В игре «Бой в памяти» компьютерные программы ведут битву за биты

А.К. ДЬЮДНИ

ДВЕ КОМПЬЮТЕРНЫЕ программы, действующие в своей родной стихии — памяти цифровой ЭВМ, рыщут от адреса к адресу, «атакуя» друг друга. Иногда они проводят разведывательные действия в поисках противника, иногда бомбардируют память «числовыми бомбами», а иногда копируют сами себя, избегая опасности, или приостанавливаются, чтобы исправить полученные повреждения. Это игра, которую я назвал «Бой в памяти». Она отличается от большинства других компьютерных игр тем, что люди, по существу, не принимают в ней участия. Противоборствующие программы, разумеется, написаны людьми, но, когда начинается «сражение», создателю программы не остается ничего другого, как просто наблюдать за тем, как продукт его многочасового труда либо живет, либо погибает на экране дисплея. Исход борьбы зависит лишь от того, какая программа первой получит повреждение в уязвимой области.

Название игры (Core War — «Бой на ферритовых сердечниках») происходит от ныне устаревшей технологии производства оперативной памяти ЭВМ. В 50-х и 60-х годах оперативные запоминающие устройства компьютеров состояли из тысяч ферритовых сердечников — маленьких колечек, нанизанных на тонкие проволочки. Каждый такой сердечник предназначался для запоминания одного бита информации, или двоичного разряда — основной информационной единицы. Теперь запоминающие устройства изготовляются на полупроводниковых кристаллах, однако оперативную память, в которой хранится программа в ходе своего выполнения, все еще часто называют ферритовой памятью (core memory или просто core).

Программы для игры «Бой в памяти» пишутся на специальном языке, который я назвал Редколом. Он относится к категории языков программирования, известной как языки ассемблера. В настоящее время большинство программ для ЭВМ пишется на языках высокого уровня, таких, как Паскаль, Фортран или Бейсик. В этих языках один оператор может привести к выполнению целой последовательности машинных команд. Кроме того, эти операторы легче читаются и программисту проще понять логику программы. Однако, чтобы программу можно было выполнить, ее сначала нужно перевести на «машинный язык», в котором каждая команда представляется довольно длинной цепочкой двоичных разрядов. Составлять программу на таком языке — занятие скучное и утомительное.

Языки ассемблера занимают промежуточное положение между языками высокого уровня и машинными языками. Как правило, в программе, написанной на языке ассемблера, каждый оператор соответствует одной машинной команде и, следовательно, какой-то конкретной цепочке двоичных разрядов. Но вместо того чтобы записывать эти двоичные числа, программист представляет их короткими словами, сокращенными мнемоническими обозначениями команд. Такие слова значительно проще запомнить, чем двоичные числа. Перевод на машинный язык осуществляет программа, называемая ассемблером.

Языком ассемблера пользуются сравнительно редко, потому что программы, написанные на этом языке, длиннее и в них труднее разобраться по сравнению с программами, написанными на языках высокого уровня. Однако существуют задачи, для которых язык ассемблера — идеальное средство. Если программа должна занимать как можно меньше памяти или должна выполняться как можно быстрее, то обычно ее пишут на языке ассемблера. Более того, некоторые возможности машины можно реализовать, пользуясь только языком ассемблера, — языки высокого уровня не позволяют этого сделать. Например, программы, написанные на языке ассемблера, могут модифицировать свои собственные команды или перемещать самих себя на другое место в памяти.

ИДЕЯ игры «Бой в памяти» впервые возникла у меня несколько лет назад, после того как мне рассказали о шутке одного программиста, работавшего в крупной исследовательской лаборатории, которую я условно назову «лабораторией X». Этот программист написал на языке ассемблера программу под названием «Сорняк», которая создавала свою точную копию каждый раз, когда выполнялась. Она также обладала способностью переходить из одного компьютера в другой, распространяясь по всей сети ЭВМ лаборатории X. У программы не было никакого другого назначения, кроме как «увековечить» самое себя. Через некоторое время появилось так много копий Сорняка, что более полезным программам и наборам данных стало не хватать места. Такое разрастание Сорняка продолжалось до тех пор, пока кто-то не придумал воспользоваться программой, основанной на том же принципе: была создана еще одна самовоспроизводящаяся программа под названием «Косарь» для обнаружения и уничтожения копий Сорняка, а когда их уже нельзя было найти, программа Косарь должна была уничтожить самое себя. Косарь выполнил свою задачу, и порядок в лаборатории X был таким образом восстановлен.

