**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 №17

Виконав:

Ст. групи КІ-33

Скрипник Д.О.

Прийняв:

Грицик І. В.

**Львів – 2019**

**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.

**Зміст**

[**Вступ 4**](#_Toc26707615)

[**1. Конкретизовані та розширені вихідні дані для проектування 6**](#_Toc26707616)

[**2. Аналіз основних принципів побудови комп’ютерів. 8**](#_Toc26707617)

[2.1. Архітектура CISC 8](#_Toc26707618)

[2.2. Система команд 12](#_Toc26707619)

[2.3. Способи адресації 13](#_Toc26707620)

[2.4. Індексна адресація 15](#_Toc26707621)

[**3. Алгоритм розробленого емулятора та асемблера 18**](#_Toc26707622)

[**4. Аналіз принципів побудови спрощеного комп’ютера 21**](#_Toc26707623)

[4.1. Структура СК до модифікації 21](#_Toc26707624)

[4.2. Система команд 23](#_Toc26707625)

[4.3. Структура СК після модифікації 25](#_Toc26707626)

[4.4. Потактове виконання команд 27](#_Toc26707627)

[**5. Результат роботи 30**](#_Toc26707628)

[1. Знайти суму чисел від 0 до 25 30](#_Toc26707629)

[**Список використаної літератури 34**](#_Toc26707630)

[**Додаток А. Вихідний код транслятора. 35**](#_Toc26707631)

[**Додаток Б. Вихідний код симулятора. 41**](#_Toc26707632)

**Вступ**

В ході виконання даного курсового проекту передбачає ознайомлення та опанування архітектуру CISC – комп’ютера. Основні принципи даної архітектури, які запропонував Джон фон Нейман:

1. Інформація кодується в двійковому представленні.

2. Інформація в комп’ютері ділиться на команди і дані.

3. Різнотипні за змістом слова розрізняються за способом застосування, а не по способу кодування.

4. Слова інформації розміщаються в комірках пам’яті та ідентифікуються номерами комірок – адресами слів.

5. Пам’ять є лінійною.

6. Пам’ять має довільну адресацію.

7. Команди і дані зберігаються в одній пам’яті.

8. Алгоритми представляються у вигляді послідовності керуючих слів, як називаються командами. Команда визначається найменуванням операції та слів інформації, які в ній приймають участь. Алгоритм записаний у вигляді послідовності команд, називається програмою.

9. Весь набір виконуваних комп’ютером команд називається системою команд комп’ютера.

10. Виконання обчислень, які визначені алгоритмом, являють собою послідовне виконання команд в порядку визначеному програмою.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, 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. Cтруктура 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. Більшість комп'ютерів мають в складі процесора тригери для зберігання бітів стану процесора, або, як їх іще називають, прапорців. Кожен прапорець має спеціальне призначення. Частина прапорців вказує на результати арифметичних і логічних операцій: додатній результат (Р), від'ємний результат (N), нульовий результат (Z), перенос (С), арифметичне переповнення (V), і т. д. В системі команд комп'ютера є команди, які вказують процесору коли встановити чи скинути ці тригери. Інша частина прапорців вказує режими захисту пам'яті. Існують також прапорці, які вказують пріоритети виконуваних програм. В деяких процесорах додаткові тригери служать для зберігання кодів умов, формуючи регістр кодів умов. Взяті разом описані прапорці формують слово стану програми (ССП), а відповідні тригери - регістр ССП. Регістри загального призначення (РЗП) є програмно доступними. Зазвичай 'їх називають регістровим файлом. Вони можуть використовуватись програмістом в якості регістрів для зберігання вхідних та вихідних даних, а також проміжних результатів обчислень, в якості адресних та індексних регістрів при виконанні операцій модифікації адрес.

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

Кожен варіант складається з наступних завдань:

8 додаткових інструкцій без використання регістрів стану:

3 – арифметичні

3 – логічні

2 – керування

3 додаткові інструкції з використання регістрів стану.

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

Таблиця 1 – Варіант №17

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **№** | **Розряд**  **ність** | **Арифметичні** | | | **Логічні** | | | **Керування** | | **Прапорці** | | | | **Адре**  **сація** |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **1** | **2** | **3** |
| 17 | 3 | 3 | 5 | 9 | 1 | 2 | 17 | 2 | 10 | ZF | 2 | 3 | 5 | 1 |

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

Таблиця 2

|  |  |  |  |
| --- | --- | --- | --- |
| № | Розрядність шини даних | Розмір пам’яті  Байт | Розмір регістрового файлу(к-сть регістрів) |
| 3 | 32 | 65536 | 16 |

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

**Арифметичні**

Таблиця 3

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 3 | DIV regA regB destReg | Беззнакове ділення destReg=regA/regB |
| 5 | IMUL regA regB destReg | Знакове множення destReg=regA\*regB |
| 9 | XIDIV regA regB destReg | Знакове ділення і обмін операндів місцями destReg=regA/regB |

**Логічні**

Таблиця 4

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 1 | AND regA regB destReg | Побітове логічне І: destReg=regA & regB |
| 2 | XOR regA regB destReg | Додавання по модулю 2: destReg=regA # regB |
| 17 | CMPGE regA regB destReg | Порівняти regA regB destReg= regA >= regB |

**Керування. Умовні переходи**.

Таблиця 5

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 2 | JMAE regA regB offSet | Беззнакове більше/рівно if (regA>= regB) PC=PC+1+offset |
| 10 | JMNAE regA regB offSet | Беззнакове не більше/рівно if (regA!>= regB) PC=PC+1+offset |

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)

Таблиця 6

|  |  |
| --- | --- |
| **№** | **Адресція** |
| 1 | Безадресна – реалізація стеку. Максимальна глибина 32 слова по 32 розряди. |

Безадресні команди.

Таблиця 7

|  |  |
| --- | --- |
| **Мнемонічнй код** | **Зміст** |
| POP | Зчитати з стеку в 1 регістр |
| PUSH | Записати в стек з 1 регістру |

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

**Регістр переносу (ZF)**

Таблиця 8

|  |  |  |
| --- | --- | --- |
| **№** | **Мнемонічнй код** | **Зміст** |
| 2 | BSR regA destReg | Побітове сканування 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 молодших до старших) напрямку regA в пошуках біта з 1 , повертає номер позиції в destReg. Якщо 1 знайдено ZF=1, інакше ZF=0 |
| 5 | JNE offSet | Перейти, якщо більше чи рівно, if(ZF!=0)PC=offset |

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

**2.1. Архітектура CISC**

**CISC** ([англ.](http://uk.wikipedia.org/wiki/Англійська_мова) Complex Instruction Set Computer — комп'ютер зі складним набором команд) — це [архітектура системи команд](http://uk.wikipedia.org/wiki/Архітектура_системи_команд), в якій більшість команд є комплексними, тобто реалізують певний набір простіших інструкцій [процесора](http://uk.wikipedia.org/wiki/Центральний_процесор) або шляхом зіставлення з кожною CISC-командою певної [мікропрограми](http://uk.wikipedia.org/wiki/Мікропрограма), 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/Методи_адресації_пам'яті) 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/Архітектура_системи_команд).

