МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

А. М. Пеленицын, Н. Н. Ячменёва

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к практикуму по курсу «Архитектура компьютера»

Ростов-на-Дону
2014

Методические указания разработаны ассистентами кафедры информатики и вычислительного эксперимента А. М. Пеленицыным и Н. Н. Ячменёвой.

Печатается в соответствии с решением кафедры информатики и вычислительного эксперимента факультета математики, механики и компьютерных наук ЮФУ, протокол № ____ от «____» ______________ 2014 г.

СОДЕРЖАНИЕ

Введение. Использование утилит ассемблирования 4

1 Простейшие программы, арифметика и циклы LOOP 8
1.1 Структура программы на языке ассемблера Intel 8088 8
1.2 Режимы адресации 10
1.3 Базовые арифметические команды 12

2 Массивы. Условные и безусловные переходы 14
2.1 Массивы 14
2.2 Моделирование условного оператора 16
2.3 Полная форма условного оператора 19
2.4 Организация циклов с помощью условных переходов 19

3 Интерфейс системных вызовов. Простейшие подпрограммы 21
3.1 Интерфейс системных вызовов, вывод на консоль 21
3.2 Простейшие подпрограммы 24

4 Подпрограммы (продолжение) 28
4.1 Конвенции вызова 28
4.2 Переменное число параметров подпрограммы 30
4.3 Передача адресов подпрограмм в другие подпрограммы . . 30

5 Цепочечные инструкции 31

6 Работа с файлами 34

7 Микропрограммирование 37
7.1 Микроархитектура Mic-1 и набор инструкций IJVM 38
7.2 Пример реализации инструкции IADD 41
7.3 Работа с симулятором Mic1MMV 42

Библиография 48

ВВЕДЕНИЕ. ИСПОЛЬЗОВАНИЕ УТИЛИТ АССЕМБЛИРОВАНИЯ

Цель лабораторных работ по курсу «Архитектура компьютера» состоит в знакомстве студентов с низкоуровневым интерфейсом компьютера, набором инструкций процессора, с помощью соответствующего языка ассемблера. Для решения этой задачи использован набор инструкций достаточно простого процессора Intel 8088. Для ассемблирования и запуска программ на языке ассемблера данного процессора используются инструменты из сопроводительных материалов к учебнику [1], доступные онлайн [2]. Кроме того, в последнем разделе изложен процесс и задачи для работы с эмулятором учебной микроархитектуры Mic-1, описанной в том же издании [1]. Это позволяет перейти к изучению ещё более низкого уровня организации вычислительной системы — внутреннему устройству процессора и способу реализации некоторого набора инструкций, в данном случае IJVM, на основе этого устройства.

Набор утилит as88/s88/t88 из пакета ACK (Amsterdam Compiler Kit) предназначен для ассемблирования, запуска и пошагового исполнения программ на языке ассемблера процессора Intel 8088, который представляет собой упрощённую версию современных процессоров Intel семейства x86 и совместимых с ними (например, AMD). Данные утилиты разработаны специально для использования в учебном процессе сотрудниками Свободного университета Амстердама под началом Эверта Ваттеля (Evert Wattel) и кратко описаны в [1, Прил. В].

Утилиты скомпилированы под большинство современных операционных систем, однако в ОС семейства Windows необходима дополнительная настройка. В дисплейных классах факультета рекомендуется использовать версию под Linux. На домашних машинах студентам рекомендуется установить виртуальную машину с ОС Линукс (если её ещё нет). Дальнейшие указания приводятся для ОС Линукс.

Утилиты имеют консольный интерфейс и потому предполагают владение командами терминала на базовом уровне: переход между каталогами

(команда cd — от change directory), запуск исполняемых программ с аргументами (в нашем случае аргументами обычно являются имена файлов).

– `<имя программы> <имя файла>` — если программа лежит в «системном каталоге»,
– `<имя программы, включая полный путь> <имя файла>` — иначе.

Примеры:

– `./as88 test.s` — символы `./` здесь как раз описывают полный путь к программе и дословно означают «текущий каталог»;
– `bin/as88 test.s` — файл `as88` лежит в подкаталоге `bin` текущего каталога;
– `as88 test.s` — файл `as88` лежит в «системном» каталоге.

Дальнейшие указания приводятся для случая, когда все файлы лежат в одном каталоге.

Запуск утилит из набора АСК

В набор АСК входят три исполняемых файла:

– as88 — ассемблер для Intel 8088, эта программа принимает на вход текстовый файл с ассемблерным листингом и генерирует бинарный (исполняемый) файл, а также два вспомогательных файла, о которых будет сказано ниже;
– s88 — интерпретатор бинарного кода для Intel 8088;
– t88 — трассировщик бинарного кода для Intel 8088.

Ассемблерные листинги принято размещать в файлах с расширением .s. Пусть имеется текстовый файл test.s с ассемблерным листингом. Тогда команда в консоли

```
./as88 test.s
```

создаст три файла:

– test.88 — бинарный код для Intel 8088;

– test.# — файл с указанием, какие позиции исходного файла соответствуют каким позициям бинарного файла;

– test.$ — копия файла с исходным кодом, где подставлен текст включаемых вспомогательных файлов, а также удовлетворены соглашения по оформлению ассемблерных листингов. Учтите, что в случае ошибок ассемблирования as88 указывает номера строк именно в этом файле.

Команда

```
./s88 test.s
```

запускает полученный бинарный код. Если в программе использовался ввод-вывод, то он будет происходить в обычном режиме. Однако в первых программах ввод-вывод не используется, потому необходимо пользоваться трассировщиком t88.

Трассировка

Команда

```
./t88 test.s
```

запускает трассировку (исполнение по шагам) бинарного файла для Intel 8088. Во время такого выполнения можно видеть значения регистров, стека и других участков памяти на каждом шаге.

Для выполнения очередного шага трассировки следует нажать Enter. Для выхода из трассировщика следует нажать q, а затем Enter. Чтобы следить за изменением памяти по метке X, можно использовать команду: /X (после набора этих символов во время выполнения трассировки нужно нажать Enter).

Более полное описание возможностей t88 (в том числе использование точек останова, breakpoints) можно найти на страницах 762–766 [1].

Настройка использования утилит из АСК в Windows

В ОС семейства Windows, начиная с версии XP, необходимо найти файл config.nt (регистр букв в имени может отличаться). Обычно это

```
C:\Windows\System32\config.nt
```

(Если такой файл не нашёлся в этой папке, запустите поиск по всему компьютеру.) Также необходимо узнать расположение файла ansi.sys — обычно он лежит в том же каталоге. В файл config.nt необходимо добавить одну строчку в самом конце:

```
device=ПУТЬ_К_ANSI.SYS
```

(редактировать файл можно в Блокноте). Например,

```
device=C:\Windows\System32\ansi.sys
```

После этого нужно сохранить файл config.nt и перезапустить систему.

Если при сохранении файла вам откажут за отсутствием привилегий администратора, можно сохранить полученный файл в какой-либо временный каталог и уже потом скопировать его оттуда в каталог C:\Windows\System32 с помощью «Проводника»: при этом вас попросят выбрать функцию «Выполнить с правами администратора».

64-разрядные системы

В 64-разрядных системах требуются дополнительные настройки окружения. Для ОС Windows они непонятны. Если у вас Windows на x64, то вам рекомендуется использовать виртуальную машину.

В Линуксе на x64 необходимо доставить один программный пакет, в Ubuntu это делается следующей командой в терминале:

```
sudo apt-get install ia32-libs
```

1 ПРОСТЕЙШИЕ ПРОГРАММЫ, АРИФМЕТИКА И ЦИКЛЫ LOOP

1.1 Структура программы на языке ассемблера Intel 8088

Структура программы на языке ассемблера Intel 8088 может быть представлена следующим образом:

```
.SECT .TEXT
! <последовательность инструкций процессора>
.SECT .DATA
! <последовательность команд as88 выделения памяти с инициализацией>
.SECT .BSS
! <последовательность команд as88 выделения памяти без инициализации>
```

Символ восклицательного знака здесь обозначает начало однострочного комментария. Первая секция предназначена для написания кода, вторые две — для хранения статических данных, это полный аналог глобальных переменных в высокоуровневых языках программирования, например, в языке C. Последние две секции могут быть пустыми, но заголовки всё равно должны присутствовать.

Пример последовательности инструкций для сложения двух чисел, два и три:

```
.SECT .TEXT
    MOV     AX, 2
    ADD     AX, 3
```

Сначала в регистр AX помещается (MOV) двойка. Затем к содержимому AX добавляется тройка.

Следует в общем виде представлять организацию компьютера, которая подразумевается языком ассемблера. Записанные в текстовом файле инструкции превращаются ассемблером в бинарные коды, соответствующие этим инструкциям, при этом осуществляется почти взаимно-однозначное соответствие. Бинарный код программы при запуске с жёсткого

диска помещается в оперативную память. Далее инструкции в простейшем случае по одной последовательно загружаются из памяти на процессор и выполняются. На процессоре имеются быстрые именованные элементы памяти, называемые регистрами. В регистрах могут храниться операнды и результаты операций. Доступ к регистрам осуществляется намного быстрее, чем в оперативную память, поэтому если некоторые значения в программе используются особенно часто, их следует помещать в регистры.

Смысл инструкций MOV и ADD легко запомнить, если применять следующую мнемонику: большинство арифметических инструкций аналогичны присваивающим формам арифметических операций в языке C (в частности, MOV аналогичен операции присваивания). То есть код выше можно было бы перевести на C следующим образом:

```
AX  = 2
AX += 3
```

Попробуйте сассемблировать (с помощью as88) и выполнить по шагам (с помощью t88) программу, приведённую выше. Для этого нужно создать текстовый файл (назовём его task-0.s) с объявлением всех трёх секций и с двумя указанными инструкциями в секции кода. В ходе выполнения по шагам проследите изменение значений регистра AX. После выполнения обеих инструкций (клавиша Enter) и проверки значения регистра AX завершите работу командой q.

