VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE PROCESORU PICOBLAZE V PROSTŘEDÍ ECLIPSE

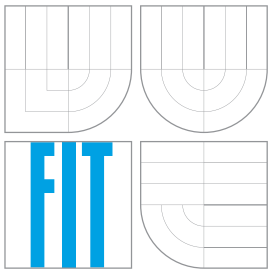
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ŠIMEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE PROCESORU PICOBLAZE V PROSTŘEDÍ ECLIPSE

SIMULATION OF PICOBLAZE MICROCONTROLLER IN ECLIPSE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ ŠIMEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2013

Abstrakt

Tato diplomová práce se zabývá návrhem a tvorbou simulátoru mikroprocesoru PicoBlaze. Cílem práce je vytvoření grafického simulátoru tohoto mikroprocesoru v prostředí Eclipse jako rozšíření existujícího vývojového prostředí VLAM IDE. V práci je uveden podrobný popis simulátoru μCsim, který byl vybrán pro implementaci jádra simulátoru mikroprocesoru PicoBlaze, a dále jsou navržena jeho rozšíření mající za cíl usnadnit použití simulátoru pro automatizované testování. Následně je představen návrh a implementace grafického simulátoru v prostředí Eclipse využívajícího vytvořené jádro simulátoru a jsou diskutovány možnosti rozšíření implementovaného jádra a grafického simulátoru.

Abstract

This thesis deals with the design and implementation of a simulator of PicoBlaze microcontroller. The aim of this thesis is to create a graphical simulator of this microcontroller in Eclipse as an extension of the existing integrated development environment VLAM IDE. The thesis describes in detail the simulator μCsim which was chosen for implementation of PicoBlaze simulator core and introduces its improvements for a better support of automatized testing. The thesis also presents a description of desing and implementation of the graphical simulator in Eclipse which uses the created simulator core and discusses possible improvements of the implemented core and graphical simulator.

Klíčová slova

PicoBlaze, simulátor, uCsim, PicoBlaze C Compiler, Eclipse, zásuvný modul

Keywords

PicoBlaze, simulator, uCsim, PicoBlaze C Compiler, Eclipse, plug-in

Citace

Jiří Šimek: Simulace procesoru PicoBlaze v prostředí Eclipse, diplomová práce, Brno, FIT VUT v Brně, 2013

Simulace procesoru PicoBlaze v prostředí Eclipse

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.

.....
Jiří Šimek
22. května 2013

Poděkování

Děkuji panu Ing. Zbyňku Křivkovi, Ph.D. za odborné vedení, rady, ochotu a trpělivost při tvorbě této práce.

Obsah

Kapitola 1

Úvod

V dnešní době se výpočetní technika nachází všude kolem nás a je nedílnou součástí každodenního života. A to nejen ve formě klasických osobních počítačů, ale v čím dál větší míře také ve formě vestavěných systémů, což jsou většinou specializované jednoúčelové integrované obvody. Kvůli své úzké specializaci na jednu konkrétní činnost je nevýhodou cena jejich návrhu a výroby a vyplatí se pouze ve velkém množství. Další nevýhodou je pak nemožnost změny vytvořeného obvodu.

Nejen z těchto důvodů se do popředí dostávají obvody FPGA (Field Programmable Gate Array), což jsou programovatelná hradlová pole umožňující tvorbu integrovaných obvodů dle konkrétních požadavků zákazníka. V rámci obvodu FPGA tak může být naprogramován mimojiné i jednoduchý mikroprocesor. Jedním z mikroprocesorů určených přímo pro FPGA obvod je jednoduchý 8bitový mikroprocesor PicoBlaze vyvinutý firmou Xilinx.

Na Fakultě informačních technologií Vysokého učení technického v Brně vzniklo pro tento mikroprocesor v rámci prostředí Eclipse integrované vývojové prostředí VLAM IDE a samostatný překladač jazyka C PicoBlaze C Compiler. Tyto nástroje umožňují tvorbu programů pro mikroprocesor PicoBlaze, ale už neposkytují funkčnost pro testování a ladění vytvořeného programu. Pro mikroprocesor PicoBlaze je sice dostupných několik simulátorů, ale jedná se o samostatné aplikace a neumožňují integraci do prostředí Eclipse.

Cílem této práce je vytvoření grafického simulátoru pro mikroprocesor PicoBlaze v prostředí Eclipse jako rozšíření vývojového prostředí VLAM IDE. Motivací je také vytvoření simulátoru pracujícího v příkazové řádce pro usnadnění možnosti automatizovaného testování programů pro mikroprocesor PicoBlaze a následné využití tohoto simulátoru pro implementaci grafického simulátoru v prostředí Eclipse.

V druhé kapitole je představen mikroprocesor PicoBlaze, jeho verze KCPSM3 a KCPSM6, dialekty jazyka symbolických instrukcí pro tento mikroprocesor, existující assemblery a simulátory.

Třetí kapitola obsahuje popis překladače Small Device C Compiler (SDCC), z něj odvozeného překladače PicoBlaze C Combiler (PBCC) pro mikroprocesor PicoBlaze a ve zbývající části je uveden detailní popis simulátoru μCsim, jeho struktura, princip fungování a obecný návod na vytvoření nového portu.

Čtvrtá kapitola se zabývá návrhem a implementací portu simulátoru μCsim pro mikroprocesor PicoBlaze a jeho rozšířením o nové funkce mající za cíl usnadnění automatizovaného testování a využití simulátoru jako jádra grafického simulátoru.

V úvodu páté kapitoly je stručně popsáno prostředí Eclipse a jeho zásuvný modul VLAM IDE. Následně jsou uvedeny požadavky na grafický simulátor a navržen princip jeho činnosti.

Šestá kapitola se věnuje samotné implementaci grafického simulátoru v prostředí Eclipse a uvádí způsob realizace jednotlivých částí.

Předposlední kapitola popisuje způsob testování a diskutuje možnosti rozšíření a vylepšení implementovaného portu simulátoru μCsim a grafického simulátoru v prostředí Eclipse.

V závěrečné kapitole jsou shrnuty dosažené výsledky.

Kapitola 2

PicoBlaze

PicoBlaze [24, 23, 11] je jednoduchý softwarový 8-bitový RISC mikroprocesor vyvinutý společností Xilinx pro použití jejich v programovatelných hradlových polích (Field Programmable Gate Array, FPGA). Tento mikroprocesor je volně dostupný a existuje ve dvou verzích KCPSM3 (dále nazývána jako PicoBlaze 3) a KCPSM6 (dále nazývána jako PicoBlaze 6). Architektura a vlasnosti jednotlivých verzí a jejich rozdíly jsou popsány v následujících částech.

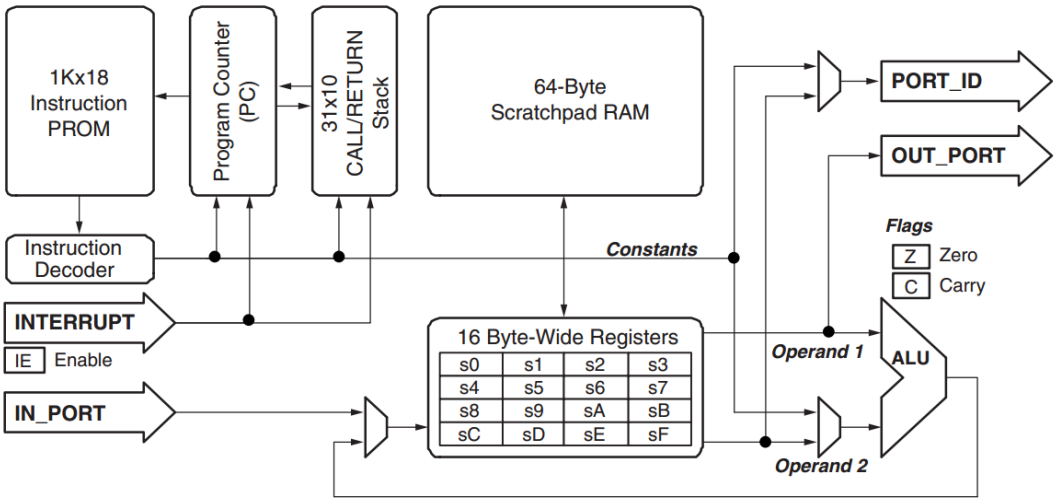
2.1 PicoBlaze 3 (KCPSM3)

Mikroprocesor PicoBlaze 3 [23] je starší a jednodušší verze mikroprocesoru PicoBlaze a byla navržena pro použití v FPGA čipech generace Spartan-3 a Virtex-5. PicoBlaze 3 zabírá v FPGA čipu generace Spartan-3 pouze 96 bloků. PicoBlaze 3 dokáže v závislosti na typu a rychlosti FPGA čipu provést 44 až 100 milionů instrukcí za sekundu (million instructions per second, MIPS).

Architektura mikroprocesoru PicoBlaze 3 je zobrazena na obrázku 2.1. Základní charakteristika mikroprocesoru je následující:

- celkem 16 obecných 8bitových registrů označených s0 až sF. Všechny registry mají stejnou prioritu a žádný z registrů nemá speciální funkci. V operacích s registry je možné použít kterýkoliv z registrů.
- programová paměť pro 1024 instrukcí o šířce 18 bitů
- paměť RAM o velikosti 64 bajtů, která je přístupná pomocí instrukcí `STORE` a `FETCH`. Paměť může být adresována přímo hodnotou nebo nepřímo obsahem jednoho z registrů (pro adresaci je použito pouze spodních 6 bitů registru)
- 8bitová aritmeticko-logická jednotka podporující základní aritmetické a bitové logické operace, porovnávání a bitové posuvy
- příznaky pro přenos (`CARRY`, `C`), nulový příznak (`ZERRO`, `Z`), jež jsou nastavovány aritmeticko-logickou jednotkou a příznak indikující, zda je povoleno přerušení (`INTERRUPT ENABLE`, `IE`)
- programový čítač obsahující adresu následující instrukce. Hodnota programového čítače nemůže být přímo změněna programem. Může být změněna pouze nepřímo instrukcemi pro skoky, volání podprogramů a událostmi přerušení a resetování.

- až 256 vstupních a 256 výstupních nebo kobminovaných vstupně-výstupních 8bitových portů. Pro adresaci portu je využíván výstup `PORT_ID`. Při operaci `INPUT` je vstup čten ze vstupního portu `IN_PORT` a podobně při operaci `OUTPUT` je výstup zapisován na výstupní port `OUT_PORT`.
- zásobník volání podprogramů pro ukládání návratových adres o velikosti 31 položek, který umožňuje zanořené volání podprogramů. Zásobník je implementován jako cyklická paměť, tzn. po přetečení je přepsána nejstarší hodnota. Ukazatel na vrchol zásobníků je nastavován automaticky a nelze ho pomocí instrukce přímo změnit.
- vstup pro externí přerušení umožňující mikroprocesoru reagovat na externí události. Deklarovaná doba reakce na přerušení je 5 hodinových cyklů procesoru.
- vstup pro reset mikroprocesoru nastavující mikroprocesor do výchozího stavu



Obrázek 2.1: Architektura mikroprocesoru PicoBlaze 3 [23]

2.2 PicoBlaze 6 (KCPSM6)

Mikroprocesor PicoBlaze 6 [11] je optimalizován pro FPGA čipy generací Spartan-6, Virtex-6 a 7. I když je o něco složitější než PicoBlaze 3, zabírá v FPGA čipu pouze 26 bloků. Srovnání počtu použitých bloků však není zcela relevantní díky rozdílným architekturám FPGA čipů Spartan-3 a Spartan-6. Reálně je však PicoBlaze 6 o 25% menší než PicoBlaze 3. PicoBlaze 6 podle typu a rychlosti FPGA čipu dokáže provést 52 až 119 milionů operací za sekundu (MIPS).

Architektura mikroprocesoru PicoBlaze 6 je znázorněna na obrázku 2.2. Charakteristika mikroprocesoru je podobná mikroprocesoru PicoBlaze 3 a hlavní rysy jsou:

- celkem 32 obecných 8bitových registrů ve dvou bankách A a B. Výchozí registrová banka je banka A. Aktivně je možné používat pouze jednu banku. Mezi bankami lze přepínat pomocí instrukce `REGBANK`. Registry jsou pojmenovány `s0` až `sF` a vždy označují registr v aktivní bance. Registry mají stejně jako u mikroprocesoru PicoBlaze

3 stejnou prioritu a při registrových operací lze použít libovolný z registrů. Hodnoty mezi registry v rozdílných bankách lze přenášet buď přes paměť RAM, nebo pomocí instrukce `STAR`.

- programová paměť až pro 4096 18bitových instrukcí
- RAM o velikosti až 256 bytů. Stejně jak u PicoBlaze 3 je paměť přístupná pouze pomocí instrukcí `STORE` a `FETCH`. Adresace probíhá buď přímo hodnotou, nebo nepřímo obsahem registrů.
- 8bitová aritmeticko-logická jednotka podporující základní aritmetické a bitové logické operace, porovnávání a bitové posuvy
- příznaky pro detekci přenosu (`CARRY`, `C`), nulový příznak (`ZERO`, `Z`), jež jsou nastavovány aritmeticko-logickou jednotkou a příznak indikující, zda je povoleno přerušení (

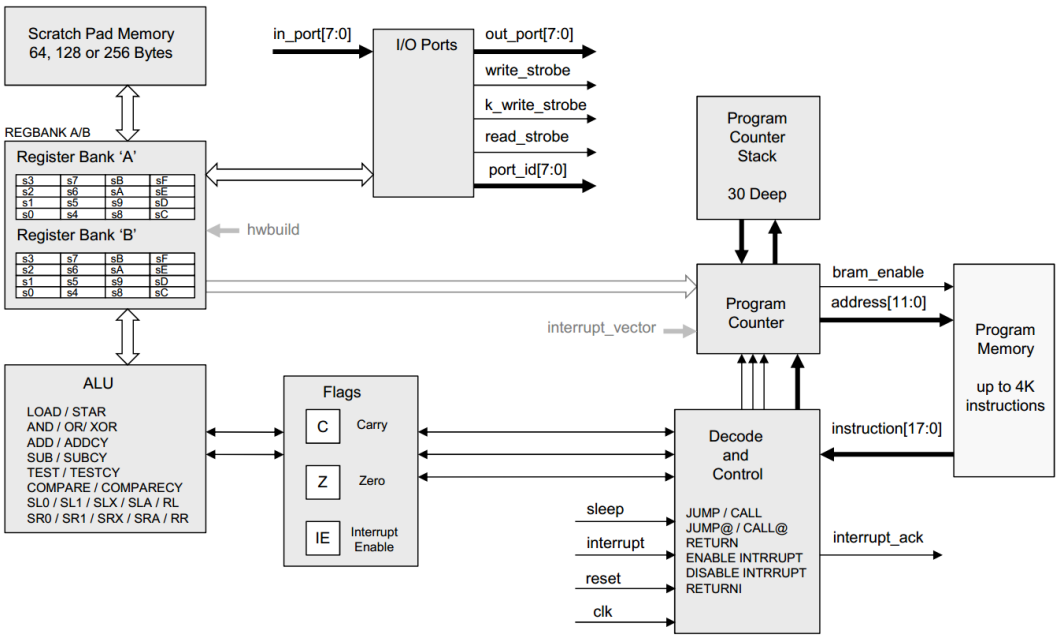
  INTERRUPT ENABLE, `IE`)
- programový čítač se stejnými vlastnostmi jako u PicoBlaze 3
- až 256 vstupních a 256 výstupních nebo kobminovaných vstupně-výstupních 8bitových portů se stejnými vlastnostmi jako u PicoBlaze 3
- zásobník volání podprogramů umožňující až 30 zanořených volání podprogramů. Zásobník je automatický a přímo nelze měnit hodnotu ukazatele vrcholu zásobníku.
- vstup pro externí přerušení
- vstup pro reset mikroprocesoru
- vstup pro přepnutí mikroprocesoru do režimu spánku umožňující snížit spotřebu energie. Pokud je signál sleep nastaven, mikroprocesor dokončí právě prováděnou instrukci a přepne se do režimu spánku. V režimu spánku jsou ignorovány všechny vstupy kromě vstupu reset. Po probuzení z režimu spánku pokračuje mikroprocesor v provádění programu.

2.3 Rozdíly mezi PicoBlaze 3 a PicoBlaze 6

Výše byly popsány hlavní charakteristiky mikroprocesorů PicoBlaze 3 a 6. Nyní budou shrnuty jednotlivé rozdíly mezi těmito mikroprocesory.

Z hlediska harwarových rozdílů má mikroprocesor PicoBlaze 6 oproti PicoBlaze 3 navíc jeden vstupní pin `sleep` a 2 výstupní piny `k_write_strobe` a `bram_enable`. Výstup `k_write_strobe` slouží podobně jako výstup `write_strobe` k určení platnosti výstupních dat. Výstup `k_write_strobe` je však nastavován instrukcí `OUTPUTK`, narozdíl od výstupu `write_strobe`, který je nastavován instrukcí `OUTPUT`. Výstup `bram_enable` slouží k aktivaci či deaktivaci programové paměti. Při aktivním signálu reset nebo sleep je tento signál nastaven na 0, čímž dojde k deaktivaci programové paměti a k dalšímu snížení spotřeby mikroprocesoru.

Podstatným rozdílem u PicoBlaze 6 je možnost parametrizace velikosti programové paměti, RAM paměti, vektoru přerušení a definice vestavěné hardwarové konstanty. PicoBlaze



Obrázek 2.2: Architektura mikroprocesoru PicoBlaze 6

3 podporuje program maximálně o 1024 instrukcích, kdežto PicoBlaze 6 až o 4096 instrukcích. Z tohoto důvodu musela být rozšířena adresace programové paměti z 10 na 12 bitů. PicoBlaze 6 umožňuje nastavit velikost paměti RAM na 64, 128 nebo až 256 bajtů. Adresace paměti RAM tak byla u PicoBlaze 6 rozšířena z 6 na 8 bitů. Z důvodu kompatibility jsou výchozí hodnoty velikosti programové paměti a paměti RAM u PicoBlaze 6 shodné s PicoBlaze 3 - programová paměť pro 1024 instrukcí, pamět RAM o velikost 64 bajtů a vektor přerušení s hodnotou 0x3FF. Definice harwarové konstanty je pouze vlastnost PicoBlaze 6 a konstanta může být definována jako libovolná 8bitová hodnota.

Výše uvedené rozdíly se týkají spíše hardwarových vlastností mikroprocesorů. Rozdíly jsou však i v oblastí vývoje softwaru pro tyto mikroprocesory. PicoBlaze 6 podporuje všech 30 instrukcí, jako PicoBlaze 3. PicoBlaze 6 přidává do instrukční sady 9 nových instrukcí (`TESTCY`, `COMPARECY`, `REGBANK`, `STAR`, `OUTPUTK`, `JUMP@`, `CALL@`, `LOAD&RETURN`, `HWBUILD`). Programy napsané pro PicoBlaze 3 je však nutné znovu přeložit do strojového kódu assemblerem pro PicoBlaze 6. Mikroprocesory sice podporují stejné instrukce, ale kódy instrukcí se pro jednotlivé mikroprocesory liší.

Provedení instrukcí na obou mikroprocesorech je stejné kromě dvou instrukcí - `ADDCY` a `SUBCY`. Rozdíl je ve způsobu určování nulového příznaku `Z`. U PicoBlaze 3 je příznak nastaven, pokud je výsledek operace `ADDCY` nebo `SUBCY` rovný nule. U PicoBlaze 6 je nulový příznak nastaven v případě, že výsledek operace `ADDCY` nebo `SUBCY` je roven nule a zároveň nulový příznak byl nastaven před provedením instrukce `ADDCY` nebo `SUBCY`.

Dalším podstatným rozdílem je, že PicoBlaze 6 disponuje dvěma sadami registrů. Celkem obsahuje tedy 32 registrů. Nicméně aktivně lze používat vždy pouze jednu sadu registrů a mezi sadami se přepíná pomocí speciální instrukce.

Poslední změnou je velikost a chování zásobníku volání podprogramů. PicoBlaze 3 podporuje až 31 vnořených volání podprogramů a nijak nedetekuje přetečení či podtečení zásobníku. PicoBlaze 6 podporuje pouze 30 vnořených volání, ale provádí detekci přetečení a podtečení. V případě detekce jednoho z těchto stavů dojde k resetování mikroprocesoru.

2.4 Dialekty jazyka symbolických instrukcí

Pro mikroprocesor PicoBlaze existují dva dialekty jazyka symbolických instrukcí - KCPSM a pBlazIDE. KCPSM je jazyk navržený společností Xilinx a dialekt pBlazIDE byl navržen společností Mediatronix pro použití v jejich vývpojovém prostředí pBlazIDE (viz 2.6.1). Oba dialekty jsou si však velice podobné a liší se pojmenováním a syntaxí několika instrukcí a direktiv assebmleru. Přehled rodílů mezi dialekty je uveden v tabulce 2.1.

