Глава 3. Учебная машина

Никто не заставит тебя учиться. Учиться ты будешь тогда, когда захочешь этого.

Ричард Бах.
«Карманный справочник Мессии»

Ученье без размышления бесполезно, но и размышление без ученья опасно.

Конфуций, V век до н.э.

В этой главе, используя принципы фон Неймана в качестве базовых, нами будет сконструирована собственная учебная ЭВМ. Для понимания особенностей архитектуры ЭВМ нам понадобятся даже несколько различных учебных машин, но одна из них, однако, будет изучаться более подробно, чтобы иметь возможность писать для неё простые полные программы. Будем называть эту машину УМ-3, смысл этого названия – учебная машина трёхадресная. УМ-3 будет удовлетворять *всем* принципам фон Неймана.

Для построения конкретной ЭВМ необходимо строго определить характеристики памяти этой машины, задать набор машинных команд, описать устройство процессора, определить правила взаимодействия с «внешним миром» (операции ввода/вывода) и т.д.

Память нашей УМ-3 будет состоять из 512 (это 2^9) ячеек, имеющих адреса от 0 до 511. Каждая ячейка состоит из 32 двоичных разрядов. По современным понятиям это очень маленькая память (всего 2 Kb), отметим, однако, что память одной из первых «настоящих» ЭВМ EDSAC была 225 байт.

Будем предполагать, что целые числа имеют тот же формат, что и тип `Longint`, а вещественные – тип `Single` в языке Free Pascal, внутреннее (машинное) представление этих форматов будет описано в следующей главе. Записанное в ячейке машинное слово может трактоваться (рассматриваться) как одно целое или вещественное *число* или как одна *команда*. Машинное слово, трактуемое как команда, будет разбиваться на четыре поля, и представляться в следующей форме:

КОП	A1	A2	A3
5 разрядов	9 разрядов	9 разрядов	9 разрядов

Номера ячеек, кодов операций и адресов операндов будем для удобства записывать в десятичном виде, хотя первые программисты использовали для этого шестнадцатеричную или восьмеричную системы счисления. Первое поле с именем `КОП` (код операции – operation code, opcode) содержит число от 0 до 31. Это число задает номер (код) операции машинной команды, а поля с именами `A1`, `A2` и `A3` задают адреса операндов (это целые числа от 0 до 511). Таким образом, в каждой команде могут задаваться адреса двух аргументов (это `A2` и `A3`) и адрес результата операции `A1` (это и означает, что УМ-3 трёхадресная машина).

Определим далее, какие регистры находятся в устройстве управления.

- `RA` – регистр, называемый счётчиком адреса, он имеет 9 разрядов и во время выполнения текущей команды будет содержать адрес следующей команды;
- `RK` – регистр команд имеет 32 разряда и содержит текущую выполняемую команду, которая, как уже говорилось, содержит код операции `КОП` и адреса операндов `A1`, `A2` и `A3`;
- ω – (произносится как «омега») регистр длиной два бита, он ещё называется регистром-признаком результата. В этот регистр после выполнения арифметических команд сложения, вычитания, умножения и деления, записывается число от 0 до 2 по следующему правилу (`S` – числовой результат арифметической операции, полученной на регистре-сумматоре АЛУ, см. далее):[1]

---

[1] Аналогом регистра ω в процессорах фирмы Intel для *вещественных* чисел является комбинация бит `C3` и `C0` в так называемом регистре состояния сопроцессора, они принимают значения: 00 (результат > 0), 01 (результат < 0), 10 (результат = 0), 11 (числа между собой не сравнимы 🙄). А для целых чисел это флаги `ZF`, `SF` и `CF` в специальном регистре флагов, это мы будем изучать далее.

$$w := \begin{cases} 0, & S = 0 \\ 1, & S < 0 \\ 2, & S > 0 \end{cases}$$

Все остальные команды (кроме указанных арифметических операций) не меняют ω.

- `Err` – регистр ошибки, содержит нуль (**`false`**) в случае успешного выполнения очередной команды и единицу (**`true`**) в противном случае.

В таблице 3.1 приведёны все команды учебной машины УМ-3. Множество всех допустимых форматов команд называется, как уже говорилось, *языком машины*.

Таблица 3.1. Команды учебной машины УМ-3

КОП	Смысл операции и её мнемоническое обозначение
01	СЛВ – СЛожение Вещественных чисел: <A1>:=<A2>+<A3>
11	СЛЦ – СЛожение Целых чисел: <A1>:=<A2>+<A3>
02	ВЧВ – ВыЧитание Вещественных чисел: <A1>:=<A2>-<A3>
12	ВЧЦ – ВыЧитание Целых чисел: <A1>:=<A2>-<A3>
03	УМВ – УМножение Вещественных чисел: <A1>:=<A2>*<A3>
13	УМЦ – УМножение Целых чисел: <A1>:=<A2>*<A3>
04	ДЕВ – ДЕление Вещественных чисел: <A1>:=<A2>/<A3>
14	ДЕЦ – ДЕление Целых чисел (то же, что и **`div`** в Паскале)
15	МОД – остаток от деления (то же, что и **`mod`** в Паскале)
00	ПЕР – ПЕРесылка: <A1>:=<A3>
10	ЦЕЛ – вещественное в ЦЕЛое: <A1>:=Round(<A3>)
30	ВЕЩ – целое в ВЕЩественное: <A1>:=Single(<A3>)
09	БЕЗ – БЕЗусловный переход: **`goto`** A2, т.е. RA:=A2
19	УСЛ – УСЛовный переход: **`case`** ω **`of`** 0: RA:=A1; 1: RA:=A2; 2: RA:=A3 **`end`**
20	ПР – Переход по Равно 0: **`if`** ω=0 **`then`** RA:=A2
21	ПНР – Переход по НеРавно 0: **`if`** ω<>0 **`then`** RA:=A2
22	ПБ – Переход по Больше 0: **`if`** ω=2 **`then`** RA:=A2
23	ПМ – Переход по Меньше 0: **`if`** ω=1 **`then`** RA:=A2
24	ПБР – Переход по Больше или Равно 0: **`if`** ω<>1 **`then`** RA:=A2
25	ПМР – Переход по Меньше или Равно 0: **`if`** ω<2 **`then`** RA:=A2
31	СТОП – остановка выполнения программы
06	ВВМ – ВВод A2 Машинных слов в память, начиная с адреса A1 **`for`** i:=1 **`to`** A2 **`do`** Read(<A1+i-1>)
07	ВЫВ – ВЫвод A2 Вещественных чисел, начиная с адреса A1 **`for`** i:=1 **`to`** A2 **`do`** Writeln(<A1+i-1>)
17	ВЫЦ – ВЫвод A2 Целых чисел, аналогично ВЫВ

