

tiny-lemon
для чайников



автор Поэт Лучник

СОДЕРЖАНИЕ

предисловие	2
Для кого эта книга?	2
Зачем мне это всё?	2
Как пользоваться книгой?	2
Последующие разделы	2
вступление	3
центральный процессор	4
Оперативная память	4
Стек	4
Прерывания	5
Регистры	5
Числовые значения	6
Работа процессора	7
Режимы адресации	7
Операции	8
Операции над числами размером больше байта	11
Инструкции	12
статическая библиотека	14
Заголовочный файл	14
Как пользоваться?	18
примеры программ	25
Cat	25
Hello, World!	25
Quine	26
Последовательность Фибоначчи	26
Машина истинности	27
Сумма 64-бит чисел	27

ПРЕДИСЛОВИЕ

Для кого эта книга?

Эта книга предназначена для новичков в программировании которые, не иначе как чудом, наткнулись на `tiny-lemon` и хотят узнать об этом больше. В любом случае это книга может быть использована любым человеком вне зависимости от уровня знаний и преследуемых целей.

Зачем мне это всё?

`tiny-lemon` создавался в первую очередь как относительно простая среда для обучения программированию снизу-вверх, то есть чтобы самостоятельно пройти путь от программирования машинным кодом до создания своего собственного языка программирования. Во вторую очередь как среда для программирования в ограниченных условиях, то есть проверка своих навыков оптимизации программ с грамотным использованием всех доступных возможностей. В любом случае `tiny-lemon` не ограничивается этим, ведь это полноценная расширяемая система, которая может быть использована для любых целей.

Как пользоваться книгой?

Читать книгу нужно по порядку, сначала и до конца. Однако, предусмотрена возможность пропускать отдельные абзацы. То есть, если поднимаемая в абзаце тема вам уже известна при желании его можно пропустить. Так же в книге присутствуют подчеркивания там, где предполагается самостоятельное ознакомление читателя с внешними источниками информации, без которых опыт будет не полный.

Последующие разделы

Вступление – раздел содержит определение общих терминов.

Центральный процессор – в разделе описано устройство процессора `tiny-lemon` в отрыве от самой реализации библиотеки.

Статическая библиотека – раздел о том, как пользоваться `tiny-lemon`, так же раздел содержит перечень структур данных и функций доступных пользователю.

Примеры программ – раздел, где можно найти программы с объяснениями того, как они работают.

ВСТУПЛЕНИЕ

tiny-lemon – это статическая библиотека написанная на языке программирования C которая является реализацией абстрактной виртуальной машины.

Виртуальная машина (сокращенно ВМ) – это программа, которая полностью или частично повторяет поведение некоторого вычислительного устройства. Виртуальную машину называют абстрактной если упомянутого устройства не существует в виде физической реализации.

Язык программирования C (читается как «Си») – как и любой другой язык, это набор слов и символов, которые используются для передачи смысла, только в отличии от человеческих языков смысл передается не от человека к человеку, а от человека к компьютеру. У языка C существуют правила написания текстов эти правила называются – синтаксис. Если текст написан синтаксически правильно, его можно считать полноценным кодом. Но это всего лишь текст, компьютеру нужна программа что бы прочесть его, и такая программа называется компилятор.

Компилятор – это программа, которая на вход получает синтаксически правильно написанный код, а на выходе создается исполняемый файл. Это основная функция компилятора, но она может быть не единственной, в зависимости от конкретной реализации, компилятор может получать на вход дополнительные данные, а также может на выходе давать что-то кроме исполняемого файла.

В мире существует несколько компиляторов для языка программирования C и в любой момент могут появиться новые поэтому в этой книге не будет объяснений как использовать каждый из существующих компиляторов. Читателю предлагается ознакомиться с тем или иным экземпляром самостоятельно, путем чтения официальной документации.

Не смотря на выше сказанное, в книге можно найти несколько примеров использования компилятора GCC. Предполагается, что этого будет достаточно для начала работы со статической библиотекой tiny-lemon.

Статическая библиотека – это файл или несколько файлов которые содержат машинные инструкции, а также некоторые данные. Это похоже на описание обычной программы, но статическая библиотека это не программа, это независимый модуль, который может быть вставлен в вашу программу для дальнейшего использования его функционала. Файлы библиотеки имеют расширение .lib на операционной системе Windows и – .a на ОС Linux.

Статическая библиотека написанная на C – включает в себя файлы самой библиотеки, но также как минимум один заголовочный файл (расширение .h). Заголовочные файлы содержат код написанный на C который является перечислением доступных структур данных и функций. В первую очередь заголовочные файлы нужны компилятору.

Работа компилятора делится на этапы. Сначала идет предварительная обработка текста (препроцессор). Затем генерация промежуточного результата который представляет из себя набор файлов объектников (расширение .obj или .o), такие файлы хранят машинный код с дополнительными данными. В конце происходит линковка – соединение всех объектников в один исполняемый файл, а также присоединение файлов библиотек.

ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР

Процессор – это устройство, выполняющее инструкции. Каждый процессор имеет свой конечный набор инструкций, где каждая инструкция имеет точное определение.

Центральный процессор – это процессор, который является главной частью компьютера.

Процессор не может существовать сам по себе, ему обязательно нужно что-то откуда можно получать инструкции. И самым частым решением является специальная память, где хранятся инструкции, представляющие собой программу.

Как раз таки tiny-lemon представляет собой комбинацию процессора и памяти.

Оперативная память

Оперативная память – это память, предназначенная для записи, хранения и чтения информации во время работы процессора. Эта память может содержать как инструкции, так и данные. У tiny-lemon память составляет 256 байт.

Байт – это 8 бит. Бит это единица измерения информации, которая представляет из себя одну цифру двоичной системы счисления, то есть 0 или 1. Перебирая все возможные комбинации из восьми бит, можно прийти к выводу что байт может хранить 256 разных значений. Обычно их представляют как числа от 0 до 255, или же как числа от -128 до 127. Так же байты удобно воспринимать в шестнадцатеричной системе счисления, как числа от 0x00 до 0xFF.

В байтах можно хранить любую информацию, любые данные. Комбинация байт может означать что угодно. Поэтому о значениях байтов всегда говорят в контексте программы, алгоритма или устройства, которые могут интерпретировать данные тем или иным способом. К примеру, текстовый редактор интерпретирует байты как набор символов, а редактор изображений рассматривает байты как информацию о изображении. И вы можете открыть файл изображения в текстовом редакторе, тем самым используя данные от одной программы в другой программе. Тот же файл можно открыть через программу архиватор что бы создать файл архив. Только что было сказано про различные файлы, возникают они из-за того, что существует программная реализация файловой системы, то есть программа которая может интерпретировать байты на диске как информацию о файлах и папках.

