tiny-lemon
для чайників

автор Поет Лучник

ЗМІСТ

ПЕРЕДМОВА

Для кого ця книга?

Ця книга призначена для новачків у програмуванні котрі, не інакше як дивом, натрапили на tiny-lemon та хочуть дізнатись про це більше. У будь якому випадку ця книга може бути використана ким завгодно у незалежності від рівня знань та переслідуваних цілей.

Нащо мені все це?

tiny-lemon створювався у першу чергу як відносно просте середовище для навчання програмуванню знизу-уверх, тобто щоб самостійно пройти шлях від програмування машинним кодом до створення своєї власної мови програмування. У другу чергу як середовище для програмування в обмежених умовах, тобто перевірка своїх здібностей до оптимізації програм з розумним використанням усіх доступних можливостей. У будь якому випадку tiny-lemon не обмежується цими двома, бо це повноцінна розширювана система, котра може бути використана для будь чого.

Як користуватися книгою?

Читати книгу треба по порядку, від початку і до кінця. Але, передбачена можливість ігнорувати окремі абзаци. Тобто, якщо тема абзацу вам вже відома то при бажанні його можна проігнорувати. Також у книзі можна зустріти підкреслення у місцях де передбачається самостійне ознайомлення читача із зовнішніми джерелами інформації, без яких досвід буде не повний.

Наступні розділи

Вступ – розділ містить визначення загальних термінів.

Центральний процесор – у розділі розповідається про те як влаштований процесор tiny-lemon у відриві від реалізації самої статичної бібліотеки.

Статична бібліотека – розділ про те як користуватись tiny-lemon, також розділ містить перелік структур даних та функцій доступних користувачу.

Приклади програм – розділ, де можна знайти програми з поясненнями того, як вони працюють.

ВСТУП

tiny-lemon – це статична бібліотека яка написана на мові програмування С та є реалізацією абстрактної віртуальної машини.

Віртуальна машина (скорочено ВМ) – це програма, котра повністю або частково відтворює поведінку деякого обчислювального пристрою. Віртуальну машину називають абстрактною якщо згаданий обчислюваний пристрій не має фізичної реалізації.

Мова програмування С (вимовляється як «Сі») – як і будь яка інша мова, це набір слів та символів, котрі використовують для передачі сенсу, тільки на відмінну від людських мов сенс передається не від людини до людини, а від людини до комп'ютера. У мові С існують правила написання текстів, ці правила називаються – синтаксис. Якщо текст написаний синтаксично правильно то його можна вважити повноцінним кодом. Але це лише текст, комп'ютеру потрібна програма щоб прочитати його, і така програма називається компілятор.

Компілятор – це програма котра на вхід отримує синтаксично правильно написаний код, а на виході створює виконуваний файл. Це основна функція компілятора, але вона може бути не єдиною функцією, у залежності від конкретної реалізації, компілятор може отримувати на вхід додаткові данні, і також може на виході давати щось окрім виконуваного файлу.

У світі існує декілька компіляторів для мови програмування С та у бідь який момент можуть з'явитися нові, тому у цій книжці не буде пояснень як користуватись кожним з існуючих компіляторів. Читачу пропонується ознайомитися з тим чи іншим екземпляром самостійно, шляхом читання офіційної документації.

Не зважаючи на вже сказане, у книзі можна знайти кілька прикладів використання компілятора GCC. Передбачається, що цього буде достатньо для початку роботи зі статичною бібліотекою.

Статична бібліотека – це файл або декілька файлів, які містять машинні інструкції, а також деякі дані. Це схоже на опис звичайної програми, але статична бібліотека це не програма, це незалежний модуль, який може бути вставлений у програму для подальшого використання його функціоналу. Файли бібліотеки мають розширення .lib на операційній системі Windows та – .a на ОС Linux.

Написана на С статична бібліотека – включає у собі файли як самої бібліотеки, так і хочаб один файл заголовку (розширення .h). Файли заголовку містять код написаний на С який є переліком доступних структур даних та функцій. Насамперед файли заголовку потрібні компілятору.

Робота компілятора ділиться на етапи. Спочатку йде попередня обробка тексту (препроцесор). Потім генерація проміжного результату який представляє собою набір файлів об'єктиників (розширення .obj або .o), такі файли зберігають машинний код з додатковими даними. Наприкінці відбувається лінковка – з'єднання всіх об'єктників в один виконуваний файл, а також приєднання файлів бібліотек.

ЦЕНТРАЛЬНИЙ ПРОЦЕСОР

Процесор – це пристрій, який виконує інструкції. Кожен процесор має кінцевий набір інструкцій, де кожна інструкція має точне визначення.

Центральний процесор - це процесор, який є головною частиною комп'ютера.

Процесор не може існувати у «вакуумі», йому обов'язково потрібно щось, звідки можна отримувати інструкції. І найчастішим рішенням цієї проблеми є спеціальна пам’ять, де зберігаються інструкції, які є програмою.

Саме tiny-lemon є комбінацією процесора і пам'яті.

Оперативна пам’ять

Оперативна пам’ять - це пам’ять, яка призначена для запису, зберігання та читання інформації під час роботи процесору. Ця пам’ять може містити інструкції та дані. У tiny-lemon пам’ять становить 256 байт.

Байт – це 8 біт. Біт це одиниця вимірювання інформації, яка є однією цифрою двійкової системи чисел, тобто 0 або 1. Перебираючи всі можливі комбінації з восьми біт, можна дійти висновку, що байт може зберігати 256 різних значень. Зазвичай їх уявляють як числа від 0 до 255, або як числа від -128 до 127. Також байти зручно сприймати у 16-ій системі чисел, як числа від 0x00 до 0xFF.

У байтах можна зберігати будь-яку інформацію, будь-які дані. Комбінація байт може означати будь-що. Тому про значення байтів завжди говорять у контексті програми, алгоритму чи пристрою, який може інтерпретувати дані тим чи іншим способом. Наприклад, текстовий редактор інтерпретує байти як набір символів, а редактор зображень розглядає байти як інформацію про зображення. І ви можете відкрити файл зображення в текстовому редакторі, використовуючи дані від однієї програми в іншій програмі. Той самий файл можна відкрити через програму архіватор щоб створити файл архів. Щойно було сказано про різні файли, виникають вони через те, що існує програмна реалізація файлової системи, тобто програма, яка може інтерпретувати байти на диску як інформацію про файли та директорії.

Стек

Стек – структура даних, яка працює за принципом "останній прийшов, перший пішов". Існують можливості покласти або взяти з вершини стека значення. У tiny-lemon стек це окрема пам’ять, яка становить 256 байт. Але оскільки переривання переповнення буде викликано при спробі запису в останній байт, можна вважати що доступні тільки 255 байт.

Переривання

Переривання – це реакція процесора на певну подію і супроводжується передачею управління спеціальній підпрограмі обробнику переривань. Переривання може бути викликане у будь-який момент роботи процесора. Це дозволяє процесору миттєво реагувати на події. Коли обробник переривань закінчує роботу, він повертає виконання програми на те місце, на якому відбулося переривання. У tiny-lemon переривання може бути викликане як внутрішнім станом процесора, так і зовнішнім пристроєм, незалежно від типу переривання адрес підпрограми обробника знаходиться у пам'яті за адресом 255 (останній байт). Варто зазначити, що під час зовнішнього переривання пристрій посилає 1 байт, який потрапить у регістр A і це можна використовувати для визначення того який пристрій викликав переривання.

Регістри

Регістр – це окрема комірка пам'яті, з можливістю читання та запису значень, що використовується процесором для різноманітних інструкцій. У tiny-lemon вісім регістрів і кожен по вісім біт. Назви та значення регістрів знаходяться у таблиці «Регістри процесору tiny-lemon». Усі регістри напочатку мають значення 0.