Несмотря на то что эта история казалась маловероятной, я в нее поверил, возможно потому, что мне захотелось в нее поверить. Я затратил некоторое время на то, чтобы докопаться до реальных фактов, скрывавшихся за этим анекдотом. (О том, что мне удалось узнать, я расскажу ниже. А сейчас заметим лишь, что в основе моего желания поверить этой истории лежала интригующая идея о двух программах, «сражающихся в темных и бесшумных лабиринтах памяти машины».)

В прошлом году я решил все-таки реализовать идею, лежащую в ее основе, и разработал первую версию игры «Бой в памяти». Мне помогал Дэвид Джонс, студент моего отделения в Университете Зап. Онтарио. С тех пор мы продолжали дорабатывать эту игру и довели ее до довольно интересного уровня.

У игры «Бой в памяти» четыре основных компонента: массив памяти на 8000 адресов, язык ассемблера Редкод, программа-монитор под названием «Марс» (MARS — Memory Array Redcode Simulator) и набор «боевых» программ. Две противоборствующие программы вводятся в память на случайным образом выбранные позиции, причем ни одна программа не знает, где находится программа-противник. Программа-монитор Марс управляет выполнением таких программ, реализуя простейший вариант системы разделения времени, которая распределяет ресурсы компьютера по многочисленным пользователям, работающим одновременно с машиной. Программы-противники «ходят» по очереди. Сначала выполняется одна команда первой программы, затем одна команда второй программы и т.д.

Действия, совершаемые программой в отведенные для нее промежутки времени, полностью определяются авто-

ром программы. Цель же этих действий заключается в том, чтобы вывести из строя другую программу, повредив ее команды. Однако можно придерживаться и защитной стратегии: программа может заниматься восстановлением полученных повреждений или, «попав под огонь противника», переходить на другое место в памяти. Игра заканчивается, когда программа-монитор Марс встречает в одной из программ команду, которую невозможно выполнить. Программа, содержащая эту недействительную команду (следствие полученного в «бою» повреждения), объявляется проигравшей.

О ПРОГРАММЕ можно многое узнать, анализируя ее поведение в уме или пользуясь просто карандашом и бумагой. Однако, чтобы по-настоящему испытать программу в действии, необходимы компьютер и программное обеспечение игры (система Марс). Играть можно и на персональном компьютере — мы с Джонсом подготовили инструкции для тех, кто захочет реализовать игру «Бой в памяти» на своем компьютере.

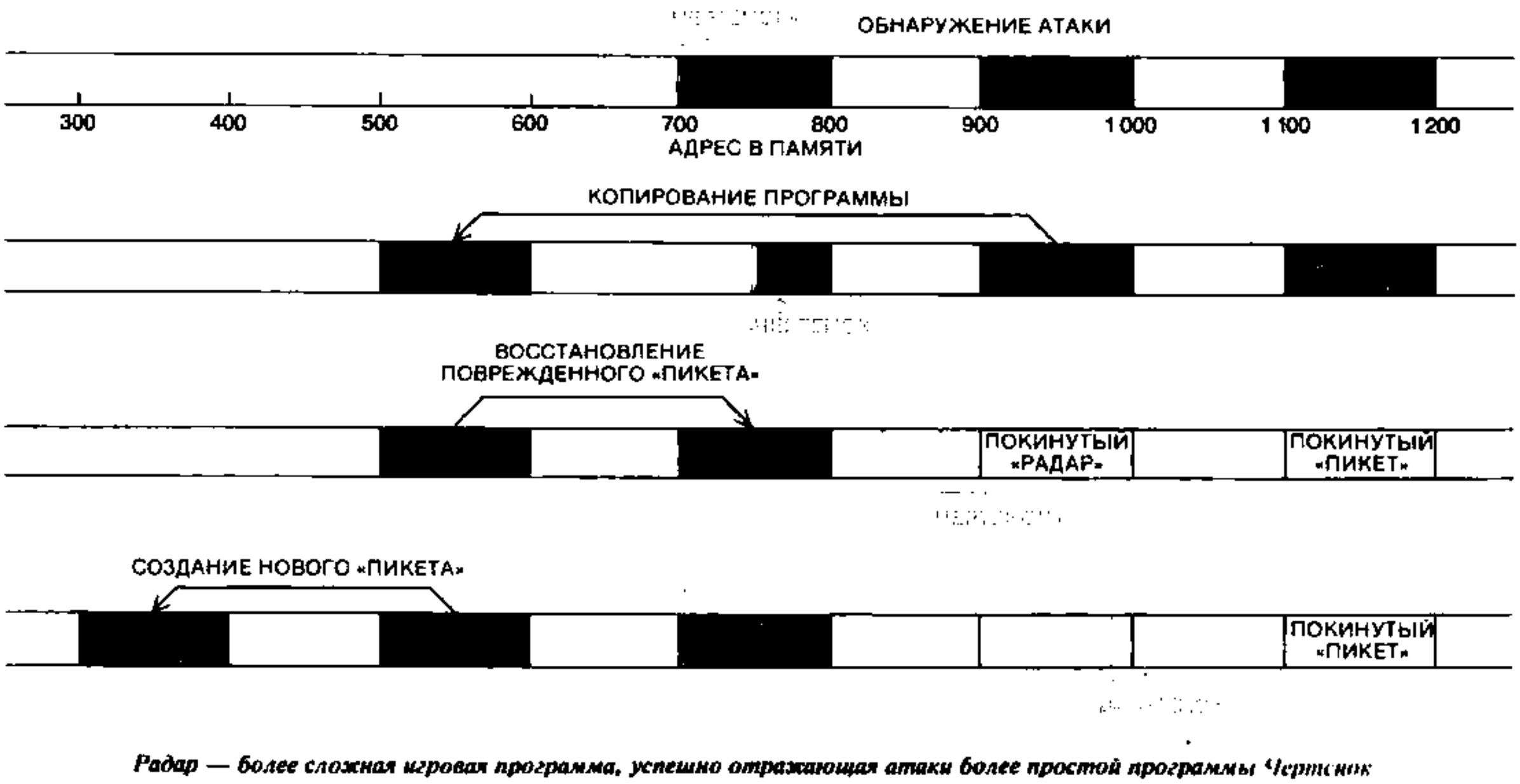
Прежде чем перейти к описанию языка Редкод и представить несколько примеров простых программ для этой игры, мне хотелось бы немного подробнее остановиться на массиве памяти. Хотя выше отмечалось, что она состоит из 8000 адресуемых ячеек, в этом числе нет ничего магического; игра может проходить и в памяти меньшего размера. Существенным отличием от большинства запоминающих устройств является то, что массив памяти для системы Марс имеет круговую структуру. Адреса ячеек памяти нумеруются последовательно от 0 до 7999, а следующий адрес 8000 указывает опять на ячейку 0; таким образом, память замыкается сама на себя. Адреса, большие, чем 7999, Марс всегда «загоняет» в пределы от 0 до 7999, для чего значение адреса делится на 8000, а полученный при делении остаток берется в качестве действительного адреса. Так, например, когда программа «стреляет» в ячейку с адресом 9378, Марс интерпретирует этот адрес как 1378.

