Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Diplomová práce

Multiplatformní grafická aplikace pro simulaci mikroprogramovaného procesoru DOP

Bc. Vojtěch Miškovský

Vedoucí práce: Ing. Pavel Kubalík, Ph.D

29. dubna 2015

Poděkování

Na tomto místě bych chtěl poděkovat svému vedoucímu práce panu Pavlu Kubalíkovi za rady a podporu během realizace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 29. dubna 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Miškovský, Vojtěch. *Multiplatformní grafická aplikace pro simulaci mikroprogramovaného procesoru DOP*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem této práce je vytvoření multiplatformní aplikace pro simulaci mikroprogramovaného procesoru DOP sloužící zejména pro účely výuky. Aplikace by měla studentům umožnit pochopení fungování mikroprogramovaného procesoru a implementaci vlastní instrukce do mikroprogramu.

Klíčová slova DOP, mikroprogramovaný procesor, simulátor, mikroinstrukce, mikrokód, instrukce

Abstract

This work aims to create cross-platform application for simulating microprogrammed processor DOP used for educational purpose. Using this application students should be able to understand principals of microprogrammed processors and implement their own instruction to DOP instruction set.

Keywords DOP, microprogrammed processor, simulator, microinstruction, microcode, instruction

Obsah

Úvod **1**

1 Cíl práce **3**
1.1 Současné simulátory 3
1.2 Nový simulátor 4

2 Analýza **7**
2.1 Procesor 7
2.2 Překladač 14
2.3 Uživatelské rozhraní 18

3 Návrh **23**
3.1 Výběr vývojové platformy 23
3.2 Struktura 24
3.3 Procesor 25
3.4 Překladač 28
3.5 Ovladač procesoru 30
3.6 Uživatelské rozhraní 31

4 Realizace **35**
4.1 Procesor 35
4.2 Překladač 40
4.3 Ovladač procesoru 44
4.4 Uživatelské rozhraní 47

5 Výstupy práce **63**

6 Testování **65**
6.1 Testování simulace 65
6.2 Testování překladu 66
6.3 Testování uživatelského rozhraní 68
6.4 Komplexní testy 68

Závěr **69**

Literatura **71**

A Seznam použitých zkratek **73**

B Obsah přiloženého CD **75**

Seznam obrázků

2.1 Digram případů užití simulátoru 7
2.2 Procesor DOP 8
2.3 Řadič procesoru DOP 9
2.4 Datová cesta 9
2.5 Aritmeticko-logická jednotka 10
2.6 Klopný obvod pro práci s přenosy a jeho logika 11
2.7 Klopný obvod pro umožnění/zamezení zápisu a jeho logika 11
2.8 Paměťový řadič 12
2.9 Grafické znázornění handshake protokolu pro komunikaci s pamětí 13
2.10 Grafické znázornění protokolu pro komunikaci s řadičem přerušení 14

3.1 Diagram struktury aplikace 24
3.2 Sekvenční diagram spolupráce aritmeticko-logické jednotky a stavového registru 25
3.3 Diagram tříd procesoru 27
3.4 Diagram tříd překladače 29
3.5 Diagram tříd uživatelského rozhraní 31

4.1 Podoba uživatelského rozhraní 48
4.2 Diagram aktivit metody *closeEvent* 49
4.3 Widget textového editoru 53
4.4 Widget pro zobrazení a úpravu vnitřních hodnot procesoru 55
4.5 Widget výstupu překladače 56
4.6 Widget návěstí 57
4.7 Widget řídící paměti 58
4.8 Widget hlavní paměti 59
4.9 Widget diagramu aritmeticko-logické jednotky 61
4.10 Widget diagramu multiplexoru následující adresy 62

Úvod

Procesor DOP začal vznikat na fakultě elektrotechnické již v devadesátých letech. Jeho architektura byla navrhována tak, aby se pro ni snadno tvořily kompilátory. Procesor byl implementován v FPGA a byl pro něj vytvořen překladač z jazyka C. Porovnání překladače s obdobnými překladači pro procesory Intel, Siemens, VAX a SPARC prokázalo vhodnost koncepce.

Řídící část původně obvodového procesoru byla pro potřeby výuky později vyměněna za horizontální mikroprogramovaný řadič a procesor byl dále zdokonalován. Byl mu například přidán řadič přerušení a vylepšena práce s příznaky.

Dále byly vytvořeny simulátory tohoto procesoru, které měly za cíl studentům lépe přiblížit funkčnost procesoru a umožnit jim implementovat v mikrokódu procesoru novou instrukci. Tyto simulátory se při výuce nadále využívají, ovšem ani jeden z nich není bezproblémově použitelný.

Je tedy potřeba vytvořit nový simulátor procesoru DOP, který skloubí dobré vlastnosti současných simulátorů, bude funkčně shodný s procesorem, uživatelsky přívětivý a schopný běhu na nejrozšířenějších moderních platformách. Díky tomuto simulátoru by již nemělo být potřeba při výuce volit mezi neoptimálními simulátory, či kombinovat jejich použití.

Kapitola 1

Cíl práce

Cílem práce je vytvořit simulátor procesoru DOP, který vyřeší všechny nedostatky současných simulátorů a bude moci být využíván pro výuku.

1.1 Současné simulátory

V současnosti jsou pro výuku používány dva simulátory, SIMDOP určený pro MS-DOS [1] a jSimDOP v jazyce Java [2].

1.1.1 SIMDOP

SIMDOP je prvním ze simulátorů procesoru DOP. Byl vytvořen v rámci diplomové práce v roce 2000 a i přes svou značnou zastaralost je dodnes používán.

1.1.1.1 Výhody

- správná funkčnost

1.1.1.2 Nevýhody

- běh pod MS-DOS — na moderních systémech nutnost emulace
- nízký výkon
- neobsahuje editor
- nepodporuje breakpointy
- nízká uživatelská přívětivost

1.1.2 jSimDOP

jSimDOP je simulátor napsaný v jazyce Java v rámci bakalářské práce v roce 2006. I tento simulátor je využíván pro výuku, ale z důvodu chyb pouze v kombinaci se SIMDOP. S některými verzemi Javy je dokonce problém se samotným spuštěním aplikace.

1.1.2.1 Výhody

- multiplatformnost
- integrovaný editor
- podpora breakpointů
- vyšší uživatelská přívětivost

1.1.2.2 Nevýhody

- problémy se spuštěním
- chyby v simulaci
- vyžaduje Java runtime
- zbytečná robustnost a vysoké systémové nároky (založený na náročném IDE NetBeans)
- místy neintuitivní ovládání

1.1.3 Shrnutí

V současnosti existují dva simulátory procesoru DOP. Jeden se velice nekomfortně používá z důvodu zastaralosti, druhý má problémy se správností samotné simulace, působí těžkopádně a navíc se objevují i problémy s jeho spuštěním. Nelze tedy bezproblémově používat ani jeden z nich.

1.2 Nový simulátor

Ze seznamu nevýhod současných řešení jsem vyvodil, že je třeba vytvořit simulátor nový, který bude kombinovat výhody současných simulátorů a vyvaruje se jejich nevýhod.

Mým cílem bude vytvořit simulátor s těmito vlastnostmi:

- správná funčnost
- multiplatformnost

- jednoduchá instalace
- integrovaný editor
- uživatelská přívětivost
- přehledné a intuitivní rozhraní
- maximální zpětná kompatibilita
- podpora breakpointů
- nízké systémové nároky

Splněním těchto cílů bude možné k výuce používat výhradně můj simulátor, což ušetří a zpříjemní práci studentům i vyučujícím. Budu se snažit o co největší kompatibilitu se simulátorem SIMDOP a zároveň vytvořit příjemné simulační prostředí.

Kapitola 2

Analýza

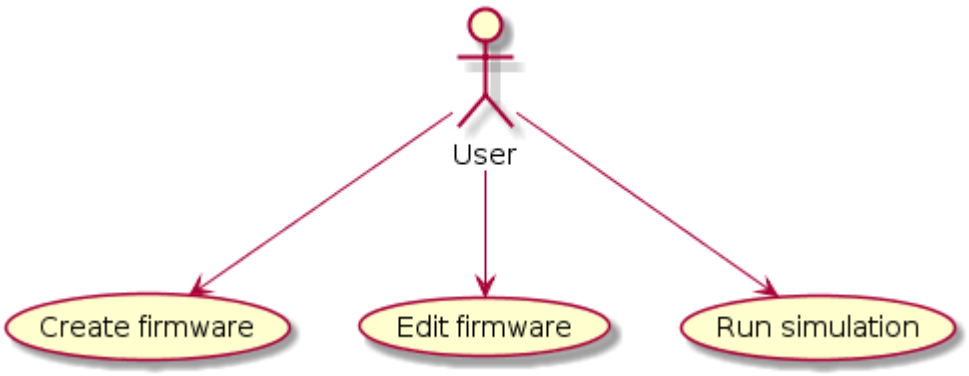
Simulátor bude sloužit k vytvoření a úpravě zdrojového kódu jazyka symbolických mikroinstrukcí, překladu zdrojového kódu do mikroprogramu a simulaci mikroprogramu a její detailní zobrazení a krokování. Dále bude umožňovat export mikroprogramu do binární podoby, vytvoření logu s informacemi o překladu či ukládání stavu procesoru a paměti.

Simulátor lze podle funkčnosti rozdělit na tři základní celky: překladač, procesor a uživatelské rozhraní. Tyto tři nezávislé části si podrobněji rozebereme.

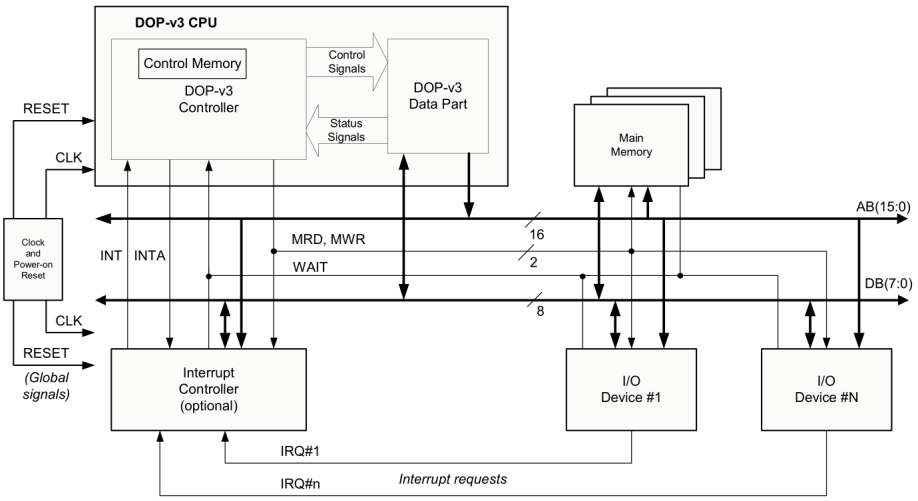
2.1 Procesor

Základním stavebním kamenem simulátoru procesoru je programová jednotka, která reprezentuje samotný procesor a simuluje jeho chování. Tuto jednotku budu pro účely této analýzy označovat jako procesor. Jeho znázornění je vidět na obrázku 2.2.

Obrázek 2.1: Digram případů užití simulátoru



Obrázek 2.2: Procesor DOP



Nejzásadnějším problémem, který budu muset vyřešit, je přemodelování paralelního (ve smyslu šíření signálu) hardwarového návrhu do čistě sekvenčního softwarového návrhu. Procesor DOP naštěstí neobsahuje kombinační smyčky, což přemodelování poměrně zjednodušuje.

Dalším souvisejícím rozhodnutím bude zvolení vhodné úrovně abstrakce procesoru, tedy do jaké hloubky bude simulátor přesně kopírovat hardware.

Procesor můžeme dále funkčně rozdělit na řadič, datovou cestu, paměť (která sice není reálně součástí procesoru, ale pro účely simulace ji tak budu považovat) datovou a adresní sběrnici a řadič přerušení.

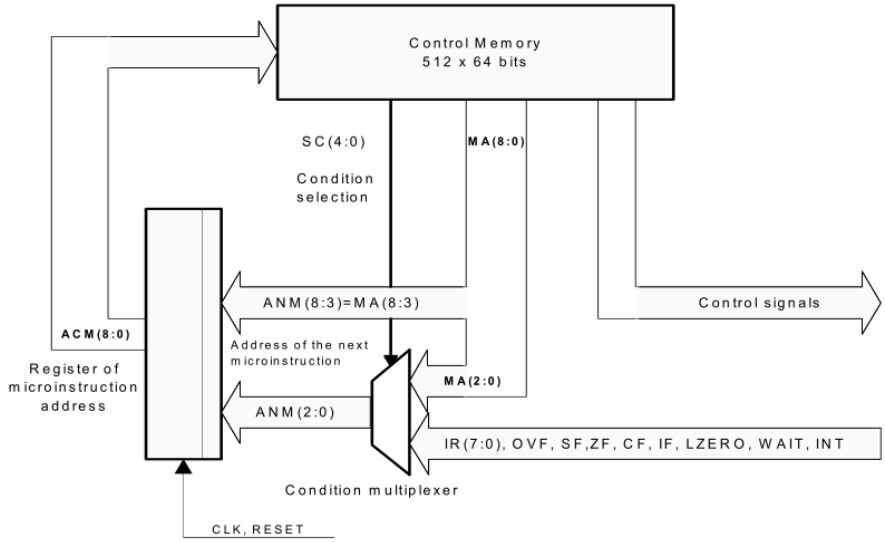
Použitá schemata procesoru a průběhy jsou převzata ze souboru schemat [3].

2.1.1 Řadič

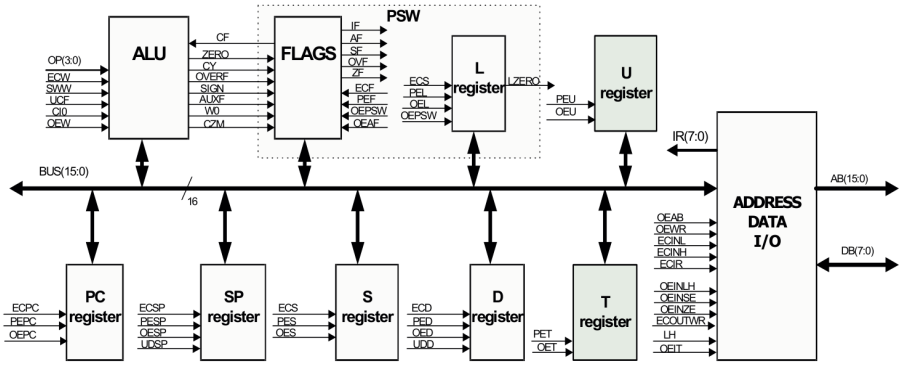
Řadič procesoru DOP je horizontální mikroprogramovaný řadič. Jeho stěžejní částí je paměť obsahující takzvané mikroinstrukce, registr obsahující adresu aktuálně prováděné mikroinstrukce a multiplexor pro podmíněné skoky.

Každá mikroinstrukce představuje jeden takt procesoru. Obsahuje logickou hodnotu všech řídících signálů datové cesty v daném taktu, operační znak pro aritmeticko-logickou jednotku, adresu následující mikroinstrukce a řídící signály pro multiplexor. V následujícím taktu je spuštěna mikroinstrukce, jejíž adresa je získána z adresy následující mikroinstrukce a dopočítána na základě řídících a stavových signálů multiplexorem.

Obrázek 2.3: Řadič procesoru DOP



Obrázek 2.4: Datová cesta



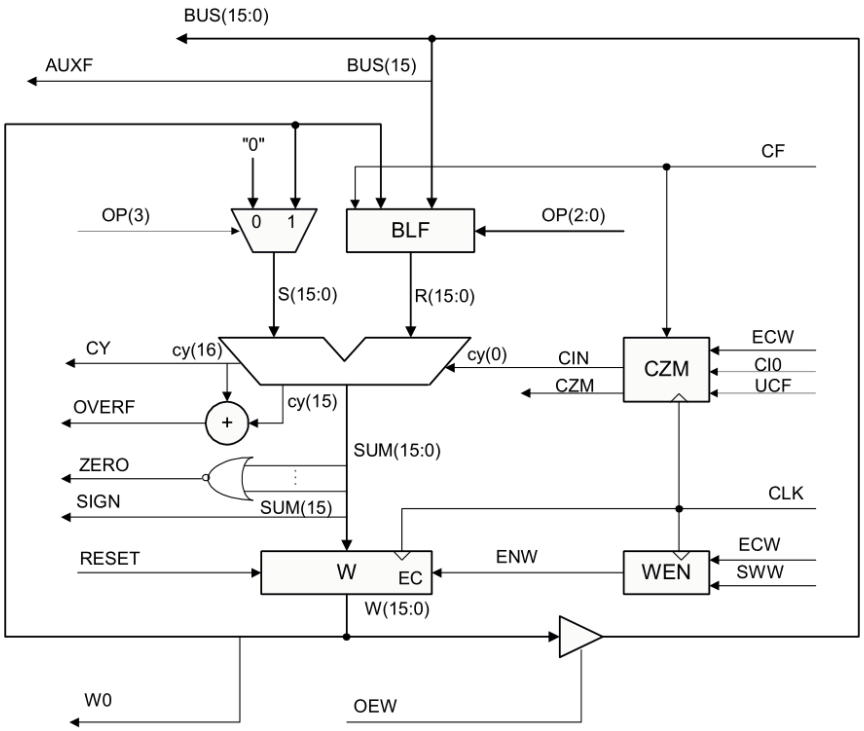
Vzhledem k jednoduchosti takového řadiče by jeho návrh a implementace měly být triviální.

Diagram řadiče je na obrázku 2.3.

2.1.2 Datová cesta

Základem datové cesty jsou registry a paměťový řadič. Vše je propojeno sběrnicí. Délka slova procesoru je 16 bitů. Detail datové cesty je na obrázku 2.4.

