Dokumentace k projektu
Assembler via JavaScript

David Nápravník

2018

1 Zadání

Mým úkolem bylo naprogramovat emulátor, jenž bude simulovat chování assemblerovského kódu, včetně jeho překladu do strojového kódu, práce s virtuální pamětí a registry.

2 Program

2.1 Specifikace

Program je napsán v jazyce **JavaScript** (verze: ECMAScript 6) s podporou drtivé většiny moderních prohlížečů. Tudíž aplikace je multiplatformní, včetně mobilních platforem.

Časová náročnost je tak vysoká, že programi obsahující více než 1000 instrukcí (do toho se počítají i průchody cyklů) bude probíhat až 1 vteřinu.
Ohledně pameťi by program neměl spotřebovat více než 50 MB operační paměti v prohlížeči chrome, čistě můj program pak zabírá kolem 1 MB operační paměti. Má to za následek právě to, že javascript si o poli drží více údajů, než pole samotné obsahuje dat.

Na vstupu program dostane **MIPS**ovský assembler, který přečte z pole textarea. Výstupem je pak pamět a registry s výslednými hodnotami.

2.2 Uživatelská příručka

Jedná se o webovou responzivní stránku, skládající se ze tří částí.

- Vstup
- Ovládací prvky
- Výstup (Paměť a registry)

Uživatel zadá (nebo použije ukázkový hard-coded) kód assembleru MIPS do levého textového pole (resp. textarea).
Poté spustí buď pouhé přeložení assebleru do paměti, nebo jej „spustí“ čímž se kód nejen přeloží, ale také vykoná. (Volá se funkce „insertCodeIntoMemory()“ pro překlad a funkce „runEmulator() pro spuštění emulace“)
Tlačítko vymazat pak jednoduše vynuluje paměť. (paměť se vynuluje i při použití tlačítka spustit) (Volá se funkce „resetEmulator()“)

Aby se předešlo nekonečným smyčkám, program má vlastní counter vykonaných příkazů. Jakmile se jeho hodnota přehoupne přes 1000, program spadne a ohlásí chybu.
Podobně pak program odmítá i assembler delší než 64 řádků (příkazů), právě protože by se neměli kam ukládat. Pokud chcete zadat kód s více než 32 řádky, bylo by vhodné posunout i ukazatel (global pointer „$gp“) na pozici kde nebude strojový kód.

2.3 Algoritmy a Rozbor kódu

2.3.1 převody mezi soustavami

Program pracuje s čísly ve dvojkové, desítkové a šestnáctkové soustavě. Pro převod mezi těmito soustavami slouží šest funkcí.

Při volání funkce se musí předávat informaci o tom, zda se převádí znaménkově nebo bezznaménkově. Při převodu do binárky je poté ještě zapotřebí předat údaj jak dlouhý má výsledek být, jelikož výsledek se ukládá jako string.

Celkem problematické bylo manipulovat s čísly v JavaScriptu. Jelikož tento jazyk je dynamicky typovaný, tudíž při každém převodu je nutné z proměnné vytvořit **32-bitový** integer.

Podívejme se na následující kód:

```
1   function bin2Dec(bin,signed = true){
2       var uint = parseInt(bin, 2);
3       if(bin.length<32){
4           uint <<= 32 - bin.length;
5           uint >>= 32 - bin.length;
6       }
7       if(signed){
8           if(uint>2147483648){
9               return uint-4294967296;
10          } else {
11              return uint;
12          }
13      } else {
14          return parseInt(bin, 2);
15      }
16  }
```

Jedná se o funkci jež převádí čísla z dvojkové do desítkové soustavy.
Většina kódu je tu pouze proto, že JavaScript v základu převádí binární čísla bezznaménkově, tudíž si to musíme dodělat sami.
Prvně doplníme znaménka po celé délce (32 bitů). Na konci nám vyleze bezznaménkové číslo dlouhé 32-bitů. Pokud je větší než 2^{31} (2147483648) tak to znamená, že na první bitu je jednička $->$ číslo je záporné a po odečtení 2^{32} (4294967296) se dostaneme na správný výsledek, který pak vracíme.
Celkově tedy pracujeme jak s kladnými tak i zápornými čísly, kde je ale pouze jedna nula (posloupnost samých jedniček bude rovna „-1“)

2.3.2 překlad assembleru do strojového kódu

Překlad se provádí po řádcích, tudž co řádek to instrukce. I kdyby byl řádek prázdný, nebo pouze s komentářem, bude přeložen jako instrukce která nic nedělá (instrukce „noop“).

Textový řetězec, je na začátku rozdělen do částí: instrukce, parametry a komentář. K instrukci se pomocí switche najde odpovídající šablona pro její překlad a umístění parametrů.
Oficiální tabulku lze najít na stránce:opencores.org

Na konci dostaneme string s binárně zapsanou instrukcí. Tu poté převedeme do soustavy šestnáctkové, rozčtvrtíme (32 bitů na čtyři 8 bity) a uložíme do paměti (myšleno pole s názvem „memory“).

2.3.3 vykonávání strojového kódu

Pro vykonání instrukce, se musí nejdříve zjistit o kterou instrukci se jedná. A jelikož jsou tři typy instrukcí, musíme si je předem rozdělit. Ony skupiny vypadají takto:

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	address				

Další podrobnosti o formátování onstrukcí zde: MIPS_Green_Sheet.pdf[5]

Poté se na dvouvrstvém switchy hledá odpovídající formát a opcode. Po vykonání se ukazatel „pc“ posune o 4 bajty dále. A to i v případě, že se předtím použil skok pomocí *jumperu* jinam.

