Копирование Бита – Простейшая Вычислительная Машина

Олег Мазонка
2009
mazonka@gmail.com

Абстракт. Статья представляет доказательство, что операции ссылки и копирования бита достаточные для полного по Тюрингу вычисления. Представлен язык программирования состоящий из всего лишь одной инструкции - копирования бита. Этот язык достаточно мощный что бы исполнять такие программы как “Hello, World!” с итерациями по массиву и вычисления факториалов. Идея в том, что вычисления производятся без логических операций *AND* и *OR*. В Интернете есть компилятор и эмулятор этого языка с примерами программ.

Введение

В поиске языков с минимальным набором инструкций было изобретено несколько императивных языков с одной лишь инструкцией (One Instruction Set Computer, OISC) [1]. Например, Subleq [2] определен как язык, работающий на бесконечном массиве ячеек памяти с инструкцией, которая имеет 3 операнда. Процессор читает из памяти 3 последовательные ячейки ***A*** ***B*** ***C***, вычитает значение ячейки адресованной ячейкой ***A*** из значения ячейки адресованной ячейкой ***B*** и помещает результат в ту же ячейку адресованную ячейкой ***B***. Если результат меньше или равен нулю, выполнение переходит по адресу ***C*** и процессор читает следующую инструкцию оттуда. В противном случае следующие 3 операнда читаются из памяти. Доказано что этот язык полный по Тюрингу. Существуют несколько вариантов этого языка, но они одинаковы в своей сущности. Был написан компилятор компилирующий программу в код Subleq из простого C-образного языка [3].

Хотя языки OISC имеют одну инструкцию, эта инструкция, как правило, выполняет несколько манипуляций и вычислений. Таким образом возникает вопрос: какой язык имеет самую простую инструкцию и возможно ли построить язык с еще более простой инструкцией?

Другой интересный вопрос относится к логическим операциям. Общеизвестно, что любые классические вычисления обычно используют битовые логические операции такие как *AND*, *OR*, *XOR*, и *NOT*. Эти операции – ни полный набор, ни минимальный набор необходимый для вычисления. *OR* и *XOR* могут легко быть выражены через *AND* и *NOT*, и наоборот. Однако, обычно считается, что для настоящих вычислений нужны по крайней мере операции типа *AND* или *OR*. Невозможно скомбинировать только *OR* и *XOR* операции для очистки одного бита. Поэтому они неспособны выполнять классические вычисления. Следовательно,

вопрос: возможны ли программируемые вычисления без использования логических операций типа *AND* или *OR*?

1. Ссылка как Вычислительная Операция

Удивительно, что *OR* и *XOR* обратимые операции могут производить необратимый результат, если их использовать в комбинации с ссылкой. В следующей таблице:

000	001	010	011	100	101	110	111
100	011	011	111	110	100	010	101

верхняя строчка определяет три начальных бита. Нижняя строчка показывает те же биты с одним инвертированным (применена операция *NOT*). Бит, который инвертирован, определяется всеми тремя как индекс равный бинарному выражению взятому по модулю 3. Как видно, два начальных состояния (001 и 010) создают тот же результат (011), что делает всю операцию необратимой.

В другом примере

0000	0001	0010	0011	0100	0101	0110
0000	0001	1010	1011	1100	0001	...

в нижней строчке один бит изменен по формуле

$$(\ p[A]\ XOR\ p[B]\ ) \rightarrow p[A],$$

где ***A*** значение первых двух битов, ***B*** значение последних двух битов, и ***p[]***бит взятый по индексу.

Обратимые операции применяются к битам, но ссылка на биты превращает весь процесс в необратимый. Даже если кажется что этот процесс может выполнить вычисления, им трудно воспользоваться для преднамеренных запрограммированных вычислений.

2. Язык Копирования Бита

Оказывается, если пойти другим путем: скомбинировать необратимую операцию копирование бита с ссылкой, то программируемые вычисления становятся возможными.

Инструкция, копирующая бит, всегда очищает один бит. На первый взгляд кажется что полное количество информации в системе будет уменьшатся. Но это не так. Например, в процессе: *делай всегда (**a→b**, **c→a**, **b→c**)*, где ***a***, ***b***, и ***c*** биты и стрелка означает копирование, биты циркулируют неограниченно долго. Это может быть названо равномерным, но не стационарным состоянием. Что бы получить что-то более интересное, придадим смысл набору бит как адрес какого-то из этих битов.