Как вскоре станет ясно, даже этот маленький набор команд является полным: его достаточно для программирования всех задач, для решения которых хватает ресурсов этой учебной машины. И пусть Вас не смущает отсутствие многих типов данных, привычных в языках высокого уровня, так как, например, символы можно хранить в виде целых чисел – номеров символов в алфавите, логические значения – в виде целых чисел 0 и 1 и т.д. Более того, внимательное рассмотрение языка нашей машины выявит даже его *избыточность*. Так, например, вместо команды безусловного перехода 19 (БЕЗ) можно использовать команду условного перехода, в которой все три адреса совпадают. Все команды условных переходов, кроме команды с мнемоникой УСЛ, также являются избыточными.

Можно сказать, что при разработке архитектуры этой учебной машины отдавалось некоторое предпочтение удобству программирования в ущерб экономии электронных схем процессора. Хорошо, однако, вовремя остановиться и не пойти по этому пути дальше, например, не вводить в язык машины команду вычисления абсолютной величины или квадратного корня (такие команды часто реализовывались в первых ЭВМ).

Отметим, что у нас при вводе команды, целые и вещественные числа представляются на устройстве ввода во внутреннем машинном представлении, т.е. как 32-разрядные двоичные машинные слова. В то же время при выводе на печатающее устройство целые и вещественные числа преобразуются из внутреннего машинного представления и выводятся уже в десятичном формате. Отсюда следует,

что нам нужна одна команда ввода чисел, и две команды вывода, одна для целых, и одна для вещественных чисел. Для удобства программирования в языке нашей машины предусмотрены команды ввода и вывода массива машинных слов, при этом в поле команды A1 задаётся адрес начала этого массива, а в поле A2 – длина массива.

3.1. Схема выполнения команд

Большинство людей находят концепцию программирования очевидной, но само программирование невозможным.

Алан Перлис,
первый лауреат премии Тьюринга

Рассмотрим теперь схему работы процессора нашей учебной машины. В соответствии с принципом фон Неймана последовательного выполнения команд, процессор выполняет одну команду за другой, пока не выполнит команду СТОП или же пока очередная команда не зафиксирует аварийную ситуацию, установив в единицу регистр Err в устройстве управления. Модифицируем цикл работы машины фон Неймана, тело этого цикла соответствует выполнению одной команды:

repeat

1. RK := <RA>; чтение очередной команды из ячейки памяти с адресом RA на регистр команд RK в устройстве управления.
2. RA := RA+1; увеличение счётчика адреса на единицу, чтобы следующая команда (если текущая команда не является командой перехода) выбиралась из следующей ячейки памяти.
3. Выполнение операции, заданной в коде операции (КОП) в АЛУ, при необходимости знак результата записывается на регистр ω. При несуществующем коде операции или других ошибках при выполнении команды (например, делении на ноль или переполнении) выполнение команды прекращается и производится присваивание Err := 1.

until (Err=1) **or** (КОП=СТОП)

Уточним теперь для нашей вычислительной системы схему работы арифметико-логического устройства. Все *бинарные* операции (т.е. те, которые имеют два аргумента и один результат) выполняются в АЛУ нашей учебной машины по схеме: <A1>:=<A2>⊗<A3> (⊗ – любая бинарная операция). Для реализации таких операций в арифметико-логическом устройстве предусмотрены регистры первого (R1) и второго (R2) операндов, а также регистр сумматора (S), на котором размещается результат операции. Таким образом, как и в машине фон Неймана, АЛУ, подчиняясь управляющим сигналам со стороны УУ, выполняет бинарную операцию по схеме

R1 := <A2>; R2 := <A3>; S := R1⊗R2; <A1> := S;
Формирование регистра ω для арифметических операций.

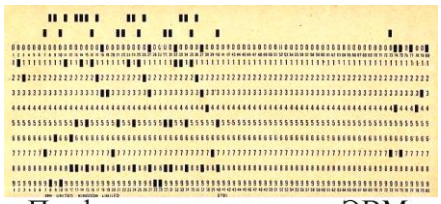
Теперь осталось определить условие начала работы программы. Прежде всего, программа каким-то образом должна оказаться в памяти нашей ЭВМ. Вообще говоря, исходя из принципа неразличимости команд и данных, нашу программу можно, например, представить в виде массива целых чисел и ввести в память, используя команду ввода (например, с кодом операции ВВМ=06, см. таблицу 3.1). Заметим, однако, что для *начального* ввода программы в память нельзя использовать *команды ввода* из языка машины, так как для этого в памяти уже должна находиться такая команда. Чтобы разорвать этот замкнутый круг, в современных ЭВМ обычно часть памяти отводят для постоянного хранения специальной небольшой программы, которая называется программой *начальной загрузки*. Эта программа начинает автоматически выполняться при включении ЭВМ. Память, занимаемая программой начальной загрузки, недоступна для записи, такую память, как уже говорилось, обычно обозначают английской аббревиатурой ROM – Read Only Memory. Ясно, что наличие программы начальной загрузки нарушает принцип фон Неймана однородности памяти.

