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 №19

Виконав:

ст. групи КІ-305

Лихограй А. В.

Прийняв:

Пісоцький М. О.

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.

В другій частині здійснюється розробка симулятора, який може відсимулювати машинний код, а також розробка тестів для тестування розробленої програми.

Зміст

Вихідні дані для проектування:

Перелік всіх інструкцій, розрядностей, кількості регістрів та розміру памяті:

1. 8 додаткових інструкцій без використання регістрів стану:

   3 – арифметичні;

   3 – логічні;

   2 – керування.

2. 3 додаткові інструкції з використання регістрів стану.

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

Згідно мого варіанту таблиця вхідних даних:

Таблиця 1.1 – Варіант №19.

№	Розряд-ність	Арифметичні			Логічні			Керування		Прапорці				Адре-сація
		1	2	3	4	5	6	7	8		1	2	3	
19	5	1	3	10	2	3	13	2	15	CF	3	4	6	3

1. Розрядність шини, розмір пам’яті та регістрового файлу:

Таблиця 1.2 – Розрядність.

№	Розрядність шини даних	Розмір пам’яті Байт	Розмір регістрового файлу(к-сть регістрів)
5	48	16777216	64

2. Додаткові 8 команд: 3 арифметичні, 3 логічні та 2 команди керування згідно варіанту. Команди не мають повторюватися.

Таблиця 1.3 – Арифметичні команди.

№	Мнемонічнй код	Зміст
1	DEC regA	Зменшити regA на 1
3	DIV regA regB destReg	Беззнакове ділення destReg=regA/regB
10	XIMUL regA regB destReg	Знакове множення і обмін операндів місцями destReg=regA*regB

Таблиця 1.4 – Логічні команди.

№	Мнемонічний код	Зміст
2	XOR regA regB destReg	Додавання по модулю 2: destReg=regA # regB
3	SHL regA regB destReg	Логічний зсув вліво destReg=regA << regB
13	MOV regA destReg	Переміщення даних destReg= regA

Таблиця 1.5 – Команди керування.

№	Мнемонічний код	Зміст
2	JMAE regA regB offSet	Беззнакове більше/рівно if (regA>= regB) PC=PC+1+offSet
15	JMNGE regA regB offSet	Знакове не більше/рівно if (regA !>= regB) PC=PC+1+offSet

3. Додатковий спосіб адресації, який буде підтримувати лише дві команди.

Таблиця 1.6 – Додаткова адресація.

№	Адресація
3	Непряма

4. Регістри стану: CF –регістр переносу.

Таблиця 1.7 – регістр переносу CF.

<table>
<tr><th>№</th><th>Мнемонічний код</th><th>Зміст</th></tr>
<tr><td>3</td><td>BT regA regB</td><td>Копіює значення біта з regA по позиції regB в CF = regA[regB]</td></tr>
<tr><td>4</td><td>CMP regA regB</td><td>Порівняти regA regB і встановити прапорці:
<table>
<tr><td></td><td>CF</td></tr>
<tr><td>regA < regB</td><td>1</td></tr>
<tr><td>regA = regB</td><td>0</td></tr>
<tr><td>regA > regB</td><td>0</td></tr>
</table></td></tr>
<tr><td>6</td><td>RCL regA regB destReg</td><td>Зсунути циклічно вліво через CF destReg=regA << regB</td></tr>
</table>

1. Аналітичний розділ

1.1 Архітектурні принципи

CISC (англ. ComplexInstructionSetComputer — комп'ютер зі складним набором команд) — це архітектура системи команд, в якій більшість команд є комплексними, тобто реалізують певний набір простіших інструкцій процесора або шляхом зіставлення з кожною CISC-командою певної мікропрограми, або принаймні можуть бути зведені до набору таких простих інструкцій.

Приведемо 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.

1.2 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.

1.3 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.

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. Такий 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. Проектний розділ

2.1 Алгоритми роботи розробленого асемблера

Перш ніж почати проектування, потрібно визначитись з синтаксисом асемблерної програми, яка буде перетворювати асемблерний код в машині команди.

Формат лінійки асемблерного коду буде наступний:

мітка <*пробіл*> *інструкція* <*пробіл*>*поле№1*<*пробіл*>*поле№2*<*пробіл*>*поле№3*<*пробіл*>*коментар*

<*пробіл*> означає послідовність табуляцій і/або пробілів.

Поле№1-№3 – номер регістру або адреса (символьна або числова) в залежності від команди.

Алгоритм перетворення асемблерних команд у машинні інструкції передбачає 2 «проходи» по вхідній текстовій (асемблерній) програмі. У процесі першого проходу програма перевіряє коректність синтаксису команд: перевіряє кожну вхідну команду на наявність її у множині команд комп'ютера, перевіряє відповідність кількості операндів вхідної команди, правильність зазначених у команді операндів.

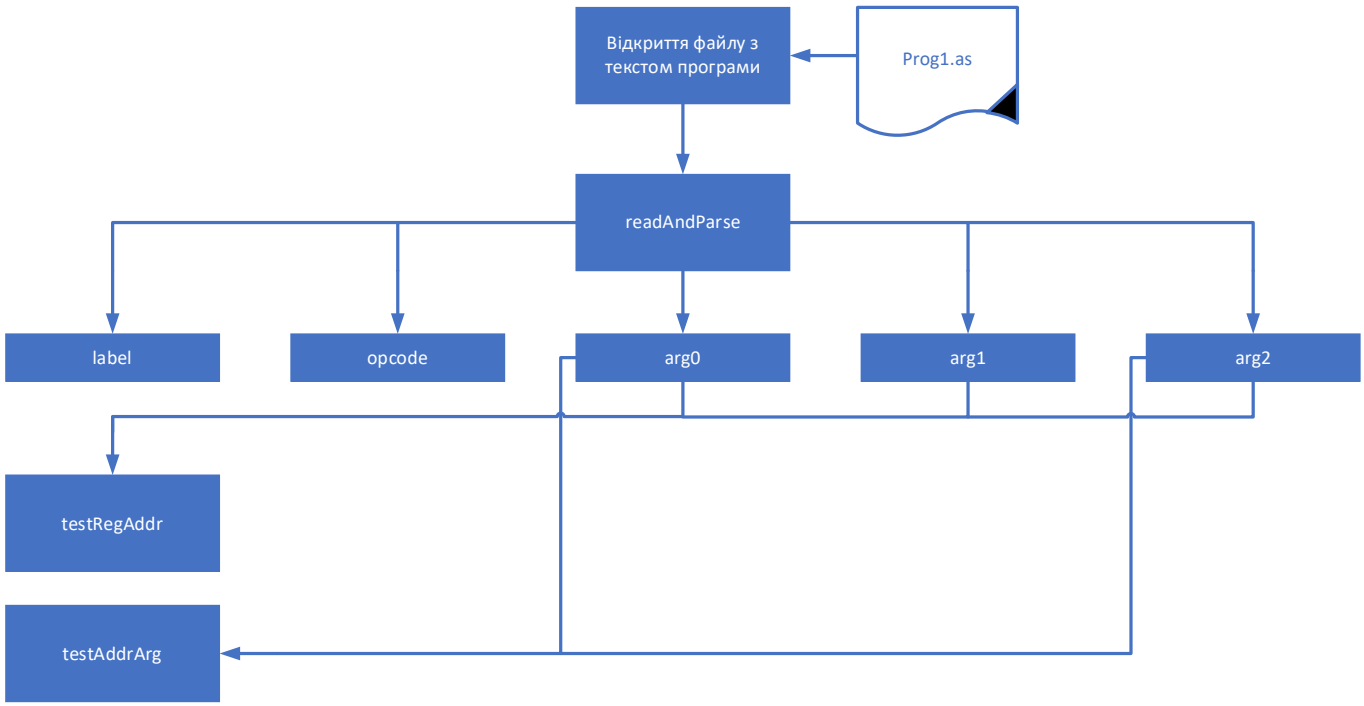

Рис. 2.1.1. Алгоритм першого проходу асемблера.

