Kris Kaspersky

Тонкости дизассемблирования

ДИЗАССЕМБЛИРОВАНИЕ В УМЕ

Мне известны политические аргументы,

Но меня интересуют человеческие доводы.

Очень часто под рукой не оказывается ни отладчика, ни дизассемблера, ни даже компилятора, чтобы набросать хотя бы примитивный трассировщик. Разумеется, что говорить о взломе современных защитных механизмов в таких условиях просто смешно, но что делать если жизнь заставляет?

Предположим, что у нас есть простейший шестнадцатеричный редактор, вроде того, который встроен в DN и, если очень повезет, то debug.com, входящий в поставку Windows и часто остающийся не удаленным владельцами машины. Вот этим-то мы и воспользуемся. Сразу оговорюсь, что все описанное ниже требует для своего понимания значительного упорства, однако, открывает большие практические возможности. Вы сможете, например, поставить на диск парольную защиту, зашифровать несколько секторов, внести вирус или разрушающую программу и все это с помощью «подручных» средств, которые наверняка окажутся в вашем распоряжении.

Должен напомнить вам, что многие из описываемых действий в зависимости от ситуации могут серьезно конфликтовать с законом. Так, например, разрушение информации на жестком диске может повлечь за собой большие неприятности. Не пытайтесь заняться шантажом. Если вы можете зашифровать жесткий диск и установить на него пароль, то это еще не означает, что потом за сообщение пароля можно ожидать вознаграждения, а не нескольких лет тюремного заключения.

Поэтому все нижеописанное разрешается проделывать только над своим собственным компьютером или с разрешения его владельца. Если вы соглашаетесь с данными требованиями, то приступим.

СТРУКТУРА КОМАНД INTEL 80x86

Потому ты и опасен, что овладел своими страстями…

Дизассемблирование (особенно в уме) невозможно без понимания того, как процессор интерпретирует команды. Конечно, можно просто запомнить все опкоды (коды операций) команд и записать их в таблицу, которую потом выучить наизусть, но это не самый лучший путь. Уж слишком много придется зубрить. Гораздо легче понять, как стояться команды, чтобы потом с легкостью манипулировать ими.

Для начала разберемся с форматом инструкций архитектуры Intel (рис. 1).

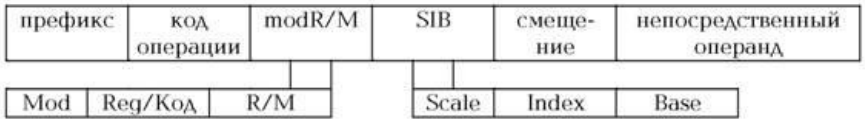


Рис. 1. Формат команд процессором x86 фирмы Intel.

Заметим, что кроме поля кода операции все остальные поля являются необязательными, т.е. в одних командах могут присутствовать, а в других нет.

Само поле кода занимает восемь бит и часто (но не всегда) имеет следующий формат (рис. 2):

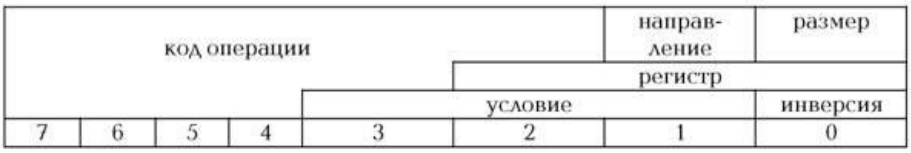


Рис. 2. Формат поля "Код операции".

Поле размера равно нулю, если операнды имеют размер один байт. Единичное значение указывает на слово (двойное слово в 32-битном или с префиксом 0x66 в 16-битном режиме).

Направление обозначает операнд-приемник. Нулевое значение присваивает результат правому операнду, а единица левому. Рассмотрим это на примере инструкции **mov bx,dx**:

```
8BDA            mov     bx,dx
10001011b
89DA            mov     dx,bx
10001001b
```

Если теперь флаг направления установить в единицу, то произойдет следующие:

```
8306230166      add     w,[00123],0066
00000011
```

Не правда ли, как по мановению волшебной палочки мы можем поменять местами операнды, изменив всего один бит? Однако, давайте задумаемся, как это поле будет вести себя, когда один из операндов непосредственное значение? Разумеется, что оно не может быть приемником и независимо от содержимого этого бита будет только источником. Инженеры Intel учили такую ситуацию и нашли оригинальное применение, часто экономящее в лучшем случае целых три байта. Рассмотрим ситуацию, когда операнду размером слово или двойное слово присваивается непосредственное значение по модулю меньшее 0100h. Ясно, что значащим является только младший байт, а стоящие слева нули по правилам математики можно отбросить. Но попробуйте объяснить это процессору! Потребуется пожертвовать хотя бы одним битом, что бы указать ему на такую ситуацию. Вот для этого и используется бит направления. Рассмотрим следующую команду:

```
810623016600    add     w,[00123],0066
00000001
```

Таким образом, мы экономим один байт в 16-разрядном режиме и целых три – в 32-разрядом. Этот факт следует учитывать при написании самомодифицирующегося кода. Большинство ассемблеров

генерируют второй (оптимизированный) вариант, и длина команды оказывается меньше ожидаемой. На этом, кстати, основан очень любопытный прием против отладчиков. Посмотрите на следующий пример:

```
00000100: 810600010200  add     w,[00100],00002
00000106: B406          mov     ah,006
00000108: B207          mov     dl,007
0000010A: CD21          int     021
0000010C: C3            retn
```

После выполнения инструкция в строке 0100h приобретет следующий вид:

```
00000100: 8206000102    add     b,[00100],002
00000105: 00B406B2      add     [si][0B206],dh
                  ↖ip после исполнения команды процессором
```

То есть, текущая команда станет на байт короче! И «отрезанный» ноль теперь стал частью другой команды! Но при выполнении на «живом» процессоре такое не произойдет, т.к. следующие значение ip вычисляется еще **до выполнения** команды на стадии ее декодирования.