В процессорах Intel программа начальной загрузки обычно начинается с адреса 0FFFFFFF0h (как для 32-х битных, так и для 64-х битных машин). При включении многоядерной ЭВМ сначала начинает работать только одно ядро, при этом программа «думает», что она работает на самых древ-

них процессорах Intel, где длина адреса 16 бит и всего 1Мб оперативной памяти. Целью этой небольшой программы является включение современного 32-х или 64-битного режима работы, и загрузки (обычно с диска или флешки) «настоящей» программы начальной загрузки. При этом сначала работает служебная процедура POST (Power On Self Test), она проверяет правильность функционирования основных узлов ЭВМ и выполняет ещё некоторые другие вспомогательные действия.

В нашей учебной ЭВМ пойти по этому пути нельзя по двум причинам. Во-первых, память и так очень маленькая, и отводить часть её для постоянного хранения программы начальной загрузки нерационально, а, во-вторых, в первой учебной ЭВМ мы договорились не отступать от принципа фон Неймана однородности памяти. В УМ-3 первичной загрузкой программы в память и формированием начальных значений регистров в устройстве управления занимается *устройство ввода*. Для этого на устройстве ввода имеется специальная кнопка ПУСК (). При нажатии этой кнопки устройство ввода самостоятельно (без сигналов со стороны устройства управления, которое ещё не функционирует) выполняет следующую последовательность действий:

1. Производит ввод расположенного на устройстве ввода массива машинных слов в память, начиная с *первой ячейки*; этот массив заканчивается специальным машинным словом-признаком *конца ввода*.
2. `RA:=001`; первой будет выполняться команда из первой ячейки.
3. `ω:=0`; начальное значение признака результата нулевое.
4. `Err:=0`; признак ошибки сбрасывается (делается равным **`false`**).


Перфокарта для первых ЭВМ

Для ввода программы в ЭВМ первые программисты кодировали все машинные слова в двоичном виде и переносили их на какой-нибудь *носитель данных* для устройства ввода, например, на перфоленты или картонные перфокарты (punched card, см. фото слева), на которых отверстиями кодировались хранимые данные. После этого оставалось снабдить такую, как говорили, *колоду перфокарт*, специальной перфокартой-признаком конца ввода, поместить на устройство ввода и нажать кнопку ПУСК.

Например, для нашей УМ-3 каждое машинное слово будет занимать на перфокарте 32 позиции, причём в тех позициях, где машинное слово имеет разряд единицу, пробивается отверстие. Признак конца ввода массива является специальным словом, которое имеет дополнительные (сверх 32-х) позиции (обычно признак конца ввода располагался на отдельной перфокарте, в память он не вводился). Заметим, что такой массив с признаком конца ввода аналогичен современному понятию последовательного файла машинных слов.

Как видно, при нажатии кнопки ПУСК устройство ввода выполняет основные функции упомянутой выше программы начальной загрузки ЭВМ, при этом работает только устройство ввода и память. А вот после этого всё готово для автоматической работы процессора по загруженной в память программе, и за дело принимается устройство управления.

Отметим, что для того, чтобы включить в работу устройство управления и арифметико-логическое устройство, надо начать подавать на их схемы тактовые импульсы. Таким образом, можно временно выключать те или иные устройства ЭВМ, переставая подавать на них эти импульсы. Как Вы, наверно, уже знаете, сейчас так можно заставить временно «заснуть» почти весь компьютер, оставив в работе, например, только схемы управления клавиатурой и мышкой (а иногда ещё и сетевую карту), активируя которые пользователь может заставить компьютер «проснуться».

Таким образом, полностью определены условия начала и конца работы этой учебной машины как алгоритмической системы. Здесь по аналогии хорошо вспомнить, как условия начала работы определялись, скажем, для машины Тьюринга: на ленте расположено входное слово, головка установлена на первый слева символ этого слова и машина находится в первом состоянии.

По своей архитектуре УМ-3 похожа на первые ЭВМ, построенные в соответствии с принципами фон Неймана. Например, первая отечественная ЭВМ МЭСМ (Малая Электронная Счётная Машина) 1951 года выпуска была трёхадресной ЭВМ с памятью 94 машинных слова. В языке машины было всего 13 кодов операций, она работала со скоростью около 50 оп./сек. и содержала примерно 6000 электронных ламп.


ЭВМ Стрела, МГУ, 1953 г.

Отметим также отечественную серийную ЭВМ Стрела, выпускавшуюся в средине прошлого века. Стрела содержала 6200 электронных ламп и 60000 полупроводниковых диодов, работала со скоростью 2000 операций в секунду(такт 500 мкс), ввод производился с перфокарт (по 12-ть 43-разрядных слов на перфокарте). Эта трёхадресная машина имела оперативную память в 2048 43-разрядных слов (каждое слово являлось командой или числом). Эта память была реализована на электронно-лучевых трубках со временем доступа 20 мкс. Постоянная память объёмом 256 слов хранила 15 стандартных подпрограмм. Язык машины содержал 64 команды. Внешняя память была на двух магнитных лентах по 2^{20} машинных слов. Машина занимала площадь 300 кв.м. и потребляла 150 кВт (из них половину на вентиляцию и охлаждение). В то время Стрела была самой быстродействующей и первой серийной ЭВМ в Европе.

Именно на Стреле рассчитывались траектории полёта первых спутников и первого космонавта Юрия Гагарина. Эти компьютеры стояли в Математическом институте им. В.А. Стеклова и в Вычислительном центре АН СССР. Они же работали в центрах Советского Союза по разработке ядерного оружия Арзамас-16 (Серов) и Челябинск-70 (Снежинск). Заметим, что ЭВМ Стрела с серийным №4 была установлена в Научно-исследовательском вычислительном центре (НИВЦ) МГУ на Ленинских Горах (см. фото).

3.2. Примеры программ для УМ-3

При изучении наук примеры полезнее правил.
Исаак Ньютон

Далее рассматриваются несколько простых полных программ для данной учебной машины. Эти программы призваны проиллюстрировать основные приёмы программирования на языке машины для первых ЭВМ. Такие примеры позволят Вам лучше понять способ работы учебной ЭВМ и, следовательно, её архитектуру. Итак, представим себя первыми программистами 🙂.

