

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"



КАФЕДРА ЕЛЕКТРОННИХ ОБЧИСЛЮВАЛЬНИХ МАШИН

МЕТОДИЧНІ ВКАЗІВКИ

до курсової роботи на тему

ПРОЄКТУВАННЯ КОМП'ЮТЕРА

з дисципліни

"Архітектура комп'ютерів"

для студентів спеціальності F7 "Комп'ютерна інженерія"

Львів 2025

ЗМІСТ

- Вступ
- Архітектурні принципи
- Система команд
- Способи адресації
- Система машинних інструкцій СК
- Асемблерна мова та асемблер
- Поведінкова симуляція
- Асемблерне множення
- Вихідні дані на проєктування
- Вимоги щодо оформлення матеріалів проєкту
- Література
- Додатки

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 – комп'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 частині розробляється поведінковий стимулятор результуючого машинного коду. В третій частині розробляється невеличка програма на асемблерній 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:

Принципи архітектури:









1.  Інформація кодується в двійковому представленні;
2.  Інформація в комп'ютері ділиться на команди і дані;
3.  Різноманітні за змістом слова розрізняються за способом застосування, а не по способу кодування;
4.  Слова інформації розміщуються в комітках 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 необхідно:







1.  забезпечити вибірку команди програми із його пам'яті в заданій послідовності, організувати звернення до неї за відповідними адресами;
2.  Розпізнати тип виконуваної операції;
3.  Організувати звернення до пам'яті за відповідними адресами для вибірки необхідних для виконання кожної команди даних;
4.  Організувати виконання над даними операцій відповідно до вказівок команд;
5.  Зберегти результати обчислень.

Комп'ютер виконує кожну команду як послідовність простих операцій.




Етапи виконання кожної команди:


1.  Вибірка команди із пам'яті;
2.  Визначення типу вибраної команди, тобто її дешифрування;
3.  Визначення адрес потрібних даних;
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 для запису результату;
7.  Запис результату;
8.  Підготовка до наступної команди;







Особливості CISC-процесора:

1.  виконання команди за багато тактів, оскільки для цього потрібно здійснити багаторазові операції звернення до основної пам'яті та до програмно-доступних регістрів процесора;
2.  орієнтація АЛП на виконання великої кількості операцій, що пов'язано з розширеним складом системи команд;
3.  складна система розпізнавання команди, що пов'язано з великою кількістю методів адресації та великою кількістю форматів команд різної розрядності;
4.  Програмне дешифрування для економії апаратури;
5.  складна організація конвеєризації виконання команд, що пов'язано, в першу чергу, з різноманітністю їх виконання;
6.  орієнтація структури на виконання команд типу регістр-пам'ять та пам'ять-пам'ять.





Основні елементи процесора:

-  Арифметико-логічний пристрій (АЛП)
-  Пристрій керування
-  Регістрова пам'ять (надоперативний ЗП)

 До регістрової пам'яті належать:

-  Програмний лічильник (ПЛ)
-  Регістр адреси (РГА)
-  Регістр команд (РГК)
-  Регістр даних (РГД)
-  Регістр слова стану програми (ССП)
-  Регістровий файл (загальні регістри)






Пояснення призначення:


-  РГА — зберігає адресу даних або команди.
-  РГК — зберігає команду після її зчитування.
-  РГД — зберігає операнд.
-  ПЛ — підраховує команди та зберігає адресу поточної.


Більшість комп'ютерів мають в складі процесора тригери для зберігання бітів стану процесора, або, як їх ще називають, прапорців. Кожен прапорець має спеціальне призначення. Частина прапорців

вказує на результати арифметичних і логічних операцій:


Прапорці (тригери стану):

-  P — додатний результат
-  N — від'ємний результат
-  Z — нульовий результат
-  C — перенос
-  V — переповнення


 Також є прапорці для режимів захисту пам'яті, пріоритетів задач, умовних переходів тощо. Разом вони формують ССП — слово стану програми.


 **Регістровий файл (РЗП)** складається з регістрів загального призначення. Вони використовуються для:

- зберігання даних і проміжних результатів,
- адресної арифметики,
- передачі аргументів та результатів між командами.




 **Підсумок:** CISC-процесори забезпечують гнучке та потужне середовище виконання завдяки багатій системі команд та глибокій організації процесів обробки. Ці архітектурні принципи формують фундамент сучасних комп'ютерних систем.

