Debreceni Egyetem
Informatikai Kar

FPGA programozása magas szintű nyelven

Témavezető:
Dr. Herendi Tamás
egyetemi adjunktus

Készítette:
Laczkó Sándor
PTI szakos hallgató

Debrecen
2010

Tartalomjegyzék

1. Az FPGA eszközök

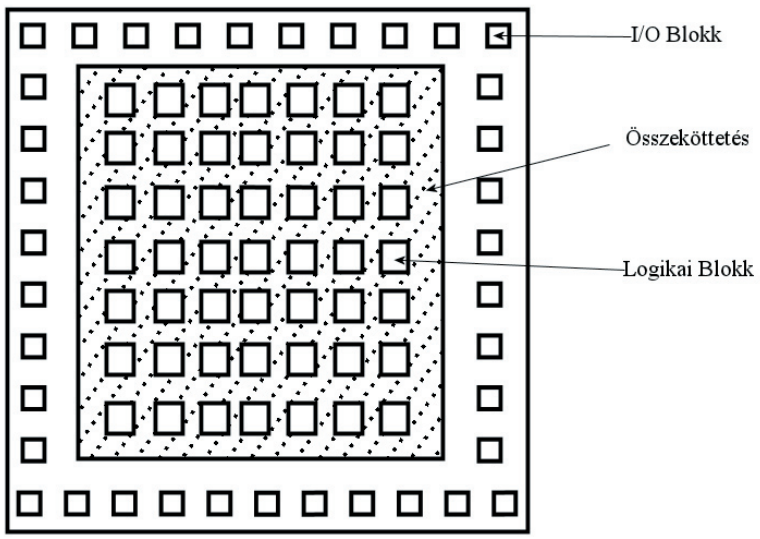
Az FPGA (Field-Programmable Gate Array) eszközök olyan speciális integrált áramkörök, melyekkel különböző feladatok elvégzését megvalósító áramköröket lehet létrehozni. Az eredmény nem sokban különbözik a hagyományos nyomtatott áramköröktől, viszont az előállítása sokkal egyszerűbb és gyorsabb, illetve később tetszés szerint újraprogramozható, ami a hagyományos műanyag lapokra integrált áramkörökről egyáltalán nem mondható el. Széles körű felhasználási területe folyamatosan bővül. Jellemzően alkalmazzák digitális jelfeldolgozásra, orvosi képalkotásra, beszédfelismerésre, kriptográfiára, bioinformatikára, hardver emulációra, stb. Ezek az eszközök processzorként jelennek meg egy-egy speciálisan erre a célra készített alaplapon, melyre különböző perifériákat integrálnak, ezzel biztosítva a felhasználhatósági skála szélesítését. Ezek a perifériák esetenként szabványos csatolóeszköz (pl. SDRAM foglalat) segítségével cserélhetőek, bővíthetőek. Az FPGA processzor és az azt felhasználó egyéb komponensek közötti kommunikáció számtalan módon megvalósítható.

Az eszközök programozása több módon is történhet. Mivel eredetileg ez egy mérnökök által, mérnökök számára kifejlesztett eszköz, ezért lehetőség van megtervezni egy áramkört, amit egy erre a célra kifejlesztett fejlesztői program segítségével feltölthetünk a processzorra, ahol ez az áramkör fizikailag megvalósul. Egy másik megközelítés szerint az FPGA processzor egy PC processzorhoz hasonló, programozható eszköz. Egy hagyományos PC processzorban megtalálható logikai kapuk fizikailag rögzítve vannak, ezekhez igazítva és optimalizálva található meg az alap utasításkészlet, amelyek segítségével tetszőleges bonyolultságú algoritmus hajtható végre. Ettől eltérően egy FPGA processzorban viszonylag nagy szabadsággal rendezhetünk el logikai kapukat, melyeket az általános processzorokhoz hasonlóan felhasználhatunk a kívánt algoritmus lépéseinek végrehajtására. Alapvető különbséget jelent azonban, hogy az architektúra kialakítása miatt a hagyományos processzorok esetében egyetlen órajel alatt egyetlen utasítás hajtható végre (ha eltekintünk a többmagos processzorok előnyeitől), az FPGA esetében viszont ez a szám megsokszorozható, mivel az egyes kódrészeknek megfelelő logikai kapuk egyszerre több helyen elhelyez-

hetők, így ezek egymással párhuzamosan hajtódhatnak végre. A logikai kapuk kialakításához különböző, alacsony szintű nyelvekhez hasonló programnyelvek állnak rendelkezésre. Ezen nyelveken ugyanúgy programozhatunk egy fejlesztői környezetben, mint bármely más elterjedt programnyelv fejlesztői környezetében. A kódolást követően szintaktikai és szemantikai elemzésen esik át a forrásszöveg. A fordítás eredményeként egy áramkör fog létrejönni, melyet később feltölthetünk az eszközre. Mivel az áramkör fizikailag megvalósul, ezért nem mindegy, hogy az egyes részek hogyan vannak kialakítva, illetve egymáshoz képest hogyan helyezkednek el. Ebből kifolyólag az optimalizálás a fordítás egy fontos eleme, melynek során az áramkör egyes elemei a megfelelő helyre kerülnek. A jelentős optimalizálási folyamatoknak köszönhetően egy modern számítógépen is percekig, akár órákig tarthat a végleges elrendezés kialakítása. Természetesen ezt az időt jelentősen befolyásolja a kialakítandó áramkör bonyolultsága és a rendelkezésre álló hardver mérete.

1.1 Az FPGA processzorok felépítése

Egy FPGA processzor felépítése a gyártóktól és típusoktól függően sokféle lehet, azonban van pár alapvető jellemvonásuk, melyekben megegyeznek a különböző modellek.



1. ábra - Az FPGA processzor felépítése

Az 1. ábrán az FPGA processzorok egy általános sematikus felépítése látható. Az információk átvitelét az úgynevezett I/O Blokkok segítségével oldja meg, melyek praktikusan a processzor külső peremén körben helyezkednek el. Ezek a blokkok alkotják a kommunikációs platformot. A belső négyzetben találhatóak a logikai blokkok, melyek funkcióját a fejlesztő tetszés szerint konfigurálhatja. A processzorokban létrehozható logikai utasításokat gyakorlatilag ezen blokkok segítségével hozhatjuk létre. Ezek belső felépítése már erősen gyártófüggő, de működési elvben azonosak. Minden blokknak egyformán van néhány bemeneti és néhány kimeneti csatlakozása. Értelem szerűen a bemenő jeleken végrehajtott műveleteket a kimeneti csatornákon továbbítják. A végrehajtható műveletek alatt általában az alap logikai műveletekre kell gondolni, melyeket a blokk belső szerkezete által meghatározott szélességű bitsorozaton hajthatunk végre. A blokkok közötti kommunikációt segíti elő a blokkok között elhelyezkedő bonyolult érhálózat, mely tetszőleges logikai blokkok között megteremti a kapcsolatot. Egy processzor teljesítményét, illetve a rajta létrehozható áramkör bonyolultságát alapvetően befolyásolja a logikai blokkok száma és belső felépítése.

Egyes hardvergyártók ezeket az alapvető elemeket még egyéb, kevésbé általános elemekkel bővítik ki. Ezek az elemek általában olyan speciális részfeladat elvégzésére alkalmasak, melyek gyakran hasznosak bizonyos bonyolultabb algoritmusok implementálása esetében. Egy ilyen speciális elem például a Xilinx processzoraiban található beépített szorzó, mely hardveresen implementálva képes bizonyos méretet meg nem haladó regiszterek gyors összeszorzására.

1.2 Szoft processzorok

Amint az látható, az FPGA processzorok felépítése eltér a PC gépekben található processzoroktól, viszont a számítási ereje nem kevesebb annál, ugyan azon logikai műveleteken alapszik, mint a PC, viszont nem feltétlenül követi a Neumann elvű architektúrát. Ebből következik, hogy a PC processzorainak utasításkészlete megvalósítható FPGA környezetben, ez által szimulálható azok működése. A nagyobb fejlesztő cégek lehetővé teszik, hogy előre kidolgozott komplett processzort hozzunk létre az eszközünkön. Mivel ezek a processzorok

csak szimulálják az eredeti működését, az utasításkészletének FPGA környezetbeli megoldásokkal, ezért az így létrehozott processzorokat szoft processzoroknak nevezik. A szoft processzorok lehetővé teszik a hagyományos programozási technika ötvözését a hardverprogramozás előnyeivel, továbbá lehetőség nyílik a hagyományos gépekre megírt programok átültetésére is. A gyártók általában külön fejlesztői környezetet biztosítanak a szoft processzoron való fejlesztésre, ahol közvetlenül elérhetjük a kártya perifériáit és általában C vagy C-hez hasonló nyelvben fejleszthetjük a programunkat. A szoft processzorokra való fejlesztés eredménye egy olyan kód, melyet a fejlesztői környezet fordítója állít elő, tölt fel a céleszközre, majd az FPGA eszközön implementált szoft processzor futtat. A szoft processzorok előnye, hogy megszokott környezetben fejleszthetünk, és könnyen kezelhetjük az eszköz perifériáit, mindemellett hivatkozhatunk olyan modulokat is, melyek szintén az FPGA processzorán, de a szoft processzoron kívül találhatóak. A szoft processzorok teljesítménye általában lassabb az általuk modellezett processzoréhoz képest.

1.3 Alprogramok FPGA processzoron

Egy komplex feladat elvégzéséhez jól ismert módszer, hogy a feladatot apróbb részekre bontjuk, amíg egy-egy részfeladatot méretéből adódóan egyszerűen meg tudunk oldani. Ezen részfeladatok megoldásából pedig létrejön a komplex feladat megoldása. Ezt a módszert alkalmazhatjuk az FPGA processzorok esetén is. Amennyiben egy feladatot az eljárás orientált paradigmából ismert módon részfeladatokra bontunk, és ezen részfeladatokat a függvényekhez hasonló módon időnként különböző bemeneti adatokra meg kell ismételnünk, több lehetőségünk is adódik a kivitelezésre.

Az első, kézen fekvő megoldás, hogy a részfeladatot elvégző áramkörön újra és újra elvégezzük a műveleteinket a különböző bemenetekkel. Ez a megoldás viszonylag lassú, mert nem használjuk ki a párhuzamos futás lehetőségét, viszont helytakarékos, mert a feladat n-szer való végrehajtásához is csak egyszer kell a processzorban kialakítani az áramkört.

A második megoldás, hogy a részfeladatot elvégző áramkört annyiszor hozzuk létre a processzor különböző pontjain, ahányszor azt alkalmazni szeretnénk. Ez a megoldás sokkal nagyobb teljesítménnyel oldja meg a problémát, hiszen gyakorlatilag egyetlen lépésben elvégzi a feladatot, viszont ha n különböző bemenetre szeretnénk kiszámítani a feladatot, akkor n különböző helyen kell implementálnunk az áramkört, ami a meglehetősen szűkös fizikai korlátok miatt nem minden esetben valósítható meg.