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/Структура_даних). В ті часи, за відсутності повноцінних [мов програмування](http://uk.wikipedia.org/wiki/Мова_програмування)високого рівня та відповідних [компіляторів](http://uk.wikipedia.org/wiki/Компілятор), така апаратна підтримка високорівневого інструментарію програмування могла підвищити [продуктивність праці](http://uk.wikipedia.org/wiki/Продуктивність_праці) 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/Мікропроцесор) [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/Англійська_мова) 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.

**2.2. 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.

**2.3. 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.

**2.4. Індексна адресація**

Індексна адресація використовується при виконанні циклів, коли потрібно збільшення або зменшення адреси на деяку величину. Цей спосіб адресації подібний до відносної адресації, при якій адреса може автоматично змінюватися в процесі виконання програми. Індексація є засобом для багатократного виконання одних і тих же відрізків програми над різними наборами (масивами) вхідних даних. Тим самим забезпечується мінімальна залежність довжини програми від кількості повторюваних відрізків програми. При цьому коди команд програми залишаються без змін. Для отримання виконавчої адреси адресна частина команди додається до вмісту спеціального регістра, в якому зберігається номер оброблюваного масиву чисел. Ці регістри називають індексними, а їх вміст - індексною величиною, або індексом.

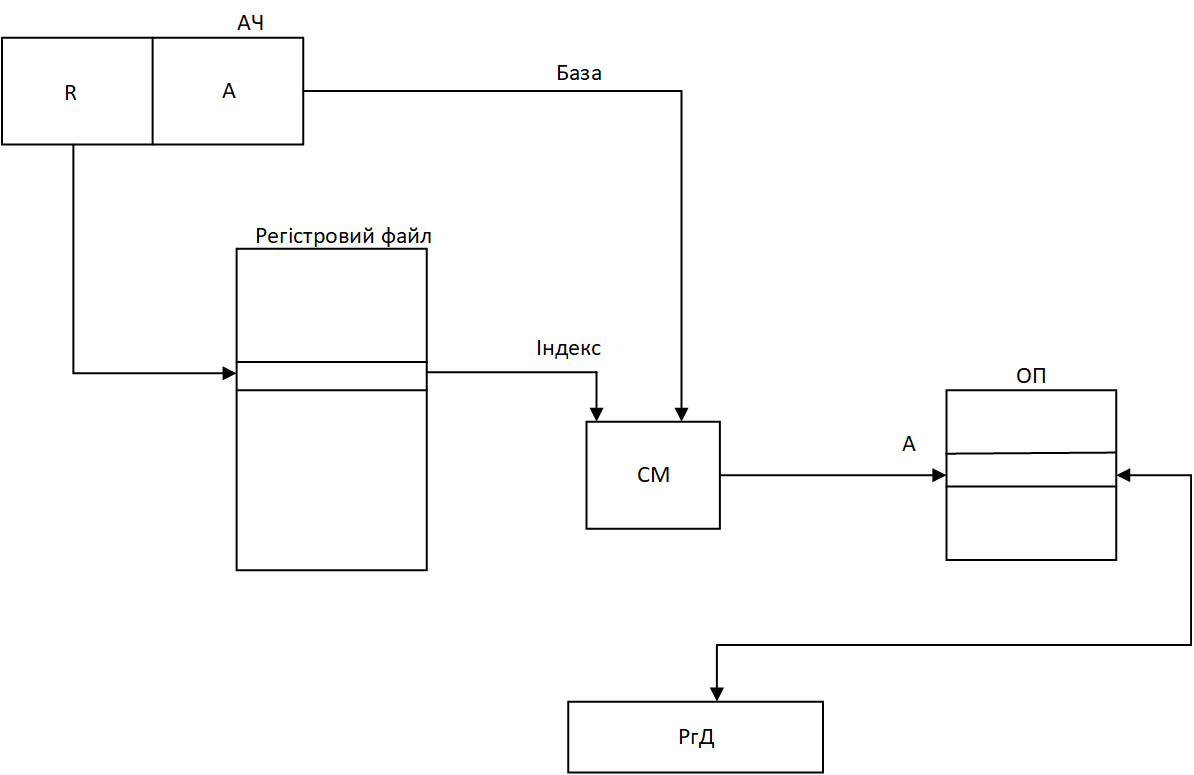


Рис.2.1. Формування адреси при індексній адресації.

Вказівка про індексну адресацію вміщується в полі типу адресації. При наявності кількох індексних регістрів в цих розрядах команди вказується номер того індексного регістра, в якому зберігається значення індексу оброблюваного в даний час масиву інформації. Індекси можуть зберігатися як в спеціальних індексних регістрах, так і в регістровій пам'яті процесора. Вміст індексних регістрів змінюється після закінчення деякого циклу обробки. При цьому до попереднього значення індексу добавляється приріст, значення якого залежить від розміщення операндів в пам'яті.

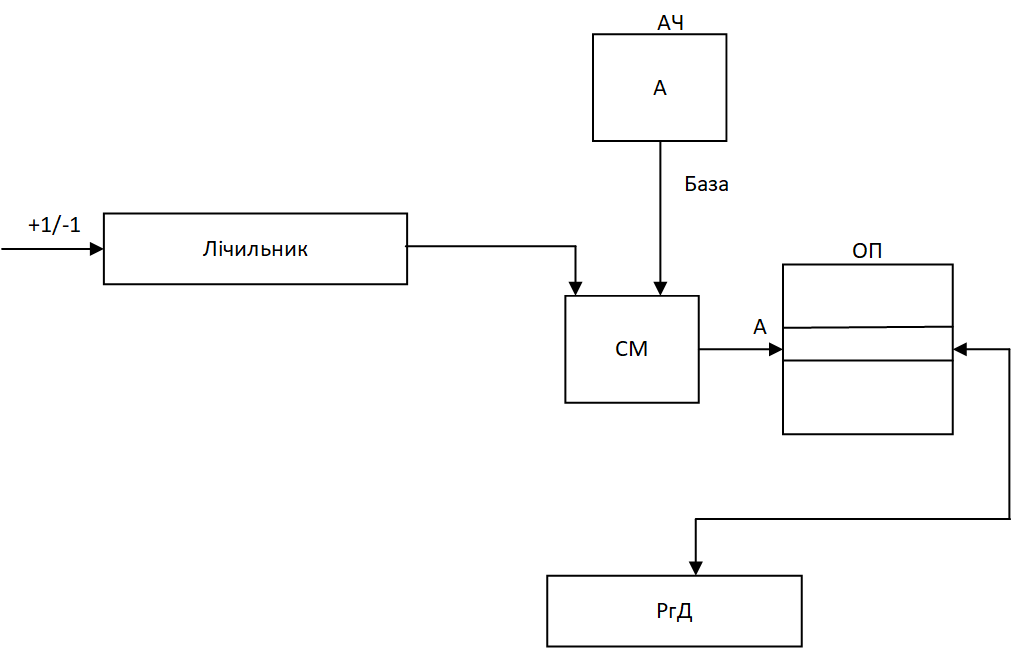
Різновидністю індексної адресації є автоіндексація, при якій значення індексу є відомим наперед. Найчастіше операнди розміщуються в пам'яті послідовно і тому це значення рівне + 1 (так звана автоінкрементна адресація) або -1 (так звана автодекрементна адресація). Порядок формування адреси при використанні автоінкрементної та автодекрементної адресації показано на рис. 2.2, де для забезпечення переміщення по комірках пам'яті використовується лічильник. 

