ЛАБОРАТОРНЫЕ РАБОТЫ 1 и 2
ПРОГРАММИРОВАНИЕ АРИФМЕТИЧЕСКОГО СОПРОЦЕССОРА

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Сопроцессорные конфигурации

Использование сопроцессора позволяет значительно ускорить работу программ, выполняющих расчеты с высокой точностью, тригонометрические вычисления и обработку информации, которая должна быть представлена в виде действительных чисел. Сопроцессор подключается к системной шине параллельно с центральным процессором (CPU) и может работать только совместно с ним. Все команды попадают в оба процессора, а выполняет каждый свои. Сопроцессор не имеет своей программы и не может осуществлять выборку команд и данных. Это делает центральный процессор. Сопроцессор перехватывает с шины данные и после этого реализует конкретные действия по выполнению команды. Два процессора работают параллельно, что повышает эффективность системы. Но возникают ситуации, когда их работа требует синхронизации (из-за разницы во времени выполнения команд).

Синхронизация по командам. Когда центральный процессор выбирает для выполнения команду FPU, последний может быть занят выполнением предыдущей команды. Поэтому перед каждой командой сопроцессора в программе должна стоять специальная команда (wait), которая только проверяет текущее состояние FPU и, если он занят, переводит центральный процессор в состояние ожидания. Соответствующую команду в программу может вводить либо ассемблер, либо компилятор языка без указаний программиста.

Синхронизация по данным. Если выполняемая в FPU команда записывает операнд в память перед последующей командой CPU, которая обращается к этой ячейке памяти, требуется команда проверки состояния FPU. Если данные еще не были записаны, CPU должен переходить в состояние ожидания. Автоматически учесть такие ситуации довольно сложно, поэтому вводить команды, которые проверяют состояние сопроцессора и при необходимости заставляют центральный процессор ожидать, должен программист.

Программная модель сопроцессора

В программную модель любого процессора включаются только те регистры, которые доступны программисту на уровне машинных команд. Основу программной модели FPU образует регистровый стек из восьми 80-битных регистров R0-R7. В них хранятся числа в вещественном формате. В любой момент времени 3-битное поле ST в слове состояния определяет регистр, являющийся текущей вершиной стека и обозначаемый ST(0). При

занесении в стек (push) осуществляется декремент поля ST и загружаются данные в новую вершину стека. При извлечении из стека (pop) в получатель, которым чаще всего является память, передается содержимое вершины стека, а затем инкрементируется поле ST .

В организации регистрового стека FPU есть отличия от классического стека.

1. Стек имеет кольцевую структуру. Контроль за использованием стека должен осуществлять программист. Максимальное число занесений без промежуточных извлечений равно 8.

2. В командах FPU допускается явное или неявное обращение к регистрам с модификацией или без поля ST. Например, команда fsqrt замещает число из вершины стека значением корня из него. В бинарных операциях допускается явное указание регистров. Адресация осуществляется относительно текущей вершины стека и обозначение ST (i) $0<i<7$, считая от вершины.

3. Не все стековые команды автоматически модифицируют указатель вершины стека.

С каждым регистром стека ассоциируется 2-битный тег (признак), совокупность которых образует слово тегов. Тег регистра R0 находиться в младших битах, R7 – в старших. Тег фиксирует наличие в регистре действительного числа (код 00), истинного нуля (код 01), ненормализованного или бесконечности (код 10) и отсутствие данных (код 11). Наличие тегов позволяет FPU быстрее обнаруживать особые случаи (попытка загрузить в непустой регистр, попытка извлечь из пустого) и обрабатывать данные.

Остальными регистрами FPU являются регистр управления, регистр состояния, два регистра состояния команды и два регистра указателя данных. Длина их всех 16 бит.

Форматы численных данных

Арифметический FPU К1810ВМ87 оперирует с семью форматами численных данных, образующих три класса: двоичные целые, упакованные десятичные целые и вещественные числа. Во всех форматах старший (левый) бит отведен для знака.

Форматы различаются длиной, следовательно, диапазоном допустимых чисел, способом представления (упакованный и неупакованный формат), способом кодировки(прямой и дополнительный код).

Двоичные целые числа. Три формата целых двоичных (целое слово (16 бит), короткое целое (32 бита), длинное целое (64 бита)) отличаются длиной, следовательно, диапазоном чисел. Только в этих форматах применяется стандартный дополнительный код. 0 имеет единственное кодирование. Наибольшее положительное число кодируеться как 011…1, а Наибольшее по модулю отрицательное как 100..0.

Упакованные десятичные целые. Числа представлены в прямом коде и упакованном формате, т.е. в байте содержится две десятичные цифры в коде 8421. Старший бит левого байта – знак, остальные игнорируются, но при записи в них помещаются нули. Но надо учитывать, что при наличии в тетраде запрещающих комбинаций 1010 – 1111 результат операции не определен. Т.е. сопроцессор не контролирует правильность результата.

Вещественные числа. Различают короткие вещественные (КВ) (мантиса – 24 бита, порядок – 8 бит), длинные вещественные (ДВ) (мантиса – 53 бита, порядок – 11 бит) и временные вещественные (ВВ) (мантиса – 64 бита, порядок – 15 бит). Для них применяется формат с плавающей точкой. Значащие цифры находятся в поле мантисы, порядок показывает фактическое положение двоичной точки в разрядах мантисы, бит знака S определяет знак числа. Порядок дается в смещенной форме :

E = истинный порядок + смещение

Смещение для соответствующих форматов равно 127, 1023, 16383 это упрощает операцию сравнения. Операция с целыми числами быстрее операции над плавающей точкой. Это важно в алгоритмах.

Значение числа равно

$$(1)^S \times 2^{\text{E-смещение}} \times F_0F_1F_2\ldots F_n,$$

где n для разных форматов равно 23,52 или 63.

Порядок имеет фиксированную длину, определяя один диапазон представимых чисел. Мантиса – правильная дробь. В коротком и длинном вещественном формате бит Fo при передаче чисел и хранении их в памяти не фигурирует. Это скрытый бит, который в нормализованных числах содержит 1. Скрытый бит не дает представить в этих форматах нуль и он должен кодироваться как спец значение.

Числа во временном вещественном формате имеют явный бит Fo. Формат повышает скорость выполнения операций благодаря простоте представления чисел, не являющихся ненормализованными. При загрузке из памяти в регистр FPU оно преобразуется во временный вещественный формат. А при записи в память – обратный формат. Временной вещественный формат – единственный, в котором абсолютно точно кодируется любые загружаемые из памяти числа.

Режимы работы. Состояние

Сопроцессор имеет 2 доступных 16-битных регистра, содержимое которых определяет его режим работы и текущее состояние. Форматы регистров содержат слово управления CW и слово состояния SW. Регистр управления содержит 6 бит масок особых случаев. Регистр состояния – 6 бит флажков.

Регистр управления:

15	14	13	12	11 10	9 8	7	6	5	4	3	2	1	0
X	X	X	IC	RC	PC	IEM	X	PM	UM	OM	ZM	DM	IM

Регистр состояния:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	ST			C2	C1	C0	IR	X	PE	UE	OE	ZE	DE	IE

Рис 1. Форматы слова управления и слова состояния

Регистр управления содержит 6 бит масок особых случаев, а регистр состояния 6 бит флажков особых случаев:

P – потеря точности

U – антипереполнение

O – переполнение

Z - деление на нуль

D - денормализованный операнд

I - недействительный операция

Слово управления. Оно определяет для FPU один из нескольких вариантов обработки численных данных. Программа центрального процессора может сформировать в памяти образ слова управления, а затем заставить сопроцессор загрузить его в регистр CW. Рассмотрим значение полей.

Шесть младших бит слова управления - индивидуальные маски особых случаев. Т.е. особых ситуаций, обнаруженных FPU при выполнении команд. Если бит=1, то не будет вызвано прерывание CPU. Иначе FPU устанавливает в 1 бит запроса прерывания в слове состояния и при общем разрешении прерываний генерирует сигнал int прерывания CPU.

Бит 7 слова управления содержит маску управлени прерыванием IEM, которая разрешает (IEM=0) или запрещает(IEM=1) прерывание центрального процессора.

Двухбитное поле управления точностью (PC) определяет точность вычислений в 24 бита(PC=00), 53 бита (PC=10) или 64 бита (PC=11). По умолчанию вводиться режим с максимальной точностью в 64 бита.

Двухбитное поле управления округлением RC определяет один из четырех возможных вариантов округления результатов операций сопроцессора.

Бит 12 управляет режимом бесконечности IC. Когда IC=0, сопрцессор обрабатывает два специальных значения “плюс бесконечность” и “минус бесконечность” как одно и то же значение “бесконечность”, не имеющее знака.

