ПРОЦЕССОР IBN KIKKA 24

(программный эмулятор)

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Научная группа:

Аматэрасу-о-миками

Сергей Львович

Вера Алексеевна

Кошка Юко

Ивакава Ацуко

2024 年 11 月 16 日

Посвящается памяти Ады Лавлейс, Алана Тьюринга, Эмиля Поста и Кэтлин Бут

Содержание:

Предисловие

Если Вы читаете этот текст, дорогой пользователь, то у нас есть две новости, хорошая и плохая. Начнём с хорошей: этот процессор полон по Тьюрингу, то есть Вы сможете написать на нём всё, что хотите. А теперь плохая: этот процессор — трясина Тьюринга, *в которой можно сделать всё, но ничего интересного нельзя сделать просто.* Так почему же и зачем был создан этот процессор? Все известные человечеству электронные вычислительные устройства, реализованные на практике, в своей основе используют концепцию машины, описанной Аланом Тьюрингом в его работе «О вычислимых числах и их применении к проблеме разрешимости» (эта работа впервые опубликована на русском языке в первом источнике). Эта машина описывается, как устройство с бесконечной лентой и считывателем, который записывает и стирает значения в ячейках и перемещается между ячейками, согласно заранее предустановленным правилам. Независимо от Алана Тьюринга в 1936 году Эмиль Пост предложил свою машину (второй источник), которая эквивалента машине Тьюринга (далее МТ) в вопросе выяснения алгоритмической разрешимости той или иной проблемы. Эта машина более проста, чем МТ. Её «язык управления» описывается тем, что считыватель возможно сдвигать влево и вправо по бесконечной ленте, записывать и стирать метку в ячейке, на которую наведён считыватель, а также проверять содержимое ячейки, на которую наведён считыватель, чтобы если оно пусто перейти на одну строчку программы, а если не пусто, то есть метка есть, перейти на другую строчку программы. Это условный оператор. После каждого указанного действия указывается, какую строчку программы исполнять дальше. Эта машина так же детерминирована, как и МТ, однако существуют и недетерминированные МТ (третий источник). Они интересны тем, что могут на практике решить задачу быстрее, чем детерминированные (353-я страница третьего источника). Это объясняется тем, что недетерминированная МТ может за то же время работы детерминированной угадать и проверить сразу несколько решений тех или иных проблем (третий источник, страница 429), так как всякая детерминированная МТ — та же, что недетерминированная, но без возможности случайного перехода на каждом такте работы, то есть если детерминированная на основе состояния считывателя и символа в ячейке однозначно определяет, что будет сделано, куда будет сдвинут считыватель, и какое у него будет состояние, то недетерминированная задаёт не менее, чем две возможности для определения состояния считывателя того, что будет записано в ячейку.

Современные процессоры, которые, разумеется, позволяют смоделировать работу машины Тьюринга, работают детерминированно. Возможность определять случайным образом последующие действия напрямую в рамках формальной системы языка не предусматривается — случайность вводится опосредованно через вызов генератора псевдослучайных чисел. Использование условных операторов сравнения переменных со случайным значением позволяет определить случайное поведение программы, однако это не позволяет задать случайность для исполнения всего кода. Однако, что будет, если мы обоснуем свои вычисления и программы не только на детерминированных принципах, но и на стохастических? Очевидно, что имея возможность использовать псевдослучайные числа, мы можем симулировать работу недетерминированных МТ (и машин Поста тоже), а значит и более сложных вычислителей.

Но не только вероятность представляет интерес для проектирования вычислителей будущего. Данные и программа обычно разделены, то есть программа меняет данные, но данные не меняют программу. Существуют эволюционные алгоритмы, которые прогоняют через себя данные, проверяют то, что получилось на выходе, и если оно не соответствует определённым условиям, они, алгоритмы, подают свой выход на свой же вход, чтобы данные вновь изменились. Существует и другой подход: в программе изначально реализовано несколько возможностей, которые исполняются в зависимости от того, какие данные пришли на вход. Однако в самой своей основе, что данные, что программа, представляют собой числа. Это единство проявляется в метапрограммировании, когда программы пишут другие программы, однако даже тогда то, что пишется, определяет человек — человек во всех случаях является действующей причиной происходящего, предопределяя то, что должно произойти. Также стоит заметить, что цикличность программ не предусматривается на самом базовом уровне — циклы реализуются в рамках

формальной системы, описываемой языком программирования/языком управления определённого вычислителя.