Замечание по оформлению кода (правило «четырёх столбцов»). Код рекомендуется оформлять в 4 столбца: столбец меток, столбец инструкций и команд ассемблера, столбец операндов, комментарии. Это правило используется, даже если столбец пуст (например, первая колонка с метками почти всегда пуста и потому инструкции пишутся с отступом). Исключением являются начала секций — директива .SECT идёт без отступа. Столбцы разделяются достаточным для визуального разделения количеством нажатий Tab (1 или 2 в зависимости от настроек редактора). Между секциями следует оставлять пустые строки.

1.2 Режимы адресации

Под термином «режимы адресации» в языках ассемблеров понимается способ передачи («адресация») операндов инструкций. В примере выше уже были использованы два режима адресации — перечислим их, а также третий режим, который понадобится далее, последовательно.

а) Непосредственная адресация («непосредственные операнды») — операнд передаётся процессору из оперативной памяти прямо вместе с инструкцией. Так заданы операнды 2 и 3 в примере выше.

б) Регистровая адресация — операнд находится в указанном регистре процессора (пока мы использовали один регистр — AX).

в) Прямая адресация в памяти — операнды располагаются в оперативной памяти (в наших примерах чаще всего — в сегменте данных) по известному на этапе ассемблирования адресу. Такой адрес обычно задаётся с помощью символического имени: «метки». Ниже приводится соответствующий пример.

```
.SECT .TEXT
    MOV     AX, (x)
    ADD     AX, (y)

.SECT .DATA
x:  .WORD   2
y:  .WORD   3
```

В данном примере суммируемые значения размещены в сегменте данных по меткам x и y. Метка отделяется от команды двоеточием. Метка очень похожа на константный указатель C++: она представляет адрес нужного значения (в примере x это адрес значения 2, y это адрес значения 3). Для обращения по этому адресу используется запись (x), (y) и т. п., что аналогично операции разыменования указателя * в языке C. Часто для краткости метки наподобие x и y называют «переменными».

Заметим, что слово «адрес» выше используется неформально, для упрощения изложения. Адреса в рассматриваемом наборе инструкций x86 состоят из двух частей: сегмента и смещения. Метки представляют смещение, а сегмент обычно выводится из контекста. В простейших примерах эти подробности можно опустить, считая смещение собственно адресом. В описании ассемблера для Intel 8088 [1, Прил. B] данный вопрос рассматривается подробнее. Студентам рекомендуется ознакомиться с этим описанием.

Задача 1 (task-1.s). Создайте программу, которая вычитает из числа 3 число 2 по аналогии с примером выше (числа размещены в сегменте данных). Инструкция вычитания `SUB` получает операнды полностью аналогично `ADD`. ♢

Задача 2 (task-2.s). Добавьте в предыдущую программу (используйте команду «Сохранить как...») к переменным `x` и `y` переменную `res`. Поскольку у `res` нет исходного значения, её разумно поместить в секции неинициализированных данных `BSS`. В этой секции вместо команды `.WORD` нужно использовать команду `.SPACE`, после которой указывается количество байт под переменную. Все целочисленные значения имеют размер 2 байта.

После получения результата в регистре `AX` скопируйте содержимое `AX` в `res` с помощью инструкции `MOV`. Следить за значением `res` можно после исполнения команды:

```
/res
```

Её нужно ввести после запуска t88. ♢

Задача 3. Не все комбинации режимов адресации допустимы. Проверьте, можно ли решить предыдущую задачу без использования регистра AX. Вариант 1 (task-3-1.s): поместить (MOV) в res значение 3, а затем вычесть (SUB) из res число 2. Если операция не удалась, запишите в файл с программой в комментарии (после символа ! до конца строки) текст ошибки ассемблирования. Данные в секции BSS инициализируются нулями, поэтому,

чтобы получить в res исходную 3, можно воспользоваться ADD (прибавляем к 0 число 3). Затем вычитаем из res число 2. Напишите два варианта этой логики: когда для вторых операндов ADD и SUB (3 и 2) используется прямая адресация в памяти (task-3-2.s) и непосредственные операнды (task-3-3.s). В случае неудачи в каком-то из случаев добавьте текст ошибки ассемблирования в комментарии. ♢

Задача 4 (task-4.s). Метки представляют собой константные адреса, то есть целые числа, которые определены на этапе ассемблирования — а значит, при каждом запуске программы они будут одинаковыми. По сути это просто символические константы аналогично `#define` в языке C.

Загрузите в регистры AX, BX и CX значения меток `x`, `y`, `res` (не путать со значениями *по* меткам), объявленных как в программе из позапрошлой задачи (task-2.s): используйте MOV и имена меток без круглых скобок. Проанализируйте полученные значения. Добавьте после `x` переменную `x1` (то есть метку `x1` и ещё одну инструкцию `.WORD`). Как изменились значения меток? ♢

1.3 Базовые арифметические команды

Задача 5 (task-5.s, умножение). Инструкция умножения MUL обладает отличным от ADD и SUB интерфейсом. Она имеет один (явный) операнд, который играет роль одного из множителей. Второй сомножитель берётся (неявно) из регистра AX. Результат умножения может не поместиться в один регистр, потому для него используется пара регистров: младшие 16 бит результата помещаются в регистр AX, старшие — в DX. В задачах будут использоваться достаточно маленькие числа, потому регистр DX существенно использоваться не будет, однако такое устройство инструкции MUL может приводить к проблемам, если в регистре DX хранится некая полезная информация: после выполнения MUL она непременно затрётся (если результат умножения положителен и поместился в 16 бит, то в DX будет записан ноль). Поэтому программист должен сохранить полезную

информацию из DX в каком-либо месте перед использованием MUL — в переменной или в другом регистре.

Следуя данному описанию, составьте программу, которая перемножает два числа, два и три, заданные в сегменте данных, и помещает результат в переменную `res`, объявленную в секции BSS. ◊

Задача 6 (task-6.s, целочисленное деление). Инструкция целочисленного деления DIV имеет аналогичный MUL интерфейс. Так же используется один явный параметр, играющий роль делителя. Делимое считается 32-битным и берётся из пары регистров DX:AX (старшие биты в DX). Если делимое помещается в 16 бит, то его нужно загрузить в регистр AX, а DX заполняется 1-битами, если делимое отрицательно, и 0-битами в противном случае (такая операция называется знаковым расширением). Для этого непосредственно перед использованием инструкции DIV следует передать инструкцию CWD (Convert Word to Doubleword) без операндов. Вновь следует помнить о том, что хранившаяся в DX информация будет утеряна.

Частное от деления помещается в регистр AX, а остаток от целочисленного деления — в DX.

Следуя данному описанию, составьте программу, которая вычисляет сумму: $55/10 + 55\%10$ и помещает результат в переменную `res`, объявленную в секции BSS. (Здесь и далее / означает целочисленное деление, а $\%$ — остаток от целочисленного деления). ◊

Задача 7 (task-7.s, простейший цикл с LOOP). Для организации циклов используются метки и специальные инструкции перехода по этим меткам. Простейшая из таких инструкций — LOOP.

```
        MOV     CX, (n)       ! количество шагов загружаем в CX —
                              !   это требование LOOP
L1:     ADD     AX, (x)
        LOOP    L1            ! уменьшает CX на 1, и если CX не равен 0,
                              !   то переходит по метке L1
```

Этот код `(n)` раз прибавляет к AX значение `(x)` (предполагается, что `n` и `x` это метки из сегмента данных, указывающие на некоторые числа).

Следуя данному описанию, составьте программу, которая считает сумму $5+10+15+20+25$. Очередное слагаемое удобно хранить в отдельном регистре (например, BX) и увеличивать его на 5 на каждом шаге цикла. ♢

Дополнительные задачи

Задача 8 (task-extra-1.s). Вычислите значение многочлена $2x^4-3x^2+x-5$ в точке $x = 7$ (задано в сегменте данных). *Указание*: четвёртая степень x должна вычисляться с помощью возведения в квадрат второй. ♢

Задача 9 (task-extra-2.s). Вычислите значение выражения

$$3x^4/8 - x^3\%5 + x - 1$$

в точках $x = 3, 6, 9$ (в цикле LOOP). ♢

2 МАССИВЫ. УСЛОВНЫЕ И БЕЗУСЛОВНЫЕ ПЕРЕХОДЫ

2.1 Массивы

Принципы организации и обработки массивов в языке ассемблера аналогичны языку C. Основная идея заключается в том, что массивом считается непрерывный участок памяти, в котором друг за другом размещены значения фиксированного размера.

Пример объявления массива:

```
.SECT .DATA
arr:    .WORD   2, 5, 3, 8, 1
```

Многие задачи на обработку массивов связаны с организацией цикла, количество итераций которого равно длине массива. Для указания длины

можно было бы использовать явное числовое значение (5 для массива `arr` из примера), однако рекомендуется вычислять это значение по формуле:

```
<длина массива> = (<адрес конца>-<адрес начала>) / <размер элемента>
```

Адрес начала задаётся меткой перед объявлением массива (`arr` в примере выше). Адрес конца указывает любая метка, поставленная сразу после объявления массива; если таковой нет, то её вводят вместе с выделением фиктивной переменной. В случае массива слов (`.WORD`) размер элемента равен двум байтам, а для деления на 2 удобно использовать битовый сдвиг вправо на одну позицию. Таким образом, для организации цикла по массиву с помощью `LOOP` необходимо сделать так:

```
        MOV     CX, end - arr
        SHR     CX, 1           ! shift right, деление на 2
LA:
        ! действия в цикле ...
        LOOP    LA
```

Здесь `end` это имя метки фиктивной переменной, объявленной сразу после массива.

Для доступа к очередному элементу массива в самом простом случае удобно использовать *четвёртый режим адресации* (помимо трёх, рассмотренных в первой лабораторной работе), а именно, косвенную адресацию. По форме она похожа на прямую адресацию в памяти (используются круглые скобки), однако нужный адрес указывается не меткой, а хранится в одном из регистров: `BX`, `SI`, `DI`.

```
        MOV     BX, arr
        ADD     (BX), 3         ! добавили 3 к первому элементу
        ADD     BX, 2           ! переместились к следующему элементу
        SUB     (BX), 2         ! отняли 2 от второго элемента
```

Эта логика совершенно аналогична обработке массива в языке C с использованием указателя (`int * p = arr`), скользящего вдоль массива (`++p`) и

выполняющего обращение к отдельным элементам с помощью разыменования (*p). Для организации цикла по массиву в языке ассемблера следует загрузить адрес начала массива в один из допустимых для косвенной адресации регистров (например, BX) и увеличивать значение этого регистра на два в конце каждой итерации цикла.