Слово состояния. В нем младшие 6 бит отведены для регистрации особых случаев. Бит 7 – запроса прерывания (IR), устанавливается в 1 при возникновении любого незамаскированного особого случая. Бит C3-C0 фиксирует код условия в операциях сравнения, проверки условия и анализа. Три бита ST указатели стека. Стековые операции сопровождаются модификацией поля ST. Наконец, флажок занятости B устанавливается в состояние 1 когда численное операционные устройство выполняет операцию. Большую роль играют биты кода условия, которые фиксируют особенности результата (табл. 1.). Коды условия привлекаются для реализации условных

переходов. Сопроцессор самостоятельно не может влиять на ход выполнения программ. Поэтому для условных переходов по результатам операций сопроцессора приходится сначала передавать код условия в память, а затем загружать один из регистров центрального процессора. После этого код условия передается в регистр флагов, производится условный переход.

Таблица 1. Интерпетация кода условия в операциях сравнения и проверки (FCOM,FCOMP,FTST)

C3	C2	C1	C0	Описание
0	X	X	0	(ST)>источника (src)
0	X	X	1	(ST)Бисточника (src)
1	X	X	0	(ST)=источнику (src)
1	X	X	1	Не сравнимы

Система команд

Система команд сопроцессора содержит 6 групп команд: команды передач данных, арифметические команды, команды сравнения, трансцендентных операций, команды загрузки констант и управления сопроцессором. Операнды некоторых команд определяются неявно. Другие команды допускают или требуют явного задания операнда.

При рассмотрении ситемы команд сопроцессора будем пользоваться следующими обозначениями: src обозначает источник, т.е. операнд, значение которого не модифицируеться, а dst – получатель, т.е. операнд, значение которого замещается результатом операции.

Особенности задания команд. Команды бинарных операций допускают несколько форм задания**.** При пустом поле операнда операция выполняется с двумя верхними элементами стека ST(источник) и ST(1)(получатель). После производства операции осуществляеться инкрементирование указателя стека и результат помещаеться в новую вершину стека, заменяя исходное содержимое ST(1). Когда в бинарной команде определен один операнд, операция выполняется с привлечением указанного в команде регистра или ячейки памяти и содержимого вершины стека. Результат загружается в старую вершину стека и указатель стека не модифицируется. Если в бинарной команде указаны 2 операнда, ими является содержимое 2-х регистров стека, причём одним из них будет ST, а 2-м ST(i).

Альтернативные формы операндов условно показываются с помощью наклонной черты, причем черта без последующей спецификации означает отсутствие явно задаваемых операндов. Например команда FADD имеет следующий общий вид:

FADD //src/dst,src

Такая запись подразумевает три возможных формы команд: без операндов, с одним источником, с получателем и источником.

В мнемониках команд сопроцессора приняты следующие соглашения: первая буква всегда F; вторая буква I обозначает операцию с целыми числами, буква B – операцию с десятичным целым операндом, а пустая – операцию с вещественными числами; предпоследняя или последняя буква R

указывает обратную операцию (например, в обычной форме команды деления получатель делится на источник, а в обратной форме источник делится на получатель; в обоих формах результат помещается в получатель); последняя буква P идентифицирует команду, заключительным действием которой является извлечение из стека.

Команды передачи данных. Команды этой группы производят передачу данных между регистрами стека, а так же между вершинами стека и памятью. Одной командой число из памяти преобразуется во ременный вещественный формат и загружается в стек. Таким же образом, но в обратном порядке осуществляется передача числа в память.

Команды загрузки. 3 команды загрузки имеют следующий вид:

вещественное: FLD src

двоичное целое FILD src

десятичное целое FBLD src

Эти команды осуществляют декремент указания стека и передачу в новую вершину стека содержимого источника. В команде FLD источником может быть один из регистров стека или вещественное число. В командах FILD и FBLD – только операнд в памяти.

Команды запоминания.

Вещественное: FST dst

Двоичное целое: FIST dst

Они производят передачу содержимого вершины стека в память без модификации указателя ST и содержимого ST(0). В команде FST получатель регистр стека или вещественная переменная в памяти. В команде FIST получателем является переменная в памяти имеющая формат коротко целого и целого слова. Рассмотренные команды не допускают получателем формат длинного целого, временного вещественного и неупакованного десятичного.

Команды запоминания с извлечением из стека. Три команды, помимо передачи содержимого ST(0) осуществляют извлечение из стека. Регистры бывшей вершины стека отмечается как пустой и производится инкремент указателя стека:

Вещественное: FSTP dst

Двоичное целое: FISTP dst

Десятичное целое: FBSTP dst

Действия команды FSTP очень похожи на действия FST с добавлением извлечения из стека. Однако, FSTP может передать в память слово во временном вещественном формате, чего не может сделать FST. Команда FIST обеспечивает передачу в память числа в любом формате целого двоичного, включая длинное целое. Последний формат не допустим в формате FSTP. Команда FBST преобразует операнд из вершины стека в упакованное десятичное число, передает его в память и производит извлечение из стека.

Команда обмена содержимого регистров.

FXCH//dst ST(0)<->(dst) обменивает содержимое получателя ST(i) с вершиной ST(0). При пустом поле операнда обменивается содержимое

регистров ST(1) и ST(0).

Команды управления.

Команда FINIT / FNINIT - инициализировать сопроцессор.

Команда FLDCW src – загрузить слово управления. Источником является целое число в памяти.

Команды FSTCW dst и FSTSW dst – запомнить слово управления и состояния в ячейке памяти, определяемой получателем dst.

Команда FSTENV dst – запомнить среду. Под средой сопроцессора К1810ВМ87 понимаеся содержимое регистров управления, состояния, тегов, указателя команды и указателя операнда. Команда FSTENV передаёт его в область памяти с начальным адресом, указаным в команде. Формат хранения среды в памяти показан на рис. 2.

Начальный адрес →	15 … 0	Смещение
	Слово управления	+0
	Слово состояния	+2
	Слово тегов	+4
	Указатель команды	+6 +8
	Указатель операнда	+10 +12

Рис. 2. Формат хранения среды сопроцессора в памяти.

До выборки из очереди следующей команды сопроцессора выполнение команды FSTENV должно закончиться.

Команда FLDENV src – загрузить среду. Парная предыдущей команде команда FLDENV src осуществляет загрузку среды сопроцессора из области памяти, определяемой src. После команды FLDENV не требуется команда FWAIT, так как сопроцессор автоматически контролирует завершение передачи всех слов среды до перехода к своей следующей команде.

Команда FSAVE dst - сохранить полное состяние сопроцессора. Полное состояние сопроцессора представляет собой содержимое всех регистров програмной модели – среды и восьми регистров стека. Размер полного состояния сопроцессора составляет 94 байта. Команда FSAVE передаёт его в область памяти с начальным адресом, указаным в команде. Формат размещения полного состояния сопроцессора в памяти (его иногда называют «образом в памяти») показан на рис.3.

Начальный адрес →	15 … 0	Смещение
	Среда сопроцессора	+0
	Верхний элемент стека st(0)	+14
	Следующий элемент стека st(1)	+24
	...	...
	Последний элемент стека st(7)	+12

Рис. 3. Формат хранения полного состояния сопроцессора в памяти.

Команда FRSTOR src – восстановить полное состояние сопроцессора.

Для сохранения и восстановления состояния сопроцессора обычно применяется следующий ассемблерный фрагмент:

```
SUB     SP,94       ;Зарезервировать пространство в стеке
MOV     BP,SP       ;BP является базой для состояния
FSAVE   [BP]        ;Сохранить полное состояниеъ
  …
MOV     BP,SP       ;BP является базой для состояния
FRSTOR  [BP]        ;Восстановить состояние
ADD     SP,94       ;Освободить пространство в стеке
```

Арифметические команды.

Необходимо отметить, что вещественные числа в памяти, не могут быть в формате временного вещественного, а целые числа – в формате длинного целого. Здесь сказывается недостаточность наборов кодов операций.

Команды сложения. Операция сложения реализуется командами со следующими формами:

вещественные числа FADD //src/dst,src

вещественные числа с извлечением из стека FADDP dst,src

целые числа FIADD src

Отметим, что команда FADD ST,ST(0) удваивает содержимое вершины стека.

Команды вычитания. Обычное вычитание $dst \leftarrow (dst) - (src)$ осуществляют команды:

вещественные числа FSUB //src/dst,scr

вещественные числа с извлечением из стека FSUBP dst,src

целые числа FISUB src

Для производства обратного вычитания $dst \leftarrow (src) - (dst)$ предназначены команды FSUBR, FSUBRP, FISUBR, имеющие аналогичные формы.

Команды умножение. Операция умножение реализуется следующими командами:

вещественные числа FMUL //src/dst,scr

вещественные числа с извлечением из стека FMULP dst,src

целые числа FIMUL src

Команды деления. Для выполнения обычной операции деления предусмотрены команды:

вещественные числа FDIV //src/dst,scr

вещественные числа с извлечением из стека FDIVP dst,src

целые числа FIDIV src

Соответствующие команды обратного деления FDIVR, FDIVRP, FIDIVR загружают в получатель частное от деление источника на получатель.

