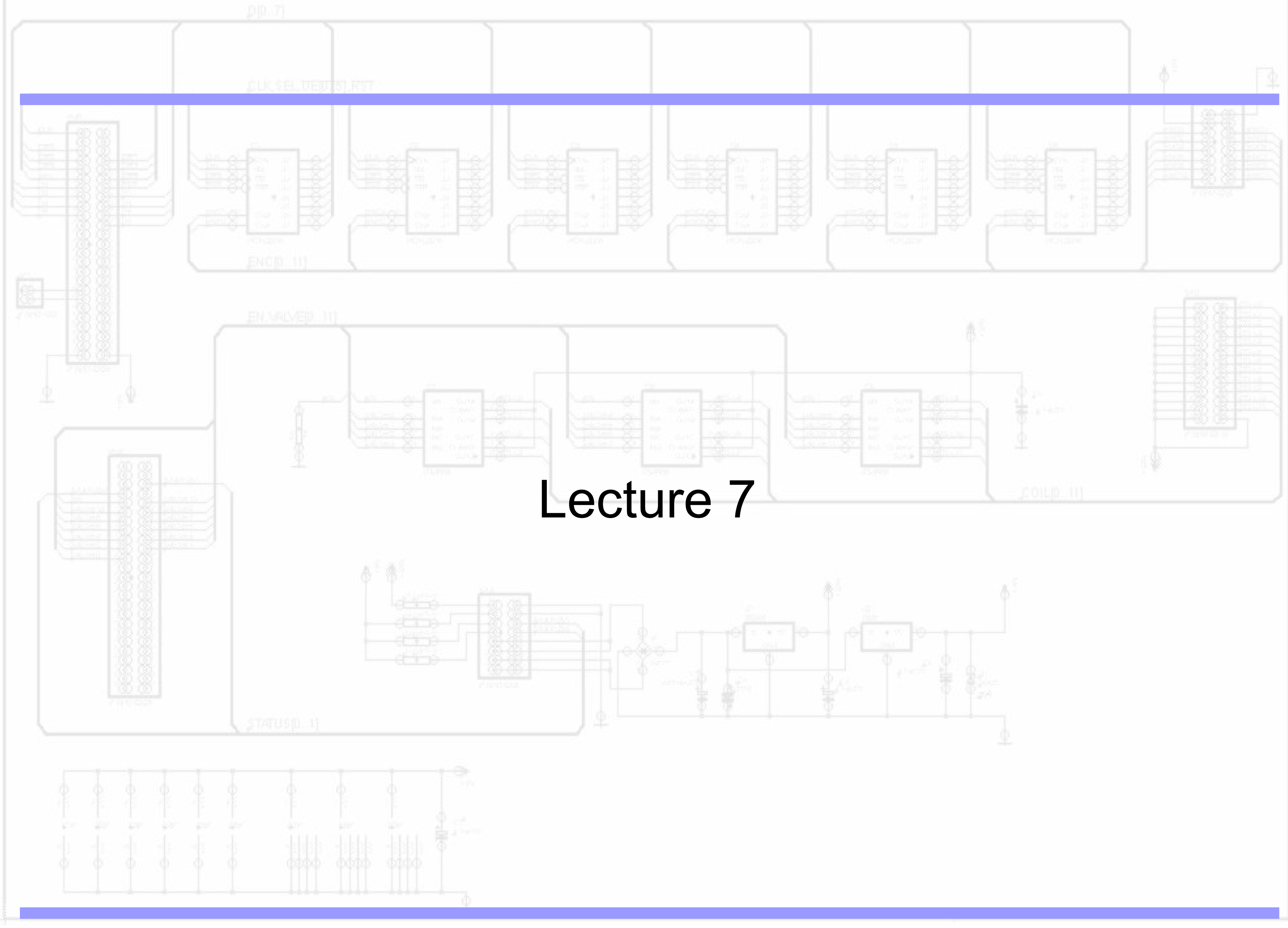


Lecture 7

SIMD instruction extensions

SIMD

SIMD (*Single Instruction Multiple Data*) – technique allowing to achieve data processing parallelism. In this approach one operation (addition, multiplication, etc.) is applied to several data points organized as the data vector. When processing long data blocks that require the same operations such approach can result in significant reduction of computational time. For example when we want to change the brightness of an image of the following pixel color format: red, green and blue components each 8 bits long, we can perform an addition (subtraction) operation to 2 and 2/3-rd of pixels at the same time.

In IA-32 and Intel 64 architecture processors the following two groups of SIMD instruction are available for software developers: **MMX** (*MultiMedia eXtension*) and **SSE** (*Streaming SIMD Extensions*). MMX instructions are predestined for integer arithmetic signal processing while SSE instruction operate upon floating point data types.