Стек

Стек – структура данных, которая работает по принципу «последний пришел, первый ушел». Имеются возможности положить на или взять с вершины стека значение. В tiny-lemon стек это отдельная память, которая составляет 256 байт. Но поскольку прерывание переполнения будет вызвано при попытке записи в последний байт, можно считать что доступной памяти стека 255 байт.

Прерывания

Прерывание – это реакция процессора на некоторое событие и сопровождается передачей управления специальной подпрограмме обработчику прерываний. Прерывание может быть вызвано в любой момент работы процессора. Это позволяет процессору моментально реагировать на события. Когда обработчик прерываний заканчивает работу он возвращает выполнение программы на то место, на котором произошло прерывание. В *tiny-lemmon* прерывание может быть вызвано как внутренним состоянием процессора, так и внешним устройством, вне зависимости от типа прерывания адрес подпрограммы обработчика находится в памяти по адресу 255 (последний байт). Стоит отметить, что во время внешнего прерывания устройство посылает 1 байт, который попадет в регистр А и это можно использовать для определения какое устройство вызвало прерывание.

Регистры

Регистр – это отдельная ячейка памяти, с возможностью чтения и записи значений, которая используется процессором для различных инструкций. У *tiny-lemmon* восемь регистров и каждый по восемь бит. Названия и значения регистров описаны в таблице «Регистры процессора *tiny-lemmon*». Все регистры изначально имеют значение 0.

Регистры процессора *tiny-lemmon*

Имя	Полное имя	Значение
A	Аккумулятор	Используется как аргумент некоторых инструкций и для хранения результата различных операций.
D	Ди	Не имеет заранее определенного значения и может быть использован для любых целей.
F	Флаги	Хранит информацию о результатах операций, а также используется для управления прерываниями и условными переходами. Каждый из восьми бит этого регистра имеет свое значение, которое можно посмотреть в таблице «Биты регистра F».
I	Индекс Ввода	Индекс порта ввода, используется операциями ввода.
K	Кэй	Не имеет заранее определенного значения и может быть использован для любых целей.
O	Индекс Вывода	Индекс порта вывода, используется операциями вывода.
P	Программный Счётчик	Адрес байта инструкции, которая будет выполнена в следующий цикл процессора. Во время выполнения инструкции значение этого регистра автоматически увеличивается на один. Поскольку изначально регистр равен 0 процессор выполняет программу начиная с первого байта оперативной памяти.
S	Указатель Стека	Указывает на вершину стека и используется операциями работы со стеком. Регистр указывает именно на самый верхний не занятый байт в памяти стека. Изначально это адрес 0, ведь на стек еще ничего не положили. При попытке взять со стека значение, когда регистр S равен 0, будет вызвано прерывание по причине недополнения стека (англ. <i>Stack Underflow</i>). С другой стороны. При попытке что-то положить на стек, когда регистр S равен 255 (максимальному значению), это вызовет прерывание, в связи с переполнением стека (англ. <i>Stack Overflow</i>).

Биты регистра F

Разряд	Имя	Полное имя	Значение
1	Z	Ноль	Значение этого бита автоматически изменяют некоторые операции, если бит равен 1 это указывает на то, что результат операции равен нулю.
2	C	Перенос	Значение этого бита автоматически изменяют некоторые операции, если бит равен 1 это указывает на то, что результат операции вышел за пределы байта, с левой или правой стороны.
3	N	Отрицательный	Значение этого бита автоматически изменяют некоторые операции, если бит равен 1 это указывает на то, что результат операции был отрицательным или, что тоже самое, результат был больше, чем 127
4	O	Смена Знака	Значение этого бита автоматически изменяют некоторые операции, если бит равен 1 это указывает на то, что во время операции произошла некорректная смена знака - сумма двух положительных чисел дала отрицательный результат или сумма отрицательных дала положительный результат.
5	I	Запрет Прерываний	Изменяя значение этого бита, можно управлять внешними прерываниями. Пока бит равен 1 процессор будет игнорировать сигналы прерываний от внешних устройств.
6	V	Перерыв	Инструкции внутренних прерываний автоматически выставляют значение этого бита на 1. Это позволяет отличить такие прерывания от любых других.
7	S	Прерывание Стека	Значение бита выставляется на 1 во время прерываний в связи с переполнением или недополнением стека.
8	U	Неизвестный	Не имеет какого-либо определенного значения и может быть использован для любых целей.

Числовые значения

Числа в байтах кодируются как обычные двоичные, то есть вот это 0b00000000 означает 0, а вот это 0b01011010 означает $64 + 16 + 8 + 2 = 90$, и вот это 0b11111111 число 255. О переводе из одной системы счисления в другую можете узнать самостоятельно.

Любую последовательность бит можно воспринять как исключительно положительное число, или как число со знаком. В tiny-lemoн смена знака определяется как $-x = \text{not}(x) + 1$. Это удобно, потому что любые операции суммы и вычитания которые работают корректно для беззнаковых чисел будут работать корректно и для чисел со знаком. К примеру, $0b00000001 + 0b11111111 = 0b00000000$ в беззнаковых будет восприниматься как $1 + 255 = 0$ (256, но произошло переполнение байта), в знаковых это $1 + (-1) = 0$ что тоже является правильным результатом. Это касается только суммы и вычитания чисел, для умножения и деления нужно два разных алгоритма один для чисел со знаком другой для беззнаковых.

Работа процессора

Работа процессора делится на шаги. Шаг – это выполнения одной инструкции, но также включает в себя действия, которые гарантированно выполняются перед и после инструкции.

Каждая инструкция в tiny-lemoн делится на две независимых части: режим адресации и операция. Это означает что нет необходимости заучивать значение каждой инструкции, достаточно запомнить режимы адресации и операции по отдельности и легко понимать любую комбинацию этих двух частей. Об этом будет сказано в следующих двух главах: «Режимы адресации» и «Операции».

Один шаг процессора tiny-lemoн можно описать такой последовательностью действий:

1. Послать сигнал на специальный выход – оповестить внешние устройства о начале шага.
2. Получить инструкцию из оперативной памяти по адресу из регистра Р.
3. Увеличить значение регистра Р на 1.
4. Узнать по значению инструкции режим адресации и выполнить его.
5. Узнать по значению инструкции операцию и выполнить её.
6. Увеличить счетчик шагов на 1.

Важно еще раз отметить, что регистр Р увеличивается на 1 после получения инструкции, но перед её интерпретацией. Это значит, что на этапе режима адресации регистр Р будет хранить адрес уже следующего байта.

Во время любого прерывания процессор выполняет такую последовательность действий:

1. Кладет на стек значение регистра Р.
2. Кладет на стек регистры А, D, I, К, О, F в соответствующем порядке (что аналогично выполнению операции PSR, но об этом позже)
3. Читает значение байта по адресу 0xFF (последний байт памяти) и помещает прочитанное значение в регистр Р. То есть передает управление подпрограмме обработчику прерываний.