Система команд

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





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


 **Система команд характеризується трьома аспектами:**

1.  формат,
2.  способи адресації,
3.  система операцій.

Формат команди



Формат включає: довжину команди, кількість, розмір, положення, призначення та спосіб кодування полів. Команди містять такі поля:


1.  тип операції (КОП)
2.  місце першого операнду (A1)
3.  місце другого операнду (A2)
4.  місце запису результату (A3)

 Кожному полю відповідає частина двійкового слова. Реальна система команд має команди різних форматів, тип яких визначається КОП.


Структура команди

Команда зберігається у двійковій формі та складається з:

-  коду операції (КОП),
-  адресної частини (адреси операндів та результату).




 КОП займає k розрядів, що дозволяє закодувати до 2^k операцій. Якщо процесор підтримує N_c операцій, мінімальне k визначається як $k = \lceil \log N_c \rceil$ (округлення вгору).

Адресна частина займає m розрядів. Кожна адреса — m_i біт, де $i = 1, 2, \dots, l$ (l — кількість адресних полів).

 Повна довжина команди: $k + m$ — повинна відповідати розміру даних або бути кратною йому (наприклад: 8, 16, 32 біти).

Форми представлення команд

Окрім двійкової, використовуються:


-  вісімкова (3 біти = 1 цифра)
-  шістнадцяткова (4 біти = 1 цифра)
-  мнемонічна (символьна)

Приклад перетворення:

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



Мнемонічне кодування

- Кожна команда — 3-4 літерний символ (наприклад: **ADD**, **SUB**, **MPY**, **DIV**).
- Операнди — теж символічні (наприклад: **ADD R Y**).

 Операція виконується над **вмістом**, а не над адресами.



Таким чином, можна писати програми у **символічній формі** — на **асемблерній мові**.

Асемблер

-  Перетворює мнемонічний код у машинний.
-  Команди асемблера вказують, як інтерпретувати назви, розмістити програму, скільки пам'яті потрібно.

Високорівневі мови програмування

Асемблерна мова складна, тому з'явилися:

-  мови високого рівня (наближені до людської мови)
-  компілятори (перетворюють програми в машинний код)

✅ Переваги мов високого рівня:

1. 🗣️ Легше писати (ближчі до мови спілкування)
2. 🕒 Швидше розробляти
3. 💻 Незалежні від типу комп'ютера

Це дозволяє переносити програми між різними машинами і звільняє програміста від знань архітектури.

Способи адресації

📌 Визначення: 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.

📌 Основні способи адресації операндів:

- ◆ Пряма – адресне поле зберігає адресу операнда. 📌 Пряма регістрова адресація адресує не комірку пам'яті, а номер регістру.
- ◆ Безпосередня – в полі адреси зберігається сам операнд, а не його адреса.
- ◆ Непряма – адресне поле містить адресу, за якою знаходиться адреса операнда. 📌 Непряма-регістрова адресація — адреса зберігається в регістрі загального призначення.
- ◆ Відносна – адреса формується як сума бази (в регістрі) та зміщення (у полі команди). 📌 Індексна адресація — базу замінює індексний регістр, який автоінкрементується. 📌 Базова-індексна — сума бази, індексу та зміщення.
- ◆ Безадресна – команда не має адресного поля. 📌 Операнд або відсутній, або визначений за замовчуванням (наприклад, акумулятор). 📌 Один із варіантів — використання стеку.

🔧 Примітка: Практично у всіх існуючих комп'ютерах використовується один або декілька з цих способів адресації. Тому в команду вводяться спеціальні ознаки, щоб пристрій керування міг розпізнати спосіб адресації — через:

- 🧩 додаткові розряди в команді;
- 📌 закріплення способів адресації за типами команд.

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.

