Нижегородский государственный университет
им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Параллелизм как основа архитектуры ВС

Раздел 9

Векторное процессирование

Кудин А.В., к.т.н.

Содержание

- Ограничение производительности систем класса SISD
- Сферы применения векторной обработки
- Архитектура векторного конвейера
- Технология SIMD в IA32: техника MMX
- Технология SIMD в IA32: расширения SSE, SSE2 и SSE3

Ограничение производительности систем класса SISD

CPU 3 GHz → такт $\frac{1}{3\times10^9}$ сек

с ≈ 300 тыс. км / сек → путь за такт $3\times10^8 \times \frac{1}{3\times10^9}$ м = 10 см

перспективный путь дальнейшего повышения производительности – **распараллеливание операций**, т.е. выполнение процессором большего числа операций за такт

гиперконвейеризация

суперскаляризация

технология SIMD

позволила достигнуть производительности порядка GFLOPS

Классификация Флинна

Flynn M. (1966)

		Data Stream	
		Single	Multiple
Instruction Stream	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Flynn M.J. “Very High-Speed Computing System”, Proceedings IEEE, #54, 1966

Векторное процессирование

Вектор – набор однотипных данных (обычно FLOAT-массив)

Длина вектора – обычно переменная величина
(например, до 64 элементов)

Векторный процессор – поддержка векторных операндов

Позитивные факторы, сказывающиеся на производительности:
- эффективное декодирование
- удобные данные

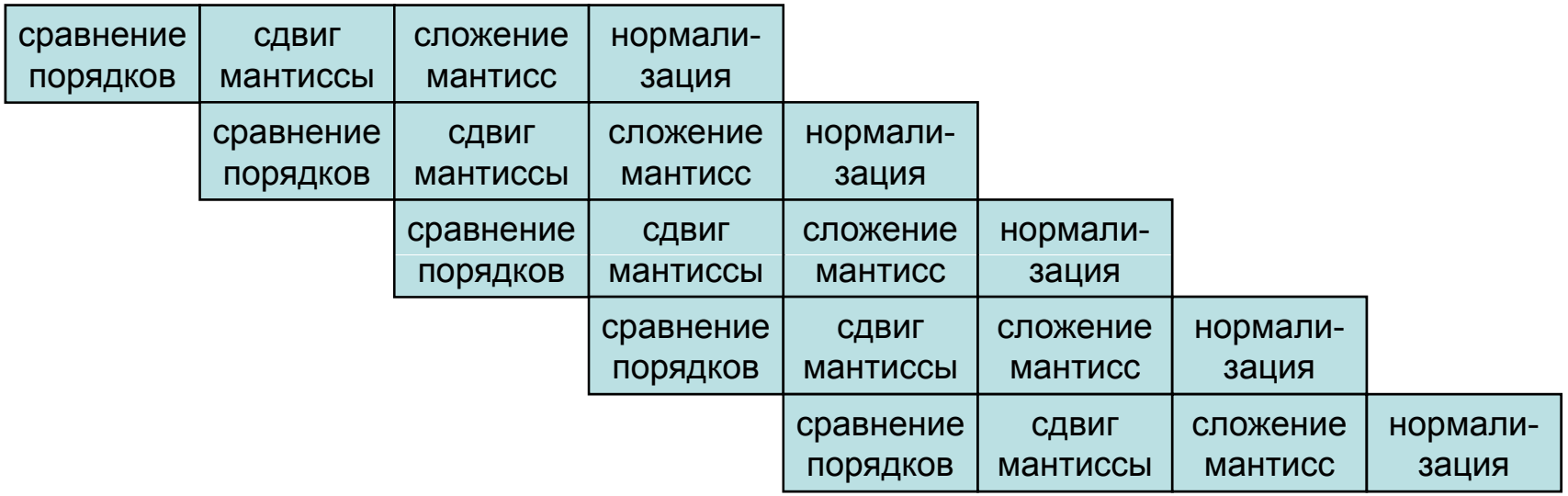
Примеры процессоров:
- Intel P55C (Pentium MMX)
- Cray C90
- NEC SX-3
- PowerPC AltiVec

Сферы применения векторной обработки

- Особенности обработки больших регулярных массивов чисел
- Особенности обработки аудиоинформации
- Особенности обработки видеоинформации

Архитектура векторного конвейера

Реализация в виде конвейерного ALU



Архитектура векторного конвейера

Реализация в виде массива ALU

сравнение порядков	сдвиг мантиссы	сложение мантисс	нормали-зация
сравнение порядков	сдвиг мантиссы	сложение мантисс	нормали-зация
сравнение порядков	сдвиг мантиссы	сложение мантисс	нормали-зация

Векторное процессирование в IA32

Вектор – набор однотипных данных

Сфера применения – моделирование процессов, для которых характерна обработка больших регулярных массивов чисел

MMX (MultiMedia eXtension) – SIMD-технология, появившаяся в процессоре Intel Pentium MMX и вышедшая на массовый рынок в 1997

SSE, SSE2, SSE3 – последующее развитие технологии MMX в более поздних процессорах семейства IA32

Векторное процессирование в IA32

“Alas, SIMD extensions are more an example of good marketing than outstanding achievement of hardware/software co-design”

John L. Hennessy, David A. Patterson

Computer Architecture, 3rd Edition

Технология MMX

Условия применения:

- данные имеют небольшое число разрядов
  (ограничение MMX)
- однотипные операции над многими данными
  (идеология SIMD)
- поток команд не зависит от данных
  (MMX-инструкции способствуют устранению ветвлений)
- необходима высокая производительность
  (максимальное ускорение в 16 раз на P55C)

Технология MMX в дизайне CPU

Определение поддержки MMX: при помощи команды **CPUID**

MMX-регистры реализованы на базе регистров FPU

Любая команда MMX/FPU разрушает состояние FPU/MMX

Сохранение контекста FPU/MMX: команда **FSAVE**

Восстановление контекста FPU/MMX: команда **FRSTOR**

Выход из “режима” MMX с инициализацией FPU: команда **EMMS**

EFLAGS не используется и не модифицируется

Используются идентичные схемы адресации памяти

Технология MMX

Регистры MMX

Восемь 64-разрядных регистров: **MM0**, **MM1**, …, **MM7**

Форматы данных

Packed bytes: 8 × INT8
Packed words: 4 × INT16
Packed doublewords: 2 × INT32
Quadword: 1 × INT64