Блок *readAndParse* зчитує з вхідного файлу 1 рядок і отримує з нього значення таких полів: *мітка, код операції, операнд1, операнд2* та *операнд3*. Блок *testRegArg* перевіряє правильність номеру регістру. Блок *testAddrArg* перевіряє коректність вказання адреси (символьної або числової). На схемі, до цього блоку підведені тільки

перший та третій операнди. Це обумовлено специфікацією команд розробленого комп'ютера. Тобто, у наборі інструкцій мого комп'ютера немає такої інструкції, у якій другим операндом буде адреса. Перший операнд, як адресу, може використовувати тільки одна інструкція - *.fill* (збереження у мітці числа або адреси). З третім операндом, як адресою, працюють операції завантаження/збереження даних, операції керування *(lw,sw,jmae,jmnge,beq).*

У процесі другого проходу рядку програми виконується генерування відповідних машинних команд, тобто числового еквіваленту асемблерним командам.

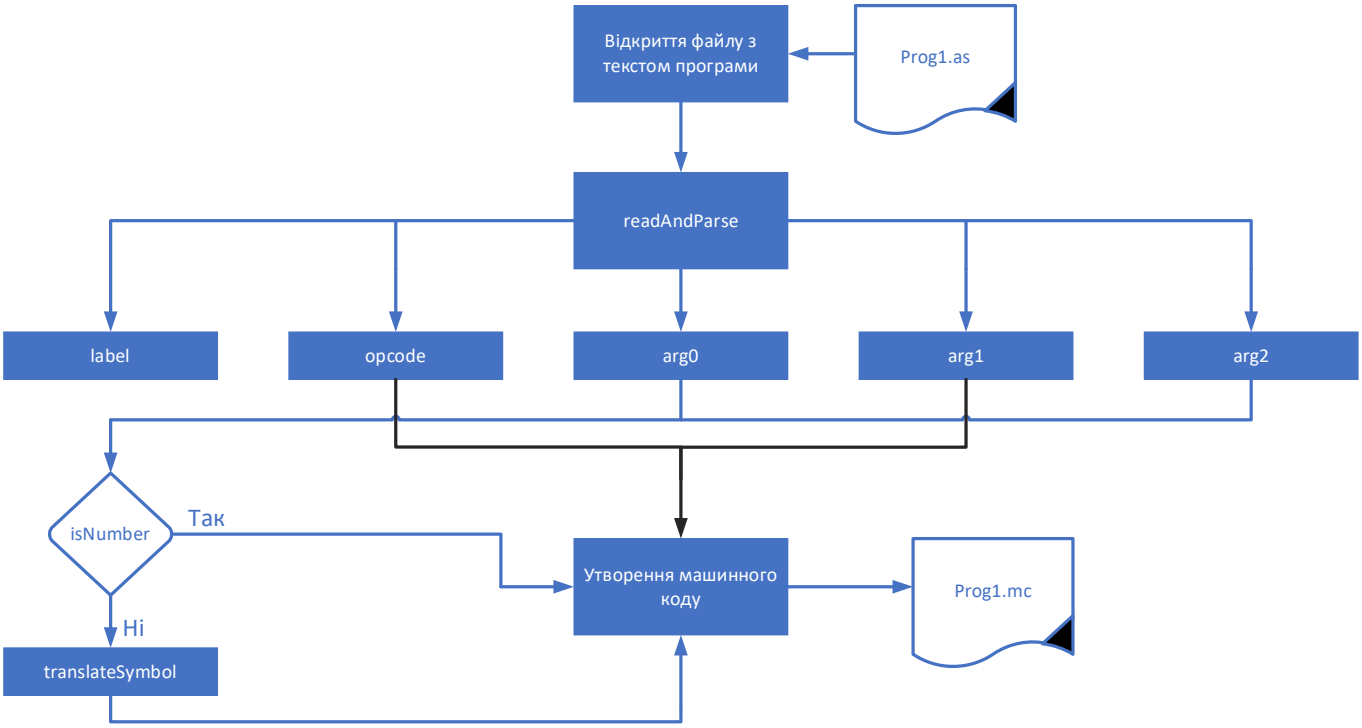

Рис. 2.1.2. Алгоритм другого проходу асемблера.

Блок *isNumber* перевіряє, чи є операнд числом. Якщо так, то програма записує це число у відповідній позиції адресного поля команди. Якщо ні, то переходить до блоку translateSymbol

translateSymbol – підставляє замість символьної адреси (мітки) числову. Числова адреса береться з, сформованої на першому проході, своєрідної таблиці міток, де вказані адреси для кожної мітки. Після виконання переходить до блоку формування машинного коду.

Утворення машинного коду – формує машинний код з коду операції та операндів. Цей блок генерує *«Prog1.mc»* - файл, де записана послідовність машшиних команд.