Редкод представляет собой очень упрощенный специализированный язык программирования, относящийся к классу языков ассемблера. Он содержит команды, позволяющие перенести содержимое одной ячейки памяти в другую, команды, производящие арифметические операции над содержимым ячеек памяти, а также команды управления последовательностью выполнения команд в программе. В отличие от ассемблера, который переводит программы в двоичные коды машинных команд, Марс переводит программы, написанные на Редкоде, в последовательность больших целых десятичных чисел, хранящихся затем в массиве памяти. Каждая ячейка памяти может хранить одно такое число. При выполнении программ Марс интерпретирует эти целые числа как машинные команды и производит соответствующие операции.

Список элементарных команд языка Редкод приведен на рисунке с. 98. Каждую команду программист должен снабдить по крайней мере одним операндом или конкретным значением. Большинство команд требуют двух операндов. Например, команда JMP −7 (JMP — мнемоническое обозначение от слова jump — прыгнуть, перескочить) имеет один операнд со значением −7. Эта команда сообщает Марсу, что нужно передать управление на ячейку памяти с адресом, меньшим текущего адреса на 7 единиц, т.е. выполнить команду под номером на 7 меньшим, чем номер самой команды JMP −7. Если эта команда находилась бы сама в ячейке с адресом 3715, то выполнение программы продолжилось бы с ячейки с адресом 3708.

Такой способ вычисления адреса памяти называется относительной адресацией; это единственный способ адресации, используемый в языке Редкод. Таким образом, программа никогда не знает абсолютных значений адресов, по которым располагаются ее команды.

Команда MOV 3 100 сообщает Марсу, что нужно продвинуться вперед на 3 адреса, прочитать содержимое ячейки и перенести его в ячейку с адресом на 100 бóльшим, чем адрес самой команды, стерев то, что там находилось до сих пор. В этой команде операнды заданы в «прямой» форме, т.е. они прямо указывают на адреса ячеек, над содержимым которых производятся действия. Разрешаются также еще два способа адресации. Если перед операндом стоит значок @, то это означает, что используется «косвенная» адресация. Команда MOV @3 100 означает, что целое число, которое нужно поместить в ячейку с относительным адресом 100, находится уже не в ячейке с относительным адресом 3, а в ячейке, *адрес которой* хранится по относительному адресу 3. (Механизм косвенной адресации проиллюстрирован на с. 98 внизу.) Значок # указывает на то, что операнд следует понимать как «не-