Dialekt KCPSM	Dialekt pBlazIDE
ADDCY	ADDC
COMPARE	COMP
DISABLE INTERRUPT	DINT
ENABLE INTERRUPT	EINT
FETCH sX, (sY)	FETCH sX, sY
INPUT sX, (sY)	IN sX, sY
INPUT sX, kk	IN sX, kk
OUTPUT sX, (sY)	OUT sX, sY
OUTPUT sX, kk	OUT sX, kk
RETURN	RET
RETURN C	RET C
RETURN NC	RET NC
RETURN NZ	RET NZ
RETURN Z	RET Z
RETURNI DISABLE	RETI DISABLE
RETURNI ENABLE	RETI ENABLE
STORE sX, (sY)	STORE sX, sY
SUBCY	SUBC
ADDRESS <adresa>	ORG $<adresa>
NAMEREG <registr>, <nové jméno registru>	<jméno registru> EQU <registr>
CONSTANT <název konstanty>, <hodnota>	<název konstanty> EQU $<hodnota>

Tabulka 2.1: Rozdíly v názvech a syntaxi instrukcí a direktiv assembleru

2.5 Assemblery pro mikroprocesor PicoBlaze

Pro mikroprocesor PicoBlaze existuje několik volně dostupných assemblerů, které se liší podporovanými verzemi PicoBlaze, dialektů a možností použití na různých platformách. Všechny assemblery jsou schopny na základě VHDL nebo Verilog šablony vygenerovat hardwarový popis mikroprocesoru PicoBlaze. Dále generují obsah programové paměti v HEX formátu a soubor s mapováním instrukcí zdrojového kódu na adresy v programové paměti.

2.5.1 KCPSM3 a KCPSM6

KCPSM3 a KCPSM6 jsou oficiální assemblery společnosti Xilinx pro mikroprocesory PicoBlaze 3 a PicoBlaze 6. Jsou k dispozici pouze v binární podobě jako konzolová aplikace pro operační systém Windows. Tyto assemblery podporují pouze dialekt KCPSM.

2.5.2 pBlazASM

PBlazASM [7] je open-source assembler společnosti Mediatronix pro mikroprocesory PicoBlaze 3 a 6. PBlazASM je primárně určen pro operační systém Windows. Při překladu pro PicoBlaze 3 dokáže pBlazASM pracovat s oběma dialekty. Při překladu pro PicoBlaze 6 pak jen s dialektem pBlazIDE.

Na základě dostupnosti zdrojových kódů byla prověřena možnost kompilace a spuštění assembleru pBlazASM i v operačním systému Linux. S drobnými úpravami je možné assembler zkompilovat a spustit i na tomto operačním systému.

2.5.3 PicoAsm

Dalším open-source assemblerem určeným primárně pro operační systém Linux je PicoAsm [1]. Picoasm je určen pouze pro PicoBlaze 3 a umí pracovat pouze s dialektem KCPSM. Tento assembler není dále vyvíjen.

2.5.4 OpenPICIDE

OpenPICIDE [17] je vývojové prostředí pro mikroprocesor PicoBlaze pro platformy Windows, Linux a Mac. Nejde tedy o samostatný assembler ale o komplexnější open-source projekt, v němž je integrovaný i assembler pro PicoBlaze 3 a 6. OpenPICIDE podporuje oba dialekty jazyka symbolických instrukcí.

2.6 Simulátory mikroprocesoru PicoBlaze

V následující části jsou stručně popsány simulátory nebo vývojová prostředí dostupná pro simulátor PicoBlaze.

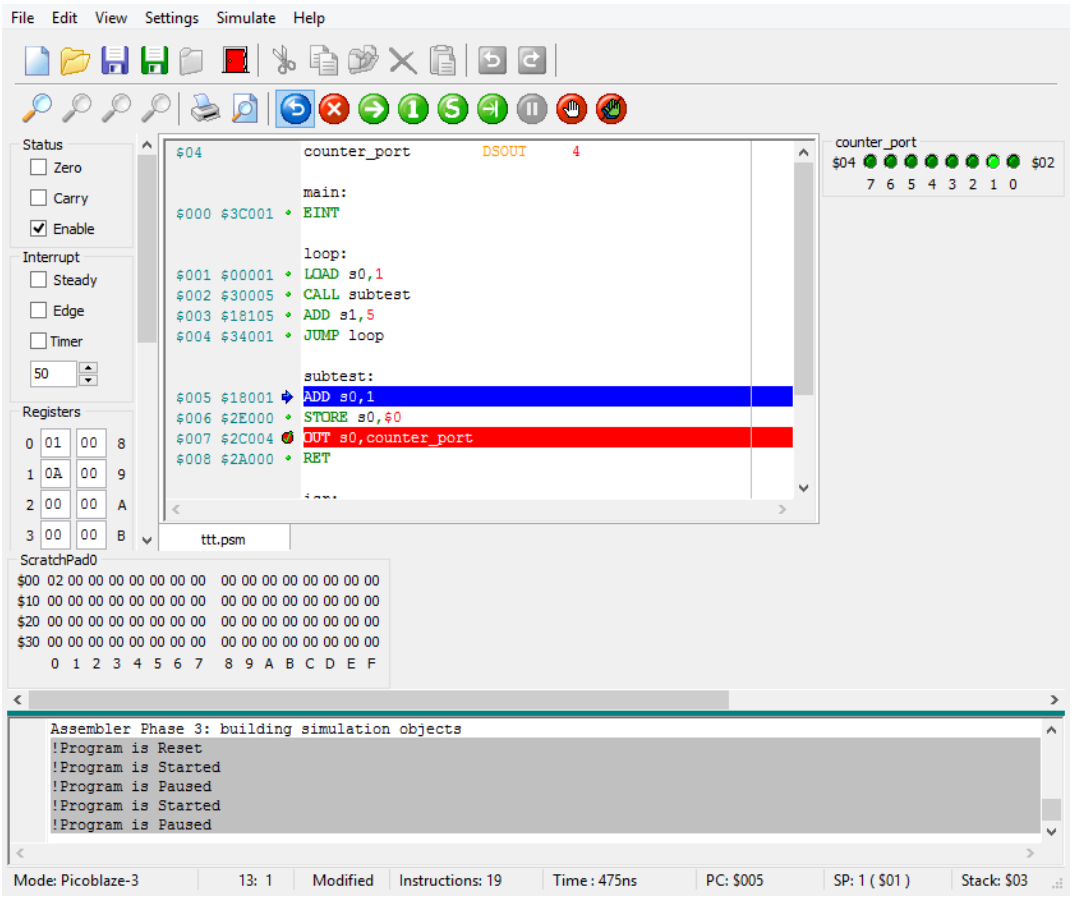
2.6.1 pBlazIDE

PBlazeIDE [8] je dnes již nepodporované integrované vývojové prostředí společnosti Mediatronix pro mikroprocesor PicoBlaze 3 a starší. Poskytuje editor kódu se zvýrazňováním syntaxe a simulátor.

V režimu simulace jsou v pBlazeIDE zobrazovány hodnoty registrů, příznaků a paměti RAM s možností přímé změny hodnoty. Dále je zobrazován obsah zásobníku volání podprocedur a obsah definovaných vstupně/výstupních portů. Zobrazení jednotlivých částí je však z uživatelského pohledu nejednotné a nepřehledně uspřádáné. V rámci editoru kódu jsou v režimu simulace k jednotlivým instrukcím zobrazovány odpovídající instrukční kódy a adresy v paměti ROM.

Simulátor umožňuje simulaci žádosti o přerušení, a to třemi způsoby - stálá žádost o přerušení, žádost aktivní jen při následujícím hodinovém cyklu a žádost aktivní po zadanou dobu.

K ovládání běhu simulace lze použít běh k bodu přerušení, běh na místo označené kurzorem, pozastavení simulace, krokování po instrukcích a krokování s přeskočením volání podprogramů a skoků (krok na následující řádek ve zdrojovém souboru). Při simulaci není možné editovat program. Pro editaci je nutné simulaci ukončit a po editaci znovu spustit.



Obrázek 2.3: Vývojové prostředí pBlazIDE

2.6.2 pBlazSIM

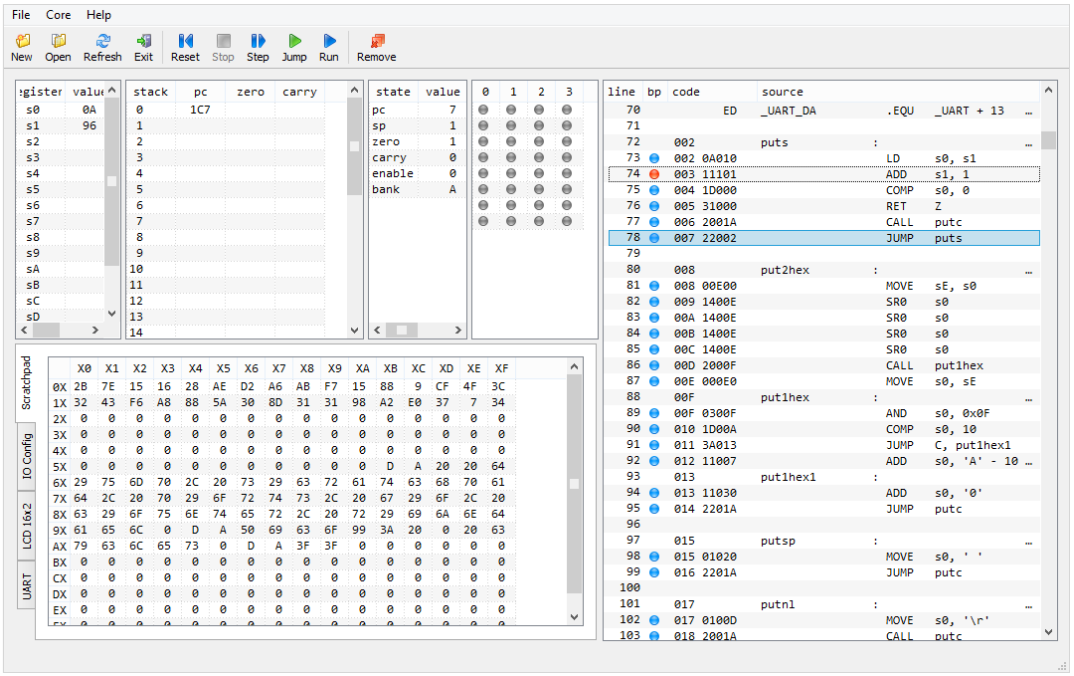
PBlazSIM [9] je nástroj vyvíjený taktéž společností Mediatronix a jde nástupce vývojového prostředí pBlazIDE. I když je PBlazSIM označován jako nástupce pBlazIDE, nejde o vývojové prostředí, ale pouze o grafický simulátor. PBlazSIM podporuje mikroprocesor PicoBlaze 3 i 6 a simulace je prováděna na základě `.lst` souboru a volitelně `.scr` souboru s obsahem paměti RAM.

Z hlediska uživatelského prostředí je pBlazSIM přehlednější než pBlazIDE v režimu simulace, ale v některých oblastech nedisponuje takovou funkcionalitou jako pBlazIDE. Není např. možné nastavit parametry mikroprocesoru PicoBlaze nebo nelze simulovat žádost o přerušení.

pBlazSIMcl

V době psaní této práce byl simulátor pBlazSIM rozšířen také o možnost spouštění jako konzolové aplikace bez grafického uživatelského rozhraní. Toto rozšíření se nazývá pBlazSIMcl a má umožnit tvorbu automatizovaných testů [6].

pBlazSIMcl má však oproti grafickému pBlazSIM omezenou funkčnost. Vstupem simulátoru pBlazSIMcl je soubor `.lst`, ve kterém je uložen mimojiné obsah programové paměti, a volitelně soubor `.scr` nesoucí obsah paměti RAM. pBlazSIMcl také neumožňuje simulaci přerušení a pomocí implementované vstupně výstupní komponenty UART dokáže pouze



Obrázek 2.4: Simulátor pBlazSIM

vypsat výstupy mikroprocesoru PicoBlaze na standardní výstup při instrukci `OUTPUT`, a to pouze na portech `0xEC` a `0xED`. Pro instrukci `INPUT` není možné zadat vstupní hodnoty. Při simulaci nebo po skončení simulace (automaticky je skončena pouze při nestandardním stavu mikroprocesoru) není možné zjistit vnitřní stav mikroprocesoru. Vzhledem k uvedeným nedostatkům je možné pBlazSIMcl použít pouze pro simulaci a testování jednoduchých programů.

2.6.3 KPicoSim

KPicoSim [2] je open-source vývojové prostředí pouze pro mikroprocesor PicoBlaze 3 určené pro operační systém Linux a prostředí KDE. Kpicosim poskytuje podobnou funkcionalitu jako pBlazIDE a není dále vyvíjen.

KPicoSim obsahuje editor zdrojového kódu se schopností zvýrazňování syntaxe, integrovaný překladač picoasm a simulátor.

KPicoSim při simulaci poskytuje zobrazení hodnot registrů a příznaků, RAM paměti a vstupně/výstupních portů.

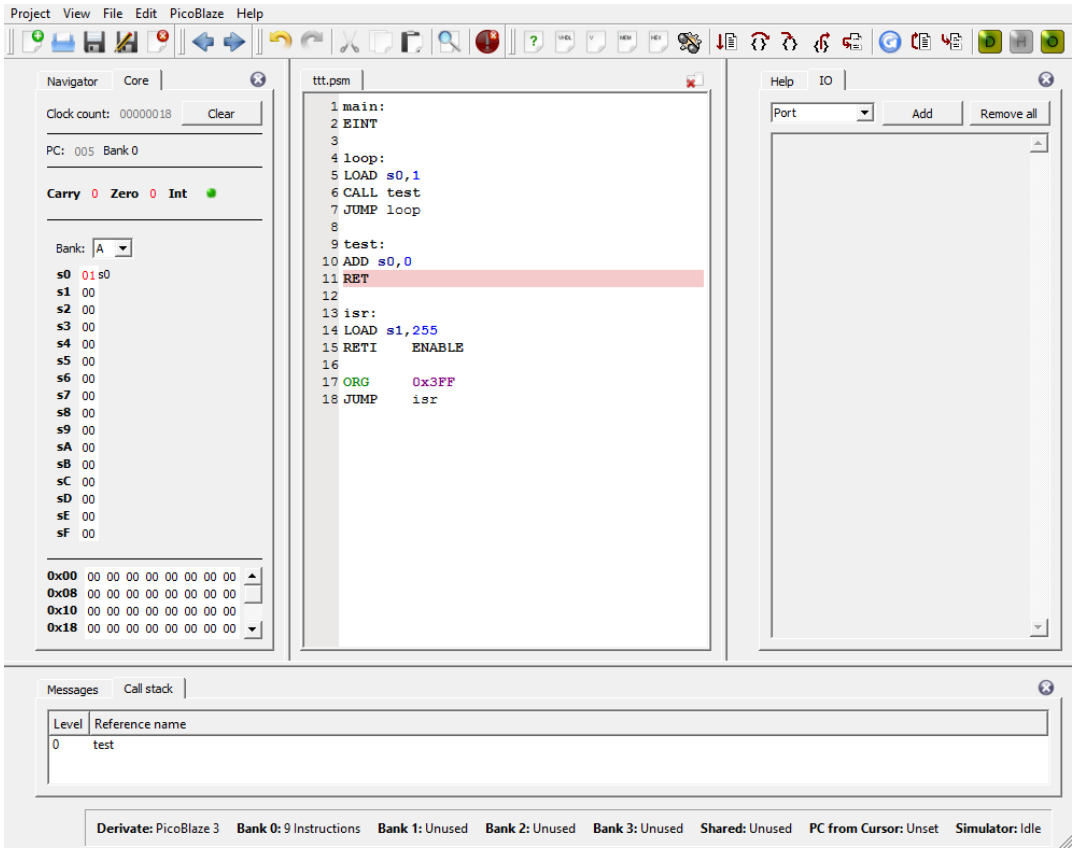
2.6.4 OpenPICIDE

OpenPICIDE [17] je další open-source vývojové prostředí pro mikroprocesor PicoBlaze. OpenPICIDE je k dispozici pro platformy Linux. Windows i Mac a umožňuje vývoj a simulaci programů pro PicoBlaze 3 a 6 i starší verze mikroprocesoru PicoBlaze.

OpenPICIDE poskytuje editor zdrojového kódu se schopností zvýrazňování syntaxe, správu projektů, možnost volby parametrů simulovaného mikroprocesoru PicoBlaze, export programu do `.hex`, `.mem` nebo `.vhdl` souboru a v neposlední řadě také propracovaný simulátor.

OpenPICIDE v režimu simulace zobrazuje hodnoty registrů, příznaky a paměť RAM s možností volby zobrazení hodnot v desítkové, osmičkové nebo šestnáctkové soustavě. Hodnoty registrů a paměti lze během simulace uživatelsky přímo měnit. V případě, že instrukce ovlivňuje hodnotu registru, příznaku nebo paměťové buňky, je uživatel na změnu upozorněn barevným zvýrazněním příslušné hodnoty. Kromě toho OpenPICIDE zobrazuje i zásobník volání podprogramů a zobrazení vstupně/výstupních portů.

Simulátor poskytuje standardní funkčnost podporující ladění (body přerušení, krokování po instrukcích, zvýraznění prováděné instrukce). Navíc umožnuje i krokování po řádcích ve zdrojovém souboru programu, kdy jsou volání a provedení podprogramu bráno jako jedna instrukce. Podprogram je v takovém případě proveden, ale simulace je pozastavena až po provedení celého podprogramu. Dalším formou krokování je běh simulace do konce podprogramu (po instrukci `RETURN`). Simulátor implementuje i žádost o přerušení, reset mikroprocesoru a během simulace je možné upravit simulovaný program a do simulátoru jej znovu nahrát, přičemž stav mikroprocesoru zůstane zachován.



Obrázek 2.5: Vývojové prostředí OpenPICIDE

2.6.5 Simulace v Modelsim

Výše uvedené simulátory jsou specializovány přímo pro mikroprocesor PicoBlaze. Vzhledem k tomu, že PicoBlaze je softwarový mikroprocesor určený pro použití v FPGA čipech a assemblery pro PicoBlaze umožňují vygenerování komponenty popisující mikroprocesor v jazyce VHDL nebo Verilog, je možné simulaci provádět i v pokročilém nástroji ModelSim [20]. Jedná se však o nízkoúrovňovou simulaci na úrovni signálů a pro testování pouze

programů pro mikroprocesor PicoBlaze jde o zbytečně složité řešení. ModelSim je naopak vhodné využít např. pro otestování komunikace mikroprocesoru PicoBlaze s externími komponentami pomocí vstupně/výstupních portů a žádostí o přerušení.

Kapitola 3

Small Device C Compiler (SDCC)

Small Device C Compiler (SDCC) [4, 15] je volně dostupný překladač jazyka C (či přesněji sada nástrojů) určený zejména pro 8bitové mikroprocesory. Na rozdíl od jiných volně dostupných překladačů jazyka C (například GCC) určených pro obecné procesory, je SDCC navržen dle specifických potřeb a omezení 8bitových mikroprocesorů. Ty používají ve většině případů Harwardskou architekturu, která odděluje kód a data programu do samostatných adresových prostorů. Pro podporu těchto odlišností zavádí SDCC do jazyka C nová klíčová slova a konstrukce. SDCC oficiálně podporuje tyto rodiny mikroprocesorů:

- Intel MCS51
- Maxim DS80C390
- Freescale HC08
- Zilog Z80
- PIC16 a PIC18 (ve vývoji)

Díky své struktuře však může být rozšířen o podporu dalších mikroprocesorů. Jednotlivá rozšíření pro mikroprocesory se nazývají porty.

Součástí SDCC je mimo jiné i debugger SDCDB [3] umožnující ladění a testování programů v příkazovém řádku. Pomocí SDCDB je možné ladit programy na úrovni kódu v jazyce C nebo na úrovní jazyka symbolických instrukcí. Při ladění využívá debugger SDCDB jednak přeložený program na instrukce mikroprocesoru, a dále soubor .cdb generovaný překladačem a linkerem, který obsahuje informace o symbolech v překládaném programu. Pro provádění programu využívá SDCDB samostatný simulátor ucSim, který bude blíže popsán v následující části. SDCDB tak slouží jako rozhraní, pomocí kterého je spouštěn a ovládán simulátor ucSim.

SDCC je sice primárně určen pro systém Linux (resp. obecněji systémy UNIX), ale je možné jej přeložit a spustit i na systému Windows.

3.1 PicoBlaze C Compiler (PBCC)

PicoBlaze C Compiler (PBCC) [5, 18] je port překladače SDCC pro mikroprocesor PicoBlaze vytvořený na Fakultě informačních technologií Vysokého učení technického v rámci projektu VLAM. PBCC je založen na SDCC verze 3.0.1 a v současnosti je podporován pouze PicoBlaze 3. Překlad je možný do obou dialektů jazyka symbolických instrukcí.