2.2 Алгоритми роботи розробленого симулятора

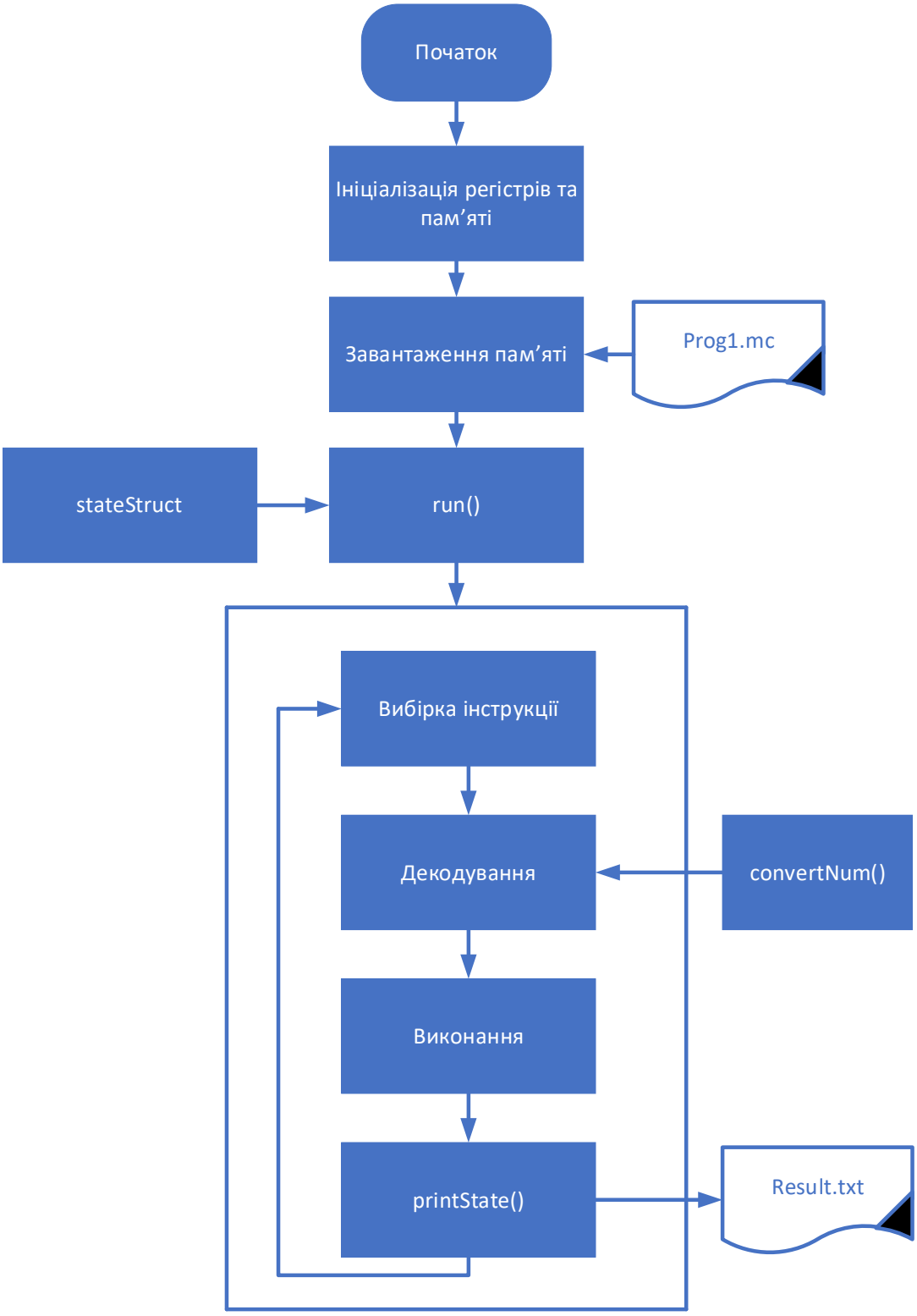

Рис. 2.2.1. Алгоритм роботи симулятора.

Симулятор починає свою роботу ініціалізацією пам’яті та регістрів нульовими значеннями. Наступним кроком відбувається завантаження програми у машинних кодах у пам’ять з текстового файлу. Далі відбувається покрокове виконання програми

(один крок – одна команда) та вивід стану у файл. У блоці *stateStruct* зберігається стан машини – значення регістрів, пам’яті, програмного лічильника та прапорців (CF, ZF, SF).

run() – виконує вибірку інструкції з пам’яті, декодує інструкцію та виконує її. У кінці кожного *run* циклу викликається блок *printState*, який «звітує» про проробленую роботу.

Додатковим завдання було - розробити додатковий спосіб адресації та передбачити, що 3 інструкції його підтримують. Адресація згідно з варіантом – **непряма**.

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

2.3 Функціональна схема комп’ютера 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. У СК є спеціальний регістр – лічильник команд (programCounter), в якому зберігається адреса інструкції. За прийнятою домовленістю нульовий регістр завжди містить 0.

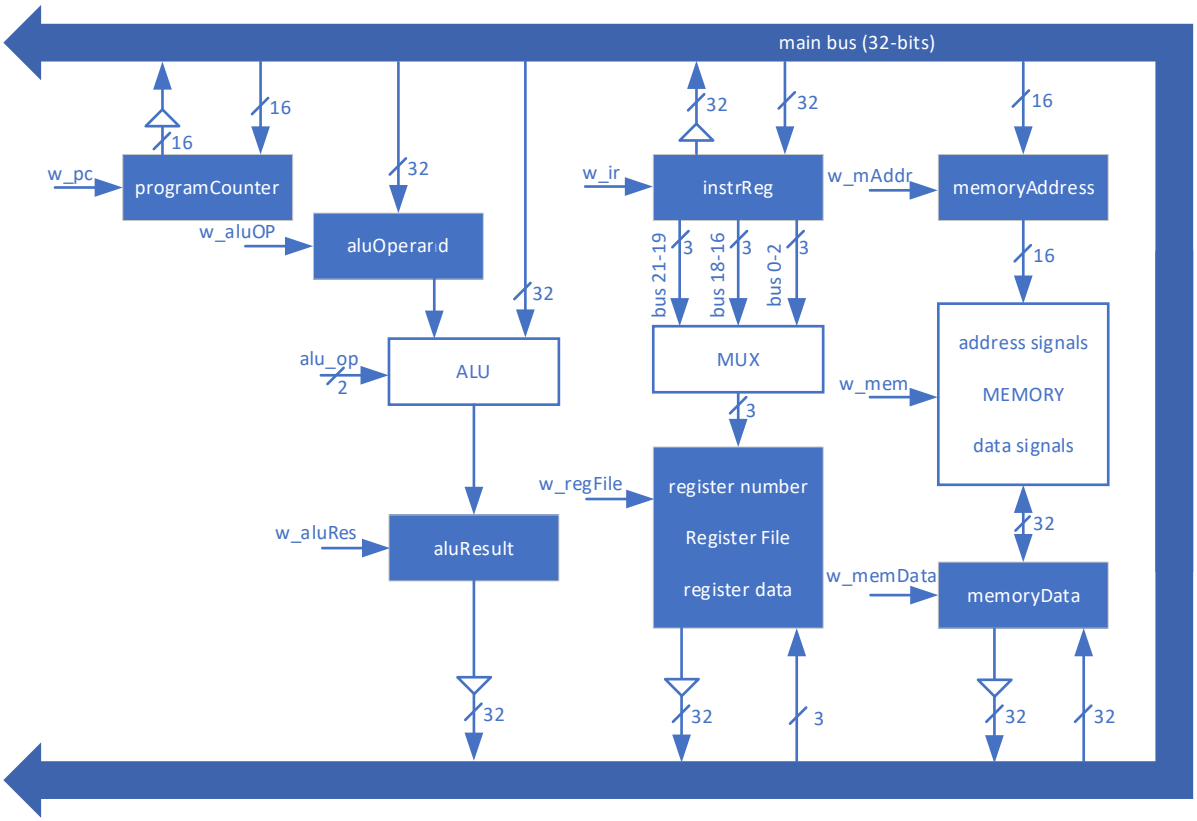

Рис. 2.3.1. Функціональна схема СК до модифікації.

В регістрі *instReg* міститься код інструкції, яка виконується. При роботі з пам'яттю *(MEMORY)* використовуються два регістри – в *memAddress* зберігається адреса пам'яті, а в *memData* значення, яке читається/записується в пам'ять.

Комп'ютер містить арифметико-логічний пристрій *(ALU)*, який може виконувати 3 дії: додавання, логічне І-НЕ і перевірку на рівність. *ALU* керується двохрозрядним сигналом *alu_op*.