Радар — более сложная игровая программа, успешно отражающая атаки более простой программы Чертеник

КОМАНДА	ОБОЗНАЧЕНИЕ	КОД	ОПЕРАНДЫ	ПОЯСНЕНИЯ
Переместить	MOV	1	A B	Переместить содержимое ячейки *A* в ячейку *B*
Сложить	ADD	2	A B	Сложить содержимое *ячеек A и B*
Вычесть	SUB	3	A B	Вычесть содержимое ячейки *A* из содержимого ячейки *B*
Перейти	JMP	4	A	Передать управление на ячейку *A*
Перейти, если 0	JMZ	5	A B	Передать управление на ячейку *A*, если в ячейке *B* находится 0
Перейти, если больше	JMG	6	A B	Передать управление на ячейку *A*, если в *B* находится величина, большая 0
Уменьшить и перейти, если 0	DJZ	7	A B	Вычесть 1 из содержимого ячейки *B* и передать управление по адресу *A*, если содержимое *B* становится нулевым
Сравнить	CMP	8	A B	Сравнить величины в ячейках *A* и *B*, если они не равны, пропустить следующую команду
Определить	DAT	0	B	Невыполняемая команда; *B*—значение элемента данных

Система команд Редкода — языка ассемблера для игры «Бой в памяти»

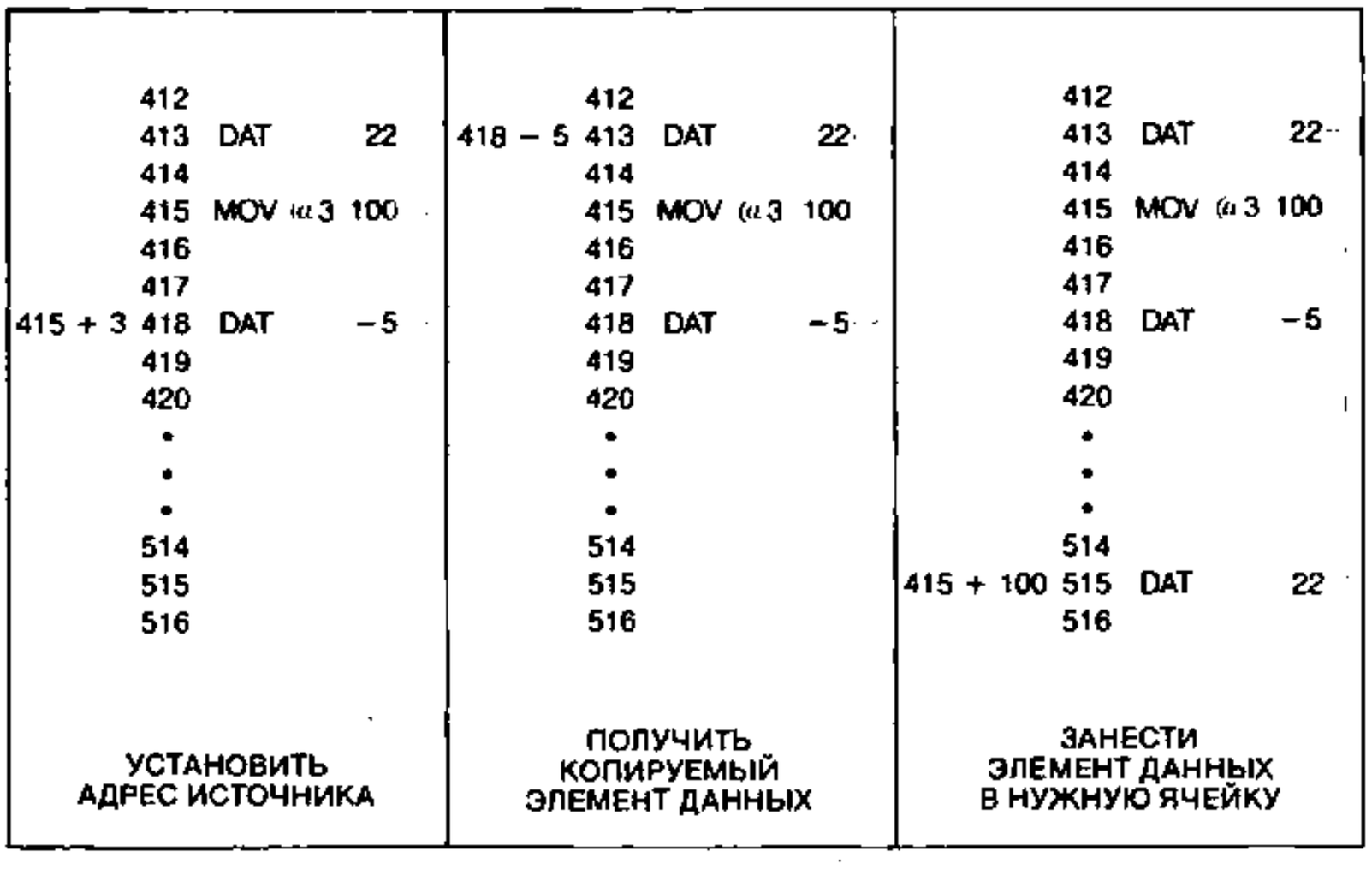
ОБОЗНАЧЕНИЕ	ОПЕРАНД *A*	ОПЕРАНД *B*	КОД ОПЕРАЦИИ	РАЗРЯД АДРЕСАЦИИ: ОПЕРАНД *A*	РАЗРЯД АДРЕСАЦИИ: ОПЕРАНД *B*	ОПЕРАНД *A*	ОПЕРАНД *B*
DAT		−1	0	0	0	0000	7999
ADD	# 5	−1	2	0	1	0005	7999
MOV	# 0	@ −2	1	0	2	0000	7998
JMP	−2		4	1	0	7998	0000

