Программные системы и вычислительные методы
Правильная ссылка на статью:
Гусенко М.Ю. Создание обобщенной нотации программного интерфейса процессоров x86 для автоматизированного построения дизассемблера // Программные системы и вычислительные методы. 2024. № 2. DOI: 10.7256/2454-0714.2024.2.70951 EDN: EJJSYT URL: https://nbpublish.com/library_read_article.php?id=70951

Создание обобщенной нотации программного интерфейса процессоров x86 для автоматизированного построения дизассемблера

Гусенко Михаил Юрьевич

ORCID: 0009-0007-0524-5604

кандидат технических наук

доцент; кафедра "Прикладная и бизнес-информатика"; Московский технологический университет

119454, Россия, г. Москва, ул. Вернадского, 78, оф. 418

mikegus@yandex.ru

Статья из рубрики "Математическое и программное обеспечение новых информационных технологий"

DOI:

10.7256/2454-0714.2024.2.70951

EDN:

EJJSYT

Дата направления статьи в редакцию:

04-06-2024

Аннотация: Предметом исследования является процесс обратного инжиниринга (обратной разработки) программ с целью получения их исходного кода на языках низкого или высокого уровня для процессоров с архитектурой x86, программный интерфейс которых разрабатывается компаниями Intel и AMD. Объектом исследования являются технические спецификации команд, представленные непосредственно в документации, выпускаемой этими компаниями. Исследована интенсивность обновления документации на процессоры и обоснована необходимость разработки технологических подходов, направленных на автоматизированное построение дизассемблера с учетом регулярно выпускаемых и частых обновлений программного интерфейса процессоров. В статье представлен способ обработки документации с целью получения обобщенной, формализованной и единообразной спецификации команд процессоров для дальнейшей автоматизированной трансляции ее в программный код дизассемблера. Исследована документация производителей процессоров семейства x86 непосредственно в виде,

публикуемом Intel и AMD. На основе встроенных средств автоматизации Microsoft Office написан ряд программ верификации текстов и генерирования выходной спецификации команд процессоров. В статье представлены два основных результата: первый – это разбор различных вариантов описания команд, представленных в документации Intel и AMD, и лаконичное сведение этих описаний к однообразной форме представления; второй – комплексный синтаксический анализ нотаций описания машинного кода и формы представления каждой команды на языке ассемблера. С учетом с некоторых дополнительных деталей описания команд (например, допустимого режима работы процессора при исполнении команды), это позволило создавать обобщенное описание команды для трансляции описания в код дизассемблера. К числу результатов исследования можно отнести выявление ряда ошибок в как текстах документации, так и в работе существующих промышленных дизассемблеров, построенных, как показывает анализ их реализации, с применением кодирования вручную. Выявление таких ошибок в существующем инструментарии обратного инжиниринга является косвенным результатом авторского исследования.

Ключевые слова:

дизассемблер, реинжиниринг программ, ассемблер, микропроцессоры, спецификация команд, синтаксис машинного кода, синтаксис ассемблера, документация x86 Intel, документация x86 AMD, режим работы процессора

1. Введение

При обратном инжиниринге (реинжиниринге) исполняемых программ для машины фон Неймана необходимо решать задачу разделения бинарного кода команд и данных, переводя при этом машинный код команд в эквивалентное символическое представление на языке ассемблера и, тем самым, осуществляя дизассемблирование программы.

Построение программы перевода машинного кода в символьную форму для процессоров семейства x86 хотя и имеет столь же длинную историю, как и история самого этого семейства, но на взгляд автора не имеет изящного технологичного решения. Доступные тексты программ реализации дизассемблеров показывают [1,2], что его разработчики вручную кодируют процедуры распознавания машинного кода отдельных команд. Т.е. их реализации дизассемблеров технологически сложны как в написании собственно программного кода, так и в его тестировании. С учетом того, что программный интерфейс процессоров непрерывно расширяется и имеет линейную динамику приращения множества команд на протяжении более чем 20 лет, что отражается в довольно частом обновлении документации производителей, то такие программы дизассемблеров морально устаревают довольно быстро. Существующие технологические решения построения дизассемблеров не позволяют производить обновления этих программ синхронно с обновлениями программного интерфейса процессоров, следовательно, необходимо разрабатывать новые подходы к созданию таких программ.

2. Тенденция к расширению программного интерфейса процессоров

Процессоры архитектуры x86 на протяжении многих лет остаются самыми распространёнными устройствами для создания персональных компьютерных систем. Согласно [3] доля компьютерных систем, построенных на базе процессоров этой архитектуры, в настоящий момент составляет около 70% от общего числа применяемых

процессоров. Такая популярность этих устройств у потребителей и, соответственно, у производителей компьютерной техники объясняется как историческими причинами (обеспечивается программная совместимость с ранее разработанными и проданными компьютерами), так и постоянным стремлением производителей процессоров совершенствовать вычислительный потенциал своих продуктов. Результатом этих устремлений являются архитектурные, технологические и интерфейсные обновления процессоров. Архитектурные обновления выражаются, например, в размещении на одном кристалле процессора нескольких вычислительных ядер; технологические - в переходе на более плотную компоновку транзисторов на кристалле процессора; интерфейсные - в расширении множества исполняемых процессором команд.

x86 - это архитектура со сложным набором команд (complex instruction set computing, CISC), которая традиционно предполагает поддержку специализированных для каждого рода вычислений команд, совершающих арифметические и логические действия над данными, содержащимися в регистровой памяти небольшого объема.

Флагманы среди производителей процессоров x86 - компании Intel и AMD - за прошедшие два десятилетия разработали и внедрили целый ряд вычислительных технологий, включающих расширение регистровой памяти и множества команд. Нет оснований сомневаться в том, что подобные усовершенствования прекратятся в будущем, поскольку завоеванные на рынке позиции эти компании вряд ли решаться утратить, а CISC архитектура не предполагает иного решения, кроме как добавление новых специализированных команд к уже существующему их множеству. Единственным заметным конкурентом для Intel и AMD в части производства x86-процессоров является китайская компания Zhaoxin [4], наладившая производство с помощью тайваньской компании Via, но это скорее производитель собственно чипов, нежели разработчик, влияющий на программный интерфейс процессоров.

Предполагая, что тенденция в совершенствовании аппаратной части сохранится, логично ожидать изменений и в программном обеспечении, конкретно в том, что в нем постепенно будут использоваться вновь появившиеся возможности процессоров. Это означает, что и решение задачи по исследованию программ, например, на наличие недокументированных возможностей также потребует разработки новых или модернизации существующих средств анализа.

Традиционно анализ поведения программы включает обратный инжиниринг, составной частью которого является дизассемблирование. Расширение множества поддерживаемых процессорами команд и регистров указывает на необходимость обновления средств дизассемблирования. Очевидным путем в этом направлении является совершенствование технологии создания дизассемблеров при соблюдении следующих требований:

1. охват всего множества команд процессора;

2. возможности тонкой настройки дизассемблера на работу с командами определенного множества или поддерживаемых при определенном режиме работы процессора;

3. минимизация ручного кодирования процедур дизассемблера.

Естественным источником сведений о командах и регистровой памяти является документация производителей процессоров. Динамика обновления документации от Intel [5] указывает на периодическое обновление ее примерно через каждые 106 дней (рис. 1). Если сопоставить этот график с данными, представленными в табл. 1, то можно

заметить корреляцию дат выхода обновленной документации и новой вычислительной технологии, представленной Intel. Реализация вычислительной технологии предполагает реализацию определенного набора команд в процессоре, а значит и уточнений в документации. Однако как показывает изучение версий документации, публикация описаний новых команд происходит не единовременно. Версии документации выходят регулярно, и в них постепенно уточняются или заново публикуются описания команд программного интерфейса процессора.

AMD не следует полностью в фарватере Intel и разрабатывает собственные технологии обработки данных, реализуя поддержку оригинальных наборов команд в своих процессорах. AMD обновляет документацию реже, чем Intel – в среднем каждый 176 дней (рис. 1). И так же, как у Intel, публикация сведений о новых командах происходит постепенно.

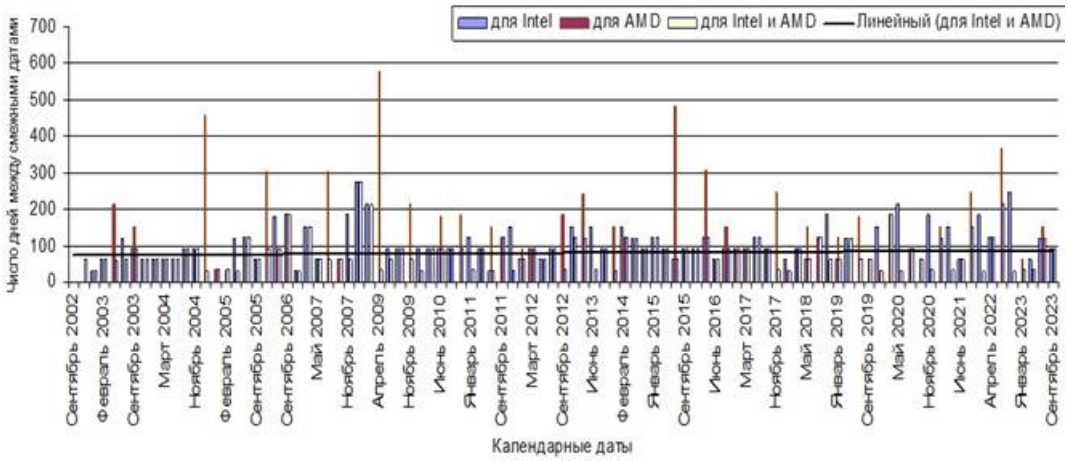


Рис. 1 Динамика выхода обновлений документации на процессоры x86 от Intel и от AMD с линейной интерполяцией среднего периода ее обновлений

Fig. 1 Dynamics of the release of documentation updates for x86 processors from Intel and AMD with linear interpolation of the average period of its updates

Объединенная динамика обновления документации Intel и AMD на рис. 1 показана через линейную интерполяцию (жирная линия). Видно, что в период с 2002 по 2023 годы она поддерживается практически неизменной - в среднем частота выхода обновлений в этом случае составляет 80 дней. Таким образом, если рассматривать официальную документацию от Intel и AMD как основной источник данных о программном интерфейсе процессоров и выпускать обновления средств дизассемблирования, поддерживающих наборы команд от обоих производителей, то это, очевидно, придется делать с периодом в 80 дней.

Отметим, что даты выхода обновлений документации непосредственно связано с публикацией описаний новых команд. Например, версия документации Intel от сентября 2023 содержит описания 4343 команд [6], это на 283 команды больше, чем версия документации от июня 2021. Между ними было выпущено еще пять версий документации. В среднем - это 56 новых команд в очередном выпуске. Следует признать, что это весьма динамичная тенденция расширения программного интерфейса процессоров.