Определим императивный язык, абстрактная машина которого работает на одностороннем бесконечном массиве памяти, состоящей из адресованных битов. Биты сгруппированы в слова определенного размера – ячейки памяти. Например, 8-битовые слова:

Memory	01010101	00001111	10101010	11001100	00110011	...
Address	0	8	16	24	32	40

Каждая инструкция состоит из трех операндов: ***A B C***, где каждый операнд это одно слово. Инструкция копирует бит по адресу ***A*** в бит по адресу ***B***, затем процесс переходит по адресу ***C***. Операнд ***C*** читается после того как копирование бита закончено. Это позволяет инструкции быть самомодифицируемой (несмотря на то что только один бит может быт изменен самой инструкцией).

Поскольку каждое слово представляет адрес бита в памяти, точная интерпретация адреса может быт оставлена неопределенной без ограничения самой идеи. Однако, в настоящей реализации она предполагается little-endian – младший бит первый.

Другая неопределенная до реализации деталь это группировка битов памяти в слова – физическая: все слова выровнены по адресу размера ячейки памяти, или логическая: слова группируются начиная с текущего адреса. Во втором случае операнду ***C*** позволено иметь любой адрес, а не только кратный размеру ячейки памяти.

Такая одиночная инструкция не делает больше чем просто копирование бита из одного места в другое, и в то же время она достаточна что бы язык был способен выполнять запрограммированную последовательность операций. Абстрактная машина очевидно делает нечто большее чем копирование бита. А именно, она ссылается на бит и переносит процесс на другой адрес выполнения. Однако эта работа не обязательно требует операций *AND* и *OR*, и выполняется вне модели. Это означает, что она может быть эмулирована тем же самым процессом копирования бита.

3. Язык Ассемблера

Для упрощения представления инструкций языка допустим следующее обозначение ассемблера. Каждое слово обозначается как **L:*V'x***, где **L** необязательная метка, служащая для адресации этой ячейки памяти, ***V*** – значение этой ячейки памяти, ***x*** – необязательное битовое смещение внутри слова (ячейки памяти). Каждая инструкция пишется с новой строчки. Например:

```
A'0 B'1 A
A:18 B:7 0
```

Здесь две инструкции. Первая инструкция копирует младший бит ячейки ***A*** (значение 18) во второй бит ячейки ***B*** (значение 7), затем переходит по адресу ***A***, которым является адрес следующей инструкции. После того как первая инструкция выполнена, значение ячейки ***B*** изменено на 5. Например, в 8-битовом представлении эти две инструкции будут выглядеть как

```
24 33 24 18 7 0.
```

Если битовое смещение опущено, оно предполагается равным нулю. Так что, ***A*** то же что и ***A'0***.

Если операнд ***C*** отсутствует то его предполагаемое значение это адрес следующей ячейки, то есть:

```
A B
```

то же что и

```
A B C
C: ...
```

и то же что

```
A B ?
```

Вопросительный знак (**?**) – это адрес следующей ячейки памяти, или другими словами, адрес первого бита в последующем слове. Пусть (**n?**) обозначает кратное размеру ячейки смещение от текущей позиции. Так что (**0?**) значит адрес текущей позиции; (**1?**) то же что и (**?**) и адрес следующей ячейки; (**2?**) адрес следующей через одну; и (**-2?**) адрес ячейки стоящей перед предыдущей. Например, инструкция

```
A B -2?
```

та же что и

```
C:A B C
```

и является бесконечным циклом (предполагая что ***C*** не модифицируется), поскольку после того как бит скопирован контроль передается по адресу ячейки ***C:A***, которая есть началом инструкции. Стоит напомнить что ***A*** – это значение и ***C*** – это адрес ячейки ***C:A***.

Что бы выполнить программу, представленную в виде текста ассемблера, необходима среда выполнения. Поэтому требуется два шага: 1) компиляция текста в бинарный код в виде битового массива; 2) выполнение инструкций представленных битовым массивом на абстрактной машине. Программа называющаяся ассемблер выполняет первый шаг, и эмулятор может выполнить второй.