Obrázek 2.5: Aritmeticko-logická jednotka



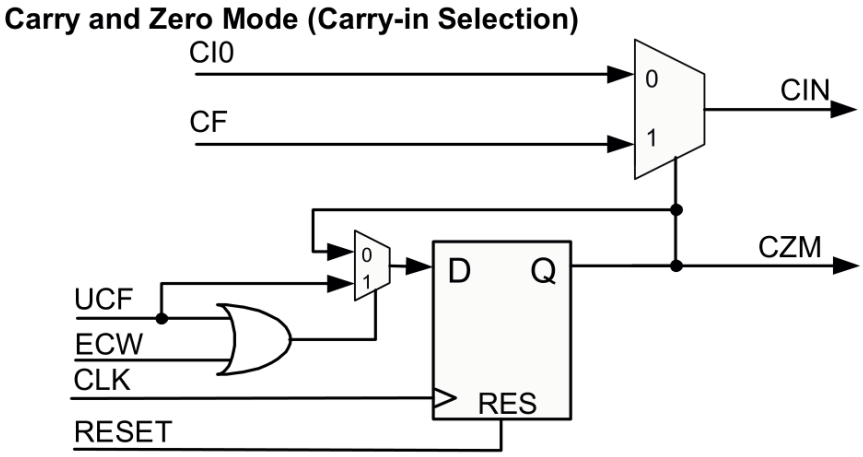
2.1.2.1 Registry

Základní registry slouží pro zápis dat na sběrnici a uložení dat ze sběrnice. Některé umožnují inkrementaci nebo dekrementaci. Speciálními registry jsou aritmeticko-logická jednotka (ALU) a stavový registr (PSW).

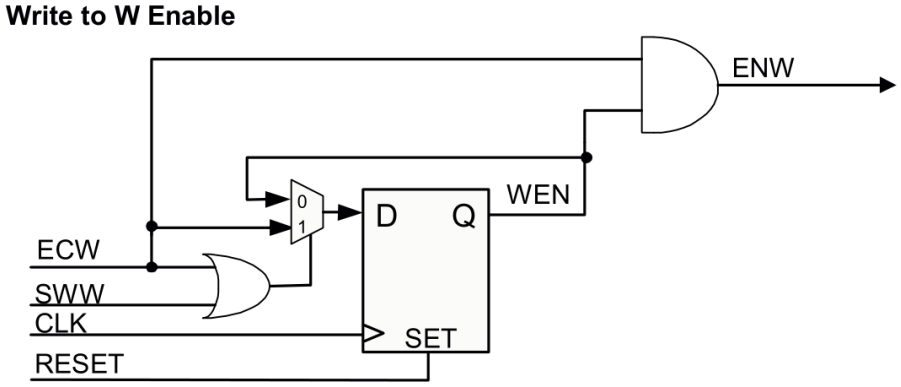
ALU jednotka na základě operačního znaku daného mikroinstrukcí vykoná aritmetickou či logickou operaci. Operační znak také určuje vstupní hodnoty této operace. Dále obsahuje dva jednobitové klopné obvody, jeden slouží k přenosu mezi více slovy, druhý k potlačení zápisu do registru ALU. ALU jednotka pracuje s hodnotami příznaků, které získává ze stavového registru. Podrobněji lze ALU jednotku prozkoumat na obrázku 2.5, funkci klopného obvodu pro práci s přenosy na obrázku 2.6 a klopného obvodu pro řízení zápisu na obrázku 2.7.

PSW je registr, který obsahuje pole příznaků. Tyto příznaky získává z ALU jednotky na základě výsledku operace. Dále obsahuje osmibitový dekremen-

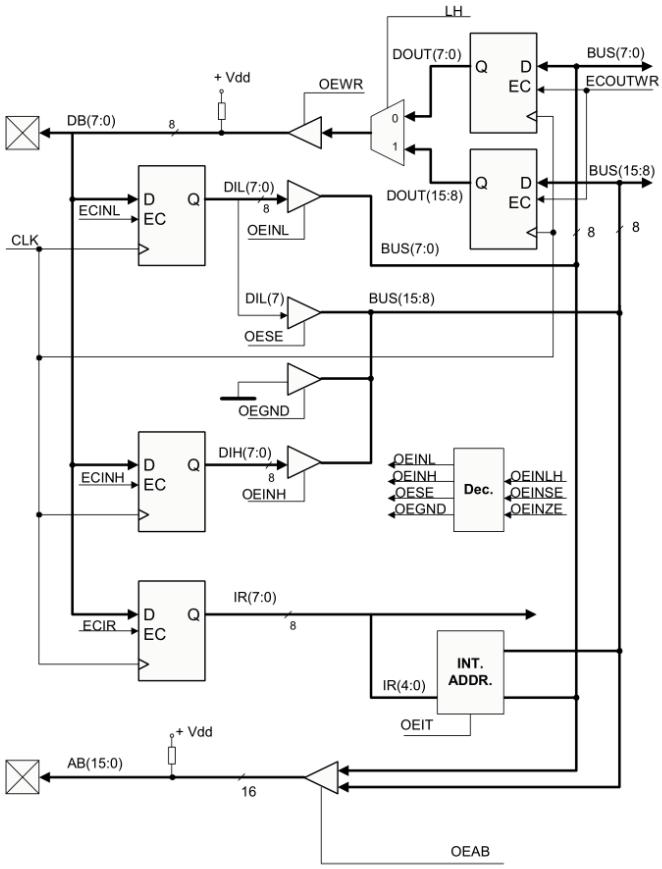
Obrázek 2.6: Klopný obvod pro práci s přenosy a jeho logika



Obrázek 2.7: Klopný obvod pro umožnění/zamezení zápisu a jeho logika



Obrázek 2.8: Paměťový řadič



tační registr L se stavovým signálem indikujícím nulovost registru. Tento je vhodný zejména pro implementaci cyklů.

Registry jsou návrhově jednoduchou záležitostí. Je však potřeba vyřešit cyklickou vazbu mezi ALU a PSW.

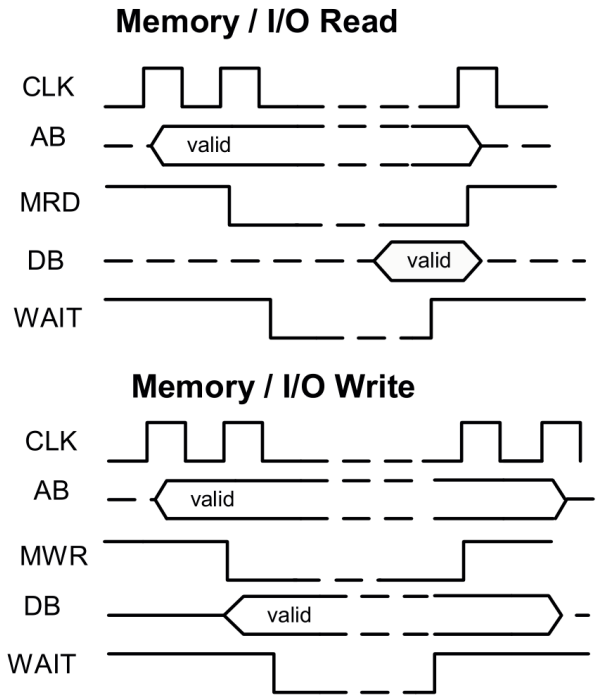
2.1.2.2 Sběrnice

Sběrnice je vodič napojený na třístavové budiče registrů. Ke kolizi by nemělo dojít vzhledem k tomu, že zápis z více registrů je zakázán při překladu, viz sekce 2.2.

2.1.2.3 Řadič paměti

Paměťový řadič je soustava registrů, které propojují interní sběrnici s datovou a adresní sběrnicí paměti. Podoba řadiče je znázorněna na obrázku 2.8

Obrázek 2.9: Grafické znázornění handshake protokolu pro komunikaci s pamětí



2.1.3 Paměť

Paměť je implementačně relativně jednoduchou záležitostí. Délka slova v paměti je 8 bitů.

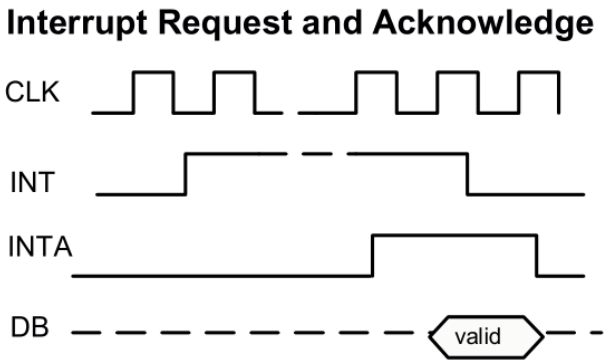
Pro komunikaci se využívá jednoduchého handshake protokolu, kdy je třeba s předstihem jednoho taktu umístit požadovanou adresu do adresní sběrnice, dále přivést aktivní hodnotu na požadovaný řídící signál (MRD pro čtení respektive MWR pro zápis) a následně čekat na aktivní signál WAIT. V případě zápisu je třeba ještě ponechat adresu na adresní sběrnici po jeden další takt. Pro lepší porozumění protokolu slouží obrázek 2.9.

Jediným problémem zde bude alespoň částečně odsimulovat asynchronní přístup a detekovat chyby protokolu.

2.1.4 Sběrnice

Adresní a datová sběrnice pro přístup do paměti jsou v podstatě shodné s interní sběrnicí (2.1.2.2). Jediným rozdílem je šířka datové sběrnice, která je (vzhledem k délce slova paměti) pouze osmibitová.

Obrázek 2.10: Grafické znázornění protokolu pro komunikaci s řadičem přerušení



2.1.5 Řadič přerušení

Řadič přerušení komunikuje s procesorem podobným protokolem jako paměť. Procesor je o přerušení informován signálem INT, signálem INTA potvrzuje přečtení signálu a na datovou sběrnici se poté načte číslo přerušení. Podrobný průběh je vidět na obrázku 2.10.

2.2 Překladač

Překladač bude sloužit k překladu jazyka symbolických mikroinstrukcí do binární podoby firmwaru procesoru DOP respektive do mezistupně určeného pro potřeby simulátoru k zobrazování čitelnější podoby firmwaru.

2.2.1 Jazyk symbolických mikroinstrukcí

Jazyk symbolických mikroinstrukcí slouží k usnadnění psaní mikroprogramu pro procesor DOP. Syntaxe jazyka vychází ze specifikace [4].

Pro potřeby vytvoření překladače je nezbytné analyzovat syntaxi jazyka a vytvořit jemu odpovídající gramatiku.

2.2.1.1 Analýza jazyka

Při analýze specifikace jazyka jsem objevil několik nejasností a nepřesností, kterým je potřeba věnovat pozornost. Nyní tedy popíšu hierarchickou strukturu mikrokódu danou specifikací a případné nejasnosti rozeberu.

Mikroprogram
Popis mikroprogramu ve specifikaci:

```
Mikroprogram ::= Blok1 Blok2 ... BlokN EOF
EOF ::= konec souboru
```

Mikroprogram se skládá z posloupností bloků zakončených koncem souboru. Zde ve specifikaci chybí informace o tom, čím mohou být bloky odděleny.

Na základě pokusů se simulátorem SIMDOP jsem dospěl k tomu, že bloky mohou být odděleny neomezeným počtem bílých znaků včetně nových řádků a celořádkových komentářů.

Blok

Popis bloku ve specifikaci:

```
Blok ::= <Návěstí bloku:> Mikroinstrukce

nebo

Blok ::= <Návěstí bloku:> { CR
                  Mikroinstrukce 0
                  Mikroinstrukce 1
                  ...
                  Mikroinstrukce k-1
} CR
Kde k=0..7
```

Každý blok tedy může obsahovat jednu mikroinstrukci, nebo více mikroinstrukcí umístěných ve složených závorkách.

Opět zde chybí informace, čím mohou být odděleny jednotlivé části bloku. V tomto případě se jedná o neomezený počet bílých znaků včetně nových řádků a celořádkových komentářů.

Ještě bych upozornil na překlep, kde poslední mikroinstrukce by měla mít číslo *k*, nikoli *k-1*.

Mikroinstrukce

Popis mikroinstrukce ve specifikaci:

```
Mikroinstrukce ::=
<Seznam akt. Signálů>,<kód operace ALU>, <typ podmínky>
   <,návěstí následujícího bloku> ; <komentář> CR

< ... >  nepovinné položky
CR ::= konec řádku
```

Opět chybí informace, čím mohou být odděleny jednotlivé části mikroinstrukce. Jedná se o bílé znaky vyjma nového řádku.

Také zde nesedí použití čárek mezi jednotlivými částmi instrukce. Tyto čárky jsou (dle příkladů na konci specifikace a fungování simulátoru SIMDOP)

všechny povinné pouze v případě, že následuje další část mikroinstrukce. Pouze v případě nezadání žádné části mikroinstrukce je třeba uvést před koncovým středníkem nejméně jednu čárku pro odlišení od celořádkového komentáře. Dokonce jsem odhalil, že o platnou mikroinstrukci se jedná i v případě, že za návěstím přímo následuje středník. Syntax zápisu mikroinstrukce by pak vypadala například takto:

```
Mikroinstrukce ::=
<Seznam akt. Signálů><,<kód operace ALU><,<typ podmínky
    ><,<návěstí následujícího bloku>>>> ; <komentář> CR
```

Návěstí bloku
Umožňuje odkázat se na daný blok. Návěstí je case sensitive.

Ve specifikaci není uvedeno, jaké znaky může návěstí obsahovat. Simulátor SIMDOP umožňuje, aby návěstí obsahovala velká a malá písmena bez diakritiky a čísla. Teoreticky by však nebyl problém, aby obsahovala i jakékoli jiné tisknutelné znaky, které nemají v jazyce symbolických mikroinstrukcí speciální funkci.

Seznam aktivních signálů
Právě signály uvedené v tomto seznamu jsou aktivní. Seznam všech řídících signálů lze nalézt například v blokových schématech procesoru DOP [3].

Dle specifikace mohou být signály odděleny pouze mezerami, simulátor SIMDOP však umožňuje i oddělení tabulátory.

Specifikace dále vymezuje signály, které nesmějí být současně aktivní kvůli kolizím na sběrnicích. Toto musí být nezbytně kontrolováno překladačem.

Kód operace ALU
Číslo rozsahu 0–15 určující operaci aritmetickologické jednotky.

Typ podmínky
Číslo v rozsahu 0–27 určující nastavení multiplexoru podmínky. Multiplexor poté určuje posledních 0–3 bitů následující adresy v závislosti na typu podmínky (takzvané 3bitové, 2bitové, 1bitové a 0bitové skoky).

Dle specifikace by délka bloku cílové adresy měla odpovídat typu skoku, tedy *m-bitovému* skoku by měl následovat blok o k instrukcích, kde $m = \log_2 k$. Tento požadavek však nesplňuje ani simulátor SIMDOP, který pouze zahlásí chybu, pokud by došlo k přepsání jiné mikroinstrukce nevhodně velkým blokem, ani vzorový mikroprogram, který používá přímé skoky pro přístup k různě velkým blokům.

Návěstí následujícího bloku
Určuje adresu následujícího bloku, ze které je na základě typu podmínky vybrána mikroinstrukce prováděná v následujícím taktu.

Zde je ve specifikaci nesprávně formulováno, že v případě, že cílové návěstí není uvedeno, předpokládá se provádění bezprostředně následujícího bloku. Z příkladů uvedených níže ve specifikaci i z fungování simulátoru SIMDOP však vyplývá, že dojde k vykonání bezprostředně následující mikroinstrukce. Z tohoto důvodu je ve specifikaci uvedeno doporučení, aby uvnitř jednoho bloku měly všechny mikroinstrukce návěstí následujícího bloku uvedené.

2.2.1.2 Gramatika jazyka

Na základě analýzy syntaxe jazyka symbolických mikroinstrukcí jsem vytvořil jeho gramatiku. Tato gramatika nepočítá s komentáři a přebytečnými bílými znaky, které je zpravidla vhodné odebrat v dřívější fázi překladu.

Terminály
NUM (číslo), ID (identifikátor), COL (:), LBR ({), RBR (}), SEM (;), COM (,), CR (nový řádek)

Neterminály
Prog (mikroprogram), Block (blok), InstSet (množina mikroinstrukcí), Inst (mikroinstrukce), Label (návěstí), Sig (seznam signálů), Alu (číslo operace ALU), Cond (číslo podmínky), Dest (cílové návěstí)

Počáteční symbol
Prog

Odvozovací pravidla

```
Prog -> Block Prog | e
Block -> Label Inst |
         Label { CR InstSet } CR
Label -> ID COL | e
InstSet -> Inst InstSet | e
Inst -> Sig SEM CR |
        Sig COM Alu SEM CR |
        Sig COM Alu COM Cond SEM CR |
        Sig COM Alu COM Cond COM Dest SEM CR
Sig -> ID Sig | e
Alu -> NUM | e
Cond -> NUM | e
Dest -> ID | e
```

2.2.2 Umístění mikroinstrukcí do řídící paměti

Vzhledem k tomu, že každá mikroinstrukce obsahuje adresu následující mikroinstrukce, není třeba, aby v řídící paměti následovaly mikroinstrukce ve stejném pořadí jako v jazyce symbolických mikroinstrukcí. Na první místo v paměti však musí být přiřazena první mikroinstrukce.

Dále je třeba dodržet umístění bloků mikroinstrukcí pro správné větvení mikroprogramu pomocí rozskoků. Jak bylo zmíněno v sekci 2.2.1.1, poslední až tři bity adresy následující mikroinstrukce mohou být upraveny pomocí multiplexoru podmínky. Za účelem snadné implementace tohoto větvení se mikroinstrukce zapisují do různě dlouhých bloků. v řídící paměti pak tyto mikroinstrukce ze zjevných důvodů musí být umístěny za sebou a to počínaje adresou dělitelnou velikostí bloku.

Po rozmístění mikroinstrukcí do řídící paměti je potřeba nahradit symbolická návěstí konkrétními adresami mikroinstrukcí.