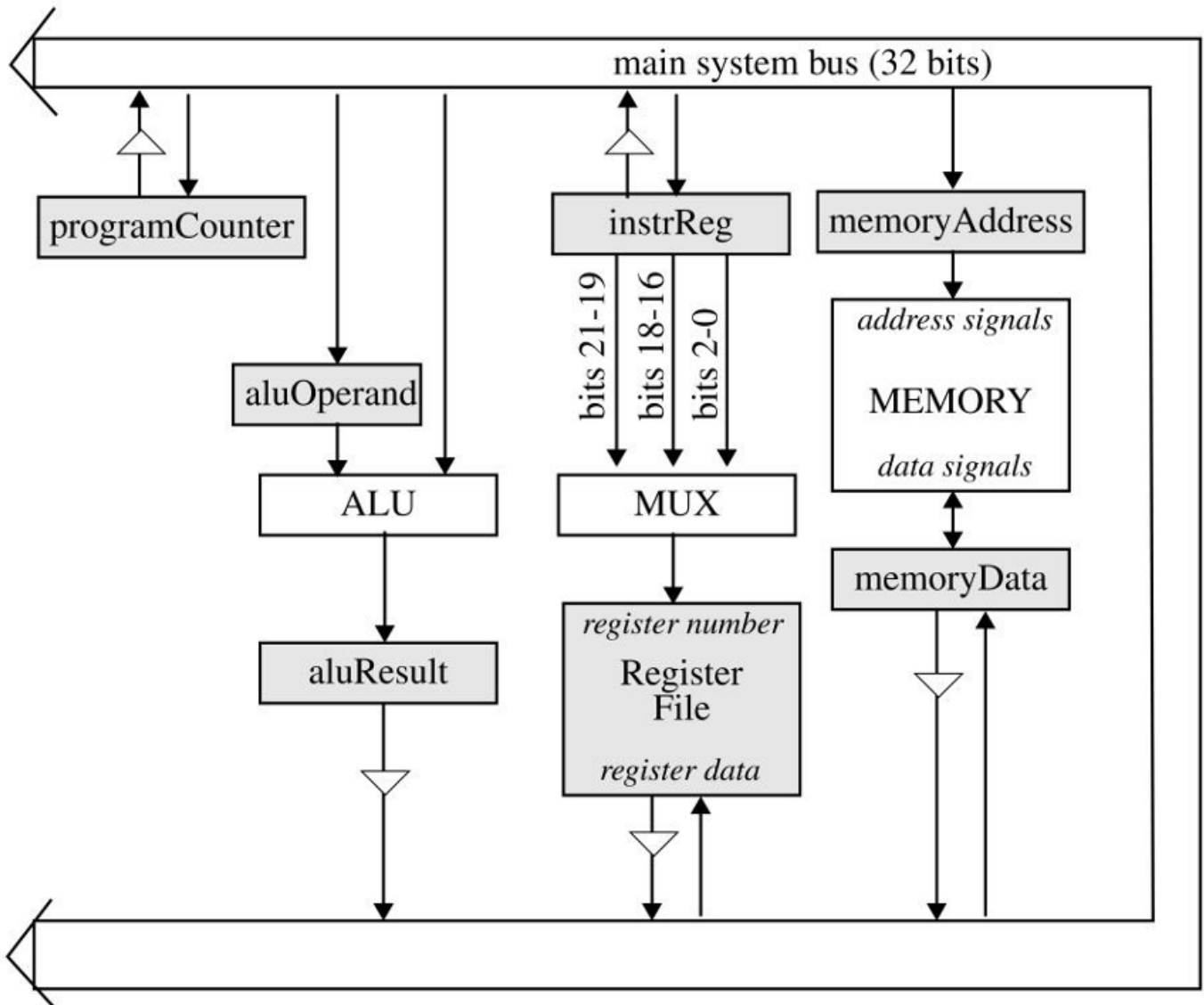


Рис 1. Функціональна схема СК.