Множества команд от Intel и от AMD пересекаются лишь частично, каждый производитель реализуется свои оригинальные подмножества команд, различающиеся как по строению машинного кода, так и по функциональному назначению.

Документация от Intel и от AMD имеет такой объем (4595 команды), что ее автоматизированная обработка для составления спецификаций машинного кода команд представляется наиболее перспективным подходом к автоматизации построения дизассемблера.

Помимо объема текста документации следует отметить и качество исполнения ее текста, в котором регулярно присутствуют опечатки, неточности и, кроме того, Intel может поменять синтаксис и способы описания команд. Проделать анализ таких особенностей построения текста, выявить и устранить неоднозначности и ошибки в описаниях команд без автоматизированного подхода практически невыполнимо. Такие же претензии можно адресовать и к документации от AMD.

Табл. 1. Обозначения основных технологий обработки данных в процессорах x86 и даты их анонсирования компанией Intel

Table 1. Designations of the main data processing technologies in x86 processors and the date of their announcement by Intel

Технология	Дата анонсирования
SSE	26.02.1999
SSE2	20.11.2000
SSE3	02.02.2004
SSSE3	26.07.2006
SSE4.1	27.09.2006
SSE4.2	27.09.2006
AESNI	01.03.2008
PCLMULQDQ	01.03.2008
AVX	01.03.2008
AVX2	04.06.2013
AVX-512	01.07.2013
RDRAND	29.04.2012
AMX	28.06.2020

Единообразный и выверенный список команд вместе с их детализированными нотациями может позволить автоматически генерировать текст программы дизассемблера. Идея подхода, когда на вход обрабатывающей программы подается формальное описание языка программирования и эта программа автоматически генерирует программу распознавания текстов на этом языке, апробирована в таких системах как YACC, Bison и LEXX. В них по грамматическому описанию языка на автоматной или контекстно-свободной грамматике строится часть программы разбора текстов на этом языке.

Эту идею можно применить в отношении автоматной грамматики машинного кода процессоров x86 и, подавая на вход специального транслятора спецификации команд, получить сгенерированный код программы дизассемблера на некотором языке программирования высокого уровня.

Чтобы детально изложить эту идею, необходимо вначале рассмотреть строение документации процессоров с архитектурой x86 от Intel и AMD.

3. Структура документации Intel и AMD

Автору удавалось знакомиться с описанием программного интерфейса процессоров по фирменной документации Intel, начиная с 1995 года. Содержание и объем документации,

естественно, менялись, а вот структура описания отдельных команд процессора остается практически неизменной. В версии, используемой для написания статьи [6], описание отдельной команды имеет следующий типовой вид (рис. 2), где характерными элементами являются:

1) **заголовок**, в котором указаны: мнемоника (иногда мнемоники семейства команд) команды и краткое описание ее семантики (рис. 2(1)): 'ADD' - мнемоника, 'Add' - краткое описание семантики команды;

2) таблица описания мнемоники команд (рис. 2(2)), включая синтаксическую нотацию машинного кода в столбце "Opcode"; синтаксическую нотацию ассемблерного кода команды (или мнемонику) в столбце "Instruction" и прочую информацию. Более детальное описание столбцов таких таблиц представлено в п. 4. Кроме того, на рис. 2(2) представлен только один из вариантов таблицы, которых в документации Intel шесть видов (рис. 5, 6, 7, 8, 9, 10). Отметим, что структура машинного кода не отделена в некоторых таблицах (табл. 6) от синтаксиса команды и такое разделение необходимо будет произвести;

3) таблица кодирования операндов команды (рис. 2(3)), которая всегда следует за таблицей описания команды, и в которой представлены варианты кодирования явных операндов команды и способ (чтение/запись) их использования;

4) развернутое текстовое описание работы команды (рис. 2(4));

5) описание операционной семантики команды на Algol-подобном языке (рис. 2(5)). В документации представлено описание элементов этого языка, однако его фактическое применение для описания команд несколько произвольное, применяется авторами каждого описания команды часто без согласования с исходной нотацией языка и апеллирует скорее к интуитивному восприятию читателя, чем подходит для формальной трансляции;

6) описание воздействия на флаги из регистра [R|E]FLAGS (Здесь и далее в статье используется метаязык для описания грамматики в расширенной нотации Бэкуса-Наура (РБНФ) [7]) процессора, которое производит исполнение команды (рис. 2(6));

7) различные дополнительные аспекты поведения процессора в случае распознавания им ошибочного вычислительного контекста исполнения команды (рис. 2(7)).

Из всего этого многообразия информации для построения входных спецификаций дизассемблера существенным является только содержимое таблицы описания мнемоники команд (рис. 2(2)) и таблицы кодирования операндов команды (рис. 2(3)). Из-за разных временных моментов создания и обновления документации описание команд представляется таблицами, отличающимися по строению друг от друга. В рассматриваемой версии документации описание представлено в таблицах шести видов (рис. 5, 6, 7, 8, 9, 10). Для некоторых таблиц показана связь между знаками сносок (используются различные знаки) и текстовым представлением необходимых частностей. Большинство таких сносок указывает на ограничения в машинном коде, которые влияют на дизассемблированное представление команды. Эти ограничения необходимо учитывать, поскольку они позволяют отличить корректный машинный код команды от некорректного, который при попытке исполнения заставит процессор сгенерировать исключение #UD (Invalid Opcode/Undefined Opcode). Соответственно, дизассемблер, решая задачу разделения бинарного кода команд и данных в программе для машины фон Неймана, должен отличать корректный код от некорректного. И указанные детали

описания команд могут быть использованы для этого.

Описания команд у Intel сгруппированы в следующих разделах [6]:

· Группа глав, описывающих общие команды архитектуры под общим заголовком "Instruction set reference". На момент написания статьи рассматривалось 4289 синтаксических нотаций команд (см. далее) этого вида совместно у Intel и у AMD. Из них оригинальными для Intel являются 2383 команды, оригинальными для AMD являются 252 команды, общие для Intel и AMD - это 1654 команды. И это подсчеты для команд, исключая дублирование, т.е., например, 'CMPS m8, m8' и 'CMPSB' считаются одной командой, т.к. машинный код у них одинаковый.

· Команды, относящиеся к расширению SAFER MODE, встречаются только у Intel и, по сути, представлены только одной командой GETSEC, входными параметрами для которой является значение регистров, задающих выполнения частной функциональности (leaf functions).

· Команды, уникальные для процессоров INTEL® XEON PHI™, встречаются только у Intel и это 33 команды.

· Команды поддержки расширения Virtual Machine Extensions (VMX) встречаются только у Intel и это 17 команд.

· Команды поддержки расширения Intel® Software Guard Extensions (Intel® SGX) встречаются только у Intel и это три команды, допускающие на основе входных значений выполнение leaf-функций, как и для GETSEC.

В документации от AMD [8] описание команды представлено на рис. 3 и включает:

1) заголовок, который как и у Intel включает мнемонику операции команды и краткое описание ее семантики;

2) описание операционной семантики в словесной форме;

3) синтаксические формы отдельных команд;

4) описание воздействия на флаги из регистра [R|E]FLAGS процессора;

5) различные дополнительные аспекты поведения процессора в случае распознавания им ошибочного вычислительного контекста исполнения команды.

Если сравнить эти два варианта описания команд, то предпочтительным вариантом для построения дизассемблера представляется описание от Intel. Во-первых, оно более детализированное и включает подробное описание синтаксиса машинного кода и режимов работы процессора, в которых допустимы те или иные команды. Во-вторых, оно содержит удобное для автоматизированного анализа описание кодирования отдельных операндов команд в систематическом виде, в отличие от словесного описания у AMD. В-третьих, документация от Intel содержит описание бóльшего числа команд, в частности расширение Intel AVX-512, GETSEC, Intel Xeon PHI, VMX, SGX, которые не поддерживаются AMD.

Однако следует отметить, что описание используемых флагов регистра [R|E]FLAGS процессора в документации от AMD удобнее для автоматизированной обработки при составлении схемы информационных связей команд реальной программы, чем вариант от Intel.

Кроме того, у AMD описаны команды, которые отсутствуют в документации Intel (подробности далее). Поэтому спецификации множества описаний команд, создаваемое по документации Intel, можно дополнить описаниями команд от AMD и получить обобщенное множество команд для архитектуры x86.

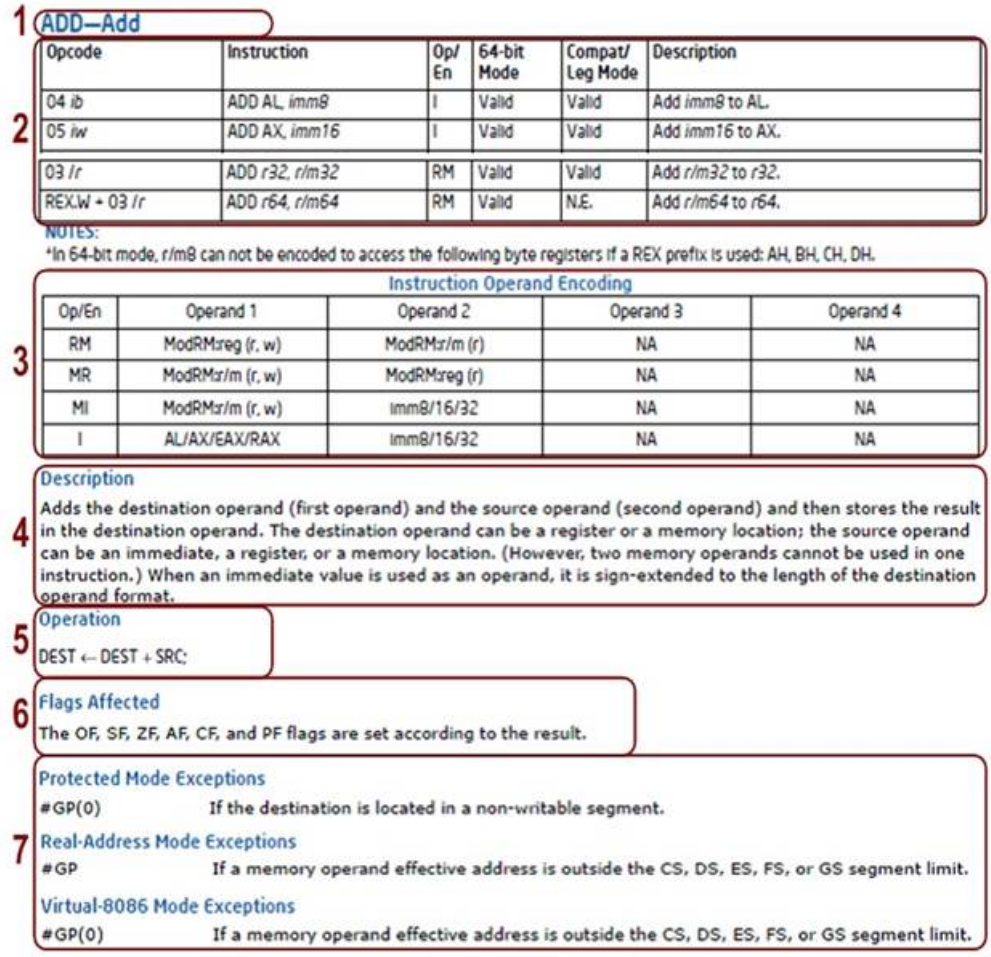
И у Intel, и у AMD семантическая форма команды может быть как в префиксном представлении, например, ADD DEST, SRC, где ADD - обозначение операции, DEST, SRC - обозначения операндов приемника (**DEST**ination) и источника (**S**ou**RC**e) значений, так и в инфиксном представлении вида: DEST ¬ DEST + SRC (рис. 2(5)).