Совсем другое дело отладчики, и особенно отладчики-эмуляторы, которые часто вычисляют значение ip **после выполнения** команды (это легче запрограммировать). В результате чего наступает крах. Маленькая тонкость – **до** или **после** оказалась роковой, и вот вам в подтверждение дамп экрана:

```
cs:0100 8306000102      add     word ptr [0100],02 | ax 0000  c=0
cs:0105 00B406B2        add     [si-4DFA],dh       | bx 0000  z=0
cs:0109 07              pop     es                 | cx 0000  s=0
cs:010A CD21            int     21                 | dx 0000  o=0
cs:010C C3              ret                        | si 0000  p=0
```

Заметим, что этот прием может быть бессилен против трассирующих отладчиков (debug.com, DeGlucker, Cup386), поскольку значение ip за них вычисляет процессор и делает это правильно.

Однако, на «обычные» отладчики управа всегда найдется (см. соответствующую главу), а с эмуляционными справиться гораздо труднее, и приведенный пример один из немногих, способных возыметь действие на виртуальный процессор.

Перейдем теперь к рассмотрению префиксов. Они делятся на четыре группы:

блокировки и повторения:

0F0h	LOCK	префикс
0F2h	REPNZ	(только для строковых инструкций)
0F3h	REP	(только для строковых инструкций)

переопределения сегмента

02Eh	CS:
036h	SS:
03Eh	DS:
026h	ES:
064h	FS:
065h	GS:

переопределения размеров операндов:
066h

переопределение размеров адреса:
067h

Если используется более одного префикса из той же самой группы, то действие команды не определено и по-разному реализовано на разных типах процессоров.

Префикс переопределения размера операндов используется в 16-разрядном режиме для манипуляции с 32-битными операндами и наоборот. При этом он может стоять перед любой командой, например **0x66:CLI**будет работать! А почему бы и нет? Интересно, но отладчики этого не учитывают и отказываются работать. То же относиться и к дизассемблерам, к примеру IDA Pro:

```
seg000:0100      start   proc near
seg000:0100      db      66h
seg000:0100      cli
seg000:0102      db      67h
seg000:0102      sti
seg000:0104      retn
```

На этом же основан один очень любопытный прием противодействия отладчикам, в том числе и знаменитому отладчику-эмулятору cup386. Рассмотрим, как работает конструкция **0x66:RETN**. Казалось бы, раз команда **RETN** не имеет операндов, то префикс **0x66** можно просто игнорировать. Но, на самом деле, все не так просто. **RETN** работает с неявным операндом-регистром ip/eip. Именно его и изменяет префикс. Разумеется, в реальном и 16-разрядном режиме указатель команд всегда обрезается до 16 бит, и поэтому, на первый взгляд, возврат сработает корректно. Но стек-то окажется несбалансированным! Из него вместе одного слова взяли целых два! Так нетрудно получить и исключение 0Ch – исчерпание стека. Попробуйте отладить чем-нибудь пример crack1E.com – даже cup386 во всех режимах откажется это сделать, а Turbo-Debuger вообще зависнет! IDA Pro не сможет отследить стек, а вместе с ним все локальные переменные.

Любопытно, какой простой, но какой надежный прием. Впрочем, следует признать, что перехват **INT0Ch**под операционной системой windows бесполезен, и, не смотря на все ухищрения, приложение, породившие такое исключение, будет безжалостно закрыто. Однако, в реальном режиме это работает превосходно. Попробуйте убедиться в этом на примере crack1E.com. Забавно наблюдать реакцию различных эмулирующих отладчиков на него. Все они либо неправильно работают (выбирают одно слово из стека, а не два), либо совершают очень далекий переход по 32-битному eip (в результате чего виснут), либо, чаще всего, просто аварийно прекращают работу по исключению 0Ch (так ведет себя cup386).

Еще интереснее получится, если попытаться исполнить в 16-разрядном сегменте команду CALL. Если адрес перехода лежит в пределах сегмента, то ничего необычно ожидать не приходится. Инструкция работает нормально. Все чудеса начинаются, когда адрес выходит за эти границы. В защищенном 16-разрядном режиме при уровне привилегий CL0 с большой вероятностью регистр EIP «обрежется» до шестнадцати бит, и инструкция сработает (но, похоже, что не на всех процессорах). Если уровень не CL0, то генерируется исключение защиты 0Dh. В реальном же режиме эта инструкция может вести себя непредсказуемо. Хотя в общем случае должно генерироваться прерывание **INT0Dh**. В реальном режиме его нетрудно перехватить и совершить дальний 'far' переход в требуемый сегмент. Так поступает, например, моя собственная операционная система OS\7R, дающая в реальном режиме плоскую (flat) модель памяти. Разумеется, такой поворот событий не может пережить ни один отладчик. Ни трассировщики реального режима, ни v86, ни protect-mode debugger, и даже эмуляторы (ну, во всяком случае, из тех, что мне известны) с этим справиться не в состоянии.

Одно плохо – все эти приемы не работают под Windows и другими операционными системами. Это вызвано тем, что обработка исключения типа «Общее нарушение защиты» всецело лежит на ядре операционной системы, что не позволяет приложениям распоряжаться им по своему усмотрению. Забавно, но в режиме эмуляции MS-DOS некоторые EMS-драйверы ведут себя в этом случае совершенно непредсказуемо. Часто при этом они не генерируют ни исключения 0Ch, ни 0Dh. Это следует учитывать при разработке защит, основанных на приведенных выше приемах.