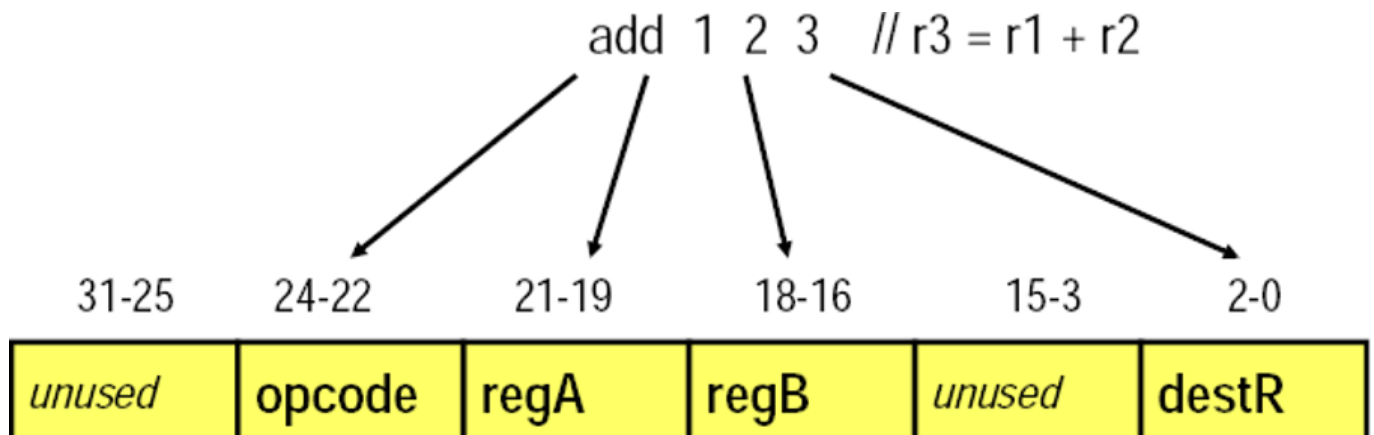
В спрощеному комп'ютері (СК) в пам'яті зберігаються, як дані так і інструкції.

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 регістровій 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 лічильник команд (ЛК).

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



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

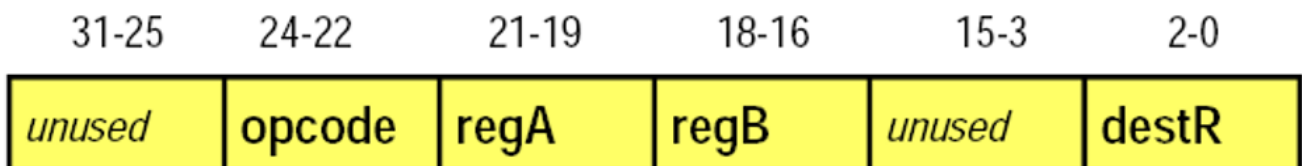
Інструкції R-типу (add nand):

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

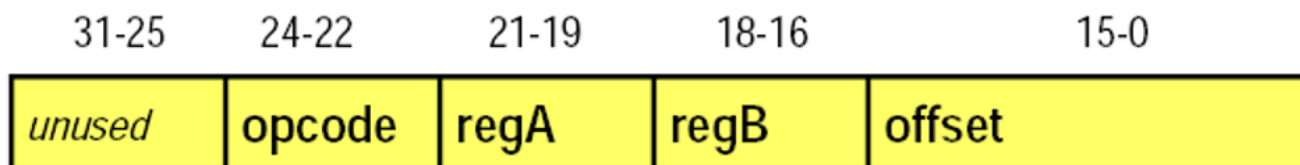


I-тип інструкцій (lw, sw, beq):

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)

0-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	2ко ве	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.

№ пп	Код інструкції	2ко ве	СУТНІСТЬ ІНСТРУКЦІЙ МАШИНИ
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 символні 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 лїве поле лїнійки асемблерного коду – поле мїтки. Коректна мїтка має складатися максимуму з 6 символів, символами можуть бути лїтери або цифри, але починатися з букви. Поле мїтки є необов'язковим, проте пробїл пїсля даного поля є обов'язковим. Мїтки дозволяють значно спростити процес написання асемблерних програм, в іншому випадку прийшлося бї модифїкувати всї адресні частини кожен раз коли додавався рядок коду!

Пїсля не обов'язкової мїтки їде обов'язковий пробїл. Далї їде поле назви їнструкції, в якому може бути їм'я будь якої асемблерної їнструкції зазначенї вище в таблицї. Пїсля пробїлів їдуть відповідні поля. Всї поля можуть зберїгати або десятковї значення або мїтки. Кїлькїсть полів залежить від їнструкції, поля якї не використовуються їгноруються (подїбно до коментарїв).

Їнструкції г-типу (add, and) потребуєть наявностї 3 полів: поле№1 – regA, поле№2 regB, поле№3 destReg.

Їнструкції ї-типу (lw, sw, beq) вимагають 3 полів: поле№1 – regA, поле№2 regB, поле№3 – числове значення змїщення чи символїна адреса. Числове значення може бути як додатнїм так ї вїдеднїм. Символїні адреси описанї нижче.