Vzhledem k minimální velikosti mikroprocesoru PicoBlaze plyne pro překladač PBCC několik omezení. V rámci PBCC není implementován linker. Lze však použít vkládání hlavičky nebo zdrojového kódu pomocí direktivy `#include`. Pro PicoBlaze byla implementována pouze malá podmnožina některých standardních knihoven jazyka C (např. `ctype`, `time`, `math` nebo `stdlib`). Implementace dalších knihoven, jako např. `stdio` nebo `string`, by se kvůli velikosti do programové paměti mikroprocesoru nevlezla. Jelikož je PBCC stále ve vývoji, překladač neobsahuje následující funkčnost:

- dynamická alokace pomocí funkcí malloc a free
- klíčová slova `__reentrant` a `__data __at` definované v rámci SDCC
- vkládání funkcí označené klíčovým slovem inline
- problémy při předávání polí jako argumenty funkcí
- pouze částečná podpora globálních proměnných

3.2 μCsim

μCsim [13] je open-source nástroj sloužící k simulaci mikroprocesorů. Oficiálně podporuje tyto rodiny mikroprocesorů:

- Intel MCS51
- Zilog Z80
- Freescale HC08
- Atmel AVR
- Philips XA

μCsim je k dispozici pro platformu UNIX ve formě konzolové aplikace, která umožňuje simulaci všech výše uvedených mikroprocesorů. Dále je k dispozici verze pro platformu DOS disponující grafickým uživatelským rozhraním, ale podporující pouze mikroprocesory rodiny MCS51. μCsim jako samostatný nástroj se však od roku 2004 prozatím nedočkal dalšího vývoje. μCsim však byl integrován jako jedna z částí SDCC a v rámci vývoje SDCC je rozšiřován i simulátor μCsim.

V následující části jsou popsány implementační detaily simulátoru μCsim. Popis se vztahuje na μCsim dostupný v SDCC verze 3.0.1. K simulátoru μCsim existuje pouze uživatelský manuál a většina informací popsaných v následující části byla zjištěna na základě zdrojových kódu simulátoru.

3.2.1 Logická struktura μCsim

Z hlediska struktury je možné μCsim rozdělit na tři hierarchické logické části:

1. aplikace
2. simulátor
3. mikroprocesor (mikrokontroler)

Aplikace je definována třídou `cl_app` a zastřešuje všechny další části. V rámci inicializace aplikace probíhá zpracování parametrů, inicialializace a nastavení vlastností aplikace (třídy `cl_options` a `cl_option`), inicializace podporovaných příkazů (`cl_cmdset` a `cl_cmd`) a subsystému pro zpracování příkazů (`cl_commander`). Aplikace může být spuštěna jako konzolová aplikace nebo jako server naslouchající na zadaném portu na příchozí spojení, jimiž je aplikace ovládána stejně jako v případě konzolové varianty. Při spuštění jako server může být k simulátoru připojeno nezávisle více klientů.

Simulátor (třída `cl_sim`) je část provádějící a ovládající simulaci mikroprocesoru. Hlavní programová smyčka, kdy je buď čekáno na vstup uživatele nebo je prováděna simulace programu instrukce po instrukci, dokud nedojde k zastavení simulace výskytem bodu přerušení, k neočekávané situaci v simulovaném mikroprocesoru (např. nerozpoznaná instrukce), nebo ke vstupu uživatele, může být definována jak v aplikační třídě (metoda `cl_app::run()`), tak ve třídě simulátoru (`cl_sim::main()`). Je pak na tvůrci konkrétního portu, kterou možnost využije, ale většina portů využívá tu v aplikační třídě.

Jádro simulátoru pak tvoří samotný mikroprocesor definovaný třídou `cl_uc` definující architekturu a funkčnost simulovaného mikroprocesoru nebo rodiny mikroprocesorů.

3.2.2 Vlastnosti aplikace

Při spuštění simulátoru mohou být pomocí parametrů nastaveny některé vlastnosti aplikace. Jedná se například o číslo portu, na kterém má simulátor naslouchat, cesta k souboru s obsahem paměti ROM, typ procesoru apod. Každá vlastnost je identifikována svým jedinečným názvem, pomocí kterého je možné se na vlastnost odkazovat. Nové vlastnosti je možné vytvořit a přidat kdekolik v programu, ale standardně jsou vlastnosti jsou vytvářeny buď v metodě `cl_appp::mk_options()`, nebo v metodě `cl_app::proc_arguments()` určené pro zpracování parametrů.

Vlastnost je reprezentovaná bázovou třídou `cl_option`, ze které jsou podle typu odvozeny konkrétní třídy `cl_bool_option`, `cl_string_option`, `cl_pointer_option`, `cl_number_option` a `cl_float_option` nesoucí popořadě logickou a řetězcovou hodnotu, ukazatel na soubor, celočíselnou nebo desetinnou hodnotu. Seznam vlastností je uložen v rámci třídy `cl_options` a v aplikaci (třídě) je dostupný v proměnné `options`.

3.2.3 Příkazy simulátoru

Jak bylo uvedeno výše, v rámci spouštění aplikace probíhá inicializace příkazů ovládajících aplikaci. Systém je však navržen tak, že aplikace může být jednoduše rozšířena o další příkazy. Nové příkazy nemusí být přidáváný pouze při inicializaci aplikace, ale mohou být přídány při inicializaci simulátoru nebo mikroprocesoru (metody `build_cmdset()`). Každý příkaz může mít určený rámec své působnosti, tzn. která část aplikace je příkazem ovlivňována. Možnosti jsou aplikace, simulátor nebo mikroprocesor. Příkazy mohou být definovány jako jednoduché příkazy (např. `run`, nebo seskupovány do složených příkazů (např. `info hardware`, `info memory`). Jednoduché příkazy jsou pak rozšířením třídy `cl_cmd` a složené příkazy (příkaz `info` uvedený jako příklad výše) jsou rozšířením třídy `cl_super_cmd`. Třída `cl_super_cmd` obsahuje v proměnné typu `cl_cmdset` seznam podřízených příkazů, což může být jednoduchý příkaz (z příkladu výše příkazy `hardware` `memory` nebo dále složený příkaz.

Pro každý jednoduchý příkaz musí být definována samostatná třída implementující funkčnost příkazu. Dle zavedené jmenné konvence jsou třídy reprezentující příkazy pojmenovávány `cl_<celý název příkazu>_cmd`. Pro zjednodušení implementace je možné pou-

žít makra `COMMAND` nebo `COMMAND_ON` pro deklaraci třídy a makra `COMMAND_DO_WORK` nebo `COMMAND_DO_WORK_{UC, SIM, APP}` pro definici třídy a metody `do_work()`, ve které je implementována funkčnost příkazu.

3.2.4 Paměť mikroprocesoru

Simulovaná paměť mikroprocesoru se skládá ze tří částí:

- adresový prostor
- paměťový čip
- dekodér adres

Adresový prostor je definován třídou `cl_address_space` a je určen svým identifikátorem, počáteční adresou, velikostí a šířkou. V μCsim jsou v souboru `stype.h` definovány identifikátory těchto základních adresových prostorů (výčet `mem_class`, konstanty `MEM_xxx_ID`:

- programová paměť (rom)
- speciální funkčí registry (sfr)
- interní RAM (iram)
- externí RAM (xram)
- jiná interní paměť (ixram)

Hodnoty simulované paměti nejsou uloženy přímo v adresovém prostoru, ale jsou ukládány v paměťových čipech. Každý adresový prostor může být tvořen několika paměťovými čipy. Paměťové čipy jsou definované třídou `cl_memory_chip` a podobně jako u adresovým prostorů musí být určen jejich identifikátor, velikost a šířka.

Pro přiřazení paměťových čipů do adresových prostorů slouží dekodéry adres. Ty jsou reprezentovány třídou `cl_address_decoder`. Při vytváření dekodéru adres musí být kromě odkazu na adresový prostor a paměťový čip definován rozsah adres z adresového prostoru (počáteční a koncová adresa), do kterého má být paměťový čip mapován, a počáteční adresa v paměťovém čipu.

Pro inicializaci pamětí simulovaného mikroprocesoru je určena metoda `cl_uc::make_memories()`.

Ilustrativní příklad definice 128 bajtové paměti RAM může vypadat následovně:

```
class cl_address_space *ram;
class cl_memory_chip *chip;
class cl_address_decoder *ad;

ram = new cl_address_space(MEM_IRAM_ID, 0, 128, 8);
ram->init();
address_spaces->add(ram);

chip = new cl_memory_chip("ram_chip", 128, 8);
chip->init();
```

```
memchips->add(chip);

ad = new cl_address_decoder(ram, chip, 0, 127, 0);
ad->init();
ram->decoders->add(ad);
ad->activate(0);
```

Práce s pamětí, například v implementaci instrukcí, probíhá výhradně přes adresové prostory. Hodnoty v paměti je možné číst a nastavovat přímo, nebo přes třídu `cl_memory_cell` reprezentující paměťovou buňku. S hodnotami se manipuluje pomocí metod `get()` a `set()` nebo `read()` a `write()`. Jediný rozdíl mezi těmito metodami je ten, že u metod `read()` a `write()` může být zaznamenáván počet přístupů k paměti. Při přístupu k paměti neprobíhá kontrola, zda požadovaná adresa spadá do rozsahu adresového prostoru.

Při simulaci lze pomocí příkazů získat nebo nastavit hodnoty v paměti nebo i jednotlivých bitů paměťových buněk. Lze také vypsat kompletní obsah paměti.

3.2.5 Rozhraní mikroprocesoru

μCsim umožňuje u simulovanému mikroprocesoru definovat hardwarové komponenty a vstupně/výstupní rozhraní. Každá komponenta nebo rozhraní je definováno jako hardwarový element a je reprezentováno třídou `cl_hw`. Hardwarový element musí mít určenu kategorii, do které spadá. Kategorie jsou definované výčtem `hw_cath` a jedná se například o časovač, sériové rozhraní, port, žádost o přerušení apod. Hardwarový element musí být dále označen jednoznačným řetezovým identifikátorm (např. "port") a doplňkovým číselným identifikátorem (např. 0 a 1 pro rozlišení dvou vstupních portů).

Seznam hardwarových elementů je udržován v třídě mikroprocesoru v proměnné `hws` a vytvoření a inicializace jednotlivých harwdarových elementů probíhá v metodě `cl_uc::mk_hw_elements()`.

3.2.6 Registry

Základní třída mikroprocesoru `cl_uc` definuje pouze jeden interní registr, a to programový čítač `PC`. Ostatní registry mikroprocesoru jsou definovány až v jednotlivých portech. K simulaci registrů jsou v existujících portech použity dle architektury mikroprocesoru dva různé způsoby. U obou je však zachována stejná konvence a definice typů, konstant a maker týkajících se registrů jsou uvedeny v souboru `regs<název portu>.h`.

Prvním způsobem je mapování registrů do paměti. K tomuto účelu je určen jeden z typů paměti - paměť pro speciální funkční registry (special function registers, SFR). Pro jednotlivé registry jsou definovány konstanty, jejichž jméno je shodné s názvem registru a hodnota konstanty představuje adresu v SFR paměti, kde je uložena hodnota registru. K registrům se pak přistupuje pomocí stejných metod jako k paměti.

Při mapování registrů do SFR paměti je možné nastavovat hodnoty registrů během simulace pomocí příkazu simulátoru `set memory sfr`. Aby bylo možné v příkazu použít název registru a ne pouze jeho adresu v SFR paměti, je nutné implementovat překladovou tabulku. K vytvoření překladové tabulky je určena struktura `name_entry`. Položky struktury jsou typ mikroprocesoru, adresa registru v SFR paměti a jeho název. Překladová tabulka je zpravidla definována v souboru `glob.cc` a ukazatel na tuto tabulku musí být

předán metodou `cl_uc::sfr_tbl()`. Samotný překlad názvu registru na adresu je implementován v jádru μCsim. Podobným způsobem mohou být pojmenovány i jednotlivé bity v registrech (např. v registru příznaků).

Druhým způsobem implementace registrů (použitým např. u portu hc08 nebo z80) je definice vlastní struktury, jejíž položky symbolizují konkrétní registry. V třídě mikroprocesoru je pak definována proměnná, přes kterou je přistupováno k registrům. Nevýhodou tohoto řešení je, že při simulaci nemůže být pomocí příkazů simulatoru přistupováno k jednotlivým registrům nebo dokonce jejich bitům. Pomocí příkazu `info registers` může být pouze vypsán stav všech registrů.

U obou způsobů je nutné implementovat metodu `cl_uc::print_regs()`, která je volána po zadání příkazu `info registers` a slouží k výpisu hodnot všech registrů.

3.2.7 Instrukční sada

Implementace instrukční sady v simulátoru je rozdělena do několika částí. První z nich je definice tabulka všech podporovaných instrukcí. Tabulka instrukcí je zpravidla definována v souboru `glob.cc` jako pole struktur `dis_entry`. Struktura představuje záznam o jedné instrukci a obsahuje tyto položky:

- instrukční kód
- bitovou masku
- typ skoku
- délku instrukce
- název instrukce

Bitová maska určuje, které bity z instrukčního kódu identifikují danou instrukci. Typ skoku slouží pro účely analyzátoru kódu. Délka instrukce značí, kolik buněk v paměti ROM instrukce zabírá včetně operandů. Název instrukce označuje pojmenování instrukce v jazyce symbolických instrukcí, který je uživateli zobrazován během simulace. V názvu instrukce se mohou nacházet i formátovací značky ve tvaru `\%<znak>`. Místo formátovacích značek jsou při zpracování instrukce doplněny hodnoty operandů, názvy registrů apod. Formátovací značky nejsou nijak standartizované a jejich volba je na autorovi. Po každou značku však musí být definován požadovaný výstup v metodě `cl_pblaze::disass()`. V rámci tvorby portu může dle podporovaných typů mikroprocesoru existovat i více tabulek instrukcí.

Další částí je implementace funkčnosti jednotlivých instrukcí. V třídě simulátoru je zpravidla pro každou instrukci nebo skupinu instrukcí, které se liší např. pouze svými operandy, definována metoda s názvem `inst_<název instrukce>` a v rámci této metody implementována funkčnost instrukce. Deklarace a definice těchto metod je dle zavedené konvence umístěna v samostatných souborech `instcl.h` a `inst.cc`. V případě většího počtu instrukcí je u některých portů z důvodu větší zvoleno ještě jemnější členění do více souborů.

Poslední částí je rozhodnovací mechanizmus, na jehož základě se pro aktuálně prováděnou instrukci zavolá metoda implementující její funkčnost. Rozhodování probíhá v metodě `cl_uc::exec_inst()`, kde je na základě instrukčního kódu volána patřičná metoda implementující chování instrukce.

3.2.8 Průběh simulace

Spuštěním simulace příkazem `run` se přepne simulace do stavu běhu (stav `SIM_GO`), kdy jsou prováděny jednotlivé instrukce programu. Běžící simulace může být přerušena uživatelským vstupem, dosažením bodu přerušení nebo výskytem nestandardní nebo chybové situace v mikroprocesoru (např. neplatný instrukční kód, neočekávaný stav při provádění instrukce apod.).

V simulátoru je možné nastavovat dva typy bodů přerušení:

- Bod přerušení na dosažení určitého místa v programu. Jedná se o standardní bod přerušení, který je nastaven na určitou adresu v programu a při dosažení tohoto místa je běh pozastaven.
- Událostní bod přerušení. Tento typ bodu přerušení se vztahuje k paměti na pomocí něj může být běh pozastaven při čtení nebo zápisu hodnoty z určitého místa v paměti.

K provedení instrukce slouží metoda `cl_uc::do_inst()`. V rámci ní jsou pak volány metody `pre_inst()`, `exec_inst()` a `post_inst()`. Samotné provedení instrukce probíhá v metodě `exec_inst()`, kde je na základě hodnoty programového čítače nejprve zjištěno, zda pro aktuální adresu není nastaven bod přerušení. Pokud ano, simulace je pozastavena. V opačném případě je z paměti načten kód aktuální instrukce a simulováno její provedení. Dle výsledku provedení instrukce pak simulace pokračuje dále nebo je pozastavena. Metody `pre_inst()` a `post_inst()` mohou sloužit k provedení podpůrných akcí v rámci simulátoru (např. vyhodnocení chyb po provedení instrukce).

Základní implementace výše uvedených metod ve třídě `cl_uc` neposkytuje podporu zpracování žádostí o přerušení. Toto rozšíření bylo implementováno v portu s51 tak, že byla přepsána medota `do_inst()` a po provedení instrukce je kontrolováno, zda není aktivní žádost o přerušení. Pokud ano, je žádost o přerušení zpracována.

3.2.9 Simulace času

Z hlediska simulace času pracuje μCsim na nejnižší úrovni s hodinovými takty mikroprocesoru. V μCsim je pro práci s hodinovým taktem určena třída `cl_ticker`. Simulace hodinového taktu se provede voláním metody `tick()` a počet provedených hodinových taktů je uložen v proměnné `ticks`.

Při spuštění simulátoru lze přepínačem `-X` specifikovat frekvenci mikroprocesoru. Na základě frekvence a počtu hodinových taktů pak lze sledovat reálnou celkovou dobu běhu mikroprocesoru od posledního resetování. Kromě celkové doby běhu je v μCsim rozlišena ještě doba strávená obsluhou přerušení a doba nečinnosti mikroprocesoru. Tyto hodnoty je možné při simulaci zobrazit příkazem `state`.

Na úrovni simulace jednotlivých instrukcí se pak nepracuje přímo s hodinovými takty mikroprocesoru, ale s instrukčními cykly. Počet provedených instrukčních cyklů je uložen v proměnné `inst_ticks` třídy `cl_uc` a provedení instrukčního cyklu je simulováno voláním metody `cl_uc::tick()`. V ní je pak proveden odpovídající počet hodinových taktů mikroprocesoru na základě hodnoty vrácené metodou `cl_uc::clock_per_cycle()`, která představuje počet hodinových taktů na jeden instrukční cyklus.

3.2.10 Struktura zdrojových kódů

Zdrojové kódy μCsim se v rámci SDCC nacházejí ve složce `sim/ucsim/`.

- adresář μCsim - obsahuje definice a deklarace týkající se celé aplikace a různé podpůrné třídy a funkce. Důležité soubory jsou:
  - `ddconfig.h` - globální konfigurace dle systémového prostředí. Soubor je generován příkazem `configure` na základě detekovaných vlastností systému.
  - `globals.{h, cc}` a `stypes.h` - v těchto souborech je definována řada struktur, výčtů a konstant pro reprezentaci typů paměti, typů mikroprocesorů, instrukcí, registrů, stavů simulace, stavů mikroprocesoru, výsledků provedení instrukcí, typů hardwarových komponent, bodů přerušení apod.
  - `appcl.h` a `app.cc` - aplikační třída
  - `optioncl.h` a `option.cc` - definice tříd reprezentující vlastnosti aplikace

  jako například aplikační třída (), vlastnosti aplikace ()
- cmd.src - implementace systému pro zpracování příkazů simulátoru a implementace příkazů
- gui.src - k této části zdrojových kódů se mi bohužel nepodařilo zjistit nic bližšího. Zřejmě obsahuje nepoužitou nebo nedokončenou část implementace grafického rozhraní. Existují nedokumentované příkazy simulátoru `gui start` a `gui stop`, jejichž funkčnost však není implmenentována.
- sim.src - obsahuje zdrojové kódy týkající se simulátoru, simulovaného mikroprocesoru a jeho částí. Důležité soubory jsou:
  - `hwcl.h`, `hw.cc` - třídy pro hardwarové komponenty a rozhraní mikroprocesoru
  - `memcl.h`, `mem.cc` - třídy reprezentující paměť, paměťové prostory apod.
  - `simcl.h`, `sim.cc` - implementace simulátoru
  - `uccl.h`, `uc.cc` - implementace mikroprocesoru
- porty - zdrojové soubory každého portu jsou uloženy ve složce s názvem `<název portu>.src`. Struktura zdrojových kódu jednotlivých portů se liší, ale je možné sledovat toto schéma:
  - `<název portu>cl.h`, `<název portu>.cc` - třída mikroprocesoru (rozšiřuje třídu `cl_uc`)
  - `glob.h`, `glob.cc` - deklarace a definice instrukční sady (tabulky instrukcí), pojmenování registrů a bitů
  - `instcl.h`, `inst.cc` - metody implementující funkčnost jednotlivých instrukcí. U některých portů je implementace instrukcí členěna do více samostatných souborů dle logických skupin instrukcí
  - `regs<název portu>.h` - definice registrů nebo jejich adres v SFR paměti
  - `s<název portu>.cc` - soubor obsahující funkci `main()`
  - `sim<název portu>cl.h`, `sim<název portu>.cc` - třída simulátoru (rozšiřuje třídu `cl_sim`)

3.2.11 Tvorba nového portu

Tato část popisuje nejnutnější minimum pro vytvoření nového portu simulátoru μCsim pro jednoduchý mikroprocesor a lze použít jako obecný návod pro vytvoření libovolného portu. Při vytváření nového portu je kromě vytvoření nových souborů nutné modifikovat některé existující soubory jádra simulátoru společného pro všechny porty. Pojmenování souborů, tříd apod. použité v návodu vychází z nastudovaných zvyklostí použitých u existujících portů.