The MMX was introduced in year 1997 with Pentium MMX processor. It includes 57 instructions and four new vectorized data types. It found its applications in image, video and audio processing applications, 2D and 3D graphics, communications, speech recognition and many others.

The SSE first built into Pentium III from 1999 processor. It introduced 48 new instructions and 16 floating point registers with 128 bits precision. At the present moment also some later extended versions: SSE2, SSE3 and SSE4 are available in modern processors. The SSEs are vastly applied to computer graphics and signal processing.

MMX instructions extension

MMX registers
The MMX instructions operate on eight 64-bits registers mm0-mm7 that are aliased with FPU (*Floating Point Unit*) registers f0-f7 (see Fig. 6.1). Hence simultaneous usage of FPU and MMX in our applications is hindered (disadvantage).

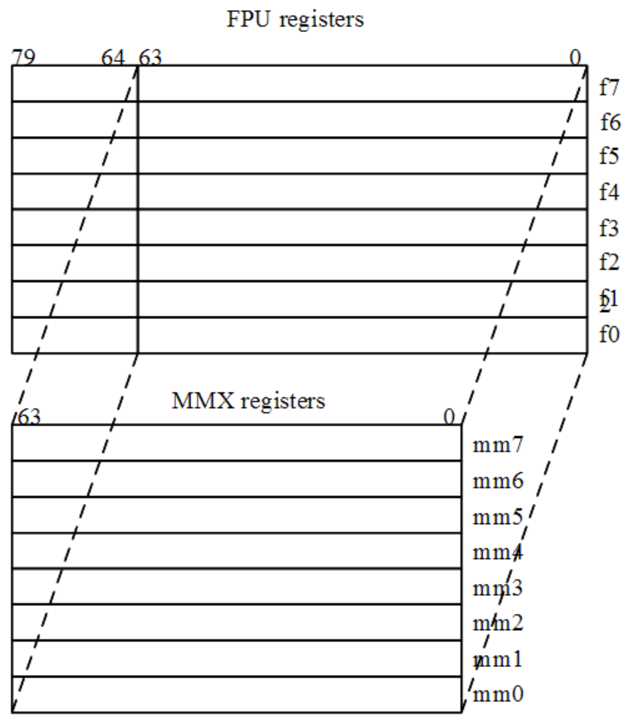


Fig. 6.1. Mapping of MMX registers into
FPU 80-bit wide registers (significant part)

The MMX registers are freely accessible from instruction level, it means that they are not organized in stack as FPU registers f0-f7.

MMX instructions extension

MMX data types
MM0-MM7 registers are 64-bit wide however they enable to operate upon smaller parts of data organized in blocks (**data vectors**). The following vectorized data types are available:

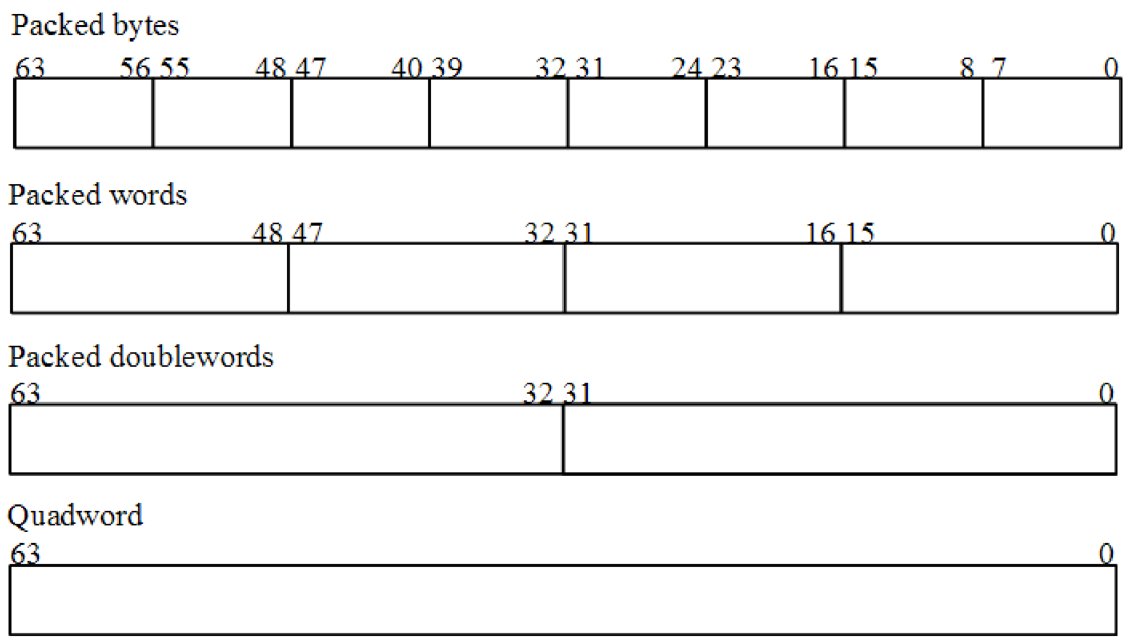


Fig. 6.2. MMX data types

When such data structures are read/written to memory we must remember that they are accessed in accordance with *little-endian* order. It means that first less important than more important part of data is stored.

MMX instructions extension

MMX instructions

1. Data transfer instructions
In subsequent slides the following abbreviations will be assumed:
reg32 – 32-bit general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI),
mem32 – 32-bit value from memory,
mem64 – 64-bit value from memory,
mmreg – MMX register.

movd mmreg1, reg32/mem32
This instruction copies 32-bit value from general purpose register or memory to MMX register. Bits 32-63 are padded with 0.

movd reg32/mem32, mmreg1
It copies 32-bit value from MMX register to general purpose one or memory location.

movq mmreg1, mmreg2/mem64
movq mmreg2/mem64, mmreg1
These instruction operate just like the ones previously discussed but on 64-bit data.

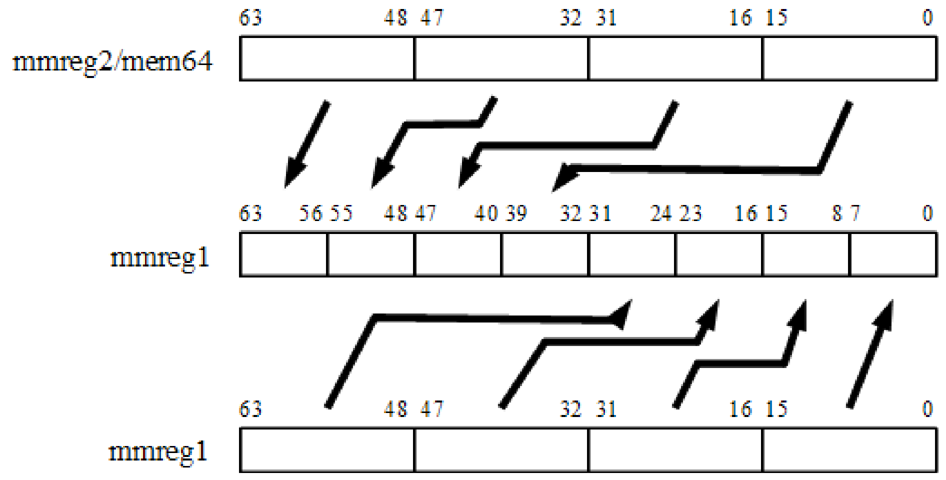
MMX instructions extension

Examples

```
movd      mm0, eax
movd      ebx, mm1
movd      [bx], mm0
movq      mm0, mm1
movq      mm0, [si+4]
```

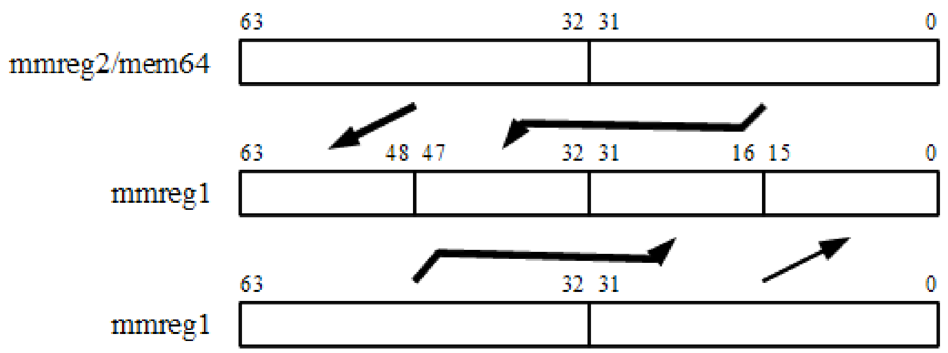
2. Data conversion/handling instructions

packsswb mmreg1, mmreg2/mem64
This instruction packs 16-bit data chunks into 8-bit ones with saturation. It means that any value from beyond an interval <-128, 127> will be replaced with the closest allowed value. Data packing scheme is explained in the following figure.