2.3 Uživatelské rozhraní

V uživatelském rozhraní simulátoru bude třeba navrhnout tyto součásti: textový editor, seznam návěstí ve zdrojovém souboru mikroprogramu, ovladač vnitřních hodnot procesoru a paměti, online diagram aritmeticko-logické jednotky, online diagram adresního multiplexoru, zobrazovač výstupů překladače, zobrazovač obsahu řídící paměti v čitelné podobě. Dále bude potřeba vytvořit hlavní menu a stavový řádek.

Jednotlivé prvky uživatelského rozhraní by měly být snadno pozicovatelné, aby vyhovovaly individuálním požadavkům uživatelů. S důrazem na modularitu bude tedy potřeba zvolit vhodné rozměry jednotlivých prvků a vybrat vhodné vývojové prostředky.

2.3.1 Textový editor

Textový editor bude sloužit k zobrazování a úpravě zdrojových souborů mikrokódu v jazyce symbolických mikroinstrukcí a také k zobrazování aktuálně zpracovávané mikroinstrukce pro názornější simulaci.

Textový editor by měl mít tyto vlastnosti:

- možnost vytvářet, otevírat a ukládat soubory
- standardní možnosti editace souboru
- zobrazování čísla řádek včetně aktuální pozice kurzoru
- zvýrazňování syntaxe jazyka symbolických mikroinstrukcí
- automatické doplňování slov (návěstí a signály)

- přidávání, odebírání a zvýrazňování breakpointů
- pokročilé vyhledávání a nahrazování v textu
- zvýrazňování aktuálně zpracovávané mikroinstrukce
- navigace na definici vybraného návěstí
- detekce vnější úpravy souboru pro možnost využití externího editoru

Textový editor by měl v uživatelském rozhraní vyplnit největší plochu. Měl by být ústředním prvkem celého rozhraní.

2.3.2 Seznam návěstí

Seznam návěstí bude sloužit k zobrazování všech návěstí nalezených při překladu zdrojového kódu. Dále by měl umožňovat navigaci na definici daného návěstí v textovém editoru.

Seznam návěstí by měl být vertikálně orientovaný prvek uživatelského rozhraní.

2.3.3 Ovladač vnitřních hodnot procesoru

Ovladač vnitřních hodnot procesoru bude sloužit k zobrazování a úpravě hodnot registrů, sběrnic a stavových signálů procesoru. Pro optimální simulaci by měl umět zobrazit a editovat co nejvíce těchto hodnot.

Tento prvek se bude skládat z dvojic obsahujících popisek a řádkový editor hodnot. Tyto dvojice budou dále rozděleny do skupin podle umístění a funkce v procesoru. Řádkové editory needitovatelných prvků budou sloužit pouze k zobrazování.

Další užitečnou vlastností by mělo být zvýraznění hodnot, které byly změněny během posledního úkonu simulátoru.

Ovladač vnitřních hodnot bude přibližně čtvercových rozměrů.

2.3.4 Ovladač hlavní paměti procesoru

Ovladač paměti bude sloužit k zobrazování a úpravě hodnot v hlavní paměti procesoru. Bude to jednoduchá tabulka, kde v prvním sloupci budou needitovatelné adresy a v druhém sloupci editovatelné hodnoty odpovídající paměťové buňky.

Ovladač paměti bude vertikálně orientovaný prvek. Jednou z možností je i jeho integrace do ovladače vnitřních hodnot procesoru.

2.3.5 Online diagram aritmeticko-logické jednotky

Diagram ALU bude sloužit pro přehlednější zobrazení aktuálních hodnot v aritmeticko-logické jednotce. Bude vycházet z diagramu ALU na obrázku 2.5 a u každého signálu či sběrnice bude zobrazovat jeho aktuální hodnotu.

Pro lepší přehlednost by mohly být zvýrazněny hodnoty změněné během posledního úkonu simulátoru.

Diagram ALU bude prvek přibližně čtvercových rozměrů.

2.3.6 Online diagram adresního multiplexoru

Diagram adresního multiplexoru bude sloužit k názornému vyobrazení stavu adresního multiplexoru v aktuálním taktu. Diagram bude vyobrazovat, ze kterého signálu či registru vedou jednotlivé vodiče reprezentující adresu následující mikroinstrukce v řídící paměti.

Diagram multiplexoru bude prvek přibližně čtvercových rozměrů.

2.3.7 Zobrazovač výstupů překladače

Zobrazovač výstupů překladače bude sloužit k zobrazování informačních zpráv a průběhu překladu mikroprogramu. S jeho pomocí se budou vypisovat informační zprávy, varování a chyby.

Zobrazovač výstupů bude tabulka obsahující typ zprávy, text a případně další hodnoty, například číslo řádky.

Tento prvek bude vhodné umístit horizontálně.

2.3.8 Zobrazovač obsahu řídící paměti

Zobrazovač obsahu řídící paměti bude v čitelné podobě vypisovat celou řídící paměť procesoru. Bude obsahovat návěstí, adresu v řídící paměti, seznam aktivních signálů, operační znak aritmeticko-logické jednotky, kód podmínky a adresu následující mikroinstrukce. Dále by měl sloužit k navigaci na danou mikroinstrukci v textovém editoru.

Zobrazovač obsahu řídící paměti bude vhodné umístit horizontálně, ovšem s ohledem na jeho přehlednost přímo úměrnou jeho výšce.

2.3.9 Hlavní menu

Hlavní menu bude obsahovat několik podmenu. Těmi nejdůležitějšími jsou:

File Bude sloužit zejména pro otevírání a ukládání souborů a zavírání aplikace.

Edit Bude obsahovat akce zpět a vpřed, akce pro práci se schránkou, vyhledávání a nahrazování a přechod na konkrétní řádek.

View Bude sloužit pro volení viditelných prvků uživatelského rozhraní.

Simulation Bude obsahovat akce, s jejichž pomocí bude možné spouštět samotnou simulaci v různě dlouhých krocích či manipulovat s breakpointy.

Core Bude obsahovat možnost resetu, načtení či uložení vnitřního stavu procesoru.

Memory Bude obsahovat možnost resetu, načtení či uložení obsahu paměti.

Help Bude sloužit k zobrazování nápovědy a dalších informací o programu.

2.3.10 Stavový řádek

Stavový řádek bude vhodný k zobrazování pozice kurzoru textového editoru či informace o poslední vykonané akci.

Kapitola 3

Návrh

3.1 Výběr vývojové platformy

Při výběru vývojové platformy jsem se řídil zejména s ohledem na nenáročnost výsledné aplikace, multiplatformnost a v neposlední řadě také na své zkušenosti s daným prostředím.

Potřeba jednoduchosti a nenáročnosti aplikace vyloučila interpretované jazyky Java a C#. Ty jsou sice přirozeně multiplatformní, ovšem vyžadují pro svůj běh runtime rozhraní a jsou řádově náročnější než nativní aplikace.

Rozhodl jsem se tedy využít jazyka C++, který umožňuje psát velmi výkonné aplikace, ale zároveň je také pro mě osobně velice pohodlný na psaní. Navíc je velmi rozšířený a existuje pro něj několik knihoven určených pro psaní multiplatformních grafických aplikací.

Dalším krokem tedy bylo zvolit vhodnou knihovnu pro grafické rozhraní. Po krátkém průzkumu možností jsem se rozhodl pro framework Qt, který nabízí velmi dobrou dokumentaci, rozsáhlé funkce nejen pro grafické rozhraní a také multiplatformní práci s datovými typy, které budou velmi užitečné pro implementaci vnitřní struktury procesoru DOP.

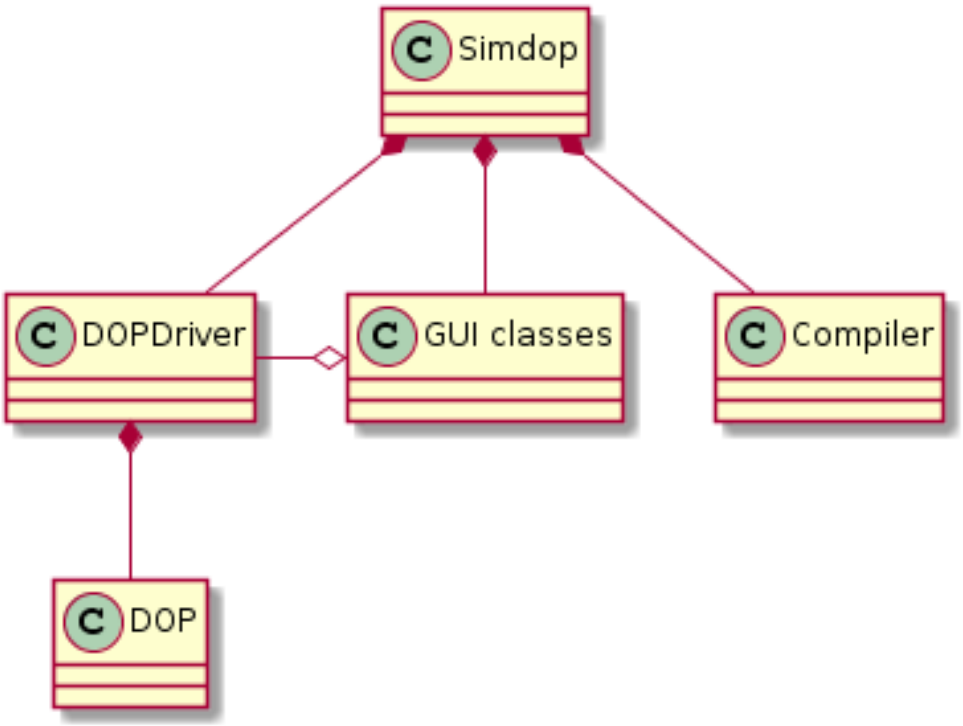
3.1.1 Mechanismy knihovny Qt

V implementaci uživatelského rozhraní bude využito některých specifických mechanismů knihovny Qt. Jde zejména o signály a sloty, které slouží pro zjednodušení komunikace mezi jednotlivými objekty, zejména pak mezi prvky uživatelského rozhraní, které jsou v Qt označovány jako widgety.

Každý objekt, který je potomkem třídy *QObject*, může využít speciálního makra *Q_OBJECT*. Díky tomuto makru je takovému objektu vygenerován ještě takzvaný metaobjekt, který umožňuje využívat zmíněné speciální mechanismy Qt, jako jsou signály a sloty.

Signály bych připodobnil k ukazatelům na metody a sloty jsou speciální metody, které umožňují být připojeny na signály pomocí metody *connect* třídy

Obrázek 3.1: Diagram struktury aplikace



QObject. Pokud je pak vyvolán signál, jsou postupně zavolány všechny sloty na daný signál připojené.

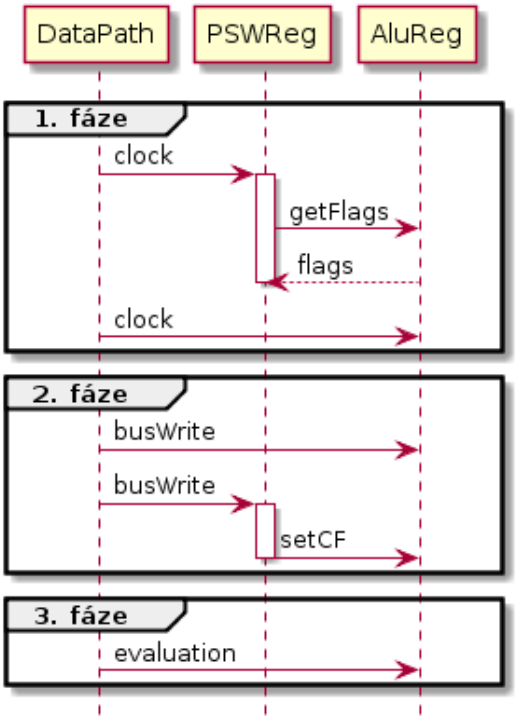
Díky tomuto mechanismu pak lze snadno implementovat komunikaci mezi jednotlivými prvky uživatelského rozhraní, aniž by musely být příliš návrhově provázány a celý návrh je tím zjednodušen stejně jako samotná implementace.

3.2 Struktura

Nejvyšším objektem aplikace bude objekt třídy *Simdop*, která bude potomkem třídy *QMainWindow* reprezentující v Qt hlavní okno aplikace. Tato třída bude řídit hlavní programovou smyčku a bude obsluhovat a propojovat ostatní komponenty.

Třída *Simdop* bude obsahovat objekty tříd jednotlivých částí uživatelského rozhraní. Dále bude obsahovat objekt třídy *Compiler*, tedy překladač jazyka symbolických mikroinstrukcí, a *DOPDriver*, který bude sloužit k ovládání objektu třídy *DOP*, tedy třídy implementující samotný procesor. Struktura aplikace je znázorněna na obrázku 3.1.

Obrázek 3.2: Sekvenční diagram spolupráce aritmeticko-logické jednotky a stavového registru



3.3 Procesor

Procesor bude navržen tak, aby jeho struktura co nejpřesněji odpovídala hardwarové implementaci. Díky tomu bude snadněji implementován, předejde se tím funkčním chybám a kód bude lépe čitelný.

Vykonávání mikroinstrukcí bude rozděleno na tři fáze. První fáze bude reprezentovat sekvenční logiku. Zde dojde k zapsání vstupů registrů na výstupy. Ve druhé fázi dojde k vystavení hodnot na sběrnici. Ve třetí fázi dojde k výpočtu nových hodnot všech ostatních signálů.

Díky tomuto rozdělení taktů na jednotlivé fáze lze zajistit správnou simulaci paralelně šířených signálů v sekvenčním prostředí simulátoru.

Křížovou závislost aritmeticko-logické jednotky a stavového registru lze nyní vyřešit tak, že ve fázi zápisu na sběrnici předá stavový registr hodnotu příznaku přenosu (CF) aritmeticko-logické jednotce, ve fázi vyhodnocení dojde k výpočtu a ve fázi zápisu do registru získává stavový registr výstupní hodnoty aritmeticko-logické jednotky. Navíc to umožňuje roztržení návrhově nekorektní křížové vazby mezi dvěma objekty, kdy stačí stavovému registru poskytnout referenci aritmeticko-logické jednotky a již ne naopak. Názorně je průběh výměny dat zobrazen na obrázku 3.2.

Parametry metod tříd reprezentujících konkrétní část procesoru, které vykonávají jednotlivé fáze taktu, budou řídící signály, které v dané fázi danou část procesoru ovládají.

Jako datový typ pro ukládání šestnáctibitových respektive osmibitových hodnot v procesoru jsem zvolil datový typ *quint16* a *quint8*, což je šestnáctibitový respektive osmibitový bezznaménkový datový typ frameworku Qt, který garantuje správnou funkčnost na všech podporovaných platformách. Jednobitové signály budou datového typu *bool*.

3.3.1 Struktura

Struktura procesoru je vyobrazena na diagramu tříd na obrázku 3.3.

3.3.1.1 Třída DOP

Třída *DOP* bude hlavní třídou celého procesoru. Bude implementovat funkci řadiče procesoru, pomocí kterého bude ovládat další objekty, které bude obsahovat. Těmi budou objekty tříd *Memory* (hlavní paměť), *DataPath* (datová cesta), *IntCtrl* (řadič přerušení), *Bus<quint16>* (adresní sběrnice) a *Bus<quint8>* (datová sběrnice).

3.3.1.2 Třída DataPath

Třída *DataPath* bude reprezentovat datovou cestu procesoru. Bude obsahovat objekty tříd *Bus<quint16>* (vnitřní sběrnice), *MemCtrl* (řadič paměti), *BaseReg* (registry T a U), *IncReg* (registry PC a S), *IncDecReg* (registry SP a D), *AluReg* (aritmeticko-logická jednotka) a *PSWReg* (stavový registr). Dále bude obsahovat reference na *Bus<quint16>* (adresní sběrnice) a *Bus<quint8>* (datová sběrnice)

3.3.1.3 Třída Memory

Třída *Memory* bude reprezentovat hlavní paměť procesoru. Bude obsahovat referenci na *Bus<quint16>* (adresní sběrnice) a *Bus<quint8>* (datová sběrnice).

