Визитка

АЛЕКСЕЙ ВТОРНИКОВ, программист. Разработчик ПО для банков и страховых компаний (хотя не отказывается от интересных задач в других областях). Основной «недостаток» – предпочитает командную строку любым IDE

Стек: скрытые таланты и возможности

Программирование богато разнообразными структурами данных, однако даже среди них выделяется своими возможностями стек — неотъемлемая часть «внутренностей» любого компилятора или интерпретатора. Стек может выполнять не только технические функции, но и выступать в качестве самостоятельной вычислительной среды

> Работа предстоит мелкая. Хуже вышивания
>
> **Евгений Шварц «Убить дракона»**

Эта статья посвящена описанию небольшого, но вместе с тем достаточно мощного стекового компьютера, пригодного для реализации широкого круга алгоритмов. Внешне программа для стекового компьютера напоминает программу, составленную на ассемблере. Не будучи, однако, привязанным к определенному типу процессора, стековый компьютер не зависит от особенностей его архитектуры (порой причудливой и труднообъяснимой) и значительно проще обычных ассемблеров.

Система команд стекового компьютера проста и логична: фактически все, что требуется новичку, – это освоить способы манипулирования данными в стеке и привыкнуть к использованию стека, как к разновидности памяти. Замечательной особенностью стекового компьютера является его расширяемость – новые возможности добавляются путем весьма простого изменения исходных кодов его интерпретатора.

Первая часть статьи посвящена неформальному изложению теории и описанию архитектуры стекового компьютера. Вторая часть содержит описание реализации стекового компьютера. На сайте журнала можно найти исходные коды интерпретатора и примеры нескольких программ.

Часть 1. Немного теории

Для начала я напомню некоторые определения (которые, хочется верить, окажутся читателю знакомыми), расскажу об используемой в статье системе обозначений и немного затрону вопросы реализации стеков.

Дональд Кнут в своем «Искусстве программирования» определяет стек так:

«Стек – это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка».

Проще говоря, стек – это память, доступ к элементам которой организован по принципу «последним зашел, первым вышел» (или LIFO – от Last In First Out). В теории формальных языков и методов компиляции обычно используется термин «магазин» (push-down) или «магазинный список» – по аналогии с магазином огнестрельного оружия.

Стек представляет собой упорядоченную структуру данных, состоящую из отдельных элементов. Элементы служат местом хранения данных (а также, возможно, дополнительной информации, например, указателей на другие структуры данных).

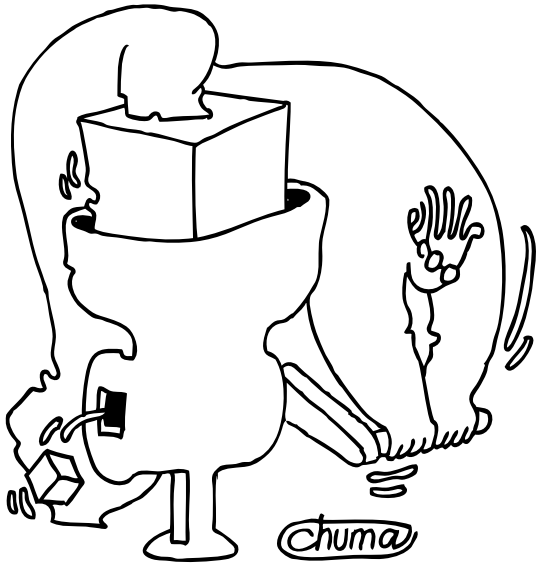
Данные, хранящиеся в элементах стека, могут быть любыми: числами, символами или другими структурами; это определятся назначением стека и способом его реализации.

У стека имеется дно – место, начиная с которого в нем накапливаются элементы; в пустом стеке нет ни одного элемента. Новый элемент всегда помещается (проталкивается, заносится) на место, указываемое вершиной стека; ранее находившиеся в стеке данные утапливаются в глубь стека. Вместо термина «вершина» обычно используется термин «голова».