Задача 1 (task-1.s). Напишите программу, которая суммирует все элементы массива. Результат остаётся в регистре AX. ◊

2.2 Моделирование условного оператора

Условный оператор в языке ассемблера Intel 8088 моделируется с помощью группы команд условных переходов J** (сравнение на больше, меньше, больше или равно и т.п., см. рис. 1), каждая из которых принимает один операнд — имя метки, к которой осуществляется переход при выполнении нужного условия. Этим команды J** похожи на команду LOOP. Конкретные величины, которые сравниваются, задаются в команде CMP, которую следует вызвать перед J**.

Пример кода, который увеличивает значение регистра BX на единицу, если значение в регистре AX больше пяти:

```
        CMP     AX, 5
        JLE     LIF     ! LE = less or equal: если AX <= 5
                        !    "перепрыгиваем" (jump) INC
        INC     BX
LIF:
        ! продолжение программы...
```

Задача 2 (task-2.s). Напишите программу, которая суммирует все элементы массива, большие или равные N, где $N = 3$ — переменная, заданная в секции данных.

Указание: обращаться в память по метке N на каждом шаге цикла неэффективно, перед началом цикла загрузите это значение в один из свободных регистров. ◊

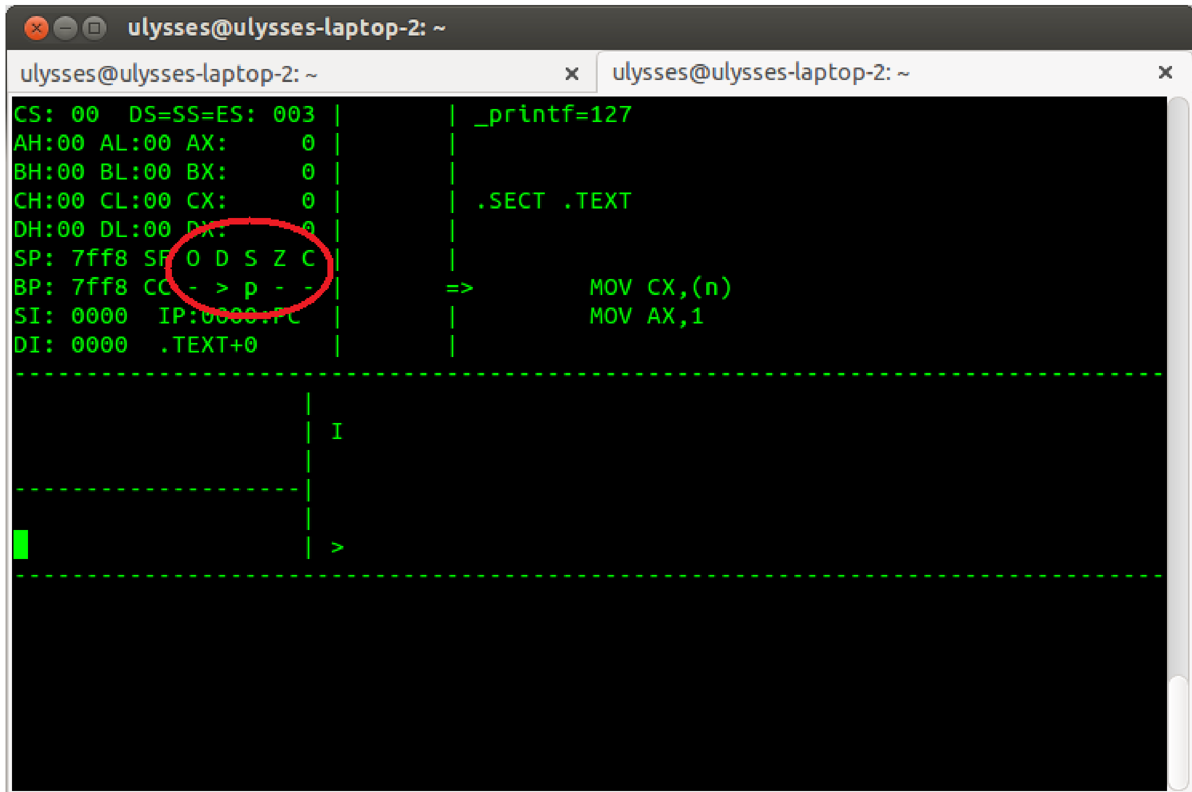
Рисунок 1 — Инструкции условных переходов.

Команда	Описание	Условие перехода
JNA, JBE	Ниже или равно	CF = 1 или ZF = 1
JNB, JAE, JNC	Не ниже	CF = 0
JE, JZ	Нуль, равно	ZF = 1
t@tabl_body = JNLE, JG	Больше чем	SF = OF и ZF = 0
JGE, JNL	Больше или равно	SF = OF
JO	Переполнение	OF = 1
JS	Отрицательный знак	SF = 1
JCXZ	Значение CX равно нулю	CX = 0
JB, JNAE, JC	Ниже	CF = 1
JNBE, JA	Выше	CF = 0 и ZF = 0
JNE, JNZ	Не равно нулю, не равно	ZF = 0
JL, JNGE	Меньше чем	SF ≠ OF
JLE, JNG	Меньше или равно	SF ≠ OF или ZF = 1
JNO	Без переполнения	OF = 0
JNS	Неотрицательно	SF = 0

Разберёмся, как устроены команды `J**`, а именно, каким образом они получают информацию от инструкции сравнения `CMP`. В процессорах семейства Intel x86 (как и во многих других) существует специальный регистр флагов, в котором каждый бит имеет имя и представляет собой специальное условие (является «флагом»). Примеры флагов, связанных с арифметическими операциями: `ZF` — флаг нуля (zero), `CF` — флаг переноса (carry), `SF` — знаковый флаг (sign), `OF` — флаг переполнения (overflow). Следить за значением этих флагов можно в окне t88 (см. рис. 2). Большинство арифметических инструкций кроме выполнения нужной операции взводят соответствующие флаги. Инструкция `CMP` похожа на арифметические инструкции: она выполняет вычитание второго операнда из первого, результат игнорируется, но взводятся соответствующие флаги.

Инструкции `J**` анализируют текущее состояние флагов. Однако имена этих инструкций даны так, что при использовании перед ними

Рисунок 2 — Индикатор флагов в окне t88.



инструкции CMP упомянутый анализ флагов точно отражается именами J**. Рассмотрим пример.

```
    CMP     AX, BX
    JLE     LBL1
```

Здесь переход по метке LBL1 происходит, если AX меньше или равен (Less or Equal — JLE) BX. В этом случае программисту можно вообще ничего не знать про существование флагов. Однако часто бывает удобно (и более эффективно) использовать другие арифметические инструкции, взводящие флаги, кроме CMP и на их основе выполнять переходы J**. В этом случае нужно разобраться, какие флаги затрагивает интересующая арифметическая инструкция и какие из переходов J** анализируют именно эти флаги.

Задача 3 (task-3.s). Напишите программу, которая суммирует все чётные элементы массива. Для проверки чётности используйте деление на 2 с помощью SHR: проследите, какой флаг может взводится этой командой в случае нечётных или чётных чисел и найдите в приведённой выше таблице команду J**, которая анализирует именно этот флаг. *Указание*: модифицировать решение задачи 2. ◊

2.3 Полная форма условного оператора

В предыдущих задачах на условные переходы моделировалась сокращённая форма условного оператора (без ветки `else`). Для получения полной формы в комбинации с одной инструкцией J** используется инструкция безусловного перехода JMP. Модифицируем первый пример на условные переходы.

```
        CMP     AX, 5
        JLE     LELSE
        INC     BX      ! ветка then  (если AX > 5)
        JMP     LIF     ! безусловный переход к концу
                        !    "условного оператора"
LELSE:
        DEC     BX      ! ветка else (уменьшаем BX, если AX <= 5)
LIF:
```

Задача 4 (task-4.s). Напишите программу, которая отдельно суммирует все элементы массива, больше или равные N, где $N = 3$ — переменная, заданная в секции данных, и отдельно суммирует элементы, меньшие N. Первый и второй результат в конце программы должны находиться в AX и BX соответственно. *Указание*: модифицировать решение задачи 2. ◊

2.4 Организация циклов с помощью условных переходов

Выше отмечалось, что инструкции J** аналогичны LOOP. Легко заметить, что с помощью инструкций J** тоже можно организовывать циклы. На самом деле, промышленные компиляторы практически никогда не генерируют инструкцию LOOP для организации цикла (например, цикла `for`

в языках C или Pascal), а всегда обходятся J**. Это связано с тем, что LOOP занимает один регистр (CX) и использует его строго определённым образом, который не слишком удобен для большинства задач. В таком случае этот регистр просто «потерян» для программиста. В случае замены LOOP на J** этого можно избежать.

Задача 5 (task-5.s). Напишите программу, которая вычисляет значение выражения $4x^2 - x\%3$ для $x = 4, 8, 12$.

Указания. Не использовать LOOP. Выбрать регистр для хранения очередного значения x и делать переход J** к началу тела цикла, пока значение этого регистра меньше или равно 12. Для умножения на 4 используйте битовый сдвиг влево SHL: для сдвига на 2 и более (до 15) величину сдвига следует передавать в регистре CL (нижняя половинка регистра CX), чтобы загрузить в CL двойку, следует использовать инструкцию MOVB (B от byte: явно указывается размер операнда). ♢

Дополнительные задачи

Во всех задачах «найти» означает поместить в AX к концу программы.

Задача 6 (task-extra-1.s). Напишите программу, которая находит минимум в заданном массиве. ♢

Задача 7 (task-extra-2.s). Напишите программу, которая находит минимальный из чётных элементов массива в предположении, что такие элементы в массиве есть. ♢

Задача 8 (task-extra-3.s). Найти первый элемент массива, кратный четырём.