Рис.2.2. Формування адреси при автоінкрементній та автодекрементній адресації.

Операції індексної арифметики виконуються в спеціальному індексному арифметичному пристрої, або в арифметико-логічному пристрої процесора. Команди індексної арифметики входять до складу команд керування. Вони забезпечують зміну значення індексу шляхом добавлення до нього приросту, перевірку закінчення індексного циклу та засилання початкових значень індексу. Значення індексів, приростів та інформаційні біти циклів формують керуюче слово, яке розміщується в одній або декількох командах керування. Послідовність таких управляючих слів забезпечує роботу з масивами даних.

1. **Алгоритм розробленого емулятора та асемблера**

  
Рис.3.1. Схема роботи асемблера

Загальна схема роботи асемблера (рис. 3.1) складається з 2 проходів. На першому проході асемблер перевіряє коректність синтаксису команд. На другому виконується генерування відповідних машинних команд, тобто числового представлення асемблерної команди.

Функція *readAndParse* виконує зчитування рядку асемблерної програми і декодування на відповідні поля: мітка, код операції, операнди. Отримана таким чином і декодована інструкція перевіряється на коректність: існування команди, відповідна кількість аргументів, існування міток та т. п.

Функція *testRegArg* перевіряє коректність використання назви регістра.

Функція *testAddrArg* перевіряє коректність використання адреси.

Функція *labelArray* перетворює відповідну мітку у адресу.

*Program.as* та *program.mc* – відповідно вхідний та вихідний файли.

  
Рис.3.2. Функціональна схема симулятора.

Симулятор починає свою роботу ініціалізацією пам’яті та регістрів 0 значеннями (рис. 3.2). Наступним кроком відбувається заванаження програми у машинних кодах в пам’ять. Далі відбувається покрокове виконання інструкційта вивід стану на зовнішній пристрій (чи на екран консолі чи у файл).

У stateStruct зберігається стан машини – значення регістрів, пам’яті та програмний лічільник. *stateStruct*

Функція *Run* виконує обробку інструкцій з пам’яті, функція *printState* виводить поточний стан машини, а функція *convertNum* виконує перетворення числа у доповняльний код.

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



Рис. 4.1. 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 0-вий регістр завжди містить 0 (це не обмовлено апаратними вимогами проте асемблерна програма ніколи не має змінювати значення 0ого регістра, який ініціалізуються 0 ).



Рис.4.2. 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.4.3. 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.4.4. 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 9. 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 | DIV regA regB destReg | 00111 | Беззнакове ділення destReg=regA/regB |
| 9 | IMUL regA regB destReg | 01000 | Знакове множення destReg=regA\*regB |
| 10 | XIDIV regA regB destReg | 01001 | Знакове ділення і обмін операндів місцями destReg=regA/regB |
| 11 | AND regA regB destReg | 01010 | Побітове логічне І: destReg=regA & regB |
| 12 | XOR regA regB destReg | 01011 | Додавання по модулю 2: destReg=regA # regB |
| 13 | CMPGE regA regB destReg | 01100 | Порівняти regA regB destReg= regA >= regB |
| 14 | JMAE regA regB offSet | 01101 | Беззнакове більше/рівно if (regA>= regB) PC=PC+1+offSet |
| 15 | JMNAE regA regB offSet | 01110 | Беззнакове не більше/рівно if (regA!>= regB) PC=PC+1+offSet |
| 16 | POP | 01111 | Зчитати з стеку в 1 регістр |
| 17 | PUSH | 10000 | Записати в стек з 1 регістру |
| 18 | BSR regA destReg | 10001 | Побітове сканування 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 |
| 19 | BSF regA destReg | 10010 | Побітове сканування в прямому( від молодших до старших) напрямку regA в пошуках біта з 1 , повертає номер позиції в destReg. Якщо 1 знайдено ZF=1, інакше ZF=0 |
| 20 | JNE offSet | 10011 | Перейти, якщо більше чи рівно, if(ZF!=0)PC=offset |

* 1. **Структура СК після модифікації**



Рис. 4.5. Функціональна схема СК після модифікації.

Після модифікації відбулись зміни :

1. Додано 8 додаткових операцій:

* 3 арифметичні операції (DIV, IMUL, XIDIV);
* 3 логічні операції (XOR, AND, CMPGE);
* 2 операції керування (JMAE, JMNAE).

1. Було додано прапорець ZF – (Zero flag), а також команди BSR, BSF, JNE, які працюють в парі з прапорцем.
2. Збільшилось число регістрів (з 8 до 16).
3. Додано додатковий спосіб адресації — безадресна, та модифіковані команди XADD, XIDIV, XSUB для роботи з прямою адресацією

Отже , СК є 32 розрядним комп’ютером. У СК є 16 регістрів по 32 розряди. Пам’ять складається з 65535 слів по 32 розрядів. Він підтримує 20 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 7 форматів інструкцій.

Інструкції R-типу(XIDIV, IMUL, DIV, AND, XOR, CMPGE, NAND, ADD):

біти 25-22: код операції

біти 21-19: reg A

біти 18-16: reg B

біти 15-3: не використовуються

біти 2-0: destReg

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31-26 | 25-22 | 21-19 | 18-16 | 15-3 | 2-0 |
| unused | opcode | regA | regB | unused | destReg |

Рис.4.6. Інструкція R-типу після модифікації.