4. Макрокоманды

Что бы сделать описание программы короче и более читабельным определим механизм макроподстановки как представлено в следующем примере:

```
.copy A B
...
.def copy X Y
X'0 Y'0
X'1 Y'1
...
```

```
X'w Y'w
.end
```

Первая строчка это макрокоманда, которая заменяется телом макроопределения начинающегося с ".def" и заканчивающегося на ".end". Имя после ".def" становится именем макроса и все последующие имена формальными аргументами к макросу. После макроподстановки код превращается в:

```
A'0 B'0
A'1 B'1
...
A'w B'w
```

Здесь ***w*** индекс старшего бита. Он равен размеру ячейки памяти минус один. Пусть ***W*** размер слова, тогда ***W***=***w***+1.

Два других полезных макроопределения – это сдвиг и вращение. shiftL сдвигает биты в ячейке памяти на 1 от младшего к старшему, и младший бит устанавливается в ноль. Эта операция соответствует арифметическому умножению на 2, или оператору "<<=1" в языке программирования C.

```
.def shiftL X : ZERO
X'(w-1) X'w
X'(w-2) X'(w-1)
...
X'1 X'2
X'0 X'1
ZERO X
.end
```

ZERO определено как ZERO:0. Это имя появляется после двоеточия в конце списка аргументов макроопределения, для обозначения что это имя определено вне макроопределения. Необходимость этого в том, что ассемблер пытается разрешить все имена внутри тела макроопределения или связать их с формальными аргументами. Нужно отметить, что последняя инструкция копирует только младший бит ячейки ZERO в младший бит ячейки ***X***.

shiftR то же что и shiftL но работает в противоположном направлении, и соответствует целочисленному делению на 2, или оператору ">>=1" языка C.

Макросы вращения похожи на макросы сдвига, за исключением того что они копируют последний выпадающий бит обратно в первый. Они могут быть определены через макроопределения сдвига:

```
.def rollR X : TMP
X TMP
.shiftR X
TMP X'w
.end

.def rollL X : TMP
X'w TMP
.shiftL X
TMP X
.end
```

Ячейка памяти `TMP` служит для временного сохранения бита и определена во внешней библиотеке.

Макросы копирования, сдвига и вращения полезны, но им не хватает логики для выполнения полезных вычислений.

5. Условный Переход

Рассмотрим следующий код:

```
.def jump01 A b
A'b 2?'k
0 J'0
A'b 2?'k
1 J'1
A'b 2?'k
2 J'2
...
A'b 2?'k
(w-2) J'(w-2)
A'b 2?'k
(w-1) J'(w-1)
A'b 2?'k
w J'w J:0
.end
```

Смещение ***k*** определено как 2^k=***W***. Поскольку слово это адрес бита в памяти, то ***k*** бит соответствуют смещению внутри слова. Остальные биты слова определяют адрес ячейки памяти. Например, для 32-битного слова, ***k*** будет 5, потому что изменения в шестом бите и выше изменяют адрес ячейки памяти, но не смещения внутри ячейки. Отметим, что если ***k***=5, то запись по смещению ***k*** изменяет шестой бит.

Первая строчка копирует бит **b** ячейки памяти ***A*** в ***k***-тый бит первого операнда следующей инструкции. После ее выполнения первый операнд следующей инструкции равен нулю или ***W***. Следующая инструкция копирует значение первого бита ячейки по адресу либо ноль либо ***W*** в ячейку помеченную ***J*** – которая является последней ячейкой памяти в этом списке инструкций, и которая будет содержать адрес, куда перейдет контроль после того как последняя инструкция будет исполнена. Последующая строчка `[(A'b 2?'k)(1 J'1)]` копирует второй бит в ячейку ***J*** , и так далее.

Когда последний бит скопирован, ячейка ***J*** содержит то же значение как и ячейка по адресу 0 (первое слово в памяти) или ячейка по адресу ***W*** (второе слово в памяти) в зависимости от того значение ***A'*****b** было 0 или 1.

Обозначая первые две ячейки памяти в программе специальными, в том смысле что они могут быть использованы только для этой операции, возможно написать универсальный тест одного бита.

```
Z0:0 Z1:0
.def test A b B0 B1 : Z0 Z1
.copy L0 Z0
```

```
        .copy L1 Z1
        .jump01 A b
        L0:B0 L1:B1
        .end
```

Этот код определяет макрос, который тестирует бит ***b*** ячейки памяти ***A*** и переходит по адресу либо ***B0***, либо ***B1*** в зависимости бит 0 или 1 соответственно.