Регістри процесору tiny-lemon

Ім'я	Повне ім'я	Значення
A	Акумулятор	Використовується як аргумент деяких інструкцій та для збереження результатів операцій.
D	Ді	Не має попередньо визначеного значення і може бути використаний для будь-яких цілей.
F	Прапори	Зберігає інформацію про результати операцій, а також використовується для керування перериваннями та умовними переходами. Кожен із восьми біт цього регістру має своє значення, яке можна подивитися в таблиці «Біти регістру F».
I	Індекс Введення	Індекс порту введення використовується операціями введення.
K	Кей	Не має попередньо визначеного значення і може бути використаний для будь-яких цілей.
O	Індекс Виведення	Індекс порту виведення використовується операціями виведення.
P	Програмний Лічильник	Адреса байту інструкції, яка буде виконана у наступний цикл процесору. Під час інструкції значення цього регістру автоматично збільшується на 1. Оскільки спочатку регістр має значення 0, процесор виконує програму починаючи з першого байту оперативної пам'яті.
S	Вказівник Стеку	Вказує на вершину стеку та використовується операціями роботи зі стеком. Регістр вказує саме на верхній не зайнятий байт у пам'яті стеку. Спочатку це адрес 0, адже на стек ще нічого не поклали. При спробі взяти зі стеку значення, коли регістр S дорівнює 0, буде викликано переривання через недоповнення стеку (англ. *Stack Underflow*). З іншого боку, при спробі щось покласти на стек, коли регістр S дорівнює 255 (максимальному значенню), виникне переривання, у зв'язку з переповненням стеку (англ. *Stack Overflow*).

Біти регістру F

Розряд	Ім’я	Повне ім’я	Значення
1	Z	Нуль	Значення цього біту автоматично змінюють деякі операції, якщо біт має значення 1, це вказує на те, що результат операції дорівнює нулю.
2	C	Перенос	Значення цього біту автоматично змінюють деякі операції, якщо біт має значення 1, це вказує на те, що результат операції вийшов за межі байту, з лівого або правого краю.
3	N	Від’ємний	Значення цього біту автоматично змінюють деякі операції, якщо біт має значення 1 це вказує на те, що результат операції був від’ємний або, що теж саме, результат був більший за 127.
4	O	Зміна Знаку	Значення цього біту автоматично змінюють деякі операції, якщо біт має значення 1, це вказує на те, що під час операції відбулася некоректна зміна знаку – сума двох позитивних чисел дала від’ємний результат або сума від’ємних дала позитивний результат.
5	I	Заборона Переривань	Змінюючи значення цього біту, можна керувати зовнішніми перериваннями. Поки біт дорівнює 1 процесор ігноруватиме сигнали переривань від зовнішніх пристроїв.
6	B	Перерва	Інструкції внутрішніх переривань автоматично змінюють значення цього біту на 1. Це дозволяє відрізнити такі переривання від інших.
7	S	Переривання Стеку	Значення біта змінюється на 1 під час переривань через переповнення або недоповнення стеку.
8	U	Невідомий	Не має певного значення і може бути використаний для будь-яких цілей.

Числові значення

Числа у байтах кодуються як звичайні двійкові, тобто ось це 0b00000000 означає 0, а ось це 0b01011010 означає 64 + 16 + 8 + 2 = 90, і ось це 0b11111111 число 255. Про те як переводити з однієї системи чисел у іншу пропонується дізнатись самостійно.

Будь-яку послідовність біт можна сприйняти як винятково позитивне число, або як число зі знаком. В tiny-lemon зміна знаку визначається як **-x = not(x) + 1**. Це зручно, бо будь-які операції додавання та віднімання котрі працюють правильно для чисел без знаку будуть працювати правильно і для чисел зі знаком. Наприклад, 0b00000001 + 0b11111111 = 0b00000000 у позитивних буде сприйматись як 1 + 255 = 0 (256, але вийшло переповнення), зі знаком це 1 + (-1) = 0 що також є правильним результатом. Це стосується тільки додавання та віднімання, для множення та ділення потрібно два різних алгоритму один для чисел зі знаком другий для виключно позитивних.

Робота процесору

Робота процесора ділиться на кроки. Крок – це виконання однієї інструкції, а також дії, які обов'язково виконуються перед і після інструкції.

Кожна інструкція в tiny-lemon ділиться на дві незалежні частини: режим адресації та операція. Це означає, що немає необхідності запам'ятовувати значення кожної інструкції, достатньо запам'ятати режими адресації та операції окремо та легко розуміти будь-яку комбінацію цих двох частин. Про це буде написано у наступних двох главах: «Режими адресації» та «Операції».

Один крок процесору tiny-lemon можна описати такою послідовністю дій:

1. Надіслати сигнал на спеціальний вихід – сповістити зовнішні пристрої про початок кроку.
2. Отримати інструкцію з оперативної пам'яті по адресу із регістру P.
3. Збільшити значення регістру P на 1.
4. Дізнатись по значенню інструкції режим адресації и виконати його.
5. Дізнатись по значенню інструкції операцію та виконати її.
6. Збільшити лічильник циклів на 1.

Важливо ще раз відзначити, що регістр P збільшується на 1 після отримання інструкції, але перед її інтерпретацією. Це означає, що на етапі режиму адресації регістр P матиме адрес наступного байту.

Під час будь-якого переривання процесор виконує таку послідовність дій:

1. Кладе на стек значення регістру P.
2. Кладе на стек регістри A, D, I, K, O, F у відповідній послідовності (що аналогічно до виконання операції PSR, але про це пізніше).
3. Читає значення байту за адресом 0xFF (останній байт пам'яті) і поміщає прочитане значення у регістр P. Тобто передає управління підпрограмі обробнику переривань.

Режими адресації

Режим адресації вказує процесору, звідки отримати адрес аргументу (далі просто, адрес) і значення аргументу (далі просто, аргумент). Адрес та аргумент використовуються різними операціями. Усі режими адресації наведено у таблиці «Режими адресації».

Режими адресації

Ім'я	Адрес	Аргумент	Додаткові дії
(немає)	З регістру P	З регістру A	
__A	З регістру A	Із байту по адресу	
_LA	Із байту по адресу з регістру A	Із байту по адресу	
__D	З регістру D	Із байту по адресу	
_LD	Із байту по адресу з регістру D	Із байту по адресу	
__K	З регістру K	Із байту по адресу	
_LK	Із байту по адресу з регістру K	Із байту по адресу	
__P	З регістру P	Із байту по адресу	Збільшити регістр P на 1
_LP	Із байту по адресу з регістру P	Із байту по адресу	Збільшити регістр P на 1

Безіменний режим адресації мається на увазі скрізь, де не вказаний жоден з восьми інших.

Не варто плутати збільшення регістра P на 1 те, що відбувається кожен крок з тим, що зазначено в додаткових діях режиму адресації. Перше вже відбулося до початку режиму адресації, а друге відбудеться після отримання адресу та аргументу.

Операції

Операція – це наказ процесору що зробити, які обчислення виконати, які маніпуляції з тією чи іншою пам'яттю зробити, як і які прапори регістру F оновити.

Умовні позначення у назвах та описах операцій:

1. Літера b у назві означає, що є дві групи цієї операції. У першій групі замість b підставлено F, а в другій – T. Можливо вам буде простіше зрозуміти так boolean, False, True. В описі операції буква b означає 0 або 1 відповідно.
2. Літера f у назві та описі означає, що замість неї можна підставити назву будь-якого з восьми бітів регістру прапорів F.
3. Літера r у назві та описі означає, що замість неї можна підставити назву будь-якого з восьми регістрів процесора.