У обоих производителей процессоров одной семантической форме описания команды ADD DEST, SRC соответствует, как правило, несколько синтаксических форм вида: ADD AL, imm8; ADD AX, imm16; …; ADD r64, r/m64. По числу синтаксических форм можно судить о мощности программного интерфейса процессора.

Например, в группе глав, описывающих общие команды архитектуры x86 у Intel [6], число семантических форм 764, а синтаксических форм 3975.

Синтаксические формы команд кодируются различными вариантами машинного кода. И поскольку для многих аспектов статьи будет необходимо ссылаться на конкретные примеры команд, то для этого будет использоваться нотация вида: O(pcode)='машинный код'; I(nstruction)='синтаксис команды'. Например, для команды на рис. 2 эта нотация будет иметь вид: O='04 ib'; I='ADD AL, imm8'. Для команды на рис. 3 эта нотация будет иметь вид: O='14 ib'; I='ADC AL, imm8' и т.д.

В нотации машинного кода числа представлены в шестнадцатеричном виде, т.е. в записи O='37'; I='AAA' число '37' - это число в шестнадцатеричном представлении. В некоторых случаях, когда будут упоминаться биты машинного кода, то они будут записаны суффиксом 'b' (binary), например, '101b' - это три бита '1', '0' и '1'.



1 ADD—Add

2

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
03 */r*	ADD *r32*, *r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 */r*	ADD *r64*, *r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

3 Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA

4 Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

5 Operation

DEST ← DEST + SRC;

6 Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

7 Protected Mode Exceptions

#GP(0) If the destination is located in a non-writable segment.

Real-Address Mode Exceptions

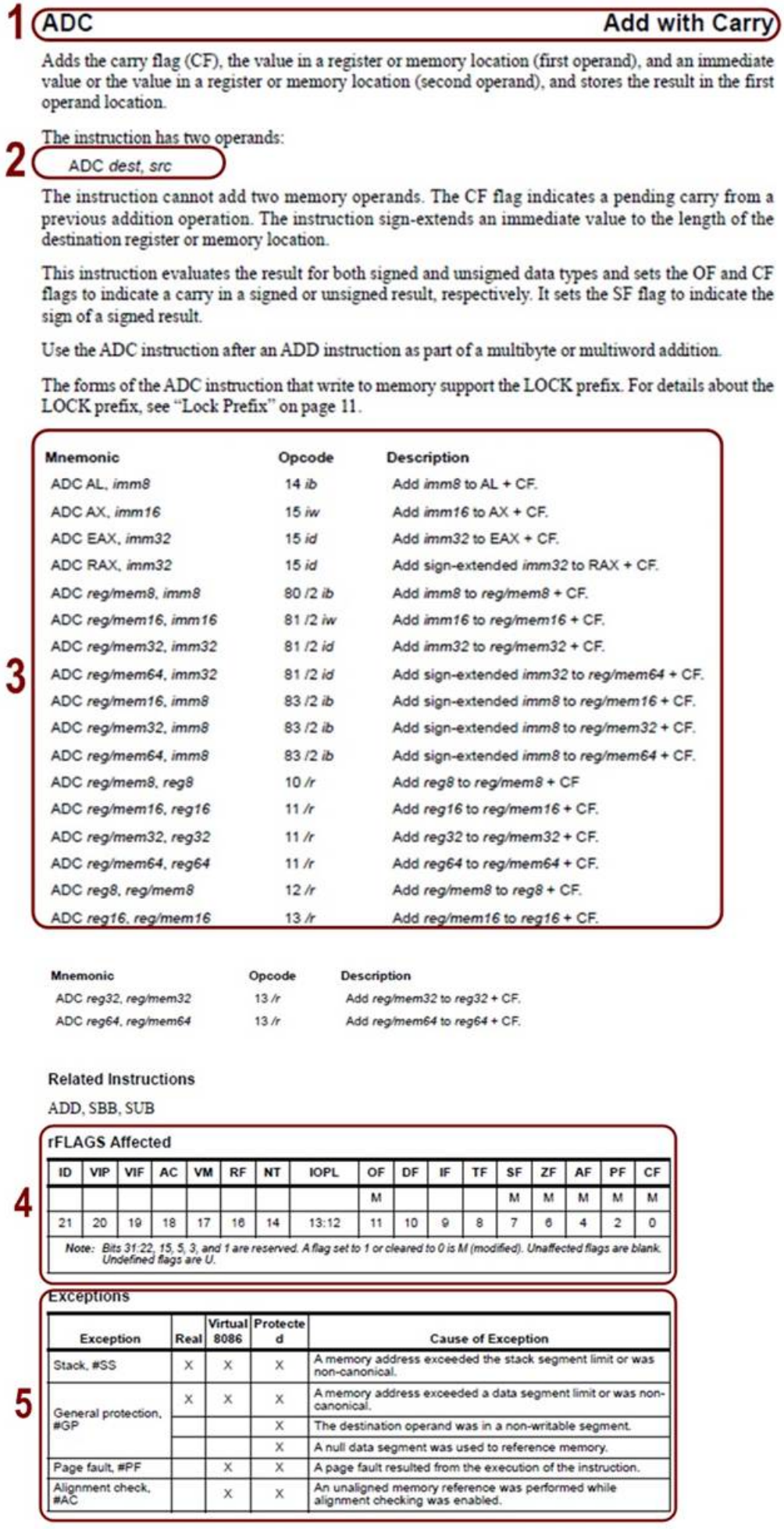
#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Рис. 2. Типовой вид описания команды в документации Intel

Fig. 2. A typical view of the instruction description in the Intel documentation



1

ADC — Add with Carry

Adds the carry flag (CF), the value in a register or memory location (first operand), and an immediate value or the value in a register or memory location (second operand), and stores the result in the first operand location.

The instruction has two operands:

2

ADC *dest, src*

The instruction cannot add two memory operands. The CF flag indicates a pending carry from a previous addition operation. The instruction sign-extends an immediate value to the length of the destination register or memory location.

This instruction evaluates the result for both signed and unsigned data types and sets the OF and CF flags to indicate a carry in a signed or unsigned result, respectively. It sets the SF flag to indicate the sign of a signed result.

Use the ADC instruction after an ADD instruction as part of a multibyte or multiword addition.

The forms of the ADC instruction that write to memory support the LOCK prefix. For details about the LOCK prefix, see "Lock Prefix" on page 11.

3

Mnemonic	Opcode	Description
ADC AL, *imm8*	14 *ib*	Add *imm8* to AL + CF.
ADC AX, *imm16*	15 *iw*	Add *imm16* to AX + CF.
ADC EAX, *imm32*	15 *id*	Add *imm32* to EAX + CF.
ADC RAX, *imm32*	15 *id*	Add sign-extended *imm32* to RAX + CF.
ADC *reg/mem8, imm8*	80 /2 *ib*	Add *imm8* to *reg/mem8* + CF.
ADC *reg/mem16, imm16*	81 /2 *iw*	Add *imm16* to *reg/mem16* + CF.
ADC *reg/mem32, imm32*	81 /2 *id*	Add *imm32* to *reg/mem32* + CF.
ADC *reg/mem64, imm32*	81 /2 *id*	Add sign-extended *imm32* to *reg/mem64* + CF.
ADC *reg/mem16, imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem16* + CF.
ADC *reg/mem32, imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem32* + CF.
ADC *reg/mem64, imm8*	83 /2 *ib*	Add sign-extended *imm8* to *reg/mem64* + CF.
ADC *reg/mem8, reg8*	10 /*r*	Add *reg8* to *reg/mem8* + CF
ADC *reg/mem16, reg16*	11 /*r*	Add *reg16* to *reg/mem16* + CF.
ADC *reg/mem32, reg32*	11 /*r*	Add *reg32* to *reg/mem32* + CF.
ADC *reg/mem64, reg64*	11 /*r*	Add *reg64* to *reg/mem64* + CF.
ADC *reg8, reg/mem8*	12 /*r*	Add *reg/mem8* to *reg8* + CF.
ADC *reg16, reg/mem16*	13 /*r*	Add *reg/mem16* to *reg16* + CF.

Mnemonic	Opcode	Description
ADC *reg32, reg/mem32*	13 /*r*	Add *reg/mem32* to *reg32* + CF.
ADC *reg64, reg/mem64*	13 /*r*	Add *reg/mem64* to *reg64* + CF.

Related Instructions

ADD, SBB, SUB

rFLAGS Affected

4

ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
								M				M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to 1 or cleared to 0 is M (modified). Unaffected flags are blank. Undefined flags are U.