Тестирование бита – ключевое требование для вычислений более высокого уровня описанных ниже. В большинстве случаев тестирование бита заключается в проверке младшего или старшего бита в слове:

```
        .def testL A B0 B1
        .test A 0 B0 B1
        .end
        .def testH A B0 B1
        .test A w B0 B1
        .end
```

6. Арифметика

Одна из базовых операций, которая требуется для определения других более сложных конструкций, это операция инкремент. Что бы инкрементировать ячейку ***A*** можно скомбинировать макросы вращения, сдвига и теста следующим образом:

```
        .copy ONE ctr

begin:  .testL A test0 test1

test0:  ONE A rollback
test1:  ZERO A
        .testH ctr next rollback

next:   .shiftL ctr
        .rollR A
        0 0 begin

rollback: .testL ctr roll End

roll:   .shiftR ctr
        .rollL A
        0 0 rollback

        End:0 0
        ...
        ctr:0 0
```

Первая строчка инициализирует счетчик `ctr` в 1. Младший бит операнда ***A*** инвертируется. Затем операнд и счетчик вращаются до тех пор пока бит операнда ноль, либо единица счетчика достигает позиции старшего бита, что означает что все ***w*** биты операнда были просмотрены. После этого операнд может подвергнуться обратному вращению к первоначальной позиции битов.

В этом коде `ZERO` и `ONE` определены как `ZERO:0` и `ONE:1`. Инструкция `0 0 label` используются как безусловный переход по адресу `label`. Она копирует первый бит в памяти в него же, то есть не изменяет его значения.

Сумма может быть определена похожим образом за исключением того что обрабатываются четыре операнда: два аргумента, результат и сумматор для передачи экстра бита. Младшие биты сумматора и двух аргументов складываются и дают в результате два бита, старший из которых переходит во второй бит сумматора, а младший в бит результата. После этого все четыре операнда вращаются и процесс продолжается. Код суммы – команда `add` – приведен в аппендиксе B1.

Вычитание определяется очень просто как:

```
.def sub X Y Z
.inv Y
.inc Y
.add X Y Z
.end
```

где `inc` инкрементирование, `add` суммирование (***Z=X+Y***) описанные выше, и `inv` операция инверсии, которая просто инвертирует все биты в ячейке памяти. Операция инверсии проще чем инкремент. В аппендиксе B2 проведен код для `inv`.

7. Управление и Указатели

Для тестирования величин переменных можно использовать следующие определения:

```
.def ifeq X Y yes no
.sub X Y Z
.ifzero Z yes no
Z:0 0
.end

.def ifzero Z yes no
.testH Z cont no
cont: .copy Z A
.inv A
.inc A
.testH A yes no
A:0 0
.end

.def iflt A B yes no
.sub A B Z
.testH Z no yes
Z:0 0
.end
```

Первый макрос `ifeq`, тестируя результат вычитания, проверяет равны ли оба аргумента друг другу. Второй макрос `ifzero` проверяет равен ли аргумент нулю. Это выполняется следующим образом: проверить старший бит (отрицательное число), если бит нулевой – выполнить операцию отрицания аргумента (в простом бинарном знаковом представлении

инверсия и инкремент производят операцию отрицания) и перетестировать старший бит. Аргумент копируется перед тем, как выполняется отрицание, потому что аргумент не должен быть изменен. Третий макрос `iflt` тестирует меньше ли первый аргумент чем второй.

Для написания классической программы “Hello, World!” с итерацией указателя по массиву ячеек памяти необходимо определить операцию дереференцирования указателя. Рассмотрим следующий фрагмент:

```
        Z0:0 Z1:0

start:  .deref p X
        .testH X print -1
print:  .out X
        .add p W p
        0 0 start

        p:H X:0
        H:72 101 108
        108 111 44
        32 87 111
        114 108 100
        33 10 -1
```

Метка ***H*** адрес строчки содержащей ASCII код “Hello, World!” со знаком новой строки. ***p*** указатель – ячейка инициализированная адресом строки.