Режимы адресации

Режим адресации указывает процессору откуда получить адрес аргумента (далее просто, адрес) и значение аргумента (далее просто, аргумент). Адрес и аргумент используют различные операции. Все режимы адресации приведены в таблице «Режимы адресации».

Режимы адресации

Имя	Адрес	Аргумент	Доп. действия
(нет)	Из регистра Р	Из регистра А	
<u> </u> А	Из регистра А	Из байта по адресу	
<u> </u> LA	Из байта по адресу из регистра А	Из байта по адресу	
<u> </u> D	Из регистра D	Из байта по адресу	
<u> </u> LD	Из байта по адресу из регистра D	Из байта по адресу	
<u> </u> К	Из регистра К	Из байта по адресу	
<u> </u> LK	Из байта по адресу из регистра К	Из байта по адресу	
<u> </u> Р	Из регистра Р	Из байта по адресу	Увеличить регистр Р на 1
<u> </u> LP	Из байта по адресу из регистра Р	Из байта по адресу	Увеличить регистр Р на 1

Безымянный режим адресации подразумевается везде, где не указан ни один из восьми других.

Не стоит путать увеличение регистра Р на 1 то которое происходит каждый шаг с тем которое указано в дополнительных действиях режима адресации. Первое уже гарантированно произошло до режима адресации, а второе произойдет после получения адреса и аргумента.

Операции

Операция – это указание процессору что сделать, какие вычисления выполнить, какие манипуляции с той или иной памятью произвести, как и какие флаги регистра F обновить.

Условные обозначения в названиях и описаниях операций:

1. Буква *b* в названии означает что есть две группы этой операции. В первой группе вместо *b* подставлено F, а во второй – T. Возможно вам будет проще понять так *boolean*, *False*, *True*. В описании операции буква *b* означает 0 или 1 соответственно.
2. Буква *f* в названии и описании означает что вместо неё можно подставить название любого из восьми бит регистра флагов F.
3. Буква *r* в названии и описании означает что вместо неё можно подставить название любого из восьми регистров процессора.

Перечисление операций с их описанием (в алфавитном порядке):

ADD – (англ. *ADDition*), прибавляет к регистру A значение аргумента (то есть это изменит значение регистра A), обновляет флаги Z и N исходя из полученного в A значения. Флаг C становится 1, если сумма получается больше, чем 255, в противном случае флаг C станет 0, тем самым сохраняя потребность в переносе. Флаг O станет 1 если A и аргумент положительны, а результат суммы отрицательный, и наоборот. Во всех других случаях флаг O станет 0.

ADC – (англ. *ADDition with Carry*), действует аналогично операции ADD, но добавляя в общую сумму регистра и аргумента также единицу в случае если флаг C равен 1.

AND – (англ. *bitwise AND*), вычисляет побитовое И для регистра A с аргументом, сохраняя результат в регистре A, обновляя флаги Z и N в соответствии с результатом.

BRK – (англ. *BReaK*), процессор совершает прерывание, флаг B при этом становится 1.

CMР – (англ. *CoMPare*), вычисляет разницу между значениями регистра A и аргумента, обновляет флаги Z, N, C, O аналогично операции SUB, но не сохраняет результат вычисления в регистр A.

DCA – (англ. *DeCrement A register*), действует аналогично операции SUB с аргументом равным 1, обновляя флаги Z, N, C, O соответственно.

DCD – (англ. *DeCrement D register*), уменьшает значение регистра D на 1.

DCK – (англ. *DeCrement K register*), уменьшает значение регистра K на 1.

DEC – (англ. *DeCrement byte*), уменьшает значение байта по адресу на 1. Обновляет флаги аналогично операции SUB, будто от аргумента отнимают 1, и результат сохраняют в байт.

Fbf – (англ. *set Flag to b, name of flag is f*), флаг f становится равен b.

HLT – (англ. *HaLT*), посылает сигнал что бы оповестить об остановке, затем уменьшает значение регистра R на 1, что создает бесконечный цикл выполнения этой операции. То есть процессор останавливается на текущей инструкции и интерпретирует её бесконечно.

ICA – (англ. *InCrement A register*), действует аналогично операции ADD с аргументом равным 1, обновляя флаги соответственно.

ICD – (англ. *InCrement D register*), увеличивает значение регистра D на 1.

ICK – (англ. *InCrement K register*), увеличивает значение регистра K на 1.

INA – (англ. *INput register A*), посылает устройству ввода по индексу из регистра I сигнал, что бы устройство отреагировало и отправило процессору значение которое затем будет сохранено в A.

INC – (англ. *INCrement byte*), увеличивает значение байта по адресу на 1. Обновляет флаги аналогично операции ADD, будто это сумма аргумента и единицы, сохраняя результат в байт.

Jbf – (англ. *Jump if value b in flag f*), регистр R становится равен адресу, но только если флаг f равен значению b. Ака «Условный переход».

JMP – (англ. *JuMP*), регистр R становится равен адресу. Ака «Безусловный переход».

JSR – (англ. *Jump to Sub-Routine*), сначала помещает значение регистра R на вершину стека, затем регистр R становится равен адресу. Ака «Вызов подпрограммы».

LDr – (англ. *LoaD r register*), обновляет флаги Z и N исходя из аргумента, затем регистру r присваивается значение аргумента.

LUP – (англ. *Loop*), уменьшает значение регистра A на 1, затем если A не равен 0, регистр R примет значение адреса.

MVx – (англ. *MoVe x register*), операция присвоения, такая что, регистр A становится равен регистру x (вместо x можно подставить название любого регистра кроме A). После присвоения обновляются флаги Z и N исходя из значения которое попало в A.

NEG – (англ. *NEGative byte*), меняет знак значения байта по адресу на противоположный. Обновляет флаги Z и N исходя из результата и флаг O станет 1 в любом случае, кроме случая, когда значение байта 0, ведь смена знака у нуля ничего не меняет.

NGA – (англ. *NeGative A register*), меняет знак регистра A на противоположный. Обновляет флаги Z и N исходя из результата и флаг O станет 1 в любом случае, кроме случая, когда значение регистра это 0, ведь смена знака у нуля ничего не меняет.

NOP – (англ. *No OPeration*), никакой операции.

NOR – (англ. *bitwise NOR*), вычисляет побитовое ИЛИ для регистра A с аргументом, результат делает побитово противоположным и затем сохраняет в A. Обновляет флаги Z и N исходя из значения попавшего в регистр A.

NOT – (англ. *bitwise NOT*), меняет значение байта по адресу побитово на противоположное (не путать со сменой знака). Обновляет флаги Z и N исходя из результата, и во всех случаях флаг O станет 1.

