VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULÁTOR A DEBUGGER PROCESORU PICOBLAZE

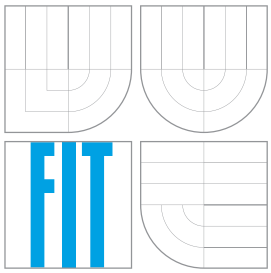
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL VAMPOLA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SIMULÁTOR A DEBUGGER PROCESORU PICOBLAZE

PICOBLAZE SIMULATOR AND DEBUGGER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL VAMPOLA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VAŠÍČEK ZDENĚK, Ph.D.

BRNO 2013

Abstrakt

Tato práce se zabývá procesorem PicoBlaze a možnostmi jeho simulace a ladění. Představí čtenáři strukturu procesoru, jeho instrukční sadu a existující vývojové nástroje. Popisuje návrh a implementaci pluginu pro QDevKit, který umožňuje simulaci a krokování procesoru na počítači i na přípravku FITkit.

Abstract

This thesis describes processor PicoBlaze and his possibilities of simulation and debugging. Structure, intruction set and existing developing tools of the processor will be presented to the reader. The thesis describes design and implementation of plugin for QDevKit, which allows simulating and stepping the procesor on personal computer or FITkit.

Klíčová slova

simulátor, debugger, PicoBlaze, assembler, QDevKit, plugin, FITkit, Qt

Keywords

simulator, debugger, PicoBlaze, assembler, QDevKit, plugin, FITkit, Qt

Citace

Pavel Vampola: Simulátor a debugger procesoru PicoBlaze, bakalářská práce, Brno, FIT VUT v Brně, 2013

Simulátor a debugger procesoru PicoBlaze

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Vašíčka, Ph.D. Uvedl jsem všechny literární prameny publikace, ze kterých jsem čerpal.

.....
Pavel Vampola
13. května 2013

Poděkování

Děkuji vedoucímu práce Ing. Zdeňku Vašíčkovi, Ph.D. za odbornou pomoc a konzultace, díky kterým byla tato práce úspěšně dokončena.

Obsah

Kapitola 1

Úvod

S rozvojem programovatelných hradlových polí rostly i možnosti jejich použití. Začaly se vyvíjet soft-core procesory, které jsou popsány výhradně v softwaru a mohou být syntetizovány do hardwaru, například do FPGA. Hlavní výhodou těchto procesorů je jednodušší implementace stavových automatů a podobných algoritmů, další nesporná výhoda je rekonfigurovatelnost na míru vyvíjené aplikace.

Mezi jeden z nejpoužívanějších soft-core procesorů patří procesor PicoBlaze, jehož vzrůstající popularita dala vzniknou mnoha nástrojům usnadňující vývoj a tvorbu aplikací pro něj. Avšak i přes snahu vytvářet nové vývojové prostředky pro procesor PicoBlaze, neexistuje žádný profesionální nástroj, který by splnil požadavky na skutečně kvalitní vývojové prostředí.

Cílem mé práce je vyvinout simulátor a debugger procesoru PicoBlaze, který umožňuje tvorbu kódu, jeho simulaci, krokování a možnost kód nahrát do přípravku FITkit, kde je umožněno komunikovat s periferiemi jako je LCD displej nebo klávesnice.

Tato práce má za úkol přiblížit čtenáři studii a vývoj simulátoru pro procesor PicoBlaze. Struktura procesoru a jeho vlastnosti budou probrány v druhé kapitole. Popis současných vývojových nástrojů a jejích možností je probírán v třetí kapitole. Návrh řešení je nastíněn ve čtvrté kapitole. Implementace celé aplikace a její vzhled je představen v páté kapitole.

Kapitola 2

Procesor PicoBlaze

Picoblaze je 8bitový procesor RISC. Protože se jedná o soft-core procesor, je zcela implementován v poli FPGA. Proto se dá funkcionalita lehce rozšiřovat pomocí připojení dalších FPGA komponent na vstupně/výstupní porty mikrokontoléru.

Veškerý popis a vlastnosti procesoru jsou popsány v manuálu [2] a uživatelské příručce [6], ze kterých budu v této kapitole vycházet.

Časté použití tohoto procesoru je v algoritmických úlohách, u nichž není kritické časové hledisko. Vhodné je pak řešení, ve kterých se řídí procesy, kde tento procesor nahradí stavový automat.

Protože PicoBlaze není klasickým obvodem na plošném spoji, jak to bývá u jiných procesorů, ale programuje se do FPGA, je rychlé a levné případné chyby v návrhu opravit nahráním nové konfigurace FPGA.

Velkou výhodou procesoru PicoBlaze je jeho velikost. Díky tomu, že je optimalizovaný pro FPGA rodiny Xilinx zabírá na obvodu Spartan-3 pouze 96 SLICE[1] což odpovídá 12,5% z verze XC3S50, 7,5% z XC3S400 a méně než 0,3% z XC3S5000.

Procesor PicoBlaze má několik verzí. V této práci se vždy bude psát o verzi KCPSM3, protože tato verze je optimalizována pro Spartan-3, jehož obvod je na FITkitu.

2.1 Architektura procesoru PicoBlaze

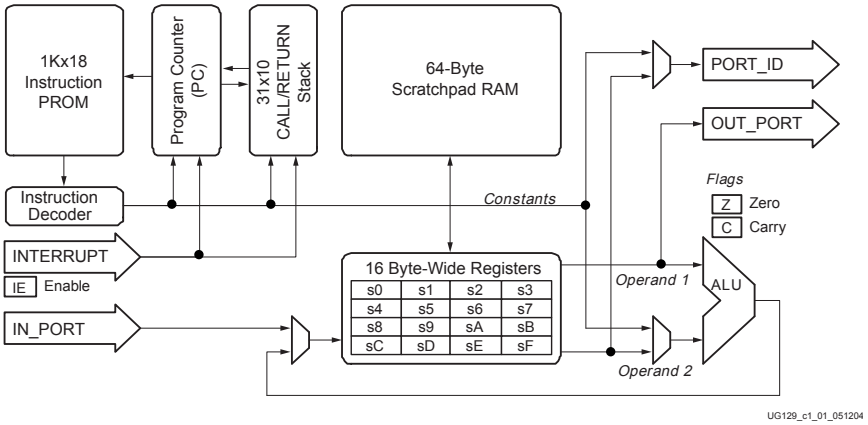
Na obrázku 2.1 je vidět architektura procesoru a jednotlivé logické bloky procesoru, které budu postupně vysvětlovat v této kapitole.

Procesor obsahuje registrové pole čítající *16 registrů pro obecné použití*, tyto registry jsou pojmenovány `s0, s1...sF`. Pro lepší srozumitelnost kódu lze registry přejmenovat direktivou `EQU`. Žádný z registrů nemá speciální funkci, jako jsou například akumulátor nebo ukazatel do paměti atd. Všechny výpočty jsou tedy ukládány jenom do registru určeného programátorem.

Další důležitou součástí procesoru je *aritmeticko-logická jednotka (ALU)*, která provádí všechny výpočty procesoru. ALU při výpočtech používá až dva operandy, kde do prvního ukládá i výsledek. Druhý operand může být registr, nebo konstanta. ALU nastavuje příznaky `ZERO` a `CARRY` podle výsledku dané operace. Nastavení příznaků bude podrobněji popsáno v kapitole 2.2.

Pro uchování dat je kromě registrů dostupná *datová paměť(scratchpad)*, která uchovává až 64 bytů dat. K datům se přistupuje pomocí instrukcí `STORE` a `FETCH`. Adresa místa

[1] Nejmenší logický blok uvnitř FPGA logiky[3].