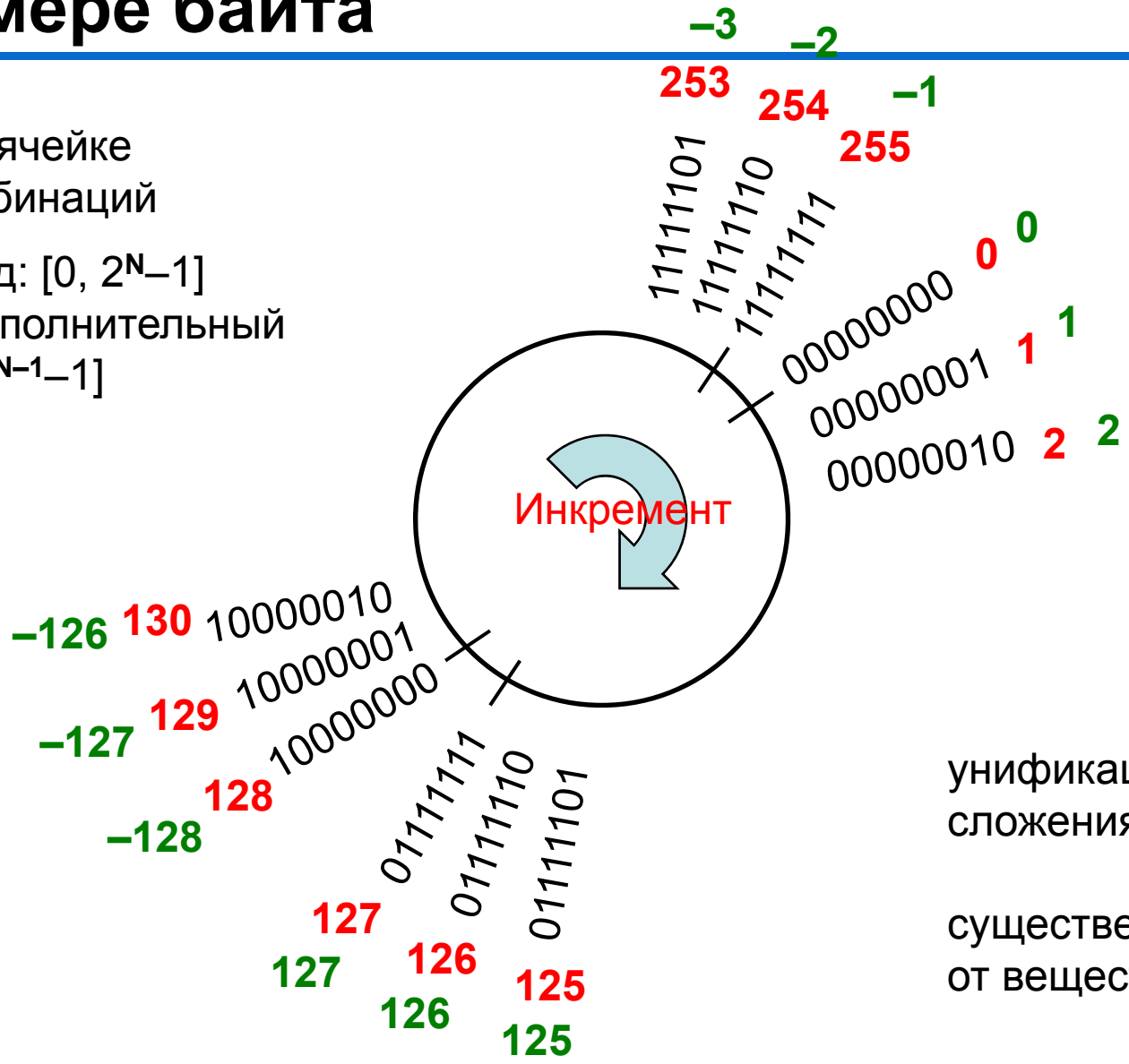
Арифметики

Wraparound arithmetic (кольцо целых чисел, max+1=min)
Saturation arithmetic (арифметика с насыщением)