A harmadik megoldás az előző kettő egyfajta kombinációja, miszerint lehetőségeinkhez mérten többször elhelyezzük a részfeladatot megoldó áramkört és ezeket alkalmazzuk időben többször.

Az előzőekben feltételeztük, hogy az egyes részfeladatok függetlenek egymástól, nem támaszkodnak egymás eredményeire. Amennyiben a komplexebb feladatokat sikerül olyan részfeladatokra bontanunk, melyben az egyes részfeladatok függetlenek, akkor jól párhuzamosítható algoritmusokat hozhatunk létre.

2. A párhuzamos programozás

A számítógépek kezdetben egy időpillanatban egy utasítást hajtottak végre. A párhuzamos programozás alap ötlete, hogy egy időpillanatban egyszerre több utasítást is végrehajthassunk. Idővel különböző modellek alakultak ki a párhuzamos számítások végrehajtására.

2.1 SISD – Single Instruction, Single Data

A modell lényege, hogy egy adott időpillanatban egyetlen utasítás segítségével egyetlen adatelemen hajthatunk végre műveletet. A műveleteket vezérlési utasítások segítségével bonyolultabb struktúrákba szervezhetjük, így áll össze egy komplex feladat megoldása. Sok párhuzamos programozást megvalósító modell olyan megoldást kínál, mely kompatibilis az SISD alapelveivel. Ennek következményeként a legtöbb erre épülő szoftver a párhuzamosságot egyfajta időosztásos megoldással valósítja meg, mely valójában nem párhuzamos, csak a futtató környezet szimulálja ennek megfelelően a működését.

2.2 SIMD – Single Instruction, Multiple Data

A szuperszámítógépek világában jellemzően megtalálható modell, melynek lényege, hogy egy központi irányító egység sok viszonylag sok processzort irányít, melyek mindegyike egymástól függetlenül egy speciális feladatot lát el. Ezek a processzorok mind a saját adatukon operálnak, viszont azonos műveletet hajtanak végre.

2.3 MIMD – Multiple Instructions on Multiple Data

A párhuzamos végrehajtás felé egy újabb lépést téve eljuthatunk ehhez a modellhez, melyben minden processzor külön műveletet hajt végre a saját adatrészén. Ez hangzik a leghatékonyabb megoldásnak, viszont kezelhetőség szempontjából meglehetősen bonyolult, nehéz összehangolni a processzorok működését egy komplex feladat megoldása esetén.

2.4 Osztott memóriás MIMD

Mivel az MIMD modell legnagyobb problémája, hogy a processzorok között bonyolult a kommunikáció megteremtése, ezért alternatív megoldást kellett keresni az információk átadására. Így alakult ki az üzenet átadásos architektúra, melyben a processzorok nem egy központi memóriából veszik ki az adatokat, és írják vissza, hanem az adat az egyik folyamattól a másikhoz csomagok formájában vándorol át.

2.5 Párhuzamos programozás FPGA eszközökkel

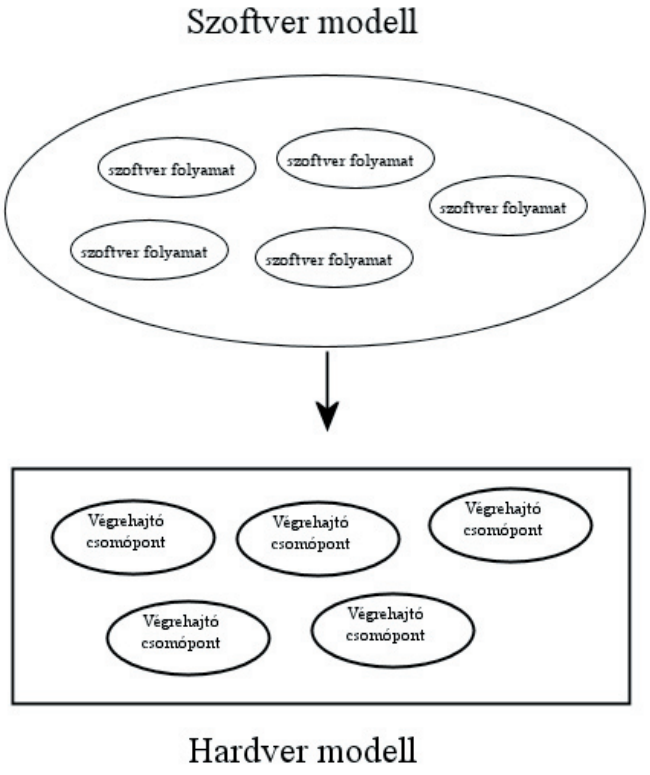
Ahhoz, hogy egy FPGA eszközt megfelelő hatékonysággal tudjunk párhuzamosan végrehajtható feladatokra programozni, szükséges egy arra alkalmas eszköz, és szükséges egy erre a célra alkalmas programozási modell. A feladatot olyan módon kell részfeladatokra bontani, hogy az illeszkedni tudjon az eszközre. A részfeladatok külön fizikai blokkok formájában fognak megjelenni, melyek nem rendelkeznek bonyolult vezérlési szerkezetekkel, viszont nagyon gyorsan végrehajtódnak, és egy időpillanatban egyszerre működnek. Ahhoz, hogy egy ilyen programot le tudjunk programozni, szükségünk van egy olyan programozási modellre, mely egyszerre biztosít lehetőséget a párhuzamos és az eljárás orientált programozásra. Kiválaszthatunk egy magas szintű nyelvet, amely rendelkezik a szükséges eszközökkel. Az ANSI C egy nagyon elterjedt, széles körben alkalmazott eljárás orientált nyelv, mely önmagában nem rendelkezik párhuzamos programozásra alkalmas eszközökkel, viszont függvénykönyvtárakkal bővítve képes ellátni a feladatot. A C azért is alkalmas választás, mert nagyon gyakran alkalmazzák beágyazott rendszerekben, illetve a szoft processzorok is programozhatók ezen a nyelven.

Tudjuk, hogy bizonyos programozási feladatok nem feltétlenül igényelnek párhuzamos megoldásokat, sőt, esetenként csak komplikáltabbá teszik a kódolást. Ezekben az esetekben praktikus lenne egy eljárás orientált környezetet alkalmazni, amiben leprogramozhatóak az olyan utasítások, melyek sokkal jobban illeszkednek abba a környezetbe. A szoft processzorok használata tökéletesen megfelel erre a célra. A szoft processzorok segítségével általában egyszerűen,

magas szintű nyelv segítségével kezelhetőek az eszközön található egyéb egységek, mint pl. memória, kommunikációs portok, stb. Ezen kívül lehetőség van ilyen módon a felkonfigurált logikai blokkok működését is irányítani, ezáltal ötvözve a párhuzamos programozás hatékonyságát az eljárás orientált nyelvek kényelmével, létrehozva így egy durva szemcsézettséggel rendelkező heterogén modellt. A durva szemcsézettség egy pozitív mellékhatása, hogy a folyamatok között kevesebb kommunikációra van szükség, mely tovább csökkentheti a program végrehajtásához szükséges időt, hiszen minden végrehajtási egységnek lehet saját memóriája, ahol elvégzi a számítást. Ezek a jól elkülönített blokkok akár külön órajellel is rendelkezhetnek, teljesen önállóan képesek működni. A részfeladatok által létrejött részeredményt a központi szoft processzor pedig összegezheti, vagy továbbadhatja más, elkülönített moduloknak.

2.6 Párhuzamos programozás modelljei

A magas szintű programnyelvek segítségével történő párhuzamos programozás alapvető problémája, hogy a nyelvekben nincs eszköz a párhuzamosság megfelelő kifejezésére. A C-ben egy ehhez közel álló lehetőség a szálak programozása, vagy a kevésbé közismert üzenettovábbítás interfész (Message-Passing Interface). A másik oldalon ott vannak azok az alacsony szintű nyelvek, mint a Verilog vagy VHDL, melyek kifejezetten erre a célra jöttek létre. Az egyes modulok létrehozására jó eszközként állnak rendelkezésünkre, viszont alacsony szintű nyelvek, így meglehetősen nehéz velük a komplex rendszerek megírása. A két különböző módszer ötvözésével létrehozható egyfajta hibrid nyelv. Az előzőeket összegezve, fizikai szinten szükségünk van egy absztrakt modellre, melyre programozunk, szoftver szinten pedig egy olyan modellre, amely illeszkedik a fizikai modellre.



2.ábra: Szoftver és hardver modell

A 2. ábrán látható egy hardver modell, mely félig autonóm végrehajtó csomópontokból áll, illetve egy erre illeszkedő programozási modell, mely lehet például a Communiting Sequential Process (CSP) modell.

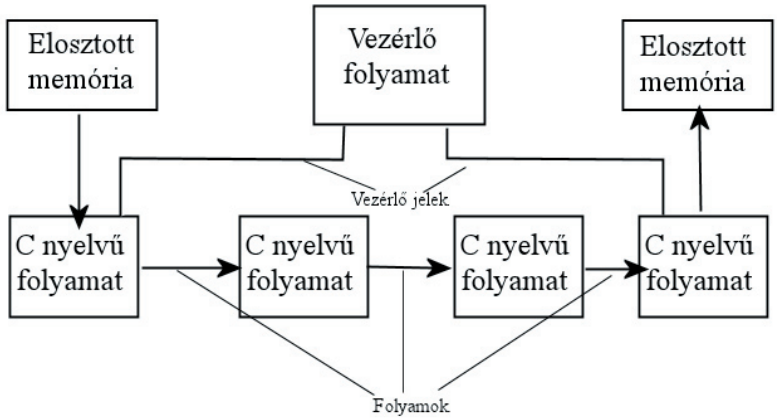
2.7 Communiting Sequential Process (CSP) modell

Ebben a programozási modellben minden folyamat egy önálló hagyományos szoftverhez hasonlítható, viszont a folyamatok közötti kommunikáció csak korlátozott mértékben lehetséges, jól definiált csatornákon keresztül zajlik. Az egyes folyamatok egymástól függetlenek, a Neumann architektúrával megegyező módon épülnek fel. Minden folyamatnak van egy saját memóriája, amivel szabadon gazdálkodhat. A hagyományos programozástól eltérően, ezen folyamatokat nem irányítja egy külső egység, ami időről időre meghívja őket, hanem a rendszerben fixen helyezkednek el, az adat pedig átáramlik a rendszeren. Egy ideális világban a folyamatok közötti adatáramlás folyamatos, nincs késleltetés az egyik folyamat kimenete és a másik bemenete között. Egy FPGA környezet-

ben azonban ez nem így történik, az egyes folyamatok áteresztőképessége eltérő lehet a folyamat összetettségétől függően. A CSP modell egy lényeges eleme, hogy a lokális memória hozzáférés sokkal kevésbé költséges, mint egy megosztott memóriát használó rendszer, így amikor csak lehet, kerülendő a megosztott memória használata. Ezt a párhuzamos programozási koncepciót nevezik lokalitásnak.