Obrázek 2.1: Architektura procesoru PicoBlaze (převzato z uživatelské příručky [6])

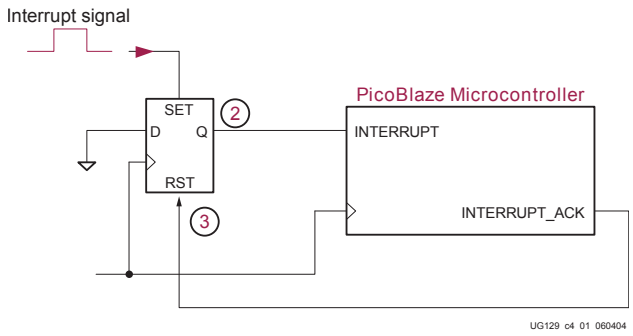
v paměti je určena registrem, nebo konstantou. Protože je paměť jenom 64 bytová, je adresována vždy šesti nejnižšími byty adresovacího operandu.

Vstupně/výstupní porty umožňují rozšířit funkčnost mikrokontroléru připojením dalších periférií, se kterými procesor komunikuje pomocí instrukcí `INPUT` a `OUTPUT`. Tyto instrukce mají dva operandy, přičemž první je zdrojový/cílový registr a druhý operand označuje adresu portu. Po vykonání instrukce `INPUT` se na výstup `PORT_ID` dostane adresa periferie a ze vstupu `IN_PORT` se data uloží do cílového registru. V případě instrukce `OUTPUT`, se periferie adresuje stejně, ale data ze zdrojového registru se pošlou na výstup `OUT_PORT`.

Pro adresování instrukcí v paměti se používá 10bitový *programový čítač*. K hodnotě čítače nelze v kódu přistoupit ani ji měnit. Při vykonání instrukce se čítač inkrementuje, tak aby ukazoval na další instrukci, která je umístěna v paměti. Pouze instrukce skoku(`JUMP`), volání rutin(`CALL`), návratu(`RETURN, RETURNI`) a reset mohou měnit hodnotu čítače neinkrementálně.

Aby bylo možné v kódu volat rutiny obsahuje procesor *návratový zásobník* o velikosti až 31 adres, kde se ukládá obsah programového čítače při volání rutiny, což umožňuje používat až 31 vnořených rutin. Při volání rutiny instrukcí `CALL` nebo jejími variantami se na zásobník uloží aktuální adresa inkrementována o jedničku zároveň s příznaky `ZERO` a `CARRY`. Pro návrat z rutiny se používá instrukce `RETURN`, která obnoví hodnotu programového čítače z návratového zásobníku.

Obsluha přerušení je vyvolána jako reakce vnější signál připojený na vstup `INTERRUPT`. Procesor PicoBlaze má pouze jedno přerušení, pokud aplikace potřebuje vyvolávat přerušení z více zdrojů je potřeba vytvořit odpovídající mechanismus v FPGA logice. Při detekci žádosti o přerušení se vnutí instrukce `CALL 3FF`. Na adrese `3FF` je očekávána rutina, která přerušení obslouží a poté ukončí obsluhu instrukcí `RETURNI`. Procesor také disponuje signálem `INTERRUPT_ACK`, který potvrzuje vyvolání obsluhy přerušení. Tento signál může být použit i pro zpětnou vazbu periferním zařízením nebo pro vynulování externího přerušení viz obrázek 2.2.



Obrázek 2.2: Ukázka využití signálu `INTERRUPT_ACK` (převzato z uživatelské příručky [6])

2.2 Instrukční sada

Instrukční sada procesoru PicoBlaze se skládá z 57 instrukcí. Podrobný popis instrukcí lze najít v manuálu [2].

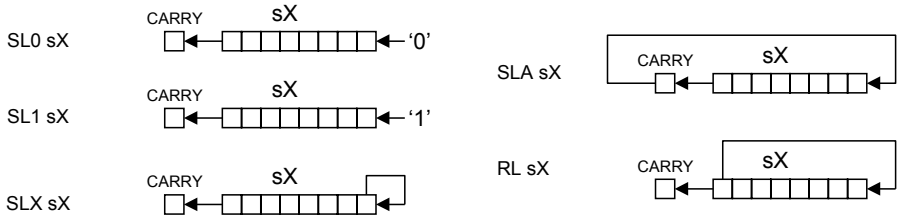
Instrukční sada se dá rozdělit do několika sekcí, které budu v následujícím textu rozebírat.

Logické operace - logický součet (`OR`), součin (`AND`) a výlučný součet (`XOR`). Tyto instrukce mají dva operandy, prvním je cílový registr, kde bude výsledek uložen, druhý je registr nebo konstanta, se kterou se bude operace prováděna. V případě, že je po provedení operace výsledek nula, je nastaven příznak `ZERO`, příznak `CARRY` je vždy vynulován.

Aritmetické operace - sčítání (`ADD`, `ADDCY`) a odečítání (`SUB`, `SUBCY`). Syntaxe těchto operací je stejná jako u logických operací, ale příznak `CARRY` je nastaven, pokud operace způsobila přetečení. Instrukce `ADDCY` a `SUBCY` ve výpočtu ještě přičtou/odečtou příznak `CARRY`, pokud je nastaven.

Porovnávací operace - aritmetické porovnání (`COMPARE`) a testy na shodu (`TEST`). Instrukce `COMPARE` porovnává dva operandy a to buď dva registry nebo registr a konstantu. Při této operaci se nemění hodnoty v registrech, ale pouze se nastavují příznaky. Příznak `ZERO` se nastaví v případě rovnosti. Příznak `CARRY` je nastaven pokud druhý operand je větší než první. Instrukce `TEST` provádí logický součin nad operandy, ale výsledek se neukládá do žádného registru. Pokud jsou operandy shodné, je nastaven příznak `ZERO`. A příznak `CARRY` indikuje lichou paritu ve výsledku logického součinu.

Operace bitového posunu a rotace - instrukce posouvající bity registru vpravo (`SR0, SR1, SRX, SRA, RR`) a vlevo (`SL0, SL1, SLX, SLA, RL`). Při posunu doleva je MSB bit uložen do příznaku `CARRY` a na místo bitu LSB je nastaven bit podle varianty použité instrukce, jak lze vidět na obrázku 2.3. Analogicky fungují i instrukce posunu vpravo.



Obrázek 2.3: Instrukce posunu(převzato z manuálu [2])

Operace pro komunikaci s periferiemi - instrukce pro zápis (`OUTPUT`) a čtení (`INPUT`) dat. Tyto instrukce zapisují/čtou data do/z registru určeného prvním operandem, přičemž adresa periferie je dána druhým operandem.

Operace pro práci s datovou pamětí - instrukce `STORE` uloží data registru daného prvním operandem do paměti a instrukce `FETCH` načte data z paměti do cílového registru. Adresovaná paměť je určena šesti bity druhého operandu, kterým může být registr nebo konstanta.

Operace obsluhy přerušení - instrukce pro povolení/zakázání přerušení (`ENABLE/DISABLE INTERRUPT`) a návratu z přerušení (`RETURNI ENABLE/DISABLE`). Instrukce `RETURNI` je modifikací instrukce `RETURN`, která nastaví programový čítač na hodnotu v posledním prvku návratového zásobníku a zároveň obnoví příznaky `ZERO` a `CARRY`, které se uložily v době vzniku přerušení a nastaví příznak `ENABLE` do stavu daného variantou instrukce `RETURNI`.

Operace skoku - instrukce ovlivňující sekvenci příkazů, které procesor vykonává, protože nastavují programový čítač na novou adresu. Dají se rozdělit na skoky nepodmíněné(`JUMP`) a podmíněné(`JUMP Z/NZ/C/NC`), přičemž u podmíněných instrukcí je skok podmíněn (ne)nastaveným příznakem `CARRY` nebo `ZERO`.