NTA – (англ. *bitwise NoT A register*), меняет значение регистра A побитого на противоположное (не путать со сменой знака). Обновляет флаги Z и N исходя из результата, и во всех случаях флаг O станет 1.

ORA – (англ. *bitwise OR*), вычисляет побитовое ИЛИ для регистра A с аргументом, сохраняя результат в A. Обновляет флаги Z и N исходя из результата.

OUT – (англ. *OUTput A register*), отправляет значение аргумента вместе с специальным сигналом на порт устройства вывода по индексу из регистра O.

PIK – (англ. *PeeK to byte*), берет копию значения со стека и помещает его в байт по адресу.

PKr – (англ. *PeeK r register*), берет копию значения со стека и помещает его в регистр r.

POP – (англ. *POP to byte*), берет значение со стека и помещает его в байт по адресу.

PPr – (англ. *PoP r register*), берет значение со стека и помещает его в регистр r.

PPR – (англ. *PoP all Registers*), берет со стека 6 значений которые помещает в регистры в соответствующем порядке F, O, K, I, D, A. Среди перечисленных нет регистров S и P, ведь изменение первого приведет к потере последовательности стека, а изменение второго приведет к потере последовательности инструкций. Эта операция не может вызвать прерываний в связи с ошибкой стека.

PSH – (англ. *PuSH*), помещает значение аргумента на стек.

PSr – (англ. *PuSh r register*), помещает значение регистра r на стек.

PSR – (англ. *PuSH all Registers*), помещает на стек значения регистров A, D, I, K, O, F в соответствующем порядке. Эта операция не может вызвать прерываний в связи с ошибкой стека.

RTI – (англ. *ReTurn from Interrupt*), берет со стека 7 значений которые помещает в регистры в соответствующем порядке F, O, K, I, D, A, P. Тем самым совмещает в себе PPR и PPP, что позволяет совершить возврат из прерывания. Эта операция не может вызвать прерываний в связи с ошибкой стека.

SBC – (англ. *SuBtraction with Carry*), действует аналогично операции SUB, но отнимает от общей суммы дополнительно единицу в случае если флаг C равен 1. Тем самым обеспечивая корректный перенос по правилам вычитания.

SHL – (англ. *bitwise SHift Left*), записывает значение самого левого бита регистра A в флаг C, сдвигает все биты регистра A на 1 бит влево, обновляет флаги Z и N исходя из результата операции.

SHR – (англ. *bitwise SHift Right*), записывает значение самого правого бита регистра A в флаг C, сдвигает все биты регистра A на 1 бит вправо, обновляет флаги Z и N исходя из результата операции.

SLC – (англ. *bitwise Shift Left with Carry*), запоминает значение самого левого бита регистра A, сдвигает все биты регистра A на 1 бит влево, добавляет 1 к регистру A если флаг C равен 1, обновляет флаги Z и N исходя из результата операции. Записывает в флаг C значение, которое ранее запомнил.

SRC – (англ. *bitwise Shift Right with Carry*), запоминает значение самого правого бита регистра A, сдвигает все биты регистра A на 1 бит вправо, добавляет 128 к регистру A если флаг C равен 1, обновляет флаги Z и N исходя из результата операции. Записывает в флаг C значение, которое ранее запомнил.

STA – (англ. *STore A register*), записывает значение регистра A в байт по адресу.

SUB – (англ. *SUBtraction*), прибавляет к регистру A отрицательное значение аргумента, то есть по сути это вычитание. Обновляет флаги аналогично операции ADD. При чем флаг C будет обновлен корректно имея в виду вычитание – если от меньшего отнять большее произойдет перенос, который сохраниться в флаге C как 1 и наоборот флаг будет 0 в случаях, когда перенос не произошел.

TST – (англ. *TeST*), выполняет операцию побитового AND регистра A с аргументом, обновляет флаги Z и N, но не сохраняет результат самой операции в регистр A.

XCB – (англ. *eXChange Byte*), меняет местами значения регистра A и байта по адресу, затем обновляет флаги Z и N исходя из значения попавшего в A.

XCr – (англ. *eXChange r register*), меняет местами значения регистра A и регистра r, затем обновляет флаги Z и N исходя из значения попавшего в A.

XOR – (англ. *bitwise eXclusive OR*), вычисляет побитовое исключающее ИЛИ для регистра A с аргументом, результат сохраняет в A и обновляет флаги Z и N исходя из полученного значения.

ZRB – (англ. *ZeRo Byte*), байт по адресу становится равен нулю.

ZRr – (англ. *ZeRo r register*), флаг Z станет 1, флаг N станет 0, затем регистр r становится 0.

Операции над числами размером больше байта

Операция над числами размером больше байта может быть реализована через последовательное выполнение операций над каждым байтом числа.

Сумма при помощи операций ADD и ADC. К примеру, шестнадцатеричной системе счисления $0x0201 + 0x00FF = 0x0300$ это сумма байтов $0x01 + 0xFF$ при помощи ADD (или же ADC с нулевым флагом C), в результате $0x100$, но поскольку это вышло за пределы байта результат будет обрезан до $0x00$, а флаг C станет 1, тем самым указывая на необходимость переноса. Затем суммируем байты $0x02 + 0x00$ именно через ADC что бы учесть перенос от предыдущей суммы, в результате $0x03$, и если соединить полученные однобайтные результаты будет $0x0300$ что является корректным результатом суммы.

Вышесказанное справедливо и для вычитания целых чисел, только уже с использованием SUB и SBC операций.

Побитовые сдвиги выполняются похожим методом. Тут только важно понять, как двигаться по байтам что бы учесть перенос. Что бы сдвинуть число в лево нужно идти по байтам начиная с самого правого и идя до левого, на первом байте использовать операцию SHL (или SLC с нулевым флагом C), а затем использовать только SLC. А вот что бы сдвинуть в право нужно начинать с самого левого байта и идти к правому, на первом байте использовав SHR (или SRC с нулевым флагом C), а далее только через SRC.

Для таких побитовых операций как AND, OR, NOR, XOR, NOT всё еще проще, можно рассматривать число как массив байт и выполнять операцию для каждого байта по отдельности в любом порядке.

Инструкции

В таблице ниже приведены все возможные комбинации операций с режимом адресаций, каждая из таких комбинаций это инструкция, каждая инструкция процессора tiny-lemon имеет свое числовое значение, которое можно представить как две шестнадцатеричных цифры или же один байт.

Пустые клетки означают отсутствие операции. Использование таких значений в программах не рекомендуется, так как они могут в будущих обновлениях приобрести смысл. Если вам нужна инструкция, которая ничего не делает используйте 0x00 (NOP)

Инструкции и их значения

	*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*																

К примеру, инструкция AND__P это значение 0xC4 что в десятичном варианте 196.