3. Impulse C

Az Impulse C egy CSP-re épülő programozási modell, melyben az egyes folyamatok külön szinkronizált, konkurens komponensek. Gondolhatunk rájuk úgy, mint alprogramokra, melyek között adatfolyamok valósítják meg a kommunikációt. Az így létrejött modulok azonban nem hívódnak meg, ahogyan azt az eljárás orientált nyelvekben megszokhattuk, hanem folyamatosan jelen vannak, és ha a bemenetükre adat érkezik, az átáramlik rajtuk, ezzel új adatokat állítanak elő, melyek a kimenő adatfolyamokon át távoznak.



3. ábra: Impulse C vezérlési séma

A 3. ábrán látható, hogy egy vezérlő folyamat irányítja a C nyelvben megírt folyamatokat, melyek egymás között adatfolyamok, illetve elosztott memória segítségével kommunikálhatnak. Az adatfolyamok alkotják az elsődleges kommunikációs interfészt, és jelentős feladatot látnak el a folyamatok szinkronizációjában. Egy jól megtervezett program attól lesz hatékony, hogy az adatfolyamok megfelelő hatékonysággal tudják kiszolgálni a folyamatokat.

Az Impulse C kiterjeszti az ANSI C függvényeket előre definiált, C kompatibilis függvényekkel, melyek lehetővé teszik a párhuzamos folyamatok közötti kommunikáción alapuló programozási modell megvalósítását. Ez a fajta modell az egyes modulok közötti kommunikáció vonatkozásában hasonló az adatfolyam modellekhez. A nyelv első sorban adatfolyam központú megközelítést vall, de kellően rugalmas, így akár elosztott memória, vagy vezérlőjelek használatával

is irányíthatjuk a folyamatainkat. A nyelvben megírt program jól alkalmazkodik az ismert fejlesztői környezetekhez, szimuláció céljából akár a szabványos gcc fordító segítségével lefordított programot is futtathatjuk.

3.1 Folyamatok az Impulse C-ben

Az Impulse C-ben létrehozható folyamatok hasonlítanak a szálak programozásához, ebből kifolyólag egy szálakon alapuló alkalmazás viszonylag könnyen átírható Impulse C alkalmazássá. Van azonban néhány fontos különbség a két modell között:

Szálprogramozás	Impulse C
A globális változók és a heap memória megosztható a szálak között	A heap memória implicit módon megosztottá tehető, a globális változók használatára viszont általában nincs lehetőség
A szálak ugyanazon a központi processzoron futnak	Minden folyamat egy külön processzoron vagy logikai blokkban kap helyet
A kommunikáció strukturált adatok megosztása, illetve szemaforok használata segítségével történik	A kommunikáció alapvetően a hardver által támogatott pufferek segítségével (FIFO) történik
A programot alkotó egységek dinamikusan épülnek fel, futnak, majd érnek véget	A program részegységei az inicializálás idejében fizikailag létrejönnek

3.2 Az Impulse C programok felépítése

Az Impulse C programok utasításkészlete megegyezik az ANSI C utasításkészletével, a különbség az, hogy olyan függvénykönyvtárakat használ, melyek nem részei az ANSI C szabványnak. Egy egyszerű program két fájlból áll. Az egyik tartalmazza szoftveres, vezérlő utasításokat, míg a másik a hardveren implementálni kívánt folyamatokat definiálja. A szoftveres utasítások futhatnak a szoft processzoron, vagy egy tesztelő környezetben a PC-n.

3.3 HelloWorld Impulse C-ben

Az [Függelék 1.]-ben található HelloFPGA példaprogramon keresztül bemutatásra kerül, hogyan lehet egy egyszerű Impulse C programot készíteni. A `HelloFPGA_sw.c` fájl a szoftveres részt írja le.

```
#include “co.h”
#include <stdio.h>
```

A program általában a `co.h` függvénykönyvtár hivatkozásával kezdődik, ami az Impulse C alapvető deklarációit, függvényeit tartalmazza. Az ebben található függvények mind `co_` preffixűek.

```
void Producer(co_stream output_stream) {
co_stream_open(output_stream, O_WRONLY, CHAR_TYPE);
p = HelloWorldString;
while (*p) {
     printf(“Producer writing output_stream with: %c\ n”, *p);
     co_stream_write(output_stream, p, sizeof(char));
     p++;
}
co_stream_close(output_stream);
```

A `Producer` függvény futtatja a szoftver bemeneti oldalát. Jelen esetben generál egy karaktersorozatot, amit majd átad a hardvernek egy adatfolyam segítségével.

```
void Consumer(co_stream input_stream) {
  char c;

  co_stream_open(input_stream, O_RDONLY, CHAR_TYPE);
  while (co_stream_read(input_stream, &c, sizeof(char)) == co_err_
none ) {
     printf(“Consumer read %c from input stream\ n”, c);
  }
  co_stream_close(input_stream);
}
```

A `Consumer` függvény a `Producer`-hez hasonlóan egy folyamon keresztül kommunikál a hardverrel, ez a függvény fogja feldolgozni az onnan visszaérkező jelsorozatot. A függvény a `printf` segítségével a konzolra írja a hardvertől fogadott jelsorozatot.

```
int main(int argc, char *argv[]) {
    int param = 0;
    co_architecture my_arch;

    printf("HelloFPGA starting...\ n");

    my_arch = co_initialize(param);
    co_execute(my_arch);

    printf("HelloFPGA complete.\ n");
    return(0);
}
```

A `main` függvény meghívja a `co_initialize` függvényt, mely minden Impulse C programban szerepel, gyakorlatilag az Impulse C alkalmazás belépési pontját határozza meg. Az átadott paramétert a programozó szabadon felhasználhatja. A függvény visszatérési értéke az architektúra leírása, melyet a `co_execute` függvénynek átadva tudjuk elindítani az alkalmazásunkat.

A program szoftveres részén kívül szükség van egy hardveres részre is, mely a `HelloFPGA_hw.c` fájlban kapott helyet [Függelék 2].A fájl elején a szoftveres részhez hasonlóan a `co.h` header fájl hivatkozása található, majd a `DoHello` függvény következik.

```
void DoHello(co_stream input_stream, co_stream output_stream) {
  char c;

  co_stream_open(input_stream, O_RDONLY, CHAR_TYPE);
  co_stream_open(output_stream, O_WRONLY, CHAR_TYPE);
  while (co_stream_read(input_stream, &c, sizeof(char)) ==   co_
err_none ) {
    // Do something with the data stream here
    co_stream_write(output_stream,&c,sizeof(char));
  }
```

```
  co_stream_close(input_stream);
  co_stream_close(output_stream);
}
```

A `DoHello` függvény két adatfolyam referenciát kap paraméterül, melyek közül az `input_stream` egy 8 bites bemenő adatfolyam, mely a `Producer` függvénnyel van összekapcsolva. Az `output_stream` a reprezentálja a feldolgozott adatokat, ezek szintén 8 bites karakterek, amelyek a `Consumer` függvénynek kerülnek átadásra. A beérkező adatfolyamot a lokális c változóba olvassa, majd módosítás nélkül a kimenetre írja. Végezetül lezárja az adatfolyamokat.

```
void config_hello(void *arg) {  // Konfigurációs függvény
  co_stream s1,s2;
  co_process producer, consumer;
  co_process hello;

  s1 = co_stream_create("Stream1", CHAR_TYPE, 2);
  s2 = co_stream_create("Stream2", CHAR_TYPE, 2);
  producer = co_process_create("Producer",
                (co_function) Producer, 1, s1);
  hello = co_process_create("DoHello",
                (co_function) DoHello, 2, s1, s2);
  consumer = co_process_create("Consumer",
                (co_function) Consumer, 1, s2);
  co_process_config(hello, co_loc, "PE0");  // Assign to PE0
}
```

A konfigurációs eljárás minden Impulse C alkalmazás kötelező eleme. Ebben definiáljuk az alkalmazás struktúráját, azaz, hogy milyen folyamatokat fogunk használni, és ezeket hogyan kötjük össze egymással. A `co_process_create` függvény segítségével hozzuk létre a folyamat példányokat, amik rendre a `producer`, `consumer`, illetve `hello` azonosítót kapják. A `co_stream_create` függvény létrehozza az adatfolyamokat, melyek a `producer` és `hello`, illetve `hello` és `consumer` között közvetítik az adatokat. Végezetül a `co_process_config` segítségével a `PE0` nevű hardver elemhez rendeljük a `hello` folyamot. Ez az egyetlen rész a forrásban, ami a cél eszközhöz kapcsolódik.

```
co_architecture co_initialize(int param) {
  return(co_architecture_create("HelloArch","generic",
                                config_hello,(void *) param));
}
```

A fenti inicializáló kódrészletben található `co_architecture_create` függvényhívás szintén minden Impulse C alkalmazás része. Ebben a függvényben egy egyszerű eljáráshívás kapott helyett, mely összeköti az előbb definiált konfigurációt az alkalmazás cél architektúrájával. Ez a cél architektúra jelen esetben egy általános hardver/szoftver platform. Az, hogy ténylegesen milyen hardverre fordítunk, nem a forráskód része, hanem a fordító beállításaihoz tartozik.

3.4 Az Impulse C függvények

3.4.1 Folyamatok létrehozása

Egy hardveres vagy szoftveres folyamatot a `co_process_create(arg1, arg2, arg3 [, args]...)` függvény segítségével definiálhatunk, ahol `arg1` egy sztringet hivatkozó mutató, ami a folyamat nevét tartalmazza. Ez a név monitorozáskor azonosítja az aktuális folyamatot. A második paraméter egy `co_function` típusú függvényre hivatkozó mutató, mely megmondja, hogy mely függvényhez lesz hozzárendelve a folyamat. A harmadik paraméter a bemeneti és kimeneti portok számait jelzi. Ez után a paraméter után annyi paraméternek kell következnie, amennyi a harmadik paraméter értéke. Egyúttal a második paraméter által hivatkozott függvénynek pontosan ugyanennyi paraméterrel kell rendelkeznie.

