VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

WEBOVÝ SIMULÁTOR PROCESORU ARCHITEKTURY MIPS

WEB-BASED MIPS SIMULATOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MATĚJ HŮLEK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VAŠÍČEK ZDENĚK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Hůlek Matěj**
Program: Informační technologie
Název: **Webový simulátor procesoru architektury MIPS**
Web-Based MIPS Simulator
Kategorie: Web

Zadání:

1. Seznamte se s problematikou řetězeného zpracování instrukcí, s architekturou MIPS a detailně s principem činnosti řetězeného procesoru DLX. Dále se seznamte s architekturou RISC-V a odlišnostmi oproti architektuře MIPS.
2. Navrhněte webový simulátor zřetězeného procesoru, který bude vhodně demonstrovat činnost procesoru na úrovni zřetězené linky pro potřeby výuky. Vstupem nechť je kód v jazyku symbolických instrukcí.
3. Zpracujte studii na téma uvedené v prvním bodě zadání.
4. Simulátor implementujte s využitím jazyka Typescript a vhodného frameworku pro UI (např. VueJS, React, apod.). Implementaci proveďte tak, aby byla vhodně oddělena simulační a prezentační vrstva.
5. Vytvořte sadu demonstračních programů, vyhodnoťte parametry navrženého řešení a diskutujte možná rozšíření.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Vašíček Zdeněk, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Práce se zaobírá simulací zřetězeného procesoru architektury MIPS za účelem demonstrace činnosti procesoru na úrovni zřetězené linky pro účely výuky. Cílem práce je tedy implementovat webovou aplikaci, která bude umožňovat uživateli vložit kód symbolických instrukcí a vhodným způsobem demonstrovat činnost simulačního jádra.

Abstract

The work deals with the simulation of a pipelined processor of the MIPS architecture in order to demonstrate the activity of the processor at the level of a pipeline for teaching purposes. The aim of this work is to implement a web application that will allow the user to enter the code of symbolic instructions and demonstrate the operation of the simulation core in a suitable way.

Klíčová slova

MIPS architektura, webový simulátor, zřetězené zpracování instrukcí, pipelining

Keywords

MIPS architecture, web simulator, instruction processing, pipelining

Citace

HŮLEK, Matěj. *Webový simulátor procesoru architektury MIPS*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vašíček Zdeněk, Ph.D.

Webový simulátor procesoru architektury MIPS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením doc. Vašíčka Zdeňka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Matěj Hůlek
10. května 2022

Obsah

Kapitola 1

Úvod

Člověk se během života učí pořád něčemu novému a je v našem zájmu, aby byl proces učení co nejefektivnější, tedy aby člověk správně pochopil a co nejjednodušeji pojmul dané učivo. K tomu existuje celá řada různých technik, zdrojů a způsobů. Tak je tomu i při výuce architektur a činnosti procesorů. Zde je klíčové, aby studenti správně pochopili základní koncepty a funkcionality na kterých jsou postaveny i dnešní procesory. Toto se dá vyučovat mnoha způsoby, ale asi nejvhodnějším by byl simulátor, na kterém by se dala demonstrovat funkcionalita procesoru. Tomu se věnuje tato práce.

První část bakalářské práce se zaměřuje na seznámení s problematikou a funkcionalitou architektury procesoru. To, jakým způsobem pracují jednotlivé části, které chceme simulovat.

Druhá část se věnuje návrhu samotného simulátoru a použitým konstrukcím, které by měly sloužit pro jednoduchou implementaci programu.

V třetí části se věnuju popisu vlastní implementaci simulátoru MIPS procesoru, která využívá koncepty popsané v návrhu implementace.

Kapitola 2

Moderní zřetězené architektury

Tato kapitola se věnuje vysvětlení základních pojmů používané v této práci, funkcionalitě klíčových částí simulovaného procesoru a dalším nezbytným informacím pro tuto práci. Kromě popisu simulovaného procesoru jsou zde popsány i alternativní architektury.

2.1 Architektura MIPS

MIPS je jednou z nejvíce populárních architektur procesorů. Proto je často vyučována na univerzitách a technických školách a z toho důvodu je perfektní pro simulaci. [11]

MIPS je architektura založená na registrech, což je malá a velmi rychlá paměť, která je přístupná procesoru pro provádění operací. MIPS podobně jako většina procesorů funguje na principu, kdy z paměti nahraje data do registrů, provede nad nimi operace a výsledek uloží do paměti. Každý registr MIPS architektury má velikost 32 bitů.

2.1.1 Registry

MIPS obsahuje celkem 64 registrů. 32 registrů je vyhrazeno pro práci s celočíselnými hodnotami, známé též jako univerzální registry. Dalších 32 registrů pro práci s pohyblivou řádovou čárkou. Názvy registrů vždy začínají znakem $. Registry pro práci s pohyblivou řadovou čárkou jsou $f0 až $f31. Univerzální registry mají své jméno a číslo, jejich výpis je v tabulce 2.1.

Tabulka 2.1: Univerzální registry architektury MIPS

Číslo	Název	Popis a využití
$0	$zero	vždy 0 (pouze pro čtení)
$1	$at	rezervováno pro assembler
$2-$3	$v0-$v1	návratové hodnoty funkce
$4-$7	$a0-$a3	argumenty funkce
$8-$15	$t0-$t7	pomocné výpočty, bez zachování
$16-$23	$s0-$s7	chráněné hodnoty, nutno zachovat
$24-$25	$t8-$t9	pomocné výpočty, bez zachování
$26-$27	$k0-$k1	rezervováno pro operační systém
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$fp	frame pointer
$31	$ra	návratová adresa

Při psaní programu lze použít kterýkoliv z těchto registrů, ale neměla by být porušena konvence volání. Ta specifikuje, jak by s registry měla pracovat volaná funkce a jak volající. Zabezpečuje, aby volaná funkce nepřepsala hodnoty registrů používané volajícím.

Dočasné registry jsou určeny pro pomocné výpočty a mohou být volně využívány a přepisovány volanými funkcemi. Chráněné registry naopak slouží pro bezpečné uložení hodnot, které by neměly být přepsány volanými funkcemi. Pokud chce volaná procedura používat tyto registry, měla by hodnoty v těchto registrech uložit a na konci procedury opět obnovit. Nejjednodušší je ale s těmito registry vůbec nepracovat.

Registr $zero je statický registr, který obsahuje hodnotu 0. Tento registr nemůže být použit jako cíl pro uložení hodnoty, ale dá se použít pro vytvoření dalších operací.

Dále se zde nacházejí registy bez možného přímého přístupu programátora. Jedním z těchto registrů je například registr PC (Program Counter - čítač instrukcí), jež obsahuje adresu další instrukce, která se má provést. Jediná instrukce schopná číst hodnotu registru PC je instrukce jal. Ta se používá při volání funkcí a ukládá hodnotu PC+4 do registru $ra.

MIPS obsahuje i dva speciální registry $hi a $lo, které slouží pro uložení výsledku dělení a násobení. S těmito registry nelze přímo pracovat jako s univerzálními registry. Jediný způsob, jak přistoupit k hodnotám uloženým v těchto registrech je pomocí instrukcí `mfhi` a `mflo`. Tyto instrukce přesunou hodnotu uloženou v $hi nebo $lo do cílového univerzálního registru.

2.1.2 Instrukce

Všechny MIPS instrukce jsou kódovány pomocí slov o délce 32 bitů a spadají do jednoho ze tří formátů [15]. Prvních 6 bitů každé instrukce (tzv. `opcode`) specifikuje konkrétní instrukci. Význam ostatních bitů závisí na formátu intrukce.

R formát

R formát je určený pro kódování operandu aritmetických a logických instrukcí.

Tabulka 2.2: R formát instrukce

31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
opcode	rs	rt	rd	shamt	funct

Všechny instrukce tohoto formátu mají opcode roven nule. `Rs` a `Rt` jsou zdrojové registry a `Rd` slouží jako cílový registr. Pole `shamt` specifikuje velikost posunu, slouží instrukcím provádějící bitový posun. `funct` (funkční hodnota) specifikuje typ aritmetické/logické operace, např. hodnota 32 má význam operace součtu.

Tabulka 2.3: Příklad R instrukcí

instrukce	opcode	rs	rt	rd	shamt	funct	příklad
add	0	2	3	1	0	32	add $1, $2, $3
or	0	2	3	1	0	37	or $1, $2, $3
sll	0	0	2	1	8	0	sll $1, $2, 8

I formát

Formátem I jsou kódované veškeré instrukce, které vyžadují kromě případného cílového registru `rt` a zdrojového registru `rs` i 16-bitovou přímou hodnotu `imm` ve dvojkovém doplňku, která je součástí instrukce.

Tabulka 2.4: I formát

31 - 26	25 - 21	20 - 16	15 - 0
opcode	rs	rt	imm

Mezi tyto instrukce patří i některé aritmetické a logické operace:

Tabulka 2.5: Příklad aritmetických a logických instrukcí I formátu

instrukce	opcode	rs	rt	imm	příklad
addi	8	2	1	20	addi $1, $2, 20
slti	10	1	2	20	slti $1, $2, 20
xori	14	2	1	20	xori $1, $2, 20
lui	15	0	1	100	lui $1, 100

Instrukce pro podmíněné skoky:

Tabulka 2.6: Příklad instrukcí formátu I pro podmíněné skoky

instrukce	opcode	rs	rt	imm	příklad
beq	4	1	2	25	beq $1, $2, 100
bne	5	1	2	25	beq $1, $2, 100

Ty provedou skok pouze v případě, že je splněna podmínka. Adresa cíle se vypočítá pomocí hodnoty `imm`, která funguje jako offset. Tento offset se znaménkově rozšíří na 32 bitů a následně se provede logický posun o dvě pozice doleva. Nakonec se k hodnotě ještě přičte hodnota 4.

$$destination\_address = PC + 4 + (signext(imm) << 2)$$

Instrukce pro práci s pamětí:

Tabulka 2.7: Příklad instrukcí formátu I pro práci s pamětí

instrukce	opcode	rs	rt	imm	příklad
lw	35	2	1	100	lw $1, 100($2)
sw	43	2	1	100	sw $1, 100($2)
lh	33	2	1	100	lh $1, 100($2)

Ty buďto čtou data z paměti (`lw, lh, lb`) nebo ukládají data do paměti (`sw, sh, sb`). Adresa paměti, se kterou se pracuje, se vypočítá jako součet hodnoty registru `rs` a hodnoty `imm`. registr `rt` se v případě čtení z paměti použije jako cílový nebo se v případě uložení hodnota v registru `rt` načte do paměti.

J formát

Instrukce formátu J (`j, jal`) slouží k zakódování instrukcí, které slouží pro předání toku řízení do jiné části programu.

Tabulka 2.8: J formát

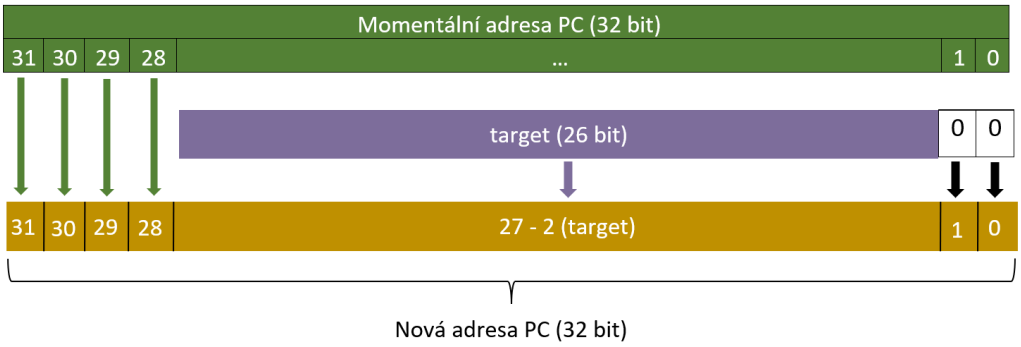
31 - 26	25 - 0
opcode	target

Mezi tyto instrukce patří:

Tabulka 2.9: Příklad J instrukcí

instrukce	opcode	target	příklad
j	2	2500	j 10000
jal	3	2500	jal label

32 bitová adresa na kterou se provede skok se skládá z vrchní 4 bitů aktuální adresy, 26 bitové hodnoty `target` a dvou nul.



Obrázek 2.1: Výpočet cílové adresy skoku.

Pseudoinstrukce

Pseudoinstrukce jsou instrukce, které jsou validní MIPS asembler instrukcemi, ale nemají svojí přímou hardwarovou implementaci. Jejich účelem je zjednodušit práci programátorovi při psaní kódu. Při překladu programu jsou tyto instrukce přeloženy na ekvivalentní hardwarově implementované instrukce.

Tabulka 2.10: Příklad pseudoinstrukcí

pseudoinstrukce	popis	příklad	přeloženo
move `<cíl>, <zdroj>`	Nahraje obsah jednoho registru do druhého	move $t0, $s0	addu $t0, $zero, $s0
la `<cíl>, <návěští>`	Nahraje adresu návěští do registru	la $s0, label	lui $at, 0x1001 ori $s0, $at, 16
blt `<r1>, <r2>, <label>`	Provede skok pokud je r1 < r2	blt $t0, $t1, label	slt $at $t0, $t1 bne $at, $zero, label

Systémová volání

Systémová volání slouží k vyžádání obslužné rutiny od systému. To se dá využít například pro získání uživatelského vstupu nebo vypsaní zprávy pro uživatele. K tomu, abychom mohli použít konkrétní obslužnou rutinu musíme nahrát její číslo do registru `$v0`. Případné parametry musíme nahrát do registru `$a0` nebo `$a1`. Nakonec vyžádáme obslužnou rutinu pomocí instrukce `syscall`.

Příklad existujících systémových volání:

Tabulka 2.11: Příklad systémových volání

služba	kód (hodnota v $v0)	argumenty	popis
print_int	1	$a0	Vypíše hodnotu uloženou v registru $v0
print_string	4	$a0	Vypíše řetězec, který začíná na adrese uložené v registru $a0
read_int	5		Přečtě číslo ze vtupu a uloží jej do registru $v0
read_string	8	$a0, $a1	Přečte řetězec o délce `$a1` a uloží jej na adresu `$a0`
sbrk	9	$a0	Alokuje `$a0` bajtů paměti a vrátí její adresu v $v0

Příklad systémového volání pro výpis hodnoty 36 na výstup:

```
li $v0, 1
li $a0, 36
syscall
```

2.1.3 Assembler MIPS

MIPS asembler kód má několik základních vlastností, které jsou popsány v této části.

Komentáře

Komentáře začínají znakem `#`. Veškerý obsah za tímto znakem až po konec řádku je lexikálním analyzátorem ignorován.

Návěští

Návěští velmi usnadňuje psaní kódu. Slouží pro odkazování části kódu, tím že mu dáme jméno. Díky tomu můžeme jednoduše vytvářet cykly, podmínky a skoky. Podmínkou je, že název návěští nesmí být stejný jako je název MIPS instrukce.

MIPS návěští fungují prakticky stejně, jako návěští v jazyce C.

Sekce

Sekce slouží k rozdělení asembler programu na instrukce a data. Program se načítá do paměti a procesor musí vědět, kde se nachází vstupní bod programu, odkud má začít vykonávat instrukce a posouvat čítač instrukcí. Jelikož data a instrukce můžou být v kódu různě rozmístěna, je potřeba je od sebe odlišit. K tomu slouží sekce.

V MIPS existují 2 hlavní sekce: `text` a `data`. Text sekce obsahu asembler kód a začíná řetězcem `.text`. Data sekce obsahuje naše data a začíná řetězcem `.data`.

Direktivy

Datové direktivy se nacházejí v `.data` sekci a slouží k inicializaci paměti pro naše data. Syntax je následující:

[název:] direktiv inicializátor [, inicializátor]...

`Název` slouží k tomu abychom se mohli v kódu k dané paměti jednoduše odkazovat. Je to nepovinná položka a zapisuje se stejně jako návěští.

`direktiv` je množství paměti, kterou chceme alokovat. Například `.word` pro 32 bitů, nebo `.byte` pro alokaci jednoho bajtu.

`inicializátor` je počáteční hodnota. Pokud jsou hodnoty odděleny čárkou, tak se alokuje pole hodnot.