I-тип інструкцій (lw, sw, beq, jmae, jmnae):

біти 25-22: код операції

біти 21-19: reg A

біти 18-16: reg B

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31-26 | 25-22 | 21-19 | 18-16 | 15-0 |
| unused | opcode | regA | regB | Offset |

Рис.4.7. Інструкція I-типу після модифікації.

J-тип інструкцій (jalr):

біти 31-26: не використовується

біти 25-22: код операції;

біти 21-19: reg A;

біти 18-16: reg B;

біти 15-0: не використовуються.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31-26 | 25-22 | 21-19 | 18-16 | 15-0 |
| unused | opcode | regA | regB | Unused |

Рис.4.8. Інструкція J-типу після модифікації.

O-тип (halt):

біти 25-22: код операції;

біти 21-0: не використовуються.

|  |  |  |
| --- | --- | --- |
| 31-26 | 25-22 | 21-0 |
| unused | opcode | Unused |

Рис.4.9. Інструкція O-типу після модифікації.

S-тип інструкцій (push, pop):

біти 25-22: код операції

біти 21-19: reg A

біти 18-0: не використовується.

|  |  |  |  |
| --- | --- | --- | --- |
| 31-26 | 25-22 | 21-19 | 18-0 |
| unused | opcode | regA | unused |

Рис.4.7. Інструкція I-типу після модифікації.

Z-тип інструкцій (JNE):

біти 25-22: код операції

біти 21-16: не використовується

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

|  |  |  |  |
| --- | --- | --- | --- |
| 31-26 | 25-22 | 21-16 | 15-0 |
| unused | opcode | unused | offset |

Рис.4.7. Інструкція I-типу після модифікації.

Інструкції Z1-типу(BSF, BSR):

біти 25-22: код операції

біти 21-19: reg A

біти 18-3: не використовуються

біти 2-0: destReg

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31-26 | 25-22 | 21-19 | 18-3 | 2-0 |
| unused | opcode | regA | unused | destReg |

Рис.4.6. Інструкція R-типу після модифікації.

## Потактове виконання команд

1. Інструкції XIDIV, IMUL, DIV, AND, XOR, CMPGE, NAND, ADD:
2. memAddr <= PC;
3. PC++; mem (встановлення адреси);
4. instrReg <= mem[memAddr];
5. aluOP <= Reg[instrReg[18-16]]
6. aluRes <=aluOP operation Reg[instrReg[21-19]];
7. Reg[instrReg[2-0]] <= aluRes;
8. Інструкціїlw, sw, beq, jmae, jmnae, jne:

* sw/lw

1. memAddr <= PC;
2. PC++; mem (встановлення адреси);
3. instrReg <= mem[memAddr];
4. memAddr <= Reg[instrReg[18-16]] + instrReg [15-0];
5. memData <= mem[mamAddr]; or mem[mamAddr] <= memData then Reg[instrReg[18-16]] <= mem[Addr];

* умовних переходів

1. memAddr <= PC;
2. PC++; mem (встановлення адреси);
3. IR <= mem[memAddr];
4. aluRes <= Reg[instrReg [21-19]] logic\_operation Reg[instrReg [18-16]];

Якщо умова виконується, то*:*

1. PC <= PC + instrReg[15-0] + 1;

* JNE

1. memAddr <= PC;
2. PC++; mem (встановлення адреси);
3. IR <= mem[memAddr];
4. aluRes <= ZF != 0

Якщо умова виконується, то*:*

1. PC <= instrReg[15-0];
2. Інструкція jalr:
3. memAddr <= PC;
4. PC++; mem (встановлення адреси);
5. IR <= mem[memAddr];
6. A) Reg[instrReg [18-16]] <= PC;

Б) Reg[instrReg [18-16]] <= - Reg[instrReg [21-19]]

1. PC <= Reg[instrReg[21-19]];
2. If Reg[instrReg[21-19]] = Reg[instrReg [18-16]] then:

A.1) Reg[instrReg[21-19]] <= PC+1;

A.2) PC <= Reg[instrReg[21-19]];

1. Інструкціїhalt**:**
2. memAddr <= PC;
3. PC++; mem (встановлення адреси);
4. IR <= mem[memAddr];
5. Stop.
6. Інструкції push/pop:
   1. push

memAddr <= PC;

PC++; ROM (встановлення адреси);

IR <= ROM[memAddr];

Stack<=RF[IR[18-16]]; or RF[IR[18-16]]<=Stack;

б)

1. memAddr <= PC;
2. PC++; ROM (встановлення адреси);
3. IR <= ROM[memAddr];
4. AX<=Stack[RF[IR[18-16]]];
5. Result<=AX operation Stack[RF[IR[18-16]]];
6. Stack<=Result;
7. Інструкції BSF/BSR:
8. memAddr <= PC;
9. PC++; mem (встановлення адреси);
10. IR <= mem[memAddr];
11. aluRes <= AluOP operation Reg[instrReg [21-19]]
12. Reg[instrReg[2-0]] <= aluRes;
13. ZF = ALU find ‘1’ in Reg ? 1 : 0
14. **Результат роботи**

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

**1. Знайти суму чисел від 0 до 25**

1. Формуємо програму.

