in  std_logic_vector (cl downto 0);
        cq : out std_logic_vector ((cl - 1) downto 0);
        clk: in  std_logic);
end sum;

architecture Behavioral of sum is
  signal accum : std_logic_vector ((width - 1) downto 0) :=
    conv_std_logic_vector(0,width);
  signal dc: integer := 0;
  signal cdelay: std_logic_vector ((cl - 1) downto 0);
begin
  process(clk)
  begin
    if(rising_edge(clk))then
      if (cl(1)='1') then
        dc <= 0;
        cdelay <= cl(cl downto 1);
      else
        dc <= dc+1;
      end if;

      if (dc=delay-1) then
        q <= accum;
        cq <= cdelay;
      else
        cq <= conv_std_logic_vector(0,cl);
      end if;

      if (cl(1)='1') then
        accum <= in1;
      elsif (cl(0)='1') then
        accum <= accum+in1;
      end if;
    end if;
  end process;
end Behavioral;
```

F.1.1. ábra. Sum művelet VHDL modulja

```

entity enumfromto is
  generic(cl: integer; width: integer; rate: integer; delay: integer);
  port (in1: in std_logic_vector ((width - 1) downto 0);
        in2: in std_logic_vector ((width - 1) downto 0);
        q : out std_logic_vector ((width - 1) downto 0);
        c1 : in std_logic_vector ((cl - 1) downto 0);
        c2 : in std_logic_vector ((cl - 1) downto 0);
        cq : out std_logic_vector (cl downto 0);
        clk: in std_logic);
end enumfromto;

architecture Behavioral of enumfromto is

  signal dc: integer := 0;
  signal rc: integer := rate;
  signal cqh: std_logic_vector ((cl - 1) downto 0);
  signal qc: std_logic_vector ((width - 1) downto 0);

  begin

  cqh <= c1 and c2;
  q <= qc;

  process(clk)
  begin
    if(rising_edge(clk))then
      cq(cl downto 1) <= cqh;
      if (cqh(0)='1') then
        dc <= 0;
        rc <= 0;
        cq(0) <= '1';
        qc <= in1;
      else
        if (dc>=delay-1) then
          dc <= 0;
          if (rc=rate-1) then
            cq(0) <= '0';
            qc <= qc;
          elsif (rc>rate-1) then
            cq(0) <= '0';
            qc <= qc;
          else
            cq(0) <= '1';
            qc <= qc+1;
          end if;
          rc <= rc+1;
        else
          dc <= dc+1;
          cq(0) <= '0';
          qc <= qc;
        end if;
      end if;
    end if;
  end process;
end Behavioral;

```

F.1.2. ábra. *EnumFromTo* művelet VHDL modulja

F.2. Tömb műveletek VHDL kódja

```
entity nth is
  generic(cl:integer; width: integer);
  port (blockram: inout std_logic_vector ((width - 1) downto 0);
        in2: in std_logic_vector ((width - 1) downto 0);
        q : out std_logic_vector ((width - 1) downto 0);
        oe : out std_logic;
        c2 : in std_logic_vector (cl downto 0);
        cq : out std_logic_vector (cl downto 0);
        clk: in std_logic);
end nth;

architecture Behavioral of nth is

  signal cbuf : std_logic_vector (cl downto 0);

begin

process1: process (clk)
  begin
    if (falling_edge(clk)) then
      if (c2(0)='1') then
        oe <= '0';
        blockram <= in2;
      else
        oe <= '1';
        blockram <= (others => 'Z'); -- regex miatt TODO
      end if;
    end if;

    if (rising_edge(clk)) then
      cbuf <= c2;
      cq <= cbuf;
      if (cbuf(0)='1') then
        q <= blockram;
      end if;
    end if;
  end process;
end Behavioral;
```

F.2.1. ábra. *Nth művelet VHDL modulja*

F.3. Az Haskell nyelven implementált szintézis program fordítói naplója

