МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

КАФЕДРА ЕЛЕКТРОННИХ ОБЧИСЛЮВАЛЬНИХ МАШИН

МЕТОДИЧНІ ВКАЗІВКИ

до курсової роботи на тему

ПРОЕКТУВАННЯ КОМП'ЮТЕРА

з дисципліни
“ Архітектура комп’ютерів ”
для студентів базового напрямку 6.050102
“Комп’ютерна інженерія ”.

Затверджено
на засідання кафедри
“Електронні обчислювальні машини ”.
Протокол № 6 від 09.02.2011 р.

Львів – 2021

Методичні вказівки до курсової роботи на тему «Проектування комп’ютера» з дисципліни "Архітектура комп’ютера" для студентів базового напрямку 6.050102 “Комп’ютерна інженерія ” /Укл.: Мельник А.О., Троценко В.В., Клименко В.А.- Львів: Видавництво Національного університету “Львівська політехніка ”, 2021.- 28с.

Укладачі: Мельник А.О., док. техн. наук., проф.
Троценко В.В., канд. техн. наук., доц.
Клименко В.А., ас.

Відповідальний за випуск Березко Л. О., канд. техн. наук., доц.

Рецензент: Дунець Р.Б., проф. док. техн. наук., завідувач кафедри СКС,

ЗМІСТ

Вступ

Вступ містить короткі теоретичні відомості, архітектурний опис комп'ютера, архітектура якого досліджується.

Мета курсового проектування полягає в опануванні студентом знань про принципи дії та архітектуру прототипних варіантів CISC – ком'ютера. Крім того курсовий проект допоможе зрозуміти інструкції простої асемблерної мови та як транслювати програми в машинну мову.

Проект складається з трьох частин. В першій частині розробляється програма, яка перетворює вхідну асемблерну програму у відповідний код на мові машинних інструкцій. В другій частини розробляється поведінковий стимулятор результуючого машинного коду. В третій частині розробляється невеличка програма на асемблер ній мові.

Архітектурні принципи.

В ході виконання даного курсового проекту студент має ознайомитись та опанувати архітектуру CISC – комп'ютера. 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.

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.

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 (P), від'ємний результат (N), нульовий результат (Z), перенос (C), 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.

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.

КОП займає k розрядів. Ним може бути закодовано до $N = 2^k$ різних операцій. Кількість двійкових розрядів, які відводяться під код операції, вибирається таким чином, щоб ними можна було закодувати всі виконувані в даному комп'ютері операції. Якщо деякий комп'ютер може виконувати N_c різних операцій, то мінімальна розрядність поля коду операції k визначається наступним чином: $k = [\log N_c]$, де вираз в дужках означає заокруглення до більшого цілого.

Поле адреси (адресна частина) займає m розрядів. В ньому знаходяться адреси операндів. Кожна адреса займає m_i розрядів, де i - номер адреси (i=1,2,...1), 1 - кількість адресних полів. Кожною адресою можна адресувати пам'ять ємністю 2ті слів.

Розмір команди k + m повинен бути узгодженим з розміром даних, тобто бути з ним однаковим або кратним цілому числу, що спрощує організацію роботи з пам'яттю. Як правило, розмір команди рівний 8, 16, 32 біти.

При написанні програми крім двійкової можуть використовуватись й інші форми представлення команд: вісімкова, шістнадцяткова, символьна (мнемонічна). Використання вісімкового і шістнадцяткового кодування дозволяє скоротити записи і спростити роботу програміста. Як відомо 3 двійкових розряди (тріада) замінюються на 1 вісімковий, а 4 двійкових розряди (тетрада) - на 1 шістнадцятковий. Приклад:

$(000011111111)_2 = (0377)_8 = (0FF)_{16}$;

Мнемонічне кодування спрощує процес написання, читання і відлагодження програми. Основний принцип такого кодування - кожна команда представляється 3-х або 4-х буквеним символом, який показує назву команди. Деякі приклади мнемонічного кодування:

ADD - додати (add),

SUB - відняти (subtract),

MPY - перемножити (multiply),

DIV - поділити (divide),

Операнди також представляються символічно. Наприклад команда ADD R Y означає додавання вмісту комірки пам'яті Y до вмісту регістра R. Зауважимо, що операція виконується над вмістом, а не над адресою комірки пам'яті та регістра.

Таким чином, з'являється можливість написання машинних програм в символічній формі. Повний набір символічних назв і правила їх використання утворюють мову програмування, відому як асемблерна мова. Символічні імена називаються мнемонічними, а правила їх використання для створення команд і програм називаються синтаксисом мови.

Програма, яка переводить із мнемонічного коду асемблерної мови в машинний, називається асемблером. Команди, які використовуються для переводу вихідної програми в асемблерну, називаються командами асемблера. Ці команди вказують як інтерпретувати назви, де розмістити програму в пам'яті, яка кількість комірок пам'яті необхідна для зберігання даних.