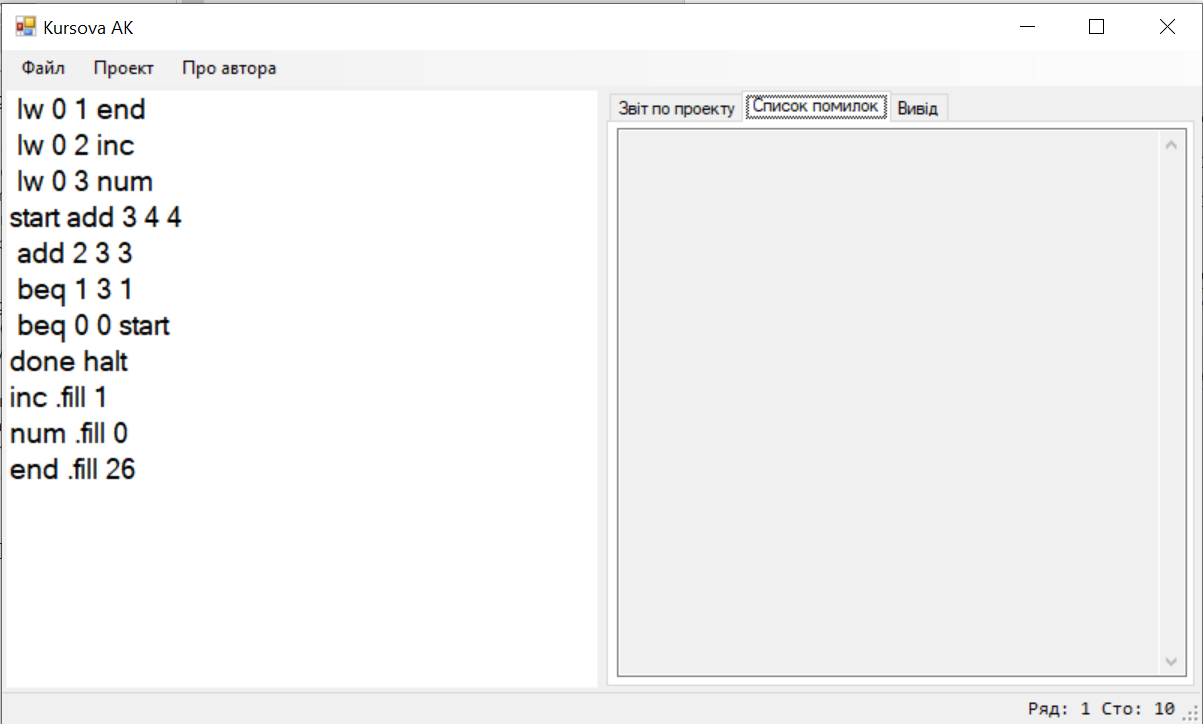
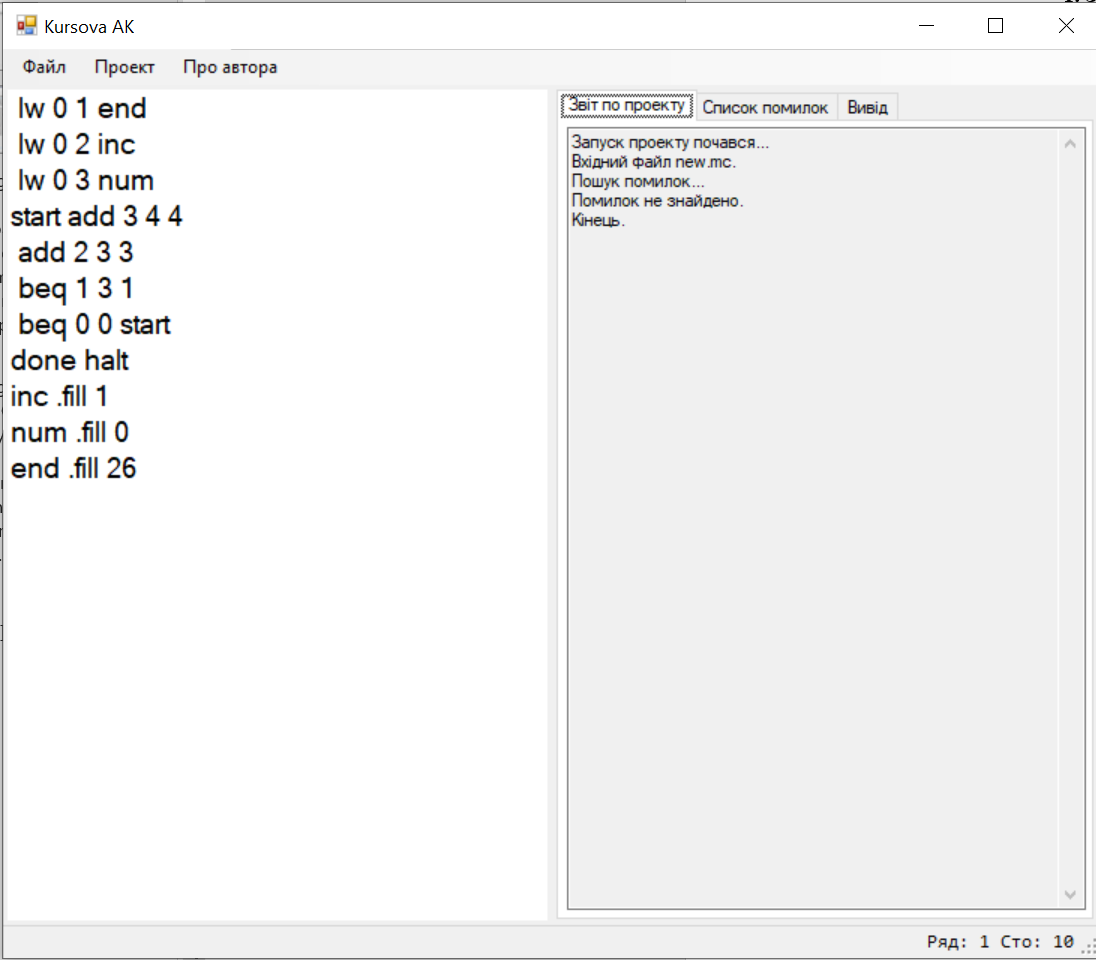


