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

Jani Lázár

ALKALMAZÁS-SPECIFIKUS MIKROPROCESSZOROK ARCHC-ALAPÚ LOGIKAI SZINTÉZISE

KONZULENS
Horváth Péter

BUDAPEST, 2013

Tartalomjegyzék

1 Bevezetés

Gordon Moore az 1965-ben megjelent tanulmányában arra a következtetésre jutott, hogy az elkövetkezendő tíz év során az egy lapkára helyezett tranzisztorok száma másfél-két évente duplázódni fog. Megfigyelése irányadóvá vált az ipar számára, így Moore „törvénye” önbeteljesítő jóslat lett, amely nemcsak tíz, hanem az azóta eltelt csaknem ötven év során mindvégig helyesnek bizonyult.

A chipek összetettségének növekedése és a szigorú time-to-market követelmények miatt a tervezés során törekedni kell arra, hogy a megtervezett funkcionális blokkokat más áramkörökben is fel lehessen használni. A *System on Chip* (SoC) áramkörök egyes feladatait ellátó fix funkciójú áramkörök (ASIC, *Application-Specific Integrated Circuit*, alkalmazás-specifikus integrált áramkör) előnyösek az alacsony fogyasztás és gyors működés szempontjából, azonban jelentős módosítások nélkül ezeket nem lehet újra felhasználni.

E problémára egy lehetséges megoldás az ASIP (*Application-Specific Instruction set Processor*, alkalmazás-specifikus processzor) rendszerek használata. Ezek a processzorok az egyedi, egy adott szoftver alkalmazásra, vagy alkalmazási területre optimalizált utasításkészletből adódóan energiahatékonyság és teljesítmény terén kedvezőbb tulajdonságokkal rendelkeznek az általános célú és a DSP (*Digital Signal Processor*, digitális jelfeldolgozó processzor) processzoroknál, programozhatóságuk pedig a széleskörű újrafelhasználhatóságot is biztosítja. Az ASIP-ok tervezését segítő CAD (*Computer-Aided Design*) eszközök elsősorban a tervezendő mikroprocesszoros rendszer szoftver komponenseire (fordító, assembler, utasításkészlet-szimulátor) koncentrálnak, az automatizált hardvergenerálás kevésbé hangsúlyos.

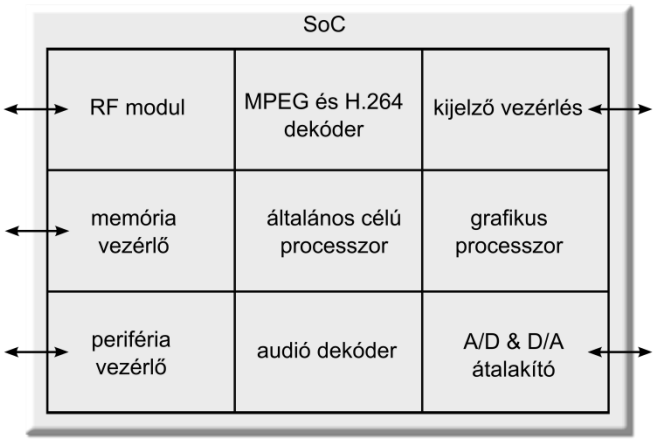
A dolgozat első része az ArchC hardverleíró nyelvvel foglalkozik, amely egy nyílt forrású, SystemC-re épülő osztálykönyvtár. Segítségével alkalmazás-specifikus mikroprocesszorok funkcionális és ciklushelyes szimulátorainak előállítása lehetséges, automatizált logikai szintézis bemeneteként szolgáló hardvermodellek generálására azonban a szükséges algoritmusok hiányában nem alkalmas.

A dolgozat második részében egy ArchC-alapú, hardverszintézisre optimalizált keretrendszer kerül bemutatásra. E keretrendszer egy modelltranszformációs algoritmusból és egy generikus assembler fordítóból áll. Az algoritmus a mikroprocesszor ArchC leírása alapján egy köztes, Verilog nyelvű hardvermodellt állít elő, amely a már rendelkezésre álló logikai szintézis eszközökkel feldolgozható. A funkcionális verifikációhoz szükséges tesztprogramok előállítását egy konfigurálható assembler fordító segíti, amely egyaránt képes az ArchC és az abból generált Verilog leíráshoz illeszkedő programtár-modellek előállítására.

2 Irodalmi áttekintés

2.1 SoC rendszerek megvalósítása

A *System on Chip* (SoC) olyan integrált áramkör, amely egy szilíciumlapkán valósít meg egy teljes funkcionális rendszert, így egy chip tartalmazhat digitális, analóg és RF (*Radio Frequency*) részeket, programozható logikai cellákat, általános célú processzorokat.



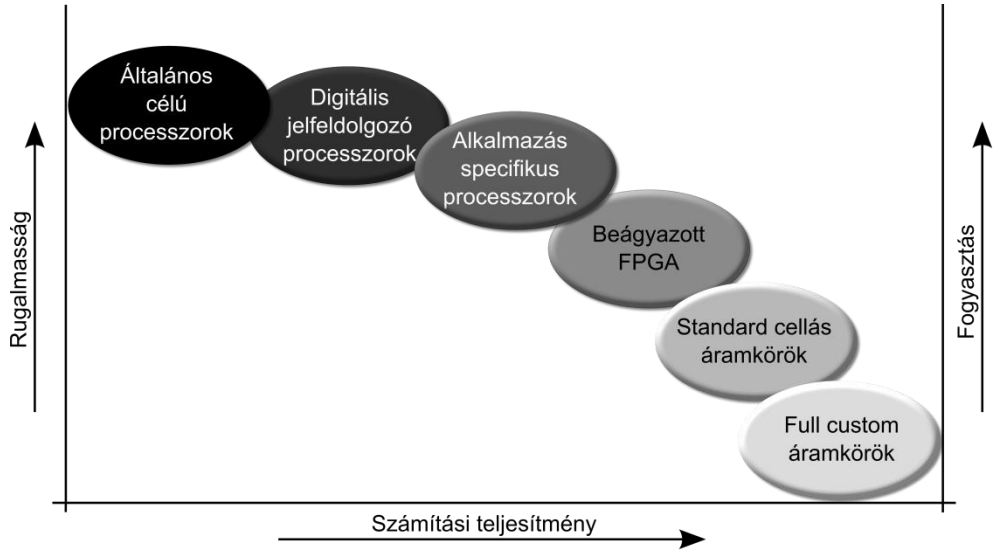
2.1. ábra: System on a Chip általános blokkvázlata

A 2.1. ábra egy általános SoC blokkvázlatot mutat be. Egy ilyen rendszer sok funkcionális modult tartalmaz, amelyek buszrendszeren keresztül kapcsolódnak egymáshoz. A fix funkcionalitással (például MPEG vagy H.264 dekóder, periféria vezérlő stb.) rendelkező blokkokat olyan integrált áramkörökkel valósítják meg, amelyeket az alkalmazás igényei szerint terveztek. Ezeket az integrált áramköröket alkalmazás-specifikus integrált áramköröknek nevezik (ASIC, *Application Specific Integrated Circuit*) és a számítási teljesítmény, illetve energiahatékonyság terén nagyon kedvező tulajdonságokkal rendelkeznek. Az adatfeldolgozó rendszerek ASIC áramkörei jellemzően adatútból és vezérlő egységből állnak [1]. Az adatút a funkció elvégzéséhez szükséges aritmetikai és adattároló erőforrásokat és azok összeköttetéseit tartalmazza, a vezérlő egység pedig általában egy állapotgép, amely az adatút megfelelő működtetéséhez szükséges időzített vezérlőjelek előállítását végzi.

Az ASIC áramkörök modellezésének elsődleges eszközei – a rendszerszintű modellezést követő tervezési szakaszban – a hardverleíró nyelvek (HDL, *Hardware Description Language*). Bár a hardverleíró nyelvek több különböző elvonatkoztatási szinten képesek hatékonyan leírni az áramkörök viselkedését és szerkezetét, a nagy bonyolultságú rendszerek fejlesztése rendkívül sok munkaórát igényel az egyes tervezési fázisok közötti információfüggőségből adódó iterációs lépések és a verifikációs nehézségek miatt. A nagy költségekből adódóan a rugalmas újrafelhasználhatóság lehetősége egyre fontosabb szempont, amelyet az összetett digitális rendszerek tervezése során figyelembe kell venni. Egy fix funkcionalitású ASIC esetén a kedvező fogyasztás és a nagy számítási kapacitás ára éppen a

rugalmasság. Ezekben az áramkörökben sem a vezérlési feladatot ellátó állapotgép, sem az adatmanipulációt végző erőforrások nem változtathatók meg a gyártás után. Egy-egy új funkció bevezetése, egy új kódolási algoritmus vagy egy új kommunikációs protokoll implementálása csak egy új ASIC időigényes kifejlesztésével és költséges legyártásával lehetséges.

Az adatfeldolgozó rendszerek rugalmassága kétféle úton biztosítható. Az egyik lehetőség az újrakonfigurálható technológia (FPGA, CPLD), a másik az általános célú, tárolt programot végrehajtó hardver (mikroprocesszor). A SoC rendszerek esetén a fix funkciójú áramkörök, a tárolt programú gépek és az újrakonfigurálható logikai cellák egy lapkára való integrálásának lehetősége széleskörű optimalizációt tesz lehetővé. A 2.2. ábra az adatfeldolgozó rendszerek különböző implementációs lehetőségeit hasonlítja össze számítási kapacitás, rugalmasság és fogyasztás alapján.



2.2. ábra: Különböző architektúrák fogyasztás, sebesség és rugalmassági jellemzői [1]

Az általános célú és a jelfeldolgozó processzorok ugyan különböző céloknak megfelelően programozhatóak, azonban az energiahatékonyságuk és a számítási teljesítményük nagyon elmarad az standard cellás és full-custom ASIC áramköröktől. A SoC-ba ágyazott FPGA megoldás nagy rugalmasságot kínál, számítási teljesítménye azonban viszonylag alacsony, fogyasztása pedig jelentős.