Приведём несколько примеров арифметических команд:

```
FADD ST, ST(5)                  ; Сложить содержимое регистров
FIADD   WORD PTR COUNT [SI]     ; Прибавить целое слово
FSUBP   ST(2), ST               ; Вычесть содержимое регистров
```

FDIVR DWORD PTR [SI] ;Разделить короткое вещественное

FIDIVR DWORD PTR [BX+5] ; Разделить короткое целое

Команда извлечения квадратного корня. Команда FSQRT извлечения квадратного корня заменяет число, находящееся в вершине стека, значением квадратного корня:

FSQRT ST(0) <- $ST(0)^{1/2}$.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Могут ли процессор и сопроцессор работать параллельно?
2. В чем суть синхронизации по данным?
3. В чем суть синхронизации по командам?
4. Какие регистры образуют программную модель сопроцессора?
5. С какими форматами численных данных может оперировать сопроцессор?
6. Что подразумевается под средой сопроцессора?
7. Что подразумевается под состоянием сопроцессора?

ЛАБОРАТОРНЫЕ ЗАДАНИЯ

Лабораторная работа 7.

Написать программу, находящую решение квадратного уравнения

$ax^2 + bx + c = 0$

с помощью сопроцессора.

Лабораторная работа 8.

Значение аргумента **x** изменяется от ***a*** до ***b*** с шагом ***h***. Для каждого ***x*** найти значения функции Y(x), суммы S(x) и число итераций n, при котором достигается требуемая точность ε = |Y(x)-S(x)|. Результат вывести в виде таблицы. Значения ***a, b, h*** и ***ε*** вводятся с клавиатуры.

1. $S(x) = \sum_{k=1}^{n} \frac{\cos(kx)}{k}$, $Y(x) = -\ln\left|2\sin\frac{x}{2}\right|$.

2. $S(x) = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$, $Y(x) = \sin(x)$.

3. $S(x) = \sum_{k=0}^{n} \frac{\cos(\frac{k\pi}{4})}{k!} x^k$, $Y(x) = e^{x\cos\frac{\pi}{4}} \cos(x\sin\frac{\pi}{4})$.

4. $S(x)=\sum_{k=0}^{n}(-1)^k\frac{x^{2k}}{(2k)!}$, $\quad Y(x)=\cos(x)$.

5. $S(x)=\sum_{k=1}^{n}x^k\sin(\frac{\pi k}{4})$, $\quad Y(x)=\frac{x\sin(\pi/4)}{1-2x\cos\frac{\pi}{4}+x^2}$.

6. $S(x)=\sum_{k=0}^{n}\frac{x^{4k+1}}{4k+1}$, $\quad Y(x)=\frac{1}{4}\ln\frac{1+x}{1-x}+\frac{1}{2}arctgx$.

7. $S(x)=\sum_{k=0}^{n}\frac{\cos(kx)}{k!}$, $\quad Y(x)=e^{\cos x}\cos(\sin(x))$.

8. $S(x)=\sum_{k=0}^{n}\frac{2k+1}{k!}x^{2k}$, $\quad Y(x)=(1+2x^2)e^{x^2}$.

9. $S(x)=\sum_{k=1}^{n}\frac{x^k\cos\frac{k\pi}{3}}{k}$, $\quad Y(x)=-\frac{1}{2}\ln(1-2x\cos\frac{\pi}{3}+x^2)$.

10. $S(x)=\sum_{k=0}^{n}\frac{1}{2k+1}\left(\frac{X-1}{X+1}\right)^{2K+1}$, $\quad Y(x)=\frac{1}{2}\ln(x)$.

11. $S(x)=\sum_{k=1}^{n}(-1)^k\frac{\cos(kx)}{k^2}$, $\quad Y(x)=\frac{1}{4}\left(x^2-\pi^2/3\right)$.

12. $S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k+1}}{4k^2-1}$, $\quad Y(x)=\frac{1+x^2}{2}arctg(x)-x/2$.

13. $S(x)=\sum_{k=0}^{n}\frac{x^{2k}}{(2k)!}$, $\quad Y(x)=\frac{e^x+e^{-x}}{2}$.

14. $S(x)=\sum_{k=1}^{n}\frac{\cos(2kx)}{4k^2-1}$, $\quad Y(x)=\frac{1}{2}-\frac{\pi}{4}\left|\sin(x)\right|$.

15. $S(x)=\sum_{k=0}^{n}\frac{k^2+1}{k!}(x/2)^k$, $\quad Y(x)=(\frac{x^2}{4}+\frac{x}{2}+1)e^{\frac{x}{2}}$.

16.

$$S(x)=\sum_{k=0}^{n}(-1)^k\frac{2k^2+1}{(2k)!}x^{2k},\qquad Y(x)=(1-\frac{x^2}{2})\cos(x)-\frac{x}{2}\sin(x).$$

17. $$S(x)=\sum_{k=1}^{n}(-1)^k\frac{(2x)^{2k}}{(2k)!},\qquad Y(x)=2(\cos^2 x-1).$$

18. $$S(x)=\sum_{k=1}^{n}x^k\cos(\frac{k\pi}{4}),\qquad Y(x)=\frac{x\cos\frac{\pi}{4}-x^2}{1-2x\cos\frac{\pi}{4}+x^2}\quad .$$

19. $$S(x)=\sum_{k=1}^{n}\frac{\cos[(2k-1)x]}{(2k-1)^2};\qquad Y(x)=\frac{\pi^2}{8}-\frac{\pi}{4}|x|.$$

20.

$$S(x)=\sum_{k=1}^{n}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)},\qquad Y(x)=x\,arctg(x)-\ln\sqrt{1+x^2}\,.$$

ЛАБОРАТОРНАЯ РАБОТА 9.1.
ТЕХНОЛОГИЯ MMX

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Назначение MMX

Технология Intel MMX представляет собой набор расширений к архитектуре Intel, которые были разработаны для того, чтобы увеличить производительность средств мультимедиа и коммуникаций.

Расширение MMX предназначено для ускорения выполнения приложений типа «подвижное видео», комбинированной графики с видеообработкой изображений, звуковым синтезом, синтезом и сжатием речи, телефонией, видео, конференц-связью, и 2D и 3D графикой, которые обычно используют алгоритмы с интенсивными вычислениями, чтобы выполнять повторяющиеся действия на больших множествах простых элементов данных.

Технология MMX определяет простую и гибкую модель программного обеспечения без нового режима или видимого состояния для операционной системы.

Следующие разделы этой работы описывают основную окружающую среду программирования технологии MMX, включая набор регистров MMX, типы данных и набор инструкций.

Модель SIMD

MMX-технология использует методику «одиночная команда, множественные данные» (Single Instruction Multiple Data – SIMD) для выполнения арифметических и логических операций над байтами, словами или двойными словами, упакованными в 64-разрядные регистры MMX. Например, команда paddsb складывает 8 знаковых байт источника с 8 знаковыми байтами в приемнике и сохраняет их в операнде-приемнике. Эта технология ускоряет эффективность выполнения программ, позволяя одну и ту же операцию выполнять параллельно на множестве элементов данных.

Модель выполнения SIMD, обеспечиваемая в MMX-технологии, удовлетворяет потребностям современных средств связи и графических приложений, которые часто используют сложные алгоритмы, в которых выполняются одни и те же операции над большим количеством данных. Например, большие звуковые данные представляются в 16-разрядных словах. Команды MMX могут обрабатывать сразу 4 из этих слов одновременно в одной команде. Видео- и графическая информация обычно представляются как пакетированные байты. Одна MMX-команда может оперировать над 8 из этих байтов одновременно.

Программная модель расширения MMX

Состав программной модели

Технология MMX обеспечивает следующие новые расширения к окружающей среде программирования архитектуры IA-32:

- восемь 64-разрядных MMX-регистров MM0-MM7;
- четыре типа данных MMX (упакованные байты, слова, двойные слова и учетверенное слово);
- систему команд MMX.

Регистры MMX

Набор регистров MMX состоит из восьми 64-разрядных регистров MM0-MM7. Эти регистры могут использоваться только для выполнения вычислений над MMX типами данных и не могут использоваться для адресации памяти. Адресация операндов MMX-команды в памяти осуществляется, используя стандартные способы адресации и регистры общего назначения. Хотя регистры MMX определены в архитектуре IA-32 как отдельные регистры, они являются псевдонимами младших 64-разрядных частей регистров R0..R7 стека математического сопроцессора как изображено на рисунке 1.