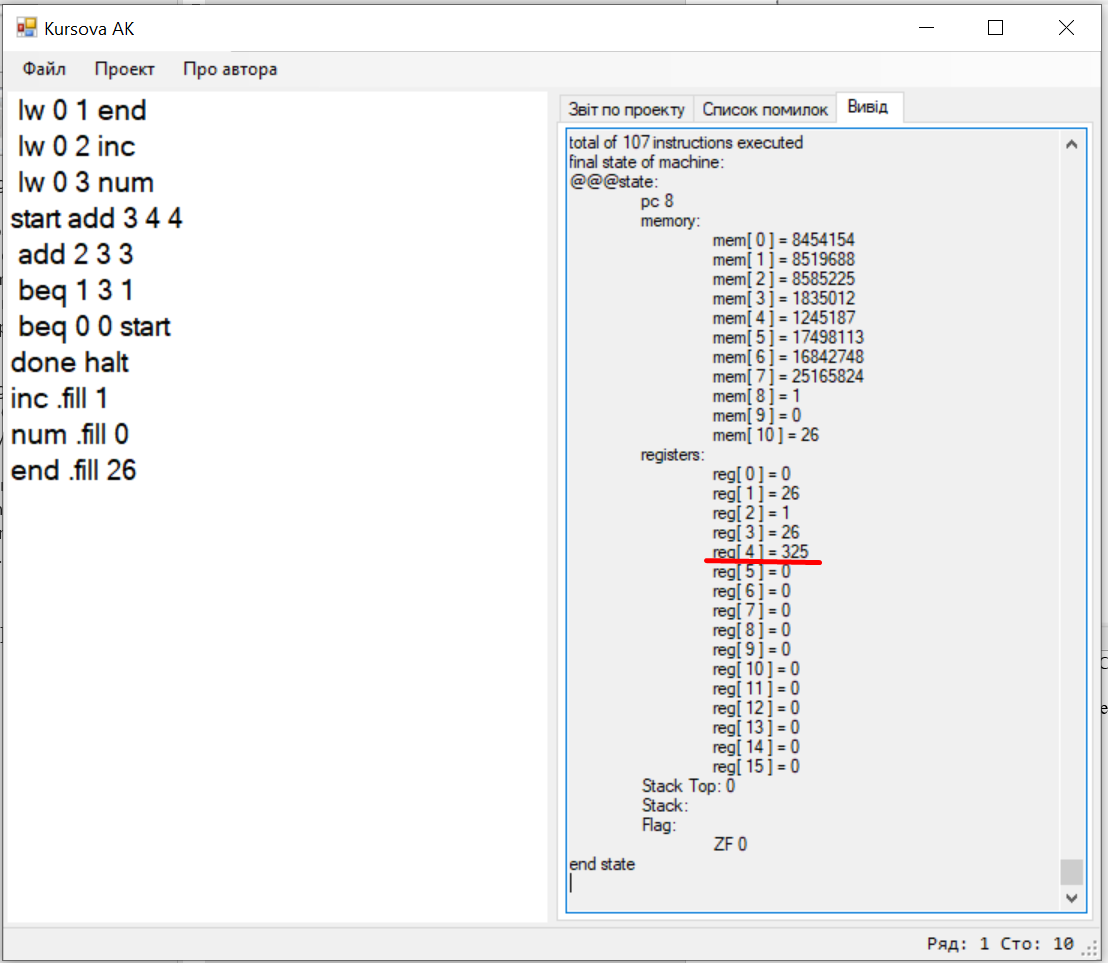
Рис. 5.1 Програма

1. Компілюємо програму.



5.2 Скомпільована програма

1. Помилок не знайдено, тому можемо запустити програму.



5.3 Результат програми

Отримали:

107 виконаних інструкцій;

Результат виконання 325;

Сума перших 25 натуральних чисел дорівнює **325,** отже ми переконались, що програма працює вірно.

**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.

Програми були протестовані та працюють 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.**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Runtime.Serialization.Formatters.Binary;

using System.Text.RegularExpressions;

using System.Text;

namespace Assembler

{

public class ASol : ISol

{

public void Run(string compiled, string to)

{

string label, opcode, arg0, arg1, arg2;

var labelArray = new List<Label>(CommonUtil.MAXNUMLABELS);

StreamReader inFile = null;

StreamWriter outFile = null;

try

{

inFile = new StreamReader(compiled);

outFile = new StreamWriter(to);

/\* map symbols to addresses \*/

/\* assume address start at 0 \*/

int address;

bool isNoReg;

for (address = 0; ReadAndParse(inFile, out label, out opcode, out arg0, out arg1, out arg2); address++)

{

/\* check for illegal opcode \*/

if ( CommonUtil.Commands.All(c => opcode != c.Value))

{

throw new MessageException($"Unrecognized opcode {opcode} at address {address + 1}");

}

isNoReg = CheckCommands(opcode, new[] { Command.XOR, Command.AND, Command.IMUL }) && string.IsNullOrEmpty(arg0);

/\* check register fields \*/

if (CheckCommands(opcode, new[] { Command.ADD, Command.NAND, Command.LW, Command.SW,

Command.BEQ, Command.JALR, Command.MUL, Command.DIV,

Command.IMUL, Command.XIDIV, Command.XOR, Command.AND,

Command.CMPGE, Command.JMAE, Command.JMNAE, Command.PUSH,

Command.POP, Command.BSF, Command.JNE, Command.BSR}))

{

if (!isNoReg)

TestRegArg(arg0);

}

if (CheckCommands(opcode, new[] { Command.ADD, Command.NAND, Command.LW, Command.SW,

Command.BEQ, Command.JALR, Command.MUL, Command.DIV,

Command.IMUL, Command.XIDIV, Command.XOR, Command.AND,

Command.CMPGE, Command.JMAE, Command.JMNAE, Command.JNE}))

{

if (!isNoReg)

TestRegArg(arg1);

}

if (CheckCommands(opcode, new[] { Command.ADD, Command.NAND, Command.MUL, Command.IMUL,

Command.AND, Command.XOR, Command.DIV, Command.XIDIV,

Command.CMPGE}))

{

if (!isNoReg)

TestRegArg(arg2);

}

/\* check addressField \*/

if (CheckCommands(opcode, new[] { Command.LW, Command.SW, Command.BEQ, Command.JMAE, Command.JMNAE}))

{

TestAddrArg(arg2);

}

/\* check addressField \*/

if (CheckCommands(opcode, new[] { Command.FILL, Command.JNE}))

{

TestAddrArg(arg0);

}

/\* check for enough arguments \*/

if (!isNoReg &&

(!CheckCommands(opcode, new[] { Command.HALT, Command.FILL, Command.JALR, Command.PUSH, Command.POP, Command.BSR, Command.BSF}) && string.IsNullOrEmpty(arg2)) ||

(CheckCommands(opcode, new[] { Command.BSF, Command.BSR }) && string.IsNullOrEmpty(arg1) ) ||

(CheckCommands(opcode, new[] { Command.FILL, Command.PUSH, Command.POP, Command.JNE }) && string.IsNullOrEmpty(arg0) ) )

{

throw new MessageException($"Error at address {address + 1}: not enough arguments");

}

if ( !string.IsNullOrEmpty(label) )

{

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

if (label.Length >= CommonUtil.MAXLABELLENGTH)

{

throw new MessageException($"Label {label} is too long!(max length: {CommonUtil.MAXLABELLENGTH})");

}

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

if (!Regex.IsMatch(label, "^[a-zA-Z]"))

{

throw new MessageException($"Label {label} doesn't start with letter!");

}

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

if (!Regex.IsMatch(label, "[a-zA-Z0-9]"))

{

throw new MessageException($"Label {label} has character other that letters and numbers!");

}

/\* look for duplicate label \*/

var duplicate = labelArray.FirstOrDefault(l => l.label == label);

if (duplicate != null)

{

throw new MessageException($"Error: duplicate label {label} at address {duplicate.Address + 1}");

}

labelArray.Add(new Label { label = label, Address = address });

}

}

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

addresses) \*/

inFile.DiscardBufferedData();

inFile.BaseStream.Seek(0, SeekOrigin.Begin);

for (address = 0; ReadAndParse(inFile, out label, out opcode, out arg0, out arg1, out arg2); address++)

{

var opCommand = (Command)Enum.Parse(typeof(Command), opcode.ToUpper().Replace(".", ""));

isNoReg = CheckCommands(opcode, new[] { Command.XOR, Command.AND, Command.IMUL }) && string.IsNullOrEmpty(arg0);

int num = 0;

if (CheckCommands(opcode, new[] { Command.ADD, Command.NAND, Command.MUL, Command.DIV,