Příklad použití:

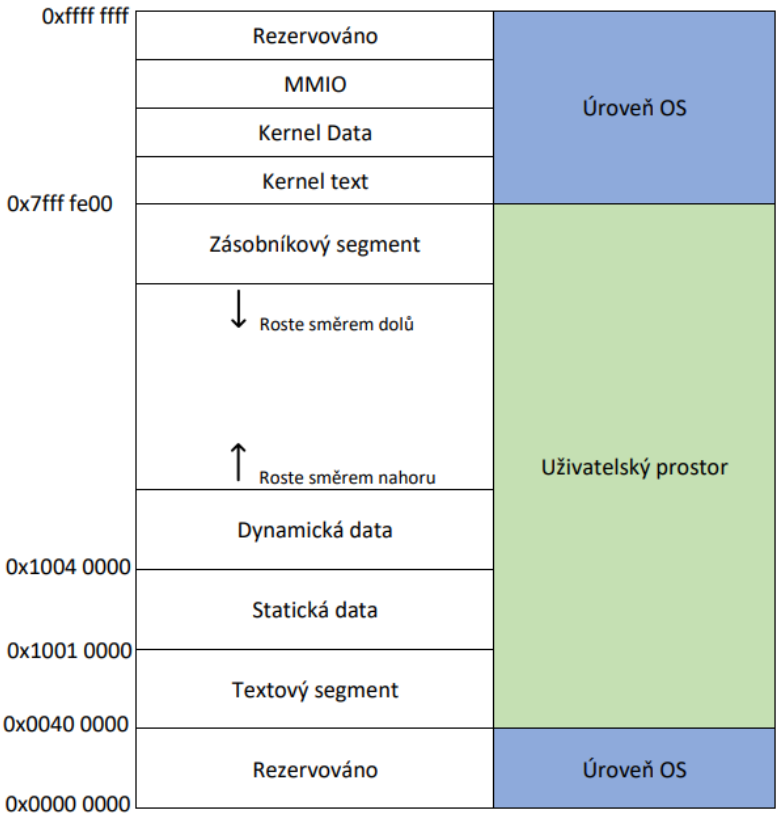
```
.data
    num: .word 42
    arr_word: .word 0, 1, 2, 3
    arr_char: .byte 'a', 'b'
    text: .asciiz "lorem ipsum"
```

Možnosti zápisu registrů

Při zápisu registrů lze použít, jak číslo registru, tak i jeho název. např. registr `$4` lze zapsat i jako `$a0`.

2.1.4 Organizace paměti

MIPS program se ukládá do paměti odkud se při exekuci i čte. Proto musíme znát strukturu paměti, aby bylo možné kód uložit na adresu, kde procesorové jádro po restartu začíná. MIPS počítače dokážou adresovat až 4 GB paměti, od adresy `0x00000000` až po adresu `0x7fffffff`. Paměť pojme jak programové instrukce, tak i data. Běží-li nad MIPS operační systém, pak uživatel nemá přistup k celému adresovému prostoru, ale pouze jeho části, zbytek adresového prostoru je přiřazeno operačnímu systému. [4]



Obrázek 2.2: Typické rozložení paměti v případě použití OS.

- Textový segment - (Adresa 0x0040 0000 - 0x1000 0000) Prostor pro uložení uživatelského kódu, každá instrukce je uložena jako `word` (4 bajty).
- Statická data - (Adresa 0x1001 0000 - 0x1004 0000) Prostor pro uložení dat z datového segmentu (`.data`) programu. Velikost elementů je přiřazena během sestavení programu a nemůže být dále změněna při běhu programu.
- Dynamická data - (Adresa 0x1001 0004 - dokud nedosáhne zásobníkového segmentu, roste nahoru) Prostor pro dynamická data, která se alokují po čas běhu programu.
- Zásobníková data - (Adresa 0x7fff fe00 - dokud nedosáhne bloku s dynamickými daty, roste dolů) Zásobník, kam může uživatelský program ukládat dočasná data. Například pro uložení hodnot registrů volanou funkcí.

Zarovnání paměti

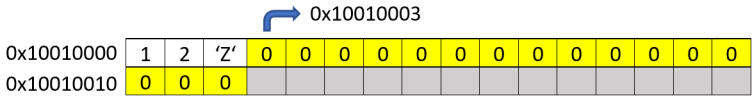
Jednotlivé prvky v paměti musí být zarovnány na správné místo. Základní pravidla pro zarovnání hodnot v paměti jsou:

- Hodnoty o velikosti jednoho slova (32 bitů) musí začínat na adrese dělitelné čtyřmi.
- Hodnoty o velikosti půlky slova (16 bitů) musí začínat na adrese dělitelné dvěma.

- Hodnoty o velikosti jednoho bajtu (8 bitů) mohou začínat kdekoli.

Jakýkoli pokus o čtení nebo zápis do paměti, který porušuje tyto pravidla způsobí chybu. V `.data` sekci se dá použít direktiv `.align n`, pomocí kterého se přepíší výchozí pravidla zarovnání pro další proměnnou. Ta bude zarovnána na adresu, která je dělitelná číslem 2^n. Tento direktiv se hodí použít v kombinaci s direktivem `.space n`, který alokuje `n` bajtů paměti. Těchto `n` bajtů ovšem může začínat kdekoli a to by byl problém. V případě, kdy bychom odtud chtěli číst jinak něž po bajtech, by mohlo dojít k porušení pravidla pro zarovnání. Tomuto problému se dá předejít právě použitím direktivu `.align`.
Příklad paměti bez použití direktivu `.align` je demonstrován na obrázku 2.3.

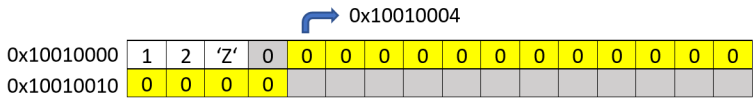
```
.data
var1: .byte 1, 2, 'Z'
var2: .space 16
```



Obrázek 2.3: Zarovnání paměti bez `.align`.

V tomto případě by paměť nebyla zarovnána a pokus o čtení nebo zápis z `var2` po slovech či půl-slovech by způsobil chybu.
Příklad paměti s použitím direktivu `.align` je demonstrován na obrázku 2.4.:

```
.data
var1: .byte 1, 2, 'Z'
.align 2
var2: .space 16
```



Obrázek 2.4: Zarovnání paměti s `.align`.

V tomto případě byla paměť zarovnána na adresu dělitelnou čtyřmi a je možné číst a zapisovat do `var2` libovolně. bajt mezi `var1` a `var2` je přeskočen a nevyužit.

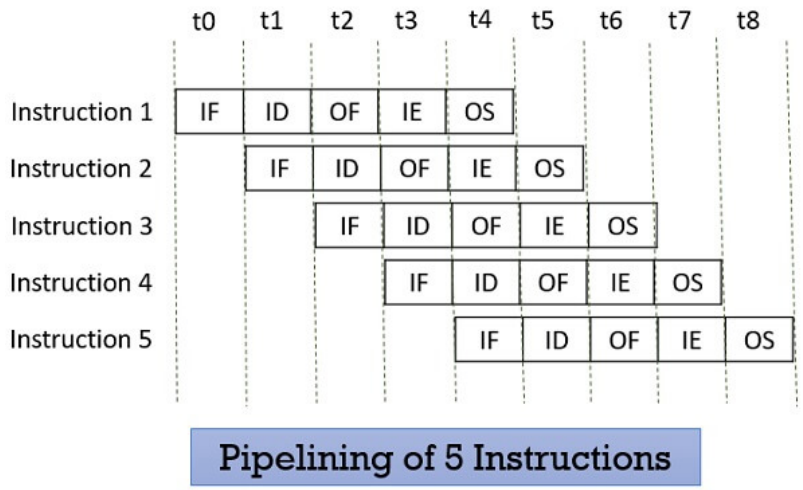
2.1.5 Hardwarová realizace

Architektura MIPS (obrázek 2.6) patří mezi tzv. skalární architektury. To znamená, že je logika procesoru rozdělena na několik fází, které se provádějí paralelně. Základním rysem paralelismu na úrovni instrukcí, kterého se dosahuje za pomocí tzv. zřetězeného zpracování instrukcí je, že se v jednom taktu načte maximálně jedna instrukce a dokončí se maximálně jedna instrukce. V každé fázi se nachází maximálně jedna instrukce.

Zřetězenou linku MIPS tvoří pět na sobě nezávislých stupňů:

- IF: Instruction fetch - načtení instrukce z paměti
- ID: Instruction decode - dekódování instrukce a načtení registrů
- EX: Execute - provedení instrukce
- MEM: Memory access - čtení z paměti
- WB: Write back - zápis výsledku do paměti

Provádění instrukce je tedy rozděleno do 5 fází. Když je první instrukce dekódována, tak je druhá instrukce načtena. Pakliže je pipeline zaplněna, tak se provádí 5 různých akcí paralelně. Toto je vyobrazeno na obrázku 2.5.



Obrázek 2.5: Ukázka zřetězeného zpracování instrukcí. [13]

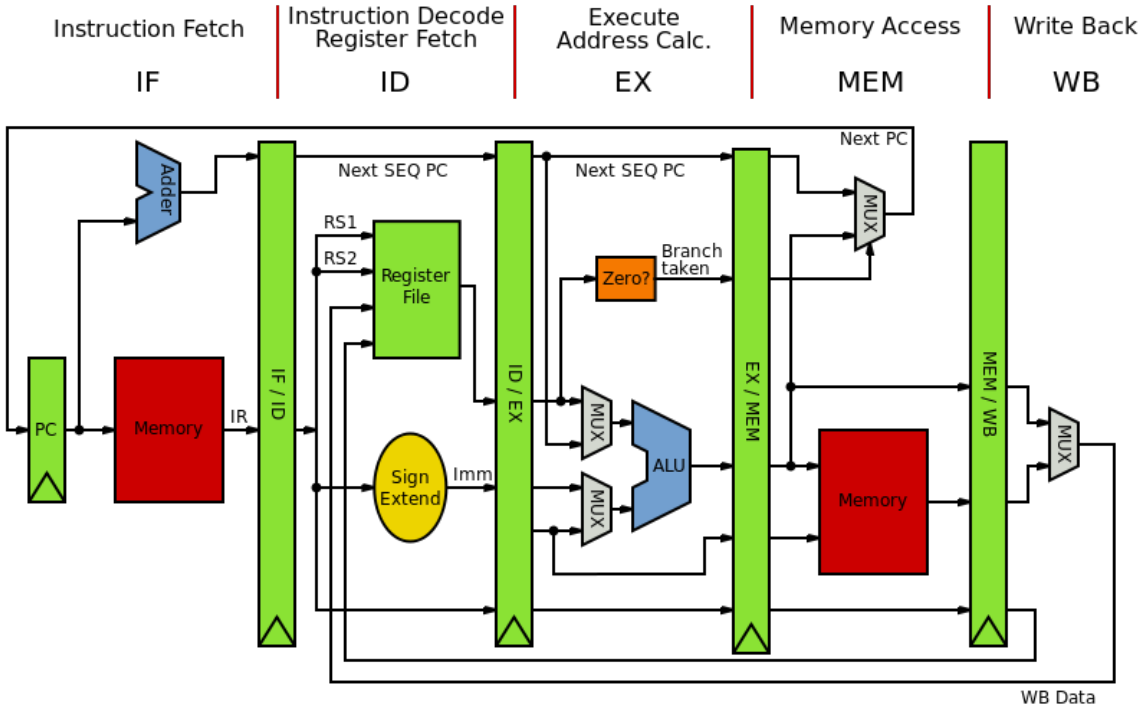
Základní vlastnosti zřetězeného zpracovní:

- Pipelining nepomáhá latenci, pouze pomáhá propustnosti celé pracovní zátěže. To je způsobeno tím, že je více instrukcí prováděno zároveň. Latence se však může zhoršit kvůli přidané režii z rozdělení výpočtu na oddělené kroky.
- Rychlost pipeline je omezena nejpomalejší fází. Důvodem k tomuto je, aby mohla fáze provádět další instrukci, tak předchozí musí být dokončena.
- Více úloh se provádí současně.
- Využívá paralelismus mezi instrukcemi v sekvenčním toku instrukcí.

Vyrovnávací paměti

Mezi jednotlivými fázemi se nacházejí vyrovnávací paměti (registry pro uložení konkrétních položek). Každá fáze přijímá data z vyrovnávací paměti, zpracovává je a zapisuje do další vyrovnávací paměti. Například, během cyklu 4, jsou informace ve vyrovnávacích pamětech následující:

- V paměti IF/ED se nachází instrukce 4, které byla načtena ve fázi 4.
- V paměti ID/EX se nachází dekódovaná instrukce 3 a zdrojové operandy pro tuto instrukci.
- V paměti EX/MEM se nachází výsledek instrukce 2 a informace potřebné pro zápis této instrukce.
- V paměti MEM/WB se nachází data načtená z paměti pro instrukci 1. Pro aritmeticko logické operace se zde nachází výsledek z fáze `EX`.



Obrázek 2.6: MIPS pipeline architektura [6]

Hazardy

V případě sub-skalárního procesoru je rozpracována jedna instrukce a další se začne zpracovávat až po jejím dokončení. V případě zřetězeného procesoru tento předpoklad neplatí, neboť je v jednom okamžiku zpracováno více instrukcí, každá v jiné fázi zpracování. Tento způsob zpracování vede ke vzniku tzv. hazardů, tj. že instrukce pro své provedení vyžaduje výsledek předchozí instrukce, který ještě nebyl uložen. Jednotka pro odhalování hazardů se jmenuje `Hazard unit`. Rozlišujeme tři hlavní typy hazardů dle toho, o jakou skupinu instrukcí se jedná.

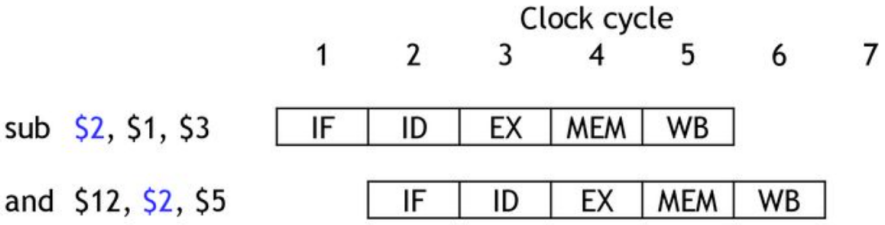
- Strukturální hazardy - hardware nepodporuje danou kombinaci instrukcí. Současný přístup k prostředku z více stupňů pipeline.
- Datové hazardy - instrukce nemá k dispozici data na vykonání, neboť závisí na výsledku předchozí instrukce. Provede-li se v tuto chvíli výpočet, tak dostane chybný výsledek.
- Řídící hazardy - nutno učinit rozhodnutí před vykonáním instrukce u podmíněných skoků, které jsou zpracovány až ve 3. kroku.

Datové hazrdy

Obrázek 2.7 obsahuje ukázkovou situaci datového hazardu. Instrukce `sub` ukládá výsledek do registru `$2`, hodnotu tohoto registru ale pro výpočet používá následující instrukce `and`. Použití výsledku instrukce `sub` v následující instrukci způsobuje hazard, jelikož se výsledek operace zapisuje do registru, až v 5. cyklu. Ovšem instrukce `and` s ním počítá již ve 4. cyklu. Bez ošetření by tato situace způsobila chybný výsledek druhé instrukce.

Forwarding

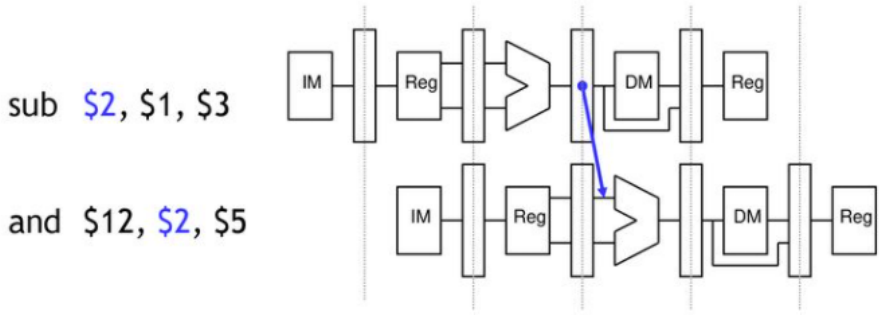
Jedním z efektivních způsobů řešení je tzv. forwarding. Jedná se o koncept zpřístupnění dat vstupu ALU pro následující instrukce i přes to, že se instrukce ještě nedostala do fáze `WB`, kde by se výsledek do registru uložil.



Obrázek 2.7: Posloupnost isntrukcí způsobující datový hazard. [2]

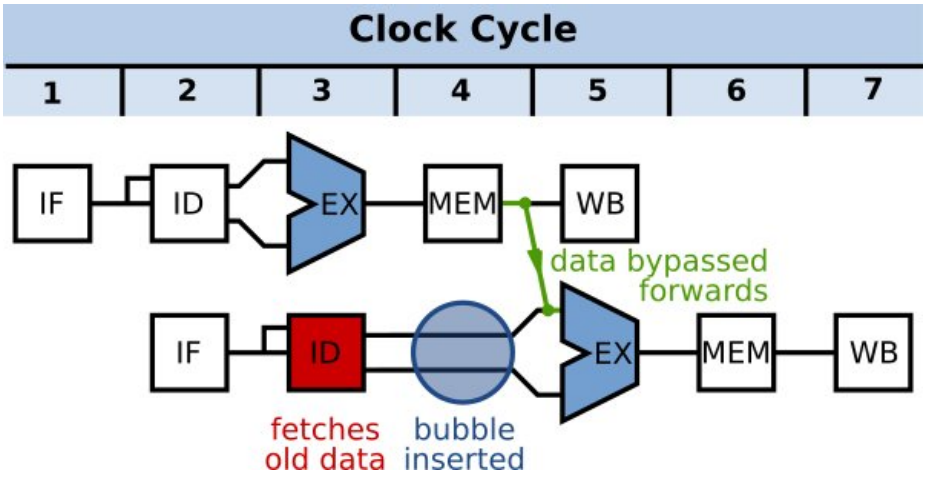
Vzhledem k tomu, že `EX/MEM` vyrovnávací paměť již obsahuje výsledek faze `EX`, můžeme jí jednoduše přeposlat dalším instrukcím.

