

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Konzulens

BUDAPEST, 2018

Tartalomjegyzék

[Kivonat 6](#_Toc531814017)

[Abstract 7](#_Toc531814018)

[1 Bevezetés 8](#_Toc531814019)

[1.1 PipeComp keretrendszer 8](#_Toc531814020)

[1.2 HLS Backend 9](#_Toc531814021)

[2 Megismert technológiák 10](#_Toc531814022)

[2.1 VHDL hardverleíró nyelv jellegzetességei 10](#_Toc531814023)

[2.1.1 Felépítés 10](#_Toc531814024)

[2.2 Eclipse alapú modellezési technológiák 13](#_Toc531814025)

[2.2.1 Eclipse project 13](#_Toc531814026)

[2.3 Template-nyelvek 15](#_Toc531814027)

[2.3.1 Összehasonlítás szempontjai 15](#_Toc531814028)

[2.3.2 Freemarker 16](#_Toc531814029)

[2.3.3 StringTemplate 17](#_Toc531814030)

[2.3.4 Xtend 19](#_Toc531814031)

[2.3.5 Konklúzió 20](#_Toc531814032)

[3 Specifikáció 22](#_Toc531814033)

[3.1 HIG adatmodell 22](#_Toc531814034)

[3.2 Elvárások a generátorral szemben 24](#_Toc531814035)

[4 Architektúra 26](#_Toc531814036)

[4.1 VHDL könyvtár 27](#_Toc531814037)

[4.1.1 BasicEntityReader 27](#_Toc531814038)

[4.2 HIG olvasó 27](#_Toc531814039)

[4.3 HIG Component és a VHDL kapcsolata 27](#_Toc531814040)

[4.4 HIG előfeldolgozó 28](#_Toc531814041)

[4.5 VHDL builder 28](#_Toc531814042)

[4.5.1 HIG Generátor 28](#_Toc531814043)

[4.5.2 Loop Generátor 29](#_Toc531814044)

[4.5.3 Selection Generátor 30](#_Toc531814045)

[4.5.4 Tömbkezelés 31](#_Toc531814046)

[4.6 Kimenet felügyelő 34](#_Toc531814047)

[5 Fejlesztés 35](#_Toc531814048)

[5.1 VHDL könyvtár 35](#_Toc531814049)

[5.1.1 Alap műveletek kezelése 36](#_Toc531814050)

[5.2 HIG olvasó 36](#_Toc531814051)

[5.3 VHDL generátorok 36](#_Toc531814052)

[5.3.1 HIG Generátor 37](#_Toc531814053)

[5.3.2 Loop Generátor 38](#_Toc531814054)

[5.3.3 Selection Generátor 39](#_Toc531814055)

[6 Tesztelés 41](#_Toc531814056)

[6.1 HIG 42](#_Toc531814057)

[6.2 LoopComp 43](#_Toc531814058)

[6.3 SelComp 44](#_Toc531814059)

[6.4 Tömbkezelés 44](#_Toc531814060)

[7 Összefoglalás 45](#_Toc531814061)

[7.1 Az általam elért eredmények 45](#_Toc531814062)

[7.2 Továbbfejlesztési lehetőségek 45](#_Toc531814063)

[8 Köszönetnyílvánítás 47](#_Toc531814064)

[Irodalomjegyzék 48](#_Toc531814065)

Hallgatói nyilatkozat

Alulírott **Kovácsvölgyi Dávid**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet (nem kívánt törlendő) 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 hitelesített 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.

Kelt: Budapest, 2018. 12. 05.

...…………………………………………….

Kivonat

Egyre nagyobb az igény, hogy általános célú, magas szintű programozási nyelvekből is lehessen hardvert szintetizálni, minimális beavatkozással. A magas szintű szintézis (HLS, High-level synthesis) sok esetben gyorsabb és kevesebb költségekkel járó hardverfejlesztést tesz lehetővé.

Ilyen rendszer az Irányítástechnika és Informatika Tanszéken fejlesztett PipeComp, amely magas szintű nyelvekből hardverleírást generál. A PipeComp programozási nyelveket feldolgozó frontendekből és hardverleírást generáló backendekből áll. A PipeComp köztes reprezentációja a HIG (HLS Intermediate Graph) adatfolyamgráf.

A dolgozat témája a PipeComp részeként egy VHDL backend fejlesztése, ami HIG adatfolyamgráfról fordít. A szoftver alapvetően Java nyelven készítettem, de a HIG modell értelmezésére EMF (Eclipse Modeling Framework) keretrendszert és a VHDL kód generálásához Xtendet használtam, mint sablonnyelv.

A dolgozatom ismerteti a VHDL hardver leírónyelv sajátosságait és kódgenerálásra alkalmas sablonnyelveket továbbá bemutatja a HIG adatfolyamgráf modellezésére használt frameworkot. Leírja a fejlesztendő szoftver pontos specifikációját, a szoftver tervezésének lépéseit, az implementáció fontosabb részleteit, és a szoftver tesztelésének lépéseit is bemutatja.

Abstract

There is a growing need for applications that can synthesis harware from high level programming languages. The HLS (High-level Synthesis) is faster and cheaper than the actual hardwaredevelopment in most cases.

One of the implementation of these applications is the PipeComp system which was developed by the Department of Control Engineering and Information Technology department. The PipeComp has frontends which processes high level lenguages into the common HIG dataflow graph, and it has backends that generates hardware definition (e.g. VHDL, Verilog, etc.) from HIG.

In this paper I would like to document an implementation of a PipeComp backend which creates VHDL from dataflow graph. The software has been written in Java and for reading the HIG model as input it uses EMF (Eclipse Modeling Framework) and it generates the output using Xtend as a template language

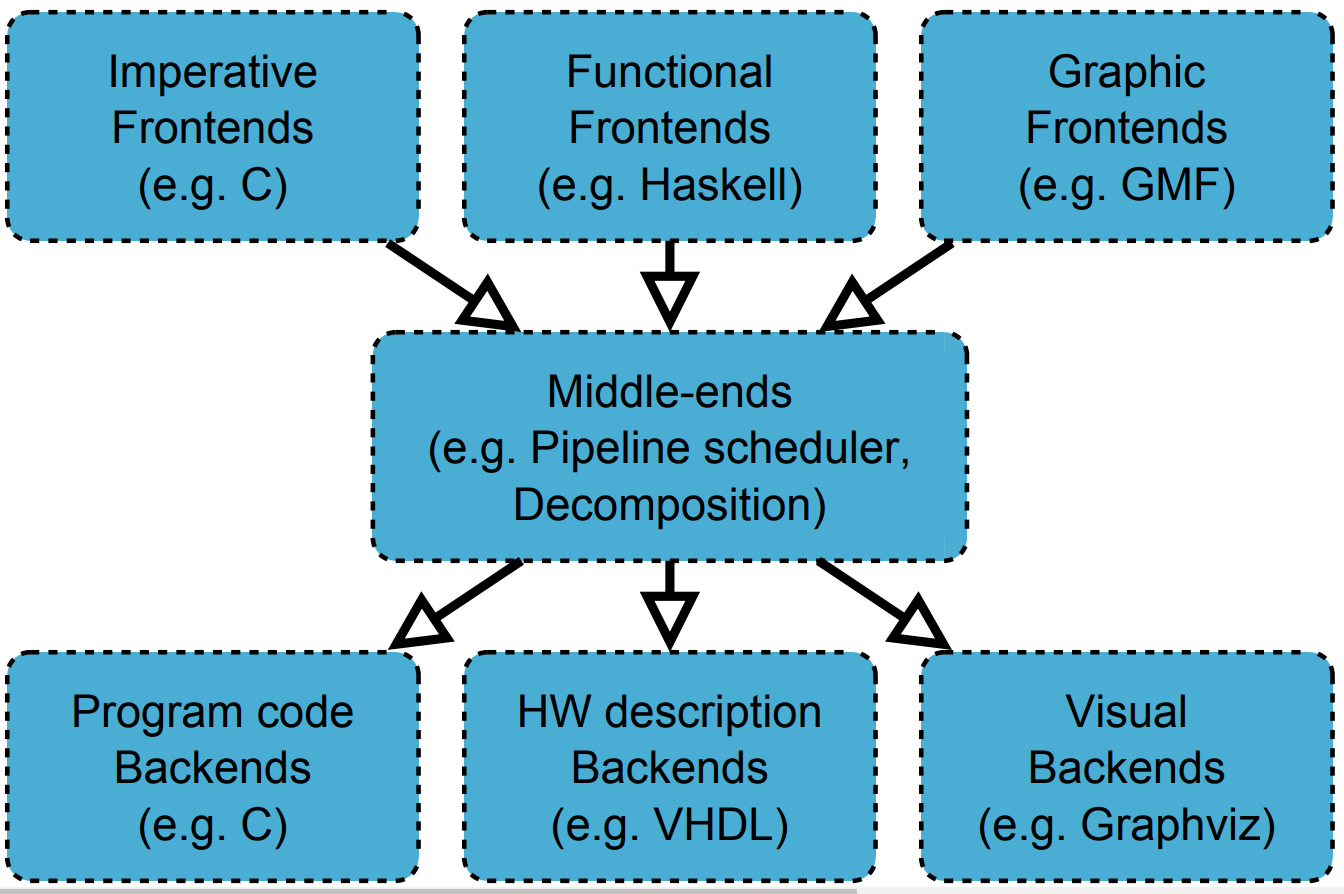
My paper will review the VHDL hardware definition language and the template languages which can generate code easily. It will help understand the framework used for modelling HIG, and provides documentation about the architecture and implementation of the software. It will reveal how the testing and planning was done.

# Bevezetés

Mivel egy áramkörnek a kapuszintű tervezése lassú és nagy odafigyelést igényel, ezért régóta foglalkoztatja a mérnököket, hogy a tervezést egy magasabb absztrakciós szinten tudják elvégezni, ahol a rendszer viselkedését képesek előírni, és ebből hardverleírást generálni egy arra alkalmas fordító segítségével. Az első generációt a Synopsys cég készítette el 1994-ben Behavioral Compiler néven, ami VHDL-t és Verilogot használt bemeneti nyelveként. Viszont az eszközök, amik viselkedést leíró VHDL-t és Verilogot alkalmazták nem terjedtek el, mert a nyelvek nem voltak alkalmasak magasszinten való algoritmus leírásra, ezért 2004-ben leállították a Behavioral Compiler támogatsát és tovább fejlesztését. Eközben megjelent az igény egy ténylegesen magas szintű nyelvet (pl. C-t) használó szintézer kifejlesztésére, ezt hívják ma magas szintű szintézisnek [11].

## PipeComp keretrendszer

A BME IIT tanszékén folyó HLS projekt célja egy magasszintű szintézis keretrendszer megalkotása. Ennek neve a PipeComp[7], ami a sok bemeneti és kimeneti nyelv és formátum támogatása érdekében 3 rétegű architektúrát valósít meg (1‑1. ábra). A bemenet feldolgozásáért a frontend réteg felelős, ami egy köztes, algoritmust leíró adatmodellbe fordítja, ez az ún. HIG adatfolyamgráf. A middle-end végrehajt bizonyos nyelvfüggetlen optimalizációs lépéseket, illetve különböző transzformációkat hajt végre az adatfolyamgráfon. A backend réteg végzi az adatfolyamgráf feldolgozását, és kívánt nyelvre való fordítását.