Указание. Можно использовать LOOP, но в теле цикла при нахождении нужного элемента делать переход J** из тела цикла к первой инструкции после цикла. Если в вашей программе после цикла не нужно делать ничего, то после цикла можно вставить инструкцию NOP (No OPeration — отсутствие действия). ♢

3 ИНТЕРФЕЙС СИСТЕМНЫХ ВЫЗОВОВ. ПРОСТЕЙШИЕ ПОДПРОГРАММЫ

3.1 Интерфейс системных вызовов, вывод на консоль

Программы на микрокомпьютерах («персональных компьютерах») обычно запускаются поверх операционной системы. Это позволяет осуществлять ряд операций (в первую очередь, ввод-вывод), используя интерфейс ОС и не прибегая к более низкоуровневому взаимодействию с периферийными устройствами (BIOS, прямая запись в память устройств и т. п.).

Средства ACK (as88/s88/t88) поддерживают 7 системных вызовов и 5 функций, совместно моделирующих базовые средства ввода-вывода ОС UNIX и подобных (полное перечисление см. в [1, Приложение B]). Для обращения к любому из этих средств применяется одна инструкция SYS. При этом для указания конкретного системного вызова (функции), а также его аргументов, используется стек. Обычно номера системных вызовов выносятся в константы.

Приведём пример системного вызова для корректного завершения программы (с явным сообщением ОС о том, что программа завершена успешно).

```
_EXIT = 1
.SECT .TEXT
        PUSH        0       ! return code
        PUSH        _EXIT
        SYS
```

(Секции DATA и BSS опущены для краткости). Системный вызов для завершения программы имеет номер 1 (для наглядности используется именованная константа). Он имеет один аргумент, код возврата (return code): в ОС UNIX каждая программа по завершении должна была сообщать ОС целое число, описывавшее результат работы программы. Обычно это число было нулём, если программа завершалась удачно, иначе число обозначало «код

ошибки», который следовало расшифровывать с помощью документации к программе. Заметим, что данное соглашение используется в большинстве современных операционных систем.

Для вывода на консоль целых чисел следует использовать системный вызов с кодом 127. Под ним скрывается известная из языка C функция форматного вывода `printf`, которая требует передачи строки, содержащей формат вывода, и аргументов, которые будут подставлены в нужные места форматной строки (если таковые имеются). Для примера смоделируем следующий вызов функции `printf` из языка C:

```
printf("%d\n", 42);
```

на языке ассемблера as88:

```
_PRINTF = 127
.SECT .TEXT
        PUSH        42
        PUSH        fmt
        PUSH        _PRINTF
        SYS

.SECT .DATA
fmt:    .ASCIZ       "%d\n"
```

В этом примере показан способ объявления строковых литералов, завершающихся нулём (в стиле C) — с помощью псевдокоманды ассемблера `.ASCIZ`. При использовании `_PRINTF` следует помнить о том, что это функция, она возвращает количество выведенных символов в регистре `AX` (который, таким образом, «портится»).

Задача 1 (task-1.s). Составьте программу, которая печатает на консоль числа от 1 до 10 через пробел, а затем печатает символ перехода на новую строку (`\n`).

Указания.

а) Для организации цикла использовать условные переходы `J**`, а не `LOOP`.

б) В конце программы добавить код завершения программы (как в первом примере с `_EXIT`).

в) Выполнить программу с помощью t88, а затем s88. ♢

Задача 2 (task-2.s). При выполнении предыдущей программы в t88 видно, что стек сильно захламляется в цикле — это некорректное использование стека: считается, что после того, как данные, помещённые на стек, использованы, их следует удалить со стека. Простой и неэффективный способ очистки стека — использование инструкции `POP`. Более разумный и широко используемый способ очистки стека — непосредственное манипулирование адресом вершины стека, который хранится в регистре `SP`. Для этого следует знать одну особенность устройства стека в интеловских процессорах: «стек растёт вниз». Это означает, что при добавлении элементов на стек адрес вершины стека уменьшается. И наоборот: при снятии элементов со стека адрес вершины стека увеличивается. В данном случае для снятия трёх двухбайтовых значений следует увеличить значение адреса вершины стека (регистр `SP`) на 6 ($= 3 \times 2$).

Добавьте инструкцию увеличения `SP` на 6 сразу после системного вызова, выполняемого в цикле для печати чисел. Проследите с помощью t88, что захламления стека теперь не происходит. ♢

Задача 3 (task-3.s). Постоянное добавление-снятие элементов с вершины стека, как в прошлой программе, не вполне эффективно, если учесть, что меняется лишь третий аргумент системного вызова (печатаемое число). Этот аргумент можно менять, используя косвенную адресацию на стеке — для неё используется регистр `BP` (в отличие от косвенной адресации в сегменте данных, для которой используются `BX`/`SI`/`DI`).

Указания.

а) Скопируйте решение предыдущей задачи, вынесите код добавления на стек аргументов и код очистки стека за границы цикла (до и после, соответственно).

б) Перед циклом занесите значение вершины стека в регистр `BP` для последующей косвенной адресации на стеке.

в) На каждом шаге цикла записывайте в третий аргумент на стеке очередное число для печати. Для этого используйте косвенную адресацию со сдвигом: `N(BP)` вместо обычной косвенной адресации `(BP)`. Здесь `N` означает необходимый сдвиг относительно вершины стека. Первый аргумент лежит на вершине стека — `(BP)`, второй аргумент — со сдвигом два, то есть обратиться к нему можно так: `2(BP)`, третий аргумент — со сдвигом четыре, обращение: `4(BP)`. ◊

Задача 4 (task-4.s). В прошлой задаче происходило обращение к одному и тому же (третьему с вершины) элементу на стеке. Постоянно указывать сдвиг в этом случае не вполне эффективно: проще сразу же, до начала цикла, увеличить значение `BP` на четыре, а внутри цикла использовать обычную косвенную адресацию `(BP)` без сдвига. Выполните это преобразование программы. ◊

3.2 Простейшие подпрограммы

Чтобы создать программу, использующую подпрограмму (процедуру или функцию), нужно должным образом организовать две части программы: код вызова подпрограммы (из основной программы) и код самой подпрограммы.

Код вызова похож на код осуществления системного вызова: необходимые параметры (обычно) складываются на стек, а затем используется инструкция

```
        CALL      <имя_метки_подпрограммы>
```

В этом месте происходит непосредственно вызов подпрограммы. После её окончания продолжается выполнение инструкций, следующих за CALL. Обычно первая такая инструкция выполняет очистку стека от аргументов (например, ADD SP, < количество аргументов * 2 >).

Для понимания того, как должна быть устроена подпрограмма, следует разобраться в механизме работы CALL. По сути CALL совершает безусловный переход по заданной метке, как JMP. Однако если бы этим действие CALL ограничивалось, то невозможно было бы вернуться из подпрограммы обратно в место вызова (сделать «обратный J(u)MP»). Чтобы было ясно, куда возвращаться, CALL кладёт на стек адрес следующей за ней инструкции. Это необходимо учитывать при доступе к аргументам подпрограммы: они оказываются на 2 байта глубже, чем хотелось бы, из-за лежащего на вершине стека адреса возврата (который разместила там CALL).

Теперь разберёмся, как определяется подпрограмма. В самом простом случае она размещается в том же файле, что и основная программа, после инструкций, составляющих основную программу, которые завершаются системным вызовом _EXIT.

Подпрограмма начинается с метки, которая будет использоваться при вызове (CALL) — «имени подпрограммы». Далее в подпрограмме могут идти любые инструкции, но обычно первым делом выполняются два действия, предназначенные для организации доступа к аргументам подпрограммы. Этот доступ организуется с помощью косвенной адресации со сдвигом через регистр BP (как в задаче 3). Для этого в BP заносится текущий указатель на вершину стека (также аналогично задаче 3). Однако сделать это первым же действием нельзя, так как процедура может быть вызвана из другой процедуры, которая использует BP для доступа к своим аргументам. Так что первым действием следует сохранить старое значение BP на стеке и только после этого заносить в BP текущее значение SP.

```
MY_PROC:
        PUSH    BP
        MOV     BP, SP
```

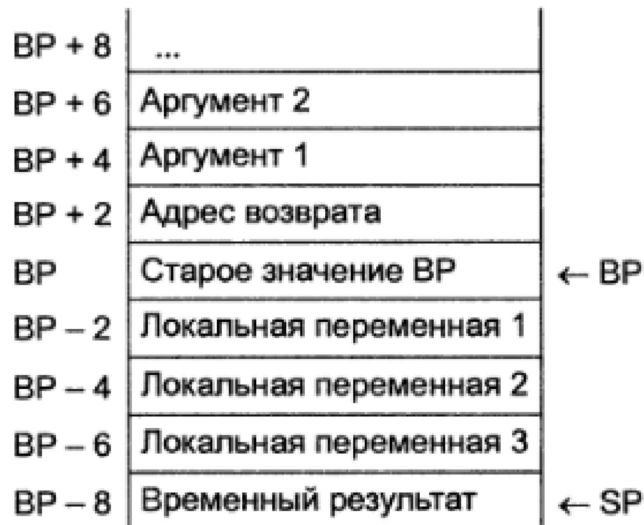
Такая последовательность действий называется прологом подпрограммы. В конце подпрограммы следует восстановить старое значение BP и выполнить инструкцию RET, которая снимет с вершины стека адрес возврата и передаст управление программой по нему. Если подпрограмме требуются какие-либо локальные переменные, они размещаются также на стеке, в таком случае перед завершением вершину стека следует переместить на то место, где она была после первой инструкции (PUSH BP). В этом случае завершение или эпилог подпрограммы выглядит так:

```
        MOV     SP, BP  ! можно опустить, если на вершине стека
                        !    и так находится старое значение BP
        POP     BP
        RET
```

Обратите внимание на зеркальное подобие пролога и эпилога. Очень важно представлять, что происходит со стеком перед вызовом подпрограммы, во время вызова и по её завершении. В течение работы подпрограммы стек выглядит примерно так, как показано на рисунке 3. Здесь стек изображён растущим вниз, что соответствует указанному ранее принципу «стек растёт вниз». Однако следует учесть, что в окне t88 стек показан растущим вверх.