Na obrázku 2.8 je vidět, jak je v cyklu 4 možné získat hodnoty $1 - $3 z `EX/MEM` paměti a předat jí ALU při provádění instrukce `and`.



Obrázek 2.8: Řešení situace z obrázku 2.7 pomocí forwardingu. [2]

Jedním z případů, kde forwarding nemůže pomoci při odstranění hazardů je, když se instrukce snaží číst z registru, do kterého předchozí instrukce načítá hodnotu z paměti (například instrukcí `lw`). Data se z paměti čtou až ve fázi `MEM`. Následující instrukce by se ale už měla nacházet ve fázi `EX`. Takovýto případ musíme řešit pozastavením pipeline, aby se data stihla načíst z paměti (obrázek 2.9).



Obrázek 2.9: Příklad pozastavení pipeline. [5]

Dalším případem, kde nelze forwarding použít, je v případě registrů $hi a $lo. Ty jsou totiž fyzicky odděleny od univerzálních registrů. Proto situaci, kde by instrukci `div` následovala instrukce `mfhi`, nešlo řešit forwardingem. Buďto by se musela pozastavit pipeline nebo by se muselo změnit pořadí instrukcí, aby k hazardu vůbec nedošlo.

Alternativním způsobem řešení datových hazardů je pozastavit pipeline a počkat, až se výsledek předchozí instrukce zapíše do cílového registru ve fázi `WB`. Tento způsob je ovšem velmi neefektivní.

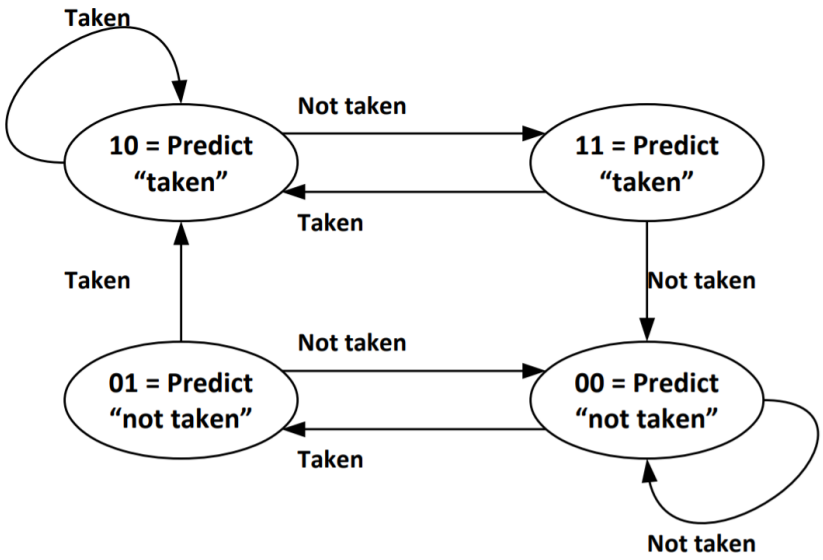
Řídící hazardy

Dalším hazardem pipelingu jsou skoky. Instrukce se načítají každý cyklus, dokud není skok dokončen. Dokud ovšem není instrukce skoku provedena, tak nevíme, odkud získat další instrukci. Možnosti řešení:

- Zpozdit pipeline do dokončení skoku - Takové řešení je velmi neefektivní, ale jednoduché na implementaci.
- Předpokládat, že se skok neprovede - pipeline prostě čte další instrukce a pokud se skok provede, tak vyprázdní pipeline. Redukuje zpoždění při provádění skoku.
- Predikce skoku - pokusit se odhadnout, zdali se skok provede nebo ne.

2-bitová prediktor skoků

Tento prediktor mění predikci pouze u dvou po sobě následujících chybných predikcích. V predikční vyrovnávací paměti jsou udržovány dva bity a existují čtyři různé stavy. Dva stavy odpovídající převzatému stavu a dva odpovídající nepřevzatému stavu. Stavový diagram prediktoru je uveden níže:



Obrázek 2.10: Stavový diagram 2-bitového prediktoru skoků. [11]

Vyhodnocení různých operací

Aritmetické a logické instrukce (`and, or, sll...`) jsou vyhodnocené ve fázi `EX` a jejich výsledek je zapsán do cílového registru ve fázi `WB`. Ve fázi `MEM` nic neprovádí.

Instrukce pro práci s pamětí (`lw, sw, lh...`) se vykonají ve fázi `MEM` a v případě čtení z paměti zapíší výsledek do cílového registru ve fázi `WB`. Ve fázi `EX` se nic neprovádí.

U instrukcí podmíněného skoku záleží na implementaci. Buďto může být jejich podmínka vyhodnocena ve fázi `EX` a skok případně proveden na začátku dalšího cyklu.

Druhou možností je skok provést již ve fázi `ID`. V takovém případě, by se musela přidat komponenta, která by porovnala načtené hodnoty z registrů. Pakliže nevzniká datový hazard mezi instrukcí podmíněného skoku a jinou instrukcí v pipeline, je tato možnost efektivnější.

Instrukce skoku jsou vyhodnoceny již ve fázi `ID`. Instrukce `j` může provést skok bez omezení. Instrukce `jal` provede skok na adresu a nastaví hodnotu registru `$ra` na hodnotu PC + 4. Nastavení registru `$ra` se provede taky ve fázi `ID`. To může zapříčinit hazard, kdy instrukce předcházející instrukci `jal`, taky mění hodnotu registru `$ra`. Příklad takového hazardu:

```
.text
add $ra, $zero, $zero
jal label
...
```

V takovém případě by se v cyklu 3 nastavila hondota registru `$ra` pomocí instrukce `jal` ve fázi `ID`. Ve stejnou chvíli by se instrukce `add` nacházela ve fázi `EX`. Dva cykly nato, by se instrukce `add` nacházela ve fázi `WB`, kde by se nastavila hodnota registru `$ra` na 0. Tento problém je nutné řešit buď odstraněním zbytečné instrukce nebo pozastavením pipeline.

Instrukce `jr` provede skok na adresu uloženou v registru. Zde opět záleží na implementaci. Skok může být proveden již ve fázi `ID` stejně jako instrukce `j` a `jal`. V takovém případě

je vhodné, aby nevznikal datový hazard, jinak by bylo nutné pozastavit pipeline, jelikož forwarding ve fázi `ID` nefunguje.

Druhou možností je provést skok pomocí instrukce `jr` až ve fázi `MEM`. V takovém případě, by se dal použít forwarding, ale tento způsob není tak efektivní.

2.2 Architektura DLX

DLX (Deluxe) je RISC architektura založená na principech MIPS architektury. Jedná se prakticky zjednodušenou verzi MIPS procesoru. Protože byl DLX určen především pro výukové účely, je architektura DLX široce používána v kurzech počítačové architektury na univerzitní úrovni. Hlavním rozdílem oproti MIPS je syntax DLX Assembler kódu.

2.3 Architektura RISC-V

RISC-V [10] je otevřená instrukční sada (ISA), která staví na principech RISC. Cílem RISC-V je kompletně otevřená ISA, která není omezená na konkrétní technologii (TTL, FPGA, ASIC...). Její využití je zcela zdarma, takže není třeba platit za licenční poplatky. RISC-V ISA je rozdělena na moduly, které jsou vzájemně nezávislé a nesdílí operační kódy.

Existují čtyři varianty MIPS ISA:

1. `RV32I` - 32bitová varianta.
2. `RV32E` - Zmenšená 32 bitová varianta určená pro vestavěné systémy. Nejvýraznějším rozdílem oproti variantě RV32I je snížený počet registrů z 32 na pouhých 16.
3. `RV64I` - 64bitová varianta.
4. `RV128I` - 128bitová varianta.

Instrukční sada RISC-V je velmi minimalistická. Obsahuje pouze 49 instrukcí pro 32bitový procesor a nabízí přes 20 možných rozšíření, která se dělí na 2 kategorie:

1. `Standardní rozšíření`, která jsou obecně použitelná. Použití těchto rozšíření není v konfliktu se základní ISA ani s ostatními standardními rozšířeními. Mezi standardní instrukční rozšíření patří např. rozšíření pro násobení a dělení označené písmenem `M` (Math). Rozšíření `F` (Floating point) obsahuje instrukce a registry pro práci s plovoucí řadovou čárkou.
2. `Nestandardní rozšíření` jsou vysoce specializovaná rozšíření, které mohou být v konfliktu s ostatními rozšířeními. Např. rozšíření `C` (Compressed) obsahuje 16bitové verze některých instrukcí, díky čemuž umožňuje zmenšit velikost programu.

Na obrázku 2.11 jsou vyobrazeny formáty 32bitových RISC-V instrukcí.

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

Obrázek 2.11: Formáty 32bitových RISC-V instrukcí. [3]

2.3.1 Porovnání RISC-V a MIPS

Architektury RISC-V a MIPS si jsou velmi podobné. Mezi společné vlastnosti patří:

- Všechny instrukce jsou dlouhé 32 bitů.
- Obě architektury disponují 32 univerzálními registry.
- Jediným způsobem, jak přistoupit k paměti za pomocí instrukcí pro čtení a zápis.
- Narozdíl od jiných architektur neobsahuje MIPS ani RISC-V instrukce pro čtení nebo zápis více instrukcí naráz.

Znatelným rozdílem je to, že architektura RISC-V je modernější, všestrannější a rozšiřitelnější. Velmi důležitým rozdílem je, že instrukční sada RISC-V je kompletně otevřená.

Jedním z rozdílů mezi MIPS a RISC-V architekturou jsou i podmíněné skoky. RISC-V obsahuje instrukce, které přímo porovnají hodnoty dvou registrů. MIPS instrukce nejsou tak generické a mohou pouze porovnat, jestli jsou hodnoty v registrech stejné nebo ne.

Kapitola 3

Návrh simulátoru

Tato kapitola je věnována návrhu simulátoru. Pokusím se zde navrhnout konstrukce a základní koncepty simulátoru MIPS procesoru bez ohledu na cílenou platformu a využité technologie.

V úvodu je nutné si specifikovat s jakou granularitou bude výsledný simulátor pracovat.

V případě jednoduššího simulátoru, který demonstruje činnost procesoru pouze na úrovni registrů a paměti je implementace poměrně jednoduchá. Činnost jednotlivých instrukcí je primitivní a jejich realizace poměrně přímočará. Co je nutné u takového simulátoru řešit, je korektní demonstrace aritmetických a logických operací s 32bitovými hodnotami, práci s pamětí, registry, program a zpracování vstupního programu.

V případě komplexnějšího simulátoru, který by demonstroval chování procesoru na úrovni zřetězené linky s vizualizací stavů jednotlivých komponent pipeline, je implementace výrazně komplikovanější. Realizace registrů a paměti není o moc složitější. Ovšem vizualizace průběhu instrukce v samotné pipeline znamená, že se vykonávání všech instrukcí bude muset rozdělit na jednotlivé fáze, jako u skutečné MISP pipeline.

Bez ohledu na granularitu je při vytváření simulátoru vhodné použít návrhový vzor, který odděluje simulační jádro a uživatelské rozhraní. To by měl být stavební kámen pro strukturovaný a snadno udržitelný simulátor. S tím ovšem přichází problém propojení těchto dvou vrstev.

V mém případě jsem se rozhodnul implementovat simulátor, tak aby demonstroval činnost MIPS procesoru na úrovni zřetězené linky.

3.1 Návrh implementace simulačního jádra

Tato část se věnuje návrhu implementace simulační vrstvy. Jsou zde popsány způsoby a algoritmy, které mohou být použity při implementaci různých částí simulátoru.

Je vhodné, aby jednotlivé části MIPS procesoru byli rozděleny do samostatných modulů. To je především náležité pro škálování aplikace a její údržbu.

3.1.1 Instrukce

U implementaci instrukcí je patřičné vytvořit strukturu, kterou je možné popsat libovolnou instrukci. Strukturu je vhodné navrhnout tak, aby obsahovala zcela všechno, co k ní patří, aby se dala implementovat na jednom místě a nemusela se skládat z několika funkcí, které by se navzájem provolávali. Díky tomu bude přidání nové instrukce jednoduché a zjištění nebo úprava stávající instrukce nenamáhavé.

Základní parametry, které je třeba definovat:

- `Název instrukce` - to může fungovat i jako její unikátní identifikátor.
- `Způsoby zápisu instrukce` - to kolik očekává parametrů a jakého typu. Zároveň se musí dát pozor na to, že některé instrukce jde zapsat různými způsoby.
- `Popis exekuce` - jak se má daná instrukce zpracovat.

V případě komplexního simulátoru s pipeline je třeba definovat výrazně víc parametrů:

- `Širší popis` - definice toho, jestli se jedná o instrukci skoku, čtení z paměti, zapisování do paměti atd...
- `Hodnoty které mění` - Jaké hodnoty budou změněny danou instrukcí, zda-li se jedná o registr předaný v parametru instrukce, nebo se jedná o instrukce, které nejsou ani součástí syntaxe instrukce.
- `Bitové hodnoty instrukce (oplen, funct)` - V případě, že chceme definovat instrukce na úrovni bitů.

Spousta těchto parametrů je obzvlášť vhodná pro odhalování hazardů v pipeline.

3.1.2 Zpracování vstupních dat

Vstupem simulátoru by měl být kód jazyka symbolických instrukcí MIPS. Nad vstupním kódem se provede lexikální analýza a syntaktická analýza. V případě chyby, by měl být uživatel upozorněn na chybu ve vstupním kódu, jinak by měl být kód uložen. Tato část je velmi podobná pro většinu typů simulátorů. Měla by se opírat o modul s informacemi o instrukcích. Důležité je vytvořit strukturu pomocí, která se bude dát vhodným způsobem předat modulu programu instrukce a paměti zpracovaná data.

3.1.3 Registry

Důležitou částí simulátoru jsou registry. V simulátoru by se měli chovat stejně jako v reálném MIPS procesoru, kde je každý registr dlouhý 32 bitů. A je tedy potřeba zajistit, aby výpočty byly korektně provedeny. Některé, především vysokoúrovňové jazyky v základu pracují s čísli na více než 32 bitech. To by mohlo zapříčinit chybné výpočty v případech přetečení, bitových posunů a práci se zápornými čísli. Tento problém je nejlepší řešit vestavěnými funkcemi pro práci s 32bitovými čísli. Případně je nutné použít knihovnu.

3.1.4 Paměť

Další nezbytnou částí je paměť. V reálném MIPS počítači lze do paměti uložit až několik milionu 8bitových hodnot. Vzhledem k tomu, že podpora takto velkého úložiště by znamenala příliš velké nároky na systém, je nutné simulovanou paměť zjednodušit.

Textový segment, který slouží pro uložení uživatelského kódu je potřeba vytvořit jen pokud chcete umožnit načítat kód jako hodnoty. V realitě ale toto dělat nechcete a čtení z textového segmentu je pouze výsledkem nesprávné operace s pamětí. Proto stačí, když případný pokus o čtení z textového segmentu skončí jako chyba.

Zbytek uživatelského prostoru paměti je třeba naimplementovat. Ovšem i tak by se jednalo o miliony bajtů adresového prostoru, které v rámci simulátoru nejsou potřeba.

Proto stačí, když každý segment bude moct být adresován jen v určitém rozmezí. Například prostor pro dynamická data, bude pouze v rozmezí adres 0x1004 0000 - 0x1FFFFFFF. Případ, kdy by se uživatel pokusil číst data z adresy např. 0x3fffffff, je nejpravděpodobněji pouze špatnou operací s pamětí a jedná se o chybu.

U způsobu implementace samostatné paměti je klíčové, aby bylo možné číst a ukládat alespoň na úrovni bajtů. Implementace takové paměti, by se dalo dosáhnout pomocí pole, kde by každý prvek představoval jeden byte. K tomu by byla potřeba řada dalších metod, které by zajistili správnou demonstraci paměti. Nejlepší je ale opět použít zabudované funkce jazyka nebo použít knihovnu, která by toto podporovala.

3.1.5 Program

Program by měl být samostatný modul, který drží zpracovaný program překladače. Jednou z prvních věcí, které je zde potřeba řešit, jsou pseudointrukce. Buď se dají implementovat jak ve skutečném MIPS procesoru a transformovat je na adekvátním sekvenci vestavěných MIPS instrukcí, Nebo se s pseduinstrukcemi dá pracovat jako s ostatními vestavěnými instrukcemi. Kromě programu, by tento modul měl držet informace o návěštích a hodnotu čítače instrukcí (PC - Program counter).