A processzorok rugalmasságát és az alkalmazás-orientált integrált áramkörök energiahatékonyságát és számítási teljesítményét ötvözik az ASIP (*Application Specific Instruction-set Processor*, alkalmazás-specifikus utasításkészletű processzor) áramkörök [1].

2.2 Alkalmazás-specifikus mikroprocesszorok

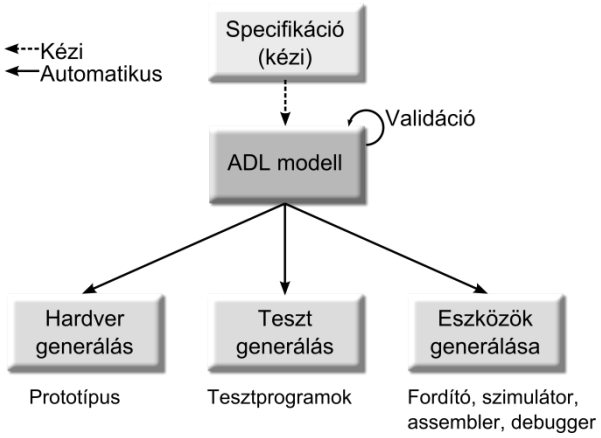
Az ASIP olyan tárolt programú mikroprocesszort jelent, amelynek utasításkészlete és mikroarchitektúrája egy konkrét algoritmusra, vagy egy problémakörre optimalizált. Ilyen terület lehet például a digitális jelfeldolgozás, ahol a beérkező digitalizált jelet kell manipulálni valós időben. Két eltérő szemlélet létezik ASIP-ek tervezési módszerében. Az egyik megközelítésben csak az algoritmus vagy

alkalmazás által igényelt utasításokat implementálják, jelentősen lecsökkentve a szükséges erőforrások mennyiségét, a másik esetben különleges gépi utasítások terveznek, és ezekhez egyedi hardver egységeket is illesztenek a mikroarchitektúrába.

Az alkalmazás-specifikus processzorokat a legkülönfélébb területeken használnak, de az előnyös tulajdonságaik (fogyasztás, teljesítmény, rugalmasság) miatt elsősorban a rendkívül gyorsan fejlődő szegmensekben találhatók meg. Ilyen terület például a mozgókép feldolgozás, ahol kódolási és dekódolási funkciókat láthatnak el [2], elmozdulás becslést végeznek [3], valamint a képfeldolgozás, ahol például Retinex szűrést hajtanak végre [4]. Speciális jelfeldolgozási feladatokban egy ASIP hatékonyabb lehet egy DSP-nél. Előbbi felhasználásával akár valós idejű szinguláris érték szerinti és QR felbontás is megvalósítható [5]. A vezeték nélküli technológiák rendkívüli mértékben fejlődnek, egyre több szabvány kínál mind nagyobb adatátviteli sebességet. Az alkalmazás-specifikus processzorok hozzájárulnak ehhez az evolúcióhoz, például demapper [6], vagy hibajavítási funkciók [7] [8] ellátásával. ASIP áramköröket használhatnak reaktív rendszerekben léptetőmotorok vezérléséhez [9], multi-rail DC-DC konverterek vezérléséhez [10], vagy titkosításhoz is [11]. Változatos felhasználhatóságukat a tervezéshez szükséges erőfeszítések csökkenése teszi lehetővé. Az ASIP-ok fejlesztését ADL (*Architecture Description Language*, architektúra leíró nyelv) nyelveket támogató CAD (*Computer Aided Design*) eszközökkel lehet gyorsítani.

2.3 Architektúra leíró nyelvek

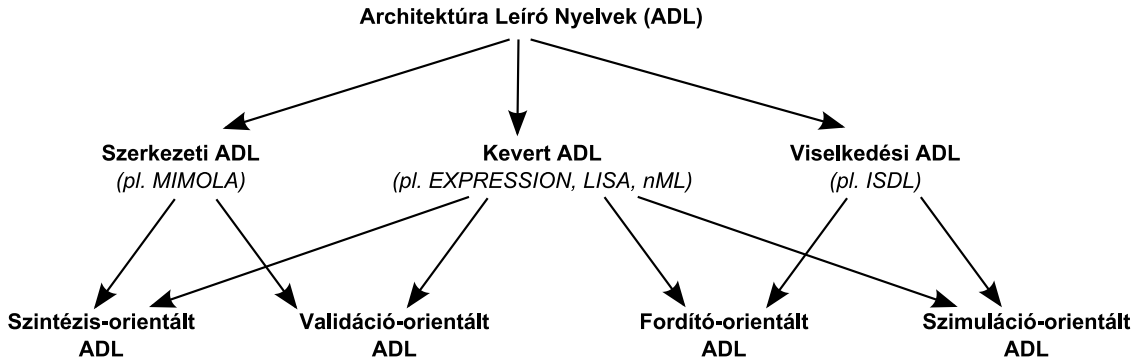
A hagyományos tervezési eljárásban a tervezők a processzor viselkedését egy magas szintű programnyelven (például C, C++) specifikálják, majd ezt referenciaként felhasználva tervezik meg az alacsonyabb absztrakciós szintű modellt hardver leíró nyelven (VHDL, Verilog), valamint a processzorhoz szükséges szoftvereszközöket. Végül az áramkör logikai kapu szintű leírása az RTL modellből készül automatizált logikai szintézissel. Ezzel a módszerrel a fejlesztés hosszadalmas és sok hibalehetőséget rejt magában.



2.3. ábra: Architektúra leíró nyelvek funkciói [12]

Az ADL-ek olyan speciális célú modellező nyelvek, amelyek nyelvi eszközökkel támogatják az alkalmazás-specifikus processzormodellek tervezését. A magas absztrakciós szinten leírt utasításkészletet már az előtt lehet tesztelni, mielőtt a processzor architektúrája ki lenne dolgozva. Ez lehetővé teszi szoftvereszközök automatikus generálását. Bizonyos architektúra leíró nyelvek alkalmasak a rendszerszintű modellezésen és a szoftvergeneráláson kívül hardver szintézisére is. Ezeknek a nyelveknek a kívánt architektúra strukturális leírását is tartalmazniuk kell, ezáltal az RTL leírást gépi úton lehet generálni (2.3. ábra).

Az architektúra leíró nyelvek csoportosíthatóak tartalom és felhasználási terület szempontjából (2.4. ábra).



2.4. ábra: Az architektúra leíró nyelvek csoportosítása [12]

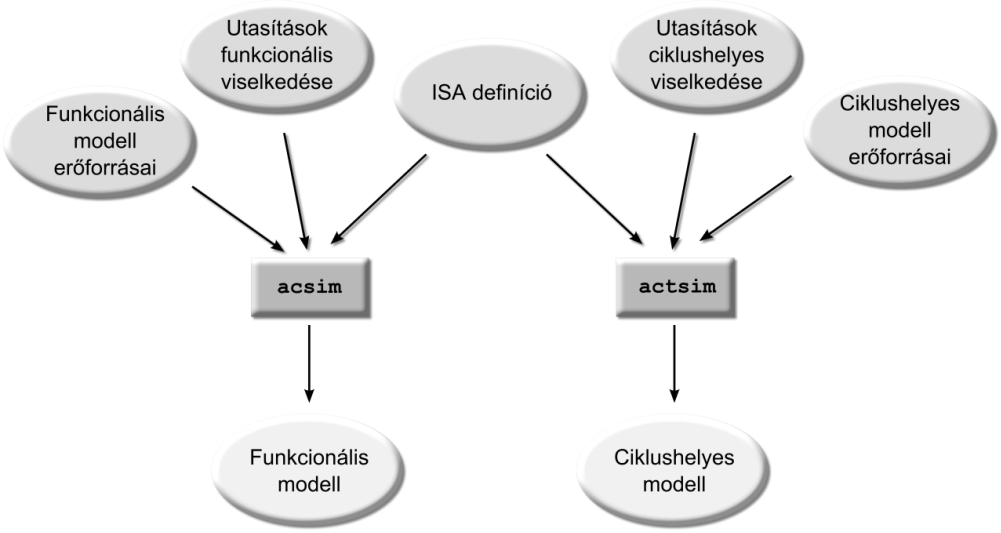
Az ADL modell tartalmazhatja a processzor szerkezeti vagy viselkedési leírását, esetleg mindkettőt. Ez alapján az információ alapján a különböző architektúra leíró nyelvek más célokra alkalmazhatóak. A szerkezeti ADL-ek alapján szintetizálni lehet a processzorok prototípusát, ezáltal fogyasztásra és területre vonatkozó információk nyerhetőek ki belőle, viselkedési ADL felhasználásával pedig az utasításkészletet lehet szimulálni, illetve fordítót generálni. A kevert ADL-ek szerkezeti és viselkedési információkat is tartalmaznak, így az összes feladatot képesek ellátni.

2.4 Az ArchC architektúra leíró nyelv

Az ArchC architektúra leíró nyelv a SystemC alapjaira épülő nyílt forrású C++ osztálykönyvtár. A nyelv támogatja funkcionális és ciklushelyes szimulátorok, valamint assemblerek automatikus generálását. Ez lehetőséget teremt, hogy a tervező funkcionális szintű processzormodellt hozzon létre, majd azt a tervet finomítva ciklushelyes modellt készíthessen. A modellezés ilyen felbontása lehetővé teszi, hogy az utasításkészletet alapos tesztelésnek lehessen alávetni anélkül, hogy túlzottan sok architektúrális részlettel kelljen foglalkozni. A processzor funkcionális modelljéhez lehet készíteni szoftver eszközöket (assembler, fordító stb.), így tesztprogramokat már ebben a fázisban is lehet készíteni. A ciklushelyes modellt érdemes egy tesztelt funkcionális modell alapján megtervezni, ezáltal a hibalehetőségek száma redukálható.

Funkcionális szimulátort az ArchC csomag `acsim` és `accsim` eszközével lehet generálni. Előbbi SystemC-re interpretált, utóbbi közvetlen lefordított szimulátort generál.