Задача 5 (task-5.s). Оберните решение задачи 2 в подпрограмму PRINT_N, которая печатает числа от 1 до N. Число N передаётся как параметр на стеке. Следуя рисунку 3, при правильной организации подпрограммы доступ к N осуществляется так: 4(BP) (так как это первый и единственный аргумент). Основная программа (сразу после .SECT .TEXT) должна состоять из четырёх действий: положить на стек число 10 (значение параметра N), вызвать подпрограмму (CALL), очистить стек

Рисунок 3 — Кадр стека подпрограммы.



от числа 10, завершить программу системным вызовом `_EXIT`. После этого располагается текст подпрограммы: метка с именем, пролог, цикл печати из задачи 2 (условие окончания цикла немного меняется: счётчик сравнивается не с 10, а с `4(BP)`), эпилог. ◊

Задача 6 (task-6.s). В этой задаче вам предлагается создать простейшую функцию. Функция отличается от процедуры наличием возвращаемого значения. Есть несколько вариантов размещения возвращаемого значения подпрограммы. Самый простой и одновременно широко используемый вариант (который примем и мы) — помещать возвращаемое значение в регистр AX. Создайте функцию сложения двух чисел (аргументы передаются через стек). В основной программе вызовите функцию и распечатайте её результат. ◊

Дополнительные задачи

Задача 7 (task-extra-1.s). Добавление символа перехода на новую строку в подпрограмме из задачи 6 можно реализовать без помощи `printf`, воспользовавшись более простым системным вызовом 122, который моделирует работу функции `putchar`. Она имеет единственный аргумент:

символ, который необходимо вывести. Нужный символ можно класть на стек примерно так:

```
PUSH    'a'
```

(для размещения на стеке символа a). Организуйте печать символа перехода на новую строку с помощью системного вызова 122 (объявите для 122 именованную константу по аналогии с `_EXIT` и `_PRINTF`). ◊

Задача 8 (task-extra-2.s). Создайте функцию печати всех элементов данного массива (передаётся адрес массива и его длина в словах). ◊

Задача 9 (task-extra-3.s). Создайте функцию суммирования всех элементов данного массива. ◊

4 ПОДПРОГРАММЫ (ПРОДОЛЖЕНИЕ)

4.1 Конвенции вызова

При формулировке пролога и эпилога подпрограммы указывалось на необходимость сохранения «старого» значения регистра BP на стеке. Возникает вопрос:

а) Нужно ли сохранять значения других регистров и если да, то всех или некоторых?

На самом деле, можно задать ещё несколько похожих «инфраструктурных» вопросов:

б) Должны ли передаваться параметры через стек или через регистры, если последних хватает (ведь это быстрее)?

в) Если параметры передаются через стек, нужно ли складывать их туда в обратном порядке (как это делалось в прошлом разделе) или в прямом?

г) Где должен располагаться код очистки стека от аргументов подпрограммы, в вызывающем коде после вызова (как в прошлом разделе) или в самой подпрограмме в эпилоге (для этого существует специальная форма инструкции RET с одним целочисленным параметром, который указывает, сколько байтов надо снять в дополнение к снятому адресу возврата)?

Набор конкретных ответов на эти четыре вопроса называется *конвенцией вызова подпрограмм*. Существует несколько вариантов конвенций, каждая носит своё имя. Очень важно, чтобы подпрограммы, которые обычно пишутся одними людьми, и вызывающий код (в том числе в других подпрограммах), который пишется другими людьми, использовали одну конвенцию вызова. При использовании языков высокого уровня типа C на это можно влиять с помощью компилятора.

По умолчанию большинство компиляторов с языка C используют конвенцию, которая известна под именем *cdelc* (от C declaration), её мы и будем придерживаться далее:

а) Подпрограмма может «портить» регистры AX, CX, DX, значения остальных регистров она обязана сохранять (обычно по аналогии с сохранением BP для этого используется стек). Результат функций возвращается в AX. В этих условиях, если в вызывающем коде используются перечисленные регистры, перед вызовом любой подпрограммы их значения должны сохраняться (также на стеке).

б) Параметры всегда передаются через стек.

в) Параметры складываются на стек справа-налево (как показано в прошлом разделе, например, с функцией `printf`).

Примем ещё одно соглашение, которое стараются выдерживать в большинстве профессиональных проектов:

д) Подпрограмма не должна использовать глобальные переменные. Если подпрограмме нужны переменные для хранения промежуточных вычислений, она использует с этой целью стек.

Задача 1 (task-1.s). Создайте процедуру `SWAP`, которая получает адреса двух слов и меняет их местами. Помните о необходимости сохранять старые значения регистров, которые вы собираетесь использовать (кроме `AX`/`CX`/`DX`). ◊

Задача 2 (task-2.s). Создайте процедуру `SWAP_ARRAYS`, которая меняет местами содержимое двух массивов одинаковой длины поэлементно. Параметры процедуры: адреса двух массивов и их длина. Для перестановки пары элементов используйте процедуру `SWAP` из предыдущего задания. В основной программе используйте `SWAP_ARRAYS` для двух массивов, объявленных в секции данных. После этого распечатайте их с помощью процедуры печати массива из занятия 3 (task-extra-2.s). ◊

4.2 Переменное число параметров подпрограммы

Задача 3 (task-3.s). Создайте функцию `SUM`, которая принимает целое число N и складывает N чисел, которые лежат на стеке после N. В основной программе загрузите на стек массив и его размер и вызовите `SUM`. Результат распечатайте на консоль. ◊

4.3 Передача адресов подпрограмм в другие подпрограммы

Задача 4 (task-4.s). Создайте функцию `F`, которая вычисляет значение выражения $4x^4 - 9x^2 - 5$, x — параметр функции. Умножение/деление на степени двойки реализуются с помощью битовых сдвигов. ◊

Задача 5 (task-5.s на основе task-4.s). Создайте процедуру `MAP`, которая получает два массива одинаковой длины и функцию, которую она применяет к каждому элементу первого массива, а результат записывает в соответствующий элемент второго массива. Для передачи адреса применяемой функции через стек значение её метки нужно положить на стек, а внутри `MAP` вызвать эту функцию косвенно, указав в качества операнда `CALL` место на стеке, где лежит адрес этой функции (например `4(BP)`, если адрес функции передан в качестве первого аргумента `MAP`). В основной

программе продемонстрируйте работу `MAP` для произвольного массива и функции `F` из прошлой задачи. ♢

Дополнительные задачи

Задача 6 (task-extra-1-cdecl.s, task-extra-1-pascal.s). Создайте функцию `MAX` для определения наибольшего из двух аргументов в двух вариантах: один вариант с конвенцией вызова cdecl, а второй — который использует конвенции вызова pascal. Справку по последним можно получить в Википедии. Для второго случая вам понадобится использовать форму инструкции `RET` с целочисленным операндом, указывающим количество снимаемых со стека байт для очистки стека от аргументов. ♢

Задача 7 (task-extra-2.s). Создайте функцию `DIVIDES_2_TO_N`, которая в качестве аргументов принимает два целых числа, X и N, и возвращает 1, если X делится на 2^N, и 0 в противном случае. *Указание*: в цикле из не более чем N шагов следует делать сдвиг вправо на одну позицию и, если взведён флаг `CF`, то завершать цикл вместе со всей функцией, возвращая результат 0; если цикл пройдёт все N шагов и этот флаг ни разу не взведётся, то результат равен 1. ♢

5 ЦЕПОЧЕЧНЫЕ ИНСТРУКЦИИ

Отличительная черта CISC-архитектур, к которым относится и изучаемая нами архитектура x86, заключается в наличии большого числа специализированных команд, которые, с одной стороны, являются избыточными в том смысле, что их функции можно смоделировать с помощью «базовых» команд, однако, с другой стороны, такие команды облегчают жизнь программисту на ассемблере и иногда могут быстрее исполняться. По вопросу сравнения скорости исполнения цепочечных инструкций с обычными циклами можно проконсультироваться в [3], короткий ответ состоит в том, что

преимущество в скорости зависит от версии используемого процессора и размера обрабатываемых данных.

В данном разделе рассматриваются специализированные команды организации циклов для обработки последовательностей (или «цепочек», string) элементов размером в два или один байт, что обычно используется для обработки массивов целых чисел и строк соответственно. Приведём краткую сводку цепочечных инструкций.

а) Имена команд:

```
MOVS*, CMPS*, SCAS*, LODS*, STOS*
```

Каждая такая команда выполняет один шаг цикла с соответствующей семантикой (MOVS копирует, CMPS сравнивает соответствующие элементы цепочек, SCAS сравнивает элемент цепочки с эталоном и т.д.) и сдвигают указатель(и).

* — возможное окончание указывает на размер элемента цепочки. Если отсутствует или равно ‘w’, подразумевается слово (два байта), если окончание равно ‘b’, то работа происходит побайтово.

б) Итерации задаются префиксом REP*, например:

```
    REP MOVSB
```

Окончания: REPE/REPZ — выполнять пока равны/пока ноль, REPNE/REPNZ — наоборот.

Префикс REP* позволяет организовать цикл, в теле которого исполняется только одна цепочечная инструкция (действие + сдвиг указателя + уменьшение CX). В случае, если тело цикла должно быть сложнее, необходимо организовывать цикл самостоятельно (например, с помощью инструкции LOOP, как в задаче 3 ниже).

в) Длина цепочки задаётся в регистре CX до вызова цепочечной инструкции (и до префикса REP, если таковой имеется).

г) Расположение цепочек: SI, DI — источник (source) и приёмник (destination), а вернее, их адреса. Исходные значения также задаются заранее, до вызова цепочечных инструкций.

д) Направление обхода цепочек (флаг DF): DF=0 — в направлении увеличения адресов, DF=1 — уменьшения. Инструкция сброса флага — CLD (CLear Direction flag), взвода флага — STD (SeT Direction flag).

Более подробно см. [4, 2.3.8].

Во всех задачах ниже предполагается демонстрация созданных подпрограмм в основной программе. Кроме того, если явно не указано иное, все подпрограммы должны удовлетворять конвенциям вызова cdecl.