1‑1. ábra PipeComp keretrendszer architektúra[7]

## HLS Backend

A feladatom egy VHDL-t generáló HLS Backend tervezés és implementálása volt, amivel alapműveleteket (összeadás, szorzás, osztás, stb), vezérlési szerkezeteket (összetett művelet, elágazás, ciklus) és memóriát kellett tudnom kezelni. Az általam elkészített program bemenetként a HIG adatfolyamgráfot fogadja, bejárja és feldolgozza. Végezetül VHDL leírókat generál, amik alkalmasak a HIG által leírt algoritmus végrehajtására.

# Megismert technológiák

Ebben a fejezetben szeretném leírni a tapasztalataimat a feladatom elvégzése közben megismert technológiákat.

## VHDL hardverleíró nyelv jellegzetességei

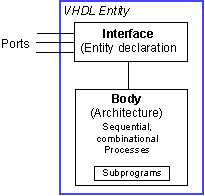
A VHDL (VHSIC Hardware Description Language) az egyik legelterjedtebb hardver leíró nyelv napjainkban. A hardver leíró nyelvek céljukat tekintve eltérnek magasabb szintű szoftveres nyelvektől, mert utóbbiak a fordítás után a processzorban futnak, ami sorosan hajtja végre a fordított kód utasításait. Míg a hardver leíró nyelvek esetén az általa definiált modulok egymás mellett párhuzamosan működnek. Tehát nem egy algoritmus lépésről lépésre történő végrehajtásával, hanem annak elemi blokkjaival és azok összeköttetéseivel írják le azt.

### Felépítés

Főleg FPGA és ASIC áramkörök tervezésére és szimulálására használják a hardverleíró nyelveket, mivel a bonyolult és ebből fakadóan nehezen áttekinthető kapcsolásirajzokkal szemben a hardverleíró nyelvek gépek és ember által is jól olvasható kódot eredményeznek. Az áttekinthetőséget növeli az is, hogy a VHDL lehetőséget ad több modul definiálására és ezek egymásba ágyazására is.

#### Entity és Architecture

A modulok két fő részből állnak (2‑1. ábra). Az egyik az ún. Entity ami a modul a környezete felé mutatott tulajdonságait tartalmazza, ilyenek a modul ki- és bemenetei (továbbiakban port) illetve azok típusa és bitszáma. Opcionálisan tartalmazhat generikus paramétereket is, ezeket tényleges értékek helyett felhasználhatjuk a modul funkcionális leírásában és a portjai bitszámát is megadhatjuk generikus paraméter formájában. A modul példányosítása során megadhatjuk milyen értéket szeretnénk a generikus paraméterek helyére behelyettesíteni, így azonos logikájú modulokat nem szükséges többször újra megírni. A másik fő rész az ún. Architecture, ez a modul működését és belső összeköttetéseit írja le, itt definiálhatók a modul almoduljai is, ezekre component-ként hivatkoznunk kell az almodulok Entity részében definiált interfész pontos megadásával. Ezután már többet is példányosíthatunk az adott modulból ki- és bemeneteire kötött vezetékek és generikus paramétereinek megadásával.



2‑1. ábra A VHDL strukturális felépítése[12]

#### Példányosítás

A modulokban definiálhatunk más modulokat komponensként, amennyiben szükségünk van egy másik modul funkcionalitására, így a VHDL kód sokkal strukturáltabbá, áttekinthetőbbé válik, és lehetőségünk nyílik egyes modulok újra felhasználására. Akár több példányt is létrehozhatunk ugyanabból a modulból miután felvettük a komponens listára az entity-jében megadott portokat és generikus paraméterek pontos másolatát. A példányosításkor meg kell adni a generikus paraméterek pontos értékét, továbbá a komponens példány portjaihoz kell rendelni a kívánt signalokat vagy portokat (2‑2. ábra).

2‑2. ábra Példányosítás

add1 : add

PORT MAP(rst, rst\_loop, clk, input1, input1\_rdy, input2, input2\_rdy, signal\_output, signal\_output\_rdy);

#### Főbb adattípusok és konstansok

A VHDL szabványban összesen hat egyszerű adattípus létezik és ebből csak három szintetizálható a végleges áramkörbe (a többit inkább tesztelésnél használhatók), ezek a következők:

* BIT: 0,1 értéket vehet fel
* BOOLEAN: true, false értéket vehet fel. Az előző típustól a hordozott információ tekintetében nem tér, az áttekinthetőség érdekében használják.
* INTEGER: 32 bites előjeles számot reprezentál

Ezeken felül általánosan használatos egy külön importálandó könyvtár, ami a IEEE 1164-es szabványban [1] rögzítettek. Ez az std\_logic és std\_logic\_vectort tartalmazza az első egy adattípus a másik annak vektorba fűzött formája. A std\_logic és a bit között az a különbség, hogy a std\_logic 0 és 1 értékeken kívül más értékeket is felvehet: ’U' – Uninitialized, 'X' - Forcing Unknown, 'Z' - High Impedance, 'W' - Weak unknown, 'L' - Weak 0, 'H' - Weak '1', '-' - Don't care. Ezek közül csak a ’Z’ érték kerül szintetizálásra a végső áramkörben, a többi a modulok szimulációjának és a tesztelésének könnyítése miatt van jelen.

A modulokban definiálhatók konstans értékek is a megfelelő adattípus megadásával (2‑3. ábra), illetve értékük megadásával. Ezek később értékül adhatók portoknak és signaloknak.

constant constant\_5 :

STD\_LOGIC\_VECTOR (31 downto 0) := "00000000000000000000000000000000";

2‑3. ábra Konstans definiálása

#### Signal

A signal a modulon belüli „vezetékeket” reprezentálja, a meghajtó és fogadó között. A vezetéket egyszerre csak egy forrás tudja meghajtani adattal, de több nyelőt is köthetünk az adatok fogadására. A forrás és nyelő bármi lehet, ami a signal definiálásakor megadott adattípust ad eredményeként vagy vár bementeként (signal definiálásra példa: 2‑4. ábra). Bár egyszere csak egy forrás tudja meghajtani a signalt, de ez a forrás időben változtatható pl: process segítségével.

signal signal\_out\_9 : STD\_LOGIC\_VECTOR (31 downto 0);

signal signal\_out\_9\_rdy : BOOLEAN;

2‑4. ábra Signal definiálása

#### Processek

A process a modul elvárt viselkedését írja le. A processben leírt viselkedést egy végtelen ciklus sorról sorra hajtja végre, ezt a wait kulcsszó használatával megállíthatjuk és újra indulását egy feltételhez köthetjük. Továbbá a ciklustörzs lefutását egy a modulban deklarálttag (signal, port) változásához is köthetjük, ha azt a process sensitivity listájába felvesszük. Ezt mutatja a 2‑5. ábra ahol mindkét process ugyanazt a működést írja le, feltéve, ha statements helyére ugyanazokat a kifejezéseket írjuk.

|  |  |
| --- | --- |
| sensitivity\_list : PROCESS (A, B)  BEGIN  --statements  END PROCESS; | wait\_statement : PROCESS  BEGIN  --statements  WAIT ON A, B;  END PROCESS; |

2‑5. ábra Sensitivity list és wait utasítás példa

## Eclipse alapú modellezési technológiák

Az Eclipse Foundation[3] egy non profit, a tagjai által támogatott cég, amit 2004 januárjában hozták létre, hogy segítse és irányítsa az Eclipse közösség munkáját. Ez a közösség mára már több nyíltforráskódú project fejlesztéséért felelős, ilyen az Eclipse Foundation megalakítása előtt már elindult Eclipse project is.

### Eclipse project

Az Eclipse project[4] egy nyíltforráskódú többnyire java alapú projekt, amit az IBM hozott létre 2001-ben az Eclipse SDK fejlesztése és karbantartása céljából. 2004-ben a projectvezetését a többek között erre a célra létrehozott Eclipse Foundation vette át, és a mai napig ez a szervezet egyengeti útjait. Mára a project jelentősen kinőte magát, és az idő előre haladtával már inkább mint "Eclipse top-level project"-ként hivatkoznak rá a projectek résztvevői, mivel mostanra már több alprojectre bontották a nagy projectet. Jelen pillanatban öt alproject fut egymás mellett, ezek a Platform, a Java development tools, a Plug-in Development Environment, E4 és az Orion.

#### Eclipse platform

Az Eclipse platform[5] nem más, mint olyan keretrendszerek és szolgáltatások összessége, amik alkalmazások elkészítését segítik, mert így minden, ami a fejlesztés szükséges egy helyen elérhető vagy beszerezhető. A platform támogatja a külső szolgáltatások/keretrendszerek integrációját pluginek formájában így minél jobban testre szabható és felhasználóbarát élményt nyújt. A pluginek megoldást nyújthatnak különböző, a fejlesztés közben felmerülő problémákra.

#### Eclipse Modelling Framework

A rendszerek modellezésénél mindig is nagy problémát jelentett a modell megalkotása és módosítása után a modellt reprezentáló kód elkészítésének időigénye és az a tény, hogy az így készülő boiler-plate kód megírása több lehetőséget adott az emberi hibázásra. Tehát a fejlesztési idő és a hiba lehetőségének csökkentésére szükségünk van egy olyan megoldásra, ami mindezt magából a modellből kiindulva automatikusan elő tudja állítani. Erre több megoldás is létezik, az egyik ilyen az Eclipse Foundation által kínált EMF (Eclipse Modelling Framework). A modellünk definiálása után az EMF elkészíti a modellnek megfelelő java osztályokat és hozzájuk tartozó utility osztályokat is, így nekünk erre nem kell időt szakítanunk.

Az EMF segítségével van lehetőségünk egy grafikus felület segítségével definiálni magas szintű modellünket, az így létrehozott modellt az Ecore modell írja le. Az Ecore egy metamodell ami a strukturális felépítését és lehetséges tulajdonságait tartalmazza az Ecore segítségével létrehozható példánymodellnek. Ezen példánymodellek létrehozását az EMF az Ecore alapján generálható szerkesztővel is segíti.

Az Ecore metamodell főként az alábbi entitásokból áll:

* EObject: Minden EMF által generált osztály az EObject leszármazottja. Igazából a java.lang.Object EMF reprezentációjaként tekinthetünk rá.
* EPackage: Ez tartalmazza a modelleket a legfelső szinten, a modellek elemei egy EPackage példányhoz tartoznak.
* EClass: Ezt tekinthetjük az osztályok reprezentációinak. Egy Java interfész és azt implementáló osztályként generálódik le modellünkből.
* EAttribute: egy attribútumot modellez, amelynek neve és típusa van.
* EReference: Két osztály közötti kapcsolatot reprezentálja, tartalmazza a kapcsolat multiplicitását és nevét. Ezen felül asszociációt és aggregációt is jelképezhet.

Ezeket a komponenseket szemlélteti a 3‑1. ábra, amin a HIG adatfolyamgráf Ecore reprezentációja látható. A példánymodelleket fa struktúrában tárolja el az EMF. Ezek bejárására az EMF által generált Switch osztály és Iterator segítségével van lehetőség, az Iterator bejárja a referenciákon keresztül a modellt és EObject formájában adja vissza a következő elemet. Az *instanceof* operátor használata helyett az EObjecteket a Switch osztály segítségével tudjuk azonosítani, aminek minden modellben található osztályhoz tartalmaz egy case metódust, amit felülírva tudjuk a saját algoritmusunkat érvényre juttatni.