Ciklushelyes modellt az `actsim` eszközzel lehet létrehozni. Ennek a modellnek az absztrakciós szintje nem egyezik meg a HDL-ekben (*Hardware Description Language*, hardver leíró nyelv) megszokott RTL (*Register-Transfer Level*, regiszter-átviteli szint) leírással, mert több magas szintű ArchC nyelvi elem is megtalálható benne, amelyek közvetlen implementációjára a tervezőnek nincs befolyása (pl. utasításfelhozás és dekódolás, pipeline megvalósítás, megszakításkezelés).



2.5. ábra: Az ArchC alapú tervezéshez szükséges források és felhasználható eszközök

A 2.5. ábra mutatja be az ArchC-alapú tervezéshez szükséges leírásokat és a használható szoftvereszközöket. A nyelv két leírásra támaszkodik a modellezés során: az egyikben az architektúra erőforrásai, a másikban az utasításkészlet architektúra (ISA, *Instruction Set Architecture*) kifejtése szerepel, így az ArchC kevert architektúra leíró nyelvnek mondható.

2.4.1 Funkcionális modellezés

```
AC_ISA(proc){
  ac_format type_L = "%op:2 %dat:6";
  ac_format type_J = "%op:2 %adr:6";
  ac_instr <type_L> and, or, inv;
  ac_instr <type_J> jmp;
  ISA_CTOR(proc){
    inv.set_decoder(op = 0x0);
    and.set_decoder(op = 0x1);
    or.set_decoder(op =0x2);
    jmp.set_decoder(op = 0x3);
  };
};
```

2.6. ábra: Utasításkészlet deklarálása ArchC-ben

Funkcionális modellezés során az utasításkészlet megvalósítása a cél (2.6. ábra), az architektúra erőforrásai nincsenek kidolgozva részletesen (nincsenek időzítések, pipeline stb.). Az ArchC szimulátoraiban az utasítások egy SystemC szimulációs ciklus

alatt hajtódnak végre. A fejlesztés következő lépéseként a modellt finomítani lehet architekturális részletek hozzáadásával (ciklushelyes modellezés).

```
AC_ARCH(proc){
  ac_wordsize 8;
  ac_mem PMEM:64;
  ac_reg <8> acc;
  ARCH_CTOR(proc){
    ac_isa("proc_isa.ac");
    set_endian("big");
  };
};
```

2.7. ábra: Architektúra erőforrásainak leírása ArchC-ben

ArchC-ben a funkcionális modellt egyszerűen létre lehet hozni. A tervezőnek meg kell határoznia az architektúra erőforrásait (2.7. ábra, ezek nélkül az utasításkészlet nem használható) és az utasításkészletet, majd generáltatni kell ArchC-vel a modellfájlokat.

A modellfájlok generálása mellett az ArchC létrehoz egy sablont is, amely tartalmazza az utasítások működését modellező függvények vázát (2.8. ábra).

```
...
//!Behavior executed before simulation begins.
void ac_behavior( begin ){
  printf("@@@ begin simulation @@@\n");
};

//Behavior executed after simulation ends.
void ac_behavior( end ){
  printf("@@@ end simulation @@@\n");
};

//! Generic instruction behavior method.
void ac_behavior( instruction ){};

//! Instruction Format behavior methods.
void ac_behavior( type_L ){}
void ac_behavior( type_J ){}

//! Instruction and behavior method.
void ac_behavior( and ){
  acc = acc & dat & 0x3F;
  printf("PC=%#x OPC=%#x ACC=%#x",(int)ac_pc,(int)op,(int)acc);
}
...
```

2.8. ábra: Utasítások viselkedési leírása ArchC-ben

Az utasítások viselkedését három függvény határozza meg. Egy általános függvény, amely minden utasítás esetén lefut (2.8. ábra, argumentuma `instruction`), az utasítás formátumához tartozó függvény (2.8. ábra, `type_L`), valamint az utasítás egyedi függvénye (2.8. ábra, `and`). A tervezőnek elég az utasítások viselkedésének megfelelően definiálni a függvényeket, majd lefordítani a projektet. A szimulációt a fordítást követően kapott futtatható állomány segítségével lehet elvégezni.

2.4.2 Ciklushelyes modellezés

Ciklushelyes modellezés során olyan modell létrehozása a cél, amelyben az architektúra összes erőforrása megtalálható, és ezeknek működési sorrendje megfelel a valóságnak (ami párhuzamos működésű, az párhuzamosan van modellezve, a szekvenciális működés pedig egymás után hajtódik végre a szimulációban is). A ciklushelyes modellt érdemes a funkcionális modellből kiindulva megvalósítani, így az utasításkészletet már előre le lehetett tesztelni, és kész tesztalkalmazások is rendelkezésre állnak (ezek azonban nem feltétlen működnek ugyanúgy egy csővezetékezett processzoron és egy funkcionális modellen).

```
AC_ARCH(proc){
  ac_wordsize 8;
  ac_mem PMEM:64;
  ac_reg <8> acc;
  ac_pipe pipe = {IF, ID, EX};
  ARCH_CTOR(proc){
    ac_isa("proc_isa.ac");
    set_endian("big");
  };
};
```

2.9. ábra: Ciklushelyes processzormodell architektúra leírása ArchC-ben

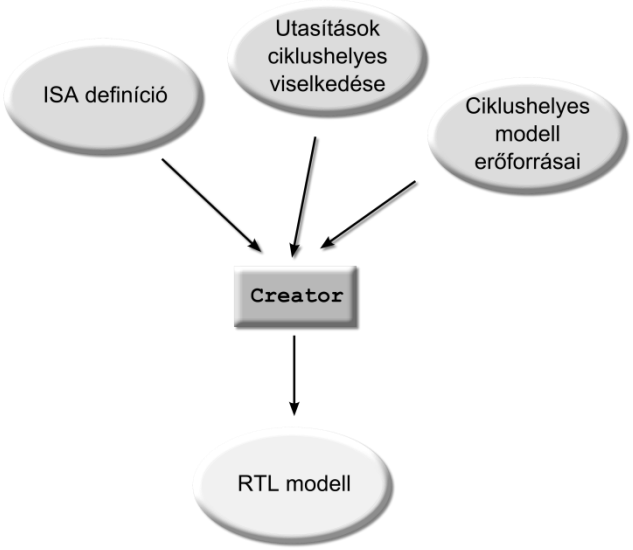
ArchC-ben a ciklushelyes modellt a funkcionális modellhez hasonlóan lehet létrehozni. A tervezőnek meg kell határoznia az architektúra erőforrásait (2.9. ábra), és létre kell hoznia az utasításkészletet definiáló fájlt. Amennyiben egy funkcionális modellt finomít a tervező ciklushelyes modellre, akkor elég kiegészíteni az architektúra erőforrásait a csővezetékkel, és az ahhoz tartozó regiszterekkel, az utasításkészlet leírását módosítás nélkül át lehet venni. A ciklushelyes modell és szimulátor generálásához az `actsim` eszköz használható. Ez az eszköz az `acsim`-hez hasonlóan generálja a processzormodellt és az utasítások viselkedésének sablonját. Amennyiben a tervező deklarált egy csővezetéket az architektúra erőforrásai között, akkor az ArchC az utasítások viselkedését modellező függvényekbe egy `switch-case` szerkezetet is megvalósít, amely az egyes csővezeték fokozatokat modellezi (2.10. ábra). A fokozatok közötti utasítás átadást az ArchC automatikusan kezeli, tehát az utasítás, amelyik egy ciklusban az 1. fokozatban volt, a következő ciklusban átkerül a következő fokozatba. A csővezeték kezelésére (fokozat megállítása, kiűrítés) léteznek beépített függvények.

```
...
//!Behavior executed before simulation begins.
void ac_behavior( begin ){};

//Behavior executed after simulation ends.
void ac_behavior( end ){};

//! Generic instruction behavior method.
void ac_behavior( instruction ){};

//! Instruction Format behavior methods.
void ac_behavior( type_L ){...}
void ac_behavior( type_J ){...}

//! Instruction and behavior method.
void ac_behavior( and ){
switch(stage){
  case id_pipe_IF:
    break;
  case id_pipe_ID:
    break;
  case id_pipe_EX:
    acc = acc & dat & 0x3F;
    printf("PC=%#x OPC=%#x ACC=%#x",(int)ac_pc,(int)op,(int)acc);
    break;
  default:
    break;
}
...
```

2.10. ábra: Utasításkészlet viselkedésének leírása ArchC-ben ciklushelyes modellezés esetén

3 ArchC-alapú hardverszintézis - `Creator`

Az ArchC nyelv és a hozzá tartozó szoftver eszközök hatékony funkcionális modellezést tesznek lehetővé, automatizált logikai szintézisre azonban nem alkalmasak. A dolgozat további részeiben bemutatásra kerül egy új, C++ nyelven kifejlesztett szoftver eszköz, a `Creator`, amely képes a ciklushelyes ArchC modellt regiszter-transzfer szintű Verilog leírássá transzformálni. E köztes reprezentáció a már rendelkezésre álló logikai szintézis eszközökkel feldolgozható, az áramkör kapuszintű modellje előállítható.



3.1. ábra: A Creator algoritmus be- és kimenete

A `Creator` ugyanazokat a leírásokat használja (3.1. ábra), mint az ciklushelyes modellt generáló `actsim` eszköz (2.5. ábra), ezáltal az RTL modell generálása nem igényel külön erőfeszítéseket a tervező részéről.

3.1 Az ArchC nyelvi elemek transzformációja

A `Creator` algoritmus a forrásfájlok feldolgozása után az ArchC modellt RTL szintű Verilog leírássá transzformálja. Ahhoz, hogy ez a lépés egyértelmű legyen, egy megfeleltetési szabályt kellett definiálnom. A 3.1. táblázat összefoglalja az ArchC és Verilog elemek kapcsolatát.