Обратим внимание так же и на последовательности типа **0x66 0x66 [xxx]**. Хотя фирма intel не гарантирует корректную работу своих процессоров в такой ситуации, но фактически все они правильно интерпретируют такую ситуацию. Иное дело некоторые отладчики и дизассемблеры, которые спотыкаются и начинают некорректно вести себя.

Есть еще один интересный момент связанный с работой декодера микропроцессора.

Декодер за один раз считывает только 16 байт и, если команда «не уместиться», то он просто не сможет считать «продолжение» и сгенерирует исключение «Общее нарушение защиты». Однако, иначе ведут себя эмуляторы, которые корректно обрабатывают «длинные» инструкции.

Впрочем, все это очень процессорно-зависимо. Никак не гарантируется сохранение и поддержание этой особенности в будущих моделях, и поэтому злоупотреблять этим не стоит, иначе ваша защита откажется работать.

Префиксы переопределения сегмента могут встречаться перед любой командой, в том числе и не обращающейся к памяти, например, **CS:NOP** вполне успешно выполнится. А вот некоторые дизассемблеры сбиться могут. К счастью, IDA Pro к ним не относится. Самое интересное, что комбинация '**DS: FS: FG: CS: MOV AX,[100]**' работает вполне нормально (хотя это и не гарантируется фирмой Intel). При этом последний префикс в цепочке перекрывает все остальные. Некоторые отладчики, наоборот, ориентируются на первый префикс в цепочке, что дает неверный результат. Этот пример хорош тем, что великолепно выполняется под Windows и другими операционными системами. К сожалению, на декодирование каждого префикса тратится один такт,

и все это может медленно работать.

Вернемся к формату кода операции. Выше была описана структура первого байта. Отметим, что она фактически не документирована, и Intel этому уделяет всего два слова. Формат разнится от одной команды к другой, однако, можно выделить и некоторые общие правила. Практически для каждой команды, если регистром-приемником фигурирует AX (AL), существует специальный однобайтовый код, который содержит в трех младших битах регистр-источник. Этот факт следует учитывать при оптимизации. Так, среди двух инструкций **XCHGAX,BX** и **XCHG BX,DX** следует всегда выбирать первую, т.к. она на байт короче. (Кстати, инструкция **XCHGAX,AX** более известна нам как **NOP**. О достоверности этого факта часто спорят в конференциях, но на странице 340 руководства №24319101 «Instruction Set Reference Manual» фирмы Intel это утверждается совершенно недвусмысленно. Любопытно, что, выходит, никто из многочисленных спорщиков не знаком даже с оригинальным руководством производителя).

Для многих команд условного перехода четыре младших бита обозначают условие операции. Точнее говоря, условие задается в битах 1-3, а установка бита 0 приводит к его инверсии (таблица 1).

Таблица 1.

КОД	МНЕМОНИКА	УСЛОВИЕ
0000	O	Переполнение
0010	B, NAE	Меньше
0100	Z	Равно
0110	BE, NA	Меньше или равно
1000	S	Знак
1010	P, PE	Четно
1100	L, NGE	Меньше (знаковое)
1110	LE, NG	Меньше или равно (знаковое)

Как видим, условий совсем немного, и проблем с их запоминанием обычно не возникает. Теперь уже не нужно мучительно вспоминать '**jz**' – это 74h или 75h. Так как младший бит первого равен нулю, то '**jz**' – это 74h, а '**jnz**', соответственно, 75h.

Далеко не все коды операций смогли поместиться в первый байт. Инженеры Intel задумались о поиске дополнительного места для размещения еще нескольких бит и обратили внимание на байт modR/M. Подробнее он описан ниже, а пока рассмотрим приведенный выше рисунок (рис. 1). Трех-битовое поле reg, содержащие регистр-источник, очевидно, не используется, если вслед за ним идет непосредственный операнд. Так почему бы его не использовать для задания кода операции? Однако, процессору требуется указать на такую ситуацию. Это делает префикс '0Fh', размещенный в первом байте кода. Да, именно префикс, хотя документация Intel этого прямо и не подтверждает. При этом на не-MMX процессорах для его декодирования требуется дополнительный такт. Intel же предпочитает называть первый байт основным, а второй уточняющим кодом операции. Заметим, что это же поле используют многие инструкции, оперирующие одним операндом (jmp, call). Все это очень сильно затрудняет написание собственного ассемблера/дизассемблера, но зато дает простор для создания самомодифицирующегося кода и, кроме того, вызывает восхищение инженерами Intel, до минимума сокративших размеры команд. Конечно, это досталось весьма непростой ценой. И далеко не все дизассемблеры работают правильно. С другой стороны именно благодаря этому и существуют защиты, успешно противостоящие им.

Избежать проблем можно, лишь четко представляя себе сам принцип кодировки команд, а не просто работая с «мертвой» таблицей кодов операций, которую многие авторы вводят в дизассемблер и на том успокаиваются, так как внешне все работает правильно.

К тонкостям кодирования команд мы еще вернемся, а пока приготовимся к разбору поля modR/M. Два трехбитовых поля могут задавать код регистра общего назначения по следующей таблице (таблица 2):

Таблица 2.