СПОСОБЫ АДРЕСАЦИИ: НЕПОСРЕДСТВЕННАЯ # 0

ПРЯМАЯ 1

КОСВЕННАЯ @ 2

Представление команд языка Редкод десятичными целыми числами



Трехшаговый механизм косвенной относительной адресации

посредственный», т.е. рассматривать его не как адрес ячейки в памяти, а как целое число. Команда MOV #3 100 означает, что целое число 3 нужно поместить в ячейку с относительным адресом 100.

Большинство других команд не нуждается в пояснениях. Следует лишь сказать еще несколько слов о команде DAT. Она может служить как средство резервирования памяти для информации, используемой в программе. Однако, строго говоря, DAT не является командой как таковой. Любая ячейка памяти, содержащая нуль в первом десятичном разряде, может рассматриваться как команда типа DAT и потому как невыполняемая команда. Если от системы Марс потребуется выполнить подобную «команду», то она не сможет этого сделать и объявит программу проигравшей.

Десятичное число, представляющее команду Редкода, имеет несколько полей, или функциональных областей (см. рисунок слева в середине). Первая цифра представляет обозначение команды; она является как бы кодом операции. Две следующие цифры отводятся под способ адресации для первого и второго операндов соответственно (прямая, косвенная или непосредственная). Наконец, на каждый операнд отводится еще по четыре цифры. Отрицательные значения величин хранятся в дополнительном представлении: так, −1 будет представлена как 7999, поскольку в круговой памяти системы «прибавить 7999» даст тот же эффект, как «вычесть 1».

Команды, из которых построена простейшая программа «Карлик», приведены на с. 99. Несмотря на то что программа Карлик не очень разумна, она весьма опасна: систематически «прочесывает» память, «бомбардируя» каждую пятую ячейку нулем. Нуль — это число, обозначающее невыполняемую команду DAT; таким образом, попав в расположение команд программы-противника, нуль может привести к ее остановке.

Предположим, к примеру, что Карлик занимает ячейки памяти с абсолютными адресами от 1 до 4. Ячейка 1 первоначально содержит команду DAT −1, однако выполнение программы начинается со следующей команды, ADD # 5 −1. В результате число 5 прибавляется к содержимому предыдущей ячейки, а именно к команде DAT −1, после чего она превращается уже в DAT 4. Затем Карлик выполняет команду, находящуюся в ячейке 3, а именно MOV # 0 @ −2. Здесь целое число 0 указано как непосредственное значение операнда; его нужно перенести в ячейку, адрес которой вычисляется следующим образом. Сначала Марс отсчитывает от адреса 3 две единицы назад, останавливаясь на ячейке

АДРЕС	ЦИКЛ 1			ЦИКЛ 2			ЦИКЛ 9		
0									
1	DAT		−1	DAT		4	DAT		14
2	ADD	#5	−1	ADD	#5	−1	ADD	#5	−1
3	MOV	#0	@−2	MOV	#0	@−2	MOV	#0	@−2
4	JMP	−2		JMP	−2		JMP	−2	
5				—	0		—	0	
6									
7									
8									
9							—	0	
10									
11									
12									
13									
14							—	0	
15									
16									
17									

Карлик — игровая программа, систематически разбрасывающая «нулевые бомбы»

1. Прочитав значение, хранящееся в этой ячейке, а именно 4, Марс интерпретирует его как адрес, определяемый относительно адреса текущей команды. Другими словами, он отсчитывает четыре ячейки вперед от адреса 1 и помещает значение 0 в ячейку с адресом 5.