Задача 1 (task-1.s). Создайте программу, которая создаёт копию массива, объявленного в сегменте данных. Память для результата выделяется в секции BSS. ♢

Задача 2 (task-2.s). Создайте функцию `strlen` для подсчёта длины строки, завершающейся нулём. Алгоритм: обнулить AX, загрузить в CX значение -1, загрузить в DI смещение строки, повторять команду `scasb` (она ищет в строке значение из AL, то есть 0). Цепочечная команда увеличивает адрес DI и уменьшает CX. В конце CX равен отрицательному числу, которое по модулю на 2 больше, чем длина строки. Чтобы получить длину строки максимально быстро, можно использовать особенность представления отрицательных чисел в памяти: инвертирование битов отрицательного числа $-N$ даёт число $N-1$. После инвертирования битов CX (инструкция NOT) останется уменьшить его ещё на 1 (DEC). ♢

Задача 3. Создайте функцию печати данной строки в следующем виде: после каждой буквы печатается знак '!'. *Указания*: печать происходит с помощью PUTCHAR, цикл удобно организовать с помощью LOOP, на каждом шаге цикла делать LODSB (загрузка очередного символа в AL) и копирование AX в нужное место стека, где на вершине уже лежит код PUTCHAR для вызова SYS. ♢

Задача 4. Функция LAST_EVEN: найти последнее чётное число в данном массиве. *Указание*: использовать проход от конца массива к началу — инструкция STD для выставления флага прохода цепочки в обратном направлении. ♢

Дополнительные задачи

Задача 5. Создать процедуру FILTER, которая принимает две строки и предикат. Она записывает во вторую строку все символы первой, удовлетворяющие данному предикату. *Указания*: воспользоваться LODS/STOS, передача функций в качестве параметров функции обсуждалась в разделе 4 (пункт 3). Демонстрация — с предикатом проверки на то, что буква является гласной (a, e, i, o, u). ♢

6 РАБОТА С ФАЙЛАМИ

Ознакомьтесь с системными вызовами, отвечающими за работу с файлами, по рисунку 4 ([1, с. 777]): _OPEN/_CREATE/_CLOSE, _READ/_WRITE.

Задача 1 (task-1.s, создание бинарного файла). Напишите программу, которая создаёт бинарный файл task-1.dat с целыми числами от 1 до 10. Алгоритм:

а) создать файл с помощью _CREATE, указав в качестве параметров имя файла и число 0600 для задания прав доступа (смысл этого числа поясняется в курсе операционных систем, см. подробнее, например, [5]);

б) получить дескриптор fd созданного файла (возвращается после _CREATE в регистре AX);

в) в цикле двухбайтовая переменная-буфер получает очередное значение ($1, 2, \ldots$), системный вызов _WRITE записывает эти два байта в файл по дескриптору fd; помните, что системные вызовы обычно используют регистр AX для возвращаемого значения, потому нужные

Рисунок 4 — Системные вызовы ассемблера as88.

№	Имя	Аргументы	Возвращаемое значение	Описание
5	_OPEN	*name, 0/1/2	Дескриптор файла	Открытие файла
8	_CREAT	*name, *mode	Дескриптор файла	Создание файла
3	_READ	fd, buf, nbytes	# байт	Чтение n байт (nbytes) из буфера buf
4	_WRITE	fd, buf, nbytes	# байт	Запись n байт (nbytes) из буфера buf
6	_CLOSE	fd	0 в случае успешного выполнения	Закрытие файла с дескриптором fd
19	_LSEEK	fd, offset(long), 0/1/2	Положение (long)	Перемещение указателя файла
1	_EXIT	status		Закрытие файлов, остановка процесса
117	_GETCHAR		Чтение символа	Чтение символа в файле стандартного ввода
122	_PUTCHAR	char	Запись байта	Запись символа в файл стандартного вывода
127	_PRINTF	*format, arg		Запись с форматированием в файл стандартного вывода
121	_SPRINTF	but, *format, arg		Запись в буфере buf с форматированием
125	_SSCANF	buf, *format, arg		Чтение аргументов из буфера buf

программисту данные следует либо хранить в других регистрах, либо сбрасывать на стек;

г) закрыть файл с помощью системного вызова `_CLOSE` (в последующих задачах напоминаний об этом важном действии не будет).

Убедитесь, что содержимое файла task-1.dat совпадает с ожидаемым с помощью любого просмотрщика бинарных файлов. Варианты просмотра представлены ниже.

– Консольная утилита hexdump:

```
hexdump <имя_файла>
```

– Консольный файловый менеджер mc: F3 для просмотра выбранного файла, затем F4 для переключения в бинарный режим.

– Графическая утилита GHex. ♢

Задача 2 (task-2.s, чтение бинарного файла). Создайте программу, которая находит сумму первых десяти чисел в бинарном файле. *Указание*: использовать системный вызов `_OPEN` для открытия файла на чтение (аргументы: имя файла и число 0 для выбора режима чтения); использовать системный вызов `_READ` аналогично `_WRITE` из первой задачи. Для проверки можно использовать файл task-1.dat из первой задачи. ♢

Задача 3 (task-3.s, чтение с клавиатуры). Чтение с клавиатуры производится с помощью системного вызова `_READ` (см. задачу 2). В качестве дескриптора файла используется число 0 — дескриптор стандартного потока ввода (`_STDIN` — следует ввести и использовать именованную константу). После получения байтов, кодирующих символы, которые ввёл пользователь, обычно используется функция `_SSCANF`, позволяющая выделить из данной строки нужную информацию — обычно, целые числа. Указанная функция устроена так же, как `sscanf` в языке C.

Создайте программу, которая читает одно целое число с клавиатуры и выводит результат проверки его на чётность (слово Even или Odd). Алгоритм:

а) завести переменную-буфер `buf` из 6 символов;

б) прочитать с помощью `_READ` из `_STDIN` строку в `buf`;

в) вызвать `_SSCANF`, передав ей `buf`, формат "%d" и смещение (метку) переменной для получения результата;

г) проверить результат на чётность и вывести ответ. ♢

Задача 4 (task-4.s, преобразование файла). Создайте программу, которая по данному бинарному файлу целых чисел создаёт новый файл, содержащий только положительные числа из исходного файла в том же порядке. Для определения конца исходного файла проверяйте возвращаемое (в AX) значение _READ, оно равно количеству прочитанных байт; если значение равно нулю, значит файл закончился.

Дополнительные задачи

Задача 5. Дан бинарный файл целых чисел, обнулить в нём максимальный элемент. *Указание*: используйте _LSEEK и открытие файла на чтение и запись одновременно (_OPEN в режиме 2). ♢

7 МИКРОПРОГРАММИРОВАНИЕ

В данном разделе предлагается спуститься в изучении архитектуры вычислительных систем на один уровень ниже, чем уровень ассемблера. В процессорах с микропрограммным управлением каждая инструкция ассемблера (ADD, MOV, и т. д. в ассемблере x86) реализована с помощью набора микрокоманд, специфичных для микроархитектуры этого процессора. Соотношение между инструкциями процессора (это более корректное название, чем «инструкция ассемблера») и микропрограммой (совокупностью микрокоманд процессора) такое же, как между интерфейсом и реализацией. Например, набор инструкций x86 реализуется большим числом процессоров Intel и AMD, которые внутри (на уровне микроархитектуры) устроены по-разному. Соответственно, эти процессоры, если они снабжены микропрограммным, а не аппаратным управлением, будут иметь разные микропрограммы. Микропрограмма связана, с одной стороны, с реализуемым ею набором инструкций, с другой — с микроархитектурой конкретного процессора, в который она встроена на заводе-производителе.

Мы рассмотрим задачу реализации нескольких простых инструкций на конкретной (учебной) микроархитектуре Mic-1, описанной в [1, гл. 4]. Ядром микроархитектуры любого процессора является *тракт данных*, который, как правило, включает несколько регистров для хранения операндов микрокоманд и комбинационных схем — в простейшем случае, арифметико-логическое устройство — для осуществления операций над содержимым регистров. Рисунок 5 представляет тракт данных микроархитектуры Mic-1.

Перечислим назначение основных регистров тракта данных Mic-1.

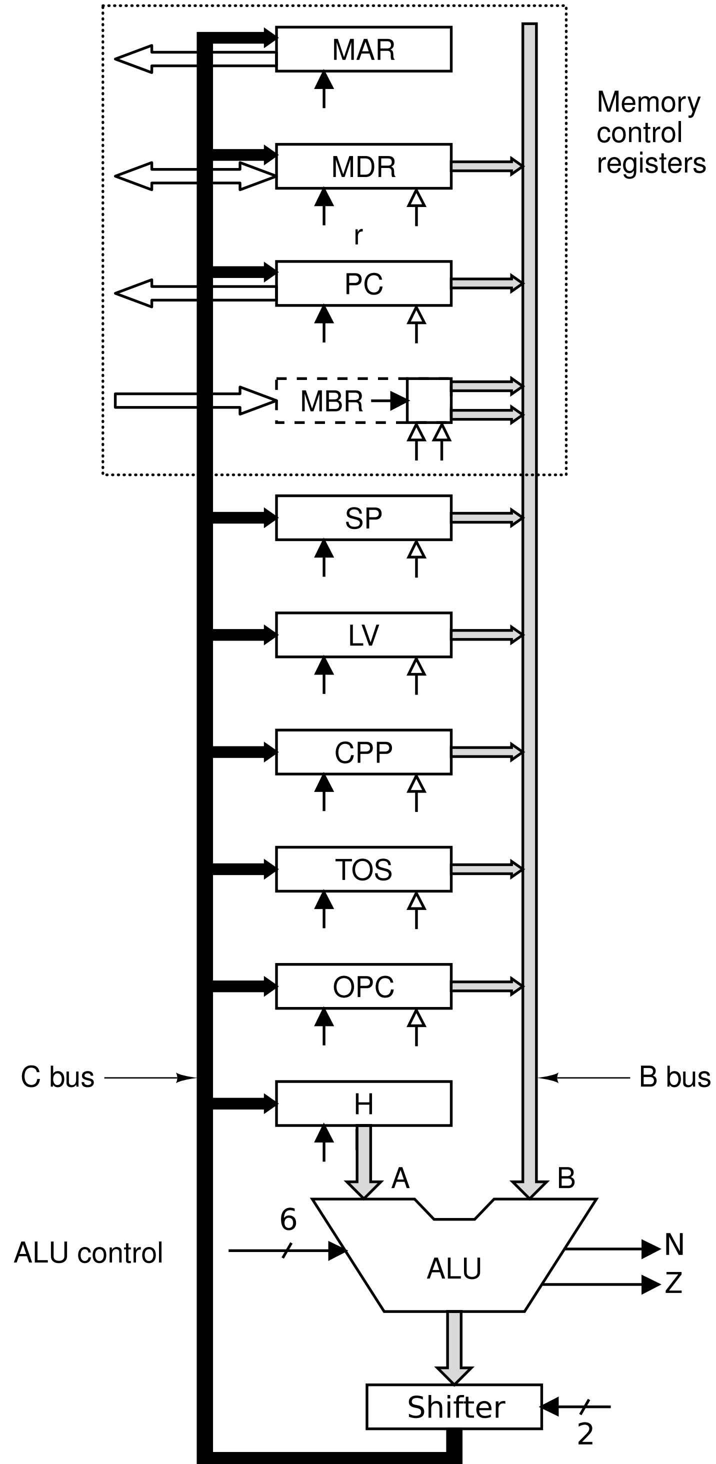
– SP — адрес вершины стека в памяти;