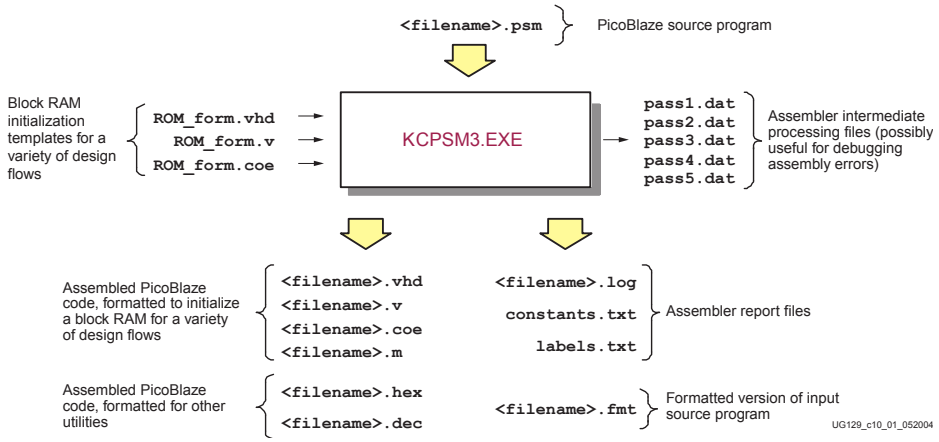
Operace pro volání rutin - instrukce `CALL`, která je podobná operaci skoku, ale navíc původní obsah programového čítače spolu s příznaky uloží na návratový zásobník. V případě vykonání instrukce `RETURN`, se pak do programového čítače nastaví adresa uložena na vrcholu návratového zásobníku. Instrukce `CALL` má také podmíněné varianty `CALL Z/NZ/C/NC`, jejichž vykonání závisí na příznacích, podobně jako tomu bylo u operací skoku.

Kapitola 3

Dostupné vývojové nástroje

3.1 Xilinx KCPSM3

KCPSM3 je jednoduchá utilita, pracující v příkazové řádce. Jedná se o překladač kódu napsaného v assembleru do strojového kódu. Vstupem programu KCPSM3 je zdrojový kód procesoru PicoBlaze a soubory, které obsahují šablony pro vytvoření VHLD, Verilog a Core Generator souborů. Výstupem u bezchybného překladu je celkem 15 souborů, které jsou popsány na obrázku 3.1.



Obrázek 3.1: KCPSM3 překlad (převzato z uživatelské příručky[6])

3.2 Mediatronix pBlazeIDE

PBlazeIDE je grafické vývojové prostředí, které umožňuje tvorbu a následnou simulaci kódu. Má mírně odlišnou syntaxi psaného kódu než program KCPSM3.

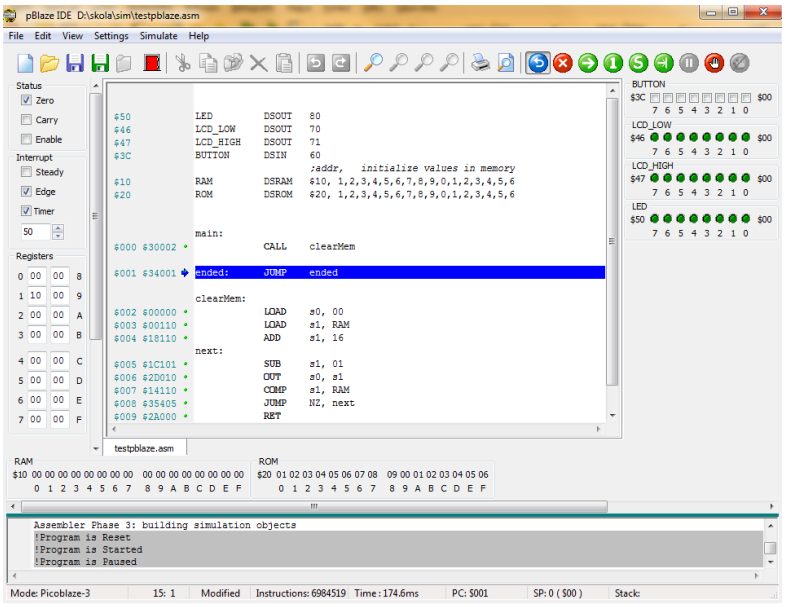
Pro přehledné zobrazení dat, které se posílají na periferie lze použít direktivy `DSIN`, `DSOUT` a `DSIO`, za které se napíše číslo portu periferie, jak lze vidět na obrázku 3.2 ve zdrojovém kódu na prvních řádcích. Zobrazení dat na periferiích je pak k dispozici v pravém panelu.

Pokud chce uživatel jako periferií použít paměť ROM nebo RAM, může využít direktivy `DSRAM` a `DSROM`, které simulují paměť. Číslo portu, na kterém se paměť nachází a inicializační hodnoty v paměti se zapisují za danou direktivu, jak lze vidět na obrázku 3.2 ve zdrojovém kódu. Obsah daných pamětí je pak zobrazen pod kódem.

PBlazeIDE je vhodné pro rychlý vývoj kódu, ale kromě paměti neobsahuje další nástroje, které by simulovali práci s periferiemi.

Kvůli odlišné syntaxi obsahuje nástroj pro konvertování kódu z KCPSM do pBlazeIDE assembleru.

Kód můžeme spouštět po jednotlivých krocích nebo nechat běžet dokud se nezastaví na breakpointu. Případně můžeme procesor zastavit sami. V době běhu simulace je na levé straně panel, kde je možné sledovat stav registrů a příznaků.