Первая инструкция не меняет состояния памяти поскольку она копирует бит по адресу 0 в него же. Она необходима из-за условного перехода, который использует первые два слова и который является частью других макрокоманд. Следующая команда дереференцирует ***p***, копируя содержание ячейки, чей адрес содержится в ***p***, в ячейку ***X*** (эта операция обсуждена ниже). Процесс проверят отрицательна ли ***X***. Если да, то переходит по адресу (**-1**), в противном случае переходит на следующую строчку. Адрес (**-1**) специальный, в том смысле что выполнение программы останавливается если процесс переходит по адресу (**-1**). Подобным образом определяется остановка, например, в Subleq [2]. Следующая строчка печатает символ из ячейки ***X***. [Реализация печатания будет обсуждена позже в секции ввод/вывод.] Если ***X*** не неотрицательна, указатель ***p*** не достиг конца массива и все еще указывает на какой-то элемент. Указатель инкрементируется на размер ячейки памяти. Процесс повторяется до тех пор, пока инструкция остановки не выполнена.

Оказывается возможным скопировать ячейку памяти референцированную другой ячейкой, установив итеративную инструкцию с адресами источника и назначения, и повторяя эту инструкцию ***W*** раз, с каждым разом увеличивая адреса до тех пор, пока целое слово не скопировано. Например, как в следующем фрагменте:

```
        .copy ONE ctr

        .copy P A
        .copy L B

begin:  A:0 B:0
        .testH ctr next End

next:   .shiftL ctr
```

```
        .inc A
        .inc B
        0 0 begin

        End: ...
        L:X ctr:0
```

Этот фрагмент кода выполняет то же, что и выражение ***X=*P*** языка программирования C. Счетчик приготовлен как в предыдущих примерах. Значение указателя копируется в первый операнд итеративной инструкции (`A:0 B:0`), затем адрес ячейки результата копируется во второй операнд итеративной инструкции. Теперь эта инструкция исполняется ***W*** раз, с каждым разом увеличивая адреса – значения операндов.

Такой же подход может быть использован для копирования значения в ячейку указываемую другим указателем. Все дело только в том, что бы поменять местами операнды ***A*** и ***B*** в итеративной инструкции.

8. Больше Арифметики

Операция умножения относительно проста, если операции сдвига и сложения готовы [4]:

```
        .copy ZERO Z

begin:  .ifzero X End L1
L1:     .testL X next L2
L2:     .add Z Y Z
next:   .shiftR X
        .shiftL Y
        0 0 begin

        End:0 0
```

Код сдвигает первый множитель влево и второй множитель вправо, одновременно накапливая результат, и добавляя второй множитель, если младший бит первого множителя 1. Этот алгоритм выражается простой формулой:

$$X \times Y = \begin{cases} X/2 \times 2Y, & \text{if } X \text{ even;} \\ (X-1)/2 \times 2Y + Y, & \text{if } X \text{ odd.} \end{cases}$$

Деление немного сложнее. Имея два числа ***X*** и ***Y***, будем увеличивать ***Y*** в 2 раза до тех пор пока последующее увеличение ***Y*** даст больше чем ***X***. Во время увеличения ***Y***, будем увеличивать переменную ***Z*** в 2 раза, которая инициализирована единицей. В конце ***Z*** содержит часть результата деления, а остальную часть нужно вычислить используя ***X-Y*** и ***Y***, что повторяется накапливая все ***Z***. На последнем шаге когда ***X<Y***, ***X*** это остаток от деления. Код деления представлен в аппендиксе B3.

Операция деления необходима для печатания чисел в виде десятичной строчки. Алгоритм выполняющий эту функцию делит число на 10 и сохраняет остатки в массиве. Когда значение становится 0, цифры выводятся из массива в обратном порядке.

```
        .testH X begin negate
```

```
negate: .inv X
        .inc X
        .out minus

begin:  .div X ten X Z
        .toref Z p
        .add p W p
        .ifzero X print begin

print:  .sub p W p
        .deref p Z
        .add Z d0 Z
        .out Z
        .ifeq p q End print

        End:0 0
        ...
        Z:0 d0:48 ten:10
        p:A q:A minus:45
        A:0 0 0
        ...
```

Первая секция помеченная `negate` проверяет аргумент меньше ли чем 0. Если да, то аргумент превращен в положительный и знак минуса печатается. Вторая секция циклично делит аргумент и сохраняет результаты в массив ***A***, дереференцируя указатель ***p***. Команда `div` делит ***X*** на 10, сохраняет результат в ***X*** и остаток в ***Z***. Последующая команда `toref` пишет значение ***Z*** в ячейку, указанную указателем ***p***. Этот процесс повторяется пока X не равно нулю. В следующей секции помеченной меткой `print`, указатель ***p*** движется в обратном направлении до тех пор, пока он не равен ***q***, который инициализирован в ***A*** – начало массива. Команда `deref` копирует значение из массива в ***Z***. Затем ASCII код 48 символа 0 добавляется и байт готов к печатанию. [Предполагается что ячейка памяти не меньше чем 8-ми битовый байт.]