```

GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package array-0.3.0.2 ... linking ... done.
Loading package containers-0.4.0.0 ... linking ... done.
Loading package bytestring-0.9.1.10 ... linking ... done.
Loading package Win32-2.2.0.1 ... linking ... done.
Loading package filepath-1.2.0.0 ... linking ... done.
Loading package old-locale-1.0.0.2 ... linking ... done.
Loading package old-time-1.0.0.6 ... linking ... done.
Loading package directory-1.1.0.0 ... linking ... done.
Loading package pretty-1.0.1.2 ... linking ... done.
Loading package process-1.0.1.5 ... linking ... done.
Loading package Cabal-1.10.1.0 ... linking ... done.
Loading package ghc-binary-0.5.0.2 ... linking ... done.
Loading package bin-package-db-0.0.0.0 ... linking ... done.
Loading package hpc-0.5.0.6 ... linking ... done.
Loading package template-haskell ... linking ... done.
Loading package ghc-7.0.3 ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[ 1 of 14] Compiling CoreVisiting   ( CoreVisiting.hs, interpreted )
[ 2 of 14] Compiling DotOutput      ( DotOutput.hs, interpreted )
[ 3 of 14] Compiling Typconv       ( Typconv.hs, interpreted )
[ 4 of 14] Compiling CorePrint     ( CorePrint.hs, interpreted )
[ 5 of 14] Compiling Common        ( Common.hs, interpreted )
[ 6 of 14] Compiling SearchCoreNode ( SearchCoreNode.hs, interpreted )
[ 7 of 14] Compiling VarRename      ( VarRename.hs, interpreted )
[ 8 of 14] Compiling AppRename      ( AppRename.hs, interpreted )
[ 9 of 14] Compiling MapConversion  ( MapConversion.hs, interpreted )
[10 of 14] Compiling IterateConversion ( IterateConversion.hs, interpre
[11 of 14] Compiling CaseReduction  ( CaseReduction.hs, interpreted )
[12 of 14] Compiling SimplCore     ( SimplCore.hs, interpreted )
[13 of 14] Compiling Flpc          ( Flpc.hs, interpreted )
[14 of 14] Compiling Main           ( Main.hs, interpreted )
Ok, modules loaded: SimplCore, Main, Flpc, CaseReduction,
IterateConversion, MapConversion, AppRename, VarRename, Common,
CorePrint, Typconv, DotOutput, SearchCoreNode, CoreVisiting.
*Main> Loading package transformers-0.2.2.0 ... linking ... done.
Loading package mtl-2.0.1.0 ... linking ... done.
Loading package regex-base-0.93.2 ... linking ... done.
Loading package regex-posix-0.94.4 ... linking ... done.
Loading package regex-compatible-0.93.1 ... linking ... done.
Loading package time-1.2.0.3 ... linking ... done.
Loading package random-1.0.0.3 ... linking ... done.
Loading package haskell98-1.1.0.1 ... linking ... done.
Loading package ghc-paths-0.1.0.8 ... linking ... done.
Loading package HUnit-1.2.2.3 ... linking ... done.
Loading package parsec-3.1.1 ... linking ... done.
Loading package network-2.3.0.2 ... linking ... done.
Loading package hslogger-1.1.5 ... linking ... done.
Loading package MissingH-1.1.0.3 ... linking ... done.
```

Tárgymutató

- α -konverzió, 10
- β -redukció, 10
- η -konverzió, 10
- összetett függvény, 23
- újraindítási idő, 52

- adatifolyam, 23
- alkalmazás, 10
- AppRename, 38
- argumentum azonosító, 40
- AST, 13

- backend, 31
- behelyettesítés, 16
- blokk, 50
- blokk sajátfrekvenciája, 50
- BNF, 79
- boxing, 15

- C λ asH, 7
- call-by-name, 11
- call-by-need, 11
- call-by-reference, 11
- call-by-value, 11
- CaseReduction, 34
- CoreToTree, 38
- csomagolt típus, 15
- csupasz típus, 15

- egysebességű (single-rate) adatfolyam, 21
- elágazás, 33
- elemi függvény, 23
- EOG, 21
- erősen típusos, 9

- függvény, 10

- frontend, 31

- GenerateVHDL, 44
- globális azonosító, 40
- gyorsító függvény, 54

- IterateConversion, 37

- kötött/szabad változó, 10

- lambda lifting, 10
- lambda-absztrakció, 10
- lambda-kifejezés, 10
- lappangási idő, 52
- lassító függvény, 54
- latency, 24
- lokális azonosító, 40
- lusta kiértékelésű (lazy evaluation), 9

- művelet, 23
- művelethalmaz, 21
- MakeModuleList, 43
- MapConversion, 57
- mintaillesztés, 33
- MREOG, 50

- OpTree, 38

- ParseEOG, 43
- polimorf típus, 9
- ProduceComponents, 43
- ProduceOperations, 44
- ProduceSignals, 43

- saját művelet, 50
- scrutinee, 14
- SDF, 21
- SFB, 71

stream, 54
strukturális leírás, 18
supercompiling, 38

többsebességű (multi-rate) adatfolyam, 49
típusalternatíva, 12
típusosztály, 12
tisztán funkcionális (pure functional), 9
TreeSimplifier, 42

unboxing, 15

változó, 10
viselkedési leírás, 18