Az argumentumként átadható portok típusa a következők egyike lehet:

- `co_stream`: Egy pont-pont interfész, mely FIFO puffer segítségével továbbítja az adatokat.
- `co_signal`: Egy pufferelt pont-pont interfész, melyen folyamatok által küldött üzenetek továbbítódnak
- `co_memory`: Egy megosztott memória interfész, mely blokkok írását és olvasását teszi lehetővé

- `co_register`: Egy alacsony szintű, nem pufferelt hardver interfész
- `co_parameter`: Egy fordítás idejű paraméter

Példa egy folyamat létrehozására:

```
#define BUFSIZE 4
co_process procHost1;
co_stream s1;
s1 = co_stream_create(“s1”, INT_TYPE(16), BUFSIZE);
procHost1=co_process_create(“Host1”, (co_function)Host1, 1, s1);
```

3.4.2 Adatfolyamok létrehozása

Az adatfolyamok olyan egyirányú csatornák, melyeken keresztül különböző folyamatok kommunikálhatnak, legyenek azok hardveresek, vagy szoftveresek. Egy ilyen adatfolyamot a `co_stream_create(arg1, arg2, arg3)` függvény segítségével hozhatunk létre. Az első paraméter egy sztring, mely az adatfolyam monitorozáskor megjelenített nevet fogja jelölni. Második paraméterként egy típust és egy méretet kell megadnunk, ami az adatfolyam által közvetített adatelemek típusát és méretét jelöli. Az egyes típusok megjelöléséhez makrókat használhatunk, mint `INT_TYPE`, `UINT_TYPE`, vagy `CHAR_TYPE`. A harmadik paraméter definiálja az adatfolyam pufferének méretét. Amennyiben 1-et adunk meg, az adatfolyam nem fog puffert használni, így rögtön továbbítja az adatokat, amennyiben az lehetséges. Egy nagyobb puffer méret biztosabb, gördülékenyebb megoldást eredményez, viszont a nagyobb puffer méret nagyobb helyet igényel a hardverből is. Egy alkalmazás fejlesztésekor meg kell találni az alkalmazáshoz jól igazodó értéket az optimális teljesítmény érdekében. Egy példa a `co_stream_create` használatára:

```
co_stream_create(“IMG_VAL”, INT_TYPE(16), BUFSIZE);
```

3.4.3 I/O adatfolyamok

Ahhoz, hogy egy létrehozott adatfolyamot használni tudjunk, meg kell nyitnunk a csatornát, melyet a `co_stream_open(arg1, arg2, arg3)` függvény segítségével tehetünk meg. A függvény első paramétere az adatfolyam, amit meg szeretnénk nyitni, a második az adatfolyam típusa, ami lehet `O_RDONLY` vagy `O_WRONLY`, a harmadik pedig a kommunikáció során felhasznált adat típu-

sa és mérete. Mivel az adatfolyamok két pont közötti egyirányú csatornák, ezért pontosan egy folyamat írhatja, egy pedig olvashatja egy időben. Egy példa az adatfolyam megnyitására:

```
co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));
```

Amennyiben egy olyan adatfolyamot szeretnénk megnyitni, mely már meg van nyitva, akkor a `co_err_already_open` hibakódot kapjuk visszatérési értékként. Ha már nincs többé szükségünk az adatfolyamra, lezárhatjuk azt a `co_stream_close` függvény segítségével. A függvény egy `EOS` tokent küld a fogadó fél felé, aki ez által érzékeli, hogy az adatfolyam véget ért.

3.4.4 Adatfolyam írása

Egy írásra megnyitott adatfolyamon adatot küldhetünk a `co_stream_write(arg1, arg2, arg3)` függvény segítségével. Az első paraméter az adatfolyam, amin az adatot küldeni szeretnénk, a második a küldeni kívánt adat, a harmadik pedig az adatfolyam mérete. Amikor a függvényt meghívjuk, először megvizsgálja, hogy van-e az adatfolyam pufferében hely újabb adat számára. Amennyiben van, belekerül az adat, ha viszont nincs, akkor mindaddig blokkolja az írást, amíg fel nem szabadul a szükséges hely. Ez egy rosszul megtervezett architektúra esetén könnyen holtponthoz vezethet, amiből az alkalmazás sosem fog tudni kilépni. Példa az adatfolyamra való írásra:

```
co_stream_open(output_stream, O_WRONLY, INT_TYPE(32));
for (i=0; i < ARRAYSIZE; i++) {
    co_stream_write(output_stream, &data[i], sizeof(int32));
}
```

3.4.5 Adatfolyam olvasása

Egy adatfolyamon kétféle olvasási műveletet lehet végrehajtani. Tesztelhetjük, hogy az adatfolyam lezárásra került-e már, illetve adatot olvashatunk belőle. A `co_stream_read` függvény segítségével olvashatunk az adatfolyamból. Amennyiben az olvasást egy lezárt adatfolyamon hajtjuk végre, logikai igaz értéket kapunk eredményül. Ennek megfelelően egy adatfolyam végéig tartó olvasást elvégző ciklus a következőképpen nézhet ki:

```
co_stream_open(input_stream, O_RDONLY, INT_TYPE(32));
while(co_stream_read(input_stream) == co_err_none) {
. . . // Process the data here
}
co_stream_close(input_stream);
```

Amennyiben az adatfolyamot lezárjuk a `co_stream_close` segítségével, az összes el nem olvasott adat eldobásra kerül, és az `EOS` tokent küld a folyamatnak. Amennyiben nincs az adatfolyamban `EOS`, sem adat, az olvasó blokkolódik, amíg `EOS` vagy adat nem érkezik. Fontos, hogy az `EOS` token csak akkor íródik az adatfolyamra, ha az író fél zárja le azt.

3.4.6 Regiszterek használata

Bizonyos esetekben előfordulhat, hogy két folyamat közötti kommunikációt nem az adatfolyam csatornáin, hanem egy közösen olvasható memóriarész segítségével szeretnénk megvalósítani. Ennek a kivitelezésében segítenek a regiszterek. Egy regisztert egy folyamat írhat, de akár több folyamat is olvashat. A regiszterek tipikusan a hardveren létrejövő eszközök, a szoftveres környezetből általában nem elérhetőek. Egy regisztert a `co_register_create` függvénnyel hozhatunk létre, mely első paraméterben egy nevet tartalmazó sztringet, második paraméterben pedig egy típust, illetve méretet vár. A regiszterek módosításához a `co_register_put`, illetve a `co_register_write`, míg olvasásához a `co_register_get`, illetve `co_register_read` függvények használhatók.

3.4.7 Megosztott memória használata

Adatfolyamok helyett használhatunk megosztott memóriát is a folyamatok kommunikációjához. Ezzel a módszerrel akár az FPGA kártyára elhelyezett memóriát is hivatkozhatjuk a programunkból. A megosztott memória segítségével hardveres, vagy szoftveres folyamatok közötti kommunikációt is megvalósíthatunk. A memória használatához a `co_memory` típust használhatjuk, létrehozásához pedig a `co_memory_create(arg1, arg2, arg3)` függvényt. Az első paraméter szokás szerint egy sztring, mely a monitorozáskor lehet segítségünkre. Második paraméterként a hardveren található memória fizikai helyét kell megadnunk, utolsó argumentumként pedig a létrehozandó méretet adjuk át. Egy egyszerű példa a megosztott memória létrehozására:

```
co_memory memory;
memory = co_memory_create(“Memory”, “mem0”, MAXLEN*sizeof(char));
```

A memóriából való olvasáshoz a `co_memory_readblock`, írásához pedig a `co_memory_writeblock` függvények használhatók. Ezek 4 paraméterrel dolgoznak. Első paraméterként az írandó/olvasandó memóriát várja. A második paraméterként átadott offszet érték azt jelöli, hogy a címzett memória mely részéről szeretnénk olvasni. A harmadik paraméter egy mutató, mely arra a lokális területre mutat, amelyből írni, vagy amelybe olvasni szeretnénk. Az utolsó paraméter pedig az adat mennyiségét jelöli. Lényeges, hogy megosztott memória használata esetén a hozzáférés nem szinkronizált, így explicit módon kell biztosítani, hogy az olvasást akkor végezzük el, miután az írás véget ért. Ezt könnyen megtehetjük például egy vezérlőjel (`co_signal`) segítségével. A `HelloFPGA` program átiratát, melyben a `Producer` és `DoHello` közti adatfolyamot lecseréljük megosztott memória általi kommunikációra, megtalálhatjuk az [Függelék 3] és [Függelék 4]-ben.

3.5 Matematikai műveletek

Az Impulse C lehetőséget biztosít arra, hogy valós számokon hajtsunk végre műveleteket. Az FPGA-hoz hasonló rendszereknél jellemzően a fix pontos számábrázolást választják erre a célra. Nincs ez másként ennél a fejlesztői eszközrendszernél sem. Az Impulse C-ben használhatunk előjeles és előjel nélküli fix pontos számokat. Ezeknek három különböző bitszélességű megvalósítása létezik. Az előjeles típusok a következőek: `co_int8`, `co_int16`, `co_int32`, illetve az előjel nélküliek: `co_uint8`, `co_uint16`, `co_uint32`. Az Impulse C a fix pontos számokon végezhető műveleteket a `co_math.h` függvénykönyvtárban található függvényekkel valósítja meg. Ezek a függvények értelemszerűen fix pontos számokat várnak. Egy fix pontos változónak értéket adhatunk a szám hexadecimális ábrázolásával, illetve egy már meglévő egész típusú számot átkonvertálhatunk függvények segítségével. Egy `int` típusú számot fix pontos számmá konvertálhatunk az erre a célra létrehozott makrók segítségével. Ezek a makrók a következők: `FXCONST8`, `FXCONST16` és `FXONST32`, melyek rendre a `co_uint8`, `co_uint16` és `co_uint32` típusú, az átadott paraméterrel egyenérté-

kű visszatérési értékkel rendelkeznek. Az így előállított számot típuskényszerítéssel előjelessé tehetjük. A `co_math.h` makróinak mindegyike két, vagy három paramétert vár, melyek közül az első, vagy az első kettő a szám, amin a műveletet el kívánjuk végezni, az utolsó pedig a tizedes pont mögötti törtrész hossza. Példaként tekintsük az alábbi kódot:

```
co_int16 a = (co_int16) FXCONST16(96, 7);
```

A fenti kódrészletben az a 16 bites előjeles változó a 96 számot fogja tartalmazni, ahol az első bit jelöli a pozitív előjelet, a következő 8 az egészrészt, és a fennmaradó 7 bit pedig a tört részt. Az ilyen módon tárolt számokon makrók segítségével alapvető matematikai műveleteket végezhetünk el.

3.5.1 Összeadás

Összeadáshoz az `FXADD8`, `FXADD16`, `FXADD32` makrókat használhatjuk. Két szám összeadása történhet például az alábbi módon:

```
co_int16 a, b, c;
 a = 0xFF00;            // -1.0
 b = 0x0180;            //  1.5
 c = FXADD16(a, b, 8);  // 0x0080 == 0.5
```

A változók értékét a 16 bites alakjuk segítségével adtuk meg. A deklarációjukból látszik, hogy előjel nélküliek, az összeadásnál pedig jelöltük, hogy 8 biten ábrázoljuk a tört részt, így a szám egész részére 7 bit jut.