Любая программа для процессора tiny-lemoн состоит из байтов, которые содержат значения инструкций, а также любые другие данные. Что бы процессор выполнил программу она должна быть загружена в оперативную память tiny-lemoн. Процессор интерпретирует байты памяти начиная с самого первого (по адресу 0).

Ниже приведен пример небольшой программы. Больше примеров можно посмотреть в разделе «Примеры программ».

Пример программы

Адрес	0x	Код	Объяснения
00	7C	INA	Принимает значение с порта ввода по индексу 0, значение оказывается в регистре A, сохраняет это значение в байте по адресу 0x02, принимает еще одно значение с того же порта, значение попадет в регистр A, прибавляет к регистру A значение байта по адресу 0x02, отправляет результат в порт вывода по индексу 0 и останавливается на бесконечном выполнении инструкции HLT. То есть суммирует два байта со входа и отправляет результат на выход.
01	1E	STA__P	
02	00	0x00	
03	7C	INA	
04	C8	ADD_LP	
05	02	0x02	
06	7D	OUT	
07	7E	HLT	

СТАТИЧЕСКАЯ БИБЛИОТЕКА

В этом разделе будет рассмотрена реализация процессора `tiny-lemon` в виде статической библиотеки которая так и называется `tiny-lemon` и представляет из себя три файла «`tiny-lemon.h`», «`tiny-lemon.a`» и «`tiny-lemon.lib`».

Если у вас операционная система Windows вам нужен файл «`tiny-lemon.lib`», если у вас ОС Linux и подобные то вам нужен файл «`tiny-lemon.a`». Вне зависимости от операционной системы вам нужен файл «`tiny-lemon.h`». В итоге у вас будет два файла: файл заголовка «`.h`» и файл библиотеки «`.lib`» или «`.a`».

В файле заголовка находится немного кода на языке программирования C который нужно рассмотреть. Внимание: не редактируйте файл «`tiny-lemon.h`», внесение любых изменений приведет к ошибкам времени компиляции или ошибкам времени выполнения.

Заголовочный файл

Ниже показана и прокомментирована каждая строка файла «`tiny-lemon.h`» который является заголовочным файлом библиотеки.

```
#ifndef TINY_LEMON_HEADER_INCLUDED
```

Директива препроцессора `ifndef` (англ. *if not defined*), позволяет рассматривать текст из конструкции `ifndef-endif`, в случае если конкретный макрос не определен. Поскольку изначально макроса с названием `TINY_LEMON_HEADER_INCLUDED` нету, текст внутри `ifndef-endif` будет рассмотрен, то есть директивы будут проинтерпретированы, а код попадет на этап компиляции.

```
#define TINY_LEMON_HEADER_INCLUDED
```

Директива препроцессора `define`, позволяет определять значение макросов. В данном случае определяет существование `TINY_LEMON_HEADER_INCLUDED`. Что в комбинации с предыдущей строкой гарантирует что текст из конструкции `ifndef-endif` будет рассмотрен компилятором лишь один раз. Такая комбинация часто используется для файлов, которые вставляются (англ. *include*) несколько раз и при этом нет смысла компилировать их несколько раз или же в случаях когда множественное компилирование приводит к ошибке.

```
#ifdef __cplusplus
```

```
extern "C" {
```

```
#endif
```

Код «`extern "C" {}`» будет проигнорирован директивой `ifdef` (англ. *if defined*) если макрос «`__cplusplus`» не определен. Макрос `__cplusplus` как правило заранее определяется C++ компиляторами. Это позволяет отличить компиляцию C кода через C компилятор от компиляции через C++ компилятор. В данном случае для C++ компилятора добавляется подсказка что код идущий дальше стоит рассматривать как C код. В самом конце будет аналогичная конструкция `ifdef-endif` для добавления закрывающей скобки «`}`».

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

```
#include<stdint.h>
```

Директива препроцессора `include`, позволяет вставлять содержимое одних файлов внутрь других файлов. Это лишь вставка текста, поэтому не стоит рассматривать это как подключение библиотеки. В данном случае вставляется текст из файлов `«stdlib.h»` и `«stdint.h»`, это файлы стандартной библиотеки языка программирования C, из которых в `tiny-lemop` используются типы `uint8_t` и `size_t`.

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

Определяется макрос `TL_INTH_PTR_ADR` с текстом `«(0xFF)»`. Это адрес байта в памяти, где храниться указатель на подпрограмму обработчик прерываний.

```
typedef uint8_t TLbyte;
```

Определение типа `TLbyte` который аналогичен типу `uint8_t` (из файла `«stdint.h»`) и представляет собой целое беззнаковое число размеров в 8 бит, то есть 1 байт.

```
typedef size_t TLcounter;
```

Определение типа `TLcounter` как `size_t`. Тип `size_t` представляет собой целое беззнаковое число и который может содержать максимальный допустимый индекс.

```
typedef int TLbool;
```

Определение типа `TLbool` как `int`. `TLbool` используется как подсказка программисту что полученное значение будет либо 0, либо 1.

```
struct TLdata;
```

Декларация структуры `TLdata` представляющей все данные необходимые для работы виртуального процессора `tiny-lemop`. Это не определение типа, это лишь подсказка компилятору что этот тип будет определен в будущем и не стоит выдавать ошибку в случае его использования до его фактического определения.

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

Определение типа `TLin` как указателя на функцию, принимающую указатель на структуру `TLdata` и `TLbyte`, с типом возвращаемого значения – `TLbyte`.

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

Определение типа `TLout` как указателя на функцию, принимающую указатель на структуру `TLdata` и два `TLbyte`, с типом возвращаемого значения – `void`.

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

Определение типа `TLclock` как указателя на функцию, принимающую указатель на структуру `TLdata`, с типом возвращаемого значения – `void`.

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

Определение типа `TLhalt` как указателя на функцию, принимающую указатель на структуру `TLdata`, с типом возвращаемого значения – `void`.


```
typedef struct TLdata {
```

Определение структуры TLdata и назначение ей псевдонима TL. TLdata – это составной тип, то есть структура данных содержащая в себе под элементы. Эта структура предназначена для хранения всего состояния процессора tiny-lemon и его памяти. Использовать этот тип можно для объявления как глобальной, так и локальной переменной вашей программы. Все под элементы определяются внутри фигурных скобок, после которых указан псевдоним структуры, а именно TL.

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

Массив из 256 байт под названием «ram» представляет собой оперативную память для процессора tiny-lemon. Этот массив, как и все прочие элементы структуры доступны для получения и изменения значений в любой момент.