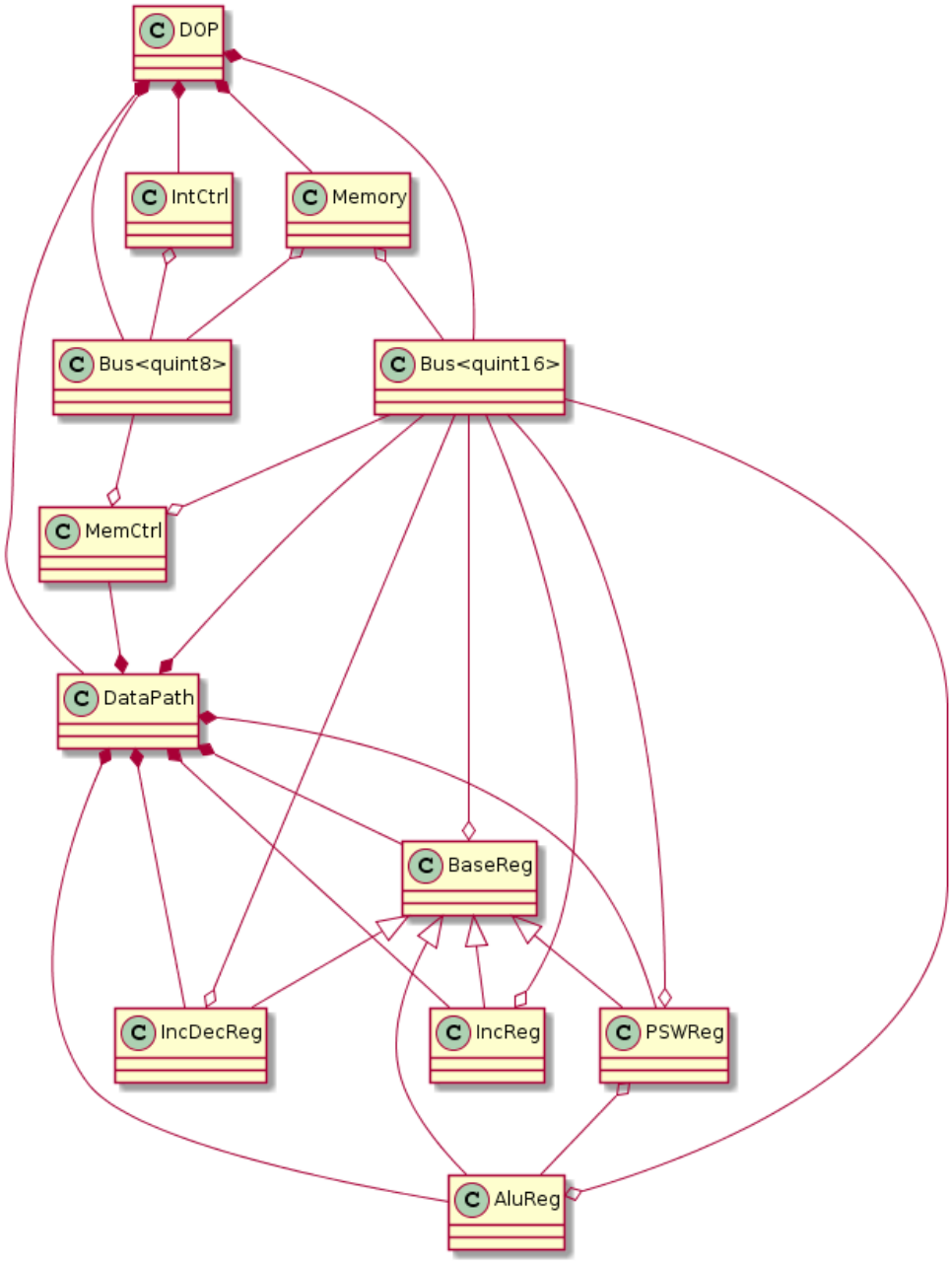
3.3.1.4 Šablona Bus

Šablona *Bus* bude reprezentovat sběrnici variabilní šířky. Pro potřeby simulátoru využiji třídu *Bus<quint8>* a *Bus<quint16>*.

3.3.1.5 Třída MemCtrl

Třída *MemCtrl* bude reprezentovat paměťový řadič procesoru. Bude obsahovat reference na *Bus<quint16>* (adresní a interní sběrnice) a *Bus<quint8>* (datová sběrnice).

Obrázek 3.3: Diagram tříd procesoru



3.3.1.6 Třída BaseReg

Třída *BaseReg* bude reprezentovat základní registr procesoru. Bude obsahovat referenci na *Bus<quint16>* (interní sběrnice).

3.3.1.7 Třída IncReg

Třída *IncReg* bude potomkem třídy *BaseReg* a bude reprezentovat registr umožňující inkrementaci.

3.3.1.8 Třída IncDecReg

Třída *IncDecReg* bude potomkem třídy *BaseReg* a bude reprezentovat registr umožňující inkrementaci a dekrementaci.

3.3.1.9 Třída AluReg

Třída *AluReg* bude potomkem třídy *BaseReg* a bude reprezentovat aritmeticko-logickou jednotku.

3.3.1.10 Třída PSWReg

Třída *PSWReg* bude potomkem třídy *BaseReg* a bude reprezentovat stavový registr obsahující osmibitový registr L a registr příznaků. Bude obsahovat referenci na *AluReg*.

3.3.1.11 Třída IntCtrl

Třída *IntCtrl* bude reprezentovat řadič přerušení. Bude obsahovat referenci na *Bus<quint8>* (datová sběrnice).

3.4 Překladač

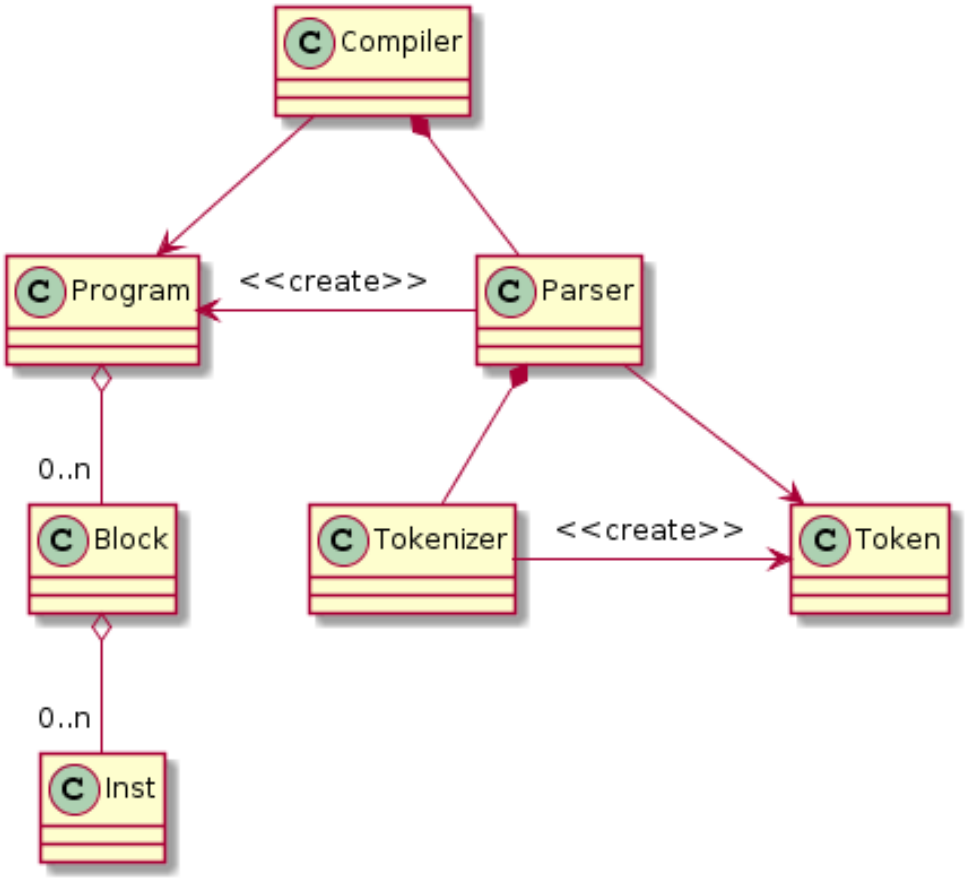
Překladač bude soubor tříd umožňující překlad jazyka symbolických mikroinstrukcí do obsahu řídící paměti procesoru DOP a do podoby vhodné k čitelnému zobrazení tohoto obsahu.

Překladač se bude skládat ze tří hlavních tříd, *Compiler*, *Parser* a *Tokenizer*. Dále bude využívat pomocné struktury pro reprezentaci jednotlivých částí mikrokódu, *Program*, *Block*, *Inst*, strukturu reprezentující token jazyka symbolických mikroinstrukcí, *Token*, a strukturu reprezentující zprávu z průběhu překladu, *CompileMessage*.

3.4.1 Struktura

Struktura překladače je vyobrazena na diagramu tříd na obrázku 3.4.

Obrázek 3.4: Diagram tříd překladače



3.4.1.1 Třída Compiler

Třída *Compiler* bude hlavní třídou překladače. Jejím úkolem bude přeložit soubor symbolických mikroinstrukcí a poskytovat výstupy z tohoto překladu. Bude obsahovat objekt třídy *Parser*.

3.4.1.2 Třída Parser

Třída *Parser* bude sloužit k parsování vstupního souboru jazyka symbolických mikroinstrukcí do objektu typu *Program* a ke kontrole jeho syntaktické správnosti. Bude obsahovat objekt třídy *Tokenizer*.

3.4.1.3 Třída Tokenizer

Třída *Tokenizer* bude sloužit k rozložení zdrojového souboru na objekty typu *Token*.

3.4.1.4 Struktura Token

Struktura *Token* bude reprezentovat jeden token ve zdrojovém souboru jazyka symbolických mikroinstrukcí. Bude sloužit také k vytvoření textové reprezentace tohoto tokenu za účelem jeho čitelného výpisu.

3.4.1.5 Struktura Program

Struktura *Program* bude potomkem třídy *QList<Block*>* a bude reprezentovat jazyk symbolických mikroinstrukcí v programově čitelné podobě.

3.4.1.6 Struktura Block

Struktura *Block* bude potomkem třídy *QList<Inst*>* a bude reprezentovat blok mikroinstrukcí pro využití ve struktuře *Program*.

3.4.1.7 Struktura Inst

Struktura *Inst* bude reprezentovat jednu mikroinstrukci pro účely parsování, překladu a výpisu.

3.4.1.8 Struktura CompileMessage

Struktura *CompileMessage* bude reprezentovat zprávu z průběhu překladu.

3.5 Ovladač procesoru

Ovladač procesoru bude sloužit jako rozhraní mezi procesorem (třídou *DOP*) a uživatelským rozhraním. Jeho hlavními funkcemi bude ovládání simulace procesoru a umožnění přístupu k vnitřním hodnotám procesoru.

3.5.1 Struktura

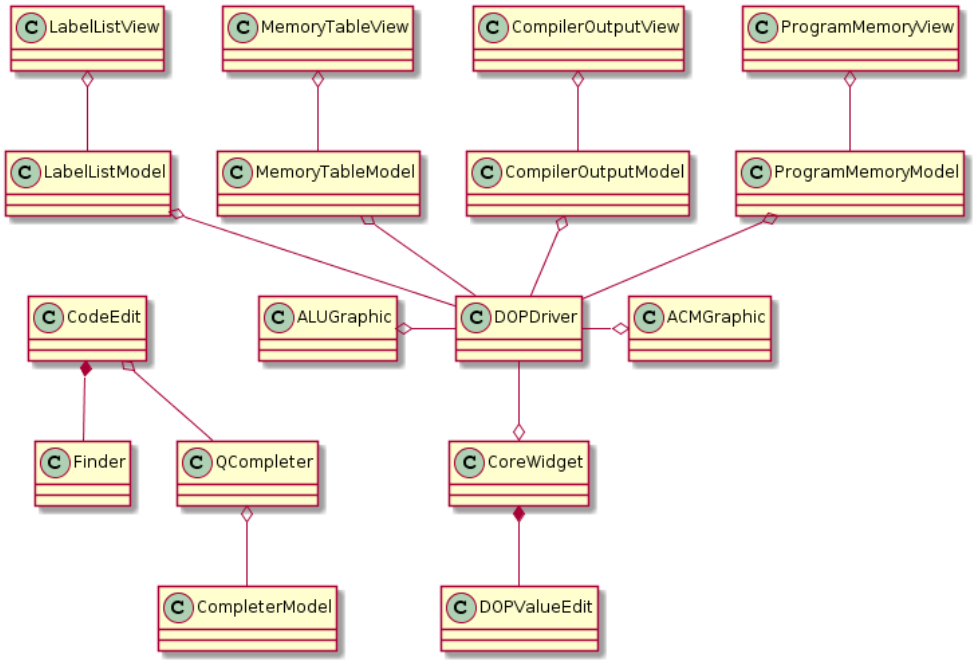
Ovladač procesoru bude reprezentován jedinou třídou *DOPDriver*. Pro snadné předávání informací o dostupných hodnotách procesoru bude sloužit struktura *DOPValueInfo*.

3.5.1.1 Třída DOPDriver

Třída *DOPDriver* bude potomkem třídy *QObject* a bude vyžívat speciálního makra *Q_OBJECT*, díky čemuž jí bude vytvořen takzvaný metaobjekt. Bližší informace lze nalézt v sekci 3.1.1 nebo v dokumentaci knihovny Qt [5].

Třída bude obsahovat objekt *DOP* a umožňovat spouštění simulace, zastavování a restartování simulace a bude poskytovat signály informující o vnitřních změnách procesoru, aby na ně mohly reagovat prvky uživatelského rozhraní.

Obrázek 3.5: Diagram tříd uživatelského rozhraní



3.5.1.2 Struktura DOPValueInfo

Struktura *DOPValueInfo* bude sloužit pro popis vlastností vnitřních hodnot procesoru přístupných pomocí třídy *DOPDriver*.

3.6 Uživatelské rozhraní

Uživatelské rozhraní bude navrženo tak, aby odpovídalo model/view architektuře, kterou využívá knihovna Qt.

3.6.1 Struktura

Struktura uživatelského rozhraní je vyobrazena na diagramu tříd na obrázku 3.5. Základem bude třída *Simdop* reprezentující hlavní okno aplikace. v něm lze zvolit takzvaný centrální widget, který zabírá maximální možnou plochu okna, a vytvořit takzvané dokovací widgety, které se dají libovolně rozmístit okolo centrálního widgetu po stranách okna. Dále bude hlavní okno obsahovat hlavní menu aplikace a stavový řádek.

Centrálním widgetem aplikace bude textový editor reprezentovaný třídou *CodeEdit*. Ostatní prvky uživatelského rozhraní budou umístěny jako dokovací widgety. Tímto dosáhnu vysoké modularity a flexibility rozhraní. Konkrétní rozvržení uživatelského rozhraní tedy bude záležet na uživateli.

3.6.1.1 Třída Simdop

Třída *Simdop* bude potomkem třídy *QMainWindow*. Třída nebude implementovat žádné veřejné metody ani sloty, její komunikace s okolím bude probíhat pomocí akcí (objektů třídy *QAction*), které jsou navázány na položky v hlavním menu aplikace.

3.6.1.2 Třída CodeEdit

Třída *CodeEdit* bude potomkem třídy *QPlainTextEdit*, což je widget základního textového editoru. V něm budou implementované funkce jako zobrazení číslování řádek, zvýrazňování aktuální mikroinstrukce, zvýrazňování syntaxe (*Highlighter*), automatické doplňování textu (*QCompleter*) či pokročilé vyhledávání (*Finder*).

3.6.1.3 Třída CompleterModel

Třída *CompleterModel* bude potomkem třídy *QAbstractListModel* a bude zprostředkovávat seznam nabízených slov pro automatické doplňování. Těmi bude seznam všech návěstí a řídících signálů.

3.6.1.4 Třída Finder

Třída *Finder* bude představovat widget pro obsluhu vyhledávání a nahrazování v textovém editoru.

3.6.1.5 Třída Highlighter

Třída *Highlighter* bude potomkem třídy *QSyntaxHighlighter* a bude sloužit ke zvýrazňování syntaxe jazyka symbolických mikroinstrukcí v textovém editoru.

3.6.1.6 Třída CoreWidget

Třída *CoreWidget* bude zobrazovat objekty třídy *DOPValueEdit* rozdělené do několika skupin.

3.6.1.7 Třída DOPValueEdit

Třída *DOPValueEdit* bude potomkem třídy *QLineEdit* a bude sloužit pro zobrazení a úpravu jedné konkrétní hodnoty procesoru na základě objektu třídy *DOPValueInfo* (viz sekce 3.5.1.2).

3.6.1.8 Třída LabelListView

Třída *LabelListView* bude sloužit k zobrazování seznamu návěstí v aktuálním souboru jazyka symbolických mikroinstrukcí. Dvojklik na návěstí způsobí

přesunutí kurzoru textového editoru na definici daného návěstí. Jako datový model jí bude sloužit třída *LabelListModel*.

3.6.1.9 Třída LabelListModel

Třída *LabelListModel* bude sloužit ke zprostředkování seznamu návěstí a jim odpovídajících čísel řádek.

3.6.1.10 Třída MemoryTableView

Třída *MemoryTableView* bude sloužit k zobrazování a úpravě obsahu hlavní paměti procesoru. Jako datový model jí bude sloužit třída *MemoryTableModel*.

3.6.1.11 Třída MemoryTableModel

Třída *MemoryTableModel* bude sloužit ke zprostředkování obsahu hlavní paměti procesoru.

3.6.1.12 Třída CompilerOutputView

Třída *CompilerOutputView* bude sloužit k zobrazování zpráv překladu. Jako datový model jí bude sloužit třída *CompilerOutputModel*.

3.6.1.13 Třída CompilerOutputModel

Třída *CompilerOutputModel* bude sloužit ke zprostředkování seznamu zpráv překladu.

3.6.1.14 Třída ProgramMemoryView

Třída *ProgramMemoryView* bude sloužit k zobrazování obsahu řídící paměti procesoru. Jako datový model jí bude sloužit třída *ProgramMemoryModel*.

3.6.1.15 Třída ProgramMemoryModel

Třída *ProgramMemoryModel* bude sloužit ke zprostředkování obsahu řídící paměti procesoru.

3.6.1.16 Třída ALUGraphic

Třída *ALUGraphic* bude sloužit k zobrazování diagramu aritmeticko-logické jednotky s aktuálními hodnotami.

3.6.1.17 Třída ACMGraphic

Třída *ACMGraphic* bude sloužit k zobrazování diagramu multiplexoru adresy následující mikroinstrukce s aktuálními hodnotami.

Kapitola 4

Realizace

Realizace probíhala v prostředí programu Microsoft Visual Studio 2013 s rozšířením Qt Add-in 1.2.3. Následně byl projekt převeden do programu Qt Creator a kompilován prostřednictvím qmake a nmake (Windows) respektive make (Linux). Použita byla staticky kompilovaná knihovna Qt ve verzi 5.4.

4.1 Procesor

Procesor byl realizován za minimálního využití knihovny Qt pro případné znovupoužití v jiných projektech. Výjimkami jsou pouze základní datové typy *quint8*, *quint16*, *quint32* a *quint64* s multiplatformně garantovanou velikostí.

4.1.1 Třída DOP

Třída *DOP* integruje řadič procesoru jako pole o 512 prvcích typu *quint64* a uchovává adresu aktuální a následující mikroinstrukce. Obsahuje objekty reprezentující další vnitřní součásti (viz sekce 3.3.1.1).