		8 бит операнд	16 бит операнд	32 бит операнд
код	000	AL	AX	EAX
	001	CL	CX	ECX
	010	DL	DX	EDX
	011	BL	BX	EBX
	100	AH	SP	ESP
	101	CH	BP	EBP
	110	DH	SI	ESI
	111	BH	DI	EDI

Опять можно восхищаться лаконичностью инженеров Intel, которые ухитрились всего в трех битах закодировать столько регистров. Это, кстати, объясняет, почему нельзя выборочно обращаться к старшим и младшим байтам регистров SP, BP, SI, DI и, аналогично, к старшему слову всех 32-битных регистров. Во всем «виновата» оптимизация и архитектура команд. Просто нет свободных полей, в которые можно было бы «вместить» дополнительные регистры. Сегодня мы вынуждены расхлебывать результаты архитектурных решений, выглядевшими такими удачными всего лишь десятилетие назад.

Обратите внимание на порядок регистров: AX, CX, DX, BX, SP, BP, SI, DI. Немного не по алфавиту, верно? И особенно странно в этом отношении выглядит регистр BX. Но, если понять причины, то никакой нужны запоминать это исключение не будет, т.к. все станет на свои места: BX – это индексный регистр, и первым стоит среди индексных.

Таким образом, мы уже можем «вручную» без дизассемблера распознавать в шестнадцатеричном дампе регистры-операнды. Очень неплохо для начала! Или писать самомодифицирующийся код. Например:

```
00000000: 800E070024    or      b,[00007],024
00000005: FA            cli
00000006: 33C0          xor     ax,ax
00000008: FB            sti
```

Он изменит 6 строку на **XOR SP,SP**. Это «завесит» многие отладчики, и, кроме того, не позволит дизассемблерам отслеживать локальные переменные адресуемые через SP. Хотя IDA Pro и позволяет скорректировать стек вручную, для этого надо сначала понять, что SP обнулился. В приведенном примере это очевидно (но в глаза, кстати, не бросается), а если это произойдет в многопоточной системе? Тогда изменение кода очень трудно будет отследить, особенно в листинге дизассемблера. Однако, нужно помнить, что самомодифицирующийся код все же уходит в историю. Сегодня он встречается все реже и реже.

2-битная кодировка	3-битная кодировка
00 ES	000 ES
01 CS	001 CS
10 SS	010 SS
11 DS	011 DS
	100 FS
	101 GS
	110 Зарезервировано
	111 Зарезервировано

Первоначально сегментные регистры кодировались всего двумя битами и этого с вполне хватало, т.к. их было всего четыре. Позже, когда количество их увеличилось, перешли на трехбитную кодировку. При этом две кодовые комбинации (110b и 111b) в настоящее время не применяются и вряд ли будут добавлены в ближайшем будущем. Но что же будет, если попытаться их использовать? Генерация **INT 06h**. А вот отладчики-эмуляторы могут вести себя странно. Одни не генерируют при этом прерывания, чем себя и выдают, а другие – ведут себя непредсказуемо, т.к. при этом требуемый регистр может находится в области памяти, занятой другой переменной (это происходит, когда ячейка памяти определяется по индексу регистра, при этом считываются три бита и суммируются с базой, но никак не проверяются пределы).

Поведение дизассемблеров так же разнообразно. Вот, например,

Hiew:
```
00000000: 8E            ???
00000001: F8            clc
00000002: C3            retn
```

qview:
```
00000000: 8EF8          mov     !s,ax
00000002: C3            ret
```

IDA Pro:
```
seg000:0100 start       db      8Eh
seg000:0101             db      0F8h
seg000:0102             db      0C3h
```

Кстати, IDA Pro вообще отказывается анализировать весь последующий код. Как это можно использовать? Да очень просто – если эмулировать еще два сегментных регистра в обработчике **INT 06h**, то очень трудно это будет как отлаживать, так и ди-зассемблировать программу. Однако, это опять-таки не работает под win32!

Управляющие/отладочные регистры кодируются нижеследующим образом:

	Управляющие регистры	Отладочные регистры
000	CR0	DR0
001	Зарезервировано	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Зарезервировано
101	Зарезервировано	Зарезервировано
110	Зарезервировано	DR6
111	Зарезервировано	DR7

Заметим, что коды операций mov, манипулирующих этими регистрами, различны, поэтому-то и возникает кажущееся совпадение имен. С управляющими регистрами связана одна любопытная мелочь. Регистр CR1, как известно большинству, в настоящее время зарезервирован и не используется. Во всяком случае, так написано в русскоязычной документации. На самом деле регистр CR1 просто не существует! И любая попытка обращения к нему вызывает генерацию исключение **INT 06h**. Например, cup386 в режиме эмуляции процессора этого не учитывает и неверно исполняет программу. А все дизассемблеры, за исключением IDA Pro, неправильно дизассемблируют этот несуществующий регистр:

IDA Pro:
```
seg000:0100 start       db      0Fh
seg000:0101             db      20h
seg000:0102             db      0C8h
seg000:0103             db      0C3h
```

Sourcer:
```
43C5:0100 start:
43C5:0100 0F 20 C8      mov     eax,cr1
43C5:0103 C3            retn
```
или:
```
43C5:0100 start:
43C5:0100 0F 20 F8      mov     eax,cr7
43C5:0103               C3      retn
```

Все эти команды на самом деле не существуют и приводят к вызову прерывания **INT06h**. Не так очевидно, правда? И еще менее очевидно обращение к регистрам DR4-DR5. При обращении к ним исключения не генерируется. Между прочим, IDA Pro 3.84 дизассемблирует не все регистры. Зато великолепно их ассемблирует (кстати, ассемблер этот был добавлен другим разработчиком).

Пользуясь случаем, акцентируем внимание на сложностях, которые подстерегают при написании собственного ассемблера (дизассемблера). Документация Intel местами все же недостаточно ясна (как в приведенном примере), и неаккуратность в обращении с ней приводит к ошибкам, которыми может воспользоваться разработчик защиты против хакеров.