Az ArchC modellben deklarált regiszterek, regisztertömbök és memóriák a Verilog modellben `reg` erőforrásként példányosodnak, a csővezeték pedig fokozatonként egy `always` blokkot kap. A fokozatok kiürítését ArchC-ben egy függvény végzi. Ezt a funkciót Verilogban egy engedélyező jel felhasználásával valósítottam meg. Minden fokozatnak van egy ilyen változója, ami tiltani tudja az adott fokozat működését.

Az utasítás felhozatalát és továbbítását egy shiftregiszter végzi. A shiftregiszter szélessége a szóhosszal, a mélysége pedig a fokozatok számával egyezik meg.

A programszámláló az egyetlen olyan erőforrás, amelyet több fokozatból is lehet módosítani. Ezért minden fokozatban van egy címregiszter, ami a fokozat által írt címet tárolja. Az utasításmutató prioritásos alapon választja ki az egyik címregisztert (működés szempontjából későbbi fokozatnak nagyobb a prioritása).

ArchC-ben az értékvizsgálat, értékadás és a feltételes szerkezetek hasonlóak a Veriloghoz. Az értékadás nem blokkoló jellegű, az értékvizsgálat és a feltételes szerkezetek szintaxisa hasonló, így a Verilog modell kialakítása konverzióval megoldható.

ArchC modell	Verilog nyelvi elem
regiszter, regiszterbank	regiszter, regiszter vektor (`reg` erőforrás)
memória	regiszter vektor
csővezeték	külön `always` blokk az egyes csővezeték fokozatok számára
csővezeték fokozatainak kiürítését végző függvény	dedikált vezérlőjel az egyes fokozatok számára
utasítás felhozatal és az utasítás propagálása a csővezetékben	shift regiszter, amely mindig a fokozatban pillanatnyilag végrehajtandó utasítás kódját tárolja
programszámláló	dedikált címregiszter minden fokozatban, az utasításmutató értéke prioritásos alapon dől el
értékadás	nem blokkoló értékadás
feltételes szerkezet, értékvizsgálat	feltételes szerkezet, értékvizsgálat

3.1. táblázat: Az ArchC nyelvi elemek Verilog nyelvű reprezentációja

A 3.2. ábra egy egyszerű csővezeték példáján keresztül mutatja be a `Creator` által megvalósított leképezést. Az ArchC modellben a csővezeték az utasítások viselkedési leírásában szerepel, a Verilog modellben viszont minden fokozat működése külön `always` blokkban van leírva. Az algoritmusnak fel kell ismernie az ArchC modellben, hogy az utasítások viselkedési leírásán belül melyik fokozatban milyen tevékenységet végez a processzor, és ezeket a tevékenységeket a Verilog modellben a megfelelő `always` blokkban az utasításdekóder által meghatározott helyre kell illesztenie. A Verilog kódban láthatók az egyes fokozatok engedélyező jelei (`pipe_c`). Ezeknek az értéke annak függvényében változik, hogy a csővezeték egyes fokozatai milyen utasítást hajtanak végre. Például amennyiben az `EX` fokozatban jmp utasítás van, akkor az `IF` és `ID` fokozatok tiltásra kerülnek. A `pipe_c` változó értékét kombinációs logika állítja elő.

A `rst` jel hatására a fokozatban állított összes változónak a kezdeti értéket kell adni. Az ábrán látható ArchC példamodellben a szimuláció előtt lefutó függvény (`ac_behavior(begin)`) nem tartalmaz értékadást, így minden változónak nulla értéket kell kapnia.

```
void ac_behavior(begin){}

void ac_behavior(end){}

void ac_behavior(instruction){
 switch (stage){
  case id_pipe_IF:
   IF_ID.op = op;
   IF_ID.adr = adr;
   break;
  case id_pipe_ID:
   break;
  case id_pipe_EX:
   break;
  default:
   break;
  }
}

void ac_behavior(type_1byte){
 switch (stage){
  case id_pipe_IF:
   ac_pc = ac_pc + 1;
   break;
  case id_pipe_ID:
   ID_EX.dat=DMEM.read_byte(IF_ID.adr);
   ID_EX.adr=IF_ID.adr;
   break;
  case id_pipe_EX:
   break;
  default:
   break;
 }
}

void ac_behavior(jmp){
 switch (stage){
  case id_pipe_IF:
   break;
  case id_pipe_ID:
   break;
  case id_pipe_EX:
   pipe_IF.flush();
   pipe_ID.flush();
   ac_pc = ID_EX.adr;
   break;
  default:
   break;
 }
}
```

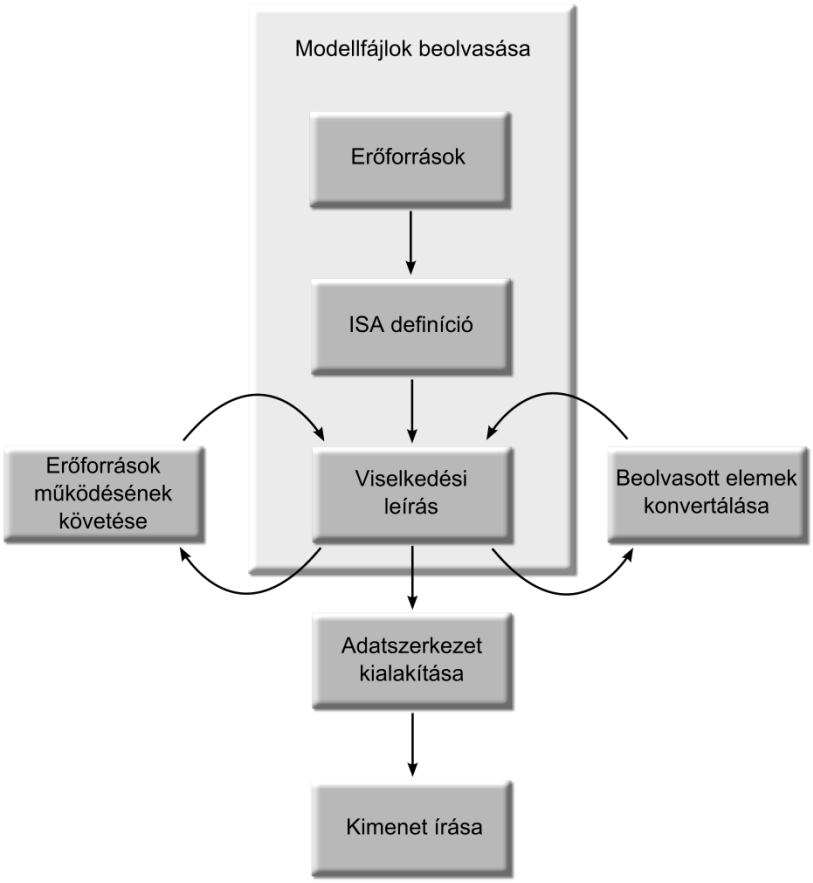
```
always@ (posedge clk) begin //IF
 if(rst == 0) begin
   IP_IF <= 0;
   IF_ID_adr <= 0;
   IF_ID_op <= 0;
   end
 else begin
  if (pipe_c[0] == 1) begin
   case(inst_word_IF[15:14])
    0:
    1:
    2:
    3: begin
     IP_IF = IP + 1;
     IF_ID_op <= inst_word_IF[15:14];
     IF_ID_adr = inst_word_IF[13:8];
     end
   end
  end
 end

always@ (posedge clk) begin //ID
 if(rst == 0) begin
  IP_ID <= 0;
  ID_EX_adr <= 0;
  ID_EX_dat <= 0;
  end
 else begin
  if (pipe_c[1] == 1) begin
   case(inst_word_ID[15:14])
    0:
    1:
    2:
    3: begin
     ID_EX_dat <= DMEM[IF_ID_adr];
     ID_EX_adr <= IF_ID_adr;
     end
   end
  end
 end

always@ (posedge clk) begin //EX
 if(rst == 0) begin
  IP_EX <= 0;
  acc <= 0;
  end
 else begin
  if (pipe_c[2] == 1) begin
   case(inst_word_EX[15:14])
    0:
    1:
    2:
    3: begin
     IP_EX <= ID_EX_adr;
     end
   end
  end
 end
```

3.2. ábra: Egyszerű csővezeték leírása ArchC és Verilog modellben

3.2 A `Creator` algoritmus

Az algoritmust a 3.3. ábra mutatja be. A `Creator` első lépésben beolvassa az erőforrásokat, és az utasításkészletet deklaráló fájlt. Ezt követően megkezdi az utasítások viselkedését leíró fájl beolvasását, és ezzel párhuzamosan a beolvasott konstansokat, feltételes szerkezeteket (`if-else`, `switch-case`), értékvizsgálatokat és értékadásokat a Verilog nyelv szintaxisának megfelelő formátumba alakítja.

Ebben a műveleti fázisban az algoritmus az architektúra erőforrásainak működését is követi. A regisztereket és memóriákat az adat konzisztencia megőrzése érdekében nem javasolt több csővezeték fokozatban is írni. A `Creator` figyelmeztető üzenetet ad, ha ez a feltétel nem teljesül. Ez alól a feltétel alól az utasításmutató kivétel, ugyanis ennek az erőforrásnak az ugrások és feltételes elágazások miatt előfordulhat, hogy több fokozatban is állítani kell az értékét.