Ovládání procesoru probíhá prostřednictvím metody *clock*, která reprezentuje jeden takt procesoru, metody *reset*, která slouží k uvedení do výchozího stavu, a getterů a setterů všech vnitřních hodnot procesoru.

4.1.1.1 Metoda clock

Metoda *clock* je hlavním prvkem řídícím chod procesoru a implementuje všechny tři fáze taktu (viz sekce 3.3). Její průběh je následující:

- přivedení hodin do sekvenční logiky - volání metody *clock* na objekt třídy *DataPath* (první fáze taktu)
- přepsání adresy aktuální mikroinstrukce (skok v novou adresu)
- resetování sběrnic

- zapsání dat na sběrnice - volání metody *busWrite* na objekt třídy *DataPath* (druhá fáze taktu)
- vyhodnocení kombinační logiky - volání metody *evaluation* na objekt třídy *DataPath* (třetí fáze taktu)
- paměťová operace - volání metody *clock* na objekt třídy *Memory*
- vypočítání adresy následující mikroinstrukce

Všechny metody objektů uvnitř procesoru mají v parametrech všechny řídící signály, které se daného objektu v dané fázi taktu týkají.

4.1.2 Třída Memory

Třída *Memory* implementuje paměť jako pole prvků typu *quint8*.

Pro sledování správné funkčnosti komunikačního protolu uchovává hodnotu adresní sběrnice v předchozím taktu a uchovává hodnotu výčtového typu reprezentující chybu v protokolu. Ta může nabývat hodnoty pro bezchybnost, změnu adresy během paměťové operace, soušasnou aktivitu signálů MRD a MWR, či předčasnou deaktivaci jednoho z těchto signálů.

Pro ovládání paměti slouží metoda *clock*, metoda *reset*, getter a setter obsahu paměti, getter signálu WAIT a getter chyby v protokolu.

4.1.2.1 Metoda clock

Metoda *clock* je implementována stavovým automatem, jehož vstupy jsou rovnost předchozí a aktuální hodnoty na adresní sběrnici a signály MRD a MWR. Výstupy jsou signál WAIT a hodnota chyby protokolu. Pro detailní podobu automatu doporučuji nahlédnout do zdrojového kódu (soubor *dopmemory.cpp*).

4.1.3 Třída DataPath

Třída *DataPath* implementuje metody pro jednotlivé fáze taktu (viz 3.3). První fázi implementuje metoda *clock*, druhou fázi metoda *busWrite* a třetí fázi metoda *evaluation*. Dále implementuje gettery a settery všech hodnot, které implementují v ní obsažené objekty.

4.1.3.1 Metoda clock

Metoda *clock* představuje první fázi hodinového taktu na datové cestě. Nejprve dojde k zápisu ve stavovém registru, poté v aritmeticko-logické jednotce, následně v ostatních registrech a na závěr v paměťovém řadiči (ve všech případech volání metody *clock* na daném objektu).

4.1.3.2 Metoda busWrite

Metoda *busWrite* představuje druhou fázi hodinového taktu na datové cestě. Nejprve je vyresetována vnitřní sběrnice metodou *reset*, následně zavolána metoda *busWrite* všech registrů a nakonec metoda *busWrite* paměťového řadiče.

4.1.3.3 Metoda evaluation

Metoda *evaluation* představuje třetí fázi hodinového taktu na datové cestě. Jejím jediným krokem je vyhodnocení kombinační logiky aritmeticko-logické jednotky metodou *evaluation*.

4.1.4 Šablona Bus

Šablona *Bus* obsahuje aktuální hodnotu a také příznak, zda bylo na sběrnici zapisováno. Ten slouží k ověření, že nedošlo k vícenásobnému zápisu na sběrnici a je třeba ho v každém taktu vynulovat.

Šablona implementuje getter a setter hodnoty a metodu *reset*, která ruší příznak zápisu na sběrnici.

4.1.5 Třída MemCtrl

Třída implementuje metodu *clock* reprezentující první fázi taktu a metodu *busWrite* reprezentující druhou fázi taktu (viz 3.3). Vzhledem k tomu, že řadič není kombinačně závislý na ostatních částech datové cesty jinak než přes sběrnice, třetí fázi taktu není potřeba implementovat.

Dále bude třída implementovat gettery a settery svých vnitřních registrů (IR, DOUT, DIL, DIH) a metodu *reset*, která vynuluje všechny vnitřní registry řadiče.

4.1.5.1 Metoda clock

Metoda *clock* umožňuje načítání hodnot ze sběrnic do vnitřních registrů.

4.1.5.2 Metoda busWrite

Metoda *busWrite* zapisuje hodnoty registrů na sběrnice, popřípadě (na závěr) hodnotu vnitřní sběrnice na adresní sběrnici.

4.1.6 Třída BaseReg

Třída implementuje metodu *clock* reprezentující první fázi taktu a metodu *busWrite* reprezentující druhou fázi taktu (viz 3.3). Vzhledem k tomu, že základní registr není kombinačně závislý na ostatních částech datové cesty jinak

než přes sběrnice, třetí fázi taktu není potřeba implementovat. Dále třída implementuje getter a setter hodnoty klopného obvodu a metodu *reset* nulující hodnotu klopného obvodu.

4.1.6.1 Metoda clock

Metoda *clock* umožňuje zapsat do klopného obvodu hodnotu na vstupu.

4.1.6.2 Metoda busWrite

Metoda *busWrite* umožňuje zápis hodnoty klopného obvodu na sběrnici.

4.1.7 Třída IncReg

Třída *IncReg* je potomkem třídy *BaseReg* a reimplementuje metodu *clock*.

4.1.7.1 Metoda clock

Metoda *clock* umožňuje inkrementaci hodnoty registru, nebo volání rodičovské metody *clock* pro zápis do klopného obvodu.

4.1.8 Třída IncDecReg

Třída *IncDecReg* je potomkem třídy *BaseReg* a reimplementuje metodu *clock*.

4.1.8.1 Metoda clock

Metoda *clock* umožňuje inkrementaci a dekrementaci hodnoty registru,nebo volání rodičovské metody *clock* pro zápis do klopného obvodu.

4.1.9 Třída AluReg

Třída *AluReg* je potomkem třídy *BaseReg* a reimplementuje metodu *clock*. Dále implementuje metodu *evaluation* reprezentující třetí fázi taktu (viz 3.3), gettery vnitřních signálů, settery a gettery hodnot pomocných klopných obvodů CZM a WEN a metodu *setCF*.

4.1.9.1 Metoda clock

Metoda *clock* ukládá výsledky aritmeticko-logické operace do hlavního klopného obvodu a pomocných klopných obvodů CZM a WEN při splnění podmínek vyplývajících z blokových schémat na obrázcích 2.6 a 2.7.

4.1.9.2 Metoda evaluation

Metoda *evaluation* slouží k samotnému vykonání operace aritmeticko-logické jednotky. Její průběh je následující:

- načtení operandů v závislosti na operačním znaku do proměnných typu *quint32* (pro možnost uložení 17. bitu s přenosem)
- vypočtení výsledku
- vyhodnocení přenosu, přetečení, znaménka a nulovosti výsledku (hodnoty CY, OVERF, SIGN a ZERO)
- oříznutí výsledku na typ *quint16*

4.1.9.3 Metoda setCF

Pomocí metody *setCF* může být aritmeticko-logické jednotce třídou stavového registru nastavena aktuální hodnota příznaku přenosu (CF). Podrobněji je způsob komunikace mezi aritmeticko-logickou jednotkou a stavovým registrem znázorněn na obrázku 3.2.

4.1.10 Třída PSWReg

Třída *PSWReg* je potomkem třídy *BaseReg* a reimplementuje metodu *clock* a *busWrite*. Dále implementuje gettery a settery jednotlivých příznaků a registru L a getter stavového signálu LZERO (L rovno 0).

4.1.10.1 Metoda clock

Metoda *clock* kromě uložení hodnoty celého PSW registru umožňuje také uložení pouze registru L, dekrementaci registru L, nebo uložení pouze příznaků na základě výsledku operace aritmeticko-logické jednotky.

4.1.10.2 Metoda busWrite

Metoda *busWrite* kromě zapsání hodnoty celého PSW registru na sběrnici umožňuje také zapsání pouze registru L, pouze příznaků, nebo rozšířeného auxiliary příznaku (AF). Poté předává pomocí metody *setCF* třídy *AluReg* aktuální hodnotu příznaku přenosu (CF) aritmeticko-logické jednotce. Podrobněji je způsob komunikace mezi aritmeticko-logickou jednotkou a stavovým registrem znázorněn na obrázku 3.2.

4.1.11 Třída IntCtrl

Pro ovládání řadiče slouží metoda *clock* určená k provádění v první fázi taktu, která jako parametr dostává řídící signál INTA. Dále třída implementuje getter a setter stavového signálu INT a čísla přerušení.

4.1.11.1 Metoda clock

Metoda *clock* zapíše v případě aktivního signálu INTA číslo přerušení na datovou sběrnici.

4.2 Překladač

Podobně jako u procesoru jsem se v případě implementace překladače snažil z důvodu případné znovupoužitelnosti kódu o minimalizaci využití knihovny Qt. Využil jsem pouze multiplatformní číselné datové typy, řetězce typu *QString* a šablony datových struktur *QList* a *QMap*, které jsou kompatibilní s obdobnými datovými typy v *STL*.

4.2.1 Třída Compiler

Pro vložení vstupu překladače bude sloužit metoda *setInput*. Nejdůležitejší metodou třídy *Compiler* bude metoda *compile*. Dále bude třída *Compiler* implementovat několik metod pro získání informací o překladu.

4.2.1.1 Metoda compile

Metoda *compile* provede překlad zdrojového kódu jazyka symbolických mikroinstrukcí do firmwaru v binární podobě a vytvoří struktury s informacemi o průběhu překladu, seznamem návěstí a čitelnou podobou firmaru. Pracuje se strukturou *Program* vytvořenou metodou *parse* třídy *Parser*. Průběh metody *compile* je následující:

1. parsuje soubor - metoda *parse* třídy *Parser*
2. v případě neúspěchu (chyba při parsování) přidá chybu do výstupu překladu a přejde na krok 11
3. vytvoří seznam návěstí a detekuje případné duplicity (v případě chyby tuto přidá do výstupu překladu)
4. doplní nepojmenovaná návěstí
5. doplní nevyplněné cíle skoku
6. prohledáváním do šířky prochází všechny dosažitelné mikroinstrukce u nichž:
   - zkontroluje operaci aritmeticko-logické jednotky v rozsahu 0–15
   - zkontroluje typ podmínky v rozsahu 0–27
   - zkontroluje seznam signálů na jejich existenci a splnění omezení
   - vytvoří binární podobu mikroinstrukce

- najde cílovou mikroinstrukci aktuální mikroinstrukce, přidá ji do fronty prohledávání (pokud tam ještě není) a uloží si její ukazatel
- zkontroluje, zda velikost cílového bloku/mikroinstrukce odpovídá typu podmínky
- v případě chyby v některém kroku přidá tuto do výstupu překladu

7. vymaže všechny nedostupné bloky programu a za každý takový přidá varování do seznamu výstupů
8. v případě chyby v kterémkoli kroku přejde na krok 11
9. rozmístí mikroinstrukce do pole struktur *Inst* a velikosti bloků do pole typu *int*:
   - první blok umístí na začátek pole
   - umisťuje bloky do pole mikroinstrukcí tak, že v pořadí od největších bloků umístí mikroinstrukce každého bloku za sebou od první volné pozice s indexem dělitelným velikostí bloku a velikost bloku uloží do pole velikostí na odpovídající index
   - v případě, že se nenajde vhodné umístění pro daný blok, přidá chybu do výstupu překladu a přejde na krok 11
   - do struktury každé takto umístěné mikroinstrukci uloží její index
10. projde v předchozím kroku vytvořené pole mikroinstrukcí a každé:
    - přidá do pole pro seznam řádek na její index její číslo řádky
    - přidá do její binární podoby index její cílové mikroinstrukce
11. v případě chyby v kterémkoli kroku přidá do výstupu překladu informaci, že překlad byl neúspěšný, a vrátí *false*, v opačném případě přidá do výstupu překladu informaci, že překlad byl úspěšný, a vrátí *true*

4.2.1.2 Metody pro získání dat z překladu

getFirmware vrací pole o 512 prvcích (délka řídící paměti) o šířce 64 bitů (počet řídících signálů) s obsahem přeložené řídící paměti (návratová hodnota typu ukazatel na *quint64*)

getOutput vrací ukazatel na seznam zpráv o průběhu překladu pro zobrazení výpisu v uživatelském rozhraní (návratová hodnota typu ukazatel na *QList<CompileMessage>*)

getLabels vrací seznam návěstí s jejich umístěním ve zdrojovém kódu (návratová hodnota typu ukazatel na *QMap<int,QString>*) (Tento seznam se nepřepíše, pokud při dalším překladu dojde k chybě v překladu. Díky tomu lze používat seznam návěstí i při chybě.)

getSegmentation vrací rozmístění instrukcí v řídící paměti pro zobrazení řídící paměti v uživatelském rozhraní v čitelné podobě (návratová hodnota typu ukazatel na ukazatel na *Inst*)

getLines vrací číslování řádek jednotlivých položek v řídící paměti pro jejich odkazování do textového editoru (návratová hodnota typu ukazatel na *int*)

getBlocks vrací délku bloků začínajících na jednotlivých adresách v řídící paměti pro možnost exportu překladového logu kompatibilního se simulátorem SIMDOP (návratová hodnota typu ukazatel na *int*)

4.2.2 Třída Parser

Základem této třídy je metoda *parse*.

4.2.2.1 Metoda parse

Metoda *parse* slouží k parsování zdrojového souboru, který dostává jako parametr. Návratovou hodnotou je ukazatel na objekt typu *Program*. Metoda chytá výjimku typu *CompileMessage*, při jejímž zachycení se hodnota přidá do struktury *Program* a parsování se ukončí.

Parsování probíhá přímočaře podle gramatiky jazyka symbolických mikroinstrukcí, kterou naleznete v sekci 2.2.1.2. Tokeny získává metodou *next* třídy *Tokenizer*, správnost tokenu ověřuje metoda *compare*.

4.2.2.2 Metoda compare

Metoda *compare* porovnává požadovaný typ tokenu, který je jí předán parametrem, s typem tokenu získaným z tokenizeru. Pokud se tyto typy neshodují a nelze je mezi sebou převést (převést jde pouze typ číslo na typ identifikátor), vytvoří strukturu *CompileMessage*, kterou vyhodí jako výjimku.

4.2.3 Třída Tokenizer

Třída implementuje metodu *setInput* pro nastavení vstupních dat a metodu *next*.

4.2.3.1 Metoda next

Metoda *next* vrací hodnotu typu *Token* určující další token nalezený v toku dokumentu.

Metoda hledá následující token od pozice za předchozím hledáním. Vynechává přitom všechny bílé znaky (kromě nového řádku) a komentáře.

4.2.4 Struktura Token

Struktura obsahuje tyto položky:

type položka výčtového typu určující typ tokenu (identifikátor, číslo, čárka, levá složená závorka, pravá složená závorka, středník, dvojtečka, nový řádek, konec souboru)

line položka typu *int* udávající číslo řádky výskytu tokenu

name položka typu *QString* obsahující v případě tokenu typu identifikátor jeho název

value položka typu *int* obsahující v případě tokenu typu číslo jeho hodnotu

Dále struktura implementuje metodu *toString*, která vypíše textovou reprezentaci tokenu, a metodu *typeToString*, která vypíše textovou reprezentaci typu tokenu.

4.2.5 Struktura Program

Struktura *Program* je potomkem třídy *QList<Block*>* a obsahuje jedinou položku *parserError* typu *CompileMessage* obsahující případnou chybu vzniklou během parsování.

4.2.6 Struktura Block

Struktura *Block* je potomkem třídy *QList<Inst*>* a obsahuje následující položky:

maxCond položka typu *int* udávající velikost největšího rozskoku podmínky z mikroinstrukcí, které mají tento blok jako cílový

line položka typu *int* udávající číslo rádky obsahující návěstí tohoto bloku

label položka typu *QString* obsahující návěstí bloku

4.2.7 Struktura Inst

Struktura obsahuje tyto položky:

line položka typu *int* udávající číslo rádky obsahující mikroinstrukci

label položka typu *QString* obsahující návěstí mikroinstrukce

signalList položka typu *QStringList* obsahující seznam aktivních signálů v textové podobě

alu položka typu *int* udávající kód operace aritmeticko-logické jednotky

cond položka typu *int* udávající kód podmínky skoku

dest položka typu *QString* udávající návěstí následující mikroinstrukce

binary položka typu *quint64* obsahující binární reprezentaci dané mikroinstrukce

destInst položka typu ukazatel na *Inst* ukazující na následující mikroinstrukci

addr položka typu *int* udávající výslednou adresu mikroinstrukce v řídící paměti procesoru

4.2.8 Struktura CompileMessage

Struktura *CompileMessage* obsahuje tyto položky:

component položka výčtového typu udávající kompomentu, ve které zpráva vznikla (*Compiler*, *Parser*, *Tokenizer*, jiná)

type položka výčtového typu udávající typ zprávy (varování, chyba, informace, žádná zpráva)

line položka typu *int* udávající číslo řádky, ke které se zpráva váže

text položka typu *QString* obsahující text zprávy

argList položka typu *QStringList* obsahující hodnoty argumentů v textu zprávy

4.3 Ovladač procesoru

Ovladač procesoru je rozhraním mezi widgety uživatelského rozhraní a třídou *DOP* a je reprezentován třídou *DOPDriver*.

4.3.1 Třída DOPDriver

Třída *DOPDriver* je potomkem třídy *QObject*. v konstruktoru vyžaduje ukazatele na číslování řádek jednotlivých položek v řídící paměti (viz sekce 4.2.1: návratová hodnota metody *getLines*) a ukazatel na seznam breakpointů (viz sekce 4.4.2: návratová hodnota metody *getBreakpoints*).

