Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Fejér Attila

INTEGRÁLT FEJLESZTŐI KÖRNYEZET EGYEDI SOFT-CORE PROCESSZORHOZ

KONZULENS

Raikovich Tamás

BUDAPEST, 2013

Tartalomjegyzék

Összefoglaló

A digitális technika oktatásában kiemelt fontosságú a processzorok működési elvének megértése, ehhez célszerű egyszerű felépítésű eszközt bemutatni. A szakdolgozat témám a PicoBlaze soft-core processzorhoz készült integrált fejlesztői környezet volt, ami megkönnyítette az asm nyelvű programok írását, ám a processzor HDL forráskódja nehezen értelmezhető, így oktatási célokra nem a legalkalmasabb választás.

Az ötletet továbbgondolva Raikovich Tamás elkészítette a MiniRISC nevű 8 bites mikroprocesszort, ami nagyon egyszerű felépítésének köszönhetően ideális a HDL forráskód szintű bemutatásra is. Ehhez a processzorhoz eredetileg egy parancssoros fordító készült, ami a mai környezetekkel összehasonlítva idegen a hallgatók számára. Kézenfekvő volt a PicoBlaze-hez készült fejlesztői környezetet alapnak használni és továbbfejleszteni, hogy egy kényelmesen használható eszközt adjunk a hallgatók kezébe.

A MiniRISC elsődlegesen a szintén tanszéki fejlesztésű Logsys Spartan-3E kártyán fut, így ennek a panelnak a lehetőségeit vettem alapul.

A fejlesztői környezet tartalmazza a szokásos fejlesztői és debug funkciókat, úgy mint fordítás, eszköz felprogramozása, futtatás, töréspontok elhelyezése stb. Ezen túl hozzáférést biztosít a memóriatartalomhoz, a processzor erőforrásaihoz (programszámláló, státuszbitek, regiszterek, verem teteje), valamint a perifériákhoz (LED-ek, nyomógombok, kapcsolók, hétszegmenses kijelző, 5x7-es pontmátrix, bővítőcsatlakozók, USRT).

A kezdeti lépésekhez nincs feltétlenül szükség fizikai eszközön való futtatásra, így a szoftver részét képezi egy szimulátor is, aminek a segítségével könnyen elkezdhető a platformmal való ismerkedés.

A környezet tervezése során nagy hangsúlyt fektettem a továbbfejleszthetőségre is, így könnyen illeszthető a felhasználói felülethez új periféria, vagy akár lecserélhető a fordító.

Abstract

In the education of Digital Technology it's really important to understand how processors work. It's easy to achieve this with presenting a simple device. The topic of my thesis was an integrated development environment to the PicoBlaze soft-core processor, which made easier developing programs in asm. However, the HDL source of this CPU is hardly understandable. Because of this it isn't a good choice for educational purposes.

After further considerations Tamás Raikovich developed an 8 bit microprocessor called MiniRISC with a really simple structure, so it could be presented in HDL level. For this processor he made a console assembler, but it was strange to use compared to the modern development tools. It was an obvious choice to use the PicoBlaze IDE as a base. After further developments we could provide an easy-to-use tool for the MiniRISC processor.

The primary platform of the MiniRISC is the Logsys Spartan-3E FPGA board, which was also developed on the department.

The IDE contains the usual developer and debugger functions, like compilation, device programming, run, breakpoints etc. It also provides access to the data memory, the processor resources (program counter, flags, registers, top of the stack), and the peripherals (LEDs, push buttons, switches, 7 segment display, 5x7 dot matrix, GPIO ports, USRT).

To make the first steps it isn't required to run the program on a physical device, so a simulator is part of the software too. Using this beginners can easily make the first steps with the platform.

When designing the software I considered making further developments simple as a top priority goal, so it will be easy to implement new peripherals or replace the compiler.

1 Bevezetés

Az egyetemi informatikus és villamosmérnök képzések során kiemelt fontosságú, hogy a hallgatók megismerjék a processzorok alapvető működési elveit. Ehhez szükségük van valamilyen mintarendszerre, aminek a hardveres és szoftveres tulajdonságai egyaránt könnyen megismerhetők.

A hardveres részt tekintve célszerű valamilyen HDL nyelven írt megoldást bemutatni, hiszen így könnyebben megérthető az egységek működése. Az FPGA gyártók kínálnak egyszerűbb és bonyolultabb processzorokat, például a Xilinx cég PicoBlaze-t (1) és a MicroBlaze-t (2), a Lattice a Mico8-at (3) és Mico32-t (4), az Altera pedig a Nios II-t (5). Ezeknek megvan az a hátránya, hogy jellemzően a saját eszközeikre tervezték őket, így ha el is érhető a forráskódjuk, az nem fordítható le más FPGA-kra (ez alól kivétel a Mico8 és Mico32). További problémát jelent a fejlesztőeszközök jellege is: az egyszerűbb, ingyenes processzorok csak minimális támogatást élveznek, míg a bonyolultabbakhoz ugyan bőven kapunk fejlesztési támogatást, de ezek forráskódja nem elérhető.

Természetesen elérhetők harmadik fél által fejlesztett processzorok is, például az OpenCores (6) rengeteg ilyen eszközt gyűjtött össze. Azonban ezekhez a megoldásokhoz még nehezebb egy tisztességes fejlesztőkörnyezetet találni, így ez sem megfelelő megoldás.

A probléma elhárítására a tanszéken az a javaslat született, hogy egy saját fejlesztésű eszközkészlettel kellene kiváltani az eddigi megoldásokat. A hardveres részt tekintve a Raikovich Tamás által fejlesztett LOGSYS Spartan-3E FPGA panel (7) és a 8 bites MiniRISC (8) processzorok adták az alapot. Ehhez elkészített egy egyszerű, parancssoros assembly fordítót is.

Ezt a fordítót felhasználva fejlesztettem tovább a már meglévő, PicoBlaze-hez készült fejlesztői környezetemet (9), ami a LOGSYS panelen futó alkalmazás hibakeresését támogatta. Ez egy fontos funkció, hiszen a felhasználó szereti látni, hogy a gyakorlatban hogyan működik a megoldása.

Azonban a kezdeti lépésekhez nem feltétlenül van szükség a hardveren való futtatáshoz, ahhoz adott esetben elegendő lehet egy szimulátor is. Ez egy teljesen új

igény volt a fejlesztőkörnyezettel szemben, hiszen a PicoBlaze-hez nem készítettem ilyet. A cél a teljes utasításkészletet és a perifériákat a valós környezethez hasonló módon szimuláló szoftver elkészítése volt, így ezzel már otthon is bárki elkezdhet ismerkedni az assembly nyelvű szoftverfejlesztéssel egy minden részletekig ismert platformon.

2 Az elérhető technológiák áttekintése

2.1 Kemény- és lágymagos processzorok

Korábban a mikrovezérlő alapú rendszerek készítésének egyetlen módja a perifériák és egyéb áramköri elemek dedikált processzor köré való építése volt. Természetesen ekkor is rendelkeztek különböző képességű és felszereltségű chipek, de az elektronikai ipar fejlődésével a költséghatékonyság és az integráltság iránti igény nőtt, megjelentek az úgynevezett SoC rendszerek, ahol az erőforrások döntő többsége egy tokba került. Már ezek az eszközök is forradalminak számítottak, de az igazi áttörést az FPGA-k megjelenése hozta meg, amiknek köszönhetően megszülettek az SoPC.

Lehetővé vált az ún. lágymagos processzorok alkalmazása. Ezeknél az eszközöknél a processzor és a perifériák is HDL szinten kerülnek implementálásra, így a teljes rendszer egy tokban foglal helyet. Az FPGA újrakonfigurálásával lehetővé vált az eddigi rendszer funkcióinak rugalmas bővítése, továbbá az esetleges tervezési hibák könnyű és olcsó javítása, hiszen a hardvert nem, csak a konfigurációt kellett lecserélni.

Természetesen ahogy a kemény-, úgy a lágymagos processzorokból is léteznek különböző képességű változatok. A belépő szintű termékek általában ingyen elérhetők, a képességeik rengeteg esetben a gyakorlati alkalmazásokban is elegendőnek bizonyulnak. Viszont a gyártók érdeke a nagyobb képességű, fizetős termékek eladása, ezért valamivel rá kell venni a vásárlókat az ingyenes modellek mellőzésére. Ezt a fejlesztési támogatás megvonásával érik el, vagyis míg a fizetős termékekhez igen komplex, a legtöbb igényt kielégítő fejlesztőeszközöket nyújtanak, addig az ingyenes processzorokhoz csak az abszolút minimum érhető el, amikkel a fejlesztés lassú és nehézkes, rengeteg funkció (például a hibakeresés) pedig egyáltalán nem áll rendelkezésre. (9)

2.1.1 PicoBlaze

Az első lágymagos processzor a Xilinx által készített PicoBlaze volt. Ez egy kis méretű, sokrétű, költséghatékony, beágyazott 8 bites CPU, RISC utasításkészlettel,

kifejezetten a Xilinx FPGA-k képességeihez igazítva[1]. Mindössze 96 slice-ot foglal el, ami a Spartan-3 család XC3S50-es tagjánál az erőforrások 12,5%-át, míg az XC3S5000 esetében csupán 0,3%-át jelenti!

A program memória tipikusan egy block RAM, ami 1024 utasítás tárolását teszi lehetővé. A processzor erőforrását képezi a 16 darab 8 bit széles általános célú regiszter, a 64 bájtos memória és a 8 bites aritmetikai és logikai egység ZERO és CARRY státuszbitekkel.

A programmemóriát a felhasználónak kell a processzorhoz illesztenie (természetesen az FPGA-n belül), ahogyan a perifériákat is. Bemeneti és kimeneti perifériából egyaránt 256-256 darabot támogat a processzor.

A CPU támogatja a szubrutinhívást, 31 szintű, automatikus kezelésű veremmel rendelkezik. Ezen felül egyszintű megszakításkezeléssel is ellátták.

Teljesítményét tekintve minden utasítás 2 órajel alatt fut le, ami akár 200MHz-es is lehet (pl. Virtex-4 esetén). A megszakításra legkésőbb 5 órajelciklus alatt válaszol. (1)

2.1.2 MicroBlaze

A MicroBlaze szintén a Xilinx terméke, azonban ennek a forráskódja ingyen nem elérhető. Fontosabb különbség, hogy 32 bites és sokkal nagyobb teljesítmény elérésére tervezték. Részletesen konfigurálható, így kiválasztható a buszok típusa, a memória mérete, az illesztett perifériák stb. Összetettsége miatt (és mivel a forráskódja nem hozzáférhető) alapszintű oktatási célokra nem alkalmas. (2)

2.1.3 Mico8

A Mico8 a Lattice PicoBlaze-hez hasonló megoldása, nyílt forráskóddal, ami más eszközökre is lefordítható. Itt több erőforrás és több konfigurációs lehetőség áll a rendelkezésünkre:

- 18 bit széles utasítások,
- 16 vagy 32 db általános célú regiszter,

[1] Olyannyira, hogy a HDL kódja FPGA primitívek példányosításából áll, így csak arra a családra implementálható, amelyikhez készítették.