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

Массив из 256 байт под названием «stk» представляет собой память стека для процессора tiny-lemon.

```
    struct {
```

Объявление анонимной подструктуры, содержащей все регистры процессора. Анонимной означает что у этого типа нет названия, но есть определение, после которого, как правило, идет использование. После закрывающей фигурной скобки указано название переменной – reg.

```
        TLbyte A;
```

```
        TLbyte D;
```

```
        TLbyte F;
```

```
        TLbyte I;
```

```
        TLbyte K;
```

```
        TLbyte O;
```

```
        TLbyte P;
```

```
        TLbyte S;
```

Восемь регистров процессора tiny-lemon и каждый является одним байтом.

```
    } reg;
```

Вышеупомянутое название под-элемента.

```
    struct {
```

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

```
        TLin in;
```

```
        TLout out;
```

```
        TLclock clock;
```

```
        TLhalt halt;
```

Объявлены переменные `in`, `out`, `clock`, `halt` соответствующие ранее определенным типам `TLin`, `TLout`, `TLclock`, `TLhalt`. Виртуальный процессор `tiny-lemon` использует эти указатели на функции для реализации отправки различных сигналов.

```
} callback;
```

Все указатели на функции объединятся под одним названием – `callback`.

```
TLcounter clock_counter;
```

Объявление счетчика циклов процессора.

```
void* additional_data;
```

Объявление указателя на что угодно. Библиотека `tiny-lemon` никак не использует этот указатель. Если убрать эту строчку вы получите ошибку, что возможно покажется вам контринтуитивным, но на самом деле нет ничего странного, даже несмотря на то, что эта переменная не используется библиотекой, сама библиотека манипулирует памятью с расчетом на определенный размер структуры данных в байтах, если уменьшить размер – библиотека выйдет за границу предоставленной вами памяти. Вы можете использовать этот указатель для «прикрепления» дополнительных данных, которые нужны для реализации вашей виртуальной машины.

```
TLbyte arg_address;
```

В этой переменной храниться адрес байта, полученный из режима адресации.

```
TLbyte arg_value;
```

В этой переменной храниться значение, полученное из режима адресации.

```
} TL;
```

Вышеупомянутый псевдоним структуры `TLdata`.

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

Декларация функции `tlInit` принимающую указатель на экземпляр структуры `TLdata` и с типом возвращаемого значения `void`. Эта функция нужна для инициализации экземпляра структуры `TLdata`, так же это можно назвать настройкой по умолчанию. Функция записывает значение ноль во все регистры, обнуляет счетчик шагов, выставляет нули на байтах оперативной памяти и памяти стека, но главное выставляет указатели на «нулевые» функции обратного вызова. Нулевые они, потому что являются функциями, которые ничего не делают, и реализованы они внутри статической библиотеки. Это позволяет указать только те функции обратного вызова, которые вам действительно необходимы. Не забывайте использовать функцию `tlInit` при работе с библиотекой.

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

Декларация функции `tlClock` принимающую указатель на экземпляр структуры `TLdata` и с типом возвращаемого значения `void`. В предыдущей главе было сказано, что работа процесса делиться на шаги, так вот функция `tlClock` это программная реализация одного такого шага. Вы можете использовать эту функцию для реализации своего собственного цикла работы процессора, например, просто вызывать её внутри `while (1) { ... }`. Так же это позволяет иметь несколько процессоров `tiny-lemon` работающих «параллельно», то есть в теле цикла вызывать `tlClock` для каждого процессора.

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

Декларация функции `tlRun` принимающую указатель на экземпляр структуры `TLdata` и с типом возвращаемого значения `void`. Реализует бесконечный цикл из вызовов `tlClock`. Это самый простой способ получить бесконечно работающий процессор. Очевидно, что таким способом и без многопоточности не выйдет получить больше одного работающего процессора. Функцией `tlRun` не предоставляет никакого способа выйти из бесконечного цикла, так что решение этой проблемы ложиться на пользователя, например можно использовать функцию `exit` из «`stdlib.h`» заголовочного файла стандартной библиотеки языка программирования C.

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

Декларация функции `tlDoExternalInterrupt` принимающую указатель на экземпляр структуры `TLdata` и байт данных, а тип возвращаемого значения – `void`. Эта функция реализует механизм внешнего прерывания, байт данных при этом попадет в регистр A. Само прерывание будет гарантированным, и будет вызвано даже если флаг I, запрещающий внешние прерывания, равен единице. По этой причине, использовать данную функцию не рекомендуется и лучше использовать вариант ниже.

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

Декларация функции `tlTryExternalInterrupt` которая аналогична `tlDoExternalInterrupt`, но прерывание произойдет только если внешние прерывания не запрещены флагом I. Если прерывание удалось, то функция вернет один, в противном случае – ноль. Таким образом можно написать реализацию внешнего устройства так что бы то продолжало попытки сделать прерывание, до успешной попытки.

```
#ifdef __cplusplus
```

```
}
```

```
#endif
```

Добавление закрывающей скобки для конструкции `extern "C" { ... }`, естественно только в тех случаях когда используется C++ компилятор.

```
#endif
```

Последняя строка в заголовочном файле библиотеки. Директива `endif` закрывающая конструкцию `ifndef-endif` открытую в самом начале файла.

Как пользоваться?

В коде вашей программы должен быть объявлен как минимум один экземпляр структуры `TLdata`, это может быть локальная или глобальная переменная или же память, выделенная другими методами, например при помощи [алокатора памяти](#).

Объявление переменной типа `TLdata` будет выглядеть так:

```
TL mytl;
```

Сначала указываете тип `TLdata` или псевдоним `TL`, затем через пробел – имя «`mytl`», и в конце точка с запятой которая является знаком окончания утверждения (англ. *Statement*) и не является частью

имени переменной. Имя переменной может быть любым, главное, чтобы оно соответствовало правилам синтаксиса языка программирования C, то есть начиналось с буквы либо знака подчеркивания и содержало в себе только буквы, цифры и знаки подчеркивания. Рассматриваемые ниже примеры будут подразумевать что название переменной это «mytl». Взаимодействовать с переменной возможно только после её объявления, то есть объявление должно находиться выше, чем использование переменной, а также с учитывая область видимости.

Перед тем как что-либо делать с полученной переменной нужно сначала провести инициализацию. Делается это путем вызова функции `tlInit`. Вызвать функцию нужно только один раз для каждого экземпляра структуры `TLdata`.

Пример инициализации:

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

Обратите внимание на префиксный оператор «&» он нужен для того, чтобы получить указатель на переменную и передать его в функцию. Если не хотите каждый раз использовать этот оператор можно объявить переменную, хранящую указатель, например так:

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

В строке выше не только объявляется переменная `tl_ptr`, также происходит присвоение или другими словами – запись значения в переменную. Знак равно «=» это оператор присвоения. В данном случае переменной `tl_ptr` присваивается значение указателя на переменную `mytl`. То, что переменная `tl_ptr` является указателем можно понять по символу звезды «*» и указывает он на структуру `TLdata`.