3.3. ábra: A `Creator` algoritmus működése

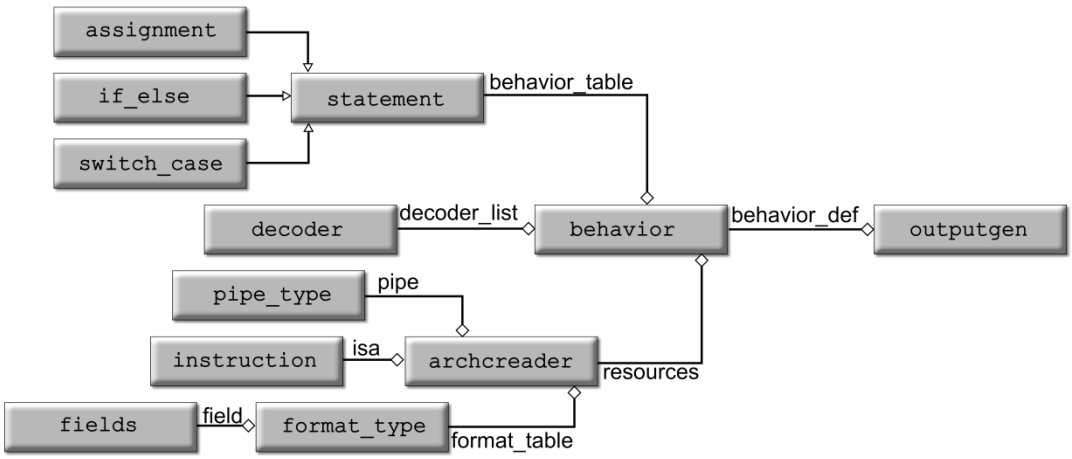
Az ArchC nyelv a csővezeték kezelését nagymértékben megkönnyíti a tervező számára. Az utasítások ciklusonként automatikusan lépnek a következő fokozatba, és a memóriából egy új utasítás kerül az első fokozatba. Előfordulhat, hogy be kell avatkozni ebbe a folyamatba, mert ki kell üríteni a csővezeték bizonyos fokozatait. Ez az eset áll elő feltételes elágazásoknál, ha elágazásbecslést alkalmaz a processzor. Hibás becslés esetén a csővezetékbe nem megfelelő utasítások kerülnek, ennek következtében ki kell üríteni a csővezeték bizonyos fokozatait. ArchC-ben ezt egy függvényhívással

lehet megtenni. A `Creator` algoritmus figyeli, hogy a processzormodellben ezek a függvények milyen feltételek mellett hívódnak meg, és ezeket az információkat elmenti. Az RTL Verilog modellben az egyes fokozatok működéséhez engedélyező jelre van szükség, ezeket a jeleket az elmentett feltételek állítják be.

Az utasítások viselkedését leíró fájl alapján felépített adatszerkezetet át kell alakítani a Verilog modell igényeinek megfelelően. Az ArchC modellben az egyes utasítások viselkedését három függvény határozza meg (generikus, utasításformátumnak megfelelő, és specifikus függvény, bővebb információ 2.4.1 fejezetben). Az RTL modellben nem ilyen hierarchikus az utasítások végrehajtása, így az egyes utasításhoz tartozó függvényeket a `Creator` algoritmusnak össze kell vonnia.

Ezen túl a Verilog modellben figyelni kell a tároló elemek inicializálására is. ArchC-ben a változók automatikusan nulla értéket kapnak, de lehetőség van a szimuláció előtt értékadásokra is (2.8. ábra, 2.9. ábra). A `Creator` algoritmus eltárolja ezeket, és RTL modellben ezeket az értékeket adja a változóknak reset esetén. A megfelelő adatszerkezet kialakítása után a `Creator` előállítja az RTL modellt.

A 3.4. ábra mutatja be a `Creator` algoritmust alkotó osztályok kapcsolatát. Az architektúra erőforrások és az utasításkészlet beolvasásáért az `archcreader` osztály a felelős, az adatokat a `fields`, `format_type`, `instruction` és `pipe_type` típusú változók tárolják. A `decoder` típusú `decoder_list` változó menti el az utasításkészlet alapján inicializált utasításdekódert. Az utasítások viselkedési leírásának beolvasásáért a `behavior` osztály felel. A viselkedési információkat a `behavior_table` tárolja, amely a `statement` osztály alapján inicializált STL (*Standard Template Library*) vector típusú változó. A `statement` ősosztály virtuális, az adatokat a belőle származó `assignment`, `if_else` és `switch_case` osztályok tárolják, ezáltal az adatszerkezet egy heterogén kollekciót alkot. A viselkedési leírás beolvasása után az adatszerkezetet módosítani kell az ArchC és a Verilog modellek eltérő felépítése miatt. Ezt követően az `outputgen` osztály függvényei generálják a kimenetet.



3.4. ábra: A `Creator` osztályainak kapcsolata

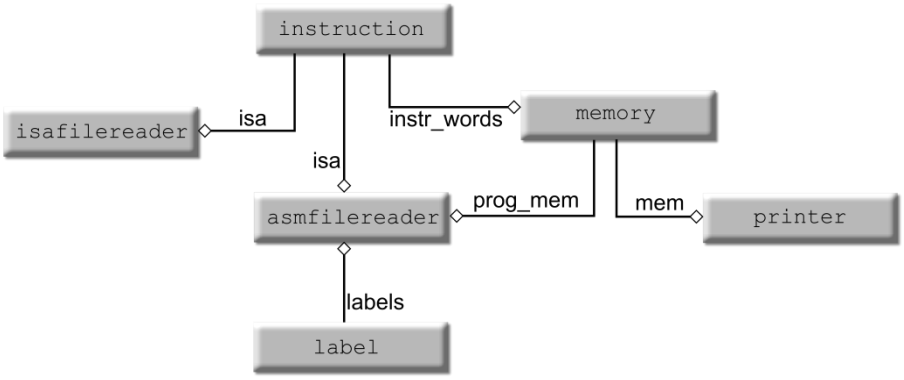
4 Assembler

Az ArchC nyelv támogatja assemblerek automatikus generálását, de ezek az eszközök nem használhatóak Verilog nyelvű modellekhez. Ebben a fejezetben bemutatásra kerül a `Creator` modelltranszformáló eszköz mellé készült generikus assembler fordító, amely lehetővé teszi tesztprogramok készítését ArchC és Verilog nyelvű modellekhez is.

4.1 Az assemblerrel szemben támasztott követelmények

Az tervezés során nagy figyelmet fordítottam arra, hogy az assembler tetszőleges utasításkészlethez tudjon adaptálódni. Ezen kívül szerettem volna elérni, hogy ne csak ArchC, hanem Verilog nyelven tervezett processzorokhoz is használható legyen az assembler fordító (emiatt az nem támaszkodhat az ArchC utasításkészlet definíciójára).

További követelmény volt az assemblerrel szemben különböző, az assembly program előállítását megkönnyítő funkciók támogatása. Ilyenek a különböző címkekezelési módok, a módosítható pozíciószámláló (ezzel meg lehet határozni tetszőleges kódrészlet pozícióját a memóriában), valamint konstansok használatának lehetősége.



4.1. ábra: Az assembler osztályainak kapcsolata

Az assemblert C++ nyelven implementáltam, az osztályok kapcsolatát a 4.1. ábra mutatja be. Az `instruction` osztály egy utasításra vonatkozó adatok tárolására szolgál (szóhossz, név, fix és operandusmezők adatai). Az `isafilereader` osztály végzi el az utasításkészletet specifikáló fájl beolvasását. Az utasításkészletet egy STL vector típusú változó tárolja el, amelynek sablonparamétere az `instruction` osztály. A `label` osztály egy címkéhez rendelhető információk tárolására való (adott címkének mennyi az értéke, valamint melyik utasításokban hivatkoztak rá). Az `asmfilereader` osztály olvassa be az assembly fájlt, majd egy `memory` típusú objektumot hoz létre. A beolvasásnak két menete van, mivel egy menetben nem biztos, hogy fel lehet oldani az összes címke értékét. Végül a `printer` osztály a beállításoktól függően generálja az ArchC vagy Verilog formátumának megfelelő memória-inicializáló fájlt.

4.2 Utasításkészlet beolvasása

Az általános felhasználhatóság érdekében saját utasításkészletet specifikáló fájlformátumot határoztam meg. Egy utasítás bináris kódjában vannak fix mezők, amelyek azonosítják az utasítást, továbbá lehetnek operandus mezők, amelyeken az utasítás valamilyen műveletet végez. Ezért az ISA-t specifikáló fájlnak tartalmaznia kell az utasításszó hosszát, az utasítás mnemonikját (az utasításra jellemző, informativ rövid név), a fix és az operandusmezők számát, hosszát és pozícióját. Ezt az információt az `instruction` osztály tartalmazza, és ezek alapján az adatok alapján meghatározható tetszőleges utasítás bináris kódja.

```
16 i2rf f1 (15:12) 0 o2 (11:8) 0 (7:0) 0
16 scall f2 (15:12) 13 (11:8) 3 o1 (7:0) 0
16 ret f3 (15:12) 13 (11:8) 6 (7:0) 0 o0
```

4.2. ábra: Utasításkészlet definíció az assembler számára

Az 4.2. ábra egy utasításkészlet definíciós fájl részlete látható. Az első oszlopban az utasítás hossza szerepel bitben megadva, őt pedig az utasítás mnemonikja követi. Ezután meg kell adni a fix mezők számát (például f1), a fix mezők intervallumát és az egyes fix mezők értékét. Ezt követően ugyanezt meg kell tenni az operandus mezőire is. Amennyiben az utasításnak nincs operandusa, „o0” jelzést kell adni, enélkül ugyanis az utasítás nem kerül mentésre, ami az assembly fájl beolvasásakor hibát okozhat.

Az utasításkészlet beolvasását az `isafilereader` osztály végzi. Ebben az osztályban tárolom az utasításkészletet egy vector típusban, amelynek az `intruction` típust adtam sablonparaméterként.

4.3 Az assembly fájl beolvasása

Az assembly fájl beolvasásáért az `asmfilereader` osztály a felelős. Ennek az osztálynak a konstruktora egy stringet (assembly fájl neve), és a beolvasott utasításkészletre mutató referenciát kap. Ahhoz, hogy egy kezelhető nagyságú állapotgéppel meg lehessen valósítani az assembly fájl beolvasását, a fájl formátumára bizonyos megkötéseket határoztam meg.

Az assembly fájlban egy sorban csak egy assembly utasítás lehet. Az utasítások végét ’`;`’ karakterrel kell jelezni, ami ezt a karaktert követi, az kommentnek minősül. A címkéknek vagy „`r_`” vagy „`a_`” karakterekkel kell kezdődniük, és a címke inicializálásakor ’:’-ra kell végződniük. Az „`r_`” az IP-relatív (*Instruction Pointer*, utasítás mutató), „`a_`” az abszolút címzést jelzi. Előbbi esetén a címke értéke az inicializálás helyétől vett távolságot lesz, míg utóbbi esetén a pozíciószámláló értékét kapja meg. Ezzel a funkcióval abszolút és relatív címzésű architektúrákra is egyszerűen lehet tesztprogramot írni.