Перелік операцій з описом (за абеткою):

ADD – (англ. *ADDition*), додає до регістру A значення аргументу (тобто це змінить значення регістру A), оновлює прапори Z і N виходячи з отриманого значення A. Прапор C стає 1, якщо сума виходить більше, ніж 255, в іншому випадку прапор C стане 0, тим самим зберігаючи потребу в перенесенні. Прапор O стане 1, якщо A і аргумент позитивні, а результат суми від'ємний, або навпаки. У інших випадках прапор O буде 0.

ADC – (англ. *ADDition with Carry*), діє аналогічно операції ADD, але додаючи в загальну суму регістру та аргументу також одиницю у випадку, якщо прапор C дорівнює 1.

AND – (англ. *bitwise AND*), обчислює побітове I для регістру A з аргументом, зберігаючи результат у регістрі A, оновлюючи прапори Z і N відповідно до результату.

BRK – (англ. *BReaK*), процесор здійснює переривання, прапор B при цьому ставатиме 1.

CMP – (англ. *CoMPare*), обчислює різницю між значеннями регістру A та аргументу, оновлює прапори Z, N, C, O аналогічно операції SUB, але не зберігає результат обчислення регістр A.

DCA – (англ. *DeCrement A register*), діє аналогічно операції SUB з аргументом 1, оновлюючи прапори Z, N, C, O відповідно.

DCD – (англ. *DeCrement D register*), зменшує значення регістру D на 1.

DCK – (англ. *DeCrement K register*), зменшує значення регістру K на 1.

DEC – (англ. *DeCrement byte*), зменшує значення байту за адресом на 1. Оновлює прапори аналогічно операції SUB, ніби від аргументу відняли 1, і результат зберегли у байт.

Fbf – (англ. *set Flag to b, name of flag is f*), прапор f прийме значення b.

HLT – (англ. *HaLT*), посилає сигнал щоб сповістити про зупинку, потім зменшує значення регістра P на 1, що створює нескінченний цикл виконання цієї операції. Тобто процесор зупиняється на поточній інструкції та інтерпретує її нескінченно.

ICA – (англ. *InCrement A register*), діє аналогічно операції ADD з аргументом 1, оновлюючи прапори відповідно.

ICD – (англ. *InCrement D register*), збільшує значення регістру D на 1.

ICK – (англ. *InCrement K register*), збільшує значення регістру K на 1.

INA – (англ. *INput register A*), посилає пристрою введення по індексу з регістра I сигнал, щоб пристрій відреагував і відправив процесору значення яке потім буде збережено в A.

INC – (англ. *INCrement byte*), збільшує значення байту за адресом на 1. Оновлює прапори аналогічно операції ADD, ніби це сума аргументу та одиниці, та зберігаючи результат у байт.

Jbf – (англ. *Jump if value b in flag f*), регістр P приймає значення адресу, але тільки якщо прапор f має значення b. Ака «Умовний перехід».

JMP – (англ. *JuMP*), регістр P приймає значення адресу. Ака «Безумовний перехід».

JSR – (англ. *Jump to Sub-Routine*), спочатку кладе значення регістру P на вершину стеку, потім регістр P прийме значення адресу. Ака «Виклик підпрограми».

LDr – (англ. *LoaD r register*), оновлює прапори Z і N виходячи з аргументу, а вже потім регістру r надається значення аргументу.

LUP – (англ. *LooP*), зменшує значення регістру A на 1, потім якщо A не дорівнює 0, регістр P прийме значення адресу.

MVx – (англ. *MoVe x register*), операція присвоєння, така що, регістр A приймає значення регістру x (замість x можна підставити назву будь-якого регістру крім A). Після присвоєння оновлюються прапори Z і N виходячи із значення, яке потрапило в A.

NEG – (англ. *NEGative byte*), змінює знак байту на протилежний. Оновлює прапори Z та N виходячи з результату і прапор O стане 1 у будь-якому випадку, крім випадку, коли значення байту 0, адже зміна знаку у нуля нічого не змінює.

NGA – (англ. *NeGative A register*), змінює знак регістру A на протилежний. Оновлює прапори Z та N виходячи з результату і прапор O стане 1 у будь-якому випадку, крім випадку, коли значення регістру це нуль, адже зміна знаку у нуля нічого не змінює.

NOP – (англ. *No OPeration*), ніякої операції.

NOR – (англ. *bitwise NOR*), обчислює побітове АБО для регістру A з аргументом, результат робить побітно протилежним і потім зберігає в A. Оновлює прапори Z і N виходячи зі значення потрапив у регістр A.

NOT – (англ. *bitwise NOT*), змінює значення байту по адресу побітно на протилежне (не плутати зі зміною знаку). Оновлює прапори Z та N виходячи з результату, у всіх випадках прапор O стане 1.

NTA – (англ. *bitwise NoT A register*), змінює значення регістру A побітно на протилежне (не плутати зі зміною знаку). Оновлює прапори Z та N виходячи з результату, у всіх випадках прапор O стане 1.

ORA – (англ. *bitwise OR*), обчислює побітове АБО для регістру A з аргументом, зберігаючи результат в A. Оновлює прапори Z і N виходячи з результату.

OUT – (англ. *OUTput A register*), відсилає значення аргументу разом із спеціальним сигналом на порт пристрою виведення за індексом з регістру O.

PIK – (англ. *PeeK to byte*), бере копію значення зі стеку та зберігає у байті по адресу.

PKr – (англ. *PeeK r register*), бере копію значення зі стеку та зберігає у регістр r.

POP – (англ. *POP to byte*), бере значення зі стеку та зберігає у байті по адресу.

PPr – (англ. *PoP r register*), бере значення зі стеку та зберігає у регістр r.

PPR – (англ. *PoP all Registers*), бере зі стеку 6 значень які зберігає в регістри F, O, K, I, D, A у відповідній послідовності. Серед перерахованих немає регістрів S та P, адже зміна першого призведе до втрати послідовності стеку, а зміна другого призведе до втрати послідовності інструкцій. Ця операція не може викликати переривань через помилку стеку.

PSH – (англ. *PuSH*), кладе значення аргументу на стек.

PSr – (англ. *PuSh r register*), кладе значення регістру r на стек.

PSR – (англ. *PuSH all Registers*), кладе на стек значення регістрів A, D, I, K, O, F у відповідній послідовності. Ця операція не може викликати переривань через помилку стека.

RTI – (англ. *ReTurn from Interrupt*), бере зі стеку 7 значень які зберігає у регістри F, O, K, I, D, A, P у відповідній послідовності. Тим самим поєднує в собі PPR і PPP, що дозволяє здійснити повернення із переривання. Ця операція не може викликати переривань через помилку стеку.

SBC – (англ. *SuBtraction with Carry*), діє аналогічно операції SUB, але віднімає від загальної суми одиницю у випадку, якщо прапор C дорівнює 1. Тим самим забезпечуючи коректне перенесення за правилами віднімання.

SHL – (англ. *bitwise SHift Left*), записує значення лівого біту регістру A у прапор C, зсуває всі біти регістру A на 1 біт вліво, оновлює прапори Z і N виходячи з результату операції.

SHR – (англ. *bitwise SHift Right*), записує значення правого біту регістру A у прапор C, зсуває всі біти регістру A на 1 біт вправо, оновлює прапори Z і N виходячи з результату операції.

SLC – (англ. *bitwise Shift Left with Carry*), запам'ятовує значення лівого біту регістра A, зсуває всі біти регістра A на 1 біт вліво, додає 1 до регістру A якщо прапор C дорівнює 1, оновлює прапори Z і N виходячи з результату операції. Записує у прапор C значення біту, яке раніше запам'ятав.

SRC – (англ. *bitwise Shift Right with Carry*), запам'ятовує значення правого біту регістру A, зсуває всі біти регістру A на 1 біт вправо, додає 128 до регістру A якщо прапор C дорівнює 1, оновлює прапори Z і N виходячи з результату операції. Записує у прапор C значення біту, яке раніше запам'ятав.