Возможен иной подход — можно описать процессор/вычислитель, полный по Тьюрингу, который будет следовать следующим принципам:

1. Программа, управляющая вычислителем, безусловно циклическая и исполняется, пока не встретится оператор конца.
2. Переход по строкам программы может быть задан вероятностью.
3. Считыватель не только записывает и читает данные, но и обрабатывает их. Обработка связана с самоизменением состояний считывателя, основанных на данных, зафиксированных в определённых ячейках. При этом возможно задать влияние вероятности на это.
4. Лента конечной длины замкнута в том смысле, что при выходе за пределы количества ячеек в ней, считывание переходит на противоположную часть ленты.
5. Возможно исполнять части кода из любой точки программы.

И мы создали эмулятор процессора на C++, который дважды считывает ассемблерный код из файла: в первый раз он определяет, где начинаются блоки; во второй раз он построчно интерпретирует, то есть исполняет, код. Если есть оператор начала программы на какой-нибудь строчке, то исполнение начинается с неё, а не с нулевой. При достижении конца программы исполнение начинается с нулевой строчки, если не был исполнен оператор конца программы. Присутствует также оператор перехода на случайную строчку программы, что позволяет добавить случайность в алгоритм, погрузив его в хаос и бессмыслицу при непродуманном использовании этого оператора. Этот процессор реализует перечисленную пятёрку принципов.

Несмотря на то, что проведение даже базовых операций по типу сложения и умножения на этом процессоре представляет невероятно трудную задачу для программиста, сам этот процессор можно использовать для эмпирических проверок тех или иных гипотез, связанных со скоростями выполнения того или иного алгоритма. Также этот процессор позволяет на самом низком уровне абстракции описывать стохастические алгоритмы и системы, не привлекая высокие уровни абстракции.

1. Бог создал целые числа : математические открытия, изменившие историю. / Перевод с англ. О. С. Сажиной [и др.]. — М. : АСТ, 2022. — 804, [6] с. : ил. — (Мир Стивена Хокинга) — ISBN 978-5-17-113541-6.
2. Успенский Владимир Андреевич. Машина Поста / Гл. ред. физ.-мат. лит.. — 2-е изд., испр.. — М.: Наука, 1988. — 96 с. — (Популярные лекции по математике). — ISBN 5-02-013735-9.
3. Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман. Введение в теорию автоматов, языков и вычислений = Introduction to Automata Theory, Languages, and Computation. — М.: «Вильямс», 2002. — 528 с. — ISBN 0-201-44124-1.

Описание процессора: лента

Процессор состоит из двух элементов: считыватель и 256-битная лента, каждая ячейка которой может содержать в себе либо ноль, либо единицу. При этом лента процессора замкнута, то есть после 256-й ячейки следует нулевая, а перед нулевой располагается 256-я.

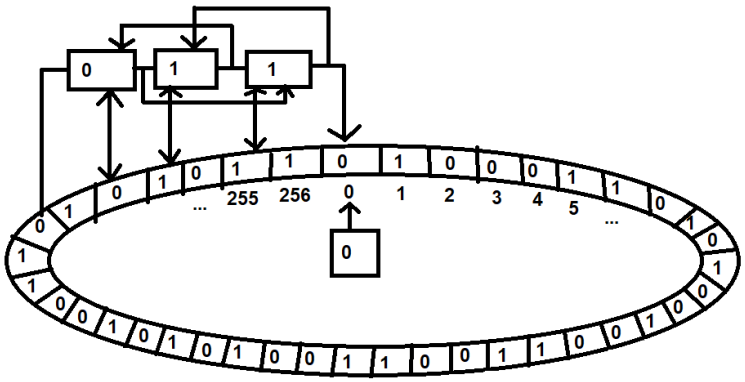

Схема процессора

Считыватель может двигаться по ленте влево и вправо. При достижении граничных значений (ноль и 256) считыватель, если продолжит свой путь, переместится на противоположный конец. Если сдвинуться влево от нуля, то адрес станет равным 256. Если с 256 сдвинуться вправо, то будет совершён переход на нулевой адрес. На этом описание ленты заканчивается.