– TOS — значение вершины стека; то есть это значение хранится как в TOS, для удобства, так и в памяти по адресу SP; таким образом, при изменении вершины стека результат должен записывать как в TOS, так и в память по адресу SP;

– MAR — адрес в памяти, по которому следует сделать команду чтения (rd) или записи (wr); таким образом, при каждом чтении/записи следует предварительно заполнить MAR; самые частые случаи — когда требуется 1) записать новое значение вершины стека, тогда в MAR нужно записать SP, 2) прочитать слово ниже вершины стека, тогда в MAR следует записать SP - 1 (это же значение обычно следует записать и в SP, чтобы «вытолкнуть» из стека старую вершину);

– MDR — значение, которое следует записать в память или прочитать из памяти; если выполняется чтение, то прочитанное по адресу MAR значение появится в MDR только через полный цикл, то есть только через одну микрокоманду после того, как было запрошено wr; на следующем же после wr цикле ещё доступно старое значение MDR; такое поведение нужно учитывать, когда выполняется серия чтений из памяти (несколько команд wr по разным адресам);

Рисунок 5 — Тракт данных Mic-1



- H — сюда помещается левый операнд АЛУ для любой бинарной операции; учтите, что увеличение или уменьшение на 1 является унарной операцией и может быть выполнено без использования H;
- PC — адрес следующей инструкции или её операнда; практически всегда изменяется последовательно, увеличением на 1; для однобайтовых инструкций IJVM увеличение PC происходит в начале обработки инструкции стандартной микрокомандой Main1 и больше не требуется;
- MBR — значение следующей инструкции или первого байта операнда текущей инструкции.
- остальные регистры нам не понадобятся.

Микроархитектура Mic-1 предназначена для реализации конкретного набора инструкций, а именно, IJVM. Этот учебный набор инструкций, описанный в том же источнике, что и Mic-1 [1, гл. 4], является упрощённой версией набора инструкций виртуальной машины Java. Отметим некоторые *особенности набора инструкций IJVM*.

В отношении типов данных набор IJVM предоставляет средства для целочисленной арифметики и побитовой логики. В этом он схож с набором инструкций процессора Intel 8088, который изучался в предыдущих разделах. В IJVM нет поддержки ни чисел с плавающей точкой (как и в Intel 8088), ни других специализированных типов данных (к примеру, как цепочки в Intel 8088, изученные в разделе 6). Это максимально упрощает работу по реализации такого набора инструкций на микропрограммном уровне. IJVM является 32-разрядной архитектурой: и числа, над которыми выполняются арифметические действия, и адреса в памяти занимают по 32 бита.

В IJVM имеется поддержка вызова подпрограмм (по аналогии с инструкциями `call`/`ret` в x86, в том числе в Intel 8088), однако они не будут рассматриваться в дальнейшем.

Главное отличие IJVM от x86 состоит в том, что он моделирует *стековую машину*. Это выражается в первую очередь в способе передачи операндов для арифметических и прочих инструкций. Если в типичной *регистровой машине* x86 операнды могли передаваться через регистры или лежать в памяти по заданным адресам, то в стековых машинах, в том числе в IJVM, операнды инструкций должны передаваться на стеке. Обычно инструкция стековой машины снимает несколько аргументов со стека и кладёт на вершину результат операции.

7.2 Пример реализации инструкции IADD

Рассмотрим пример реализации инструкции `IADD`, входящей в набор IJVM, на микроархитектуре Mic-1. Данная инструкция снимает два 32-разрядных числа со стека и кладёт на вершину стека их сумму.

```
MAR = SP = SP - 1; rd
H = TOS
MDR = TOS = H + MDR; wr; goto Main1
```

Разберём первую микрокоманду, то есть первую строку. Поскольку вершина стека хранится в регистре `TOS`, достаточно загрузить из памяти лишь одно 32-разрядное слово, а именно то, которое располагается под вершиной («второе слагаемое»). Для этого в регистр `MAR`, отвечающий за адрес загружаемых из памяти слов, записывается адрес этого слова, то есть значение `SP - 1`. Новая вершина стека после завершения инструкции сложения будет находиться по этому же адресу — это легко вычислить: два слова-слагаемых снимаются со стека, одно кладётся, в итоге адрес уменьшается на единицу. Поэтому значение `SP - 1` сразу же записывается как новое содержимое адреса вершины стека `SP`. Синтаксис многократного присваивания здесь имеет тот же смысл, что и в языке C, то есть `A = B = C` означает `A = (B = C)`, после этого и `A`, и `B` равны `C`.

Следует отметить, что адреса в регистрах `MAR` и `SP` исчисляются в словах, а не в байтах, иначе значение `SP` следовало бы уменьшить на 4.

Первая микрокоманда завершается запросом (rd) к контроллеру памяти на чтение слова из оперативной памяти по адресу, только что записанному в MAR. Результат такого запроса в Mic-1 всегда приходит через один полный цикл в регистр MDR. В данном случае это означает, что второе слагаемое будет доступно в MDR только на третьей микрокоманде.

Вторая микрокоманда помещает одно из слагаемых, а именно вершину стека, хранившуюся в TOS, в регистр H. Необходимость этого действия становится очевидной после более пристального рассмотрения рисунка 5. На рисунке видно, что тракт данных Mic-1 устроен таким образом, что любая бинарная операция, выполняемая АЛУ, один из своих аргументов берёт в регистре H.

В третьей микрокоманде происходит собственно сложение. Операнды находятся в H (подготовлен второй микрокомандой) и MDR (результат запроса rd, выполненного первой микрокомандой). Сумма записывается в TOS (так как она становится новой вершиной стека), а также в MDR, посредством которого выгружается в память (напомним, что вершина стека хранится в двух местах: в регистре TOS на процессоре и в оперативной памяти по адресу, который содержится в регистре SP). Сама запись в память инициируется запросом wr, адрес для записи берётся из регистра MAR, в котором ещё со времени выполнения первой микрокоманды хранится нужное значение SP - 1.

В конце третьей микрокоманды происходит переход к началу цикла выборки инструкций, который отмечен меткой Main1. Каждая последовательность микрокоманд, реализующая одну инструкцию, должна завершаться таким переходом. Это позволяет процессору прочесть следующую инструкцию из оперативной памяти и перейти к последовательности микрокоманд, которые реализуют её. Такой цикл выполняется условно бесконечно.

7.3 Работа с симулятором Mic1MMV

В материалах [2] (каталог Mic1MMV) к учебнику [1], доступных онлайн, находится симулятор процессора с микроархитектурой Mic-1 под

названием Mic1MMV. Симулятор представляет из себя Java-приложение, поэтому его запуск из консоли должен выглядеть так:

```
java -jar Mic1MMV.jar
```

Симулятор имеет графический интерфейс и объединяет в себе несколько инструментов, а именно:

– ассемблер для IJVM с возможностью конфигурации (в частности, добавления новых инструкций);

– микроассемблер MAL, который транслирует код, аналогичный приведённому для IADD в предыдущем подразделе, в бинарный код;

– загрузчик бинарного кода микропрограммы;

– загрузчик бинарного кода, полученного от ассемблера IJVM, и непосредственно симулятор последовательного исполнения каждой инструкции в этом коде в соответствии с загруженной заранее микропрограммой.

Схему работы с симулятором можно представить графически (рисунок 6).