Кольцо целых чисел на примере байта

в **N**-битовой ячейке
всего 2^N комбинаций

двоичный код: $[0, 2^N-1]$
двоичный дополнительный код: $[-2^{N-1}, 2^{N-1}-1]$

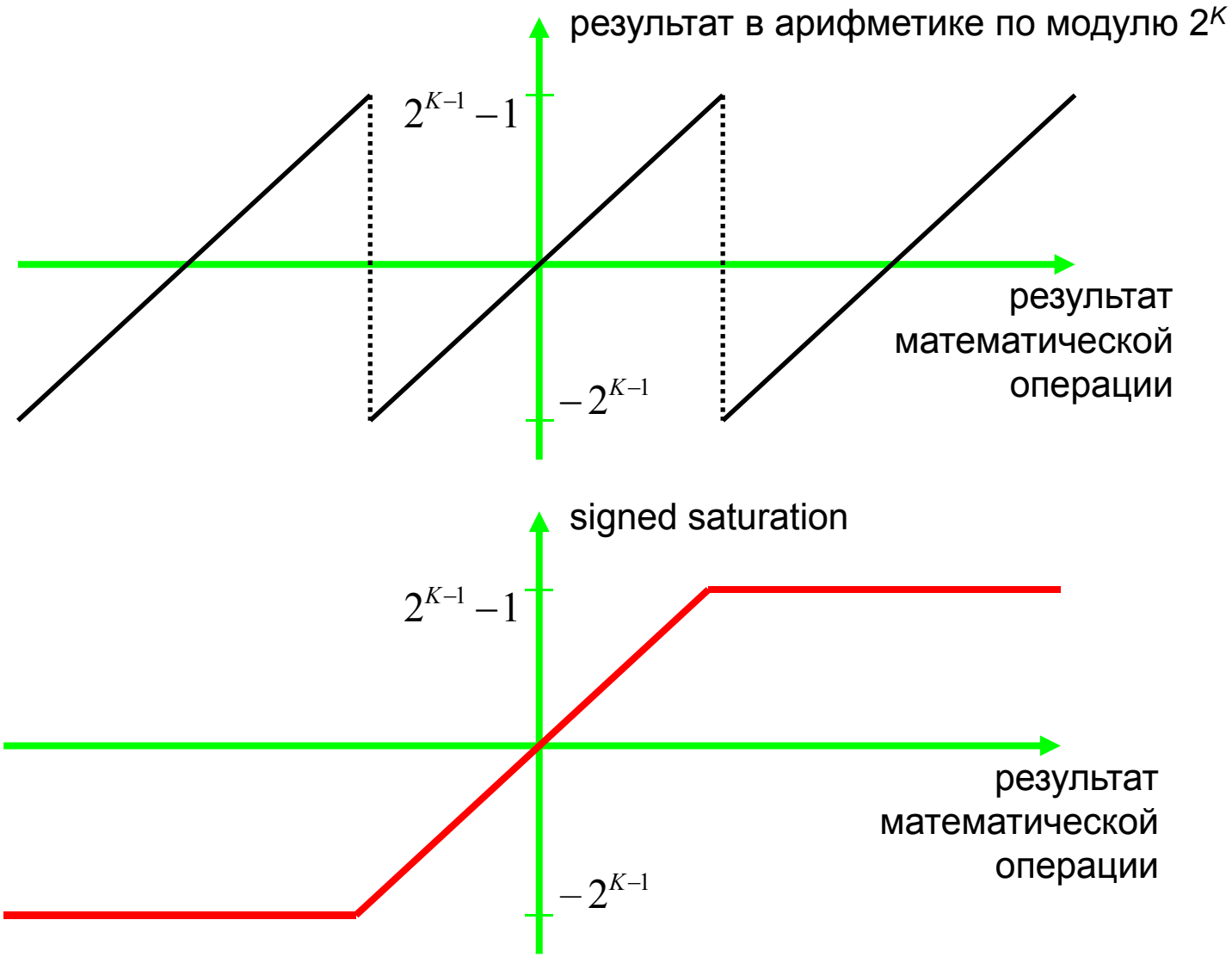


унификация операций сложения/вычитания

существенное отличие от вещественных чисел

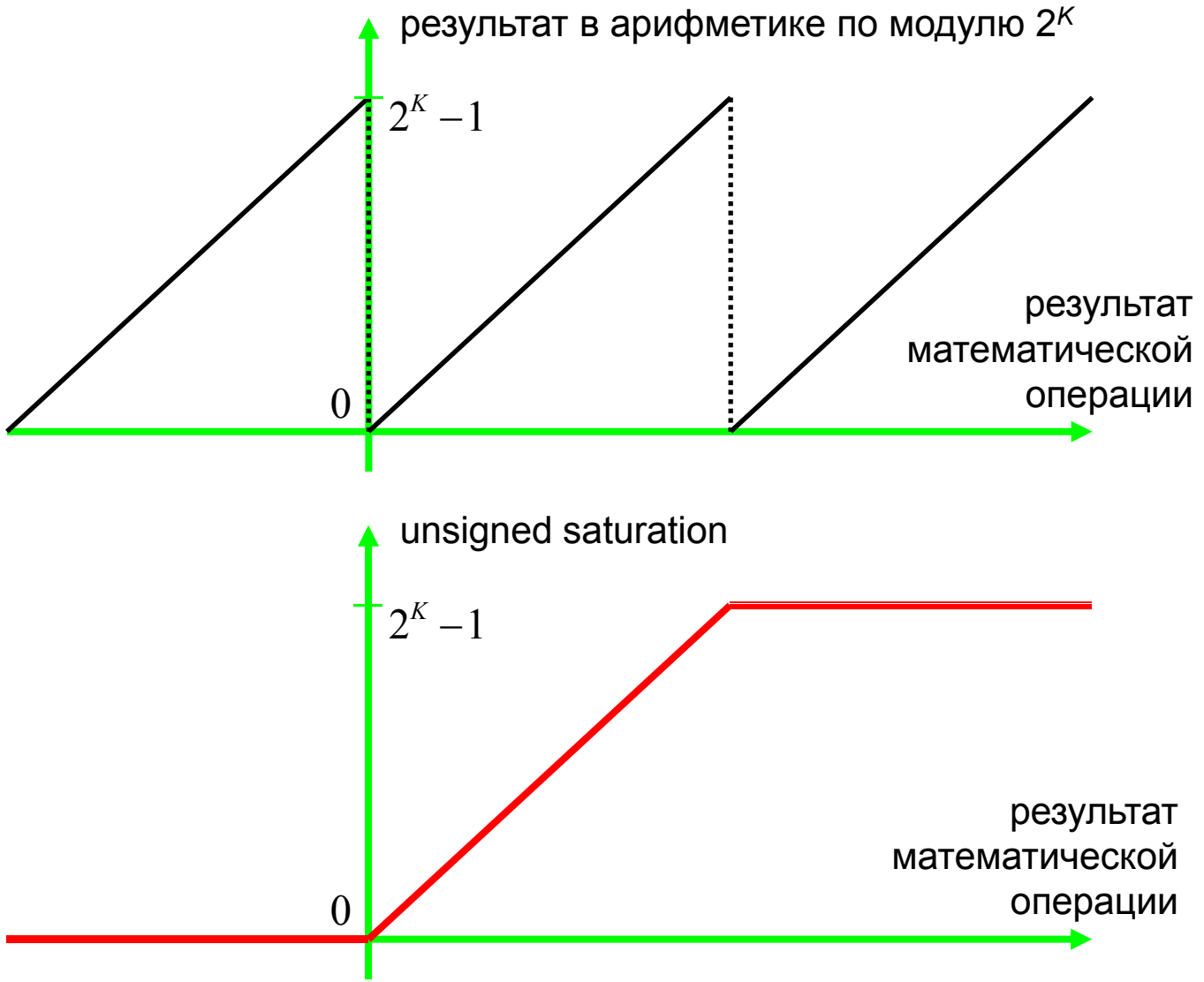
Знаковое насыщение

например, обработка аудиоинформации



Беззнаковое насыщение

например, обработка видеоинформации



Технология MMX – мнемоника

Синтаксис MMX-инструкций

P = Packed
S = signed Saturation
US = Unsigned Saturation

B – операнд из однобайтовых данных
W – операнд из двухбайтовых данных
D – операнд из четырёхбайтовых данных
Q – восьмибайтовый операнд

Технология MMX – машинный код

Кодирование MMX-инструкций

трёхбитовое поле mmx:
- 000 – mm0
- 001 – mm1
- 010 – mm2
- 011 – mm3
- 100 – mm4
- 101 – mm5
- 110 – mm6
- 111 – mm7

трёхбитовое поле reg:
- 000 – eax
- 001 – ecx
- 010 – edx
- 011 – ebx
- 100 – esp
- 101 – ebp
- 110 – esi
- 111 – edi

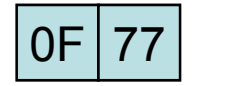
двухбитовое поле gg:
- 00 – B
- 01 – W
- 10 – D
- 11 – Q

Технология MMX – инициализация FPU

Команда **EMMS** (Empty MMX State)