Теперь перейдем к описанию режимов адресации микропроцессоров Intel. Тема очень интересная и познавательная не только для оптимизации кода, но и для борьбы с отладчиками.

Первым ключевым элементом является байт modR/M.

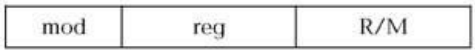


Рис. 3. Формат байта 'modR/M'.

Если 'mod' содержит '11b', то два следующий поля будут представлять собой регистры. (Это так называемая регистровая адресация). Например:

```
00000000: 33C3    xor     ax, bx
                          11 000 011
00000000: 32C3    xor     al, bl
                          11 000 011
```

Как отмечалось выше, по байту modR/M нельзя точно установить регистры. В зависимости от кода операции и префиксов размера операндов, результат может коренным образом меняться.

Биты 3-5 могут вместо определения регистра уточнять код операции (в случаи, если один из операндов представлен непосредственным значением). Младшие три бита всегда либо регистр, либо способ адресации, что зависит от значения 'mod'. А вот биты 3-5 никак не зависят от выбранного режима адресации и задают всегда либо регистр, либо непосредственный операнд.

Формат поля R/M, строго говоря, не документирован, однако достаточно очевиден, что позволяет избежать утомительного запоминания совершенно нелогичной на первый взгляд таблицы адресаций (таблица 3).

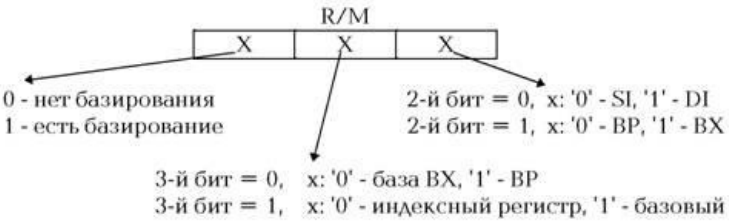


Рис. 4. Формат поля 'R/M'.

Возможно, кому-то эта схема покажется витиеватой и трудной для запоминания, но зубрить все режимы без малейшего понятия механизма их взаимодействия еще труднее, кроме того, нет никакого способа себя проверить и проконтролировать ошибки.

Действительно, в поле R/M все три бита тесно взаимосвязаны, в отличии от поля mod, которое задает длину следующего элемента в байтах.

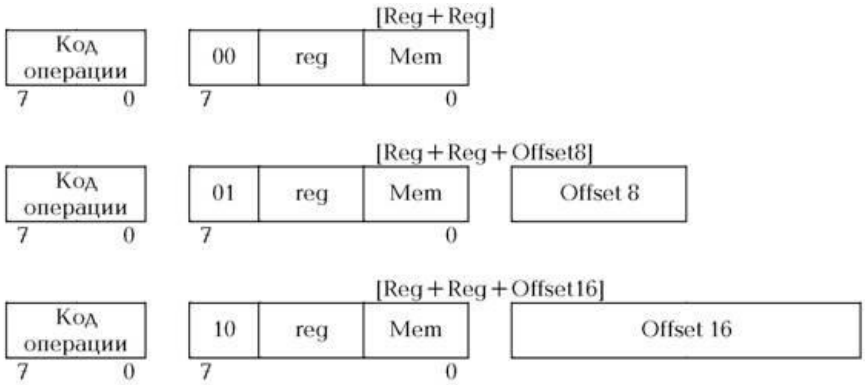


Рис. 5. Формат команды в зависимости от поля 'mod'.

Разумеется, не может быть смещения 'offset 12', (т.к. процессор не оперирует с полуторными словами), а комбинация '11' указывает на регистровую адресацию.

Может возникнуть вопрос, как складывать с 16-битным регистром 8 битное смещение? Конечно, непосредственному сложению мешает несовместимость типов, поэтому процессор сначала расширяет 8 бит до слова с учетом знака. Поэтому, диапазон возможных значений составляет от —127 до 127. (или от —0x7F до 0x7FF).

Все вышесказанное проиллюстрировано в приведенной ниже таблице 3. Обратим внимание на любопытный момент – адресация типа [BP] отсутствует. Ее ближайшим эквивалентом является [BP + 0]. Отсюда следует, что для экономии следует избегать непосредственного использования BP в качестве индексного регистра. BP может быть только базой. И **mov ax,[bp]** хотя и воспринимается любым ассемблером, но ассемблируется в **mov ax,[bp+0]**, что на байт длиннее.

Исследовав приведенную ниже таблицу 1, можно прийти к выводу, что адресация в процессоре 8086 была достаточно неудобной. Сильнее всего сказывалось то ограничение, что в качестве индекса могли выступать только три регистра (BX, SI, DI), когда гораздо чаще требовалось использовать для этого CX (например, в цикле) или AX (как возвращаемое функцией значение).

Поэтому, начиная с процессора 80386 (для 32-разрядного режима), концепция адресаций была пересмотрена. Поле R/M стало всегда выражать регистр независимо от способа его использования, чем стало управлять поле 'mod', задающие, кроме регистровой, три вида адресации:

mod	адрес
00	[Reg]
01	[Reg + 08]
10	[Reg + 32]
11	Reg

Видно, что поле 'mod' по-прежнему выражает длину следующего поля – смещения, разве что с учетом 32-битного режима, где все слова расширяются до 32 бит.

Напомним, что с помощью префикса 0x67 можно и в 16-битном режиме использовать 32-битный режимы адресации, и наоборот. Однако, при этом мы сталкиваемся с интересным моментом – разрядность индексных регистров остается 32-битной и в 16-битном режиме!