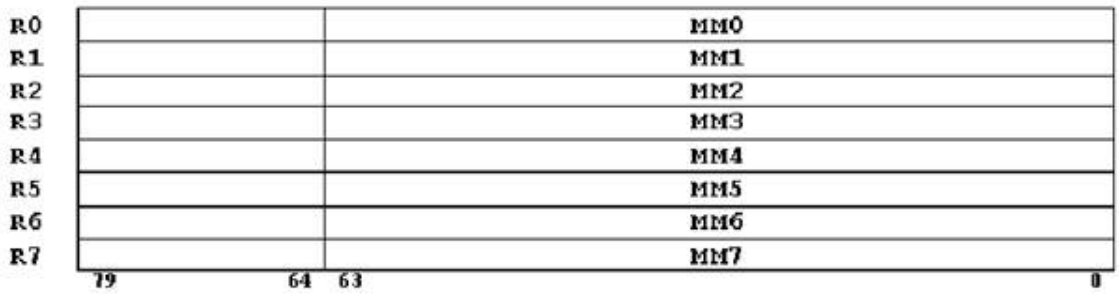

Рисунок 1 – Регистры MMX и регистры сопроцессора

Когда регистры сопроцессора играют роль MMX-регистров, то доступными являются лишь их младшие 64 бита. К тому же, при работе стека сопроцессора в режиме MMX-расширения, он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению и как MMX-расширение. Забота о его разделении и корректной работе с ним полностью ложится на программиста.

Типы данных MMX

Технология MMX определяет следующие новые 64-разрядные типы данных (рисунок 2):

- упакованные байты – восемь байт, упакованные в одно 64-разрядное поле;
- упакованные слова – четыре слова, упакованные в одно 64-разрядное поле;
- упакованные двойные слова – два двойных слова, упакованные в одно 64-разрядное поле;
- учетверенное слово – одно 64-разрядное поле.

Рисунок 2 – Типы данных MMX

При выполнении арифметических или логических операций над упакованными типами данных MMX-команды оперируют параллельно над индивидуальными байтами, словами, или двойными словами, содержащихся в 64-разрядном MMX-регистре. При операциях над байтами, словами и двойными словами внутри упакованных типов данных, MMX-команды оперируют как со знаковыми, так и беззнаковыми целыми байтами, словами, двойными словами.

Для определения 64-разрядных данных в программах на ассемблере используется директива dq. Однако существует множество других способов определения MMX-данных.

Например:

```
.data
m64_1       dq          0011223344556677h
m64_2       label       qword
_2_dword        dd      00112233h, 44556677h
m64_3       label       qword
_4_word     dw          0011h, 2233h, 4455h, 6677h
m64_4       label       qword
_8_byte     db          00h, 11h, 22h, 33h, 44h, 55h, 66h, 77h
union  mmxdata
  m64       dq          0
  m32       dd          2 dup(0)
  m16       dw                4 dup(0)
  m8        db          8 dup(0)
ends
m64_5           mmxdata   <-1, 2, 3, 4, -29, 100, 84, 53>
```

Система команд MMX-расширения

Арифметика с насыщением и арифметика цикличности

Технология MMX поддерживает новую арифметическую возможность, известную как *арифметика с насыщением* (Saturated Arithmetics). Арифметику с насыщением лучше всего определить, противопоставляя ее *арифметике цикличности* (Wraparound Arithmetic). В арифметике цикличности результаты, которые *переполняются* или *антипереполняются*, усекаются и возвращаются только самые младшие биты результата (только те

которые входят в разрядную сетку соответствующего типа), т.е. перенос игнорируется. В режиме насыщения результаты операции, которые переполняются или антипереполняются, приводятся к соответствующим значениям границ диапазона для данного типа данных (таблица 1). Результат операции, который превышает верхнюю границу диапазона типа данных, насыщается к максимальному значению диапазона, а результат, который оказывается меньше нижней границы, – к минимальному значению диапазона. Этот метод обработки переполнения и антипереполнения применяется во многих приложениях.

Например, когда результат превышает диапазон данных для знаковых байтов, он обрезается до 7fh для знаковых байтов и 0ffh для байтов без знака. Если значение меньше диапазона, оно обрезается до 80h для знаковых байтов и 00h для байтов без знака.

Таблица 3.1

Тип данных	Нижний предел		Верхний предел	
	шестнадцат.	десятичн.	шестнадцат.	десатичн.
Знаковый байт	80h	-128	7fh	127
Знаковое слово	8000h	-32768	7fffh	32767
Беззнаковый байт	00h	0	0ffh	255
Беззнаковое слово	0000h	0	0ffffh	65535

Система команд

Система команд MMX состоит из 57 команд, сгруппированных в следующие категории:

- команды пересылки данных;
- арифметические команды;
- команды сравнения;
- логические команды;
- команды сдвига;
- команды упаковки и распаковки;
- дополнительные команды;
- команда инициализации.

При оперировании над упакованными данными внутри регистра MMX данные приводятся в соответствии с типом, определенным командой. Например, команда paddb (сложить упакованные байты) обрабатывает упакованные данные как 8 упакованных байтов, в то время как команда paddw (сложить упакованные слова) обрабатывает упакованные данные как 4 упакованных слова.

Операнды команд

Все команды MMX, за исключением команды emms, оперируют двумя операндами: первый операнд – приемник, второй операнд – источник. Результат команды записывает в операнд-приемник.

Операнд-источник для всех MMX-команд (за исключением команд передачи данных) может быть в памяти или в одном из регистров MM0-

MM7. Операнд-приемник всегда должен находиться в регистре MMX. Для команд передачи данных операндом-источником и операндом-приемником может также быть целочисленный регистр (для команды movd) или память (для команд movd и movq).

Команды пересылки данных

MMX-команды пересылки данных работают с 32- и 64-разрядными операндами. В данную группу входят следующие команды:

- movd dst, src – пересылает 32-разрядные данные из памяти в регистры MMX и обратно или из целочисленных регистров процессора в регистры MMX и обратно.
- movq dst,src – пересылает 64-разрядные упакованные данные из памяти в регистры MMX и обратно или между регистрами MMX.

Арифметические команды

Упакованное сложение и вычитание выполняют следующие команды:

- paddsb, paddsw, paddwd – выполняют сложение знаковых или беззнаковых упакованных байтов, слов, двойных слов.
- psubb, psubw и psubd – выполняют вычитание знаковых или беззнаковых упакованных байтов, слов, двойных слов.

Команды paddsb и paddsw (упакованное сложение с насыщенностью) и psubsb и psubsw (упакованное вычитание с насыщенностью) выполняют сложение или вычитание знаковых элементов данных операнда источника и операнда адресата и приводят результат к граничным значениям диапазона знакового типа данных. Эти команды поддерживают упакованные байты и упакованные слова.

Команды paddusb и paddusw (упакованное сложение без знака с насыщенностью) и psubusb и psubusw (упакованное вычитание без знака с насыщенностью) выполняют сложение или вычитание элементов данных без знака операнда источника и операнда адресата и приводят результат к граничным значениям диапазона типа данных без знака. Эти команды поддерживают упакованные байты и упакованные слова.

Упакованное умножение. Команды упакованного умножения выполняют четыре умножения на парах 16-разрядных знаковых операндов, производя 32-разрядные промежуточные результаты. Пользователи могут выбирать старшие или младшие части каждого 32-разрядного результата.

Команды pmulhw и pmullw умножают знаковые слова операндов источника и адресата и записывают старшую или младшую часть результата в операнд адресата.

Упакованное умножение/сложение. Команда pmaddwd вычисляет произведение знаковых слов операндов адресата и источника. Четыре промежуточных 32-разрядных произведения суммируются в парах, чтобы произвести два 32-разрядных результата.

Команды сравнения

Команды pcmpeqb, pcmpeqw, pcmpeqd (упакованное сравнение на равенство) и pcmpgtb, pcmpgtw, pcmpgtd (упакованное сравнение на «больше») сравнивают соответствующие элементы данных в операндах источника и адресата на равенство или оценивают, кто из них больше. Эти команды генерируют маску единиц или нулей, которые записываются в

операнд адресата. Логические операции могут использовать маску, чтобы выбрать элементы. Это можно использовать, чтобы выполнить упакованную условную операцию пересылки без ветвления или набора команд ветвления. Никакие флаги не устанавливаются. Эти команды поддерживают упакованные байты, упакованные слова и упакованные двойные слова.

Команды преобразования

Команды преобразования преобразовывают элементы данных внутри упакованного типа данных.

Команды packsswb и packssdw (упакованный со знаковой насыщенностью) преобразовывают знаковые слова в знаковые байты или знаковые двойные слова в знаковые слова в режиме знаковой насыщенности.

Команда packuswb (упакованный насыщенностью без знака) преобразовывает знаковые слова в байты без знака в режиме насыщенности без знака.

Команды punpckhbw, punpckhwd и punpckhdq (распаковать старшие упакованные данные) и punpcklbw, punpcklwd и punpckldq (распаковать младшие упакованные данные) преобразовывают байты в слова, слова в двойные слова или двойные слова в четверное слово.