3.5.2 Szorzás

Fixpontos számok szorzása az `FXMUL8`, `FXMUL16`, `FXMUL32` makrók segítségével végezhető el. A szorzás tulajdonságaira való tekintettel a művelet eredményét egy kétszeres pontosságú számban tárolja. Visszatérési értékként ennek az alsó felét adja vissza. Ebből kifolyólag, ha a cél eszköz nem támogatja a 64 bites számokat, akkor a 32 bites számok összeszorzása sem lesz lehetséges. Példa két 32 bites fixpontos szám összeszorzására:

```
co_int32 a, b, c;
 a = 0x00002000;        // 32.0
 b = 0x80000080;        // -0.5
 c = FXMUL32(a, b, 8);  // 0x80001000 == -16.0
```

3.5.3 Osztás

Fixpontos számok osztását az `FXDIV8`, `FXDIV16`, illetve `FXDIV32` makrók teszik lehetővé. Az osztás a szorzáshoz hasonlóan dupla pontossággal hajtódik végre. Két fixpontos szám osztása történhet az alábbi módon:

```
co_int16 a, b, c;
 a = 0x1000;             // 4.0
 b = 0x0100;             // 0.25
 c = FXDIV16(a, b, 10);  // 0x4000 == 16.0
```

3.5.4 Nem szabványos bitszélességű számok használata

Lehetőség van a szabványostól eltérő bitszélességű számok használatára is. Fontos lehet tudni, hogy szoftveres szimuláció esetében ezek a számok felminősítésre kerülnek a legközelebbi szabványos típusra, így eltérő eredményt adhatnak, mint a hardveres implementáció. Bizonyos FPGA processzorok tartalmaznak hardveresen implementált szorzást végrehajtó egységeket. A szorzást - amennyiben lehetséges - érdemes ezeken az egységeken végrehajtani, mivel nagyon hatékony megoldást nyújt. Ezek az egységek általában maximum 18x18-as méretű adatokat tudnak összeszorozni. Érdemes tehát 18 bites típusokat használni, amennyiben szorzásokat szeretnénk a számokon elvégezni. A következő kódrészlet bemutatja, hogyan lehet hatékonyan használni ezeket az egységeket:

```
co_int18 a,b;
 co_int36 c;
 c = (int64)a * (int64)b;
```

A 18 bites számokat a szorzás elvégzése előtt 64 bitessé kényszerítjük, hogy a 32 bites változóhoz való hozzárendeléskor megfelelően vágja le a felső biteket a rendszer.

3.6 Optimalizációs eljárások

Az Impulse C-ben megírt programok alapvető tulajdonsága a szerkezete, amelynek köszönhetően az egyes folyamatok egymástól függetlenül léteznek, és valamilyen csatornán továbbítják egymásnak az adatokat. Az egyes folyamatok belső szerkezete azonban szintén a párhuzamos programozásnak megfelelő logika szerint kerül kialakításra. Az Impulse C fordítója számos módszert alkalmaz arra, hogy az egyébként szekvenciális utasításkészlettel rendelkező C nyelvben megírt kódot párhuzamos végrehajtáshoz optimalizálja. Lehetőség van kifejezés-szintű optimalizálására, C függvényben található blokkok optimalizálására, csővezeték kialakítására, illetve a fejlesztői környezet lehetőséget biztosít RTL sematikus ábra szerkesztésére.

3.6.1 Kifejezés szintű optimalizálás

A fordító optimalizáló eljárása automatikusan párhuzamosítja az egymás után következő, egymástól független kifejezések sorozatát, illetve egy összetett kifejezésen belül lévő részkifejezések végrehajtását. Vegyük alapul a következő utasítássorozatot:

```
X = a + b + c;
X = X << 2;
Y = a - c;
```

Az optimalizáló egyesíti az első sorban lévő összeadást, a második sorban lévő eltolást és a harmadik sorban lévő kivonást egyetlen kifejezéssé. Érdemes tudni, hogy sok hasonló kódrészlet esetén a fordító által létrehozott kód ugyan egy lépéses végrehajtást eredményezhet (amennyiben nincs egymásra épülő utasításrészlet), ugyanakkor nagy késleltetést okozhat a futásban, a viszonylag bonyolult összetettséggel rendelkező logikai blokk miatt.

3.6.2 Blokkok optimalizálása

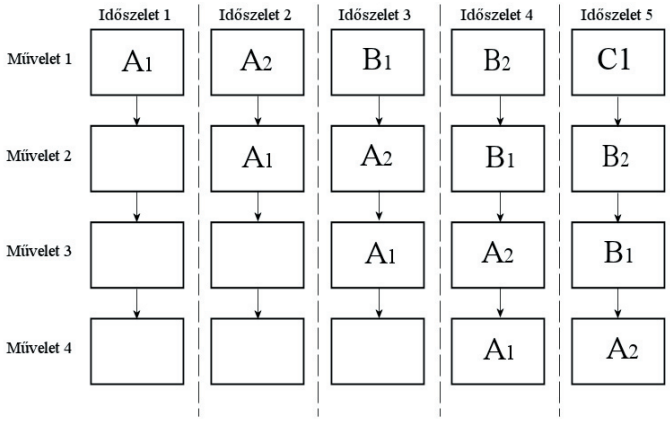
Egy blokkon belüli utasítássorozat a kifejezés szintű optimalizáláshoz hasonlóan kerül feldolgozásra, továbbá például ha egy elágaztatáshoz ér, akkor az elágazás mindkét ágát végrehajtó folyamatot legenerálja. Tekintsük az alábbi kódot:

```
if (odd) {
    x = a + b + c;
    x = x << 2;
}
y = a - c;
```

Ebben az esetben a fordító előállít egy olyan folyamatot, ami kiszámítja az a + b + c << 2 értékét, továbbá egy olyat, ami az kiszámítja a - c értékét.

3.6.3 Csővezeték kialakítása

A csővezeték kialakítása egy elég elterjedt módja a szekvenciális utasítások párhuzamos végrehajtásának. Az elképzelés szerint, az egymásra nem épülő részfeladatok egymással párhuzamosan is elvégezhetőek. Amennyiben n különböző műveletet kell végrehajtanunk m különböző adaton, az m*n utasítást jelentene a szekvenciális végrehajtás során. A csővezeték használata esetén az n műveletet n egymástól függetlenül, folyamatosan dolgozó részegység végzi. Rendre veszi a következő bemenő értékeket, végrehajtja az utasításait, majd továbbadja az eredményt egy ugyanilyen elven működő részegységnek. Ezzel a módszerrel nem kell megvárni egy-egy kifejezés kiértékelésének eredményét, hogy a következő kifejezés kiértékelése kezdetét vegye.



4. ábra: A csővezeték szemléltetése

A 4. ábrán látható egy műveletsorozat, melyet a csővezeték technikáját alkalmazva hajtunk végre. Impulse C-ben a `CO PIPELINE` pragmát használva jelölhetünk meg egy ciklust, jelezve ezzel, hogy a fordító olyan kódot állítson elő belőle, amely a csővezeték technikáját alkalmazza. Példaként nézzük az alábbi kódrészletet:

```
while (1) {
#pragma CO PIPELINE
  if (co_stream_read(istream,&data,sizeof(data)) != co_err_none)
      break;
  sum += data;
  co_stream_write(ostream,&sum,sizeof(sum));
}
```

Ebben az esetben a ciklusban minden lépésnél olvas egy újabb adatot, illetve minden lépésben összegez és a kimenő folyamra írja az eredményt. A `CO PIPELINE` pragma megadása nélkül az összes lépést külön óraciklusban végezné el. A csővezeték használatával 100 bemenő jel esetén 101 ciklus alatt fejeződik be a ciklus, míg csővezeték nélkül 200 lépésbe telik mindez.

3.6.4 Ciklusok kibontása

Bizonyos esetekben a ciklusok által felvett értékek fordítás időben eldönthetőek. Ebben az esetben egy makróhoz hasonló eljárás által kibonthatjuk a cik-

lusok tartalmát. A `CO UNROLL` pragma megadásával kérhetjük a fordítót, hogy hajtsa végre a kibontást. Vegyük például az alábbi kódrészletet:

```
int i;    // Az index típusa int kell hogy legyen
...
for (i=0; i<10; i++) {
#pragma CO UNROLL
 sum += A[i];
}
```

Ha a fordító kibontja a ciklust, akkor a következő utasítássorozatot generálja:

```
sum += A[0];
sum += A[1];
...
sum += A[9];
```

Az ilyen módon nyert kódot még mindig ugyanúgy tíz lépésbe telik elvégezni, mivel az `A` tömbnek egy lépésben csak egy értékét olvashatjuk ki. A fordító viszont tovább optimalizálja a kódot, és a tömbök minden egyes értékét külön regiszter változóba helyezi, ez által a végső kód valahogy így néz ki:

```
sum += A_0;
sum += A_1;
...
sum += A_9;
```

Az ilyen módon generált kód viszont már egyetlen lépésben elvégezhető, ezzel nagyban javítva a teljes kód futásának teljesítményét.

3.7 HDL függvény beágyazása

Az Impuse C kifejezőereje nagyon nagy ugyan, de bizonyos esetekben szükség lehet alacsony szintű megoldáshoz fordulni. Más magas szintű nyelvek esetében megszokhattuk, hogy lehetőség van úgynevezett inline kódok beszúrására, mely más, általában alacsonyabb szintű nyelven írt kód beágyazását jelenti a programunkba. Ezzel az eszközzel nagyban megnöveli a programozó szabadsá-

gát, lehetőséget ad arra, hogy a futási teljesítmény szempontjából különösen kényes részeket saját kézzel, a magas szintű nyelv bizonyos optimalizációs eljárásait megkerülve írjuk meg. Az Impulse C is rendelkezik ehhez hasonló eszközzel. A `CO IMPLEMENTATION` pragma segítségével VHDL vagy Verilog nyelven írt függvényeket hivatkozhatunk. Ha egy Impulse C függvényben elhelyezzük a `CO IMPLEMENTATION` pragmát, azzal azt jelezzük a fordító számára, hogy az adott függvénynek létezik egy másik, hardver közeli nyelven megírt implementációja. A fordító a HDL függvények három különböző típusát különbözteti meg: kombinációs logika, regisztrált-aszinkron logika (nem determinisztikus késleltetéssel) és csővezetékes logika (determinisztikus késleltetéssel).

3.7.1 Kombinációs függvények és eljárások

Ha kombinációs logikát szeretnénk hivatkozni a forrásszövegben, akkor a logic segítségével jelezhetjük ezt. A szintaxis a következő:

```
#pragma CO implementation myfunction logic
```

Ahol a myfunction a hivatkozni kívánt függvény neve. Vegyük szemügyre az alábbi kódrészletet:

```
co_int8 combFun(co_int4 a, co_int4 b)
 {
 #pragma CO implementation combFun logic
   co_int8 r=a;
   r=(r<<4)|b;
   return r;
 }
```

Ennek az egyszerű függvénynek a VHDL megfelelője a következőképen nézhet ki:

```
entity combFun is
   port (
     signal a : in std_ulogic_vector(3 downto 0);
     signal b : in std_ulogic_vector(3 downto 0);
     signal r_e_t_u_r_n : out std_ulogic_vector(7 downto 0));
 end;
```

```
architecture test of combFun is
begin
  r_e_t_u_r_n <= a & b;
end test;
```

Látható, hogy a két nyelvben megadott függvény neve, paramétereinek neve és bitszélessége megegyezik. Mivel a C nyelvben is egy függvényt definiáltunk, melynek szükségszerűen van egy visszatérési értéke, ezért a HDL nyelvben megírt függvényben is meg kell adnunk, hogy mit szeretnénk visszatérési értékként átadni. Ezt a fenti példában a `r_e_t_u_r_n` nevű változóval tettük meg.