Описание процессора: считыватель

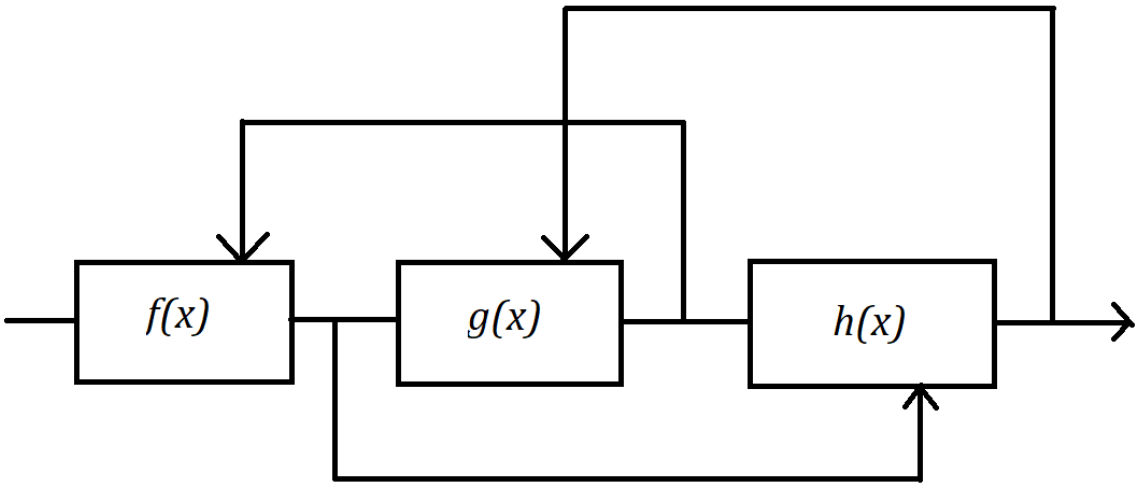

Схема считывателя

Считыватель процессора — это абстрактное вычислительное устройство, представленное на схеме выше представляет собой последовательное применение трёх функций к входным данным, а затем изменение этих функций друг другом на основе их выходных результатов.

У нас есть три логические функции: $f(x), g(x), h(x)$. Каждая из функций может быть, как тавтологией, так и отрицанием:

$$f(x) = x$$

$$f(x) = not(x)$$

Каждой из двух возможностей соответствует либо ноль, либо единица. Изменение одной функцией другой функцией можно записать, как применение изменяющей функции к изменяемой. Например, $f(x)$ изменяет $g(x)$ можно записать, как $f(g(x))$. Изменение же на основе выходных результатов возможно записать через применение к изменяемой функции такой функции, что какой соответствует выход изменяющей функции. Запишем в общем виде соответствия:

$$a \to q(x) = x$$

$$b \to q(x) = not(x)$$

$$a, b \in \{0, 1\}, a \neq b$$

Тогда применение некоей функции $f(x)$ к $g(x)$ на основе выходного результата $f(x)$ будет записываться следующим образом:

$$f(c) = d, c \wedge d \in \{0,1\}$$

$$d \to w(x) = (q(x) = x) \vee (q(x) = not(x))$$

$$f(c) = d, d(g(x)) = w(g(x))$$

c – входные данные в $f(x)$.

Применение $f(x)$ к $g(x)$ на основе выходных данных $f(x)$ будет записывать в дальнейшем так:

$$f * g$$

Тогда можно следующим образом описать работу вычислительного устройства со схемы выше следующим образом:

x – входные данные.

y – выходные данные.

$$y = h(g(f(x)))$$

$$h * g$$

$$g * f$$

$$f * h$$

В случае, если $x = y$, мы получаем замыкание, а если они не равны, мы на вход всегда подаём одно и то же. Устройство прекращает свою работу, если состояния функций, которые можно описать одним числом (либо 0, либо 1) и входное значение одинаковые.