## Template-nyelvek

A Java-ba beépített a szövegek dinamikus generálására nyújtott lehetőségek meglehetősen kezdetlegesek, mert nem hosszú szövegek generálásra találták ki őket, ezért nehezen olvasható kódot eredményeznek, és ebből fakadóan nehéz használni is őket erre a célra. Ezért szükséges volt egy template-nyelvet keresnem, ami képes egy nagyobb statikus szövegbe beágyazódni, így dinamikusan változtatni egyes részein. Erre több módszert kipróbáltam ezek leírása következik.

### Összehasonlítás szempontjai

#### Java és Eclipse támogatás

Mivel a Javat választottam a program elkészítéséhez, a template-nyelvnek is szükséges támogatnia azt.

Továbbá a HIG adatmodell egy Eclipse-s keretrendszerben lett modellezve, ezért adott volt az Eclipse fejlesztőkörnyezet használata is. Így egy olyan template-nyelvet szerettem volna választani, ami Eclipse-ben megfelelően támogatott.

#### Listák, ciklusok és Elágazások

A HIG modellben sok listás elem található, ezért előnyös, ha a nyelv támogatja azok bejárását. A ciklusok támogatottsága is fontos szempont volt számomra, mert szeretnék minél több lehetőséget a végtermék alakításán.

Az elágazások segíthetnek bizonyos opcionális karakterek kiírásában, vagy teljes kódsorok megváltoztatásában valamilyen feltétel szerint.

#### A templatenyelv kifejezései

A belső függvények támogatottságát is figyelembe szeretném venni, mivel sokszor kell majd a kapott adaton műveleteket végrehajtanom, és jobbnak ítélem meg, ha azt helyben a template-ben tudom végrehajtani.

### Freemarker

A Freemarker[8] egy ingyenes Java alapú template motor, eredetileg dinamikus weboldalak generálására készítették, mivel független a webes technológiáktól. Gyakran használják forrás file-ok és e-mailek generálására is.

#### Java és Eclipse támogatás

Található Eclipse plugin a FreeMarkerhez, ami szintaktikai kiemelést, ellenőrzést és kódkiegészítő funkcióval is rendelkezik. Használata viszonylag egyszerűbb, de a template változóit kívülről egy Map segítségével kell a template-be juttatni. Minden használatnál fel kell olvasni a háttértárolóról a tisztán szöveges template file-t ezzel nem elhanyagolható overheadet eredményezve más binárisra fordított template nyelvekkel szemben.

#### A templatenyelv kifejezései

A template kifejezéseket elhatároló eleme a ${KIFEJEZÉS}, az ilyen szerkezetbe írt kifejezések a template kiértékelése során a kívülről adott adatok alapján kapnak értéket. Akár objetumokat is átadhatunk, amelyek függvényeihez és belső adatstruktúráihoz is hozzáférhetünk.

#### Listák és ciklusok

A Freemarker támogatja a listák és ciklusok kezelését is, végig iterálhatunk rajtuk és az éppen soron következő elemhez, mint az objektumokhoz férhetünk hozzá. A listákat a <#list *listaneve* as *objektumneve*> kifejezéssel kezelhetjük.

<#list lista as elem>

Tetszőleges szöveg${elem.attribute1} - ${elem.attribute2}

</#list>

2‑6. ábra Freemarker lista kezelés

A ciklusos működést is <#list…> kifejezéssel tudjuk előhívni, itt valójában egy szám listán iterálunk végig ezzel imitálva az n iterációig tartó ciklikus működést.

#### Elágazások

Ha csak feltételesen van szükségünk egy adott szövegrészre, lehetőség van elágazások használatára is, ezzel azt egy bizonyos feltételhez köthetjük a template-n belül. Az elágazás leírásához a <#if…> kifejezéssel használhatjuk, végezetül pedig </#if> záró kifejezéssel tudjuk megadni a végét. Az if kifejezés után használható a <#elseif…>,<#else>, mint *egyébként ha* és *egyébként* ágak kezelésére.

<#if x == 1>

x értéke 1.

<#elseif x == 2>

x értéke 2.

<#else>

x nem 1, x nem 2.

</#if>

2‑7. ábra Freemarker elágazás kezelés

### StringTemplate

A StringTemplate[10] egy java template motor, amit forrásfájlok, weboldalak, emailek és más formázott szövegek előállításához használnak. Célkitűzései között szerepel a modell és a nézet (model-view) teljes szeparációja, ezért a template nem engedi a modellt befolyásoló műveletek végrehajtását a template belső kifejezései között. Így a szintaxisa letisztult és egyszerű így könnyen átlátható, ebből fakadóan tudástára szerény, de készítője azt állítja ennél több nem is kell.

#### A templatenyelv kifejezései

Mivel a modell és a nézet teljes szétválasztására törekedtek a nyelv kialakítása során, így minden adatnak rendelkezésre kell állnia a template kiértékelésekor, ezért a nyelv nem enged meg külső függvényhívásokat vezérlőszerkezeteiben, csak a modellben szereplő objektumok és azok attribútumaihoz tudunk hozzá férni. A nyelv a <,> jelek közé írt kifejezéseket értelmezi fordításkor.

#### Java és eclipse támogatás

Mivel ez egy java alapú template motor így kifinomult java könyvtárral rendelkezik, ami némi utána olvasás után egyszerűen használható. A template belső változóit (modell) a java API-val tudjuk „injektálni” a template számára. Ellentétben a FreeMarkerrel a StringTemplate fájljait binárisra tudjuk fordítani ezzel növelve a hatékonyságát a szöveg generálásnak.

Az eclipse-hez ugyan találtam plugint, de azt nem sikerült kipróbálnom technikai nehézségek miatt. Így ezekután StringTemplate-t eclipse-ben nem megfelelően támogatottnak ítéltem meg.

#### Listák és ciklusok

Klasszikus ciklusok nem támogatottak a nyelv előbb említett alapelve miatt, hiszen modellnek már rendelkezésre kell állnia mire fordítás megkezdődik, így a következő iteráció megkezdése előtt vagy után nincs lehetőség a ciklus feltételben szereplő kifejezéseket kiértékelni.

Ez a hiányosság áthidalható listák használatával, hiszen listákon való iteráció támogatott. Ezt a template nevű elem segítségével lehet elérni, ami lehet anonim vagy előre definiált. A templateknek bemenetként előre meghatározott számú listát várnak. A listák elemeit 2 fajta módon tudja kezelni, vagy minden elemhez külön-külön legenerálja a hozzátartozó stringet, vagy a template-ben az adatnak kihagyott lyukat az lista elemeinek felsorolásával tölti ki.

#### Elágazások

2‑8. ábra StringTemlate template elemének szintaxisa és használata

parens(x) ::= "(<x>)" <!template definíció!>

["a", "b", "c"]:parens() -> (a)(b)(c) <!!>

parens(["a", "b", "c"]) -> (abc)

<["a","b"]:{v | <v>=<i>;}> -> a=1;b=2; <!anonim template!>

<["a","b"]:{v | <v>=<i>;}>

Az elágazások is támogatottak StringTemplate-ben, használatuk hasonló a FreeMarkerben bemutatotthoz, itt az elágazást a <if…> kifejezéssel kezdhetjük, illetve a <endif> kifejezéssel kell zárni azt. Továbbá itt is van lehetőség *egyébként* és *egyébként ha* ágak kezelésére az <else> és <elseif…> kifejezések használatával.

2‑9. ábra Elágazás elemének szintaxisa

<if(boolexpr1)>subtemplate

<elseif(boolexpr2)>subtemplate2

...

<elseif(boolexprN)>subtemplateN

<else>defaultsubtemplate

<endif>

### Xtend

Az Xtend[14] egy statikusan típusos programozási nyelv, ami egy vékony rétegként csomagolja be a JDK-t (Java development kit) kiegészítéseket és új funkciókat rendelve hozzá. Az egyik ilyen funkció a Xtend saját template nyelve, amit Xtendes kifejezésként, nem pedig külön template fájlként definiálhatunk. Mivel a célom template nyelvek bemutatása, ezért leginkább az Xtend template komponensével szeretnék foglalkozni ebben a fejezetben.

#### Java és eclipse támogatás

Az Xtend Java 8 kompatibilis forráskódra fordul, ezért az Xtendes osztályok java-s osztályokban import segítségével hivatkozhatók és ugyanez fordítva is megtehető az Xtendes osztályokon belül.

Rendelkezik saját eclipse pluginnel, ami szintaxis ellenőrzést, kód kiemelést és a java-ra való fordítást is elvégzi. Továbbá kifejezetten kényelmessé és egyszerűvé tette az Xtend használatát a beépített kódkiegészítés.

#### A templatenyelv kifejezései

A templateket kifejezésként tudjuk definiálni amelynek elejét és végét 3 db ’ jel jelzi (2‑10. ábra). Az Xtend template kifejezései a «» karakterek közé írt kifejezéseket értékelik ki, minden mást statikus szövegként kezel. bármilyen java/Xtend függvény hívás megengedett és azok által visszaadott érték String reprezentációja fog a végső szövegbe kerülni.

2‑10. ábra Xtend template függvény definiálása

**def** **static** String printVhdl(HIG hig)

**var** entity = *preprocessor*.getEntity(hig);

**var** vhdlOutput = '''

«PrintUtils.*includes*»

«PrintUtils.*printEntity*(entity)»

«*printArchitecture*(hig,entity.name)»

'''

**return** vhdlOutput;

}

#### Listák és ciklusok

A listák kezelése a *«FOR…»* kifejezéssel történik, mint ahogy a 2‑11. ábra is mutatja a *FOR* kifejezésében az iteráló elem hivatkozását és a listát ’:’-val elválasztva kell beírni, majd *«ENDFOR»* kifejezéssel kell lezárni azt. A közbülső kifejezés annyiszor kerül a végső szövegbe ahány elemű a megadott lista. Megadhatunk még szeparáló karaktereket is amivel az egyes iterációk eredményeit fogja elválasztani (így az utolsó elem után már nem szúrja be), ezt a *SEPARATOR* kifejezéssel tudjuk megadni a *FOR* kifejezésben.

2‑11. ábra Listák kezelése Xtendben

**def** **static** printPorts(VhdlEntity entity) {

'''

«**FOR** port : entity.ports **SEPARATOR** ";"»

«port.name»:«port.getInOut»STD\_LOGIC\_VECTOR(«port.bitWidth-1» downto 0)

«**ENDFOR**»

'''

}

#### Elágazások

Az elágazások definiálását az *«IF…»* kifejezéssel tudjuk elkezdeni, és lezárásképpen a *«ENDIF»* kifejezést kell használni. Továbbá itt is van lehetőség *egyébként* és *egyébként ha* ágak kezelésére az *«ELSE»* és *«ELSEIF…»* kifejezések használatával.