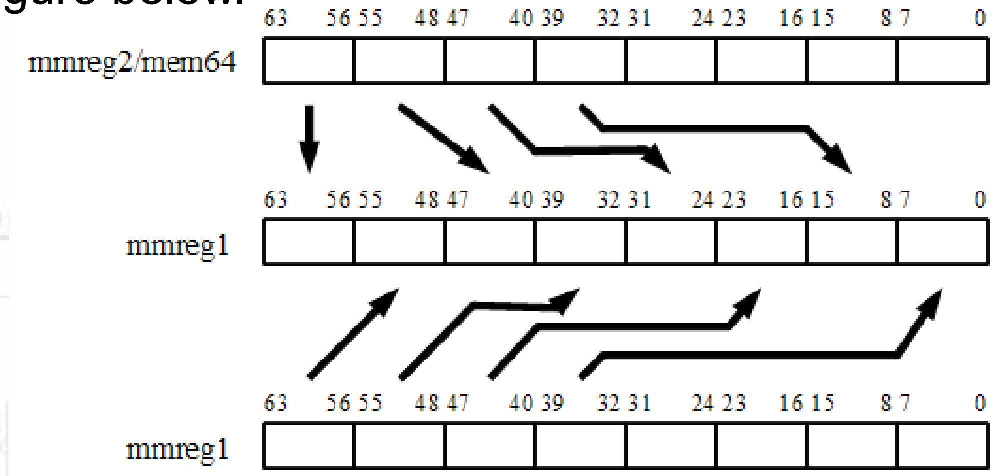
MMX instructions extension

packsswd mmreg1, mmreg2/mem64
For this instruction 32-bit chunks of data are packed with saturation to 16-bit blocks. See figure below.



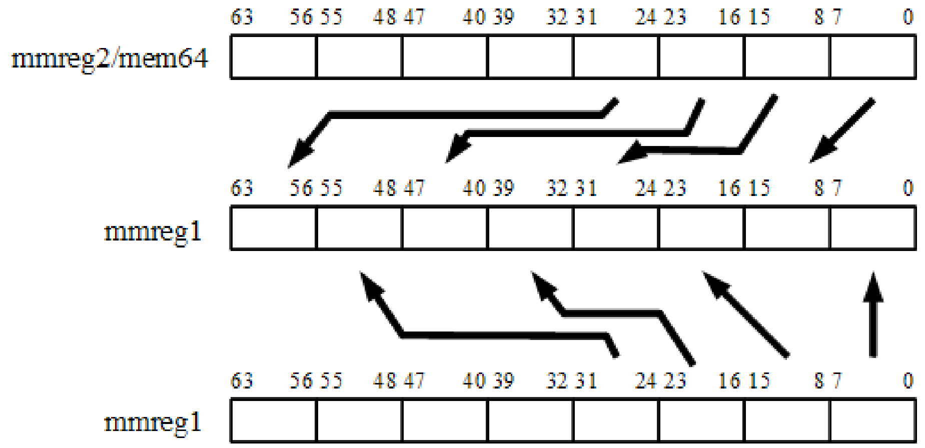
packuswb mmreg1, mmreg2/mem64
Operates like packsswb instruction but the allowed interval changes to <0, 255>.

punpckhbw mmreg1, mmreg2/mem64
For explanation see figure below.

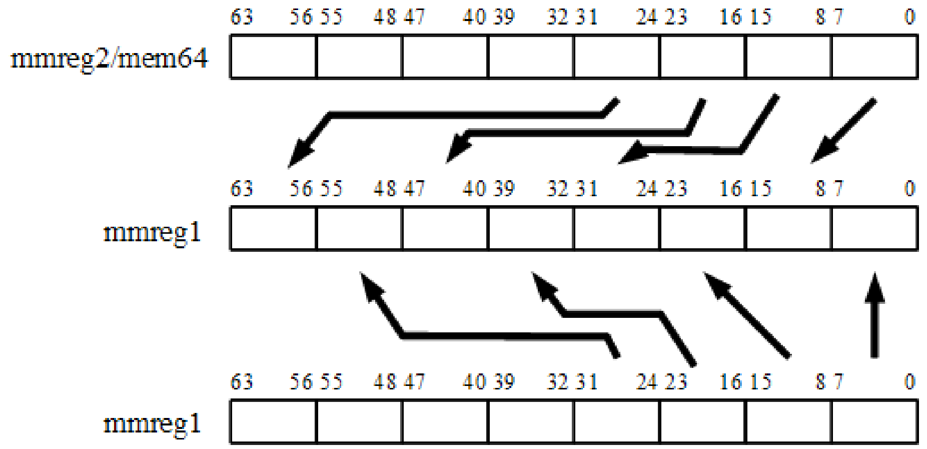


MMX instructions extension

punpcklbw mmreg1, mmreg2/mem64
For explanation see figure below.