Command.XIDIV, Command.AND, Command.CMPGE, Command.XOR, Command.IMUL}))

{

num = !isNoReg ? CreateNumber(opCommand, arg0, arg1, arg2) : CreateNumber(opCommand);

}

else if (CheckCommands(opcode, new[] { Command.JALR, Command.BSF, Command.BSR }))

{

num = CreateNumber(opCommand, arg0, arg1);

}

else if (CheckCommands(opcode, new[] { Command.PUSH, Command.POP}))

{

num = CreateNumber(opCommand, arg0);

}

else if (opcode == (CommonUtil.Commands[Command.HALT]))

{

num = CreateNumber(opCommand);

}

else if (opcode == (CommonUtil.Commands[Command.FILL]))

{

if (int.TryParse(arg0, out num))

{

num = int.Parse(arg0);

}

else

{

num = TranslateSymbol(labelArray, arg0);

}

}

else if ( CheckCommands(opcode, new[] { Command.SW, Command.BEQ, Command.LW, Command.JNE, Command.JMAE, Command.JMNAE }) )

{

int addressField;

if (opCommand == Command.JNE)

{

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

addressField = !IsNumber(arg0) ? TranslateSymbol(labelArray, arg2) : int.Parse(arg2);

}

else

{

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

if (!IsNumber(arg2))

{

addressField = TranslateSymbol(labelArray, arg2);

if ( CheckCommands(opcode, new[] { Command.BEQ, Command.JMAE, Command.JMNAE }) )

{

addressField = addressField - address - 1;

}

}

else

{

addressField = int.Parse(arg2);

}

}

if (addressField < CommonUtil.MINADDRESSFIELD || addressField > CommonUtil.MAXADDRESSFIELD)

{

throw new MessageException($"Error: offset {addressField + 1} out of range");

}

addressField &= CommonUtil.MaskForAddr;

num = opCommand == Command.JNE ? CreateNumber(opCommand, addressField) : CreateNumber(opCommand, arg0, arg1, addressField);

}

outFile.WriteLine(num);

}

}

finally

{

inFile?.Close();

outFile?.Close();

}

}

/\*

\* 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.

\*/

private static bool ReadAndParse( StreamReader inFile, out string label, out string opcode, out string arg0, out string arg1, out string arg2 )

{

label = opcode = arg0 = arg1 = arg2 = string.Empty;

/\* delete prior values \*/

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

if (inFile.EndOfStream)

{

/\* reached end of file \*/

return false;

}

var line = inFile.ReadLine();

var index = 0;

if (line == null) return false;

var split = line.Split(new[] {'\t', '\n', '\r', ' '}, StringSplitOptions.RemoveEmptyEntries);

if (!Regex.IsMatch(line, "^[\t\n ]"))

{

label = split[index++];

}

opcode = split.ElementAtOrDefault(index++) ?? string.Empty;

arg0 = split.ElementAtOrDefault(index++) ?? string.Empty;

arg1 = split.ElementAtOrDefault(index++) ?? string.Empty;

arg2 = split.ElementAtOrDefault(index) ?? string.Empty;

return true;

}

private static int TranslateSymbol( IEnumerable<Label> labelArray, string symbol )

{

/\* search through address label table \*/

var label = labelArray.FirstOrDefault(l => l.label == (symbol));

if (label == null)

{

throw new MessageException($"Error: missing label {symbol}");

}

return label.Address;

}

private bool IsNumber( string str)

{

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

return int.TryParse(str, out \_);

}

/\*

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

\*/

private void TestRegArg( string arg )

{

if (!IsNumber(arg)) return;

if (!Regex.IsMatch(arg, "[0-9]"))

{

throw new MessageException("Bad character in register argument!");

}

var num = int.Parse(arg);

if (num < 0 || num > CommonUtil.COUNTREGISTERS)

{

throw new MessageException($"Error: register out of range ({num})");

}

}

/\*

\* Test addressField argument.

\*/

private void TestAddrArg( string arg )

{

/\* test numeric addressField \*/

if (IsNumber(arg) && !Regex.IsMatch(arg, "[0-9]"))

{

throw new MessageException("Bad character in addressField!");

}

}

private static bool CheckCommands(string opcode, Command[] opcodes)

{

return opcodes.Any(op => CommonUtil.Commands[op] == opcode);

}

private static int CreateNumber(Command com, string arg0, string arg1, string arg2)

{

return ((int)com << CommonUtil.OffsetOpcode) | (int.Parse(arg0) << CommonUtil.OffsetArg0) | (int.Parse(arg1) << CommonUtil.OffsetArg1)

| int.Parse(arg2) << CommonUtil.OffsetArg2;

}

private static int CreateNumber(Command com, string arg0, string arg1, int arg2)

{

return ((int)com << CommonUtil.OffsetOpcode) | (int.Parse(arg0) << CommonUtil.OffsetArg0) | (int.Parse(arg1) << CommonUtil.OffsetArg1)

| arg2 << CommonUtil.OffsetArg2;

}

private static int CreateNumber(Command com, string arg0, string arg1)

{

return ((int)com << CommonUtil.OffsetOpcode) | (int.Parse(arg0) << CommonUtil.OffsetArg0) | (int.Parse(arg1) << CommonUtil.OffsetArg1);

}

private static int CreateNumber(Command com, string arg0)

{

return ((int)com << CommonUtil.OffsetOpcode) | (int.Parse(arg0) << CommonUtil.OffsetArg0);

}

private static int CreateNumber(Command com, int arg0)

{

return ((int)com << CommonUtil.OffsetOpcode) | (arg0 << CommonUtil.OffsetArg0);

}

private static int CreateNumber(Command com)

{

return ((int)com << CommonUtil.OffsetOpcode);

}

}

}

**Додаток Б. Вихідний код симулятора.**

using System;

using System.Collections;

using System.IO;

namespace Assembler

{

public class SSol : ISol

{

private StreamWriter sw = null;

public void Run(string compiled, string report)

{

var state = new State();

StreamReader file = null;

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

address 0) \*/

try

{

file = new StreamReader(compiled);

sw = new StreamWriter(report);

for (state.numMemory = 0; !file.EndOfStream;state.numMemory++)

{

var line = file.ReadLine();

if (state.numMemory >= CommonUtil.NUMMEMORY)

{

throw new MessageException("Exceeded memory size!");

}

if (int.TryParse(line, out int t))

{

state.mem[state.numMemory] = t;

}

else

{

throw new MessageException($"error in reading address {state.numMemory + 1}");

}

sw.WriteLine("memory[ {0} ]={1}", state.numMemory, state.mem[state.numMemory]);