Рассмотрим ситуацию, когда на вход всегда подаём ноль, единица соответствует отрицанию, а изначально функции описываются строкой 010. Мы получим следующее:

Текущее состояние функций: 010

Выходы: 0 1 1

Текущее состояние функций: 100

Выходы: 1 1 1

Текущее состояние функций: 011

Выходы: 0 1 0

Текущее состояние функций: 111

Выходы: 1 0 1

Следующим состоянием будет 100, а на входе будет ноль, потому цикл завершается.

Однако это можно не ограничиваться таким поведением этой системы (режим работы «01»).

Можно сначала проводить изменения функций (тогда их выходы полагаются на первой итерации равными их изначальным состояниям), а только потом проход данных через них (режим работы «10»).

Либо же можно пропустить данные через первую функцию, первая изменит третью, затем данные пройдут через вторую функцию, вторая изменит первую, а потом данные пройдут через третью функцию, которая затем изменит вторую (режим работы «11»).

Можно также сделать соответствие отрицание единице или нулю, основанным на вероятности.

Режим работы «00» просто копирует значение из одной ячейки в другую.

Ассемблер Kikka: общие принципы

Описанный процессор (его эмулятор) в отличие от обыкновенных процессоров, которые можно назвать интерпретаторами машинных кодов, принимает инструкции не в виде шестнадцатеричных данных, и даже не в виде двоичных кодов — IBN KIKKA 24 напрямую интерпретирует ассемблерный язык, описанный ниже.

Программы на ассемблере Kikka (яп. 菊花 — цветок хризантемы; далее просто ассемблер) состоят из конечного числа строчек, имеющих следующий синтаксис:

оператор операнд1 операнд2

Оператор и операнды отделены друг от друга одним пробелом. Если в начале строки стоит пробел, то строка игнорируется. Язык чувствителен к регистру.

В общем случае операнды — это адреса/номера ячеек на ленте, то есть числа от нуля (0) до двухсот пятидесяти шести (256). В случае, если число будет указано минус один (-1), то значением операнда полагается значение, сохранённое в отдельной ячейке (не на ленте), которое равно заранее определённому адресу ячейки (по умолчанию это значение равно нулю, далее мы будем назвать это «взятием/записью значения по адресу»). Если же оператором являются block, label, do и to, то единственным операндом является названием блока или метки, которое может быть представлено строкой произвольной длины, не содержащей в себе пробела и управляющих символов.

Процессор исполняет программу построчно, выполняя действие, указанное на каждой строчке. При этом сами программы безусловно цикличны, то есть, когда будет выполнена последняя строчка, программа начнёт выполнятся с нулевой строчки. И так до тех пор, пока по каким-то причинам не будет исполнена строчка с оператором конца программы (owari, яп. 終わり — конец). Возможно так же при запуске программы начинать исполнение с указанной строчки через использование оператора hajimaru (яп. 始まる — начало). В случае, если операндом является адрес ячейки, то минус единица в данном контексте означает взятие номера ячейки из переменной, в которой хранится адрес, на который указывает считыватель (то самое «взятие по адресу»).

Ассемблер Kikka: таблица команд/операторов

После рассмотрения общих принципов работы мы приведём таблицу всех операторов ассемблера с примером их применения и объяснением, что они делают.

Оператор	Пример применения	Действие
hajimaru (始まる — начало)	hajimaru	При запуске программа начинает исполняться со строчки, на которой расположен этот оператор (затем этот оператор игнорируется даже в случае использования безусловной цикличности). Если этого оператора в программе нет, то программа начинает исполняться с нулевой строчки.
owari (終わり — конец)	owari	При его исполнении программа заканчивается. Необходим для того, чтобы программа закончилась.
<	<	Уменьшает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен нулю, то после применения он будет равен 256.
>	>	Увеличивает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен 256, то после применения он будет равен нулю.
addr	addr 249	Делает адрес, равный указанному значению. Если значение указано больше 256, либо меньше нуля, то значением станет число, появившееся после учёта цикличности ленты.
inaddr	inaddr	Запрашивает ввод адреса у пользователя. В остальном делает всё то же, что и addr.
goto	goto 17	Переход на указанную строчку программы Учитывайте, что нумерация начинается с нуля и не следует переходить на строчки, которых нет — если у вас в программе последняя строчка седьмая (нумерация учитывает нулевую), то бессмысленно пытаться перейти на, например, тринадцатую, которой нет.