Třída implementuje gettery akcí (více o akcích v sekci 4.4.1), getter a setter jednotlivých položek hlavní paměti a metodu *setFirmware* pro načtení řídící paměti procesoru. Dále bude implementovat metody *getValues*, *setValue*, *getValue*, *getSignal* a *getSignalNames*, které si rozebereme podrobněji.

4.3.1.1 Metoda getValues

Metoda *getValues* slouží k získání informací o všech dostupných vnitřních hodnotách procesoru. Návratová hodnota je typu *QList<DOPValueInfo>*, v němž každá položka obsahuje informace o jedné vnitřní hodnotě a s jehož pomocí lze snadno algoritmicky se všemi hodnotami pracovat. Toho využívá třída *CoreWidget*.

4.3.1.2 Metody getValue a setValue

Metoda *getValue* slouží k získání jedné konkrétní vnitřní hodnoty procesoru. Jejím parametrem je identifikátor výčtového typu, který hodnotu v rámci procesoru jednoznačně identifikuje. Návratová hodnota je typu *quint16*, jsou jí však předávány i parametry typu *quint8* a *bool*.

Podobně metoda *setValue* má jako první parametr identifikátor výčtového typu a jako druhý novou hodnotu typu *quint16*.

4.3.1.3 Metody getSignal a getSignalNames

Metoda *getSignal* bere jako parametr název signálu (datový typ *QString*) a vrací hodnotu typu *quint64*, která obsahuje masku daného signálu ve slově řídící paměti, nebo nulu, pokud signál s daným názvem neexistuje.

Metoda *getSignalNames* pak vrací hodnotu typu *QStringList* s názvy všech řídících signálů.

4.3.1.4 Signály

lineChanged je vyslán při změně aktuální mikroinstrukce a jako parametr předává číslo řádku, na kterém se aktuální mikroinstrukce nachází, je na něj napojen slot *setCurrentAddressLine* třídy *CodeEdit*

addrChanged je vyslán při změně aktuální mikroinstrukce a jako parametr předává adresu v řídící paměti, na které se aktuální mikroinstrukce nachází, je na něj napojen slot *addrChanged* třídy *ProgramMemoryModel*

dataChanged je vyslán při změně vnitřních hodnot procesoru, je na něj připojen slot *revert* třídy *DOPValueEdit* a slot *updateValues* třídy *ALUGraphic*

memoryToChange je vyslán těsně před změnou hodnoty v hlavní paměti procesoru, je na něj napojen slot *submit* třídy *MemoryTableModel*

memoryChanged je vyslán po změně hodnoty v hlavní paměti procesoru, je na něj napojen slot *revert* třídy *MemoryTableModel*

4.3.1.5 Sloty

setAddr slouží k nastavení aktuální mikroinstrukce na adresu v řídící paměti danou parametrem slotu, je připojen na signál *doubleClicked* třídy *ProgramMemoryModel*

enableSimulation slouží k povolení akcí spouštějících simulaci, je volán po úspěšné kompilaci

disableSimulation slouží k zakázání akcí spouštějících simulaci, je volán po neúspěšné kompilaci a je připojen na signál *textChanged* třídy *CodeEdit* respektive jejího rodiče

Sloty pro manipulaci s pamětí a stavem procesoru:

resetState resetuje všechny hodnoty procesoru

saveState uloží všechny editovatelné hodnoty procesoru do zvoleného souboru, je připojen na odpovídající akci hlavního menu

loadState načte všechny editovatelné hodnoty procesoru ze zvoleného souboru, je připojen na odpovídající akci hlavního menu

resetMemory nastaví všechny položky hlavní paměti na 0, je připojen na odpovídající akci hlavního menu

saveMemory uloží obsah hlavní paměti do zvoleného souboru, je připojen na odpovídající akci hlavního menu

loadMemory načte obsah hlavní paměti ze zvoleného souboru, je připojen na odpovídající akci hlavního menu

Sloty pro ovládání běhu simulace

runClock proběhne jeden takt procesoru, je připojen na odpovídající akci hlavního menu

runTo umožní zvolit určitou hodnotu programového čítače, nebo adresu mikroinstrukce, poté procesor běží do zvolené hodnoty, je připojen na odpovídající akci hlavního menu

runToPC procesor běží do změny hodnoty programového čítače, je připojen na odpovídající akci hlavního menu

runToBreakpoint procesor běží do nejbližšího breakpointu, je připojen na odpovídající akci hlavního menu

abortSimulation slouží k přerušení aktuálně běžící simulace, je připojen na odpovídající akci hlavního menu

Ve slotech implementujících běh simulace jsem musel vyřešit situaci, kdy zvolená podmínka nemůže nastat. V takový moment došlo k zablokování činnosti programu z důvodu uváznutí v nekonečné smyčce. Proto jsem do všech cyklů umístil příkaz, který spustí smyčku událostí knihovny Qt, díky čemuž se uživatelské rozhraní stalo použitelným i v případě dlouhého běhu simulace. Poté již stačilo přidat možnost zrušení běžící simulace nastavením příznaku, který jsem v každém kroku simulačního cyklu kontroloval.

4.3.2 Struktura DOPValueInfo

Struktura obsahuje následující položky:

id položka výčtového typu obsahující identifikátor hodnoty

type položka výčtového typu obsahující typ hodnoty (dvoubitová, osmibitová, šestnáctibitová)

category položka typu *int* sloužící pro ukládání binárního součtu hodnot výčtového typu osahujícího kategorii hodnoty (registr, alu, paměťový řadič, ...)

editable položka typu *bool* udávající, zda lze danou položku editovat, tedy zda pro ni *DOPDriver* implementuje setter

tristate položka typu *bool* udávající, zda může hodnota nabývat stavu *odpojeno* (tedy zda se jedná o sběrnici)

4.4 Uživatelské rozhraní

Jedna z možných podob uživatelského rozhraní je na obrázku 4.1. Další varianty lze získat přesouváním jednotlivých widgetů

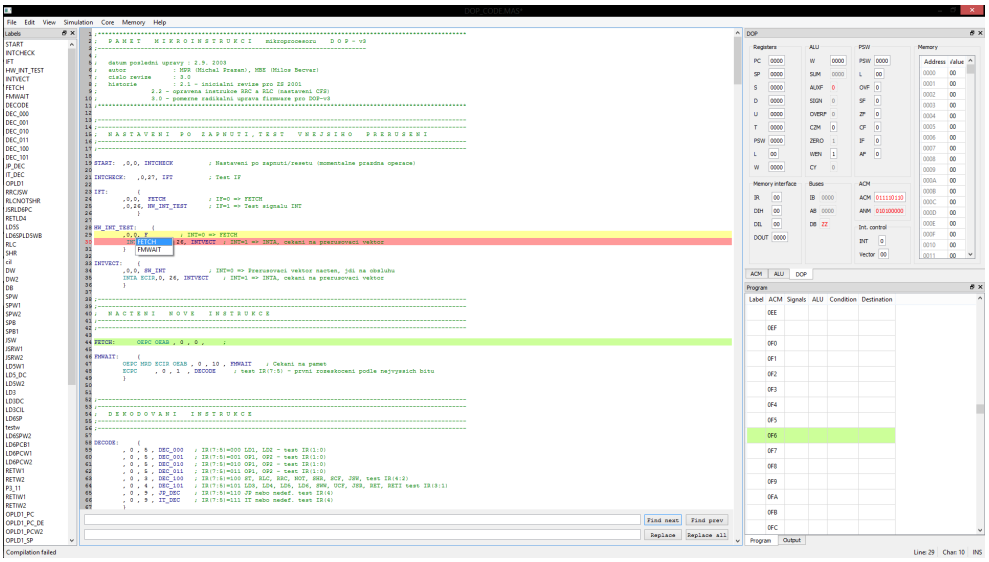
Minimální rozlišení obrazovky pro bezchybný chod aplikace je 1024×768, doporučené rozlišení je minimálně 1366×768.

4.4.1 Třída Simdop

Třída *Simdop* je potomkem třídy *QMainWindow*. Hlavní funkce třídy spočívá v konstruktoru, kde vytváří ostatní objekty uživatelského rozhraní, a v akcích a na ně navázaných slotech.

Kromě widgetů obsahuje *Simdop* také objekt třídy *QTimer* pro automatický překlad. Ten je napojen na signál *textChanged* textového editoru třídy *CodeEdit* a po zvoleném intervalu vyvolá překlad souboru. Dále obsahuje třída objekt typu *QFileSystemWatcher* nastavený na sledování právě otevřeného souboru na změnu mimo simulátor.

Obrázek 4.1: Podoba uživatelského rozhraní



4.4.1.1 Konstruktor

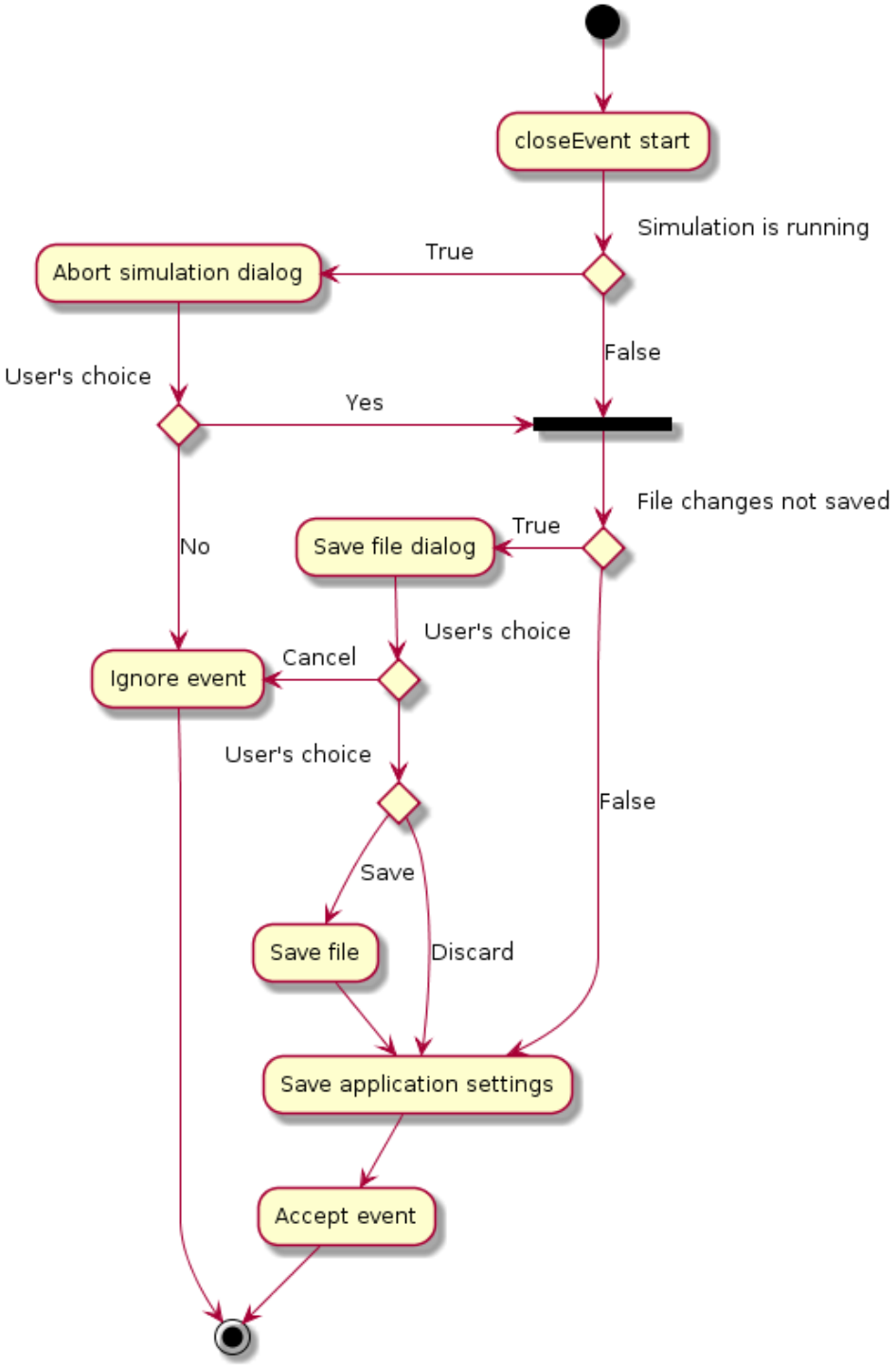
Průběh konstruktoru třídy *Simdop* je následující:

1. vytvoří objekty tříd *QTimer* a *QFileSystemWatcher*
2. vytvoří objetky tříd *Compiler* a *DOPDriver*
3. vytvoří centrální widget (*CodeEdit*)
4. vytvoří všechny dokovací widgety a jejich modely
5. vytvoří status bar
6. vytvoří objekty typu *QAction* a hlavní menu
7. načte persistentní nastavení pomocí třídy *QSettings*
8. vyvolá hlavní okno aplikace

4.4.1.2 closeEvent

Další důležitou metodou hlavní třídy aplikace je metoda *closeEvent*. Ta je volána knihovnou Qt při pokusu o zavření aplikace. Tato událost může být v závislosti na stavu textového editoru a simulace potvrzena (v takovém případě je nejdříve pomocí *QSettings uložen stav a nastavení aplikace*), nebo zrušena. Průběh metody je zobrazen na obrázku 4.2.

Obrázek 4.2: Diagram aktivit metody *closeEvent*



4.4.1.3 Sloty

newFile zavře současný soubor a zobrazí prázdný editor, je připojen na odpovídající akci hlavního menu

openFile otevře soubor vybraný pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

saveFile uloží soubor a pokud se jedná o nový soubor, zvolí se cílové umístění pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

saveAs uloží soubor do umístění zvoleného pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

setFileChange nastaví příznak, že soubor byl změněn od posledního uložení, je připojen na signál *modificationChanged* třídy *CodeEdit* respektive jeho rodiče

updateCursorPosition slouží k aktualizování aktuální pozice kurzoru textového editoru ve stavovém řádku, je připojen na signál *cursorPositionChanged* třídy *CodeEdit* respektive jeho rodiče

updateOverwriteMode slouží k aktualizování způsobu zadávání do textového editoru ve stavovém řádku, je připojen na signál *overwriteModeChanged* třídy *CodeEdit*

exportLogFile vytvoří log soubor z překladu kompatibilní s programem SIMDOP do úmístění zvoleného pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

exportBinary vytvoří binární soubor s obsahem řídící paměti procesoru do úmístění zvoleného pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

exportASCII vytvoří textový soubor s obsahem řídící paměti procesoru do úmístění zvoleného pomocí *QFileDialog*, je připojen na odpovídající akci hlavního menu

fileChangeHandler slouží k vyvolání dialogu v případě, že je otevřený soubor upraven vně simulátoru, je připojen na signál *fileChanged* třídy *QFileSystemWatcher*

compile je připojen na odpovídající akci hlavního menu, průběh slotu je následující:

- vyvolá slot *submit* objektů tříd *LabelListModel*, *CompilerOutputModel* a *ProgramMemoryModel*
- spustí překlad metodou *compile* třídy *Compiler*

- v případě úspěchu nastaví třídě *DOPDriver* nový firmware a umožní spuštění simulace metodou *enableSimulation* třídy *DOPDriver*, v případě neúspěchu znemožní spuštění simulace metodou *disableSimulation* třídy *DOPDriver*
- do stavového řádku vyšle zprávu o úspěchu či neúspěchu
- vyvolá slot *revert* objektů tříd *LabelListModel*, *CompilerOutputModel* a *ProgramMemoryModel*

setSimulationRunning nastaví příznak, že běží simulace, a znemožní úpravy v textovém editoru, je připojen na signál *simulationStarted* třídy *CodeEdit*

unsetSimulationRunning zruší příznak, že běží simulace, a umožní úpravy v textovém editoru, je připojen na signál *simulationStopped* třídy *CodeEdit*

showAbout zobrazí okno s informacemi o programu, je připojen na odpovídající akci hlavního menu

showHelp otevře v prohlížeči uživatelskou příručku k programu

4.4.1.4 Akce hlavního menu

Každá položka v hlavním menu odpovídá jedné akci. Některé akce jsou získávány z jiných objektů (*DOPDriver*, *CodeEdit*).