2‑12. ábra Elágazások kezelése Xtendben

**def** **static** printType(VhdlPort port) {

'''

«**IF** port.bitWidth==1»

«port.vhdlPortType»

«**ELSE**»

«port.vhdlPortType»\_vector(«port.bitWidth-1» downto 0)

«**ENDIF**»

'''

}

### Konklúzió

A választási szempontok szerint az Xtend bizonyult a legjobb választásnak, mivel jobb Eclipse támogatása, könnyebben kezelhető script nyelve és könnyen elsajátítható szintaktikája van. A fejlesztés közben volt alkalmam megismerkedni a többi újdonsággal is, amit a Javahoz ad, pl. @Accessors annotációval megjelölhető egy osztály és fordítási időben legenerálódnak getter/setter metódusai.

# Specifikáció

A generált hardverleírásnak a magasszinten definiált algoritmus szerint kell működnie. A program bementeként magát az algoritmust kapja egy adatfolyamgráf (Data Flow Graph [9]) formájában, ami egy olyan irányított gráf, amelynek csúcsai egy-egy műveletet, élei pedig az adatáramlás irányát fejezik ki. Az adatfolyamgráf leírásához a HIG-t[7] (HLS intermediate graph) használtam.

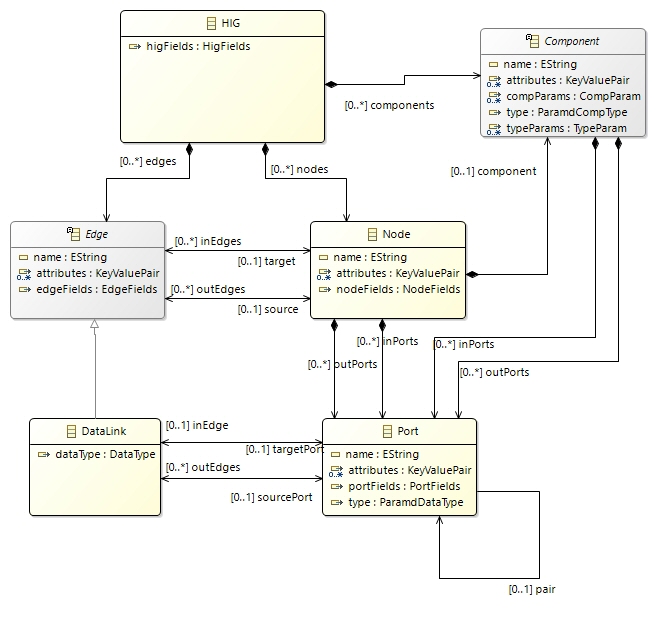
## HIG adatmodell

A HIG a PipeComp keretrendszer köztes nyelve, eltérően más adatfolyamgráfokhoz képest a ciklusokat műveltként, nem pedig irányított körként kezeli. A modellen belül minden műveletnek vannak a működését befolyásoló belső tulajdonságai, mint a végrehajtásiidő ami hardverre történő fordítás esetén a szükséges órajelciklusokat adja meg. Ilyen tulajdonság a ki- és bemeneti portok, amik az adott művelet külvilággal való kommunikációját mutatják, illetve az adott csúcs típusa is, ami megadja, hogy milyen műveletet reprezentál. A művelettípusok a Component nevű absztrakt osztályból származnak (3‑2. ábra).

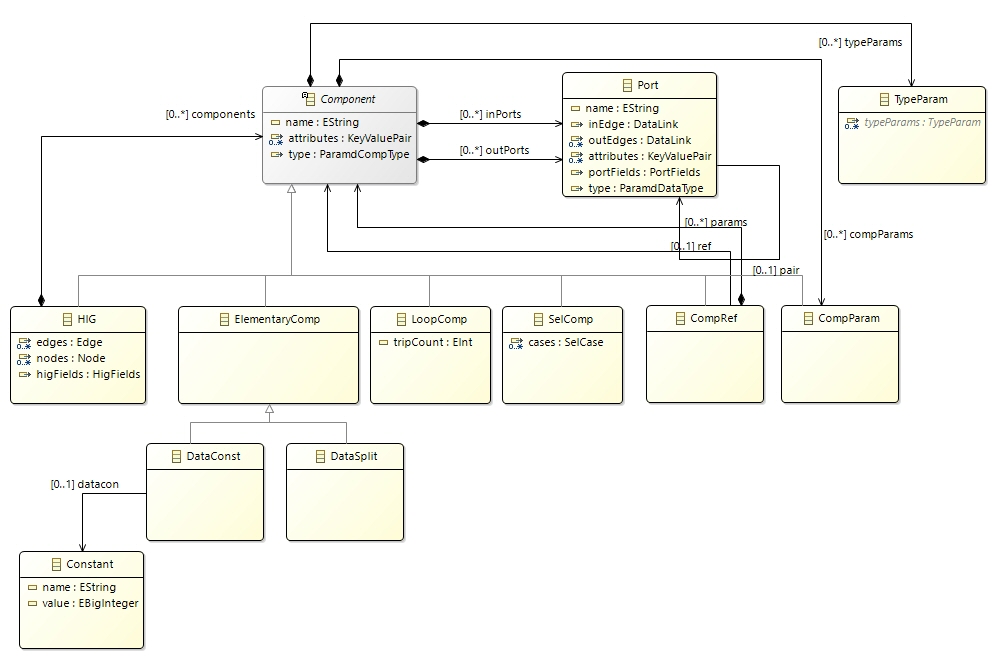
Művelet típusok:

* HIG: Ez maga az adatfolyamgráfot modellezi, minden modell legkülsőbb eleme. Több más Componentet és azok összeköttetéseit foglalja magába. Mostantól az egyértelműség kedvéért HIGCompként fogok rá hivatkozni, mivel magát a modellt is HIG-nek hívják, így szeretném elkerülni a félreértéseket (felépítés: 3‑1. ábra).
* ElementaryComp: Elemi műveleteket reprezentálja (pl. összeadás, szorzás)
* LoopComp: A ciklus leírására használt osztály. Ez magába foglal pontosan egy HIGCompot ami a ciklusmagját adja meg.
* SelComp: Elágazás megjelenítésére használt osztály, ami magába foglal több másik Componentet is amik az eseteket jelentik (felépítés: 3‑3. ábra).

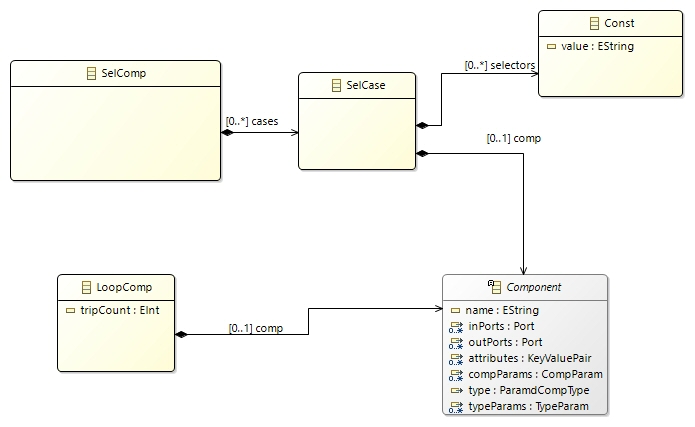
A műveletek portjait DataLink élek kötik össze, amik egy forrás és egy fogadó port között lehetnek. Természetesen egy forrás portból több él kiindulhat, de a fogadó portokba csak egy él érkezhet. Itt szeretném megjegyezni, hogy a forrás-/fogadó port és a ki-/bementi port nem feletethető meg egymásnak mivel egy komplex műveletből nézve a művelet saját bementi portjai forrásként, kimenti portjai fogadóként viselkednek, míg a belső node-k portjai pont fordítva működnek.



3‑1. ábra HIG Ecore metamodellje



3‑2. ábra Component Ecore metamodellje

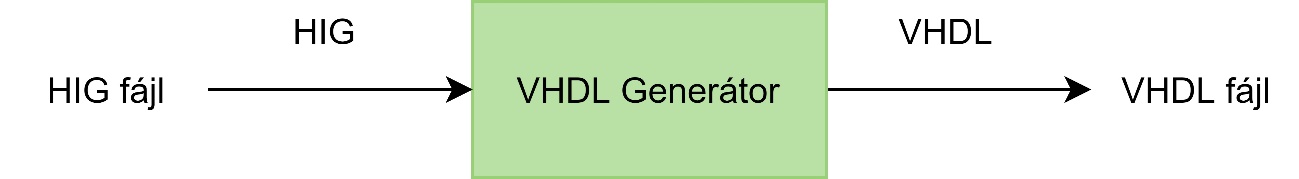


3‑3. ábra SelComp Ecore metamodelje

## Elvárások a generátorral szemben

A feladatkiírásnak megfelelően specifikáltam a rendszert (3‑4. ábra Specifikáció diagram), így az elkészítendő programnak képesnek kell lennie:

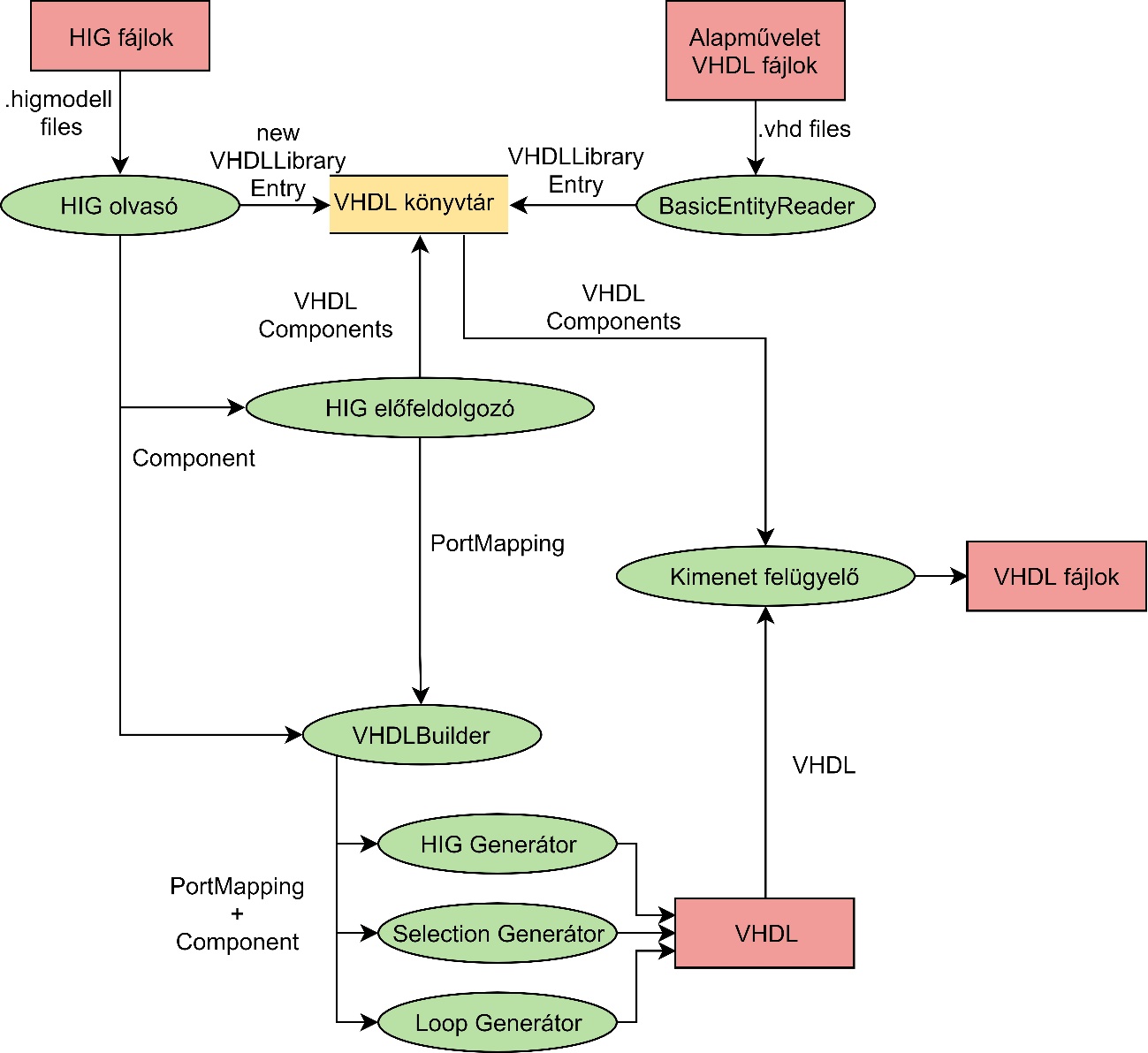
* Felolvasni a .higmodel fájlokat a megadott input mappából.
* A beolvasott adatmodellt betölteni a memóriába.
* A HIG, LoopComp, SelComp és ElementaryComp komponensek mellett a speciális tömbkezelési Componenteket is kezelje.
* A HIG komponensekből típusuktól függően megfelelő VHDL leírást generálni.
* A kimenetek közé minden olyan VHDL fájlt mellékelnie kell, ami példányosításra került bármelyik, már a kimenetek között található VHDL fájlban.



