# Глава 4. Форматы команд и данных

*Компьютеры – это не только усилители человеческого разума. В той же мере они усиливают и человеческую глупость.*
*Ю.И. Манин*

В этой главе вводятся основные определения и рассматриваются важные базовые понятия, которые будут необходимы для дальнейшего изучения архитектуры ЭВМ. С этой целью будут построены ещё несколько различных учебных машин, они как бы кратко повторяют историю начального этапа развития вычислительной техники.

## *4.1. Адресность ЭВМ*

*Программировать – значит понимать.*
*Кристин Нюгард*

Число адресов в машинной команде называется адресностью ЭВМ. Разнообразие архитектур ЭВМ предполагает, в частности, и разную адресность их систем команд. Рассмотрим схему выполнения команд на компьютерах с различным числом адресов операндов. По сравнению с изученной ранее ЭВМ УМ-3 несколько увеличим аппаратные возможности (объём памяти и число возможных кодов операций), чтобы приблизить эти параметры к характеристикам «настоящих» машин второго поколения.

Будем предполагать, что для хранения кода операции в команде учебной машины отводится один байт (8 бит, что достаточно для представления 256 различных кодов операций), а для хранения каждого из адресов – по 3 байта, что обеспечивает доступ к объёму адресуемой памяти в $2^{24}$ (примерно 16 миллионов) байт.

При разработке архитектур ЭВМ определились две тенденции по вопросу о том, сколько различных команд должен уметь выполнять процессор ЭВМ. Одни разработчики считают, что все полезные команды надо стараться реализовать в языке машины, говорят, что такие ЭВМ имеют архитектуру CISC (Complex Instruction Set Computer). Компьютеры этой архитектуры имеют очень богатый язык машины, например, современные ЭВМ фирмы Intel могут выполнять около 1500 различных команд. Другие разработчики полагают, что процессор должен уметь выполнять совсем немного команд (2-3 десятка), что позволит сделать его компактным, быстрым и дешёвым. ЭВМ такой архитектуры называют RISC (Complex Instruction Set Computer) машинами, в основном эти процессоры устанавливаются в смартфоны и планшеты.

Ниже приведены форматы команд для ЭВМ различной адресности и схемы выполнения этих команд для случая бинарных (двуместных) операций (у таких операций два операнда и один результат, это, например, привычные для нас арифметические операции сложения, вычитания, умножения и деления). Как и ранее, для обобщённого обозначения любой бинарной операции будет использоваться символ ⊗.

- **Трёхадресная машина**

Трёхадресные команды являются наиболее естественными для человека при составлении программы на машинном языке, так как большинство элементарных шагов в вычислительных алгоритмах являются именно бинарными операциями (т.е. имеют два аргумента и один результат). Пусть команда в новой трёхадресной учебной машине имеет такой формат:

| КОП | A1 | A2 | A3 |
|---|---|---|---|
| 8 разрядов | 24 разряда | 24 разряда | 24 разряда |

= 10 байт

Это модификация уже знакомой Вам трёхадресной ЭВМ УМ-3, однако, теперь длина каждой команды увеличена до 10 байт. Длина целых и вещественных чисел по-прежнему будет 32 бита или 4 байта, кроме того в дальнейшем нужно будет предусмотреть хранение в памяти символов, которые занимают по одному байту. Итак, здесь команды и обрабатываемые данные имеют разную длину, что типично для современных ЭВМ, поэтому предполагается, что память состоит из коротких ячеек длиной в один байт, при этом *адресом* команды или числа будет адрес её первой ячейки (байта).

Схема выполнения команд такой машины её процессором нам уже известна, она такая же, как и на старой учебной ЭВМ УМ-3:[1]

```
R1 := <A2>; R2 := <A3>; S := R1 ⊗ R2;
<A1> := S; { ⊗ – любая бинарная операция }
```

В сокращенном виде эту схему можно записать как `<A1>:=<A2>⊗<A3>`, здесь всего три обращения в память.

- **Двухадресная машина**

Сократим теперь число адресов в машинной команде с трёх до двух и мысленно построим двухадресную учебную машины (по аналогии с предыдущей учебной машиной назовём её УМ-2). Длина каждой команды для двухадресной ЭВМ будет равна 7 байт:

| КОП | A1 | A2 |
|---|---|---|
| 8 разрядов | 24 разряда | 24 разряда |

= 7 байт

Схема выполнения бинарных двухадресных команд в процессоре нашей машины УМ-2 будет такой:

```
R1 := <A1>; R2 := <A2>; S := R1 ⊗ R2; <A1> := S
```

В сокращенном виде эту схему можно записать как `<A1>:=<A1>⊗<A2>`, здесь тоже три обращения в память. Заметим, что теперь, так как адресов у нас только два, для выполнения бинарной операции первый и второй операнды задаются в команде *явно* в качестве адресов, а местоположение результата задаётся *неявно* или, как говорят, *по умолчанию*. В рассмотренном выше случае двухадресных команд результат бинарной операции по умолчанию помещается на место первого операнда, уничтожая его старое значение. Оправданием такого «странного» выполнения команд может служить тот факт, что многие операторы языков высокого уровня выполняются именно по такой схеме, например:

```
N := N+1; S := S + X[i]; X := X * Y; и т.д.
```

К сожалению, бинарную операцию, например, `z:=x+y`, теперь приходится программировать тремя двухадресными командами (всего шесть обращений в память):

| | |
|---|---|
| ПЕР R x | рабочая переменная R:=x |
| СЛ R y | R := x+y |
| ПЕР z R | z := x+y |

Двухадресные команды имеют примерно на 30% меньшую длину, чем трёхадресные, но менее удобны для программиста, так как *затирают первый операнд*. Как только отпала необходимость экономить память, архитекторы стали чаще использовать трёхадресные команды. Например, большинство старых команд в процессоре Intel двухадресные, а новые команды векторной арифметики трёхадресные и даже четырёхадресные (см. главу 17).