Prvním krokem je vytvoření složky nového portu s názvem `<název portu>.src`, ve které budou uloženy zdrojové kódy tvořeného portu. Ve složce nového portu je nutné vytvořit tyto soubory:

`s<název portu>.cc`
Soubor obsahující funkci `main()`.

`sim<název portu>cl.h, sim<název portu>.cc`
Definice třídy simulátoru s názvem `cl_sim<název portu>` dědící ze třídy `cl_sim`. V třídě simulátoru je nutné předefinovat tyto metody:

- konstruktor
- `mk_controller()` - vytvoření instance mikroprocesoru.

`<název portu>cl.h, <název portu>.cc`
Definice třídy mikroprocesoru s názvem `cl_<název portu>` a dědící ze třídy `cl_uc`. Zde je potřeba jednak deklarovat proměnné pro přístup k adresovým prostorům mikroprocesoru, proměnnou pro přístup k registrům (pokud nejsou mapovány do SFR paměti. Dále musí být předefinovány tyto metody:

- konstruktor - v něm probíhá nastavení typu mikorprocesoru
- `init()` - inicializace paměti, hardwarových komponent apod.
- `id_string()` - indentifikátor mikroprocesoru
- `make_memories()` - vytvoření paměťových prostorů mikroprocesoru
- `mk_hw_elements()` - vytvoření hardwarových komponent a rozhraní mikroprocesoru
- `reset()` - operace a nastavení výchozích hodnot p5i resetování mikroprocesoru
- `exec_inst()` - provedení aktuální instrukce dle hodnoty programového čítače
- `clock_per_cycle()` - určení počtu hodinových taktů na jeden instrukční cyklus
- `inst_length()` - získání délky aktuální instrukce (počet paměťových buněk v paměti ROM)
- `longest_inst()` - délka nejdelší instrukce
- `print_regs()` - vypsání stavu všech registrů
- `dis_tbl()` - získání tabulky instrukcí
- `sfr_tbl()` - získání tabulky názvů a adres registrů
- `bit_tbl()` - získání tabulky názvů a adres bitů

- `get_disasm_info()` - získání informací o aktuálně prováděné instrukci z tabulky instrukcí
- `disass()` - získání textové reprezentace aktuálně prováděné instrukce (její název a skutečné hodnoty parametrů)

`instcl.h, inst.cc`

Definice metod představující jednotlivé instrukce. Název souborů je poněkud zavádějící a funkčnost instrukcí není implementována v samostatné třídě, ale jako součást třídy mikroprocesoru, kde jsou tyto soubory přiloženy pomocí direktivy `#include`.

`glob.h, glob.cc`

Definice tabulky typů mikroprocesoru, tabulky instrukcí, nebo tabulek instrukcí pro jednotlivé typy mikroprocesoru, definice tabulky jmen registrů (pokud jsou registry mapovány do SFR paměti) a definice tabulky bitů.

`regs<název portu>.h`

Definice registrů mikroprocesoru. Dle architektury mikroprocesoru zde jsou v existujících portech definovány buď konstanty představující adresu registru v SFR paměti, anebo struktura s položkami reprezentujícími registry.

`Makefile.in`

Soubor je používán pro vygenerování souboru `Makefile` příkazem `configure`. Nejjednodušší způsobem vytvoření tohoto souboru je jeho zkopírování z existujícího portu a následná modifikace cest k portu a názvů zdrojových souborů. Vzor souboru `Makefile.in` je uveden v příloze B.

Do kódu společného pro všechny mikroprocesoru musí být pouze přidána definice jednotlivých typů mikroprocesorů vytvářeného portu.

`stypes.h`

Definice konstant pro všechny typy mikroprocesoru (konstanty `CPU_<typ>`) a konstanty `CPU_ALL_<název portu>` jako bitového součtu konstant představujících typy mikroprocesoru.

Jako poslední krok je potřeba modifikovat konfigurační soubory používané příkazem `configure` pro globální nastavení simulátoru dle systémových vlastností a vygenerování souborů `Makefile`. Čísla řádků v souborech uvedených níže představují pouze přibližné místa, kam je potřeba vložit nový kód.

`configure`

- řádek 812 - definice proměnné `enable_<název portu>`
- řádek 867 - definice proměnné `enable_<název portu>`
- řádek 1510 - rozšíření nápovědy o parametr pro zamezení kompilace nového portu

```
--disable-<název portu> do not compile simulator for <mikroprocesor>
```

- řádek 2889 - zpracování parametru pro povolení nebo zamezení kompilace portu

```
# Check whether --enable-<název portu> was given.
if test "${enable_<název portu>+set}" = set; then :
```

```
  enableval=$enable_<název portu>;
  if test $enable_<název portu> != "no"; then
    enable_<název portu>="yes"
  fi
else
  enable_xa="no"
fi
```

- řádek 16227 - přidání cesty k souboru `Makefile` nového portu - `<složka portu>/Makefile`
- řádek 17285 - přidání cesty k souboru `Makefile` nového portu - `"<složka portu>/Makefile")`
  `CONFIG_FILES="$CONFIG_FILES <složka portu>/Makefile" ;;`

`configure.in`

- řádek 88 - definice nového parametru pro možnost volby, zda má být daný port kompilován.

```
AC_ARG_ENABLE(<název portu>,
[  --disable-<název portu>          \\
do not compile simulator for <název mikroprocesoru>],
if test $enable_<název portu> != "no"; then
    enable_<název portu>="yes"
fi,
enable_<název portu>="yes")
```

- řádek 96 - příkaz `AC_SUBST(enable_<název portu>)`
- řádek 660 - přidání cesty k souboru Makefile pro kompilaci portu

`packages_in.mk`

- řádek 8 - definice proměnné s informací, zda má být port kompilován
  `enable_<název portu>    = @enable_<název portu>@`
- řádek 38 - přidání nastavenípro nový port

```
ifeq ($(enable_<název portu>),yes)
<název portu>        = <složka portu>
else
<název portu>=
endif
```

- řádek 39 - přidání proměnné pro port do seznamu zdrojových kódů - `$(<název portu>)`
- řádek 41 - přidání složky portu do seznamu zdrojových kódů

Kapitola 4

Návrh a implementace simulátoru PicoBlaze

V této kapitole je popsán návrh a implementace nového portu simulátoru μCsimpro mikroprocesor PicoBlaze. Nový port je založen na standardní struktuře a funkčnosti poskytované simulátorem μCsim, která je popsána v předchozí kapitole, a v této kapitole jsou popsány dílčí části a rozšíření vztahující se k mikroprocesoru PicoBlaze.

4.1 Typy mikroprocesoru

Jedním z výchozích bodů návrhu simulátoru bylo rozhodnutí, pro jaký typ nebo typy mikroprocesoru PicoBlaze bude simulátor určen a jakým způsobem budou tyto typy implementovány. I když je PBCC implementováno pouze pro PicoBlaze 3, s ohledem na budoucí použitelnost simulátoru jsem se rozhodl jej implementovat i pro PicoBlaze 6. Simulátor μCsimnavíc může být použit i samostatně nezávisle na PBCC nebo obecněji SDCC.

Vzhledem k výrazné podobnosti architektury a instrukční sady mikroprocesorů PicoBlaze 3 a 6 by bylo zbytečné vytvářet v μCsimsamostatné třídy mikroprocesorů reprezentující PicoBlaze 3 a 6, ale byla zvolena varianta, kdy je pro oba typy implementován pouze jedna třída mikroprocesoru a rozdílnosti v chování mikroprocesoru se řeší v rámci implementace této třídy. Na základě této volby byl ovlivněn další návrh jednotlivých částí mikroprocesoru a specifika jednotlivých typů jsou popsány v následujících částech.

Výběr typu mikroprocesoru pro simulaci probíhá při spuštění simulátoru pomocí přepínače `-t` a názvu konkrétního typu. Názvy typů byly určeny `kcpsm3` pro PicoBlaze 3 a `kcpsm6` pro PicoBlaze 6. Pokud typ při spuštění není určen, je bude brán jako výchozí typ PicoBlaze 3.

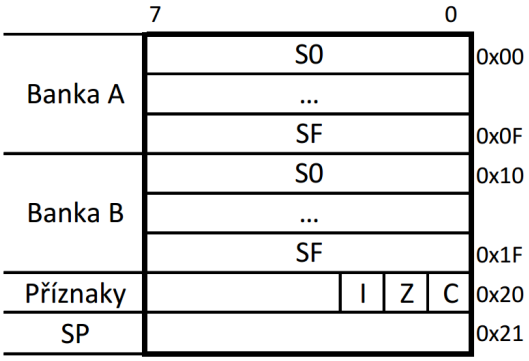
4.2 Registry

Na základě architektury mikroprocesoru PicoBlaze se nabízí možnost implementovat registry pomocí struktury s položkami reprezentujícími jednotlivé registry (viz kapitola 3.2.6). Nedostatkem tohoto řešení je však možnost práce s hodnotami registrů při simulaci. Lze pouze vypsat obsahy registrů přikazem `info registers`. Pro nastavování hodnot registrů by musel být implementován nový příkaz (nebo příkazy). Při implementaci registrů v SFR paměti je automaticky k dispozici stejná funkčnost a příkazy jako u ostatních druhů pamětí (nastavování hodnod i bitů, výpis paměti).

Díky výše uvedeným výhodám byla pro implementaci registrů mikroprocesoru PicoBlaze vybrána druhá varianta, kdy jsou registry mapovány do SFR paměti. Tato varianta sice nereprezentuje architekturu mikroprocesoru PicoBlaze, ale po funkční stránce je ekvivalentní s první variantou. V SFR paměti může být také snadněji implementován pomocí vhodné adresace přístup k registrům v případě více registrových bank.

Návrh SFR paměti s adresami mapovaných registrů je ilustrován na obrázku 4.1. Struktura SFR pam2ti je navržena s ohledem na mikroprocesor PicoBlaze 6 a u mikroprocesoru PicoBlaze 3 bude využita pouze její část. Adresy v SFR paměti reprezentují následujíí registry:

- `0x00` až `0x0F` - v případě PicoBlaze 3 jde o registry `S0` až `SF`. U mikroprocesoru představují registry `S0` až `SF` banky A
- `0x10` až `0x1F` - U PicoBlaze 3 nejsou využity, u PicoBlaze 6 představují registry `S0` až `SF` banky B
- `0x20` - registr s příznaky přenosu (C), nulový příznak (Z) a příznak povolení přerušení (I). Tyto příznaky jsou uloženy na bitech 0, 1 a 2.
- `0x21` - ukazatel vrcholu zásobníku volání podprogramů (SP).



Obrázek 4.1: Struktura SFR paměti

Navržená struktura SFR paměti obsahuje oproti skutečné architektuře mikroprocesoru PicoBlaze navíc registr ukazatele vrcholu zásobníku (stack pointer, SP). Registr SP slouží pro interní implementaci záobníku volání podprogramů, ale tím, že je umístěn v SFR paměti, může být uživatelem při simulaci manipulováno s jeho hodnotou bez nutnosti implementace dodatečných příkazů simulátoru.

Pro přístup k registrům při simulaci pomocí jejich jmen musela být také definována tabulka jmen registrů a tabulka jmen bitů pro přístup k bitům registru příznaků. Registry v obou bankách mikroprocesoru PicoBlaze 6 nazývají stejně, ale v simulátoru musely být rozlišeny. S ohledem na kompatibilitu s PicoBlaze 3 byly registry v bance A pojmenovány dle svého skutečného názvu `S0` až `SF` a registry v bance B byly pojmenovány `S0b` až `SFb`. Registr příznaků a ukazatel vrcholu zásobníku jsou přístupny pomocí jmen `FLAGS` a `SP`. Bity registru příznaků (tzn. jednotlivé příznaky) jsou přístupny pomocí názvů `C`, `Z` a `I`.

4.3 Paměť

V mikroprocesoru PicoBlaze se nachází tři typy pamětí:

- programová paměť o šířce 18 bitů
- RAM paměť o šířce 8 bitů
- zásobník volání podprogramů o šířce 10 bitů (PicoBlaze 3) nebo 12 bitů (PicoBlaze 6)

Pro každý typ paměti musí být při inicializaci simulovaného mikroprocesoru vytvořen adresový prostor, paměťový čip a dekodér adres. Z důvodu podpory mikroprocesorů PicoBlaze 3 i 6 je šířka zásobníku nastavena na 12 bitů. Pro vytvoření zásobníku byl definován nový typ paměti `MEM_STACK`.

Menší komplikaci způsobuje programová paměť se svou šířkou 18 bitů. V μCsim je v souboru `stypes.h` definován typ `t_mem`, který reprezentuje hodnotu v paměti. Typ je však definován jako 16 bitový a nedokáže pojmout hodnotu programové paměti mikroprocesoru PicoBlaze. Z tohoto důvodu musela být změněna definice typu `t_mem` na 32 bitový a v konstruktoru třídy `cl_memory_cell` v souboru `sim.src\\mem.c`. upravena hodnota proměnné `width` na 32;

Mikroprocesor PicoBlaze 3 má pevně danou velikost programové a RAM paměti. U mikroprocesoru PicoBlaze 6 jsou velikosti těchto pamětí ve výchozím stavu stejné jako u PicoBlaze 3, ale mohou být změněny. Aby simulátor mikroprocesoru PicoBlaze 6 podporoval i tento aspekt, byly navrženy nové přepínače, pomocí nichž je možné při spuštění simulátoru u mikroprocesoru PicoBlaze 6 nastavit velikost programové a RAM paměti. Tyto přepínače jsou:

- `-m` - velikost RAM paměti (64, 128 nebo 256 bajtů)
- `-M` - velikost programové paměti (1024, 2048 nebo 4096 instrukcí)

Obdobným způsobem byly implementovány i zbývající modifikovatelné vlastnosti mikroprocesoru PicoBlaze 6, a to hodnota vektoru přerušení a vestavěná hardwarová konstanta. Přepínače pro tyto vlastnosti jsou:

- `-I` - hodnota vektoru přerušení. Výchozí hodnota je 0x3ff.
- `-w` - nastavení hardwarové konstanty pro použití instrukcí `HWBUILD`

4.4 Instrukční sada

Kvůli odlišnosti instrukčních kódů mezi mikroprocesory PicoBlaze 3 a 6 nemohla být definována společná tabulka instrukcí, na jejímž základě jsou jednotlivé instrukce v paměti ROM dekódovány a prováděny. Pro definici tabulky instrukcí však nebyl použit typ `dis_entry` popsaný v kapitole 3.2.7, ale tento typ byl rozšířen o další dvě nové položky - typ instrukce a typ operandů. Pro určení typu instrukce a operandů byly v souboru `glob.h` definovány výčty, které obsahují seznam všech instrukcí a kombinací parametrů. Toto rozšíření má ryze praktický účel pro zjednodušení dekódování a provádění instrukcí, kdy pro každou variantu instrukce (např. operandem je registr nebo konstanta, případně různé druhy skoků) nemusí být implementována samostatná metoda.

Narozdíl od instrukčních kódů je chování instrukcí u obou typů mikroprocesoru až na odlišnosti popsané v kapitole 2.3 stejné. Z tohoto důvodu byly metody reprezentující instrukce implementovány společně pro obě verze mikroprocesoru a pouze v některých případech (např. instrukce `ADDCY` nebo `SUBCY`), je provedení metody rozděleno do dvou větví podle typu mikroprocesoru.

4.5 Přerušení

Pro podporu přerušení musel být simulátor rozšířen ve dvou rovinách. Nejdříve byla vytvořena hardwarová komponenta `irq[0]` simulující vstupní signál žádosti o přerušení. Při simulaci lze žádost o přerušení nastavit příkazem `set hardware irq[0] <hodnota>`, kde `<hodnota>` může být buď 0 pro deaktivaci signálu, nebo 1 (obecně nenulové číslo) pro aktivaci signálu přerušení. Pro získání aktuálního stavu žádosti o přerušení slouží příkaz `info hardware irq[0]`.

Vytvoření hardwarové komponenty pouze umožnilo nastavení žádosti o přerušení. Dále muselo být upraveno zpracování jednotlivých instrukcí. V simulovaném mikroprocesoru poté muselo být přepracováno provádění instrukcí (metoda `do_inst()`) tak, aby pro provedené instrukci probíhala kontrola, zda není aktivní žádost o přerušení. Pokud ano (a přerušení je povoleno), je provedeno zpracování přerušení (metoda `do_interrupt()`).

Výše uvedené změny umožňují simulaci přerušení pouze při aktivní simulaci uživatelem, kdy uživatel v požadované chvíli pozastaví simulaci a pomocí příkazu nastaví žádost o přerušení. Do simulátoru však byla implementována i podpora pro načtení seznamu žádostí přerušení z externího souboru při spuštění simulátoru (přepínač `-i`) a nebo pomocí příkazu `import interrupts "soubor"` za běhu simulátoru.

Při načítání seznamu žádostí o přerušení je na vstupu očekáván XML soubor, který v elementech interrupt obsahuje číslo instrukčního cyklu, ve kterém má dojít k aktivaci žádosti o přerušení. Příklad XML souboru je uveden níže. Pokud je do simulátoru načten seznam žádostí o přerušení, simulátor pak při běhu automaticky kontroluje, zda pro aktuální instrukční cyklus existuje v seznamu přerušení položka a pokud ano, nastaví žádost o přerušení v hardwarové komponentě `irq[0]` jako aktivní. Při simulaci lze kombinovat nastavování žádostí o přerušení pomocí XML souboru i ručního nastavení pomocí příkazu.

```
<interrupts>
    <!-- žádost o přerušení bude nastavena při provádění 10. instrukce -->
    <interrupt>10</interrupt>
    <!-- žádost o přerušení bude nastavena při provádění 35. instrukce -->
    <interrupt>35</interrupt>
</interrupts>
```

4.6 Vstupy

Pro simulaci vstupních operací (instrukce INPUT) byly v simulovaném mikroprocesoru vytvořeny dvě komponenty - `input_port[0]` a `port_id[0]`. Pomocí komponenty `input_port` je simulován 8bitový vstupní port a komponenta `port_id` simuluje výstup udávající 8bitovou adresu portu při vstupních a výstupních operacích.

Před vstupní operací je možné nastavit vstupní hodnotu pomocí příkazu `set hardware input_port[0] <hodnota>`. Tato hodnota se nastavuje obecně pro následující vstupní operaci a je nezávislá na adrese portu určeném instrukcí `INPUT`. Po zpracování vstupu už

nebude zadaná hodnota použita při příští vstupní operaci. Pro příští vstupní operaci musí být nastavena nová hodnota. V případě, že není nastavena žádná hodnota, simulátor vypíše informaci, že není specifikován vstup a jako standardní hodnota je použita 0.

Podobně jako pro přerušení byla navíc implementována funkčnost pro import vstupních dat z XML souboru kvůli usnadnění automatizovaného spouštění simulátoru. Data pro vstupní operace lze načíst při spuštění simulátoru pomocí přepínače `-n` anebo za běhu simulátoru příkazem `import input "soubor"`.

Pro specifikaci vstupů v XML souboru je určen element input. Element má povinný atribut type určující typ vstupních informací. Atribut může nabývat těchto hodnot:

- hex - vstupem je hodnota v hexadecimálním formátu. Hodnota musí být reprezentována dvěma znaky, např 01, FE apod.
- int - vstupem je číslo v desítkové soustavě
- char - vstupem je znak

Pro usnadnění zadávání vstupních dat může být u typů hex a char v rámci definice vstupu zadáno (zřetězeno) více hodnot. Zápis je pak ekvivalentní zápisu, ve kterém by byla každá hodnota v samostatném elementu input. Jednotlivé vstupní hodnoty budou zpracovávány v takovém pořadí, v jakém byly zadány v XML.

Kromě atributu type může element input obsahovat ještě nepovinné atributy port a tick. Atribut port omezuje vstupní data pouze pro určitý port a pomocí atributu tick lze přiřadit vstupní hodnotu ke konkrétnímu instrukčnímu cyklu. Pokud je však zadán atribut tick, u hodnot typu hex a char nelze použít zřetězené zadání, resp. je použita pouze první hodnota. Může být zadán pouze jeden z atributů port nebo tick, nebo oba atributy.

V průběhu simulace jsou pak při vstupní operaci INPUT prohledávány načtené vstupy. Nejvyšší prioritu mají vstupy, u nichž je určen port. Pokud je u vstupu zadán instrukční cyklus, má vyšší prioritu než vstup bez zadání instrukčního cyklu. Prohledávání vstupů probíhá v následujícím pořadí:

1. podle portu a aktuálního instrukčního cyklu
2. podle portu
3. podle aktuálního instrukčního cyklu
4. vstupy bez určeného portu a instrukčního cyklu

Pokud nebyl nalezen ani jeden vyhovující vstup, jako vstup je použita hodnota 0 a uživatel je informován, že pro vstupní operaci nebyl specifikován žádný vstup.