В реальном режиме, где нет понятия границ сегментов, это действительно будет работать так, как выглядит, и мы сможем адресовать первые 4 мегабайта памяти (32 бита), что позволит преодолеть печально известное ограничение размера сегмента 8086 процессоров в 64K. Но такие приложения окажутся нежизнеспособными в защищенном или V86 режиме. Попытка вылезти за границу 64K сегмента вызовет исключение 0Dh, что приведет к автоматическому закрытию приложения, скажем, под управлением Windows. Аналогично поступают и отладчики (в том числе и многие эмуляторы, включая cup386).

Сегодня актуальность этого приема, конечно, значительно снизилась, поскольку «голый DOS» практически уже не встречается, а режим его эмуляции Windows крайне неудобен для пользователей.

Таблица 3.

16-разрядный режим

адрес	Mod	R/M
[BX + SI]	00	000
[BX + DI]	00	001
[BP + SI]	00	010
[BP + DI]	00	011
[SI]	00	100
[DI]	00	101
смещ16	00	110
[BX]	00	111
[BX + SI] + смещ8	01	000
[BX + DI] + смещ8	01	001
[BP + SI] + смещ8	01	010
[BP + DI] + смещ8	01	011
[SI] + смещ8	01	100
[DI] + смещ8	01	101
[BP] + смещ8	01	110
[BX] + смещ8	01	111
[BX + SI] + смещ16	10	000
[BX + DI] + смещ16	10	001
[BP + SI] + смещ16	10	010
[BP + DI] + смещ16	10	011
[SI] + смещ16	10	100
[DI] + смещ16	10	101
[BP] + смещ16	10	110
[BX] + смещ16	10	111

32- разрядный режим

адрес	Mod	R/M
[EAX]	00	000
[ECX]	00	001
[EDX]	00	010
[EBX]	00	011
[--][--]	00	100
смещ32	00	101
[ESI]	00	110
[EDI]	00	111
смещ8[EAX]	01	000
смещ8[ECX]	01	001
смещ8[EDX]	01	010
смещ8[EBX]	01	011
смещ8[--][--]	01	100
смещ8[ebp]	01	101
смещ8[ESI]	01	110
смещ8[EDI]	01	111
смещ32[EAX]	10	000
смещ32[ECX]	10	001
смещ32[EDX]	10	010
смещ32[EBX]	10	011
смещ32[--][--]	10	100
смещ8[ebp]	10	101
смещ8[ESI]	10	110
смещ8[EDI]	10	111

Изучив эту таблицу, можно прийти к заключению, что система адресации 32-битного режима крайне скудная, и ни на что серьезное ее не хватит. Однако, это не так. В 386+ появился новый байт SIB, который расшифровывается 'Scale-Index Base'.

Процессор будет ждать его вслед за R/M всякий раз, когда последний равен 100b. Эти поля отмечены в таблице как '[-]'. SIB хорошо документирован, и назначения его полей показаны на рисунке 6. Нет совершенно никакой необходимости зазубривать таблицу адресаций.

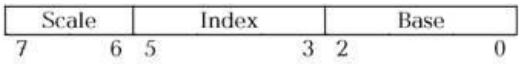


Рис. 6. Фортам поля 'SIB'.

'Base' это базовый регистр, 'Index' – индексный, а два байта 'Scale' – это степень двойки для масштабирования. Поясним введенные термины. Ну, что такое индексный регистр, понятно всем. Например SI. Теперь же в качестве индексного можно использовать любой регистр. За исключением, правда, SP; впрочем, можно выбирать и его, но об этом позже.

Базовый регистр, это тот, который суммируется с индексным, например, [BP+SI]. Аналогично, базовым теперь может быть любой регистр. При этом есть возможность в качестве базового выбрать SP. Заметим, что если мы выберем этот регистр в качестве индексного, то вместо 'SP' получим – «никакой»: в этом случае адресацией будет управлять только базовый регистр.

Таблица 4.

Base			EAX 000	ECX 001	EDX 010	EBX 011	ESP 100	EBP* 101	ESI 110	EDI 111
Index		S	шестнадцатеричные значения SIB							
[EAX]	000	00	00	08	10	18	20	28	30	38
[ECX]	001		01	09	11	19	21	29	31	39
[EDX]	010		02	0A	12	1A	22	2A	32	3A
[EBX]	011		03	0B	13	1B	23	2B	33	3B
нет	100		04	0C	14	1C	24	2C	34	3C
[EBP]	101		05	0D	15	1D	25	2D	35	3D
[ESI]	110		06	0E	16	1E	26	2E	36	3E
[EDI]	111		07	0F	17	1F	27	2F	37	3F
[EAX*2]	000	01	40	48	50	58	60	68	70	78
[ECX*2]	001		41	49	51	59	61	69	71	79
[EDX*2]	010		42	4A	52	5A	62	6A	72	7A
[EBX*2]	011		43	4B	53	5B	63	6B	73	7B
нет	100		44	4C	54	5C	64	6C	74	7C
[EBP*2]	101		45	4D	55	5D	65	6D	75	7D
[ESI*2]	110		46	4E	56	5E	66	6E	76	7E
[EDI*2]	111		47	4F	57	5F	67	6F	77	7F
[EAX*4]	000	10	80	88	90	98	A0	A8	B0	B8
[ECX*4]	001		81	89	91	99	A1	A9	B1	B9
[EDX*4]	010		82	8A	92	9A	A2	AA	B2	BA
[EBX*4]	011		83	8B	93	9B	A3	AB	B3	BB
нет	100		84	8C	94	9C	A4	AC	B4	BC
[EBP*4]	101		85	8D	95	9D	A5	AD	B5	BD
[ESI*4]	110		86	8E	96	9E	A6	AE	B6	BE
[EDI*4]	111		87	8F	97	9F	A7	AF	77	BF
[EAX*8]	000	11	C0	C8	D0	D8	E0	E8	F0	F8
[ECX*8]	001		C1	C9	D1	D9	E1	E9	F1	F9
[EDX*8]	010		C2	CA	D2	DA	E2	EA	F2	FA
[EBX*8]	011		C3	CB	D3	DB	E3	EB	F3	FB
нет	100		C4	CC	D4	DC	E4	EC	F4	FC
[EBP*8]	101		C5	CD	D5	DD	E5	ED	F5	FD
[ESI*8]	110		C6	CE	D6	DE	E6	EE	F6	FE
[EDI*8]	111		C7	CF	D7	DF	E7	EF	F7	FF