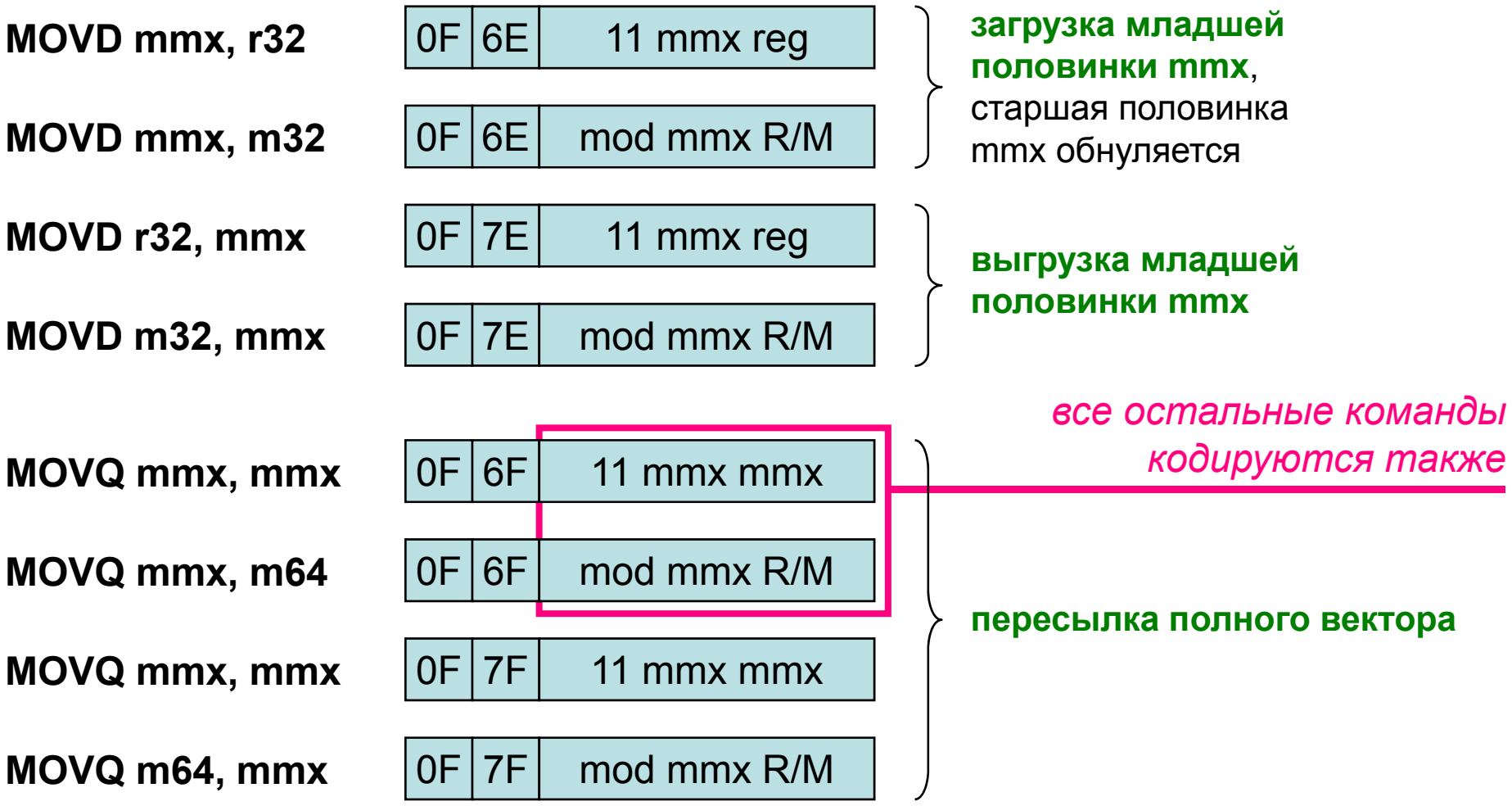
EMMS

мнемоника



машинный код

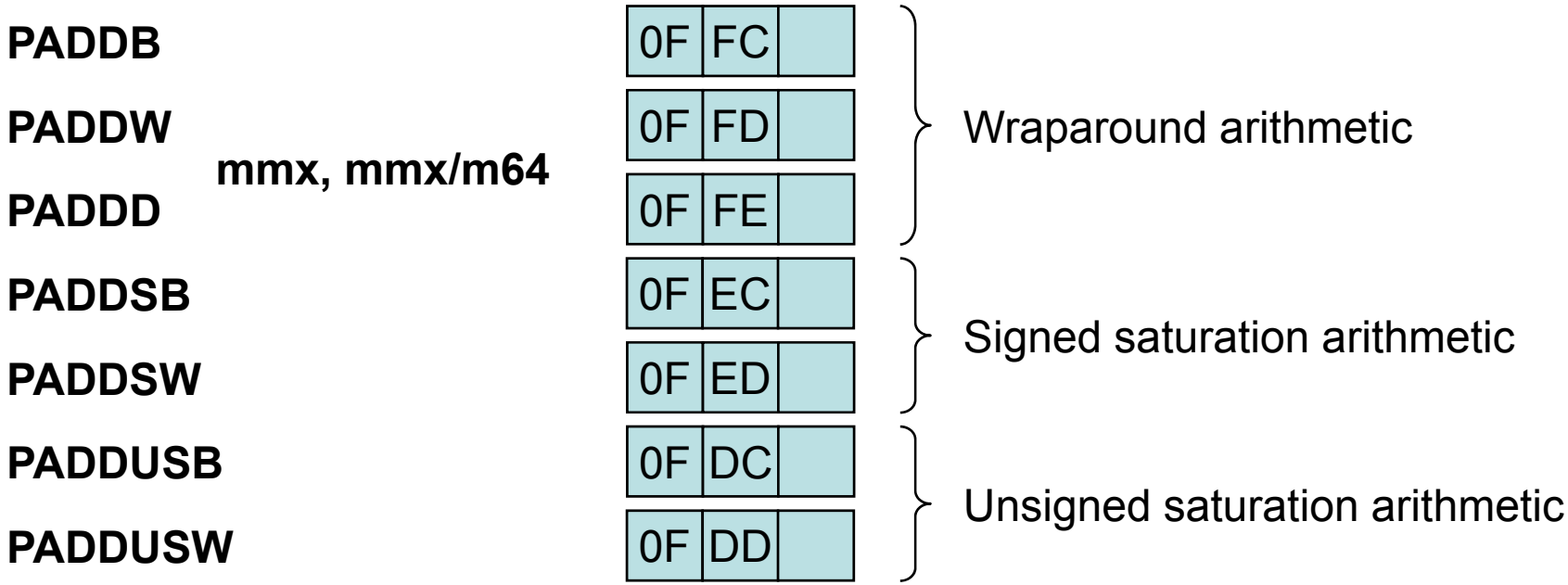
Технология MMX – пересылка



Технология MMX – побитовые операции

PAND		0F	DB		
POR	**mmx, mmx/m64**	0F	EB		
PXOR		0F	EF		
PANDN		0F	DF		$приёмник = \overline{приёмник} \ \& \ источник$