addrwokaku (addr を書く — написать адрес)	addrwokaku	Вывод в консоль, чему равен адрес.
mojiwokaku (文字を書く — написать символ)	mojiwokaku	Вывод в консоль символа, чей ASCII-код равен значению адреса.
kyouki (狂気 — безумие)	kyouki	Переход на случайную строчку программы. Действует так же, как и goto, но строка, на которую делает переход выбирается случайно (не беспокойтесь — выход за количество строчек не произойдёт).
		Пустая строка, которая игнорируется при исполнении (не используйте пробел в начале строки для создания строки комментария!).
;	; 温子は狂気なです。	Комментарий.
loop	loop	Адрес становится равным числу количества пройденных программой циклов. Изначально программа прошла ноль циклов, однако, каждый раз доходя до последней строчки, не встретив owari, она возвращается в начало (нулевая строка), при этом счётчик циклов увеличивается на единицу. Оператор loop приравнивает значение адреса количеству пройдённых циклов с учётом замкнутости ленты.
f1	f1 147	Указывает, из какой ячейки берётся значение для первой функции в считывателе.
f2	f2 37	То же, что и f1, но для второй функции.
f3	f3 54	То же, что и f1, но для третьей функции.
prob	prob 27	Вероятность того, что в считывателе единица будет соответствовать отрицанию. При вводе любого целого числа (за исключением минус единицы) берёт остаток от деления его абсолютного значения на 101 и записывается в качестве вероятности. При значении, равным -1, берёт остаток от деления значения адреса на 101 и записывает в качестве

		вероятности. Лучше указывайте число от нуля до сотни.
cycle	cycle 217	Берёт значение из ячейки с указанным номером (минус один — по адресу) и, в зависимости от полученного значения, определяет, будет ли считыватель циклично подавать себе на вход выходное значение, пока не достигнет устойчивого состояния. Ноль — нет, единица — да.
conf1	conf1 3	Берёт один разряд (ноль/один) из указанной ячейки для составления конфигурации (левый знак — «x1»). Если операнд — минус единица, то берёт по адресу.
conf2	conf2 56	То же, что и conf1, но для правого знака («1x»).
conf	conf 7	Эквивалентно случаю, когда для conf1 и conf2 указана одна и та же ячейка. Если в ячейке лежит единица, то конфигурация будет «11», иначе — «00».
kaku (書く — написать)	kaku 23 75	Выводит в консоль значения ячеек ленты из указанного диапазона. Настоятельно рекомендуем следить за тем, чтобы второй операнд был больше первого. Минус один (-1) означает взятие значения по адресу. Если оба операнда равны, то выведена будет только одна ячейка.
bunkiten (分岐点 — поворотный момент)	bunkiten 0 76	Сравнивает значения в двух ячейках. Если они равны, выполняет следующую строчку программы, а если нет — пропускает/перескакивает следующую строчку. Если оба операнда равны, то следующая строчка не будет пропущена. Минус единица (-1) указывает ячейку по адресу.
henkamono (変化物 — изменитель)	henkamono 88 44	Считыватель берёт значение из ячейки, чей номер передан вторым операндом, обрабатывает его, согласно своим настройкам, и записывает результат в ячейку, чей номер передан первым операндом. Минус единица

		означает ячейку по адресу. **Обратите внимание, что этот оператор делает адрес, равным первому операнду!**
ugoku (動く — переместить)	ugoku 93 127	Копирует значение из ячейки, чей номер передан вторым операндом в ячейку, чей номер передан вторым операндом. Минус единица означает взятие значения по адресу. В случае равенства операндов ничего не изменится (ячейка скопируется сама в себя). Оператор является синтаксическим сахаром и может быть реализован через применение считывателя с функциями 000 и вероятностью 100 (либо применение считывателя в режиме работы «00»). **Обратите внимание, что этот оператор делает адрес, равным первому операнду!**
label	label tape7	Объявление метки с именем, переданным первым операндом, которое не должно содержать пробелов и управляющих символов. Запоминает номер строчки, на которой объявлена метка.
to	to tape7	Переход на метку с именем, переданным первым операндом. В совокупности с bunkiten позволяет реализовывать циклические конструкции.
block	block q1	Начало блока с именем, переданным первым операндом (ограничения на имя такие же, как для label). Всё, что идёт после block до break игнорируется, если не был исполнен оператор do, вызывающий исполнение блока.
break	break q1	Конец блока. Всё, что после break, уже не игнорируется и исполняется, как обычно.
do	do q1	Вызов исполнения блока с именем, переданным первым операндом. После исполнения кода, расположенного между block и break, процессор возвращается к строке, с которой был совершён вызов