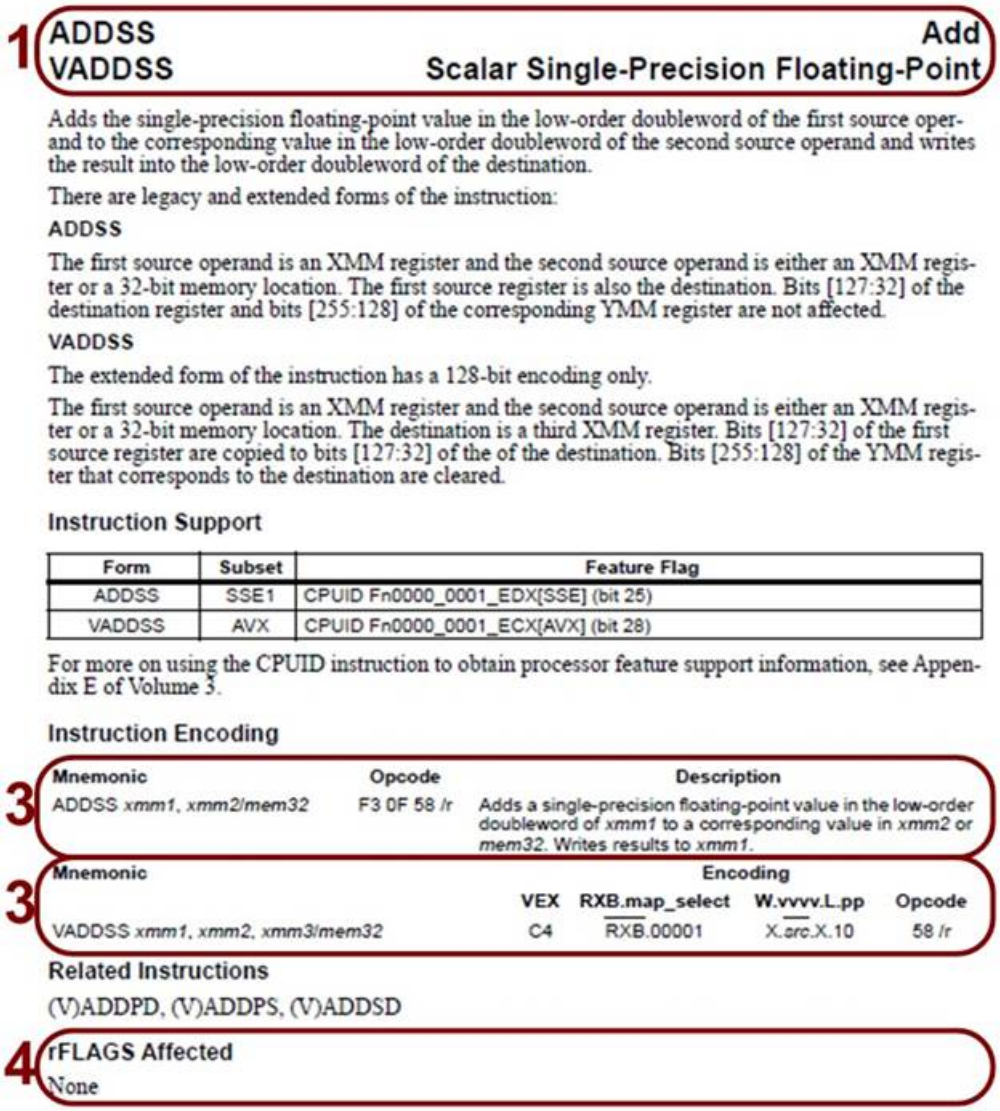
Exceptions

5

Exception	Real	Virtual 8086	Protected	Cause of Exception
Stack, #SS	X	X	X	A memory address exceeded the stack segment limit or was non-canonical.
General protection, #GP	X	X	X	A memory address exceeded a data segment limit or was non-canonical.
			X	The destination operand was in a non-writable segment.
			X	A null data segment was used to reference memory.
Page fault, #PF		X	X	A page fault resulted from the execution of the instruction.
Alignment check, #AC		X	X	An unaligned memory reference was performed while alignment checking was enabled.

Рис. 3 Типовой вид описания legacy-команды в документации AMD

Fig. 3. A typical description of the legacy instruction in the AMD documentation



1

ADDSS **Add**
VADDSS **Scalar Single-Precision Floating-Point**

Adds the single-precision floating-point value in the low-order doubleword of the first source operand to the corresponding value in the low-order doubleword of the second source operand and writes the result into the low-order doubleword of the destination.

There are legacy and extended forms of the instruction:

ADDSS

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The first source register is also the destination. Bits [127:32] of the destination register and bits [255:128] of the corresponding YMM register are not affected.

VADDSS

The extended form of the instruction has a 128-bit encoding only.

The first source operand is an XMM register and the second source operand is either an XMM register or a 32-bit memory location. The destination is a third XMM register. Bits [127:32] of the first source register are copied to bits [127:32] of the of the destination. Bits [255:128] of the YMM register that corresponds to the destination are cleared.

Instruction Support

Form	Subset	Feature Flag
ADDSS	SSE1	CPUID Fn0000_0001_EDX[SSE] (bit 25)
VADDSS	AVX	CPUID Fn0000_0001_ECX[AVX] (bit 28)

For more on using the CPUID instruction to obtain processor feature support information, see Appendix E of Volume 3.

Instruction Encoding

3

Mnemonic	Opcode	Description
ADDSS *xmm1, xmm2/mem32*	F3 0F 58 /r	Adds a single-precision floating-point value in the low-order doubleword of *xmm1* to a corresponding value in *xmm2* or *mem32*. Writes results to *xmm1*.

3

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDSS *xmm1, xmm2, xmm3/mem32*	C4	$\overline{\text{RXB}}$.00001	X.*src*.X.10	58 /r

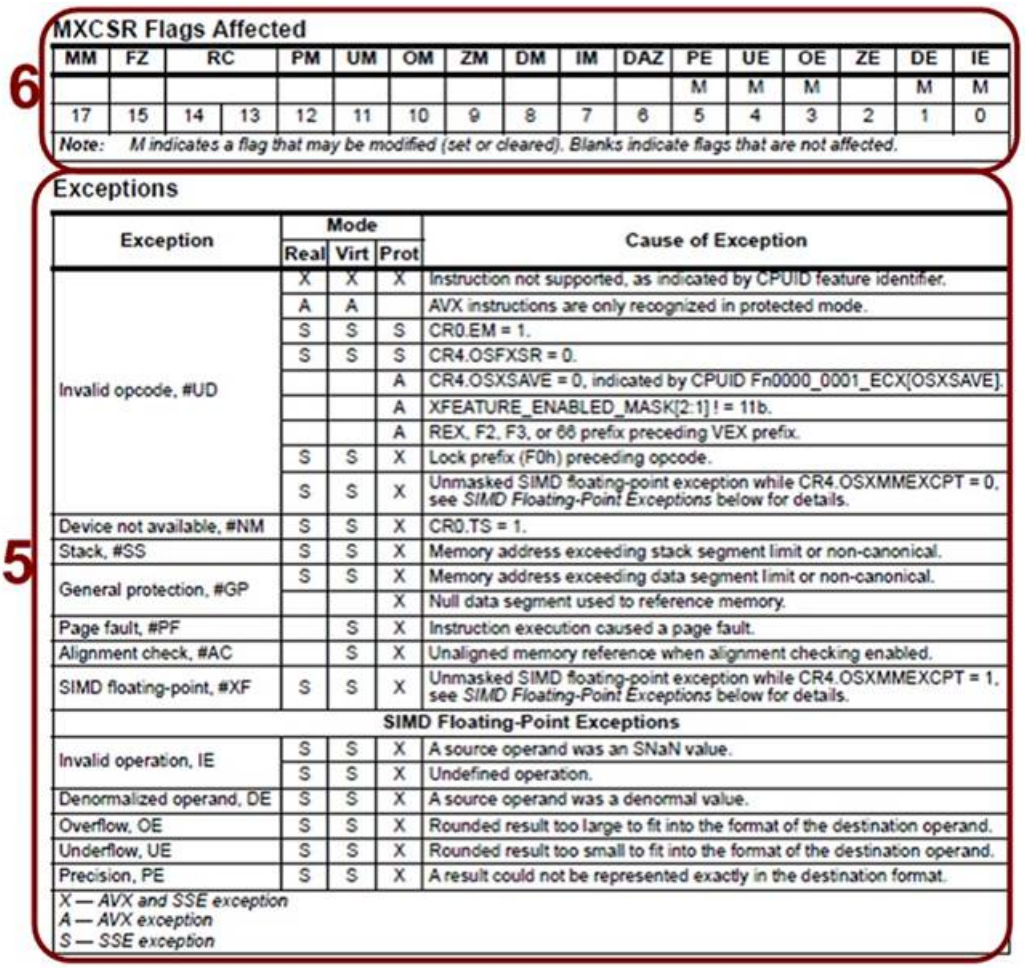
Related Instructions

(V)ADDPD, (V)ADDPS, (V)ADDSD

4

rFLAGS Affected

None



MXCSR Flags Affected

MM	FZ	RC		PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE
											M	M	M		M	M
17	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Note: M indicates a flag that may be modified (set or cleared). Blanks indicate flags that are not affected.

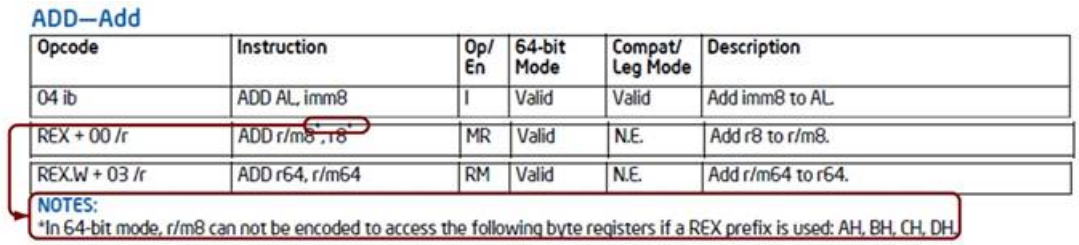
Exceptions

Exception	Mode			Cause of Exception
	Real	Virt	Prot	
Invalid opcode, #UD	X	X	X	Instruction not supported, as indicated by CPUID feature identifier.
	A	A		AVX instructions are only recognized in protected mode.
	S	S	S	CR0.EM = 1.
	S	S	S	CR4.OSFXSR = 0.
			A	CR4.OSXSAVE = 0, indicated by CPUID Fn0000_0001_ECX[OSXSAVE].
			A	XFEATURE_ENABLED_MASK[2:1] ! = 11b.
			A	REX, F2, F3, or 66 prefix preceding VEX prefix.
	S	S	X	Lock prefix (F0h) preceding opcode.
	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 0, see *SIMD Floating-Point Exceptions* below for details.
Device not available, #NM	S	S	X	CR0.TS = 1.
Stack, #SS	S	S	X	Memory address exceeding stack segment limit or non-canonical.
General protection, #GP	S	S	X	Memory address exceeding data segment limit or non-canonical.
			X	Null data segment used to reference memory.
Page fault, #PF		S	X	Instruction execution caused a page fault.
Alignment check, #AC		S	X	Unaligned memory reference when alignment checking enabled.
SIMD floating-point, #XF	S	S	X	Unmasked SIMD floating-point exception while CR4.OSXMMEXCPT = 1, see *SIMD Floating-Point Exceptions* below for details.
SIMD Floating-Point Exceptions				
Invalid operation, IE	S	S	X	A source operand was an SNaN value.
	S	S	X	Undefined operation.
Denormalized operand, DE	S	S	X	A source operand was a denormal value.
Overflow, OE	S	S	X	Rounded result too large to fit into the format of the destination operand.
Underflow, UE	S	S	X	Rounded result too small to fit into the format of the destination operand.
Precision, PE	S	S	X	A result could not be represented exactly in the destination format.

X — AVX and SSE exception
A — AVX exception
S — SSE exception

Рис. 4 Типовой вид описания VEX- или XOP-команд в документации AMD

Fig. 4. A typical description of VEX or XOP instructions in AMD documentation



ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
REX + 00 /r	ADD r/m8*, r8	MR	Valid	N.E.	Add r8 to r/m8.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Рис. 5. Пример таблицы описания команд вида 1. Формат используется для описания команд ранних версий процессоров x86 (legacy-команд)

Fig. 5. An example of a instruction description table of the form 1. The format is used to describe instructions from early versions of x86 processors (legacy instructions)

ADCX—Unsigned Integer Addition of Two Operands With Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.