2.4 Функціональна схема комп'ютера після модифікації

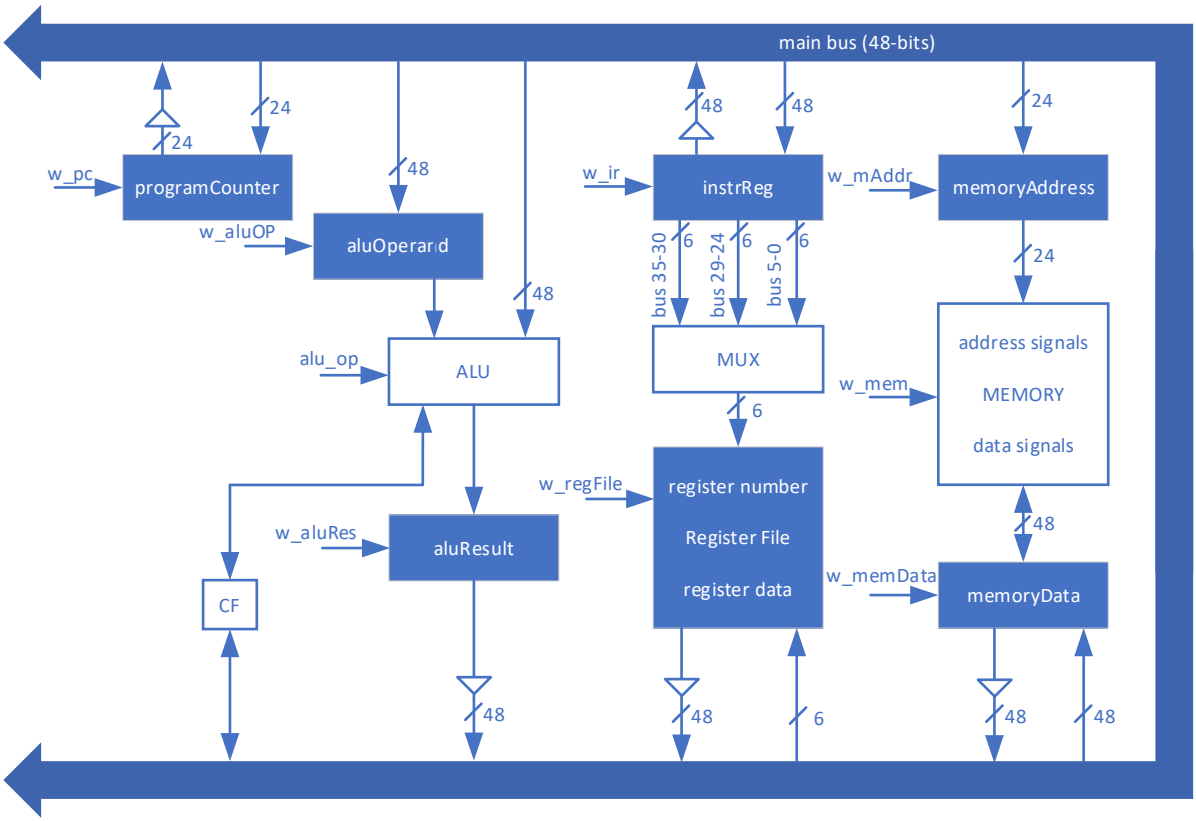

Рис. 2.4.1. Функціональна схема СК після модифікації.

Після модифікації відбулись зміни :

1. Шина даних збільшилась (48 бітів).
2. Додано 8 додаткових операцій:
   - 3 арифметичні операції (DEC, DIV, XIMUL);
   - 3 логічні операції (XOR, SHL, MOV);
   - 2 операції керування (JMAE, JMNGE).
3. Було додано прапорець CF, а також команди BT, CMP, RCL, які працюють в парі з прапорцем.
4. Встановлено розмір пам'яті: 16777216.
5. Збільшилось число регістрів з 8 до 64.

Отже , СК є 48 розрядним комп'ютером. У СК є 64 регістрів по 48 розрядів. Пам'ять складається з 16777216слів по 48 розрядів. Він підтримує 18 інструкцій, кожна з яких буде розписана нижче. За прийнятою домовленістю $0^{вий}$ регістр завжди містить 0 (це не обмовлено апаратними вимогами проте асемблерна програма ніколи не має змінювати значення 0-ого регістра, який ініціалізуються 0).

2.5 Формати інструкцій СК

СК підтримує 6 форматів інструкцій.

Інструкції R-типу (add, nand, div, ximul, xor, shl, rcl):

біти 47-42: не використовуються (= 0);

біт 41: opcodeType;

біти 40-36: код операції;

біти 35-30: reg A;

біти 29-24: reg B;

біти 32-6: не використовуються (= 0);

біти 5-0: destReg.

47-42	41	40-36	35-30	29-24	23-5	5-0
unused	opcodeType	opcode	regA	regB	*unused*	destReg

Рис.2.5.1. Інструкція R-типу після модифікації.

Інтсрукції I-типу (lw, sw, beq, jmae, jmnge):

біти 47-41: не використовуються (= 0);

біти 40-36: код операції;

біти 35-30: reg A;

біти 29-24: reg B;

біти 23-0: зміщення (23 біт, значення від -8388608 до 8388607).

47-41	40-36	35-30	29-24	23-0
unused	opcode	regA	regB	*offset*

Рис.2.5.2. Інструкція I-типу після модифікації.

Інструкції J-типу (jalr, bt, cmp):

біти 47-41: не використовуються (= 0);

біти 40-36: код операції;

біти 35-30: reg A;

біти 29-24: reg B;

біти 23-0: не використовуються (= 0).

47-41	40-36	35-30	29-24	23-0
unused	opcode	regA	regB	*unused*

Рис.2.5.3 Інструкція J-типу після модифікації.

Інструкції O-типу (halt):

біти 47-41: не використовуються (= 0);

біти 40-36: код операції;

біти 35-0: не використовуються (= 0).

47-41	40-36	35-0
unused	opcode	*unused*

Рис.2.5.4. Інструкція O-типу після модифікації.

Інструкції Y-типу (mov):

біти 47-41: не використовуються (= 0);

біти 40-36: код операції;

біти 35-30: reg A;

біти 29-6: не використовуються (= 0);

біти 5-0: destReg.

47-41	40-36	35-30	29-6	5-0
unused	opcode	regA	*unused*	destReg

Рис.2.5.5. Інструкція Y-типу після модифікації.

Інструкції M-типу (dec):

біти 47-41: не використовуються (= 0);

біти 40-36: код операції;

біти 35-30: reg A;

біти 29-0: не використовуються (= 0).

47-41	40-36	35-30	29-0
unused	opcode	regA	*unused*

Рис.2.5.6. Інструкція M-типу після модифікації.

2.6 Множина інструкцій комп’ютера

В таблиці наведено множину інструкцій СК:

Табл.2.6.1. Множина інструкцій.

<table>
<tr><th>№</th><th>Код інструкції</th><th>2-кове</th><th>Сутність інструкцій машини</th></tr>
<tr><td colspan="4">Інструкції R-типу</td></tr>
<tr><td>1</td><td>add regA regB destReg</td><td>00000</td><td>Додає вміст регістру regA до вмісту regB, та зберігає в destReg</td></tr>
<tr><td>2</td><td>nand regA regB destReg</td><td>00001</td><td>Виконує логічне побітове І-НЕ вмісту regA з вмістом regB, та зберігає в destReg</td></tr>
<tr><td>3</td><td>div regA regB destReg</td><td>01000</td><td>Беззнакове ділення destReg=regA/regB</td></tr>
<tr><td>4</td><td>ximul regA regB destReg</td><td>01001</td><td>Віднімання і обмін операндів місцями: destReg=regA-regB</td></tr>
<tr><td>5</td><td>xor regA regB destReg</td><td>01010</td><td>Додавання по модулю 2: destReg=regA # regB</td></tr>
<tr><td>6</td><td>shl regA regB destReg</td><td>01011</td><td>Логічний зсув вліво destReg=regA << regB</td></tr>
<tr><td>7</td><td>rcl regA regB destRg</td><td>10001</td><td>Зсунути циклічно вліво через CF destReg=regA >> regB</td></tr>
<tr><td colspan="4">I-тип</td></tr>
<tr><td>8</td><td>lw regA regB offSet</td><td>00010</td><td>Завантажує regB з пам’яті. Адреса пам’яті формується додаванням зміщення до вмісту regA.</td></tr>
<tr><td>9</td><td>sw regA regB offSet</td><td>00011</td><td>Зберігає вміст регістру regB в пам’ять. Адреса пам’яті формується додаванням зміщення до вмісту regA.</td></tr>
<tr><td>10</td><td>beq regA regB offSet</td><td>00100</td><td>Якщо вміст регістрів regA та regB однаковий, виконується перехід на адресу програмний лічильник(ПЛ) + 1 + зміщення, в ПЛ зберігається адреса поточної тобто beq інструкції.</td></tr>
<tr><td>11</td><td>jmae regA regB offSet</td><td>01101</td><td>Беззнакове більше/рівно if (regA>= regB) PC=PC+1+offSet</td></tr>
<tr><td>12</td><td>jmnge regA regB offSet</td><td>01110</td><td>Знакове не більше/рівно if (regA !>= regB) PC=PC+1+offSet</td></tr>
<tr><td colspan="4">J-тип</td></tr>
<tr><td>13</td><td>jalr regA regB</td><td>00101</td><td>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.</td></tr>
<tr><td>14</td><td>cmp regA regB</td><td>10000</td><td><table><tr><td></td><td>CF</td></tr><tr><td>regA < regB</td><td>1</td></tr><tr><td>regA = regB</td><td>0</td></tr><tr><td>regA > regB</td><td>0</td></tr></table></td></tr>
</table>

Продовження табл.2.6.1.