Obrázek 3.2: Vývojové prostředí Mediatronix pBlazeIDE

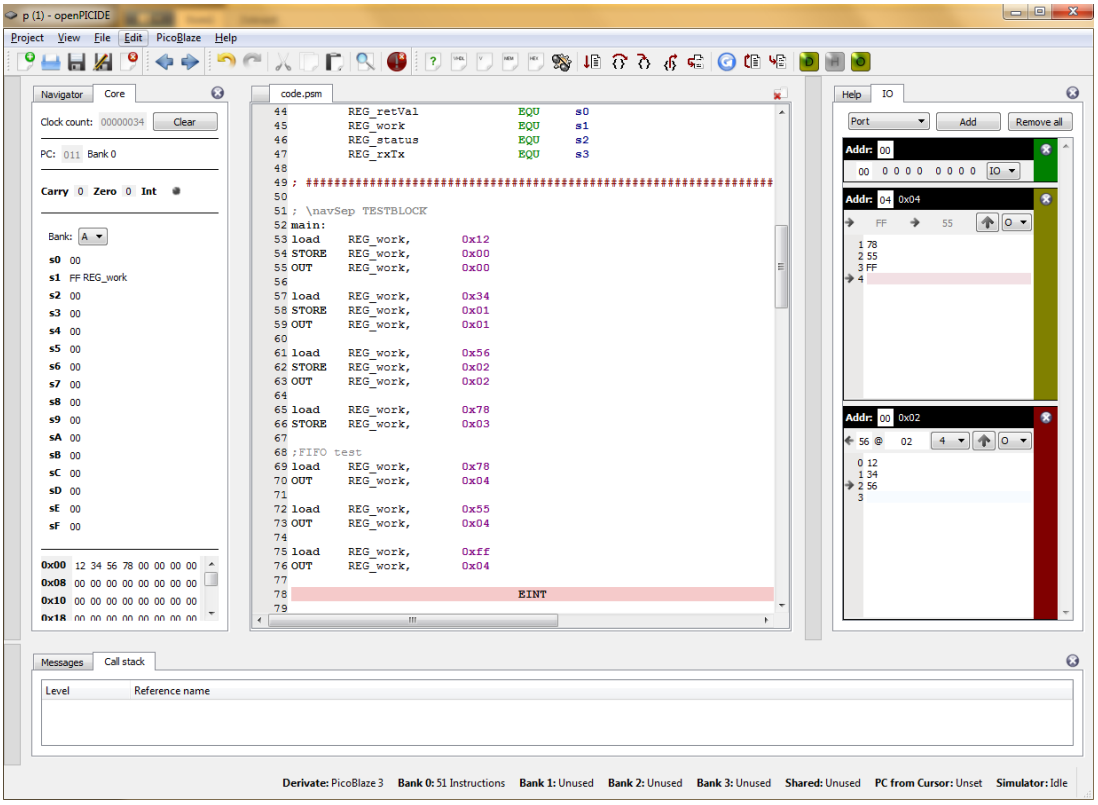
3.3 openPICIDE

Program openPICIDE je grafické vývojové prostředí, které v porovnání s PBlazeIDE má trochu bohatší výbavu ve formě navigace v kódu díky direktivám, které vytvoří záchytné body v programu a v záložce navigator, pak vytvoří osnovu, která umožní programátorovi rychle přepínat mezi jednotlivými sekcemi a návěštími v kódu.

Zdrojový kód, který tento program překládá má stejnou syntaxi jako program pBlazeIDE, ale neobsahuje direktivy pro označování portů periferií, protože ty se dají nastavit v panelu IO, pokud nejsou nastaveny ručně, program automaticky zobrazí příslušnou tabulku, pokud je vykonán příkaz, který adresuje danou periferiemi. Simulace periferií napojených na procesor se skládá ze tři možnosti, které jsou zobrazení dat, paměť a FIFO fronta, jak lze vidět na obrázku 3.3 v pravém panelu.

Program má zároveň pěknou dokumentaci a návody k práci s aplikací, bohužel k jednotlivým instrukcím chybí popisy, co která instrukce provádí.

Bohužel má aplikace velice pomalou simulaci kódu. Například smyčka, která má zpozdit výpočet o jednu sekundu, se na počítači simuluje více než jednu hodinu.



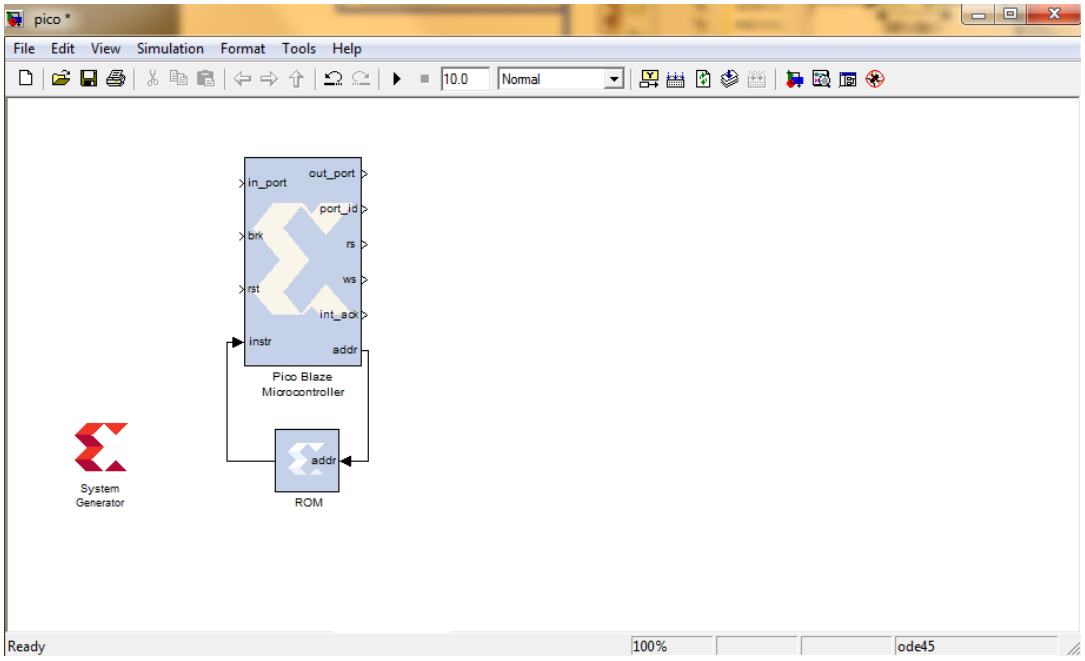
Obrázek 3.3: Vývojové prostředí OpenPICIDE

3.4 Simulink a Xilinx system generator

Simulink je nástavba nad programem Matlab, která umožňuje simulaci a modelování dynamických systémů. Proto je vhodná i pro návrh a simulaci velkých systému uvnitř logiky FPGA. Základní bloky pro tvorbu FPGA systémů v Simulinku jsou obsaženy v Xilinx system generatoru. Bohužel mezi tyto bloky nepatří komponenta procesoru PicoBlaze, která musí být dodatečně vytvořena nástrojem Xilinx EDK a kód vytvořený uživatelem je pak nutno překládat a propojit s pamětí, která je vytvořena v modelu a napojena na procesor. Celý návod na zprovoznění tohoto řešení je v manuálu Xilinx system generatoru [5].

Jak lze vidět, pro tento typ simulace je použito několik programů a než uživatel nasimuluje první kus kódu, musí věnovat značné úsilí, aby zprovoznil celý systém. Navíc programy Matlab a Xilinx system generator jsou komerční produkty, jejichž zakoupení stojí desetitisíce korun.

Uživatel sice nemůže kód upravovat a ladit tak pohodlně jako u softwaru, který je vytvořen na míru procesoru PicoBlaze. Nespornou výhodou je fakt, že toto řešení umožňuje simulovat širokou škálu periferií a libovolně je připojovat na procesor. Uživatel má pak možnost sledovat jak funguje celá FPGA logika a závislostí v ní.



Obrázek 3.4: Vývojové prostředí Xilinx system generator

Kapitola 4

Návrh řešení

Zadání této práce zahrnuje návrh a implementaci pluginu pro program QDevKit, který bude umožňovat tvorbu kódu v assembleru a jeho simulaci. Zároveň bude mít uživatel možnost naprogramovat procesor PicoBlaze běžící na FITkitu.

Aby bylo možno s procesorem komunikovat a ladit jej za běhu, bude nutné vytvořit i aplikaci pro FITkit. Tato aplikace bude mít dvě části, jedna bude implementována v FPGA a bude obalovat procesor PicoBlaze takovým způsobem, aby procesor mohl být programován, spouštěn a zastavován, podle toho, jaký pokyn vydá plugin v aplikaci QDevKit. A druhá část bude v MCU, které pak bude řídit komunikaci mezi pluginem a logikou v FPGA.

Protože by nebylo vhodné nutit uživatele, aby při každé simulaci programoval procesor na FITkitu, bude mít možnost provádět simulaci, která bude probíhat pouze na počítači a proto ji budu v této práci označovat za offline simulaci. Naproti tomu online simulace bude podporovat běh kódu na FITkitu a tomu odpovídající simulaci běžící v pluginu pro QDevKit.