Рис. 6. Пример таблицы описания команд вида 2. Формат используется для различных команд

Fig. 6. An example of a instruction description table of type 2. The format is used for various instructions

F2XM1—Compute 2^x-1

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with ($2^{ST(0)}$ – 1).

Рис. 7. Пример таблицы описания команд вида 3. Формат используется для команд математического сопроцессора (Numeric Data Processor, NDP)

Fig. 7. An example of a instruction description table of the form 3. The format is used for instructions of a mathematical coprocessor (Numeric Data Processor, NDP)

XACQUIRE/XRELEASE—Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.

Рис. 8. Пример таблицы описания команд вида 4. Формат используется для различных команд

Fig. 8. An example of a instruction description table of the form 4. The format is used for various instructions

GETSEC[CAPABILITIES] - Report the SMX Capabilities

Opcode	Instruction	Description
NP 0F 37 (EAX = 0)	GETSEC[CAPABILITIES]	Report the SMX capabilities. The capabilities index is input in EBX with the result returned in EAX.

Рис. 9. Пример таблицы описания команд вида 5. Формат используется для команд семейства GETSEC

Fig. 9. An example of a instruction description table of the form 5. The format is used for instructions of the GETSEC family

INVEPT— Invalidate Translations Derived from EPT

Opcode/ Instruction	Op/En	Description
66 0F 38 80 INVEPT r64, m128	RM	Invalidates EPT-derived entries in the TLBs and paging-structure caches (in 64-bit mode).
66 0F 38 80 INVEPT r32, m128	RM	Invalidates EPT-derived entries in the TLBs and paging-structure caches (outside 64-bit mode).

Рис. 10. Пример таблицы описания команд вида 6. Формат используется для команд VMX

Fig. 10. An example of a instruction description table of the form 6. The format is used for VMX instructions

а

Mnemonic	Encoding			
	VEX	RXB.map_select	W.vvvv.L.pp	Opcode
VADDPD *xmm1, xmm2, xmm3/mem128*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.0.01	58 /r
VADDPD *ymm1, ymm2, ymm3/mem256*	C4	$\overline{\text{RXB}}$.00001	X.$\overline{src}$.1.01	58 /r

б

Mnemonic	Encoding			
	XOP	RXB.map_select	W.vvvv.L.pp	Opcode
VFRCZPD *xmm1, xmm2/mem128*	8F	$\overline{\text{RXB}}$.09	0.1111.0.00	81 /r
VFRCZPD *ymm1, ymm2/mem256*	8F	$\overline{\text{RXB}}$.09	0.1111.1.00	81 /r

Рис. 11. Пример таблицы описаний (а) VEX-, (б) XOP-команд в документации AMD

Fig. 11. An example of a table of descriptions of (a) VEX, (b) XOP instructions in the AMD

documentation

По строению машинного кода и по нотации его описания команды Intel можно разделить на варианты:

· EVEX-команды. В нотации машинного кода присутствует префикс 'EVEX': O='EVEX.512.66.0F38.WIG DF /r; I=VAESDECLAST zmm1, zmm2, zmm3/m512';

· VEX-команды. В нотации машинного кода присутствует префикс 'VEX': O='VEX.256.66.0F38.WIG DF /r; I=VAESDECLAST ymm1, ymm2, ymm3/m256';

· REX-команды. В нотации машинного кода присутствует префикс 'REX': O='F2 REX 0F 38 F0 /r; I=CRC32 r32, r/m8';

· Legacy-команды. В нотации машинного кода не присутствует на один из указанных префиксов: O='0F 38 F0 /r; I=MOVBE r16, m16'.

Еще одной особенностью описания команд от Intel является указание на некоторые ограничения/исключения в строении машинного кода команд. Такие указания, как правило, передаются сноской из элемента таблицы мнемоник (табл. 5), в которой ссылка может быть указана как на столбец таблицы целиком, так и на отдельные значения в ее ячейках. Учесть такие ограничения довольно легко в постобработке дизассемблированной команды (п. 9).

В части legacy-команд документация от AMD придерживается представления, схожего с Intel'овским, но в отношении остальных команд, например, VEX-команд, AMD использует для описания табличное представление (рис. 11,а), что осложняет сопоставление с Intel'овским эквивалентом. По нотации машинного кода от AMD, представленной в виде таблицы (рис. 11), можно сгенерировать представление, похожее на то, какое использует Intel. Но это будет лишь примерный вариант, поскольку Intel не придерживается строгих правил для обозначения машинного кода команд. Так, O='VEX.128.0F38.W0 F5 /r'; I='BZHI reg32, reg/mem32, reg32' в варианте от AMD будет соответствовать O='VEX.LZ.0F38.W0 F5 /r'; I='BZHI r32a, r/m32, r32b' в варианте от Intel.

Кроме того, в документации от AMD присутствует описание команд, которые не встречаются у Intel. К числу их относятся:

· команды поддержки технологии 3DNow! Машинный код их похож на legacy-строение. Например, O='0F 0F /r 0C'; I='PI2FW mmx1, mmx2/mem64'. Отличие касается байтов кода операции - они не размещаются на смежных позициях;

· XOP-команды, включающие префикс XOP, имеющие строение, похожее на VEX-команды (рис. 11,б), но все же различающиеся в деталях (здесь Intel-эквивалента нет вообще, поэтому команды представлены в авторской интерпретации). Например, O='XOP.128.MAP9.W0 81 /r'; I='VFRCZPD xmm1, xmm2/mem128';

· некоторые VEX-команды, не описанные у Intel. Например, O='VEX.128.66.0F3A.W0 78 /r /is4'; I='VFNMADDPS xmm1, xmm2, xmm3/mem128, xmm4'. В этой же категории присутствуют команды с пятью операндами. Такого вида команды вообще не встречаются у Intel. Это, например, O='VPERMIL2PD xmm1, xmm2, xmm3/mem128, xmm4, m2z';I='VEX.128.66.0F3A.W0 49 /r is5';

· некоторые команды legacy-формата, не описанные у Intel. Например, O='0F 77'; I='EMMS'.

4. Некоторые элементы таблиц описания команд

Таблицы синтаксического описания команды (таблицы мнемоник) в документации Intel (рис. 5, 6, 7, 8, 9, 10) имеют следующие заголовки столбцов:

· "Opcode" - заголовок столбца, содержащего нотацию машинного кода команды. Чаще всего является отдельным столбцом (рис. 5, 6, 7), но может быть совмещен с описанием синтаксиса команды - со столбцом "Instruction" (рис. 8).

· "Instruction" - заголовок столбца, содержащего синтаксическую нотацию команды. Чаще всего является отдельным столбцом (рис. 5, 6, 7), но может быть совмещен с описанием синтаксиса команды - со столбцом "Opcode" (рис. 8).

· "Op/En" - заголовок столбца, содержащего отсылку к строкам таблицы "Instruction Operation Encoding", следующей за таблицей описания синтаксиса команды (рис. 2(3)).

· "64-bit Mode" или "64/32bit Mode Support" - заголовок столбца указывает на поддержку команды в 64-битном и/или 32-битном режимах работы процессора. Значения, представленные в таких столбцах, описаны в п. 6.

· "Compatibility/Legacy Mode" - заголовок столбца указывает на поддержку команды в режиме совместимости и/или указание на вхождение в семейство legacy-команд IA-32. Значения, представленные в таких столбцах, описаны в п. 6.

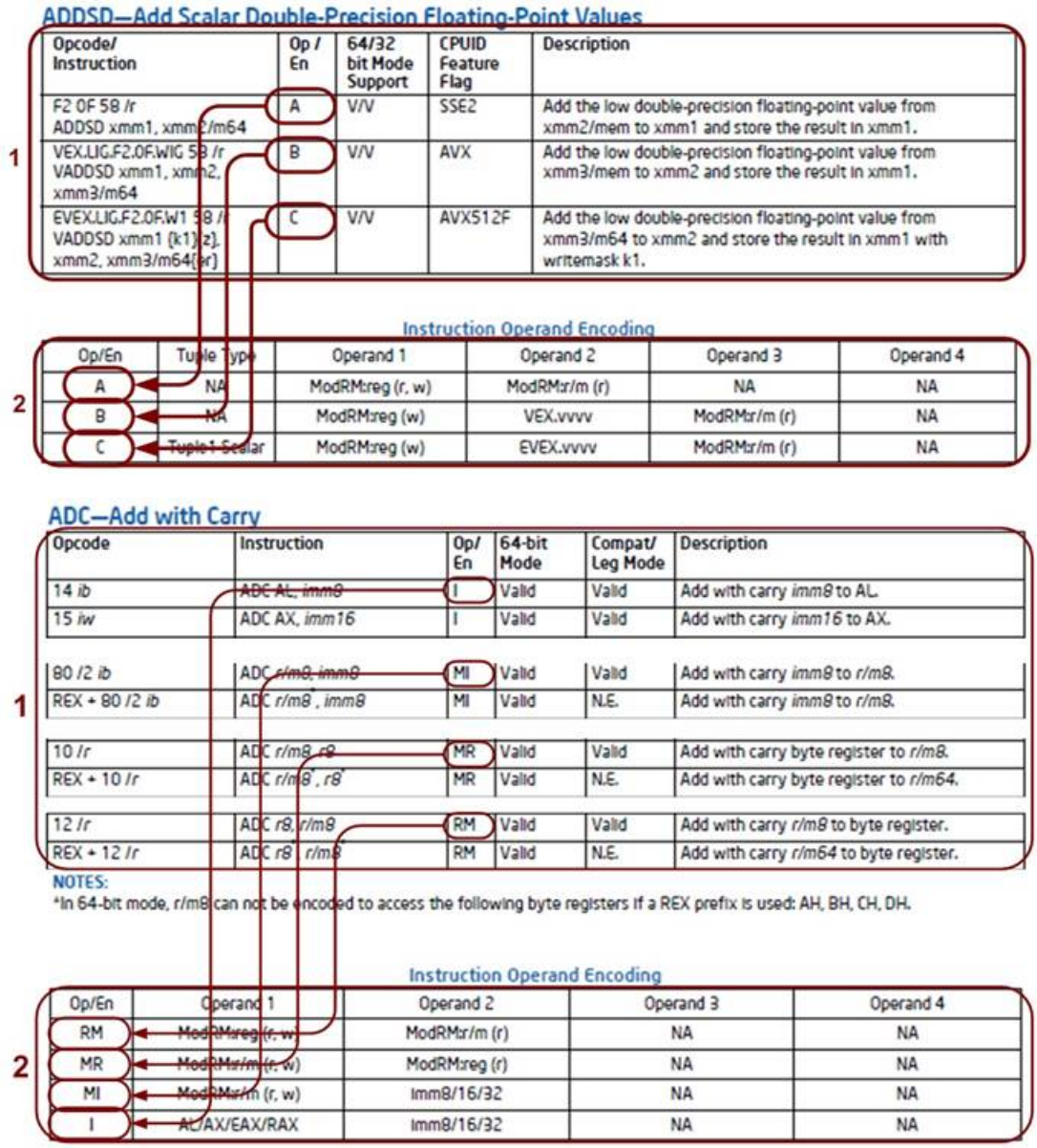
· "CPUID Feature Flag" - заголовок столбца указывает на флаг в значениях, выдаваемых командой O='0F A2'; I='CPUID' процессора, и указывает на поддержку в процессоре определенных технологий обработки данных. Значения, представленные в таких столбцах, разбираются в п. 7.