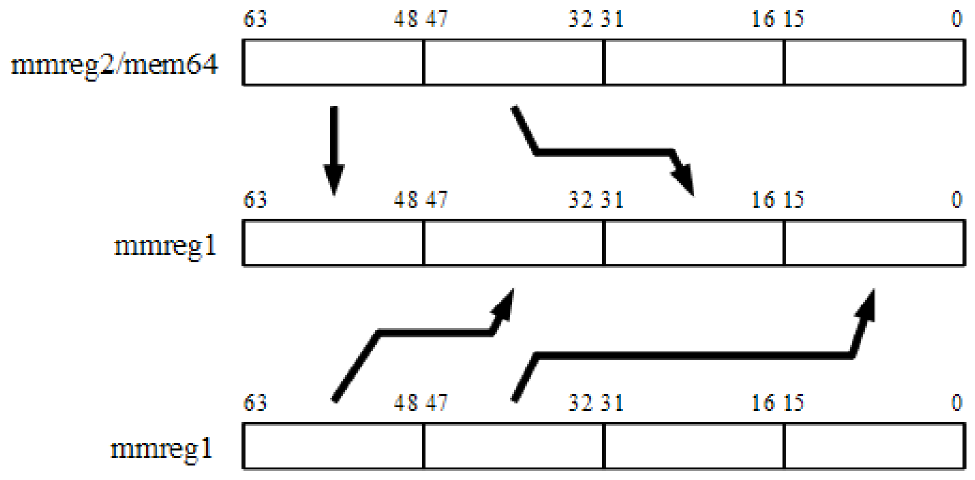


punpcklbw mmreg1, mmreg2/mem64
For explanation see figure below.

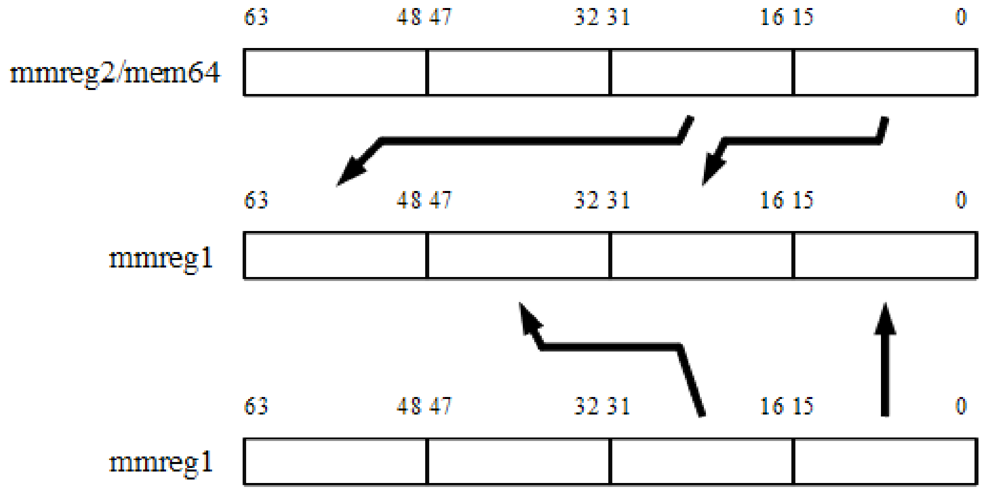


MMX instructions extension

punpckhwd mmreg1, mmreg2/mem64
For explanation see figure below.