STA – (англ. *STore A register*), зберігає значення регістру A у байт по адресу.

SUB – (англ. *SUBtraction*), додає до регістру A від'ємне значення аргументу, тобто це віднімання. Оновлює прапори аналогічно до операції ADD. При чому прапор C буде оновлений правильно маючи на увазі віднімання – якщо від меншого відняли більше відбудеться перенесення, яке збережеться у прапорі C як 1 і навпаки прапор буде 0 у випадках, коли перенесення не відбулося.

TST – (англ. *TeST*), виконує операцію побітового І регістру A з аргументом, оновлює прапори Z і N, але не зберігає результат самої операції в регістр A.

XCB – (англ. *eXChange Byte*), міняє місцями значення регістру A та байту по адресу, потім оновлює прапори Z і N виходячи зі значення, що потрапило до A.

XCr – (англ. *eXChange r register*), міняє місцями значення регістру A та регістру r, потім оновлює прапори Z і N виходячи зі значення, що потрапили до A.

XOR – (англ. *bitwise eXclusive OR*), обчислює побітове виключне АБО для регістру A та аргументу, результат зберігається в A, оновлює прапори Z і N виходячи з отриманого значення.

ZRB – (англ. *ZeRo Byte*), байт по адресу стає нуль.

ZRr – (англ. *ZeRo r register*), флаг Z стає 1, флаг N стає 0, потім регістр r приймає значення 0.

Операції з числами розміром більше байту

Операція з числами розміром більше байту може бути реалізована через виконання операцій з кожним байтом числа.

Додавання за допомогою операцій ADD та ADC. Наприклад, у 16-ій системі чисел **0x0201 + 0x00FF = 0x0300** це сума байтів **0x01 + 0xFF** за допомогою ADD (або ADC з нульовим прапором C), у результаті **0x100**, але оскільки це виходить за межі байту результат буде обрізаний до **0x00**, а флаг C стане 1, тим самим вказуючи на необхідність переносу. Потім додаємо байти **0x02 + 0x00** саме через ADC щоб урахувати перенос від попередньої суми, у результаті **0x03**, и якщо об'єднати отримані однобайтні результати буде **0x0300** що є правильним результатом додавання.

Сказане працює і для віднімання цілих чисел, але вже з використанням SUB і SBC операцій.

Побітові зсуви виконуються схожим методом. Тільки важливо зрозуміти, як рухатися по байтах щоб врахувати перенесення. Щоб зробити зсув числа вліво, потрібно йти по байтах починаючи з самого правого і йдучи до лівого, на першому байті використовувати операцію SHL (або SLC з нульовим прапором C), а потім використовувати тільки SLC. А ось щоб зробити зсув в право потрібно починати з самого лівого байту і йти до правого, на першому байті використавши SHR (або SRC з нульовим прапором C), а далі тільки через SRC.

Для таких побітових операцій як AND, OR, NOR, XOR, NOT все іще простіше, можна розглядати число як масив байт і виконувати операцію для кожного байту окремо в будь-якому напрямку.

Інструкції

У таблиці нижче наведені всі можливі комбінації операцій з режимами адресацій, кожна з таких комбінацій це інструкція, кожна інструкція процесора tiny-lemon має своє числове значення, яке можна представити як дві 16-і цифри або один байт.

Порожня клітинка означає відсутність операції. Використання таких значень у програмах не рекомендується, оскільки вони можуть у майбутніх версіях набути нового сенсу. Якщо вам потрібна інструкція, яка нічого не робить, використовуйте 0x00 (NOP)

Інструкції та їх значення

	*0	*1	*2	*3	*4	*5	*6	*7	*8	*9	*A	*B	*C	*D	*E	*F
0*	NOP	MVD	MVF	MVI	MVK	MVO	MVP	MVS	LDA	LDD	LDF	LDI	LDK	LDO	LDP	LDS
1*	LDA __A	LDA _LA	LDA __D	LDA __LD	LDA __K	LDA _LK	LDA __P	LDA _LP	STA __A	STA _LA	STA __D	STA _LD	STA __K	STA _LK	STA __P	STA _LP
2*	XCA	XCD	XCF	XCI	XCK	XCO	XCP	XCS	XCB __A	XCB _LA	XCB __D	XCB _LD	XCB __K	XCB _LK	XCB __P	XCB _LP
3*	ZRA	ZRD	ZRF	ZRI	ZRK	ZRO	ZRP	ZRS	ZRB __A	ZRB _LA	ZRB __D	ZRB _LD	ZRB __K	ZRB _LK	ZRB __P	ZRB _LP
4*	FFZ	FFC	FFN	FFO	FFI	FFB	FFS	FFU	FTZ	FTC	FTN	FTO	FTI	FTB	FTS	FTU
5*	JFZ _LP	JFC _LP	JFN _LP	JFO _LP	JFI _LP	JFB _LP	JFS _LP	JFU _LP	JTZ _LP	JTC _LP	JTN _LP	JTO _LP	JTI _LP	JTB _LP	JTS _LP	JTU _LP
6*	JMP __A	JMP _LA	JMP __D	JMP _LD	JMP __K	JMP _LK	JMP __P	JMP _LP	JSR __A	JSR _LA	JSR __D	JSR _LD	JSR __K	JSR _LK	JSR __P	JSR _LP
7*	LUP __A	LUP _LA	LUP __D	LUP _LD	LUP __K	LUP _LK	LUP __P	LUP _LP	PSR	PPR	RTI	BRK	INA	OUT	HLT	
8*	PSA	PSD	PSF	PSI	PSK	PSO	PSP	PSS	PSH __A	PSH _LA	PSH __D	PSH _LD	PSH __K	PSH _LK	PSH __P	PSH _LP
9*	PPA	PPD	PPF	PPI	PPK	PPO	PPP	PPS	POP __A	POP _LA	POP __D	POP _LD	POP __K	POP _LK	POP __P	POP _LP
A*	PKA	PKD	PKF	PKI	PKK	PKO	PKP	PKS	PIK __A	PIK _LA	PIK __D	PIK _LD	PIK __K	PIK _LK	PIK __P	PIK _LP
B*	CMP __A	CMP _LA	CMP __D	CMP _LD	CMP __K	CMP _LK	CMP __P	CMP _LP	TST __A	TST _LA	TST __D	TST _LD	TST __K	TST _LK	TST __P	TST _LP
C*	ADD __P	ADC __P	SUB __P	SBC __P	AND __P	ORA __P	NOR __P	XOR __P	ADD _LP	ADC _LP	SUB _LP	SBC _LP	AND _LP	ORA _LP	NOR _LP	XOR _LP
D*	ICA	DCA	NTA	NGA	SHL	SLC	SHR	SRC	INC _LP	DEC _LP	NOT _LP	NEG _LP	ICD	DCD	ICK	DCK
E*																
F*																

Наприклад, інструкція AND__P це значення 0xC4 що у десятковому варіанті 196.

Будь-яка програма для tiny-lemon процесора складається з байтів, які містять значення інструкцій та будь-які інші дані. Щоб процесор виконав програму вона повинна бути завантажена в оперативну пам'ять tiny-lemon. Процесор інтерпретує байти пам'яті починаючи з першого (по адресу 0).

Нижче наведено приклад невеликої програми. Більше прикладів можна знайти у розділі «Приклади програм».