· Столбец "Description" содержит краткое словесное описание семантики вычислений, выполняемых командой. Примеры такого описания представлены в рис. 5, 6, 7, 8, 9, 10.

5. Кодирование операндов в описании команд

Кодировка операндов каждой команды специальным образом описывается в отдельной таблице, следующей за таблицей описания команд данного вида (рис. 12). Кодировка операндов в исходной таблице представлена в столбце "Op/En" и включает специальные буквенные обозначения, адресующие к методу кодирования операндов в отдельной команде. Значения в столбце "Op/En" (рис. 12(1)) применяются только для соотнесения с описаниями в следующей за ней таблице кодирования операндов (рис. 12(2)). И в этой таблице (Согласно п. "3.1.1.4 Operand Encoding Column in the Instruction Summary Table" [6]) обозначение "r" в паре круглых скобок указывает на чтение операнда, обозначение "w" в паре круглых скобок указывает на запись в этот операнд при совершении вычислений процессором.

Примеры многообразных описаний кодирования операндов в документации Intel представлены в столбце 1 (табл. 2), а полный набор обозначений типов кодирования, которые используются для построения спецификации команд в авторской обобщенной нотации, представлен в столбце 2 (табл. 2). Таким образом, многообразие текстовых описаний от Intel может быть сведено к восьми типам декодирования в дизассемблере.



ADDSD—Add Scalar Double-Precision Floating-Point Values

1

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

2

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

ADC—Add with Carry

1

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	I	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	I	Valid	Valid	Add with carry *imm16* to AX.
80 /2 *ib*	ADC *r/m8*, *imm8*	MI	Valid	Valid	Add with carry *imm8* to *r/m8*.
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	MI	Valid	N.E.	Add with carry *imm8* to *r/m8*.
10 /r	ADC *r/m8*, *r8*	MR	Valid	Valid	Add with carry byte register to *r/m8*.
REX + 10 /r	ADC *r/m8**, *r8**	MR	Valid	N.E.	Add with carry byte register to *r/m64*.
12 /r	ADC *r8*, *r/m8*	RM	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /r	ADC *r8**, *r/m8**	RM	Valid	N.E.	Add with carry *r/m64* to byte register.

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

2

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA

Рис. 12. Основные таблицы (1) описания команды с семантическими мнемониками ADDSD и ADC, и таблицы (2) "Instruction Operand Encoding" кодирования операндов в документации Intel. Показаны связи значений из столбца "Op/En" основных таблиц (1) и таблиц (2)

Fig. 12. The main tables (1) describe instructions with semantic mnemonics ADD and ADC, and tables (2) "Instruction Operand Encoding" encoding operands in Intel documentation. The relationships of the values from the "Op/En" column of the main tables (1) and tables (2) are shown

Табл. 2. Представление вариантов кодирования операндов и эквивалентные типы декодеров операндов

Tabl. 2. Representation of operand encoding options and equivalent types of operand decoders

Операнд в описании Intel	Тип декодера операнда в дизассемблере	Пример команды
1	2	3
AL/AX/EAX/RAX		O='REX.W + 15 id'; I='ADC

AL/AX/EAX/RAX ST(0)	Value	O='REX.W + 15 id'; I='ADC RAX, imm32' O='DC C0+i'; I='FADD ST(i), ST(0)'
ModRM:r/m; ModRM:rm BaseReg(R): VSIB ST(i)	ModRM	O='NP 0F AE /2'; I='LDMXCSR m32' O='NP 0F 1A /r'; I='BNDLDX bnd, mib' O='D8 C0+i '; I='FADD ST(0), ST(i)'
ModRM:reg	ModReg	O='VEX.L0.F2.0F.W0 93 /r'; I='KMOVD r32, k1'
VEX.vvvv VEX.1vvv EVEX.vvvv	Vvvv	O='VEX.L1.66.0F.W1 4A /r'; I='KADDD k1, k2, k3' O='EVEX.128.66.0F3A.W0 21 /r ib'; I='VINSERTPS xmm1, xmm2, xmm3/m32, imm8'
imm/imm8/16/32 offset Segment+Absolute Address	Imm	O='REX.W + 15 id'; I='ADC RAX, imm32' O='0F 84 cd'; I='JZ rel32' O='EA cp'; I='JMP ptr16:32'
opcode + rd	Opcode	O='40+ rd'; I='INC r32'
Moffs	Moffs	O='A1'; I='MOV EAX,moffs32'
NA implicit	None	O='NP 0F 38 CB /r'; I='SHA256RNDS2 xmm1,xmm2/m128, '

Отметим, что Intel в документации не всегда придерживается единообразия в описании кодировки операндов. Таблица кодирования операндов вместо типового представления (рис. 12(2)) может иметь иной вид (рис.13), в нем операнды 2-9 (Operands 2-9) являются неявными и потому в дизассемблированном представлении команды не указываются. Поскольку в рассматриваемой версии документации только шесть таких случаев описания команд, то они правкой вручную могут быть сведены к типовому описанию (рис.14).

AESDECWIDE128KL—Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

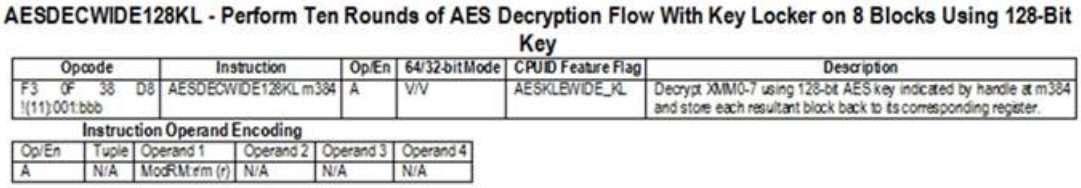
Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):001:bbb AESDECWIDE128KL m384, <XMM0-7>	A	V/V	AESKLEWIDE_KL	Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operands 2–9
A	N/A	ModRM:r/m (r)	Implicit XMM0-7 (r, w)

Рис. 13. Нетипичное описание операндов команды

Fig. 13. An atypical description of the instruction operands



AESDECWIDE128KL - Perform Ten Rounds of AES Decryption Flow With Key Locker on 8 Blocks Using 128-Bit Key

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 38 D8 !(11):001:bbb	AESDECWIDE128KL m384	A	V/V	AESKLEWIDE_KL	Decrypt XMM0-7 using 128-bit AES key indicated by handle at m384 and store each resultant block back to its corresponding register.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	N/A	ModRM:rm (r)	N/A	N/A	N/A

Рис. 14. Отредактированное описание нетипичного описания команды и ее операндов для исходной версии на рис. 13

Fig. 14. An edited description of the atypical description of the instruction and its operands for the original version in Fig. 13

6. Совместимость команд с режимами работы процессора

Столбец "64/32-bit Mode" указывает на поддержку команды в 64-битном режиме (64-bit mode), в режиме совместимости (Compatibility mode) и других режимах IA-32, которые применяются в сочетании с флагом, возвращаемым командой O='0F A2'; I='CPUID', связанным с конкретными расширениями функциональности команд. Значения в столбце (табл. 3) показаны согласно п. "3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table" [6].

Табл. 3. Значения столбца "64/32-bit Mode" на рис. 2(2)

Tabl. 3. The values of the column "64/32-bit Mode" in Fig. 2(2)

Обозначение	Описание
V или Valid	Команда поддерживается (Valid) в данном режиме
I, Invalid или Inv.	Команда не поддерживается (Invalid) в данном режиме
NE или N.E.	Указывает на то, что команда не поддерживается (Not encodable) в 64-битном режиме, но может быть частью поддерживаемых команд в других режимах
N.S.	Указывает на синтаксис команды, которой требуется префикс замены адреса в 64-битном режиме. Использование префикса замены адреса в 64-битном режиме может привести к специфическим результатам, зависящим от конкретной модели процессора.

Возможные значения столбца "Compatibility/Legacy Mode" представленны в табл. 4.

Табл. 4. Значения столбца "Compatibility/Legacy Mode" на рис. 2(2)

Fig. 4. Values of the "Compatibility/Legacy Mode" column in Fig. 2(2)

Обозначение	Описание
V	Команда поддерживается (Valid) в данном режиме
I	Команда не поддерживается (Invalid) в данном режиме
NE или N.E.	Указывает на то, что команда не поддерживается (Not encodable) в 64-битном режиме, машинный

	код команды не допускается в режиме совместимости или в режиме IA-32. Машинный код может представлять допустимую команду семейства IA-32 legacy команд.

7. Нотация вычислительных опций процессора

Согласно п. "3.1.1.6 CPUID Support Column in the Instruction Summary Table"[6] значения в столбце "CPUID Feature Flag" (рис. 15) передают поддержку процессором соответствующей технологии. Значение в столбце эквивалентно названию флага, который устанавливается в данных, выдаваемых процессором при исполнении команды O='0F A2'; I='CPUID'. Иначе говоря, если процессор поддерживает некоторую технологию вычислений, то соответствующий флаг/бит установлен.

Значения этого столбца, например, 'ADX' на рис. 15, могут быть отражены на множество значений, обозначающих технологии обработки данных, реализованных в процессорах (табл. 1). Соответственно, задавая фильтрацию команд по технологиям, отмеченным в столбце "CPUID Feature Flag", в дизассемблере можно реализовать проверку кода анализируемой программы на поддержку соответствующей технологии обработки данных.

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Рис. 15. Пример значений в столбце "CPUID Feature Flag"

Fig. 15. Example of values in the "CPUID Feature Flag" column

8. Некоторые типовые ошибки, выявляемые в документации Intel и AMD

Детальный анализ показал, что ни одну из версий документации любого производителя невозможно рассматривать как эталон. И у Intel, и у AMD в описаниях команд выявляются синтаксические и семантические ошибки. Некоторые из них удается выявлять и исправлять при перекрестном сопоставлении описаний команд обоих производителей.

Далее представлены некоторые типовые ошибки, которые приходится исправлять в тексте документации при составлении входной спецификации для построения дизассемблера.

8.1 Ошибочное кодирование операнда в документации Intel

В описании команды O='VEX.128.66.0F.WIG 12 /r'; I='VMOVLPD xmm2, xmm1, m64 кодировки операндов описаны неверно: одновременно указано кодирование через ModRM:r/m и для Operand 1, и для Operand 3 в "Instruction Operand Encoding".