4.1 Grafické uživatelské rozhraní pluginu

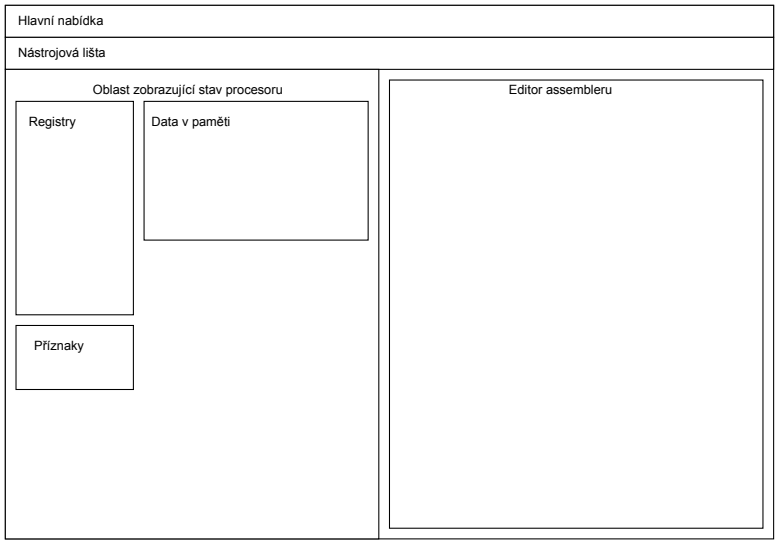
Při návrhu GUI jsem se inspiroval jednoduchými simulátory jako Mediatronix pBlazeIDE a openPICIDE, které mají podobnou funkčnost jako mnou navrhované řešení. Tyto aplikace kladou důraz na jednoduchost a intuitivnost ovládání. Pokud má uživatel alespoň malou zkušenost se simulátory nebo debuggery, měl by si základní funkce osvojit velice brzo po zapnutí aplikace. Z tohoto důvodu jsem se rozhodl svou aplikaci od tohoto jednoduchého konceptu moc neodlišovat. Uživatel má tedy všechny nástroje pro generování a ladění kódu po ruce, nikam se nemusí proklikávat a nic nastavovat. Návrh GUI, které bude popsáno v následujícím textu, je na obrázku 4.1.

Editor kódu

Největší prostor v GUI bude zabírat editor kódu, u kterého se předpokládá, že jej bude uživatel využívat nejčastěji.

Editor musí zvládat klasické operace práci s textem, na které je uživatel zvyklý při práci s jinými editory textu.

Při psaní zdrojového kódu je vhodné mít očíslované řádky, proto by ani u této aplikace neměl chybět blok na okraji editoru, který bude čísla jednotlivých řádků zobrazovat.



Obrázek 4.1: Návrh GUI

Do editoru musí být vhodný způsob, jak vkládat a odebírat breakpointy. Zároveň by měl umožňovat přepnutí do simulačního módu, kdy program zkontroluje funkčnost napsaného programu a pokud je kód napsán bez chyb, zakáže editaci kódu a umožní simulovat procesor. Při simulaci pak vhodně indikoval, že je kód vykonáván a na kterém řádku v kódu se právě nachází vykonávaná instrukce.

Oblast zobrazující stavu procesoru

Pro zobrazení dat uvnitř procesoru je vyhrazená levá část uživatelského rozhraní, která bude zobrazovat vnitřní stav procesoru PicoBlaze jako jsou registry, příznaky, data v paměti, programový čítač, návratový zásobník a případně data poslaná na periferie.

Tuto oblast jsem navrhl tak, aby bylo možné vkládat jednotlivé segmenty s různými informacemi o vnitřním stavu procesoru. Tento návrh se může zdát zbytečný, protože procesor má jen omezenou množinu prvků, které může ovlivnit. Pokud, ale vezmeme v úvahu, že procesor PicoBlaze má 256 vstupních a 256 výstupních portů, u kterých by uživatel mohl chtít vidět stav, je množství informace pro přehlednost a velikost GUI neúnosné.

Nástrojová lišta

Nástrojová lišta je prvkem GUI, který není příliš velký, přesto však dokáže svému uživateli poskytnout značnou míru pohodlí a rychlosti práce. V této liště by se měli nacházet akce, které uživatel využívá nejčastěji. V našem případě budou nejvíc využívány akce spojené se simulací jako je krokování, spouštění a pozastavování simulace. Pro dané akce by se měly používat jednoduché ikony, které uživateli sdělí co daná akce provádí, zároveň je dobré používat pro ikony rozdílné barvy, protože díky barvám uživatel vnímá tyto ikony i periferně a dokáže rychle nalézt akci, kterou chce vykonat.

4.2 Hardwarová podpora pro online simulaci

Při podpoře online simulace, musí být v logice FPGA kromě samotného procesoru PicoBlaze i prostředí, které procesor obaluje a umožní jej programovat, spouštět, zastavovat a krokovat. Programování a řízení procesoru je prováděno z pluginu, který komunikuje přes UART, ale logika v FPGA přijímá data přes rozhraní SPI, tudíž je nutné provést konverzi těchto komunikačních rozhraní.

Konverzi bude provádět mikroprocesor MSP430, který se nachází na FITkitu a je propojen s rozhraním UART i SPI. Pro usnadnění komunikace skrz mikroprocesor budu používat knihovnu libfitkit, která bude podrobněji představena v kapitole 4.3.

4.3 Použité technologie

Pro QDevKit se dají programovat pluginy v jazycích Python a C++ nebo je i kombinovat. Rozhodl jsem se pro implementaci v C++, protože je to jazyk kompilovaný na rozdíl od Pythonu, který je skriptovací, což znamená, že rychlost simulace v jazyce C++ by měla být rychlejší.

Zároveň budu využívat framework Qt, který umožňuje vytvořit multiplatformní aplikace a nabízí širokou škálu prvků pro tvorbu GUI.

Framework Qt

Qt je komplexní framework pro jazyk C++, který umožňuje vytvářet multiplatformní aplikace pro Windows, Linux, Mac OS X, Solaria a mnoho dalších verzí Unixu s X11.

Framework je licencován pod dvěma licencemi. Pro volně šířitelný kód to je GNU Lesser General Public License v. 2.1. Pro komerční využití, kde se vytvořený kód nemusí sdílet, existuje komerční licence Qt. [1] Obsahuje silné API, mnoho c++ tříd a nástroje pro tvorbu GUI. [1]

Knihovna libfitkit

Knihovna libfitkit zastřešuje základní funkce pro použití MCU a komunikaci MCU s komponentami na FITKitu nebo PC. Knihovna poskytuje platformu pro použití FITKitu a základní rozhraní:

- *Komunikaci s terminálem* přes USB a UART MCU - V knihovně je implementovaná komunikace přes sériové rozhraní, které je připojeno přímo na port B čipu FT2232C. Nad sériovou komunikací je implementován jednoduchý příkazový interpret, do kterého lze zadávat příkazy pro MCU, které knihovna zpracuje. Pokud se jedná o příkaz knihovny, knihovna daný příkaz vykoná. V opačném případě se příkaz zadaný do příkazového řádku pošle funkci `USER_CMD_Decode`, kterou si uživatel musí nadefinovat.

- *Podpora SPI rozhraní* - Rozhraní slouží ke komunikaci s FLASH pamětí, FPGA a na FITkitu 2.0 ke konfiguraci zvukového kodeku.[4]

[1] http://qt-project.org/products/licensing

Kapitola 5

Implementace simulátoru

Výsledkem této práce je plugin do QDevKitu, který umožňuje uživateli otevřít nové okno, které poskytuje nástroje pro tvorbu kódu, jeho simulaci a programování procesoru PicoBlaze na FITkitu.

V této kapitole bude ukázáno grafické rozhraní pluginu a jeho možnosti překladu a simulace. Budou popsány základní principy v logice FPGA. Na příkladu bude ukázán princip komunikace mezi pluginem a aplikací na FITkitu. A nakonec bude vyhodnocena výkonnost celé aplikace.

5.1 Grafické uživatelské rozhraní pluginu

Při tvoření grafického rozhraní jsem neodbočoval od návrhu a vytvořil jej tak, aby rozhraní bylo jednoduché, přehledné a uživatel všechny dostupné nástroje pro simulaci měl přímo před očima.