Lehetséges konstansok definiálása, például a „`const alma 5`” egy olyan konstans, amelynek értéke 5, a neve pedig alma. Konstans segítségével egyszerűen lehet hivatkozni regisztertömb egy elemére, vagy egy memóriacímre. A felhasználónak a pozíciószámláló módosításával lehetősége van arra, hogy egyes utasítások az általa

kiválasztott memóriacímre kerüljenek. A pozíció módosítását a ’@’ karakterrel lehet jelezni. A ’@’ karaktert követő szám jelzi az őt követő utasítás memóriacímét. Címzési módtól függően ez bájtcímet, vagy szócímet is jelenthet (a „@32” beállítástól függően vagy a 32-es bájtcímet, vagy a 32-es szócímet jelöli).

A 7. fejezet bemutat három tesztprogramot, amelyeket az elkészült assemblerrel implementáltam.

5 Eredmények – A Juliet példarendszer

Ebben a fejezetben bemutatom az általam tervezett Juliet mintaprocesszort, amelynek funkcionális és ciklushelyes modelljét ArchC-ben terveztem, majd a `Creator` algoritmus felhasználásával generáltam a szintetizálható Verilog leírását. A különböző reprezentációjú modellekhez tesztprogramokat készítettem, azok bináris kódját az assembler felhasználásával generáltam és funkcionális teszteket végeztem. A programok assembly forráskódja a 7. fejezetben található.

5.1 Az architektúra erőforrásai

A processzor hardvard archiktetúrájú (512 bájt utasításmemória, 256 bájt adatmemória). A memóriákon kívül a processzor architektúrájának része egy 48 elemű regisztertömb (`RF`), 8 bites stack pointer (`stackp`), egy regiszter, amellyel a regisztertömb elérhető elemeit lehet ablakozni (`RegW`), valamint 3 státusz flag (eredmény nulla, eredmény negatív, privilégiumszint). Az architektúra erőforrások deklarálását az 5.1. ábra mutatja be.

```
AC_ARCH (juliet){
 ac_wordsize 16;
 ac_mem DATA_MEM:256;
 ac_mem INST_MEM:512;
 ac_regbank <8>RF:48;
 ac_reg <8>stackp;
 ac_reg <8>RegW;
 ac_reg <1>zero;
 ac_reg <1>neg;
 ac_reg <1>priv;
 ARCH_CTOR (juliet) {
  ac_isa("juliet_isa.ac");
  set_endian("big");
 };
};
```

5.1. ábra: Funkcionális processzormodell architektúra erőforrásai

5.2 A processzor utasításkészlete

A tervezett processzor utasításkészlete 3 címes, (az utasításszóban meg kell adni az operandusok mellett az eredményregisztert is, nincs rögzített eredményregiszter vagy akkumulátor).

Mnemonik	Művelet	Operandusok		
i2rf	RF[rf_a]=dat	rf_a	dat	
ijmp	PC=PC+RF[rf_a]	func	rf_a	dc
m2rf	RF[rf_a1]=DATA_MEM[RF[rf_a2]+RF[rf_a3]]	rfa_1	rf_a2	rf_a3
rf2m	DATA_MEM[RF[rf_a2]+RF[rf_a3]]=RF[rf_a1]	rfa_1	rf_a2	rf_a3
jmpeq	RF[rf_a2]=RF[rf_a3]?PC=PC+RF[rf_a1]:PC++	rfa_1	rf_a2	rf_a3
jmpneq	RF[rf_a2]=RF[rf_a3]?PC++:PC=PC+RF[rf_a1]	rfa_1	rf_a2	rf_a3

add	RF[rf_a1]=RF[rf_a2]+RF[rf_a3]	rfa_1	rf_a2	rf_a3
sub	RF[rf_a1]=RF[rf_a2]-RF[rf_a3]	rfa_1	rf_a2	rf_a3
and	RF[rf_a1]=RF[rf_a2]&RF[rf_a3]	rfa_1	rf_a2	rf_a3
or	RF[rf_a1]=RF[rf_a2]|RF[rf_a3]	rfa_1	rf_a2	rf_a3
xor	RF[rf_a1]=RF[rf_a2]^RF[rf_a3]	rfa_1	rf_a2	rf_a3
shfl	RF[rf_a1]=RF[rf_a2]<<RF[rf_a3]	rfa_1	rf_a2	rf_a3
shfr	RF[rf_a1]=RF[rf_a2]>>RF[rf_a3]	rfa_1	rf_a2	rf_a3
jmp	PC=PC+adr		adr	
bz	ZERO ? PC=PC+adr : PC++		adr	
bn	NEG ? PC=PC+adr : PC++		adr	
scall	PRIV=0 ? (PC=PC+adr, PRIV=1, ST--) : PC++		adr	
ucall	PC=PC+adr, ST--		adr	
eoj	PC = 512		dc	
ret	ST++		dc	
chrw	REGW = ofs		ofs	

5.1. táblázat: A mintaprocesszor utasításkészlete

A regisztertömb címzésekor az operandusokhoz automatikusan hozzáadódik a RegW regiszter tartalma, amely az eltolást tartalmazza. Ezt az eltolást a chrw utasítással lehet módosítani 0-s privilégiumszint mellett. Az scall és ucall utasítások végrehajtásakor a stack-re elmentődik a programszámláló értéke, valamint a regiszterablak értéke és a privilégiumszint. Az scall csak 0-s privilégiumszinten hajtódik végre, ucall pedig átállítja 1-re a privilégiumszintet. A ret utasítás végrehajtásakor az elmentett értékek betöltődnek a stackről. A címzés IP-relatív módon történik, az új cím a korábbi utasításmutató-érték és az utasítás által meghatározott adat összegeként áll elő (az adat kettes komplemens, így az eltolás lehet negatív is). Az m2rf és rf2m utasítások adat mozgatását valósítják meg az adatmemória és a regisztertömb között. Az adatmemória címe a két operandus által meghatározott regiszterek értékeinek összegéből képződik. A negatív és nulla eredményt jelző flagek értékét azok az utasítások változtathatják, amelyek a regisztertömbbe írás műveletet hajtanak végre, ez a i2rf, m2rf, add, sub, and, or, xor, shfl és shfr (logikai shift) utasítás. A bz és bn utasítások a negatív és nulla eredményt jelző flagek értékének függvényében hajtanak végre ugrást. A funkcionális modellt több programmal is teszteltem. Ezek maradékos osztást, legnagyobb közös osztó keresést és szorzást valósítanak meg.

5.3 Ciklushelyes modell fejlesztése

5.3.1 Csővezeték implementálása

ArchC-ben egyszerűen lehet létrehozni csővezetéket, annak a kezelésével a tervezőnek nem kell foglalkoznia (például az utasítások léptetésével egyik fokozatból a másikba). A pipeline erőforrás deklarálása után az `actsim` eszköz által generált viselkedést leíró fájl sablonjában az utasítások viselkedését modellező függvények

kiegészülnek a csővezeték fokozatoknak megfelelő `switch-case` szerkezettel. Ezzel lehet meghatározni, hogy az egyes utasítások melyik fokozatban hogyan viselkedjenek.

A processzor egy háromfokozatú csővezetéket kapott, az adatokat két fokozat között egyéni formátumú regiszterek tárolják. Az első fokozatban történik az utasítás felhozatala (fetch), az utasításban található memória- és regisztercímek, illetve közvetlen adatok regiszterekbe mentődnek. Amennyiben ugró (vagy rutinhívó) utasítás van ebben a fokozatban, akkor a programszámláló értéke is módosul. A második fokozatban az operandusok bekerülnek a csővezetékbe az adatmemóriából vagy a regisztertömbből. Ekkor azonban előfordulhat, hogy olyan helyről olvasunk be adatot, amit az előző utasítás a végrehajtás fázisban éppen írni fog, így a beolvasott adat érvénytelen lesz. Ezt a problémát küszöböli ki az adat előrecsatolás (*data forwarding*). A harmadik fokozatban történik az utasítás tényleges végrehajtása (kivéve az ugró és a szubrutinhívó utasításokat, azok már az első fokozatban átállítják a programszámlálót). A BZ és BN utasítások kivételesek, ugyanis hiába ugró utasítások, nem derül ki, hogy melyik címről kell felhozni a következő utasítást, amíg az őket megelőző utasítás nem értékelődik ki. A processzor statikus elágazásbecslést végez, vagyis feltételes elágazás esetén mindig végrehajtódik az ugrás. Amennyiben végrehajtási fázisban kiderül, hogy hibás volt a feltételezés, akkor a csővezeték kiürül, és a megfelelő címről folytatódik az utasítások végrehajtása.

5.3.2 Adat-előrecsatolás

A csővezetékkel rendelkező processzorok egyik problémája az adat hazárd (fokozatok közötti adatfüggőség) kiküszöbölése. A bemutatott processzorban akkor lép fel adat hazárd, ha a csővezetékben közvetlenül követik egymást olyan utasítások, amelyek azonos adatokon hajtanak végre valamilyen műveletet. Amennyiben egy utasítás a harmadik fokozatban egy regiszter vagy az adatmemória egy rekeszét írja, és az őt követő utasítás a második fokozatban ugyanezt a regisztert vagy memóriaterületet olvassa, akkor a második utasítás nem a módosított értéket olvassa be, hiszen az érték csak akkor frissül, ha végrehajtódott az utasítás. Ez a probléma kezelhető hardveres és szoftveres úton is.

Amennyiben a modellben nincs kiküszöbölve a hazárd, akkor programok írásakor figyelembe kell venni, hogy az egymást követő utasítások között ne legyen adatfüggőség. Ezzel egyszerűvé válik a processzormodell megtervezése, azonban a csővezetékezés előnyei nem jelentkeznek (egymást követő utasítások adatfüggése esetén NOP utasításokkal kellene beilleszteni), és a funkcionális modell teszteléséhez használt programokat újra is kellene írni.