Логические команды

Команды pand (поразрядное логическое И), pandn (поразрядное логическое И-НЕ), por (поразрядное логическое ИЛИ) и pxor (поразрядное логическое исключающее ИЛИ) выполняют поразрядные логические операции над 64-разрядных данными.

Команды сдвига

Команды логического сдвига влево, логического сдвига вправо и арифметического сдвига право сдвигают каждый элемент на определенное число битов. Логические левые и правые сдвиги также дают возможность перемещать 64-разрядное поле как один блок, что помогает в преобразованиях типа данных и операциях выравнивания.

Команды psllw, pslld (упакованный логический сдвиг влево) и psrlw, psrld (упакованный логический сдвиг вправо) выполняют логический левый или правый сдвиг и заполняют пустые старшие или младшие битовые позиции нулями. Эти команды поддерживают упакованные слова, упакованные двойные слова и четверное слово.

Команды psraw и psrad (упакованный арифметический сдвиг вправо) выполняют арифметический сдвиг вправо, копируя знаковый разряд в пустые разрядные позиции на старшем конце операнда. Эти команды поддерживают упакованные слова и упакованные двойные слова.

Команда EMMS

Команда emms освобождает состояние MMX. Эта команда должна использоваться, чтобы очистить состояние MMX (чтобы освободить tag-слово регистров FPU) в конце MMX подпрограммы перед вызовом других подпрограмм, которые могут выполнять операции с плавающей точкой.

Наличие этой команды обязательно, если MMX-команды комбинируются с командами сопроцессора.

Использование MMX-команд в программах

Когда необходимо использовать MMX-команды в программе на ассемблере в начале программы указывается директива .mmx. Например:

```
.686
.mmx
.model   flat
.data
m64_1  dq       0a0d0b0c12ef093ch
m64_2  dq       1f05db0c93eee9a0h
m64_3  dq       0
.code
          movq          mm0, m64_1
          paddd                mm0, m64_2
          movq          m64_3, mm0
end
```

Однако, как правило, различные алгоритмы, которые используют MMX-команды, значительно меньше по объему кода, чем окружающая их среда программы, основное назначение которой организовать интерфейс для доступа к данным и взаимодействия ее с пользователем. Поэтому целесообразно использовать либо совместное программирование на языке высокого уровня (например, на языке C++) и ассемблере, либо использовать механизм ассемблерных вставок.

Рассмотрим, как можно использовать MMX в программах на C++, создаваемых в среде программирования Visual Studio 6.0/2003/2005.

Типы данных __m64 и __int64 предназначены для определения в программе 64-разрядных данных, которые могут затем быть использованы в MMX-командах. Представим предыдущий фрагмент кода в виде функции на языке C++.

```
void mmx_func()
{
        __int64 m64_1 = 0x0a0d0b0c12ef093ch;
        __int64 m64_2 = 0x1f05db0c93eee9a0h;
        __int64 m64_3 = 0;

 __asm
 {
          movq    mm0, m64_1
          paddd  mm0, m64_2
          movq    m64_3, mm0
 }
}
```

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Для выполнения данной работы необходимо выполнить следующее:

а) изучить программную модель MMX;

б) изучить систему команд MMX;

в) обработать массивы из 8 элементов по следующему выражению (в зависимости от варианта):

1) F[i]=(A[i]+B[i])*C[i]+D[i] , i=1...8;

2) F[i]=(A[i]+B[i])*C[i]-D[i] , i=1...8;
3) F[i]=A[i]-B[i]*C[i]+D[i] , i=1...8;
4) F[i]=A[i]-B[i]*C[i]-D[i] , i=1...8;
5) F[i]=A[i] *C[i]+B[i] *D[i] , i=1...8;
6) F[i]=A[i] *B[i] +C[i] -D[i] , i=1...8.
7) F[i]=A[i] -B[i] +C[i] *D[i] , i=1...8.
8) F[i]=A[i] -B[i] +C[i] -D[i] , i=1...8.
9) F[i]=A[i] *B[i] +C[i] -D[i] , i=1...8.
10) F[i]=(A[i]+B[i])*(C[i]+D[i]) , i=1...8;
11) F[i]=(A[i]+B[i])*(C[i]-D[i]), i=1...8;
12) F[i]=(A[i]-B[i])*(C[i]+D[i]), i=1...8;
13) F[i]=(A[i]-B[i])*(C[i]-D[i]) , i=1...8;
14) F[i]=(A[i] *C[i])+(B[i] *D[i]) , i=1...8;
15) F[i]=(A[i] *B[i]) +(C[i] -D[i]) , i=1...8.
16) F[i]=(A[i] -B[i]) +(C[i] *D[i]) , i=1...8.
17) F[i]=(A[i] -B[i]) +(C[i] -D[i]), i=1...8.
18) F[i]=(A[i] *B[i])+(C[i] -D[i]) , i=1...8.
19) F[i]=A[i]+(B[i])*C[i])+D[i] , i=1...8;
20) F[i]=A[i]+(B[i])*C[i])-D[i] , i=1...8;
21) F[i]=A[i]-(B[i]*C[i])+D[i] , i=1...8;
22) F[i]=A[i]-(B[i]*C[i])-D[i] , i=1...8;
23) F[i]=A[i] *(C[i])+B[i]) *D[i] , i=1...8;
24) F[i]=A[i] *(B[i]) +C[i])-D[i] , i=1...8.
25) F[i]=A[i] –(B[i]) +C[i]) *D[i] , i=1...8.
26) F[i]=A[i] –(B[i]) +C[i]) -D[i] , i=1...8.
27) F[i]=A[i] *(B[i])+C[i]) -D[i] , i=1...8.

Используются следующие массивы:

A, **B** и **C** – 8 разрядные целые знаковые числа (_int8);

D – 16 разрядные целые знаковые числа (_int16).

Полученный результат отобразить на форме с использованием соответствующих элементов. При распаковке знаковых чисел совместно с командами распаковки использовать команды сравнения (сравнивать с нулём перед распаковкой).

ЛАБОРАТОРНАЯ РАБОТА 9.2.

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ РАСШИРЕНИЙ SSE/SSE2

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Особенности расширений SSE/SSE2

Расширение SSE

SSE (англ. Streaming SIMD Extensions, потоковое SIMD-расширение процессора) — это набор SIMD инструкций, разработанный Intel , и впервые представленный в процессорах серии Pentium III.

Технология SSE позволяет преодолеть основную проблему MMX — при использовании MMX невозможно одновременно использовать инструкции сопроцессора, так как его регистры используются и для MMX и для работы FPU.

Расширение позволяет выполнять векторные (пакетные) и скалярные инструкции. Векторные инструкции реализуют операции сразу над четырьмя комплектами операндов. Скалярные инструкции работают только с одним комплектом операндов – младшим 32-битным словом.

SSE включает в архитектуру процессора восемь 128-битных регистров xmm0…xmm7, каждый из которых трактуется как 4 последовательных значения с плавающей точкой одинарной точности. Расширение позволяет выполнять векторные (пакетные) и скалярные инструкции. *Векторные инструкции* реализуют операции сразу над четырьмя комплектами операндов. *Скалярные инструкции* работают только с одним комплектом операндов – младшим 32-битным словом.

Реализация блоков SIMD осуществляется распараллеливанием вычислительного процесса между данными. То есть когда через один блок проходит поочередно множество потоков данных.

Расширение SSE2

SSE2 (англ. Streaming SIMD Extensions 2, потоковое SIMD-расширение процессора) – это SIMD (англ. Single Instruction, Multiple Data, Одна инструкция – множество данных) набор инструкций, разработанный Intel , и впервые представленный в процессорах серии Pentium 4.

SSE2 использует те же восемь 128-битных регистров xmm0…xmm7 что и расширение SSE, каждый из которых трактуется как 2 последовательных значения с плавающей точкой двойной точности. SSE2 включает в себя набор инструкций, которые производят операции со скалярными и упакованными типами данных. Также SSE2 содержит инструкции для потоковой обработки целочисленных данных в тех же 128-битных xmm регистрах, что делает это расширение более предпочтительным для целочисленных вычислений, нежели использование набора инструкций MMX.

Программная модель SSE/SSE2

Регистры SSE/SSE2

Все три расширения работают с одним набором 128-битных регистров, обозначаемых XMM0…XMM7, как показано на рисунке 1.