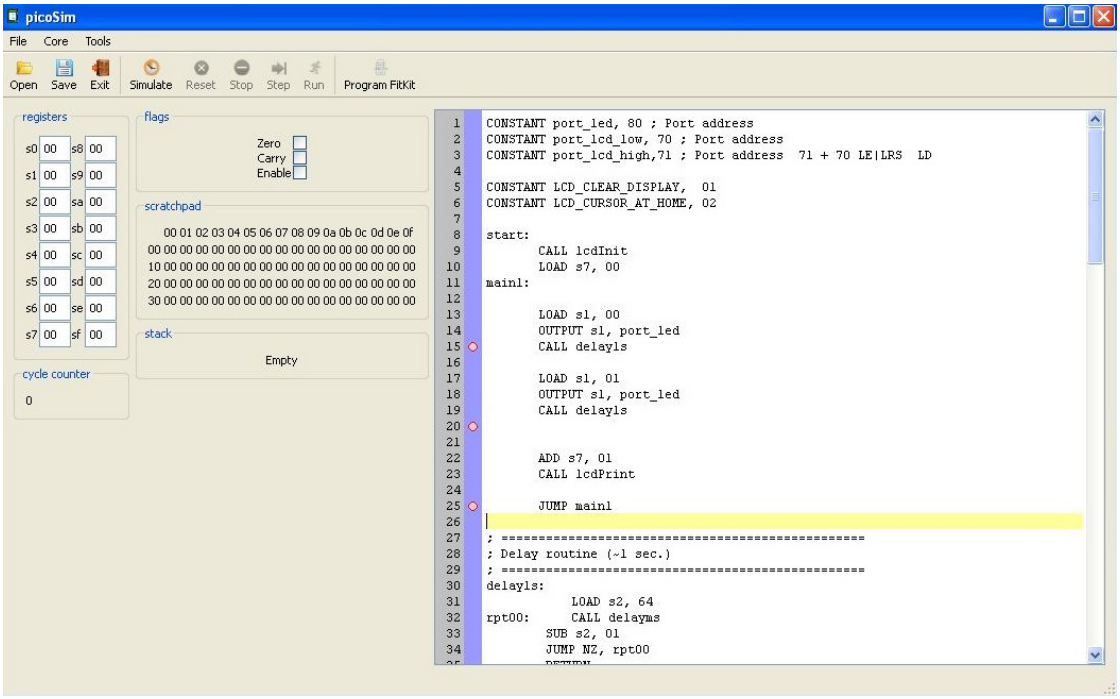
Pro tvorbu hlavního okna, menu a nástrojové lišty jsem používal nástroj Qt Designer, který je součásti balíčku frameworku QT. Tento nástroj umožňuje tvořit grafické rozhraní jen pomocí klikání myší a výsledný kód v C++ je vygenerován automaticky. Pro další části grafického rozhraní, jako editor kódu nebo jednotlivé komponenty zobrazující vnitřní stav procesoru, jsem již Qt Designer nepoužíval, protože jeho možnosti jsou omezené a složitější prvky v grafickém rozhraní je lepší psát ručně.

Výsledné grafické rozhraní je zobrazeno na obrázku 5.1, jehož základní prvky budu popisovat v následujícím textu.

Editor assembleru

Editor zdrojového kódu je reprezentován třídou `CodeEditor`, která dědí z třídy `QPlainTextEdit`, protože ta poskytuje funkcionalitu klasického editoru textu včetně podpory klávesových zkratek.

Funkčnost tohoto textového editoru rozšiřuji přidáním grafické komponenty, která zobrazuje čísla řádků a kliknutím do této oblasti se přidává breakpoint na příslušný řádek, kde se vykonávaný kód při simulaci zastaví. Při spuštění simulace, pak tato komponenta připisuje na řádky s kódem adresy v paměti, kde jsou instrukce uloženy.



Obrázek 5.1: Hlavní okno programu

Oblast zobrazující stav procesoru

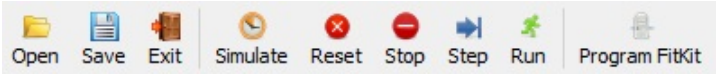
O vykreslování vnitřního stavu procesoru se stará třída `StatusArea`, která pracuje s jednotlivými komponentami zobrazující data procesoru.

Kvůli tomuto řešení jsem implementoval mechanizmus, při kterém se jednotlivé komponenty zaregistrují třídě `StatusArea` metodou `registerStatusWidget`, která pak tyto komponenty vyzývá k aktualizaci zobrazovaných dat.

Mimo jiné jsem vytvořil abstraktní třídu `StatusWidget`, která vytváří rozhraní pro komponenty tím, že vynucuje implementaci metody `updateState` v těle každé komponenty. Díky tomuto způsobu implementace se nemusí tělo třídy `StatusArea` měnit, když v této oblasti vznikne nová komponenta.

Nástrojová lišta

Tato lišta slouží jako snadný přístup pro hlavní akce, které může uživatel vykonat. Dělí se na tři části, které oddělují práci se soubory, simulaci a programování procesoru na FITkitu.



Obrázek 5.2: Nástrojová lišta

Na obrázku 5.2 je nástrojová lišta, která obsahuje tlačítka:

- **Open -** Otevře vybraný textový soubor na disku a zobrazí jej v editoru kódu.

- **Save -** Uloží rozpracovaný zdrojový kód na disk.
- **Exit -** Ukončí aplikaci.
- **Simulate -** Přeloží zdrojový kód, pokud byl přeložen bez chyb, umožní simulovat procesor pomocí následujících akcí.
- **Reset -** Uvede procesor do počátečního stavu.
- **Stop -** Zastaví provádění instrukcí.
- **Step -** Provede jednu instrukci.
- **Run -** Začne postupně provádět instrukce dokud není zastaven akcí *Stop* nebo nenarazí na breakpoint.
- **Program FITkit -** Přeloží zdrojový kód, pokud byl přeložen bez chyb, naprogramuje procesor na FITkitu a zahájí simulaci.

5.2 Překlad assembleru

Správný překlad assembleru je klíčovým prvkem u tohoto simulátoru, protože špatný překlad by znamenal nefunkčnost celého řešení. Z tohoto důvodu jsem se rozhodl použít již hotový program pro překlad assembleru a to KCPSM3, u kterého lze předpokládat, že překládá bez chyb a případné chyby budou opraveny, protože tento program je pod záštitou firmy Xilinx.

Nicméně, aby bylo možné tento program nahradit jiným, případně implementovat vlastní překladač, je použití externího překladače zabaleno do třídy `PBlazeCompiler`, která může být nahrazena za jiné řešení překladu.

Úspěšný výsledek překladu je interpretován přepnutím simulátoru do ladícího módu. Jestliže překlad skončí syntaktickou chybou, zobrazí se chybová hlášku umístěna pod editorem kódu.

5.3 Emulace procesoru PicoBlaze

Jádro procesoru emuluje třída `PBlaze` obsahující metody, které odpovídají procesům probíhajícím na skutečném procesoru jako například `init`, `reset` a `step`.