3.1.6 Pipeline

V případě simulátoru, který demonstruje chování MIPS procesoru na úrovni zřetězené linky, bude nejvhodnější implementovat jednotlivé části pipeline, tak aby fungovala jako reálné. Společně by měli fungovat jako funkční celek. Jednotlivé části pipeline by měly být rozděleny do samostatných modulů, které mezi sebou komunikují. Vhodný způsob, je vytvořit jeden modul, který zodpovídá za komunikaci mezi částmi pipeline. Tento modul může fungovat i jako vstupní bod celého simulačního jádra a spravovat komunikaci mezi pipeline, registry, pamětí a programem.

Komunikace mezi fázemi

Komunikaci mezí fázemi lze zajistit pomocí vyrovnávacích pamětí 2.1.5, v kterých se budou předávat hodnoty. To pro implementaci znamená vytvořit vhodnou strukturu, ze které jednotlivé fáze budou moct číst a následně do ní opět zapisovat.

Fáze pipeline

Jednotlivé fáze jsou popsány v sekci 2.1.5. V případě velmi detailního simulátoru, který by měl zobrazovat stav na úrovni logických hradel je možné implementaci provést s využitím bitových operací a posunů. To ovšem může přinést zbytečné komplikace.

Druhou možností je jednotlivé fáze naimplementovat jednoduše a provádět základní operace, které se dají shrnout ve 3 bodech:

1. Získá instrukce z předcházející vyrovnávací paměti. (U `IF` načíst instrukci z programu.).
2. Provést operaci nad instrukcí voláním funkce, která je součástím popisu instrukce.
3. Nahrát upravenou instrukci do další vyrovnávací paměti. (U `WB` jí nahrát do cílového registru.).

Následně je možné stav hradel vypočítat samostatně bez dopadu na fungování pipeline. Jednou z klíčových funkcionalit, na kterou je třeba dát pozor, je hodinový signál. Některé operace jsou totiž provedeny během náběžné hrany a některé během sestupné hrany. To hraje velkou roli v tom, jak se určité fáze ovlivňují. Například instrukce skoku zapříčiní nastavení hodnoty čítače instrukcí při sestupné hraně ve fázi `ID`. Ovšem načtení instrukce ve fázi `IF` se provede už při náběžné hraně. Proto je třeba obsah vyrovnávací paměti `IF/ID` vyprázdnit, jelikož obsahuje nesprávnou instrukci.

Kontrolní jednotku (`Contril unit`) není třeba implementovat. Z informací o instrukci by mělo být jednoduché vyčíst, o jakou instrukci se jedná a co by se s ní v jednotlivých fázích mělo dít.

Forwarding Unit

Forwarding řeší závislosti mezi instrukcemi a byla popsána v sekci 2.1.5. Implementace Forwarding jednotky je celkem přímočará a řídí se jednoduchými pravidly. Pokud je v `id/ex` hodnota, do které zapisuje jedna ze 2 předchozích instrukcí, tak se provede forwarding. Ten uloží hodnotu do `id/ex` před tím, než se provede fáze EX.

Pseudokód pro implementaci forwardingu:

```
if(changeValueUsedInId_ex(mem_wb)){
     id_ex.r1 = changeR1(mem_wb) ? mem_wb.res : id_ex.r1;
     id_ex.r2 = changeR2(mem_wb) ? mem_wb.res : id_ex.r2;
}
if(changeValueUsedInId_ex(ex_mem)){
     id_ex.r1 = changeR1(ex_mem) ? ex_mem.res : id_ex.r1;
     id_ex.r2 = changeR2(ex_mem) ? ex_mem.res : id_ex.r2;
}
```

Důležité je vzít výsledek operace, která jako poslední zapisovala. Pokud tedy obě předchozí instrukce změnily hodnotu použitého registru, tak je nutné, aby se převzala hodnota z ex/mem. Tam je totiž aktuální hodnota.

Hazard Unit

Tato jednotka doplňuje Forwarding jednotku. Kontroluje, jestli nevzniká závislost mezi instrukcemi, kde první instrukce načítá data z paměti a nelze tedy použít forwarding. V takovém případě je potřeba pozastavit pipeline, konkrétně fáze `IF` a `ID`.

```
if(memReadInst(id_ex)){
    if(if_id.r1 == id_ex.r1 || if_id.r2 == id_ex.r1){
        fetch.stall();
        decode.stall();
    }
}
```

Tato jednotka by měla odhalovat i hazardy, které byly zmíněny v sekci 2.1.5. K odhalování závislostí mezi instrukcemi se hodí dobrý návrh instrukce, který by definoval, jaké

registry instrukce mění, které vyžaduje a v jaké části se provádí její exekuce (instrukce `add` v `EX`, instrukce `lw` v `MEM` atd...). Pokud simulátor umožňuje vypínat `Forwarding jednotku`, tak je nutné aby s tím `Hazard jednotka` počítala a náležitě pozastavovala pipeline.

Pořadí volání

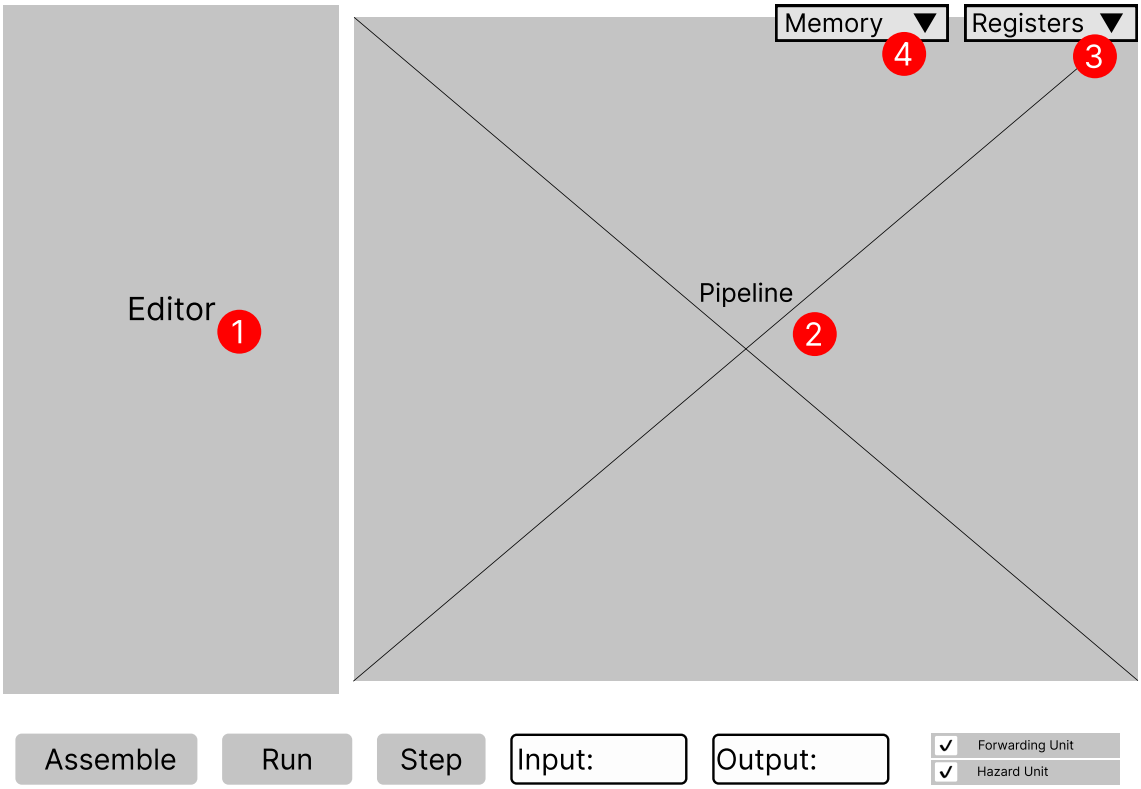
Jak již bylo naznačeno, při sekvenčním zpracování pipeline, kde se jednotlivé fáze zpracovávají paralelně je nutné správně definovat pořadí provádění jednotlivých fází. Proto je dobré provádění některých fází rozdělit na dvě části (funkce) podle toho, jestli se dějou během náběžné (runRisingEdge) nebo sestupné (runFallingEdge) hrany. Pořadí volání funkcí se musí řídit i obecnou podmínkou závislosti mezi fázemi. Tedy, že fáze může být ovlivněna pouze fází, která jí následuje. To znamená, že fáze `ID`, může být ovlivněna fázemi `EX, MEM a WB`. Nikoliv ale fází `IF`. Před vykonáním jednotlivých fází je ovšem nutné, aby své úkony provedly `Hazard unit` a `Forwarding unit`. První musí být volána `Hazard unit`, jelikož ta se rozhoduje i na základě toho, jestli je forwarding dostupný. Následně je možné volat `Forwarding unit`.

Pořadí volání v jednom cyklu by mělo být následující:

1. HazardUnit.run()
2. ForwardingUnit.run()
3. WriteBack.runRisingEdge()
4. Memory.runRisingEdge()
5. Execute.runRisingEdge()
6. Decode.runRisingEdge()
7. Fetch.runRisingEdge()
8. Memory.runFallingEdge()
9. Execute.runFallingEdge()
10. Decode.runFallingEdge()
11. Fetch.runFallingEdge()

3.2 Návrh uživatelského rozhraní

Uživatelské rozhraní se dá implementovat různorodě, při jeho realizace by se měli dodržovat základní pravidla tvorby uživatelského rozhraní [9]. U simulátoru MIPS procesoru jsou některé elementární prvky, které je potřeba implementovat.



Obrázek 3.1: Wireframe uživatelského rozhraní

Editor kódu

Editor kódu je zvýrazněn na obrázku 3.1 pod číslem `1`. Dá se uvažovat nad implementací simulátoru, který by získával vstupní kód ze souboru, který by uživatel musel nahrát. To je ovšem velmi nepraktické a mnohem lepší možností je vytvořit vlastní editor. Ten by měl vhodně zvýrazňovat části kódu syntaxe. V současné době existuje celá řada volně dostupných editorů zdrojových kódů, které je možné zaintegrovat do vlastní aplikace. Mezi nejznámější patří např. Ace editor, CodeMirror nebo Monaco Editor.

Registry

Rozbalovací menu s registry je zvýrazněno na obrázku 3.1 pod číslem `3`. Zobrazení registrů je velmi důležité pro všechny simulátory, jelikož se na nich nejjednodušeji pozoruje, jak jednotlivé instrukce fungují. Pro uživatele může být také užitečné, možnost přepínat mezi zobrazením hodnot v šestnáctkové a desítkové soustavě.

Paměť

Rozbalovací menu s pamětí je zvýrazněno na obrázku 3.1 pod číslem `4`. Při práci s mips můžeme používat i paměť, bylo by tedy vhodné, aby do ní uživatel mohl nahlédnout. Paměť je vhodné reprezentovat jako tabulku, kde na každém řádku je adresa a hodnota. Pro ušetření místa je vhodné paměť reprezentovat po slovech (32 bitech). Nutností je uživateli umožnit přepínání mezi segmenty paměti. Není vhodné, aby se vytvořila tabulka s tisíci položkami,

jelikož by se v takovém množství špatně orientovalo. Možným řešením je implementovat tzv. `Infinite loop`, který umožňuje načíst další data pokud se uživatel dostane ke spodní hranici vykreslených dat.

Pipeline

Pipeline je zvýrazněna na obrázku 3.1 pod číslem 2. Pakliže simulátor pracuje na úrovni pipeline, tak je třeba, aby byla vhodně zobrazena. V první řadě je třeba vybrat technologii, pomocí které bude pipeline vykreslena. Nejlepší je zvolit některý z vektorových grafických formátů, aby se zachovala kvalita a čitelnost bez ohledu na rozměry.

Dále je potřeba rozhodnout o tom, jak bude pipeline demonstrována. Jednou z možností je zobrazit pipeline jako MIPS architekturu s jednotlivými komponentami 2.6. Druhou možností je vizualizovat, jak se v čase jednotlivé instrukce nacházejí v různých fázích pipeline 2.5.

Co může být pro uživatele přívětivé je vizualizace toho, kde se jaká instrukce nachází nejen v samotné komponentě vizualizujíc pipeline, ale taky přímo v editoru kódu (případně textovém segmentu).

Kapitola 4

Implementace

V této kapitole se budu věnovat vlastní implementaci simulátoru MIPS procesoru, který demonstruje činnost MIPS procesoru na úrovni zřetězené linky. Především jsem se zaměřil na to, aby simulátor podporoval a korektně demonstroval, co nejvíce různorodých instrukcí. Zároveň jsem se soustředil na to, aby simulátor byl univerzální, snadno udržitelný a především lehce rozšířitelný. Svůj simulátor jsem implementoval jakožto webovou aplikaci. Očividná výhoda je, že uživateli k používání mého simulátoru stačí libovolné zařízení s webovým prohlížečem a připojením k internetu.

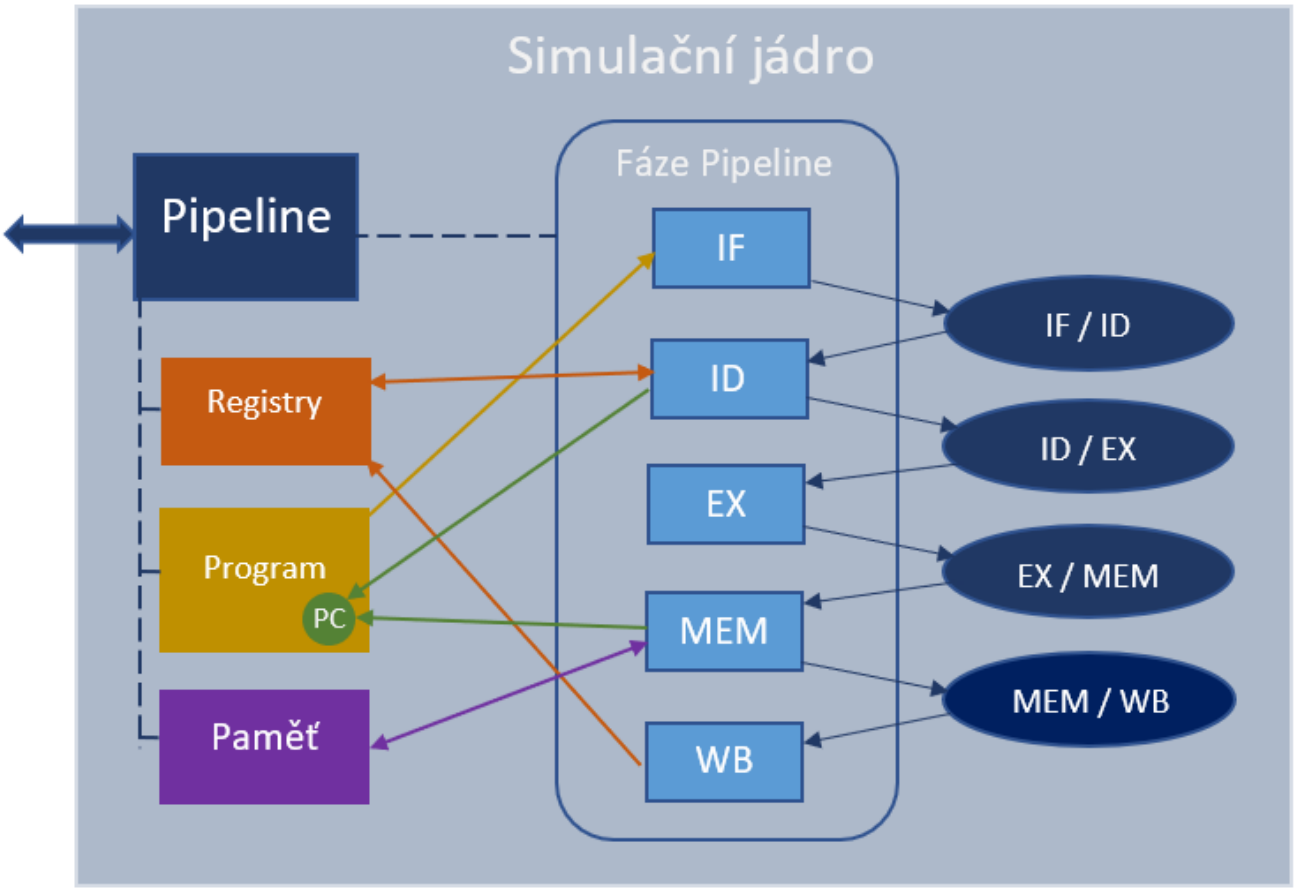
Jakožto jednu z technologií pro implementaci jsem zvolil programovací jazyk `TypeScript`. Jedná se o skriptovací jazyk, který je nadstavbou jazyka JavaScript. Rozšiřuje jej o statické typování a další atributy, které se objevují v objektově orientovaném programování. Samostatný kód napsaný v TypeScriptu se transpiluje na ekvivalentní JavaScript kód.

Druhou použitou technologií je framework `React`. Jedná se o javaScriptovou knihovnu pro tvorbu uživatelského rozhraní. React umožňuje rozdělit části uživatelského rozhraní do samostatných komponent, které mezi sebou komunikují a dynamicky mění svůj obsah. Každá komponenta obsahuje svoje vlastnosti a spravuje svůj stav.