Технология MMX – сложение



Технология MMX – вычитание

PSUBB	**mmx, mmx/m64**	0F F8	Wraparound arithmetic	
PSUBW		0F F9		
PSUBD		0F FA		
PSUBSB		0F E8	Signed saturation arithmetic	
PSUBSW		0F E9		
PSUBUSB		0F D8	Unsigned saturation arithmetic	
PSUBUSW		0F D9		

Технология MMX – сравнение

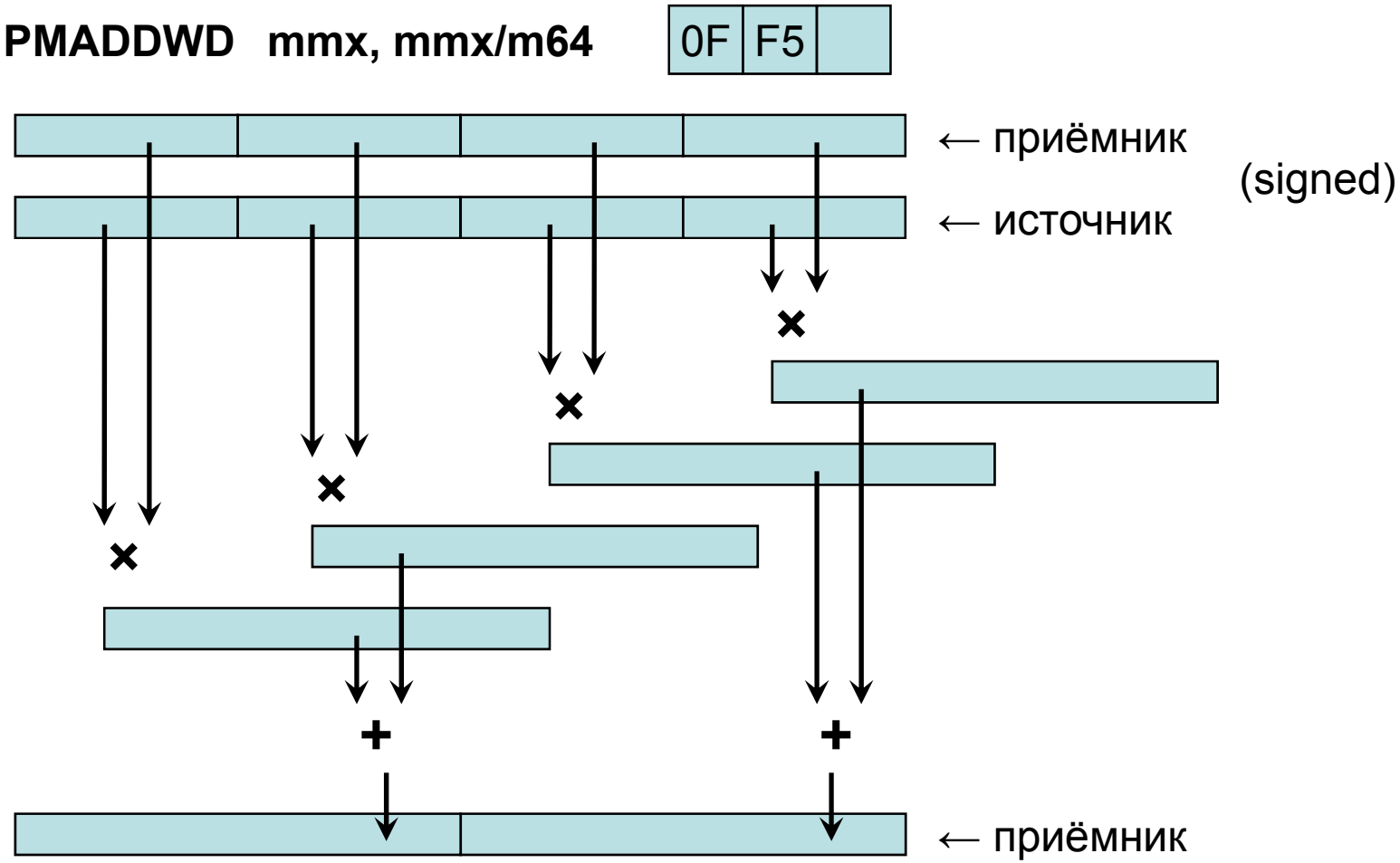
PCMPEQB	**mmx, mmx/m64**	0F	74		**проверка на равенство**
PCMPEQW		0F	75		
PCMPEQD		0F	76		
PCMPGTB		0F	64		**проверка на больше**
PCMPGTW		0F	65		
PCMPGTD		0F	66		

Если условие истинно, соответствующий элемент результирующего вектора заполняется единицами, иначе – нулями.

Технология MMX – умножение (знаковое)

PMULHW	**mmx, mmx/m64**	0F E5	**получение старших 16 бит результата умножения**
PMULLW	**mmx, mmx/m64**	0F D5	**получение младших 16 бит результата умножения**

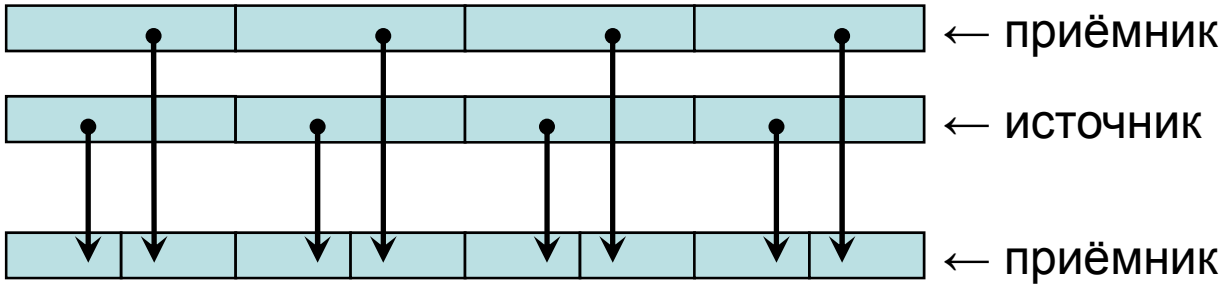
Технология MMX – умножение и сложение



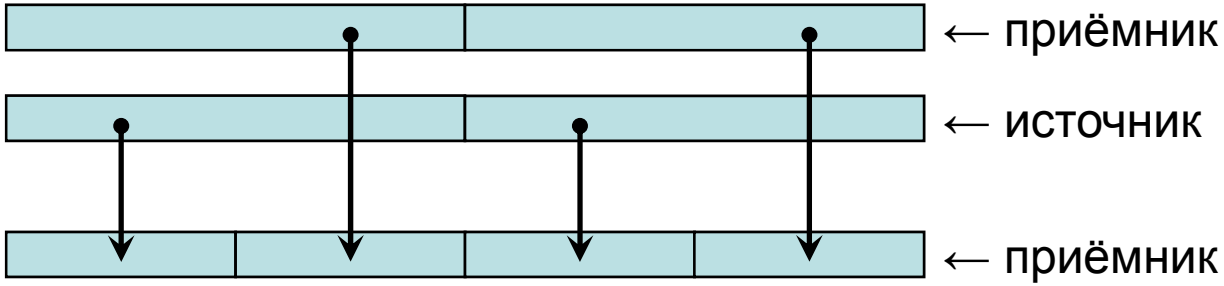
Технология MMX – упаковка

PACKSSWB **mmx, mmx/m64** | 0F | 63 | | Signed saturation arithmetic

PACKUSWB **mmx, mmx/m64** | 0F | 67 | | Unsigned saturation arithmetic
(знаковый word в беззнаковый byte)