Pro nastavení vstupních dat lze kombinovat oba způsoby zadání - XML souborem a příkazem. Nastavení aktuální vstupní hodnoty pomocí příkazu má však vždy vyšší prioritu a nastavená hodnota je použita i v případě, že v XML souboru je pro vstupní port nebo instrukční cyklus specifikována hodnota.

```
<inputs>
    <input type="int">128</input>
    <input type="int" port="8">128</input>
    <input type="int" tick="25">128</input>
    <input type="int" port="6" tick="150">128</input>
```

```
    <input type="hex">0A</input>
    <input type="hex">0A0B0C0D</input>

    <input type="char">a</input>
    <input type="char">abCD 01</input>
</inputs>
```

4.7 Výstupy

Pro simulaci výstupů při instrukci `OUTPUT` byla vytvořena v hardwarová komponenta `output_port[0]` a již zmíněná komponenta `port_id[0]`. Při provádění instrukce `OUTPUT` je výstupní hodnota nastavena v komponentě `output_port[0]` a adresa výstupního portu v komponentě `port_id[0]`. Tyto hodnoty lze přečíst pomocí příkazů `info hardware output_port[0]` a `info hardware port_id[0]`.

Jednotlivé výstupy jsou také průběžně zaznamenávány a kompletní přehled výstupů v XML formátu lze získat pomocí příkazu get output "soubor". Pokud není zadán název souboru, XML je vypsáno na standardní výstup. XML obsahuje element port pro každý port, na kterém došlo k výstupu. Atribut id udává adresu portu. Element port obsahuje pro každý výstup, který byl na portu proveden, element output udávající hodnotu výstupu. Atribut tick udává, ve kterém instrukčním cykly k výstupu došlo. Adresa portu je číslo v desítkové soustavě a hodnota výstupu je zapsaná hexadecimálně.

```
<outputs>
    <port id="97">
        <output tick="28">bb</output>
    </port>
    <port id="111">
        <output tick="3">ab</output>
        <output tick="8">01</output>
    </port>
</outputs>
```

4.8 Stav mikroprocesoru

Pro možnost automatizovaného spouštění simulátoru nebo napojení na grafické rozhraní byla v simulátoru implementována možnost pro import a export vnitřního stavu mikroprocesoru zahrnujícího obsah RAM paměti, obsah a ukazatel vrcholu zásobníku, hodnoty registrů, programového čítače a příznaků. Stav procesoru neobsahuje obsah programové paměti, protože ten je určen simulovaným programem v HEX formátu. Pro export stavu mikroprocesoru slouží příkaz `pbstate "soubor"`. Pokud není zadán název souboru, je XML se stavem procesoru vypsáno na standardní výstup. Pro import stavu mikroprocesoru slouží příkaz `import pbstate "soubor"`.

Import i export pracuje se stejnou strukturou XML souboru. Kořenový element `picoblaze` s atributem `type` udává typ mikroprocesoru. Elementy `ram` a `stack` obsahují výpis paměti RAM a zásobníku. Jejich atribut `size` udává velikost paměti resp. zásobníku a atribut `bytes` říká, kolik bajtů reprezentuje jednu hodnotu. V obsahu elementu je jedna hodnota reprezentována počtem znaků, který se rovná dvojnásobku hodnoty atributu `bytes` (2 znaky

představují 1 bajt) a hodnoty jsou uloženy postupně od adresy 0 po nejvyšší adresu. Element `registers` sdružuje hodnoty registrů (`regbank`), programového čítače (`pc`), ukazatele zásobníku (`sp`), a příznaků (`flags` a podřízené elementy `carry`, `zero` a `interruptEnable`). Pokud je exportován stav mikroprocesoru PicoBlaze 6, element `registers` obsahuje atribut `activeRegbank` udávají, která z registrových bank je aktivní. Hodnoty registrů jsou sdruženy v elementu `regbank` (u PicoBlaze 6 atribut `name` udává, zda se jedná o banku A nebo B) a jsou zapsány podobným způsobem jako paměť RAM nebo zásobník. Element `regbank` obsahuje postupně hodnoty registrů `s0` až `sF` a každý registr je reprezentován jedním bajtem (dvěma znaky).

```
<picoblaze type="KCPSM6">
    <!-- RAM[0] = 0xbb, RAM[1] = 0x83, ... , RAM[63] = 0x00 -->
    <ram size="64" bytes="1">bb838f1de43a39cc...cc7a00</ram>
    <!-- zásobník[0] = 0xaa2e, zásobník[1] =0xa3e6, ...
    <stack size="30" bytes="2">aa2ea3e6....9d7f7600</stack>
    <registers activeRegbank="A">
        <regbank name="A">eec254f81be8e78d765a2e63339fc99a</regbank>
        <regbank name="B">ffc6697351ff4aec29cdbaabf2fbe346</regbank>
        <pc>0</pc>
        <sp>0</sp>
        <flags>
            <carry>0</carry>
            <zero>0</zero>
            <interruptEnable>0</interruptEnable>
        </flags>
    </registers>
</picoblaze>
```

4.9 Použité knihovny

Výše navržená rozšíření funkčnosti simulátoru μCsim pracují s XML soubory, avšak existující porty ani jádro simulátoru s XML soubory nepracují, a tudíž neobsahují knihovny nebo metody čtení a tvorbu XML souborů. Pro tyto účely byla v portu pro mikroprocesor PicoBlaze použita knihovna TinyXML [10].

Pro interní reprezentaci vstupů a žádostí o přerušení, které byly načteny z XML souborů, a provedených výstupů mikroprocesoru byly použity kontejnery (seznam - list, asociativní pole - map, množina - set) z knihovny Standard Template Library (STL) jazyka C++. Z použití této knihovny však bohužel vyplynuly problémy při kompilaci simulátoru způsobené definicemi v jádru simulátoru μCsim.

Problém způsobuje definice `#define bool int` na řádku 36 v souboru `pobjt.h`, kterou je definován typ `bool` jako celočíselný typ. Při použití knihovny STL však tato definice typu způsobuje kolizi v definicích některých funkcí knihovny STL. Problém částečně vyřešilo zakomentování řádku v souboru `pobjt.h`, ale v důsledku toho vyplynuly na povrch nekonzistentnosti v kódu jádra μCsim. V hlavičkových souborech byly parametry nebo návratové hodnoty některých metod deklarovány jako typ `bool` a v definici metod pak jako typ `int` (případně opačně). Pokud nebyla definice v souboru `pobjt.h` zakomentována, typy `bool` a `int` byly ekvivalentní a nekonzistentnosti v deklaracích a definicích metod ničemu

nevadily. Kvůli zakomentování této definice tak musela být upravena řada deklarací nebo definicí metod jádra μCsim.

Z hlediska funkčnosti tyto změny nemají žádný vliv. Problém by však nastal v situaci, kdy by bylo potřeba port pro mikroprocesor PicoBlaze přenést do novější verze simulátoru μCsim, nebo v případě snahy integrovat port pro PicoBlaze do oficiální distribuce simulátoru μCsim. Řešení tohoto problému jsou v zásadě dvě:

1. Reimplementovat port pro PicoBlaze tak, aby nevyužíval knihovnu STL, případně parametrizovat kompilaci portu tak, aby bylo možné deaktivovat rozšíření využívající knihovnu STL.

2. Snažit se prosadit oficiální změnu zdrojových souborů jádra simulátoru μCsim, která by vyřešila problém s problémovou definicí typu a deklaracemi a definicemi metod.

Kapitola 5

Návrh grafického simulátoru v Eclipse

Pro odlišení grafického simulátoru v Eclipse a portu μCsim pro PicoBlaze bude dále v textu nazýván simulátor v Eclipse pouze jako grafický simulátor a port μCsim pro PicoBlaze jako simulátor spblaze.

5.1 Prostředí Eclipse

Pod pojmem Eclipse [12, 19] si mnozí vývojáři představí integrované vývojové prostředí (integrated development environment, IDE) pro vývoj aplikací v jazyce Java, ale Eclipse není pouze vývojové prostředí pro aplikace v jazyce Java. Eclipse ve významu Eclipse Software Development Kit (Eclipse SDK) představuje kombinaci několika projektů, z nichž nejpodstatnější je platforma Eclipse, Java Development Tools (JDT) [21] a Plug-in Development Environment (PDE) [22].

Platforma Eclipse představuje univerzální a základní část systému, která poskytuje prostředky a služby pro vývoj jak jednoduchých aplikací, tak komplexních nástrojů. Platforma Eclipse byla již od základu navržena s ohledem na svou rozšiřitelnost pomocí zásuvných modulů (plug-ins). Všechny nástroje vyvinuté nad platformou Eclipse tak představují její zásuvné moduly. Zásuvnými moduly je možné rozšířit nejen samotnou platformu, ale také jiné zásuvné moduly. Komplexní vývojové prostředí tak může představovat velké množství kooperujících zásuvných modulů.

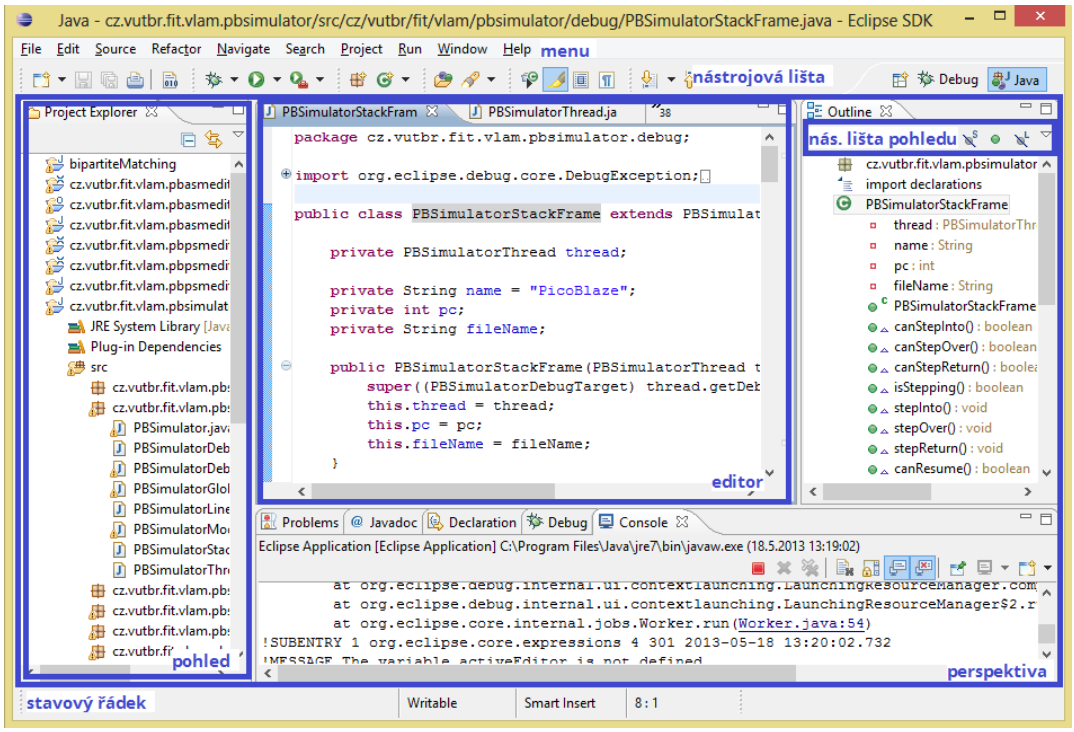
JDT představuje sadu zásuvných modulů rozšiřující základní platformu o nástroje pro podporu vývoje, testování a ladění aplikací v jazyce Java. PDE pak obsahuje nástroje pro vývoj zásuvných modulů pro platformu Eclipse. Pomocí jiných zásuvných modulů může být platforma Eclipse rozšířena i o podporu vývoje v dalších programovacích jazycích, jako například C/C++ nebo PHP.

Samotné zásuvné moduly musí být ale vytvořeny v jazyce Java. Zásuvný modul se skládá z kódu v souboru JAR (Java archive), případných dalších souborů jako jsou obrázky, ikony apod. a souborů `MANIFEST.MF` a `plugin.xml`. Soubor `MANIFEST.MF` obsahuje obecné informace o zásuvném modulu (název, identifikátor, verze apod.) a dále informace týkající se spuštění modulu (např. závislosti na ostatních zásuvných modulech). Soubor `plugin.xml` obsahuje informace o závislostech zásuvného modulu na jiných modulech nebo knihovnách, seznam rozšíření, pomocí kterých zásuvný modul rozšiřuje funkcionalitu platformy Eclipse a jiných zásuvných modulů, a seznam rozšiřujících bodů, na základě nichž může být naopak

rozšířena funkčnost vytvářeného modulu jinými zásuvnými moduly.

5.1.1 Uživatelské prostředí Eclipse

Uživatelské prostředí platformy Eclipse [16] se skládá z různých částí, které jsou zobrazeny na obrázku 5.1.



Obrázek 5.1: Uživatelské rozhraní Eclipse

Pracovní prostředí (workbench) se skládá z jednoho nebo více oken (window). Každé okno obsahuje menu (menu bar), nástrojovou lištu (tool bar) a jednu nebo více perpektiv (perspective). Perspektiva definuje seznam komponent (part), které budou zobrazeny v okně, a jejich rozložení a uspořádání v okně. V danou chvíli je viditelná vždy jen jedna perspektiva.

Komponenty zobrazené v perspektivě mohou být dvou typů - editor a pohled (view). Editor zpravidla slouží k úpravám zdrojových souborů (nebo obecněji zdrojů) a může se jednat například o textový nebo grafický editor. Změny provedené v editoru nejsou do zdrojového souboru zapisovány ihned, ale až na základě akce uživatele, případně v rámci periodicky spouštěného automatického ukládání. Pohled zpravidla nepracuje se zdrojovými soubory, ale slouží pro zobrazování informací nebo navigaci. Změny provedené v pohledu jsou narozdíl od editoru uloženy ihned. Pohledy a editory mohou mít také svou nástrojovou lištu.

V rámci tvorby zásuvného modulu je možné rozšířit kteroukoliv z výše uvedených částí uživatelského prostředí Eclipse.

5.1.2 VLAM IDE

VLAM IDE [14] je balík zásuvných modulů pro Eclipse, které vznikl na Fakultě informačních technologií Vysokého učení technického v Brně v rámci projektu VLAM, a který rozšiřuje schopnosti Eclipse o návrh malých vestavěných systémů s možností souběžného návrhu hardware a software.

Vývojové prostředí VLAM IDE nabízí tyto části a funkcionalitu:

- grafický komponentní editor pro platformu FITkit obsahující různé komponenty (např. mikroprocesor PicoBlaze, LCD displej apod.) se schopností generování VHDL kódu
- editor jazyka symbolických instrukcí pro PicoBlaze 3. Editor podporuje oba dialekty - KCPSM i pBlazIDE.
- integraci s nátrojem QDevKit pro programování platformy FITkit
- integraci překladače PBCC pro překlad programu z jazyka C

VLAM IDE bylo vyvíjeno pro Eclipse ve verzi 3.6.2 (Helios). V době tvorby této práce bylo k dispozici Eclipse ve verzi 4.2.2 (Juno). VLAM IDE však prozatím nebylo uvolněno i pro novější verze Eclipse z důvodu kompatibility a závislosti na jiných rozšířeních.

5.2 Funkčnost a uživatelské rozhraní grafického simulátoru v prostředí Eclipse

Na základě schopností simulátoru spblaze a vlastností vývojového prostředí Eclipse byla navržena následující sada funkcí, kterou by měl grafický simulátor v Eclipse disponovat:

- zobrazením stavu mikroprocesoru - pohled, ve kterém by byly přehledně zobrazeny aktuální hodnoty registrů, příznaků, RAM paměti, zásobníku volání podprogramů a případně vstupních a výstupních portů. Uživatel by měl mít možnost v případě pozastavené simulace změnit jednotlivé hodnoty registrů, paměťových buněk atd. Při krokování by mohly být jednotlivé registry, buňky atd., u nichž došlo ke změně hodnoty, pro upozornění uživatele zvýrazněny.
- nastavením parametrů simulátoru - možnost definovat parametry simulovaného mikroprocesoru (typ mikroprocesoru PicoBlaze, velikost paměti RAM, ROM, atd.; odpovídá parametrům simulátoru spblaze). Na základě zvolených parametrů by měl být adekvátně vykreslen i pohled se stavem mikroprocesoru.
- ovládáním simulace - spuštění a ukončení simulace, pozastavení a spuštění běhu simulovaného programu a možnost krokování simulace (vykonání pouze jedné instrukce), resetování mikroprocesoru a možnost aktivace žádosti o přerušení.
- body přerušení - vkládání a mazání bodů přerušení v simulovaném programu (v editoru zdrojového souboru). Body přerušení by měly být zobrazeny i ve standardním pohledu Breakpoints.
- zobrazením prováděné instrukce - v případě pozastavení simulace (na základě bodu přerušení, nebo explicitní pozastavení uživatelem) musí být zvýrazněna instrukce odpovídající aktuální hodnotě programového čítače.

Pro implementaci výše uvedených požadavků lze z části využít standardní funkčnosti poskytované platformou Eclipse, některé části však budou muset být nově vytvořeny. Např. zobrazení stavu mikroprocesoru je specifická záležitost a pro tyto účely nelze využít žádný existující pohled nebo editor. V první fázi bylo nutné také rozhodnout, jakým způsobem bude simulátor integrován do vývojového prostředí Eclipse. Jedná se zejména o způsob spouštění a ovládání simulátoru. Veškerou funkčnost by bylo možné implementovat samostatně a ovládací prvky související se simulací realizovat jako součást pohledu se stavem mikroprocesoru. Výhodou by byla kompaktnost uživatelského rozhraní, protože zobrazení stavu mikroprocesoru a ovládání simulace by byly pohromadě na jednom místě.

Na druhou stranu simulace se po funkční stránce podobá ladění programu a Eclipse již disponuje infrastrukturou, na jejímž základě může být vytvořen vlastní debugger/simulátor. Jde například o konfigurace spouštění, kde je možné nastavit parametry laděného programu, nebo o pohled Debug, který poskytuje základní ovládací prvky pro ladění. Výhodou je právě využití standardní funkcionality Eclipse, díky němuž bude pro uživatele znalého Eclipse snadnější pracovat se simulátorem. Vzhledem k mnohem vyšší integrovatelnosti a využití existující infrastruktury byla vybrána tato možnost.

Podpora bodů přerušení a zvýraznění aktuálně prováděné instrukce se z hlediska uživatelského rozhraní týkají editoru zdrojového kódu. V rámci projektu VLAM IDE byly vytvořeny dva editory pro zdrojové soubory programů pro PicoBlaze v jazyce symbolických instrukcí - editor PBPsm pro dialekt KCPSM a PBAsm pro dialekt pBlazeIDE. Oba editory jsou téměř shodné, jediné rozdíly plynou z rozdílů dialektů jazyka. Pro oba editory však bude muset být implementována podpora možnosti vkládat a odebírat body přerušení a zobrazovat aktuálně prováděnou instrukci.

Ve výchozím zobrazení vývojového prostředí je hlavní částí editor zdrojového kódu, případně jiný editor. Během simulace je však potřeba odlišné rozložení vývojového prostředí se dvěma hlavními částmi (pohledy) - editor zdrojového kódu a pohled se stavem mikroprocesoru. Pro tyto účely bude vhodné vytvořit nové rozložení vývojového prostředí - perspektivu, která bude automaticky otevřena při spuštění simulace. Kromě zmíněného pohledu a editoru by součástí editoru měl být pohled Debug pro ovládání simulace, pohled Breakpoints se zobrazením všech bodů přerušení a pohled *Console*, kam bude uživateli zapisován průběh komunikace mezi grafickým simulátorem a simulátorem spblaze.

Z důvodu návaznosti na editory PBPsm a PBAsm z rozšíření VLAM IDE bude nutné grafický simulátor vyvíjet také pro Eclipse v poněkud starší verzi 3.6. V případě, že by měl být grafický simulátor vyvinut i pro novější verzi Eclipse, bylo by nutné opustit myšlenku využití editorů PBPsm a PBAsm a simulátor implementovat nezávisle na VLAM IDE. V tomto případě by musel být pro možnost vkládání bodů přerušení a zobrazování aktuálně prováděné instrukce použit jiný editor zdrojového kódu. V případě použití obecného textového editoru by postrádal zvýrazňování syntaxe a chyb ve zdrojovém kódu, případně by tato funkčnost musela být implementována. Opětovná implementace editoru je vzhledem k existujícím editorům PBPsm a PBAsm zbytečná, proto byla i přes nutnost použití starší verze Eclipse zvolena implementace formou rozšíření VLAM IDE.

5.3 Proces simulace

V předchozích kapitolách byl popsán návrh a implementace simulátoru spblaze. Již od počátku návrhu bylo uvažováno, že simulátor spblaze bude možné použít jako samostatný simulátor, ale také že bude použit pro implementaci grafického simulátoru ve vývojovém

prostředí Eclipse. S ohledem na tyto skutečnosti byly navrženy a implementovány některá rozšíření simulátoru spblaze oproti standardní funkčnosti poskytované μCsim.

V rámci implementace grafického simulátoru v Eclipse by nebylo nutné použít simulátor spblaze. Simulátor by mohl být implementován například jako interpret assembleru přímo v rámci grafického simulátoru. Výhodou by byla volnost při návrhu a možnost mnohem větší provázanosti grafického prostředí a samotného simulátoru, nicméně by se jednalo o implementaci velmi podobné funkcnionality jako při tvorbě simulátoru spblaze. Právě z tohoto důvodu bylo již od začátku pro tvorbu grafického simulátoru v Eclipse uvažováno o použití vytvářeného simulátoru spblaze.