3‑4. ábra Specifikáció diagram

# Architektúra

Ebben a fejezetben a tervezés lépéseit és a kialakult architektúra fontosabb döntési szituációit mutatom be. A HIG-VHDL fordítóprogramot a Specifikáció fejezetben leírtakat figyelembe véve terveztem a programot, aminek három főbb szerepet kell betöltenie: bemenet olvasó, feldolgozó és kimenet szolgáltató. Mivel a szerepeket már tervezéskor igyekeztem minél jobban elkülöníteni egymástól, ezért szükséges volt egy tároló egység létrehozására is, amin keresztül ezek a modulok kommunikálni tudnak egymással.



4‑1. ábra Architektúra DFD (data flow diagram) formátumban

## VHDL könyvtár

Egy központi tárolóként funkciónál, ahol minden, a program által használt és generált VHDL fájl referenciája megtalálható. A referenciák név alapján vannak tárolva és a HIG Componentek nevei megegyezenek a hozzájuk tartozó VHDL nevével. Továbbá minden bejegyzés tartalmazza a VHDL fejlécét (az Entity és Architecture fejezetben bemutatott entity-ről van szó), illetve az általa komponensként hivatkozott VHDL-k VHDLLibrary bejegyzéseit.

### BasicEntityReader

Az alapműveleteket (összeadás, szorzás …stb) előre megírt VHDL fájlokban tárolja a program, egy meghatározott mappában. Innen az induláskor azokat felolvassa és a VHDLLibrary-t feltölti velük, így ezeket is be lehet tenni a VHDL bejegyzés hivatkozásai közé.

## HIG olvasó

A HIG olvasó modul egy megadott bemeneti mappából felolvassa az összes .higmodel fájlt és azok tartalmát EMF objektumként betölti a memóriába. Az EMF objektumokat bejárással felbontja olyan egységekre, amikből külön VHDL készül, továbbá mindegyikhez generál egy bejegyzést a VHDLLibrary-ben.

## HIG Component és a VHDL kapcsolata

A tervezéskor meg kellet vizsgálnom, hogyan tudok a HIG bemenetből azt reprezentáló, szintaktikailag helyes VHDL kódot generálni. Elsőkörben azt találtam, hogy minden HIG Component megfeleltethető egy azt leíró VHDL kódrészletnek, mivel a HIG által reprezentált viselkedés leírható VHDL kóddal. A Component és a VHDL felépítése is hasonló: ki- és bementeik vannak, hivatkozhatnak más modulokra/componentekre, ezeket belül példányosíthatják és kommunikálhatnak velük ki és bementi portjaikon keresztül. Kombinációs hálózatok esetében ez a megfeleltetés elégnek is bizonyult, de a ciklus és az osztás művelet megvalósítása így nem lehetséges. Így a VHDL-eknek egy szinkron sorrendi hálózatot kellett megvalósítaniuk, ezért bevezetésre került a ’clk’ (órajel) bemenet minden modulon, ami a mintavételezés és a kimenet ütemezésére használ az adott modul. Az időigény miatt még szükséges volt minden kimenethez egy ’rdy’ (kész) jelet rendelni, amivel jelezhette az adott eredmény elkészült és az már más rákapcsolódó modul által felhasználható, illetve minden modulban bevezetni egy ’rst’(reset) jelet, ami felkészíti, hogy újabb bemenet érkezik minden addigi eredményt dobjon el és kezdje ellőről működését.

## HIG előfeldolgozó

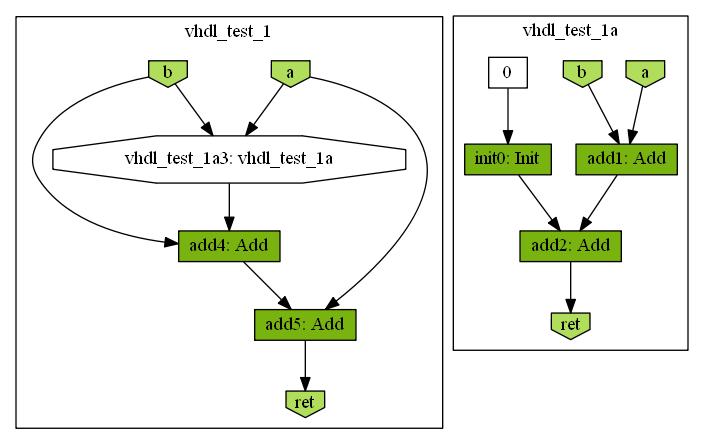
A generált VHDL komponenseit példányosítani kell, hogy minden példánynak a portjai a Componentnek megfelelően legyenek bekötve. A VHDL nyelvben ez úgy oldható meg, hogy a példányok kimeneteit és a modul bementeit signalokhoz rendelem, és a Port map-nél (2‑2. ábra) ezeket a signalokat adom át. Majd komponensek bementeit és modul kimenteit a HIG modell szerint a megfelelő kimenethez rendelt signalhoz rendelem. A signalok generálásához, és a signal porthoz rendeléséért az előfeldolgozó lesz felelős. Mivel ezen feladat közben az adott komponens alkomponensei is előkerülnek itt volt célszerű megvalósítani a VHDL könyvtárba való betöltésüket is.

## VHDL builder

Az előfeldolgozó által elkészített port-signal összerendelés és a HIG olvasóból érkező Component alapján a megfelelő generátor elkészíti a VHDL-t és a könyvtárban található fájlba kiírja azt.

### HIG Generátor

A HIG Genrátor feladata, hogy String formában visszaadja a VHDL megfelelőjét a bemenetén kapott port-signal összerendelésnek és HIGComp (példa: 4‑2. ábra). A HIGComp felépítésében egy Component listában tér el a többi Componenttől, a lista elemeit node-oknak hívják. Ezeket a node-kat kell majd VHDL-ben példányosítani és összeköttetéseiket kezelni, ezeket is a HIG tartalmazza, de ezt az előfeldolgozó már értelmezte a port-signal összerendelésben, így ezzel itt nem kell foglalkozni.



**int** **vhdl\_test\_1a**(**int** a, **int** b)

{ **int** c=**0**;

c+=a+b;

**return** c;

}

**int** **vhdl\_test\_1**(**int** a, **int** b)

{

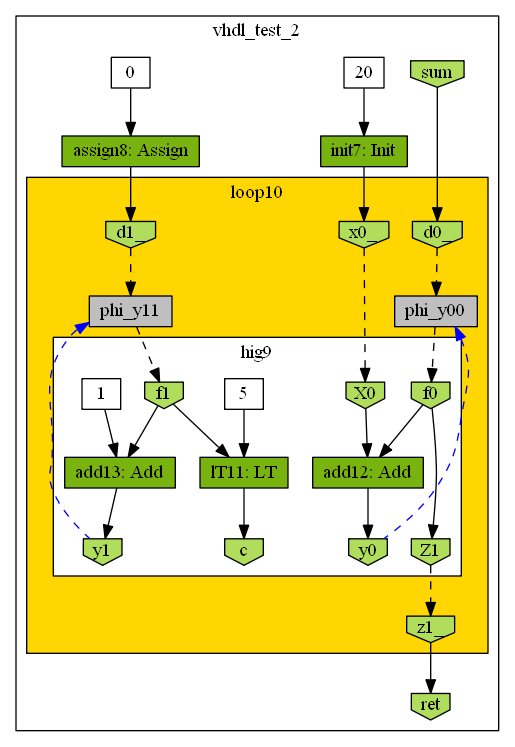
**return** vhdl\_test\_1a(a,b)+b+a;

}

4‑2. ábra HIG modell példa és a hozzátartozó C kódrészlet (felette)

### Loop Generátor

A Loop generátor is String formában adja vissza a LoopComp (példa: 4‑3. ábra) VHDL megfelelőjét. A LoopCompnak viszont HIGComptól eltérően csak egy belső Componentje van ez a ciklusmagja, ami egy HIG Compként szerepel a struktúrában, ezért a hozzátartozó VHDL-t a HIG generátor generálja így azzal itt nem kell foglalkoznom. A LoopComp struktúrájában a ciklusmag adja a leállás feltéllét, egy erre dedikált kimeneti porton, ezt a végső VHDL-ben úgy alakítottam ki, hogy amennyiben nulla érkezik a kimeneten úgy hamis értéket képvisel, bármilyen más érték esetén igazként van értelmezve. A ciklusnak szüksége lehet előző állapotának visszacsatolására, ennek megvalósítására egy saját VHDL modult használtam, ami az előző iteráció eredményét adta kimenetén, vagy egy LoopComp modulból származó kezdőértéket az első iteráció megkezdésekor.



int vhdl\_test\_2(int sum)

{

int i=0;

int b = 20;

for (i=0; i<5; i++) {

sum+=b;