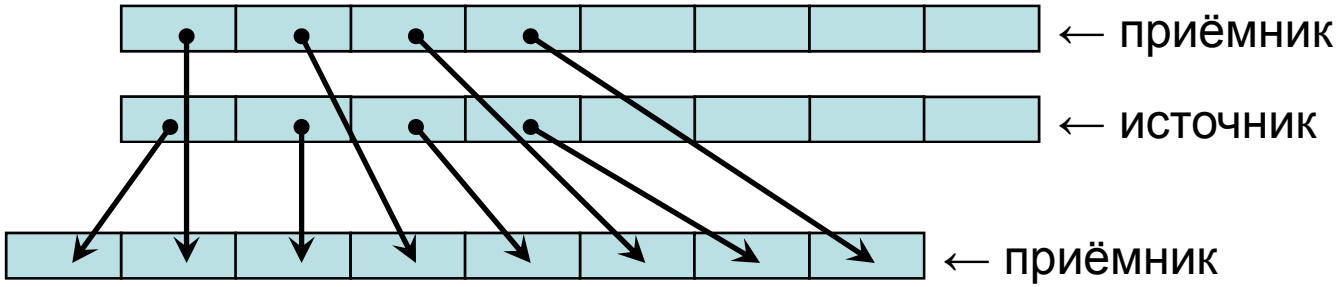


PACKSSDW **mmx, mmx/m64** | 0F | 6B | | Signed saturation arithmetic



Технология MMX – распаковка

PUNPCKHBW mmx, mmx/m64 | 0F | 68 | | **high bytes → words**



PUNPCKLBW	**mmx, mmx/m64**	0F	60	**low bytes → words**
PUNPCKHWD	**mmx, mmx/m64**	0F	69	**high words → doublewords**
PUNPCKLWD	**mmx, mmx/m64**	0F	61	**low words → doublewords**
PUNPCKHDQ	**mmx, mmx/m64**	0F	6A	**high doublewords → quadword**
PUNPCKLDQ	**mmx, mmx/m64**	0F	62	**low doublewords → quadword**

Технология MMX – сдвиги

PSLLW	**mmx, mmx/m64**	0F F1	}	**Shift left logical** (shl)
PSLLD		0F F2		
PSLLQ		0F F3		
PSRLW		0F D1	}	**Shift right logical** (shr)
PSRLD		0F D2		
PSRLQ		0F D3		
PSRAW		0F E1	}	**Shift right arithmetic** (sar)
PSRAD		0F E2		

64-битный источник – количество разрядов для сдвига

Технология MMX – сдвиги

PSLLW	mmx, imm8	0F	71	11110 mmx	i8
PSLLD		0F	72	11110 mmx	i8
PSLLQ		0F	73	11110 mmx	i8
PSRLW		0F	71	11010 mmx	i8
PSRLD		0F	72	11010 mmx	i8
PSRLQ		0F	73	11010 mmx	i8
PSRAW		0F	71	11100 mmx	i8
PSRAD		0F	72	11100 mmx	i8

Shift left logical (shl)

Shift right logical (shr)

Shift right arithmetic (sar)

Спаривание MMX-команд на P55C

P55C имеет **два конвейера (U и V)** для выполнения MMX-команд.

MMX-команды
- **Класс U** – с обращением к памяти или РОН
- **Класс UV** – без обращения к памяти или РОН

MMX-команды класса U спариваются только MMX-командами класса UV.

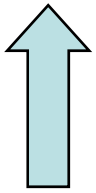
MMX-команды класса UV спариваются с любыми командами.

P55C имеет только одно устройство для MMX-умножения и только одно устройство для MMX-сдвига.

Пример
Модуль разности двух массивов байт

Цикл:

```
        MOVQ     mm0, sour   ; mm0 = вектор B
        MOVQ     mm1, dest   ; mm1 = вектор A
спаривание
        PSUBUSB  mm1, mm0    ; mm1 = a–b , если a>b (иначе ноль)
        PSUBUSB  mm0, dest   ; mm0 = b–a , если a<b (иначе ноль)
        POR      mm1, mm0    ; mm1 = | a–b |
        MOVQ     dest, mm1   ; выгрузка результата
```



параллельное исполнение 16 арифметических команд

ветвлений нет, хотя операция взятия модуля предполагает ветвление

Пример
Знаковое расширение

Цикл:

```
        MOVQ      mm0, sour   ; исходный вектор A
спаривание
        PXOR      mm2, mm2    ; нули
        MOVQ      mm1, mm0    ; копия исходного вектора
        PCMPGTB   mm2, mm0    ; mm2 = a<0 ? 0xFF : 0x00
спаривание
        PUNPCKLBW mm0, mm2    ; расширение младших четырёх байт
        PUNPCKHBW mm1, mm2    ; расширение старших четырёх байт
```

ветвлений нет, хотя операция предполагает ветвление

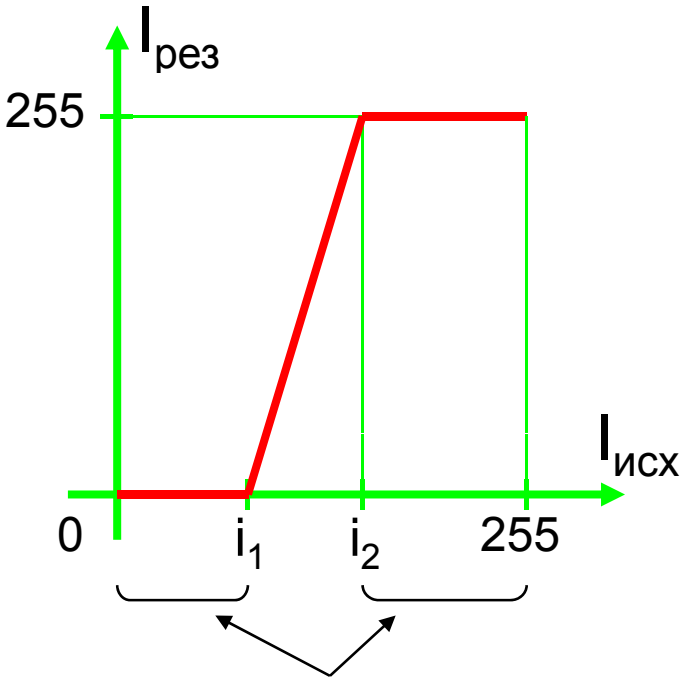
Пример
Наложение графического объекта на фон