A második lehetőség hardveres megoldás, vagyis a modellt fel kell készíteni az egymást követő utasítások adatfüggőségeinek vizsgálatára. Így ha egymást követően egy utasítás ír egy regisztert, vagy egy memóriaterületet, és a következő utasítás ugyanezt a regisztert vagy memóriatartalmat használná fel, akkor a második fokozatban behozott operandusértéket figyelmen kívül hagyja a modell, és az előző utasítás eredményét használja fel helyette.

Én az utóbbi megoldást alkalmaztam.

5.3.3 Megvalósítás

Az 5.2. ábra és az 5.3. ábra az XOR utasítás megvalósításának egy részletét mutatja be ArchC-ben és Verilogban. Az első feltételvizsgálat az adat-előrecsatolás miatt szükséges. Minden utasítás az eredményét a célregiszteren kívül egy eredményregiszterbe is menti, így ha a következő utasítás az előző eredményére hivatkozna, akkor nem szükséges az adatot újra betöltenie, hanem fel tudja használni az eredményregisztert operandusként.

```
void ac_behavior(xor){
switch (stage){
 case id_pipe_IF:
  ac_pc += 2;
  break;
 case id_pipe_ID:
  break;
 case id_pipe_EX:
  if((regEX.regwrite==1)&&(regEX.rdest==regID.rdest2)){
   if ((regEX.res^regID.op3)==0)status.zero=1;
   else status.zero=0;
   if ((regEX.res^regID.op3)&0x80)status.neg=1;
   else status.neg=0;
   RF.write((status.RegW+regID.rdest1),(regEX.res^regID.op3));
   regEX.res=regEX.res^regID.op3;
   regEX.rdest=regID.rdest1;
   }
  else if((regEX.regwrite==1)&&(regEX.rdest==regID.rdest3)){
   if ((regID.op2^regEX.res)==0)status.zero=1;
   else status.zero=0;
   if ((regID.op2^regEX.res)&0x80)status.neg=1;
   else status.neg=0;
   RF.write((status.RegW+regID.rdest1),(regID.op2^regEX.res));
   regEX.res=regID.op2^regEX.res;
   regEX.rdest=regID.rdest1;
   }
  else{
   if((regID.op2^regID.op3)==0)status.zero=1;
   else status.zero=0;
   if ((regID.op2^regID.op3)&0x80)status.neg=1;
   else status.neg=0;
   RF.write((status.RegW+regID.rdest1),(regID.op2^regID.op3));
   regEX.res=regID.op2^regID.op3;
   regEX.rdest=regID.rdest1;
   }
  break;
 default:
  break;
 }
return;
}
```

5.2. ábra:Részlet az XOR utasítás megvalósításából ArchC-ben

Az ArchC és a Verilog szintaxisa nagyon hasonló, azonban az ArchC speciális típusait a Verilog nem ismeri, így azoknak a Verilog implementációjára külön figyelni kell. Ilyen például az RF regisztertömb, amelyet write és read függvények hívásával lehet írni és olvasni. A `status_zero` és `status_neg` regiszterek értéke abban az esetben egy, ha az eredmény nulla, vagy negatív. Mivel az eredmény több operandusból

is származhat, ezért azt az összes kombináció esetére meg kell vizsgálni. Ez a megoldás nem hatékonyan használja az erőforrásokat, de az ArchC nyelv nem elég fejlett ciklushelyes modellezés tekintetében ahhoz, hogy optimális megoldást adjon az adat-előrecsatolás megvalósításához.

```
if((regEX_regwrite==1)&&(regEX_rdest==regID_rdest2)) begin
 if((regEX_res^regID_op3)==0) begin
  status_zero <= 1;
  end
 else begin
  status_zero <= 0;
  end
 if((regEX_res^regID_op3)&8'h80) begin
  status_neg <= 1;
  end
 else begin
  status_neg <= 0;
  end
 RF[(status_RegW+regID_rdest1)] <= (regEX_res^regID_op3);
 regEX_res <= regEX_res^regID_op3;
 regEX_rdest <= regID_rdest1;
 end
else begin
 if((regEX_regwrite==1)&&(regEX_rdest==regID_rdest3)) begin
  if((regID_op2^regEX_res)==0) begin
   status_zero <= 1;
   end
  else begin
   status_zero <= 0;
   end
  if((regID_op2^regEX_res)&8'h80) begin
   status_neg <= 1;
   end
  else begin
   status_neg <= 0;
   end
  RF[(status_RegW+regID_rdest1)] <= (regID_op2^regEX_res);
  regEX_res <= regID_op2^regEX_res;
  regEX_rdest <= regID_rdest1;
  end
 else begin
  if((regID_op2^regID_op3)==0) begin
   status_zero <= 1;
   end
  else begin
   status_zero <= 0;
   end
  if((regID_op2^regID_op3)&8'h80) begin
   status_neg <= 1;
   end
  else begin
   status_neg <= 0;
   end
  RF[(status_RegW+regID_rdest1)] <= (regID_op2^regID_op3);
  regEX_res <= regID_op2^regID_op3;
  regEX_rdest <= regID_rdest1;
  end
 end
end
```

5.3. ábra: Részlet az XOR utasítás megvalósításából Verilogban

5.3.4 A logikai szintézis eredményei

Az RTL modell generálása után elvégeztem a logikai szintézist Altera és Xilinx FPGA-ra, illetve ASIC technológiára. FPGA esetén a gyártók fejlesztőkörnyezetét használtam (Quartus II 13.0, illetve ISE 14.6), ASIC-et pedig a Cadence Encounter RTL Compiler (v11.21) programmal szintetizáltam az AMS (*Austria Micro Systems*) 180nm-es technológiájára. Az eredményeket az 5.2. táblázat foglalja össze.

Xilinx (Spartan 6 XC6SLX75T)		**Altera** (Cyclone II EP2C70F896I8)		**Cadence**	
Flip-flop	**LUT**	**Regiszter**	**Logikai elem**	**Cella**	**Terület [µm^2]**
2483	15681	2466	15151	26270	998505

5.2. táblázat: A szintézis eredménye

Mivel az utasítás- és adatmemória az FPGA-n belül lett implementálva, valamint a szintézer nem használt speciális erőforrásokat (például szorzó áramkört, blokkramot), így a feltüntetett erőforrásigény nagynak mondható. ASIC esetén RTL Compiler úgynevezett „timing-driven” szintézist végez, ami azt jelenti, hogy egy kívánt órajelfrekvencia elérésére törekszünk minimális erőforrásigény mellett. A táblázatban látható adatok 100MHz-es cél órajelfrekvenciára vonatkoznak. Nagyobb frekvencia esetén több cellát igényelne az áramkör, így a területigény nagyobbnak adódna.

A leszintetizált áramkörök funkcionális helyességét kapuszintű (poszt-szintézis) szimulációval ellenőriztem.

6 Összefoglalás

A dolgozat az alkalmazás-specifikus mikroprocesszorok (*Application-Specific Instruction Set Processor*, ASIP) automatizált logikai szintézisének lehetőségeivel foglalkozik. A szakirodalomban számos olyan modellező nyelv és szoftver eszköz megtalálható, amelyek e tárolt programú gépek viselkedésének leírását, szimulációját, a hozzájuk tartozó szoftver eszközök (assembler, compiler, debugger stb.) létrehozását segítik, a magas szintű leírásból kiinduló automatizált hardverszintézis azonban kevésbé hangsúlyos.

A dolgozat egy, az ArchC modellezési módszerre épülő szoftver eszközt, a *Creator*-t mutatja be. Az ArchC architektúra leíró nyelv ez idáig csak utasításkészlet tesztelésére, és ciklushelyes modellezésre volt alkalmas, de a dolgozatban bemutatott szoftver felhasználásával a ciklushelyes modellből automatikusan generálható egy RTL szintű leírás, amely a már rendelkezésre álló logikai szintézis eszközök felhasználásával kapuszintű modellé transzformálható. Ezáltal a szoftver kiküszöböli az időigényes kézi RTL szintű modellezést, amely sok hibalehetőséget rejt magában az eltérő absztrakciós szintű modellek konzisztenciája tekintetében.

Bemutatásra került továbbá egy szintén saját fejlesztésű generikus assembler fordító, amely lehetővé teszi a fejlesztés alatt álló mikroprocesszor architektúrák funkcionális verifikációjához szükséges tesztprogramok létrehozását ArchC és Verilog nyelvű modellhez.

Köszönetnyilvánítás

Ezúton köszönöm konzulensemnek, Horváth Péternek a dolgozat elkészítése során adott útmutatását és a folyamatos segítséget.

7 Függelék

Ez a fejezet bemutatja az ismert számelméleti algoritmusokat realizáló három tesztprogram assembly kódját, amelyeket a Juliet példarendszer funkcionális tesztjéhez használtam. A programok nem hajtanak végre teljes körű tesztelést, de elég nagy a kódlefedésük ahhoz, hogy a címzési, vagy időzítésbeli hibák kiderüljenek. Az ArchC és Verilog modellhez szükséges bináris kódot az általam tervezett assembler fordító állította elő.

