**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**

Кафедра ЕОМ



**Пояснювальна записка**

**до курсової роботи**

З дисципліни

“Архітектура комп’ютерів”

На тему:

«Проектування комп’ютера»

Варіант №8

Виконав:

Ст. гр. КІ-304

Кісіль В. І.

Прийняв:

Жолубак І.М.

Львів – 2023

**Анотація**

Курсова робота складається з трьох частин: перша частина являє собою розробку програми, яка перетворює вхідну програму на мові асемблер в мову машинних кодів. В цій частині проводиться аналіз всіх команд які потрібно реалізувати, а також розробка тестів для тестування даної програми.

В другій частині здійснюється розробка симулятора, який може відсимулювати будь яку машинного коду, а також розробка тестів для тестування розробленої програми.

Третє частина являє собою розробку асемблерної програми для перевірки коректності роботи моделі спрощеного комп’ютера. Програма має бути ефективною:

- не перевищувати 50 рядків;

- і не має виходити за 1000 інструкцій для будь яких вхідних даних.

Зміст

[**1.** **Конкретизовані та розширені вихідні дані для проектування** 24](#_Toc531108265)

[**2.** **Аналіз основних принципів побудови комп’ютерів.** 26](#_Toc531108266)

[2.1. Система команд 29](#_Toc531108267)

[2.2. Способи адресації 30](#_Toc531108268)

[**3.** **Аналіз принципів побудови спрощеного комп’ютера** 32](#_Toc531108269)

[3.1. Структура СК до модифікації 32](#_Toc531108270)

[3.2. Система команд 34](#_Toc531108271)

[3.3. Структура СК після модифікації 37](#_Toc531108272)

[3.4. Потактове виконання команд 40](#_Toc531108273)

[**4.** **Результат роботи** 43](#_Toc531108274)

[1. Програма для перевірки операторів CMBSR, BSF. 43](#_Toc531108275)

[2. Перевірка виконання базово-індексної адресації (команди LOAD, SAVE). 43](#_Toc531108276)

[3. Пошук максимального числа в масиві (команди JMA, JMNBE). 24](#_Toc531108277)

[4. Перевірка виконання логічних команд (SHL, ROL, NOT). 32](#_Toc531108278)

[5. Перевірка виконання команд SUB, XSUB, DIV. 33](#_Toc531108279)

[**Висновок** 35](#_Toc531108280)

[**Список використаної літератури** 36](#_Toc531108281)

[Додаток А.Вихідний код транслятора. 37](#_Toc531108282)

[Додаток Б.Вихідний код симулятора. 45](#_Toc531108283)

1. **Конкретизовані та розширені вихідні дані для проектування**

Кожен варіант складається з наступних завдань:

1. 8 додаткових інструкцій без використання регістрів стану:

3 – арифметичні

3 – логічні

2 – керування

1. 3 додаткові інструкції з використання регістрів стану.
2. Передбачити на власний вибір 3 інструкцій (з розроблених в п. 1, 2), які підтримують додатковий тип адресації.

Таблиця 1 – Варіант №5

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **№** | **Розряд-**  **ність** | **Арифметичні** | | | **Логічні** | | | **Керування** | | **Прапорці** | | | | **Адре-**  **сація** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **1** | **2** | **3** |
| 8 | 1 | 3 | 7 | 11 | 3 | 9 | 11 | 1 | 12 | ZF | 1 | 2 | 3 | 8 |

1. Визначити формати команд згідно розрядності шини даних, розміру пам’яті та регістрового файлу.

Таблиця 2 - Розрядність

|  |  |  |  |
| --- | --- | --- | --- |
| № | Розрядність шини даних | Розмір пам’яті  Байт | Розмір регістрового файлу(к-сть регістрів) |
| 1 | 16 | 256 | 8 |

1. Реалізація додаткових команд. Необхідно реалізувати 8 додаткових команд. Серед них 3 арифметичні, 3 логічні та 2 команди керування згідно варіанту. Команди не мають повторюватися.

Таблиця 3 – Арифметичні команди

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 3 | DIV regA regB destReg | Беззнакове ділення: destReg=regA/regB |
| 7 | SUB regA regB destReg | Віднімання: destReg=regA-regB |
| 11 | XSUB regA regB destReg | Віднімання і обмін операндів місцями: destReg=regAregB |

Таблиця 4 – Логічні команди

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 3 | SHL regA regB destReg | Логічний зсув віліво: destReg=regA << regB |
| 9 | ROL regA regB destReg | Циклічний зсув вправо: destReg= regA >> regB |
| 11 | NOT regA destReg | Логічне побітове НЕ: destReg=~ regA |

Таблиця 5 – Команди керування

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 1 | JMA regA regB offSet | Беззнакове більше if (regA> regB) PC=PC+1+offSet |
| 7 | JMNBE regA regB offSet | Беззнакове не менше/рівно if (regA!=< regB) PC=PC+1+offSet |

1. Реалізувати додатковий спосіб адресації. Передбачити, що 3 інструкції підтримують інший вид адресації згідно варіанту. Визначення операндів, які підтримують інший спосіб адресації узгодити з викладачем.(крім безадресної)

Примітка: безадресний варіант передбачає створення стеку та реалізацію 2 додатковий команд наведених в таблиці.

Таблиця 6 – Додаткова адресація

|  |  |
| --- | --- |
| **№** | **Адресація** |
| 8 | Базово – індексна (розробити IR та BR – базовий регістр) передбачити команду чи дириктиву встановлення базового регістру, індексний регістр має інкрементуватися/декрементуватися після кожного звернення\* |

1. Регістри стану: CF –регістр переносу, SF – регістр знаку, ZF – регістр 0.

Таблиця 7 – регістр переносу CF

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 1 | CMP regA regB | Порівняти regA regB і встановити прапорці |
| 2 | BSR regA destReg | Побітове скануання в зворотньому напрямку (від старших до молодших) regA в пошуках біта з 1, повертає номер позиції в destReg. Якщо 1 знайдено ZF =1, інакше ZF=0 |
| 3 | BSF regA destReg | Побітове сканування в прямому (від молодшого до старших) напрямку regA в пошуках біта з 1, повертає номер позиції в destReg. Якщо 1 знайдено ZF=1, інакше ZF=0 |

1. **Аналіз основних принципів побудови комп’ютерів.**

**CISC** ([англ.](http://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) Complex Instruction Set Computer — комп'ютер зі складним набором команд) — це [архітектура системи команд](http://uk.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4), в якій більшість команд є комплексними, тобто реалізують певний набір простіших інструкцій [процесора](http://uk.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BD%D1%82%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80) або шляхом зіставлення з кожною CISC-командою певної [мікропрограми](http://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B0), або принаймні можуть бути зведені до набору таких простих інструкцій. Крім того, ознаками CISC-архітектури можна вважати також наявність великої кількості [методів адресації пам'яті](http://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%B8_%D0%B0%D0%B4%D1%80%D0%B5%D1%81%D0%B0%D1%86%D1%96%D1%97_%D0%BF%D0%B0%D0%BC%27%D1%8F%D1%82%D1%96) з можливістю безпосередньої роботи з операндами в основній пам'яті комп'ютера. Тобто, CISC-архітектури відносяться, як правило, до класу [двохадресних](http://uk.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4).

Архітектури з комплексними наборами команд, розвиток яких припав на кінець 60-х — 70-ті роки пропонували програмісту досить різноманітний набір порівняно високорівневих інструкцій машинної мови, таких, наприклад, як «виклик підпрограми» або «відняти одиницю та перейти, якщо результат ненульовий», а також велику кількість способів звертання до операндів в пам'яті для полегшення роботи зі складними [структурами даних](http://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85). В ті часи, за відсутності повноцінних [мов програмування](http://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)високого рівня та відповідних [компіляторів](http://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%96%D0%BB%D1%8F%D1%82%D0%BE%D1%80), така апаратна підтримка високорівневого інструментарію програмування могла підвищити [продуктивність праці](http://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D1%96%D1%81%D1%82%D1%8C_%D0%BF%D1%80%D0%B0%D1%86%D1%96) програміста. До того ж, програма, складена з таких команд займала небагато в пам'яті комп'ютера.