- külső vagy belső utasításmemória Wishbone interfésszel, 256, 512, 1K, 2K vagy 4K utasítással,
- külső vagy belső memória Wishbone interfésszel, maximum 4GB kapacitásig lapozással (256 bájtos lapokkal),
- minimum 2 órajelciklus utasításonként,
- 8, 16 vagy 32 mély call stack,
- 8 külső interrupt támogatása,
- előre elkészített perifériák, mint például UART, SPI, DMA stb.

Ami még a Mico8 különlegessége, hogy kapunk hozzá egy Eclipse alapú C/C++ fejlesztői környezetet, azonban ez az asm nyelvű programok írását nem teszi lehetővé. (3)

2.1.4 Mico32

A Mico8-Mico32 viszony hasonló a PicoBlaze-MicroBlaze-éhoz, vagyis a Mico32 a Lattice kereskedelmi céllal létrehozott, jobban támogatott, 32 bites beágyazott processzora. Ez is egy sokrétűen konfigurálható CPU. Összetettsége miatt oktatási célokra ez sem egy ideális választás. A Lattice nyílt forrású elveivel összhangban ennek az eszköznek is ingyen elérhető a forráskódja (4)

2.1.5 Nios II

A Nios II képességeit tekintve egy ligában játszik a MicroBlaze és Mico32 párossal, nagyon hasonlatos a másik két CPU-hoz. Ez sem egy ingyenes megoldás, és bonyolultsága miatt szintén nem használható oktatási célokra. (5)

2.2 Fejlesztőeszközök a PicoBlaze-hez

A PicoBlaze-el kapcsolatos tapasztalataim okán ezt a processzort választottam a további vizsgálatok alanyának, így a PicoBlaze esetén elérhető megoldásokat fogom a következő szakaszokban bemutatni.

2.2.1 KCPSM3

A gyár fejlesztőeszközei nem kényeztetik a felhasználókat a funkciók gazdagságával: kapunk egy egyszerű assemblert és szimulációs lehetőséget a Xilinx egyéb szoftvereiben. Minket most az előbbi érint, ezért erről ejtenék néhány szót.

A Spartan-3 FPGA-khoz tervezett PicoBlaze (KCPSM3) esetén a fordító egy DOS-os program (KCPSM3.EXE), ami 64 bites operációs rendszer alatt el sem indul! Az újabb, Spartan-6 eszközökhöz készített változat már ablakos alkalmazás, bár az is csak a legalapvetőbb funkciókat támogatja. A LOGSYS rendszer Spartan-3E FPGA-val van ellátva, így csak a KCPSM3-al foglalkozom.

A KCPSM3 fordító csak a legalapvetőbb funkciókat támogatja:

- konstansok definiálása,
- regiszterek átnevezése,
- memória adott területére „ugrás”.

Így például már a makrók támogatása is hiányzik a repertoárból, a hibakeresésről és a fejlesztői környezetről nem is beszélve! A program feladata csupán a felhasználói kód lefordítása.

A KCPSM3-at használva a fejlesztés menete a következő:

1. A felhasználó tetszőleges szövegszerkesztőben megírja a program forráskódját,
2. a KCPSM3 fordítóval lefordítja,
3. a generált HDL fájlok egyikét a hardver tervbe illeszti,[2]
4. szintetizálja a hardver tervet,
5. amivel végül felprogramozza az FPGA-t.

Az esetleges hibákat rövid, viszonylag egyértelmű üzenetekkel jelzi a fordító a folyamat végén.

Ez a fejlesztési módszer rendkívül kényelmetlen a felhasználó számára, hiszen legalább 3 szoftvert kell használnia! Ezen túl csak a szintézis 1-2 percet vesz igénybe,

[2] Verilog vagy VHDL, attól függően melyiket részesíti előnyben a fejlesztés során

vagyis ha a hiba oka nem egyértelmű, és nem a jó helyen keresi a felhasználó, akkor két futtatás között 2-3 perc is eltelhet. Ez már önmagában elég ahhoz, hogy a kezdő vagy tapasztalatlan felhasználók kedvét elvegye az eszköz használatától, de egy-egy hiba megtalálása a rutinos fejlesztők dolgát is könnyen megnehezíti. (10)

2.2.2 KCAsm

A KCAsm egy Java nyelven íródott, egyszerű assembler, ami funkcionalitását tekintve megegyezik a KCPSM3 fordítóval. A Java implementációnak köszönhetően nem csak a 32 bites Windows-hoz, hanem egyáltalán az operációs rendszerhez való kötöttség is megszűnt.

A KCAsm a PacoBlaze projekt része, aminek célja az eredeti PicoBlaze processzor Verilog nyelven történő implementálása FPGA primitívek használata nélkül. Ennek vitathatatlan előnye, hogy nincs annyira korlátozva a felhasználható FPGA-k köre, viszont hátránya a kevésbé hatékony szintézis. (11)

2.2.3 JTAG Loader

Ez az eszköz néhány DOS-os programból és egy kötegfájlból áll, a PicoBlaze csomaggal együtt érkezik. A célja az lenne, hogy kiküszöbölje az újraszintetizálást, és a felkonfigurált FPGA-ban csak a programmemória tartalmát frissítse a KCPSM3-al való fordítás után.

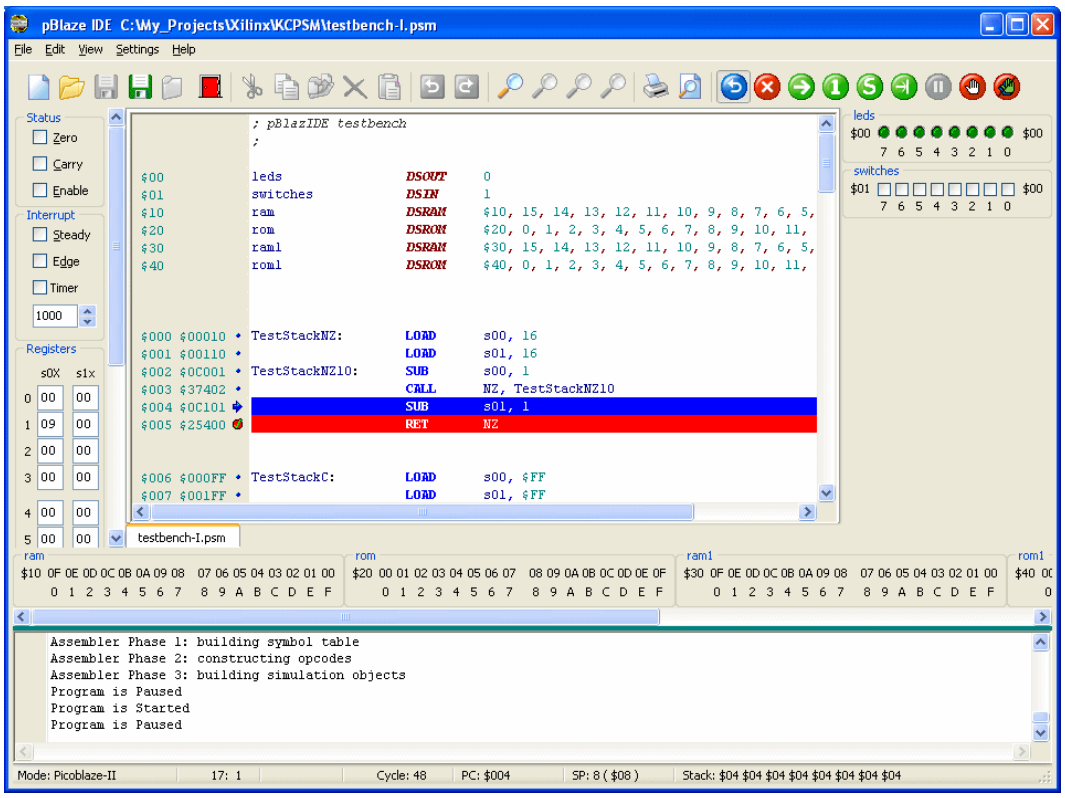
Sajnos 64 bites operációs rendszer alatt ez az eszköz sem működik, de a nagyobb baj az, hogy nem képes együttműködni a LOGSYS rendszerrel, így nem is tudtam kipróbálni a képességeit. Továbbá a párhuzamos portot használja alaphelyzetben a konfiguráláshoz, az USB-s kábel használatához újabb szoftvert kell használni, így megint legalább 3-ra nőtt a használandó alkalmazások száma, de legalább (elvileg) kikerülhető a lassú újraszintetizálás. (1)

2.2.4 Mediatronix pBlazIDE

A Mediatronix pBlazIDE a nevével ellentétben nem nevezhető igazi integrált fejlesztői környezetnek, hiszen a program eszközre töltését nem támogatja. Mégis nagy előrelépés a KCPSM3-hoz képest, hiszen nyújt egy kényelmes kódszerkesztőt a kód megírásához, egy fordítót és egy szimulátort. A szimulátornak köszönhetően a felhasználó alapszintű hibakeresést is tud végezni a programon.

A fejlesztés menete hasonló a KCPSM3-hoz, hiszen a pBlazIDE által generált VHDL vagy Verilog fájl hardver tervbe való illesztése és szintetizálása után tudjuk konfigurálni a processzort. (1) (12)

Az 1. ábra mutatja az alkalmazás képernyőképét működés közben.



1. ábra A Mediatronix pBlazIDE képernyőképe

2.2.5 kpicosim és openPICIDE

Ezen két termék funkcionalitását tekintve többé-kevésbé megegyezik a pBlazIDE-vel. A különbség az operációs rendszerben van, hiszen míg a pBlazIDE Windows alapú, addig a kpicosim és az openPICIDE Linux operációs rendszerekre íródott.

Mind a kettő nagyjából a következő szolgáltatásokat nyújtja: kódszerkesztő, fordító, szimulátor, perifériák kezelése, memóriadefiníciós fájl exportálása. A felületük is nagyon hasonlít Windowsos testvérükére. (13) (14)

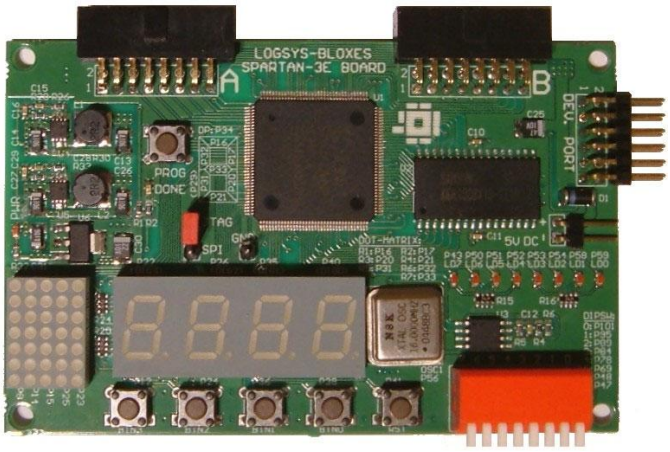
2.3 A LOGSYS rendszer

A LOGSYS rendszer több részből áll: a Spartan-3E FPGA kártyából, a fejlesztői kábelből és a következő szoftverekből:

- meghajtóprogramok,
- LOGSYS GUI,
- FLASH programozó,
- Xilinx USB letöltőkábel emulátor.

Ezek közül minket csak a Spartan-3E kártya, a fejlesztői kábel és a LOGSYS GUI érint, ezért a következőkben ezeket fogom röviden bemutatni.