Jedním z problémů použití simulátoru spblaze je skutečnost, že v Eclipse je pracováno se zápisem programu v jazyce symbolických instrukcí, ale spblaze předpokládá na vstupu již přeložený program v HEX formátu. Před spuštěním simulace musí tedy nejprve dojít k překladu programu. Pro překlad je možné využít některý z existujících assemblerů pro mikroprocesor PicoBlaze, případně je možné překladač navrhnout a implementovat samostatně.

V důsledku použití dvou různých reprezentací programu (v jazyce symbolických instrukcí v editoru a v HEX formátu v simulátoru spblaze) je nutné disponovat mechanizmem pro mapování jedné reprezentace na druhou. Pro účely přidávání bodů přerušení je například potřeba znát pro jednotlivé řádky kódu (instrukce) odpovídající adresu v ROM paměti, nebo naopak zobrazení aktuálně prováděné instrukce v editoru kódu je potřeba z adresy v programovém čítači určit odpovídající řádek kódu.

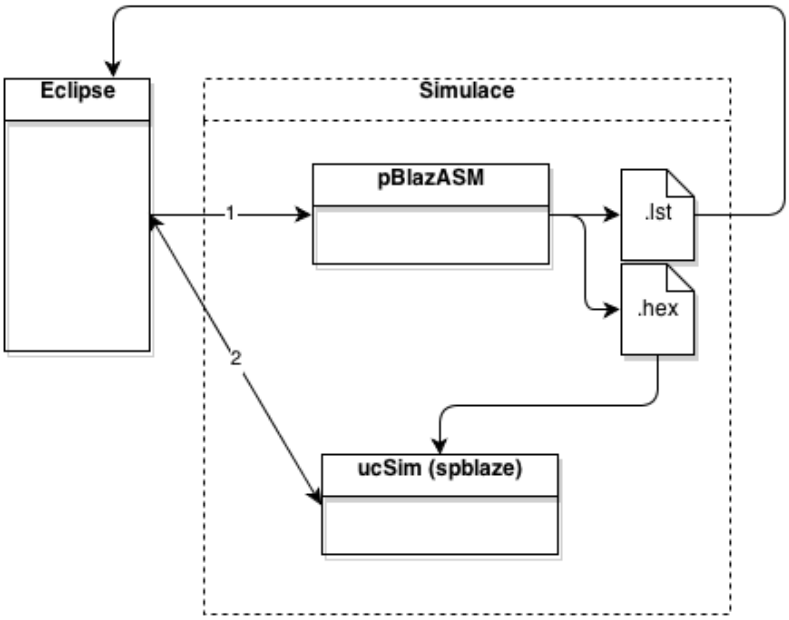
Překladač by měl také ideálně podporovat obě verze mikroprocesoru PicoBlaze a oba dialekty jazyka symbolických instrukcí.

Vzhledem ke složitosti implementace vlastního překladače jsem se z dostupných překladačů popsaných v kapitole 2.5 rozhodl využít assembler pBlazASM, a to hned z několika důvodů:

- podporuje PicoBlaze 3 i 6
- částečně podporuje oba dialekty KCPSM i pBlazeIDE
- překladač je možné zkompilovat pro platformu Windows i Linux
- kromě výstupu v HEX souboru poskytuje i `.lst` soubor mapující řádky zdrojového souboru na adresy v paměti ROM a instrukční kódy (bližší popis je uveden v kapitole 5.3.1)

Z ostatních assemblerů by díky podpoře obou verzí i dialektů mikroprocesoru PicoBlaze připadal v úvahu ještě openPICIDE, ale problémem je to, že jde o integrované vývojové prostředí, a překladač je jeho pevnou součátí a nelze použít samostatně. Dalším problémem, nejen vývojového prostředí openPICIDE ale i ostatních assemblerů, je to, že neposkytují výstup, ve kterém by byly řádky zdrojového souboru v originálním formátování mapovány na instrukční kódy a adresy.

Nevýhodou použití assembleru pBlazASM je skutečnost, že při překladu pro mikroprocesor PicoBlaze 6 nedokáže zpracovat zdrojový soubor v dialektu KCPSM. Tento problém by bylo možné z části obejít úpravou zdrojových souborů assembleru pBlazASM a povolit spuštění pro verzi PicoBlaze 6 a zdrojový soubor v dialektu KCPSM. Tato změna ale nepřidá assembleru novou funkčnost, pouze bude možné přeložit zdrojové soubory v dialektu KCPSM i pro PicoBlaze 6, avšak se schopností zpracovat zdrojový kód pouze v rozsahu PicoBlaze 3.



Obrázek 5.2: Návrh procesu simulace v Eclipse

5.3.1 Struktura LST souboru

Jak bylo zmíněno výše, LST soubor generovaný assemblerem pBlazASM obsahuje mapování instrukcí zdrojového souboru na jejich instrukční kódy a umístění v paměti ROM.

Každý LST soubor obsahuje na prvním řádku identifikaci verze mikroprocesoru PicoBlaze. Možnými hodnotami jsou `PB3` pro PicoBlaze 3 a `PB6` pro PicoBlaze 6. Ve zbývající části LST souboru je pak uvedena reprezentace zdrojového souboru nebo souborů. Assembler pBlazASM umožňuje překlad programu z více zdrojových souborů (například soubor s definicemi konstant a inicializacemi a soubor se samotný programem). Jednotlivé zdrojové soubory jsou v LST souboru uvedeny řádkem

```
---------- source file: <název souboru>
```

Za identifikací souboru pak následuje jeho obsah v nezměněné podobě z hlediska počtu řádků, komentářů, prazdných řádků apod. Na začátek řádků však mohly být přidány doplňující informace. Nejpodstatnější jsou informace uváděné u jednotlivých instrukcí a jejich struktura je následující:

```
AAA IIIII  [návěští :] instrukce
```

kde `AAA` je adresa instrukce v ROM paměti, `IIIII` je kód dané instrukce. Obě hodnoty jsou zapsány hexadecimálně. `[návěští :]` je nepovinná položka reprezentuje definované návěští ve zdrojovém souboru a `instrukce` je zápis dané instrukce ze zdrojového souboru. Jednotlivé prvky jsou vždy zarovnány pod sebou. V případě, že se návěští nachází na samostatném řádku, jsou ze zápisu vynechány prvky `IIIII` a `instrukce`.

Pro lepší představu je uveden následující příklad LST souboru.

```
PB3
---------- source file: test.psm
```

```
000 00141  main            : LOAD   s1, 0x41
001 2C1EC                    OUTPUT s1, 0xEC
002        test            :
002 18102                    ADD    s1, 2

003 34000                    JUMP   main
```

Zdrojový program obsahuje celkem 4 instrukce, které se ve zdrojovém souboru nachází na 1., 2., 4. a 6. řádku a ROM paměti uloženy na adresách 0 až 3. Návěští `main` pak odkazuje na adresu 0 a návěští `test` na adresu 2.

Na základě zpracování LST souboru tam může být jednotuše získáno mapování řádků ve zdrojovém souboru na adresy v ROM paměti a opačně. Uplatnit lze také i znalost návěští a odpovídající adresy např. pro přehlednější zobrazení zásobníku volání podprogramů.

Kapitola 6

Implementace grafického simulátoru

V této kapitole jsou popsány dílčí části implementace grafického simulátoru v prostředí Eclipse na základě požadavků a návrhu uvedeného v předchozí kapitole.

6.1 Perspektiva a pohled se stavem mikroprocesoru

Dle návrhu byla pro účely simulace vytvořena nová perspektiva s názvem PB Simulation jako rozšíření `org.eclipse.ui.perspectives` a třída `PBSimulatorPerspective`. Určení pohledů a editorů, které budou v rámci perpektivy zobrazeny, je definováno pomocí rozšíření `org.eclipse.ui.perspectiveExtensions`.

Do definované perspektivy PB Simulation byly zahrnuty tyto části:

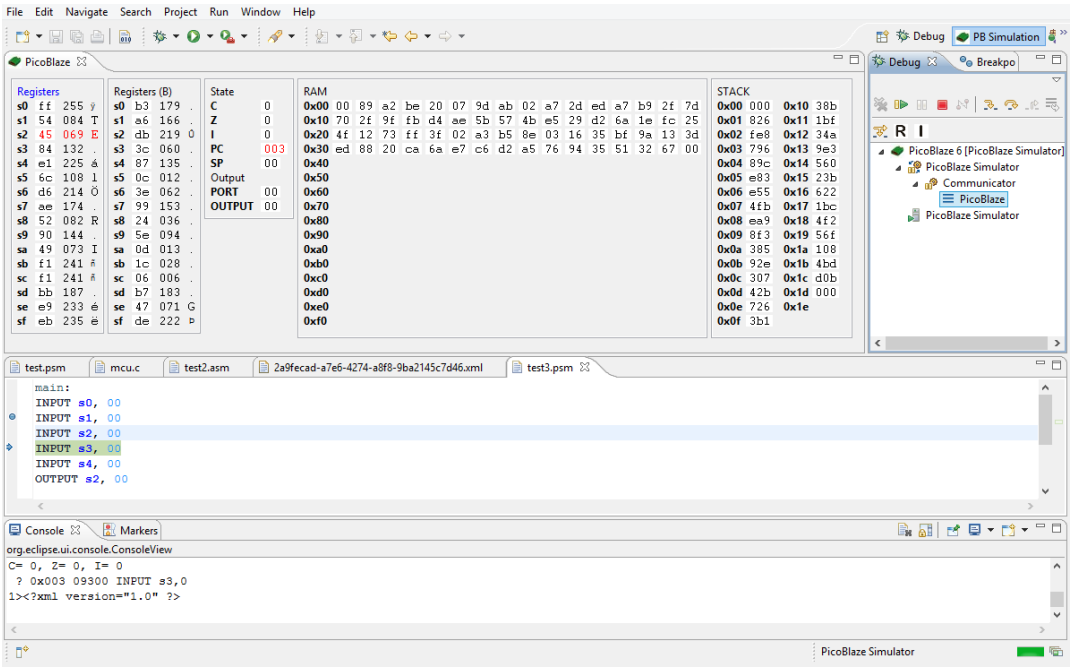
- pohled PicoBlaze zobrazující aktuální stav mikroprocesoru
- část s editory zdrojových kódů
- pohled Debug pro ovládání simulace
- pohled Breakpoints zobrazující seznam všech bodů přerušení
- pohled Console, do kterého je vypisována komunikace se simulátorem spblaze
- pohled Markers, ve kterém jsou zobrazovány kromě řady jiných značek také body přerušení, avšak s podrobnějšími informacemi než v pohledu Breakpoints

Celkový pohled na perspektivu PB Simulation je možné vidět na obrázku 6.1.

6.1.1 Pohled PicoBlaze

Na obrázku 6.1 je možné vidět implementovaný pohled PicoBlaze, který zobrazuje stav mikroprocesoru při simulaci.

V plném zobrazení (pro mikroprocesor PicoBlaze 6) pohled obsahuje části *Registers* a *Registers (B)* zobrazující hodnoty obou registrových bank, *State* se zobrazením hodnot příznaků, programového čítače a ukazatele vrcholu zásobníku, část *RAM* obsahující výpis hodnot v paměti RAM a nakonec část *Stack* se zobrazením zásobníku volání podprogramů.



Obrázek 6.1: Perspektiva PB Simulation

Jednotlivé části jsou vykreslovány v závislosti na zvoleném typu mikroprocesoru a nastavených parametrech. Týká se to části *Registers (B)*, která není u PicoBlaze 3 vykreslena vůbec, a částí *RAM* a *Stack*, kde se mění počet vykreslených hodnot. Hodnoty jednotlivých buněk jsou zobrazovány hexadecimálně a pouze u registrů je kromě hexadecimální hodnoty zobrazována i hodnota v desítkové soustavě a ASCII znak, případně tečka, pokud se jedná o netisknutelný znak.

Při pozastavené simulaci je možné hodnoty jednotlivých buněk měnit (pouze hexadecimální hodnotu) a změnu je nutné potvrdit stisknutím klávesy Enter nebo přepnutím kurzoru do jiné buňky. V případě, že je zadána neplatná hodnota, není změna provedena.

Během simulace jsou hodnoty, u nichž došlo ke změně, označovány červeně. V případě mikroprocesoru PicoBlaze 6 je navíc modrým písmem označena aktivní registrová banka.

6.2 Simulace

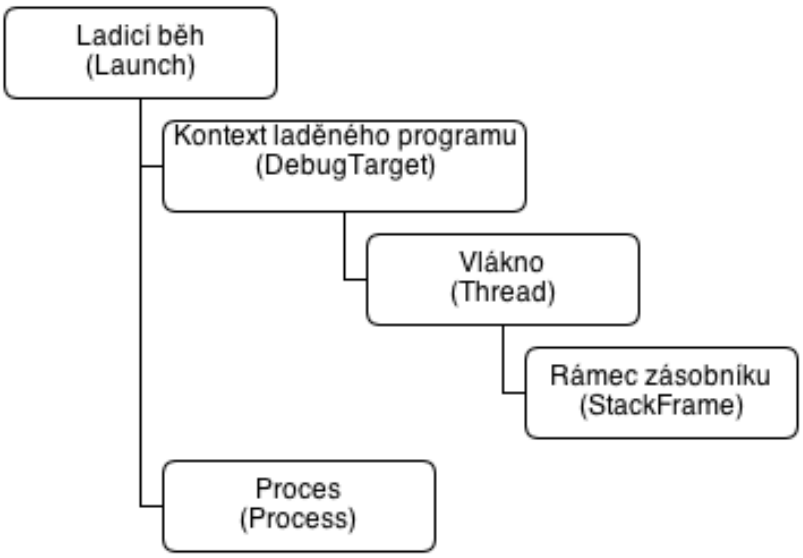
V následujících kapitolách je popsána implementace částí vztahujících se k ovládání simulace.

6.2.1 Zobrazení v pohledu Debug

V pohledu Debug se zobrazují informace o aktuálně laděném programu a pomocí nástrojové lišty je ovládáno samotné ladění/simulace. Pohled je navržen obecně pro zobrazování složitějších vícevláknových aplikací, kde jsou pro každé vlákno zobrazeny rámce zásobníku vztahující se k volaným funkcím. Při simulaci mikroprocesoru toto členění postrádá smysl, nicméně pro plné využití funkčnosti musí být jednotlivé elementy (vlákno, rámec zásobníku) implementovány.

Obecná struktura elementů zobrazených v pohledu Debug je ukázána na obrázku 6.2. Kořenový uzel reprezentuje ladicí běh (launch) spuštěný uživatelem. V rámci běhu je zob-

razen systémový proces, který obstarává běh laděného programu, a kontext laděného programu. Kontext pak obsahuje již zmíněný seznam všech vláken programu a v případě, že běh vlákna je při ladění pozastaven (uživatelsky nebo bodem přerušení), jsou pro vlákno zobrazeny jednotlivé rámce zásobníku odpovídající volaným funkcích. Rámce zásobníku korespondují s konkrétními místy ve zdrojových kódech a na základě nich je v editoru kódu zobrazována aktuálně prováděná instrukce. Na rámec zásobníku pak mohou být navázány proměnné spadající do tohoto rámci. Proměnné však již nejsou zobrazovány v pohledu Debug, ale v samostatném pohledu. Z důvodu provázánosti rámce zásobníku a zdrojového kódu bylo nutné i pro grafický simulátor mikroprocesoru PicoBlaze implementovat všechny elementy zobrazované v pohledu Debug.



Obrázek 6.2: Obecná struktura pohledu Debug

V balíčku `cz.vutbr.fit.vlam.pbsimulator.debug` byly implementovány jednotlivé třídy `PBSimulatorDebugTarget`, `PBSimulatorThread` a `PBSimulatorStackFrame` reprezentující výše uvedené elementy pohledu Debug. Po funkční stránce třída `PBSimulatorStackFrame` obsahuje pouze informaci o zdrojovém souboru a řádku, ke kterému se rámec zásobníku vztahuje, a jinak je veškará funkčnost delegována na třídu `PBSimulatorThread` a z ní dále na `PBSimulatorDebugTarget`. Teprve v této třídě je obsažena veškerá logika pro ovládání simulace a komunikaci se simulátorem.

6.2.2 Body přerušení

Platforma Eclipse poskytuje základní stavební prvky pro implementaci možnosti vkládání bodů přerušení (breakpoints) v editoru zdrojového kódu. Implementace vkládání bodů přerušení i při využití poskytovaných prvků není zcela triviální a pro možnost vkládání bodů přerušení v editoru assembleru PicoBlaze musela být definována řada rozšíření a tříd.

Prvním krokem byla definice bodu přerušení a značky reprezentující bod přerušení v editoru kódu. Značka je definována jako rozšíření `org.eclipse.core.resources.markers`. Pro reprezentaci bodu přerušení je vhodné značku definovat jako potomka značky `org.eclipse.debug.core.breakpointMarker` nebo `org.eclipse.debug.core.lineBreakpointMarker`, která již definují standardní chování a vykreslování značky reprezentující bod přerušení. Jelikož je kód assembleru orientován řádkově (jedna instrukce

na jednom řádku), byl vybrán druhý typ bodu přerušení. Značka byla také nastavena jako perzistentní, aby byly nastavené body přerušení zachovány i po zavření Eclipse.

Samotný bod přerušení musí být definován jako rozšíření `org.eclipse.debug.core .breakpoints` a k němu přiřazena třída provazující bod přerušení se značkou. Třída byla implementována s názvem `PBSimulatorLineBreakpoint` v balíčku `cz.vutbr.fit.vlam .pbsimulator.debug` jako potomek třídy `LineBreakpoint` a v konkstruktoru třídy jsou nastavovány parametry značky, na základě nichž se bod přerušení zobrazí v editoru kódu a v pohledu *Breakpoints*.

Definice bodu přerušení a jeho zobrazení v editoru je pouze prvním krokem. Dále musela být pro každý editor kódu, ve kterém má být podpora bodů přerušení, zaregistrována továrna a adaptér mající na starosti přidávání a odebírání bodů přerušení. Editory pro assembler mikroprocesoru PicoBlaze jsou rozšířením editoru XtextEditor, proto byla továrna a adaptér zaregistrována k tomuto typu. Adaptér musí být typu `org.eclipse.debug.ui .actions.IToggleBreakpointsTarget`. Samotná továrna byla implementována ve třídě `LineBreakpointAdapterFactory` a v metodě `getAdapter()` poskytuje adaptér pouze pro editor, ve kterém je otevřen `.psm` nebo `.asm` soubor.

Adaptér je implementován ve třídě `LineBreakpointAdapter` a pomocí něj je řízeno, zda může být bod přerušení na vybraném řádku programu vytvořen nebo nikoliv. K tomuto účelu slouží metoda `canToggleLineBreakpoints()`. Vzhledem k tomu, že program pro mikroprocesor PicoBlaze nemusí obsahovat pouze instrukce, ale např. prázdné řádky, komentáře, návěští, direktivy assembleru apod., muselo být omezeno přidávání bodů přerušení pouze na ty řádky kódu, pro které to má význam. Pro rozpoznávání řádků kódu obsahujících instrukci jsem se rozhodl využít implementace editorů PBPsm a PBAsm. Tyto editory jsou vygenerovány na základě specifikované gramatiky popisující syntaxi assembleru PicoBlaze a za běhu je z editoru možné získat jméno symbolu gramatiky, který reprezentuje aktuální výběr v editoru. Symbol lze získat voláním metory `EObjectAtOffsetHelper .resolveElementAt()`. Všechny instrukce jsou v editorech reprezentovány symboly končící slovem `Key` (např. `AddKey`, `CallJumpKey`, apod.). Přidávání bodu přerušení tak bylo omezenou pouze na ty řádky, pro které jsou v editorech vráceny symboly reprezentující instrukce. Definice gramatik v editorech PBPsm a PBAsm se v některých částech liší, avšak z hlediska názvu symbolů reprezentujících instrukce jsou tyto gramatiky shodné. Nevýhodou zvoleného řešení je fakt, že vkládání bodů přerušení je závislé na implementaci (či spíše správnosti definice gramatiky) editorů assembleru.

Další důležitou metodou v adaptéru `LineBreakpointAdapter` je metoda `toggleLineBreakpoints()`, ve které probíhá vytváření nových nebo mazání existujících bodů přerušení. Pro vytvoření nového bodu přerušení nestačí pouze vytvořit nový objekt `PBSimulatorLineBreakpoint`, ale bod přerušení musí být přidán i do objektu typu `IBreakpointManager` majícího na starosti správu všech bodů přerušení. Přidání nového bodu přerušení se provede pomocí volání metody `DebugPlugin.getDefault() .getBreakpointManager().addBreakpoint()`.