Типовими прикладами CISC-архітектур були системи [VAX](http://uk.wikipedia.org/wiki/VAX), [PDP-11](http://uk.wikipedia.org/wiki/PDP-11), [IBM System/360](http://uk.wikipedia.org/wiki/IBM_System/360), сімейства [мікропроцесорів](http://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80) [Motorola 68000](http://uk.wikipedia.org/wiki/Motorola_68000) та [Intel x86](http://uk.wikipedia.org/wiki/Intel_x86).

Але з появою високорівневих мов та оптимізуючих компіляторів, розвитком електроніки, який спричинив здешевлення комп'ютерної пам'яті виявилось, що використання високорівневих машинних команд суттєво обмежує можливості до оптимізації програми, підвищення її швидкодії.

Зокрема, складні команди потребували багато часу на процедуру декодування, потребували багато апаратурних ресурсів для реалізації, що негативно відображалось на загальній швидкодії та складності системи. Далі, наявність спеціальних команд, таких як «виклик підпрограми» не завжди виправдовувала себе, в багатьох випадках доцільніше було б замість такої загальної команди використати набір елементарніших інструкцій, які в результаті спричиняли б виконання меншої кількості обчислень процесором. Це стосувалось і обчислювальних команд, які підтримували роботу з операндами в повільній пам'яті, що далеко не завжди давало оптимальний результат. Набагато ефективніше було б відокремити процедури роботи з пам'яттю (завантаження та збереження операндів) від проведення обчислень, що дало б можливість оптимізувати процедури звертання до запам'ятовуючого пристрою. Зрозуміло, що ці оптимізації повинні були виконуватись вже компіляторами з мов високого рівня, які до того часу досягли досить високого рівня функціональності.

Ці та інші проблеми CISC-архітектур призвели до створення в 80-ті роки [RISC](http://uk.wikipedia.org/wiki/RISC)-архітектур (від [англ.](http://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) Reduced Instruction Set Computer — комп'ютер зі скороченим набором команд), які вибудовують прямо протилежну модель системи команд з максимальним спрощенням семантики машинної команди, зведенням її до елементарної, мінімізацією методів адресації пам'яті тощо та принциповою орієнтацією на мови високого рівня й оптимізуючі компілятори з них, аніж на програмування безпосередньо в машинних мовах.

Сучасні CISC-архітектури, такі як останні втілення сімейства процесорів x86, хоч і відповідають CISC-концепції на рівні архітектури системи команд, але всередині процесора реалізують якраз пристосованішу до сьогоднішніх реалій RISC-модель, трансформуючи потік CISC-команд в процесі виконання в набори з простіших RISC-мікрооперацій, які й виконуються процесором.

Для виконання задачі на комп’ютері необхідно:

* + забезпечити вибірку команди програми із його пам’яті в заданій послідовності, організувати звернення до неї за відповідними адресами;
  + забезпечити розпізнавання типів виконуваних операцій;
  + організувати звернення до пам’яті за відповідними адресами для вибірки необхідних для виконання кожної команди даних;
  + організувати виконання над даними операцій відповідно до вказівок команд;
  + запам’ятати результат обчислень.

Комп'ютер виконує кожну команду як послідовність простих операцій:

1. Вибірка чергової команди із основної пам'яті.
2. Визначення типу вибраної команди, тобто її дешифрування.
3. Визначення адрес даних, необхідних для виконання цієї команди.
4. Виконання операцій пересилання даних (зчитування даних із пам'яті в регістри процесора).
5. Виконання операції відповідно до її коду в полі коду операції команди.
6. Визначення адрес, за якими запам'ятовуються результати.
7. Запам'ятовування результатів.
8. Підготовка до виконання наступної команди, тобто обчислення її адреси.

Для процесора комп'ютера із складною системою команд характерні наступні особливості:

* + виконання команди за багато тактів, оскільки для цього потрібно здійснити багаторазові операції звернення до основної пам'яті та до програмно-доступних регістрів процесора;
  + орієнтація АЛП на виконання великої кількості операцій, що пов'язано з розширеним складом системи команд;
  + складна система розпізнавання команди, що пов'язано з великою кількістю методів адресації та великою кількістю форматів команд різної розрядності;
  + програмне дешифрування команд з метою зменшення затрат обладнання;
  + складна організація конвеєризації виконання команд, що пов'язано, в першу чергу, з різнотипністю їх виконання;
  + орієнтація структури на виконання команд типу регістр-пам'ять та пам'ять-пам'ять.

Основні елементи процесора - арифметико-логічний пристрій, пристрій керування і регістрова пам'ять або, як її ще називають, надоперативний запам'ятовуючий пристрій. До складу регістрової пам'яті, в свою чергу, входять наступні вузли - програмний лічильник, регістри: адреси, команди, даних, слова стану програми, а також регістровий файл, який складається з програмно доступних регістрів.

Структура регістрової (надоперативної) пам'яті процесора складається з регістрів спеціального та зального призначення. До регістрів спеціального призначення належать:

* + регістри адреси (РгА);
  + регістри команд (РгК);
  + програмний лічильник(ПЛ)
  + регістри даних (РгД).

РгА зберігає адресу даного або команди при зверненні до основної пам'яті. РгД зберігає операнд при його запису або зчитуванні з основної пам'яті. В ролі операнда може бути дане, команда або адреса. РгК зберігає команду після її зчитування з основної пам'яті. ПЛ підраховує команди та зберігає адресу поточної команди. Комп'ютер з архітектурою Джона фон Неймана має один програмний лічильник.

2.1. Система команд

Різноманітність типів даних, форм представлення та опрацювання, необхідні дії для обробки та керування ходом виконання обчислень призводить до необхідності використання різноманітних команд – набора команд.

Кожен процесор має власний набір команд, який називається системою команд процесора.

Система команд характеризується трьома аспектами:

* + формат,
  + способи адресації,
  + система операцій.

Форматом команди – є довжина команди, кількість, розмір, положення, призначення та спосіб кодування полів. Команди мають включати наступні види інформації:

* + тип операції, яку необхідно реалізувати в даній команді (поле команду операції - КОП);
  + місце в пам’яті звідки треба взяти перший операнд (А1);
  + місце в пам’яті звідки треба взяти другий операнд (А2);
  + місце в пам’яті куди треба помістити результат (А3).

Кожному з цих видів інформації відповідає своя частина двійкового слова – поле. Реальна система команд зазвичай має команди декількох форматів, тип формату визначає КОП.

Команда в комп'ютері зберігається в двійковій формі. Вона вказує тип операції, яка має бути виконаною, адреси операндів, над якими виконується операція, та адреси розміщення результатів виконання операції. Відповідно до цього команда складається з двох частин, коду операції та адресної частини.

* 1. . Способи адресації

Варіанти інтерпретації бітів (розрядів) поля адреси з метою знаходження операнда називаються способами адресації. Коли команда вказує на операнд, він може знаходитись в самій команді, в основній або зовнішній пам'яті чи в регістровій пам'яті процесора. За роки існування комп'ютерів була створена своєрідна технологія адресації, яка передбачає реалізацію різних способів адресації, чому послужило ряд причин:

* + забезпечення ефективного використання розрядної сітки команди;
  + забезпечення ефективної апаратної підтримки роботи з масивами даних;
  + забезпечення задання параметрів операндів;
  + можливість генерації великих адрес на основі малих.

Існує велика кількість способів адресації. Розглянемо п’ять основних способів адресації операндів в командах.

**Пряма** – в цьому випадку адресне поле зберігає адресу операнда. Її різновидом є пряма регістрова адресація, яка адресує не комірку пам’яті а номер регістру.

**Безпосередня** – в поле адреси команди поміщається не адреса, а сам операнд.

**Непряма** – в полі адреси команди зберігається адреса комірки пам’яті в якій знаходиться адреса операнда. Такій спосіб дозволяє оперувати з адресами як з даними. Різновид непряма-регістрова адресація, адреса адреси зберігається в регістрі загального призначення.