Ну и, наконец, масштабирование – это уникальная возможность умножать индексный регистр на 1,2,4,8 (т.е. степень двойки, которая задается в поле Scale). Это очень удобно для доступа к различным структурам данных. При этом индексный регистр, являющийся одновременно и счетчиком цикла, будет указывать на следующий элемент структуры даже при единичном шаге цикла (что чаще всего и встречается). В таблице 4 показаны все возможные варианты значений байта 'SIB'.

Если при этом в качестве базового регистра будет выбран EBP, то полученный режим адресации будет зависеть от поля MOD предыдущего байта. Возможны следующие варианты:

mod	действие
00	смещение32 [index] — регистр EBP в адресации не участвует!
01	смещение8 [EBP] [index]
10	смещение32 [EBP] [index]

Итак, мы полностью разобрались с кодировкой команд. Осталось лишь выучить непосредственно

саму таблицу кодов, и можно отправляться в длинный и тернистый путь написания собственного дизассемблера.

За это время, надеюсь, у вас разовьются достаточные навыки для ассемблиро-вания/ дизассемблирования в уме. Впрочем, есть множество эффективных приемов, позволяющих облегчить сей труд. Ниже я покажу некоторые из них. Попробуем без дизассемблера взломать crackme01.com. Для этого даже не обязательно помнить коды всех команд!

```
00000000: B4 09 BA 77 01 CD 21 FE C4 BA 56 01 CD 21 8A 0E | „.єw.ќ!.”eV.ќ!K.
00000010: 56 01 87 F2 AC 02 E0 E2 FB BE 3B 01 30 24 46 81 | V.З.м.рт.Ъ;.0$FB
00000020: FE 56 01 72 F7 4E 02 0C 81 FE 3B 01 73 F7 80 F9 | .V.rЇN..Б.;.s.A.
00000030: C3 74 08 B4 09 BA BE 01 CD 21 C3 B0 94 29 9A 64 | “t.„.єЋ.ќ!”ЂФ)Ъd
00000040: 21 ED 01 E3 2D 2A 70 41 53 53 57 4F 52 44 00 6F | !э.у-*pASSWORD.o
00000050: 6B 01 20 2A 04 B0 20 00 00 00 00 00 00 00 00 00 | k..*.Ђ..........
00000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000070: 00 00 00 00 00 00 00 43 72 61 63 6B 20 4D 65 20 | .......Crack Me
00000080: 20 30 78 30 20 3A 20 54 72 79 20 74 6F 20 66 6F |  0x0 : Try to fo
00000090: 75 6E 64 20 76 61 6C 69 64 20 70 61 73 73 77 6F | und valid passwo
000000A0: 72 64 20 28 63 29 20 4B 50 4E 43 0D 0A 54 79 70 | rd (c) KPNC..Typ
000000B0: 65 20 70 61 73 73 77 6F 72 64 20 3A 20 24 0D 0A | e password : $..
000000C0: 50 61 73 73 77 6F 72 64 20 66 61 69 6C 2E 2E 2E | Password fail...
000000D0: 20 74 72 79 20 61 67 61 69 6E 0D 0A 24          |  try again..$
```

Итак, для начала поищем, кто выводит текст 'Crack me... Type password:'. В самом файле начало текста расположено со смещением 77h. Следовательно, учитывая, что com файлы загружаются, начиная со смещения 100h, эффективное смещение, равняется 100h+77h=177h. Учитывая обратное расположение старших и младших байт, ищем в файле последовательность 77h 01h.

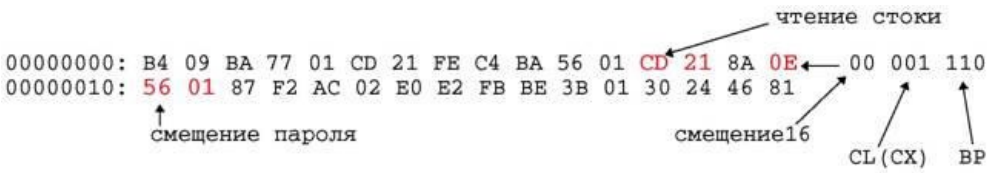
```
00000000: B4 09 BA 77 01 CD 21
```

Вот она! Но что представляет собой код 0BAh? Попробуем определить это по трем младшим битам. Они принадлежат регистру DL(DX). А 0B4h 09h – это ***AH,9**. Теперь нетрудно догадаться, что оригинальный код выглядел как:

```
MOV     AH,9
MOV     DX,0177h
```

И это при том, что не требуется помнить код команды MOV! (Хотя это очень распространенная команда и запомнить ее код все же не помешает).