Приклад програми

Адрес	0x	Код	Пояснення
00	7C	INA	Приймає значення з порту введення за індексом 0, значення зберігається в регістрі A, зберігає це значення в байті за адресом 0x02, отримує ще одне значення з того ж порту, значення потрапить у регістр A, додає до регістру A значення байту за адресом 0x02, відправляє результат у порт виведення за індексом 0 та зупиняється на нескінченному виконанні інструкції HLT. Тобто додає два байти зі входу та відправляє результат на вихід.
01	1E	STA__P	
02	00	0x00	
03	7C	INA	
04	C8	ADD_LP	
05	02	0x02	
06	7D	OUT	
07	7E	HLT	

СТАТИЧНА БІБЛІОТЕКА

У цьому розділі буде розглянуто реалізацію процесора tiny-lemon у вигляді статичної бібліотеки з тією самою назвою tiny-lemon, налічує 3 файли «tiny-lemon.h», «tiny-lemon.a» та «tiny-lemon.lib».

Якщо у вас ОС Windows вам потрібен файл «tiny-lemon.lib», якщо ОС Linux або подібна, то вам потрібен файл «tiny-lemon.a». Незалежно від операційної системи вам потрібен файл "tiny-lemon.h". Тому у вас буде два файли: файл заголовку «.h» і файл бібліотеки «.lib» або «.a».

Файл заголовку зберігає трохи коду мовою програмування С, який потрібно розглянути. Увага: не редагуйте файл «tiny-lemon.h», внесення будь-яких змін призведе до помилок часу компіляції або помилок часу виконання.

Файл заголовку

Нижче показано та прокоментовано кожен рядок файлу «tiny-lemon,h».

```
#ifndef TINY_LEMON_HEADER_INCLUDED
```

Директива препроцесору ifndef (англ. *if not defined*), дозволяє розглядати текст із конструкції ifndef-endif, тільки у тому випадку коли макрос не визначений. Оскільки напочатку макросу з назвою TINY_LEMON_HEADER_INCLUDED нема, текст усередині ifndef-endif буде розглянутий, тобто директиви будуть проінтерпретувані, а код попаде до етапу компіляції.

```
#define TINY_LEMON_HEADER_INCLUDED
```

Директива препроцесора define, дозволяє визначати значення макросів. У цьому випадку визначає існування TINY_LEMON_HEADER_INCLUDED. Що в комбінації з попереднім рядком гарантує, що текст з конструкції ifndef-endif буде розглянутий компілятором лише один раз. Така комбінація часто використовуються коли файл вставляється (англ. *include*) кілька разів і при цьому немає сенсу компілювати його кілька разів або ж у випадках коли множина компіляція призводить до помилок.

```
#ifdef __cplusplus

extern "C" {

#endif
```

Код «extern “C” {» буде проігнорований директивою ifdef (англ. *if defined*) якщо макрос «__cplusplus» не визначений. Макрос __cplusplus як правило заздалегідь визначається C++ компіляторами. Це дозволяє відрізнити компіляцію С коду через С компілятор від компіляції через C++ компілятор. У даному випадку для C++ компілятора додається підказка що код далі має сенс розглядати як С код. У самому кінці буде аналогічна конструкція ifdef-endif для додавання фігурної дужки «}» що закриває блок.

```
#include<stdlib.h>

#include<stdint.h>
```

Директива препроцесору include, дозволяє додавати вміст одних файлів всередину інших. Це лише підстановка тексту, тому не слід вважати що це підключення бібліотеки. У цьому випадку додається текст із файлів «stdlib.h» і «stdint.h», це файли стандартної бібліотеки мови програмування С, з яких в tiny-lemon використовуються типи uint8_t і size_t.

```
#define TL_INTH_PTR_ADR (0xFF)
```

Визначається макрос TL_INTH_PTR_ADR із текстом «(0xFF)». Це адреса байту в пам’яті, де зберігається вказівник на підпрограму обробник переривань.

```
typedef uint8_t TLbyte;
```

Визначення типу TLbyte який аналогічний типу uint8_t (з файлу «stdint.h») та є цілим числом без знаку розміром у 8 біт, тобто 1 байт.

```
typedef size_t TLcounter;
```

Визначення типу TLcounter як size_t. Тип size_t є цілим числом без знаку яке може містити максимально можливий індекс.

```
typedef int TLbool;
```

Визначення типу TLbool як int. TLbool використовується як підказка програмісту, що отримане значення буде 0 або 1.

```
struct TLdata;
```

Декларація структури TLdata яка представляє всі необхідні для роботи віртуального процесора tiny-lemon дані. Це не визначення типу, а лише підказка компілятору, що цей тип буде визначений у майбутньому і не варто видавати помилку у разі його використання до його фактичного визначення.

```
typedef TLbyte(*TLin)   (struct TLdata* tl, TLbyte port);
```

Визначення типу TLin як вказівника на функцію, що приймає вказівник на структуру TLdata та TLbyte, з типом значення, що повертається – Tlbyte.

```
typedef void  (*TLout)  (struct TLdata* tl, TLbyte data, TLbyte port);
```

Визначення типу TLout як вказівника на функцію, що приймає вказівник на структуру TLdata та два TLbyte, з типом значення, що повертається – void.

```
typedef void  (*TLclock)(struct TLdata* tl);
```

Визначення типу TLclock як вказівника на функцію, що приймає вказівника на структуру TLdata, з типом значення, що повертається – void.

```
typedef void  (*TLhalt) (struct TLdata* tl);
```

Визначення типу TLhalt як вказівника на функцію, що приймає вказівника на структуру TLdata, з типом значення, що повертається – void.

```
typedef struct TLdata {
```

Визначення структури TLdata та призначення їй псевдоніма TL. TLdata – це складений тип, тобто структура даних що містить у собі під елементи. Ця структура призначена для зберігання всього стану процесора tiny-lemon та його пам’яті. Використовувати цей тип можна для оголошення як глобальної, так і локальної змінної у коді. Всі елементи визначені всередині фігурних дужок, після яких вказаний псевдонім структури, а саме TL.

```
    TLbyte ram[256];
```

Масив з 256 байт під назвою «ram» є оперативною пам’яттю процесору tiny-lemon. Цей масив, як і всі інші елементи структури, доступні для читання та зміни значень у будь-який момент.

```
    TLbyte stk[256];
```

Масив з 256 байт під назвою «stk» є пам’яттю стека процесору tiny-lemon.

```
    struct {
```

Визначення анонімної підструктури, що містить усі регістри процесору. Анонімної означає, що цей тип не має назви, але є визначення, після якого, як правило, йде використання. Після фігурної дужки, що закриває блок, зазначена назва змінної – reg.

```
        TLbyte A;
        TLbyte D;
        TLbyte F;
        TLbyte I;
        TLbyte K;
        TLbyte O;
        TLbyte P;
        TLbyte S;
```

Вісім регістрів процесору tiny-lemon і кожен це один байт.

```
    } reg;
```

Вищезгадана назва під елементу.

```
    struct {
```

Еще одна анонимная структура данных. Эта содержит указатели на функции обратного вызова.