- **Одноадресная машина**

При дальнейшем сокращении числа адресов в машинной команде получится уже одноадресная учебная машина УМ-1, команды которой имеют такую структуру:

| КОП | A2 |
|---|---|
| 8 разрядов | 24 разряда |

= 4 байта

Длина каждой команды этой машины равна 4 байта. Схема выполнения одноадресных команд будет такой:

```
R2 := <A2>; S := S ⊗ R2
```

В сокращенном виде эту схему можно записать как `S:=S⊗<A2>`, здесь всего одно обращение в память! Как видим, при выполнении бинарных операций в одноадресной ЭВМ уже только один *второй* операнд A2 задаётся в команде явно, а первый операнд и результат операции задаются неявно – это регистр сумматора S в арифметико-логическом устройстве.

---

[1] В схемах выполнения команд разной адресности пока не будет учитываться, что некоторые команды могут также вырабатывать признак результата (для УМ-3 этот признак помещался в регистр ω).

Для работы в одноадресной машине необходимы ещё две специальные одноадресные команды, которые имеют один операнд и один результат. Эти команды реализуют унарные (одноместные) операции. У нас это команда чтения операнда из памяти на регистр сумматора:

```
СЧ A1
```

Она выполняется по схеме

```
S := <A1>
```

и команда записи значения из сумматора в память:

```
ЗП A1
```

Она выполняется по схеме

```
<A1> := S
```

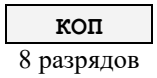
Заметим, что бинарную операцию, например, `z:=x+y`, приходится, как и в двухадресной ЭВМ, программировать тремя одноадресными командами (всего три обращения в память):

| | | |
|---|---|---|
| СЧ | x | S := x |
| СЛ | y | S := x+y |
| ЗП | z | z := S :=x+y |

Надо отметить, что в некотором смысле одноадресная архитектура самая простая для реализации. Например, здесь уже не нужен регистр R1 для первого операнда и каждая команда обращается в память только один раз. Поэтому не случайно многие первые ЭВМ, построенные в соответствии с принципами фон Неймана, были именно одноадресными машинами. Например, это машина EDSAC, построенная в 1949 году при участии А. Тьюринга. Отметим также, что по этой же схеме работают и хорошо знакомые нам калькуляторы (в смартфонах и т.д.).

- **Безадресная машина**

Логическим завершением процесса уменьшения числа адресов в команде является построение нульадресной (или безадресной) учебной ЭВМ, назовём её УМ-0. Большинство команд этой машины состоят только из кода операции и имеют длину один байт:




= 1 байт

В отличие от других рассмотренных выше учебных машин, новая безадресная машина кроме основной памяти, в которой, как обычно, хранится программа и данные, использует при своей работе также аппаратно реализованный в компьютере *стек* для хранения чисел (операндов).[1] Для обмена данными между основной памятью и стеком в язык машины вводятся две дополнительные *одноадресные* команды длиной по 4 байта. Это команда записи (добавления) машинного слова в стек из любой ячейки памяти с адресом A1

```
ВСТЕК A1
```

которая выполняется по схеме

```
R1 := <A1>; ВСТЕК(R1)
```

и команда чтения машинного слова из вершины стека в ячейку основной памяти (как обычно, при чтении машинное слово удаляется из стека)

```
ИЗСТЕКА A1
```

которая выполняется по схеме

```
R1 := ИЗСТЕКА; <A1> := R1
```

Таким образом, в этой архитектуре команды языка машины имеют разную длину. Вообще говоря, кроме указанных выше одноадресных команд записи в стек и чтения из стека, для удобства программирования в безадресной ЭВМ могут добавляться и некоторые другие одноадресные команды (например, команды безусловного и условного переходов, команда вызова процедуры и другие). В отличие от команд записи из памяти в стек и чтения из стека в память, остальные одноадресные команды являются *избыточными*. Так, например, одноадресную команду перехода на ячейку с адресом A1 можно заменить двумя последовательными командами: одноадресной командой записи в стек

[1] Обычно стек не является отдельной памятью, а располагается на части основной (оперативной) памяти.

значения адреса перехода A1 (в виде целого числа) и безадресной командой перехода по адресу, записанному в вершине стека.

Таким образом, за исключением этих двух одноадресных команд, все остальные команды являются безадресными и имеют длину 1 байт. Бинарные операции выполняются по схеме:

```
R2 := ИЗСТЕКА; R1 := ИЗСТЕКА; S := R1 ⊗ R2; ВСТЕК(S)
```
[1]

Как видно, для безадресных команд при выполнении бинарных операций уже все аргументы (два операнда и результат) задаются неявно и располагаются в стеке. Отсюда понятно, почему часто машины этой архитектуры называются *стековыми* ЭВМ.[2] На первый взгляд может показаться, что в стековых ЭВМ нарушается принцип фон Неймана однородности памяти, так как в стеке возможен доступ только к его вершине. Однако это не так: сам стек обычно является частью основной памяти, поэтому к любой ячейке стека возможен также и прямой доступ, т.е. команды `ВСТЕК A1` и `ИЗСТЕКА A1` в качестве своих операндов A1 могут указывать адрес произвольной ячейки памяти, в том числе и в любом месте самого стека.

В безадресной ЭВМ бинарную операцию, например, `z:=x+y`, приходится программировать четырьмя командами (всего шесть обращений в память):

| | |
|---|---|
| ВСТЕК y | Поместить y в стек: y |
| ВСТЕК x | Поместить x в стек: x, y |
| СЛ | В стеке: x+y |
| ИЗСТЕКА z | Запись результата в z; стек ПУСТ |

Кроме рассмотренных выше видов машин, существовали и архитектуры ЭВМ с другим числом адресов. В качестве примера упомянем *четырёхадресные* машины, в четвёртом адресе которых дополнительно хранился ещё и адрес *следующей* выполняемой команды (для таких ЭВМ вообще не нужны команды переходов). Собственно, адресов может быть и больше, с помощью таких команд можно, например, реализовать уже не бинарные операции, а функции от 3-х и более переменных. В компьютерах фирмы Intel есть четырёхадресные команды для работы с данными на векторных регистрах.