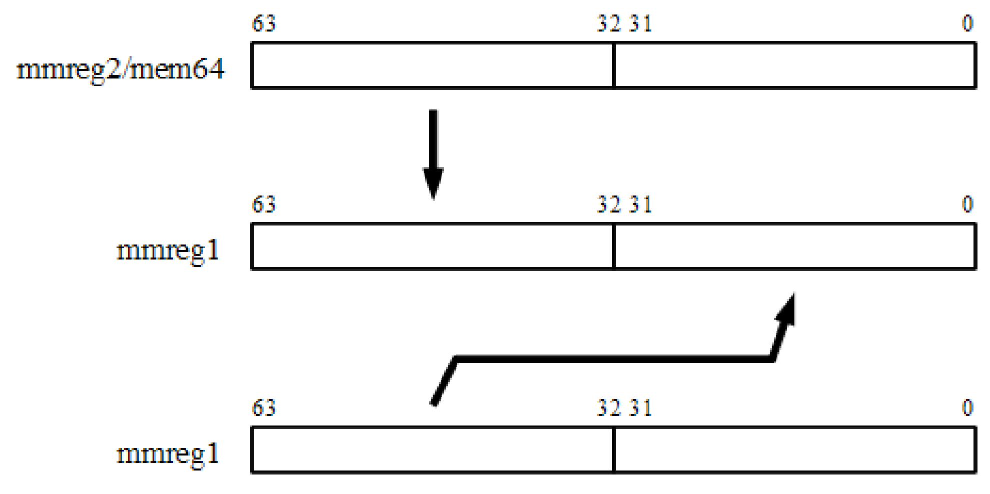


punpcklwd mmreg1, mmreg2/mem64

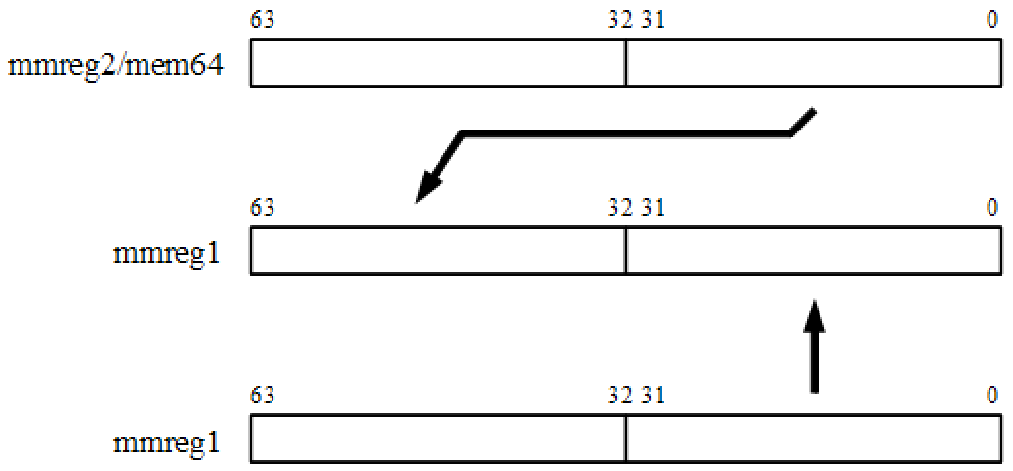


MMX instructions extension

punpckhdq mmreg1, mmreg2/mem64



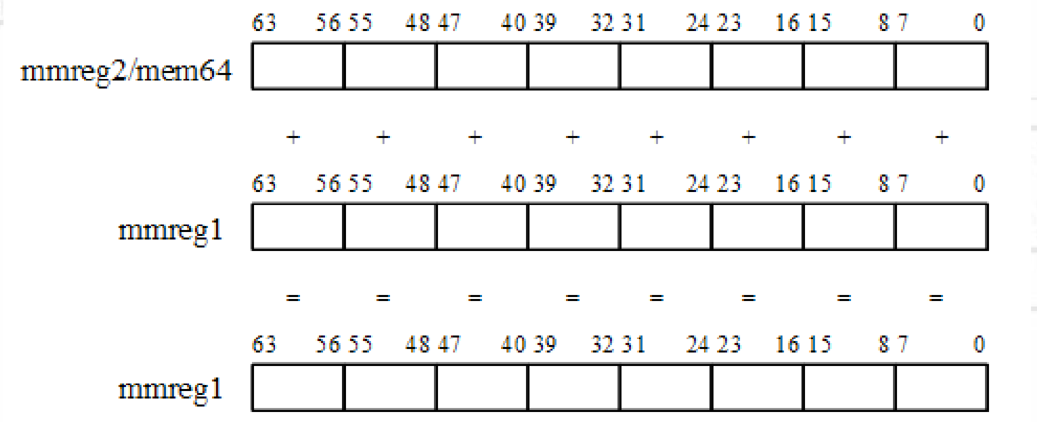
punpckldq mmreg1, mmreg2/mem64



MMX instructions extension

3. Arithmetical instructions

paddb mmreg1, mmreg2/mem64
This instruction adds (no saturation) unsigned 8-bit data blocks separately from first and second operand and the result is written in first operand.



paddw mmreg1, mmreg2/mem64,
paddd mmreg1, mmreg2/mem64
These instructions operate just like paddb instruction but on 16-bit and 32-bit data blocks respectively. While the following two instructions:
paddusb mmreg1, mmreg2/mem64, **paddusw mmreg1, mmreg2/mem64** operate with saturation. It means that the result is cut to the allowed interval with the closest value assigned to the result from beyond the interval. In the same manner but on signed data operate instructions: **paddsb mmreg1, mmreg2/mem64**, **paddsw mmreg1, mmreg2/mem64**.