№	Код інструкції	2-кове	Сутність інструкцій машини
J-тип			
15	bt regA regB	01111	Копіює значення біта з regA по позиції regB в CF = regA[regB]
O-тип			
16	halt	00110	Збільшує значення ПЛ на 1, потім припиняє виконання, стимулятор має повідомляти, що виконано зупинку.
Y-тип			
17	mov regA destReg	01100	Переміщення даних destReg= regA
M-тип			
18	dec regA	00111	Зменшити на 1

2.7 Потактове виконання команд

1) Інструкції R-типу: *add, nand, div, ximul, xor, shl, rcl*:
   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];
   3. instrReg <= memData;
   4. PC <= PC++;
   5. aluOP <= Reg[instrReg[35-30]]
   6. aluRes <=aluOP operation instrRg[29-24];
   7. Reg[instrReg[5-0]] <= aluRes;
2) Інструкції I-типу: *lw, sw, beq, jmae, jmnge*:

a) lw/sw:

   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];
   3. instrReg <= memData;
   4. PC <= PC++;
   5. memAddr <= Reg[instrReg[29-24]] + instrReg [23-0];
   6. memData <= mem[mamAddr]; or memData <= Reg[instrReg[29-24]];
   7. mem[mamAddr] <= memData; or Reg[instrReg[29-24]] <= memData.

b) beq/jmae/jmnge:

   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];

3. instrReg <= memData;
4. PC <= PC + 4;
5. aluOP <= Reg[instrReg[35-30]]
6. aluRes <= Reg[instrReg [35-30]] logic_operation Reg[instrReg [29-24]];

Якщо умова виконується, то:

7. PC <= PC + instrReg[23-0] + 1;

3) Інструкція J-типу: *jarl, bt, cmp*:

a) jarl:

1. memAddr <= PC;
2. memData <= MEMORY[memAddr];
3. instrReg <= memData;
4. PC <= PC++;
5. Reg[instrReg [29-24]] <= PC;
6. PC <= Reg[instrReg[35-30]];
7. If Reg[instrReg[35-30]] == Reg[instrReg [29-24]] then:

   True: Reg[instrReg[35-30]] <= PC+1;

   False: PC <= Reg[instrReg[29-24]];

b) bt:

1. memAddr <= PC;
2. memData <= MEMORY[memAddr];
3. instrReg <= memData;
4. PC <= PC++;
5. aluOP <= Reg[instrReg[35-30]]
6. aluRes <=aluOP operation Reg[IR[29-24]];
7. CF <= aluRes & 0x0000000001;

c) cmp:

1. memAddr <= PC;
2. memData <= MEMORY[memAddr];
3. instrReg <= memData;
4. PC <= PC++;

5. Reg[instrReg [29-24]] <= PC;
6. PC <= Reg[instrReg[35-30]];
7. If Reg[instrReg[35-30]] < Reg[instrReg [29-24]] then:

   True: CF <= 1;

   False: CF <= 0;

4) Інструкція O-типу: *halt*:
   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];
   3. instrReg <= memData;
   4. PC <= PC++;
   5. Stop.
5) Інструкція Y-типу: *mov*:
   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];
   3. instrReg <= memData;
   4. PC <= PC++;
   5. aluOP <= Reg[instrReg[35-30]];
   6. aluRes <=aluOP;
   7. Reg[instrReg[5-0]] <= aluRes;
6) Інструкція M-типу: *dec*:
   1. memAddr <= PC;
   2. memData <= MEMORY[memAddr];
   3. instrReg <= memData;
   4. PC <= PC +;
   1. Reg[instrReg[35-30]] <= Reg[instrReg[35-30]] – 1;

3. Тестування

Для перевірки коректної роботи спроектованого комп'ютера написані спеціальні програми. Ці програми використовують усі наявні команди, що були розроблені.

1. Програма для перевірки роботи арифметичних операцій (dec, div, ximul).

В регістрі reg[2] міститься результат виконання операції *div(беззнакове ділення)*, reg[3] – *ximul(знакове множення і обмін операндів місцями)*, reg[1] – *dec(зменшення на одиницю regA*).

Табл.3.1. Арифметичні операції.

Код				Машинний код	Початковий стан	Кінцевий стан
lw	0	1	x1	137455730694	STATE: PC: 0 MEMORY: mem[0] 137455730694 mem[1] 137472507911 mem[2] 550863110147 mem[3] 619582586884 mem[4] 482110078976 mem[5] 412316860416 mem[6] -24 mem[7] 6 Registers: reg[0] 0 reg[1] 0 reg[2] 0 reg[3] 0 reg[4] 0 reg[5] 0 reg[6] 0 reg[7] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 6 MEMORY: mem[0] 137455730694 mem[1] 137472507911 mem[2] 550863110147 mem[3] 619582586884 mem[4] 482110078976 mem[5] 412316860416 mem[6] -24 mem[7] 6 Registers: reg[0] 0 reg[1] 5 reg[2] -24 reg[3] 4 reg[4] -144 reg[5] 0 reg[6] 0 reg[7] 0 … reg[63] 0 CF = 0 END STATE
lw	0	2	x2	137472507911		
div	1	2	3	550863110147		
ximul	1	2	4	619582586884		
dec	1			482110078976		
halt				412316860416		
x1	.fill	-24		-24		
x2	.fill	6		6		

2. Програма для перевірки роботи логічних операцій (xor, shl, mov).

В регістрі reg[3] міститься результат виконання операції *shl(логічний зсув ліворуч)*, reg[4] – *xor(додавання по модулю два)*, reg[5] – *mov(переміщення з regA в destReg*).

Табл.3.2. Логічні операції.

Код				Машинний код	Початковий стан	Кінцевий стан
lw	0	1	x1	137455730695	STATE: PC: 0 MEMORY: mem[0] 137455730695 mem[1] 137472507912 mem[2] 137522839561 mem[3] 757021540355 mem[4] 688302063620 mem[5] 825707462661 mem[6] 412316860416 mem[7] 24 mem[8] 6 mem[9] 5 Registers: reg[0] 0 reg[1] 0 reg[2] 0 reg[3] 0 reg[4] 0 reg[5] 0 reg[6] 0 reg[7] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 7 MEMORY: mem[0] 137455730695 mem[1] 137472507912 mem[2] 137522839561 mem[3] 757021540355 mem[4] 688302063620 mem[5] 825707462661 mem[6] 412316860416 mem[7] 24 mem[8] 6 mem[9] 5 Registers: reg[0] 0 reg[1] 24 reg[2] 6 reg[3] 1536 reg[4] 30 reg[5] 24 reg[6] 0 reg[7] 0 … reg[63] 0 CF = 0 END STATE
lw	0	2	x2	137472507912		
lw	0	5	x3	137522839561		
shl	1	2	3	757021540355		
xor	1	2	4	688302063620		
mov	1	0	5	825707462661		
halt				412316860416		
x1	.fill	24		24		
x2	.fill	6		6		
x3	.fill	5		5		

3. Програма для перевірки роботи операцій керування (jmae, jmnge).

Команда *jmae (regA≥regB)* повернула значення *«true»*, тому виконалась команда *add* і результат зберігся в reg[7]. Команда *jmnge (regA!≥regB)* повернула значення *«false»*, тому виконалась команда *add* та *xor* і результати збереглися в reg[8] та reg[9].

Табл.3.3. Операції керування.