Ще одна анонімна структура даних. Ця містить вказівники на функції зворотнього виклику.

```
        TLin in;
        TLout out;
        TLclock clock;
        TLhalt halt;
```

Об’явлені змінні in, out, clock, halt раніше визначеним типам TLin, TLout, TLclock, TLhalt відповідно. Віртуальний процесор tiny-lemon використовує ці вказівники на функції для реалізації подання різних сигналів.

```
    } callback;
```

Всі вказівники на функції об’єднуються під однією назвою – callback.

```
    TLcounter clock_counter;
```

Об’явлення лічильника циклів процесору.

```
    void* additional_data;
```

Оголошення вказівника на будь-що. Бібліотека tiny-lemon не використовує цей вказівник. Якщо прибрати цей рядок ви отримаєте помилку, що, можливо, здається вам контр інтуїтивним, але насправді немає нічого дивного, навіть незважаючи на те, що ця змінна не використовується, сама бібліотека маніпулює пам’яттю з урахуванням розміру структури даних у байтах, якщо зменшити розмір – бібліотека вийде за межі наданої вами пам’яті. Ви можете використовувати цей вказівник для «причеплення» додаткових даних, які потрібні для реалізації віртуальної машини.

```
    TLbyte arg_address;
```

У цій змінній зберігатиметься адрес байту, отриманого з режиму адресації.

```
    TLbyte arg_value;
```

У цій змінній зберігатиметься аргумент, отриманий з режиму адресації.

```
} TL;
```

Вищезгаданий псевдонім структури TLdata.

```
void tlInit(TL* tl);
```

Декларація функції tlInit яка приймає вказівник на екземпляр структури TLdata та має тип значення, що повертається – void. Ця функція потрібна для ініціалізації екземпляру структури TLdata, також це можна назвати «за замовчуванням». Функція записує значення нуль у всі регістри та лічильник кроків, виставляє нулі на байтах оперативної пам’яті та пам’яті стеку, але головне це виставляє вказівники на «нульові» функції зворотнього виклику. Нульові, тому що це функції, які нічого не роблять, і вони реалізовані у файлу бібліотеки. Це дозволяє вказати лише ті функції зворотнього виклику, які дійсно необхідні. Не забувайте використовувати tlInit під час роботи з бібліотекою.

```
void tlClock(TL* tl);
```

Декларація функції tlClock яка приймає вказівник на екземпляр структури TLdata та має тип, що повертається – void. У попередньому розділі було сказано, що робота процесору ділиться на кроки, отож функція tlClock це програмна реалізація одного такого кроку. Ви можете використовувати цю

функцію для реалізації власного циклу роботи процесору, наприклад, просто викликати її всередині while (1) { … }. Так само це дозволяє мати кілька процесорів tiny-lemon, що працюють «паралельно», тобто в тілі циклу викликати tlClock для кожного процесору.

```
void tlRun(TL* tl);
```

Декларація функції tlRun яка приймає вказівник на екземпляр структури TLdata та має тип значення, що повертається – void. Реалізує нескінченний цикл із викликів tlClock. Це найпростіший спосіб отримати нескінченно працюючий процесор. Зрозуміло, що у такий спосіб і без багатопоточності не вийде отримати більше одного працюючого процесора. Функція tlRun не надає жодного способу вийти із нескінченного циклу, так що вирішення цієї проблеми лягає на користувача, наприклад, можна використовувати функцію exit з «stdlib.h» файлу заголовку стандартної бібліотеки мови C.

```
void tlDoExternalInterrupt(TL* tl, TLbyte value);
```

Декларація функції tlDoExternalInterrupt приймає вказівник на екземпляр структури TLdata та байт даних, а тип значення, що повертається – void. Ця функція реалізує механізм зовнішнього переривання, байт даних при цьому потрапить у регістр A. Переривання гарантоване, і буде викликано навіть якщо прапор I, що забороняє зовнішні переривання, дорівнює одиниці. Тому використовувати цю функцію не рекомендується і краще використовувати варіант нижче.

```
TLbool tlTryExternalInterrupt(TL* tl, TLbyte value);
```

Декларація функції tlTryExternalInterrupt яка аналогічна tlDoExternalInterrupt, але переривання відбудеться лише якщо зовнішні переривання не заборонені прапором I. Якщо переривання вдалося, то функція поверне одиницю, інакше – нуль. Таким чином можна написати реалізацію зовнішнього пристрою так, щоб він продовжував спроби зробити переривання, поки не досягне успіху.

```
#ifdef __cplusplus

}

#endif
```

Додавання дужки що закриває конструкцію extern “C” { … }, звісно лише тоді коли використовується C++ компілятор.

```
#endif
```

Останній рядок у файлі заголовку бібліотеки. Директива endif, що закриває конструкцію ifndef-endif, відкриту на самому початку файлу.

Як користуватися?

У коді вашої програми повинен бути визначений як мінімум один екземпляр структури TLdata, це може бути локальна або глобальна змінна або пам’ять, яка отримана іншими методами, наприклад за допомогою алокатора пам’яті.

Визначення змінної типу TLdata виглядатиме так:

```
TL mytl;
```

Спочатку вказано тип TLdata або псевдонім TL, потім через пропуск – назва «mytl», і в кінці крапка з комою яка є знаком закінчення ствердження (англ. *Statement*) і не є частиною назви змінної. Назва змінної може бути будь-яка, головне, щоб вона відповідала правилам синтаксису мови програмування С, тобто починалася з букви або символу підкреслення і містила в собі лише букви, цифри та символи підкреслення. Текст далі, передбачає, що назва змінної це «mytl». Взаємодія зі змінною можлива тільки після її визначення, тобто визначення повинно перебувати вище, ніж використання змінної, а також враховувати область видимості.

Перед тим, як щось робити з отриманою змінною, потрібно спочатку провести ініціалізацію. Це робиться шляхом виклику функції tlInit. Викликати функцію потрібно лише один раз для кожного екземпляра структури TLdata.

Приклад ініціалізації:

```
tlInit(&mytl);
```

Зверніть увагу на префіксний оператор «&» він потрібен для того, щоб отримати вказівник на змінну та передати його у функцію. Якщо не хочете щоразу використовувати цей оператор, можна оголосити змінну, що зберігає вказівник, наприклад так:

```
TL* tl_ptr = &mytl;
```

У рядку вище не тільки оголошується змінна tl_ptr, а також відбувається присвоєння або іншими словами – запис значення у змінну. Знак дорівнює «=» це оператор присвоєння. У разі змінної tl_ptr присвоюється значення вказівника на змінну mytl. Те, що змінна tl_ptr є вказівником, можна зрозуміти за символом зірки «*» та вказує він на структуру TLdata.

Один раз отримавши вказівник можна використовувати його замість конструкції «&mytl»:

```
tlInit(tl_ptr);
```

Пам’ятайте, що ви можете називати змінні як завгодно, але у книзі далі фігурує назва tl_ptr.

Провівши ініціалізацію, можна розпочати завантаження програми для tiny-lemon у пам’ять. Немає готової бібліотечної функції для цієї дії, тому потрібно реалізовувати це самостійно. Наприклад, надавати значення по одному:

```
mytl.ram[0] = 0x7C;

mytl.ram[1] = 0x7D;

mytl.ram[2] = 0x36;
```

Таким чином, першим трьом байтам пам'яті tiny-lemon буде присвоєно значення 0x7C, 0x7D і 0x36 відповідно. Значення представлені в коді у 16-му вигляді, це можна зрозуміти за префіксом «0x», це зручно для розшифровки інструкцій. У попередньому розділі розташована таблиця інструкцій, в якій можна знайти будь-яку інструкцію поділивши 16-ий код на праву та ліву цифри, потім використовуючи їх як X та Y координати відповідно. Наприклад, 0x7C – це інструкція INA. Дві інструкції, що залишилися, спробуйте знайти самостійно. Якщо хочете себе перевірити цю невелику програму з трьох інструкцій можна знайти в розділі «приклади програм», а саме програма «Cat».

Зрозуміло, що для досить великих програм такий спосіб запису у пам’ять не підходить через свою громіздкість. Коли будь-яка дія повторюється багато разів, наприклад запис значень у пам’ять, доцільно використовувати цикл.

```
int n = sizeof(arr) / sizeof(TLbyte);

if (n > 256) n = 256;

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

      mytl.ram[i] = arr[i];

}
```

Передбачається, що у нас вже є масив байт під назвою arr, в якому зберігається програма для процесора tiny-lemon. Використовуючи оператор sizeof можна обчислити розмір масиву і зберегти його у змінній n. Якщо розмір масиву перевищує розмір пам’яті, змінна n буде урізана під розмір пам’яті. Потім, починаючи з першого елемента, всі елементи масиву будуть записані в пам’ять tiny-lemon. Наприклад, масив arr може містити ті самі три байти:

```
TLbyte arr[] = { 0x7C, 0x7D, 0x36 };
```

Для зручності читання коду, логічного поділу програми на окремі незалежні модулі, спрощення завдання дебагу та багаторазового використання коду потрібно створити функцію load_program. Така функція буде виконувати лише завдання завантаження програми в пам'ять tiny-lemon.

```
void load_program(TL* tl_ptr, TLbyte* arr, int n) {

      if (n > 256) n = 256;

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

            tl_ptr->ram[i] = arr[i];

      }

}
```

Використовувати написану функцію можна необмежену кількість разів. Виклик функції може виглядати так:

```
load_program(tl_ptr, arr, sizeof(arr)/sizeof(TLbyte));
```

Зверніть увагу, що в функцію load_program потрібно передавати значення саме в такому порядку:

1) вказівник на структуру TLdata;
2) масив або, що теж саме, вказівник на перший елемент масиву;
3) кількість елементів масиву.