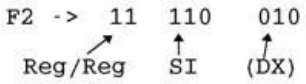
Вызов 21-го прерывания 0CDh 21h легко отыскать, если запомнить его символьное представление '=!' в правом окне дампа. Как нетрудно видеть, следующий вызов **INT21h** лежит чуть правее по адресу 0Ch. При этом DX указывает на 0156h. Это соответствует смещению 056h в файле. Наверняка эта функция читает пароль. Что ж, уже теплее. Остается выяснить, кто и как к нему обращается. Ждать придется недолго.



При разборе байта 0Eh не забудьте, что адресации [BP] не существует в природе. Вместо этого мы получим [offset16]. На размер регистра и приемник результата указывают два младших бита байта 08Ah. Они равны 10b. Следовательно, мы имеем дело с регистром CL, в который записывается содержимое ячейки [0156h].

Все, знакомые с ассемблером усмотрят в этом действии загрузку длины пароля (первый байт строки) в счетчик. Неплохо для начала? Мы уже дизассемблировали часть файла и при этом нам не потребовалось знание ни одного кода операции, за исключением, быть может, 0CDh, соответствующего команде 'INT.

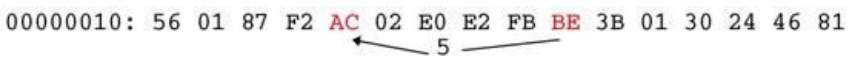
Вряд ли мы скажем, о чем говорит код 087h. (Впрочем, обращая внимание на его близость к операции **NOP**, являющейся псевдонимом '**XCHGAX,AX**', можно догадаться, что 087h – это код операции **XCHG**). Обратим внимание на связанный с ним байт 0F2h:



Как не трудно догадаться, эта команда заносит в SI смещение пароля, содержащиеся в DX. Этот вывод мы делаем, только исходя из смыслового значения регистров, полностью игнорируя код команды. К сожалению, этого нельзя сказать о следующем байте – 0ACh. Это код операции **LODSB**, и его придется просто запомнить.

0x02 – это код **ADD**, а следующий за ним байт – это **AH,AL**(не буду больше повторять, как это было получено).

0xE2 – это код операции **LOOP**, а следующий за ним байт – это знаковое относительное смещение перехода.

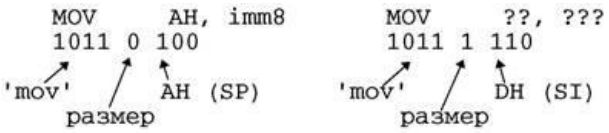


Чтобы превратить его в знаковое целое, необходимо дополнить его до нуля, (операция **NEG**, которую большинство калькуляторов не поддерживают). Тот же самый результат мы получим, если отнимем от 0100h указанное значение (в том случае, если разговор идет о байте). В нашем примере это равно пяти. Отсчитаем пять байт влево от начала **следующей команды**. Если все сделать правильно, то вычисленный переход должен указывать на байт 0ACh (команда **LODSB**), впрочем, последнее было ясно и без вычислений, ибо других вариантов, по-видимому, не существует.

Почему? Да просто данная процедура подсчета контрольной суммы (или точнее хеш-суммы) очень типична. Впрочем, не стоит всегда полагаться на свою интуицию и «угадывать» код, хотя это все же сильно ускоряет анализ.

С другой стороны, хакер без интуиции – это не хакер. Давайте применим нашу интуицию, чтобы «вычислить», что представляет собой код следующей команды. Вспомним, что 0B4h (10110100b) – это **MOV AH,imm8**.

0BEh очень близко к этому значению, следовательно, это операция MOV. Осталось определить регистр-приемник. Рассмотрим обе команды в двоичном виде:

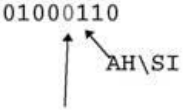


Как уже говорилось выше, младшие три бита – это код регистра. Однако, его невозможно однозначно определить без утончения размера операнда. Обратим внимание на третий (считая от нуля) бит. Он равен нулю для AH и единице в нашем случае. Рискнем предположить, что это и есть бит размера операнда, хотя этого явно и не уточняет Intel, но вытекает из самой архитектуры команд и устройства декодера микропроцессора.

Обратим внимание, что это, строго говоря, частный случай, и все могло оказаться иначе. Так, например, четвертый справа бит по аналогии должен быть флагом направления или знакового расширения, но увы – таковым в данном случае не является. Четыре левые бита это код операции '**mov reg,imm**'. Запомнить его легко – это «13» в восьмеричном представлении.

Итак, 0BEh 03Bh 001h – это **MOV SI,013Bh**. Скорее всего, 013bh – это смещение, и за этой командой последует расшифровщик очередного фрагмента кода. А может быть и нет – это действительно смелое предположение. Однако, байты 030h 024h это подтверждают. Хакеры обычно так часто сталкиваются с функций **xor**, что чисто механически запоминают значение ее кода.

Не трудно будет установить, что эта последовательность дизассемблируется как **XOR [SI],AH**. Следующий байт 046h уже нетрудно «угадать» – **INC SI**. Кстати, посмотрим, что же интересного в этом коде:



Третий бит равен нулю! Выходит команда должна выглядеть как **INC AH**! (Что кстати, выглядит непротиворечиво смысле дешифровщика). Однако, все же это **INC SI**. Почему мы решили, что

третий бит – флаг размера? Ведь Intel этого никак не гарантировала! А команда '**INC byte**' вообще выражается через дополнительный код, что на байт длиннее.

Выходит, что как ни полезно знать архитектуру инструкций, все же таблицу кодов команд хотя бы местами надо просто выучить. Иначе можно впасть в глубокое заблуждение и совершить грубые ошибки. Хотя с другой стороны, знание архитектуры порой очень и очень помогает.

Kris