Асемблерна мова є дуже далекою від мови людини і заставляє програміста думати виходячи з принципів побудови комп'ютера. Тому були створені мови високого рівня та компілятори, які переводять програми з цих мов на мову асемблера. Використання мов високого рівня має цілий ряд переваг в порівнянні з використанням асемблера. По-перше, програміст пише програми на мові, близькій до його мови спілкування. Більше того, мови високого рівня орієнтуються на класи вирішуваних задач. По-друге, скорочується час написання програм. І по-третє, мови високого рівня є незалежними від типу та архітектури комп'ютера, що дозволяє використовувати написані на цих мовах програми на всіх комп'ютерах, а програміста звільнити від знання їх структури та організації роботи.

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 опрандів 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.

Практично у всіх існуючих комп'ютерах використовується один або декілька з цих способів адресації. Тому в команду потрібно вводити 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.

Система машинних інструкцій СК

В першій частині даного курсового проекту будується архітектура «спрощеного комп'ютера», який скорочено будемо називати СК. Архітектура даного комп'ютера хоч і проста, але достатня для рішення складних задач. Для виконання даного курсового проекту необхідно володіти знаннями про набір інструкцій та формат інструкцій СК.

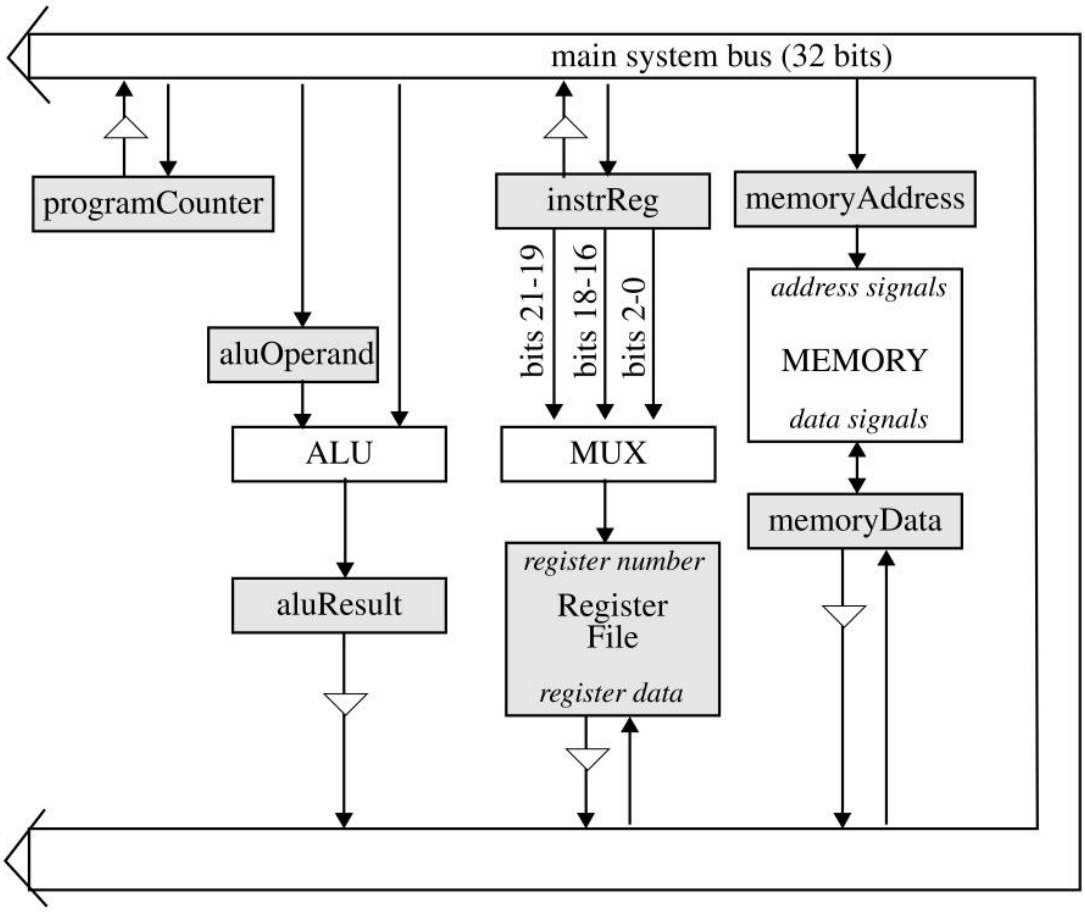
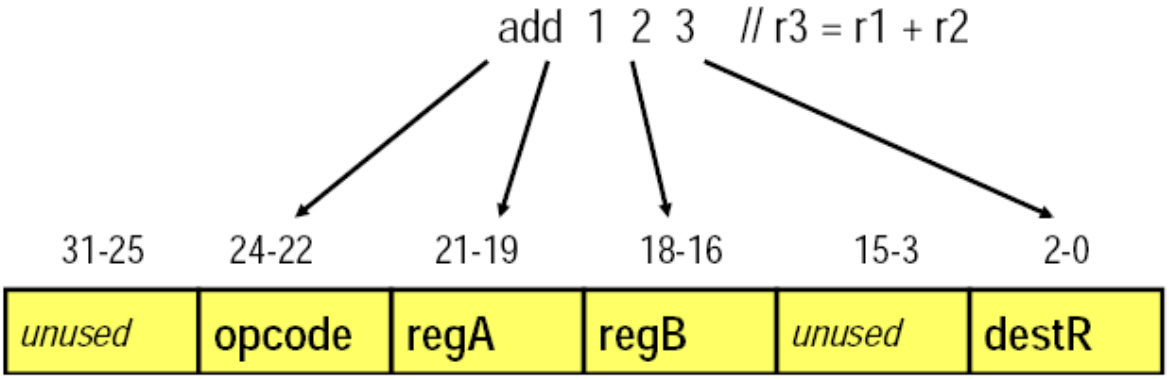


Рис 1. Функціональна схема СК.