3 Reprezentace dat

3.1 Vstupní data

Vstupní data se čtou z HTML objektu „textarea“ s id='sourceCode'.
Je požadováno, aby každý příkaz byl na právě jednom řádku.
Přijímá pouze assembler typu MIPS.
přehled podporovaných příkazů zde: opencores.org

3.2 Výstupní data

Výstup programu se ulkádá do virtuální paměti, která je poté zobrazena vpravo (dole na mobilu). V klasickém případě se v horní polovině paměti uloží program a v dolní polovině výstup assembleru. Výstup je v **šestnáctkové soustavě** (po najetí mysí na příslušnou část aměti se zobrazí i v dvojkové a desítkové soustavě).

4 Sada testovacích příkladů

Pro testování je hned po spuštění stránky předpřipraven kód pro jednoduché matematické operace a hrabání se v paměti.

```
1       ori     $t0,  $zero,  2       # vytvor 2
2       ori     $t1,  $zero,  3       # vytvor 3
3       mult    $t0,  $t1             # 2 * 3 = 6
4       mfhi    $t2                   # nacti vysledek nasobeni
5       add     $t3,  $t1,  $t2       # 3 + 6 = 9
6       addi    $t4,  $t3,  1         # 9 + 1 = 10
7       sw      $t4,  0($gp)          # uloz 10
8       addi    $gp,  $gp,  4         # posun ukazatel
9       break
```

Každý řádek má na konci komentář o tom co dělá. Nejpodstatnější je však výsledek uložený v paměti na adrese 128 - 131. Nalezneme zde „00 00 00 0a“ což je po převodu z šestnáctkové soustavy do desítkové číslo 10 a tím pádem máme ověřenou funkčnost

programu, pro operaci or, sčítání, násobení a ukládání do paměti.

Dalšími testovacími daty jsou:

```
1       ori     $t0,  $zero,  -2      # vytvor -2
2       ori     $t1,  $zero,  3       # vytvor 3
3       mult    $t0,  $t1             # -2 * 3 = -6
4       mfhi    $t2                   # nacti vysledek nasobeni
5       add     $t3,  $t1,  $t2       # 3 + (-6) = -3
6       addi    $t4,  $t3,  1         # -3 + 1 = -2
7       sw      $t4,  0($gp)          # uloz -2
8       addi    $gp,  $gp,  4         # posun ukazatel
9       break
```

Výsledkem je číslo -2 (vyjádřené jakožto „ff ff ff fe“)

```
1       ori     $t0,  $zero,  1       # vytvor 1
2       ori     $t1,  $zero,  160     # vytvor 160
3       beq     $gp,  $t1,    5       # skoc o 5 radek
4       addi    $t0,  $t0,  1         # i + 1 = i+1
5       sw      $t0,  0($gp)          # uloz cislo
6       addi    $gp,  $gp,  4         # posun ukazatel
7       j       8                     # skoc na radek 3 (3*4-4)
8       break
```

Výsledkem bude pole o 8mi položkách (od dvojky po devítku)

Nyní vyzkoušíme případ, kdy by program měl záměrně spadnout. Např. pro nekonečnou smyčku.

```
1       j 0
2       break
```

Program správně vyhodí chybu
Nebo pro odkazování mimo paměť

```
1       ori     $gp, $zero, 256     # uloz 256 do global pointeru
2       sw      $gp, 0($gp)         # zkus neco ulozit
3       break
```

Program správně vyhodí chybu

5 Závěr

Jelikož se jedná o interpretovaný skriptovací jazyk, časová náročnost převýšila mé očekávání, ačkoliv se jedná o celkem triviální operace, program je schopen rychle reagovat pouze na programy pod 1000 instrukcí (do toho se počítají i průchody cyklů), nad tuto hodnotu program dokáže „přemýšlet“ i nad jednu vteřinu, což je uživatelsky nepříjemná situace, kdy program něco dělá a uživatel pořád nevidí ani výsledek, ani průběh.

5.1 Co nebylo doděláno

Program by se dal rozšířit i pro práci s pamětí, jež nemá fixní velikost, narozdíl od paměti přesně dané rozsahem 256 bajtů. Toho by se dalo dosáhnout dynamickým polem (to není v JavaScriptu problém) a následně automatickym přesouváním global pointeru.

5.2 Co bylo doděláno

Bylo doimplementováno přetékání paměti a taktéž je kontrolováno zacyklení programu (pro více než 1000 instrukcí). Dále byla k těmto chybá, dodělána příslušná vyskakování chybová hlášení.

6 Externí odkazy

Reference

[1] Všeobecná stránka s příkazy a vysvětlením co přesně dělají
mrc.uidaho.edu

[2] Inspirativní web, ze kterého jsem převzal nápad (používa 8-bit a instrukce NASM)
schweigi.github.io/assembler-simulator

[3] Stránka o javascriptu, pomocí níž jsem implementoval bitové posuny a další nestandardní operace:
developer.mozilla.org

[4] Užitečná stránka pro testování malých části javascriptu. Velice užitečné při testování funkcí pro převod mezi soustavami:
jsfiddle.net

[5] Ačkoliv se jedná o pěkný přehled instrukcí, nejdůležitější je tabulka s formáty (R, I, J) a přehled registrů:
MIPS_ Green_ Sheet.pdf

[6] Stránka s přehledem binárního zápisu instrukcí (vzhled strojového kódu)
opencores.org