}

/\* run never returns \*/

Start(state);

}

finally

{

file?.Close();

sw?.Close();

}

}

private void Start(State state)

{

var isHalt = false;

var instructions = 0;

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

for (; !isHalt; instructions++)

{

/\* infinite loop, exits when it executes halt \*/

PrintState(state, sw);

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

{

throw new MessageException($"PC went out of the memory range {state.pc}");

}

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

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

var opcode = (Command)(state.mem[state.pc] >> CommonUtil.OffsetOpcode);

var arg0 = ( state.mem[state.pc] >> CommonUtil.OffsetArg0 ) & CommonUtil.MaskForArg;

var arg1 = ( state.mem[state.pc] >> CommonUtil.OffsetArg1 ) & CommonUtil.MaskForArg;

var arg2 = ( state.mem[state.pc] >> CommonUtil.OffsetArg2) & CommonUtil.MaskForArg; /\* only for add, nand \*/

var addressField = ConvertNum(state.mem[state.pc] & CommonUtil.MaskForAddr); /\* for beq, lw, sw \*/

state.pc++;

if (opcode == Command.ADD)

{

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

}

else if (opcode == Command.NAND)

{

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

}

else if (opcode == Command.JALR)

{

state.reg[arg1] = state.pc;

state.pc = arg0 != 0 ? state.reg[arg0] : 0;

}

else if (opcode == Command.MUL)

{

state.reg[arg2] = Math.Abs(state.reg[arg0]) \* Math.Abs(state.reg[arg1]);

}

else

{

int st0;

int st1;

if (opcode == Command.AND)

{

if (arg0 != 0 || arg1 != 0 || arg2 != 0)

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

else

{

if (state.stack.Count > 0)

{

st0 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

if (state.stack.Count > 0)

{

st1 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

state.stack.Push(st0 & st1);

}

}

else if (opcode == Command.DIV)

{

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

{

throw new MessageException($"Divide by zero({state.numMemory + 1})");

}

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

}

else if (opcode == Command.IMUL)

{

if (arg0 != 0 || arg1 != 0 || arg2 != 0)

state.reg[arg2] = state.reg[arg0] \* state.reg[arg1];

else

{

if (state.stack.Count > 0)

{

st0 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

if (state.stack.Count > 0)

{

st1 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

state.stack.Push(st0 \* st1);

}

}

else if (opcode == Command.XIDIV)

{

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

{

throw new MessageException($"Divide by zero({state.numMemory + 1})");

}

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

int tmp = state.reg[arg0];

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

state.reg[arg1] = tmp;

}

else if (opcode == Command.CMPGE)

{

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

}

else if (opcode == Command.LW)

{

if (state.reg[arg0] + addressField < 0 || state.reg[arg0] + addressField >= CommonUtil.NUMMEMORY)

{

throw new MessageException($"Address out of bounds({state.reg[arg0] + addressField})");

}

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

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

{

maxMem = state.reg[arg0] + addressField;

}

}

else if (opcode == Command.SW)

{

if (state.reg[arg0] + addressField < 0 || state.reg[arg0] + addressField >= CommonUtil.NUMMEMORY)

{

throw new MessageException($"Address out of bounds({state.reg[arg0] + addressField})");

}

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

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

{

maxMem = state.reg[arg0] + addressField;

}

}

else if (opcode == Command.BEQ)

{

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

{

state.pc += addressField;

}

}

else if (opcode == Command.JMAE)

{

if (Math.Abs(state.reg[arg0]) >= Math.Abs(state.reg[arg1]))

{

state.pc += addressField;

}

}

else if (opcode == Command.JMNAE)

{

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

{

state.pc += addressField;

}

}

else if (opcode == Command.JNE)

{

if (state.ZF != 0)

state.pc = addressField;

}

else if (opcode == Command.HALT)

{

sw.WriteLine("machine halted");

sw.WriteLine($"total of {instructions + 1} instructions executed");

sw.WriteLine("final state of machine:");

PrintState(state, sw);

isHalt = true;

}

else if (opcode == Command.PUSH)

{

if (state.stack.Count == CommonUtil.MAXSTACKSIZE)

{

throw new MessageException("Stack is full!");

}

state.stack.Push(state.reg[arg0]);

}

else if (opcode == Command.POP)

{

if (state.stack.Count == 0)

{

throw new MessageException("Stack is empty!");

}

state.reg[arg0] = state.stack.Pop();

}

else if (opcode == Command.XOR)

{

if (arg0 != 0 || arg1 != 0 || arg2 != 0)

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

else

{

if (state.stack.Count > 0)

{

st0 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

if (state.stack.Count > 0)

{

st1 = state.stack.Pop();

}

else

throw new MessageException("Stack is empty!");

state.stack.Push(st0 ^ st1);

}

}

else if (opcode == Command.BSR)

{

BitArray arr = new BitArray(new int[] { state.reg[arg0] });

int t = 0;

for (var i = 0;i<arr.Count;++i)

{

if (arr[i])

{

state.reg[arg1] = i;

t = 1;

break;

}

}

state.ZF = t;

}

else if (opcode == Command.BSF)

{

BitArray arr = new BitArray(new int[] { state.reg[arg0] });

int t = 0;

for (var i = arr.Count - 1; i >= 0; ++i)

{

if (arr[i])

{

state.reg[arg1] = i;

t = 1;

break;

}

}

state.ZF = t;

}

else

{

sw.WriteLine($"error: illegal opcode 0x{opcode:X}");

}

}

state.reg[0] = 0;

}

}

private static void PrintState(State state, TextWriter sw)

{

sw.WriteLine("\n@@@\nstate:");

sw.WriteLine($"\tpc {state.pc}");

sw.WriteLine("\tmemory:");

for (var i = 0; i < state.numMemory; i++)

{

sw.WriteLine($"\t\tmem[ {i} ] = {state.mem[i]}");

}

sw.WriteLine("\tregisters:");

for (var i = 0; i < CommonUtil.COUNTREGISTERS; i++)

{

sw.WriteLine($"\t\treg[ {i} ] = {state.reg[i]}");

}

sw.WriteLine($"\tStack Top: {state.stack.Count}");

sw.WriteLine("\tStack:");

var lst = state.stack.ToArray();

for (var i = 0;i<state.stack.Count;++i)

{

sw.WriteLine($"\t\tstack[ {i} ] = {lst[i]}");

}

sw.WriteLine("\tFlag:");

sw.WriteLine($"\t\tZF {state.ZF}");

sw.WriteLine("end state");

}

private static int ConvertNum(int num)

{

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

var test = num & ( 1 << 15 );

if ( test > 0 )

{

num -= 1 << 16;

}

return num;

}

}

}