MMX instructions extension

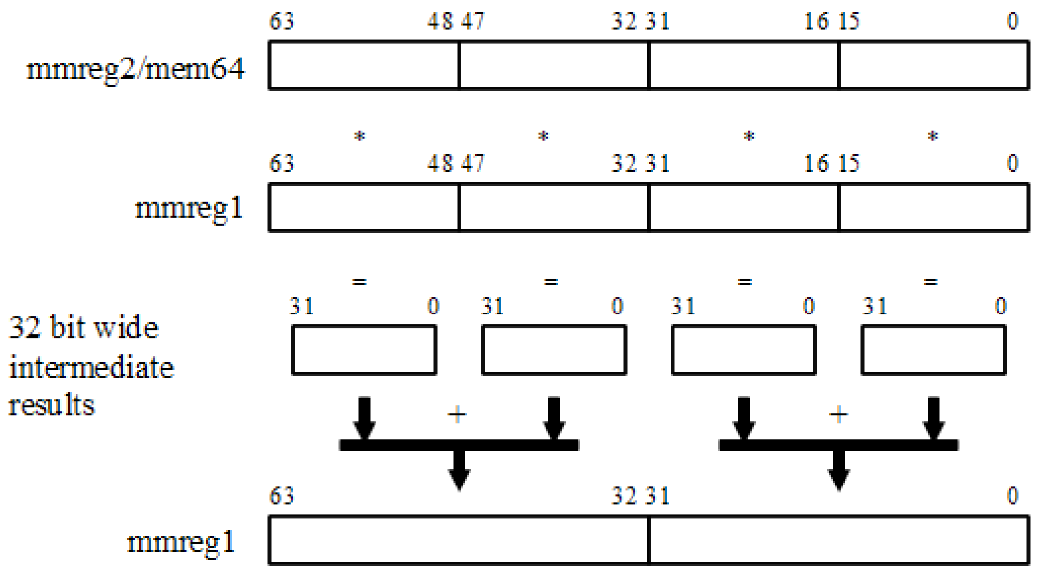
According to described data addition instructions they are also available several subtraction instructions that operate in the same manner but “+” operator must be replaced with “-”. See the list below:

> **psubb mmreg1, mmreg2/mem64**,
> **psubw mmreg1, mmreg2/mem64**,
> **psubd mmreg1, mmreg2/mem64**,
> **psubsb mmreg1, mmreg2/mem64** (with saturation),
> **psubsw mmreg1, mmreg2/mem64** (with saturation),
> **psubusb mmreg1, mmreg2/mem64** (with saturation, unsigned),
> **psubusw mmreg1, mmreg2/mem64** (with saturation, unsigned).

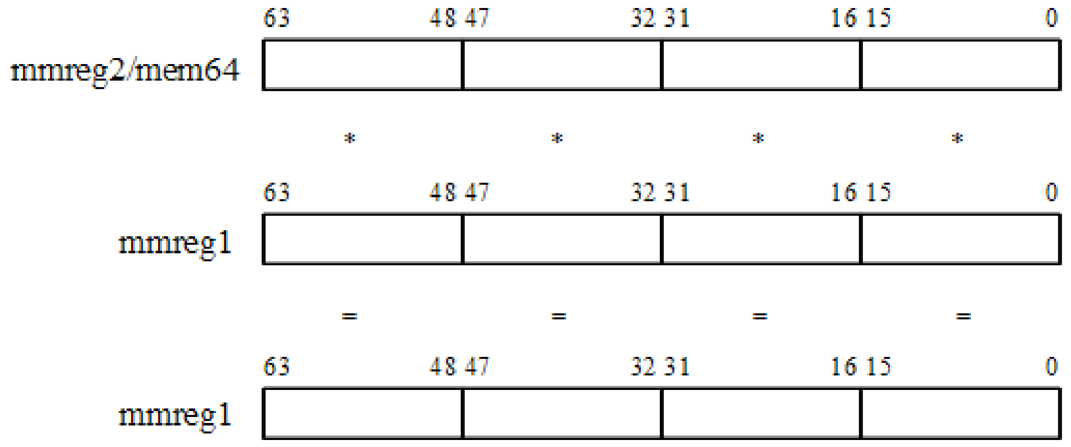
pmaddwd mmreg1, mmreg2/mem64
This instruction multiplies four 16-bit values (signed) from source operand with corresponding 16-bit values from second operand. They are four 32-bit intermediate results generated. Next the first two and the next two intermediate results are summed up resulting in two 32-bit values that are written into the destination operand (see figure in next slide).

pmulhw mmreg1, mmreg2/mem64,
pmullw mmreg1, mmreg2/mem64
Those two instructions multiply four 16-bit values from source and destination operands generating 32-bit results but to the destination operand only higher (respectively lower) 16-bit parts of those results are written (see figure in next slide).