- **init -** Inicializuje procesor do počátečního stavu. Registry, příznaky i paměť jsou vymazány a programový čítač je nastaven na první položku paměti.
- **reset -** Procesor je resetován. Data v registrech a paměti přetrvají. Programový čítač se nastaví na první adresu paměti.
- **step -** Procesor vykoná jednu instrukci, která je v reálném procesoru vykonána za dva cykly. Na přiloženém zdrojovém kódu 5.1 je ukázána implementace této metody, která vybere z paměti příslušnou instrukci a předá ji k vykonání metodě `executeInstruction`, ta pak instrukci dekóduje a provede příslušnou operaci nad procesorem.

```
void PBlaze::step ( void )  {
    int instruction = memory[pc].code;
    executeInstruction(instruction);

    pc = npc ;
    //PicoBlaze executes instruction in 2 clks
    cycleCounter += 2;
    return;
}
```

Zdrojový kód 5.1: PBlaze::step

Hlavním prvkem této třídy je metoda `executeInstruction`, která vykoná nad procesorem operaci, která odpovídá vykonané instrukci na reálném procesoru. Tato metoda má jediný argument a to je instrukce v číselném tvaru. Tato instrukce je pak dekódována na typ instrukce a její operandy. S těmito informacemi pak není problém vykonat příslušnou operaci nad procesorem.

Tuto třídu jsem implementoval takovým způsobem, aby byla přenositelná i do jiných programů, tím pádem nevolá žádné metody jiných tříd. Komunikace mezi jinými třídami, zejména s GUI, pak probíhá tak, že se ostatní třídy dotazují na stav procesoru.

5.4 Hardwarová podpora pro online simulaci

Tato hardwarová podpora se implementovala jako aplikace pro FITkit, která se skládá ze dvou částí, jedna se nachází v FPGA, ta obsahuje prostředí, které obaluje procesor PicoBlaze a jeho periferie. Pomocí tohoto prostředí lze kontrolovat procesor, programovat a krokovat jej pomocí příkazů posílaných. Část v MCU pak plní roli prostředníka pro komunikaci počítače s FPGA.

Komunikace s FITkitem pak zahrnuje pouze pár příkazů:

- **PROG -** Zahajuje komunikaci, která naprogramuje procesor PicoBlaze, přesněji přenese přeložený zdrojový kód do paměti vytvořené uvnitř FPGA.
- **CPU START -** Povolí procesoru běh, po tomto příkazu se procesor uvede do svého počátečního stavu.
- **CPU STOP -** Zastaví běh procesoru.
- **CPU STEP -** Procesor vykoná jednu instrukci.

5.4.1 Běh kódu na FITkitu

Uživatel má k dispozici dva druhy simulace. Online simulaci, ve které je procesor emulován v pluginu a offline simulaci, kde je kód nahrán na procesor ve FITkitu a může pomocí instrukcí, které posílají data na periferie ovládat LED diodu nebo displej.

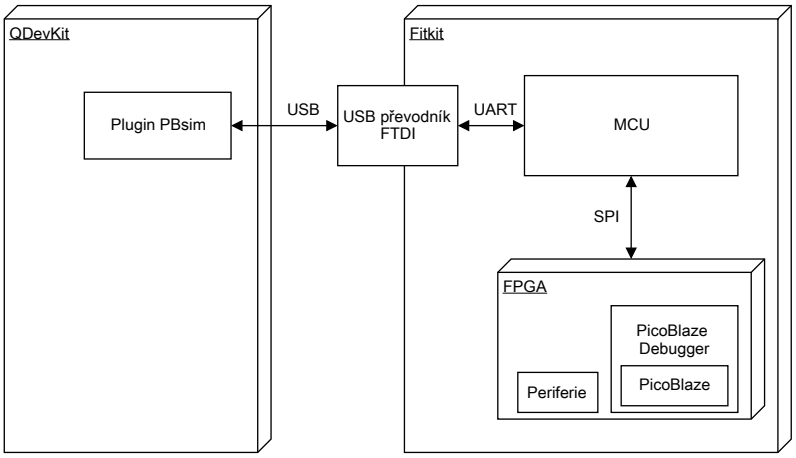
Online simulace je možná pouze, pokud je na FITkitu naprogramována aplikace PicoBlaze debugger, která se stará o přenos kódu z pluginu do paměti procesoru, propojí výstupní porty s periferiemi a zpracovává instrukce umožňující jeho ovládání a krokování.

Souběžnou simulace procesoru v pluginu a kód běžící na FITkitu jsem vyřešil tak, že plugin posílá instrukci jednoho kroku na FITkit. Tato instrukce způsobí, že procesor je

spuštěn na dva časové cykly, v nichž je provedena právě jedna instrukce na procesoru ve FITkitu. Tento způsob řízení chodu procesoru na FITkitu, není efektivní a značnou měrou zpomaluje celou online simulaci.

5.5 Komunikace mezi pluginem a aplikací na FITkitu

Komunikace pluginu s procesorem PicoBlaze prochází přes několik částí, jak je naznačeno na obrázku 5.3.



Obrázek 5.3: Architektura aplikace

Plugin komunikuje s MCU přes komunikační kanál daného FITkitu, jak je ukázáno v kódu 5.2, kde se přes kanál posílá zpráva `CPU STOP`. MCU na FITkitu zachytává zprávy

```
QtAPI::IOChannel& ch = *device->channelB();
ch.write("CPU STOP\r\n");
```

Zdrojový kód 5.2: Komunikace plugin MCU

pomocí funkce `unsigned char decode_user_cmd` (viz obrázek 5.3), která ve svých parametrech má text přijímaného příkazu. Na základě přijatého příkazu vykoná MCU danou operaci, v tomto případě zavolá funkci `PicoCPU_disable()`. V těle této funkce je také makro, které posílá data přes rozhraní SPI komunikující s logikou v FPGA.

```
unsigned char decode_user_cmd(char *cmd_ucase, char *cmd)
{
  if (strcmp(cmd_ucase, "CPU STOP") == 0) {
      PicoCPU_disable();
      return (USER_COMMAND);
    }
  return CMD_UNKNOWN;
}
```

Zdrojový kód 5.3: Zachytávání zpráv pluginem

5.6 Výkonnost simulace

Protože se mi nepovedlo přeložit QDevKit pod operačním systémem Windows 7, aplikaci jsem překládal a testoval pomocí programu VirtualBox pod nasimulovaným systémem Windows XP. Testování probíhalo na notebooku s procesorem i3-330M firmy Intel s frekvencí 2.13 GHz.

Testy měřili počet vykonaných instrukci za minutu online simulace. Výsledky ukázali, že za minutu běhu zvládne aplikace vykonat přibližně jednu miliardu instrukcí, což lze přirovnat k procesoru PicoBlaze běžícím na frekvenci 33,3 MHz.

Z tohoto usuzuji, že rychlost simulace je naprosto bezproblémová, protože se výkonnostně vyrovnává programu běžícímu v hardwaru a lze předpokládat, že rychlost pod systémem, který není simulován bude ještě o něco rychlejší.

Kapitola 6

Testování simulátoru

Vyhodnocení funkčnosti simulátoru bylo provedeno na sadě testovacích programů, které testují funkčnost online simulace a poslední testovací program testuje i offline simulaci tím, že ovládá displej FITkitu.