U implementace jsem použil konstrukce a myšlenky, které jsem popsal v Návrhu implementace [3]. Aplikace je rozdělena na dvě hlavní vrstvy. Vrstva `Simulačního jádra`, která řeší analýzu a překlad vstupního kódu a celou simulaci. Druhou vrstvou je `Uživatlské rozhraní`, kde uživatel může psát svůj kód, konfigurovat pipeline a sledovat jak je jeho program zpracován na úrovni zřetězené linky. každá vrstva se skládá z řady modulů, které mají v rámci vrstvy i aplikace specifický význam.

4.1 Simulační jádro

Vrstva simulačního jádra se skládá z hlavního modulu `pipeline.ts`, který funguje jako vstupní bod simulačního jádra a inicializuje ostatní moduly této vrstvy. Mezi přední moduly simulačního jádra patří `register.ts`, jehož úkolem je držet stav registrů během simulace a ostatním modulům poskytovat možnost upravovat a číst jejich obsah. Dalším modulem je `program.ts`, ten získává zpracovaný vstupní kód od modulu `parser.ts`, který ještě zpracuje a uloží. Drží i informaci o momentální hodnotě PC (čítači instrukcí) podle které vybírá další instrukci. Modul `memory.ts` řeší práci s pamětí, implementuje metody, pomocí kterých ostatní moduly mohou ukládat a číst z paměti. Od parseru získává statická data uložená v `.data` sekci vstupního kódu, které zpracuje a uloží. Všechny instrukce jsou popsány v modulu `instruction`.



Obrázek 4.1: Diagram simulačního jádra

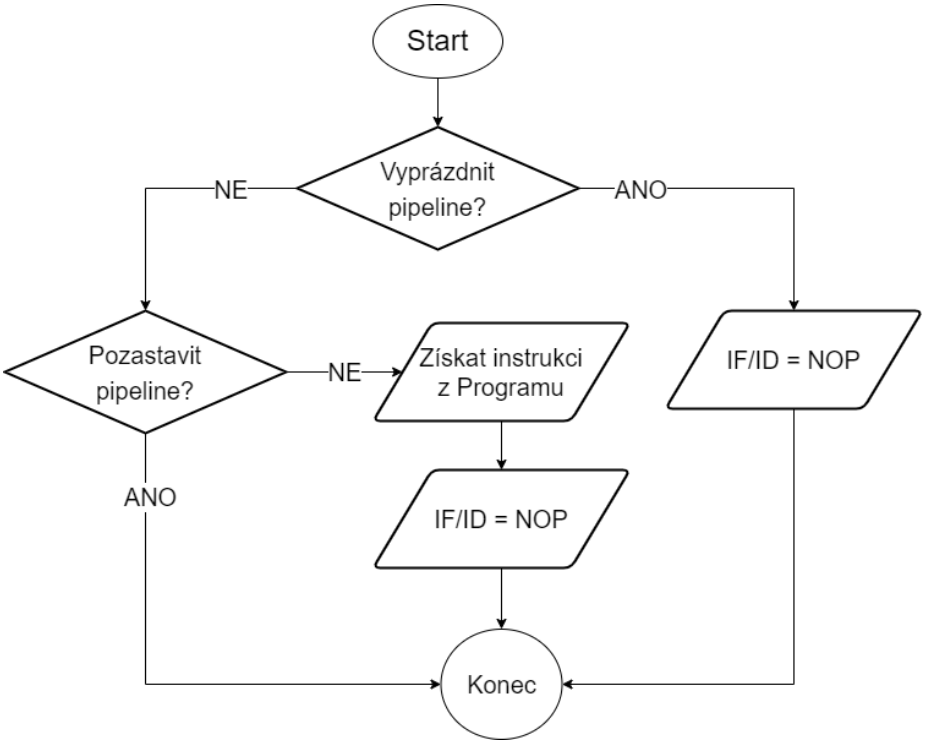
Na obrázku 4.1 je vidět zjednodušená závislost mezi prvky simulačního jádra. Jsou zde hlavně zobrazeny, jak jednotlivé fáze pipeline využívají registry, program a paměť. Zároveň je zde zobrazeno, jak mezi sebou jednotlivé fáze pipeline komunikují (jak si předávají zpracovanou instrukci). K tomu používají vyrovnávací paměti `IF/ID`, `ID/EX`, `EX/MEM`, `MEM/WB`. Ty jsou součástí komponenty `pipeline`, která implementuje metody pro čtení a zápis do těchto struktur.

4.1.1 Fáze pipeline

V této sekci popíšu a ukážu, jak jsem jednotlivé fáze pipeline implementoval. Každá fáze je implementována jako samostatný modul.

Fáze IF - Instruction fetch

Tato fáze načítá další instrukci z programu a ukládá jí do vyrovnávací paměti `IF/ID`. V případě vyprázdnění pipeline žádnou instrukci nenačítá a nastaví další instrukci na NOP. Pakliže je pipeline pozastavena, tak tato fáze nic nedělá. Toto chování je vizualizováno na obrázku 4.2.



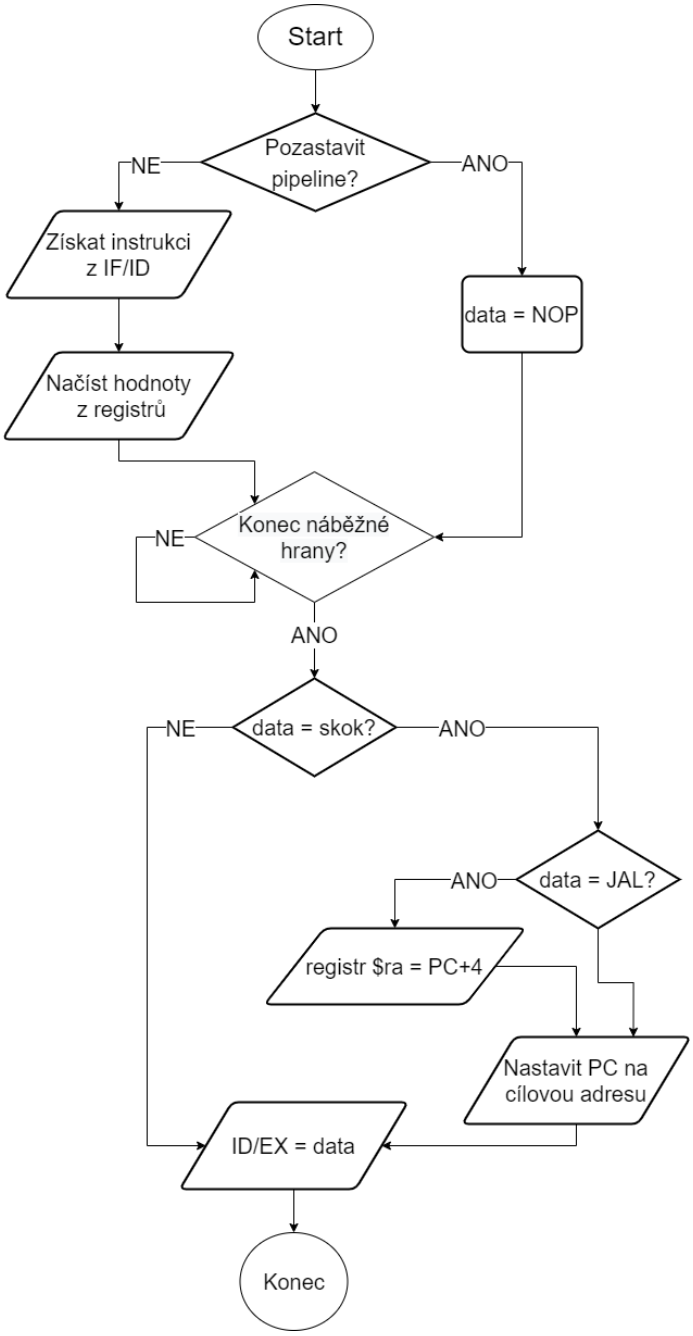
Obrázek 4.2: Diagram fáze Instruction fetch

Fáze ID - Instruction decode

Fáze `ID` je rozdělena na náběžnou a sestupnou hranu. Pro implementaci toto znamená, že existují dvě metody (runRisingEdge a runFallingEdge), které jsou volány v rozdílnou chvíli.

V náběžné hraně fáze načte instrukci z vyrovnávací paměti `IF/ID` nebo v případě pozastavení pipeline, nastaví data na NOP. Následně načte potřebné hodnoty registrů.

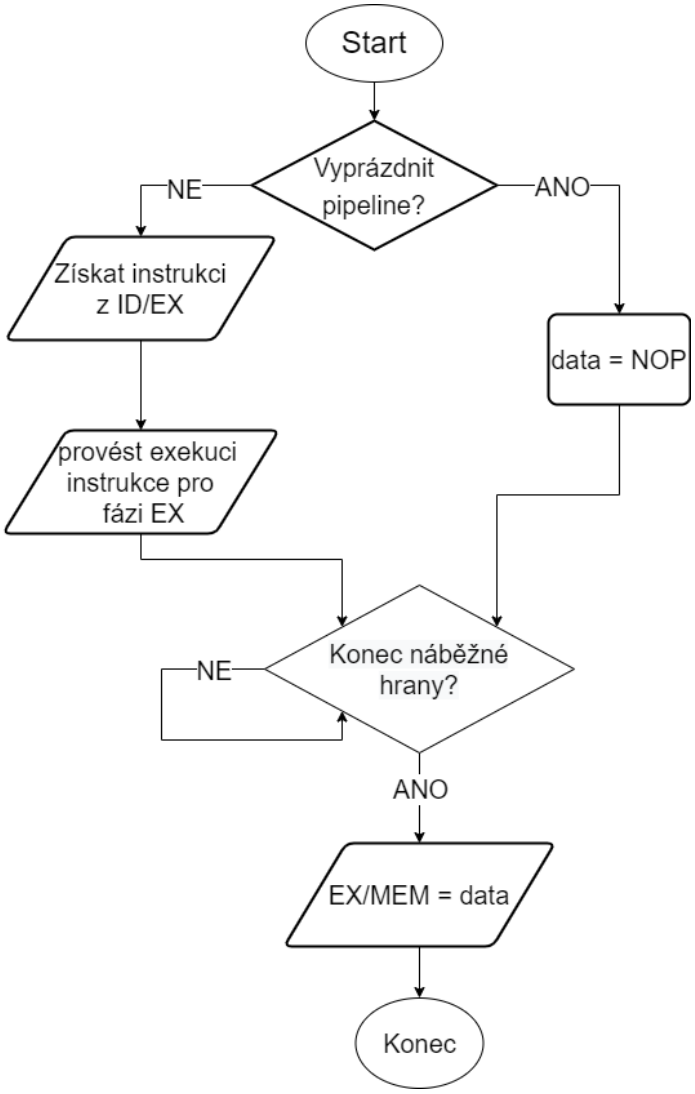
V sestupné hraně se detekuje skok. Pakliže se jedná o instrukci skoku, tak se nastaví `PC` na novou adresu. Pokud se zároveň jedná o instrukci `jal`, tak se se nastaví i hodnota registru $ra na hodnotu PC + 4 (adresa následující instrukce). Nakonec se data nahrají do vyrovnávací paměti `ID/EX`. Toto chování je vizualizováno na obrázku 4.3.



Obrázek 4.3: Diagram fáze Instruction decode

Fáze EX - Execute

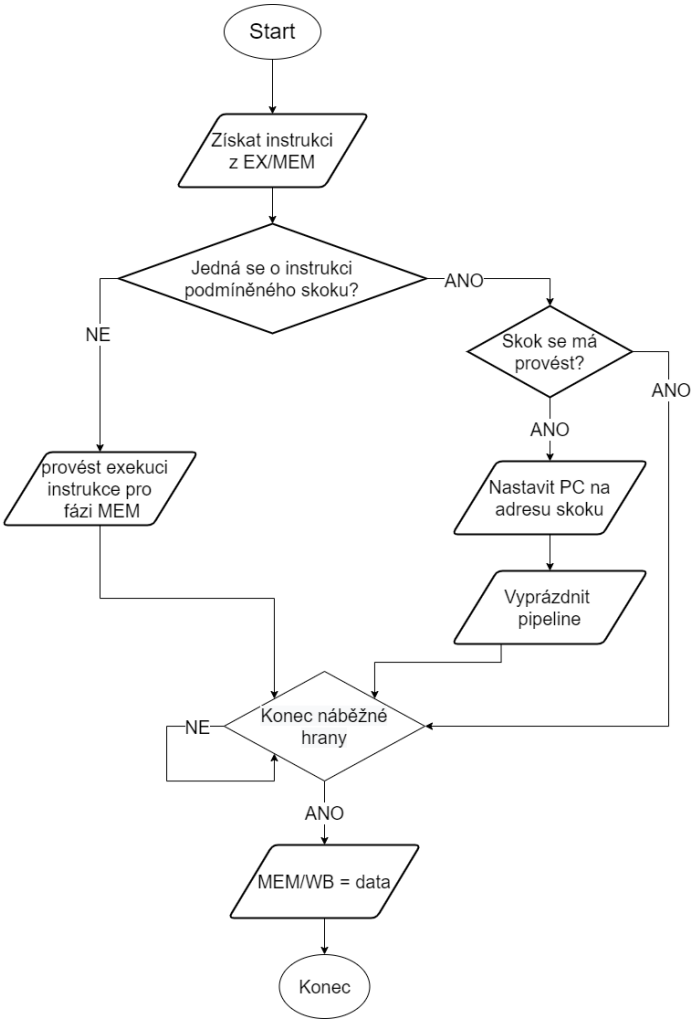
Fáze `EX` má za úkol vykonat aritmetické a logické instrukce. V případě, že se má vyprázdnit pipeline se nastaví data na NOP. Jinak se zavolá funkce instrukce, která implementuje její chování ve fázi `EX`. Aritmetické a logické funkce zde mají svojí implementaci exekuce. Pro ostatní instrukce (`LW, jal, syscall ...`) tato funkce pouze vrací je samotné beze změny. V sestupné hraně se pouze uloží data do `EX/MEM`.



Obrázek 4.4: Diagram fáze Execute

Fáze MEM - Memory access

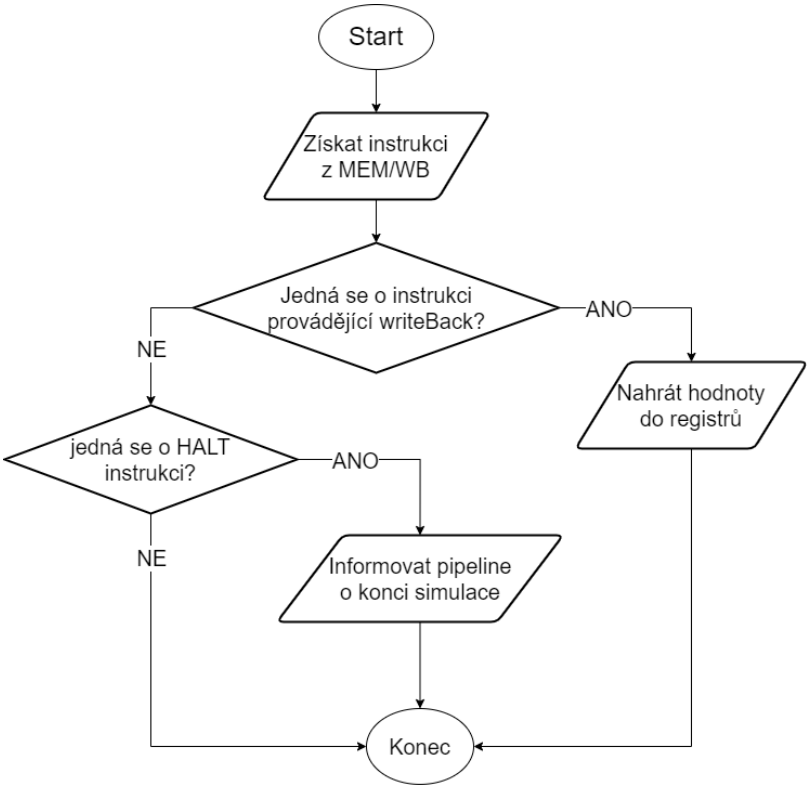
Ve fázi `MEM` se podobně jako ve fázi `EX` volá funkce instrukce, které implementuje její chování pro tuto fázi. Zároveň ře zde provádí podmíněné skoky. Původně jsem se rozhodoval mezi prováděním podmíněných skoků na konci fáze `ID` nebo na začátku fáze `MEM`. Provádět podmíněné skoky ve fázi `ID` je obecně lepší a efektivnější, jelikož se podmíněný skok provede dřív. Ovšem ve fázi `ID` se provádí již nepodmíněné skoky a ve svém simulátoru jsem chtěl provedení těchto dvou typů skoků rozlišit. Proto jsem se rozhodl, že vyhodnocení podmínky pro skok se provede až ve fázi `EX` a samotný skok se provede na začátku fáze `MEM`.