3.2.1. Пример 1. Оператор присваивания

Каждый должен учиться программированию, потому что оно учит думать.
Стив Джобс

Составим программу, которая вводит *целое* число `x` и реализует арифметический оператор присваивания Паскаля:

`y := (x+1)^2 mod (2*x)`

На языке Free Pascal эта программа может выглядеть, например, так:

```
program First;
  var x,y: Longint;
begin Readln(x); y:=sqr(x+1) mod (2*x); Writeln(y) end.
```

Текст этой программы на УМ-3 с комментариями приведён на рис. 3.1 (напомним, что все числа в программе – десятичные). Подробно прокомментируем процесс написания этой программы. Как Вы уже знаете, команды программы после начального ввода при нажатии кнопки ПУСК будут располагаться в памяти, начиная с первой ячейки. Теперь необходимо решить, в каких ячейках памяти будут располагаться переменные `x` и `y`. Эта работа называется распределением памяти под хранение переменных. При программировании на Паскале эту работу выполняла за нас Паскаль-машина, когда видела описания переменных: `var x,y: Longint;`

Теперь нам самим придётся распределять память под хранение переменных. Сделаем естественное предположение, что наша программа будет занимать не более 100 ячеек памяти. Поэтому, начиная со 101-й ячейки, память будет свободна. Пусть тогда для хранения значения переменной `x` мы выделим 101-ую ячейку, а переменной `y` – 102-ую ячейку памяти. Остальные переменные при необходимости будут размещаться в последующих ячейках памяти, так, для этой программы также понадобятся дополнительные (как говорят, *рабочие*) переменные `r1` и `r2`, которые будут размещаться в ячейках 103 и 104 соответственно.

Теперь надо понять, что при программировании на нашем машинном языке, в отличие от Паскаля, не будут существовать константы. Действительно, в какой бы ячейке не хранилось значение кон-

станты, по принципу фон Неймана однородности памяти ничто не помешает записать в эту ячейку новое значение. Поэтому будем называть *константами* такие переменные, которые имеют начальные значения, и которые не будут изменяться в ходе выполнения программы (по крайней мере, мы надеемся на это изо всех сил 🙂).

№	Команда				Комментарий
001	06	101	001	000	Readln(x)=Readln(<101>)
2	11	103	101	008	r1=103 := (x+1)
3	13	103	103	103	r1 := (x+1)^2
4	11	104	101	101	r2=104 := (x+x)=2*x
5	15	102	103	104	y=102 := r1 **mod** r2
6	17	102	001	000	Writeln(y)
7	31	000	000	000	Стоп
008	00	000	000	001	**const** 1

Рис 3.1. Программа первого примера.

Важно понять, что в начале работы программы конкретные значения имеют только те ячейки памяти, в которые произведён ввод данных по кнопке ПУСК. Отсюда следует, что константы, как и переменные, у которых нам будут нужны начальные значения, следует располагать в *тексте программы* («маскиру́я» их под команды) и загружать в память вместе с программой при нажатии кнопки ПУСК. Разумеется, такие константы, и переменные с начальными значениями следует располагать в таком месте программы, чтобы устройство управления не начало бы выполнять их как команды. Чтобы избежать этой неприятности, эти ячейки рекомендуется размещать в конце программы, после команды с кодом операции 31 (СТОП) (см. ячейку 008 на рис. 3.1).

В первых ЭВМ все ячейки памяти после включения машины обычно имели полностью нулевые значения (очищались), но в машине УМ-3 это не предполагается, так же обстоит дело и в стандарте языка Паскаль, где после порождения (размещения в памяти) переменные имеют неопределённые значения. В языке Free Pascal (как Вы уже должны знать) переменные *статического* класса памяти имеют нулевые начальные значения, а переменные остальных классов (*автоматического* и *динамического*) – неопределённые значения. Такие же соглашения приняты и в большинстве других языков высокого уровня.

Далее, используя принцип неразличимости команд и чисел, целочисленная константа 1 представлена в ячейке 008 в виде команды: 00 000 000 001 (скоро при изучении внутреннего машинного представления целых чисел Вы поймете, что как машинные слова они совпадают). Этот приём позволил ввести константу по кнопке ПУСК, «замаскирова́в» её под команду (так и поступали программисты первых ЭВМ). Можно назвать это особой, «командной» системой счисления.

Следует обратить внимание и на тот факт, что изначально программист не знает, сколько ячеек в памяти будут занимать команды его программы. Поэтому адреса программы, ссылающиеся на переменные с начальным значением, до завершения написания программы командой СТОП остаются *неизвестными* (незаполненными), и уже потом, разместив эти переменные в памяти (обычно сразу же вслед за командами программы), следует указать их адреса в тексте программы. В примерах те адреса программы учебной машины, которые заполняются программистом в последнюю очередь, будут выделяться подчёркиванием, у нас это адрес 008 в ячейке с номером 002.

Как видно из приведенного примера, для программиста запись программы состоит из строк, каждая строка снабжается номером (адресом) той ячейки, куда будет помещаться это машинное слово (команда, константа или переменная с начальным значением) при загрузке программы. Вслед за номером задаётся код операции и все три адреса команды, затем программист может указать комментарий (разумеется, номера машинных слов, и комментарии *не вводятся* в память ЭВМ, для них нет места в машинном слове). Кроме того, так как числа в памяти неотличимы от команд, то, как уже говорилось, они записываются в программе тоже в виде команд.[1]

[1] Обычно первые программисты записывали свои программы на специальных отпечатанных бланках, которые имели все эти поля.

В нашем случае надо поместить на устройство ввода два массива машинных слов – саму программу (8 машинных слов + признак конца ввода = 9 перфокарт) и число x (одна перфокарта). Как уже говорилось, первый массив заканчивался специальным признаком конца ввода, так что устройство ввода само знало, сколько машинных слов надо ввести в память по кнопке ПУСК. Для экономии на одну перфокарту (см. фото ранее) можно пробивать до 12 машинных слов (по одному слову на каждой из 12 строк перфокарты). Таким образом, наша программа будет занимать всего 3 перфокарты: 8 машинных слов (и 4 пустых строки) на первой перфокарте, признак конца ввода на второй и число x (1 машинное слово и 11 пустых строк) для ввода на третьей.