16bitová násobení

Procesor PicoBlaze neobsahuje vestavěnou násobičku a pokud ji chceme používat může být násobička naprogramovaná pomocí dostupných instrukcí nebo může být použita násobička připojena na periferie procesoru. Zdrojový kód 6.1 uvádí jednu z možných implementací násobičky, která násobí obsah registrů `s0` a `s1`. Výpočet je prováděn v 50-57 instrukčních cyklech. Výsledek je uložen jako 16-ti bitové číslo do registrů `s3` a `s4`.

```
; Multiplier Routine (8-bit x 8-bit = 16-bit product)
        ; ================================================
        ; Shift and add algorithm
        ;
mult_8x8:
        NAMEREG s0, multiplicand        ; preserved
        NAMEREG s1, multiplier          ; preserved
        NAMEREG s2, bit_mask            ; modified
        NAMEREG s3, result_msb          ; most-significant byte (MSB) of result,
        ; modified
        NAMEREG s4, result_lsb          ; least-significant byte (LSB) of result,
        ; modified

        LOAD bit_mask, 01               ; start with least-significant bit (lsb)
        LOAD result_msb, 00             ; clear product MSB
        LOAD result_lsb, 00             ; clear product LSB (not required)

        ; loop through all bits in multiplier
mult_loop:
        TEST multiplier, bit_mask       ; check if bit is set
        JUMP Z, no_add                  ; if bit is not set, skip addition

        ADD result_msb, multiplicand    ; addition only occurs in MSB

no_add: SRA result_msb                  ; shift MSB right, CARRY into bit 7,
                                        ; lsb into CARRY
        SRA result_lsb                  ; shift LSB right,
                                        ; lsb from result_msb into bit 7

        SL0 bit_mask                    ; shift bit_mask left to examine
                                        ; next bit in multiplier

        JUMP NZ, mult_loop              ; if all bit examined, then bit_mask = 0
  RETURN
```

Zdrojový kód 6.1: Násobička(převzato z uživatelké příručky [6])

Iterativní výpočet faktoriálu

Výpočet využívá 16bitovou násobičku a výsledek ukládá do dvojice registrů `s6` a `s7`. Je schopný vypočítat nejvýše faktorial čísla 8, protože pak výsledek přeteče přes 16 bitů. Při výpočtu faktoriálu čísla 6 a výš dochází k násobení 16bitového čísla s 8bitovým číslem, tuto situaci lze vyřešit dvojím použitím násobičky viz zdrojový kód 6.2.

```
      fact16b: ;Pocitany faktorial se nastavi do registru s8
               LOAD s7, 01
   fact_iter:
               LOAD s0, s8
               LOAD s1, s7
               LOAD s5, s6
               CALL mult8x16to16
               SUB s8, 01
               JUMP NZ, fact_iter
               RETURN
mult8x16to16:
               CALL mult_8x8
               LOAD s6, s3
               LOAD s7, s4

               LOAD s1, s5
               CALL mult_8x8
               ADD s6, s4
               RETURN
```

Zdrojový kód 6.2: Výpočet faktoriálu

Ovládání displeje na FITkitu

Poslední test je řízení displeje na FITkitu, který je řízen řadičem, na který procesor posílá data pomocí instrukcí `OUTPUT`. Tento program používá 8 datových vstupů řadiče, které jsou napojeny na adresu port 70, v kódu je používána pod konstantou `port_lcd_low` a řídící vstup `RS`, který rozlišuje jestli se jedná o instrukci nebo se data mají uložit do vnitřní paměti řadiče.Dalším řídícím vstupem je `E`, při jehož sestupné hraně řadič vykoná danou instrukci. Tyto dva řídící vstupy jsou napojeny na adresu port 71, v kódu je používána pod konstantou `port_lcd_high`.

Displej se nejprve inicializuje pomocí rutiny `lcdInit`, která vymaže displej, nastaví kurzor na začátek a také pošle instrukci, která iniciuje komunikaci na 8bitový komunikační mód na dvouřádkovém displeji, který bude zobrazovat font 5x8. Tato rutina volá další rutinu s názvem `lcdcmd`, která vytvoří kmit na vstupu řadiče `E` a tím řadič instrukci zpracuje.

```
CONSTANT port_led, 80 ; Port address
CONSTANT port_lcd_low, 70 ; Port address
CONSTANT port_lcd_high,71 ; Port address
        ;71 + 70 LE|LRS  LD

CONSTANT LCD_CLEAR_DISPLAY,  01
CONSTANT LCD_CURSOR_AT_HOME, 02

start:
       CALL lcdInit
       LOAD s7, 00
mainl:

       LOAD s1, 00
       OUTPUT s1, port_led
       CALL delay1s

       LOAD s1, 01
       OUTPUT s1, port_led
       CALL delay1s


       ADD s7, 01
       CALL lcdPrint

       JUMP mainl

lcdInit:
       ; LCD clear
       LOAD s6, LCD_CLEAR_DISPLAY
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ; LCD home
       LOAD s6, LCD_CURSOR_AT_HOME
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ; LCD function set - osmibitovy prenos,
        ;displej slozen ze dvou polovin, font 5x8
       LOAD s6, 38
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ; LCD display - zobrazeni kurzoru,
        ;zapnuti displeje
       LOAD s6, 0E
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ; LCD entry mode - automaticka inkrementace
        ;adresy kurzoru
       LOAD s6, 06
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       RETURN

lcdPrint:
       ;presun kurzoru na znak 0
       LOAD s6, 80
       OUTPUT s6, port_lcd_low
       LOAD s6, 00
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ;tisk znaku ':'
       LOAD s6, 3A
       OUTPUT s6, port_lcd_low
       LOAD s6, 01
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       ;tisk znaku ')'
       LOAD s6, 29
       OUTPUT s6, port_lcd_low
       LOAD s6, 01
       OUTPUT s6, port_lcd_high
       CALL lcdcmd

       RETURN

lcdcmd:
       CALL delayms

       OR s6, 02
       OUTPUT s6, port_lcd_high
       CALL delayms

       AND s6, FD
       OUTPUT s6, port_lcd_high
       CALL delayms
       RETURN
```

Zdrojový kód 6.3: Ovládání LCD

Kapitola 7

Závěr

Doposud byla jediná možnost, jak pracovat s procesorem PicoBlaze a to přenést kód v hexadecimálním tvaru do paměti v FITkitu a procesor pouštět a zastavovat skrze terminál QDevKitu. Ladění kódu tímto způsobem vyžadovalo minimálně používat další program, který umožnil tvořit kód a překládat jej do strojového kódu.

Výsledkem této práce je simulátor, který vznikl na základě rozboru existujících simulátorů. Byl implementován v jazyce C++ za pomocí frameworku Qt, který byl použit zejména k tvorbě grafického uživatelského rozhraní a knihovny libfitkit, která byla využívána pro prací s MCU. Výsledné řešení celé aplikace dovolilo vzniknout simulační knihovně procesoru PicoBlaze, která lze použít i v jiné aplikaci než QDevKit.

Simulátor umožňuje programátorovi pracovat daleko pohodlněji a dává mu nástroje pro tvorbu kódu, jeho simulaci a možnost testovat a krokovat na hardwarovém přípravku FITkit.

Jedna z možných rozšíření je podpora simulace periferií, které by se pak v simulátoru daly přiřadit k jednotlivým portům procesoru. Protože byl tento simulátor vyroben hlavně pro práci s přípravkem FITkit, bylo by nejlepší simulovat periferie, které jsou fyzicky přítomny na FITkitu jako klávesnice a LCD displej.

Literatura

[1] Blanchette, J.; Summerfield, M.: *C++ GUI Programming with Qt 4*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006, ISBN 0131872494.

[2] Chapman, K.: KCPSM3 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-IIPRO [online]. `http://www.eng.auburn.edu/štrouce/class/elec4200/KCPSM3_Manual.pdf`, Říjen 2003 [cit. 2013-3-25].

[3] Cofer, R.; Harding, B.: *Rapid System Prototyping With FPGAs*. Elsevier/Newnes, 2006, ISBN 9780750678667.

[4] Slaný, K.: Knihovna libfitkit - FITkit [online]. `http://merlin.fit.vutbr.cz/FITkit/docs/firmware/mcu_libfitkit.html`, 2009-3-22 [cit. 2013-3-25].

[5] Xilinx: Xilinx System Generator for DSP Version 9.1.01 User's Guide [online]. `http://www.xilinx.com/support/sw_manuals/sysgen_ug.pdf`, 2007-3-19 [cit. 2013-4-10].

[6] Xilinx: PicoBlaze 8-bit Embedded Microcontroller User Guide [online]. `http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf`, 2011-6-22 [cit. 2013-3-25].