File :

New vytvoří nový soubor

Open otevře existující soubor

Save uloží soubor

Save as uloží soubor do zvolené cesty

Export :

Log file vytvoří log soubor z aktuálního překladu

Binary exportuje řídící paměť do binárního souboru

Binary (ASCII) exportuje řídící paměť do textového souboru

Exit zavře okno aplikace

Edit :

Undo vrátí poslední úpravu

Redo znovu provede úpravu

Cut vyjme výběr a vloží do schránky

Copy zkopíruje výběr do schránky

Paste vloží obsah schránky

Find... otevře widget pro vyhledávání v textovém editoru

Find next najde následující výskyt hledaného

Find previous najde předchozí výskyt hledaného

Replace... otevře widget pro nahrazování v textovém editoru

Replace all nahradí všechny výskyty zvoleným textem

Go to... zobrazí dialog pro přejití na zvolený řádek

View obsahuje akce pro zobrazení/skrytí všech dokovacích widgetů

Simulation :

Add/Remove breakpoint přidá nebo odebere breakpoint na aktuálním řádku textového editoru, text akce se mění podle přítomnosti breakpointu na daném řádku

Run for one clock spustí simulaci jednoho taktu procesoru

Run to breakpoint spustí simulaci do breakpointu

Run to PC spustí simulaci do změny programového čítače

Run to... spustí simulaci do určité hodnoty zvoleného registru

Restart simulation restartuje simulaci

Stop simulation zastaví běžící simulaci

Core :

Reset memory resetuje hlavní paměť

Save memory uloží obsah hlavní paměti

Load memory načte obsah hlavní paměti

Memory :

Reset state resetuje stav procesoru

Save state uloží stav procesoru

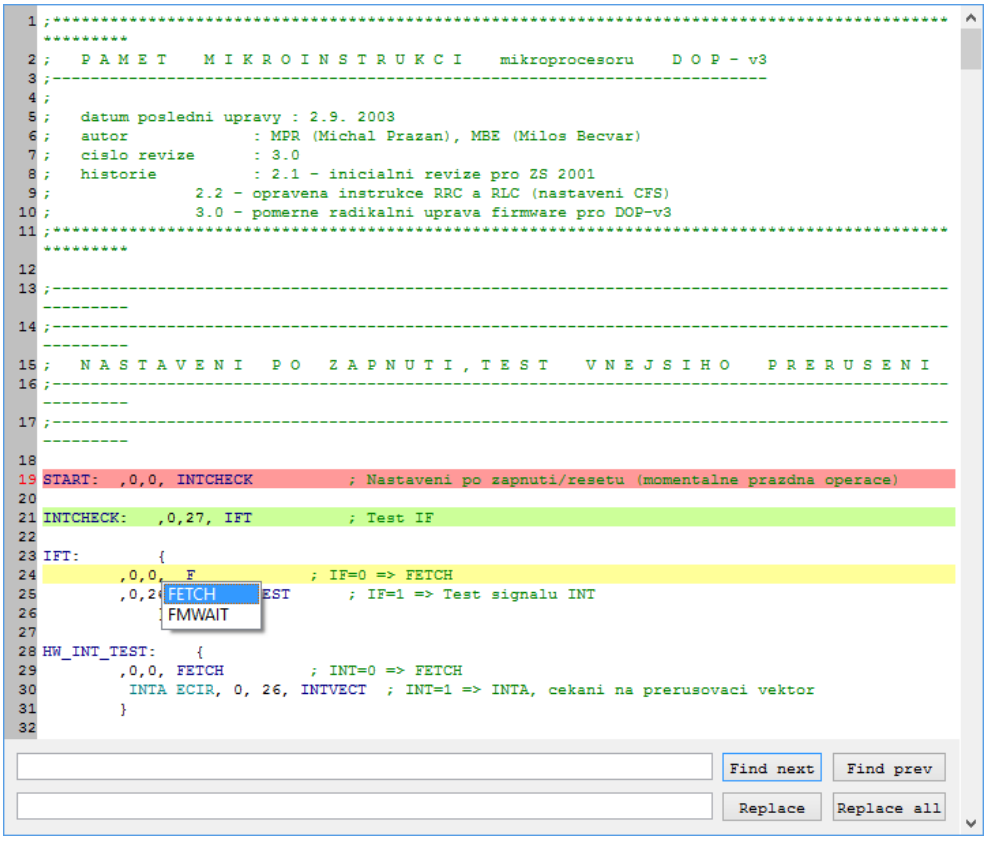
Load state načte stav procesoru

Help :

About zobrazí informace o programu

Help zobrazí nápovědu k programu

Obrázek 4.3: Widget textového editoru



4.4.2 Třída CodeEdit

Třída *CodeEdit* je potomkem třídy *QPlainTextEdit*. Pro svou funkci potřebuje ukazatel na objekt třídy *QCompleter*, pro doplňování textu, na seznam návěstí a jejich řádkování (*QMap<int,QString>*) a na seznam breakpointů (*QList<int>*). Dále bude obsahovat objet třídy *LineNumberArea* pro zobrazování číslování řádek, *Finder* pro ovládání vyhledávání v textu a *Highlighter* pro zvýrazňování syntaxe jazyka symbolických mikroinstrukcí. Podoba widgetu je vidět na obrázku 4.3.

4.4.2.1 Signály

Třída implementuje jediný signál *overwriteModeChange*, který je vyvolán, pokud dojde ke stisknutí klávesy *insert*. Je na něj připojen slot *updateOverwriteMode* třídy *Simdop*.

4.4.2.2 Sloty

Ze slotů implementuje třída sloty pro vyhledávání (*find*, *findNext*, *findPrevious*, *replace*, *replaceNext* a *replaceAll*), které jsou napojeny na odpovídající akce v hlavním menu respektive na odpovídající tlačítka widgetu *Finder*.

Dalšími sloty jsou:

toggleBreakpoint přidá/odebere breakpoint na aktuálním řádku, je připojen na odpovídající akci hlavního menu

goToLine přejde na řádek zvolený parametrem, nebo zobrazí dialog pro zvolení řádku, je připojen na odpovídající akci hlavního menu, na signál *doubleClicked* třídy *LabelListModel*, na signál *lineChanged* třídy *DOPDriver* a na akci v kontextovém menu, která se přidá po umístění kurzoru na návěstí a slouží k přesunu na jeho definici

setCurrentAddressLine slouží k nastavení aktuálně prováděné mikroinstrukce, je připojen na signál *addrChanged* třídy *DOPDriver*

highlight zvýrazňuje řádek obsahující kurzor, řádek obsahující aktuální mikroinstrukci, párování složených závorek, řádky s breakpointy a výskyty výrazu ve vyhledávacím řádku, je připojen na signál *textChanged* třídy *CodeEdit* respektive jeho rodiče *QPlainTextEdit*

4.4.3 Třída Finder

Tento widget obsahuje uživatelské vstupy pro vyhledávání v textovém editoru. Je zobrazen při spuštění vyhledávací či nahrazovací akce a skryt klávesou Escape. Obsahuje tlačítka *Find next*, *Find Prev*, *Replace* a *Replace All* a řádkové vstupy pro zadání hledaného výrazu a nahrazujícího výrazu.

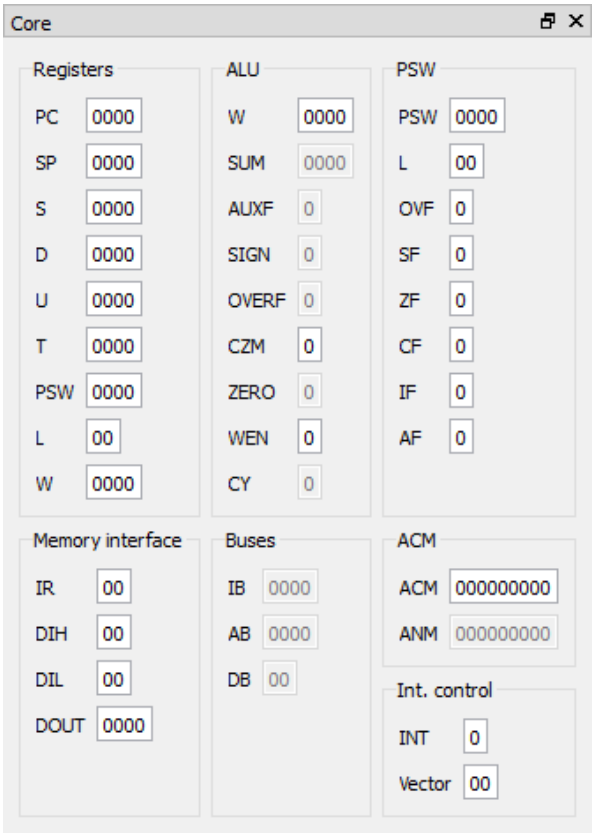
4.4.4 Třída CompleterModel

Třída *CompleterModel* je potomkem třídy *QAbstractListModel* a slouží k poskytování dat třídě *QCompleter* využívané v textovém editoru. Tento model obsahuje seznam všech jmen signálů procesoru a seznam všech návěstí použitých v textovém souboru.

4.4.5 Třída CoreWidget

Třída *CoreWidget* zobrazuje vnitřní hodnoty procesoru do několika skupin. V konstruktoru je pro každou skupinu vytvořen *QGroupBox*, poté je v cyklu procházen seznam všech vnitřních hodnot (získaný metodou *getValues* třídy *DOPDriver*) a do každé skupiny jsou přidány objekty třídy *DOPValueEdit* reprezentující hodnotu z kategorie odpovídající dané skupině. Podoba widgetu je vidět na obrázku 4.4.

Obrázek 4.4: Widget pro zobrazení a úpravu vnitřních hodnot procesoru



4.4.6 Třída DOPValueEdit

Třída *DOPValueEdit* je potomkem třídy *QLineEdit*. Parametrem konstruktoru je ukazatel na objekt typu *DOPDriver* a také reference na objekt typu *DOPValueInfo*. Z tohoto objektu získává třída identifikátor hodnoty a její typ.

4.4.6.1 Konstruktor

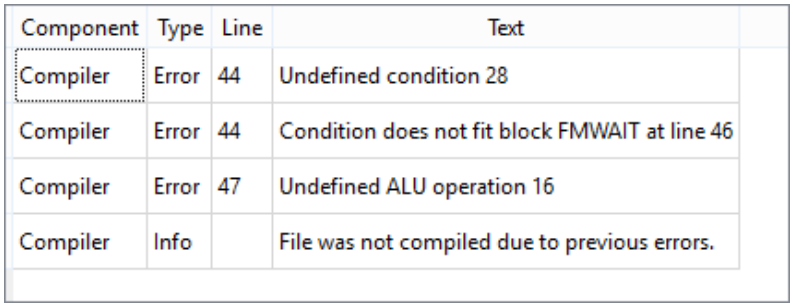
V konstruktoru je podle typu hodnoty zvolena příslušná šířka editovacího řádku a jeho validátor.

4.4.6.2 Sloty

revert obnoví hodnotu v poli a pokud došlo ke změně, je pole zvýrazněno, je připojen na signál *dataChanged* třídy *DOPDriver*

submit slouží k odeslání nové hodnoty do třídy *DOPDriver*, je připojen na signál *textEdited* svého rodiče *QLineEdit*

Obrázek 4.5: Widget výstupu překladače



Component	Type	Line	Text
Compiler	Error	44	Undefined condition 28
Compiler	Error	44	Condition does not fit block FMWAIT at line 46
Compiler	Error	47	Undefined ALU operation 16
Compiler	Info		File was not compiled due to previous errors.

4.4.7 Třída CompilerOutputView

Tuto třídu nebylo třeba implementovat, místo ní je k zobrazování dat z *CompilerOutputModel* použit standardní *QTableView*. Podoba widgetu je vidět na obrázku 4.5.

4.4.8 Třída CompilerOutputModel

Třída *CompilerOutputModel* je potomkem třídy *QAbstractTableModel* a poskytuje data z výstupu překladu. Jedná se o tabulku, kde každý řádek obsahuje jednu zprávu z překladu. Tabulka má čtyři sloupce, název zdrojové komponenty zprávy (Compiler, Parser, Tokenizer, other), typ zprávy (none, Info, Warning, Error), číslo řádky, k níž se zpráva váže, a text zprávy.

Třída reimplementuje všechny nezbytné metody. Více informací je dostupných v dokumentaci Qt [5] a zdrojovém kódu aplikace.

4.4.8.1 Signály

Třída implementuje jediný signál *doubleClicked*, který informuje o tom, že bylo kliknuto na řádek, a jako parametr udává odpovídající řádek kódu. Na tento signál je napojen slot *goToLine* třídy *CodeEdit*

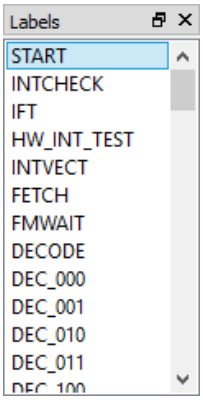
4.4.8.2 Sloty

submit připraví model na reset, je vyvolán třídou *Simdop* v případě chystaného překladu

revert dokončí reset modelu, je vyvolán třídou *Simdop* v případě dokončení překladu

doubleClick tento slot vyvolá signál *doubleClicked*, je připojen na signál *doubleClicked* třídy *QTableView* určené pro zobrazování tohoto modelu

Obrázek 4.6: Widget návěstí



4.4.9 Třída LabelListView

Tuto třídu nebylo třeba implementovat, místo ní je k zobrazování dat z *LabelListModel* použit standardní *QListView*. Podoba widgetu je vidět na obrázku 4.6.

4.4.10 Třída LabelListModel

Třída *LabelListModel* je potomkem třídy *QAbstractListModel*, zobrazuje seznam návěstí vytvořený v překladači a zařizuje přejití textového editoru na odpovídající řádek v případě dvojkliku na návěstí. Pracuje s datovou strukturou typu *QMap<int,QString>* vytvořenou překladačem.

Třída reimplementuje všechny nezbytné metody. Více informací je dostupných v dokumentaci Qt [5] a zdrojovém kódu aplikace.

4.4.10.1 Signály

Třída implementuje jediný signál *doubleClicked*, který informuje o tom, že bylo kliknuto na řádek, a jako parametr udává odpovídající řádek kódu. Na tento signál je napojen slot *goToLine* třídy *CodeEdit*

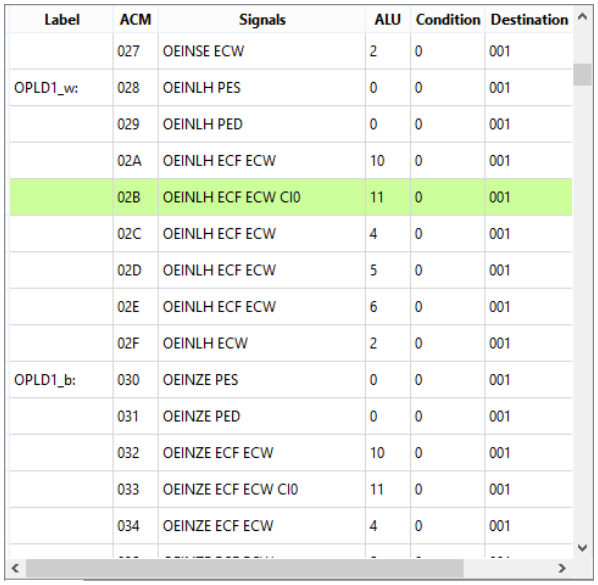
4.4.10.2 Sloty

submit připraví model na reset, je vyvolán třídou *Simdop* v případě chystaného překladu

revert dokončí reset modelu, je vyvolán třídou *Simdop* v případě dokončení překladu

doubleClick tento slot vyvolá signál *doubleClicked*, je připojen na signál *doubleClicked* třídy *QListView* určené pro zobrazování tohoto modelu

Obrázek 4.7: Widget řídící paměti



Label	ACM	Signals	ALU	Condition	Destination
	027	OEINSE ECW	2	0	001
OPLD1_w:	028	OEINLH PES	0	0	001
	029	OEINLH PED	0	0	001
	02A	OEINLH ECF ECW	10	0	001
	02B	OEINLH ECF ECW CI0	11	0	001
	02C	OEINLH ECF ECW	4	0	001
	02D	OEINLH ECF ECW	5	0	001
	02E	OEINLH ECF ECW	6	0	001
	02F	OEINLH ECW	2	0	001
OPLD1_b:	030	OEINZE PES	0	0	001
	031	OEINZE PED	0	0	001
	032	OEINZE ECF ECW	10	0	001
	033	OEINZE ECF ECW CI0	11	0	001
	034	OEINZE ECF ECW	4	0	001

4.4.11 Třída ProgramMemoryView

Tuto třídu nebylo třeba implementovat, místo ní je k zobrazování dat z *ProgramMemoryModel* použit standardní *QTableView*. Podoba widgetu je vidět na obrázku 4.7.

4.4.12 Třída ProgramMemoryModel

Třída *ProgramMemoryModel* je potomkem třídy *QAbstractTableModel* a poskytuje data reprezentující obsah řídící paměti procesoru v čitelné podobě. Pracuje s polem objektů typu *Inst*, které vytváří překladač. Jedná se o tabulku, kde každý řádek obsahuje jednu mikroinstrukci. Tabulka má šest sloupců, návěstí, adresu, seznam signálů v textové podobě, kód operace aritmeticko-logické jednotky, typ podmínky a adresu následující mikroinstrukce. Mimo to také vrací zelenou barvu pro pozadí pro právě aktivní mikroinstrukci.

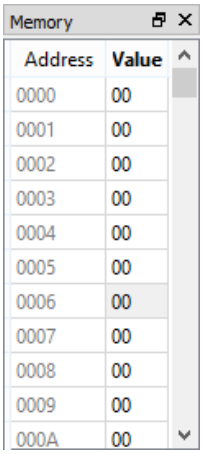
Třída reimplementuje všechny nezbytné metody. Více informací je dostupných v dokumentaci Qt [5] a zdrojovém kódu aplikace.

4.4.12.1 Signály

doubleClicked informuje o tom, že bylo kliknuto na řádek, a jako parametr udává odpovídající adresu v řídící paměti, je napojen na setter aktuální adresy třídy *DOPDriver*

itemSelected informuje o tom, že byla změněna aktuální mikroinstrukce, je napojen na slot *setCurrentIndex* třídy *QTableView* určené pro zobrazo-

Obrázek 4.8: Widget hlavní paměti



vání tohoto modelu

4.4.12.2 Sloty

submit připraví model na reset, je vyvolán třídou *Simdop* v případě chystaného překladu

revert dokončí reset modelu, je vyvolán třídou *Simdop* v případě dokončení překladu

doubleClick tento slot vyvolá signál *doubleClicked*, je připojen na signál *doubleClicked* třídy *QTableView* určené pro zobrazování tohoto modelu

addrChanged tento slot vyvolá signál *itemSelected*, je připojen na signál *addrChanged* třídy *DOPDriver*

4.4.13 Třída MemoryTableView

Třída je potomkem třídy *QTableView* a slouží k zobrazování dat z modelu *MemoryTableModel*. Reimplementuje metodu *resizeEvent* pro automatické proporcionální roztahování sloupců na šířku tabulky. Dále nastavuje takzvaný delegát, což je v mém případě objekt třídy *MemoryTableDelegate*, který nastavuje editor prvků tabulky. Podoba widgetu je vidět na obrázku 4.8.

4.4.14 Třída MemoryTableDelegate

Třída je potomkem třídy *QItemDelegate*. Implementuje delegát prvku jako objekt třídy *QLineEdit* a nastavuje mu validátor pro reprezentaci osmibitového čísla v hexadecimálním formátu.