Далее, существуют архитектуры ЭВМ, которые различаются не только количеством *адресов* в машинной команде, но и наличием в такой команде нескольких *кодов операций*. Такие ЭВМ обычно называются машинами с *очень длинным командным словом* (VLIW – Very Large Instruction Word). Заметим, что, несмотря на необычность такой архитектуры, она может удовлетворять всем принципам фон Неймана. В таких компьютерах, например, некоторые команды могут реализовывать операторы присваивания вида `x:=a*b+c` по схеме:[3]

```
R1 := <a>; R2 := <b>; S := R1*R2;
R1 := <c>; S := S+R1; <x> := S
```

В ЭВМ с такой архитектурой команда, содержащая два кода операции и четыре адреса аргументов, в наших предыдущих предположениях о размере адреса и кода операции, имеет длину 14 байт и, например, такой формат:

| КОП1 | КОП2 | A1 | A2 | A3 | A4 |
|---|---|---|---|---|---|

Можно сказать, что компьютеры этой архитектуры как бы связывают несколько последовательных команд программы в одну длинную команду, в современных ЭВМ такая связка, иногда называемая пучком (bundle) команд, может содержать до семи кодов операций и соответствующее число операндов, а длина команды может доходить до 64 и более байт. Обычно построение таких длинных машинных команд производит компилятор с языка программирования высокого уровня для эффективного вычисления выражений. Главная идея VLIW архитектуры заключается в том, что на языке машины задаётся явный параллелизм: все операции над данными в такой длинной команде можно

---

[1] В отличие от других архитектур, при обмене с памятью команды чтения из стека и записи в стек могут вызывать аварийную ситуация (попытка чтения из пустого стека и переполнение стека).

[2] Для продвинутых учащихся. Отметим, что семантика таких известных языков высокого уровня, как Java и Forth близка именно к стековой архитектуре исполнителя.

[3] Такой набор команд под названием FMA (Fused Multiply-Add) для работы с *вещественными* числами реализован в процессорах фирм Intel и AMD, начиная с 2013 года.

выполнять параллельно (в разных частях АЛУ), так как это *одна* команда.

Можно понять, что компьютеры с архитектурой VLIW будут наиболее эффективны при проведении научных расчётов в таких областях, как линейная алгебра, газовая динамика, физика твёрдого тела и т.д. Первый проект VLIW-компьютера был разработан А. Тьюрингом ещё в 1946 году (он так и не был реализован). Среди наиболее известных построенных в своё время компьютеров этой архитектуры для любознательных читателей можно назвать ЭВМ Multiflow и Cydra-5 (длина команды 16 байт), CDC6600 и специализированный процессор TriVedia. Среди отечественных разработок можно отметить машину М-10 М.А. Карцева (1973 год) и серия ЭВМ Эльбрус В.С. Бурцева. Далее архитектура VLIW получила своё дальнейшее развитие в компьютерах так называемой EPIC архитектуры (Explicitly Parallel Instruction Computing – вычисление с явно заданным параллелизмом команд).

## 4.2. Сравнительный анализ ЭВМ различной адресности

*Человек без адреса подозрителен, человек с двумя адресами – тем более.*

*Бернард Шоу*

При изучении ЭВМ с разным количеством адресов естественно встаёт вопрос, какая архитектура лучше, например, даёт программы, занимающие меньше места в памяти. Заметим, что этот критерий для первых ЭВМ с их маленькой памятью был более важным, чем, например, удобство программирования на языке машины или скорость выполнения операций. Исследуем этот вопрос, составив небольшой фрагмент программы для рассмотренных выше учебных ЭВМ с различной адресностью. В качестве примера рассмотрим оператор присваивания, который содержит типичный набор арифметических операций:

$x := a/(a+b)^2$

В наших примерах будем использовать мнемонические коды операций и мнемонические имена для номеров ячеек памяти, в которых хранятся переменные (т.е. не будем производить *явного* распределения памяти, так как это несущественно для данного исследования). Кроме того, не будем конкретизировать тип используемых величин (целые или вещественные), предположим, что это тоже не влияет на размер программы. Вам необходимо внимательно просмотреть текст этих небольших фрагментов программ, и понять, как они работают.

- **Трёхадресная машина УМ-3**

| | |
|---|---|
| СЛ x a b | x := a+b |
| УМН x x x | $x := (a+b)^2$ |
| ДЕЛ x a x | $x := a/(a+b)^2$ |

Длина этой программы в байтах: (3 команды)*10 байт = 30 байт.

- **Двухадресная машина УМ-2**

| | |
|---|---|
| ПЕР R a | R := a; копия a в рабочую переменную R |
| СЛ R b | R := a+b |
| УМН R R | $R := (a+b)^2$ |
| ПЕР x a | x := a |
| ДЕЛ x R | $x := a/(a+b)^2$ |

Длина этой программы: (5 команд)*7 байт = 35 байт (плюс одна рабочая переменная R).

- **Одноадресная машина УМ-1**

| | |
|---|---|
| СЧ a | S := a |
| СЛ b | S := a+b |
| ЗП x | x := a+b |
| УМН x | $S := (a+b)^2$ |
| ЗП x | $x := (a+b)^2$ |
| СЧ a | S := a |
| ДЕЛ x | $S := a/(a+b)^2$ |
| ЗП x | $x := a/(a+b)^2$ |

Длина этой программы: (8 команд)*4 байта = 32 байта. Как видим, длина программы для одноадресной ЭВМ получилась примерно такая же, как и для трёх и двухадресных машин. В то же время

легко понять, что процессор одноадресной ЭВМ будет устроен проще, так как схема выполнения каждой команды требует меньше служебных регистров и реже обращается в память. Далее, простота процессора позволяет, при тех же затратах, увеличить скорость его работы за счёт реализации на более дорогих интегральных схемах. Отсюда понятна привлекательность одноадресной архитектуры для разработчиков ЭВМ первых поколений. Например, одноадресная отечественная ЭВМ БЭСМ-6 в конце 60-х годов прошлого века была одной из самых быстродействующих машин в мире [3].