3.2.2. Пример 2. Условный оператор

Условные предложения используются тогда, когда мы хотим сделать предположение о том, что могло бы случится, случилось бы, и что бы мы хотели, чтобы случилось 🙄.

Составим теперь программу, реализующую условное присваивание. Пусть целочисленная переменная y принимает значение в зависимости от значения вводимой целочисленной переменной x.

$$y := \begin{cases} x+2, & \text{при } x < 2 \\ 2, & \text{при } x = 2 \\ 2*(x+2), & \text{при } x > 2 \end{cases}$$

На языке Free Pascal мы бы написали примерно такую программу:

```
program Second;
  var x,y: Longint;
begin
  Readln(x);
{ if x<2 then y:=x+2 else
  if x=2 then y:=2 else y:=2*(x+2);}
  y:=x+2; {для x<2}
  if x=2 then y:=2 else
  if x>2 then y:=y+y; {2*y}
{Pech:} Writeln(y)
end.
```

Заметим, что значение `x+2` понадобится нам как в первой, так и в третьей ветви условного оператора, поэтому более рационально вычисление выражения `x+2` расположить *перед* вычислением условного оператора, что и сделано в этом примере. Так делают и *оптимизирующие* компиляторы с языков высокого уровня, это называется *выделение общих подвыражений*.

Пусть для хранения x мы выделим 100-ю, а y 101-ю ячейку памяти. В данном и следующих примерах при записи программы на месте кода операции будем для удобства вместо числового номера указывать его мнемоническое обозначение, приведённое в таблице команд 3.1. Разумеется, потом, перед вводом программы в ЭВМ (при перенесении команд на перфокарты), необходимо будет заменить эти мнемонические обозначения соответствующими им двоичными числами.

Для определения того, является ли значение некоторого числа x больше, меньше или равным константе 2, будет выполняться команда вычитания `x-2`. В качестве побочного эффекта этой операции изменится регистр ω: `ω:=0` при `x=2`, значение `ω:=1` при `x<2` и `ω:=2` при `x>2` [i] [см. сноску в конце главы]. При этом сам результат операции вычитания `x-2` в этой программе не нужен, но по принятому в УМ-3 формату команд указание адреса ячейки для записи результата является обязательным. Для записи таких *ненужных* значений будем чаще всего использовать ячейку с номером 000. Заметим, что в нашей архитектуре программа специально вводится в память по кнопке ПУСК, начиная с *первой* ячейки, оставляя программисту для подобных целей свободной ячейку с номером ноль. В соответствии с принципом однородности памяти, эта ячейка ничем не отличается от других, то есть, доступна как для записи, так и для чтения данных.

В некоторых первых ЭВМ этот принцип нарушался: при считывании из ячейки с нулевым адресом всегда возвращался ноль, а запись в ячейку с этим адресом физически не осуществлялась, на практике такой принцип работы с этой ячейкой был удобнее. В частности, в памяти такой машине всегда была «настоящая» константа ноль.

На рис. 3.2 приведён текст этой программы.

№	Команда				Комментарий
001	ВВМ	100	001	000	`Readln(x)`
2	СЛЦ	101	100	010	`y := x+2 для x<2`
3	ВЧЦ	000	100	010	`<000> := x-2;` формирование ω
4	УСЛ	005	008	007	**`Case ω of 0: goto 005; 1: goto 008; 2: goto 007 end`**
5	ПЕР	101	000	010	`x=2 => y:=2`
6	БЕЗ	000	008	000	**`goto`** `008=Pech`
7	СЛЦ	101	101	101	`x>2 => y:=y+y`
8	ВЫЦ	101	001	000	`Pech: Writeln(y)`
9	СТОП	000	000	000	Конец работы
010	00	000	000	002	**`const`** `2`

Рис 3.2. Программа второго примера.

Для перехода на три ветви нашего алгоритма использовалась команда условного перехода `УСЛ` [1]. Предварительно предыдущая команда вычитания `ВЧЦ 000 100 010` занесла нужное значение в регистр признака результата ω. Заметим, что программист сам определяет порядок размещения в программе трёх ветвей условного оператора присваивания. В нашем примере будем сначала располагать вторую ветвь (для `x=2`), затем первую (`x<2`), а потом третью (`x>2`). Второй и третий адрес в команде условного перехода тоже подчёркнуты, так как эти значения становятся известны программисту только после написания первой (для `ω=0`) и второй (для `ω=1`) ветвей нашего алгоритма соответственно.

Обратите внимание, что целочисленная константа 2 располагается в таком месте программы (за командой безусловного перехода), где она *никогда* не будет выполняться как команда. Заметим также, что эта константа неотличима от команды `ПЕР 000 000 002` для пересылки содержимого второй ячейки памяти в нулевую ячейку, что и позволило нам записать эту константу в виде такой команды. Именно эта команда и выполнялась бы, если бы такая константа была (случайно) выбрана на регистр команд устройства управления.

Разберёмся теперь, на каком же языке написан второй пример программы для УМ-3. Ясно, что это не язык машины, так как он по определению состоит только из 32-разрядных двоичных чисел, а у нас вместо кодов операции стоят мнемонические обозначения, адреса написаны в десятичной форме записи, да ещё и комментарии присутствуют. Такой язык программирования для первых ЭВМ обычно называли *псевдокодом*, что хорошо отражало его сущность.[2] Для того, чтобы выполнить программу на псевдокоде, её надо предварительно перевести (или, как теперь принято говорить, *оттранслировать*) на язык машины. Очевидно, что для этого необходимо заменить все мнемонические обозначения соответствующими им числовыми кодами операций, затем перевести все десятичные числа в двоичную систему счисления, а также отбросить номера ячеек из первой колонки и комментарии. Для первых ЭВМ эту работу выполняли сами программисты. В дальнейшем псевдокоды постепенно развивались и превратились в языки низкого уровня – *ассемблеры* (в русскоязычной литературе их сначала называли *автокодами*, подразумевая, что соответствующая программа *автоматически* переводила (*кодировала*) с псевдокода на язык машины). Следующие программы для УМ-3 также будут для удобства писаться не на «чистом» языке машины, а на таком псевдокоде.