4.4.15 Třída MemoryTableModel

Třída *MemoryTableModel* je potomkem třídy *QAbstractTableModel* a poskytuje data reprezentující obsah hlavní paměti procesoru. Pracuje s getterem a setterem paměťových buněk třídy *DOPDriver*. Tabulka má dva sloupce, adresu a hodnotu.

Třída reimplementuje všechny nezbytné metody. Více informací je dostupných v dokumentaci Qt [5] a zdrojovém kódu aplikace.

4.4.15.1 Sloty

submit připraví model na reset, je připojen na signál *memoryToChange* třídy *DOPDriver*

revert dokončí reset modelu, je připojen na signál *memoryChanged* třídy *DOPDriver*

4.4.16 Třída ALUGraphic

Třída je potomkem třídy *QWidget*. Reimplementuje metody *paintEvent* a *resizeEvent*. Pro vykreslování diagramu využívá blokového schématu aritmeticko-logické jednotky [3]. Pro vykreslování hodnot využívá strukturu *ALUValue*. Podoba widgetu je vidět na obrázku 4.9.

V konstruktoru se vytváří datová struktura *QList<ALUValue>*, která obsahuje údaje o všech hodnotách zobrazovaných v diagramu aritmeticko-logické jednotky.

4.4.16.1 Metoda resizeEvent

Metoda *resizeEvent* slouží k přeškálování diagramu aritmeticko-logické jednotky tak, aby vyhovoval aktuálním rozměrům widgetu. Při přeškálování je zachován poměr stran diagramu.

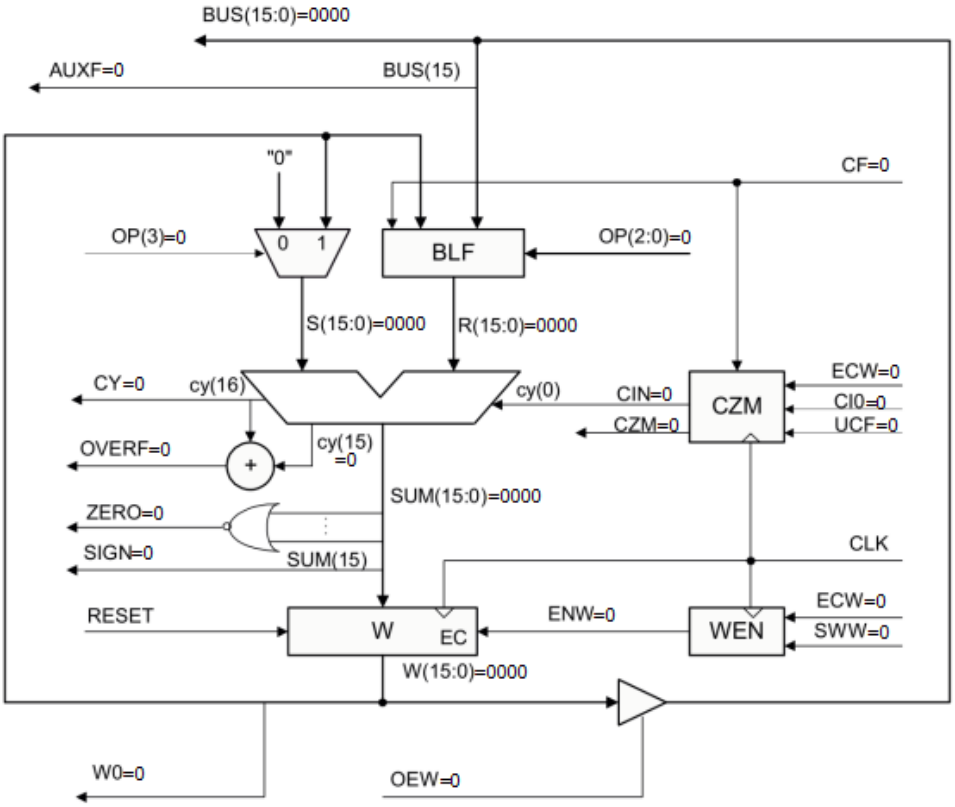
4.4.16.2 Metoda paintEvent

Metoda *paintEvent* vykreslí diagram a do něho všechny hodnoty obsažené v datové struktuře typu *QList<ALUValue>*. Hodnoty, které mají nastavený příznak *changed*, vykreslí červeně.

4.4.16.3 Sloty

Třídy implementuje jediný slot *updateValues*, který projde celý list hodnot, porovná je s aktuálními hodnotami v objektu třídy *DOPDriver* a v případě rozdílu načte nové hodnoty a nastaví příznaky *changed*. Slot je připojen na signál *dataChanged* třídy *DOPDriver*.

Obrázek 4.9: Widget diagramu aritmeticko-logické jednotky



4.4.17 Struktura ALUValue

Struktura *ALUValue* reprezentuje jednu hodnotu zobrazovanou třídou *ALUGraphic*. Obsahuje tyto položky:

id položka výčtového typu obsahující identifikátor hodnoty

value položka typu *quint16* obsahující aktuální hodnotu

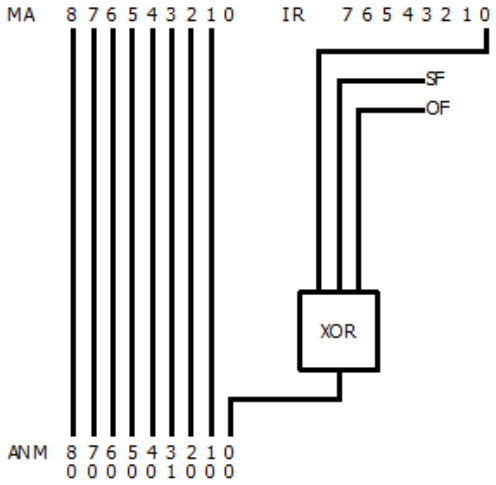
changed položka typu *bool* udávající, zda byla položka při posledním kroku simulace změněna

digits položka typu *int* udávající počet číslic potřebných pro výpis hodnoty

x položka typu *int* udávající horizontální souřadnici hodnoty v diagramu

y položka typu *int* udávající vertikální souřadnici hodnoty v diagramu

Obrázek 4.10: Widget diagramu multiplexoru následující adresy



4.4.18 Třída ACMGraphic

Třída je potomkem třídy *QWidget*. Reimplementuje metodu *paintEvent*. Slot *repaint* jejího rodiče *QWidget* je připojen na signály *addrChanged* a *dataChanged* třídy *DOPDriver*. Podoba widgetu je vidět na obrázku 4.10.

4.4.18.1 Metoda paintEvent

V metodě *paintEvent* dojde k vykreslení jednotlivých indexů cílové adresy mikroinstrukce, registru IR a případně k vykreslení dalších stavových signálů, jejichž hodnota je použita v aktuální podmínce. Ve spodní části diagramu jsou vypsány indexy adresy následující mikroinstrukce a jim odpovídající vypočítané hodnoty. Jednotlivé zdrojové bity jsou poté lomenými čarami propojeny s cílovými bity.

Kapitola 5

Výstupy práce

V rámci práce jsem vytvořil tyto produkty:

- tento text ve formátu LaTeX a PDF
- upravenou specifikace jazyka symbolických mikroinstrukcí [4] ve formátu PDF
- zdrojový kód aplikace v jazyce C++ přeložitelný na všech podporovaných platformách
- spustitelný program ve verzích pro operační systémy Windows a Linux v 32bitových i 64bitových variantách pro architekturu x86
- instalační příručku ve formátu LaTeX a PDF
- uživatelskou příručku ve formátu LaTeX, PDF a HTML
- UML diagramy použité v této práci ve formátu PlantUML a PNG

Všechny tyto produkty jsou umístěny na přiloženém mediu a jsou k dispozici na webových stránkách `http://sourceforge.net/simdop`.

Kapitola 6

Testování

Testování jsem rozdělil podle testovaných komponent na testování simulace, testování překladu a testování uživatelského rozhraní. Na závěr jsem ještě přidal několik komplexních testů pomocí složitých mikroinstrukcí. Většinu testů jsem prováděl na všech platformách.

6.1 Testování simulace

V rámci simulace procesoru jsem se zaměřil na správnou funkčnost instrukcí procesoru, správné vyhodnocování skoků a správné výsledky aritmeticko-logických operací.

6.1.1 Testování instrukcí procesoru

Na úvod jsem otestoval vybranou podmnožinu instrukcí procesoru se základním firmwarem DOPv3:

- LD a ST a jejich varianty
- ADD a její varianty
- RLC a RRC
- SWW, UCF a SCF
- JMP a všechny druhy podmíněných skoků
- CALL a RET
- INT a RETI

Všechny testy dopadly úspěšně a shodně s výsledky simulátoru SIMDOP.

6.1.2 Testování skoků

Skoky jsem testoval pomocí vlastního firmwaru. Jeho obsahem je pouze blok o osmi mikroinstrukcích a mikroinstrukce na tento blok přecházející. Postupně jsem této mikroinstrukci měnil kód skoku a poté zkoušel nastavit signály ovlivňující tento skok na všechny možné hodnoty.

Tento test byl poměrně časově náročný, nicméně vyloučil všechny chyby, které mohou nastat ve skocích. Všechny testy dopadly úspěšně a shodně s výsledky simulátoru SIMDOP.

Firmware pro otestování skoků najdete na přiloženém mediu jako *jump-test.mas*.

6.1.3 Testování aritmeticko-logické jednotky

Testování aritmeticko-logické jednotky jsem provedl pomocí vlastního firmwaru. V tomto firmwaru jsem postupně použil všechny operační znaky aritmeticko-logické jednotky s vhodně zvolenými hodnotami. Zároveň s tím jsem otestoval i správnou funkčnost příznaků.

Všechny testy dopadly úspěšně a shodně s výsledky simulátoru SIMDOP.

Firmware pro otestování aritmeticko-logické jednotky (včetně správných výsledků v komentářích) a použitý počáteční stav registrů najdete na přiloženém mediu jako *alu-test.mas* a *alu-test.sta*.

6.2 Testování překladu

Překladač jsem otestoval na správnost překladu a na odmítnutí neplatných vstupů.

6.2.1 Základní překlad

Překladač úspěšně přeložil základní firmware DOPv3 a fungoval s ním shodně se simulátorem SIMDOP. Navíc dle vyexportovaného log souboru jsou si oba přeložené soubory velmi podobné, rozdíly jsou pouze v pořadí bloků o jedné mikroinstrukci.

Vyexportované log soubory najdete na přiloženém mediu jako *DOPv3.log* (nový simulátor) a *DOPv3-old.log* (původní simulátor).

6.2.2 Testování mezních hodnot a neplatných vstupů

V této části jsem se zaměřil na testování mezních případů a neplatných vstupů a porovnával fukčnost svého simulátoru se simulátorem SIMDOP. Soubor s příklady najdete na přiloženém mediu jako *compiler-test.mas*.

6.2.2.1 Syntaktické testy

Syntaktické testy testují správnost rozpoznávání tokenů a jejich umístění. Otestoval jsem následující úkazy:

- ignorování prázdných znaků
- detekce identifikátoru a čísla
- dodržování nových řádků
- použití znaků mimo ASCII
- rozpoznání různých zápisů mikroinstrukce

Ve většině testů ukázal můj simulátor shodné chování se simulátorem SIMDOP. Vyjímkou je použití znaků mimo ASCII či znaků, které nejsou číslem ani písmenem, kdy SIMDOP při jejich výskytu mimo komentáře hlásí chybu. Můj simulátor je povoluje i v rámci identifikátoru, nicméně i s ohledem na zpětnou kompatibilitu je nedoporučuji používat.

Dále pak simulátor SIMDOP (narozdíl od mého) povoluje v rozporu se specifikací za složenou závorkou další znaky. V případě otevírací závorky případnou následující mikroinstrukci zpracuje, po zavírací závorce znaky ignoruje.

Zajímavostí je, že simulátor SIMDOP si neporadí s linuxovým zápisem nových řádek (*LF* namísto *CRLF*).

6.2.2.2 Sémantické testy

Sémantické testy se zaměřují na obsahovou správnost kódu. Otestoval jsem následující úkazy:

- neplatná hodnota aritmeticko-logické operace nebo kódu podmínky
- názvy signálů z množiny platných řídících signálů
- kolize zápisu na sběrnice
- velikost bloku odpovídající podmínkám
- přeplnění řídící paměti
- duplicita návěstí
- neexistující cílové návěstí

Výsledek většiny testů je takový, že se můj simulátor chová shodně se simulátorem SIMDOP. Jediným významným rozdílem (se kterým jsem počítal již v návrhu) je chování v případě, že podmínka skoku neodpovídá velikosti bloku. Simulátor SIMDOP takový blok umístí a pouze pokud by umístil do prostoru případného skoku jiný blok, tak zahlásí chybu. Můj simulátor překlad bloku menšího než podmínka vůbec nepovolí (i toho chování je benevolentnější než původní specifikace).

6.3 Testování uživatelského rozhraní

Uživatelské rozhraní jsem netestoval žádnými speciálními testy. Vzhledem k tomu, že jsem během předchozích testů napsal a odsimuloval několik vlastních firmwarů, předpokládám, že jsem odladil naprostou většinu chyb, které v uživatelském rozhraní byly.

6.4 Komplexní testy

Pro komplexní testy jsem použil několik složitějších instrukcí rozšiřujících základní firmware DOPv3. Jednalo se o tyto instrukce:

- určení pozice nejvyššího jedničkového bitu víceslovného operandu uloženého v hlavní paměti
- aritmetický posun vlevo víceslovného operandu uloženého v hlavní paměti
- aritmetický posun vpravo jednoslovného přímého operandu o volitelný počet pozic
- přičtení jednoslovného přímého operandu k víceslovnému operandu uloženému v paměti

Firmware s komplexními testy najdete na přiloženém mediu jako *complextest.mas*.

U všech instrukcí jsem se snažil o výpočty, s jejichž pomocí byla ověřitelná správná funkčnost co největšího množství příznaků. Všechny testy proběhly v pořádku a shodně se simulátorem SIMDOP.

Závěr

Cílem mé práce bylo vytvořit simulátor procesoru DOP, který při výuce nahradí oba současné simulátory.

Podařilo se mi vytvořit simulátor, který umožňuje pracovat se všemi vnitřními hodnotami procesoru a jeho pamětí. Tyto hodnoty lze ukládat do souborů pro pozdější načtení. Simulátor obsahuje také grafické diagramy aritmeticko-logické jednotky a multiplexoru podmínky.

Dále jsem do simulátoru implementoval editor zdrojového kódu s výrazňováním syntaxe, automatickým doplňováním kódu, pokročilým vyhledáváním a možností vytvářet breakpointy. Zdrojový kód je při editaci přímo překládán integrovaným překladačem, který poskytuje podrobné chybové výstupy a seznam návěstí s odkazy do zdrojového kódu. Také umožňuje export logu překladu a výsledného firmwaru v binární podobě.

Samotnou simulaci je možné provádět v různě dlouhých krocích, například jeden takt, do nejbližšího breakpointu, do změny či do konkrétní hodnoty programového čítače nebo do konkrétní adresy řídící paměti.

Simulátor je připraven pro běh pod operačními systémy Windows, Linux a Mac OS X, ale potencionálně i na dalších platformách podporovaných knihovnou Qt.

Ačkoli splnění vytyčeného cíle může plně ověřit až reálné nasazení ve výuce, jsem přesvědčen, že jsem cíl splnil. Program je multiplatformní, nenáročný, uživatelsky přívětivý a dle rozsáhlého testování i bez problémů funkční. Oproti dřívějším simulátorům poskytuje každá část toho mého něco navíc a značně usnadňuje simulaci. Díky tomu studentům zpříjemní a urychlí implementaci nové instrukce do procesoru DOP.

Literatura

[1] Krumphanzl, A.: *SIMDOP – simulátor procesoru pro výukové účely*. Diplomová práce, ČVUT FEL, 2000.

[2] Karas, L.: *Simulátor procesoru DOP*. Bakalářská práce, ČVUT FEL, 2009.

[3] Bečvář, M.: *DOP Block diagrams*. 2005.

[4] Bečvář, M.: *Syntaxe jazyka symbolických mikroinstrukcí (*.mas) a jeho použití*. 2003.

[5] The Qt Company Ltd.: Qt Documentation. online, 2015, http://doc.qt.io/.

[6] Pluháček, A.; Bečvář, M.: *Procesor DOP – v3 (základní charakteristika)*. 2004.

[7] Pluháček, A.; Daněček, J.; Bečvář, M.: *Strojový kód procesoru DOP-v3*. 2006.

[8] Pluháček, A.; Bečvář, M.; Daněček, J.: *DOP – historie a publikace*. 2010.

[9] Pluháček, A.: *DOP-v3 registers*. 2009.

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

FPGA Field-programmable gate array

IDE Integrated development environment

UML Unified modeling language

PDF Portable document format

CD Compact disk

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ... stručný popis obsahu CD
- `exe` ... adresář se spustitelnou formou implementace
  - `simdop_windows_32.exe` ... spustitelný soubor pro 32bitové Windows
  - `simdop_windows_64.exe` ... spustitelný soubor pro 64bitové Windows
  - `simdop_linux_32` ... spustitelný soubor pro 32bitový Linux
  - `simdop_linux_64` ... spustitelný soubor pro 64bitový Linux
- `src`
  - `impl` ... zdrojové kódy implementace
  - `thesis` ... zdrojová forma práce ve formátu LaTeX
  - `doc` ... zdrojová forma příruček ve formátu LaTeX
- `text` ... text práce
  - `thesis.pdf` ... text práce ve formátu PDF
  - `thesis.ps` ... text práce ve formátu PS
- `doc`
  - `installation-guide.pdf` ... instalační příručka ve formátu PDF
  - `user-guide.pdf` ... uživatelská příručka ve formátu PDF
  - `mas.pdf` ... specifikace jazyka symbolických mikroinstrukcí
  - `mas-changelog.txt` ... seznam úprav ve specifikaci
- `test` ... testovací soubory