Також вам може знадобитися функція fread_program для завантаження програми не з масиву, а з файлового потоку. Для цієї мети можна використовувати FILE* зі стандартної бібліотеки «stdio.h» мови програмування C, а також функції для роботи з FILE* з тієї ж бібліотеки.

```
void fread_program(TL* tl_ptr, FILE* f) {
	for (int i = 0; i < 256; i++) {
		int b = fgetc(f);
		if (b == EOF) return;
		tl_ptr->ram[i] = (TLbyte)b;
	}
}
```

Пояснення того, як працювати з файловими потоками, виходить за рамки цієї книги і рекомендується ознайомитися з можливостями стандартної бібліотеки «stdio.h» самостійно.

Після того, як програма тим чи іншим способом завантажена в пам'ять tiny-lemon можна приступати до інтерпретації цієї програми. Як було сказано можна написати свій цикл процесора, наприклад:

```
while (is_run) tlClock(tl_ptr);
```

Де is_run це глобальна змінна, яку можна змінити в будь-яку мить. Спочатку змінна має не нульове значення, таким чином цикл процесора почнеться і продовжуватиметься доки змінній is_run не буде присвоєно значення нуль.

Але найпростіше це використання функції tlRun щоб створити нескінченний цикл процесора:

```
tlRun(tl_ptr);
```

Процесор буде працювати, інтерпретуючи програму з пам'яті, але ніякої взаємодії із «зовнішнім світом» не відбудеться, оскільки все ще не були призначені функції зворотнього виклику. Для початку їх потрібно написати:

```
TLbyte my_input(TL* tl_ptr, TLbyte port) {
	return getchar();
}
void my_output(TL* tl_ptr, TLbyte data, TLbyte port) {
	putchar(data);
}
void my_clock_debug(TL* tl_ptr) {
	printf("\n%3i | ", tl_ptr->reg.P);
}
```

Залишилося лише виконати операцію присвоєння для відповідних вказівників на функції:

```
mytl.callback.in = my_input;

mytl.callback.out = my_output;

mytl.callback.clock = my_clock_debug;
```

Процесор буде використовувати функцію my_input для виконання інструкції INA, значення, яке поверне функція, потрапить у регістр A, в даному випадку – байт зі стандартного потоку введення.

У свою чергу функція my_output буде використовуватись інструкцією OUT, щоб відправити дані на вихід, у даному випадку це стандартний потік виведення.

Функції my_input і my_output діють однаково незалежно від параметру port який вказує з якого пристрою приймаються дані, та на який – відправляються. Це лише приклад, ви можете зробити різну поведінку функцій залежно від значення порту, за допомогою розгалужень if-else та switch.

Кожен крок процесор буде викликати функцію my_clock_debug яка виводить значення регістра P, таким чином можна легко простежити, які і в якій послідовності виконуються інструкції. Це зручно при пошуку помилок, але коли весь функціонал протестований, можна закоментувати рядок із присвоєнням, тим самим відключивши цей функціонал:

```
// mytl.callback.clock = my_clock_debug;
```

Якщо ви використовуєте свій власний цикл процесору і хочете мати можливість вийти з нього, то можете прив’язати вихід з циклу до виконання інструкції HLT реалізувавши, таку функцію:

```
void my_halt(TL* tl_ptr) {

      is_run = 0;

}
```

І додати ще одне присвоєння:

```
mytl.callback.halt = my_halt;
```

Виконання інструкції HLT призведе до виклику функції my_halt, і змінна is_run стане нуль. Якщо ваш цикл виглядає так:

```
while (is_run) tlClock(tl_ptr);
```

Те нульове значення змінної is_run призведе до виходу з циклу while.

Весь код із файлу main.c може виглядати так:

```
#include<stdio.h>
#include"tiny-lemon.h"
void fread_program(TL* tl_ptr, FILE* f) {
    for (int i = 0; i < 256; i++) {
        int b = fgetc(f);
        if (b == EOF) return;
        tl_ptr->ram[i] = (TLbyte)b;
    }
}
TLbool is_run = 1;
TLbyte my_input(TL* tl_ptr, TLbyte port) { return getchar(); }
void my_output(TL* tl_ptr, TLbyte data, TLbyte port) { putchar(data); }
void my_clock_debug(TL* tl_ptr) { printf("\n%3i | ", tl_ptr->reg.P); }
void my_halt(TL* tl_ptr) { is_run = 0; }
int main(int argc, char** argv) {
      TL mytl;
      TL* tl_ptr = &mytl;
      FILE* f = fopen("./data.bin", "rb");
      if (f == 0) return 1;
      tlInit(tl_ptr);
      fread_program(tl_ptr, f);
      fclose(f);
      mytl.callback.in    = my_input;
      mytl.callback.out   = my_output;
      mytl.callback.clock = my_clock_debug;
      mytl.callback.halt  = my_halt;
      while (is_run) tlClock(tl_ptr);
      return 0;
}
```

Щоб з файлу main.c отримати виконуваний файл потрібно використовувати компілятор C, наприклад GNU C Compiler тобто gcc. Мінімальна команда на ОС Linux виглядатиме так:

gcc ./main.c ./tiny-lemon.a

А для ОС Windows так:

gcc ./main.c ./tiny-lemon.lib

В загальному вигляді це:

gcc <шлях до файлу з кодом> <шлях до файлу статичної бібліотеки tiny-lemon>

Це не весь доступний функціонал компілятора gcc, але для старту цього буде достатньо. За бажанням ви можете ознайомитися з усіма можливостями gcc самостійно.

При успішній компіляції коду, ви отримаєте виконуваний файл – повноцінну віртуальну машину на базі процесору tiny-lemon.

Якщо компіляція не була успішною, компілятор виведе повідомлення про помилку і про те, де і що призвело до помилки під час компіляції.

Не варто забувати про помилки часу виконання, у цьому прикладі файлу main.c програмі необхідний роботи файл data.bin, який містить програму для tiny-lemon у тій самій папці, де знаходиться виконуваний файл.

ПРИКЛАДИ ПРОГРАМ

Cat

Програма котра відправляє на вихід все що приходить на вхід.

Адрес	0x	Код	Пояснення
00	7C	INA	Читає з порту введення значення та відправляє його на порт виведення, потім робить регістр P рівним нулю, тобто процесор почне виконувати програму спочатку і так у нескінченному циклі.
01	7D	OUT	
02	36	ZRP	

Hello, World!

Класика у світі програмування – програма, що пише текст «Hello, World!».