9. Ввод и Вывод

До настоящего момента программа может складывать, вычитать, умножать, делить, дереференцировать, выполнять итерации и условный переход. Что бы произвести вывод или получить ввод нужно сначала определить что такое ввод и вывод. Это относится к прагматике языка или, по другому, к окружению абстрактной машины, которая реализует язык. Любое определение ввода или вывода абстрактной машины будет ответственностью окружения, или в нашем случае эмулятора языка (или процессора, если он реализован в виде "железа"). Поскольку программа может копировать только один бит, то естественно определить поток битов, как биты скопированные из определенного адреса или в него. Один такой специальный адрес (**`-1`**) уже был использован как адрес остановки: программа останавливается если управление переходит по адресу (**`-1`**). Можно использовать этот же адрес без неоднозначности:

```
        .def out H
        H'0 -1
        H'1 -1
        H'2 -1
```

```
        H'3 -1
        H'4 -1
        H'5 -1
        H'6 -1
        H'7 -1
        .end

        .def in H
        -1 H'0
        -1 H'1
        -1 H'2
        -1 H'3
        -1 H'4
        -1 H'5
        -1 H'6
        -1 H'7
        .end
```

Заметим, что только младшие 8 бит копируются *из* слова и *в* слово. Это по практическим соображениям. С таким определением возможно написание ассемблерного кода независимого от размера ячейки памяти, который вводит и выводит 8-ми битовые символы.

Эмулятор содержит два буфера с восемью битами каждый. Когда программа выводит бит, он помещается в буфер. Когда буфер заполнен, символ в ASCII коде производится в стандартный вывод эмулятора и буфер очищается. Когда программа копирует бит из ввода, бит удаляется из буфера; если буфер пустой символ считывается и его биты помешаются в буфер.

Ниже представлена программа печатающая первые двенадцать факториалов.

```
      Z0:0 Z1:0

start:.prn X
      .mul X Y Y
      .out ex
      .out eq
      .prn Y
      .out eol
      .inc X

      .ifeq X TH -1 start

      X:1 Y:1 ex:33
      eol:10 eq:61 TH:13
```

Макрокоманда `prn` – команда печатания описанная в предыдущей секции. Программа печатает

```
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
```

```
11!=39916800
12!=479001600
```

Эта программа выполняется достаточно быстро на современном компьютере с настоящей реализацией ассемблера, эмулятора и библиотеки команд макроопределений. Размер слова 32 бита и размер программы (после ассемблирования) около 10000 инструкций.

10. Функции и Библиотека

Удобно собрать все макроопределения в один файл – библиотеку, и использовать ее для разных программ. Для этого определена третья ключевая команда (две другие `def` и `end`):

```
.include library_file_name
```

Любая программа, использующая библиотеку, требует ее включения и должна начинаться строчкой (`Z0:0 Z1:0`). Например,

```
        Z0:0 Z1:0

        .out H
        .out i
        0 0 -1

        H:72 i:105

        .include lib
```

печатает “Hi”.

Если бы все макроопределения описанные в этой статье были определены как макросы, то исполняемый код любой программы, даже простой, был бы огромным. Потому что макросы интенсивно определены через другие макросы. Это означает что любая команда разворачивается в каждом месте где используется. Так происходит сквозь всю иерархию макроопределений (см. аппендикс A). Что бы справиться с этой проблемой, команды могут быть определены как реальный код работающий со своими собственными аргументами. Такие блоки кода будем называть функциями. Макроопределение копирует формальные аргументы в аргументы функции и передает управление к входной точке функции. Вызывающий код также передает свой текущий адрес, что бы управление вернулось обратно к вызывающему коду. Как только управление возвращено, макроопределение может скопировать результат в аргументы если требуется. Очевидно что эти функции не могут быть рекурсивными, поскольку отсутствует стек [5].