2.3.1 Spartan-3E FPGA kártya



2. ábra A LOGSYS Spartan-3E FPGA kártya (15)

A kártya főleg kezdő felhasználók FPGA áramkörökkel való ismerkedését hivatott elősegíteni, emiatt a felépítése igen egyszerű. Ennek ellenére lehetőség van komplexebb tervek megvalósítására is a kártyán található chip képességeinek és a két bővítőcsatlakozónak köszönhetően. Az eszköz a 2. ábra látható.

Egy rövid lista a kártyán található elemekről:

- Xilinx XC3S250E-4TQ144C típusú FPGA.
- Memóriák a program és az adatok tárolására:
  - Egy 128k × 8 bites, 10 ns-os aszinkron SRAM
  - Egy 16 Mbites SPI buszos soros FLASH memória, ami konfigurációs memóriaként is szolgál az FPGA számára
- Megjelenítő eszközök:
  - 8 darab LED

- 4 digites hétszegmenses kijelző
- 7×5 pontmátrix kijelző

- Beviteli eszközök:
  - 5 darab nyomógomb
  - 8-as DIP kapcsoló
- Egy 16 MHz-es oszcillátor
- Csatlakozó a LOGSYS fejlesztői kábel számára
- 2 darab csatlakozó a kiegészítő moduloknak.

Ezek az eszközök bőségesen elegendőek a kezdő felhasználók számára, de sok bonyolultabb projekt is a kártya képességeinek csak egy részét használja ki.

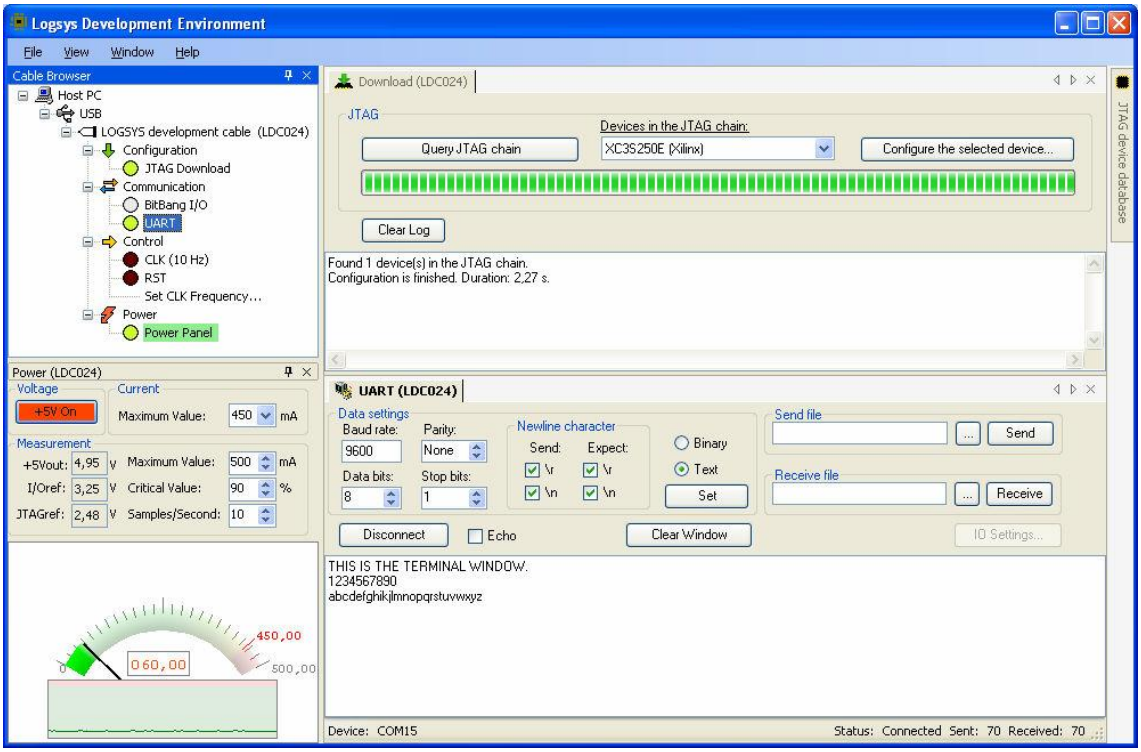
2.3.2 A fejlesztői kábel

3. ábra A LOGSYS fejlesztői kábel (7)

A fejlesztői kábel az USB porton keresztül kapcsolja össze a célrendszert a PC-vel. Biztosít konfigurációs interfészt, vezérlési interfészt (órajel és reset jel), soros kommunikációs interfészt és rendelkezik 5V-os tápfeszültség kimenettel is. Mivel a különböző rendszerek eltérő feszültségszinteket használhatnak a kommunikációhoz, ezért a fejlesztői kábel tartalmazza a szintillesztő áramköröket. Ennek köszönhetően a fejlesztői kábel sokféle célrendszerhez csatlakoztatható. (7)

Az eszköz képe a 3. ábra látható.

2.3.3 A LOGSYS GUI



4. ábra A LOGSYS GUI működés közben (7)

Az alkalmazói program jól áttekinthető és testreszabható felületen keresztül biztosítja a fejlesztői kábel funkcióinak elérését.

A programozható eszközök konfigurálásához a JTAG interfész áll rendelkezésre, mint natív konfigurációs interfész.

A LOGSYS rendszer a szabványos SVF fájlformátumot használja a JTAG láncon elvégzendő műveletek leírására. A legtöbb gyári fejlesztőrendszer lehetőséget biztosít a konfigurálást végző SVF fájl generálására. A Xilinx eszközök esetén az alkalmazás közvetlenül támogatja a BIT és a JEDEC fájlok használatát is.

A fejlesztői kábel többféle szinkron és aszinkron soros kommunikációs protokollt támogat. Alapvetően a célrendszerrel történő kommunikációra a népszerű UART használható. A virtuális soros port meghajtó elérhetővé teszi a fejlesztői kábel soros portját a windowsos alkalmazások számára. Az UART kommunikációhoz a LOGSYS környezetben megtalálható egy egyszerű terminál felület. (7)

A felhasználói felület a 4. ábra látható.

3 Előzmények

A 2.2 fejezetben áttekintést kaptunk a rendelkezésre álló fejlesztő eszközökről és azok tulajdonságairól. A legfájóbb a hibakeresés hiánya, így kiindulásként egy, a LOGSYS kártyához illeszkedő debugger kiegészítést készítettem a PicoBlaze-hez.

A tervezést a következő szempontok alapján végeztem:

- Legyen egyszerű,
- ne igényeljen speciális szoftvert a PC-n,
- a későbbiekben könnyen bővíthető legyen a funkcionalitása,
- valamint támogassa az alapvető hibakeresési műveleteket:
    - programszámláló értékének lekérése,
    - flagek értékének lekérése,
    - regiszterek olvasása/módosítása,
    - memória írása/olvasása,
    - I/O perifériák kezelése,
    - tetszőleges breakpoint beállítása futási időben,
    - a program futásának kézzel történő megállítása,
    - rendszer RESET kiváltása,
    - lépésenkénti futtatás.

Az egyszerűség jegyében minimális hardver kiegészítéssel akartam megvalósítani a debuggert, ezért egy IT szinten futó monitor programot választottam megoldás gyanánt. Ennek sajnálatos következménye volt, hogy a 16-ból csak 12 regisztert használhat a felhasználó, továbbá a PicoBlaze egyszintű megszakítás kezelése miatt az IT sem elérhető a felhasználó számára. Ezen felül természetesen a programmemóriából is elvesz némi helyet a monitor program. A legkevésbé jelentős megszorítás néhány perifériacím lefoglalása volt a hardveres kiegészítés számára: a 256 címből mindösszesen 14-et kellett erre a célra feláldozni.

Hogy ne kelljen speciális alkalmazás a hibakereséshez, ezért a kommunikációt UART protokollal oldottam meg, méghozzá úgy, hogy tetszőleges terminál programmal konzolos módban lehetővé váljon a használat. Ez egy újabb megszorítás a felhasználói programokra nézve, miszerint nem használhatják a fejlesztői kábel soros vonalát, az a debugger számára van fenntartva.

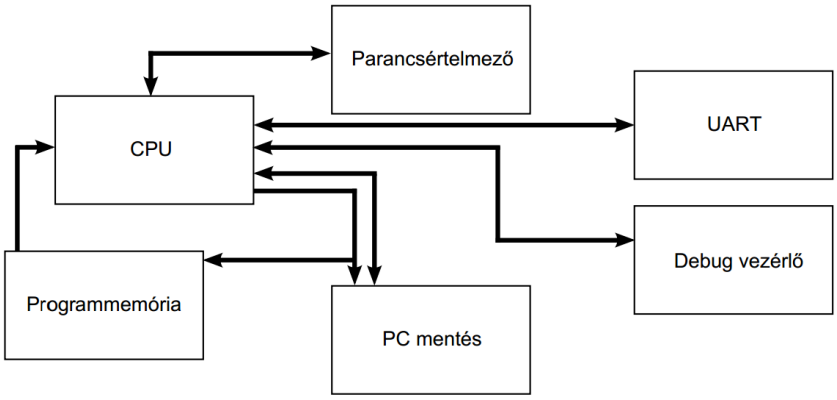
A bővíthető funkcionalitást két módon vettem figyelembe:

1. A monitor program strukturált kialakítása lehetővé teszi később az újabb funkciók utólagos beépítését.
2. Az UART kommunikáció formátuma ne csak a felhasználó számára legyen könnyen érthető, hanem egy grafikus felület számára is könnyen értelmezhető legyen, így növelve a monitor újrahasznosíthatóságát.

3.1 A debugger első verziója

3.1.1 Hardver

A hardver blokkdiagramja az 5. ábra látható.



5. ábra A hardver blokkdiagramja

Amik magyarázatra szorulnak ezek közül azok a parancsértelmező, a PC mentés és a debug vezérlő.

A parancsértelmező az UART-on érkező parancsok formátumát ellenőrzi. Azért hardveres megoldás mellett döntöttem, mert ez a funkcionalitás programozottan hosszú programkódot eredményezne, ami túlságosan lecsökkentené a felhasználói kód számára rendelkezésre álló memóriát.

A PC mentő egyszerűen a megszakítás fogadását jelző bit magas szintje esetén elmenti azt a memória címet, ahol éppen a felhasználói program áll.

A debug vezérlő feladatai közé tartozik a megfelelő pillanatban (RESET és bekapcsolás után, töréspont elérésekor) a megszakítás kiváltása, kérés esetén az eszköz RESET-elése, továbbá a felhasználói program és a monitor program közti átmenetek kezelése.

Természetesen a fenti blokkokon kívül a processzorhoz a kártyán megtalálható kimeneti és bemeneti perifériák is illesztve vannak, az áttekinthetőség kedvéért ezeket nem tüntettem fel.