3.7.2 Aszinkron regisztrált függvények és eljárások

A regisztrált aszinkron komponensek olyan ütemezett komponensek, amelyek lefutásához szükséges ciklusok száma nem determinisztikus. Egy ilyen komponens hivatkozásához az `async` használata szükséges:

```
#pragma CO implementation myfunction async
```

Egy ilyen komponens megírása már bonyolultabb kódot eredményez, mint egy kombinációs komponens létrehozása. Az alábbi kód VHDL megfelelőjét a [Függelék 5]-ben találjuk meg:

```
co_int32 asyncFun(co_int32 i1)
 {
 #pragma CO implementation asyncFun async
   return i1;
 }
```

A VHDL-ben írt kód bonyolultsága abból fakad, hogy a modulnak el kell végeznie a hardver vezérlését is, beleértve az órajel alapú ütemezést. Ehhez szükségszerű bizonyos vezérlő jeleket deklarálni. A `clk` vezérlőjel az órajelhez, míg a `reset` a reset jelhez van kötve. Végrehajtás folyamán a `request` vezérlőjel pontosan egy ciklus idejéig aktív, hogy elindítsa a végrehajtást. Az `acknowledge` vezérlőjelet akkor kell aktívra állítani, amikor a műveletek készen vannak, és ezek eredményei rendelkezésre állnak a kimenő vezérlőjeleken

keresztül. Az `acknowledge` vezérlőjelnek és az összes kimenő vezérlőjelnek aktívnak kell maradniuk a következő kérésig. Amikor a `request` jel aktívvá válik, az `acknowledge` jelnek inaktívvá kell válnia. Formálisan fogalmazva az `acknowledge` jel értékének mindig a következőképpen kell alakulnia:

```
acknowledge <= NOT request and calc_completed;
```

Ahol a `calc_completed` jelöli, hogy az elvégzendő műveletek készen állnak.

3.7.3 Csővezetékes függvények és eljárások

Meghatározott késleltetésű csővezetékes logika létrehozásához az alábbi pragma használatát vehetjük igénybe:

```
#pragma CO implementation myfunction pipeline latency=2
```

Ahol a `latency`-nek megadott érték határozza meg a késleltetést, ami jelen esetben két ciklus. Példaként tekintsük az alábbi C függvényt:

```
co_int8 combFun(co_int4 a, co_int4 b)
 {
 #pragma CO implementation combFun logic
   co_int8 r=a;
   r=(r<<4)|b;
   return r;
 }
```

Az ennek megfelelő VHDL kódú függvény a [Függelék 6] helyen található módon nézhet ki. Ahogyan az aszinkron regisztrált komponenseknek, a csővezetékeseknek is rendelkeznie kell `clk`, `reset` és a paraméterekhez tartozó vezérlőjelekkel. Ezen kívül az ilyen moduloknak szükséges egy `ce` vezérlőjel is. Amikor a `ce` vezérlőjel aktív, a csővezetéknek fogadnia kell a bejövő adatokat, és pontosan N ciklussal később ki kell küldenie az ezekhez tartozó kimenő adatokat. N jelen esetben a pragmában megadott késleltetési érték.

4. Az Impulse C bővítése függvénykövnytárakkal

Az Impulse C egy olyan eszközrendszer, mely a C alap függvényeit párhuzamos programozási eszközökkel egészíti ki függvénykönyvtárak segítségével. Mivel a rendszer felépítése moduláris, jól bővíthető. A függvénykönyvtárak átírhatóak, saját függvényekkel kiterjeszthetőek. Az Impulse C egy-egy könyvtárát egy deklarációkat tartalmazó XML fájl és egy vagy több – az ehhez tartozó implementációt tartalmazó - HDL fájl reprezentálja. A rendszer alapvető függvényeinek definícióját a `target.xml` dokumentumban találhatjuk meg. A fejlesztők azt javasolják, hogy amennyiben lehetséges ne módosítsuk az alapvető funkciókat, mivel a fejlesztői környezet későbbi kiadásaival könnyen inkompatibilissé válhat alkalmazásunk. Amennyiben ki szeretnénk bővíteni az alap könyvtárat, inkább hozzunk létre egy saját könyvtárat, és hivatkozzuk azt. A függvénykönyvtár definíciója szintén XML fájlban kerül megadásra, melyet hivatkozni a `target` fájlhoz hasonlóan lehet. Az Impulse C rendelkezik néhány előre definiált definíciós fájllal a különböző gyártók által készített kompatibilitás céljából. Tekintsünk például egy általános definíciós fájlt, a `generic.xml`-t. A `pe` elem által hivatkozott fájl jelöli az alkalmazni kívánt `target.xml`-t, és annak elérési útvonalát. A `generic.xml` ide vonatkozó részlete:

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Generic">
  <pe name="pe0" target="VHDL/target.xml"/>
…
</architecture>
```

A vastag betűvel szedett kódrészletben található a `target.xml` fájlra való hivatkozás. Az alap eszközkészlet kiterjesztéséhez ebben a dokumentumban hivatkozhatjuk saját könyvtárainkat a `library` elem segítségével. Például a Xilinx processzoron végezhető lebegőpontos számítások függvényei a `float.xml` és a `float_fast.xml` fájlokban vannak definiálva. Ezekre a fájlokra a következőképpen hivatkozik az előre definiált `xilinx_generic.xml` fájlban:

```
<?xml version='1.0'?>
<!DOCTYPE architecture PUBLIC "" "">
<architecture version="1.0" name="Xilinx Generic">
…
<library name="float" file="VHDL/Xilinx/float.xml"/>
<library name="float_fast" file="VHDL/Xilinx/float_fast.xml"/>
</architecture>
```

A `name` attribútum a könyvtár nevét, a `file` pedig a hozzá tartozó fájlt és elérési útját definiálja. A `target` és `library` fájlok szintaktikája nagyjából azonos. Legkülső szinten egy `target` vagy egy `library` elem áll, melyeknek gyermekei definiálják, hogy az egyes műveletek hogyan implementálandók HDL nyelven. Rendelkezhetnek két speciális jelentésű gyermekkel, melyek tartalma módosítás nélkül kerül a generált HDL fájlba. Ez a két elem a `header` és az `include`. Ezen elemek csak egyszer szerepelhetnek a dokumentumban. Az alábbi kódrészlet a target.xml fájlból demonstrálja ezek használatát:

```
<target version="1.0">
  <header>
-- TARGET: VHDL
  </header>
  <include>
library impulse;
use impulse.components.all;
</include>
```

A `header` és `include` elemeken kívül tartalmazhatja még az `io`, `require`, `operator`, vagy `primitive`, elemeket. Az `io` elemet az Impulse belső használatra tartja fenn, a többi elem jelentését az alábbiakban találjuk.

4.1 Forrásfájlok beágyazása

A `require` elem segítségével megadhatunk egy művelet HDL, vagy más hardverleíró nyelven írt implementálását reprezentáló külső fájlt. Ez hasonló a C előfordítója által használt `#include` direktívához, viszont itt a forrást ágyazzuk be a header fájlba. A `require` elem a következő három attribútummal rendelkezik: `file`, `dst`, `type`. Melyek rendre a hivatkozott fájl elérési útja, a projekt

lefordított hardver fájljainak könyvtára, végül a hivatkozott forrás típusa, melynek értéke kötelezően `hdl`-nek kell lennie.

4.2 Művelet vagy függvény implementálása

Egy-egy művelet, vagy függvény különböző módokon lehet implementálva. Lehet beépített, makró, VHDL függvény vagy komponens. Minden `operator` vagy `primitive` elem megadja, hogy milyen módon kell implementálni, a tartalmazó elemek pedig leírják az implementáció részleteit.

4.3 Műveletek definiálása

Egy műveletet az `operator` elem segítségével definiálhatunk. Az ilyen módon létrehozott műveletek beágyazódnak a C nyelvbe, ahol a fejlesztő header fájl vagy prototípus nélkül alkalmazhatja azt. Az `operator` elemnek az egyetlen kötelező attribútuma a `name`. Ez a név megadja, hogy melyik belső Impulse C művelet definícióját tartalmazza az elem. A használható neveknek egy előre definiált listája közül kerülhet ki az attribútum értéke. Egy művelet típusa az attribútumok értékei alapján dől el. Ha a `primary` attribútum `true` értéket vesz fel, a művelet nem fog részkifejezésként megjelenni a generált HDL fájlban. Az alapértelmezett érték `false`.

4.4 Függvény definiálása

Egy C függvény implementációját a `primitive` elem segítségével hivatkozhatjuk. Az elem három kötelező attribútummal rendelkezik: `name`, `proc`, `type`. Ezek rendre a művelet Impulse C-ben megjelenő belső neve, a C függvény implementációjának neve és az implementáció típusa (`component` vagy VHDL `function`). Ahhoz, hogy a C függvényt egy fejlesztő hardveres implementáláshoz használhassa, egy prototípust elérhetővé kell tenni (pl. egy header fájl segítségével).

4.5 Az implementáció típusa

4.5.1 Beépített HDL művelet

Ahhoz, hogy egy adott művelet natív módon a választott HDL nyelven legyen implementálva, definiálni kell a `builtin` attribútumot, és az értékét `true`-ra állítani.

4.5.2 Makrók

Lehetőségünk van a C-ben található makrókhoz hasonló makrók definiálására. Egy művelethez makrót rendelhetünk a `macro` attribútum megadásával, melynek értéke bekerül a generált HDL kódba. A rendelkezésre álló argumentumok az `arg` elemek sorrendjének megfelelően rendre a `%0`, `%1`, stb. jelöléssel hivatkozhatóak. A makró törzsében műveleteket végezhetünk ezeken az argumentumokon. Az `operator` elemnek kötelezően legalább egy `arg` elemet kell tartalmazni, mely az operandust jelöli. Az `arg` elem kötelező attribútuma a `type`, mely értékül felveheti az `in1`, `in2`, stb. egészeket jelölő értékeket, vagy a `param`-t, mely numerikus konstanst jelöl. Ha a típus `in` preffixű sztringet kap értékül, a következő attribútumok is használhatóak:

- `primary`: ha ez az érték `true`, akkor az operandus vezérlőjelhez lesz kötve, nem kifejezésként lesz átadva.
- `signed`: ha `true`-ra állítjuk az értékét, a paraméterben átadott érték előjelesként lesz értelmezve.

Ha a `type` értéke param, akkor kötelező megadni egy `name` attribútumot is, melynek értéke az argumentum belső fordító szerinti neve. Példaként nézzük meg, hogyan lehet definiálni az aritmetikai jobbra tolás műveletet a Verilog `>>>` operátora segítségével. Az operátor két operandusa a bemenet és az eltolás mértékéke. A bemenet egy előjeles egész, az eltolás mértéke pedig a belső fordító által `param`-ként ismert egész konstans.