Obrázek 4.5: Diagram fáze Memory access

Fáze WB - Write back

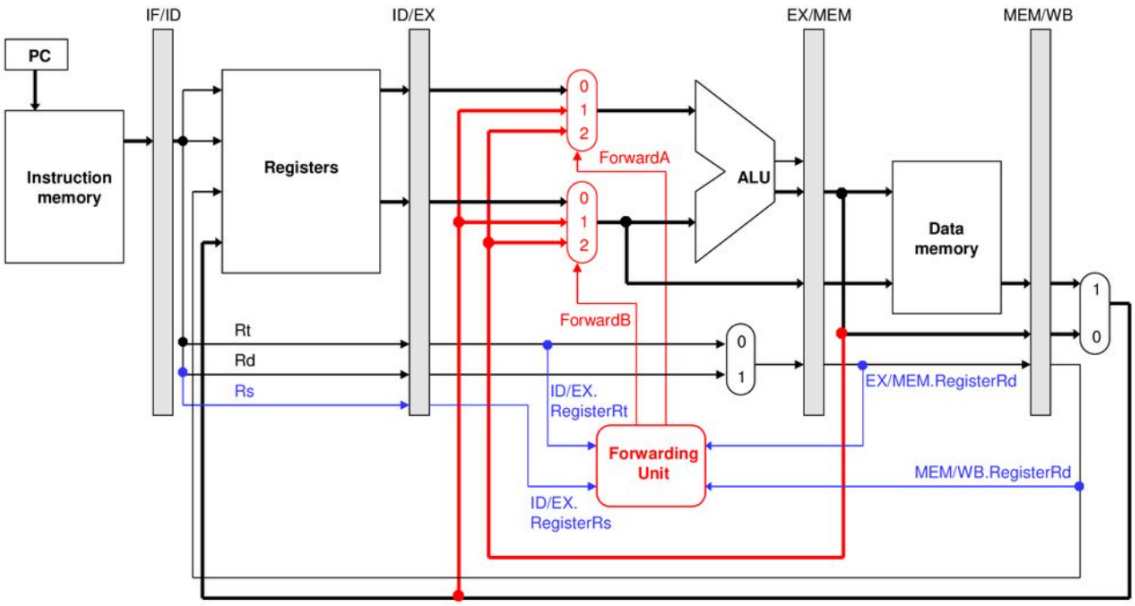
Poslední fáze `WB` načte instrukci z vyrovnávací paměti `MEM/WB`. Pro instrukce, které ukládají výsledek do některého z registrů se provede zpětný zápis za využití metod modulu `register.ts`. Pokud se jedná o instrukci `halt`, tak se informuje pipeline o konci simulace.



Obrázek 4.6: Diagram fáze Write back

Forwarding unit

Modul `forwardingUnit` zaujímá místo Forwarding jednotky. Při její implementaci jsem použil algoritmus popsaný v návrhu implementace [3.1]. Tuto jednotku je možné v mém simulátoru vypnout. Hazard jednotka s tím počítá a v případě, že vznikne datová závislost mezi instrukcí uloženou ve vyrovnávací paměti `IF/ID` a jinou instrukcí v pipeline, tak je pipeline pozastavena do doby dokud datová závislost nezmizí. V reálné MIPS pipeline jsou hodnoty vybrány pomocí multiplexorů ovládané Forwarding jednotkou, jak je zobrazeno na obrázku 4.7. V mém simulátoru se na začátku simulace jednoho hodinového cyklu zkontrolují hodnoty uložené ve vyrovnávací paměti `ID_EX` a případně se přepíší.



Obrázek 4.7: Ukázka implementace forwardingu v pipeline (červeně). [12]

Hazard unit

Modul `HazardUnit` odhaluje a řeší hazardy v pipeline. Většinu závislostí odhalí generickým algoritmem, který zjistí jaké hodnoty požaduje instrukce v `IF_ID` a jaké hodnoty jsou měněny instrukcemi v `ID_EX` a `EX_MEM`. Pakliže vznikne průnik, tak se jedná o datový hazard. Následně se podle hlavní fáze exekuce obou závislých instrukcí a dostupnosti forwardingu rozhodne, jestli má být pipeline pozdržena.

V simulátoru umožňuji tuto jednotku deaktivovat, uživatel si potom hazardy mezi instrukcemi musí uvědomovat sám a sám je taky řešit.

Pořadí volání

Pořadí volání fází pipeline společně s Forwarding a Hazard jednotkou jsem popsal v sekci 3.1.6. Má implementace se drží toho, co jsem navrhnul a pořadí volání je totožné. Jednou ze změn je, že Hazard a Forwarding jednotka se volá pouze v případě, že jsou aktivní. Druhou změnou je, že se kontroluje jestli fázi `WB` právě nedokončila instrukce `halt`. V takovém případě je simulace ukončena a uživatelské rozhraní je upozorněno pomocí `callback` funkce.

4.1.2 Instrukce

Instrukce jsou implementovány v samostatném modulu. V rámci celého simulačního jádra existují 3 úrovně struktur instrukcí, které jsou v sobě zanořené. Obrázek 4.8 ukazuje zanoření těchto struktur. Každá z těchto struktur popisující instrukci má v rámci simulátoru jiný význam a obsahuje rozdílné informace o instrukci.

Struktura IInstructionDescription

Jedná se o nejnižší strukturu popisující každou instrukci, kterou simulátor umí. Obsahuje základní neměnné údaje o instrukci.

Tato struktura obsahuje:

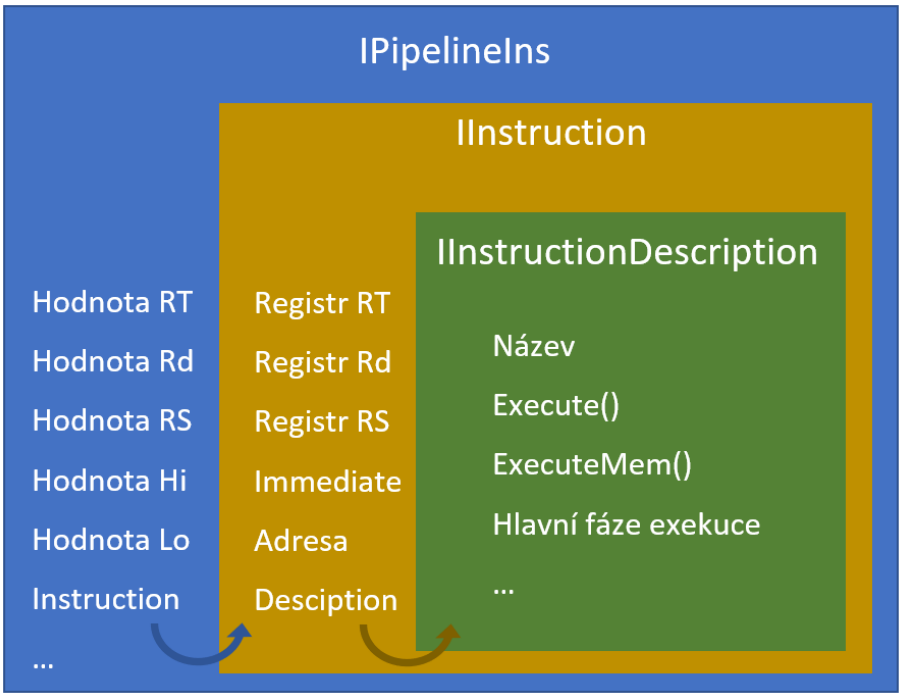
- `name` - Název instrukce.
- `isJumpInstruction, isMemoryInstruction...` - Jedná se instrukci skoku? Operuje instrukce s pamětí? atd...
- `mainExecutionStage` - Ve které fázi se provádí hlavní exekuce instrukce.
- `changedValues` - Jaké hodnoty budou změněny danou instrukcí, zda-li se jedná o registr předaný v parametru instrukce, nebo se jedná o instrukce, které nejsou ani součástí syntaxe instrukce.
- `requireSpecial` - Registry, které instrukce vyžaduje, ale nejsou součástí syntaxe.
- `paramTypes` - Všechny možné zápisy instrukce. Například instrukce pro operace s pamětí se dá zapsat čtyřmi různými způsoby.
- `execute()` - Implementace exekuce ve fázi `EX`.
- `executeMem()` - Implementace exekuce ve fázi `MEM`.

Struktura IInstruction

Tato struktura popisuje konkrétně zapsanou instrukci s konkrétními parametry. Je vytvořena při analýze zdrojového kódu modulem `parser.ts`. Obsahuje informaci o řádku v editoru kódu, typu zápisu, parametrech, cílovém registru a obsahuje odkaz na strukturu `IInstructionDescription`. Modul `program.ts` dostává instrukce v této struktuře a přidává jí ještě adresu, která funguje jako identifikátor.

Struktura IPipelineIns

Tato struktura se využívá při zpracování instrukce v pipeline. Jednotlivé fáze ji modifikují a předávají mezi sebou pomocí vyrovnávacích pamětí. Obsahuje konkrétní hodnoty, které byly načteny a vypočítány jednotlivými fázemi pipeline. Zároveň obsahuje odkaz na strukturu IInstruction.



Obrázek 4.8: Ukázka zanoření struktur instrukce

Pseudoinstrukce

Při implementaci pseudoinstrukcí jsem se rozhodl, že budou implementovány stejně jako vestavěné instrukce a nebude mezi nimi rozdíl. Negativní následek tohoto řešení je, že simulátor nedokonale demonstruje MIPS procesor. Výhodou na druhou stranu je, že uživatel přesně vidí svoji instrukci ve formátu, který zapsal a co se s ní v pipeline děje. Druhou výhodou je, že uživatelské rozhraní může zůstat jednoduché.

Pakliže bych chtěl implementovat pseudoinstrukce, tak by bylo nutné, buďto po sestavení programu upravit obsah editoru kódu, což je ze zjevných důvodů neideální praktika nebo vytvořit druhý editor, který by obsahoval přeložený program.

4.1.3 Registry

Pro práci s registry jsem implementoval modul `register.ts`, který uchovává hodnoty všech registrů a implementuje metody, pomocí kterých mohou být hodnoty konkrétních registrů modifikovány. U registrů nastává problém kvůli reprezentaci čísla v javaScriptu. Ten totiž datový typ `Number` reprezentuje jako 64-bitovou hodnotu s pohyblivou řádkovou čárkou. MISP ovšem pracuje s 32 bity a tento problém je třeba řešit, aby se výpočty prováděly korektně.

Tento problém jsem řešil pomocí zabudované třídy javaScriptu `Int32Array`, která vytváří pole 32 bitových hodnot ve dvojkovém doplňku. Stačí tedy vytvořit pole o jednom 32bitovém prvku pro každý registr.

4.1.4 Program

Modul `program.ts` uchovává informaci o návěštích, programu a čítači programu (PC). Při inicializaci dostává jako parametry pole instrukcí a pole návěští, kterým podle čísla řádku

přiřadí adresu. Obsahuje metodu `getNextInstruction()`, kterou používá fáze `IF` při čtení další instrukce. Pokud na adrese PC není žádná instrukce, tak se vrací NOP. To má za následek to, že programy neukončené instrukcí `halt` jdou ukončit jedině ručně.

4.1.5 Paměť

Operace s pamětí jsou řešeny modulem `memory.ts`. Paměť je další problematická část, jelikož je potřeba vytvořit strukturu, která bude držet hodnoty a umožňovat je číst po bajtech. K tomu jsem použil javaScriptovou třídu `ArrayBuffer`, která dokáže reprezentovat data na binární úrovni. Zapisovat a číst tyto data můžu pomocí metod třídy `DataView`.

Další problematiku, kterou jsem musel řešit a popsal v návrhu implementace je velikost alokované paměti. Uživatelský prostor v MIPS paměti zabírá přibližně 2GB. To je zbytečně mnoho. Textový segment jsem se tedy rozhodl neimplementovat, jeho roli zaujímá modul `program.ts`. Pokusy o čtení z textového segmentu jsou většinou pouze výsledkem špatné operace s pamětí. Segment pro zásobník, statická a dynamická data jsem tedy omezil adresovatelným rozsahem, který by měl být dostatečně velký pro jakýkoli smysluplný simulovaný program. Simulátor zároveň řeší zarovnání paměti a podporuje direktivy `.align` i `.space`.

4.1.6 Parser

modul `parser.ts` slouží jakožto syntaktický analyzátor vstupního kódu. Jeho výstupem je seznam instrukcí a statických dat pro program a paměť. Využívá modulu `instruction.ts` pro správnou analýzu instrukcí. V případě chyby je uživatel informován o chybném řádku.

Parser zároveň podporuje drobnou změnu v syntaxi instrukcí. Správně by parametry instrukce měly být odděleny čárkou. To ovšem dle mého mínění není dostatečně dobře čitelné a většinou je třeba oddělit parametry čárkou a mezerou. Svůj parser jsem se rozhodl implementovat tak, aby akceptoval i instrukce, které jsou odděleny pouze mezerou. Dle mého mínění zůstane syntaxe instrukce čitelná a zároveň stačí pouze jeden znak.

První dva zápisy jsou v MIPS obecně validní, poslední je zároveň validní v mém simulátoru:

```
.text
add $t0,$t1,$t2
add $t0, $t1, $t2
add $t0 $t1 $t2
```

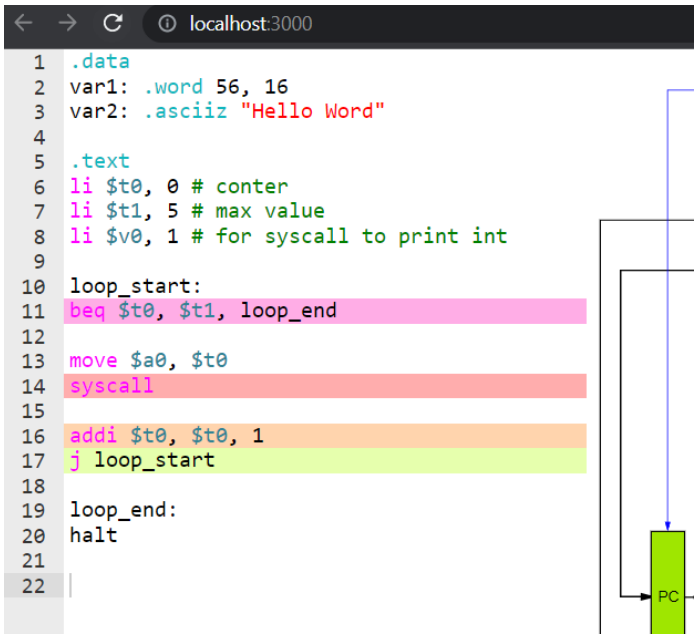
4.2 Uživatelské rozhraní

Pro implementaci uživatelského rozhraní jsem použil framework React s využitím grafické knihovny `Material UI` pro tvorbu jednotlivých komponent a jejich rozložení na stránce. Vzhled a rozložení webu je velmi jednoduché a intuitivní. Skládá se ze tří hlavních částí. Editoru kódu nacházející se na levé straně webu, obrázku pipeline na prvé straně a tlačítek nacházející se po celé šířce spodní části webu. Webová aplikace je responzivní a je možné jí používat na libovolném zařízení.

Vstupní bod uživatelského rozhraní je v modulu `App.tsx`. Ten drží stav simulátoru na úrovni uživatelského rozhraní a řeší rozložení webu. Další části webu jsou rozloženy do samostatných komponent.

4.2.1 Editor kódu

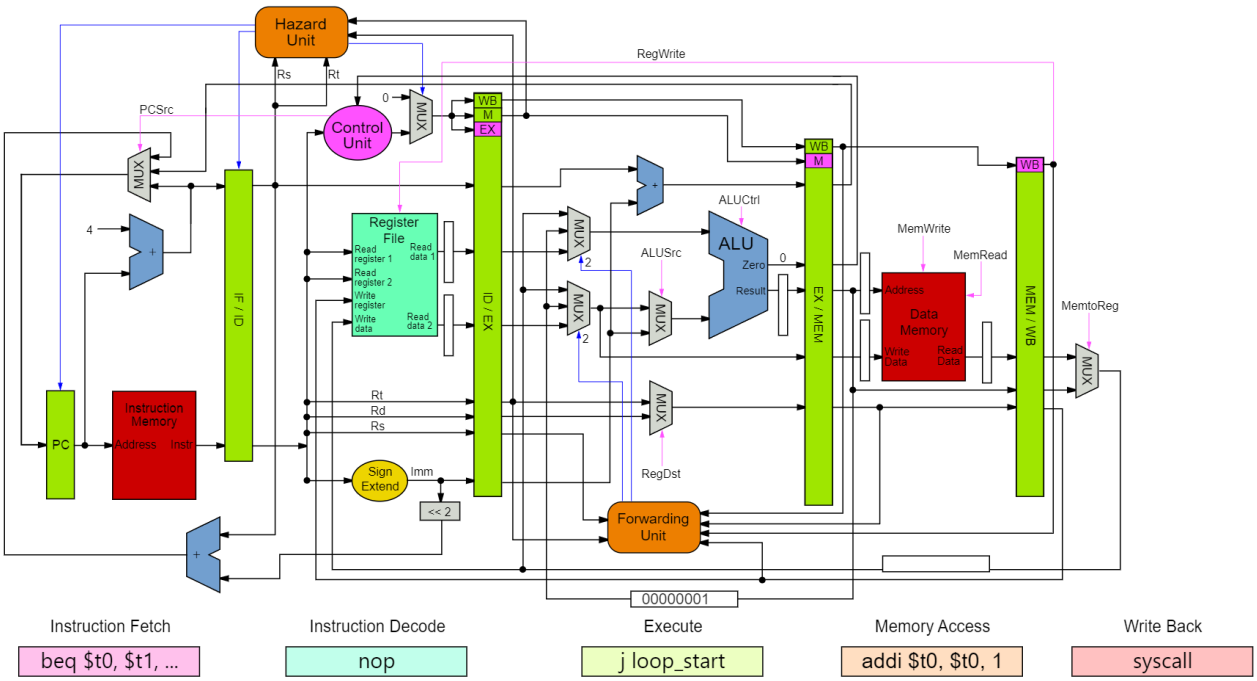
Editor kódu jsem vytvořil pomocí npm balíčku `react-ace`. Jedná se o verzi oblíbeného webového editoru pro React. Jeho hlavní výhodou je, že podporuje syntaxi jazyka MIPS, což znamená, že je text kódu barevně zvýrazňován. Jedna z užitečných funkcionalit editoru je že podporuje zvýraznění konkrétních částí textu. Díky tomu jsem schopný označit, v které fázi pipeline se určité instrukce právě nachází. Jedna z nevýhod Ace editoru je, že obsahuje chybu a nelze jeho výšku dynamicky upravovat. Tento problém jsem musel vyřešit definicí vlastních CSS pravidel, která přepisují CSS pravidla Ace editoru.