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.

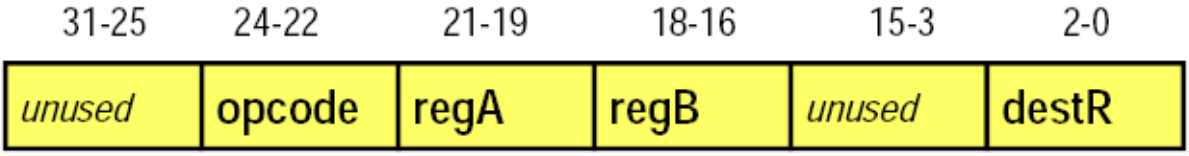
У СК є 8 регістрів по 32 розряди, пам'ять складається з 65536 слів по 32 розряди. Одже СК є 32 розрядним комп'ютером. Він підтримує 8 інструкцій, кожна з яких розписана нижче. У СК є спеціальний регістр лічільник команд (ЛК).

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

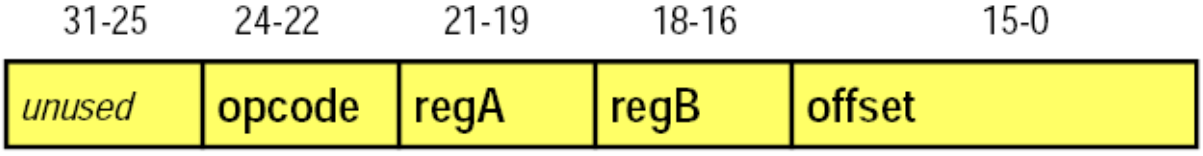


СК підтримує 4 формати інструкцій. Біти 31-25 не використовує жодна інструкція тому вони завжди мають дорівнювати 0.

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



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)



J-тип інструкцій (jalr):
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)

Повний перелік множини інструкцій надано таблицею

Таблиця 1. Множина інструкцій

№ пп	Код інструкції	2кове	СУТНІСТЬ ІНСТРУКЦІЙ МАШИНИ
Інструкції R-типу			
1	add	000	Додає вміст регістру regA до вмісту regB, та зберігає в destReg
2	nand	001	Виконує логічне побітове І-НЕ вмісту regA з вмістом regB, та зберігає в destReg
I-тип			
3	lw	010	Завантажує regB з пам'яті. Адреса пам'яті формується додаванням зміщення до вмісту regA.
4	sw	011	Зберігає вміст регістру regB в пам'ять. Адреса пам'яті формується додаванням зміщення до вмісту regA.
5	beq	100	Якщо вміст регістрів regA та regB однаковий, виконується перехід на адресу програмний лічильник(ПЛ) + 1+зміщення, в ПЛ зберігається адреса поточної тобто beq інструкції.
J-тип			
6	jalr	101	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.
O-тип			
7	halt	110	Збільшує значення ПЛ на 1, потім припиняє виконання, стимулятор має повідомляти, що виконано зупинку.
8	noop	111	Нічого не виконується

Асемблерна мова та асемблер.

В першій частині даного курсового проекту необхідно написати програму, яка перетворює вхідну програму на мові асемблер в мову машинних кодів. Програма має перетворити асемблерні імена команд в числові еквіваленти, наприклад асемблерну команду beq в 100, також перетворити символьні імена адрес в числові значення. Результуючий файл має складатися з послідовності 32 бітних інструкцій (біти 31-25 інструкції завжди рівні 0).

Формат лінійки асемблерного коду наступний (<пробіл> означає послідовність табуляцій і/або пробілів):

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

Крайнє ліве поле лінійки асемблерного коду – поле мітки. Коректна мітка має складатися максимуму з 6 символів, символами можуть бути літери або цифри, але починатися з букви. Поле мітки є необов'язковим, проте пробіл після даного поля є обов'язковим. Мітки дозволяють значно спростити процес написання асемблер них програм, в іншому випадку прийшлось би модифікувати всі адресні частини кожен раз коли додавався рядок коду!

Після не обов'язкової мітки іде обов'язковий пробіл. Далі іде поле назви інструкції, в якому може бути ім'я будь якої асемблерної інструкції зазначені вище в таблиці. Після пробілів ідуть відповідні поля. Всі поля можуть зберігати або десяткові значення або мітки. Кількість полів залежить від інструкції, поля які не використовуються ігноруються (подібно до коментарів).

Інструкції r-типу (add, nand) потребують наявності 3 полів: поле№1 – regA, поле№2 regB поле№3 destReg.