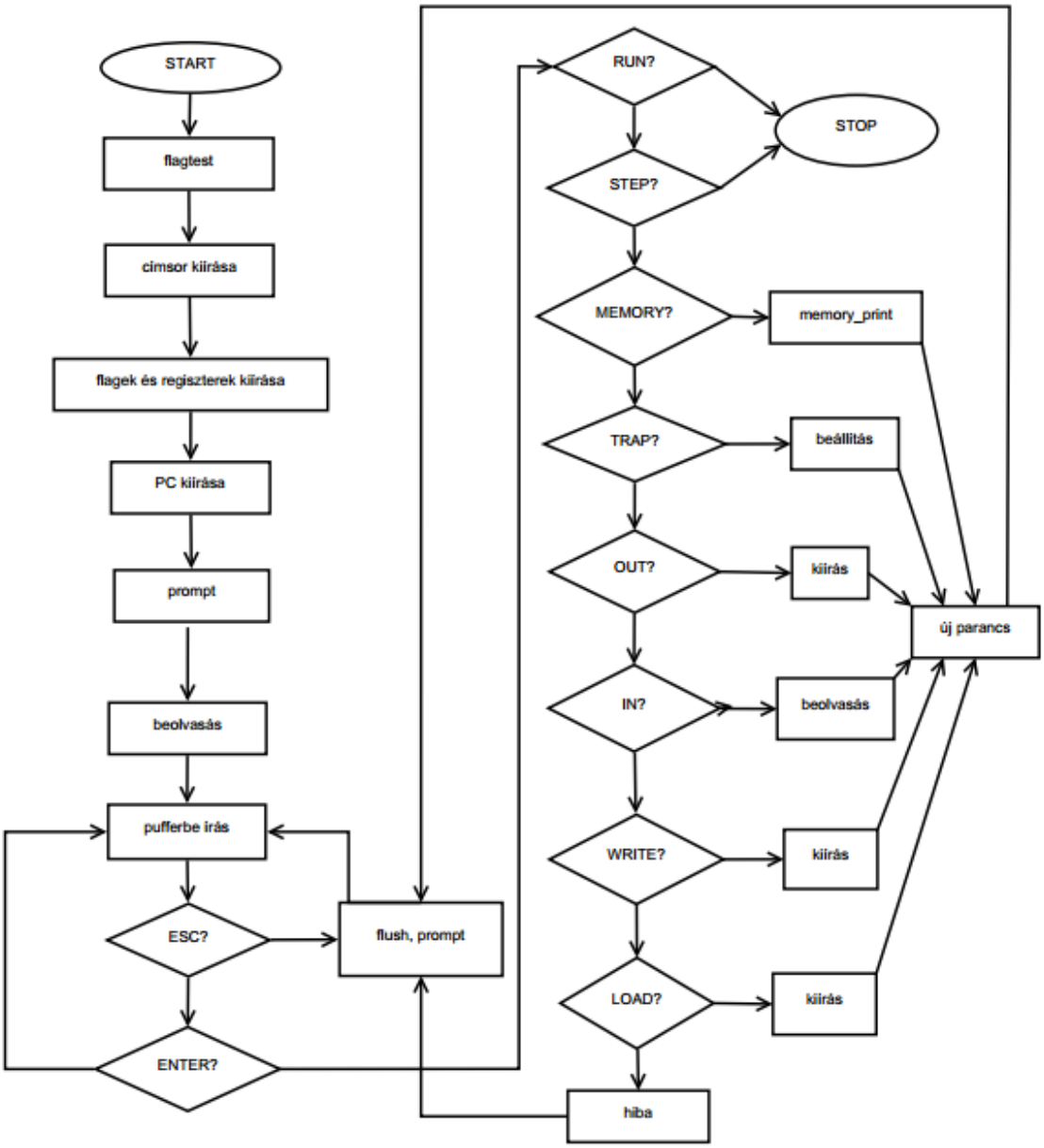
3.1.2 Szoftver

Az 1. táblázat tartalmazza a parancsok listáját, minden paraméter hexadecimális formában, két karakteren adandó meg. A processzor állapotának lekérésére (státuszbitek, programszámláló, regiszterek értéke) nincs külön parancs, ezeket az információkat a futás megszakításakor mindig kiírja a monitor. A parancsok nem kis-nagybetű érzékenyek.

1. táblázat A parancsok listája

Parancs	Formátum	Leírás
BREAK	B	Megszakítja a program futását, csak „free run” módban adható ki.
RESET	R	Alapállapotba állítja a hardvert, csak „free run” módban adható ki.
GO	G	Folytatja a program futását.
STEP	S	Végrehajtja a soron következő utasítást.
MEMORY	M	Kiírja a memória tartalmát.
WRITE	W XX YY	A memória XX címére az YY bájtot írja.
TRAP	T 0X XX	Beállít egy töréspontot az XXX címre.
IN	I XX	Beolvassa az XX című periféria értékét.
OUT	O XX YY	Az XX című perifériára az YY bájtot írja.
LOAD	L 0X YY	Az sX regiszterbe az YY bájtot írja.

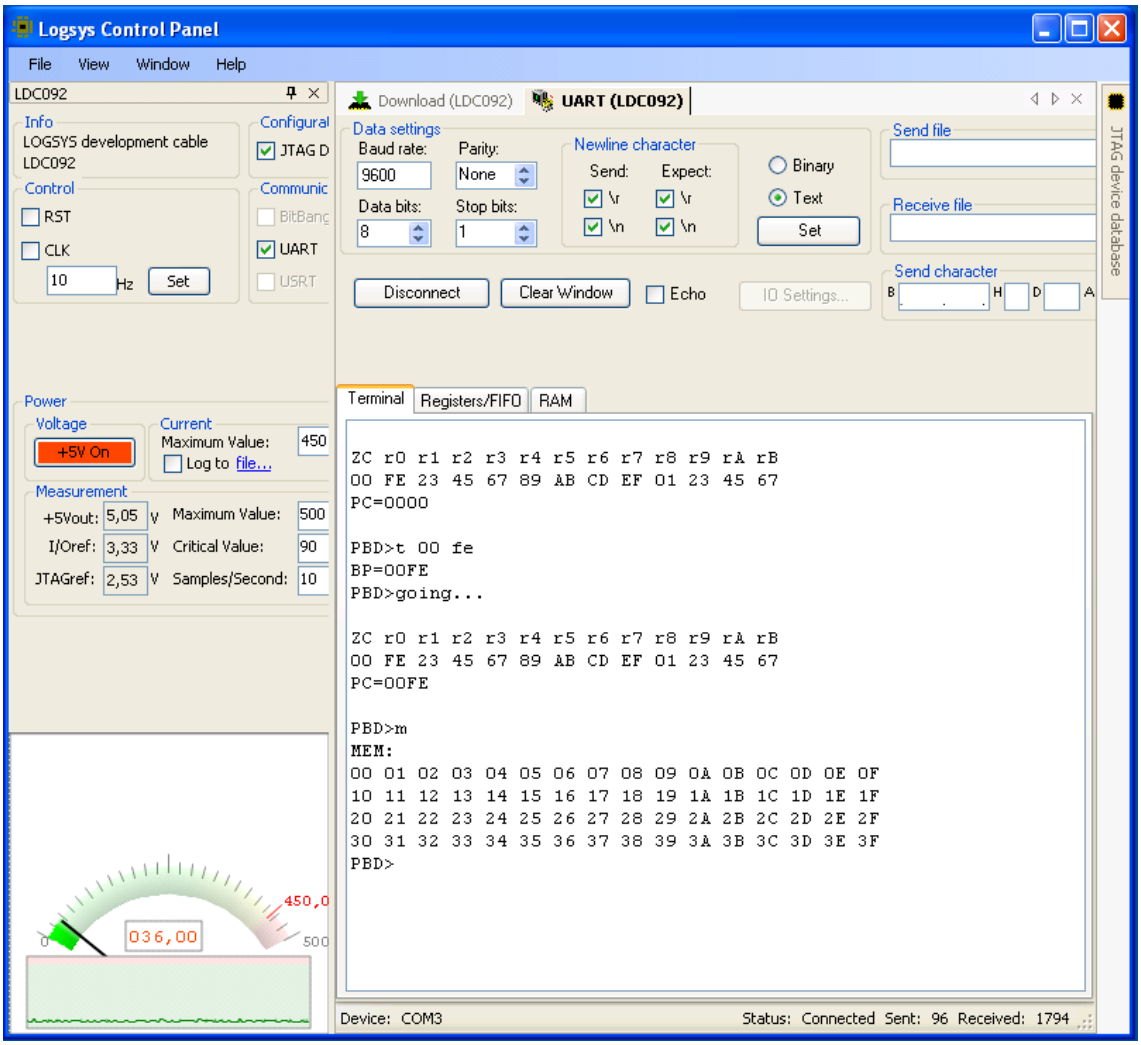
A 6. ábra látható a monitor program folyamatábrája.



6. ábra A monitor program folyamatábrája

3.1.3 Az első verzió értékelése

A 7. ábra látható az első verzió, amint a LOGSYS GUI felületén belül működik.



7. ábra A monitor működés közben

A célként kitűzött kritériumoknak kiválóan megfelelt az elkészült debugger, azonban a kezelése elég kényelmetlen több szempontból is. Egyrészt egy átlagos felhasználó nincs hozzászokva a konzolos hibakeresők használatához, másrészt ez még mindig nem küszöbölte ki a fejlesztés hosszadalmas és kényelmetlen folyamatát. Ezen felül a felhasználó csak úgy tudja nyomon követni a progamját, ha a KCPSM3 fordító által generált listafájlban követi, hogy melyik utasításhoz milyen cím tartozik a programmemóriában.

Ami szintén nagy érvágás, hogy az elkészült monitor program a programmemória több mint 1/3-át elfoglalja!

A legsúlyosabb probléma viszont csak később jelentkezett: a regiszterek és a státuszbitek nem voltak szinkronban. A PicoBlaze processzor nem támogatja a státuszbitek közvetlen elérését a programból, az értékük elérése csak feltételes elágazások segítségével lehetséges. Ezért, és a processzor belső kialakítása miatt a

monitor által jelzett értékek mindig úgy alakultak, hogy a státuszbitek a parancs lefuttatása utáni, minden más erőforrás (memória, regiszterek, perifériák) pedig az az előtti állapotot tükrözték. Ez lényegében lehetetlenné teszi az elágazások nyomon követését, hiszen a felhasználó a monitor programban bízva nem érti a program futása és a flagek állapota közti ellentmondást.

Összességében ez a monitor program jó kezdésnek tűnik mind önálló felhasználás, mind pedig egy grafikus felület és a hardver közötti közbenső réteg szempontjából. Az értékelés során átgondoltuk a továbbfejlesztési lehetőségeket. Alapvetően két út kínálkozott: az első, hogy egy teljesen hardveres megoldást készítünk, ami nem von el a felhasználói programtól erőforrásokat, cserébe az FPGA-ból használ fel több elemet. Így a felhasználó használhatná a teljes programmemóriát, az összes regisztert és a megszakításokat is.

A másik lehetőség a meglévő debuggert felhasználva egy integrált fejlesztői környezet készítése. Természetesen ehhez szükséges a státuszbitek és a többi erőforrás szinkronba hozása.

Felmerült még az UART helyett JTAG vonalon történő kommunikációra való áttérés, hogy a felhasználó a soros vonalat is használhassa az alkalmazásaiban.

Végül a második utat választottuk, a következőkben ennek a megvalósítását fogom bemutatni.