Удаление (выталкивание, снятие) элемента из стека производится в обратном порядке, т.е. прежде всего из элемента, на который указывает голова стека. Количество элементов, находящихся в стеке в данный момент времени, называется его глубиной; очевидно, что глубина пустого стека равна 0. Глубина стека может быть ограничена наперед заданной величиной или быть неограниченной.

Наглядный и знакомый всем пример стека – детская пирамидка, состоящая из разноцветных кружков, нанизанных на центральный стержень. Очевидно, что новый кружок можно нанизать только поверх ранее нанизанных. Удалять кружки можно только в порядке, обратном тому, которым они были нанизаны. Снять произвольный кружок из середины пирамидки невозможно (разве что разломав ее). В пустой пирамидке нет ни одного кружка; в целиком заполненную пирамидку невозможно ничего добавить.

Текущее содержимое стека называется его состоянием. Для отображения состояний стека и переходов стека из од-



Замечательной особенностью стекового компьютера является его расширяемость

них состояний в другие применяются стековые диаграммы. При проталкивании в стек нового элемента глубина стека увеличивается, и в этом случае часто говорят, что стек «растет»; соответственно при выталкивании из стека элемента глубина стека уменьшается, и стек «сокращается».

Я буду отображать состояние стека горизонтальными диаграммами (так, как это принято в языке программирования Forth). Дно стека (его я обозначу символом «⊦») изображается слева. Стек по мере проталкивания в него новых элементов растет слева направо.

Вот как, например, будет выглядеть изначально пустой стек после проталкивания в него трех чисел 1, 1024 и -99 (именно в таком порядке):

```
⊦ 1 1024 -99
```

Очевидно, что число 1 находится в стеке глубже всех; чуть выше него – число 1024 и, наконец, последнее число -99.

После выталкивания из стека двух элементов стек станет таким:

```
⊦ 1
```

Несколько диаграмм (если, конечно, позволяет место) можно объединять в «цепочку»:

```
⊦ 1 1024 -99 → ⊦ 1 → ⊦ 1 20 0 -2 144
```

Базовые операции

Стандартно над стеком определены следующие базовые операции:

- проталкивание нового элемента в вершину стека push (<элемент>);
- выталкивание элемента из вершины стека pop ();
- проверка стека на пустоту empty ().

Для удобства использования список базовых операций при реализации стека обычно расширяют, например, такими:

- обмен местами двух верхних элементов стека;
- копирование в вершину стека элемента, расположенного в глубине стека;
- дублирование верхнего элемента стека и т.д.

Способы реализации стека

Выше я говорил, что максимальная глубина стека может быть величиной как ограниченной, так и неограниченной. Понятно, что последнее на практике недостижимо, поскольку память реальных компьютеров, сколь бы велика она ни была, конечна: даже самый большой стек рано или поздно заполнится, и вся выделенная ему память будет занята. Такая ситуация называется переполнением стека (overflow). С другой стороны, возможна и противоположная ситуация опустошения стека (underflow) – т.е. попытка выборки из пустого стека.

При реализации и использовании стека обе эти ситуации желательно контролировать; в противном случае поведение программы легко станет непредсказуемым.

Существует два основных способа реализации стека: на основе массива и на основе связанного списка. Разумеется, второй способ более универсален, но в описанном во второй части статьи интерпретаторе стекового компьютера я буду использовать более простой способ реализации на основе массива.

Концепцию стека можно неформально описать так: «сохранить набор некоторых значений в определенном порядке по мере их поступления». «Порядок поступления» вводит в концепцию стека понятие времени: стек позволяет сделать различимыми понятия «до» и «после» или «раньше» и «позже».

Это принципиально отличает стек от обычной памяти с произвольным доступом, которая не обеспечивает строго хронологического порядка записи и считывания данных: в самом деле, память с произвольным доступом обеспечивает доступ к любой ячейке памяти в любой момент, в то время как стек предписывает записывать и считывать данные точно определенным образом. Проще говоря, стек имеет единственную точку входа – вершину, в то время как память с произвольным доступом – множество таких точек.