Їнструкція J-типу (jalr) вимагає 2 полів: поле№1 – regA, поле№2 regB. Їнструкція O-типу (noor, halt) не вимагає жодного.

Символїні адреси посилаються на відповідні мїтки. Для їнструкцій lw та sw асемблер має згенерувати змїщення, яке дорївнює адресї мїтки. Вона може використовуватися з 0 реїстром, тодї буде посилення на мїтку, або може використовуватися з не нульовим базовим реїстром у якостї їндексу масиву, який починається з мїтки. Для їнструкції beq, асемблер має перетворити мїтку в числове змїщення куди має вїдбуватися перехїд.

Пїсля останнього поля має їти пробїл за яким може розмїщуватися коментар.

Коментар закінчується з кїнцем лїнії асемблерної програми. Коментарї дуже важливї для отримання зрозумїлої асемблерної програми, тому що їнструкції самї по собї мало зрозумїлї.

Крім їнструкцій CK, асемблерна програма може мїстити директиви для асемблера. В даному курсовому проєктї для асемблера використовується лише одна директива - .fill (звернїть увагу на точку попереду). Директива .fill повїдомляє компїлятору про те, що вїн має зберегти число за адресою вїдповїдно де дана їнструкція знаходиться. Директива .fill використовує одне поле, в якому може бути як число так ї символїна адреса. Наприклад

«.fill 32» означає зберегти число 32 за адресою де дана їнструкція знаходиться. (Оскїльки в нас кожен рядок програми вїдповїдає адресї починаючи з 0, то вїдповїдно адреса буде дорївнювати номеру рядка - 1). Директива .fill з символїною адресою збереже адресу даної мїтки. В прикладї нижче ".fill start" збереже значення 2, тому що мїтка start знаходиться за адресою 2.

Асемблер має виконувати два проходи через асемблерну програму. На першому проходї, асемблер має вирахувати адреси кожної символїної мїтки. Виходячи з того, що перша їнструкція знаходить по нульовїй адресї. На другому проходї, асемблер має генерувати машиннї їнструкції (у виглядї десткових чисел) для кожного рядку асемблерної мови. Нижче подано приклад асемблерної програми, яка рахує в зворотньому порядку від 5 до 0.

Приклад асемблерної програми: Обчислення **Fibonacci(10)**

Файл: [input.as](#)

```
-----
# Fibonacci(10) -----
----
lw    0 1 ten      # R1 ← 10      ← лічильник N
lw    0 2 zero     # R2 ← 0      ← F(n-1) (a)
lw    0 3 one      # R3 ← 1      ← F(n) (b)
lw    0 4 neg1     # R4 ← -1     ← для декременту

loop   beq  1 0 done  # якщо N == 0 → результат уже в R2
       add  2 3 5     # R5 = R2 + R3 (F(n+1))
       add  3 0 2     # R2 = R3      (зсув a ← b)
       add  5 0 3     # R3 = R5      (зсув b ← c)
       add  1 4 1     # N = N - 1
       beq  0 0 loop  # безумовний перехід

done   sw    0 2 result # зберегти F(10)=55 у пам'ять
       halt              # завершити симуляцію

# --- дані -----
--
ten    .fill 10
zero   .fill 0
one    .fill 1
neg1   .fill -1
result .fill 0          # тут з'явиться 55
```

Ця програма реалізує ітеративне обчислення 10-го числа Фібоначчі. Результат буде збережено у пам'яті за міткою [result](#).

 Завантажити: [input.as](#)

Запуск асемблювання

```
python3 assemble.py # input.as → output.mc
```

Згенерований машинний код ([output.mc](#))

```
8454156    # lw 0 1 ten      → завантажити 10 у R1
8519693    # lw 0 2 zero     → завантажити 0 у R2
8585230    # lw 0 3 one      → завантажити 1 у R3
8650767    # lw 0 4 neg1     → завантажити -1 у R4
17301509   # beq 1 0 done    → якщо R1 == 0 → перехід до done
1245189    # add 2 3 5       → R5 = R2 + R3
1572866    # add 3 0 2       → R2 = R3
2621443    # add 5 0 3       → R3 = R5
786433     # add 1 4 1       → R1 = R1 - 1
16842746   # beq 0 0 loop    → безумовний перехід до loop
```

```

12714000    # sw 0 2 result    → зберегти R2 у result
25165824    # halt            → завершити виконання

# --- Дані ---
10          # ten      .fill 10
0           # zero     .fill 0
1           # one      .fill 1
-1          # neg1     .fill -1
0           # result   .fill 0 (тут з'явиться результат – 55)


```

Асемблер має визначати наступні помилки в асемблерній програмі: використання не визначених міток, використання однакових міток, використання зміщення 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 створитиможину 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 асемблерних 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 зміщення 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 частиною даної курсової роботи є створення програми, яка може відсимулювати роботу довільного машинного коду СК. Вхідним має бути файл з машинним кодом програми, якій має створити асемблер.