3.2 A második verzió

Az első verzió értékelésében (3.1.3 fejezet) beszéltem a státuszbitek nem konzisztens voltáról. Ezt a PicoBlaze CPU módosításával javítottam ki, mégpedig úgy, hogy kivezettem a processzorból a zero és carry flag-ek vezetékeit, majd azokat egy periférián keresztül, 1 órajel késleltetéssel értem el. Így a flag-ek értékének kiderítése már nem elágazások segítségével történik, hanem egy egyszerű perifériaolvasással, ráadásul szinkronban is van a többi erőforrással.

Egy másik hardveres változtatás az összes debugger specifikus hardver (PC mentő, Debug vezérlő, UART, Parancsértelmező) egy modulba csoportosítása volt, így a HDL kód átláthatóbb lett.

Ezen kívül még két apró módosítást eszközöltem a hardveren: a LOGSYS kártya RESET gombját kikötöttem a tervből, mert annak a megnyomását a grafikus felület nem

tudta volna a jelenlegi felállásban érzékelni, ami a felület lefagyásához vezethetett volna. A másik, hogy a kimeneti perifériák értékét meg szerettem volna jeleníteni a felhasználói felületen, ennek érdekében visszaolvashatóvá tettem őket.

Végül egy kicsit átstrukturáltam a monitor program kódját, aminek a célja főleg a hosszának a csökkentése volt. Ennek következtében sikerült több mint 25%-al csökkenteni a programmemóriából elfoglalt területet! A monitort érintő másik változtatás egy új parancs, a „P” bevezetése volt, ami kiírja a processzor állapotát (flag-ek, programszámláló, regiszterek értéke), ezzel párhuzamosan a monitor nem írja ki automatikusan a futás megszakítása után ezeket az adatokat. Erre azért volt szükség, hogy egy esetleges kommunikációs hiba miatt meg lehessen ismételni az utoljára kiadott parancsot, így például újra le lehessen kérni a processzor állapotát is.

3.3 A PicoBlaze IDE

A készített grafikus felülettel nem csak az volt a célom, hogy levegye a fejlesztő válláról a konzolos hibakeresés nyújtotta terhet, hanem egy olyan integrált fejlesztői környezetet szerettem volna a felhasználók kezébe adni, ami kiküszöböli az 2.2 fejezetben bemutatott eszközök hiányosságait. A tervezést ennek szellemében végeztem.

A projektben úgy gondoltam nagyobb lehetőségek rejlenek, mint ami egy félév munkájába belefér, ezért is tartottam fokozottan szem előtt egy továbbfejleszthető alkalmazás készítését. Ezt az objektumorientált tervezés lehetőségeinek kiaknázásával és az MVC (16) tervezési minta használatával kívántam elérni.

Az elsődleges eszközök, amikkel együtt kellett működni az a LOGSYS kártya és a LOGSYS fejlesztői kábel, ennek megfelelően ezen eszközök adottságaihoz igazítottam a tervet.

3.3.1 Interfészek

Az alkalmazás a korábban bemutatott monitor program 2. változatára épült, ennek megfelelően adott volt az UART mint kommunikációs interfész. Fogyatékosságai és szerény szolgáltatásai ellenére fordítónak a KCPSM3-at használtam az idő rövidsége miatt. Az is kritérium volt, hogy a lefordított programmal fel lehessen programozni az FPGA-t, így a LOGSYS fejlesztői kábellel is együtt kellett tudni működni.

Jól láthatóan ez három teljesen különböző interfész, ezért célszerű volt ezek kezelését külön-külön osztályokban implementálni. Hogy később egy másik fordítót, programozót vagy kommunikátort használhassunk, az alkalmazás vezérléséért felelős osztály csak egy jól definiált interfészét, indokolt esetben absztrakt ősosztályát ismerte ezen komponenseknek.

Hasonlóan a kezelendő perifériák listája is bármikor bővülhet, így itt is célszerű volt egy közös ősosztályt implementálni, amiből majd később származtatjuk az egyes perifériákat kezelő osztályokat.

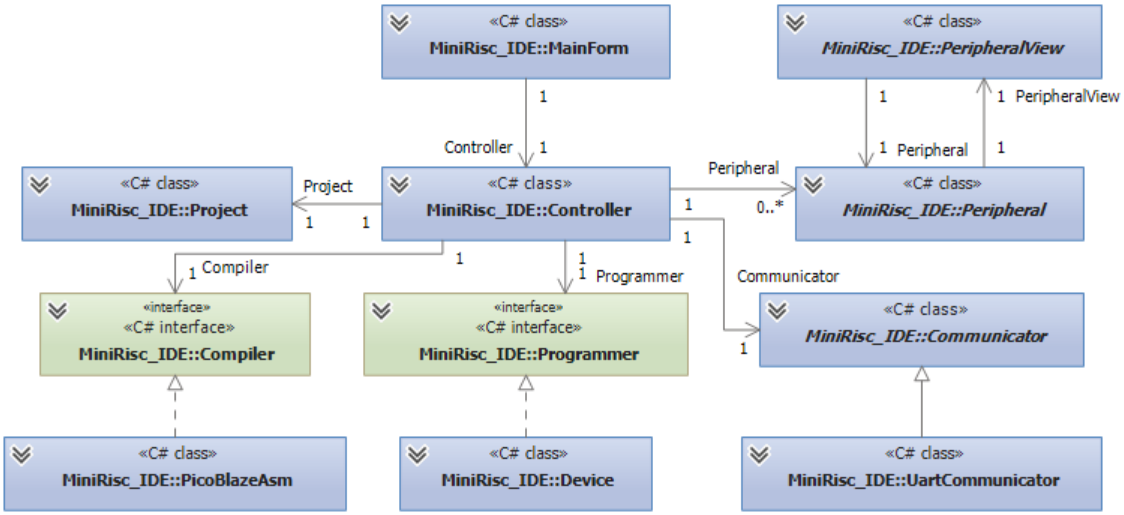
Az interfészek és absztrakt ősosztályok használatával megoldható ezen komponensek rugalmas cserélhetősége, extrém esetben akár futási időben is, ha a feladat azt kívánja. Egyelőre viszont minden komponensből csak egy volt elérhető, így ennek inkább csak a továbbfejlesztett verziók esetén lett jelentősége.

3.3.2 Platform

A fejlesztői kábel kezelése és az FPGA konfigurálása már implementálásra került a LOGSYS GUI-ban, így azokat a forráskódokat használtam. Mivel ezek C# (17) nyelven, .NET 4.0 platformon (18) vannak megírva, így egyértelműen adódott, hogy én is ezt a környezetet használtam. Ennek előnye, hogy a C# kényelmes és gyors fejlesztést tesz lehetővé, hátránya viszont, hogy kizárólag Windows operációs rendszerre érhető el.[3]

[3] Igazából elérhető egyéb rendszerekre is, de a szolgáltatásai és támogatottsága messze alacsonyabb, mint az szükséges volna, nem is beszélve a meghajtóprogramok hiányáról.

3.3.3 Az alkalmazás szerkezete

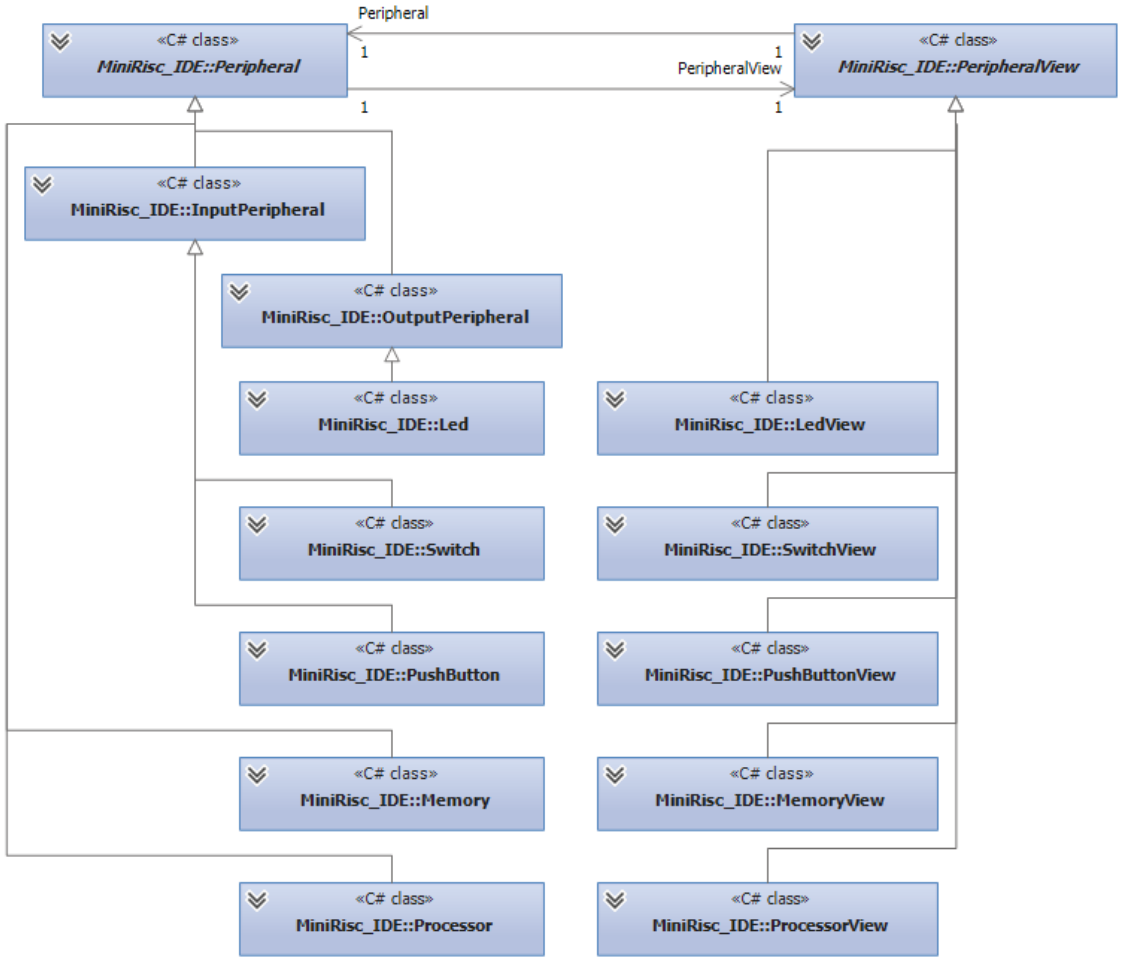


8. ábra A fontosabb osztályok áttekintése

A 8. ábra látható az alkalmazás főbb osztályait ábrázoló osztálydiagram. Megfigyelhető a 3.3 fejezetben említett MVC architektúra: a megjelenítés a `MainForm` és a `PeripheralView` osztályok leszármazottainak, a vezérlés a `Controller` osztály feladata, a modell pedig az összes többi osztályban kapott helyet.

A `MainForm` osztály semmiféle vezérlést nem végez, egyszerűen csak a `Controller`-től kapott adatokat megjeleníti és értesíti a felhasználói interakciókról, vagyis egyetlen felelőssége a megjelenítés. Így a későbbiekben könnyen lecserélhető egyéb felületre, extrém esetben akár webesre vagy konzolosra is az alkalmazás modelljének és vezérlőjének módosítása nélkül.