Стек как модель вычислений

Обычные и всем, надеюсь, знакомые области применения стека включают в себя задачи трансляции языков програм-

мирования (разбор арифметических выражений, анализ баланса скобочных структур), а также задачи поддержки среды исполнения при активации программных модулей (стеки возвратов, передача параметров в функции и процедуры, выделение памяти для локальных переменных).

Эти вопросы являются предметом отдельного рассмотрения, и в этой статье я их не буду касаться; заинтересованный читатель может обратиться к списку литературы в конце статьи.

Я рассмотрю стек в иной «ипостаси» – как модель вычислений или, проще говоря, как модель воображаемого (программно эмулируемого) компьютера. Чтобы не вдаваться в ненужные подробности, я покажу для начала, как стек может использоваться для выполнения арифметических операций.

В обычных языках программирования (таких, например, как C, Java или Pascal) предусмотрен «механизм» переменных: именованных областей оперативной памяти, предназначенных для хранения данных. Вот как могла бы выглядеть в программе на Java или C операция сложения:

```
int a = 5, b = 10;
int c = a + b;
```

При использовании стека нет необходимости определять переменные, отводить им память и размещать в памяти соответствующие значения – достаточно одного только стека. Пусть начальное состояние стека таково:

```
⊦ 10 5
```

Чтобы найти сумму двух верхних элементов стека нужно:

- вытолкнуть их из стека;
- сложить;
- результат сложения протолкнуть обратно в стек.

После этого состояние стека изменится:

```
⊦ 15
```

Видно, что сумма двух чисел (15) заняла «место» на вершине стека.

Говоря выше о том, что достаточно одного только стека, я несколько упростил ситуацию: память (а, следовательно, и переменные), подобная той, что используется в C, Java или Pascal, в определенных ситуациях бывает все-таки необходима. Не буду, однако, забегать вперед...

Аналогично сложению можно вычитать числа:

```
⊦ 10 5 → ⊦ 5
```

Обращаю внимание, что по сложившейся практике верхнее число является вычитаемым, а число под ним – уменьшаемым.

В более привычной записи рассмотренные выше арифметические операции можно переписать соответственно в виде «a b +» и «a b -». Способ записи и выполнения арифметических операций, при котором сначала записываются операнды, а после них операция, называется постфиксным. Некоторая непривычность постфиксной записи компенсируется тем, что в ней все операции имеют одинаковый приоритет и позволяют исключить скобки.

Надеюсь, что принципы выполнения арифметических операций с операндами, расположенными в стеке, совершенно ясны.

Что делать, если порядок чисел в стеке не тот, что необходим; например, в последнем примере нужно вычесть 10 из 5? Понятно, что надо каким-то образом переставить два верхних элемента стека. Для манипуляций с расположением элементов в стеке служит целый набор так называемых стековых операций. Например, для перестановки двух верхних элементов стека имеется операция SWAP:

```
⊦ 10 5 → ⊦ 5 10
```

Теперь результат вычитания будет равен -5.

Другая важная стековая операция DUP позволяет копировать вершину стека:

```
⊦ 10 5 → ⊦ 10 5 5
```

Набор таких операций может быть достаточно большим и ограничиваться лишь фантазией программиста, реализующего стек. Ниже, в разделе, посвященном архитектуре и системе команд стекового компьютера, я приведу общепринятый «канонический» набор.

Таким образом, может быть реализована привычная по другим языкам программирования арифметика. Вот, например, как можно реализовать вычисление по формуле 2*a + 2*b для a=17 и b=5:

```
⊦ 17 5 → (DUP)
⊦ 17 5 5 → (+)
⊦ 17 10 → (SWAP)
⊦ 10 17 → (DUP)
⊦ 10 17 17 → (+)
⊦ 10 34 → (+)
⊦ 44
```

Здесь для наглядности я выписал несколько стековых диаграмм одну под другой; в скобках указаны операции, применяемые к текущему содержимому стека. Таким образом, стек оказывается весьма мощной структурой и, похоже, вполне оправдывает эпитет «модель вычислений». Но подождите, то ли еще будет...