---

[1] В языках высокого уровня такой условный переход на три ветви существует, например, в языке Фортран: **`if`** `(<арифметическое выражение>) MetkaNeg,MetkaZero,MetkaPos`.

[2] Считается, что первый псевдокод был создан в 1949 г. Джоном Преспером Эккертом (John Presper Eckert) и Джоном Мок(ч)ли (John William Mauchly) для разрабатываемой ими коммерческой ЭВМ BINAC, которая, впрочем, так и не начала работать.

3.2.3. Пример 3. Реализация цикла

Если вы не можете объяснить что-то простым языком, то вы не понимаете этого.

Ричард Фейнман

В качестве следующего примера напишем программу для вычисления суммы элементов начального отрезка гармонического ряда:

$$y = \sum_{i=1}^{n} 1/i$$

Программа должна вводить значение целочисленной переменной `n≥0` и выводить в качестве результата работы вещественное значение `y`. Алгоритм естественно реализовать как *цикл с предусловием* (на случай `n=0`), на языке Free Pascal можно написать такую программу:

```
program Third;
  var i: Longint=1; y: Single=0.0;
      n: Longint;
begin { уже не надо i:=1; y:=0.0 }
  Readln(n);
  while i<=n do begin
        y:=y+1.0/i; i:=i+1
  end;
  writeln(y)
end.
```

Таким образом, здесь при вводе значения `n<1` тело цикла не будет выполняться ни одного раза, и программа будет выдавать нулевой результат, что не противоречит математическому смыслу поставленной задачи. Для хранения переменной `n` выделим ячейку 100. Параметр цикла `i`, переменную `y` и константы `1` и `1.0` будем вводить вместе с программой в виде «команд». На рис. 3.3 приведена возможная программа для решения этой задачи.

№	Команда				Комментарий
001	ВВМ	100	001	000	`Readln(n)`
2	ВЦЧ	**000**	011	100	`<0> := i-n;` формирование ω
3	ПБ	000	009	000	**if** `i>n` {ω=2} **then goto** `009`
4	ВЕЩ	000	000	011	`<0> := Single(i)`
5	ДЕВ	000	014	000	`<0> := 1.0/Single(i)`
6	СЛВ	012	012	000	`y := y+1.0/Single(i)`
7	СЛЦ	011	011	013	`i := i+1`
8	БЕЗ	000	002	000	Следующая итерация цикла
9	ВЫВ	012	001	000	`Writeln(y)`
010	СТОП	000	000	000	Стоп
1	00	000	000	001	**var** `i: Longint=1`
2	00	000	000	000	**var** `y: Single=0.0`
3	00	000	000	001	**const** `1`
4	`<1.0>`				**const** `1.0`

Рис 3.3. Программа третьего примера.

Заметим, что каждый член ряда является по смыслу задачи вещественным числом, в то время как параметр цикла `i` – целая величина. В языке машины нет команды деления вещественного числа на целое, поэтому при вычислении очередного слагаемого `1.0/i` нам пришлось воспользоваться командой `ВЕЩ 000 000 011`. Эта команда преобразует значение целой переменной `i` в вещественное значение тапа `Single` (заметим, что компилятор с Паскаля будет сам выполнять аналогичное действие). Обратите также внимание, что для нашей учебной машины ещё не определен формат представления вещественных чисел, (это будет сделано позже), поэтому в ячейке с адресом 14 стоит пока просто условное обозначение `<1.0>` константы `1.0`, а не её машинное представление. Здесь учащиеся

делают типичную ошибку, почему-то предполагая, что в машинном представлении числа 1 и 1.0 равны.

3.2.4. Пример 4. Работа с массивами

… время идёт, и за компьютеры садятся те, кто разбирается в них всё меньше и меньше …

Э. Таненбаум.
«Архитектура компьютера»

Пусть теперь нам требуется написать программу для ввода массива x из 100 *вещественных* чисел и вычисления суммы всех элементов этого массива:

$$S = \sum_{i=1}^{100} x[i]$$

На языке Free Pascal можно написать такую программу:

```
program Forth;
var n: Longint=100; i: Longint=1;
  Sum: Single=0.0;
  x: array[1..100] of Single;
begin
  Read(x); { Так в Паскале нельзя, там надо
  for i:=1 to 100 do Read(x[i]); }
{ for i:=1 to 100 do S:=S+x[i]; }
  repeat Sum:=Sum+x[i]; n:=n-1; i:=i+1
  until n=0;
  writeln(Sum)
end.
```

На Паскале наиболее естественно решить эту задачу с помощью цикла с параметром **for**, показанного в виде комментария. Такое решение легко реализовать на нашей УМ-3 (оно показано на рис. 3.5), но будет неудобно на нашей «настоящей» машине фирмы Intel. Дело в том, что параметр цикла i совмещает в Паскале две функции: он счётчик цикла и индекс элемента массива. На языке машины эти функции лучше разделить. Таким образом, мы будем использовать цикл **repeat**, где у нас счётчиком цикла будет переменная n (изменяясь от 100 до 0), а индексом переменная i (изменяясь от 1 до 100).

Сделаем естественное предположение, что длина нашей программы не будет превышать 199 ячеек, и поместим массив x, начиная с 200-ой ячейки памяти. Вещественную переменную S с начальным значением 0.0 и целую переменную n с начальным значением 100 поместим в конце текста программы (после команды СТОП), и будем вводить в память вместе с командами при нажатии кнопки ПУСК. Заметим, что так в программе будет не нужна команда обнуления переменной Sum. На рис. 3.4 приведён текст этой программы.