A `Peripheral` és `PeripheralView` osztályok azért lettek absztrakt osztályok interfész helyett, mert bizonyos alapvető funkciókat célszerű már ezen a szinten implementálni, így később biztosan nem felejtődik el. Ilyen funkció például, hogy a modell megváltozásakor automatikusan frissüljön a nézet. A 9. ábra áttekintést nyújt a perifériákat kezelő osztályokról.



9. ábra A perifériákat kezelő osztályok hierarchiája

Bevezettem két közbenső osztályt: az `InputPeripheral` és `OutputPeripheral` osztályokat. Ezek az általános I/O perifériák kezeléséért felelősek, a leszármazott osztálynak mindösszesen a megfelelő nézet, azaz `PeripheralView` osztály példányosítása és a perifériához tartozó cím(ek) megadása a feladata.

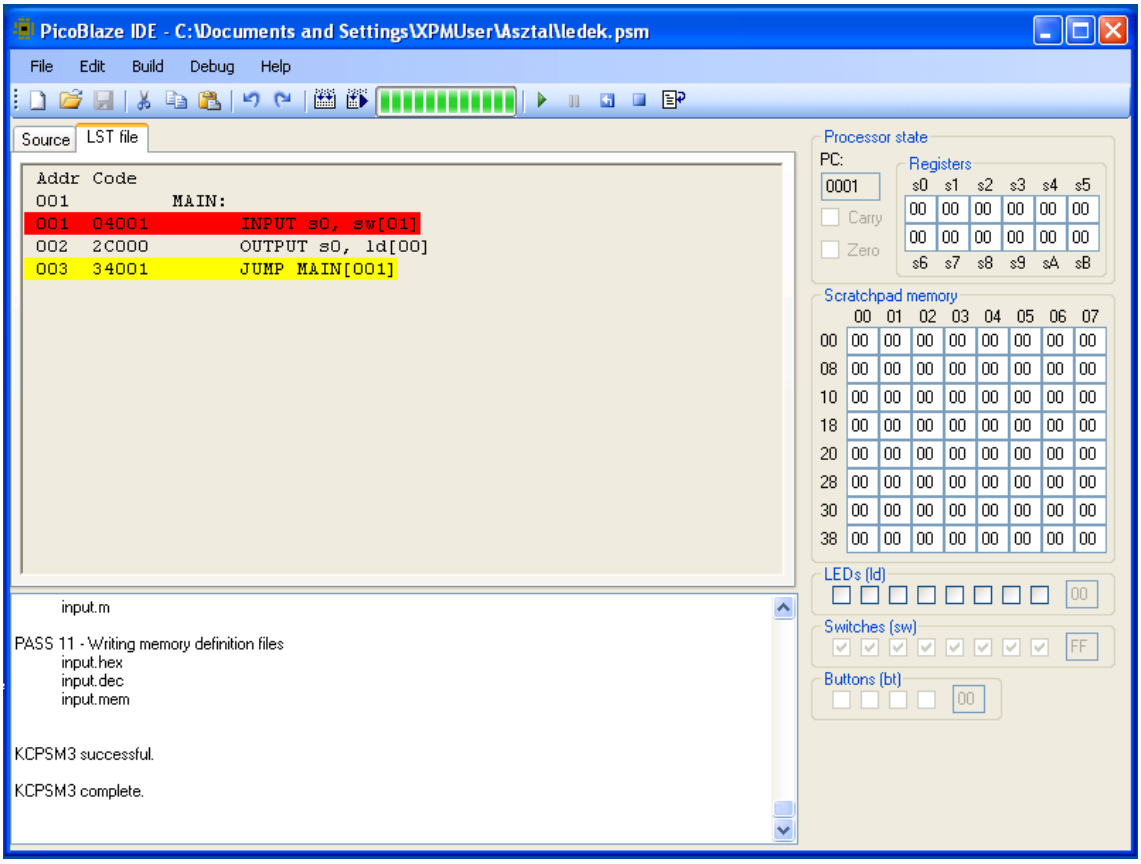
Ami még feltűnő az ábrán, hogy a memóriát és a processzort is perifériaként kezeli az alkalmazás. Ennek az az oka, hogy ezen a szinten logikailag semmi különbség nincs a processzor erőforrások és a perifériák kezelése között, hiszen a monitor program a különbségeket elfedi. Abból kiindulva, hogy ugyanolyan szerkezetű UART-os parancsokkal lehet ezen erőforrások állapotát lekérdezni és módosítani, az alkalmazás számára ezeket is perifériaként célszerű ábrázolni. Következésképpen a vezérlő osztály egyetlen heterogén kollekcióban tudja tárolni az összes erőforrás modelljét, ami az alkalmazás egyszerűsödése mellett a hibalehetőségeket is csökkenti.

A `Project` osztályról még nem ejtettem szót, ennek mindösszesen a forráskód karbantartása és a lemezen való kezelése a felelőssége.

3.3.4 A grafikus felület

A grafikus felület kialakítása során a piacon elterjedt alkalmazásokat tekintettem mintának. Így alakult ki a 10. ábra látható felépítés:

- Felül található a menüsáv és az eszközsor,
- alatta, bal oldalon a legnagyobb területet a szerkesztő terület foglalja el,
- ez alatt a fordító üzenetei láthatók,
- jobb oldalon pedig a perifériák állapota található.



10. ábra Az alkalmazás működés közben

A szerkesztő terület két részből áll, az egyikben a forráskód található („Source” fül), míg a másikban a listafájl („LST file” fül). A listafájl tartalmazza a fordítás során az egyes parancsok címének és kódjának feloldását. A programok futása során ez a terület aktív. Töréspontot itt lehet elhelyezni az utasítás során való dupla kattintással, amit a sor hátterének sárgára színeződése jelez (egyszerre csak egy aktív töréspont

lehet). Töréspont elérésekor az azon a címen lévő utasítás lefutása után a programvégrehajtás megáll, a vezérlés a monitornak adódik. Ekkor egy piros hátterű sor jelzi a következő futtatandó utasítást.

3.3.5 A fejlesztés lépései

Az elkészült alkalmazás kiváltja a 2.2 fejezetben említett összes alkalmazást, sőt: több tekintetben túl is tesz rajtuk. A fejlesztés menete a következő:

1. A felhasználó létrehoz vagy megnyit egy projektet (File → New/Open).
2. Szerkeszti a forráskódot, amit utána
3. lefordít (Build → Compile).
4. Elindítja az eszköz felprogramozását (Build → Download). Amennyiben még nem történt meg, az alkalmazás automatikusan csatlakozik a LOGSYS kártyához, bekapcsolja a tápfeszültséget, majd feltölti az alap hardverkonfigurációt. Ezt követően kerül sor a felhasználói program eszközbe töltésére.
5. Az első utasítás lefutása előtt megáll a program futása és vár a felhasználói parancsaira, aki ekkor elvégezheti a nyomkövetést (Debug menü parancsai, kivéve Stop).
6. A fejlesztő folytatja a 2-5. lépéseket, amíg el nem készült.
7. Leállítja a nyomkövetést, ezzel bontja a kapcsolatot a kártyával és kikapcsolja a tápfeszültséget (Debug → Stop).

Látszik, hogy a hardverterv újraszintetizálása kiesett a folyamatból, így takarítva meg időt a felhasználónak. Arról nem is beszélve, hogy így véletlenül sem okozhat hibát a hardvertervben.

Ami szintén kényelmes, hogy egyetlen eszköz használatát kell csak elsajátítania, ami ráadásul nagyon hasonlít a piacon lévő többi fejlesztőkörnyezethez, így akik már használtak valaha hasonló szoftvert, azoknak az átállás zökkenőmentes lesz. A kezdők dolgát pedig a konzolos felületnél sokkal barátságosabb grafikus interfész könnyíti meg.

3.3.6 Értékelés

Sikerült a célként kitűzött összes elvárásnak megfelelni, vagyis elkészült egy kényelmes és könnyen kezelhető integrált fejlesztőkörnyezet, ami nagyban elősegíti a PicoBlaze alapú rendszerek fejlesztését. Ennek köszönhetően a kezdő felhasználóknak sem fog elmenni a kedve a fejlesztéstől az első hibás program megírása után, hiszen könnyen és gyorsan megtalálhatják a hibát programjukban. Természetesen még jócskán vannak az eszköznek hiányosságai, például:

- Nem lehetséges megszakításokat tartalmazó program fejlesztése.
- A KCPSM3 fordító miatt csak és kizárólag 32 bites operációs rendszeren használható az eszköz. Ez a 64 bites architektúra miatt egyre fájóbb megszorítás, a fejlesztést nekem is virtuális gépen kellet végeznem.
- Az ASM nyelvű programozás mellett a legtöbb környezet valamilyen magasabb szintű nyelvet (általában C-t) is támogat.
- Az UART-on keresztül történő kommunikáció nem csak azért problémás, mert így a felhasználó csak a bővítő csatlakozók felhasználásával tud a számítógéppel ezen a protokollon kommunikálni, hanem fájóan lassú is, a lépésenkénti futtatásnál észrevehető válaszidő után kapja csak vissza a felhasználó a vezérlést.
- Egyelőre csak a legalapvetőbb három perifériát támogatja az alkalmazás (nyomógombok, kapcsolók, LED-ek), pedig a LOGSYS kártyán ennél sokkal több található (7 szegmenses kijelző, dot-matrix kijelző, SRAM stb.), az FPGA-ba implementálható egyéb perifériákról nem is beszélve.

Az első problémát a már említett teljesen hardveres megoldás oldaná meg. A többi hiányosság az alkalmazás felépítésének gondos tervezése következtében könnyen orvosolható:

- A KCPSM3 fordító helyettesíthető tetszőleges külső assemblerrel, ami konfigurálható utasításkészlettel rendelkezik, vagy akár készíthető egy, az alkalmazásba épített fordító is.
- Mivel a fordítónak csak egy interfészét látja a vezérlő, ezért az akár egy C fordító is lehet. Erre valószínűleg a legalkalmasabb a GCC lenne

széleskörű konfigurálhatóságának köszönhetően, de minden bizonnyal létezik a piacon más alternatíva is.

- Az UART kommunikáció kiváltható lenne JTAG kommunikációval, ami egyrészt gyorsabb és megbízhatóbb, másrészt a felhasználó számára elérhetővé válna a PC felé a kényelmes UART interfész. Ebben az esetben az alkalmazásba célszerű lenne beépíteni egy UART terminált, hogy ehhez se kelljen külön eszközt használnia a fejlesztőnek.
- A perifériák rugalmas kezelésének köszönhetően bármikor könnyedén beépíthető újabb perifériák támogatása a fejlesztőkörnyezetbe, így ennek a hiányosságnak a kiküszöbölése is csak minimális energiát igényel.

A fentieken túl további továbbfejlesztésre ad módot, hogy a programozókból csak egy interfészt, a kommunikátorokból pedig egy absztrakt ősosztályt lát a vezérlő osztály: a kettő egyszerre történő megvalósításával akár egy szimulátor is beépíthető az alkalmazásba, ezzel is tovább növelve a fejlesztést segítő eszközök számát.

4 MiniRISC IDE

Ebben a fejezetben először a MiniRISC processzor felépítésével, tulajdonságaival fogunk megismerkedni. Ezt követően bemutatom, hogy milyen lépéseket hajtottam végre a PicoBlaze IDE MiniRISC IDE-vé fejlesztése során.