MOVLPD—Move Low Packed Double Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double precision floating-point value from m64 and the high quadword of xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
B	N/A	ModRM:r/m (r)	VEX.vvvv (r)	ModRM:r/m (r)	N/A

8.2 Отсутствие способа кодирования операнда в документации Intel

В описании команды O='EVEX.LLIG.66.0F3A.W0 0A /r ib'; I='VRNDSCALESS xmm1 {k1}{z xmm2, xmm3/m32{sae}, imm8' кодировка операнда Operand 4 в таблице "Instruction Operand Encoding" не указана вообще, тогда как в синтаксисе команды указан операнд imm8. Это также касается команд VSHUFF64, VSHUFF64X2, VSHUFF32, VSHUFF32X4.

VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LLIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	N/A

8.3 Неверная нотация машинного кода команд в документации AMD

Для некоторых команд в документации AMD машинный код указан неверно. Например, для команды I='VBLENDVPD xmm1, xmm2, xmm3/mem128, xmm4' у AMD представлен код в нотации O='VEX.128.66.0F3A.WIG 4B /r', тогда как по документации Intel верный код: O='VEX.128.66.0F3A.W0 4B /r **/is4**'. Отсутствует операнд '/is4'.

8.4 Неверная нотация мнемоники команд в документации AMD

Для некоторых команд в документации AMD указана неверно мнемоника. Например, для команды I='VPINSRB xmm, reg/mem8, xmm, imm8'; O=' VEX.128.66.0F3A.WIG 20 /r ib мнемоника указана неверно. У Intel эта команда представлена в правильной нотации I='VPINSRB xmm1, xmm2, r32/m8, imm8'. Операнд 'r32/m8' должен быть третьим по порядку операндом.

В команде I='**VMOVMSKB** reg64, xmm1'; O='VEX.128.66.0F.WIG D7 /r' неверно указана мнемоника кода операции. У Intel представлен правильный вариант I='**VPMOVMSKB** reg64,xmm1'.

Для команды с O='0F 38 CD /r' мнемоника у AMD представлена в виде I='**SHA256MSG1** xmm1, xmm2/m128' тогда как правильный вариант у Intel I='**SHA256MSG2** xmm1, xmm2/m128'.

9. Обобщенное описание команд и его использование для автоматизированной обработки

Технологию автоматизированного порождения кода дизассемблера иллюстрирует рис. 16. Текстовое описание синтаксиса команд создается непосредственно по тексту документации Intel и AMD. Оно подается на вход специальному транслятору, который подвергает это описание анализу и, в случае его корректности, порождает код дизассемблера на некотором императивном языке программирования высокого уровня.

Затем порожденные файлы компилируются соответствующим компилятором в исполняемый модуль программы дизассемблера.

В качестве декларативного языка программирования оказалось удобным использовать JSON- или XML-описание. Описание синтаксиса команд в этом случае выражается в составлении массива описаний данных, выполненных в любом из этих вариантов.

Рис. 16. Иллюстрация автоматизированного порождения кода дизассемблера

Fig. 16. An illustration of automated generation of disassembler code

При создании обобщенного описания синтаксиса команд вручную и/или с применением средств автоматизации обработки текста, поддерживаемых современными текстовыми редакторами, могут быть удалены ошибки, опечатки и прочие дефекты, которыми изобилует текст документации от Intel и от AMD. Этими же средствами может генерироваться текст исходных спецификаций для транслятора (рис. 16).

Совокупность и последовательность шагов для построения обобщенной спецификации может быть следующей:

· экспорт текста из PDF-представления, например, в DOC-файл формата Microsoft Word;

· приведение текста к единому стилевому представлению;

· преобразование таблиц синтаксиса с разделением представления нотаций машинного кода и мнемоники команд;

· проверка синтаксического представления текста и кодирование всех ограничений, применимых к командам в документации;

· сопоставление описания одних и тех же команд в нотациях Intel и AMD;

· генерирование обобщенного описания синтаксиса команд (обобщенной спецификации).

Пример JSON-спецификации команды O='REX+80 /2 ib'; I='ADC r/m8, imm8' представлен в табл. 5. Фрагменты этой спецификации согласуются с соответствующими частями описаний команд в документации Intel.

Табл. 5. Пример фрагмента JSON-файла с описанием команды O='REX + 80 /2 ib'; I='ADC r/m8, imm8'

Tabl. 5. Example of a fragment of a JSON file with a description of the instruction O='REX + 80 /2 ib'; I='ADC r/m8, imm8'

Фрагмент JSON-спецификации	Пояснения
{ "_comment": "This file contains descriptions of INTEL and AMD x86 instructions" "_comment": "It based on: Intel release	Преамбула описания команд, содержащая информацию, в частности, о версиях Intel и AMD документации,

325462-081US September 2023; AMD release 40332-4.07-June 2023"	на которой она построена
"Instructions": [...	Начало массива описания синтаксиса команд
Непосредственно описание команды	
"OrderNum": "12",	Порядковый номер спецификации. Его удобно использовать для выдачи сообщений транслятора о распознанных ошибках
"Opcode": "REX + 80 /2 ib",	Синтаксис машинного кода команды. Например, из "Opcode" (рис. 2)
"Mnemonic": "ADC r/m8, imm8"	Синтаксис кода команды на ассемблере. Например, из "Instruction" (рис. 2)
"Operands": [{ "Encoding": "ModRM:r/m (r, w)", "Decoder": "ModRM" }, { "Encoding": "imm8/16/32", "Decoder": "Imm" }, { "Encoding": "NA", "Decoder": "None" }, { "Encoding": "NA", "Decoder": "None" }, { "Encoding": "NA", "Decoder": "None" }],	Описание операндов, включая их кодировку в документации рис. 2(2) и типы декодеров операндов (табл. 2)
"WidthMode": "Valid",	Поддерживаемые режимы работы процессора. Например, из столбца вида "64-bit Mode" (рис. 5 и п. 6) или вида "64/32-bit Mode Support" (рис. 8 и п. 6)
"Cpu": "",	Значение из столбца "CPUID Feature Flag" (рис. 6 и п. 7)
"CompatLegMode": "N.E.",	Значение из столбца "Compat/Leg Mode" (рис. 5 и п. 6)
"Description": "Add with carry imm8 to r/m8.",	Значение из столбца "Description" (рис. 2)
"Notes": "In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.",	Примечания, указывающие для некоторых команд на ограничения их реализации (рис. 5) Эта строка в виде комментария

	позволит корректно вручную закодировать процедуру постобработки в дизассемблере.
"Features": "",	Некоторые особенности команды, например, ее применимость только для 32-разрядного режима адресации. Например, для команды O='A5'; I='MOVSD' по сравнению с O='A5'; I='MOVSW' для 16-разрядного режима адресации.
"OEM": "INTEL",	Указание на источник описания команды
}, ...] }	Эпилоги описаний команды, массива команд и всего файла

В авторском макете транслятора JSON-файл спецификаций команд разбирается и генерируется таблица автомата, ячейки которого построены следующим образом (рис. 18).

Для понимания структуры, показанной на рис. 18, можно рассмотреть обобщенную структуру команды, которая в общем виде представляется как последовательность, включающая (рис. 17,а): факультативную последовательность { Prefix-i }, где iÎ[0..3]; последовательность { Opcode-j }, где jÎ[0..2]; последовательность прочих байтов, кодирующих операнды. Такая унифицированная структура получается в результате обобщения структур команд в EVEX-, VEX/XOP-, REX- и Legacy-вариантах. Если еще более упростить структуру, упраздняя разделения на байты, то получим вариант, представленный на рис. 17,б.

Prefix и Opcode в совокупности определяют уникальную комбинацию байтов, позволяющую однозначно распознать данную команду, что полностью согласуется с определением автоматной грамматики, в которой каждое правило вывода имеет вид: α®T β, где α – нетерминал; T – некоторый терминал; β – последовательность нетерминалов и терминалов. Фактически Prefix и Opcode в совокупности позволяют вычислить значение T, которое в конечном счете позволяет: 1) адресовать конкретную ячейку автомата дизассемблера (рис. 18); 2) выбрать элементы декодирования мнемоники и операндов в ней, и произвести (если необходимо) постобработку.

Значение терминала T в данном случае имеет двоякую роль, выполняя роль индекса (маршрутизатора) по таблице автомата и определяя контекстные условия выбора декодеров в ячейке автомата. В реальном дизассемблере таблица автомата устроена иерархически, на каждом уровне иерархии ячейки в ней адресуются соответствующим байтом Opcode-j, но для общей схемы, предающей идею построения дизассемблера, это не существенно.

Каждый декодер операнда является булевской функцией, распознающей только отдельный вид операнда и вычисляющий значение операнда на основе факультативной части машинного кода, следующей за Opcode. Постобработка применяется только в

отдельных случаях для обработки исключений из общих правил строения машинного кода.

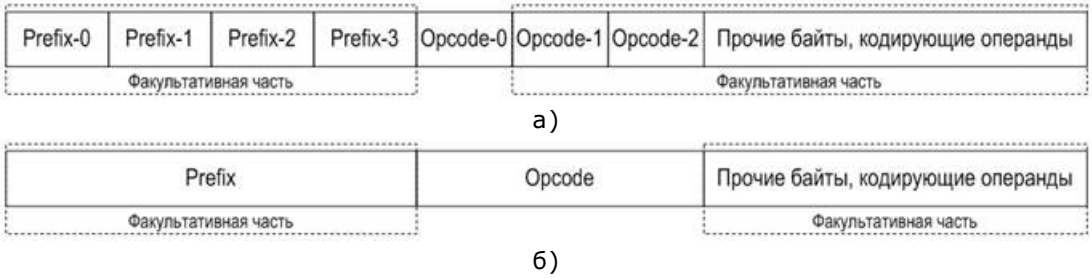


Рис. 17. Обобщенная структура команды: а) с разделением на отдельные байты; б) с разделением только на структурные элементы

Fig. 17. The generalized structure of the instruction: a) divided into individual bytes; b) divided only into structural elements

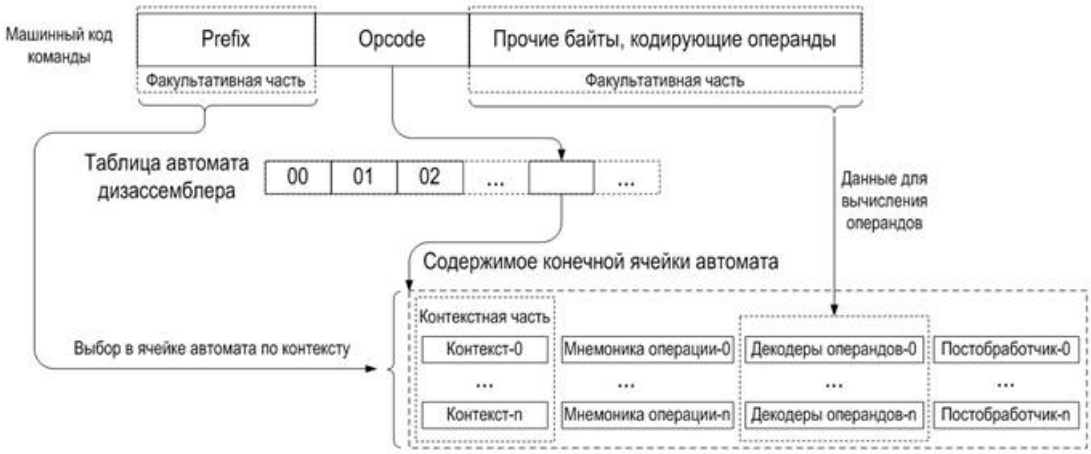


Рис. 18. Обобщенное устройство дизассемблера

Fig. 18. Generalized disassembler construction

В авторском макете дизассемблера таблица автомата (с учетом ее иерархической структуры - таблица переходов автомата), каждая ячейка этой таблицы генерируется автоматически.