- **Безадресная (нульадресная) машина УМ-0**

| | |
|---|---|
| ВСТЕК a | Поместить a в стек: a |
| ВСТЕК | Дублировать вершину стека: a,a |
| ВСТЕК b | Теперь в стеке 3 числа: b,a,a |
| СЛ | В стеке два числа: a+b,a |
| ВСТЕК | Дублировать вершину стека: a+b,a+b,a |
| УМН | В стеке два числа: $(a+b)^2$, a |
| ДЕЛ | В стеке одно число: $a/(a+b)^2$ |
| ИЗСТЕКА x | Запись результата в x; стек ПУСТ |

Отметим, что в безадресной ЭВМ есть и свои специфические безадресные «стековые» команды, например, ВСТЕК – дублировать вершину стека, ОБМЕН – поменять местами два верхних числа стека и др. В данной программе использовались команды разной длины: 3 одноадресные для обмена со стеком и 5 безадресных команд. Таким образом, длина всей программы:

(3 команды)*4 байта + (5 команд)*1 байт = 17 байт ⚠.

Безадресные ЭВМ реализуют так называемое постфиксное вычисление выражений, когда знак операции ставится *после* аргументов (такая запись выражений называемая ещё обратной польской записью – reverse Polish notation). Например, в постфиксной форме выражение (5+3)*(9-7) можно записать как 5 3 + 9 7 - *, оно будет вычисляться так: **5 3 +** 9 7 - * → 8 **9 7 -** * → **8 2 *** → 16.

Итак, с уменьшением количества адресов в команде увеличивается число команд программы, зато каждая команда становится более короткой, занимает меньше места в памяти. Наше небольшое исследование показало, что архитектура ЭВМ с безадресными командами даёт более компактные программы, кроме того, процессор у них устроен проще, чем у двух и трёхадресных ЭВМ.

Именно поэтому в начале развития вычислительной техники безадресные компьютеры были весьма распространены, их, в частности, выпускала известная фирма Барроуз (Burroughs) [3]. Безадресными были и отечественные ЭВМ Эльбрус-1 и Эльбрус-2, выпускавшиеся в 80-х годах прошлого века. Однако в дальнейшем были предложены ЭВМ с другой архитектурой, которая позволила писать не менее компактные машинные программы, и при этом обладала дополнительными достоинствами, поэтому в настоящее время *полностью* стековые ЭВМ практически не используются.

Кроме того, выпускались компьютеры и смешанной адресации. Так, уже упоминавшаяся ранее отечественная одноадресная ЭВМ БЭСМ-6, кроме одноадресных арифметических операций могла выполнять и аналогичные безадресные (стековые) операции.

Стековую организацию имеет и регистровая память для работы с вещественными числами в процессорах фирмы Intel. Вещественные регистры стека и обозначаются st(0)-st(7), где st(0) это нулевая позиция (вершина) стека. Для операции над этими числами используются стековые команды. Предыдущий пример для учебной машины УМ-0 на Ассемблере MASM будет выглядеть так:

```
; var a,b,x: Single;  После ; записывается комментарий
;   x:=a/(a+b)^2
a   dd   ?;    var a: Single;
b   dd   ?;    var b: Single;
x   dd   ?;    var x: Single;
    fld  a;     Поместить a в стек: a
    fld  st(0); Дублировать вершину стека: a,a
    fld  b;     Поместить b в стек: b,a,a
    fadd ;      В стеке два числа: a+b,a
    fld  st(0); Дублировать вершину стека: a+b,a+b,a
```

| | |
|---|---|
| **fmul** ; | В стеке два числа: $(a+b)^2, a$ |
| **fdiv** ; | В стеке одно число: $a/(a+b)^2$ |
| **fstp** x; | Запись результата в x; стек ПУСТ |

Здесь команды, обращающиеся в память, как правило имеют длину 6 байт, а безадресные команды – 2 байта.

Стековые исполнители алгоритмов весьма распространены. Например, компьютерные шрифты TrueType записаны как стековые программы на языке PostScript для «рисования» символов. Процессор PostScript встроен в аппаратуру большинства современных принтеров, а интерпретатор этого языка используется в операционных системах для вывода текста на экран.

Отметим, что сейчас стековые вычисления практически не используются в ЭВМ, так как они плохо соответствуют конвейерному принципу работы современных компьютеров (см. разд. 14.2).

## 4.3. Архитектура с адресуемыми регистрами

*Главный враг знания не невежество, а иллюзия знания.*

*Стивен Хокинг*

Далее будет рассмотрена архитектура ЭВМ с адресуемыми регистрами. Эти компьютеры дают возможность писать такие же компактные программы, как и ЭВМ с безадресной системой команд, но при этом обладать рядом дополнительных достоинств.

Компьютеры с адресуемыми регистрами нарушают принцип фон Неймана линейности и однородности памяти. В этих компьютерах память, к которой может непосредственно (по адресам) обращаться процессор за *операндами* команд («числами»), состоит из двух частей, каждая со своей независимой нумерацией ячеек (это и есть нарушение линейности памяти). Одна из этих частей называется *адресуемой регистровой памятью* и имеет небольшой объём (порядка десятков, редко сотен ячеек), а другая называется *основной (оперативной) памятью* большого объёма. Ячейка каждого из видов памяти имеет свой адрес, но в случае с маленькой регистровой памятью этот адрес имеет размер в битах в несколько раз меньший, чем адрес ячейки большой основной памяти. Кроме того, ячейки регистровой и основной памяти могут иметь разную длину.

### 4.3.1. Двухадресная машина с адресуемыми регистрами

*Возможность писать красивые программы даже на Ассемблере – именно это в первую очередь сделало меня приверженцем программирования.*

*Дональд Кнут*

Сейчас мы построим учебную двухадресную ЭВМ этой новой архитектуры, которую назовём УМ-Р (Учебная Машина с адресуемыми Регистрами). Пусть регистровая память этой учебной ЭВМ состоит из 16 ячеек по 4 байта каждая. В этом случае адрес каждого регистра лежит в диапазоне $0 \div 15$, и будет помещаться в 4 бита, а основная память пусть содержит $2^{20}$ ячеек, тогда адрес каждой ячейки занимает 20 двоичных разрядов. Команды сделаем двухадресными, в качестве адресов операндов могут быть или адреса двух регистров, или один адрес регистра и один адрес ячейки основной памяти. Заметим, что *команды* устройство управления по-прежнему может читать на регистр команд только из основной памяти. Адреса регистров на схемах команд будут обозначаться r1 и r2, а адреса основной памяти, как и раньше, A1 или A2. Первый вид команд будем называть командами формата *регистр-регистр* (коротко обозначается RR), а второй – формата *регистр-память* (обозначается RX). Вскоре выяснится, что третий логически допустимый формат команд (память-память), когда оба адреса принадлежат основной памяти, не даёт при программировании в этой архитектуре никаких преимуществ. Исходя из этого, такой формат в этой учебной машине (да и в большинстве «настоящих» машин этой архитектуры) мы не будем реализовывать.