Інструкції i-типу (lw,sw,beq) вимагають 3 полів: поле№1 – regA, поле№2 regB поле№3 – числове значення зміщення чи символьна адреса. Числове значення може бути як додатнім так і відємним. Символьні адреси описані нижче.

Інструкція J-типу (jalr) вимагає 2 полів: поле№1 – regA, поле№2 regB

Інструкція 0-типу (noop, halt) не вимагає жодного.

Символьні адреси посилаються на відповідні мітки. Для інструкцій lw та sw асемблер має згенерувати зміщення, яке дорівнює адресі мітки. Вона може використовуватися з 0 регістром, тоді буде посилання на мітку, або може використовуватися з не нульовим базовим регістром у якості індексу масиву, який починається з мітки. Для інструкції beq, асемблер має перетворити мітку в числове зміщення куди має відбуватися перехід.

Після останнього поля має йти пробіл за яким може розміщуватися коментар. Коментар закінчується з кінцем лінії асемблерної програми. Коментарі дуже важливі для отримання зрозумілої асемблерної програми, тому що інструкції самі по собі мало зрозумілі.

Крім інструкцій СК, асемблерна програма може містити директиви для асемблера. В даному курсовому проекті для асемблера використовується лише одна директива - .fill (зверніть увагу на точку попереду). Директива . fill повідомляє компілятору про те, що він має зберегти число за адресою відповідно де дана інструкція знаходиться. Директива .fill використовує одне поле, в якому може бути як число так і символьна адреса. Наприклад «.fill 32» означає зберегти число 32 за адресою де дана інструкція знаходиться. (Оскільки в нас кожен рядок програми відповідає адресі починаючи з 0, то відповідно адреса буде дорівнювати номеру рядка - 1). Директива . fill з символьною адресою збереже адресу даної мітки. В прикладі нижче ".fill start" збереже значення 2, тому що мітка start знаходиться за адресою 2.

Асемблер має виконувати два проходи через асемблерну програму. На першому проході, асемблер має вирахувати адреси кожної символьної мітки. Виходячи з того, що перша інструкція знаходить по нульовій адресі. На другому проході, асемблер має генерувати машинні інструкції (у вигляді десткових чисел) для кожного рядку асемблерної мови. Нижче подано приклад асемблерної програми, яка рахує в зворотньому порядку від 5 до 0.

```
        lw      0       1       five    завантажити в reg1 5  (символьна адреса)
        lw      1       2       3       завантажити в reg2  -1 (числова адреса)
start   add     1       2       1       зменшити  reg1 на 1
        beq     0       1       2       перейти на кінець програми коли reg1==0
        beq     0       0       start   перейти на початок циклу
        noop
        done    halt                    кінець програми
five    .fill   5
neg1    .fill   -1
stAddr  .fill   start               буде зберігати адресу мітки start (2)
```

Відповідний машинний код:

```
(адреса 0):     8454151         (hex 0x810007)
(адреса 1):     9043971         (hex 0x8a0003)
(адреса 2):     655361          (hex 0xa0001)
(адреса 3):     16842754        (hex 0x1010002)
(адреса 4):     16842749        (hex 0x100fffd)
```

(адреса 5): 29360128 (hex 0x1c00000)
(адреса 6): 25165824 (hex 0x1800000)
(адреса 7): 5 (hex 0x5)
(адреса 8): -1 (hex 0xffffffff)
(address 9): 2 (hex 0x2)

Оскільки програма завжди починається з 0, результуючий файл має складатися лише зі значень без адрес.

8454151
9043971
655361
16842754
16842749
29360128
25165824
5
-1
2

Зауважимо, що програмі імена файлів мають передаватися у якості аргументів командного рядка. Асемблер має зберігати в результуючому файлі лише машинні команди у вигляді десяткових чисел, одну команду в одному рядку. Порушення даного формату призведе до того, що вихідний файл не можна буде виконати. Інший вивід (наприклад для відладки) програма може виконувати у консоль.

Асемблер має визначати наступні помилки в асемблер ній програмі: використання не визначених міток, використання однакових міток, використання змішення яке перевищує 16 біт, не визначені команди. Асемблер має повертати 1, якщо він визначив помилку та 0 у випадку успішного виходу з програми. Асемблер не має визначати помилки виконання програми, тобто помилки які виникають під час виконання програми (наприклад безмежні цикли, чи перехід на адресу -1 і т.д.).

Одною з частин виконання даного курсової роботи створити можину тестів для того щоб протестувати роботу асемблера. Створення наботу тестів є розповсюдженою практикою при розробці ПЗ. Це дозволяє впевнитися в правильності виконання програми при її модифікаціях. Створення всебічних тестів дозволить глибше зрозуміти специфіку проекту та вашої програми, та допоможе налагодити програму.

Множиною тестів буде набір невеличких асемблер них програм в якості вхідних еталонів. При захисті курсової роботи необхідно продемонструвати не менше як 20 тестів, кожен з яких складається не менше ніж з 50 рядків.

Важливо створити тести для перевірки можливості асемблера визначити помилки в коді.