Заключительная команда программы Карлик, JMP −2, создает бесконечный цикл. Она вновь передает управление на ячейку с абсолютным адресом 2, в результате чего число в команде DAT увеличивается на 5 и теперь она уже выглядит как DAT 9. При следующем выполнении тела цикла число 0 помещается уже в ячейку с абсолютным адресом 10. После этого «нулевые бомбы» падают на ячейки 15, 20, 25 и т.д. Сама программа остается на месте, но ее «артиллерия» угрожает всему массиву памяти. В конце концов Карлик добирается до адресов 7990, 7995 и затем 8000. Последний адрес рассматривается Марсом как равный 0, и поэтому Карлик едва «не покончил жизнь самоубийством». Следующая «бомба» падает снова на ячейку 5.

К сожалению, ни одна «боевая» программа, содержащая более четырех команд, не может уйти от огня Карлика. У программы-противника остается лишь три выхода: либо, постоянно передвигаясь по памяти, попытаться уйти из-под огня, либо, приняв удар, попробовать устранить повреждение, либо успеть первой поразить Карлика. Чтобы добиться успеха, следуя последней стратегии, программа должна оказаться весьма везучей. Ведь ей не известно, в каком месте памяти находится Карлик, и в среднем потребуется произвести около 1600 «выстрелов», или выполнить 1600 программных циклов, прежде чем Карлик получит повреждение. Если Карлик сражается против Карлика, то каждая программа выигрывает в 30% случаев, а в 40% случаев ни одной программе не удается поразить противника.

Прежде чем рассмотреть две другие стратегии, мне хотелось бы представить вам забавную однострочную программу. Назовем ее «Чертенком». Вот ее текст:

```
MOV 0 1
```

Чертенок представляет собой пример простейшей программы, написанной на Редкоде и способной перемещаться в памяти. Команда MOV 0 1 считывает содержимое ячейки с относительным адресом 0, т.е. считывает самое себя, и переносит его в ячейку с относительным адресом 1, т.е. в следующую ячейку. В ходе своего выполнения программа движется по памяти со скоростью одна ячейка за одну команду, оставляя за собой пространство, полностью покрытое командами MOV 0 1.

А что будет, если выставить Чертенка против Карлика? «Зона обстрела» нулевыми бомбами Карлика распространяется по памяти быстрее, чем движется Чертенок; однако из этого еще не следует, что Карлик имеет преимущество. Поразит ли Кралик Чертенка, даже если Чертенок попадет в «зону огня», — вот в чем вопрос!

Если же Чертенок первым настигнет Карлика, то он «пропашет» команды Карлика насквозь. Когда команда Карлика JMP −2 передаст управление назад на две ячейки, там уже будет находиться команда Чертенка MOV 0 1. В результате Карлик переродится и станет вторым Чертенком, до бесконечности преследующим первого по полю памяти. Согласно правилам игры «Бой в памяти», сражение заканчивается вничью. (Отметим, что такого исхода следует, по всей вероятности, ожидать. Читатели могут проанализировать другие возможности; может быть, им удастся открыть какой-либо другой любопытный исход сражения.)

КАК ЧЕРТЕНОК, так и Карлик относятся к классу программ, которые можно охарактеризовать как небольшие и агрессивные, но неразумные. На следующем по сложности уровне находятся программы, возможно менее агрессивные, но достаточно «хитрые», чтобы успешно противостоять программам низшего класса. Эти программы обладают способностью уходить от атак противника, создавая свои копии в более безопасных местах. Каждая подобная программа должна содержать фрагмент, похожий на программу Близнецы, текст которой приведен на рисунке внизу на с. 97. Близнецы не представляют собой законченной игровой программы. Единственная цель у этой программы — создать свою копию в области памяти, отстоящей на 100 ячеек, и передать управление только что созданной копии.

Программа Близнецы состоит из трех основных частей. Две команды DAT в начале программы служат в качестве указателей. Они указывают соответственно на следующую команду, подлежащую копированию, и на адрес назначения, т.е. на ячейку, куда ее нужно поместить. Цикл, расположенный в середине программы, непосредственно осуществляет копирование, перемещая команды по очереди на 100 ячеек вперед относительно того места, откуда они берутся. После каждого прохождения цикла оба указателя увеличиваются на единицу и указывают уже на новый источник и новый пункт назначения соответственно. Команда сравнения CMP в теле цикла проверяет значение первой команды DAT. Когда оно уже увеличено 9 раз, это означает, что вся программа скопирована, и происходит выход из цикла. Теперь осталось провести небольшую коррекцию. Адрес назначения во второй команде программы имеет исходное значение 99, однако к тому времени, как оно будет скопировано, это значение окажется равным 100. Ошибку, возникшую в процессе копирования, нужно исправить (посредством команды MOV # 99 93), после чего можно передать управление на только что созданную копию программы.