В этом случае для одного кода операции (например, сложения) получаются команды двух форматов длины 2 и 4 байта соответственно:

| КОП | r1 | r2 |
|---|---|---|
| 8 бит | 4 бита | 4 бита |

= 2 байта

| КОП | r1 | A2 |
|---|---|---|
| 8 бит | 4 бита | 20 бит |

= 4 байта

В качестве преимущества этой архитектуры нужно отметить, что ячейки регистровой памяти, так как их немного, можно разместить *внутри* процессора (в арифметико-логическом устройстве, АЛУ), они будут иметь статус *регистров*. Это позволяет производить на них арифметические и логические операции (что, как Вы помните, в основной памяти по принципам фон Неймана это невозможно). Кроме того, такое расположение обеспечивает быстрый доступ к хранимым на регистрах данным (не требуется делать обмен с расположенной отдельно от процессора основной памятью). Как правило, время доступа к регистру у ЭВМ на порядок меньше, чем к ячейке основной памяти (это и есть нарушение принципа однородности памяти). Всё это позволяет существенно повысить быстродействие такого компьютера.

Требование обеспечить возможность выполнения на адресуемых регистрах всех операций над данными (сложения, вычитания, умножения, деления и т.д., да ещё и над целыми и над вещественными числами) сильно усложняет арифметико-логическое устройство ЭВМ. По существу, вместо одного регистра-сумматора для прежних архитектур, надо было реализовать столько таких сумматоров, сколько у нас адресуемых регистров. В первых моделях компьютеров фирмы Intel над восемью адресуемыми регистрами инженеры обеспечили только выполнения целочисленных операций сложения и вычитания, а операции умножения и деления смогли «припаять» только к одному из этих регистров 🙁. Через некоторое время в этой архитектуре появились дополнительные восемь регистров для работы с вещественными числами. И только в последних моделях ЭВМ фирмы Intel появились универсальные векторные регистры, которые могли выполнять все операции как с целыми, так и с вещественными числами (см. Главу 17).

Заметим, что, кроме *адресуемых* регистров, номера которых явно указываются в команде, в АЛУ по-прежнему есть и не адресуемые регистры, например, уже знакомые нам регистры первого и второго операнда `R1` и `R2`. Таким образом, команда формата регистр-память `КОП r1,A2` должна выполняться процессором, например, по схеме

`R2 := <A2>; r1 := r1` ⊗ `R2`

Отметим, что содержимое регистров, в отличие от содержимого ячеек основной памяти, принято записывать без угловых скобок, так что, например, запись `r1` в приведённом выше примере обозначает содержимое регистра `r1`, а не номер (адрес) этого регистра.

Из сказанного выше можно сделать вывод, что при программировании на таких ЭВМ желательно как можно чаще использовать регистровую память и как можно реже обращаться к большой основной памяти, этого правила мы и будем придерживаться.

В большинстве ЭВМ так называемой RISC архитектуры (мы её уже упоминали) все действия над данными производятся *только* командами формата регистр-регистр (RR), оставляя для формата регистр-память *только* операции обмена между регистрами и основной памятью.

Теперь составим для нашей машины УМ-Р фрагмент программы, который реализует, как и в предыдущих примерах, оператор присваивания `x:=a/(a+b)`$^2$. Мнемонические коды операций задают арифметические операции с обычным смыслом, а `r1` и `r2` обозначают номера адресуемых регистров.

| | |
|---|---|
| СЧ r1 a | r1:=a |
| СЧ r2 b | r2:=b |
| СЛ r2 r1 | r2:=r2+r1=b+a |
| УМН r2 r2 | r2:=(a+b)$^2$ |
| ДЕЛ r1 r2 | r1:=a/(a+b)$^2$ |
| ЗП x r1 | x:=r1=a/(a+b)$^2$ |

Длина этого фрагмента программы равна

(3 команды)*4 байта + (3 команды)*2 байта = 18 байт.

Как видим, данная архитектура, обладая отмеченными выше преимуществами, не уступает стековой (безадресной) архитектуре по компактности получаемых программ (там длина была 17 байт).

Рассмотрим теперь главный недостаток ЭВМ с адресуемыми регистрами. Заметим, что если ранее для каждой арифметической операции было необходимо реализовать по две команды (для целых и вещественных чисел), то теперь число этих команд возросло вдвое из-за необходимости реализовывать эти команды как в формате RR, так и в формате RX. Это приводит к усложнению процессора, который отныне должен поддерживать значительно бóльшее количество операций. Однако преимущества архитектуры с адресуемыми регистрами настолько очевидны, что её имеют большинство современных ЭВМ.

В этой архитектуре встречаются команды разного формата (и, соответственно, разной длины). Как говорится, современные ЭВМ обладают *многообразием форматов команд*. Например, на широко распространённых персональных компьютерах фирмы Intel реализовано около десяти форматов команд, а длина команд составляет от 1 до 15 байт [i] [см. сноску в конце главы].

Разумеется, с развитием вычислительной техники в компьютерах появилось и много новых особенностей, некоторые из которых будут рассмотрены далее в этой книге.

## 4.4. Способы адресации

*Если тебя посылают, то уточни, на всякий случай, адрес.*

Перейдём теперь к изучению другого важного понятия в архитектуре ЭВМ. Введём следующее определение. Способ адресации – это способ задания операндов внутри машинной команды. Другими словами это правила, по которым заданные в команде (двоичные) числа определяют местонахождение и значения *операндов* для данной команды. Как правило, способ адресации операндов определяется только кодом операции команды.