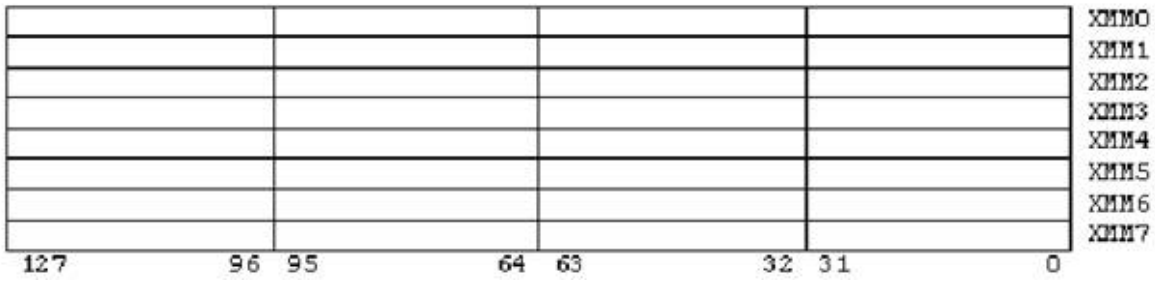

Рисунок 1 – Регистры SSE/SSE2

Типы данных SSE/SSE2

Новые расширения микропроцессора дополняют уже имеющиеся типы данных новыми упакованными типами:

- 4 упакованных вещественных числа одинарной точности;
- 2 упакованных вещественных числа двойной точности;
- 16 упакованных целых байтов;
- 8 упакованных целых слов;
- 4 упакованных целых двойных слова;
- 2 упакованных целых учетверенных слова.

Система команд SSE/SSE2

Команды SSE

Команды SSE делятся на 4 категории:

- SIMD-команды для данных одинарной точности с плавающей запятой (SPFP-команды);
- дополнительные SIMD-команды для целочисленных данных;
- команды управления кэшированием;
- команды сохранения и восстановления компонент состояния процессора.

Одна SIMD-команда с плавающей запятой может обрабатывать одновременно четыре 32-разрядных числа одинарной точности с плавающей запятой (называемых SPFP-элементами данных).

Каждое 32-разрядное число с плавающей запятой имеет 1 знаковый бит, 8 битов порядка и 23 бита мантиссы, что соответствует стандарту IEEE-754 на формат представления чисел одинарной точности с плавающей запятой.

SIMD-команды поддерживают два типа операций над упакованными данными с плавающей запятой - параллельные и скалярные.

Параллельные операции, как правило, действуют одновременно на все четыре 32-разрядных элемента данных в каждом из 128-разрядных

операндов. В именах команд, выполняющих параллельные операции, присутствует суффикс PS.

Скалярные операции действуют на младшие (занимающие разряды 0-31) элементы данных двух операндов. Остальные три элемента данных в выходном операнде не изменяются (исключение составляет команда скалярного копирования MOVSS). В имени команд, выполняющих скалярные операции, присутствует суффикс SS.

SSE-команды имеют следующий синтаксис:

instruction [dest, src]

Здесь instruction – имя команды, dest обозначает выходной операнд, src – входной операнд.

Доступны следующие SSE-команды (обозначения: xmm – XMM-регистр; mm – MMX-регистр; m – память; imm – 8-битный непосредственный операнд; ir32 – целочисленный регистр):

- ***addps xmm, xmm/m..*** Команда попарно складывает упакованные элементы данных и записывает суммы в соответствующие элементы XMM-регистра.
- ***subps xmm, xmm/m.*** Команда вычитает элементы входного операнда из элементов первого регистра и записывает полученные разности в соответствующие элементы первого регистра.
- ***mulps xmm, xmm/m.*** Команда попарно перемножает упакованные элементы. Произведения записываются в соответствующие элементы XMM-регистра.
- ***divps xmm, xmm/m.*** Команда делит элементы первого операнда на соответствующие элементы входного операнда. Результаты деления записываются в XMM-регистр на место делимых.
- ***addss xmm, xmm/m.*** Команда складывает младшие элементы данных и записывает результат в младший элемент XMM-регистра. Остальные элементы выходного операнда не меняются.
- ***subss xmm, xmm/m.*** Команда вычитает младший элемент входного операнда из соответствующего элемента выходного операнда и записывает разность в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***mulss xmm, xmm/m.*** Команда перемножает младшие элементы данных и записывает результат в младший элемент XMM-регистра. Остальные элементы выходного операнда не меняются.
- ***divss xmm, xmm/m.*** Команда делит младший элемент выходного операнда на соответствующий элемент входного операнда и записывает результат в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***sqrtps xmm, xmm/m.*** Команда вычисляет квадратный корень для каждого из четырех чисел во входном операнде и записывает результаты в выходной операнд.

- ***sqrtss xmm, xmm/m*** . Команда вычисляет квадратный корень из младшего элемента входного операнда и записывает результат в младший элемент в выходной операнд. Остальные элементы выходного операнда не меняются.
- ***rcpps xmm, xmm/m.*** Команда определяет приближенное обратное значение для каждого из четырех чисел входного операнда и записывает результаты в XMM-регистр.
- ***rsqrtps xmm, xmm/m.*** Команда вычисляет приближенное обратное значение для квадратного корня из каждого из четырех чисел входного операнда и записывает результаты в XMM-регистр.
- ***rcpss xmm, xmm/m.*** Команда определяет приближенное обратное значение для числа, находящегося в младшем элементе входного операнда и записывает результат в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***rsqrtss xmm, xmm/m.*** Команда вычисляет приближенное обратное значение для квадратного корня из числа , находящегося в младшем элементе входного операнда и записываетрезультат в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***maxps xmm, xmm/m.*** Команда попарно сравнивает элементы данных и записывает большее значение из каждой пары в соответствующий элемент выходного операнда.
- ***minps xmm, xmm/m.*** Команда попарно сравнивает элементы данных и записывает меньшие значения из каждой пары в соответствующие элементы выходного операнда.
- ***maxss xmm, xmm/m.*** Команда сравнивает младшие элементы данных и записывает большее из значений в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***minss xmm, xmm/m.*** команда сравнивает младшие элементы данных и записывает меньшее из значений в младший элемент выходного операнда. Остальные элементы выходного операнда не меняются.
- ***shufps xmm, xmm/m, imm.*** Команда с непосредственным операндом выбирает из первого операнда два элемента с 2-битными номерами из непосредственного операнда (непосредственный операнд делится справа налево на 4 2-битовых числа) и записывает их в младшие элементы выходного операнда. Из второго операнда выбираются два элемента со следующими 2-битными номерами и записываются в старшие элементы выходного операнда. Перестановка всех элементов происходит одновременно.
- ***unpckhps xmm, xmm/m.*** Команда копирует третьи элементы входного и выходного операндов в соседние младшие элементы выходного операнда, а четвертые элементы входного и выходного операндов - соответственно, в старшие элементы выходного операнда. Распаковка элементов выполняется одновременно.

- ***unpcklps xmm, xmm/m.*** Команда копирует первые элементы входного и выходного операндов в соседние младшие элементы выходного операнда, а вторые элементы входного и выходного операндов - соответственно, в старшие элементы этого операнда. Распаковка элементов выполняется одновременно.
- ***movaps xmm/m, xmm/m.*** Команда копирует четыре FP-элемента одним из следующих способов:
  - из памяти в XMM-регистр;
  - из XMM-регистра в память;
  - из одного XMM-регистра в другой.

Линейный адрес соответствует адресу младшего байта группы данных в памяти. Обращение в память должно быть по адресу, кратному 16 байтам; в противном случае генерируется исключение.

- ***movups xmm/m, xmm/m.*** Команда копирует четыре FP-элемента. Линейный адрес соответствует адресу младшего байта группы данных в памяти. Команда применяется для обращения к невыровненным данным в памяти.
- ***movhps xmm/m, xmm/m.*** Команда копирует два FP-элемента одним из следующих способов:
  - из памяти в два старшие элемента XMM-регистра
  - из двух старших элементов XMM-регистра в память

При копировании данных из памяти в XMM-регистр, два младших элемента этого регистра не изменяются. Команда работает с адресом младшего байта группы данных в памяти.

- ***movlps xmm/m, xmm/m.*** Команда копирует два FP-элемента одним из следующих способов:
  - из памяти в два младшие элемента XMM-регистра
  - из двух младших элементов XMM-регистра в память

При копировании данных из памяти в XMM-регистр, два старших элемента этого регистра не изменяются. Линейный адрес соответствует адресу младшего байта группы данных в памяти.

- ***movss xmm/m, xmm/m.*** Команда копирует один 32-рарядный FP-элемент одним из следующих способов:
  - из памяти в младший элемент XMM-регистра
  - из младшего элемента XMM-регистра в память
  - из младшего элемента одного XMM-регистра в младший элемент другого XMM-регистра

При копировании 32 битов данных из памяти в XMM-регистр, все 96 оставшихся старших битов этого регистра обнуляются. Команда работает с адресом младшего байта группы данных в памяти.

- ***movlhps xmm, xmm.*** Команда копирует 64 младших разряда входного регистра в 64 старших разряда выходного регистра, не меняя содержимое 64 младших разрядов выходного регистра.