**Відносна** – адреса формується, як сума з двох доданків: бази, яка зберігається в спеціальному регістрі чи в одному з регістрів спеціального призначення, та зміщення, яке задається в полі адреси команди. Різновид індексна та базова індексна. При індексній замість базового регістра є індексний, який автоматично модифікується (зазвичай збільшується на 1). Базова-індексна адресація формується адреса як сума трьох доданків: бази, індексу та зміщення.

**Безадресна** – поле адреси в команді відсутнє. Адреса операнда, або немає змісту або є по замовчуванню(наприклад дії на спеціальним регістром - акумулятором). Безадресні команди неможливо використати для інших регістрів чи комірок пам’яті. Одним з різновидів безадресної адресації є використання стеку.

В команду вводяться спеціальні ознаки з тим, щоб пристрій керування міг розпізнати використаний спосіб. Це можуть бути додаткові розряди в команді, або для різних типів команд закріплюватись різні способи адресації.

1. **Аналіз принципів побудови спрощеного комп’ютера**
   1. Структура СК до модифікації



Рис. 3.1. Функціональна схема СК до модифікації

В спрощеному комп’ютері (СК) в пам’яті зберігаються, як дані так і інструкції. Кожна інструкція закодована числом. Це число складається з декількох полів: поле назви команди чи код операції (КОП) та полів операндів. В СК є два види пам’яті: загальна пам’ять, та регістрова пам’ять. В загальній пам’яті зберігаються інструкції програми та дані над якими оперують інструкції. В регістровий пам’яті зберігаються дані над якими виконуються інструкції. У реальних комп’ютерах регістрова пам’ять є малою за розмірами та швидкою, працює на швидкості ядра процесора, загальна пам’ять є великою за розміром, але набагато повільніша за ядро процесора. Регістрова пам’ять підтримує лише пряму адресацію, загальна пам’ять підтримує декілька типів адресації.

У СК є 8 регістрів по 32 розряди, пам’ять складається з 65536 слів по 32 розряди. Отже СК є 32 розрядним комп’ютером. Він підтримує 8 інструкцій, кожна з яких розписана нижче. У СК є спеціальний регістр лічільник команд (ЛК).

За прийнятою домовленістю 0вий регістр завжди містить 0 (це не обмовлено апаратними вимогами проте асемблерна програма ніколи не має змінювати значення 0ого регістра, який ініціалізуються 0 ).



Рис.3.2. Формат СК

СК підтримує 4 формати інструкцій. Біти 31-25 не використовує жодна інструкція тому вони завжди мають дорівнювати 0.

Інструкції R-типу (add, nand):

біти 24-22: код операції

біти 21-19: reg A

біти 18-16: reg B

біти 15-3: не використовуються ( =0)

біти 2-0: destReg



Рис.3.3. Інструкція R-типу

I-тип інструкцій (lw, sw, beq):

біти 24-22: код операції

біти 21-19: reg A

біти 18-16: reg B

біти 15-0: зміщення (16 біт, значення від -32768 до 32767)



Рис.3.4. Інструкція I-типу

J-тип інструкцій:

біти 24-22: код операції

біти 21-19: reg A

біти 18-16: reg B

біти 15-0: не використовуються ( =0)

O-тип інструкцій (halt, noop):

біти 24-22: код операції

біти 21-0: не використовуються ( =0)

* 1. Система команд

Таблиця 8. Система команд

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Мнемонічний код**  **інструкції** | **КОП** | **Опис операції** |
| 1 | ADD regA regB destReg | 00000 | Додає вміст регістру regA до вмісту regB, та зберігає в destReg |
| 2 | NAND regA regB destReg | 00001 | Виконує логічне побітове І-НЕ вмісту regA з вмістом regB, та зберігає в destReg |
| 3 | LW regA regB offset | 00010 | Завантажує regB з пам’яті. Адреса пам’яті формується додаванням зміщення до вмісту regA. |
| 4 | SW regA regB offset | 00011 | Зберігає вміст регістру regB в пам’ять. Адреса пам’яті формується додаванням зміщення до вмісту regA. |
| 5 | BEQ regA regB offset | 00100 | Якщо вміст регістрів regA та regB однаковий, виконується перехід на адресу програмний лічильник(ПЛ) + 1+зміщення, в ПЛ зберігається адреса поточної тобто beq інструкції. |
| 6 | JALR regA regB | 00101 | Спочатку зберігає ПЛ+1 в regB, в ПЛ адреса поточної (jalr) інструкції. Виконує перехід на адресу, яка зберігається в regA. Якщо в якості regA regB задано один і той самий регістр, то спочатку в цей регістр запишеться ПЛ+1, а потім виконається перехід до ПЛ+1. |
| 7 | HALT | 00110 | Збільшує значення ПЛ на 1, потім припиняє виконання, стимулятор має повідомляти, що виконано зупинку. |
| 8 | NOOP | 00111 | Віднімання і обмін операндів місцями: destReg=regAregB |
| 9 | DIV regA regB destReg | 01000 | Ділить вміст регістру regA на вміст регістру regB і зберігає додатній результат в регістр destReg |
| 10 | SUB regA regB destReg | 01001 | Віднімає значення регістру regA від значення регістру regB і зберігає результат в регістр destReg |
| 11 | XSUB regA regB destReg | 01010 | Віднімає значення регістру regA від значення регістру regB, зберігає результат в регістр destReg, міняє значення регістрів regA і regB місцями. |
| 12 | SHL regA regB destReg | 01011 | Виконує логічний зсув ліворуч на значеня регістру regA на regB позицій та зберігає результат в регістрі destReg |
| 13 | ROL regA regB destReg | 01100 | Виконує цикліний зсув праворуч на значеня регістру regA на regB позицій та зберігає результат в регістрі destReg |
| 14 | NOT regA destReg | 01101 | Виконує операцію НЕ над значеням регістру regA та зберігає результат в регістрі destReg |
| 15 | JMA regA regB offset | 01110 | Якщо вміст регістру regA більший за вміст регістру regB, то виконується перехід на адресу програмний лічильник(ПЛ) + 1+зміщення, в ПЛ зберігається адреса поточної тобто jma інструкції. |
| 16 | JMNBE regA regB offset | 01111 | Якщо вміст регістру regA не менший/рівний за вміст регістру regB, то виконується перехід на адресу програмний лічильник(ПЛ) + 1+зміщення, в ПЛ зберігається адреса поточної тобто jmnbe інструкції. |
| 17 | CMP regA regB | 10000 | Порівнює вміст регістрів regA з regB і встановлює відповідні прапорці:  Якщо regA< regB, то ZF=0  Якщо regA= regB, то ZF=1  Якщо regA> regB, то ZF=0 |
| 18 | BSR regA destReg | 10001 | Побітово сканує в зворотньому напрямку(від старших до молодших) регістр regA в пошуках біта з 1, повертаєномер позиції в destReg. Якщо 1 знайдено ZF=1, інакше ZF=0. |
| 19 | BSF regA destReg | 10010 | Побітово сканує в прямому напрямку(від старших до молодших) регістр regA в пошуках біта з 1, повертаєномер позиції в destReg. Якщо 1 знайдено ZF=1, інакше ZF=0. |
| 20 | SETBR offset | 10011 | Встановлює регістр BR, та змінює регістр IR для переходу на вказану адресу. |
| 21 | LOAD regA regB | 10100 | Завантажує regB з пам’яті. Адреса пам’яті формується додаванням зміщення до вмісту regA. |
| 22 | SAVE regA regB | 10101 | Зберігає вміст регістру regB в пам’ять. Адреса пам’яті формується додаванням зміщення до вмісту regA. |
| 23 | NBEQ regA regB | 10110 | Якщо вміст регістрів regA та regB однаковий, виконується перехід на адресу програмний лічильник(ПЛ) +IR+BR, в ПЛ зберігається адреса поточної тобто beq інструкції. |
| 24 | NJMA regA regB | 10111 | Якщо вміст регістру regA більший за вміст регістру regB, то виконується перехід на адресу програмний лічильник(ПЛ) + IR+BR, в ПЛ зберігається адреса поточної тобто jma інструкції. |
| 25 | NJMNBE regA regB | 11000 | Якщо вміст регістру regA не менший/рівний за вміст регістру regB, то виконується перехід на адресу програмний лічильник(ПЛ) + IR+BR, в ПЛ зберігається адреса поточної тобто jmnbe інструкції. |

* 1. Структура СК після модифікації



Рис. 3.5. Функціональна схема СК після модифікації.