Код	Машинний код	Початковий стан	Кінцевий стан
lw 0 1 x1 lw 0 2 x2 lw 0 3 x3 lw 0 4 x4 jmae 1 2 pointA add 1 2 6 pointA xor 1 2 7 jmnge 3 4 pointB add 3 4 8 pointB xor 3 4 9 halt x1 .fill 10 x2 .fill 5 x3 .fill 7 x4 .fill 4	137455730699 137472507916 137489285133 137506062350 894460493825 1107296262 688302063623 965361008641 3288334344 690483101705 412316860416 10 5 7 4	STATE: PC: 0 MEMORY: mem[0] 137455730699 mem[1] 137472507916 mem[2] 137489285133 mem[3] 137506062350 mem[4] 894460493825 mem[5] 1107296262 mem[6] 688302063623 mem[7] 965361008641 mem[8] 3288334344 mem[9] 690483101705 mem[10] 412316860416 mem[11] 10 mem[12] 5 mem[13] 7 mem[14] 4 Registers: reg[0] 0 reg[1] 0 reg[2] 0 reg[3] 0 reg[4] 0 reg[5] 0 reg[6] 0 reg[7] 0 reg[8] 0 reg[9] 0 reg[10] 0 reg[11] 0 reg[12] 0 reg[13] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 11 MEMORY: mem[0] 137455730699 mem[1] 137472507916 mem[2] 137489285133 mem[3] 137506062350 mem[4] 894460493825 mem[5] 1107296262 mem[6] 688302063623 mem[7] 965361008641 mem[8] 3288334344 mem[9] 690483101705 mem[10] 412316860416 mem[11] 10 mem[12] 5 mem[13] 7 mem[14] 4 Registers: reg[0] 0 reg[1] 10 reg[2] 5 reg[3] 7 reg[4] 4 reg[5] 0 reg[6] 0 reg[7] 15 reg[8] 11 reg[9] 3 reg[10] 0 reg[11] 0 reg[12] 0 reg[13] 0 … reg[63] 0 CF = 0 END STATE

4. Програма для перевірки роботи операцій з регістром переносу (bt, cmp, rcl).

Табл.3.4. Операції з регістром переносу.

Код:				Машинний код	Початковий стан	Кінцевий стан
lw lw bt halt x1 x2	0 0 1 .fill .fill	1 2 2 10 4	x1 x2	137455730692 137472507909 103189944729 6 412316860416 10 4	STATE: PC: 0 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1031899447296 mem[3] 412316860416 mem[4] 10 mem[5] 4 Registers: reg[0] 0 reg[1] 0 reg[2] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 4 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1031899447296 mem[3] 412316860416 mem[4] 10 mem[5] 4 Registers: reg[0] 0 reg[1] 10 reg[2] 4 … reg[63] 0 CF = 0 END STATE
Код:				Машинний код	Початковий стан	Кінцевий стан
lw lw cmp halt x1 x2	0 0 1 .fill .fill	1 2 2 7 8	x1 x2	137455730692 137472507909 110061892403 2 412316860416 7 8	STATE: PC: 0 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1100618924032 mem[3] 412316860416 mem[4] 7 mem[5] 8 Registers: reg[1] 0 reg[2] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 4 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1100618924032 mem[3] 412316860416 mem[4] 7 mem[5] 8 Registers: reg[1] 7 reg[2] 8 … reg[63] 0 CF = 1 END STATE
Код:				Машинний код	Початковий стан	Кінцевий стан
lw lw rcl halt x1 x2	0 0 1 .fill .fill	1 2 2 14 3	x1 x2 3	137455730692 137472507909 116933840077 1 412316860416 14 3	STATE: PC: 0 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1169338400771 mem[3] 412316860416 mem[4] 14 mem[5] 3 Registers: reg[0] 0 reg[1] 0 reg[2] 0 reg[3] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 4 MEMORY: mem[0] 137455730692 mem[1] 137472507909 mem[2] 1169338400771 mem[3] 412316860416 mem[4] 14 mem[5] 3 Registers: reg[0] 0 reg[1] 14 reg[2] 3 reg[3] 113 … reg[63] 0 CF = 1 END STATE

5. Програма для перевірки роботи операцій з непрямою адресацією:

СК підтримує операції з непрямою адресацією лише для R-типу інструкцій: *div, ximul, xor*. В асемблерному коді виділення непрямої адресації від стандартної виглядає так: *#div, #ximul, #xor*. В кожній з операцій regA, regB одинакові і відповідно: reg[3] зберігає значення «2», що вказує на reg[1], який зберігає «54»; reg[4] зберігає значення «2», що вказує на reg[2], який зберігає «6». Аналогічно й регістри результату.

Схематично це можна зобразити так:

Табл.3.5. Схематичне зображення.

reg[reg[5]]=reg[reg3] div reg[reg4]	reg[reg[6]]=reg[reg3] xor reg[reg4]	reg[reg[7]]=reg[reg3] ximul reg[reg4]
reg[5] = reg[1] div reg[2]	reg[6] = reg[1] xor reg[2]	reg[7] = reg[1] ximul reg[2]
reg[5] = 56 div 6	reg[6] = 56 xor 6	reg[7] = 56 ximul 6

Табл.3.6. Операції з непрямою адресацією.

Код				Машинний код	Початковий стан	Кінцевий стан
lw	0	1	x1	137455730699	STATE: PC: 0 MEMORY: mem[0] 137455730699 mem[1] 137472507916 mem[2] 137489285133 mem[3] 137506062350 mem[4] 137522839567 mem[5] 137539616784 mem[6] 137556394001 mem[7] 2752067403781 mem[8] 2889506357254 mem[9] 2820786880519 mem[10] 412316860416 mem[11] 54 mem[12] 6 mem[13] 1 mem[14] 2 mem[15] 5 mem[16] 6 mem[17] 7 Registers: reg[0] 0 reg[1] 0 reg[2] 0 reg[3] 0 reg[4] 0 reg[5] 0 reg[6] 0 reg[7] 0 … reg[63] 0 CF = 0 END STATE	STATE: PC: 11 MEMORY: mem[0] 137455730699 mem[1] 137472507916 mem[2] 137489285133 mem[3] 137506062350 mem[4] 137522839567 mem[5] 137539616784 mem[6] 137556394001 mem[7] 2752067403781 mem[8] 2889506357254 mem[9] 2820786880519 mem[10] 412316860416 mem[11] 54 mem[12] 6 mem[13] 1 mem[14] 2 mem[15] 5 mem[16] 6 mem[17] 7 Registers: reg[0] 0 reg[1] 6 reg[2] 54 reg[3] 1 reg[4] 2 reg[5] 9 reg[6] 48 reg[7] 324 … reg[63] 0 CF = 0 END STATE
lw	0	2	x2	137472507916		
lw	0	3	x3	137489285133		
lw	0	4	x4	137506062350		
lw	0	5	res1	137522839567		
lw	0	6	res2	137539616784		
lw	0	7	res3	137556394001		
#div	3	4	5	2752067403781		
#xor	3	4	6	2889506357254		
#ximul	3	4	7	2820786880519		
	halt			412316860416		
x1	.fill	54		54		
x2	.fill	6		6		
x3	.fill	1		1		
x4	.fill	2		2		
res1	.fill	5		5		
res2	.fill	6		6		
res3	.fill	7		7		

Висновок

В ході виконання курсової роботи був розроблений транслятор асемблерного коду, який дозволяє переводити всі задані варіантом команди у машинні інструкції. Даною програмою виконується безліч перевірок на відповідність правилам синтаксису, семантики і граматики.

Також була розроблена програма симулятора, що дозволяє дешифрувати всі машинні інструкції і виконати потрібну операцію, а також створює звіт потактового виконання програми з виводом стану пам’яті і регістрів. Симулятор відповідає всім поставленим вимогам, робить перевірки на помилки і в разі їх виникнення виводить відповідне повідомлення.