Например, макрокоманда вычитания `sub` и ее функция определены как:

```
.def sub X Y Z : sub_f_X sub_f_Y sub_f_RET sub_f
        .copy X sub_f_X
        .copy Y sub_f_Y
        .copy L sub_f_RET
        0 0 sub_f
        L:J 0
        J:.copy sub_f_X Z
.end
```

```
:sub_f: .sub_f_def sub_f_X  sub_f_Y
sub_f_RET:0 sub_f_X:0 sub_f_Y:0

# sub internal macro definition
.def sub_f_def X Y : sub_f_RET

        .copy sub_f_RET Return

        .inv Y
        .inc Y
        .add X Y X

        End:0 0 Return:0

.end
```

Сначала идет макроопределение, которое копирует два аргумента в глобальные аргументы функции. Дальше идет глобальное определение входной точки функции. Тело функции определено через макро, только для того, что бы сохранить внутренние имена вне глобальной области видимости. Проигнорируем пока двоеточие перед меткой входной точки функции. Следующая строчка определяет ячейки памяти для адреса возврата и двух аргументов. Два аргумента достаточно, потому что результат возвращается в первом аргументе функции. Следующая строчка комментарий. Затем идет тело функции. Ее первая команда – это копирование адреса возврата в свою последнюю инструкцию – безусловный переход назад к вызываемому коду [6].

Функции позволяют одному и тому же коду исполнятся много раз, вместо повторения кода в каждом месте где требуется какая-либо операция. Однако появляется побочный эффект: поскольку каждая точка входа глобальная (не внутри макро определения), код функции будет присутствовать в программе даже если функция не используется. Это нежелательно. Маленькие программы должны оставаться маленькими после ассемблирования, и не должны включать целую библиотеку. Что бы избежать этого, дополнительный механизм должен быть добавлен в ассемблер. Ассемблер помечает команду: инструкцию или макрокоманду как условную если строчка команды начинается с двоеточия. Если имя команды – метка – становится неразрешенным именем после ассемблирования, команда добавляется в программу. Вот почему строчка (`sub_f`), в примере вверху, начинается с двоеточия.

Заключение

В настоящей статье были достигнуты две цели. Во первых, был изобретен новый OISC язык имеющий намного более простую инструкцию, чем инструкции других OISC языков известных в настоящий момент.

Во вторых, было показано что операция копирования бита с референцированием (или адресованием) достаточна для полной по Тюрингу вычислительной машины [8]. Оказалось, что она не только возможна в принципе, но так же и практически достижима. Простые программы написанные в этом языке работают в разумных пределах память-временных ресурсов.

Эту идею возможно использовать как основу для построения компьютера с минимальной потребляемой мощностью или минимально возможных размеров.

Язык представленный в статье был реализован. Ассемблер, эмулятор, и библиотека могут быть взяты из [7].

Аппендикс A

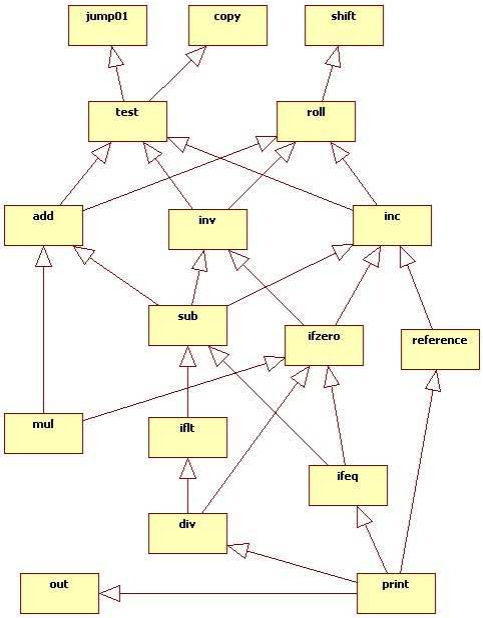

Диаграмма представляет зависимости между функциями и макросами в библиотеке в настоящей реализации [7]. Прямые зависимости соответствующие косвенным зависимостям на диаграмме отсутствуют. Различные реализации алгоритмов могут привести к другой диаграмме зависимостей, но общие уровни зависимости останутся неизменными.