Цикл:

```
          MOVQ      mm1, object
спаривание
          MOVQ      mm0, background
          PCMPEQB   mm1, mm7    ; выявление прозрачных пикселов
спаривание
          PAND      mm0, mm1    ; очистка заменяемых пикселов
          PANDN     mm1, object ; очистка прозрачных пикселов
          POR       mm0, mm1    ; наложение
          MOVQ      background, mm0
```

ветвлений нет, хотя операция предполагает ветвление

Пример
Операция контрастирования



вероятность попадания атрибута пиксела в данные интервалы мала

Цикл:

MOVQ
PSUBUSB
MOVQ
PUNPCKLBW
PUNPCKHBW
PSLLW
PSLLW
PMULHW
PMULHW
PACKUSWB
MOVQ

Необходимость загрузки/выгрузки…

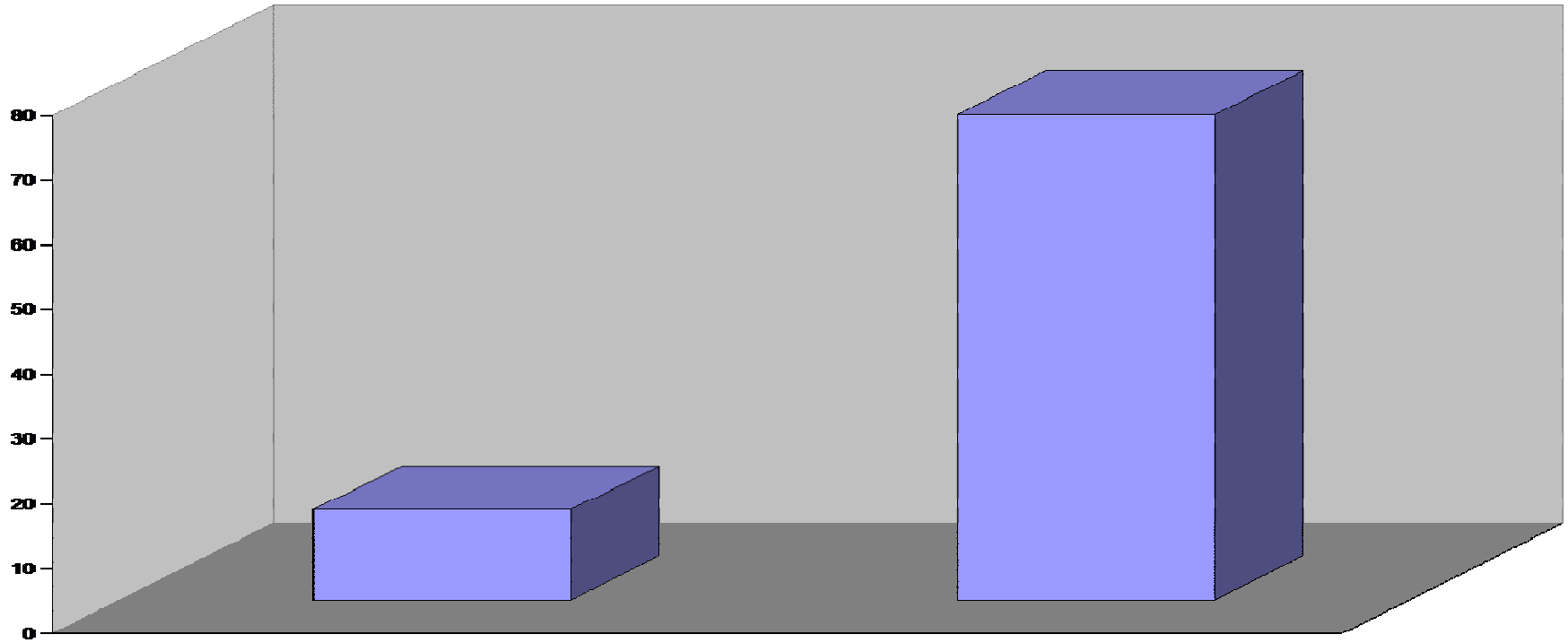
Большие накладные расходы…

Может быть, эффективнее простой (скалярный) цикл на базе XLAT по заранее вычисленной таблице замены?

Цикл:

LODSB
XLAT
STOSB

Производительность операции контрастирования на P55C (200 MHz)



14 Mb/sec
цикл на базе XLAT

75 Mb/sec
цикл по технологии MMX

Реальный пример

Со сканирующего устройства поступают отсчёты в формате 10–10–10 (в цветовой модели RGB). Требуется представить отсчёты в формате 8–8–8, осуществив нормирующее преобразование:

$$x_{рез}(i) = \begin{cases} 0, & x_{исх}(i) < x_{\min}(i) \\ 255 \times \dfrac{x_{исх}(i) - x_{\min}(i)}{x_{\max}(i) - x_{\min}(i)}, & x_{\min}(i) \le x_{исх}(i) \le x_{\max}(i) \\ 255, & x_{\max}(i) < x_{исх}(i) \end{cases}$$

$x_{исх}(i)$ – исходный отсчёт, $0 \le x_{исх}(i) \le 1023$

$x_{рез}(i)$ – результирующее значение отсчёта, $0 \le x_{рез}(i) \le 255$

$x_{min}(i)$ – минимальный уровень, полученный на чёрном эталоне

$x_{max}(i)$ – максимальный уровень, полученный на белом эталоне

Реальный пример

Проблемы

- нет инструкции деления
- нет вещественной арифметики
- умножение только 16-битовых целых чисел со знаком
- неудобный формат исходных данных

Формат исходных данных

32 bpp: 00 bb gg rr BBBBBBBB GGGGGGGG RRRRRRRR

младшие биты

Использование PMULHW в роли операции вещественного деления

$$\left[2^{12} \times \frac{255}{x_{\max}(i) - x_{\min}(i)} \right] \times 2^4 \times \left[x_{исх}(i) - x_{\min}(i) \right]$$