Язык программирования, чтобы не выглядеть как обычный калькулятор, должен располагать некоторыми структурами управления, к числу которых разумно отнести по меньшей мере условный оператор и оператор цикла. Как эти все требования можно воплотить при использовании стека?

Решения этих вопросов, разумеется, существуют, хотя выглядят они достаточно непривычно. Начну с условного оператора. В Basic-подобных языках программирования условный оператор имеет следующий синтаксис:

```
if (true) goto label
```

Иными словами, условный оператор – это оператор описания перехода: если проверяемое условие истинно, то переход осуществляется к метке label, если ложно, то выполняется часть программы непосредственно после условного оператора. Вот именно это мне и нужно. Теперь вернусь к стеку. Пусть состояние стека таково:

```
⊦ 10 5604
```

Верхнее число в стеке (т.е. 5604) – это адрес перехода, т.е. адрес оперативной памяти, по которому необходимо передать управление, если число под ним (т.е. 10) отвечает

некоторому условию. Это второе число может быть, как нетрудно догадаться, либо положительным, либо нулем, либо, наконец, отрицательным. По установившейся традиции это число называется флагом. Следовательно, и возможных условий перехода также три – по числу возможных значений флагов. Соответственно в стековом компьютере предусмотрены три условных перехода – по флагу больше 0, по флагу, равному 0, и по флагу меньше 0. И, разумеется, нужен безусловный переход, т.е. переход вне зависимости от значений каких-либо флагов. А имея в распоряжении переходы, легко реализовать и циклы (в точности так, как это делают, программируя на ассемблерах).

Искушенный читатель, возможно, заметил, что, говоря о том, что число возможных условных переходов равно трем, я упростил ситуацию. Ведь возможны условные переходы и по иным условиям, например, «больше или равно», «меньше или равно» или «не равно». Но понятно, что эти условные переходы легко реализуются путем комбинирования трех основных.

Помните, говоря об арифметических операциях над содержимым стека, я упомянул, что одного стека может оказаться недостаточно – нужен еще и доступ к оперативной памяти. Причины этого понять нетрудно – хранить все данные в стеке все-таки накладно. Для этого потребуется стек довольно большой емкости, но с этим еще как-то можно управиться. Сложнее другое: стек, это структура данных, работа которой посвящена строгой дисциплине – «последним зашел, первым вышел». Это не так страшно, если необходимые значения хранятся в элементе, на который указывает вершина стека, или неподалеку от него. А если мне необходимо значение, хранящееся в 10-м, 256-м или 3067-м элементе от вершины? Как до него добраться? Можно, конечно, предусмотреть соответствующие операции (именно так иногда и поступают), но это плохое решение: во-первых, нарушается концепция стека как упорядоченной структуры данных, и, во-вторых, расположение нужного элемента в стеке постоянно меняется – сейчас он был 256-м, а через некоторое время уже 117-м. Так что оперативная память все-таки нужна. В следующем разделе особенности использования оперативной памяти в стековом компьютере будут подробно описаны.

Стековый компьютер, архитектуру которого я опишу в следующем разделе, в некоторых отношениях напоминает так называемый P-код, предложенный Никлаусом Виртом для реализации языка программирования Pascal. Спустя 20 с лишним лет Джеймс Гослинг из Sun Microsystems обратился к идее P-кода для реализации Java Virtual Machine (JVM). Разумеется, JVM отличается от канонического P-кода Никлауса Вирта. Но основные идеи, по существу, одни и те же. Реализация языков программирования на основе стекового компьютера получила широкое распространение (в частности, тот же самый подход использовала Microsoft при реализации CIL – промежуточного языка для платформы .NET). Так что стековый компьютер – это в каком-то смысле классика, а настоящая классика никогда не устаревает.