Как Вам уже известно, часть операндов может вообще задаваться неявно т.е. кодом операции, а в самой команде нет полей, которые определяют месторасположение таких операндов. Теперь остаётся только разобраться со способами адресации *явных* операндов.

Для лучшего усвоения этого очень важного понятия модифицируем описанную нами ранее одноадресную учебную ЭВМ УМ-1, введя в её язык команды сложения целых чисел с разными способами адресации. Мнемоника кодов операций сложения будет указывать на способ адресации.

- **Прямой способ адресации**

| СЛ | 2 | S:=S+<2> |
|---|---|---|

При этом способе адресации число на месте операнда задаёт *адрес* ячейки основной памяти, в котором и содержится необходимый в команде операнд. Будем, как обычно в угловых скобках обозначать *содержимое* ячейки основной памяти с данным адресом. Так, в приведённом выше примере <2> обозначает содержимое ячейки с адресом 2. Важно понять, что в этой ячейке, конечно же, скорее всего, хранится *не* число 2.

- **Непосредственный способ адресации**

| СЛН | 2 | S:=S+2 |
|---|---|---|

При таком способе адресации поле адреса команды содержит, как говорят, *непосредственный* (immediate) операнд. Таким образом, число 2 в нашем примере обозначает не ячейку памяти с адресом 2, а непосредственно (целочисленное) значение 2. Разумеется, такие непосредственные операнды могут быть только (неотрицательными) целыми числами, по величине не превышающими максимального значения, которое можно записать в поле адреса.

Использование непосредственного метода адресации позволяет не располагать (целочисленные) константы (в отдельных) ячейках памяти, а помещать их внутрь команд, (на место адреса операнда), что может сильно сэкономить память. Заметим, что это же позволяет лучше защитить констант от случайной порчи при ошибочной записи в те ячейки памяти, где они расположены. Это, разумеется, повышает надёжность программирования на таких ЭВМ. Само программирование на языке машины также упрощается, так как теперь не надо производить распределение памяти под хранение таких констант.

Заметим, что прямая адресация использовалась во всех рассмотренных ранее учебных машинах, исключение составляли только команды ввода/вывода УМ-3, где второй адрес задавал не *номер* ячейки памяти, а *число* чисел в массиве для ввода или вывода. Таки образом, это был *непосредственный*, а не прямой операнд, например `read(x)`:

ВВЦ 100 001 000; 100–прямой адрес, 001–непосредственный

- **Косвенный способ адресации**

| СЛК | 2 | S:=S+<<2>> |
|---|---|---|

Здесь число на месте операнда задаёт *адрес* ячейки памяти, содержимое которой, в свою очередь, трактуется как целое число – адрес необходимого операнда в памяти ЭВМ. Таким образом, число 2 в нашем примере является *косвенным* (indirect) адресом операнда.[1] При таком способе адресации для доступа к операнду процессору необходимо дважды обратиться к основной памяти:

R1 := <A2>; R1 := <R1>; S := S + R1

В качестве примера выполним несколько команд сложения с различными способами адресации для этой учебной одноадресной ЭВМ, и рассмотрим значение регистра сумматора S после выполнения этих команд (см. рис. 4.1). В комментариях к каждой команде показаны производимые этой командой действия. Справа на этом рисунке показаны первые ячейки памяти и хранимые в них целые числа. В этом примере, как и в самой первой учебной машине УМ-3, предполагается, что длина команды совпадает с длиной числа и длиной машинного слова.

| | | Адрес | Число в ячейке |
|---|---|---|---|
| СЧ 2 | S:=<2>=3 | 000 | 1 |
| СЛ 2 | S:=S+<2>=3+3=6 | 001 | 2 |
| СЛН 2 | S:=S+2=6+2=8 | 002 | 3 |
| СЛК 2 | S:=S+<<2>>=S+<3>=<br>S+4=8+4=12 | 003 | 4 |

Рис. 4.1. Значение сумматора S после выполнения команд сложения с различными способами адресации.

## 4.5. Многообразие форматов данных

*Во многой мудрости много печали; и кто умножает познания, умножает скорбь.*
*Соломон. «Экклезиаст»*

Современные ЭВМ позволяют совершать операции над целыми и вещественными числами разной длины. Это вызвано чисто практическими соображениями. Например, если нужное нам целое число помещается в один байт, то неэкономно использовать под его хранение два или четыре байта (экономия будет особенно заметной, если необходимо хранить большой *массив* таких чисел). Во избежание такого неоправданного расхода памяти введены соответствующие форматы данных, отражающие представление в памяти ЭВМ чисел разной длины. Например, в зависимости от размера целого числа, оно может занимать в памяти 1, 2, 4, 8 и более байт. Приведённая ниже таблица иллюстрирует *многообразие форматов данных*, для представления целых чисел на широко распространённых ЭВМ.

| Размер (байт) | Название формата | Обозначение в Ассемблере |
|---|---|---|
| 1 | Короткое целое | **db** |
| 2 | Длинное целое (слово) | **dw** |
| 4 | Двойное слово | **dd** |
| 8 | Четверное слово | **dq** |
| 16 | Восьмерное слово | **oword** |

Многообразие форматов данных требует усложнения архитектуры как устройства управления (резко возрастает число команд в языке машины), так и арифметико-логического устройства, в частности, регистровой памяти. Теперь регистры должны уметь хранить, а само арифметико-логическое устройство – обрабатывать данные *разной длины*.

---

[1] Аналогом в языке Паскаль является двойная косвенная адресация, т.е. <2>↑=2↑↑.