Jako poslední krok v implementaci podpory bodů přerušení byly editory pro assembler PicoBlaze rozšířeny o možnost vložit/zrušit bod přerušení pomocí dvojkliku na levé svislé liště editorů a vložení/zrušení bodu přerušení přes kontextové menu levé svislé lišty editorů. Pro zavedení podpory dvojkliku slouží rozšíření `org.eclipse.ui.editorActions` a definice akce `RulerDoubleClick`. Nastavení akce muselo být provedeno zvlášť pro oba editory assembleru (PBAsm a PBPsm). Přidání možnosti přepínání bodů přerušení přes kontextové menu lišty bylo implementováno pomocí rozšíření `org.eclipse.ui.popupMenus`.

6.2.3 Prováděná instrukce

O zvýraznění aktuálně prováděné instrukce v editoru kódu lze na základě poskytnutých dat a implementace podpůrných částí využít standardní funkčnosti Eclipse. Vykreslování probíhá na základě objektu reprezentujícího rámec zásobníku, který obsahuje informaci o zdrojovém souboru, ke kterému se vztahuje, a číslo odpovídajícího řádku ve zdrojovém souboru.

Rámec zásobníku jako objekt třídy `PBSimulatorStackFrame` je tvořen v metodě `PBSimulatorDebugTarget.getStackFrames()`. Zdrojový soubor je znám od spouštění simulace a číslo řádku v souboru je získano na základě hodnoty programového čítače z posledního známého stavu mikroprocesoru a interní reprezentace programu s mapováním programových adres na řádky kódu.

Kromě poskytnutí informací, na jejichž základě je zvýrazněna aktuálně prováděná instrukce, bylo potřeba implementovat ještě podpůrné třídy pro vyhledání zdrojového souboru v rámci otevřených projektů a jeho otevření v určeném editoru. Těmito třídami jsou `PBSimulatorSourceLookupDirector`, `PBSimulatorSourceLookupParticipant`, `PBSimulatorSourcePathComputerDelegate` v balíčku `cz.vutbr.fit.vlam.pbsimulator.debug.sourceLookup`, které mají na starosti vyhledání určitého zdroje. Ve třídě `PBSimulatorModelPresentation` z balíčku `cz.vutbr.fit.vlam.pbsimulator.ui.launching` je pak v metodě `getEditorId()` určeno, v jakém editoru mají být jednotlivé soubory otevřeny. V rámci simulátoru je pracováno pouze s `.asm` a `.psm` soubory, pro než je přiřazen editor PBAsm, resp. PBPsm.

Výše uvedené třídy musely být rovněž zaregistrovány jako rozšíření `org.eclipse.debug.core.sourceLocators`, `org.eclipse.debug.core.sourcePathComputers` a `org.eclipse.debug.ui.debugModelPresentations` v souboru `plugin.xml`.

6.2.4 Krokování

V rámci simulátoru byly implementovány dva způsoby krokování:

1. krok na následující instrukci (step into)
2. krok na následující adresu (step over)

První způsob je jednoduché postupné provádění instrukce po instrukci, které následuje skoky, volání podprogramů a obsluhu přerušení. Jeden krok je proveden pomocí příkazu `next`.

Složitějším případem je druhý způsob, kdy je krok proveden na následující adresu. V jednom kroku může být provedena pouze jedna instrukce, ale v případě, že aktuální instrukce je skok nebo volání podprogramu, v jednom kroku může být proveden v podstatě libovolný počet instrukcí.

Tento způsob krokování je implementován tak, že v simulátoru spblaze je nastaven bod přerušení na adresu o jednotku vyšší, než je hodnota programového čítače. Bod přerušení je brán jako interní a není vykreslen do editoru kódu. Poté je simulace spuštěna příkazem `run`. Jakmile je simulace opět pozastavena, což může být jednak z důvodu, že bylo dosaženo nastaveného bodu přerušení a byl proveden požadovaný krok, nebo při běhu bylo dosaženo jiného bodu přerušení, případně uživatel simulaci pozastavil sám, je nastavený interní bod přerušení odebrán.

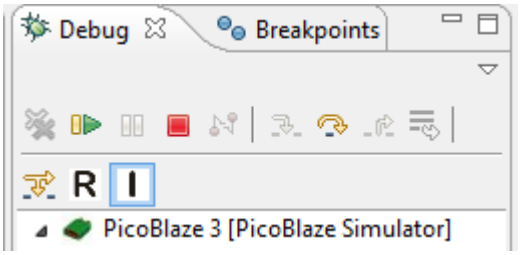
6.2.5 Reset a žádost o přerušení

Možnost resetování simulovaného mikroprocesoru a aktivace či deaktivace žádosti o přerušení byla implementována jako rozšíření nástrojové lišty v pohledu Debug. Standardní tlačítka na této nástrojové liště slouží k ovládání simulace (běh, pozastavení, krokování, atd.) a reset a ovládání žádosti o přerušení také ovlivňuje stav simulace. Proto bylo vybráno umístění těchto ovládacích funkcí do pohledu Debug.

Resetování mikroprocesoru bylo implementováno jako obyčejné tlačítko, žádost o přerušení jako přepínací tlačítko, kdy stav tlačítka definuje stav žádosti o přerušení. Tlačítka jsou viditelná pouze v perspektivě PB Simulation.

Tlačítka jsou do nástrojové lišty pohledu Debug přidány jako rozšíření `org.eclipse.ui.menus` ve formě příkazů. Samotné příkazy jsou také rozšířením `org.eclipse.ui.commands` a třídy implementující funkčnost jsou příkazům přiřazeny v rámci rozšíření `org.eclipse.ui.handlers`. Těmito třídami jsou `ResetCommandHandler` a `IrqCommandHandler` v balíčku `cz.vutbr.fit.vlam.pbsimulator.ui.views.PicoBlazeView`. Z těchto tříd je funkčnost delegována na pohled PicoBlaze a z něj dále pomocí událostí do komunikátoru se simulátorem.

Výsledek je možné vidět na obrázku 6.3, kde jsou přidány tlačítka "R"a "I"pro reset mikroprocesoru a ovládání žádosti o přerušení.



Obrázek 6.3: Rozšíření nástrojové lišty pohledu Debug

6.3 Distribuce zásuvného modulu

Vytvořený zásuvný modul může být mezi uživatele distribuován dvěma různými způsoby, které ovlivňují také způsob instalace zásuvného modulu do vývojového prostředí Eclipse.

Prvním způsobem je exportování vytvořeného modulu do `jar` souboru. Tento soubor si pak uživatel nahraje v rámci Eclipse do složky `dropins`. Z pohledu vývojáře jde o velice jednoduchý způsob, který však neposkytuje žádné pokročilejší možnosti nastavení.

Druhým způsobem, který poskytuje řadu nastavení (např. kontrolu závislostí na jiných zásuvných modulech, vložení licenčního ujednání, které se zobrazí při instalaci apod.), je vytvoření dvou speciálních projektů *Feature Projekt* a *Update Site Project*. Tyto projekty jsou navázány na vytvořený zásuvný modul a poskytují funkčnost, na jejímž základě může být zásuvný modul instalován do Eclipse pomocí standardní funkce instalace nových částí (menu *Help - Install New Software...*). Tento způsob kromě možností nastavení přináší také lepší komfort pro uživatele.

Pro zásuvný modul grafického simulátoru byl vybrán druhý způsob distribuce a pro tento účel byly vytvořeny projekty `cz.vutbr.fit.vlam.pbsimulator.feature` a `cz.vutbr.fit.vlam.pbsimulator.updatesite`. Na základě těchto projektů pak byla vygenerována sada souborů, která byla umístěna na webový server s internetovou adresou

`http://pbsimulator.jirisimek.cz`. Z této adresy je možné zásuvný modul nainstalovat do prostředí Eclipse. Podrobný postup instalace zásuvného modulu je uveden v příloze C.

Kapitola 7

Testování a možnosti rozšíření

Testování je nedílnou součástí vývoje každého softwaru a testování bulo prováděno také v rámci této práce při implementaci simulátoru spblaze i grafického simulátoru.

7.1 Testování simulátoru spblaze

Testování vytvářeného portu simulátoru μCsim pro mikroprocesor PicoBlaze bylo prováděno souběžné s jeho vývojem. Nejprve bylo nutné otestovat zpracování HEX souboru a správnost definice tabulky instrukcí, na základě které probíhá dekódování jednotlivých instrukcí. Následně bylo na jednoduchých příkladech testováno provádění jednotlivých instrukcí. Samostatnou částí pak bylo testování implementovaných rozšíření od zpracování XML souborů, přes jejich interní reprezentaci v simulátoru až po ověření, že jsou data simulátorem správně vyhodnoneny a použity (např. vstupy a jejich časování). Toto testování probíhalo manuálně.

Kromě výše uvedeného testování byla sestavena také automatizovaná sada testů pro ověření funkčnosti implementovaného portu simulátoru μCsim. Jednotlivé testy v této sadě postupně ověřují správnost chování instrukcí. Testy jsou koncipovány tak, že po spuštění simulátoru a nahrání testovacího programu je nastaven bod přerušení na poslední instrukci v testovaném programu. Poté je simulace spuštěna. Po dosažení nastaveného bodu přerušení je simulace ukončena a výstupem je stav mikroprocesoru v XML souboru. Tento stav je následně porovnán s očekávaným stavem, a pokud se shodují, je test považován za úspěšný.

Na základě sestavené sady testů byla také demonstrována možnost použití simulátoru pro automatizované testování.

7.2 Testování grafického simulátoru

Spolu s postupným vývojem grafického simulátoru v prostředí Eclipse probíhalo testování dílčích implementovaných částí. Postupně taky byly testovány nevizuální části, jako např. spuštění assembleru a následné spuštění samotného simulátoru a komunikace grafického simulátoru se simulátorem spblaze. Následně pak byla testována funkčnost částí vztahujících se prvkům uživatelského rozhraní prostředí Eclise. Zejména šlo o ovládání simulace z pohledu *Debug*, zadávání bodů přerušení a zobrazování aktuálně prováděné instrukce a v neposlední řadě také zobrazování stavu mikroprocesoru a změna hodnot v pohledu *PicoBlaze*.

7.3 Možnosti rozšíření simulátoru spblaze

Simulátor μCsim lze použít jako samostatný simulátor, ale je také distribuován jako součást překladače SDCC, kde je použit v debuggerem sdcdb. Vzhledem k tomu, že v rámci SDCC byl také vytvořen port pro mikroprocesor PicoBlaze (PBCC), nabízí se možnost konfigurovat debugger sdcdb tak, aby byl schopný používat v simulátoru μCsim i port pro PicoBlaze. Je zde však jeden zásadní problém, a tím je chybějící linker v PBCC. Vzhledem k jednoduchosti mikroprocesoru není linker prakticky potřeba, ale debugger sdcdb předpokládá na vstupu CDB soubor generovaný linkerem. Před samotným rozšířením debuggeru sdcdb by tedy musel být implementován linker pro PBCC.

Pokud by se podařilo rozšířit debugger sdcdb tak, aby byl schopný používat v rámci simulátoru μCsim i port pro PicoBlaze, byla by tímto krokem zkompletována v rámci SDCC celá sada nástrojů pro mikroprocesor PicoBlaze. Konečným krokem by mohlo být pokusit se požádat o zařazení portu pro PicoBlaze do oficiální distribuce SDCC, nicméně předtím by musely být vyřešeny problémy popsané v kapitole 4.9 plynoucí z použití knihovny STL.

7.4 Možnosti rozšíření grafického simulátoru v prostředí Eclipse

V grafickém simulátoru byla implementována možnost zadávat standardní body přerušení, které jsou založeny na dosažení určitého místa v programu při jeho běhu. Simulátor μCsim umožňuje kromě těchto standardních bodů přerušení zadávat i dočasné body přerušení, které se odlišují tím, že po jejich dosažení jsou automaticky odstraněny a při následujícím průchodu stejným místem již není běh pozastaven. Implementace dočasnýh bodů přerušení i v grafickém simulátoru by uživatelovi přinesla jemnější možnosti nastavení při simulaci.

Kromě výše zmíněných bodů přerušení disponuje simulátor μCsim i tzv. událostními body přerušení. Tyto body přerušení lze nastavit na události čtení nebo zápisu hodnoty konkrétní paměťové buňky. Pamětí je pak myšlen jeden z adresových prostorů implementovaného simulátoru. V případě simuláoru spblaze je to RAM, ROM, zásobník volání podprogramů a díky zvolenému způsobu implementace také registry. Zavedení možnosti zadávání událostních bodů přerušení do grafického simulátoru by rovněž zvýšilo jeho schopnosti a uživatelské možnosti řízení simulace. Nevýhodou událostních bodů přerušení je to, že se nedají narozdíl od klasických bodů přerušení zrušit.

Slabou stránkou implementovaného grafického simulátoru je omozená možnost práce se vstupními a výstupními porty. Simulátor umožňuje zadat vstupy pouze formou XML souboru a zobrazována je pouze poslední výstupní hodnota a číslo výstupního portu. Tento nedostatek by mohlo vyřešit rozšíření pohledu *PicoBlaze* o funkčnost, kdy by si uživatel mohl definovat seznam vstupních a výstupních portů, které chce sledovat, a hodnoty na těchto portech přímo upravovat. Dalším krokem by pak mohla být implementace komponent simulujících vstupně/výstupních zařízení, jako např. asynchronní komunikační rozhraní (universal asynchronous reciever/transmiter, UART).

Dalším možným rozšířením grafického simulátoru je možnost změny zdrojového kódu programu i při běhu simulace. Jelikož se mi bohužel nepodařilo nalézt způsob přepnutí editoru do režimu pouze pro čtení, je toto sice možné i v implementované verzi grafického simulátoru, ale změny nejsou propagovány do simulátoru a simulaci je nutné restartovat. V idálním případě by se změny provedené v editoru kódu měly ihned projevit v simulátoru. K provedení této změny za běhu by musel být upravený program nejprve znova přeložen

assemblerem a nový HEX soubor následně nahrán do simulátoru. V souvislosti s tím však vzniká potenciální problém, kdy překlad nemusí proběhnout úspěšně. Dále by bylo potřeba vyřešit, zda v upraveném programu pokračovat z aktuálního místa, na kterém se však může nové nacházet jiná instrukce, anebo se pokusit vyhledat původní instrukci v upraveném programu a podle výsledku upravit také hodnotu prográvého čítače.

Kapitola 8

Závěr

Cílem této diplomové práce bylo vytvoření grafického simulátoru mikroprocesoru PicoBlaze v prostředí Eclipse jako rozšíření existujícího vývojového prostředí VLAM IDE.

Tvorba simulátoru byla rozdělena do dvou samostatných částí - implementace jádra simulátoru a tvorba grafického simulátoru využívajícího vytvořené jádro. Pro implementaci jádra simulátoru bylo zvoleno rozšíření simulátoru μCsim o nový port pro mikroprocesor PicoBlaze. Vzhledem k nedostupnosti podrobnější dokumentace k simulátoru μCsim bylo nutné nejprve nastudovat strukturu a princip fungování simulátoru μCsim. Zjištěné poznatky byly podrobně popsány v této práci a jejich základě byl sestaven obecný návod na vytvoření nového portu v simulátoru μCsim.

Následně byl navržen a implementován port simulátoru μCsim pro mikroprocesor PicoBlaze 3 a 6 jako simulátor spblaze. V rámci návrhu byla prezentována řada rozšíření, jejichž cílem je vylepšení schopností simulátoru zejména s ohledem možnost použití simulátoru při automatizovaném testování programů. Jedná se zejména o možnost definice dat formou XML souboru pro vstupní operace s možností nastavení časování, definice žádostí o přerušení, import a export stavu mikroprocesoru a zaznamenávání a export provedených výstupních operací.

Na základě implementovaného jádra simulátoru byl poté navržen a implementován grafický simulátor v prostředí Eclipse využívající toto jádro. Návrh grafického simulátoru byl prováděn se snahou o co největší integraci a využití standardní funkčnosti prostředí Eclipse. V grafickm simulátoru byly využity editory pro jazyk symbolických instrukcí z vývojového prostředí VLAM IDE a byly rozšířeny o možnost zadávání bodů přerušení a zobrazování prováděné instrukce. Simulátor dále disponuje možností parametrizace simulátoru, zadáváním vstupů formou XML soubor, různými možnostmi ovládání simulace (pozastavení, krokování, resetování, žádost o přerušení) a v neposlední řadě také interaktivním grafickým zobrazením stavu mikroprocesoru s možností uživatelské změny jednotlivých hodnot.

Možná rozšíření implementovaných simulátorů již byla diskutována v kapitolách 7.3 a 7.4. Zde bych zmínil zejména rozšíření překladače PicoBlaze C Compiler (PBCC) o linker a následnou úpravu ladicího nástroje sdcdb o možnost využití implementovaného simulátoru spblaze, čímž by se z překladače PBCC stala ucelená sada nástrojů pro vývoj a testování programů pro mikroprocesor PicoBlaze.

Z hlediska grafického simulátoru v prostředí Eclipse se nabízí více možností a směrů rozšíření. Určitým faktorem omezujícím schopnosti grafického simulátoru je použití assembleru pBlazASM pro překlad zdrojového souboru do HEX formátu, který dokáže zpracovat programy pro mikroprocesor PicoBlaze 6 pouze v dialektu jazyka symbolických instrukcí pBlazIDE. Zde se nabízí buď rozšířit schopnosti tohoto assembleru, nebo jej plně nahradit

vlastní implementací assembleru přímo jako součást grafického simulátoru.

Dalším vhodným zaměřením pro rozšíření grafického simulátoru by mohla být lepší uživatelská podpora pro sledování a ovládání vstupně/výstupních operací.

Literatura

[1] *kpicosim. A simulator and assembler for the picoblaze* [online]. 2008-03-31 [cit. 2012-11-22].
URL http://marksix.home.xs4all.nl/picoasm.html

[2] *kpicosim. A simulator and assembler for the picoblaze* [online]. 2009-10-02 [cit. 2012-12-11].
URL http://marksix.home.xs4all.nl/kpicosim.html

[3] *SDCC Compiler User Guide* [online]. 2012-03-10.
URL http://sdcc.sourceforge.net/doc/sdccman.pdf

[4] *SDCC - Small Device C Compiler* [online]. 2012-07-09 [cit. 2012-11-20].
URL http://sdcc.sourceforge.net/

[5] *PicoBlaze C Compiler* [online]. 2012-08-23 [cit. 2012-12-01].
URL http://www.fit.vutbr.cz/~meduna/work/doku.php?id=projects:vlam:pbcc:pbcc

[6] *pBlazSIM - pblazasm - pBlazSIM, a Picoblaze simulator* [online]. 2013-02-25 [cit. 2013-03-25].
URL http://code.google.com/p/pblazasm/wiki/pBlazSIM

[7] *Mediatronix* [online]. [cit. 2012-11-22].
URL http://www.mediatronix.org/pages/pBlazASM

[8] *pBlazIDE* [online]. [cit. 2012-12-10].
URL http://www.mediatronix.org/pages/pBlazIDE

[9] *pBlazSIM* [online]. [cit. 2012-12-10].
URL http://www.mediatronix.org/pages/pBlazSIM

[10] *TinyXML-2* [online]. [cit. 2013-02-10].
URL http://www.grinninglizard.com/tinyxml2/index.html

[11] Chapman K.: *PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)* [online]. 2012-09-30.
URL http://www.xilinx.com/ipcenter/processor_central/picoblaze/member/KCPSM6_Release5_30Sept12.zip

[12] Clayberg, E.; Rubel, D.: *Eclipse: Building Commercial-quality Plug-ins*. Eclipse (Addison-Wesley), Addison Wesley Professional, 2006, ISBN 0-321-42672-X.

[13] Drótos D.: *Mikrocontroller Simulator* [online]. 1997 [cit. 2012-11-30].
URL http://mazsola.iit.uni-miskolc.hu/~drdani/embedded/ucsim/

[14] Dulík, T.; Křivka, Z.; Kadlec, J.; aj.: *Virtuální laboratoř pro vývoj aplikací s mikroprocesory a FPGA*. Akademické nakladatelství CERM, 2011, ISBN 978-80-7204-754-3, 82 s.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9727

[15] Dutta, S.: Anatomy of a Compiler: A Retargetable ANSI-C Compiler. *Circuit Cellar*, 2000.

[16] Edgar N.; Haaland K.; Li J.; Peter K.: *Eclipse User Interface Guidelines* [online]. 2004-02 [cit. 2013-04-12].
URL
http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html

[17] Fauck Ch.: *openPICIDE project site: About openPICIDE* [online]. 2010-08-14 [cit. 2012-12-11].
URL http://www.openpicide.org/content/about/

[18] Horník, J.: *Zadní část překladače podmnožiny jazyka C pro 8-bitový procesor*. Diplomová práce, FIT VUT v Brně, 2011.