Після модифікації відбулись зміни :

1. Зменшилась шина данних (з 32бітів до 16 бітів).
2. Додано 8 додаткових операцій:

* 3 арифметичні операції (DIV, SUB, XSUB);
* 3 логічні операції (SHL, ROL, NOT);
* 2 операції керування (JMA, JMNBE).

1. Було додано прапорець ZF, з яким працюють команди CMP, BSR, BSF.
2. Зменшилась пам’ять (з 65535 слів до 256).
3. Число регістрів не змінилось (8).
4. Додано 2 команди (LOAD/SAVE) для завантаження/збереження даних при базово-індексній адресації. Додано команду для роботи з базово-індексною адресацією (SETBR).

Отже , СК є 16 розрядним комп’ютером. У СК є 8 регістрів по 32 розрядів. Пам’ять складається з 256 слів по 32 розрядів. Він підтримує 25 інструкцій, кожна з яких буде розписана нижче.

За прийнятою домовленістю 0вий регістр завжди містить 0 (це не обмовлено апаратними вимогами проте асемблерна програма ніколи не має змінювати значення 0-ого регістра, який ініціалізуються 0 ).

СК підтримує 6 форматів інструкцій.

Інструкції R-типу (add, nand, div, sub, xsub, shl, rol):

біти 11-15: код операції;

біти 8-10: reg A;

біти 5-7: reg B;

біти 3-4: не використовуються (=0)

біти 0-2: destReg.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11-15 | 8-10 | 5-7 | 3-4 | 0-2 |
| opcode | reg A | reg B | unused | destReg |

Рис.3.6. Інструкція R-типу після модифікації.

I-тип інструкцій (lw, sw, beq, jma, jmnbe):

біти 11-15: код операції;

біти 8-10: reg A;

біти 5-7: reg B;

біти 0-4: зміщення (5 біт, значення від -32 до 31).

|  |  |  |  |
| --- | --- | --- | --- |
| 11-15 | 8-10 | 5-7 | 0-4 |
| Opcode | reg A | reg B | offset |

Рис.3.7. Інструкція I-типу після модифікації.

J-тип інструкцій (cmp, nbeq, njma, njmnbe, load, save, jalr):

біти 11-15: код операції;

біти 8-10: reg A;

біти 5-7: reg B/addr;

біти 0-4: не використовуються (=0)

|  |  |  |  |
| --- | --- | --- | --- |
| 11-15 | 8-10 | 5-7 | 0-4 |
| opcode | reg A | reg B | unused |

Рис.3.8. Інструкція J-типу після модифікації.

O-тип (halt):

біти 11-15: код операції;

біти 0-10: не використовуються.

|  |  |
| --- | --- |
| 11 15 | 0 10 |
| opcode | unused |

Рис.3.9. Інструкція O-типу після модифікації.

M-тип (setbr):

біти 11-15: код операції;

біти 8-10: не використовуються;

біти 0-7: зміщення (8 біт, значення від -256 до 255).

|  |  |  |
| --- | --- | --- |
| 11-15 | 8-10 | 0-7 |
| opcode | unused | offset |

Рис.3.10. Інструкція O-типу після модифікації.

Інструкції R2-типу (not, bsr, bsf):

біти 11-15: код операції;

біти 8-10: reg A;

біти 3-7: не використовуються (=0)

біти 0-2: destReg.

|  |  |  |  |
| --- | --- | --- | --- |
| 11-15 | 8-10 | 3-7 | 0-2 |
| opcode | reg A | unused | destReg |

Рис.3.10. Інструкція R2-типу після модифікації.

Прапорець 0 ([англ.](http://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0" \o "Англійська мова) Zero flag) – використовується в командах CMP, BSR, BSF.

## Потактове виконання команд

1. Інструкції add, nand, div, sub, xsub, shl, rol:
2. memAddr <= PC;
3. PC++;
4. instrReg <= mem[memAddr];
5. aluOP <= Reg[instrReg[8-10]]
6. aluRes <=aluOP operation Reg[IR[5-7]];
7. Reg[instrReg[0-2]] <= aluRes;
8. Інструкціїlw, sw, beq, jma, jmnbe:

* sw/lw

1. memAddr <= PC;
2. PC++;
3. instrReg <= mem[memAddr];
4. memAddr <= Reg[instrReg[10-8]] + instrReg [0-4];
5. memData <= mem[memAddr]; or mem[memAddr] <= memData then Reg[instrReg[5-7]] <= mem[memAddr];

* beq, jma, jmnbe

1. memAddr <= PC;
2. PC++;
3. IR <= mem[memAddr];
4. aluRes <= Reg[instrReg [8-10]] logic\_operation Reg[instrReg [5-7]];

Якщо умова виконується, то*:*

1. PC <= PC + instrReg[0-4] + 1;
2. Інструкції cmp, nbeq, njma, njmnbe, load, save, jalr:

* Jalr:

1. memAddr <= PC;
2. PC++;
3. IR <= mem[memAddr];
4. Reg[instrReg [5-7]] <= PC+1;
5. PC <= Reg[instrReg[8-10]];
6. If Reg[instrReg[8-10]] = Reg[instrReg [5-7]] then:
7. Reg[instrReg[8-10]] <= PC+1;
8. PC <= Reg[instrReg[8-10]];

* Load/Save:

1. memAddr <= PC;
2. PC++;
3. instrReg <= mem[memAddr];
4. memAddr <= Reg[instrReg[10-8]] + Reg[BR]|Reg[BR];
5. memData <= mem[memAddr]; or mem[memAddr] <= memData then Reg[instrReg[5-7]] <= mem[memAddr];

* CMP

1. memAddr <= PC;
2. PC++;
3. IR <= mem[memAddr];
4. aluRes <= Reg[instrReg [8-10]] = Reg[instrReg [5-7]];
5. Якщо умова виконується, то*:*

Reg[ZF] <= 1;

1. Інакше:

Reg[ZF]<=0;

* nbeq, njma, njmnbe
  1. memAddr <= PC;
  2. PC++;
  3. IR <= mem[memAddr];
  4. aluRes <= Reg[instrReg [8-10]] logic\_operation Reg[instrReg [5-7]];
  5. Якщо умова виконується, то*:*
  6. PC <= PC + Reg[BR]|Reg[BR];

1. Інструкціїhalt**:**
2. memAddr <= PC;
3. PC++; mem (встановлення адреси);
4. IR <= mem[memAddr];
5. Stop.
6. Інструкція setbr:
7. memAddr <= PC;
8. PC++;
9. instrReg <= mem[memAddr];
10. Reg[BR] | Reg [IR] <= memAddr;
11. Інструкції not, bsr, bsf:
12. memAddr <= PC;
13. PC++;
14. IR <= mem[memAddr];
15. aluRes <= Reg[instrReg [8-10]] logic\_operation;
16. Reg[instrReg[0-2]]<=aluRes;
17. **Результат роботи**

Для перевірки коректної роботи процесора написані спеціальні програми. Ці програми використовують усі наявні команди, що були розроблені в даному процесорі.

1. Програма для перевірки операторів CMP, BSR, BSF

Код програми:

lw 0 1 x1

lw 0 2 x2

cmp 0 0 0

bsf 1 0 3

bsf 0 0 4

bsr 2 0 5

bsr 0 0 6

cmp 0 1 o

halt

x1 .fill 5

x2 .fill 6

Результат:

state:

pc 11

memory:

mem[ 0 ] 4139

mem[ 1 ] 4172

mem[ 2 ] 32768

mem[ 3 ] 37123

mem[ 4 ] 36868

mem[ 5 ] 35333

mem[ 6 ] 34822

mem[ 7 ] 32800

mem[ 8 ] 6253

mem[ 9 ] 6318

mem[ 10 ] 12288

mem[ 11 ] 5

mem[ 12 ] 6

mem[ 13 ] 0

mem[ 14 ] 2

registers:

reg[ 0 ] 0

reg[ 1 ] 5

reg[ 2 ] 6

reg[ 3 ] 0

reg[ 4 ] 0

reg[ 5 ] 2

reg[ 6 ] 0

reg[ 7 ] 0

reg[ BR ] 0

reg[ IR ] 0

reg[ ZF ] 0

end state

1. Перевірка виконання базово-індексної адресації (команди LOAD, SAVE).