Запуск симулятора

```
python3 simulate.py # output.mc -> result.txt
```

Згенерований результат (**result.txt**)

```

- pc:11          # Лічильник команд зупинився на інструкції halt (адреса
11)
- r0:0           # Регістр 0 – завжди 0
- r1:0           # R1 – лічильник ітерацій, зменшений до 0

```

```

- r2:55          # R2 – результат F(10)
- r3:89          # R3 – F(11), обчислений на останньому кроці
- r4:4294967295 # R4 – значення -1 у 32-бітовому доповняльному коді
- r5:89          # R5 – тимчасове збереження F(n+1)
- r6:0           # Не використовувався
- r7:0           # Не використовувався

- machine halted          # Машина завершила виконання
- instructions executed: 66 # Виконано 66 інструкцій (10 обертів +
ініціалізація + завершення)

- --- memory state ---          # Стан пам'яті:
- mem[0] = 8454156             # lw 0 1 ten      → завантаження 10 у R1
- mem[1] = 8519693             # lw 0 2 zero     → завантаження 0 у R2
- mem[2] = 8585230             # lw 0 3 one      → завантаження 1 у R3
- mem[3] = 8650767             # lw 0 4 neg1     → завантаження -1 у R4
- mem[4] = 17301509            # beq 1 0 done   → перевірка завершення
- mem[5] = 1245189             # add 2 3 5     → F(n+1)
- mem[6] = 1572866             # add 3 0 2     → зсув R3 → R2
- mem[7] = 2621443             # add 5 0 3     → зсув R5 → R3
- mem[8] = 786433              # add 1 4 1     → декремент лічильника
- mem[9] = 16842746            # beq 0 0 loop  → безумовний перехід
- mem[10] = 12714000           # sw 0 2 result → запис результату
- mem[11] = 25165824           # halt          → завершення програми

- mem[12] = 10                 # .fill 10 (мітка ten)
- mem[14] = 1                   # .fill 1 (мітка one)
- mem[15] = 4294967295         # .fill -1 (мітка neg1, у двійковому представленні)
- mem[16] = 55                 # .fill result (тут збережено F(10) = 55)

```

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 зустрине 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 симулятора є простішою задачею, оскільки вже буде набір тестів для асемблера. Отже, необхідно лише 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.

 **Примітка.** 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.

```
def sext16(x: int) -> int:
```

```
    """
```

Знакове розширення 16-бітового цілого до повного 32-бітового int у Python.

У машинному коді LC-2K деякі інструкції (наприклад, lw, sw, beq) містять

16-бітове поле зміщення (offset). Але Python працює з 32-бітовими (і більше)

числами, тому від'ємні значення треба правильно інтерпретувати.

Ця функція виконує перетворення:

- 0x0000 ... 0x7FFF → 0 ... 32767 (невід'ємні значення)
- 0x8000 ... 0xFFFF → -32768 ... -1 (від'ємні значення у доповняльному

коді)

Алгоритм:

- (x & 0x7FFF) – виділяє 15 молодших біт (число без знаку)
- (x & 0x8000) – ізолює 16-й біт (знаковий біт)
- Якщо біт знаку встановлено (тобто $x \geq 0x8000$), то віднімається

0x8000

і результат буде від'ємним значенням у Python.

Приклади:

```
sext16(0x000A) → 10
sext16(0x7FFF) → 32767
sext16(0x8000) → -32768
sext16(0xFFFF) → -1
sext16(0xFFFD) → -3
```

Цей прийом – поширений метод "sign extension" у низькорівневому програмуванні.

```
    """
```

```
    return (x & 0x7FFF) - (x & 0x8000)
```

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 "msand" та "mplier". Результат має бути в першому регістрі при завершенні 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 довгими.