}

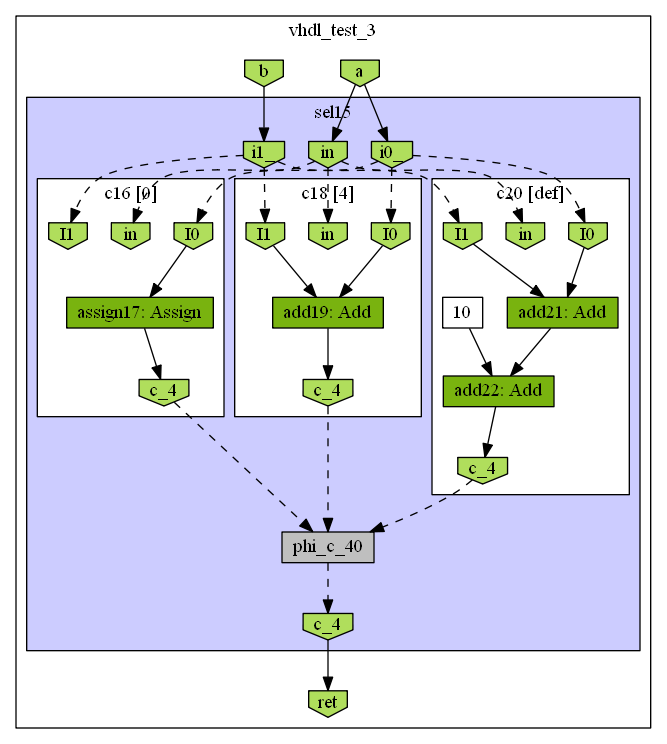
return sum;

}

4‑3. ábra LoopComp példa és a hozzátartozó C kódrészlet

### Selection Generátor

Feladata a SelComp Component (példa: 4‑4. ábra) VHDL String reprezetációjának generálása. A SelComp több ún. case-t tartalmazott, ezek írták le a különböző eseteket, amiknek kimeneteiből a case selektorai alapján lehet meghatározni a VHDL modul kimenetét. A case-k bemenetei szigorúan megegyeztek a SelComp bemeneteivel így azokat a végső VHDL-ben példányosítás port mapnél csak másolnom kellett.



int vhdl\_test\_3(int a, int b)

{ int c = 0;

switch(a){

case 0:

c= a;

break;

case 4:

c= a+b;

break;

default:

c= a+b+10;

}

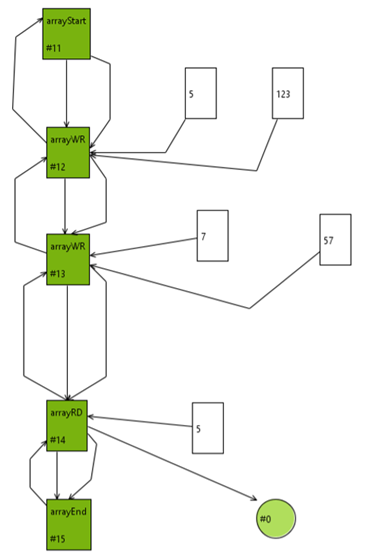
return c;

}

4‑4. ábra SelComp példa és a hozzátartozó C kódrészlet (felette)

### Tömbkezelés

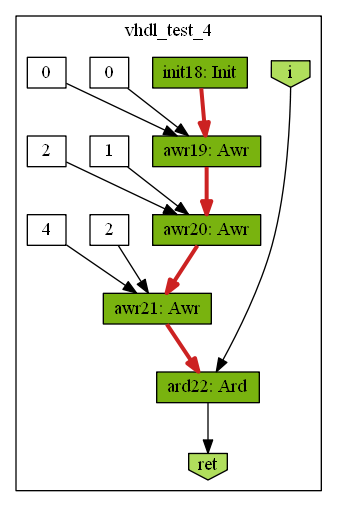
A tömbkezelés egy magasabb szintű nyelven általában a tömb referenciájának tovább adását és a referencia segítségével elérhetőek a tömb elemei. A VHDL nyelvet nem ilyen működés megvalósítására tervezték, bár van lehetőség a modulok portjain tömb típus definiálására, de ilyenkor a tömb minden elemének dedikált vezeték halmaza van, ezért a tömb elemeinek a számának növekedésével nőne a modulhoz szükséges vezetékek száma is. Továbbá a VHDL-ben nincs végrehajtási sorrend így a HIG modell által előírt tömbhozzáférés sorrendjét is biztosítani kell. A tömbhozzáférésének problémájára egy másik megoldást kellett találnom, a megoldást keresve bukkantam Olasz-Szabó Bence MSC-s Önálló Laboratórium keretei között elkészített beszámolójára[16], bár ő egy másik hardverleíró nyelvre fordítja az adatfolyamgráfot, de a tömbkezeléshez használt modul hierarchiát (4‑5. ábra) fel tudtam használni saját programomhoz.



4‑5. ábra Tömb kezelés moduldiagramja[16]

Bence dolgozatában a következőt írja: „Az arrayEnd modul tartalmazza a tényleges BlockRAM implementációt. Az arrayRD modul kiolvassa a tömbből a megadott indexű elemet, az arrayWR modul pedig beírja a megadott adatot a tömb megadott indexű elemére. A köztes arrayRD és arrayWR modulok kétféle üzemmódban képesek működni: aktív üzemmódban az adott modul kommunikál a tömbbel, transzparens üzemmódban pedig átereszti a többi modul kérését, ilyenkor tulajdonképpen vezetékként működik, azaz a kéréseket aszinkron módon, azonnal átereszti, nem órajelre továbbítja. Ez azért szükséges, hogy a megoldás sok köztes modul esetén is gyors maradhasson. Fontos biztosítani, hogy a köztes modulok közül egyszerre csak pontosan 1 lehet aktív üzemmódban, az összes többi köztes modulnak transzparensnek kell lennie, különben az arrayEnd-től messzebb lévő modulok kérései nem jutnának érvényre. Ezt a feltételt kaszkádosított engedélyező jelek segítségével oldom meg. A soros tömbelérést az arrayStart modul kezdeményezi egy kezdeti engedélyező jel hatására. A következő órajelkor az arrayStart utáni modul kapja meg az engedélyező tokent, ekkor ez a modul aktív, az összes többi transzparens. A következő órajelre az aktuális modul továbbpasszolja az engedélyező tokent a következő modulnak. Az arrayEnd előtti modul ready jele pedig felhasználható a folyamat befejezésének jelzésére.”

Bence módszere által inspirálva én is a modulokat egy láncba fűzöm és a tömb vezérlő jeleit átvezetem az író és olvasó modulokon. A lánc végén a start modul után HIGComp (4 6. ábra) által kijelölt első modul található, és ha a bementei kész állapotára elkezdeti a tömb műveletet, a többi modul transzparens állapotban van. Ennek megoldására egy foglalt jelet alkalmaztam (ez is át van vezetve minden modulon), ha foglaltat jelez akkor kimeneteit összeköti bemeneteivel, és várja amíg a foglalt jelzés megszűnik, ekkor elkezdi a sajátműveletét végrehajtani a tömbön, és ha végzett a következő reset vagy ciklus reset-ig a kimenetén megfelelő értéket tartja. A reset jelek miatt nekem nem kell foglalkoznom az újbóli hozzáféréssel hiszen a reset jel kiadásáig nem lesz olyan, így nem kell a modult transzparensen tartanom, ha megszűnt a busy jel.



**int** **vhdl\_test\_4**(**int** i)

{

**int** a[**3**];

a[**0**] = **0**;

a[**1**] = **2**;

a[**2**] = **4**;

**return** a[i];

}

4‑6. ábra Tömbkezlésre példa az adatfolyamgráfban és a hozzátartoót C kódrészlet

## Kimenet felügyelő

A kimeneti felügyelő feladata a végső kimenet szolgáltatása. A generált VHDL-ek könyvtár bejegyzés referenciáit fogja megkapni és azok komponens listáin végig iterálva megtalálja az összes szükséges VHDL fájlt. Egy fájl többször is szerepelhet, ezért többször előforduló fájlokat szűri.

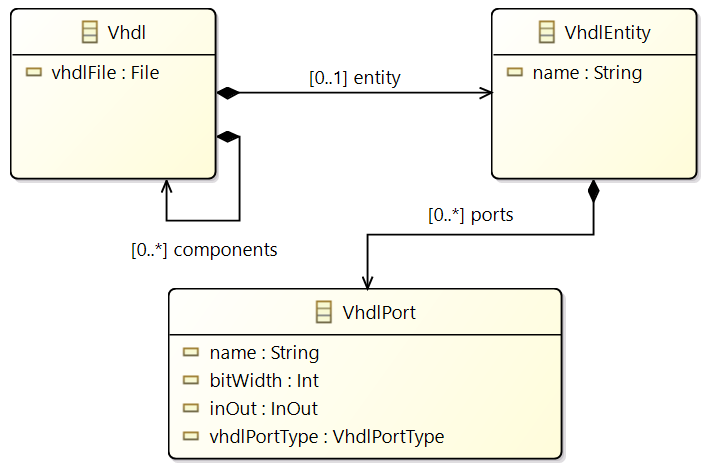
Az alapműveleteket tartalmazó VHDL fájlokat és azok komponenseit nem kell a generátornak elkészítenie, mert már előre elkészítettem őket. Ellenben a kimeneti mappában meg kell jelenniük, így hivatkozás esetén a generált VHDL fájlokkal együtt kerülnek a kimeneti mappába.

# Fejlesztés

Ebben a fejezetben a fejlesztés menetét, a közben felemerült problémákat és az ezek megoldását fogom leírni.

## VHDL könyvtár

A könyvtárat egy Map segítségével implementáltam, a Map-ben a kulcs a VHDL és Component neve (a két érték megegyezik) volt, az értéke pedig maga a VHDL objektum(5‑1. ábra), ami egy VHDL entity-t leíró objektumból, egy fájl referenciából, ami a saját VHDL fájljára mutat és egy Set amiben az általa hivatkozott VHDL objektumok referenciái voltak megtalálhatóak.



5‑1. ábra VHDL objektum EMF diagram

Ahhoz, hogy mindenhol egyszerűen elérhető legyen, és ne kelljen referenciáját továbbadni, Singleton tervezési mintát[13] használtam, és egy menedzser osztállyal oldottam meg az elérését, így a tároló maga rejtve marad a funkcióit használó osztályoktól, így implementációja könnyebben cserélhető. A menedzser osztály kényelmi funkciókat is biztosít, mert nem csak név, de Component alapján is lehet vele VHDL-t keresni. Továbbá a menedzser osztály biztosítja, ha egy olyan Component VHDL-jét akarjuk elkérni, ami még nincs benne a könyvtárban, akkor legenerál egy üres bejegyzést, amiben az adott VHDL neve és entity része található meg (Amikor a tényleges generálása történik már ezt az objektumot fogjuk oda adni).

### Alap műveletek kezelése

Minden HIG modellben leírható ElementaryOperation-höz kellett egy VHDL-t rendelnem, és ezeket is betölteni a VHDL könyvtárba. Mivel az alapműveletek VHDL-jei nem változtak dinamikusan egyik generátor hatására sem, ezért elég volt elkészítenem mindegyik statikus változatát. Ezeket egy erre kijelölt mappába helyeztem megegyező névvel, mint ahogy a VHDL-t és az ElmentaryOperation-t is hívták, és a könyvtár inicializásakor a VHDL bejegyzéseket készítettem minden művelethez. A bejegyzések entity részét sablonszerűen készítettem, így minden bemeneti portnak inputtal kezdődött a neve és egy futósorszámot raktam mögé. Ugyan ezt a szisztémát követtem az outputokkal is és a segéd jelekkel (clk, rst) is, így csak a ki-/bementek számát kellett megadnom az alapművelet betöltőnek. Természetesen erre a szabályra oda kellett figyelnem a tényleges VHDL-ek megírásánál is.