Код програми:

setbr 0 0 x1

load 0 3

setbr 0 0 x2

load 0 5

setbr 0 0 res1

save 0 3

setbr 0 0 res2

save 0 5

halt

x1 .fill 5

x2 .fill 6

res1 .fill 0

res2 .fill 0

Результат:

pc 9

memory:

mem[ 0 ] 38921

mem[ 1 ] 41056

mem[ 2 ] 38922

mem[ 3 ] 41120

mem[ 4 ] 38923

mem[ 5 ] 43104

mem[ 6 ] 38924

mem[ 7 ] 43168

mem[ 8 ] 12288

mem[ 9 ] 5

mem[ 10 ] 6

mem[ 11 ] 5

mem[ 12 ] 6

registers:

reg[ 0 ] 0

reg[ 1 ] 0

reg[ 2 ] 0

reg[ 3 ] 5

reg[ 4 ] 0

reg[ 5 ] 6

reg[ 6 ] 0

reg[ 7 ] 0

reg[ BR ] 0

reg[ IR ] 12

reg[ ZF ] 0

end state

1. Пошук максимального числа в масиві (команди JMA, JMNBE).

Код програми:

lw 0 5 max

lw 0 1 one

loop add 2 1 2

lw 6 7 x1

add 6 1 6

jma 7 3 save

jma 5 2 loop

sw 0 3 res

end halt

save sub 3 3 3

add 3 7 3

jmnbe 0 0 loop

max .fill 5

one .fill 1

x1 .fill 3

x2 .fill 2

x3 .fill 6

x4 .fill -2

x5 .fill 4

res .fill 0

Результат:

state:

pc 9

memory:

mem[ 0 ] 4268

mem[ 1 ] 4141

mem[ 2 ] 546

mem[ 3 ] 5870

mem[ 4 ] 1574

mem[ 5 ] 30563

mem[ 6 ] 30043

mem[ 7 ] 6259

mem[ 8 ] 12288

mem[ 9 ] 19299

mem[ 10 ] 995

mem[ 11 ] 30742

mem[ 12 ] 5

mem[ 13 ] 1

mem[ 14 ] 3

mem[ 15 ] 2

mem[ 16 ] 6

mem[ 17 ] -2

mem[ 18 ] 4

mem[ 19 ] 6

registers:

reg[ 0 ] 0

reg[ 1 ] 1

reg[ 2 ] 5

reg[ 3 ] 6

reg[ 4 ] 0

reg[ 5 ] 5

reg[ 6 ] 5

reg[ 7 ] 4

reg[ BR ] 0

reg[ IR ] 0

reg[ ZF ] 0

end state

1. Перевірка виконання логічних команд (SHL, ROL, NOT).

Код програми:

lw 0 5 x1

lw 0 6 op

shl 5 6 1

rol 5 6 2

not 5 6 3

sw 0 1 res1

sw 0 2 res2

sw 0 3 res3

halt

x1 .fill 5

op .fill 3

res1 .fill 0

res2 .fill 0

res3 .fill 0

Результат:

pc 9

memory:

mem[ 0 ] 4265

mem[ 1 ] 4298

mem[ 2 ] 24001

mem[ 3 ] 26050

mem[ 4 ] 27907

mem[ 5 ] 6187

mem[ 6 ] 6220

mem[ 7 ] 6253

mem[ 8 ] 12288

mem[ 9 ] 5

mem[ 10 ] 3

mem[ 11 ] 40

mem[ 12 ] -1610612736

mem[ 13 ] -6

registers:

reg[ 0 ] 0

reg[ 1 ] 40

reg[ 2 ] -1610612736

reg[ 3 ] -6

reg[ 4 ] 0

reg[ 5 ] 5

reg[ 6 ] 3

reg[ 7 ] 0

reg[ BR ] 0

reg[ IR ] 0

reg[ ZF ] 0

end state

1. Перевірка виконання команд SUB, XSUB, DIV.

Код програми :

lw 0 1 x1

lw 0 2 x2

sub 1 2 3

div 1 2 5

xsub 1 2 4

sw 0 3 res1

sw 0 4 res2

sw 0 5 res3

halt

x1 .fill 6

x2 .fill 2

res1 .fill 0

res2 .fill 0

res3 .fill 0

Результат:

state:

pc 9

memory:

mem[ 0 ] 4137

mem[ 1 ] 4170

mem[ 2 ] 18755

mem[ 3 ] 16709

mem[ 4 ] 20804

mem[ 5 ] 6251

mem[ 6 ] 6284

mem[ 7 ] 6317

mem[ 8 ] 12288

mem[ 9 ] 6

mem[ 10 ] 2

mem[ 11 ] 4

mem[ 12 ] 4

mem[ 13 ] 3

registers:

reg[ 0 ] 0

reg[ 1 ] 2

reg[ 2 ] 6

reg[ 3 ] 4

reg[ 4 ] 4

reg[ 5 ] 3

reg[ 6 ] 0

reg[ 7 ] 0

reg[ BR ] 0

reg[ IR ] 0

reg[ ZF ] 0

end state

**Висновок**

В даній курсовій роботі був реалізований спрощений прототип комп’ютера з CISC архітектурою. Мною було проведено розширення його системи команд та зміна розмірності системної шини та памяті згідно з варіантом.

Було розроблено 2 програми:

1. Транслятора асемберного коду у машинні інструкції
2. Симулятор результуючого машинного коду

Всі архітектурні нововведення були відтестовані та працюють без помилок. Також мною було розроблено 5 тестових програм, які перевіряють правильність роботи програми.

Дана програма була протестована та працює справно.

Отже, я ознайомився та опанував інструкції простої асемберної мови та особливості їх транслювання у машинну мову.

**Список використаної літератури**

1. Мельник А. О. Архітектура комп’ютера. Наукове видання. – Луцьк: Волинська обласна друкарня, 2008. – 470 с.
2. Жмакин А. П. Архитектура ЭВМ. – СПб.: БХВ-Петербург, 2006. — 320 с: ил.
3. Таненбаум Э. Архитектура компьютера. 5-е изд. (+CD). — СПб.: Питер, 2007. — 844 с: ил.
4. Patterson D., and Hennessy J. Computer Architecture. A quantitative Approach. Second Edition. - Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996. - 760 p.
5. <https://wetalldid.files.wordpress.com/2011/11/eecs-370-introduction-to-computer-organization-exam-1-w-solutions-narayanasamy-bertacco-2010.pdf>
6. [Б. Керниган, Д. Ритчи - Язык программирования Си](http://cpp.com.ru/kr_cbook/index.html)
7. <http://uk.wikipedia.org/wiki/> Індексна\_адресація
8. <http://uk.wikipedia.org/wiki/CISC>
9. https://uk.wikipedia.org/wiki/Оператори\_в\_C\_та\_C++
10. <https://uk.wikipedia.org/wiki/Код_операції>

Додаток А.Вихідний код транслятора.