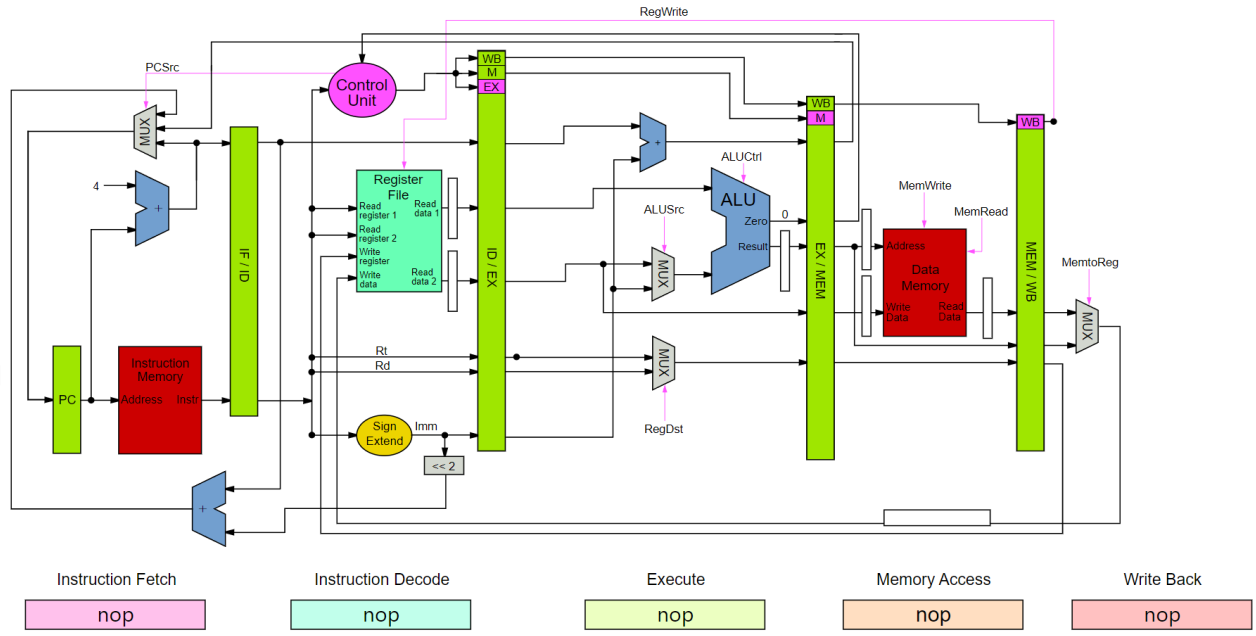
Obrázek 4.9: Ukázka editoru kódu

4.2.2 Zobrazení pipeline

Zobrazení pipeline zabírá největší část webu a je implementována v modulu `SvgContainer.tsx`. Jak název komponenty napovídá, pipeline vykresluji pomocí SVG, což přináší hned několik pozitiv. Prvně se jedná o vektorový grafický formát, takže bez ohledu na rozlišení se zachová kvalita vizualizované pipeline. Další výhodou je, že obsah SVG souboru je popsán pomocí XML. Což přináší možnost měnit obsah SVG souboru pomocí javaScriptu. Ve svém simulátoru umožňuji aktivovat a deaktivovat forwarding i hazard jednotku. To se projevuje i na samotné vizualizaci pipeline. Pakliže je některá z jednotek vypnuta, tak se pomocí selektorů korespondující části SVG schovají a některé se naopak zobrazí. Zároveň pomocí javaSciptu upravuji texty částí SVG, aby byly vidět hodnoty instrukce uvnitř pipeline. Ve fázích kde instrukce nic neprovádí (např.: instrukce add v `MEM` nebo lw v `EX`) se hodnoty na spojích nezobrazují. Výsledná pipeline s aktivními i neaktivními jednotkami je vyobrazena na obrázku 4.10 a 4.11.



Obrázek 4.10: Pipeline simulátoru s Forwarding i Hazard jednotkou



Obrázek 4.11: Pipeline simulátoru bez Forwarding a Hazard jednotky

4.2.3 Ovládací prvky

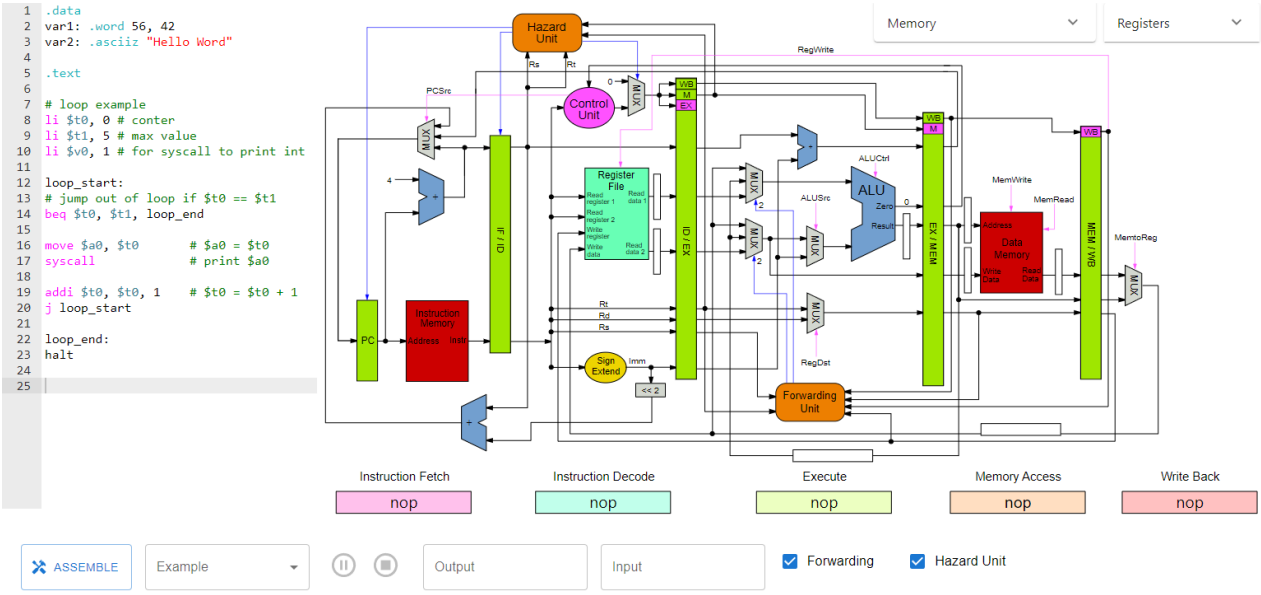
Mezi ovládací prvky UI patří tlačítka pro sestavení programu, provedení skoku a spuštění provádění simulace (provádění skoku v intervalu). Zároveň jsou zde dvě vstupní pole, jedno pro výstupní text (uživatel do něj nemůže psát) a pro vstupní text. Nakonec jsou zde dvě zaškrtávací políčka pro aktivaci a deaktivaci Forwarding a Hazard jednotky.

Pakliže program není sestaven, tak má uživatel možnost výběru z demonstračních příkladů. Každý ukázkový příklad je ve svém vlastním souboru a simulátor jeho obsah načte, až když je vybrán. Obsah vybraného demonstračního programu se vloží jako text do editoru.

4.2.4 Zobrazení registrů a paměti

V pravém horním rohu aplikace se nachází rozbalovací okna s hodnotami všech registrů a paměti. Uživatel může přepínat mezi zobrazením hodnot v šestnáctkové a desítkové soustavě. U okna s pamětí lze přepínat mezi segmenty.

Výsledný vzhled aplikace je vidět na obrázku 4.12.



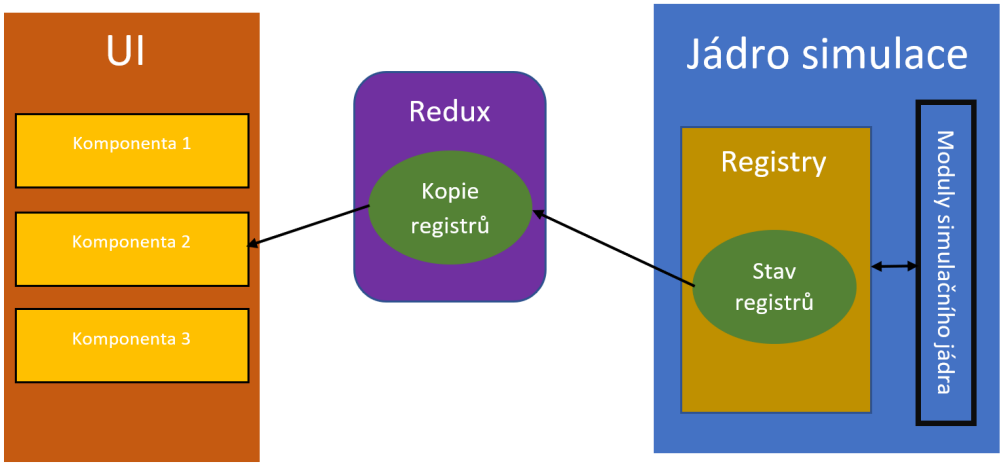
Obrázek 4.12: Výsledný vzhled aplikace

4.3 Propojení uživatelského rozhraní a simulačního jádra

Propojování vrstev uživatelského rozhraní a simulačního jádra není na první pohled nijak náročné. V modulu `App.tsx` se jednoduše vytváří instance modulu `pipeline.ts`. Problém nastává při předávání parametrů. Je potřeba simulačnímu jádru předat informaci o tom, jestli je aktivovaný forwarding a hazard unit. Také je mu potřeba předat informaci o vstupním řetězci a samozřejmě programu. Od simulačního jádra je naopak potřeba získat hodnoty registrů, paměti, stav fází na konci cyklu a výstupní řetězec. Parametry jako jsou informace o aktivitě Forwarding a Hazard jednotky stačí předat pomocí parametru modulu `pipeline.tsx`. Informaci o konci simulace jde naopak předat pomocí `callback` funkce, předané taky pomocí parametru. Diskutabilní ovšem může být předávání parametrů jako

je například vstupní řetězec. S tím se totiž pracuje až v implementaci exekuce instrukce v konkrétní fázi. To by znamenalo, předávat tento řetězec přes několik metod a funkcí až do místa, kde je potřeba.

Druhou možností je využít `Redux`. Jedná se o stavový kontejner pro javaScriptové aplikace a při tvorbě React aplikací je hojně využíván. Redux jsem se rozhodl implementovat i ve své aplikaci. Používám ho pro předávání vstupního a výstupního řetězce, informacích o chybách, hodnotách registrů a stavu jednotlivých fází. Redux funguje tak, že drží stavy definovaných hodnot a v případě změny se automaticky překreslí komponenty, které čtou tyto hodnoty. Já jsem ovšem Redux použil čistě k předání informace. Stav, například registrů, které se používají v jádru simulace, drží modul `register.ts`. Do Redux se nahrává pouze jejich kopie na konci cyklu. Tento vztah je zvýrazněný na obrázku 4.13.



Obrázek 4.13: Využití Reduxu pro předávání registrů do UI

Kapitola 5

Srovnání s existujícími simulátory

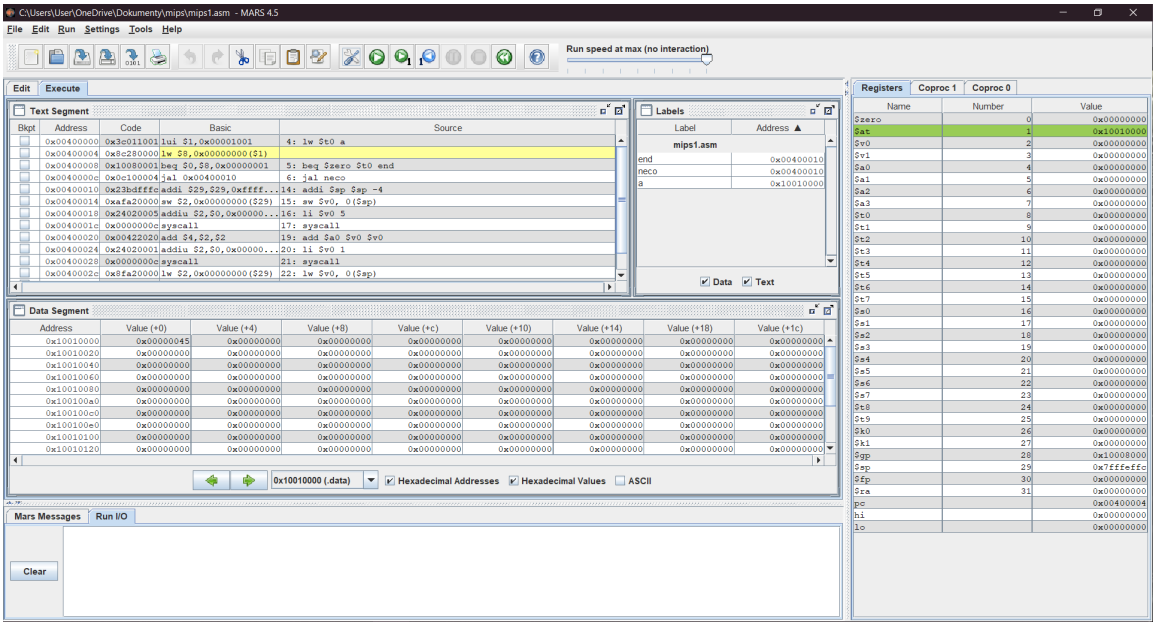
Tato sekce je zaměřena na srovnání mé implementace simulátoru s jinými existujícími simulátory MIPS procesoru. Podobných simulátorů existuje hned několik. Já se pokusím vybrat simulátory, které patří mezi populární a simulátory, které jsou implementovány pro web.

5.1 MARS

MARS (Mips Assembly and Runtime Simulator)[14] (obrázek 5.1) patří mezi nejpopulárnější simulátory MIPS procesoru. Je používán především pro výukové účely. Je implementován v programovacím jazyce Java, což má za výhodu, že je nezávislý na platformě. MARS simulátor podporuje 155 základních instrukcí, přibližně 370 pseudoinstrukcí a skoro 40 různých systémových volání. Uživatelské rozhraní distribuuje širokou škálou možností. Kromě editoru kódu a dalších základních nástrojů pro řízení simulace nabízí hned několik pokročilých nástrojů (predikátor instrukcí skoku, vizualizace cache paměti atd...). Má ovšem jednu velkou nevýhodu. MARS nedemonstruje provádění instrukcí na úrovni pipeline. K výuce detailní architektury MIPS jej použít nelze. Kromě toho, se ale jedná o skvělý simulátor, který jsem sám používal jako validátor správného provádění instrukcí. Taky jsem se jím inspiroval pří návrhu implementace paměti a omezení jednotlivých segmentů paměti.

5.1.1 Porovnání

MARS simulátor je robustní simulátor, který má za sebou několikaletou historii vývoje. Po stránce demonstrace exekuce programu se s ním nemohu srovnávat. Můj simulátor ale narozdíl od MARS simulátoru disponuje simulací na úrovni pipeline. Je tedy dle mého lepší na výuku architektury pipeline.