Модифицируя программу Близнецы, можно создать целый класс «боевых» программ. Одна из таких программ — «Идол» — создает свои копии на расстоянии 10 ячеек вместо 100. Как и Чертенок, Идол пытается «пройтись» через всю программу противника, но выигрывает гораздо чаще Чертенка и реже заканчивает игру с ничейным результатом, потому что поврежденные им программы с меньшей вероятностью смогут успешно выполнять команды, подставленные в них Идолом. В

7978	MOV	0	1	
7979	MOV	0	1	
7980	—	0		
7981	MOV	0	1	
7982	MOV	0	1	
7983	MOV	0	1	
7984	MOV	0	1	
7985	—	0		
7986	MOV	0	1	
7987	MOV	0	1	
7988	MOV	0	1	
7989	MOV	0	1	
7990	—	0		
7991	MOV	0	1	
7992	MOV	0	1	
7993	MOV	0	1	
7994	MOV	0	1	«ЧЕРТЕНОК»
7995				
7996				
7997				
7998				
7999				
0				
1	DAT		7994	
2	ADD	#5	−1	«КАРЛИК»
3	MOV	#0	@ −2	
4	JMP	−2		
5	—	0		
6				
7				
8				
9				
10	—	0		
11				

Чертенок против Карлика — кто победит?

«Скороходе» — еще одной программе, где используется механизм Близнецов, — интервал между копиями равен большому простому числу. Скорохода труднее «поймать», и он производит такое же опустошительное действие в стане противника, как и программа Идол.

СКОРОХОД и Идол не очень разумные программы. На сегодняшний день мы располагаем лишь двумя программами, относящимися ко второму уровню сложности. Они слишком громоздки для того, чтобы привести их здесь. Одна из них — назовем ее Радар — выставляет с обеих сторон своеобразные «пикеты» (см. рисунок на с. 97). Каждый пикет состоит из 100 последовательных ячеек памяти, заполненных единицами, и отделен от текста программы «буферной зоной» из 100 пустых ячеек. Радар попеременно занимается то атакующими действиями в удаленных областях памяти, то проверкой состояния своих пикетов. Если в одном из пикетов обнаруживается какое-либо изменение, Радар интерпретирует это изменение как свидетельство атаки, проведенной Карликом, Чертенком или какой-то другой неразумной программой. В этом случае Радар создает свою копию по другую сторону от поврежденного пикета, восстанавливает его, создает новый пикет на незащищенной стороне и затем продолжает нормальную последовательность действий.

Помимо создания своих копий игровая программа может быть наделена способностью восстанавливать себя от повреждений. Джонс написал самовосстанавливающуюся программу, способную выдерживать некоторые виды атак, но не все. Программа под названием «Сканер» сохраняет себя в двух экземплярах, но выполняются обычно лишь команды одного из этих экземпляров. Экземпляр программы, выполняющийся в текущий момент, периодически просматривает команды второго экземпляра, чтобы проверить, не подверглись ли они атакующим действиям программы-противника. Изменения, возникшие вследствие атаки противника, обнаруживаются путем сравнения экземпляров Сканера между собой, причем экземпляр, выполняющийся в данный момент, считается правильным. Если обнаруживаются какие-либо недействительные команды, они замещаются правильными, после чего управление передается на «отремонтированный» экземпляр, который в свою очередь приступает к проверке первого.

В настоящее время Сканер остается чисто защитной программой. Она способна выдержать атаки Карлика, Чертенка, Идола и других подобных им медленно действующих агрессивных программ, по крайней мере в тех случаях, когда противник проводит атаки с правильной (защищенной) стороны. Сейчас Джонс работает над самовосстанавливающейся программой, хранящей себя в трех экземплярах.

	DAT		0	/указатель на исходный адрес
	DAT		99	/указатель на адрес назначения
	MOV	@ −2	@ −1	/копирование
	CMP	−3	#9	/если все 10 строк уже скопированы...
	JMP	4		/... то выйти из цикла;
цикл	ADD	#1	−5	/если нет, увеличить исходный адрес...
	ADD	#1	−5	/... и адрес назначения...
	JMP	−5		/... и повторить тело цикла
	MOV	#99	93	/восстановить начальный адрес назначения
	JMP	93		/передать управление новой копии

Близнецы — программа, создающая свою копию на новом месте в памяти

Интересно, смогут ли читатели изобрести самовосстанавливающиеся программы каких-либо других типов? Например, можно подумать о программе, хранящей себя в двух или более экземплярах, но такой, что выполняется все время лишь один из них. Программа могла бы иметь «ремонтирующую часть», которая обращалась бы к другим экземплярам для восстановления поврежденных команд. Эта ремонтирующая часть могла бы даже ремонтировать сама себя, но в некоторых местах она оставалась бы уязвимой. В качестве «меры уязвимости» программы можно принять число поврежденных команд, приводящих к остановке программы. Иногда достаточно повредить одну-единственную команду; в других случаях потребуется поразить много команд. А сколько команд в среднем нужно поразить, чтобы вывести из строя программу? С точки зрения такого критерия какова должна быть наименее уязвимая самовосстанавливающаяся программа?