Один раз получив указатель можно использовать его вместо конструкции «&mytl», к примеру:

```
tlInit(tl_ptr);
```

Помните, что вы можете называть переменные как угодно, но в книге дальше будет использоваться название `tl_ptr`.

Проведя инициализацию, можно приступить к загрузке программы для `tiny-lemon` в память. Нет никакой готовой библиотечной функции для этого действия, поэтому нужно реализовывать это самостоятельно. Например, присваивать значения по одному:

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

```
mytl.ram[1] = 0x7D;
```

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

Таким образом первым трём байтам памяти `tiny-lemon` будет присвоено значения `0x7C`, `0x7D` и `0x36` соответственно. Значения представлены в коде в шестнадцатеричном виде, это можно понять по префиксу «0x», это удобно для расшифровки инструкций. В предыдущем разделе расположена таблица инструкций, в которой можно найти любую инструкцию поделив шестнадцатеричный код на правую и левую цифры, затем используя их как X и Y координаты соответственно. К примеру, `0x7C` это инструкция `INA`. Оставшиеся две инструкции попробуйте расшифровать самостоятельно. Если хотите себя проверить эту небольшую программу из трех инструкций можно найти в разделе «примеры программ», а именно программа «Cat».

Очевидно, что для достаточно больших программ такой способ записи в память не подходит из-за своей громоздкости. Когда какое-либо действие повторяется много раз, в нашем случае запись значений в память, целесообразно использовать цикл.

```
int n = sizeof(arr) / sizeof(TLbyte);  
if (n > 256) n = 256;  
for (int i = 0; i < n; i++) {  
    mytl.ram[i] = arr[i];  
}
```

Предполагается, что у нас уже есть массив байт под названием `arr` в котором храниться программа для процессора `tiny-lemon`. Используя оператор `sizeof` можно вычислить размер массива и сохранить его в целочисленной переменной `n`. Если размер массива превышает размер памяти, переменная `n` будет урезана ровно под размер памяти. Затем начиная с первого элемента все элементы массива будут записаны в память `tiny-lemon`. К примеру, массив `arr` может содержать те же самые три байта:

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

Для удобства чтения кода, логического разделения программы на отдельные независимые модули, упрощения задачи дебага и многоразового использования одного и того же кода нужно создать функцию `load_program`. Такая функция будет выполнять только задачу загрузки программы в память `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];  
    }  
}
```

Использовать написанную функцию можно неограниченное количество раз. Вызов функции может выглядеть так:

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

Обратите внимание что в функцию `load_program` нужно передавать значения именно в таком порядке:

- 1) указатель на структуру `TLdata`;
- 2) массив или, что тоже самое, указатель на первый элемент массива;
- 3) количество элементов в массиве.

Так же вам возможно понадобится функция `fread_program` для загрузки программы не из массива, а из файлового потока. Для этой цели можно использовать `FILE*` из стандартной библиотеки «`stdio.h`» языка программирования C, а также функции для работы с `FILE*` из той же библиотеки.

```
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;
    }
}
```

Объяснение того, как работать с файловыми потоками выходит за рамки этой книги и рекомендуется ознакомиться с возможностями стандартной библиотеки «`stdio.h`» самостоятельно.

После того как программа тем или иным способом загружена в память `tiny-lemon` можно приступить к интерпретации этой программы. Как уже было сказано можно написать свой цикл процессора, например:

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

Где `is_run` это целочисленная глобальная переменная, которую можно изменить в любой момент времени. Изначально переменная имеет не нулевое значение, таким образом цикл процессора начнется и будет продолжаться пока переменной `is_run` не будет присвоено значение ноль.

Но самым простым решением будет использование функции `tlRun` что бы запустить бесконечный цикл процессора:

```
tlRun(tl_ptr);
```

Процессор будет работать, интерпретируя программу из памяти, но никакого взаимодействие с «внешним миром» не произойдет так как всё еще не были назначены функции обратного вызова. Для начала их нужно написать:

```
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);
}
```

Осталось только выполнить операцию присвоения для соответствующих указателей на функции:

```
mytl.callback.in = my_input;
mytl.callback.out = my_output;
mytl.callback.clock = my_clock_debug;
```

Процессор будет использовать функцию `my_input` для выполнения инструкции `INA`, значение, которое вернет функция попадет в регистр `A`, в данном случае это будет байт из стандартного потока ввода.

В свою очередь функция `my_output` будет использоваться инструкцией `OUT` что бы отправить данные на выход, в данном случае это стандартный поток вывода.

Функции `my_input` и `my_output` действуют одинаково вне зависимости от параметра `port` который указывает из какого устройства принимаются данные, а в какое – отправляются данные. Но это лишь пример, вы можете сделать разное поведение функций в зависимости от значения порта, используя разветвления `if-else` и `switch`.

Каждый шаг процессор будет вызывать функцию `my_clock_debug` которая выводит значение регистра `P`, таким образом можно легко проследить какие и в какой последовательности выполняются инструкции. Это удобно при поиске ошибок, но когда весь функционал протестирован можно закомментировать строку с присвоением, тем самым отключив этот функционал:

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

Если вы используете свой собственный цикл процессора и хотите иметь возможность из него выйти, то можете привязать выход из цикла к выполнению инструкции `HLT` реализовав, например такую функцию:

```
void my_halt(TL* tl_ptr) {
    is_run = 0;
}
```

И добавить еще одно присвоение:

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

Выполнение инструкции `HLT` приведет к вызову функции `my_halt`, и переменная `is_run` станет равна нулю. Если ваш цикл выглядит так:

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

То нулевое значение в переменной `is_run` приведет к выходу из цикла `while`.

Весь код из файла main.c может выглядеть 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;
}
```


Что бы из файла main.c получить исполняемый файл нужно использовать компилятор C, например GNU C Compiler или же gcc. Минимальная команда на ОС Linux будет выглядеть так:

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

А для ОС Windows так:

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

В общем виде это:

```
gcc <путь к файлу с кодом> <путь к файлу статической библиотеки tiny-lemon>
```

Это далеко не весь доступный функционал компилятора gcc, но для старта этого будет достаточно. При желании вы можете ознакомиться со всеми возможностями gcc самостоятельно.

При успешной компиляции кода, вы получите исполняемый файл – полноценную виртуальную машину на базе процессора tiny-lemon.

Если компиляция не была успешной, то компилятор выведет сообщение об ошибке и о том, где и что привело к ошибке времени компиляции.

Не стоит так же забывать об ошибках времени выполнения, в конкретно этом примере файла main.c программе нужен для работы файл data.bin содержащий программу для tiny-lemon и находящийся в той же папке, где находится исполняемый файл.

ПРИМЕРЫ ПРОГРАММ

Cat

Программа, которая отправляет на вывод всё что поступает на вход.