## 4.6. Форматы команд

*А «язык» процессоров x86, между прочим, очень интересен. На сегодняшний день они имеют едва ли не самую сложную систему команд, дающую системным программистам безграничные возможности для самовыражения. Прикладные программисты даже не догадываются, сколько красок мира у них украли компиляторы!*

*Крис Касперски ака мыщъх*

Для операций с разными способами адресации и разными форматами данных необходимо введение различных *форматов команд*, которые по-разному задают местонахождение и количество операндов, и, естественно, имеют разную длину. Для широко распространённых сейчас двухадресных ЭВМ это такие форматы команд (в скобках указано их мнемоническое обозначение):

- регистр – регистр (RR);
- регистр – память, память – регистр (RX);
- регистр – непосредственный операнд в команде (RI);
- память – непосредственный операнд в команде (SI);
- память – память, т.е. оба операнда в основной памяти (SS).

Использование для мнемонического обозначения операнда в памяти сразу двух букв X (RX) и S (SS) связано с особенностями выполнения команд этих форматов (X – явный, а S – неявный, т.е. заданный по умолчанию операнд), пока надо не задумываться над этим, а просто запомнить.

Многообразие форматов команд и данных позволяет писать более компактные и эффективные программы на языке машины, однако, как уже упоминалось, усложняет процессор ЭВМ.

Возвращаясь к учебной ЭВМ УМ-3 теперь можно, используя новые изученные понятия, сказать, что это трёхадресная машина имеет ростую архитектуру с прямым способом адресации (за исключением команд ввода/вывода), одним форматом команд и одним форматом данных.

## 4.7. Сегментирование и базирование адресов

*Записал я длинный адрес на бумажном лоскутке...*

*Арсений Тарковский, 1935 г.*

Для дальнейшего уменьшения объёма программы современные ЭВМ могут использовать новый способ адресации, основанный на принципе *сегментирования адресов*. Изучение этого важного и нового понятия проведём на следующем примере. Пусть в программе на учебной машине УМ-1 необходимо реализовать арифметический оператор присваивания `Z:=X+Y`. Ниже приведена эта часть программы с соответствующими комментариями (напомним, что `S` – это регистр сумматора одноадресной ЭВМ) Операнды в команде будем разделять запятой, а точка с запятой, как это принято в языке Ассемблера, задаёт начало *комментария* к команде::

```
СЧ X; S:=X
СЛ Y; S:=X+Y
ЗП Z; Z:=X+Y
```

Пусть в этой одноадресной учебной ЭВМ имеется $2^{24}$ (примерно 16 миллионов) ячеек памяти по 32 бит в ячейке, в каждой ячейке помещается одна команда или одно число. Будем, не теряя общности, считать, что наш фрагмент программы располагается где-то примерно в средине памяти. Пусть, например, наши переменные при распределении памяти оказались помещены соответственно в следующих ячейках памяти (как и ранее, адреса для удобства даны в десятичной системе счисления):

```
X – в ячейке с адресом 10 000 000
Y – в ячейке с адресом 10 000 001
Z – в ячейке с адресом 10 000 002
```

Тогда приведённый выше фрагмент программы после замены мнемонических обозначений переменных их адресами будут выглядеть следующим образом:

```
СЧ 10 000 000; S:=X
СЛ 10 000 001; S:=X+Y
ЗП 10 000 002; Z:=X+Y
```

Из этого примера видно, что большинство адресов в нашей программе можно представить в виде выражения `Seg+Δ`, где число `Seg` назовём *сегментным адресом* программы (в нашем случае `Seg=10 000 000`), а число Δ – *смещением* адреса относительно начала этого сегмента. Здесь налицо существенная *избыточность* информации в программе. Очевидно, что в каждой команде можно указывать только короткое *смещение* Δ, а начала сегмента хранить отдельно (обычно на каком-то специальном *сегментном* регистре процессора). Исходя из этих соображений, предусмотрим в машинном языке нашей одноадресной ЭВМ команду *загрузки сегмента* (длина этой команды 4 байта):

| ЗГС | A1 |
|---|---|
| 8 бит | 24 бита |

Тогда наш фрагмент программы будет иметь такой вид:

```
ЗГС 10 000 000; Начало сегмента=10 000 000
СЧ  000; S:=X
СЛ  001; S:=X+Y
ЗП  002; Z:=X+Y
```

Как видно, в большинстве команд можно теперь вместо длинного *адреса* ячейки памяти указывать только короткое *смещение* Δ этой ячейки относительно начала сегмента. Это позволит значительно уменьшить размер программы. Заметим, однако, что теперь при выполнении *каждого* обращения за операндом в основную память, процессор должен *вычислять* значение адреса этого операнда по формуле `A=Seg+Δ`. Это вычисление производится в устройстве управления и, естественно, усложняет его, не говоря уже о том, что и выполнение всей команды может несколько замедлиться. Например, адрес переменной X вычисляется как `Адрес(X) = Seg+Δ=10^7+1=10000001`.

Осталось выбрать оптимальную длину максимального смещения **Δ** в поле адреса команды. Для этого вернёмся к рассмотрению учебной ЭВМ с адресуемыми регистрами, для которой теперь будет реализовано сегментирование адресов основной памяти (не регистровой, там это не нужно, так как адреса регистров и так маленькие). Например, пусть под запись смещения в команде выделено поле длиной в 12 бит. Будем, как и раньше, обозначать операнд в памяти A1 или A2, но помним, что теперь это только *смещение* относительно начала сегмента. Тогда все команды, которые обращаются за операндом в основную память, будут в нашей ЭВМ с адресуемыми регистрами более короткими:[1]

| КОП | r1 | A2 |
|---|---|---|
| 8 бит | 4 бита | 12 бит |

Рассмотрим схему выполнения такой команды для формата регистр-память (⊗ как обычно задаёт какой-то код бинарной операции):

```
r1 := r1 ⊗ <Seg+A2>
```

или для формата память-регистр:

```
<B+A2> := <Seg+A2> ⊗ r1
```

Итак, сегмент – это сплошной участок памяти, начало которого задаётся в некотором *сегментном* регистре.[2] Разбиения памяти на такие участки называется *сегментированием* памяти. Сегментирование позволяет уменьшить объём памяти для хранения программ, но оно имеет и один существенный недостаток: теперь каждая команда может обращаться не к любой ячейке оперативной памя-

---

[1] Таким был формат команд популярного в 60-70-х годах прошлого века семейства ЭВМ IBM-360/370 и их отечественных аналогов ЭВМ Единой серии (ЕС ЭВМ).