/\* Assembler for LC \*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#define MAXLINELENGTH 1000

#define MAXNUMLABELS 256

#define MAXLABELLENGTH 7 /\* includes the null character termination \*/

#define ADD 0

#define NAND 1

#define LW 2

#define SW 3

#define BEQ 4

#define JALR 5

#define HALT 6

#define NOOP 7

#define DIV 8

#define SUB 9

#define XSUB 10

#define SHL 11

#define ROL 12

#define NOT 13

#define JMA 14

#define JMNBE 15

#define CMP 16

#define BSR 17

#define BSF 18

#define SETBR 19

#define LOAD 20

#define SAVE 21

#define NBEQ 22

#define NJMA 23

#define NJMNBE 24

int readAndParse(FILE\*, char\*, char\*, char\*, char\*, char\*);

int translateSymbol(char labelArray[MAXNUMLABELS][MAXLABELLENGTH], int labelAddress[MAXNUMLABELS], int, char\*);

int isNumber(char\*);

void testRegArg(char\*);

void testAddrArg(char\*);

int

main(int argc, char\* argv[])

{

char\* inFileString, \* outFileString;

FILE\* inFilePtr, \* outFilePtr;

int address;

char label[MAXLINELENGTH], opcode[MAXLINELENGTH], arg0[MAXLINELENGTH],

arg1[MAXLINELENGTH], arg2[MAXLINELENGTH], argTmp[MAXLINELENGTH];

int i;

int numLabels = 0;

int num;

int addressField , newAddressField;

char labelArray[MAXNUMLABELS][MAXLABELLENGTH];

int labelAddress[MAXNUMLABELS];

if (argc != 3) {

printf("error: usage: %s <assembly-code-file> <machine-code-file>\n",

argv[0]);

exit(1);

}

inFileString = argv[1];

outFileString = argv[2];

inFilePtr = fopen(inFileString, "r");

if (inFilePtr == NULL) {

printf("error in opening %s\n", inFileString);

exit(1);

}

outFilePtr = fopen(outFileString, "w");

if (outFilePtr == NULL) {

printf("error in opening %s\n", outFileString);

exit(1);

}

/\* map symbols to addresses \*/

/\* assume address start at 0 \*/

for (address = 0; readAndParse(inFilePtr, label, opcode, arg0, arg1, arg2);

address++) {

/\*

printf("%d: label=%s, opcode=%s, arg0=%s, arg1=%s, arg2=%s\n",

address, label, opcode, arg0, arg1, arg2);

\*/

/\* check for illegal opcode \*/

if (strcmp(opcode, "add") && strcmp(opcode, "nand") &&

strcmp(opcode, "lw") && strcmp(opcode, "sw") &&

strcmp(opcode, "beq") && strcmp(opcode, "jalr") &&

strcmp(opcode, "halt") && strcmp(opcode, "noop") &&

strcmp(opcode, ".fill") && strcmp(opcode, "div") &&

strcmp(opcode, "sub") && strcmp(opcode, "xsub") &&

strcmp(opcode, "shl") && strcmp(opcode, "rol") &&

strcmp(opcode, "not") && strcmp(opcode, "jma") &&

strcmp(opcode, "jmnbe") && strcmp(opcode, "cmp") &&

strcmp(opcode, "bsr") && strcmp(opcode, "bsf") &&

strcmp(opcode, "setbr") && strcmp(opcode, "load") &&

strcmp(opcode, "save") && strcmp(opcode, "nbeq") &&

strcmp(opcode, "njma") && strcmp(opcode, "njmnbe")) {

printf("error: unrecognized opcode %s at address %d\n", opcode,

address);

exit(1);

}

/\* check register fields \*/

if (!strcmp(opcode, "add") || !strcmp(opcode, "nand") ||

!strcmp(opcode, "lw") || !strcmp(opcode, "sw") ||

!strcmp(opcode, "beq") || !strcmp(opcode, "jalr") ||

!strcmp(opcode, "div") || !strcmp(opcode, "sub") ||

!strcmp(opcode, "xsub") || !strcmp(opcode, "shl") ||

!strcmp(opcode, "rol") || !strcmp(opcode, "jma") ||

!strcmp(opcode, "jmnbe") || !strcmp(opcode, "cmp") ||

!strcmp(opcode, "load") || !strcmp(opcode, "save") ||

!strcmp(opcode, "nbeq") || !strcmp(opcode, "njma") ||

!strcmp(opcode, "njmnbe")) {

testRegArg(arg0);

testRegArg(arg1);

}

if (!strcmp(opcode, "not") || !strcmp(opcode, "bsr") ||

!strcmp(opcode, "bsf")) {

testRegArg(arg0);

}

if (!strcmp(opcode, "add") || !strcmp(opcode, "nand") ||

!strcmp(opcode, "div") || !strcmp(opcode, "sub") ||

!strcmp(opcode, "xsub") || !strcmp(opcode, "shl") ||

!strcmp(opcode, "rol") || !strcmp(opcode, "not") ||

!strcmp(opcode, "bsr") || !strcmp(opcode, "bsf")) {

testRegArg(arg2);

}

/\* check addressField \*/

if (!strcmp(opcode, "lw") || !strcmp(opcode, "sw") ||

!strcmp(opcode, "beq") || !strcmp(opcode, "jma") ||

!strcmp(opcode, "jmnbe") || !strcmp(opcode, "setbr")) {

testAddrArg(arg2);

}

if (!strcmp(opcode, ".fill")) {

testAddrArg(arg0);

}

/\* check for enough arguments \*/

if ((strcmp(opcode, "halt") && strcmp(opcode, "noop") &&

strcmp(opcode, ".fill") && strcmp(opcode, "jalr") &&

strcmp(opcode, "cmp") && strcmp(opcode, "load") &&

strcmp(opcode, "save") && strcmp(opcode, "nbeq") &&

strcmp(opcode, "njma") && strcmp(opcode, "njmnbe")

&& arg2[0] == '\0') ||

(!strcmp(opcode, "not") && !strcmp(opcode, "bsr") &&

!strcmp(opcode, "bsf") && arg1[0] == '\0') ||

(!strcmp(opcode, "jalr")

&& arg1[0] == '\0') ||

(!strcmp(opcode, ".fill") && arg0[0] == '\0')) {

printf("error at address %d: not enough arguments\n", address);

exit(2);

}

if (label[0] != '\0') {

/\* check for labels that are too long \*/

if (strlen(label) >= MAXLABELLENGTH) {

printf("label too long\n");

exit(2);

}

/\* make sure label starts with letter \*/

if (!sscanf(label, "%[a-zA-Z]", argTmp)) {

printf("label doesn't start with letter\n");

exit(2);

}

/\* make sure label consists of only letters and numbers \*/

sscanf(label, "%[a-zA-Z0-9]", argTmp);

if (strcmp(argTmp, label)) {

printf("label has character other than letters and numbers\n");

exit(2);

}

/\* look for duplicate label \*/

for (i = 0; i < numLabels; i++) {

if (!strcmp(label, labelArray[i])) {

printf("error: duplicate label %s at address %d\n",

label, address);

exit(1);

}

}

/\* see if there are too many labels \*/

if (numLabels >= MAXNUMLABELS) {

printf("error: too many labels (label=%s)\n", label);

exit(2);

}

strcpy(labelArray[numLabels], label);

labelAddress[numLabels++] = address;

}

}

for (i = 0; i < numLabels; i++) {

/\* printf("%s = %d\n", labelArray[i], labelAddress[i]); \*/

}

/\* now do second pass (print machine code, with symbols filled in as

addresses) \*/

rewind(inFilePtr);

for (address = 0; readAndParse(inFilePtr, label, opcode, arg0, arg1, arg2);

address++) {

if (!strcmp(opcode, "add")) {

num = (ADD << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "nand")) {

num = (NAND << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "jalr")) {

num = (JALR << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5);

}

else if (!strcmp(opcode, "halt")) {

num = (HALT << 11);

}

else if (!strcmp(opcode, "noop")) {

num = (NOOP << 11);

}

else if (!strcmp(opcode, "div")) {

num = (DIV << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "xsub")) {

num = (XSUB << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "sub")) {

num = (SUB << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "shl")) {

num = (SHL << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "rol")) {

num = (ROL << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| atoi(arg2);

}

else if (!strcmp(opcode, "not")) {

num = (NOT << 11) | (atoi(arg0) << 8) | atoi(arg2);

}

else if (!strcmp(opcode, "cmp")) {

num = (CMP << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5);

}

else if (!strcmp(opcode, "bsr")) {

num = (BSR << 11) | (atoi(arg0) << 8) | atoi(arg2);