## HIG olvasó

A HIG olvasó egy bemeneti mappából felolvassa az összes .higmodel fájlt, és azok modelljeit betölti a memóriába. Ezekután egy bejáró osztályt kellett implementálnom, ami segít feldarabolni Componentekre a beolvasott HIG objektumot, ebben segítségemre volt az 2.2.1.2 fejezetben bemutatott EMF bejárási módszer.

A fejlesztés közben felmerült a probléma, hogy némely port nem VHDL kompatibilis néven szerepel a HIG modellben (pl. a VHDL-ben nem kezdődhet név \_ karakterrel). A nevek cseréjét egy előiterációban oldottam meg szintén az EMF bejárási módszerrel.

## VHDL generátorok

Minden generátor fejlesztése előtt elkészítettem az adott HIG Component működését leíró VHDL-t, ezzel egy fajta specifikációt készítve az adott generátor működéséhez. Az így szerzett tapasztalatok nagyban segítették későbbi munkámat, mert találtam közös elemeket a VHDL-kben amik nem függtek a Component típusától, ezek generálását egy közös utility osztályban végzem. Ilyen a VHDL entity része, illetve a component listája is. Minden generálást az Xtend segítségével végeztem.

### HIG Generátor

A generátornak első körben a VHDL signalokat ás konstansokat kellett legenerálnia, ezeket az előprocesszor által átadott port signal összerendelésből a signalok és konstansok kiszűrésével tudtam megvalósítani. Ezekután a node-nak megfelelő VHDL modulokat kellett példányosítani, a port signal összerendelésből kiderült milyen signalt kell az adott porthoz rendelni a port mapben. A modul ki- és bemeneti portjait is signalokhoz rendelem ezek is a port signal összerendelésből jön ki (példa: 5‑2. ábra).

**LIBRARY** **IEEE**;

**USE** **IEEE.STD\_LOGIC\_1164.ALL**;

**USE** **IEEE.Numeric\_STD.ALL**;

**ENTITY** **add** **IS**

**PORT**

(

rst : **IN** **STD\_LOGIC**;

loop\_rst : **IN** **STD\_LOGIC**;

clk : **IN** **STD\_LOGIC**;

a : **IN** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

a\_rdy : **IN** **BOOLEAN**;

b : **IN** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

b\_rdy : **IN** **BOOLEAN**;

ret : **OUT** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

ret\_rdy : **OUT** **BOOLEAN**

);

**END** **add**;

**ARCHITECTURE** **Behavioral** **OF** **add** **IS**

**COMPONENT** **Add** **IS**

**PORT**

(

rst : **IN** **STD\_LOGIC**;

loop\_rst : **IN** **STD\_LOGIC**;

clk : **IN** **STD\_LOGIC**;

input1 : **IN** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

input1\_rdy : **IN** **BOOLEAN**;

input2 : **IN** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

input2\_rdy : **IN** **BOOLEAN**;

output1 : **OUT** **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**);

output1\_rdy : **OUT** **BOOLEAN**

);

**END** **COMPONENT**;

**SIGNAL** signal\_a\_0 : **STD\_LOGIC\_VECTOR** (**31** **DOWNTO** **0**);

**SIGNAL** signal\_a\_0\_rdy : **BOOLEAN**;

**SIGNAL** signal\_b\_1 : **STD\_LOGIC\_VECTOR** (**31** **DOWNTO** **0**);

**SIGNAL** signal\_b\_1\_rdy : **BOOLEAN**;

**SIGNAL** signal\_out\_2 : **STD\_LOGIC\_VECTOR** (**31** **DOWNTO** **0**);

**SIGNAL** signal\_out\_2\_rdy : **BOOLEAN**;

**BEGIN**

add0 : Add **PORT** **MAP**(rst, loop\_rst, clk, signal\_a\_0, signal\_a\_0\_rdy, signal\_b\_1, signal\_b\_1\_rdy, signal\_out\_2, signal\_out\_2\_rdy);

signal\_a\_0 <= a;

signal\_a\_0\_rdy <= a\_rdy;

signal\_b\_1 <= b;

signal\_b\_1\_rdy <= b\_rdy;

ret <= signal\_out\_2;

ret\_rdy <= signal\_out\_2\_rdy;

**END** **Behavioral**;

5‑2. ábra HIGComp VHDL eredménye egyszerű összeadás műveletre

### Loop Generátor

A loop generátorban a signalok generálása és a ciklusmag példányosításán kívül meg kellett oldanom a ciklus visszacsatolást és a ciklusmag újra indítását is, mivel a generált rendszerben nem lehet új számítást elkezdeni amíg egy reset jel nem érkezik. Ezt egy új alapjel bevezetésével értem el, aminek a ’loop\_rst’ nevet adatam. A ’loop\_rst’ alapvetően az elemi műveletvégzők működését segíti, mert így tisztában vannak milyen forrásból származik a reset jel. Ezt a jelet használja a visszacsatolást végző modul is, hogy eldöntse honnan vegyen mintát a következő iterációhoz (’rst’ felfutó él esetén az alapértelmezett bementéről veszi a mintát, ’loop\_rst’ esetén a ciklusmag rákötött kimenetéről).

A ciklikus működést egy VHDL process (5‑3. ábra) kialakításával értem el, aminek az érzékenységi listájába a ’clk’-t és az ’rst’ alapjeleket tettem. Minden felfutó órajelben megvizsgálom készen (rdy) vannak-e a kimenetei a ciklusmagnak, ha igen ellenőrzöm a ciklusfeltétel dedikált kimenetét teljesül-e a felétel, amennyiben teljesül újra indítom a ciklust a ’loop\_rst’ kiadásával, amennyiben nem teljesül a kimenetek értékét egy bufferba helyezem és innentől ebből a bufferból töltöm a loop modul kimeneteit, amíg a loop modul rst kimenetén felfutó él nem érkezik, ekkor a loop kimenetei kész jelzése hamis lesz és elkezdi a ciklus végrehajtását.

### Selection Generátor

5‑3. ábra Egy LoopCompból (4‑3. ábra) generált VHDL kód ciklus processe

loop\_process : **PROCESS** (clk, signal\_rst)

**VARIABLE** is\_loop\_finished : **BOOLEAN** := false;

**VARIABLE** buffer\_z1 : **STD\_LOGIC\_vector**(**31** **DOWNTO** **0**)

;

**BEGIN**

**IF** rising\_edge(signal\_rst) **THEN**

is\_loop\_finished := false;

z1\_rdy <= false;

**ELSIF** falling\_edge(signal\_rst) **THEN**

**ELSIF** signal\_c\_rdy **AND** signal\_y0\_5\_rdy **AND** signal\_y1\_6\_rdy **AND** signal\_Z1\_7\_rdy **THEN**

**IF** signed(signal\_c) > to\_signed(**0**, **32**) **THEN**

z1\_rdy <= false;

signal\_loop\_rst <= '1';

**ELSE**

**IF** **NOT** is\_loop\_finished **THEN**

buffer\_z1 := signal\_Z1\_7;

is\_loop\_finished := true;

**END** **IF**;

z1 <= buffer\_z1;

z1\_rdy <= true;

**END** **IF**;

**ELSE**

signal\_loop\_rst <= '0';

**END** **IF**;

**END** **PROCESS**;

Itt a signal generálás, és az esetek példányosításán (példa: 5‑4. ábra) túl meg kellett oldanom az esetek eredményei közöli választást. Ezt egy VHDL process (5‑5. ábra) segítségével oldottam meg, amiben figyelem a selcomp modul dedikált választó bemenetén érkező adatot és összehasonlítom az adott eset selectoraival, a selectorok az esetekben egy String listában találhatóak, amik alapvetően számok, de van köztük egy trükkös elem is ez a ’def’ (alapértelmezett). Az alapértelmezett elemhez nem rendelek össze hasonlítást egyszerűen egy else ágban kezelem le a processen belül.

5‑4. ábra Esetek példányosítása példa

case0 : c16 **PORT** **MAP**(rst, loop\_rst, clk, input2, input2\_rdy, i0, i0\_rdy, i1, i1\_rdy, signal\_c4\_2, signal\_c4\_2\_rdy);

case1 : c18 **PORT** **MAP**(rst, loop\_rst, clk, input2, input2\_rdy, i0, i0\_rdy, i1, i1\_rdy, signal\_c4\_3, signal\_c4\_3\_rdy);

case2 : c20 **PORT** **MAP**(rst, loop\_rst, clk, input2, input2\_rdy, i0, i0\_rdy, i1, i1\_rdy, signal\_c4\_4, signal\_c4\_4\_rdy);

selection\_process : **PROCESS** (clk)

**BEGIN**

**IF** input2\_rdy **THEN**

**IF** signed(input2) = constant\_0 **THEN**

c4 <= signal\_c4\_2;

c4\_rdy <= signal\_c4\_2\_rdy;

**ELSIF** signed(input2) = constant\_1 **THEN**

c4 <= signal\_c4\_3;

c4\_rdy <= signal\_c4\_3\_rdy;

**ELSE**

c4 <= signal\_c4\_4;

c4\_rdy <= signal\_c4\_4\_rdy;

**END** **IF**;

**ELSE**

c4\_rdy <= false;

**END** **IF**;

**END** **PROCESS**;

5‑5. ábra Egy SelComp (4‑4. ábra) választáshoz használt processe

# Tesztelés

A tesztelés fejezet során szeretném bemutatni, hogyan ellenőriztem a program működését. Minden teszthez készítettem egy tesztesetet ami leírja milyen bementeket szeretnék állítani az adott modulnak, továbbá az elvárt kimeneteket is előírja. Magukat a teszteket a Xillinx Vivado v2018.2 programban[15] végeztem, a Vivado egy olyan program, ami hardverleíró nyelvek (pl. VHDL, Verilog) szintézisét és analízisét végzi, továbbá van benne lehetőség ezek szimulációjára. A szimulációt a Vivadoban egy VHDL nyelven írt szimulációs modullal lehet elvégezni, amiben a kívánt modult/modulokat példányosítva a rájuk kötött signalok értékeit lehet manipulálni. A VHDL nyelvben a ’wait’ kulcsszó van segítségünkre ebben, amit a processekben lehet használni, és a paraméterként megadott idejű várakozásra lehet kényszeríteni a processt, a wait csak szimulációs célokat szolgál a végső hardverbe nem lehet szintetizálni (6‑1. ábra). A teszteléshez készítettem egy saját órajel modult, így nem kell nekem mindig kézzel állítgatni az órajelt, ezt modult mutatja be a 6‑2. ábra.

6‑1. ábra wait utasításra példa

proc: process

begin

signal\_input<=std\_logic\_vector(to\_unsigned(13, 32));

wait for 10ns;

signal\_input<=std\_logic\_vector(to\_unsigned(0, 32));

wait for 10ns;

end process;

A Vivado a szimuláció eredményét egy waveform diagrammon jeleníti meg, ez megmutatja melyik időpillanatban milyen értékek voltak a modul ki- és bemenetein (waveformra példa a 6‑3. ábra). A waveformban a bitvektorból képzett számok hexadecimális számrendszerben vannak ábrázolva, az önálló értékek értékük szerint vannak reprezentálva.