Примітка. Оскільки змішення складається лише з 16 біт, воно може бути в межах від -32768 до 32767. Для символічних адрес асемблер сам згенерує відповідну числову адресу.

Поведінкова симуляція.

Другою частино даної курсової роботи є створення програми, яка може від симулювати роботу любого вірного машинного коду СК. Вхідним має бути файл з машинним кодом програми, якій має створити асемблер. Наприклад, якщо назва програми

simulate та машинний код зберігається в файлі *program.mc,* програма має запускатися наступним чином:

simulate program.mc > output

При цьому весь вивід буде виконуватися в файл "output".

Симулятор має розпочинати роботи з ініціалізації вмісту всіх регістрів та ПЛ 0. Симулятор має виконувати програму доти не зустріне команду halt.

Симулятор має виводити вивід стану комп'ютера перед виконанням кожної інструкції та один раз перед виходом з програми. Вивід стану має включати вивід вмісту всіх регістрів, ПЛ, пам'яті. Пам'ять має виводитися лише для комірок визначених в файлі з машинними кодами (наприклад у наведеному вище прикладі це адреси від 0 до 9).

Так само як і для асемблера, необхідно написати набір тестів і для перевірки роботи симулятора СК.

Набір тестів для симулятра є простою задачею адже вже буде набір тестів для асемблера. Отже необхідно дише відібрати коректні програми, в якості вхідних тестових файлів для симулятора. При захисті необхідно буде представити набір тестів і для симулятора. Кожен тест має виконуватися не менше як на 200 інструкціях, набір тестів має складатися не менше ніж з 20 тестових прикладів.

Зауваження. Будьте уважними при роботі з зміщенням для lw, sw та beq. Пам'ятайте, що зміщення складається з 2 байт – 16 біт. При використанні 32 біт типів даних необхідно перетворити від'ємне 16 бітове число у від'ємне 32 бітове. Для цього можна використати наступну функцію.

```
int convertNum(int num)
{
  /* перетворення16-бітового числа в  32-бітовий integer */
  if (num & (1<<15) ) {
     num -= (1<<16);
  }
  return(num);
}
```

Асемблерне множення.

Останньою частиною курсової роботи буде створення асемблерної програми для множення двох чисел. Вхідними будуть числа, які мають зчитуватися з пам'яті розташованій в комірках з назвами "mcand" та "mplier". Результат має бутив першому регістрі при завершені виконання. Вхідні числа мають бути 15 бітовими додатними.

Програма множення має бути ефективною, тобто не має перевищувати 50 рядків та виконання не має виходити за 1000 інструкцій для будь-яких вхідних даних. Для цього можна використати цикли та зміщення, алгоритми послідовного сумування є занадто довними.

Вихідні дані на проектування.

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

№	Розрядність шини даних	Розмір пам’яті Байт	Розмір регістрового файлу(к-сть регістрів)
1	16	256	8
2	24	4096	8
3	32	65536	16
4	40	1048576	32
5	48	16777216	64
6	56	33554432	128
7	64	67108864	256

2. 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

№	**Мнемонічнй код**	**Зміст**
1	DEC regA	Зменшити regA на 1
2	INC regA	Збільшити на 1
3	DIV regA regB destReg	Беззнакове ділення destReg=regA/regB
4	IDIV regA regB destReg	Знакове ділення destReg=regA/regB
5	IMUL regA regB destReg	Знакове множення destReg=regA*regB
6	XADD regA regB destReg	Додати і обміняти операнди місцями destReg=regA+regB regA<=>regB
7	SUB regA regB destReg	Віднімання : destReg=regA-regB
8	XDIV regA regB destReg	Беззнакове ділення і обмін операндів місцями destReg=regA/regB
9	XIDIV regA regB destReg	Знакове ділення і обмін операндів місцями destReg=regA/regB
10	XIMUL regA regB destReg	Знакове множення і обмін операндів місцями destReg=regA*regB
11	XSUB regA regB destReg	Віднімання і обмін операндів місцями: destReg=regA-regB

Логічні

№	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
3	SHL regA regB destReg	Логічний зсув вліво destReg=regA << regB
4	SHR regA regB destReg	Логічний зсув вправо destReg=regA >> regB
5	CMPE regA regB destReg	Порівняти regA regB destReg= regA == regB
6	SAL regA regB destReg	Арифметичний зсув вліво destReg=regA << regB
7	SAR regA regB destReg	Арифметичний зсув вправо destReg=regA >> regB
8	ROR regA regB destReg	Циклічний зсув вліво destReg=regA << regB
9	ROL regA regB destReg	Циклічний зсув вправо destReg=regA << regB
10	OR regA regB destReg	Логічне побітове АБО destReg=regA | regB
11	NOT regA destReg	Логічне побітове НЕ destReg= ~regA
12	NEG regA destReg	Заміна знаку на протилежний
13	MOV regA destReg	Переміщення даних destReg= regA
14	CMPNE regA regB destReg	Порівняти regA regB destReg= regA != regB
15	CMPL regA regB destReg	Порівняти regA regB destReg= regA < regB
16	CMPG regA regB destReg	Порівняти regA regB destReg= regA > regB
17	CMPGE regA regB destReg	Порівняти regA regB destReg= regA >= regB
18	CMPLE regA regB destReg	Порівняти regA regB destReg= regA <= regB

Керування. Умовні переходи.