Аппендикс B

B1 .add

Этот код определяет операцию сложения как описано в секции “Арифметика”:

```
        .copy ONE ctr
        .copy ZERO adr

begin:  .copy ZERO btr
        .testL adr testx inctestx

inctestx: .inc btr
testx:  .testL X testy inctesty

inctesty: .inc btr
testy:  .testL Y testz inctestz
```

```
inctestz: .inc btr
testz:  btr Z
        btr'1 adr'1

        .testH ctr rollcont rollback

rollcont: .shiftL ctr
        .rollR adr
        .rollR X
        .rollR Y
        .rollR Z
        0 0 begin

rollback: .testL ctr roll End

roll:   .shiftR ctr
        .rollL Z
        0 0 rollback

        End:0 0
        ...
        ctr:0 adr:0 btr:0
```

Вспомогательная переменная `ctr` используется для счета количества вращений приложенных к аргументу. Переменная `adr` используется для передачи битов к следующей битовой позиции. Переменная `btr` – это сумма трех битов, взятая из одинаковых битовых позиций двух суммируемых аргументов и переменной `adr`.

B2 .inv

Код инвертирования битов в одном слове прост. `ctr` как обычно вспомогательная переменная.

```
        .copy ONE ctr

begin:  .testL ARG copy1 copy0

copy1:  ONE ARG 4?
copy0:  ZERO ARG

        .testH ctr rollcont rollback

rollcont: .shiftL ctr
        .rollR ARG
        0 0 begin

rollback: .testL ctr roll End

roll:   .shiftR ctr
        .rollL ARG
        0 0 rollback

        End:0 0
```

B3 .div

Ниже приведен рабочий пример алгоритма деления описанного в секции “Больше Арифметики”. Его аргументы: ***X*** - делимое, ***Y*** - делитель, ***Z*** - результат целочисленного деления, ***R*** - остаток.

```
        .copy ZERO Z

        .testH X L1 End
L1:     .testH Y L2 End
L2:     .ifzero Y End begin

begin:  .iflt X Y L3 L4

L3:     .copy X R
        0 0 End

L4:     .copy Y b1
        .copy ONE i1

next:   .copy b1 bp
        .copy i1 ip
        .shiftL b1
        .shiftL i1

        .iflt X b1 rec L5

rec:    .sub X bp X
        .add Z ip Z
        0 0 begin

L5:     .testH b1 next End

        End:0 0
        ...
        b1:0 bp:0 0
        i1:0 ip:0 0
```

Благодарности

Я хотел бы поблагодарить мою дочь, Софию Мазонка, за корректировку моего английского. Также хотел бы поблагодарить Джеймса Тебнефа за ценные комментарии, которые улучшили читабельность этой статьи.

Ссылки и комментарии

1. OISC (one instruction set computer), компьютер с одной инструкцией.

2. Subleq, http://esolangs.org/wiki/Subleq, один из языков OISC.

3. Higher Subleq, компилятор компилирующий из C-образного языка в код Subleq, http://mazonka.com/subleq/hsq.html

4. Алгоритм не правильно работает с отрицательными значениями. Пожертвовано в пользу простоты.

5. Это не означает что это понятие не может быть введено. Пример тому Higher Subleq.

6. Эту команду копирования можно избежать если внешняя макро команда может копировать непосредственно в ту ячейку памяти.

7. Язык BitBitJump, http://mazonka.com/bbj/

8. Точнее говоря только язык ассемблера с несколькими библиотечными макрокомандами может считаться полным по Тюрингу. Инструкция копирования бита нечетко полна по Тюрингу, или боле точно, она принадлежит к классу линейно ограниченных автоматов, который является классом обычных компьютеров. Формальное доказательство можно найти в [7] где представлен интерпретатор полного по Тюрингу языка DBFI описанного в [9]. Keymaker (пользователь esolangs.org) предположил, что язык инструкции мог бы быть полным по Тюрингу, если использовать относительную адресацию, а не абсолютную. Кажется вполне возможным переопределение языка с относительной адресацией без изменения основной идеи.

9. Oleg Mazonka, Daniel B. Cristofani, “A Very Short Self-Interpreter”, arXiv:cs/0311032v1, перевод на русский, “очень короткий самоинтерпретатор”, http://www.progz.ru/forum/index.php?showtopic=28468