Для тестування роботи обох програм створено систему тестів, що спрямована на виявлення помилок в створенні машинних інструкцій, розшифруванні машинних інструкцій, а також у виконанні команд.

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://pdfs.semanticscholar.org/4bd9/9622f8f83e80c8145dda5852b9a3e8ab3d4a.pdf
6. Б. Керниган, Д. Ритчи - Язык программирования Си
7. https://uk.wikipedia.org/wiki/Способи_адресації_пам%27яті
8. http://uk.wikipedia.org/wiki/CISC
9. 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 10000
#define MAXLABELLENGTH 15
// Initialization of commands.
#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define JALR 5
#define HALT 6
#define DEC 7
#define DIV 8
#define XIMUL 9
#define XOR 10
#define SHL 11
#define MOV 12
#define JMAE 13
#define JMNGE 14
#define BT 15
#define CMP 16
#define RCL 17

long long int num;
// Instructions that form machine code.
long long int writeInstructionTypeR(int opcodeNum, int opcodeType, char* arg0, char* arg1,
char* arg2);
long long int writeInstructionTypeJ(int opcodeNum, char* arg0, char* arg1);
long long int writeInstructionTypeO(int opcodeNum);
long long int writeInstructionTypeM(int opcodeNum, char* arg0);
long long int writeInstructionTypeY(int opcodeNum, int opcodeType, char* arg0, char* arg2);
long long int writeInstructionTypeI(int opcodeNum, char* arg0, char* arg1, int
addressField);

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 addressField;
	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.
	for (address = 0; readAndParse(inFilePtr, label, opcode, arg0, arg1, arg2);
address++) {
		// 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, "dec") &&
			strcmp(opcode, "div") && strcmp(opcode, "ximul") &&
			strcmp(opcode, "xor") && strcmp(opcode, "shl") &&
			strcmp(opcode, "mov") && strcmp(opcode, "jmae") &&
			strcmp(opcode, "jmnge") && strcmp(opcode, "bt") &&
			strcmp(opcode, "cmp") && strcmp(opcode, "rcl") &&
			strcmp(opcode, "#ximul") && strcmp(opcode, "#xor") &&
			strcmp(opcode, "#div") && strcmp(opcode, ".fill")) {
			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, "#div") ||
			!strcmp(opcode, "ximul") ||
			!strcmp(opcode, "#ximul") ||
			!strcmp(opcode, "xor") ||
			!strcmp(opcode, "#xor") ||
			!strcmp(opcode, "shl") ||
			!strcmp(opcode, "jmae") ||
			!strcmp(opcode, "jmnge") ||
			!strcmp(opcode, "bt") ||
			!strcmp(opcode, "cmp") ||
			!strcmp(opcode, "rcl")) {
			testRegArg(arg0);
			testRegArg(arg1);
		}
		if (!strcmp(opcode, "add") || !strcmp(opcode, "nand") ||
			!strcmp(opcode, "div") || !strcmp(opcode, "ximul") ||
			!strcmp(opcode, "xor") || !strcmp(opcode, "mov") ||
			!strcmp(opcode, "shl") || !strcmp(opcode, "rcl") ||
			!strcmp(opcode, "#div") || !strcmp(opcode, "#ximul") ||
			!strcmp(opcode, "#xor")) {
			testRegArg(arg2);
		}
		if (!strcmp(opcode, "dec") || !strcmp(opcode, "mov")) {
			testRegArg(arg0);
```

```
	}

	// Check addressField.
	if (!strcmp(opcode, "lw") || !strcmp(opcode, "sw") ||
		!strcmp(opcode, "beq") || !strcmp(opcode, "jmb") ||
		!strcmp(opcode, "jmae") || !strcmp(opcode, "jmnge")) {
		testAddrArg(arg2);
	}
	if (!strcmp(opcode, ".fill")) {
		testAddrArg(arg0);
	}
	// Check for enough arguments.
	if ((strcmp(opcode, "halt")
		&& strcmp(opcode, ".fill") && strcmp(opcode, "jalr")
		&& strcmp(opcode, "dec") && strcmp(opcode, "bt")
		&& strcmp(opcode, "cmp") && arg2[0] == '\0') ||
		(!strcmp(opcode, "jalr") && arg1[0] == '\0') ||
		(!strcmp(opcode, "cmp") && arg1[0] == '\0') ||
		(!strcmp(opcode, "bt") && arg1[0] == '\0') ||
		(!strcmp(opcode, ".fill") && arg0[0] == '\0') ||
		(!strcmp(opcode, "dec") && 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 %d too long!\n", address);
			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;
	}
}


// 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 = writeInstructionTypeR(ADD, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "nand")) {
            num = writeInstructionTypeR(NAND, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "div")) {
            num = writeInstructionTypeR(DIV, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "#div")) {
            num = writeInstructionTypeR(DIV, 1, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "ximul")) {
            num = writeInstructionTypeR(XIMUL, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "#ximul")) {
            num = writeInstructionTypeR(XIMUL, 1, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "xor")) {
            num = writeInstructionTypeR(XOR, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "#xor")) {
            num = writeInstructionTypeR(XOR, 1, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "shl")) {
            num = writeInstructionTypeR(SHL, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "rcl")) {
            num = writeInstructionTypeR(RCL, 0, arg0, arg1, arg2);
      }
      else if (!strcmp(opcode, "jalr")) {
            num = writeInstructionTypeJ(JALR, arg0, arg1);
      }
      else if (!strcmp(opcode, "bt")) {
            num = writeInstructionTypeJ(BT, arg0, arg1);
      }
      else if (!strcmp(opcode, "cmp")) {
            num = writeInstructionTypeJ(CMP, arg0, arg1);
      }
      else if (!strcmp(opcode, "halt")) {
            num = writeInstructionTypeO(HALT);
      }
      else if (!strcmp(opcode, "dec")) {
            num = writeInstructionTypeM(DEC, arg0);
      }
      else if (!strcmp(opcode, "mov")) {
            num = writeInstructionTypeY(MOV, 0, arg0, arg2);
      }
      else if (!strcmp(opcode, "#mov")) {
            num = writeInstructionTypeY(MOV, 1, arg0, arg2);
      }
      else if (!strcmp(opcode, "lw") || !strcmp(opcode, "sw") ||
            !strcmp(opcode, "jmae") || !strcmp(opcode, "jmnge")
            || !strcmp(opcode, "beq")) {
      /* if arg2 is symbolic, then translate into an address */
      if (!isNumber(arg2)) {
      addressField = translateSymbol(labelArray, labelAddress, numLabels, arg2);
                  if (!strcmp(opcode, "beq") || !strcmp(opcode, "jmae")
                        || !strcmp(opcode, "jmnge")) {
                        addressField = addressField - address - 1;
                  }
```

```
                }
                else {
                        addressField = atoi(arg2);
                }
                if (addressField < -8388608 || addressField > 8388607) {
                        printf("Error: offset %d out of range!\n", addressField);
                        exit(1);
                }
                // Truncate the offset field, in case it's negative.
                addressField = addressField & 0x7FFFFF;
                if (!strcmp(opcode, "jmae")) {

                        num = writeInstructionTypeI(JMAE, arg0, arg1, addressField);
                }
                else if (!strcmp(opcode, "jmnge")) {
                        num = writeInstructionTypeI(JMNGE, arg0, arg1, addressField);
                }
                else if (!strcmp(opcode, "beq")) {
                        num = writeInstructionTypeI(BEQ, arg0, arg1, addressField);
                }
                else if (!strcmp(opcode, "lw")) {
                        num = writeInstructionTypeI(LW, arg0, arg1, addressField);
                }
                else if (!strcmp(opcode, "sw")) {
                        num = writeInstructionTypeI(SW, arg0, arg1, addressField);
                }
        }
        else if (!strcmp(opcode, ".fill")) {
                if (!isNumber(arg0)) {
                        num = translateSymbol(labelArray, labelAddress, numLabels, arg0);
                }
                else {
                        num = atoi(arg0);
                }
        }
        fprintf(outFilePtr, "%lld\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) > 63) {
                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);
                }
        }
}