4.1 A MiniRISC processzor

A MiniRISC egy nagyon egyszerű, 8 bites processzor, ami jól illeszkedik a LOGSYS Spartan-3E kártyához. Főbb tulajdonságai:

- Kis erőforrásigény
- Harvard architektúra
    - 256×16 bites programmemória
    - 256×8 bites adatmemória
- RISC jellegű utasításkészlet
    - Load/store architektúra
    - Műveletvégzés csak regisztereken
    - 16×8 bites belső regisztertömb
    - Adatmozgató utasítások
    - Aritmetikai utasítások (+, -, összehasonlítás)
    - Logikai utasítások (AND, OR, XOR, bittesztelés)
    - Léptetési, forgatási és csere utasítások
    - Programvezérlési utasítások
- Operandusok: két regiszter vagy egy regiszter és egy konstans
- Abszolút és regiszter indirekt címzési módok
- Zero (Z), carry (C), negative (N), overflow (V) feltételbitek
    - Feltételes ugró utasítások a teszteléshez
- Szubrutinhívás 16 szintű hardveres verem használatával

- Programelágazás a teljes címtartományban
- Egyszintű, egyszerű megszakításkezelés
- A perifériák memóriába ágyazottan érhetők el

A MiniRISC program- és adatmemóriája felkonfigurálható JTAG-en keresztül, így nem szükséges ezek módosításához újraszintetizálni a rendszert. Ezen kívül szintén a JTAG vonalon egy saját protokoll segítségével érhetőek el a tisztán hardveres debug mag, ami így nem von el a processzortól erőforrásokat, illetve gyorsabb lehet a kommunikáció.

A MiniRISC-hez Raikovich Tamás C#-ban implementált egy parancssoros fordító programot is, ami képes SVF fájl generálására. A program feltöltése a már korábban felkonfigurált FPGA-ra történhet a LOGSYS GUI segítségével. (8)

4.2 A MiniRISC mintarendszer

Mivel az elsődleges célhardver a LOGSYS Spartan-3E panel, ezért a mintarendszer kialakításánál ennek a képességei lettek számításba véve. A memória címtartományának alsó fele (128 bájt) adatmemória, míg a felső fele a periféria címtartomány.

A következő perifériákat tartalmazza a mintarendszer, zárójelben a báziscímmel:

- LED-ek (0x80)
- DIP kapcsolók (0x81)
- Nyomógombok (0x82)
- GPIO A (0x84)
- GPIO B (0x88)
- 8 bites timer megszakításkéréssel (0x8C)
- USRT megszakításkéréssel (0x8E)
- Hétszegmenses és pontmátrix kijelző (0x90)

(8)

4.3 A fejlesztőkörnyezet

A MiniRISC IDE több lépésben került kifejlesztésre a PicoBlaze IDE-ből. Az első lépés a fordító és a debugger maggal kapcsolatot tartó osztály lecserélése volt, ez pedig már egy működő alkalmazást eredményezett.

Ezután a felhasználói felületet kellett kicsinosítani. Mivel az új assembler debug információkat is képes visszaadni (például melyik programsor hányadik programmemória címhez tartozik, szimbóluminformációk), ezért a listafájlra már nem volt szükség és kivettem az alkalmazásból. A ScintillaNET (19) nevű kódszerkesztő komponensre cseréltem az eddigi változatot, hiszen ez kifejezetten forráskódokhoz lett elkészítve. Többek között a következő szolgáltatásokat nyújtja:

- Szintaxis kiemelés
- Sorok számozása
- Margón jelek elhelyezése
- Sorok színezése
- Felugró üzenetek (tooltip) megjelenítése
- Szavak aláhúzása különböző színekkel

A komponens minőségére jellemző, hogy például a MySQL Workbench is ezt használja az SQL lekérdezések szerkesztésére.

A felugró üzeneteket a következő dolgokra használtam fel:

- Az egérrel címkére mutatva felugrik a programmemóriában való címe
- Konstansra mutatva megjelenik annak értéke
- Hibára mutatva megjelenik a hibaüzenet
- Futás közben regiszterre mutatva megjelenik annak értéke
- Futás közben indirekt címre mutatva megjelenik a cím és ha az adatmemóriába tartozik, akkor a címen lévő érték

A felhasználói felület után a perifériák bővebb támogatása következett. A következő perifériák érhetők el és vezérelhetők a felhasználó felületről:

- LED-ek

- DIP kapcsolók
- Nyomógombok
- A két bővítőcsatlakozó (GPIO A és GPIO B)
- Hétszegmenses és pontmátrix kijelzők
    - Kattintással az egyes szegmensek ki- vagy bekapcsolása
    - Az adott szegmensen megjelenítendő bájt megadása
    - Az adott szegmensen megjelenítendő karakter megadása

A timer periféria azért nem kapott megjelenítést, mert az olvasás megváltoztathatja az állapotát, ami így kihatással lehet a program futására.

Ezen túl helyet kapott az alkalmazásban egy USRT terminál ablak is, ami a következő funkciókkal rendelkezik:

- Automatikus csatlakozás az eszközhöz
- Begépelt karakter elküldése, opcionális visszhang (echo) funkcióval
- Vett karakter kiírása
- Szöveges vagy bináris adatok kezelése
- Fájl küldése
- Vett adatok fájlba naplózása
- Terminál ablak törlése

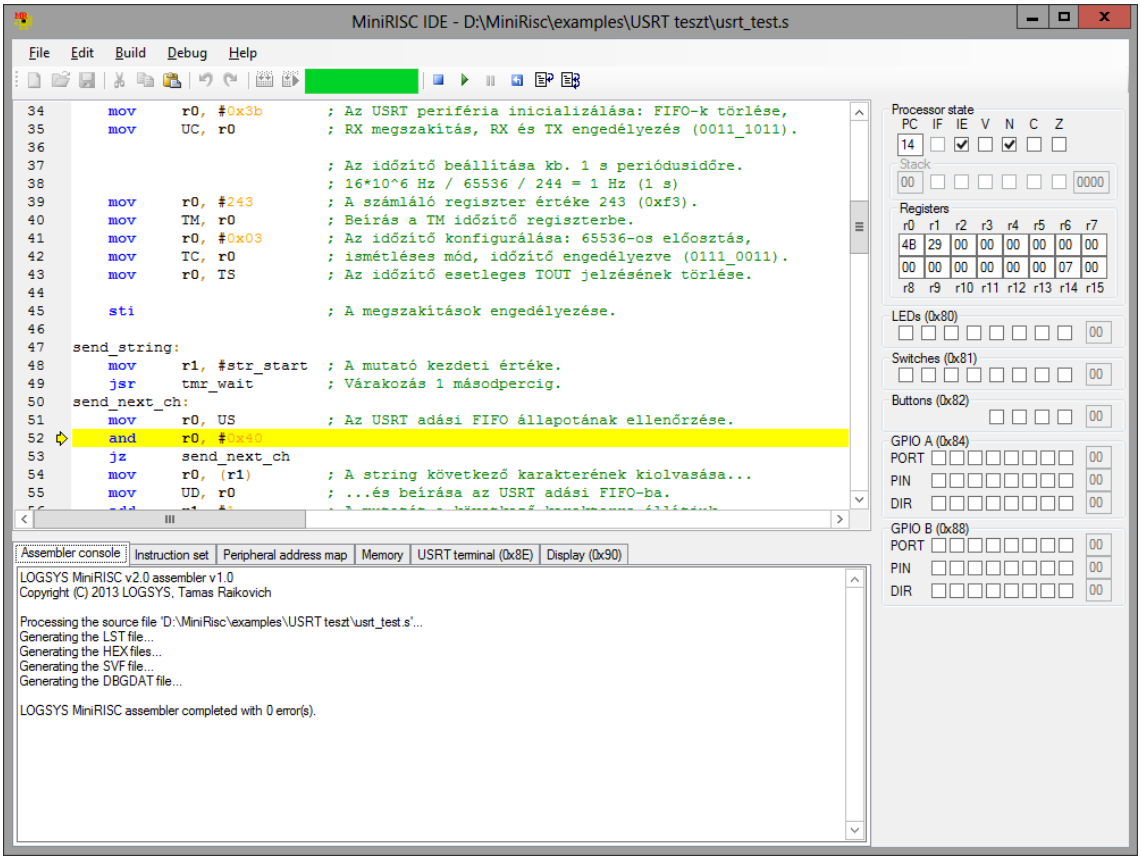
Mivel a memória itt nem 64, hanem 128 bájtos, ezért a jobb oldali sávról alulra került egy fülre. Ezt is kiegészítettem a fájlküldési és fájlba mentési funkcióval.

Végül a fejlesztést segítendő a Help menüben elhelyeztem egy linket a LOGSYS honlapra, valamint alul két új panellel bővítettem az alkalmazást: az elsőn egy összefoglaló látható az utasításkészletről, a második pedig a perifériák címkiosztását tartalmazza a báziscímekkel és az egyes regisztereik báziscímhez viszonyított címével valamint funkciójával.

Ezzel a C-fordítót és a szimulátort leszámítva megoldódott minden probléma, illetve megvalósult minden funkció, amit, a 3.3.6 fejezetben vetettem fel. Az előrelátó tervezésnek köszönhetően minden módosítást nagyon könnyű volt véghezvinni, hiszen

csupán a megfelelő működést megvalósító osztályt kellett lecserélni, vagy egy adott interfészt megvalósítva/ősosztályból leszármazva új osztályokkal kellett bővíteni a szoftvert.

Az elkészült felületet mutatja a 11. ábra.



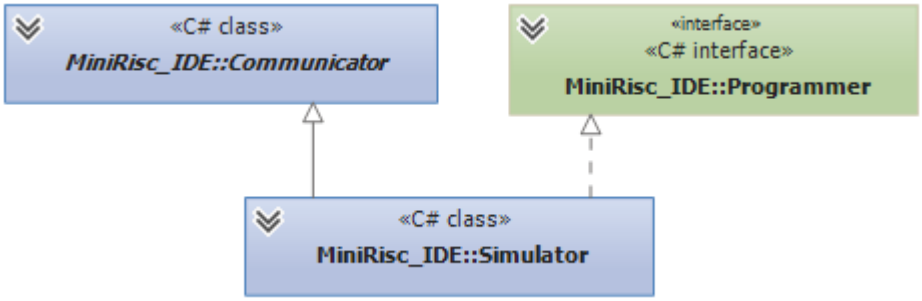
11. ábra A MiniRISC IDE felhasználói felülete

Ezek után úgy döntöttem, hogy a szimulátort is megvalósítom, hiszen ez egy nagyon hasznos eszköz azok számára, akik nem akarják, vagy nem tudják fizikai eszközön futtatni az alkalmazásukat, mégis meg szeretnének ismerkedni az assembly fejlesztés alapjaival.

4.4 Szimulátor

Ahogy a 3.3.6 fejezetben említettem ahhoz, hogy egy szimulátort építhessünk a fejlesztői környezetbe, csupán egy olyan osztályt kell megvalósítani, ami egyidejűleg implementálja a `Programmer` interfészt és leszármazik a `Communicator` absztrakt osztályból, amiben meg kell valósítani a teljes utasításkészlet, illetve a processzor egyéb tulajdonságainak szimulációját. Így az alkalmazás számára átlátszó módon lecserélhető

az FPGA panel a szimulátorra. A megvalósított statikus struktúra diagramot mutatja a 12. ábra.