$0 \div 16368$

const(i), 1021÷32640

max–min=1023 → 1021
max–min=1022 → 1022
•••
max–min=33 → 31651
max–min=32 → 32640 **< 2^{15}**

Δ≥1 всегда

ограничение: max–min ≥ 32

PMULHW
одновременно делит на 2^{16}
$2^{16} = 2^{12} \times 2^4$

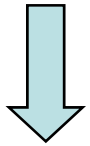
Таким образом избрана 12 степень: меньше нельзя – смыкаются константы, больше нельзя – усиливается ограничение на эталонные значения

Реальный пример

```
@@Loop:
        IN          eax, dx       ; 00bbggrr BBBBBBBB GGGGGGGG RRRRRRRR
        MOVD        mm1, eax      ; загрузка mm1
        SHR         eax, 12       ; 00000000 000000bb ggrrBBBB BBBBGGGG
        SHR         ax, 6         ; 00000000 000000bb 000000gg rrBBBBBB
        SHR         al, 6         ; 00000000 000000bb 000000gg 000000rr
        SHL         eax, 6        ; 00000000 bb000000 gg000000 rr000000
        MOVD        mm0, eax      ; загрузка mm0
        PUNPCKLBW   mm0, mm1      ; [?] [BB…B000000] [GG…G000000] [RR…R000000]
        PSRLW       mm0, 2        ; [?] [00BB…B0000] [00GG…G0000] [00RR…R0000]
        PSUBUSW     mm0, [ebx]    ; вычитание [?] [16×Bmin(i)] [16×Gmin(i)] [16×Rmin(i)]
        PMULHW      mm0, [ebx+6]  ; нормализация, константы подготовлены
        PACKUSWB    mm0, mm7      ; второе насыщение, mm7 не используется
        MOVD        eax, mm0      ; выгрузка mm0: [?] [B…B] [G…G] [R…R]
        STOSD                     ; сохранение результата в памяти
        ADD         ebx, 12       ; смещение указателя в таблице констант
        DEC         edi           ; 24 bpp
        LOOP        @@Loop
```

Резюме

- Языки высокого уровня ориентированы на кодирование алгоритмов для вычислительных систем класса SISD и не позволяют использовать технологию MMX
- Возможность генерации компиляторами эффективного кода класса SIMD сомнительна



Необходимость применения низкоуровневого программирования

Streaming SIMD Extensions (SSE)
Pentium III

MASKMOVQ **mmx, mmx**
источник, маска

0F	F7	

приёмник (m64) размещён по адресу ds:[e]di
побайтно в приёмник копируется
либо источник, если знаковый бит маски установлен,
либо нулевой байт, в противном случае

PEXTRW r32, mmx, i8

0F	C5	11 reg mmx	i8

старшие 16 бит приёмника (r32) обнуляются;
в младшие 16 бит приёмника (r32) копируется слово из источника (mmx),
номер которого в двух младших битах непосредственного операнда (i8)

PINSRW mmx, r32/m16, i8

0F	C4	mod mmx r/m	i8

младшие 16 бит источника (r32/m16) копируются в слово приёмника (mmx),
номер которого в двух младших битах непосредственного операнда (i8)

Streaming SIMD Extensions (SSE)
Pentium III

MOVNTQ	**m64, mmx**	0F E7	**аналогично MOVQ, но минуя кеш**
PMOVMSKB	**r32, mmx**	0F D7 11 reg mmx	

старшие 24 бита приёмника (r32) обнуляются;
в младшие 8 бит приёмника (r32) копируются
знаковые биты всех восьми байт источника (mmx)

PAVGB	**mmx, mmx/m64**	0F E0	**вычисление среднего**
PAVGW		0F E3	
PMAXSW	**mmx, mmx/m64**	0F EE	**максимум (signed word)**
PMAXUB		0F DE	**максимум (unsigned byte)**
PMINSW		0F EA	**минимум (signed word)**
PMINUB		0F DA	**минимум (unsigned byte)**

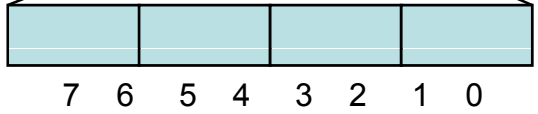
Streaming SIMD Extensions (SSE)
Pentium III

PMULHUW mmx, mmx/m64 | 0F | E4 | | **аналогично PMULHW, но unsigned**

PSADBW mmx, mmx/m64 | 0F | F6 | |

в младшее слово приёмника кладётся сумма модулей разностей байт; остальные слова приёмника обнуляются

PSHUFW mmx, mmx/m64, i8 | 0F | 70 | | i8 |



в каждое слово приёмника копируется слово из источника, номер которого находится в соответствующих двух битах непосредственного операнда (i8)

Блок XMM (eXtended MultiMedia)

Блок полностью независим.
Требуется поддержка в многозадачных ОС.

Регистры XMM

Восемь 128-разрядных регистров: **XMM0**, **XMM1**, …, **XMM7**

Форматы данных

4 × FLOAT32 – **SSE, Pentium III**

2 × FLOAT64

16 × INT8
8 × INT16
4 × INT32
2 × INT64

SSE2, Pentium 4

} в скалярном виде можно использовать вместо FPU одновременно с MMX

Дефект MMX/XMM

Фиксированная длина вектора, зависящая от кода операции!

(следующее поколение IA32 могло бы использовать более длинные вектора, выполняя тоже самый двоичный код)

Вопросы для обсуждения (x86)

- Сравните реализации векторной обработки в виде конвейерного ALU и в виде массива ALU
- Основное предназначение арифметики с насыщением вовсе не борьба с переполнениями?
- Почему MMX регистров ровно восемь?
- Почему XMM регистров ровно восемь?
- Сравните скалярные вычисления с плавающей точкой в SSE2 и в FPU
- Недостатки MMX/XMM на фоне векторных процессоров других производителей

Вопросы для обсуждения (x86) – подсказки

- конвейерное ALU – экономия транзисторов
  массив ALU – максимум производительности
- безальтернативная обработка
  (полное уничтожение точек ветвления)
- реализация на базе регистров FPU
- используются три бита в байте MOD R/M
- SSE2 – регистровая архитектура вычислений
  FPU – стековая архитектура вычислений
- жёстко фиксированная длина вектора
  короткие вектора

Темы заданий для самостоятельной работы

- Сравните производительность реализаций операции контрастирования в векторной и скалярной формах (слайд 35) на современных процессорах x86
- Сравните производительность строковых операций x86 с тождественными векторными вычислениями для следующих случаев:

REP STOS	REPZ CMPS	REPZ SCAS
REP MOVS	REPNZ CMPS	REPNZ SCAS

Следующая тема

- **Архитектура VLIW**