Адрес	0x	Код	Пояснення
00	16	LDA__P	Першими двома інструкціями завантажує в регістр D значення 0x09, що збігається з адресом тексту (див. адрес 09). Далі починається «тіло циклу». Читає байт по адресу з регістру D, тобто один символ тексту. При цьому оновлюються прапори Z та N. Відправляє прочитаний символ у порт 0, маючи на увазі, що у цей порт підключено пристрій – термінал. Збільшує значення регістру D на 1. І якщо раніше оновлений прапор Z дорівнює 0 (False) повертається на початок циклу, тобто адрес 0x03. Якщо стрибка не відбулося, то програма зупинитиметься на нескінченній інтерпретації інструкції HLT. Таким чином читання символів з пам'яті та відправлення їх на екран терміналу відбуватиметься до того моменту, поки не буде зустрінутий нульовий символ.
01	09	$09	
02	09	LDD	
03	12	LDA__D	
04	7D	OUT	
05	DC	ICD	
06	50	JFZ_LP	
07	03	$03	
08	7E	HLT	
09	48	=H	
10	65	=e	
11	6C	=l	
12	6C	=l	
13	6F	=o	
14	2C	=,	
15	20	=	
16	57	=W	
17	6F	=o	
18	72	=r	
19	6C	=l	
20	64	=d	
21	21	=!	
22	00	$00	

Quine

Програма що виводить сама себе на екран. У даному випадку не текст, а бінарні данні.

Адрес	0x	Код	Пояснення
00	12	LDA__D	Завантажує значення байту по адресу із D (спочатку D дорівнює 0). Відправляє щойно завантажене значення. Збільшує регістр D на 1 тим самим вказуючи на наступний байт. Завантажує значення регістру D у регістр A. Оновлює прапори виходячи з результату операції «A – 9», у цьому випадку важливо те, що якщо A дорівнює 9, то при відніманні вийде 0. Якщо прапор Z не дорівнює 0, повертається до початку програми, в іншому випадку зупиняється на інструкції HLT. Магічне число 9 – це розмір програми у байтах.
01	7D	OUT	
02	DC	ICD	
03	01	MVD	
04	B6	CMP__P	
05	09	+9	
06	50	JFZ_LP	
07	00	+0	
08	7E	HLT	

Послідовність Фібоначчі

Перші два числа послідовності Фібоначчі це 0 та 1, кожне наступне число це сума двох попередніх.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377…

Написанням програми, яка обчислює послідовність Фібоначчі, можна довести повноту за Тюрінгом.

Адрес	0x	Код	Пояснення
00	7D	OUT	Выводит значение регистра A, который изначально равен 0. Меняет местами значения регистра A и байта по адресу 0x0A. Назовем этот байт переменной X. Прибавляет к регистру A значение переменной X. Если значение не равно 121 вернуться к адресу 0x00, иначе остановиться. Магическое число 121 – это на самом деле число 377 которое не влезло в диапазон байта, то есть программа остановиться тогда, когда выведет на экран все числа Фибоначчи, помещающиеся в диапазон байта.
01	2F	XCB_LP	
02	0A	+10	
03	C8	ADD_LP	
04	0A	+10	
05	B6	CMP__P	
06	79	+121	
07	50	JFZ_LP	
08	00	+0	
09	7E	HLT	
10	01	+1	

Машина істинності

Програма, котра, отримав на вході 0 виведе нуль один раз, а отримавши 1 буде нескінченно відправляти одиницю на екран.

Адрес	0x	Код	Пояснення
00	7C	INA	Читає значення, оновлює прапори шляхом завантаження значення з регістру A в регістр A. У регістрі A буде значення 0 або 1 і відповідно прапор Z буде 0 або 1. Якщо прапор Z дорівнює 0 програма завершиться, якщо 1 тоді продовжить нескінченно виводити одиницю.
01	08	LDA	
02	7D	OUT	
03	50	JFZ_LP	
04	02	+2	
05	7E	HLT	

Сума 64-біт чисел

Перед тим як додати два 64-біт числа потрібно розмістити їх у пам’ять tiny-lemon. Звичайно, можна вручну перевести числа з десяткової системи числення до 16-ої, потім записати в програму значення байтів (кожен байт це дві 16-і цифри). Але коли потрібно буде по експериментувати з різними значеннями, повторювати цей процес щоразу повільно, тож краще написати функцію:

```
void write64(TL* tl_ptr, TLbyte adr, uint64_t v) {
      for (int i = 0; i < sizeof(v); i++, adr--) {
            tl_ptr->ram[adr] = v & 0xFF;
            v >>= 8;
            adr--;
      }
}
```

Зверніть увагу, що аргумент adr вказує на останній байт числа, а запис йде у зворотному напрямку, байт за байтом. У результаті, 64-біт число 0x1122334455667788 буде записано в пам’ять, зберігаючи порядок цифр, де 0x88 це значення байту, на який вказував adr:

0x11	0x22	0x33	0x44	0x55	0x66	0x77	0x88

Давайте збережемо в пам’ять два числа:

```
write64(tl_ptr, 0x80, 1020304050607080900L);
write64(tl_ptr, 0x90, 1010101010101010101L);
```

Число по адресу 0x80 назвемо Y, а число по адресу 0x90 – X.

Програма для процесору tiny-lemon обчислить суму двох чисел X та Y так, що результат замінить собою число Y. У нашому випадку сума виглядає так:

1020304050607080900

+

1010101010101010101

=

2030405060708091001

Тобто по адресу 0x80 буде знаходитись число 2030405060708091001.

Адрес	0x	Код	Пояснення
00	16	LDA__P	У регістр D зберігається адрес числа Y.
01	80	$80	
02	09	LDD	
03	16	LDA__P	У регістр K зберігається адрес числа X.
04	90	$90	
05	0C	LDK	
06	6F	JSR_LP	Викликається підпрограма по адресу 9 вона обчислює суму чисел, адреси яких записані в D і K, а результат замінить число за адресом із D. Потім процесор зупинитися на інструкції HLT.
07	09	+9	
08	7E	HLT	
09	81	PSD	На стеку зберігаються значення регістрів D і K, щоб наприкінці повернути їх початкові значення. Прапор перенесення стає нуль. У регістр A завантажується значення 8, що є розміром 64-біт числа у байтах і відповідно кількістю ітерацій циклу.
10	84	PSK	
11	41	FFC	
12	16	LDA__P	
13	08	+8	
14	80	PSA	Кладе значення регістру A на стек. Значення байту по адресу із регістру D зберігається в байт по адресу 20 який використовується як аргумент інструкції ADC__P. У регістр A завантажується значення байту по адресу із регістру K. До регістру A додається раніше завантажене значення, враховуючи прапор перенесення, та розраховуючи новий прапор перенесення для наступної суми. Результат замінює байт по адресу із регістру D. Регістри D і K зменшуються на один. Зі стека повертається значення A збережене раніше. Значення у регістрі A у комбінації з інструкцією LUP_LP забезпечує повторення цього блоку коду 8 разів. Тобто розрахує суму кожного байту числа Y з відповідним байтом числа X з урахуванням перенесення, а результат замінить собою Y.
15	12	LDA__D	
16	1F	STA_LP	
17	14	+20	
18	14	LDA__K	
19	C9	ADC__P	
20	00	+0	
21	1A	STA__D	
22	DD	DCD	
23	DF	DCK	
24	90	PPA	
25	77	LUP_LP	
26	0E	+14	
27	94	PPK	Після циклу повертаються початкові значення регістрів D і K. У регістр P записується значення зі стеку, що було залишено інструкцією JSR_LP, необхідне для повернення з підпрограми.
28	91	PPD	
29	96	PPP	

Щоб подивитися чи правильний результат нам потрібен спосіб прочитати число з пам’яті, наприклад написавши для цього спеціальну функцію:

```
uint64_t read64(TL* tl_ptr, TLbyte adr) {
	uint64_t v = 0;
	adr -= sizeof(v);
	for (int i = 0; i < 8; i++) {
		adr++;
		v <<= 8;
		v |= tl_ptr->ram[adr];
	}
	return v;
}
```

Після роботи програми прочитаємо число Y та виведемо його на екран:

```
printf("\nY = %llu\n", read64(tl, 0x80));
```