12. ábra A szimulátor ősei

Az utasításkészlet szimulációjához a hardver működését igyekeztem lemásolni, hogy lehetőleg mindenben megegyezzen a viselkedésük. Ennek megfelelően egy utasítás végrehajtása 3 virtualizált órajelciklusig tart.

A szimulátor beépítve tartalmazza a regisztertömböt, a flag-eket, a stack-et, a programszámlálót, a stack pointert, a programmemóriát, az adatmemóriát és a töréspontokat tartalmazó adatstruktúrát. Az egyes tömbök mérete konstansokkal állítható, így a CPU egy későbbi módosítása esetén könnyebben lehet azt a szimulátorban is végrehajtani.

Az utasítások végrehajtását a fetch-decode-execute mintát követtem, amit egy metódusban valósítottam meg. Először a PC által meghatározott, aktuális utasítást lekérem, majd azt dekódolom. Nem kereső tábla alapján határozom meg egy-egy kód jelentését, hanem az egyes biteket értelmezve, hiszen így áll majd a szimulátor viselkedése a lehető legközelebb a tényleges CPU-jéhoz. Az egyes utasítás típusok (adatmozgató, logikai, aritmetikai stb.) végrehajtását szintén kiszerveztem egy-egy metódusba, ezzel növelve az átláthatóságot és csökkentve a komplexitást.

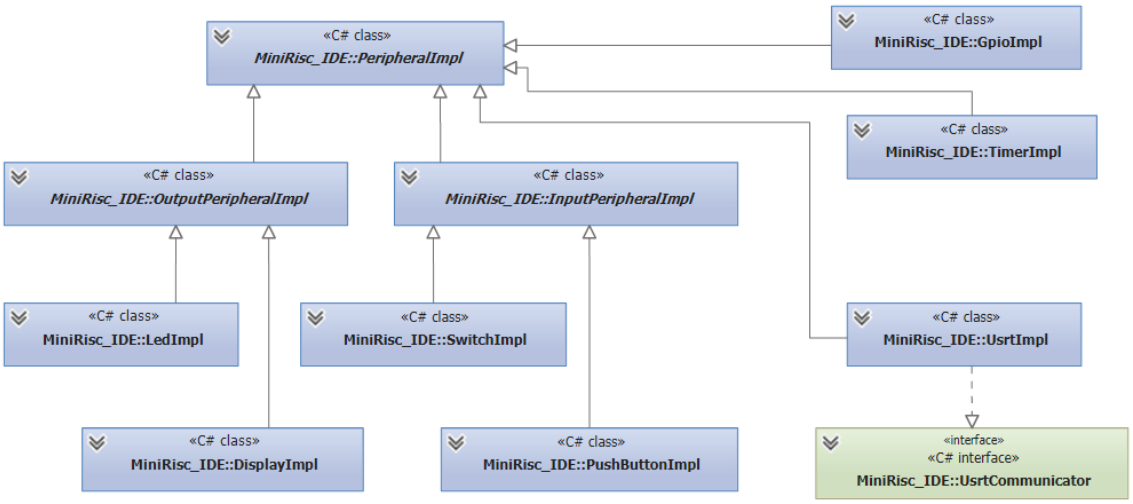
A tényleges végrehajtás egy külön szálon történik. Ebben a szálban egy állapotgép működik, aminek az állapotait rendszerint külső események változtatják meg (break, stop, run, reset). Ez alól kivétel egy töréspont elérése, ami egy belső esemény. A szálban az órajelek léptetése tovább zajlik a futás megszakadása esetén is, így a perifériák tovább működnek a háttérben (ahogy az a gyakorlatban is működik).

Ennek a szálnak a további felelőssége a perifériák értesítése az órajel váltásokról, a perifériák megszakítás kéréseinek összegyűjtése, valamint a megszakítás kiváltása.

A szál a szervezését tekintve egy végtelen ciklus, amibe be van építve egy lefutásonkénti 2ms-os késleltetés is. Ezt a 2ms-ot próbálkozással határoztam meg, hogy a processzorterhelés se legyen túl magas, illetve a szimulátoron a programok futása se legyen túl lassú.

Természetesen külön-külön minden perifériát is implementálni kell, hogy tökéletesen működjön a dolog, ezeket a szimulátor osztálynak kell ismernie és kezelnie. Ugyanazt a módszert használtam, mint a perifériák modelljének az implementálásakor: van egy absztrakt ősosztály (`PeripheralImpl`), amiből származik minden periféria implementáció. Ez az ősosztály bár bizonyos funkciókat megvalósít, mint például a regisztertömb lefoglalását. Ebből származtam az `InputPeripheralImpl` és `OutputPeripheralImpl` osztályokat, amik az egyszerű be- és kimeneti perifériákat valósítják meg, ahol csak beolvasni, illetve visszaolvashatóan kiírni kell. Az ezekből való származtatáskor már csak a regiszterek számát és a báziscímet kell megadni. Az `OutputPeripheralImpl` leszármazottai a `LedImpl` és `DisplayImpl`, az `InputPeripheralImpl` leszármazottai pedig a `SwitchImpl` és `PushButtonImpl` osztályok. A többi periféria (`GpioImpl`, `TimerImpl` és `UsrtImpl`) közvetlenül a `PeripheralImpl` leszármazottai, hiszen ezekben az esetekben összetettebb funkcionalitásról van szó, nem csak egyszerű írásról és olvasásról. A megvalósított osztálydiagramot mutatja a 13. ábra.

A perifériák saját maguk kezelik a regisztereiket (ahogy a valóságban is), így a szimulátorban csupán 128 bájtnyi memória található. Hogy egy memóriaműveletet melyik periféria fog kiszolgálni, azt belül fogja eldönteni az `PeripheralImpl` osztályban megvalósított osztály. Itt a báziscím ismeretében meghatározásra kerül, hogy melyik relatív címhez tartozik az adott írási vagy olvasási művelet, ezután pedig ellenőrzi, hogy ez a relatív cím beletartozik-e a periféria címtartományába. Amennyiben igen, végrehajtja a kért írási vagy olvasási műveletet.



13. ábra A perifériák szimulált változatának statikus struktúrája

5 Összefoglalás

A legelső, konzolos változattól hosszú út vezetett a szimulátorral is ellátott, minden perifériát támogató, komplett fejlesztőkörnyezetig. Azonban mint minden szoftvert, ezt is alapos tesztelésnek kell még alávetni, főleg olyan tekintetben, hogy az utasításkészlet ugyanúgy viselkedik-e, mint az eredeti rendszerben, illetve az egyes perifériák (különösen az USRT és a timer) megfelelően működnek-e.

Amennyiben minden hiba kijavításra került ismét számba kell benni a továbbfejlesztési lehetőségeket:

- Új perifériák implementálása, például a panelra épített SRAM
- Magasabb szintű nyelvek támogatása (például C). Természetesen mivel az erőforrások nagyon csekélyek, ezért meg kell vizsgálni, hogy van-e értelme egyáltalán ezen funkció beépítésének. Ez nagyban függ attól is, hogy a fordító mennyire hatékony kódot generál.
- Egyedi hardverterv feltöltése az FPGA-ra
- A felhasználó saját perifériáinak támogatása. Ennek a megoldása nem triviális, hiszen lehetőséget kell biztosítani legalább a periféria címtartományába eső regiszterek írására és olvasására, de jobb lenne valamilyen egyedi, grafikus felület támogatása. Ezt a felületet és a szimulátor számára a periféria működését valamilyen script nyelv segítségével lehetne megadni.

Úgy gondolom, hogy sikerült egy jól használható eszközt létrehozni, aminek a további fejlesztéseit legalább akkora lendülettel fogom végezni, mint eddig.

Irodalomjegyzék

1. **Xilinx.** PicoBlaze 8-bit Microcontroller. [Online] [Hivatkozva: 2013. október 24.] http://www.xilinx.com/products/intellectual-property/picoblaze.htm.

2. —. MicroBlaze Soft Processor Core. [Online] [Hivatkozva: 2013. október 24.] http://www.xilinx.com/tools/microblaze.htm.

3. **Lattice.** Lattice Mico8 Open, Free Soft Microcontroller. [Online] [Hivatkozva: 2013. október 24.] http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx.

4. —. LatticeMico System Development Tools. [Online] [Hivatkozva: 2013. október 24.] http://www.latticesemi.com/Products/DesignSoftwareAndIP/EmbeddedDesignSoftware/LatticeMicoSystem.aspx.

5. **Altera.** Nios II Processor: The World's Most Versatile Embedded Processor. [Online] [Hivatkozva: 2013. október 24.] http://www.altera.com/devices/processor/nios2/ni2-index.html.

6. OpenCores.org. [Online] [Hivatkozva: 2013. október 24.] http://opencores.org/projects.

7. **Fehér Béla, Raikovich Tamás, Laczkó Péter.** LOGSYS - Beágyazott rendszerek fejlesztői környezete. [Online] [Hivatkozva: 2013. október 24.] http://logsys.mit.bme.hu/sites/default/files/page/2009/09/logsys_beagyazott_rendszerek_fejlesztoi_kornyezete.pdf.

8. **Fehér Béla, Raikovich Tamás, Fejér Attila.** A MiniRISC processzor. [Online] [Hivatkozva: 2013. október 24.] http://home.mit.bme.hu/~rtamas/MiniRISC/MiniRISC_CPU.pdf.

9. **Attila, Fejér.** Fejlesztőkörnyezet kialakítása PicoBlaze processzorhoz. *Szakdolgozat.* 2011.

10. **Chapman, Ken.** KCPSM3. [Online] [Hivatkozva: 2013. október 24.] http://www.eng.auburn.edu/~strouce/class/elec4200/KCPSM3_Manual.pdf.

11. **Kocik, Pablo Bleyer.** PacoBlaze. [Online] [Hivatkozva: 2013. október 24.] http://bleyer.org/pacoblaze/.

12. **Mediatronix.** pBlazIDE. [Online] [Hivatkozva: 2013. október 24.] http://www.mediatronix.org/pages/pBlazIDE.

13. **Smith, George.** kpicosim. [Online] [Hivatkozva: 2013. október 24.] http://marksix.home.xs4all.nl/kpicosim.html.

14. **Fauck, Christoph.** openpicide. [Online] [Hivatkozva: 2013. október 24.] http://www.openpicide.org/content/about/.

15. **LOGSYS.** Logsys spartan-3e fpga kártya felhasználói Útmutató. [Online] [Hivatkozva: 2013. október 24.] http://logsys.mit.bme.hu/sites/default/files/page/2009/09/LOGSYS_SP3E_FPGA_Board.pdf.

16. *Design Patterns: Elements of Reusable Object-Oriented Software.* **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** USA : Addison-Wesley, 1994.

17. **Microsoft.** Visual C#. [Online] [Hivatkozva: 2013. október 24.] http://msdn.microsoft.com/en-us/library/vstudio/kx37x362.aspx.

18. —. .NET Framework 4. [Online] [Hivatkozva: 2013. október 24.] http://msdn.microsoft.com/en-us/library/w0x726c2(v=vs.100).aspx.

19. ScintillaNET. [Online] [Hivatkozva: 2013. október 24.] http://scintillanet.codeplex.com/.

6 Ábrák jegyzéke

7 Táblázatok jegyzéke