		блока (т.е. исполнен do) и продолжает исполнять программу со следующей строчки. Примечательно то, что блок может быть вызван из любой точки программы, даже до того, как он был объявлен, так как исполнение происходит на втором чтении, после того, как в первом чтении блоки уже были размечены.
zero (ゼロ)	zero 13	Записать ноль в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.
hitotsu (一つ)	hitotsu 7	Записать единицу в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.

Объяснения и рекомендации

Адрес (addr, −1) — номер ячейки, на которую указывает считыватель (変化物). При применении считывателя адрес изменяется на номер ячейки, в которую считыватель запишет обработанные данные. Перемещение (動く), являясь синтаксическим сахаром, реализует применение считывателя к двум ячейкам с состояниями функций 000 (каждая цифра с отдельной ячейки) и вероятностью 100 и потому также изменяет значение адреса на номер ячейки, переданной первым операндом? Однако значение адреса не меняет.

Рекомендации и факты:

- Файл с одной пустой строчкой приведёт к тому, что процессор начнёт и тут же завершит программу, однако файл с двумя пустыми строчками заставит процессор войти в бесконечный цикл, в котором ничего не будет происходить. Сочетание клавиш CTRL+C позволяет выйти из бесконечного цикла, остановив программу (это прерывание позволяет остановить исполнение программы, когда угодно).
- Оператор to позволяет перейти только на ту метку, строка с объявлением которой была исполнена ранее, то есть оператор label может быть расположен, например ниже, чем to, но, если по каким-то причинам он был исполнен, то to позволит перейти на метку. Не рекомендуем нарушать принцип использования этих двух операторов.
- Возможно описывать вложенные блоки. Например следующая программа выведет в консоль ноль:

  ```
  block q2

  block q1

  kaku 0 0

  break q1

  break q2

  do q1

  owari
  ```

  Однако мы рекомендуем очень внимательно подходить к таким конструкциям и по мере сил избегать их. В приведённом примере при вызове q2 вместо q1 программа просто завершится, ничего не выведя в консоль, так как вложенный блок не был вызван.

  Следующая программа выведет ноль дважды:

  ```
  block q2

  block q1

  kaku 0 0

  break q1

  do q1

  break q2

  do q2
  ```

owari

Это связано с технической реализацией логики блоков, которую можно понять, просмотрев исходный, код поставляемый вместе с этой инструкцией и exe-файлом эмулятора процессора. Не рекомендуем вызывать блоки, которых нет, а также использовать block и break не по назначению и поодиночке.

- Поведение процессора может быть очень и очень сложным. Один только вывод всех возможных состояний считывателя (変化物) занял 86 листов а4. И это без учёта вероятности, которая позволяет перескакивать между разными цепочками состояний, позволяя порождать, сколь угодно длинные цепочки — главное избегать повторений состояний функций и входного значения в них.
- Рекомендуем следить за значениями операндов, так как не везде реализована проверка на допустимость введённых/указанных значений. В идеале в коде должны встречаться только минус единица и числа от нуля до 256. Учитывайте, что в ленте всего 256 ячеек!
- Если в программе встретится оператор, не указанный в таблице операторов, приведённой выше, то строчка будет проигнорирована с выводом в консоль сообщения о неизвестном операторе с указанием номера строки, на которой он встретился.
- Помните: **по умолчанию все настройки, ссылки и адрес инициализированы, как нули** — вам придётся в каждой программе указывать, откуда берутся значения для настроек считывателя!