№	Команда				Комментарий
001	ВВМ	200	100	000	Read(x); массив x в ячейках 200÷299
2	СЛВ	008	008	200	Sum:=Sum+x[1]
3	СЛЦ	002	002	010	Модификация команды в ячейке 002: x[1]=200 -> x[2]=201,202,203…
4	ВЧЦ	009	009	010	n:=n-1 и **ω**=2 при n>0
5	ПБ	000	002	000	**if** n>0 {**ω**=2} **then goto** 002
6	ВЫВ	008	001	000	Writeln(Sum)
7	СТОП	000	000	000	Стоп
8	00	000	000	000	**var** Sum: Single=0.0
9	00	000	000	100	**var** n: Longint=100
010	00	000	000	001	**const** 1; Константа переадресации

Рис 3.4. Программа четвёртого примера.

Рассматриваемая программа выделяется новым приёмом программирования, она является самомодифицирующейся программой, о них говорилось при изучении машины фон Неймана. Обратим

внимание на третью строку программы. Содержащаяся в ней команда изменяет исходный код программы (меняет команду в ячейке с адресом 002) для организации цикла перебора элементов массива. При первом выполнении этой команды она адресуется к первому элементу массива, затем ко второму и т.д. Для перехода от одного элемента массива к следующему модифицируемая команда рассматривается как *целое число*, к которому прибавляется специально подобранная константа переадресации. Согласно принципу фон Неймана, числа и команды в учебной машине неотличимы друг от друга, а, значит, изменяя числовое представление команды, можно изменять и выполняемые ей действия.

У такого метода программирования есть один существенный недостаток: модификация кода программы внутри её самой повышает сложность программирования, может привести к путанице и вызвать появление ошибок. Заметим также, что такую программу нельзя повторно выполнить, просто передав управление на её первую команду,[1] так как нужно будет предварительно восстановить исходный вид всех модифицированных команд. Кроме того, самомодифицирующуюся программу трудно понимать и вносить в неё изменения. Только представьте себе, что Вам необходимо составить алгоритм для машины Тьюринга, в которой можно изменять команды в клетках таблицы (например, заменить команду движения головки по ленте влево на движение вправо, запись в клетку буквы `'a'` на запись буквы `'b'` и т.д.). Настоящий кошмар для современного программиста!

В нашей учебной машине, однако, самомодифицирующаяся программа – это *единственный* способ обработки достаточно больших массивов. В современных архитектурах ЭВМ, с которыми Вы познакомитесь несколько позже, есть и иные, более эффективные способы работы с массивами, поэтому метод программирования с модификацией команд в современных компьютерах обычно не используется.

Как уже говорилось, на УМ-3 есть ещё один способ контроля окончания цикла суммирования массива. Надо сравнивать значение команды суммирования в ячейке 002 (как целого числа) с конечным значением этой команды в ячейке 010, которое эта команда принимает после 100 суммирований (см. рис. 3.5).

№	Команда				Комментарий
001	ВВМ	200	100	000	`Read(x);` массив `x` в ячейках `200÷299`
2	СЛВ	008	008	200	`Sum:=Sum+x[1]`
3	СЛЦ	002	002	009	Модификация команды в ячейке `002:` `x[1]=200 -> x[2]=201,202,203...`
4	ВЧЦ	000	002	010	Сравнение `<002>` и `<010>`
5	ПНР	000	002	000	**`if`** `<002>` **`<>`** `<010>` **`then goto`** `002`
6	ВЫВ	008	001	000	`Writeln(Sum)`
7	СТОП	000	000	000	Стоп
8	00	000	000	000	**`var`** `S: Single=0.0`
9	00	000	000	001	**`const`** `1;` Константа переадресации
010	СЛВ	008	008	300	Конечное значение команды `<002>`

Рис. 3.5. Другая реализация четвёртого примера.

3.3. Формальное описание учебной машины

Весь математический формализм является как бы забором, следуя вдоль которого, слепой может уверенно двигаться в намеченном направлении.

Станíслав Лем. «Сумма технологии»

При описании архитектуры учебной ЭВМ УМ-3 на естественном языке многие вопросы остались нераскрытыми. Что, например, будет происходить после выполнения команды из ячейки с адресом `511`? Какое значение после нажатия кнопки ПУСК имеют ячейки, расположенные вне введённого массива машинных слов? Как представляются целые и вещественные числа? Как будет, например, выполняться такая машинная команда ввода массива: `ВВМ 500 100 000` ?

[1] Повторное выполнение находящейся в памяти программы может оказаться весьма полезным, например, при счёте нескольких вариантов задачи с разными входными данными. Так, многие диалоговые программы идут на своё повторное выполнение, задав вопрос "Повторить? (Y/N)".

Для ответа на почти все такие вопросы будет приведено *формальное описание* нашей учебной машины. При этом в качестве метаязыка будет использоваться язык Free Pascal.[1] Другими словами, будет представлена программа на языке Free Pascal, выполнение которой *моделирует* работу УМ-3, т.е. наша машина, по определению, работает «почти так же», как и эта написанная программа-модель.

Ниже приведена реализация учебной машины на языке Free Pascal:

```
program YM_3;
const
   N = 511;
type
   Address = 0..N;
   Tag = (kom, int, fl);
{ В машинном слове может хранится команда,
целое или вещественное число }
   Komanda = bitpacked record
               KOP: 0..31;
               A1, A2, A3: Address
end;
   Slovo = packed record
       case Tag of
               kom: (k: Komanda);
               int: (i: LongInt);
               fl:  (f: Single)
         end
   Memory = array[0..N] of Slovo;
var
   Mem: Memory;
   S, R1, R2: Slovo; { Регистры АЛУ }
   RK: Komanda;   { Регистр команд }
   RA: Address;   { Счётчик адреса }
   Om: 0..2;      { Регистр w }
   Err: Boolean;
begin
{ Процедура Input_Program должна вводить текст
программы с устройства ввода в память по кнопке ПУСК}
   Input_Program;
   Om := 0; Err := False; RA := 1; { Начальная установка регистров }
   with RK do
      repeat { Основной цикл выполнения команд }
   RK := Mem[RA].k;
   RA := (RA+1) mod (N+1);
   case KOP of { Анализ кода операции }
         00: { ПЕР }
           begin R1 := Mem[A3]; Mem[A1] := R1 end;
         01: { СЛВ }
           begin
             R1 := Mem[A2]; R2 := Mem[A3]; S.f := R1.f + R2.f;
             if S.f = 0.0 then OM := 0 else
             if S.f < 0.0 then OM := 1 else OM := 2;
             Mem[A1] := S; { Err := ? }
           end;
```