Obrázek 5.1: Simulátor MARS

5.2 QtMips

QtMips[8] (obrázek 5.2) je simulátor, který vznikl jako diplomová práce Karla Kočího na univerzitě ČVUT v Praze. Na simulátoru se stále aktivně pracuje a rozšiřuje se. QtMips je implementován v jazyce C++ a využívá QT knihovnu pro grafické uživatelské rozhraní. Zakládá si převážně na detailní vizualizaci pipeline. Skoro každá důležitá komponenta (Multiplexory, hodnoty registrů, obsah cache) je zde zobrazena. Kromě toho, simulátor podporuje širokou škálu instrukcí a obsahuje velké množství nástrojů. Tento simulátor bych býval považoval za nejlepší, s kterým jsem se setkal, ale obsahuje hned několik nedostatků. Často jsem se setkával s problémy UI, kdy se některé prvky nezobrazovaly a zalamovaly mimo viditelné okno. Další nevýhodou je i samotné provádění instrukcí. Některé instrukce jsou totiž špatně demonstrovány.

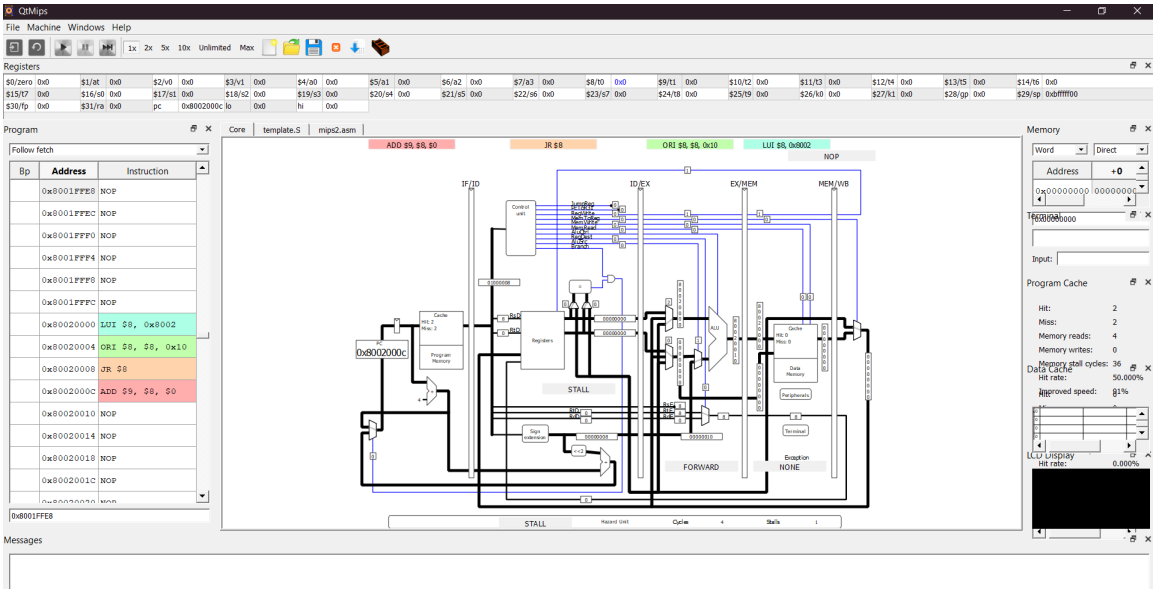
Na ukázce 5.2 je jednoduchý program, který by měl načíst adresu návěští `end` do registru `$t0` a pomocí instrukce `jr` na tuto adresu skočit. Instrukce `add` se sice načte do pipeline, ale po provedení skoku by měla být pipeline vyprázdněna.

```
.text
la $t0, end
jr $t0
add $t1, $t0, $zero
end:
```

V případě, že program zkusíme simulovat v QtMips, tak se instrukce správně načte do pipeline. Ovšem QtMips dělá tu chybu, že pipeline nevyprázdní a instrukce `add` v ní zůstane. Výsledkem je, že se do registru `$t1` uloží hodnota v registru `$t0`.

5.2.1 Porovnání

QtMips by měl být obecně lepší verzí mého simulátoru, jelikož demonstruje mnohem detailněji provádění programu na úrovni pipeline. Problém ovšem je, že obsahuje spousty chyb, které mohou vést k špatné demonstraci nejen pipeline, ale i celé exekuce programu.



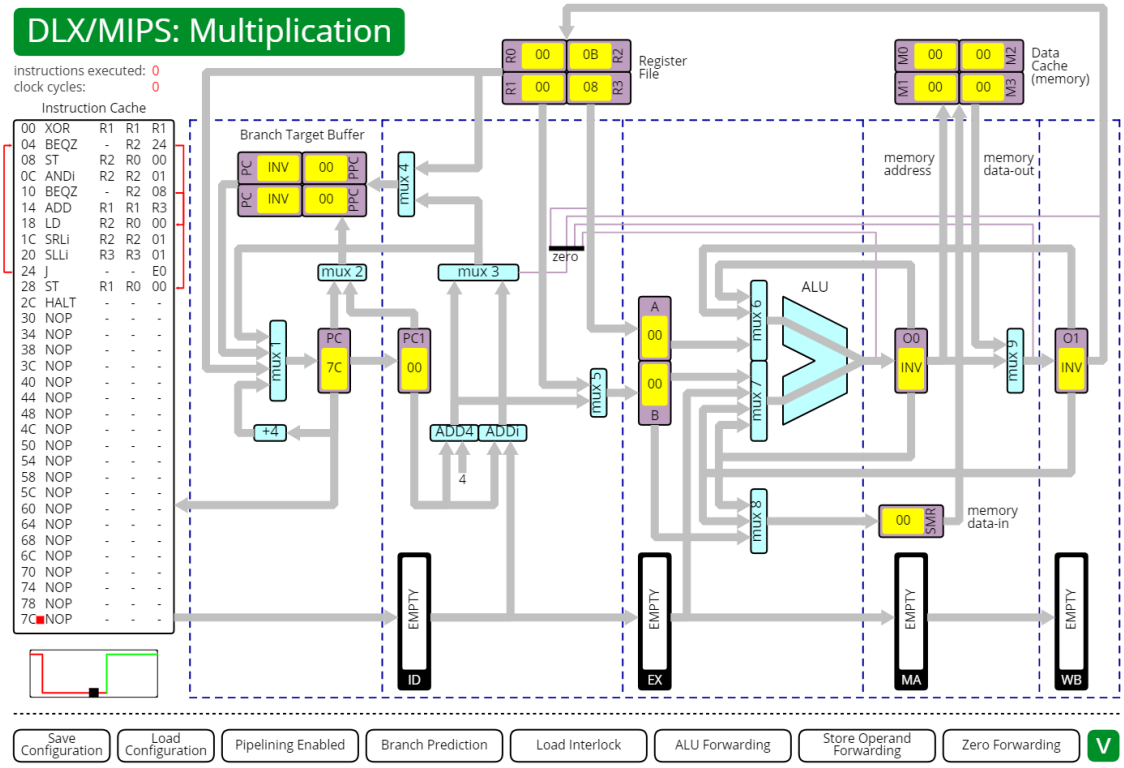
Obrázek 5.2: Simulátor QtMips

5.3 VivioJS DLX/MIPS

VivioJS DLX/MIPS[7] (obrázek 5.3) je webový simulátor MIPS pipeline. Je implementován pomocí VivioJs, což je javaScriptová implementace Vivio, která umožňuje tvorbu a ovládání animací. Narozdíl od ostatních simulátorů je VivioJS DLX/MIPS zaměřeno především na demonstraci pipeline. Umožňuje vypínat a zapínat různé části pipeline. Provádění instrukcí se demonstruje v reálném čase podle hodinového cyklu. Skvělou funkcionalitou je, že animace mohou být nejen zpomalovány a zrychlovány, ale i přehrávány pozpátku. Editace kódu je naopak naprosto katastrofální. Zpracovatelný program je velmi omezený, práce s pamětí prakticky není vůbec podporována, systémová volání použít taky nelze a dají se používat pouze čtyři registry. Program nelze psát ručně, ale musí se pro každý řádek vybírat ze seznamu dostupných instrukcí.

5.3.1 Porovnání

VivioJS DLX/MIPS je simulátor, který byl vyvíjen s cílem demonstrovat pipeline na omezeném počtu instrukcí. Z tohoto důvodu vyniká ve výuce fungování a vztahů mezi instrukcemi. Oproti mnou implementovaném simulátoru však zaostává ve všech ostatních ohledech. Můj simulátor podporuje mnohem více instrukcí, práci s pamětí, všemi registry a systémová volání. Zároveň má moje implementace mnohem lepší uživatelské rozhraní.



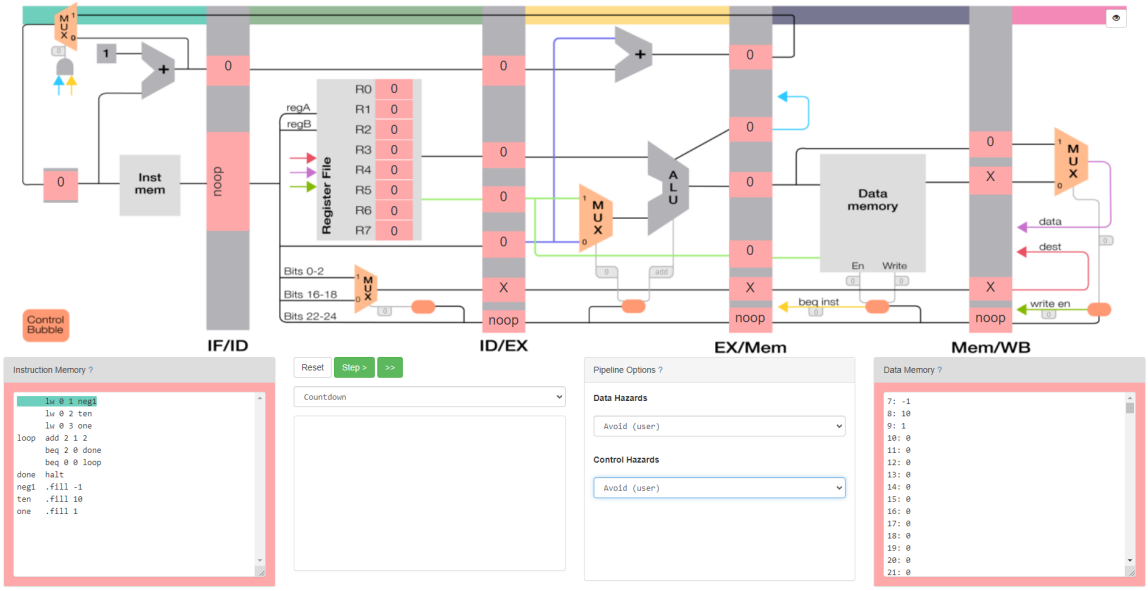
Obrázek 5.3: Simulátor VivioJS DLX/MIPS

5.4 EECS 370 pipeline simulator

EECS 370 pipeline simulator[1] (obrázek 5.4) je simulátor pipeline, který vznikl na univerzitě Mishigen v rámci kurzu EECS 370. Simulátor demonstruje chování procesoru na úrovni pipeline. Simulátor zobrazuje stav komponent a obsah vyrovnávacích pamětí na konci každého cyklu. Zároveň umožňuje nastavit řešení datových a kontrolních hazardů. Dle mého názoru je tento simulátor nejlepší webový simulátor na pochopení základního fungování pipeline s forwarding a control jednotkou. Též podporuje práci s pamětí. Největší slabinou tohoto simulátoru je použití nestandardní syntaxe postrádající dokumentaci. Simulátor naštěstí obsahuje ukázkové programy, z kterých jde syntaxe do určité míry pochopit. Dále simulátor obsahuje pouze sedm registrů a nepodporuje velkou řadu instrukcí.

5.4.1 Porovnání

Tímto simulátorem jsem se lehce inspiroval při tvoření pipeline. Každopádně si myslím, že můj simulátor je po všech stránkách lepší. Má lepší uživatelské rozhraní a korektně demonstruje MIPS syntaxi, paměť a registry. Jediné, v čem tento simulátor může konkurovat je, že lépe zobrazuje obsah vyrovnávacích pamětí.



Obrázek 5.4: Simulátor pipeline EECS 370

Kapitola 6

Závěr

Cílem této práce bylo implementovat webový simulátor, který demonstruje činnost MIPS procesoru na úrovni zřetězené linky pro výukové potřeby. To bylo úspěšně splněno bez nutnosti přistoupit ke kompromisu v podobě zmenšení instrukční sady nebo zjednodušení demonstrace pipeline.

Po teoretické části jsem pokryl všechny důležité body funkcionality architektury MIPS. V návrhu implementace jsem se opíral o zmíněnou teorii a pokusil se podle ní obecně popsat návrh implementace simulátoru MIPS. Při samotné implementaci jsem se řídil mým návrhem implementace s detailnějším popisem řešení určitých problémů, na které jsem v návrhu implementace upozorňoval. K simulátoru jsem vytvořil sadu jednoduchých příkladů. S výsledkem samotného simulátoru jsem spokojen. Podporuje většinu potřebných instrukcí. Paměť je navrhnuta ideálně, Simulátor umožňuje práci se základními i speciálními registry. Hazard jednotku i Forwarding jednotku lze deaktivovat. Činnost procesoru je demonstrována na úrovni fází pipeline. Uživatelské rozhraní je odděleno od simulačního jádra. Simulátor byl implementován tak, aby ho bylo možné snadno rozšířit. Přidání nové instrukce, přesunutí vyhodnocení podmíněného skoku do jiné fáze nebo rozšíření množství zobrazených informací o stavu pipeline v uživatelském rozhraní zabere pár málo řádku kódu. Nad rámec zadání byla implementována podpora systémových volání a možnosti vstupu a výstupu dat, což umožňuje vykonat složitější programy.

Ačkoli jsem se snažil implementovat co nejvíce funkcionalit, navržený simulátor lze dále rozvíjet a možných rozšíření je hned několik. Rád bych simulátor rozšířil o alternativní způsob zobrazení instrukcí v pipeline, predikci skoku a zobrazení více informací o stavu částí pipeline. Dalšími možnými rozšířeními, by mohla být implementace koprocesorů a překladu pseudoinstrukcí.

Literatura

[1] Beaumont, J. *EECS 370: Intro to Computer Organization* [online]. 2022 [cit. 2022-05-01]. Dostupné z: https://eecs370.github.io/.

[2] Chiang, P. *MIPS Pipeline* [online]. 2012 [cit. 2022-05-01]. Dostupné z: https://eecs.oregonstate.edu/research/vlsi/teaching/ECE472_FA12/chapter4_pipelining_END_FA11.pdf.

[3] Herbert, C. *RISC-V* [online]. 2014 [cit. 2022-05-01]. Dostupné z: https://en.wikipedia.org/wiki/RISC-V.

[4] III, C. W. K. *MIPS and Memory* [online]. LibreTexts, červenec 2020 [cit. 2022-01-24]. Dostupné z: https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Introduction_To_MIPS_Assembly_Language_Programming_(Kann)/02%3A_First_Programs_in_MIPS_Assembly/2.02%3A_MIPS_and_Memory.

[5] Inductiveload. *Data Forwarding (One Stage)* [online]. 2009 [cit. 2022-05-01]. Dostupné z: https://commons.wikimedia.org/wiki/File:Data_Forwarding_(One_Stage).svg.

[6] Inductiveload. *MIPS Architecture (Pipelined)* [online]. Wkimedia commons, leden 2009 [cit. 2022-01-24]. Dostupné z: https://commons.wikimedia.org/wiki/File:MIPS_Architecture_(Pipelined).svg.

[7] Jones, J. *VivioJS DLX/MIPS* [online]. 2017 [cit. 2022-05-01]. Dostupné z: https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/.

[8] Kočí, K. *QtMips - simulátor architektury MIPS* [online]. 2018 [cit. 2022-05-01]. Dostupné z: https://cw.fel.cvut.cz/wiki/courses/b35apo/documentation/qtmips/start.

[9] Křivánková, V. *Uživatelské rozhraní* [online]. 2016 [cit. 2022-05-01]. Dostupné z: https://wikisofia.cz/wiki/U%C5%BEivatelsk%C3%A9_rozhran%C3%AD.

[10] L., P. D. A. . H. J. *Computer Organization and Design: The Hardware/Software Interface*. 4. vyd. Morgan Kaufmann, 2008. ISBN 9780123744937.

[11] Parthasarathi, D. R. *Computer Architecture* [online]. 2018 [cit. 2022-01-24]. Dostupné z: http://www.cs.umd.edu/~meesh/411/CA-online/index.html.

[12] Somani, A. *Computer Organization and Assembly Level Programming* [online]. 2008 [cit. 2022-05-01]. Dostupné z: http://class.ece.iastate.edu/arun/Cpre381/index.html.

[13] T, N. *Pipelining in Computer Architecture* [online]. 2019 [cit. 2022-05-01]. Dostupné z: https://binaryterms.com/pipelining-in-computer-architecture.html.

[14] Vollmar, K. *MARS (MIPS Assembler and Runtime Simulator)* [online]. 2016 [cit. 2022-05-01]. Dostupné z: http://courses.missouristate.edu/kenvollmar/mars/papers.htm.

[15] Wikibooks. *MIPS Assembly* [online]. Wikibooks, prosinec 2017 [cit. 2022-01-24]. Dostupné z: https://en.wikibooks.org/wiki/Category:Book:MIPS_Assembly.