Адрес	0x	Код	Объяснение
00	7C	INA	Читает с порта ввода значение и отправляет его на порт вывода, затем делает регистр Р равным нулю, то есть процессор начнет выполнять программу сначала и так в бесконечном цикле.
01	7D	OUT	
02	36	ZRP	

Hello, World!

Классика в мире программирования – программа, выводящая текст «Hello, World!»

Адрес	0x	Код	Объяснение
00	16	LDA_P	Первыми двумя инструкциями загружает в регистр D значение 0x09 что совпадает с адресом текстового сообщения (см. адрес 09). Далее начинается «тело цикла». Читает байт по адресу из регистра D, то есть один символ сообщения. При этом обновляются флаги Z и N. Отправляет прочитанный символ в порт 0, подразумевая что в этот порт подключено устройство – терминал. Увеличивает значение регистра D на 1. И если ранее обновленный флаг Z равен 0 (False) возвращается в начало цикла, то есть адрес 0x03. Если прыжка не произошло, то программа остановиться на бесконечной интерпретации инструкции HLT. Таким образом чтение символов из памяти и отправка их на экран терминала будет происходить до того момента пока не будет встречен нулевой 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

Программа выводющая сама себя на экран. В данном случае не текст, а бинарные данные.

Адрес	0x	Код	Объяснение
00	12	LDA_D	Загружает значение байта по адресу из D (изначально D равен 0).
01	7D	OUT	Отправляет только что загруженное значение.
02	DC	ICD	Увеличивает регистр D на 1, тем самым указывая на следующий байт.
03	01	MVD	Загружает значение регистра D в регистр A.
04	B6	CMP_P	Обновляет флаги исходя из результата операции «A – 9», в данном случае важно то, что если A равен 9, то при вычитании получится 0.
05	09	+9	Если флаг Z не равен 0 возвращается к началу программы, в противном случае останавливается на инструкции HLT.
06	50	JFZ_LP	
07	00	+0	
08	7E	HLT	Магическое число 9 – это размер программы в байтах.

Последовательность Фибоначчи

Первые два числа последовательности Фибоначчи это 0 и 1, каждое следующее число это сумма двух предыдущих.

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

Написанием программы, которая вычисляет последовательность Фибоначчи можно доказать полноту по Тьюрингу.

Адрес	0x	Код	Объяснение
00	7D	OUT	Выводит значение регистра A, который изначально равен 0.
01	2F	XCB_LP	Меняет местами значения регистра A и байта по адресу 0x0A.
02	0A	+10	Назовем этот байт переменной X.
03	C8	ADD_LP	Прибавляет к регистру A значение переменной X.
04	0A	+10	Если значение не равно 121 вернуться к адресу 0x00, иначе остановиться.
05	B6	CMP_P	
06	79	+121	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.
07	50	JFZ_LP	
08	00	+0	
09	7E	HLT	
10	01	+1	

Машина истинности

Программа, которая, получив на входе 0 выведет ноль один раз, а получив 1 будет бесконечно выводить единицу на экран.

Адрес	0x	Код	Объяснение
00	7C	INA	Читает значение, обновляет флаги путем загрузки значения из регистра A в регистр A. В регистре A окажется значение 0 или 1 и соответственно флаг Z будет либо 0, либо 1. Если флаг Z равен 0 программа завершиться, если 1 тогда продолжит бесконечно выводить единицу.
01	08	LDA	
02	7D	OUT	
03	50	JFZ_LP	
04	02	+2	
05	7E	HLT	

Сумма 64-бит чисел

Перед тем как суммировать 64-бит числа нужно разместить их в память tiny-lemon. Конечно, можно вручную перевести числа из десятичной системы счисления в шестнадцатеричную, затем записать в программу значения байт (каждый байт это две шестнадцатеричной цифры). Но когда нужно будет поэкспериментировать с разными значениями, повторять этот процесс каждый раз медленно, поэтому лучше написать функцию:

```
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--;
    }
}
```

Обратите внимание что аргумент `adr` указывает на последний байт числа, а запись идет в обратном направлении, байт за байтом. В итоге, 64-бит число `0x1122334455667788` будет записано в память сохраняя порядок цифр, где `0x88` это значение байта, на который указывал `adr`:

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

Давайте запишем в память два числа:

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

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

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

Программа для процессора tiny-lemon вычислит сумму двух чисел X и Y так, что результат заменит собой число Y. В нашем случае сумма выглядит так:

1020304050607080900

+

1010101010101010101

=

2030405060708091001

То есть по адресу 0x80 будет находиться число 2030405060708091001.

Адрес	0x	Код	Объяснение
00	16	LDA_P	В регистр D загружается адрес числа Y.
01	80	\$80	
02	09	LDD	
03	16	LDA_P	В регистр K загружается адрес числа X.
04	90	\$90	
05	0C	LDK	
06	6F	JSR_LP	Вызывается подпрограмма по адресу 9 она вычисляет сумму чисел, адреса которых записаны в D и K, а результат заменит число по адресу из D. Затем процессор остановиться на инструкции HLT.
07	09	+9	
08	7E	HLT	
09	81	PSD	На стеке сохраняются значения регистров D и K, чтобы в конце вернуть изначальные значения.
10	84	PSK	
11	41	FFC	
12	16	LDA_P	В регистр A загружается значение 8, что является размером 64-бит числа в байтах, и соответственно количеством итераций цикла.
13	08	+8	
14	80	PSA	
15	12	LDA_D	Значение регистра A сохраняется на стеке. Значение байта по адресу из регистра D сохраняется в байт по адресу 20 который используется как аргумент инструкции ADC_P.
16	1F	STA_LP	
17	14	+20	
18	14	LDA_K	В регистр A загружается значение байта по адресу из регистра K. К регистру A прибавляется ранее загруженное значение, учитывая флаг переноса, и рассчитывая новый флаг переноса для следующей суммы. Результат заменяет собой байт по адресу из регистра D. Регистры D и K уменьшаются на один.
19	C9	ADC_P	
20	00	+0	
21	1A	STA_D	Со стека возвращается значение A сохраненное ранее. Значение в регистре A в комбинации с инструкцией LUP_LP обеспечивает повторение этого блока кода 8 раз. То есть каждый байт числа Y будет суммирован с соответствующим байтом числа X учитывая перенос, а результат заменит собой Y.
22	DD	DCD	
23	DF	DCK	
24	90	PPA	После цикла, возвращаются изначальные значения регистров D и K. В регистр P записывается значение со стека которое было оставлено инструкцией JSR_LP, необходимое для возврата из подпрограммы.
25	77	LUP_LP	
26	0E	+14	
27	94	PPK	
28	91	PPD	
29	96	PPP	

Что бы посмотреть правильный ли получился результат нам нужен способ прочитать число из памяти, например написав для этого специальную функцию:

```
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;  
}
```

После работы программы прочитаем число Y и выведем его на экран:

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