№	Мнемонічнй код	Зміст
1	JMA regA regB offSet	Беззнакове більше if (regA> regB) PC=PC+1+offSet
2	JMAE regA regB offSet	Беззнакове більше/рівно if (regA>= regB) PC=PC+1+offSet
3	JMB regA regB offSet	Беззнакове менше if (regA< regB) PC=PC+1+offSet
4	JMBE regA regB offSet	Беззнакове менше/рівно if (regA<= regB) PC=PC+1+offSet
5	JMG regA regB offSet	Знакове більше if (regA > regB) PC=PC+1+offSet
6	JMGE regA regB offSet	Знакове більше/рівно if (regA >= regB) PC=PC+1+offSet
7	JML regA regB offSet	Знакове менше if (regA< regB) PC=PC+1+offSet
8	JMLE regA regB offSet	Знакове менше/рівно if (regA<= regB) PC=PC+1+offSet
9	JMNA regA regB offSet	Беззнакове не більше if (regA!> regB) PC=PC+1+offSet
10	JMNAE regA regB offSet	Беззнакове не більше/рівно if (regA!>= regB) PC=PC+1+offSet
11	JMNB regA regB offSet	Беззнакове не менше if (regA!< regB) PC=PC+1+offSet
12	JMNBE regA regB offSet	Беззнакове не менше/рівно if (regA!<= regB) PC=PC+1+offSet
13	JMNE regA regB offSet	Не рівно if (regA!= regB) PC=PC+1+offSet
14	JMNG regA regB offSet	Знакове не більше if (regA !> regB) PC=PC+1+offSet
15	JMNGE regA regB offSet	Знакове не більше/рівно if (regA !>= regB) PC=PC+1+offSet
16	JMNL regA regB offSet	Знакове не менше if (regA!< regB) PC=PC+1+offSet
17	JMNLE regA regB offSet	Знакове не менше/рівно if (regA!<= regB) PC=PC+1+offSet

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

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

№	Адресція
1	Безадресна – реалізація стеку. Максимальна глибина 32 слова по 32 розряди.
2	Безпосередня
3	Непряма
4	Непряма регістрова
5	Пряма
6	Індексна (розробити IR – індексний регістр), передбачити команду чи директиву встановлення регістру, регістр має інкрементуватися/декрементуватися після кожного звернення *
7	Відносна (розробити BR – базовий регістр) передбачити команду чи директиву встановлення регістру*
8	Базово – індексна (розробити IR та BR – базовий регістр) передбачити команду чи директиву встановлення базового регістру, індексний регістр має інкрементуватися/декрементуватися після кожного звернення *

Примітка:

** узгодити з викладачем*

Безадресні команди.

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

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

Регістр переносу (CF)

<table>
<tr><th>№</th><th>Мнемонічний код</th><th>Зміст</th></tr>
<tr><td>1</td><td>ADC regA regB destReg</td><td>Додавання з переносом: destReg=regA+regB+CF</td></tr>
<tr><td>2</td><td>SBB regA regB destReg</td><td>Віднімання з переносом: destReg=regA-regB-CF</td></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><td>SF</td><td>ZF</td></tr>
<tr><td>regA < regB</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>regA = regB</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>regA > regB</td><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>5</td><td>STC</td><td>Встановити CF =1</td></tr>
<tr><td>6</td><td>RCL regA regB destReg</td><td>Зсунути циклічно вліво через CF destReg=regA << regB</td></tr>
<tr><td>7</td><td>RCR regA regB destReg</td><td>Зсунути циклічно вправо через CF destReg=regA << regB</td></tr>
<tr><td>8</td><td>CLC</td><td>Зкинути параорець CF=0</td></tr>
</table>

Регістр знаку SF

<table>
<tr><th>№</th><th>Мнемонічний код</th><th>Зміст</th></tr>
<tr><td>1</td><td>CMP regA regB</td><td>Порівняти regA regB і встановити прапорці
<table>
<tr><td></td><td>CF</td><td>SF</td><td>ZF</td></tr>
<tr><td>regA < regB</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>regA = regB</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>regA > regB</td><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>2</td><td>JL offSet</td><td>Перейти, якщо менше, if(SF==1)PC=offset</td></tr>
<tr><td>3</td><td>JGE offSet</td><td>Перейти, якщо більше чи рівно, if(SF==0)PC=offset</td></tr>
</table>

Регістр ознаки нуля ZF

<table>
<tr><th>№</th><th>Мнемонічний код</th><th>Зміст</th></tr>
<tr><td>1</td><td>CMP regA regB</td><td>Порівняти regA regB і встановити прапорці
<table>
<tr><td></td><td>CF</td><td>SF</td><td>ZF</td></tr>
<tr><td>regA < regB</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>regA = regB</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>regA > regB</td><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>2</td><td>BSR regA destReg</td><td>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</td></tr>
<tr><td>3</td><td>BSF regA destReg</td><td>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</td></tr>
<tr><td>4</td><td>JE offSet</td><td>Перейти, якщо менше, if(ZF==0)PC=offset</td></tr>
<tr><td>5</td><td>JNE offSet</td><td>Перейти, якщо більше чи рівно, if(ZF!=0)PC=offset</td></tr>
</table>