}

else if (!strcmp(opcode, "bsf")) {

num = (BSF << 11) | (atoi(arg0) << 8) | atoi(arg2);

}

else if (!strcmp(opcode, "not")) {

num = (NOT << 11) | (atoi(arg0) << 8) | atoi(arg2);

}

else if (!strcmp(opcode, "load")) {

num = (LOAD << 11) | (atoi(arg0) << 8) | atoi(arg1) << 5;

}

else if (!strcmp(opcode, "save")) {

num = (SAVE << 11) | (atoi(arg0) << 8) | atoi(arg1) << 5;

}

else if (!strcmp(opcode, "nbeq")) {

num = (NBEQ << 11) | (atoi(arg0) << 8) | atoi(arg1) << 5;

}

else if (!strcmp(opcode, "njma")) {

num = (NJMA << 11) | (atoi(arg0) << 8) | atoi(arg1) << 5;

}

else if (!strcmp(opcode, "njmnbe")) {

num = (NJMNBE << 11) | (atoi(arg0) << 8) | atoi(arg1) << 5;

}

else if (!strcmp(opcode, "lw") || !strcmp(opcode, "sw") ||

!strcmp(opcode, "beq") || !strcmp(opcode, "jma") ||

!strcmp(opcode, "jmnbe") || !strcmp(opcode, "setbr")) {

/\* if arg2 is symbolic, then translate into an address \*/

if (!isNumber(arg2)) {

addressField = translateSymbol(labelArray, labelAddress,

numLabels, arg2);

/\*

printf("%s being translated into %d\n", arg2, addressField);

\*/

if (!strcmp(opcode, "beq") || !strcmp(opcode, "jma") ||

!strcmp(opcode, "jmnbe")) {

addressField = addressField - address - 1;

}

}

else {

addressField = atoi(arg2);

}

if (addressField < -256 || addressField > 255) {

printf("error: offset %d out of range\n", addressField);

exit(1);

}

/\* truncate the offset field, in case it's negative \*/

newAddressField = addressField & 0xFF;

/\* truncate the offset field \*/

addressField = addressField & 0x1F;

if (!strcmp(opcode, "beq")) {

num = (BEQ << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| addressField;

}

else if (!strcmp(opcode, "jma")) {

num = (JMA << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| addressField;

}

else if (!strcmp(opcode, "jmnbe")) {

num = (JMNBE << 11) | (atoi(arg0) << 8) | (atoi(arg1) << 5)

| addressField;

}

else if (!strcmp(opcode, "setbr")) {

num = (SETBR << 11) | newAddressField;

}

else {

/\* lw or sw \*/

if (!strcmp(opcode, "lw")) {

num = (LW << 11) | (atoi(arg0) << 8) |

(atoi(arg1) << 5) | addressField;

}

else {

num = (SW << 11) | (atoi(arg0) << 8) |

(atoi(arg1) << 5) | addressField;

}

}

}

else if (!strcmp(opcode, ".fill")) {

if (!isNumber(arg0)) {

num = translateSymbol(labelArray, labelAddress, numLabels,

arg0);

}

else {

num = atoi(arg0);

}

}

/\* printf("(address %d): %d (hex 0x%x)\n", address, num, num); \*/

fprintf(outFilePtr, "%d\n", num);

}

exit(0);

}

/\*

\* Read and parse a line of the assembly-language file. Fields are returned

\* in label, opcode, arg0, arg1, arg2 (these strings must have memory already

\* allocated to them).

\*

\* Return values:

\* 0 if reached end of file

\* 1 if all went well

\*

\* exit(1) if line is too long.

\*/

int

readAndParse(FILE\* inFilePtr, char\* label, char\* opcode, char\* arg0,

char\* arg1, char\* arg2)

{

char line[MAXLINELENGTH];

char\* ptr = line;

/\* delete prior values \*/

label[0] = opcode[0] = arg0[0] = arg1[0] = arg2[0] = '\0';

/\* read the line from the assembly-language file \*/

if (fgets(line, MAXLINELENGTH, inFilePtr) == NULL) {

/\* reached end of file \*/

return(0);

}

/\* check for line too long \*/

if (strlen(line) == MAXLINELENGTH - 1) {

printf("error: line too long\n");

exit(1);

}

/\* is there a label? \*/

ptr = line;

if (sscanf(ptr, "%[^\t\n ]", label)) {

/\* successfully read label; advance pointer over the label \*/

ptr += strlen(label);

}

/\*

\* Parse the rest of the line. Would be nice to have real regular

\* expressions, but scanf will suffice.

\*/

sscanf(ptr, "%\*[\t\n\r ]%[^\t\n\r ]%\*[\t\n\r ]%[^\t\n\r ]%\*[\t\n\r ]%[^\t\n\r ]%\*[\t\n\r ]%[^\t\n\r ]",

opcode, arg0, arg1, arg2);

return(1);

}

int

translateSymbol(char labelArray[MAXNUMLABELS][MAXLABELLENGTH],

int labelAddress[MAXNUMLABELS], int numLabels, char\* symbol)

{

int i;

/\* search through address label table \*/

for (i = 0; i < numLabels && strcmp(symbol, labelArray[i]); i++) {

}

if (i >= numLabels) {

printf("error: missing label %s\n", symbol);

exit(1);

}

return(labelAddress[i]);

}

int

isNumber(char\* string)

{

/\* return 1 if string is a number \*/

int i;

return((sscanf(string, "%d", &i)) == 1);

}

/\*

\* Test register argument; make sure it's in range and has no bad characters.

\*/

void

testRegArg(char\* arg)

{

int num;

char c;

if (atoi(arg) < 0 || atoi(arg) >10) {

printf("error: register out of range\n");

exit(2);

}

if (sscanf(arg, "%d%c", &num, &c) != 1) {

printf("bad character in register argument\n");

exit(2);

}

}

/\*

\* Test addressField argument.

\*/

void

testAddrArg(char\* arg)

{

int num;

char c;

/\* test numeric addressField \*/

if (isNumber(arg)) {

if (sscanf(arg, "%d%c", &num, &c) != 1) {

printf("bad character in addressField\n");

exit(2);

}

}

}

Додаток Б.Вихідний код симулятора.

/\*

\* Instruction-level simulator for the LC

\*/

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define NUMMEMORY 256 /\* maximum number of words in memory \*/

#define NUMREGS 8 /\* number of machine registers \*/

#define MAXLINELENGTH 1000

#define ADD 0

#define NAND 1

#define LW 2

#define SW 3

#define BEQ 4

#define JALR 5

#define HALT 6

#define NOOP 7

#define DIV 8

#define SUB 9

#define XSUB 10

#define SHL 11

#define ROL 12

#define NOT 13

#define JMA 14

#define JMNBE 15

#define CMP 16

#define BSR 17

#define BSF 18

#define SETBR 19

#define LOAD 20

#define SAVE 21

#define NBEQ 22

#define NJMA 23

#define NJMNBE 24

typedef struct stateStruct {

int pc;

int mem[NUMMEMORY];

int reg[NUMREGS+3];

int numMemory;

} stateType;

void printState(stateType\*);

void run(stateType);

int convertNum(int);

int

main(int argc, char\* argv[])

{

int i;

char line[MAXLINELENGTH];

stateType state;

FILE\* filePtr;

if (argc != 2) {

printf("error: usage: %s <machine-code file>\n", argv[0]);

exit(1);

}

/\* initialize memories and registers \*/

for (i = 0; i < NUMMEMORY; i++) {

state.mem[i] = 0;

}

for (i = 0; i < NUMREGS; i++) {

state.reg[i] = 0;

}

state.reg[8] = 0;

state.reg[9] = 0;

state.reg[10] = 0;

state.pc = 0;

/\* read machine-code file into instruction/data memory (starting at

address 0) \*/

filePtr = fopen(argv[1], "r");

if (filePtr == NULL) {

printf("error: can't open file %s\n", argv[1]);

perror("fopen");

exit(1);

}

for (state.numMemory = 0; fgets(line, MAXLINELENGTH, filePtr) != NULL;

state.numMemory++) {

if (state.numMemory >= NUMMEMORY) {

printf("exceeded memory size\n");

exit(1);