[1] Так как учебная ЭВМ является не языком, а исполнителем алгоритмов, то в качестве метаязыка необходимо выбрать не просто формальный метаязык (например, язык синтаксических диаграмм), а формальный алгоритмический язык.

```
          09: { БЕЗ }
             RA := A2;
          15: { МОД }
           begin
             R1 := Mem[A2]; R2 := Mem[A3];
             if R2.i = 0 then Err := true else begin
             S.i := R1.i mod R2.i; Mem[A1] := S;
             if S.i = 0 then OM := 0 else
             if S.i < 0 then OM := 1 else OM := 2;
           end
           end;
          31: { СТОП };
          { Реализация остальных кодов операций }
    else
    Err := true;
    end { case }
    until Err or (KOP = 31)
end.
```

Прокомментируем эту программу, описывающую работу учебной машины. Отметим сначала, что некоторая трудность возникает при моделировании начального ввода программы в память учебной машины при нажатии кнопки ПУСК. В нашей модели для задания такого начального ввода использован вызов процедуры с именем `Input_Program`, описание этой процедуры не приводится.[1]

Для хранения машинных слов учебной машины описан тип данных `Slovo`, который является записью с вариантами языка Free Pascal. В такой записи на одном и том же месте памяти могут располагаться команды, 32-битные целые числа типа `LongInt` или же 32-битные вещественные числа типа `Single` (более привычный для программистов тип `Real` языка Free Pascal здесь не подходит, потому что имеет длину 64 бита, а не 32 бита, как нам нужно). Таким образом, этот тип данных позволяет нам реализовать в программе на Паскале неразличимость представления команд, целых и вещественных чисел учебной машины.

Эта программа-модель ведёт себя почти так же, как учебная машина. Одно из немногих мест, где это поведение расходится, показано в тексте программы комментариями с вопросительным знаком, например, при реализации команды сложения вещественных чисел. Программа на Паскале при переполнении (когда результат сложения не помещается в переменную `S`) просто производит аварийное завершение программы, а учебная машина сначала присваивает регистру `Err` значение 1, а затем останавливает выполнение программы.

Заметим, что наше формальное описание отвечает и на вопрос о том, как в учебной машине представляются целые и вещественные числа: точно так же, как в переменных соответствующих типов программы на языке Free Pascal. Это представление будет подробно изучено в этой книге несколько позже.

В приведенной модели реализовано выполнение только некоторых типичных команд из языка учебной машины. Реализацию остальных команд Вы легко можете выполнить сами.

Обратите также внимание, как в нашей модели вычисляется адрес следующей выполняемой команды: `RA:=(RA+1) mod (N+1)`. Такое правило перехода к следующей по порядку команде программы позволяет считать, что память учебной машины как бы замкнута в кольцо: после выполнения команды из ячейки с адресом 511 (если это не команда перехода) следующая команда будет выполняться из ячейки с адресом ноль. Такая организация памяти типична для многих ЭВМ.

---

[1] В архитектуре ЭВМ принято выделять центральную часть, куда входит основная (оперативная) память и центральный процессор. Вся остальная аппаратура ЭВМ относится к так называемой периферии (периферийным устройствам). Таким образом, модель на Паскале формально описывает только центральную часть учебной машины, но не её периферию.

Вопросы и упражнения

Можно ответить на любой вопрос, если вопрос задан правильно.

Платон, V век до н.э.

1. Что такое код операции ?
2. Для чего в учебной ЭВМ необходим регистр признака результата ω ?
3. Объясните, как в нашей машине УМ-3 должна выполняться такая команда ввода массива вещественных чисел `ВВМ 100 500 000`. Напишите соответствующую ветвь в формальном описании УМ-3 для реализации команды ввода вещественных чисел.
4. Реализуйте в модели на Паскале выполнение команд ввода/вывода учебной машины.
5. Почему при программировании на языке машины не существуют константы, как, например, в языке Паскаль ?
6. Что такое переменная с начальным значением и как такую переменную разместить в памяти учебной ЭВМ ?
7. Что такое псевдокод ?
8. Объясните, почему для машины УМ-3 при решении некоторой задачи нельзя сделать такое распределение памяти: «Пусть массив `X` располагается в ячейках с адресами от 100 до 199, а константа `n=100` – в ячейке с адресом 200» ?
9. Что такое самомодифицирующаяся программа ?
10. Почему в учебной машине УМ-3 обработка больших массивов возможна только при помощи самомодифицирующейся программы ?
11. Напишите программу для УМ-3, она должна вводить два вещественных массива `X` и `Y` длины `N=200` и вычислять сумму

$$S := \sum_{i=1}^{N} X[i] * Y[N - i + 1]$$

---

[i] Для продвинутых читателей. В большинстве ЭВМ команда сравнения отличается от команды вычитания, например, в процессоре Intel вычитание (целых чисел) обозначается как **`sub`**, а сравнение как **`cmp`**. Дело в том, что при сравнении (хотя оно фактически и делается как операция вычитания) результат всегда существует, а вот при операции вычитания возможна аварийная ситуация при выходе разности за допустимый диапазон (целых или вещественных чисел). Таким образом, по-хорошему нам надо было бы добавить в язык УМ-3 ещё две команды: 026 (СРЦ) для сравнения целых чисел и 027 (СРВ) для сравнения вещественных чисел.

Решение этой проблемы для архитектуры Intel мы скоро узнаем.