```
<operator name=”asr” macro=”(%0 >>> %1)”>
<arg type=”in1” primary=”true” signed=”true”/>
```

```
<arg type=”param” name=”param”/>
</operator>
```

4.5.3 VHDL függvények

Az operátorok implementálhatóak a VHDL function eszközének segítségével is. Ehhez adjuk meg a function attribútumot, értékül pedig adjuk neki az implementáló VHDL függvény nevét. Az implementáló VHDL függvény változó argumentumainak a `std_ulogic_vector` típusúnak, az egész konstans argumentumoknak pedig `natural` típusúnak kell lenniük. A függvénynek `std_ulogic_vector` típusú visszatérési értéket kell adnia. Az operátorok argumentumai fordító által definiáltak és nem módosíthatók. Példaként nézzük meg az Impulse C `sign_extend` operátorának VHDL megvalósítását a `sign_ext` nevű függvénnyel. Az operátor definiálása:

```
<target version=”1.0”>
<operator name=”sign_extend” function=”sign_ext”/>
</target>
```

4.5.4 Az operátor implementálása:

```
function sign_ext(v : std_ulogic_vector; size : natural) return
std_ulogic_vector;

  function sign_ext(v : std_ulogic_vector; size : natural) return
std_ulogic_vector is
    variable res : std_ulogic_vector (size-1 downto 0);
  begin
    res(size-1 downto v’length) := (others => v(v’left));
    res(v’length-1 downto 0) := v;
    return res;
end function;
```

4.5.5 Függvény definiálása

A következő példakód az Impulse C `satredu32` függvényét asszociálja a VHDL `satredu` függvényéhez:

```
<primitive name=”satredu” cycles=”0” proc=”satredu32”
type=”function”>
    <signal name=”i1” type=”input” carg=”0” width=”*”/>
```

```
    <signal name="i2" type="param" carg="1"/>
</primitive>
```

Amennyiben VHDL segítségével implementálunk egy függvényt a `cycles` paraméternek nullának kell lennie. A vezérlőjel `input` típusai változókat, a `param` típusai pedig konstansokat jelölnek. Minden bemenő jelnek van egy bitszélessége, amit a `width` attribútummal adhatunk meg. Az attribútum a következőket veheti fel értékül: egy egész literált, * jelet, ami tetszőleges szélességet jelöl, illetve egy hash-jelölést, ami azt jelöli, hogy az érték egy másik bemenő jelre lesz kicserélve. Például az `#i1` azt jelöli, hogy a `width` értéke azonos lesz az `i1` vezérlőjel szélességével. Hogy a függvényt használhassuk az Impulse C forrásban, deklaráljuk a C függvényt az alkalmazás kódjában prototípusként:

```
co_uint32 satredu32(co_uint32 i1, const co_uint32 i2);
```

4.5.6 Kompnensek

Az operátorok, illetve függvények HDL komponensek segítségével implementálhatók. A komponens lehet egy VHDL `entity`, vagy egy Verilog `module`. A komponensek három különböző típusúak lehetnek: kombinációs logika, regisztrált-aszinkron logika, csővezetékes logika. Ezek a típusok megegyeznek a [3.7 HDL függvény beágyazása] helyen definiáltakkal. A típusokat az XML-ben a `cylces` és `rate` attribútumok segítségével különböztethetjük meg egymástól. A HDL komponenshez tartozó interfész vezérlőjelekből áll, melyeket a `signal` elemek segítségével adhatunk meg. Minden jelnek van egy `name` és egy `type` attribútuma, de egyes vezérlőjeleknek más attribútumokra is szüksége lehet. A bemenő, kijövő jelek, megfeleltethetők a paramétereknek és visszatérési értéknek, viszont különbözőképpen értendők operátorok és primitívek esetén:

Paraméter	Művelet típusa	signal attribútumai
0	operator	type=”in1”
	primitive	type=”input” carg=”0”
1	operator	type=”in2”
	primitive	type=”input” carg=”1”
Visszatérési érték	operator	type=”out1”
	primitive	type=”return”

Lehetőség van a komponensnek paraméterezett tulajdonságokat átadni a `generic` elem segítségével. Példaként tekintsük az alábbi kódot:

```
<operator name=”dtoi” component=”dtoi_ll” cycles=”2” rate=”1”>
      <generic name=”iwidth” type=”out1_width”/>
      <signal name=”clk” type=”clock”/>
      <signal name=”a” type=”in1”/>
      <signal name=”go” type=”request” timing=”late”/>
      <signal name=”result” type=”out1”/>
      <signal name=”pipeEn” type=”pipeEn”/>
</operator>
```

4.5.7 Kombinációs logika

A kombinációs logikát használó komponensek `cycles` paramétere mindig 0. Mindig csak egy bemenő jel és egy kimenő jel van deklarálva. Példaként a lebegőpontos számok negálása:

```
<operator name=”fneg” component=”fneg_ll” cycles=”0”>
     <signal name=”a” type=”in1”/>
     <signal name=”result” type=”out1”/>
</operator>
…
<require file=”VHDL/Xilinx/lib/float_ll.vhd” dst=”lib” type=”hdl”/>
```

A `require` elem által hivatkozott fájlban pedig megtalálhatjuk a következő implementációt:

```
entity fneg_ll is
  port (
    a: in std_ulogic_vector(31 downto 0);
    result: out std_ulogic_vector(31 downto 0));
```

```
end fneg_ll;

architecture fneg_ll_a of fneg_ll is
begin
  result(31) <= a(31) xor '1';
  result(30 downto 0) <= a(30 downto 0);
end fneg_ll_a;
```

A kombinációs logikát használó függvények az operátorokhoz hasonlóan vannak deklarálva, viszont az `operator` címke helyett a `primitive` címkét használjuk.

```
<primitive name="fabs_ll" cycles="0" proc="fabsf" type="component">
    <signal name="a" type="input" carg="0" width="*"/>
    <signal name="result" type="return"/>
</primitive>
```

4.5.8 Regisztrált-aszinkron logika

Készíthetünk olyan függvényeket vagy operátorokat, amelyeknek a késési idejét nem lehet meghatározni fordítási időben. Ilyen esetekben a `cycles` attribútumnak a * jelzést adjuk. Ha tudjuk, hogy minimum N ciklus lesz a késés, használhatjuk az N* jelölést. Regisztrált-aszinkron logika esetén a `request` típusú vezérlőjel `timing` attribútuma kötelezően `early` kell, hogy legyen. Példaként tekintsük az alábbi kódot:

```
<operator name="fdivd" component="fdivd_ll" cycles="1*">
    <signal name="clk" type="clock"/>
    <signal name="a" type="in1"/>
    <signal name="b" type="in2"/>
    <signal name="go" type="request" timing="early"/>
    <signal name="result" type="out1"/>
    <signal name="done" type="acknowledge"/>
</operator>
```

4.5.9 Csővezetékes logika

A csővezetékes logikát a `cycles` és a `rates` attribútumok konstans értékei által különböztetjük meg, melyek a késleltetést és az áteresztő képességet adják meg. Egy példa egy csővezetékes logikát használó operátor deklarálására:

```
<operator name="fmuld" component="fmuld_ll" cycles="4" rate="1">
    <signal name="clk" type="clock"/>
    <signal name="a" type="in1"/>
    <signal name="b" type="in2"/>
    <signal name="go" type="request" timing="late"/>
    <signal name="result" type="out1"/>
    <signal name="pipeEn" type="pipeEn"/>
</operator>
```

A `timing` attribútum értékei a következők lehetnek:

- `early`: minden bemenet regisztrált
- `late`: a bemenetek lehetnek kombinációsak

A fentebb leírt eszközök jó eszköztárat alkotnak az Impulse C alap függvényeinek kibővítésére, ez által könnyebbé téve bizonyos specifikus feladatok megoldását a programozó számára.

5. Matematikai feladatok párhuzamos programozása

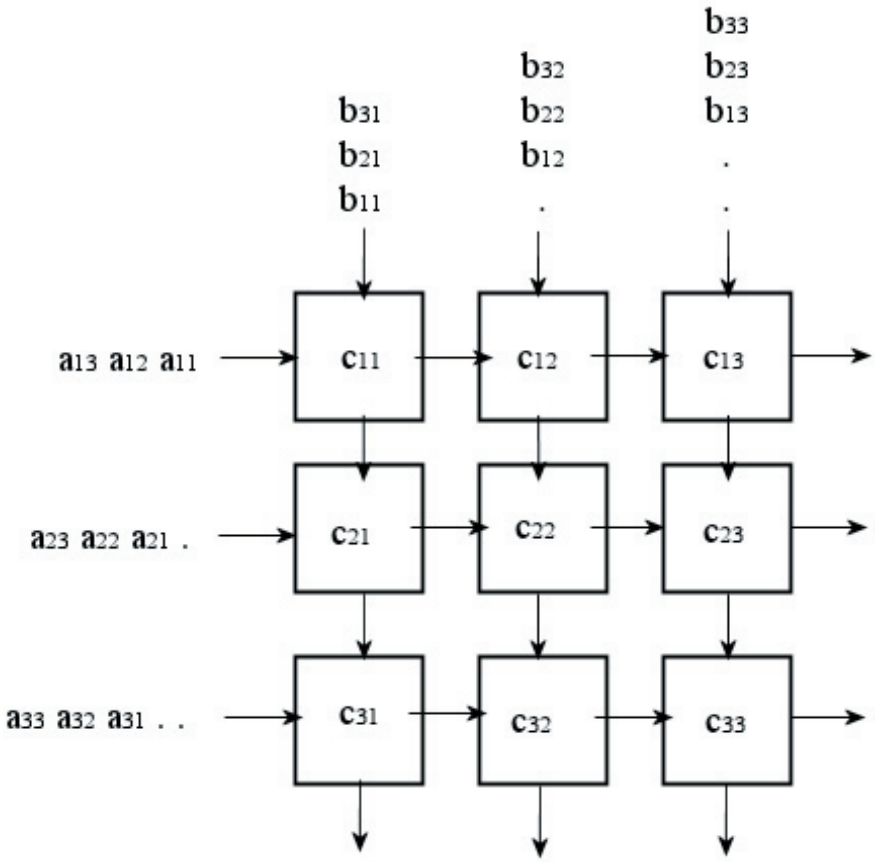
A legtöbb matematikai számítás elvégzésére ismerünk szekvenciális algoritmusokat, melyek jól használhatók mind hagyományos számítógépeken, mind FPGA eszközök esetében. Bizonyos műveletek szekvenciális algoritmussal való kiszámítása költséges, ezért léteznek párhuzamos algoritmusok is, melyek ugyanazt az eredményt állítják elő, viszont struktúrájuknak köszönhetően kevesebb idő alatt képesek elvégezni a számítást.

5.1 Mátrixok szorzása szisztolikus módszerrel

Legyen A ás B két n * n-es mátrix. Végezzük el a C = A*B műveletet a következő képlet segítségével:

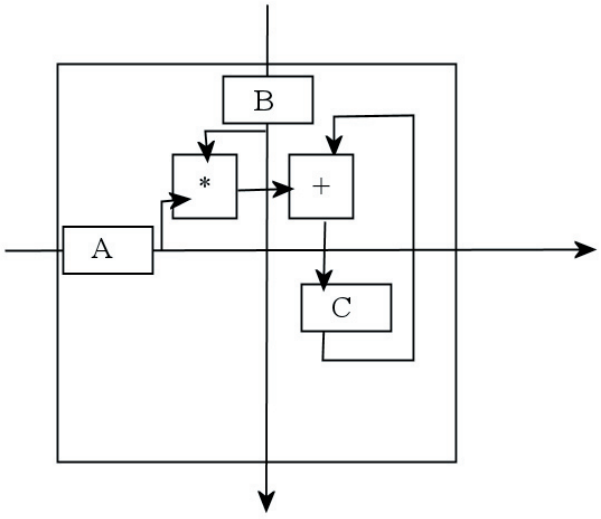
$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}, \quad i, j = 1, \ldots, n.$$

Szekvenciális módszerrel a számítás elvégzése n^3 műveletet igényel, melyet pontosan n^3 időegység alatt végezhetünk el. Ezt az időt jelentősen csökkenthetjük az úgynevezett szisztolikus architektúra segítségével. Az architektúra a csővezeték elvét veszi alapul, és egy olyan modulrendszert alkot, melyben minden egység pontosan ugyanazt a feladatot végzi el: kiszámítja két együttható szorzatát, hozzáadja a kezdetben nullaértékű regiszterhez, és továbbítja az input adatokat az output oldalon. A mátrixszorzás végeredményét (a C mátrix együtthatóit) az egyes modulokban lévő regiszterekből olvashatjuk ki. Minden egyes modulnak két bemenő és két kimenő jele van, ezek jelen esetben a mátrixok együtthatói. A szorzást végző modulok egy sematikus ábrája a következő képen látható:



5.ábra: Szisztolikus mátrixszorzás séma

Az 5. ábrán b_{ij}-vel a B mátrix, a_{ij}-vel az A mátrix, c_{ij}-vel a C mátrix együtthatóit jelöltük.



6. ábra: A műveletet elvégző modul ábrája

Az egyes modulok belső felépítése az n. ábrán látható. A bejövő jeleken érkező együtthatókat B-nek illetve A-nak jelöltük, melyek rendre a B, illetve A mátrixhoz tartoznak. C-vel jelöljük a modulban található regisztert, mely a műveletek részeredményét tárolja, továbbá innen olvashatóak le a végeredményként előálló együtthatók is.

A fentieknek megfelelően létrehozhatunk egy Impulse C alkalmazást, melyben az egyes moduloknak megfeleltethetünk egy-egy folyamatot. Pontosan annyi folyamat van, amennyi eleme a mátrixok szorzataként létrejövő mátrixnak. Minden folyamathoz tartozhat egy regiszter, melyben a részeredményeket tárolja. A folyamatokat adatfolyamok kötik össze, melyek a szorzandó mátrixok együtthatóit közvetítik. A hardveren implementált folyamatok egy szoftveres folyamattól adatfolyamokon kaphatják az együtthatókat, melyek így elvégzik a mátrixok összeszorzását. A végeredményként előálló együtthatókat a regiszterek tartalmából kiolvashatjuk.

6. Összefoglalás

A jelenlegi hardver eszközök, és a rendelkezésre álló fejlesztői környezetek lehetővé teszik, hogy a számítási tudományban kidolgozott párhuzamos algoritmusok egy részét a hétköznapi felhasználás szintjén alkalmazzuk. A dolgozatban megfogalmazott leírás jó kiindulópontot adhat egy Impulse C-ben megírt matematikai függvénykönyvtár kidolgozásához, ami a párhuzamos algoritmusok gyakorlati használhatóságát segítheti elő.

Irodalomjegyzék

1. Iványi Antal (szerk.): Informatikai Algoritmusok I.
   ELTE Eötvös Kiadó, Budapest, 2004
2. David Pellerin, Scott Thibault: Practical FPGA Programming in C
   Prentice Hall PTR, 2005
3. Ralph Bodenner: Using Hardware Libraries with Impulse C
   http://www.impulseaccelerated.com/AppNotes/

Függelék

1. HelloWorld_sw.c