Кожен варіант складається з наступних завдань:

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

Варіанти:

№	Розрядність	Арифметичні			Логічні			Керування		Прапорці				Адресація
		1	2	3	4	5	6	7	8		1	2	3	
1	1	1	3	5	1	2	3	1	3	CF	1	2	3	1
2	2	2	4	5	1	2	4	1	4	CF	1	2	4	2
3	3	3	5	6	1	3	5	1	5	CF	1	2	5	3
4	4	3	7	8	2	4	6	1	6	CF	1	2	6	4
5	5	11	9	6	2	5	7	1	7	CF	1	2	7	5
6	6	2	6	7	2	6	9	1	8	CF	1	2	8	6
7	7	1	8	10	3	8	10	1	11	SF	1	2	3	7
8	1	3	7	11	3	9	11	1	12	ZF	1	2	3	8
9	2	4	1	6	3	10	12	1	13	ZF	1	2	4	1
10	3	3	10	11	4	9	11	1	16	ZF	1	2	5	2
11	4	3	6	10	4	8	10	1	17	CF	2	3	4	3
12	5	1	4	10	4	10	13	2	3	CF	2	3	5	4
13	6	2	5	6	5	6	11	2	4	CF	2	3	6	5
14	7	1	5	7	5	12	14	2	7	CF	2	3	8	6
15	1	3	6	9	1	2	15	2	8	CF	2	3	7	7
16	2	2	7	11	1	2	16	2	9	ZF	2	3	4	8
17	3	3	5	9	1	2	17	2	10	ZF	2	3	5	1
18	4	2	3	11	1	2	18	2	13	CF	3	4	5	2
19	5	1	3	10	2	3	13	2	15	CF	3	4	6	3
20	6	3	5	7	2	3	14	2	17	CF	3	4	7	4
21	7	6	7	10	2	3	15	3	5	CF	3	4	8	5
22	1	2	6	7	2	3	16	3	6	ZF	3	4	5	6
23	2	2	9	10	2	3	17	3	11	CF	4	5	6	7
24	3	2	9	11	2	3	18	3	13	CF	4	5	7	8
25	4	2	10	11	2	4	5	3	16	CF	4	5	8	1
26	5	1	7	10	2	4	18	3	17	CF	5	6	7	2
27	6	1	6	9	2	4	17	4	5	CF	5	6	8	3
28	7	1	6	10	2	4	16	4	6	CF	6	7	8	4
29	1	3	7	10	2	4	15	4	7	SF	1	2	3	5
30	2	3	7	9	2	4	14	5	13	ZF	1	3	5	6

Вимоги щодо оформлення матеріалів проекту

Результат курсового проектування викласти у формі пояснювальної записки та креслення деталізованої структурної схеми комп'ютера.

Пояснювальна записка повинна містити:

- титульну сторінку;
- анотацію;
- зміст;
- конкретизовані та розширені вихідні дані на проектування;
- аналітичний розділ з роз'ясненням та аналізом основних принципів побудови комп'ютерів на прикладі визначених на реалізацію інструкцій;
- алгоритми роботи розробленого емулятора та асемблера
- опис виконання кожного типу розроблених інструкцій в по тактовому режимі
- функціональна схема комп'ютера до модифікації
- функціональна схема після модифікації, з визначеними сигналами керування
- опис розроблених форматів команд
- основні результати роботи (висновок);
- перелік наукових першоджерел: монографій, статей, патентів і підручників.
- вихідні коди та результати роботи тестових програм в додатках

Повний обсяг записки знаходиться в межах від 10 до 20 сторінок машинопису. Форма представлення записки – ***Word документ.*** Стиль оформлення запозичити з даних методичних вказівок. Оформлення креслення повинно задовольняти вимоги відповідних стандартів України.

Бажано електронне представлення до захисту зазначених документів у формі, що визначена вимогами системи збереження електронних документів кафедри ЕОМ. **Проекти, документи яких не відповідатимуть стандартам, до захисту не приймаються.**

Захист матеріалів курсового проектування відбувається на комісії в терміни, що відповідають складанню семестрового диференційованого заліку.

ЛІТЕРАТУРА

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.