// Functions that form machine codes.
long long int writeInstructionTypeR(int opcodeNum, int opcodeType, char* arg0, char* arg1,
char* arg2) {
        return num = ((long long int)opcodeType << 41) | ((long long int)opcodeNum << 36) |
                ((long long int)atoi(arg0) << 30) | ((long long int)atoi(arg1) << 24) |
atoi(arg2);
}

long long int writeInstructionTypeI(int opcodeNum, char* arg0, char* arg1, int
addressField) {
        return num = ((long long int)opcodeNum << 36) | ((long long int)atoi(arg0) << 30) |
((long long int)atoi(arg1) << 24) | addressField;
}

long long int writeInstructionTypeJ(int opcodeNum, char* arg0, char* arg1) {
        return num = ((long long int)opcodeNum << 36) | ((long long int)atoi(arg0) << 30) |
((long long int)atoi(arg1) << 24);
}

long long int writeInstructionTypeO(int opcodeNum) {
        return num = ((long long int)HALT << 36);
}

long long int writeInstructionTypeM(int opcodeNum, char* arg0) {
        return num = ((long long int)opcodeNum << 36) | ((long long int)atoi(arg0) << 30);
}
long long int writeInstructionTypeY(int opcodeNum, int opcodeType, char* arg0, char* arg2)
{
        return num = ((long long int)opcodeType << 41) | ((long long int)opcodeNum << 36) |
((long long int)atoi(arg0) << 30) | atoi(arg2);
}
```

Додаток Б. Вихідний код симулятора.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define NUMMEMORY 16777216 // Maximum number of words in memory.
#define NUMREGS 64 // Number of machine registers.
#define MAXLINELENGTH 1000
// Initialization of commands.
#define ADD 0
#define NAND 1
#define LW 2
#define SW 3
#define BEQ 4
#define JALR 5
#define HALT 6
#define DEC 7
#define DIV 8
#define XIMUL 9
#define XOR 10
#define SHL 11
#define MOV 12
#define JMAE 13
#define JMNGE 14
#define BT 15
#define CMP 16
#define RCL 17
typedef struct stateStruct {
	int pc;
	long long int* mem = new long long int[NUMMEMORY];
	long long int reg[NUMREGS];
	int numMemory;
	int CF;
} stateType;

void printState(stateType*);
void run(stateType);
long long 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.pc = 0;
	state.CF = 0;
	// Read machine-code file.
	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, "%lld", state.mem + state.numMemory) != 1) {
			printf("Error in reading address %d!\n", state.numMemory);
			exit(1);
		}
		printf("Memory[%d]=%lld\n", state.numMemory, state.mem[state.numMemory]);
	}
	printf("\n");
	run(state);
	return(0);
}

void run(stateType state)
{
	int arg0, arg1, arg2;
	int instructions = 0;
	int opcode, opcodeType = 0;
	long long int addressField;
	int maxMem = -1;
	for (; 1; instructions++) {
		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.
		opcodeType = state.mem[state.pc] >> 41 & 0x01;
		opcode = state.mem[state.pc] >> 36 & 0x1F;
		arg0 = (state.mem[state.pc] >> 30) & 0x3F;
		arg1 = (state.mem[state.pc] >> 24) & 0x3F;
		arg2 = state.mem[state.pc] & 0x3F;
		addressField = convertNum(state.mem[state.pc] & 0xFFFFFF);
		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 == 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 == BEQ) {
                if (state.reg[arg0] == state.reg[arg1]) {
                        state.pc += addressField;
                }
        }
        else if (opcode == JALR) {
                state.reg[arg1] = state.pc;
                if (arg0 != 0)
                        state.pc = state.reg[arg0];
                else
                        state.pc = 0;
        }
        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);
                printf("\n< Created by @Lyhotop >\n");
                exit(0);
        }
        else if (opcode == DEC) {
                state.reg[arg0]--;
        }
        else if (opcode == DIV) {
                if (opcodeType == 0)
                {
                        state.reg[arg2] = abs(state.reg[arg0] / state.reg[arg1]);
                }
                else {
                        state.reg[state.reg[arg2]] = abs(state.reg[state.reg[arg0]] /
state.reg[state.reg[arg1]]);
                }
        }
        else if (opcode == XIMUL) {
                if (opcodeType == 0)
                {
                        state.reg[arg2] = state.reg[arg0] * state.reg[arg1];
                        long long int temp = state.reg[arg0];
                        state.reg[arg0] = state.reg[arg1];
                        state.reg[arg1] = temp;
                }
                else {
                        state.reg[state.reg[arg2]] = state.reg[state.reg[arg0]] *
state.reg[state.reg[arg1]];
                        long long temp = state.reg[state.reg[arg0]];
                        state.reg[state.reg[arg0]] = state.reg[state.reg[arg1]];
                        state.reg[state.reg[arg1]] = temp;
                }
        }
        else if (opcode == XOR) {
                if (opcodeType == 0)
                {
                        state.reg[arg2] = state.reg[arg0] ^ state.reg[arg1];
                }
                else {
                        state.reg[state.reg[arg2]] = state.reg[state.reg[arg0]] ^
state.reg[state.reg[arg1]];
                }
        }
        else if (opcode == SHL) {
```

```
                state.reg[arg2] = state.reg[arg0] << state.reg[arg1];
            }
            else if (opcode == MOV) {
                if (opcodeType == 0) {
                    state.reg[arg2] = state.reg[arg0];
                }
                else {
                    state.reg[state.reg[arg2]] = state.reg[state.reg[arg0]];

                }
            }
            else if (opcode == JMAE) {
                if (abs(state.reg[arg0]) >= abs(state.reg[arg1])) {
                    state.pc += addressField;
                }
            }
            else if (opcode == JMNGE) {
                if (!(abs(state.reg[arg0]) >= abs(state.reg[arg1]))) {
                    state.pc += addressField;
                }
            }
            else if (opcode == BT) {
                long long int temp = state.reg[arg0];
                int i;
                for (i = 0; i < state.reg[arg1]; i++)
                    temp = temp >> 1;
                temp = temp & 0x000000001;
                state.CF = temp;
            }
            else if (opcode == CMP) {
                if (abs(state.reg[arg0]) < abs(state.reg[arg1])) {
                    state.CF = 1;
                }
                else {
                    state.CF = 0;
                }
            }
            else if (opcode == RCL) {
                state.reg[arg2] = state.reg[arg0];
                for (int i = 0; i < (state.reg[arg1]); i++)
                {
                    state.CF = (state.reg[arg2] & 0x20) >> 5;
                    state.reg[arg2] = state.reg[arg2] << 1;

                    if (state.CF == 1)
                    {
                        state.reg[arg2] = state.reg[arg2] | 1;
                    }
                }
            }
            else {
                printf("error: illegal opcode 0x%lli\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] %lld\n", i, statePtr->mem[i]);
        }
        printf("\tRegisters:\n");
        for (i = 0; i < NUMREGS; i++) {
                printf("\t\treg[%lld] %lld\n", i, statePtr->reg[i]);
        }
        printf("CF = %d\n", statePtr->CF);
        printf("END STATE\n");
}

long long int convertNum(int num)
{
        /* convert a 24-bit number into a 64-bit integer */
        if (num & (1 << 23)) {
                num -= (1 << 24);
        }
        return(num);
}
```