[19] International Business Machines Corp: *Eclipse Platform Technical Overview* [online]. 2006-04-19 [cit. 2013-04-10].
URL http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html

[20] Mentor Graphics: *ModelSim - Advanced Simulation and Debugging* [online]. [cit. 2012-12-11].
URL http://model.com/

[21] The Eclipse Foundation: *Eclipse Java development tools (JDT)* [online]. 2013 [cit. 2013-04-10].
URL http://www.eclipse.org/jdt/

[22] The Eclipse Foundation: *PDE* [online]. 2013 [cit. 2013-04-10].
URL http://www.eclipse.org/pde/

[23] Xilinx: *PicoBlaze 8-bit Embedded Microcontroller User Guide* [online]. 2011-06-22.
URL
http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf

[24] Xilinx: *PicoBlaze 8-bit Microcontroller* [online]. [cit. 2012-11-10].
URL http://www.xilinx.com/products/intellectual-property/picoblaze.htm

Seznam příloh

Příloha 1: CD
Obsah CD:

- `eclipse` - zdrojové soubory grafického simulátoru v prostředí Eclipse
- `pblazasm` - upravená verze assembleru pBlazASM
  - `linux` - verze pro Linux
  - `win` - verze pro Windows ve formě projektu vývojového prostředí Code-Blocks. Kromě zdrojových souborů je přiložena i spustitelná verze.
- `test` - připravená sada testů pro simulátor μCsim
- `text` - zdrojové soubory textu této práce
- `ucsim` - zdrojové simulátoru μCsim, stručný návod na kompilaci, spustitelná verze pro Windows a skripty pro aktualizaci konfiguračních souborů popsaných v kapitole 3.2.11.

Příloha A

Parametry a příkazy simulátoru PicoBlaze

V této příloze je uveden popis nově implementovaných parametrů ovlivňujících vlastnosti a chování simulátoru a nových příkazů simulátoru. Popis výchozích parametrů a příkazů můžete nalézt v [13].

Přepínač	Funkce
-t typ	Jde o standardní přepínač pro určení typu mikroprocesoru. Typ může být kcpsm3 nebo kcpsm6.
-x	Zdrojový program není v Intel HEX formátu, ale v obyčejném HEX formátu.
-i soubor	Načte XML soubor s přerušeními.
-n soubor	Načte XML soubor se vstupními daty.
-m hodnota	Velikost RAM paměti. Povolené hodnoty jsou 64, 128 nebo 256. Lze použít pouze pro PicoBlaze 6.
-M hodnota	Velikost programové paměti. Povolené hodnoty jsou 1024, 2048 nebo 4096. Lze použít pouze pro PicoBlaze 6.
-I hodnota	Adresa vektoru přerušení. Pouze pro PicoBlaze 6.
-w hodnota	Specifikace vestavěné hardwarové konstanty. Pouze pro PicoBlaze 6.
-o soubor	Při ukončení simulátoru je uložen stav mikroprocesoru a jeho výstupy do XML souborů s názvem `<soubor>_state.xml` a `<soubor>_output.xml`.

Tabulka A.1: Seznam nově implementovaných parametrů simulátoru

Příkaz	Funkce
pbstate ["soubor"]	Vypíše aktuální stav mikroprocesoru v XML formátu. Pokud není zadán název souboru, stav procesoru je vypsán na standardní výstup, v opačném případě je stav uložen do souboru.
import pbstate "soubor"	Nahraje stav procesoru ze zadaného XML souboru.
import input "soubor"	Z XML souboru nahraje vstupní data. Stejná funkčnost jako přepínač -n.
import interrupts "soubor"	Z XML souboru nahraje seznam s žádostmi o přerušení. Jde o stejnou funkci jako přepínač -i.
get output ["soubor"]	Příkaz slouží pro vypsání hodnot v XML formátu, které byly instrukcí OUTPUT nastaveny na výstupním portu. Pokud není zadán název souboru, XML s výstupy je vypsáno na standardní výstup.

Tabulka A.2: Seznam nově implementovaných příkazů simulátoru

Příloha B

Vzor souboru `Makefile.in`

Níže je uveden vzorový příklad souboru Makefile.in, který se nachází mezi zdrojovými kódy každého portu a na základě tohoto souboru je příkazem `configure` generován soubor `Makefile`. Ve vzorovém souboru musí být místo značek `<...>` doplněny odpovídající hodnoty. Jsou použity hlavn2 tyto značky:

- `<název portu>` - pojmenování nového portu, např. `pblaze`.
- `<složka portu>` - složka se zdrojovými kódy nového portu, zpravidla `<název portu>.src`, tedy např. `pblaze.src`
- `<spustitelný soubor>` - název výsledného spustitelného souboru, obvykle `s<název portu>`, tedy např. `spblaze`

```
#
# uCsim <složka portu>/Makefile
#
# <autor>
#

STARTYEAR  = <rok>

SHELL    = /bin/sh
CXX    = @CXX@
CPP    = @CPP@
CXXCPP    = @CXXCPP@
RANLIB    = @RANLIB@
INSTALL    = @INSTALL@
STRIP    = @STRIP@
MAKEDEP         = @MAKEDEP@

top_builddir  = @top_builddir@
top_srcdir  = @top_srcdir@

DEFS            = $(subs -DHAVE_CONFIG_H,,@DEFS@)
CPPFLAGS        = @CPPFLAGS@ -I$(srcdir) -I$(top_srcdir) -I$(top_builddir)
```

```
      -I$(top_srcdir)/cmd.src -I$(top_srcdir)/sim.src

      -I$(top_srcdir)/gui.src
CFLAGS          = @CFLAGS@ @WALL_FLAG@
CXXFLAGS        = @CXXFLAGS@ @WALL_FLAG@
LDFLAGS         = @LDFLAGS@
PICOPT    = @PICOPT@
SHAREDLIB  = @SHAREDLIB@
EXEEXT    = @EXEEXT@

LIBS = -L$(top_builddir) -lsim -lucsimutil -lguiucsim -lcmd -lsim @LIBS@
DL    = @DL@
dl_ok    = @dl_ok@

prefix          = @prefix@
exec_prefix     = @exec_prefix@
bindir          = @bindir@
libdir          = @libdir@
datadir         = @datadir@
datarootdir     = @datarootdir@
includedir      = @includedir@
mandir          = @mandir@
man1dir         = $(mandir)/man1
man2dir         = $(mandir)/man2
infodir         = @infodir@
srcdir          = @srcdir@
VPATH           = @srcdir@

OBJECTS_SHARED  = <seznam sdílených objektových souborů .o>
OBJECTS_EXE  = <objektový soubor výsledného programu> -
"<spustitelný soubor>.o">
OBJECTS    = $(OBJECTS_SHARED) $(OBJECTS_EXE)

enable_dlso  = @enable_dlso@
dlso_ok    = @dlso_ok@

<název portu>ASM    = <cesta k assembleru mikroprocesoru>
TEST_OBJ  = <seznam testovacích programů mikroprocesoru v HEX formátu>


# Compiling entire program or any subproject
# ------------------------------------------
all: checkconf otherlibs <složka portu>


# Compiling and installing everything and runing test
# ---------------------------------------------------
install: all installdirs
```

```
  $(INSTALL) <název výsledného spustitelného programu,
zpravidla "s<název portu>">$(EXEEXT)
$(DESTDIR)$(bindir)/<spustitelný soubor>$(EXEEXT)
  $(STRIP) $(DESTDIR)$(bindir)/<spustitelný soubor>$(EXEEXT)


# Deleting all the installed files
# --------------------------------
uninstall:
  rm -f $(DESTDIR)$(bindir)/<spustitelný soubor>


# Performing self-test
# --------------------
check: $(TEST_OBJ)

test:


# Performing installation test
# ----------------------------
installcheck:


# Creating installation directories
# ---------------------------------
installdirs:
  test -d $(DESTDIR)$(bindir) || $(INSTALL) -d $(DESTDIR)$(bindir)


# Creating dependencies
# ---------------------
dep: Makefile.dep

Makefile.dep: $(srcdir)/*.cc $(srcdir)/*.h
  $(MAKEDEP) $(CPPFLAGS) $(filter %.cc,$^) >Makefile.dep

-include Makefile.dep
include $(srcdir)/clean.mk

# My rules
# --------
.SUFFIXES: .asm .hex

<složka portu>: <spustitelný soubor>$(EXEEXT) shared_lib

<spustitelný soubor>
$(EXEEXT): $(OBJECTS) $(top_builddir)/*.a
```

```
  $(CXX) $(CXXFLAGS) $(LDFLAGS) $(OBJECTS) $(LIBS) -o $@

ifeq ($(dlso_ok),yes)
shared_lib: $(top_builddir)/<spustitelný soubor>.so
else
shared_lib:
  @$(top_srcdir)/mkecho $(top_builddir)
"No <jméno mikroprocesoru/portu> shared lib made."
  @$(top_srcdir)/mkecho $(top_builddir)
"(SHAREDLIB="$(SHAREDLIB)",dl_ok="$(dl_ok)",enable_dlso="$(enable_dlso)")"
endif

$(top_builddir)/<spustitelný soubor>.so: $(OBJECTS_SHARED)
  $(CXX) -shared $(LDFLAGS) $(OBJECTS_SHARED) -o $(top_builddir)/
<spustitelný soubor>.so

otherlibs:
  $(MAKE) -C $(top_builddir)/cmd.src all
  $(MAKE) -C $(top_builddir)/sim.src all

.cc.o:
  $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@

.asm.hex:
  $(<název portu>ASM) <parametry a vstupní soubory assembleru>


# Remaking configuration
# ----------------------
checkconf:
  @if [ -f $(top_builddir)/devel ]; then

    $(MAKE) -f conf.mk srcdir="$(srcdir)"
top_builddir="$(top_builddir)" freshconf;

  fi

# End of <složka portu>/Makefile.in
```

Příloha C

Uživatelský manuál ke grafickému simulátoru mikroprocesoru PicoBlaze

Tento uživatelský manuál popisuje způsob instalace a ovládání grafického simulátoru pro mikroprocesor PicoBlaze. Grafický simulátor byl vytvořen jako zásuvný modul rozšiřující vývojové prostředí VLAM IDE, což je také zásuvný modul rozšiřující platfomu Eclipse. Vzhledem k tomu, že VLAM IDE bylo tvořeno a testováno pro Eclipse 3.6.2, grafický simulátor také vychází z této verze Eclipse

C.1 Instalace

1. Podle postupu instalace VLAM IDE, který je dostupný na adrese `http://www.fit.vutbr.cz/měduna/work/lib/exe/fetch.php?media=projects:vlam:public:vlam_ide_uzivatelska_prirucka_2012-08-10.pdf`, nejprve stáhněte vývojové prostředí Eclipse a následně nainstalujte zásuvný modul VLAM IDE. Po úspěšné instalaci VLAM IDE bude obdobným způsobem nainstalovám zásuvný modul pro simulátor mikroprocesoru PicoBlaze.

2. V menu *Help* klikněte na položku *Install New Software ...*

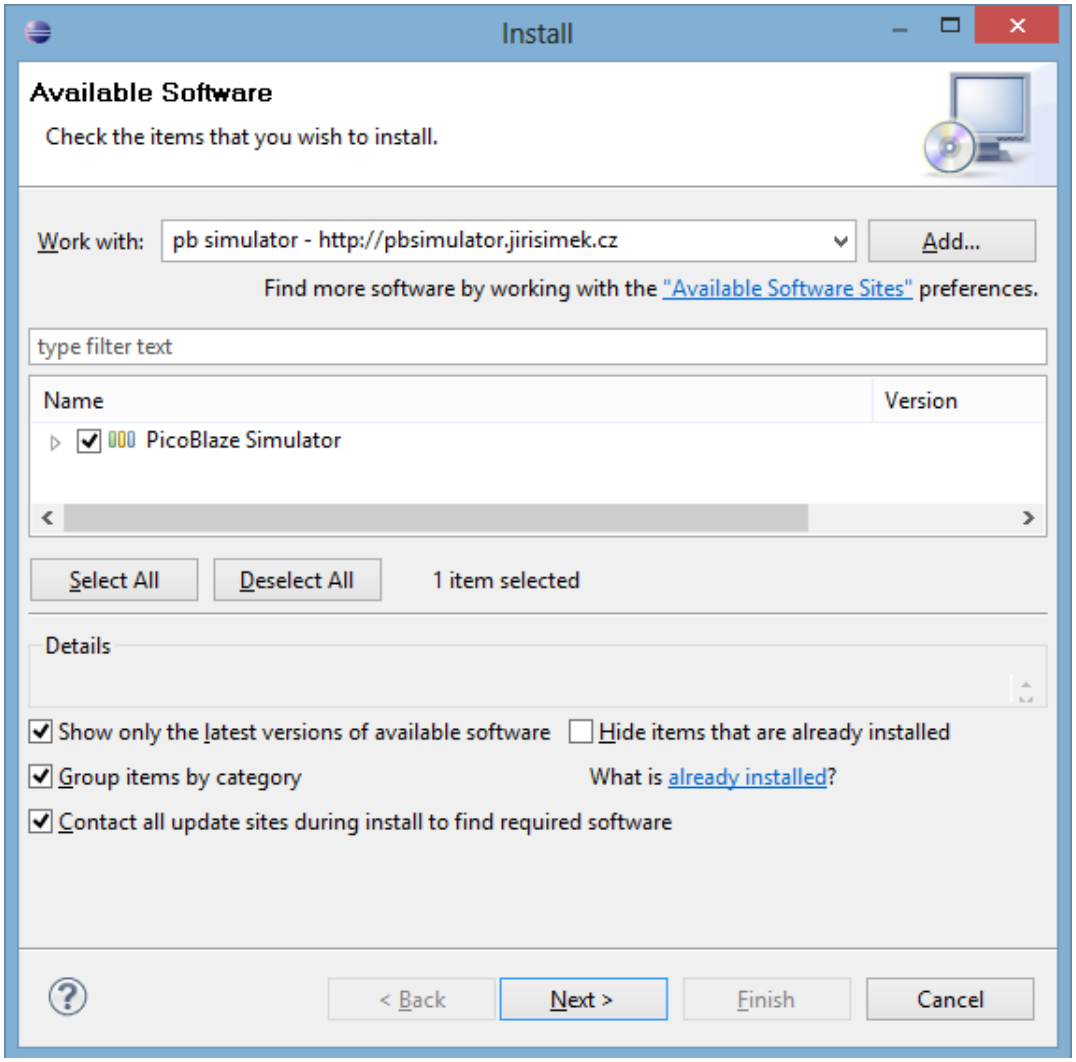
3. V zobrazeném dialogovém okně klikněte na tlačítko *Add...*

4. V dialogovém okně vyplňte v poli *Name* např. "PicoBlaze Simulator", do pole *Location* zadejte adresu `http://pbsimulator.jirisimek.cz` a klikněte na tlačítko *OK*.

5. Po přidání repozitáře by se měl načíst seznam zásuvných modulů obsažených v tomto repozitáři (viz obrázek C.1). Zaškrtněte položku *PicoBlaze Simulator* a klikněte na tlačítko *Next ¿*.

6. V následujícím kroku musíte potvrdit svůj souhlas s licencí a odsouhlasit, že instalovaný zásuvný modul není elektronicky podepsán a že si skutečně přejete jeho instalaci.

7. Po dokončení instalace budete vyzváni k restartování Eclipse a po restartu můžete ověřit správnost nainstalování simulátoru v menu *Help - About Eclipse*, kde po kliknutí na tlačítko `Installation Details` uvidíte seznam nainstalovaných zásuvných modulů a mezi nimi by měl být i *PB Simulator Feature*.



Obrázek C.1: Obsah repozitáře se simulátorem PicoBlaze

C.2 Konfigurace simulátoru

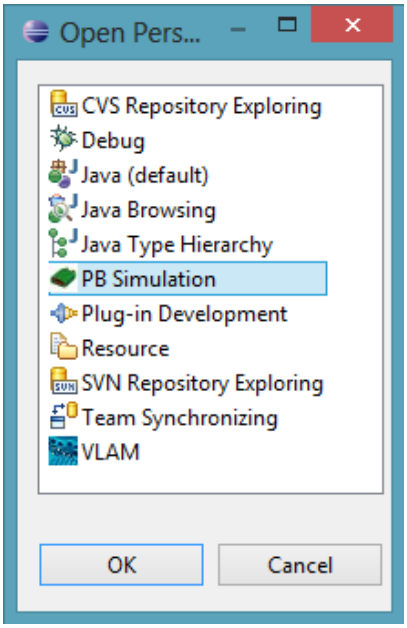
Po úspěšné instalaci zásuvného modulu simulátoru PicoBlaze je utné probvést ještě několik kroků, aby bylo možné jej používat.

1. Přidání a zobrazení perspektivy *PB Simulation* pomocí menu *Window - Open Perspective - Other...*, v zobrazeném okně vybrat *PB Simulation* a potvrdit tlačítkem *OK*. Výšěr perspektivy je ukázán na obrázku C.2.

2. Vytvoření a nastavení simulace a jejích parametrů. Toto je potřeba provést přes menu *Run - Debug Configurations...*. V zobrazeném okně (viz obrázek C.3 je potřeba přidat novou konfiguraci *PicoBlaze Simulator*.

   V rámci nastavení simulátoru je potřeba nastavit tyto položky:

   (a) Cesta ke spustitelnému souboru assembleru pBlazASM - spustitelný soubor pro Windows je možné stáhnout z adresy `http://pbsimulator.jirisimek.cz/pBlazASM.exe`.



Obrázek C.2: Výběr perspektivy *PB Simulation*

(b) Cesta ke spustitelnému souboru simulátoru spblaze - spustitelný soubor pro Windows je možné stáhnout z adresy `http://pbsimulator.jirisimek.cz/spblaze.exe`.

(c) Typ mikroprocesoru

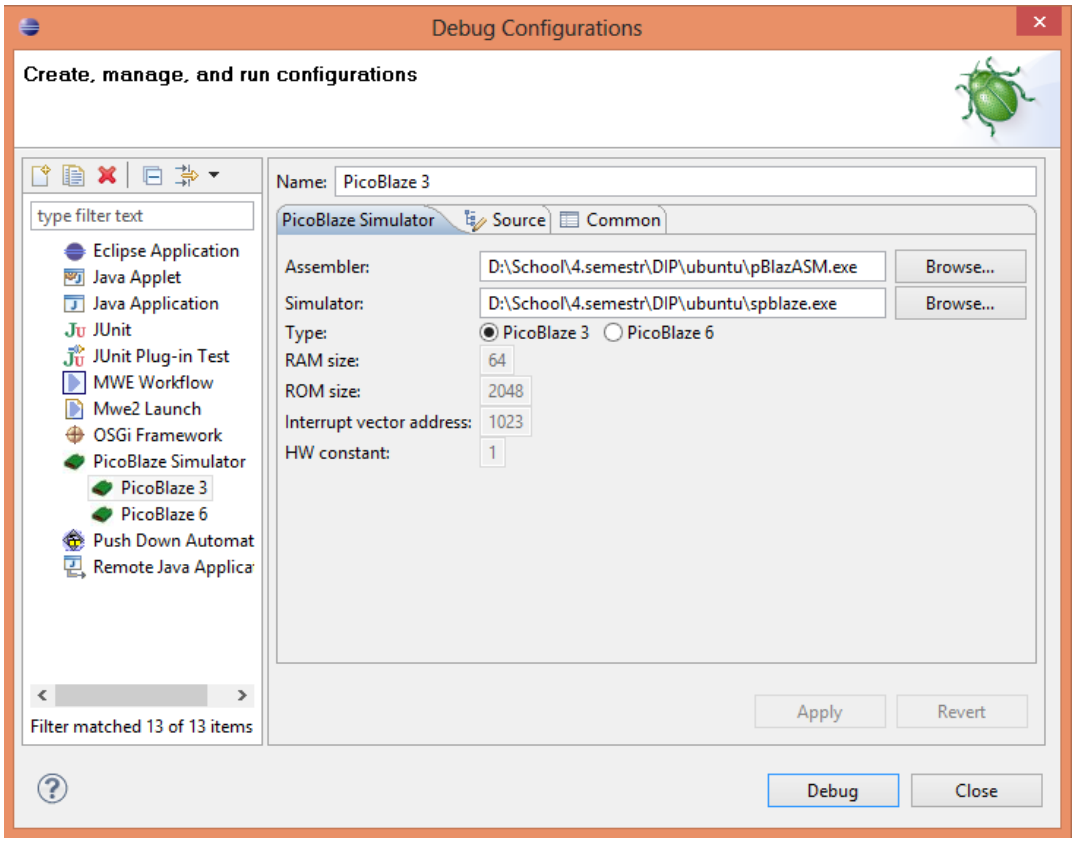
(d) V případě výběru mikroprocesoru PicoBlaze 6 je možné nastavit další parametry jako jsou velikosti RAM a ROM pamětí, vzor přerušení a harwdwarová konstanta.

(e) Nepovinně je také možné specifikova soubor se vstupními daty.

C.3 Spuštění a ovládání simulace

Pro spuštění simulace je potřeba provést následující kroky:

1. V editoru kódu otevřít zdrojový soubor programu s příponou `.psm` nebo `.asm`.
2. Přepnout se do perspektivy *PB Simulation*.
3. Zahájit simulaci z panelu nástrojů kliknutím na ikonu brouka (*Debug*) a vybrat v předchozí částí definovaný typ spuštění.
4. Následně je simulace spuštěna a z pohledu *Debug* je možné ji ovládat (pozastavit, spustit, krokovat a přerušit)



Obrázek C.3: Nastavení simulátoru