Додаток 1. Приклад асемблера.

```
      #include <stdlib.h>
      #include <stdio.h>
      #include <string.h>

      #define MAXLINELENGTH 1000

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

      int
      main(int argc, char *argv[])
      {
          char *inFileString, *outFileString;
          FILE *inFilePtr, *outFilePtr;
          char label[MAXLINELENGTH], opcode[MAXLINELENGTH],
arg0[MAXLINELENGTH],
                  arg1[MAXLINELENGTH], arg2[MAXLINELENGTH];

          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);
          }

          /* here is an example for how to use readAndParse to read a line
from
              inFilePtr */
          if (! readAndParse(inFilePtr, label, opcode, arg0, arg1, arg2) ) {
              /* reached end of file */
          }

          /* this is how to rewind the file ptr so that you start reading from
the
              beginning of the file */
          rewind(inFilePtr);

          /* after doing a readAndParse, you may want to do the following to
test the
              opcode */
          if (!strcmp(opcode, "add")) {
              /* do whatever you need to do for opcode "add" */
          }

          return(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 (by looking for a \n) */
          if (strchr(line, '\n') == NULL) {
              /* line too long */
            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 ]%[^\t\n ]%*[\t\n ]%[^\t\n ]%*[\t\n ]%[^\t\n
]%*[\t\n ]%[^\t\n ]",
              opcode, arg0, arg1, arg2);
          return(1);
      }

      int
      isNumber(char *string)
      {
          /* return 1 if string is a number */
          int i;
          return( (sscanf(string, "%d", &i)) == 1);
      }
```

Додаток 2. Приклад коду симулятора

```
/* EECS 370 LC-2K Instruction-level simulator */

#include <stdio.h>
#include <string.h>

#define NUMMEMORY 65536 /* maximum number of words in memory */
#define NUMREGS 8 /* number of machine registers */
#define MAXLINELENGTH 1000

typedef struct stateStruct {
    int pc;
    int mem[NUMMEMORY];
    int reg[NUMREGS];
    int numMemory;
} stateType;

void printState(stateType *);

int
main(int argc, char *argv[])
{
    char line[MAXLINELENGTH];
    stateType state;
    FILE *filePtr;

    if (argc != 2) {
      printf("error: usage: %s <machine-code file>\n", argv[0]);
      exit(1);
    }

    filePtr = fopen(argv[1], "r");
    if (filePtr == NULL) {
      printf("error: can't open file %s", argv[1]);
      perror("fopen");
      exit(1);
    }

    /* read in the entire machine-code file into memory */
    for (state.numMemory = 0; fgets(line, MAXLINELENGTH, filePtr) !=
NULL;
      state.numMemory++) {
      if (sscanf(line, "%d", state.mem+state.numMemory) != 1) {
          printf("error in reading address %d\n", state.numMemory);
          exit(1);
      }
      printf("memory[%d]=%d\n", state.numMemory,
state.mem[state.numMemory]);
    }

    return(0);
}

void
printState(stateType *statePtr)
{
    int i;
    printf("\n@@@\nstate:\n");
    printf("\tpc %d\n", statePtr->pc);
    printf("\tmemory:\n");
      for (i=0; i<statePtr->numMemory; i++) {
          printf("\t\tmem[ %d ] %d\n", i, statePtr->mem[i]);
```

```
    }
  printf("\tregisters:\n");
    for (i=0; i<NUMREGS; i++) {
        printf("\t\treg[ %d ] %d\n", i, statePtr->reg[i]);
    }
  printf("end state\n");
}
```

Додаток 3. Приклад виконання симулятора

```
memory[0]=8454151
memory[1]=9043971
memory[2]=655361
memory[3]=16842754
memory[4]=16842749
memory[5]=29360128
memory[6]=25165824
memory[7]=5
memory[8]=-1
memory[9]=2

@@@
state:
      pc 0
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      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
end state

@@@
state:
      pc 1
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 5
            reg[ 2 ] 0
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 2
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 5
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 3
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 4
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 4
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
```

```
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 4
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 2
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 4
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 3
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 3
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 4
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 3
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 2
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 3
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 3
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
```

```
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 2
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 4
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 2
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 2
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 2
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 3
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 1
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 4
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 1
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 2
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 1
```

```
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 3
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 0
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state

@@@
state:
      pc 6
      memory:
            mem[ 0 ] 8454151
            mem[ 1 ] 9043971
            mem[ 2 ] 655361
            mem[ 3 ] 16842754
            mem[ 4 ] 16842749
            mem[ 5 ] 29360128
            mem[ 6 ] 25165824
            mem[ 7 ] 5
            mem[ 8 ] -1
            mem[ 9 ] 2
      registers:
            reg[ 0 ] 0
            reg[ 1 ] 0
            reg[ 2 ] -1
            reg[ 3 ] 0
            reg[ 4 ] 0
            reg[ 5 ] 0
            reg[ 6 ] 0
            reg[ 7 ] 0
end state
machine halted
total of 17 instructions
executed
      final state of machine:

      @@@
      state:
            pc 7
            memory:
                  mem[ 0 ] 8454151
                  mem[ 1 ] 9043971
                  mem[ 2 ] 655361
                  mem[ 3 ] 16842754
                  mem[ 4 ] 16842749
                  mem[ 5 ] 29360128
                  mem[ 6 ] 25165824
                  mem[ 7 ] 5
                  mem[ 8 ] -1
                  mem[ 9 ] 2
            registers:
                  reg[ 0 ] 0
                  reg[ 1 ] 0
                  reg[ 2 ] -1
                  reg[ 3 ] 0
                  reg[ 4 ] 0
                  reg[ 5 ] 0
                  reg[ 6 ] 0
                  reg[ 7 ] 0
      end state
```