7.1.1 Legnagyobb közös osztót kereső tesztprogram

A programot az euklideszi algoritmus alapján valósítottam meg. Az alkamazás bemenetként két egész számot kap. Ezeknek veszi az abszolút értékét, és egy ciklusban addig képezi a két szám modulóját, amíg az nulla nem lesz. Ekkor a ciklusnak vége szakad, és az eredmény előáll.

```
const ZER  0 ; 0-at tartalmazo regiszter
const ONE  1 ; 1-et tartalmazo regiszter
const INV  2 ; invertalashoz szukeseges egy regiszter
const NbA  3 ; egyik bemeno adatot tartalmazo regiszter
const NbB  4 ; masik bemeno adatot tartalamzo regiszter
const TMP  5 ; ideiglenes adatot tartalmazo regiszter

        i2rf ZER 0       ; inicializalas
        i2rf ONE 1       ;
        i2rf INV 255     ;
        i2rf NbA 54      ;
        i2rf NbB 36      ;
        add  NbA NbA ZER ; bemeno adat elojel vizsgalata
        bn    r_L1       ; ha negativ, akkor negalni kell
        jmp  r_L2        ; nem volt negativ, nem kell negalni
r_L1:   xor  NbA NbA INV ; kettes komplemens, negalas
        add  NbA NbA ONE ; inkrementalas
r_L2:   add  NbB NbB ZER ; elojel vizsgalat
        bn   r_L3        ; negativ, akkor negalni kell
        jmp  r_cikl      ; ha nem volt negativ, akkor ugrik a ciklusra
r_L3:   xor  NbB NbB INV ; negalas
        add  NbB NbB ONE ; inkrementalas
r_cikl: add  TMP NbB ZER ; temp valtozo felveszi a B bemeno adat erteket
r_oszt: sub  NbA NbA NbB ; A=A-B
        bn   r_L4        ; negativ, akkor rollback kell
        jmp  r_oszt      ; nem volt negativ, akkor megint ki lehet vonni
r_L4:   add  NbA NbA NbB ; rollback A=A+B
        add  NbB NbA ZER ; B=A
        add  NbA TMP ZER ; A=temp
        add  NbB NbB ZER ; B elojel vizsgalat
        bz  r_veg        ; ha nulla, akkor vege
        jmp  r_cikl      ; nem volt nulla, kezdodik a ciklus elorol
r_veg:  add  NbA NbA ZER ; a legnagyobb kozos osztot tartalmazo regiszter
        eoj              ;
```

7.1. ábra: A legnagyobb közös osztó algoritmusa assemblyben megvalósítva

7.1.2 Maradékos osztást végző tesztprogram

A maradékos osztást végző algoritmus ciklikus kivonáson alapul. A program a bemeneti adatok kiértékelése után addig vonja ki az osztót az osztandóból. Ha az

eredmény kisebb, mint az osztó, akkor előállt a maradék. A hányados megegyezik a ciklus iterációjának számával.

```
const K0  0             ; 0-at tartalmazo regiszter
const K1  1             ; 1-et tartalmazo regiszter
const INV 2             ; invertalashoz szukeseges regiszter
const K4  3             ; konstans 4
const K2  4             ; konstans 2
const K3  5             ; konstans 3
const NUM 15            ; osztando
const DEN 14            ; oszto
const S_N 13            ; osztando elojele (0 poz, 1 neg)
const S_D 12            ; oszto elojele (0 poz, 1 neg)
const RES 10            ; eredmeny

r_div: i2rf K0  0       ; inicializalas
       i2rf K1  1       ;
       i2rf INV -1      ;
       i2rf K4  4       ;
       i2rf K2  2       ;
       i2rf K3  3       ;
       i2rf NUM 37      ; osztando
       i2rf DEN 7       ; oszto
       add  DEN DEN 0   ; oszto ellenorzese
       bz  r_L8         ; nulla eseten ugras r_L8ra
       i2rf S_N 0       ;
       i2rf S_D 0       ;
       i2rf RES 0       ;
       add  NUM NUM K0  ; elojelvizsgalat
       bn   r_L1        ; ha negatív, ugrás r_L1-re
       jmp  r_L2        ; egyébként ugrás r_L2-re
r_L1:  i2rf S_N 1       ; osztando negativ, S_N->1
       xor  NUM NUM INV ; kettes kompl->negalas
       add  NUM NUM K1  ; inkrementalas
r_L2:  add  DEN DEN K0  ; elojelvizsgalat
       bn   r_L3        ; ha negativ, ugras r_L3
       jmp  r_L4        ; ugras r_L4
r_L3:  i2rf S_D 1       ; oszto negativ, S_D->1
       xor  DEN DEN INV ; kettes kompl->negalas
       add  DEN DEN K1  ; inkrementalas
r_L4:  sub  NUM NUM DEN ; osztando = osztando - oszto
       bn   r_L5        ; ha az eredmeny negativ, rollback
       add  RES RES K1  ; eredmeny inkrementalasa
       jmp  r_L4        ; ciklus elejere ugras
r_L5:  add  NUM NUM DEN ; rollback: osztando mar a maradek
       sub  8 S_D S_N   ; elojelek osszehasonlitasa
       bz   r_L6        ; egyezes eseten ugras r_L6
       xor  RES RES INV ; egyebkent hanyados negativ, negalas
       add  RES RES K1  ; inkrementalas
r_L6:  add  S_N S_N K0  ; osztando elojele
       bz  r_L7         ; ha poz, ugras R_L7
       xor  NUM NUM INV ; egyebkent maradek negativ, negalas
       add  NUM NUM K1  ; inkrementalas
r_L7:  rf2m RES K0 K2   ; hanyados masolasa DMEM[2]
       rf2m NUM K0 K3   ; maradek masolasa DMEM[3]
       eoj              ; vege
r_L8:  rf2m K1 K4 K0    ; nullaval osztas, DMEM[4]=1
       jmp  r_L7        ;
```

7.2. ábra: Maradékos osztás algoritmusa assemblyben

7.1.3 Szorzó tesztprogram

A szorzó algoritmus shiftelésen és összeadáson alapul. A szorzás a szorzó legalsó helyiértékének vizsgálatával kezdődik. Ha ezen a helyiértéken 1 van, akkor az eredményhez hozzá kell adni a szorzandó számot. Ez után a szorzandó számot, és a maszkot is shiftelni kell eggyel balra. Ez a ciklus ismétlődik annyiszor, ahány bitesek voltak a bemenő számok (jelen esetben 8).

```
const ZER  0 ; 0-at tartalmazo regiszter
const ONE  1 ; 1-et tartalmazo regiszter
const INV  2 ; invertalashoz szukeseges egy regiszter
const NbA  3 ; szorzando
const NbB  4 ; szorzo
const BIT  5 ; bitek szamat tartalmazo regiszter (ciklushoz)
const MSK  6 ; maszkot tartalmazo regiszter
const RES  7 ; eredmeny regiszter
const TMP  8 ; ideiglenes adat

        i2rf NbA 3       ; inicializalas
        i2rf NbB 5       ;
        i2rf ONE 1       ;
        i2rf  BIT 8      ;
        i2rf MSK 1       ;
        i2rf ZER 0       ;
r_kezd: and  TMP NbB MSK ; szorzo legalso bitjenek vizsgalata
        bz  r_L1         ; ha nulla, akkor nem kell eredmenyhez adni
        add  RES RES NbA ; nem nulla, az eredmenyhez hoza kell adni A-t
r_L1:   shfl NbA NbA ONE ; ciklus kezeles, A shiftelese eggyel balra
        shfl MSK MSK ONE ; maszk shiftelese eggyel balra
        sub  BIT BIT ONE ; ciklus iteralo valtozojabol ki kell vonni egyet
        bz  r_veg        ; ha mar nulla, akkor vege
        jmp  r_kezd      ; nem volt nulla, meg egyszer lefut a ciklus
r_veg:  rf2m RES ZER ZER ; eredmeny masolasa a memoriaba
        eoj              ; vege
```

7.3. ábra: Shift and add szorzóalgoritmus megvalósítása

8 Irodalomjegyzék

[1] O. Schliebush, H. Meyr and R. Leupers, Optimized ASIP Synthesis from Architecture Description Language Models, Dordrecht, The Netherlands: Springer, 2007.

[2] Y. Zhang, H. He, Z. Shen and Y. Sun, "ASIP Approach for Multimedia Applications Based on Scalable VLIW DSP Architecture," *Tsinghua Science and Technology,* vol. 14, no. 1, pp. 126-132, 2009.

[3] H. Peters, R. Sethuraman, A. Berić, P. Meuwissen, S. Balakrishnan, C. A. P. Antonio, W. Kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen and G. de Haan, "Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications," *IEEE Transactions on Cirtuits and Systems for Video Technology,* vol. 15, no. 4, pp. 508-527, 2005.

[4] M. W. Ernst, S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler and E. Martin Witte, "Application-Specific Instruction-Set Processor for Retinex-Like Image and Video Processing," *IEEE Transactions on Circuits and Systems-II: Express Briefs,* vol. 54, no. 7, pp. 596-600, 2007.

[5] Z. Liu, K. Dickson and J. V. McCanny, "Application-Specific Instruction Set Processor for SoC Implementation of Modern Signal Processing Algorithms," *IEEE Transaction on Circuits and Systems-I: Regular Papers,* vol. 52, no. 4, pp. 755-765, 2005.

[6] A. R. Jafri, A. Baghdadi and M. Jézéquel, "ASIP-Based Universal Demapper for Multiwireless Standards," *IEEE Embedded Systems Letters,* vol. 1, no. 1, pp. 9-13, 2009.

[7] O. Muller, A. Baghdadi and M. Jézéquel, "From Parallelism Levels to a Multi-ASIP Architecture for Turbo Decoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 17, no. 1, pp. 92-102, 2009.

[8] T. Vogt and N. Wehn, "A Reconfigurable ASIP for Convolutional and Turbo Decoding in an SDR Environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 10, pp. 1309-1320, 2008.

[9] A. Pyttel, A. Sedlmeier and C. Veith, "PSCP: A Scalable Parallel ASIP Architecture for Reactive Systems," *Design, Automation and Test in Europe,* pp. 370-376, 1998.

[10] M. James, E. M. Abdulhussain and H. Mark, "Application-Specific Instruction-Set Processor for Control of Multi-Rail DC-DC Converter Systems," *IEEE Transaction on Circuits and Systems-I: Regular Papers,* vol. 60, no. 1, pp. 243-254, 2013.

[11] T. Good and M. Benaissa, "Very Small FPGA Application-Specific Instruction Processor for AES," *IEEE Transactions on Circuits and Systems-I: Regular Papers,* vol. 53, no. 7, pp. 1477-1486, 2006.

[12] M. Prabhat and D. Nikil, Processor Description Languages, Morgan Kaufmann, 2008.