Игра «Бой в памяти» лишь тогда станет по-настоящему интересной, когда будут созданы достаточно надежные программы, которые в основном будут обороняться, а не атаковать. Такие игровые программы должны будут отыскивать и опознавать текст программы-противника, а затем предпринимать энергичные атакующие действия в той области памяти, где он был обнаружен.

Возможно, у вас сложилось впечатление, что язык Редкод и система Марс в целом представляют собой завершенную систему. Это не так. В свободное время мы проверяли некоторые новые идеи и много экспериментировали.

Одна из новых идей состояла в том, чтобы включить еще одну команду, которая позволяла бы несколько облегчить действия, связанные с самозащитой и самовосстановлением. Команда PCT *A* защитит команду, хранящуюся в ячейке с адресом *A*, от изменений до тех пор, пока она не будет исполнена. Насколько можно будет уменьшить уязвимость программы за счет применения команды подобного рода?

В подготовленных нами инструкциях, о которых говорилось выше, мы объясняем не только правила игры «Бой в памяти», но и то, как реализовать массив памяти и написать программу-монитор Марс на различных языках высокого уровня. Инструкции содержат также предложения о том, как можно демонстрировать результаты игры. Здесь же мы опишем правила игры с достаточной точностью для того, чтобы читатели могли составлять игровые программы, используя пока что карандаш и бумагу.

1. Две боевые программы загружают в память в случайным образом выбранные позиции, но на расстоянии

друг от друга не меньше 1000 адресов.

2. Марс поочередно выполняет по одной команде из каждой программы, пока не дойдет до команды, которую невозможно выполнить. Программа, содержащая эту команду, объявляется проигравшей.

3. Программы противника можно атаковать любым доступным «оружием». В качестве «бомбы» можно воспользоваться нулем или любым другим целым числом, включая числа, представляющие дозволенные команды Редкола.

4. На продолжительность сражения накладывается определенное временно́е ограничение, зависящее от быстродействия компьютера. Если время игры уже вышло и обе программы продолжают выполняться, то исход сражения считается ничейным.

ИСТОРИЯ о Сорняке и Косаре, по-видимому, основанана двух реально существовавших программах. Одна из них представляла собой компьютерную игру под названием «Дарвин» и была изобретена М. Макилроем из AT&T Bell Laboratories. Другая программа под названием «Червяк» была написана Дж. Шохом из Исследовательского центра фирмы Xerox в Пало-Альто. Обе программы были написаны давно и стали достаточно известными. (Игра «Дарвин» была описана в журнале "Software: Practice and Experience", т. 2, с. 93—96, 1972. Довольно туманное описание того, что, по-видимому, представляет собой ту же игру, появилось в издании "Computer lib.", 1978 г.)

По правилам игры «Дарвин» каждый играющий представляет некоторое число программ, написанных на языке ассемблера, которые засылаются в память машины вместе с программами других играющих. Эти программы называют «организмами». Организмы, созданные одним игроком (и принадлежащие к одному и тому же «виду») пытаются «убить» представителей других видов и занять обитаемое ими пространство. Победителем считается тот игрок, организмы которого оказываются наиболее многочисленными к моменту остановки игры. Макилрою удалось изобрести неуязвимый организм, хотя выиграл он всего лишь «несколько игр». Этот организм не очень-то агрессивен, хотя и является «бессмертным».

Червяк — это экспериментальная программа, предназначенная для того, чтобы как можно интенсивнее использовать ресурсы сети ЭВМ фирмы Xerox. Червяк загружался в несопротивляющиеся машины под контролем программы-супервизора. Цель Червяка состояла в том, чтобы взять на себя управление машиной и в сотрудничестве с Червяками, поселившимися в других машинах, обеспечивать выполнение больших прикладных программ в образовавшейся таким образом мультипроцессорной системе. Червяк устроен так, что любой, кто хочет воспользоваться одной из занятых машин, может получить ее в свое распоряжение, не нарушая работы более объемных программ.

В легенде о Сорняке и Косаре можно увидеть некоторые элементы как игры «Дарвин», так и программы Червяк. Но лишь в игре «Бой в памяти» Косарь стал наконец реальностью.