Архитектура стекового компьютера

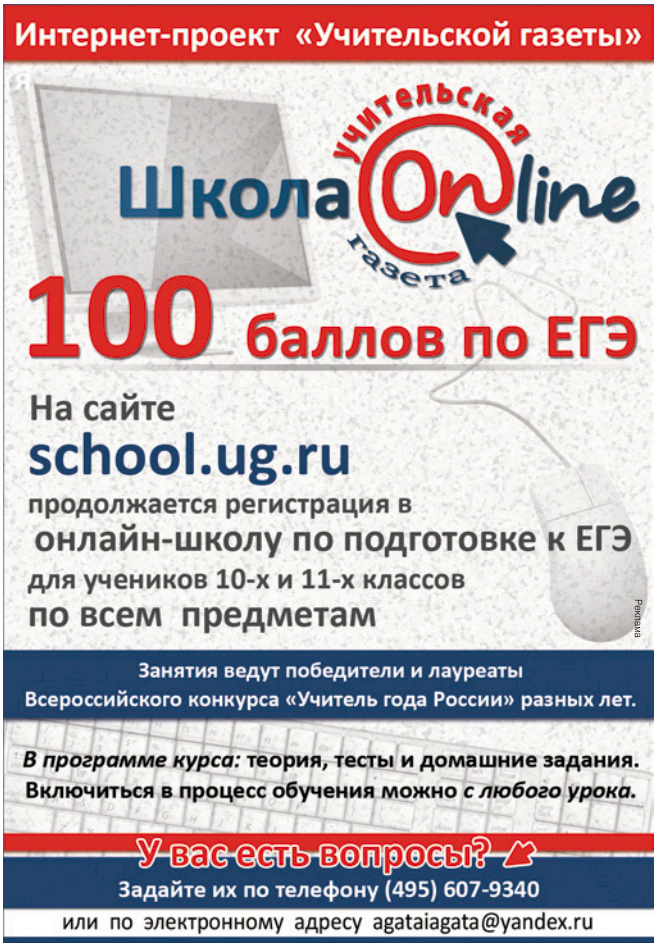
Этот раздел, возможно, покажется наиболее сложным во всей статье, поэтому я призываю читателя проявить терпение и внимание. Нам предстоит разобраться, как устроен стековый компьютер, и начать следует с оперативной памяти и памяти для стеков (ибо у нас будет не один, а целых два стека).

В дальнейшем для краткости оперативную (или основную) память я буду называть просто памятью.

Прежде всего сразу же отмечу, что стековый компьютер может хранить и обрабатывать только один тип данных – целые числа. Вспоминая, сколько типов данных поддерживается современными языками программирования (а также то, что программист может вводить свои типы данных), это может показаться сильным ограничением. Однако беспокоиться нечего. Во-первых, эта проблема разрешима, и заинтересованный читатель сможет ею заняться самостоятельно. Во-вторых, это, вообще говоря, вовсе и не проблема – это вопрос представления информации, и полное его решение читатель сможет найти в математике, точнее, в теории вычислимых (рекурсивных) функций.

Теория вычислимых функций опирается на идею кодирования, т.е. представления информации натуральными (неотрицательными целыми) числами. В частности, широко используется так называемая геделевская нумерация, введенная в 30-х годах прошлого века выдающимся австрийским математиком и логиком Куртом Геделем для доказательства знаменитых теорем о неполноте формальной арифметики. За подробностями (кстати, очень интересными, хотя и весьма сложными) я рекомендую обратиться к специальной литературе.

Очевидно, что стековому компьютеру необходим стек, в котором будут храниться данные для арифметических



вычислений, а также флаги и адреса памяти. Обычно этот стек называется стеком данных. Он может быть совсем небольшим – обычно нескольких десятков элементов вполне достаточно.

Кроме того, мне понадобится стек возвратов; в стеке возвратов будут храниться адреса возвратов из подпрограмм. Стек возвратов работает так же, как и в обычных языках программирования. Емкость стека возвратов обычно невелика (большой стек возвратов может потребоваться в основном для рекурсивных программ).

Оба стека функционируют идентично и независимо друг от друга: у каждого из них свое дно и своя вершина. Для указания на вершины стека данных и стека возвратов понадобятся два указателя, которые я обозначу соответственно как SP и RP (от data Stack Pointer и Return stack Pointer).