Это все рекомендации, которые первыми приходят в голову. Всего описать невозможно, так как тогда мы получим противоречие (теорема Гёделя). Общая рекомендация может быть описана следующим списком:

1. Соблюдайте описанные правила.
2. Избегайте вложенных меточных, циклических и блочных конструкций.
3. Старайтесь писать код, не создавая неоднозначных ситуаций, в которых Вы не уверены.
4. Все операнды должны принадлежать следующему списку:
    - Минус единица (-1).
    - Числа от нуля до 256 ($0 \leq n \leq 256$).
5. Избегайте использования оператора goto, если это возможно.
6. Разрабатывая алгоритмы/программы, сначала запишите всё на листочке и проверьте код в уме. Грамотно спроектированный код гораздо проще читать и отлаживать!

Эти рекомендации обусловлены опытом разработки языка Haikugo (俳句語) и написания на нём программ. Следуя этим рекомендациям, Вы сможете писать код, который легко читать и отлаживать.

Доказательство полноты по Тьюрингу и перспективы применения

Этот раздел посвящён чистой теории — если Вас интересует только практический аспект, то Вы можете перейти сразу к следующему разделу. Доказательство полноты по Тьюрингу описываемого процессора и его ассемблера основывается на следующих фактах:

- Любой вычислитель, на котором возможно реализовать машину Тьюринга, является полным по Тьюрингу, то есть на нём возможно реализовать любую вычислимую функцию.
- Машина поста эквивалента машине Тьюринга, являясь одной из тавтологий алгоритма.
- Вычислитель, на котором возможно реализовать другой вычислитель, который полон по Тьюрингу, сам является полным по Тьюрингу.

Основываясь на высказанных фактах можно заключить, что возможность реализовать операторы машины Поста в терминах ассемблера процессора будет доказательством полноты процессора по Тьюрингу. Таблица перехода от машины Поста к ассемблеру приведена ниже ($j \wedge j1 \wedge j2 \in N : \forall n \in N \longrightarrow 0 \leq n < \infty$).

Машина Поста	Ассемблер процессора	Действие
V j	hitotsu -1 goto j	Поставить метку, перейти к j-й строке программы.
X j	zero -1 goto j	Стереть метку, перейти к j-й строке.
← j	< goto j	Сдвинуться влево, перейти к j-й строке.
→ j	> goto j	Сдвинуться вправо, перейти к j-й строке.
? j1; j2	zero 0 bunkiten -1 0 goto j1 goto j2	Если в ячейке нет метки, то перейти к j1-й строке программы, иначе перейти к j2-й строке
!	owari	Конец программы («стоп», останов).

Из таблицы видно, что полнота по Тьюрингу гарантируется лишь небольшим подмножеством ассемблера, то есть теоретически возможно описать любую вычислимую функцию, даже не прибегая к изменителю и вероятностям.

Перспективы использования процессора по прямому назначению весьма туманны, однако можно рассматривать исключительно теоретические основы. Например, возможно проводить эмпирические тесты тех или иных алгоритмов, проводить проверки гипотез, моделировать сложные системы, связанные с вероятностями и самоопределением внутренних процессов. Ассемблер позволяет, не прибегая к высоким уровням абстракции, связать программу и данные так, чтобы сама программа и данные взаимно изменяли друг друга, образуя неразрывное целое. Использование стохастических возможностей считывателя и оператора случайного перехода по строкам программы позволяет задать неопределённое поведение процессора, которое спрогнозировать невозможно. Для вычислений эти возможности не подходят, чего нельзя сказать о простейших стохастических алгоритмических моделях. Наверное, возможно развить теорию стохастических алгоритмов, и тогда этот процессор можно использовать в качестве готовой среды для проверки и формализации положений это гипотетической теории.

В конце концов Вы можете развивать этот процессор, модифицируя исходный код, добавляя новые операторы и возможности. Этот процессор может стать основой для целого семейства архитектур.

Возможно затронуть также и темпоралогические аспекты. Программу в процессе её исполнения можно рассматривать в качестве замкнутой временной линии, тогда путь по программе (последовательность исполнения строк) описывается мировой линией в двумерном пространстве-

времени, где пространственное измерение — программа, а временное — порядок исполнения её строк. Тогда возможны точки схождения и расхождения этих мировых линий. Так, например, все строки с оператором bunkiten будут точками дивергенции и конвергенции (в зависимости от того, какие последовательности исполнения строк мы рассматриваем). То же верно и для строк с оператором kyouki. Это теорию можно развить.