[2] Чаще всего начало сегмента задаётся не в самом сегментном регистре, а дескрипторе (описателе) этого сегмента, все эти дескрипторы собраны в таблицу дескрипторов, а собственно в сегментном регистре указывается только индекс нужного дескриптора сегмента в этой таблице.

ти, как это было у нас раньше, а только к тем из ячеек, до которых «дотягивается» смещение относительно начала своего сегмента. В нашем предыдущем примере при длине смещения 12 бит каждая команда может обращаться к диапазону адресов от значения сегментного регистра `Seg` до `Seg+2`$^{12}$`-1`. Для доступа к другим ячейкам памяти необходимо записать в сегментный регистр новое значение (как говорят, *перезагрузить* сегментный регистр). Этот недостаток также существенно затрудняет в таких компьютерах работу с массивами длиной больше, чем $2^{12}$ ячеек. Однако, несмотря на указанный недостаток, ЭВМ первых поколений в целях уменьшения объёма программы часто позволяли производить сегментирование памяти.

Заметим также, что отмеченный выше недостаток исправляется путём реализации нескольких сегментных регистров, а также *переменной* длины смещения (например, разрешается смещение длиной в 1, 2, 4 или 8 байт). Необходимо, однако, понимать, что это ещё более увеличивает набор команд языка машины и усложняет процессор.

Итак, ещё раз отметим очень важное обстоятельство сегментного способа адресации. В новой архитектуре для осуществления *любого* доступа к памяти ЭВМ необходимо, чтобы ячейка, к которой осуществляется доступ, находилась в сегменте, на начало которого указывает некоторый сегментный регистр. Как уже говорилось, ЭВМ могут обеспечивать одновременную работу с несколькими сегментами памяти и, соответственно, иметь несколько сегментных регистров.

В ЭВМ используется не только сегментная организации памяти, но и так называемое *базирование* (относительная адресация команд и данных). При такой адресации некоторый *базовый* регистр ставится не на начало, а в середину участка памяти, к которому необходимо осуществлять доступ, и используется *знаковое* смещение. На рис. 4.2 показан доступ к ячейкам памяти при помощи сегментирования и базирования.

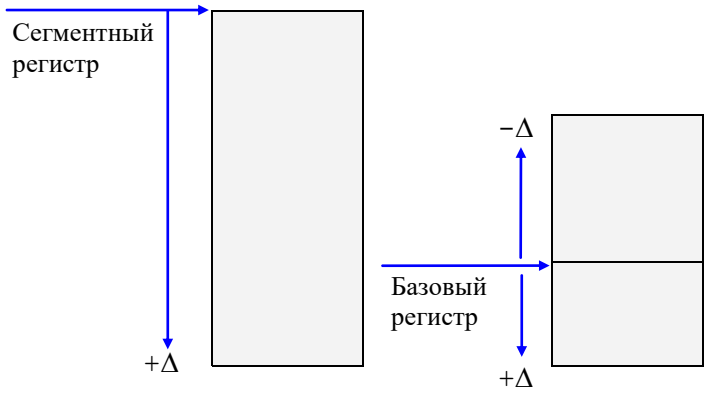



Рис. 4.2. Доступ к памяти с использованием сегментирования и базирования.

Базирование часто применяется и в нашей обычной жизни. Например, расстояние по дороге до населённых пунктов обычно указывается от главного почтамта районного или областного центра.

При изучении процедур и функций (см. разд. 6.10) Вы узнаете, что базирование широко используется для доступа к локальным переменным процедур и функций (в так называемом стековом кадре, а также для команд относительного перехода (см. разд. 6.6), когда в качестве базового регистра используется регистр счётчика адреса, который, как Вы уже знаете, указывает на следующую выполняемую команду. При величине Δ в один байт получаются очень компактные программы.

В некоторых архитектурах адресуемые регистры процессора являются *универсальными*, т.е. каждый из них может быть использован как сегментный, базовый или для выполнения любых операций над данными. Сложность процессора при этом повышается, поэтому во многих архитектурах используются *специализация* регистров, т.е. определённые регистры являются только сегментными (на них нельзя, например, складывать числа), другие не являются сегментными, но могут производить операции над данными, третьи используются в качестве счётчиков циклов и т.д.

## Вопросы и упражнения

*Невежественными бывают только те, которые решаются такими оставаться*

*Платон, V век до н.э.*

1. Что такое адресность ЭВМ ?
2. Почему в безадресной ЭВМ должны быть хотя бы две одноадресные команды ?
3. Какие ЭВМ называются компьютерами с адресуемыми регистрами ? В чём достоинства и недостатки такой архитектуры ?
4. Что такое способ адресации ? Чем отличается прямая адресация от косвенной ?
5. Добавьте в язык учебной машины УМ-3 новую команду *косвенной* пересылки:

   ```
   ПЕРК A1,A2,A3
   ```

   Эта команда использует косвенную адресацию по своему третьему адресу и выполняется по правилу:

   ```
   <A1> := <<A3>>
   ```

   Покажите, что с помощью этой команды можно обрабатывать массивы без использования самомодифицирующихся программ. Переделайте для этого программу суммирования всех элементов массива из примера на рис. 3.4.
6. Для чего нужно многообразие форматов данных ?
7. Почему в архитектуре возникает многообразие форматов команд ?
8. Почему есть формат команд память-непосредственный операнд (SI), но нет формата команд непосредственный операнд-память (IS) ?
9. В чём преимущества и недостатки сегментной организации памяти ?
10. Что такое базирование адресов ?

---

[i] Для продвинутых читателей. Переменная длина команд обеспечивает самое плотное кодирование программы, но вызывает трудности с декодированием, т.е. установлением границ команд в программе. Например, команды

```
test edi,7;                      f7 c7 07 00 00 00
setnz byte ptr[ebp-61];          0f 95 45 c3
```

при сдвиге начала декодирования на один байт даёт другие команды

```
mov dword ptr [edi],0f000000h; c7 07 00 00 00 0f
xchg ebp,eax;                  95
inc ebp;                       45
ret;                           c3
```