}

if (sscanf(line, "%d", state.mem + state.numMemory) != 1) {

printf("error in reading address %d\n", state.numMemory);

exit(1);

}

printf("memory[%d]=%d\n", state.numMemory, state.mem[state.numMemory]);

}

printf("\n");

/\* run never returns \*/

run(state);

return(0);

}

void

run(stateType state)

{

int arg0, arg1, arg2, addressField, newAddressField, BRaddressField;

int instructions = 0;

int opcode;

int maxMem = -1; /\* highest memory address touched during run \*/

int ZF = 10;

int IR = 9;

int BR = 8;

for (; 1; instructions++) { /\* infinite loop, exits when it executes halt \*/

printState(&state);

if (state.pc < 0 || state.pc >= NUMMEMORY) {

printf("pc went out of the memory range\n");

exit(1);

}

maxMem = (state.pc > maxMem) ? state.pc : maxMem;

/\* this is to make the following code easier to read \*/

opcode = state.mem[state.pc] >> 11;

arg0 = (state.mem[state.pc] >> 8) & 0x7;

arg1 = (state.mem[state.pc] >> 5) & 0x7;

arg2 = state.mem[state.pc] & 0x7; /\* only for add, nand \*/

addressField = convertNum(state.mem[state.pc] & 0x1F); /\* for beq, lw, sw \*/

BRaddressField = convertNum(state.mem[state.pc] & 0xFF); /\* for setbr \*/

newAddressField = state.reg[BR] | state.reg[IR];

state.pc++;

if (opcode == ADD) {

state.reg[arg2] = state.reg[arg0] + state.reg[arg1];

}

else if (opcode == NAND) {

state.reg[arg2] = ~(state.reg[arg0] & state.reg[arg1]);

}

else if (opcode == LW) {

if (state.reg[arg0] + addressField < 0 ||

state.reg[arg0] + addressField >= NUMMEMORY) {

printf("address out of bounds\n");

exit(1);

}

state.reg[arg1] = state.mem[state.reg[arg0] + addressField];

if (state.reg[arg0] + addressField > maxMem) {

maxMem = state.reg[arg0] + addressField;

}

}

else if (opcode == LOAD) {

if (state.reg[arg0] + newAddressField < 0 ||

state.reg[arg0] + newAddressField >= NUMMEMORY) {

printf("address out of bounds\n");

exit(1);

}

state.reg[arg1] = state.mem[state.reg[arg0] + newAddressField];

if (state.reg[arg0] + newAddressField > maxMem) {

maxMem = state.reg[arg0] + newAddressField;

}

}

else if (opcode == SW) {

if (state.reg[arg0] + addressField < 0 ||

state.reg[arg0] + addressField >= NUMMEMORY) {

printf("address out of bounds\n");

exit(1);

}

state.mem[state.reg[arg0] + addressField] = state.reg[arg1];

if (state.reg[arg0] + addressField > maxMem) {

maxMem = state.reg[arg0] + addressField;

}

}

else if (opcode == SAVE) {

if (state.reg[arg0] + newAddressField < 0 ||

state.reg[arg0] + newAddressField >= NUMMEMORY) {

printf("address out of bounds\n");

exit(1);

}

state.mem[state.reg[arg0] + newAddressField] = state.reg[arg1];

if (state.reg[arg0] + newAddressField > maxMem) {

maxMem = state.reg[arg0] + newAddressField;

}

}

else if (opcode == BEQ) {

if (state.reg[arg0] == state.reg[arg1]) {

state.pc += addressField;

state.pc = state.pc & 0x1F;

}

}

else if (opcode == NBEQ) {

if (state.reg[arg0] == state.reg[arg1]) {

state.pc = newAddressField;

}

}

else if (opcode == JMA) {

if (state.reg[arg0] > state.reg[arg1]) {

state.pc += addressField;

state.pc = state.pc & 0x1F;

}

}

else if (opcode == NJMA) {

if (state.reg[arg0] > state.reg[arg1]) {

state.pc = newAddressField;

}

}

else if (opcode == JMNBE) {

if (abs(state.reg[arg0]) >= abs(state.reg[arg1])) {

state.pc += addressField;

state.pc = state.pc & 0x1F;

}

}

else if (opcode == NJMNBE) {

if (abs(state.reg[arg0]) >= abs(state.reg[arg1])) {

state.pc = newAddressField;

}

}

else if (opcode == SETBR) {

state.reg[BR] = BRaddressField & 0x80;

state.reg[IR] = BRaddressField & 0x7F;

}

else if (opcode == JALR) {

state.reg[arg1] = state.pc;

if (arg0 != 0)

state.pc = state.reg[arg0];

else

state.pc = 0;

}

else if (opcode == NOOP) {

}

else if (opcode == HALT) {

printf("machine halted\n");

printf("total of %d instructions executed\n", instructions + 1);

printf("final state of machine:\n");

printState(&state);

exit(0);

}

else if (opcode == DIV) {

if (state.reg[arg1] != 0) {

state.reg[arg2] = abs(state.reg[arg0] / state.reg[arg1]);

}

else {

printf("error: illegal arg2 \n");

exit(1);

}

}

else if (opcode == SUB) {

state.reg[arg2] = state.reg[arg0] - state.reg[arg1];

}

else if (opcode == XSUB) {

state.reg[arg0] = state.reg[arg0] - state.reg[arg1];

state.reg[arg2] = state.reg[arg0];

state.reg[arg1] = state.reg[arg1] + state.reg[arg0];

state.reg[arg0] = state.reg[arg1] - state.reg[arg0];

}

else if (opcode == SHL) {

state.reg[arg2] = state.reg[arg0] << state.reg[arg1];

}

else if (opcode == ROL) {

int size = sizeof(state.reg[arg0]) \* 8;

int positions = state.reg[arg1] % size;

state.reg[arg2] = (state.reg[arg0] >> positions) | (state.reg[arg0] << (size - positions));

}

else if (opcode == NOT) {

for (int i = 0; i < 32; i++) {

state.reg[arg2] |= (state.reg[arg0] & (1 << i)) ? 0 : (1 << i);

}

}

else if (opcode == CMP) {

if (state.reg[arg0] < state.reg[arg1])

{

state.reg[ZF] = 0;

}

if (state.reg[arg0] == state.reg[arg1])

{

state.reg[ZF] = 1;

}

if (state.reg[arg0] > state.reg[arg1])

{

state.reg[ZF] = 0;

}

}

else if (opcode == BSR) {

for (int i = 31; i > -1; i--) {

int res = state.reg[arg0] & (1 << i);

if (res != 0)

{

state.reg[arg2] = i;

state.reg[ZF] = 1;

break;

}

else

{

state.reg[ZF] = 0;

}

}

}

else if (opcode == BSF) {

for (int i = 0; i < 32; i++) {

int res = state.reg[arg0] & (1 << i);

if (res != 0)

{

state.reg[arg2] = i;

state.reg[ZF] = 1;

break;

}

else

{

state.reg[ZF] = 0;

}

}

} else {

printf("error: illegal opcode 0x%x\n", opcode);

exit(1);

}

state.reg[0] = 0;

}

}

void

printState(stateType\* statePtr)

{

int i;

printf("\n@@@\nstate:\n");

printf("\tpc %d\n", statePtr->pc);

printf("\tmemory:\n");

for (i = 0; i < statePtr->numMemory; i++) {

printf("\t\tmem[ %d ] %d\n", i, statePtr->mem[i]);

}

printf("\tregisters:\n");

for (i = 0; i < NUMREGS; i++) {

printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);

}

printf("\t\treg[ BR ] %d\n", statePtr->reg[8]);

printf("\t\treg[ IR ] %d\n", statePtr->reg[9]);

printf("\t\treg[ ZF ] %d\n", statePtr->reg[10]);

printf("end state\n");

}

int

convertNum(int num)

{

/\* convert a 16-bit number into a 32-bit Sun integer \*/

if (num & (1 << 15)) {

num -= (1 << 16);

}

return(num);

}