Вихідні дані на проектування

1. Визначити формати команд згідно розрядності шини даних, розміру пам'яті та реєстрового файлу.

№	Розрядність шини даних	Розмір пам'яті Байт	Розмір реєстрового файлу(к-сть реєстрів)
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

№	Мнемонічний 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

№	Мнемонічний 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
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.

№	Мнемонічний 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

№	Мнемонічний код	Зміст
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 додаткових 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:

* за потреби узгодити з викладачем

Безадресні команди.

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

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

Регістр переносу (CF)

№	Мнемонічний код	Зміст
1	ADC regA regB destReg	Додавання з переносом: destReg=regA+regB+CF
2	SBB regA regB destReg	Віднімання з переносом: destReg=regA-regB-CF
3	BT regA regB	Копіює значення біта з regA по позиції regB в CF = regA[regB]
4	CMP regA regB	Порівняти regA regB і встановити прапорці
5	STC	Встановити CF =1
6	RCL regA regB destReg	Зсунути циклічно вліво через CF destReg=regA << regB
7	RCR regA regB destReg	Зсунути циклічно вправо через CF destReg=regA >> regB
8	CLC	Зкинути прапорець CF=0

Регістр знаку SF

№	Мнемонічний код	Зміст		
1	CMP regA regB	Порівняти regA regB і встановити прапорці		
		CF	SF	ZF
	regA	1	1	0
	< regB			
	regA	0	0	1
	= regB			

	0	0	0
regA			
> regB			

2	JL offSet	Перейти, якщо менше, if(SF==1)PC=offset
3	JGE offSet	Перейти, якщо більше чи рівно, if(SF==0)PC=offset

Регістр ознаки нуля ZF

№	Мнемонічний код	Зміст
1	CMP regA regB	Порівняти regA regB і встановити прапорці

Кожен варіант складається з наступних завдань:

- 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:

№	Розряд ність	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.

За потреби, власний розроблений варіант роботи можна організувати як "fork" від [основного репозиторію](#) та створити пулл ріквест.

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;
- функціональну схему комп'ютера до модифікації;
- функціональну схему після модифікації з визначеними сигналами керування;
- тести на негативні та позитивні сценарії;
- деталізований опис змін, внесених у код;
- опис реалізації додаткового способу адресації
(важливо: додатковий спосіб адресації має бути розширенням системи, а не заміною існуючого виду адресації);
- опис розроблених форматів команд;
- опис змін внесених у код;
- основні результати роботи (висновок);
- перелік наукових першоджерел: монографій, статей, патентів і підручників;
- вихідні коди та результати роботи тестових програм у додатках.

ЛІТЕРАТУРА

 Основна

1. Іванов В.В., Коваль С.В.
Мікропроцесорні системи та мікроконтролери. — Київ: Ліра-К, 2021.
 2. Морозов А.Н.
Архітектура та програмування мікропроцесорів. — Київ: КНУ ім. Шевченка, 2020.
 3. Mazidi M.A., Mazidi J.G., McKinlay R.D.
The 8051 Microcontroller and Embedded Systems. — Pearson Education, 2006.
 4. Barry B. Brey
The Intel Microprocessors: Architecture, Programming, and Interfacing. — Prentice Hall, 2008.
 5. Фурман В.М.
Програмування мікроконтролерів AVR мовою асемблера. — Львів: Новий Світ – 2000, 2015.
 6. Мельник А. О.
Архітектура комп'ютера. Наукове видання. — Луцьк: Волинська обласна друкарня, 2008. — 470 с.
-

Додаткова

7. Patterson D.A., Hennessy J.L.
Computer Organization and Design: The Hardware/Software Interface. — Morgan Kaufmann, 2021.
 8. Стогній Б.С. та ін.
Комп'ютерна електроніка та мікропроцесори. — НТУУ «КПІ», 2019.
 9. Пронін В.О.
Цифрова схемотехніка та мікропроцесорна техніка. — Харків: НТУ «ХПІ», 2020.
-

Тематичні джерела (CISC, RISC, адресація)

- [Що означають RISC та CISC? \(переклад статті\)](#)
 - [Вікіпедія: Способи адресації пам'яті](#)
 - [Вікіпедія: Цикл виконання інструкцій](#)
-

Додатки

Код асемблера

 Завантажити: [assemble.py](#)

Код симулятора

 Завантажити: [simulate.py](#)