Наконец, стековому компьютеру необходима память, аналогичная той, что используется во всех компьютерах, т.е. память произвольного доступа. В этой памяти будут находиться программы и данные, которые нецелесообразно размещать в стеке данных. Ячейки памяти адресуются посредством указателя PC (от Program Counter). Объем памяти для наших целей может быть совсем небольшим – не более нескольких десятков килобайт.

Элементами стеков данных и возврата так же, как и памяти, являются целые числа. В зависимости от компьютера размер целого может быть 8, 16, 32 или 64 бит, но для рассматриваемого стекового компьютера это не имеет значения. Подробности реализации легко извлечь из исходного кода программы, эмулирующей работу стекового компьютера.

Теперь я рассмотрю систему команд стекового компьютера. Она состоит из операций, разбитых на несколько групп.

Арифметические операции

Арифметических операций всего три, и они очень просты:

Операция ADD – сложение, выталкивает из стека данных два верхних элемента, складывает их и проталкивает в стек значение их суммы:

```
⊦ a b → ⊦ a+b
```

Операция SUB – вычитание, выталкивает из стека данных два верхних элемента, вычитает значение на вершине стека из значения элемента под вершиной стека и проталкивает в стек значение их разности:

```
⊦ a b → ⊦ a-b
```

Операция NEG – изменение знака, меняет знак самого верхнего элемента в стеке данных на противоположный:

```
⊦ a → ⊦ -a
```

Легко заметить, что среди арифметических операций нет операций умножения, деления, модуля и им подобных. Во-первых, их очень просто добавить и реализовать самостоятельно (как именно я расскажу в свое время). Во-вторых, эти операции фактически не нужны – они легко моделируются существующими.

Обращаю внимание на существенную особенность использования арифметических операций. Что если я попытаюсь применить операции ADD или SUB к пустому стеку или стеку, содержащему только один элемент? Ясно, что это ошибка, и программист обязан ее предусмотреть и предотвратить в своих программах.

Операции манипулирования стеком данных

Рассмотрим эти операции подробнее:

DUP – копирование элемента на вершине стека, применяется в случае, если необходимо сохранить значение верхнего элемента стека данных для последующих целей:

```
⊦ a b → ⊦ a b b
```

DROP – удаление элемента на вершине стека, используется для очистки стека данных от «лишних» данных:

```
⊦ a b → ⊦ a
```

Имейте в виду, что фактически никакие данные из вершины стека не удаляются; меняется лишь значение указателя стека данных SP. Это относится и ко всем другим операциям, оперирующим элементами стека.

SWAP – перестановка местами двух верхних элементов стека. Часто элементы в стеке данных находятся не в том порядке, который требуется:

```
⊦ a b → ⊦ b a
```

OVER – копирование второго элемента стека, используется для обеспечения правильного порядка элементов в стеке данных:

```
⊦ a b → ⊦ a b a
```

Часто в набор операций манипулирования стеком данных добавляют операции циклической перестановки элементов на вершине стека и доступа к элементам в глубине стека. Мне, вообще говоря, эти операции не нужны, но позже читатель без труда сможет сам их добавить.

Следует иметь в виду одну существенную особенность использования стековых операций. Что если я попытаюсь применить операции DUP или DROP к пустому стеку? Ясно, что это ошибка, и возможность ее появления необходимо контролировать в своих программах. А что произойдет, если я попытаюсь применить операции SWAP или OVER к пустому стеку или к стеку, содержащему только один элемент? Опять же ничего хорошего: это ошибка, и ответственность за нее лежит на программисте.

Операции работы с памятью

Операций работы с памятью всего две: первая копирует в стек данные из памяти, вторая копирует данные из стека в память. Опишу их по порядку.

LOAD – чтение из памяти, выталкивает из стека данных значение, интерпретирует его как адрес памяти и записывает значение, хранящееся по этому адресу в стек данных:

```
⊦ a addr → ⊦ a value
```

Здесь value – содержимое ячейки памяти с адресом addr. Содержимое ячейки памяти остается неизменным. Значение, первоначально находившееся в стеке данных под адресом, остается без изменений.