```
// HelloWorld_sw.c: Software processes to be executed on the CPU.
#include “co.h”
#include <stdio.h>
extern co_architecture co_initialize(int param);

void Producer(co_stream output_stream) {
  int32 i;
  static char HelloWorldString[] = “Hello FPGA!”;
  char *p;

  co_stream_open(output_stream, O_WRONLY, CHAR_TYPE);
  p = HelloWorldString;
  while (*p) {
    printf(“Producer writing output_stream with: %c\ n”, *p);
    co_stream_write(output_stream, p, sizeof(char));
    p++;
  }
  co_stream_close(output_stream);
}

void Consumer(co_stream input_stream) {
  char c;

  co_stream_open(input_stream, O_RDONLY, CHAR_TYPE);
  while (co_stream_read(input_stream, &c, sizeof(char)) == co_err_
none ) {
     printf(“Consumer read %c from input stream\ n”, c);
  }
  co_stream_close(input_stream);
}

int main(int argc, char *argv[]) {
    int param = 0;
    co_architecture my_arch;

    printf(“HelloFPGA starting...\ n”);

    my_arch = co_initialize(param);
```

```
    co_execute(my_arch);

    printf(“HelloFPGA complete.\ n”);
    return(0);
}
```

2. HelloWorld_hw.c

```
// HelloWorld_hw.c: Hardware processes and configuration.

#include “co.h”

extern void Consumer(co_stream input_stream);
extern void Producer(co_stream output_stream);

// Hardware process
void DoHello(co_stream input_stream, co_stream output_stream) {
  char c;

  co_stream_open(input_stream, O_RDONLY, CHAR_TYPE);
  co_stream_open(output_stream, O_WRONLY, CHAR_TYPE);
  while (co_stream_read(input_stream, &c, sizeof(char)) ==   co_
err_none ) {
    // Do something with the data stream here
    co_stream_write(output_stream,&c,sizeof(char));
  }
  co_stream_close(input_stream);
  co_stream_close(output_stream);
}

void config_hello(void *arg) {  // Configuration function
  co_stream s1,s2;
  co_process producer, consumer;
  co_process hello;

  s1 = co_stream_create(“Stream1”, CHAR_TYPE, 2);
  s2 = co_stream_create(“Stream2”, CHAR_TYPE, 2);
  producer = co_process_create(“Producer”,
               (co_function) Producer, 1, s1);
  hello = co_process_create(“DoHello”,
               (co_function) DoHello, 2, s1, s2);
  consumer = co_process_create(“Consumer”,
               (co_function) Consumer, 1, s2);
```

```
  co_process_config(hello, co_loc, “PE0”);  // Assign to PE0
}

co_architecture co_initialize(int param) {
  return(co_architecture_create(“HelloArch”,”generic”,
                                config_hello,(void *)param));
}
```

3. HelloFPGA_mem_sw.c

```
// HelloWorld_sw.c: Software processes to be executed on the CPU.
//
// In this version the Producer passes the text via a shared
// memory interface instead of on a stream.
//

#include “co.h”
#include <stdio.h>

extern co_architecture co_initialize(int param);

void Producer(co_memory shared_mem, co_signal ready) {
  int32 count;
  static char HelloWorldString[] = “Hello FPGA!”;

  count = strlen(HelloWorldString);
  co_memory_writeblock(shared_mem, 0, HelloWorldString, count);
  co_signal_post(ready, count);
}

void Consumer(co_stream input_stream) {
  char c;

  co_stream_open(input_stream, O_RDONLY, CHAR_TYPE);
  while (co_stream_read(input_stream, &c, sizeof(char)) == co_err_
none ) {
    printf(“Consumer read %c from input stream\ n”, c);
  }
  co_stream_close(input_stream);
}

int main(int argc, char *argv[]) {
   int param = 0;
```

```
   co_architecture my_arch;

   printf(“HelloFPGA starting...\ n”);

   my_arch = co_initialize(param);
   co_execute(my_arch);

   printf(“HelloFPGA complete.\ n”);
   return(0);
}
```

4. HelloFPGA_mem_hw.c

```
// HelloWorld_hw.c: Hardware processes and configuration.
//

#include “co.h”
#define MAXLEN 128

extern void Producer(co_memory shared_mem, co_signal ready);
extern void Consumer(co_stream output_stream);

// Hardware process: reads from memory, writes to stream
void DoHello(co_memory shared_mem, co_signal ready,
             co_stream output_stream) {
  int32 i, count;
  char buf[MAXLEN];
  char c;

  co_signal_wait(ready, &count);
  co_memory_readblock(shared_mem, 0, buf, count);
  co_stream_open(output_stream, O_WRONLY, CHAR_TYPE);
  for (i=0; i < count; i++) {
       c = buf[i];
       co_stream_write(output_stream,&c,sizeof(char));
  }
  co_stream_close(output_stream);
  }

void config_hello(void *arg) {    // Configuration function
  co_memory memory;
  co_signal ready;
  co_process producer, consumer, hello;
```

```
  co_stream s2;

  memory = co_memory_create("Memory", "mem0", MAXLEN*sizeof(char));
  ready = co_signal_create("Ready");
  s2 = co_stream_create("Stream2", CHAR_TYPE, 2);

  producer = co_process_create("Producer", (co_function) Producer,
2, memory, ready);
  hello = co_process_create("DoHello", (co_function) DoHello, 3,
memory, ready, s2);
  consumer = co_process_create("Consumer", (co_function) Consumer,
1, s2);

  co_process_config(hello, co_loc, "PE0");  // Assign to PE0
}
```

5. entity asyncFun

```
entity asyncFun is
   port (
     signal reset : in std_logic;
     signal clk : in std_logic;
     signal request : in std_logic;
     signal i1 : in std_ulogic_vector(31 downto 0);
     signal r_e_t_u_r_n : out std_ulogic_vector(31 downto 0);
     signal acknowledge : out std_logic);
 end;

 architecture test of asyncFun is
   signal val : std_ulogic_vector(31 downto 0);
   signal count : unsigned(31 downto 0);
   signal done : std_ulogic;
 begin
   process (clk)
   begin
     if clk'event and clk = '1' then  -- rising clock edge
       if request = '1' then
         val <= i1;
       else
         val <= '0' & val(31 downto 1);
       end if;
     end if;
   end process;
```

```
  done <= '1' when val = X"00000000" else '0';

  process (clk)
  begin
    if clk'event and clk = '1' then  -- rising clock edge
      if request = '1' then
        count <= X"00000000";
      elsif done = '0' then
        count <= count + 1;
      end if;
    end if;
  end process;

  r_e_t_u_r_n <= std_ulogic_vector(count);
  acknowledge <= not request and done;
end test;
```

6. entity pipeFun

```
entity pipeFun is
  port (
    signal reset : in std_ulogic;
    signal clk : in std_ulogic;
    signal ce : in std_ulogic;
    signal i1 : in std_ulogic_vector(31 downto 0);
    signal r_e_t_u_r_n : out std_ulogic_vector(31 downto 0));
end;

architecture test of pipeFun is
  signal s1out, s2out : std_ulogic_vector(31 downto 0);
begin

  process (clk)
  begin
    if clk'event and clk = '1' then  -- rising clock edge
      if ce = '1' then
        s1out <= i1(15 downto 0) & i1(31 downto 16);
        s2out <= s1out xor X"ff00ff00";
      end if;
    end if;
  end process;
```

```
  r_e_t_u_r_n <= s2out;
end test;
```