- ***movhlps xmm, xmm.*** Команда копирует 64 старших разряда входного регистра в 64 младших разряда выходного регистра, не меняя содержимое 64 старших разрядов выходного регистра.
- ***movmskps ir32, xmm.*** Команда копирует содержимое старших (знаковых) разрядов каждого из четырех FP-элементов, находящихся во входном XMM-регистре, в младшие разряды выходного целочисленного регистра, и формирует таким образом 4-битовую маску. Старшие 28 разрядов целочисленного регистра обнуляются.
- ***cmpeqps xmm, xmm/m; cmpltps xmm, xmm/m; cmpleps xmm, xmm/m; cmpunordps xmm, xmm/m; cmpneqps xmm, xmm/m; cmpnltps xmm, xmm/m; cmpnleps xmm, xmm/m; cmpordps xmm, xmm/m.*** Команды попарно сравнивают числа в соответствующих элементах операндов, проверяя выполнение условий равно, меньше, меньше или равно, несравнимы, не равно, не меньше, не { меньше или равно}, сравнимы соответственно. Элементы выходного операнда заполняются масками из единиц или нулей в зависимости от результата.
- ***cmpeqss xmm, xmm/m; cmpltss xmm, xmm/m; cmpless xmm, xmm/m; cmpunordss xmm, xmm/m; cmpneqss xmm, xmm/m; cmpnltss xmm, xmm/m; cmpnless xmm, xmm/m; cmpordss xmm, xmm/m.*** Команды сравнивают числа в младших элементах операндов, проверяя выполнение условий равно, меньше, меньше или равно, несравнимы, не равно, не меньше, не (меньше или равно), сравнимы соответственно. Младший элемент выходного операнда заполняется маской из единиц или нулей в зависимости от результата. Содержимое трех старших элементов выходного операнда сохраняется.
- ***andps xmm, xmm/m.*** Команда вычисляет поразрядное логическое И своих 128-битных входного и выходного операндов. Каждый бит результата полагается равным 1, если оба соответствующих бита операндов равны 1, и равным 0 в противном случае. Результат записывается в выходной операнд.
- ***andnps xmm, xmm/m.*** Команда сначала инвертирует все разряды выходного операнда (логическое НЕ), а затем вычисляет поразрядное логическое И входного и инвертированного выходного операндов. Каждый бит результата полагается равным 1, если для входного операнда соответствующий бит равен 1, а для выходного - равен 0. В противном случае присваивается 0. Результат записывается в выходной операнд.
- ***orps xmm, xmm/m.*** Команда вычисляет поразрядное логическое ИЛИ своих 128-битных входного и выходного операндов. Каждый бит результата полагается равным 0, если оба соответствующих бита операндов равны 0, и равным 1 в противном случае. Результат записывается в выходной операнд.
- ***xorps xmm, xmm/m***. Команда вычисляет поразрядное логическое исключающее ИЛИ своих 128-битных входного и выходного операндов. Каждый бит результата полагается равным 1, если соответствующие биты

операндов содержат различные значения. В случае одинаковых значений присваивается 0. Результат записывается в выходной операнд.

- ***comiss xmm, xmm/m.*** Команда выполняет сравнение двух младших элементов операндов и устанавливает для них одно из соотношений: "меньше", "равны", "больше", "несравнимы". По результатам сравнения устанавливаются следующие значения флагов состояния ZF, PF и CF.

Команды SSE2

При описании операндов инструкций использованы следующие обозначения:

- mmx – любой из восьми 64-х разрядных регистров MMX.
- xmm – любой из восьми 128-ми разрядных регистров.
- r32 – любой 32-х разрядный регистр общего назначения: EAX, EBX и так далее.
- m128, m64, m32, m8 – элемент памяти соответствующего размера в битах.
- imm8 – непосредственный способ адресации, число имеющее размер байта, например, константа сдвига.

Если в качестве операнда указано только имя регистра или только элемент памяти, то это означает, что операнд может находиться только в регистре или только в ОЗУ. Если же указано сочетание обозначений имени регистра и элемента памяти, разделенное наклонной скобкой, например, xmm/m128 то операнд может находиться либо в регистре, либо в ОЗУ.

Групповые арифметические операции, нахождение максимума и минимума, извлечение квадратного корня. Операнды (dest и src) содержат по два вещественных числа двойной точности (64-bit), соответственно в dest получаются два 64-bit результата.

- ***addpd xmm, xmm/m128.*** – сложение двух пар вещественных чисел удвоенной точности
- ***subpd xmm, xmm/m128***. – вычитание двух пар вещественных чисел удвоенной точности
- ***mulpd xmm, xmm/m128***. – умножение двух пар вещественных чисел удвоенной точности
- ***divpd xmm, xmm/m128.*** – деление двух пар вещественных чисел удвоенной точности
- ***sqrtpd xmm, xmm/m128***. – извлечение квадратного корня из двух вещественных чисел источника с записью результата в приемник
- ***maxpd xmm, xmm/m128.*** – нахождение в каждой паре большего вещественного числа удвоенной точности
- ***minpd xmm, xmm/m128***. – нахождение в каждой паре меньшего вещественного числа удвоенной точности

Скалярные арифметические операции, нахождение максимума и минимума, извлечение квадратного корня. Вещественное число двойной точности раположено в младших половинах регистров xmm. Результат

получается в младшей половине dest, содержимое его старшей половины не изменяется.

- ***addsd xmm, xmm/m64***. – сложение двух вещественных чисел удвоенной точности
- ***subsd xmm, xmm/m64.*** – вычитание двух вещественных чисел удвоенной точности
- ***mulsd xmm, xmm/m64***. – умножение двух вещественных чисел удвоенной точности
- ***divsd xmm, xmm/m64***. –деление двух вещественных чисел удвоенной точности
- ***sqrtsd xmm, xmm/m64.*** – извлечение квадратного корня из вещественного числа источника с записью результата в приемник
- ***maxsd xmm, xmm/m64.*** – нахождение большего из двух вещественных чисел удвоенной точности
- ***minsd xmm, xmm/m64.*** – нахождение меньшего из двух вещественных чисел удвоенной точности

Групповые операции преобразования форматов чисел. Возможны два типа преобразований:

- вещественные числа преобразуются в вещественные с изменением точности представления – из двойной точности в обычную или наоборот.
- целые числа преобразуются в вещественные или вещественные числа в целые. В последнем случае возможно округление или отсечение остатка. Два целых числа могут находиться в регистрах xmm или mmx.

- ***cvtpd2ps xmm, xmm/m128***. – два вещественных числа преобразуются из формата с двойной точностью в формат с обычной точностью. dest(64-127)=0
- ***cvtps2pd xmm, xmm/m64.*** – два вещественных числа преобразуются из формата с обычной точностью в формат с двойной точностью. Если источником является xmm, то преобразуются два младших числа.
- ***cvtps2dq xmm, xmm/m128***. – 4 вещественных числа представленные с обычной точностью преобразуются в 4 целых 32-х разрядных числа.
- ***cvttps2dq xmm, xmm/m128.*** – 4 вещественных числа представленные с обычной точностью преобразуются в 4 целых 32-х разрядных числа с отсечением остатка.
- ***cvtdq2ps xmm, xmm/m128.*** – 4 целых 32-х разрядных числа преобразуются в 4 вещественных числа, представленных с обычной точностью.
- ***cvtpd2dq xmm, xmm/m128.*** – два вещественных числа двойной точности преобразуются в два целых 32-х разрядных числа. dest(64-127)=0
- ***cvttpd2dq xmm, xmm/m128.*** – два вещественных числа двойной точности преобразуются в два целых 32-х разрядных числа с отсечением остатка. dest(64-127)=0

- ***cvtdq2pd xmm, xmm/m64.*** – два целых 32-х разрядных числа преобразуются в два вещественных числа, представленных с двойной точностью. Если источником является xmm, то преобразуются два младших числа.
- ***cvtpd2pi mmx, xmm/m128***. – два вещественных числа двойной точности преобразуются в два целых 32-х разрядных числа и записываются в регистр mmx
- ***cvttpd2pi mmx, xmm/m128.*** – два вещественных числа двойной точности преобразуются в два целых 32-х разрядных числа с отсечением остатка и записываются в регистр mmx
- ***cvtpi2pd xmmx, mmx/m64***. – два целых 32-х разрядных числа, находящихся в регистре mmx, преобразуются в два вещественных числа, представленных с двойной точностью.

Преобразования формата одного числа. Возможны два типа преобразований:

- изменяется точность представления вещественного числа - двойная на обычную или наоборот.
- целое число преобразуется в вещественное или вещественное число в целое. В последнем случае возможно округление или отсечение остатка.