SAVE – запись в память. Эта операция несколько сложнее, чем операция LOAD, т.к. для нее необходимо ука-

зать как адрес памяти, в которую производится запись, так и собственно записываемое значение:

```
| a value addr → | a
```

Значение, находящееся в стеке данных под его вершиной (value), записывается по адресу (addr) памяти. То, что хранилось в этой ячейке до операции SAVE, разумеется, теряется.

Операции передачи управления

Операции передачи управления позволяют организовать в программах для стекового компьютера условные операторы и циклы (аналогично тому, как это делается в ассемблерах). Таких операций всего четыре.

BR – безусловный переход, позволяет передать управление по заданному адресу в памяти без проверки каких бы то ни было условий. Откуда же операция берет этот адрес? Да все из того же стека данных:

```
| a b addr → | a b
```

Операция выталкивает с вершины стека данных число, интерпретирует его как адрес памяти и передает на него управление (т.е. программа продолжает выполняться с нового адреса). Технически передача управления выглядит как присваивание программному счетчику PC нового значения:

```
PC = addr
```

BRM – переход если минус, этой операции необходимы два числа на стеке данных – флаг и адрес перехода:

```
| a flag addr → | a
```

Если значение flag отрицательно, то управление передается на адрес addr, в противном случае программа выполняется так, как если бы ничего не произошло.

Следующие две операции совершенно аналогичны операции BRM, за исключением условия перехода: BRZ передаст управление по адресу addr, если значение флага равно 0; BRP передаст управление по адресу addr, если значение флага больше 0:

BRZ – переход, если нуль:

```
| a flag addr → | a
```

BRM – переход, если плюс:

```
| a flag addr → | a
```

Вот, собственно, и все, что касается операций передачи управления.

Обращаю внимание на то, что независимо от того, была ли выполнена передача управления по флагу или нет, из стека данных выталкиваются оба числа.

Переход к подпрограмме

Операция перехода к подпрограмме очень похожа на операцию безусловного перехода, но с одним важным исключением – в стеке возвратов запоминается адрес следующей команды (т.е. адрес программного счетчика следующей операции в программе). Итак:

CALL – переход к подпрограмме:

```
| a b addr → | a b
(стек возвратов) | pc+1
```

Из стековых диаграмм видно, что в вершину стека возвратов проталкивается адрес возврата. Вызовы подпрограмм могут вкладываться друг в друга произвольным образом, и всякий раз адрес следующей операции (т.е. команды, стоящей в программе за операцией CALL) запоминается в стеке возвратов.

RET – возврат из подпрограммы:

```
(стек возвратов) | pc+1 → |
```

При выполнении операции RET значение с вершины стека возвратов присваивается программному счетчику PC, что означает фактически безусловную передачу управления на операцию, стоящую непосредственно за соответствующей операцией CALL.

Подпрограммы в стековом компьютере могут быть рекурсивными и вызывать сами себя. Глубина вызовов подпрограмм ограничивается емкостью стека возвратов.

Нерассмотренными остались несколько операций стекового компьютера. Ниже следует их краткое описание (их семантику внимательный читатель легко извлечет из исходного кода интерпретатора стекового компьютера):

HALT – прекращение выполнения программы. наличие этой операции в исходном коде программы обязательно;

IN – ввод данных с консоли на вершину стека данных;

OUT – вывод данных на вершине стека на консоль;

OUTS – аналогично OUT, но в виде символа;

NOP – нет операции;

LPC – загрузить в стек данных текущее значение счетчика размещения PC; эта возможность позволяет динамически формировать в программе переходы, и такой возможностью следует пользоваться крайне аккуратно;

LSP – загрузить в стек данных текущее значение указателя стека SP (фактически это текущая глубина данных); операция позволяет программисту организовать проверку стека данных на пустоту. Для пустого стека операция LSP загрузит в стек данных 0.

***

В следующей части я расскажу о реализации интерпретатора стекового компьютера, архитектура которого была только что описана. EOF