Использование

Данный раздел посвящён практике. Если Вы читаете этот текст, то вы, вероятно, открыли архив IBN-KIKKA-24.zip, в котором содержатся следующие файлы:

- kikka.exe
- ascii.kikka
- ascii2.kikka
- kikka.xml
- Руководство IBN KIKKA 24.pdf
- kikka.cpp

Исполняемый файл (kikka.exe) — это сам процессор, то есть интерпретатор, который принимает .kikka-файлы. Вообще процессор принимает любой текстовый файл, в котором записан код, однако рекомендуется использовать только .kikka-файлы, так как в новых версиях процессора и других его архитектурах расширение файла может оказаться значимым.

Два .kikka-файла (ascii.kikka и ascii2.kikka) — две программы, выводящие на экран все 256 ASCII символов. Эти программы написаны двумя разными способами для демонстрации того, что безусловная цикличность позволяет сокращать количество операторов в программе при сохранении функциональности.

kikka.xml — это файл с описанием подсветки синтаксиса для редактора кода Notepad++. Вы можете открыть .kikka-файлы в Notepad++, после чего перейти в раздел «Синтаксисы» и в выпадающем списке выбрать «Пользовательский синтаксис», после чего импортировать синтаксис из kikka.xml. Тогда у Вас появится подсветка синтаксиса.

.pdf-файл — это файл с руководством, который Вы сейчас читаете.

kikka.cpp — исходный код процессора на языке C++.

Сами эти файлы располагаются в папке IBN-KIKKA-24, которую следует распаковать в удобное для Вас место.

Если вы просто нажмёте на kikka.exe, то увидите мерцание консоли. Правильный запуск и эксплуатации процессора может быть представлена следующим порядком действий (после ввода каждой команды нужно нажимать Enter, чтобы она выполнилась):

1. Откройте консоль. Это можно сделать либо через ярлыки, либо нажав сочетание WIN+R (клавиша с символом Windows и английская R). В открывшемся окне введите cmd и нажмите Enter.
2. Перейдите в директорию с kikka.exe, написав команду:

   cd C:\путь к файлу

   Например, если вы распаковали папку в корень диска C, то команда будет следующей:

   cd C:\IBN-KIKKA-24
3. Проверьте, что всё работает введя и исполнив:

   kikka

   Если Вы всё сделали правильно, то консоль напишет:

Usage: kikka <filename>

IBN-KIKKA-24 - First Temporal Processor

Version 1.0.0 Release

4. После этого Вы можете запустить какую-нибудь из программу. Для этого Вам нужно написать kikka и через пробел название файла с расширением. Синтаксис такой:

   kikka название.kikka

   Например, запуск ascii.kikka вместе с результатом выглядит так:

   C:\IBN-KIKKA-24>kikka ascii.kikka

   IBN-KIKKA-24 - First Temporal Processor

   Version 1.0.0 Release

   --
   !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopq

   rstuvwxyz{|}~□

   C:\IBN-KIKKA-24>

   Первые символы, закрытые знаками вопроса (здесь просто линия, у вас будут знаки вопроса в квадратиках), являются непечатными управляющими символами, потому они отобразились так странно.

Теперь вы можете создавать текстовые файлы, переименовывать их, заменяя расширение txt на kikka, и писать в них код, который Вы уже знаете, как запускать.

Теперь Вы знаете всё, что нужно для успешного использования процессора. Если Вы смогли запустить ascii.kikka и настроить подсветку синтаксиса в Notepad++, то Вы показываете уровень компьютерной грамотности, более высокий, чем у большинства пользователей, а это означает, что перед Вами открываются бескрайние дали — Вы можете придумывать разные стохастические алгоритмы, разные просто обычные алгоритмы. Вы можете, если знаете C++, создавать свои языки на основе ассемблера этого процессора, либо вообще создавать свои архитектуры.

Если же всё, описанное выше, покажется Вам бесполезным, то вспомните, что люди создают картины, пишут музыку… убивают друг друга… Создание очередного странного вычислителя — занятие явно более полезное, чем кошмар, который устраивает человечество в войнах.

Желаем удачи в прогулке по миру странных алгоритмов, что возможно реализовать на этом процессоре!