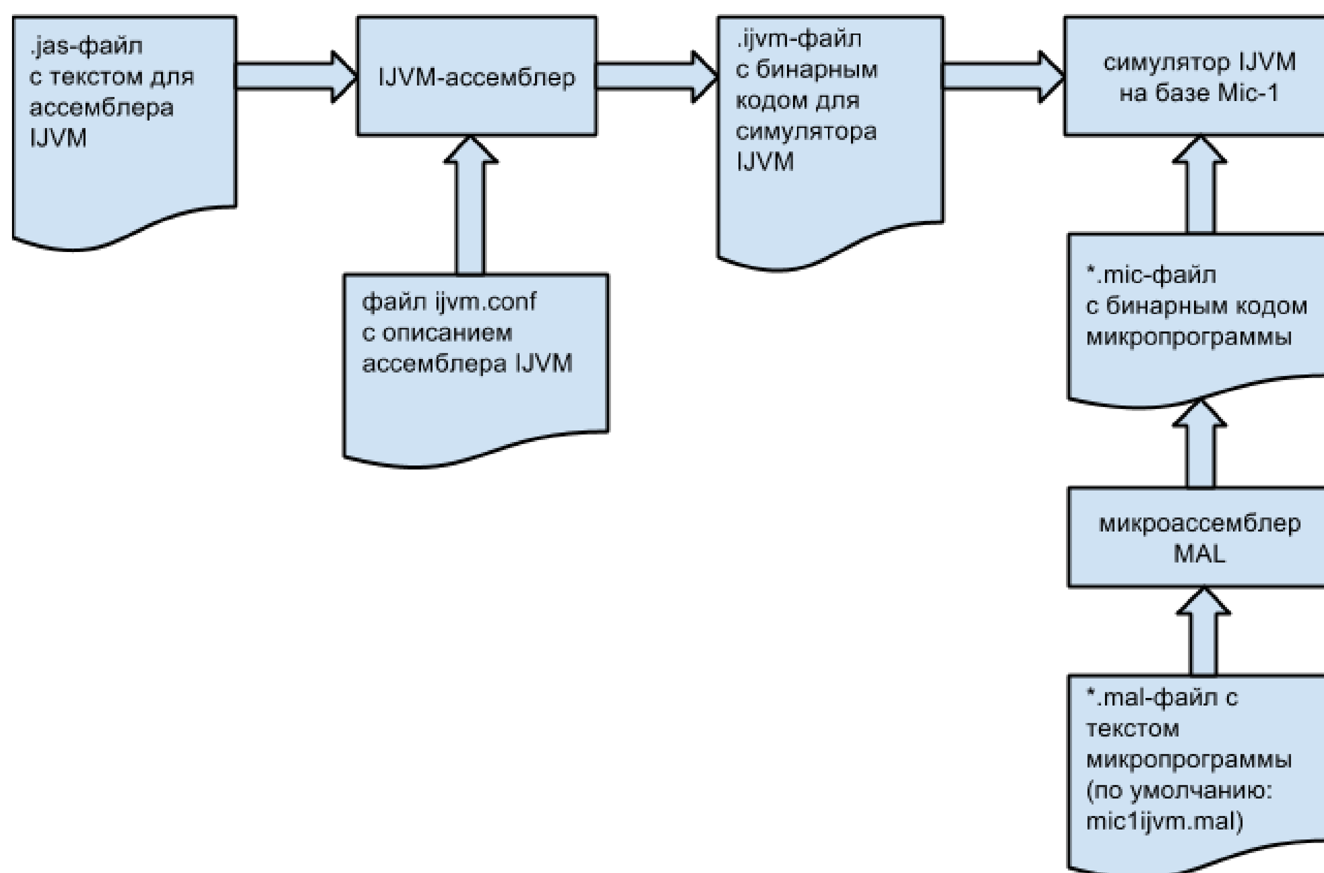
Полная микропрограммная реализация набора инструкций IJVM находится в [2] в файле mic1ijvm.mal: к ней рекомендуется обращаться для получения примеров использования тех или иных конструкций микропрограммирования. В данном разделе предлагается расширить набор инструкций IJVM, добавив в него несколько простых инструкций. Для добавления каждой инструкции следует выполнить ряд шагов.

а) Создать .jas-файл с программой (примеры будут даны в задачах) на ассемблере IJVM, где используется новая инструкция IJVM, которую нужно реализовать в данной задаче.

б) Добавить в описание ассемблера IJVM (файл ijvm-mmcs.conf) имя новой инструкции и её код (будет указан в задаче).

в) Добавить в микропрограмму (файл mic1mmv-mmcs.mal) последовательность микрокоманд, реализующих новую инструкцию.

Рисунок 6 — Схема работы с симулятором Mic1MMV



г) Сассемблировать и загрузить файл с модифицированной микропрограммой (mic1mmv-mmcs.mal, команда меню симулятора File → Assemble / Load MAL file), сассемблировать и загрузить файл с программой, созданной на первом шаге (команда меню симулятора File → Assemble / Load JAS file).

д) Запустить симуляцию (см. ниже).

У симулятора есть несколько режимов («скоростей» — найдите слово Speed в окне симулятора) работы, от выбора скорости зависит то, какое количество действий выполняется за одно нажатие синей стрелочки вправо:

– Subcycle — по фазе («подциклу») одного цикла (запуск шины B, запуск АЛУ, запуск шины C и запись в регистры — три вида подциклов внутри одного цикла);

– Cycle — по одной микрокоманде (одному «циклу»);

– IJVM — по одной инструкции процессора;

– Prog — непрерывно (до конца программы или до очередной операции ввода с клавиатуры).

К сожалению, переключение между скоростями не доступно по ходу выполнения программы. Поэтому разумный алгоритм проверки правильности решения на данном симуляторе выглядит так:

а) Первый запуск программы проводить на скорости Prog. Результат в выданных .jas-программах всегда будет помещаться на стек и в конце выполнения программы будет видно, правильное там оказалось значение или нет (вершина стека выделена красным в поле Stack Area). Если оно правильное, то можно решать следующую задачу.

б) Если значение на стеке неправильное, то нужно переключиться на скорость Cycle (или даже Subcycle) и выполнять программу по шагам, наблюдая за тем, какие выполняются микрокоманды (видно в поле под переключателем скоростей) и как изменяются значения регистров. Это позволит найти ошибку в вашей микропрограмме, хотя понадобится много нажатий на синюю стрелочку вправо.

в) В начале работы программы выполняются три стандартных цикла (Main1 | nop1 | Main1), которые позволяют подхватить из памяти первую инструкцию этой программы.

Задачи

Задача 1. Добавьте в набор инструкций инструкцию DEC (предлагаемый код: 0x16), которая уменьшает слово на вершине стека на единицу. Выполните необходимые действия для микропрограммной реализации этой инструкции (см. список из четырёх пунктов выше). Программа, использующая эту инструкцию, может выглядеть так:

```
.main
start:
    BIPUSH 0x43
```

```
    DEC
.end-main
```

Указания. Действуйте по аналогии с тем, что уже есть в .conf и .mal-файлах.

а) В .conf-файл добавляется имя новой инструкции (`DEC`) и её код.

б) В .mal-файле нужно вставить директиву `.label`, которая жёстко задаёт адрес микрокоманды с заданной меткой. Это делается для выполнения *соглашения*, используемого в данной микропрограмме: адрес первой микрокоманды для выполнения инструкции `X` должен совпадать с бинарным кодом `X`.

в) Алгоритм для микропрограммы: уменьшить содержимое `TOS` на `1` и результат записать в `TOS` и в `MDR` (в `MDR` — для будущего «сброса» в память: вершина стека хранится в двух местах, в `TOS` и в памяти по адресу `SP`). Загрузить в `MAR` значение `SP` и выдать запрос `wr`, чтобы записать новое значение вершины стека в память. В этом же месте сделать `goto Main1` (делается в конце каждого отрезка микрокоманд для каждой инструкции — это можно видеть в .mal-файле).

Реализацию данного алгоритма стоит поместить после реализации инструкции `IADD` в .mal-файле, оформлять по аналогии с ней. ◊

Задача 2. Добавьте в набор инструкций инструкцию `IADD3` (предлагаемый код: 0x20), которая берёт со стека три числа и кладёт на стек их сумму. Напишите программу, использующую эту инструкцию, самостоятельно по аналогии с программой из первой задачи.

Указания. Действуйте по аналогии с реализацией `IADD`.

а) Имя метки первой микрокоманды удобно выбрать таким: `iadd31`. Последующие инструкции отмечать `iadd32`, `iadd33` и т. д. (повторим: см. реализацию `IADD`, которая выполнена командами `iadd1`, `iadd2`, `iadd3`, свою реализацию разместите под `iadd3` через пустую строку).

б) Поскольку требуется три числа со стека, одно из которых (вершина) уже есть в регистре `TOS`, то потребуется два чтения из памяти (`rd`).

в) Решение этой задачи может уложиться в пять микрокоманд без пустых циклов ожидания данных из памяти. Для этого нужно вспомнить, что чтение из памяти (запрос `rd`) занимает один полный цикл. Это, в частности, означает, что на следующем после вызова `rd` цикле ещё можно использовать старое значение MDR. ♢

Задача 3. Добавьте в набор инструкций инструкцию `UMUL` (выберите любой свободный код инструкции), которая перемножает два слова, лежащие на стеке, и кладёт результат на стек в предположении, что исходные два слова представляют неотрицательные числа.

Указания.

а) Умножение реализуется через многократное сложение.

б) Используйте две инструкции типа `goto label`. Примеры даны в тексте микропрограммы.

в) Для организации цикла следует использовать микрокоманду безусловного перехода по метке `goto метка`. Условие остановки реализуется с помощью микрокоманды:

```
Z = регистр; if (Z) goto конец_цикла; else goto начало_цикла
```

Что означает: если указанный регистр содержит нуль, то перейти по метке конца цикла, иначе — по метке начала цикла. ♢

Задача 4. Добавьте в набор инструкций инструкцию `POPN` с одним однобайтовым аргументом, эта инструкция должна снимать со стека `N` слов, где `N` — это значение аргумента. Составьте jas-программу, которая использует эту инструкцию — например, кладёт три значения на стек, а потом вызывает POPN 3. Идеи для микропрограммной реализации стоит брать

из реализации POP и реализации любой инструкции с однобайтовым параметром, например, BIPUSH. ♢

Дополнительные задачи

Задача 5. Реализуйте инструкцию-префикс REP, которая ставится перед любой другой инструкцией X без аргументов и повторяет инструкцию X количество раз, заданное значением на вершине стека.

Указания. Используйте в качестве примера реализацию префикса WIDE из полного текста микропрограммы. ♢

Задача 6. Разберите пример сложения двух чисел, введённых с клавиатуры, из архива с материалами к учебнику [1] (файл add.jas в каталоге examples). Модифицируйте его так, чтобы вводилось три числа и сложение выполнялось с помощью инструкции IADD3, а серии инструкций DUP и OUT сократите с помощью префикса REP. ♢

БИБЛИОГРАФИЯ

[1] Таненбаум Э., Остин Т. Архитектура компьютера / 6-е изд.(+CD) — СПб.: Питер, 2013. — 816 с.

[2] Structured Computer Organization, 6/E [Электронный ресурс] http://www.pearsonhighered.com/educator/product/Structured-Computer-Organization-6E/9780132916523.page

[3] StackOverflow: Performance of x86 rep instructions on modern (pipelined / superscalar) processors [Электронный ресурс] http://stackoverflow.com/q/8425022/465100

[4] Зубков С.В. Assembler. Для DOS, Windows и Unix. 2-е изд. / М.:ДМК — 2000.

[5] Права доступа к файлам в Unix-подобных операционных системах [Электронный ресурс] http://younglinux.info/rwx