- ***cvtsd2ss xmm, xmm/m64.*** – вещественное число преобразуется из формата с двойной точностью в формат с обычной точностью. У dest не изменяется содержимое разрядов 32-127.
- ***cvtss2sd xmm, xmm/m32.*** – вещественное число преобразуется из формата с обычной точностью в формат с двойной точностью. Содержимое старшей половины dest не изменяется.
- ***cvtsd2si r32, xmm/m64***. – вещественное число двойной точности преобразуются в целое 32-х разрядное число.
- ***cvttsd2si r32, xmm/m64***. – вещественное число двойной точности преобразуется в целое 32-х разрядное число.
- ***cvtsi2sd xmm, r32/m32.*** – целое 32-х разрядное число преобразуется в вещественное число, представленное с двойной точностью. Содержимое старшей половины dest не изменяется.

Сравнения без изменения состояния разрядов EFLAGS. Для каждой сравниваемой пары вещественных чисел (dest[h],src[h] и dest[l],src[l]) при положительном результате сравнения устанавливаются все разряды dest[h] или dest[l] соответственно, в противном случае они очищаются. Например, если содержимое регистра xmm сравнить с самим собой, то при проверке отношения "равно" все 128 разрядов будет установлены, а при проверке отношения "не равно" все 128 разрядов будут очищены.

- ***cmppd xmm, xmm/m128, imm8***. –независимо друг от друга сравниваются две пары вещественных чисел. Результат помещается в соответствующие половины dest. Проверяемое условие задает третий операнд (см. ниже).

- ***cmpsd xmm, xmm/m128, imm8***. – сравниваются два вещественных числа. Результат помещается в младшую половину dest[l], содержимое старшей половины dest[h] не изменяется. Проверяемое условие задает третий операнд.

При сравнении одной пары вещественных чисел можно изменить состояние разрядов EFLAGS. Это делают две инструкции, различающиеся по способу реагирования на случаи, когда один или оба операнда не являются числами (NaN)

- ***comisd xmm, xmm/m64*** – сравниваются два вещественных числа. Если операнд QNaN или SNaN то возникает исключительная ситуации.
- ***ucomisd xmm, xmm/m64*** – сравниваются два вещественных числа. Если операнд SNaN, то возникает исключительная ситуация.

Обмен 64-bit кодами между оперативной памятью (ОЗУ) и регистрами xmm. Если данные "выровнены", то обращение к ОЗУ занимает 1 такт, в противном случае 2 такта. Выравнивание означает, что адрес 64-bit кода должен быть кратен 8.

- ***movsd xmm, xmm/m64***. Пересылка из ОЗУ в младшие 64-разряда xmm
- ***movsd xmm/m64, xmm.*** Пересылка в ОЗУ младших 64-х разрядов xmm

Важно: при копировании содержимого одного регистра xmm в другой, содержимое старшей половины dest не изменяется.

- ***movhpd xmm, m64.*** Пересылка из ОЗУ в старшие 64-разряда xmm
- ***movhpd m64, xmm***. Пересылка в ОЗУ старших 64-х разрядов xmm
- ***movlpd xmm, m64***. Пересылка из ОЗУ в младшие 64-разряда xmm
- ***movlpd m64, xmm***. Пересылка в ОЗУ младших 64-х разрядов xmm

Обмен 128-bit кодами между оперативной памятью (ОЗУ) и регистрами xmm. Для выравнивания 128-bit кодов их надо располагать по адресам кратным 16-ти. Обычно инструкции работают не с ОЗУ, а с кэш. В данном случае предусмотрена специальная инструкция для записи кодов в ОЗУ без кеширования, т.е. при ее выполнении содержимое кэш не изменяется.

- ***movntpd m128, xmm.*** Пересылка 128-ми разрядного кода в ОЗУ без кеширования

Важно: пересылаемый код должен иметь адрес кратный 16-ти, в противном случае возникает аварийная ситуация (нарушение защиты).

- ***movapd xmm, xmm/m128***. Пересылка 128-bit кода из ОЗУ в регистр xmm
- ***movapd xmm/m128, xmm***. Пересылка 128-bit кода из регистра xmm в ОЗУ

Важно: пересылаемый код должен иметь адрес кратный 16-ти, в противном случае возникает аварийная ситуация (нарушение защиты).

- ***movupd xmm, xmm/m128***. Пересылка 128-bit кода из ОЗУ в регистр xmm
- ***movupd xmm/m128, xmm***. Пересылка 128-bit кода из регистра xmm в ОЗУ

Замечание: эта инструкция предназначена для применения в тех случаях, когда данные не выровнены или неизвестно выровнены они или нет.

Распаковка и перегруппировка кодов являются своеобразной разновидностью пересылок.

- ***unpckhpd xmm, xmm/m128.*** – распаковка: в dest копируется содержимое старших половин src и dest.
- ***unpcklpd xmm, xmm/m128***. – распаковка: в dest копируется содержимое младших половин src и dest.
- ***shufpd xmm, xmm/m128, imm8***. – перегруппировка: bit 0 imm8 указывает какая половина dest копируется в его младшие разряды, bit 1 imm8 указывает какая половина src копируется в старшие разряды dest. В обоих случаях 0 соответствует младшей половине, а 1 - старшей. Например, если dest и src находятся в одном регистре, то imm8=1 вызовет перестановку чисел в dest.

Логические операции вычисляют функции булевой алгебры. Обычно они применяются для выделения или обединения отдельных частей кода.

- ***andpd xmm, xmm/m128***. – логическая функция конъюнкция ("И").

Аналоги ANDPS xmm, xmm/m128 и PAND xmm, xmm/m128

- ***andnpd xmm, xmm/m128.*** – логическая функция без определенного названия.

Аналоги ANDNPS xmm, xmm/m128 и PANDN xmm, xmm/m128.

- ***orpd xmm, xmm/m128***. – логическая функция дизюнкция ("ИЛИ").

Аналоги ORPS xmm, xmm/m128 и POR xmm, xmm/m128.

- ***xorpd xmm, xmm/m128***. – логическая функция "исключенное ИЛИ".

Аналоги XORPS xmm, xmm/m128 и PXOR xmm, xmm/m128.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Для выполнения данной работы необходимо выполнить следующее:

а) изучить программную модель MMX;

б) изучить систему команд MMX;

в) обработать массивы из 8 элементов по следующему выражению (в зависимости от варианта):

1. F[i]=(A[i]+B[i])*C[i]+D[i] , i=1...8;
2. F[i]=(A[i]+B[i])*C[i]-D[i] , i=1...8;
3. F[i]=A[i]-B[i]*C[i]+D[i] , i=1...8;
4. F[i]=A[i]-B[i]*C[i]-D[i] , i=1...8;
5. F[i]=A[i] *C[i]+B[i] *D[i] , i=1...8;
6. F[i]=A[i] *B[i] +C[i] -D[i] , i=1...8.
7. F[i]=A[i] -B[i] +C[i] *D[i] , i=1...8.
8. F[i]=A[i] -B[i] +C[i] -D[i] , i=1...8.
9. F[i]=A[i] *B[i] +C[i] -D[i] , i=1...8.
10. F[i]=(A[i]+B[i])*(C[i]+D[i]) , i=1...8;
11. F[i]=(A[i]+B[i])*(C[i]-D[i]), i=1...8;
12. F[i]=(A[i]-B[i])*(C[i]+D[i]), i=1...8;
13. F[i]=(A[i]-B[i])*(C[i]-D[i]) , i=1...8;
14. F[i]=(A[i] *C[i])+(B[i] *D[i]) , i=1...8;
15. F[i]=(A[i] *B[i]) +(C[i] -D[i]) , i=1...8.
16. F[i]=(A[i] -B[i]) +(C[i] *D[i]) , i=1...8.

17. F[i]=(A[i] -B[i]) +(C[i] -D[i]), i=1...8.
18. F[i]=(A[i] *B[i])+(C[i] -D[i]) , i=1...8.
19. F[i]=A[i]+(B[i])*C[i])+D[i] , i=1...8;
20. F[i]=A[i]+(B[i])*C[i])-D[i] , i=1...8;
21. F[i]=A[i]-(B[i]*C[i])+D[i] , i=1...8;
22. F[i]=A[i]-(B[i]*C[i])-D[i] , i=1...8;
23. F[i]=A[i] *(C[i])+B[i]) *D[i] , i=1...8;
24. F[i]=A[i] *(B[i]) +C[i])-D[i] , i=1...8.
25. F[i]=A[i] –(B[i]) +C[i]) *D[i] , i=1...8.
26. F[i]=A[i] –(B[i]) +C[i]) -D[i] , i=1...8.
27. F[i]=A[i] *(B[i])+C[i]) -D[i] , i=1...8.

Используются следующие массивы:

A, **B** и **C** – 8 разрядные целые знаковые числа (_int8);

D – 16 разрядные целые знаковые числа (_int16).

Полученный результат отобразить на форме с использованием соответствующих элементов. При распаковке знаковых чисел совместно с командами распаковки использовать команды сравнения (сравнивать с нулём перед распаковкой).