Альтернативные проекты дизассемблеров [1,2] реализуются иначе, используя путь объемного (в силу большого числа команд архитектуры x86) и весьма трудоемкого программирования вручную, что делает и обновление кода в этих проектах тоже трудоемким занятием. А значит и поддержка актуальности версий дизассемблера в таких вариантах реализации может происходить не оперативно с обновлениями документации вообще, тем более не синхронно с темпами ее обновления как было представлено выше.

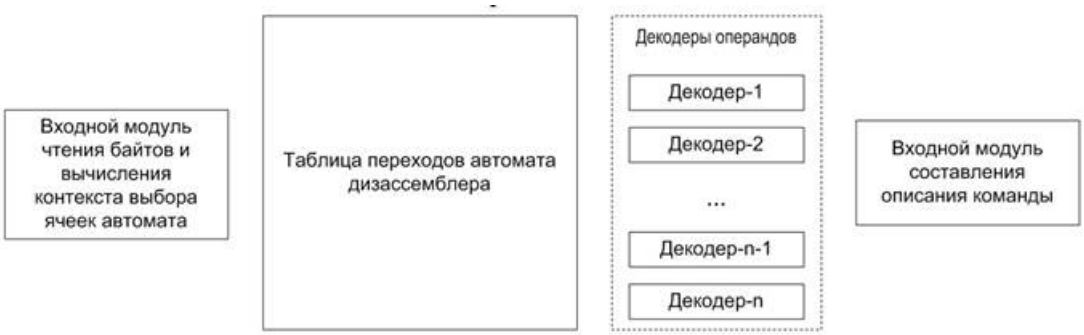


Рис. 19. Модули, составляющие дизассемблер

Fig. 19. Modules that make up the disassembler

На рис. 19 показана структура программных элементов, которые составляют макет дизассемблера, построенный по предлагаемой технологии на языке C. Он включает:

· входной модуль, который написан вручную и остается неизменным при обновлении версии документации (расширения множества команд процессора);

· таблица переходов автомата дизассемблера - наиболее сложный и объемный элемент дизассемблера, который как раз генерируется автоматизировано транслятором (рис. 16) на основе входных спецификаций, получаемых из документации;

· декодеры операндов создаются полуавтоматизированно: каркас создается автоматически, тело пишется вручную. Однажды написанный декодер далее не нуждается в обновлении. Например, для команды O='13 /r'; I='ADC r32, r/m32' однажды написанный декодер для операнда 'r32' будет использоваться для дизассемблирования аналогичных операндов во всех командах с таким же обозначением, вида O='03 /r'; I='ADD r32, r/m32', O='3B /r'; I='CMP r32, r/m32', ..., O='33 /r'; I='XOR r32, r/m32';

· выходной модуль написан вручную и не изменяется.

В идеальной ситуации, когда в новой версии документации не появилось нового формата команд и новых вариантов представления операндов все модули дизассемблера генерируются автоматически. В реальности, например, при реализации дизассемблера для версии документации Intel от сентября 2023 по сравнению с версией для документации от июня 2021 необходимо было немного изменить входной модуль для работы с командами вида O='EVEX.128.NP.MAP5.W0 58 /r'; I='VADDPH xmm1{k1}{z}, xmm2, xmm3/m128/m16bcst' и O='EVEX.128.66.MAP6.W0 13 /r'; I='VCVTPH2PS xmm1{k1}{z}, xmm2/m64/m16bcst', в машинном коде которых появились новые элементы 'MAP5' и 'MAP6'.

10. Заключение

Доля применения процессоров архитектуры x86 на мировом рынке персональных компьютеров хотя и снизилась примерно на тринадцать процентов за шесть лет с 2016 по 2022, но x86 остается все еще самой массово используемой микропроцессорной архитектурой [3]. Основные ее дизайнеры - компании Intel и AMD - непрерывно совершенствуют выпускаемые процессоры, в том числе, и расширяя их программный интерфейс. Эти усовершенствования направлены на повышение производительности при исполнении приложений.

Современная технология разработки программ в целом основывается на использовании языков высокого уровня, что заставляет совершенствовать процедуры генерирования исполняемого кода в компиляторах, в которых задействуются новые команды. Вместе с этим относительная легкость создания приложений на языках высокого уровня дает возможность также легко реализовать в них недокументированные возможности, создающие риск непредсказуемого поведения приложения на некотором массиве входных данных. Вероятность такого поведения приложений ведет к необходимости проверять, по крайней мере, критически важный код программ на корректность выполняемых им вычислений. Сделать это в ситуации недоступности исходного кода можно методами реинжиниринга программ, где одним из действий является дизассемблирование исполняемого кода.

Естественно, что дизассемблеры должны обновляться с учетом обновлений программного интерфейса процессоров. Объем документации на процессоры, в частности, архитектуры x86, таков, что обработать ее текст и собрать описания команд в целостный и единообразный массив данных можно только автоматизированными методами. Исследованию возможности создавать такие спецификации и посвящена данная статья.

Целью же построения таких спецификаций является последующее автоматизированное генерирование кода дизассемблера для поддержания актуальной его версии по отношению к версиям документации.

Библиография

1. Версия исходного кода операционной системы Windows 2000. Доступно по ссылке: https://github.com/pustladi/Windows-2000/ (дата обращения: 23.02.2022)
2. Проект Intel X86 Encoder Decoder (Intel XED). Доступно по ссылке: https://github.com/intelxed/xed (дата обращения: 23.08.2022)
3. Distribution of Intel and AMD x86 computer central processing units (CPUs) worldwide from 2012 to 2022, by quarter. Доступно по ссылке: https/www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/ (дата обращения: 23.09.2022)
4. Китай научился легально клонировать процессоры Intel. Помог старинный соперник. Доступно по ссылке: https://www.cnews.ru/news/top/2023-06-09_kitajtsy_nauchilis_legalno (дата обращения: 14.12.2023)
5. Intel® 64 and IA-32 Architectures. Software Developer's Manual. Documentation Changes. September 2023. Document Number: 252046-073 Доступно по ссылке: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-documentation-changes.html (дата обращения: 21.10.2023)
6. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Order Number: 325462-081US September 2023. Доступно по ссылке: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html (дата обращения: 20.10.2023)
7. Расширенная форма Бэкуса-Наура. Доступно по ссылке: https://ru.wikipedia.org/wiki/Расширенная_форма_Бэкуса-Наура (дата обращения: 23.09.2022)
8. AMD64 Technology. AMD64 Architecture Programmer's Manual. Volumes 1–5. Publication No. 40332 Revision 4.07 Date June 2023. Доступно по ссылке: https:// https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf (дата обращения: 20.11.2023

Результаты процедуры рецензирования статьи

В связи с политикой двойного слепого рецензирования личность рецензента не раскрывается.
Со списком рецензентов издательства можно ознакомиться здесь.

Статья посвящена разработке обобщенной нотации программного интерфейса процессоров x86 для автоматизированного построения дизассемблера. Основная цель исследования – создание системы, которая позволит автоматизировать процесс создания дизассемблеров, что значительно упростит и ускорит работу с программным

кодом различных процессоров семейства x86. Автор использует метод анализа и синтеза существующих подходов к разработке дизассемблеров, основанных на документации процессоров от Intel и AMD. В статье приводится детальный обзор структуры документации, описываются типовые ошибки и неточности, а также предлагаются способы их автоматизированного исправления. Актуальность исследования обусловлена постоянным развитием процессоров архитектуры x86 и увеличением множества команд, поддерживаемых этими процессорами. Существующие подходы к разработке дизассемблеров не всегда могут оперативно адаптироваться к изменениям в программном интерфейсе процессоров, что делает предложенное автором решение крайне востребованным. Новизна работы заключается в разработке обобщенной нотации, которая позволяет автоматизировать процесс создания дизассемблеров. Впервые предложен подход, при котором на основе формальных спецификаций команд процессоров создаются автоматизированные трансляторы, генерирующие исходный код дизассемблера. Статья написана в научном стиле, отличается логической последовательностью и четкостью изложения материала. Структура статьи включает введение, обзор текущих решений, описание методологии, анализ документации процессоров, предложение обобщенной нотации и выводы. Содержание статьи соответствует заявленной тематике, каждая часть логически связана с предыдущей, что делает работу целостной и последовательной. В статье сделан вывод о возможности и необходимости автоматизации процесса создания дизассемблеров для процессоров x86. Автор показывает, что предложенная нотация и методология позволяют существенно сократить время и усилия, необходимые для обновления и разработки дизассемблеров в соответствии с изменениями в документации процессоров. Статья будет интересна широкому кругу специалистов в области разработки программного обеспечения, занимающихся вопросами обратного инжиниринга и анализа исполняемого кода. Работа также представляет интерес для разработчиков компиляторов и специалистов по информационной безопасности, занимающихся анализом программного обеспечения на наличие уязвимостей. Для дальнейшего развития работы автору рекомендуется сосредоточиться на нескольких ключевых направлениях. Во-первых, следует провести более детальное тестирование предложенной методологии на различных версиях процессоров от Intel и AMD, чтобы подтвердить универсальность и надежность созданной обобщенной нотации. Во-вторых, стоит разработать и реализовать инструменты для автоматизированного обновления спецификаций дизассемблера на основе новых версий документации процессоров. Это позволит обеспечить синхронизацию с последними изменениями в программном интерфейсе процессоров и повысит актуальность и оперативность обновлений дизассемблеров. Также необходимо рассмотреть возможность интеграции предложенного подхода с существующими системами разработки и анализа программного обеспечения. Это позволит значительно расширить область применения предложенной методологии и сделает ее доступной для широкого круга пользователей. Важно также уделить внимание созданию подробной документации и обучающих материалов, которые помогут другим разработчикам быстро освоить и эффективно использовать предложенные инструменты и методики. Наконец, перспективным направлением является проведение сравнительного анализа эффективности предложенного подхода с другими современными методами разработки дизассемблеров. Это позволит объективно оценить преимущества и недостатки разработанной методологии, а также выявить возможности для ее дальнейшего совершенствования и адаптации к новым вызовам и требованиям рынка. Рекомендуется принять статью к публикации, так как она представляет собой значительный вклад в область автоматизации разработки инструментов для анализа программного обеспечения, обладает высокой актуальностью и новизной, а также интересна для

широкой научной и профессиональной аудитории.