A teszteléshez az adatfolyamgráfokat egy már kész, a PipeComp keretében készített, frontend segítségével készítettem el, ami C kódról fordította a HIG adatfolyamgráfot. Az adatfolyamgráfon kívül egy szemléletes ábrát is készít a C-ben leírt algoritmusról, ezzel emberiszem számára is könnyebben értelmezhetők az által generált adatfolyamgráfok.

A tisztánlátás érdekében szeretném megjegyezni, hogy a tesztelésnél használt, generált és szimulációs VHDL kódokat, a dolgozatom végén található Függelék fejezetben helyeztem el, mert azok viszonylag nagy helyet foglalnak és nehezen olvashatóvá tennék a dolgozatot.

6‑2. ábra Szimulációhoz használt órajelmodul VHDL kódja

**LIBRARY** **IEEE**;

**USE** **IEEE.STD\_LOGIC\_1164.ALL**;

**ENTITY** **sim\_clock** **IS**

**PORT** (clk : **OUT** **std\_logic**);

**END** **sim\_clock**;

**ARCHITECTURE** **Behavioral** **OF** **sim\_clock** **IS**

**SIGNAL** clock : **std\_logic** := '0';

**BEGIN**

**PROCESS**

**BEGIN**

**WAIT** **FOR** **10**ns;

clock <= **NOT** (clock);

**END** **PROCESS**;

clk <= clock;

**END** **Behavioral**;

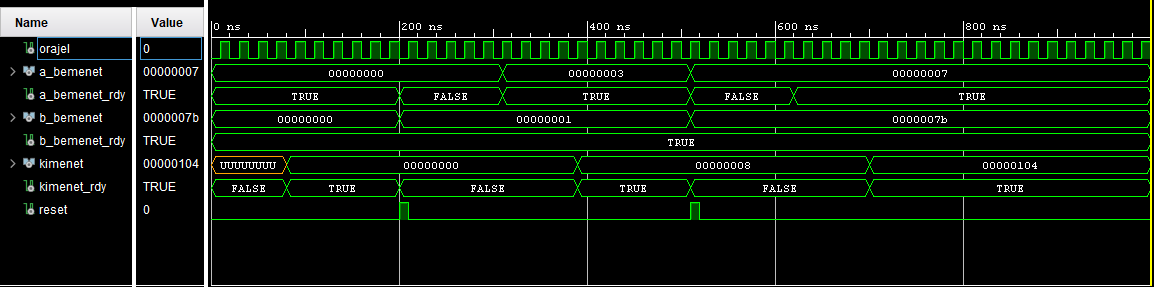
## HIG

Ebben a fejezetben bemutatom a HIGComp tesztelésének lépéseit. A teszteléséhez a 4‑2. ábra által demonstrált HIG modellt fogom használni. A teszt során szeretném megvizsgálni hogyan működik a modul induláskor, miképp viselkedik reset jel kiadása után, illetve egy következő reset után más bemenetekre. A vizsgált HIG modell által leírt algoritmus a következő matematikai kifejezéssel írható le: azaz . A szimulációt három részre osztja a reset jel kiadása, ezek eredményeit külön fogom elemezni.

Az első részben tesztelem az indulást bemenetek mellett, a modul kimenetétől azt várom el, hogy csak akkor jelenjen meg rajta a kész jelzés amikor a tényleges eredmény is szolgáltatva van, illetve a tényleges eredményre pedig: . A 6‑3. ábra az elvárt működést mutatja.

A második részben tesztelem a reset jelre adott reakciót illetve itt azt is szeretném vizsgálni mi történik ha az egyik bemenet nem készült el (hamis rdy jelet ad). A bemeneteket a értékekre állítom, illetve a *b* bemenetet és kész jelzését később állítom, mint az *a* bemenetet. A kimeneten azt várom, hogy addig nem kezdődik meg a számolás amíg a b bemenet kész állapotba nem kerül, illetve eredménynek a várom. A 6‑3. ábra az elvárt eredményt mutatja.

A harmadik részben nagyobb számok, illetve újabb újra indítás kezelését szeretném megvizsgálni. Bemenetek: , Kimenet: . A 6‑3. ábra az elvárt eredményt mutatja.

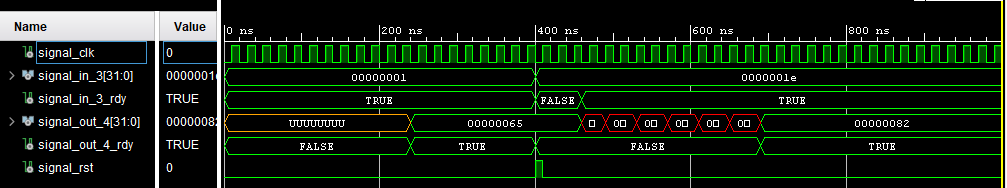


6‑3. ábra vhdl\_test\_1 tesztelés waveform diagramja

## LoopComp

Ebben a fejezetben bemutatom a LoopCompból generált VHDL kód tesztelésének lépéseit és eredményeit. A teszteléséhez a 4‑3. ábra által demonstrált HIG modellt fogom használni. A teszt során szeretném megvizsgálni mennyire valósult meg a ciklikus működés, és külső reset hatására újra indul-e a végrehajtás. A végrehajtandó algoritmus egy ciklust ír le, amit a modul ötször fog végrehajtani. Ennyiszer fogja hozzáadni a húszat a *sum* bemenethez, amit aztán eredményként visszaad, képlettel: . A szimulációt 2 részre oszthatjuk a rst jel segítségével, ezeket külön fogom megvizsgálni.

Az első esetben, ha akkor kimenetnek eredményt várok. A 6‑4. ábra szerint jó eredményt ad a VHDL modul. A második esetben a bemenet és kimenetre a eredményt várom. A 6‑4. ábra szerint ezt az eredményt adja a VHDL modul, de itt az eredmény megjelenése előtt látható pirosszínnel jelölt kimenetek, amiknek nem szabad gondot jelenteniük hiszen a kimenet kész jelzése csak a kimenet megérkezése után vált igazra.

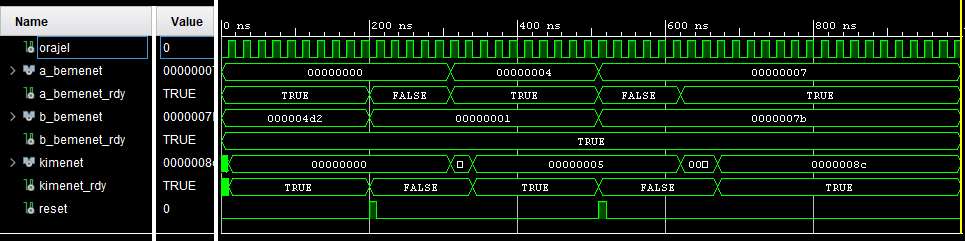


6‑4. ábra LoopComp szimulációjának waveform diagramja

## SelComp

Ebben a fejezetben szeretném letesztelni a SelComp generátor által generált VHDL kódot. A teszteléséhez a 4‑4. ábra által demonstrált HIG modellt fogom használni. Szeretném megvizsgálni, hogy a vizsgált, SelCompból generált, VHDL kód minden case-re a megfelelő eredményt adja. A HIG modell által leírt algoritmus egy elágazást valósít meg ami az *a* bemenet értékétől függő case eredményét adja vissza. Tehát ha vagy, ha , minden egyéb esetben . A teszt során mindhárom esetet megvizsgálom, a 6‑5. ábraán a reset jelek ezt a három esetet választják külön.

Az első esetben a bemenet , a kimeneten a eredményt várom, a második esetben a bemenet , a kimeneten a eredményt várom, és a harmadik esetben a bemenet a kimeneten a eredményt várom. A 6‑5. ábraán látható a VHDL modul az elvárt módon viselkedik.



6‑5. ábra SelComp szimulációjának waveformdiagramja

## Tömbkezelés

# Összefoglalás

A dolgozatban ismertettem a VHDL nyelv jellegzetességeit, bemutattam az EMF-et, leírtam három manapság használatos template nyelvet, és megmutattam az általam elkészített VHDL kódgenerátor specifikációját, tervezési lépéseit, a megvalósítás módját és a tesztelés lépéseit.

## Az általam elért eredmények

* A VHDL nyelv munkámhoz szükséges kifejezéseit sikerült készség szinten elsajátítanom. Továbbá sikerült megismerkednem egy olyan eszközzel, amit hardverleíró nyelvek fejlesztéséhez használnak.
* Megértettem a template nyelvek hasznosságát forráskódok generálásában, az általam bemutatott három nyelvből a végül választott Xtendet sikerült készség szinten is elsajátítanom.
* Gyakorlatban is kipróbálhattam az egyetemen szerezett ismereteimet a kombinációs és sorrendi hálózatokról.
* Sikerült egyszerűbb algoritmusokból (ciklus, elágazás, összeadás stb.) automatikusan generálni azt végrehajtó VHDL kódot.
* Munkám során megtapasztaltam mennyivel nehezebb leírni egy egyszerűbb algoritmust (ciklus) egy hardverleíró nyelven, mint egy magasabb szintű nyelven, ezt a teszteléshez generált kód mennyisége is mutatja viszonylag egyszerű tesztesetekhez.

## Továbbfejlesztési lehetőségek

Zárásként a fejlesztett szoftver néhány továbbfejlesztési lehetőségét szeretném bemutatni. Ezek felsorolása következik:

* Nem csak egészszámok kezelése a modulok bemenetein.
* Konfigurálható legyen az elágazás esetén, hogy a case-k eredményei közül válasszon a modul, vagy már rögtön csak az a case számolja ki végeredményét, amelyik kiválasztásra kerül.

# Köszönetnyílvánítás

Szeretnék köszönetet mondani konzulensemnek, Suba Gergelynek a türeleméért és a belém fektetett munkájáért.

Irodalomjegyzék

1. Multivalue Logic System for VHDL Model Interoperability, https://en.wikipedia.org/wiki/IEEE\_1164
2. Eclipse modelling framework, <https://eclipsesource.com/blogs/tutorials/emf-tutorial/>
3. Eclipse Foundation, http://www.eclipse.org/org/
4. Eclipse Project, <http://wiki.eclipse.org/Eclipse_Project>
5. Eclipse Platform, <http://wiki.eclipse.org/Platform>
6. High-level synthesis, <https://en.wikipedia.org/wiki/High-level_synthesis>
7. G. Suba, and P. Arató: Concept of the system-level synthesis framework PipeComp, <http://hls.iit.bme.hu/lib/exe/fetch.php/hu/pipecomp.pdf>
8. Freemarker, <https://freemarker.apache.org/>
9. Data Flow Graph, <https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Data_Flow_Graph.html>
10. StringTemplate, <https://www.stringtemplate.org/about.html>
11. High-level synthesis, <https://en.wikipedia.org/wiki/High-level_synthesis>
12. VHDL felépítés, <https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer_files/image003.gif>
13. Singleton pattern, <https://en.wikipedia.org/wiki/Singleton_pattern>
14. Xtend, <https://www.eclipse.org/xtend/documentation/index.html>
15. Xilinx vivado, <https://www.xilinx.com/products/design-tools/vivado.html>
16. Olasz-Szabó Bence: Automatikus hardverszintézis adatfolyamgráfból, <http://hls.iit.bme.hu/lib/exe/fetch.php/doc/onlab1_msc_oszb.pdf>