MMX instructions extension



pmaddwd instruction



pmulhw, pmullw instructions

MMX instructions extension

4. Logical instructions

pand mmreg1, mmreg2/mem64
This instruction performs **and** operation on two 64-bit data structures.

pandn mmreg1, mmreg2/mem64
First **not** on the first operand. Next **and** operation on two operands (64 bits).

Analogously:
por mmreg1, mmreg2/mem64 – **or** operation,
pxor mmreg1, mmreg2/mem64 – **xor** operation.

5. Shifting instructions

psllw mmreg1, mmreg2/mem64
It shift left four 16-bit values from first operand by the number of bits in the second operand. Freed bits are filled with zeros. The result is written in the first operand.

pslld mmreg1,mmreg2/mem64 – two 32 bit values,
psllq mmreg1, mmreg2/mem64 – one 64-bit value.

MMX instructions extension

psraw mmreg1, mmreg2/mem64
This instruction shifts four 16-bit long values from the first operand right by the number of bits from the second operand. Freed bits are filled with the sign bit. The results is written in the first operand.

psrad mmreg1, mmreg2/mem64 – two 32-bit values.

psrlw mmreg1, mmreg2/mem64,
psrld mmreg1, mmreg2/mem64,
psrlq mmreg1, mmreg2/mem64

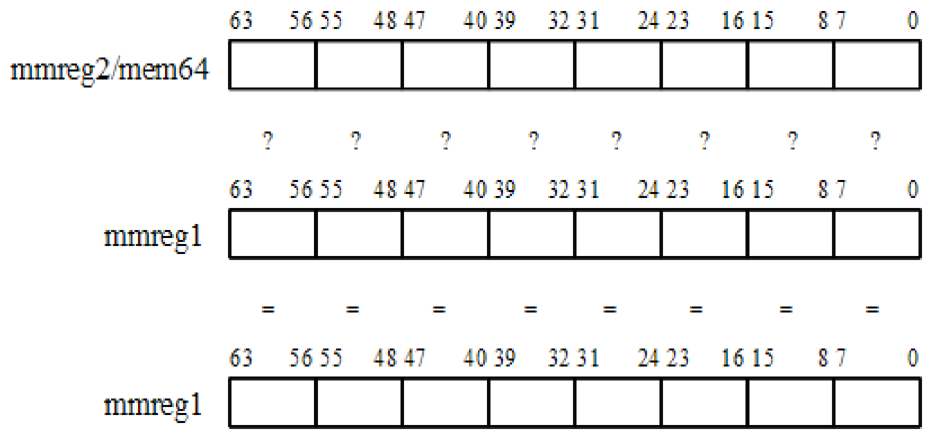
Just like previously described but freed bits are filled with zeros.

6. Data comparison instructions

pcmpeqb mmreg1, mmreg2/mem64
This instruction compares eight 8-bit values from the first operand with the corresponding values from the second operand. If both compared values are identical the proper byte in the first operand is filled with 1s otherwise it is filled with 0s (see figure in next slide).

pcmpeqw mmreg1, mmreg2/mem64 – four 16-bit values,
pcmpeqd mmreg1, mmreg2/mem64 – two 32-bit values.

MMX instructions extension



pcmpeqb instruction

The following instructions operate similarly to previously described with the difference that the greater relation is tested (signed values). It means that resulting value is filled with 1s when the value from the first operand is greater than the one from the second operand. Otherwise it is filled with 0s.

pcmpgtb mmreg1, mmreg2/mem64 – eight 8-bit values,
pcmpgtw mmreg1, mmreg2/mem64 – four 16-bit values,
pcmpgtd mmreg1, mmreg2/mem64 – two 32-bit values.

Exemplary applications of MMX

Applications of MMX to image processing – elementary examples
In this part of the presentation we will indicate and explain several exemplary applications of MMX instructions to image processing.

1. Image representation
An image is usually represented by two-dimensional array of data points each representing pixel color directly or with use of color palette. Hence we have the following possibilities.

Image pixels can be described directly with three components: **R**ed, **G**reen and **B**lue. Here we have the following formats (see Fig. 6.3):
- 24 bits per pixel: 8-bit for each component RGB,
- 16 bpp: 5-6-5 bits for RGB respectively.

There are also bitmap formats with color palettes where pixel data does not carry directly the information about pixel color but it is an index into the array (palette) of RGB triples (palette entries) (see Fig. 6.4). See below list:
- 8 bits per pixel: 256 entries in color palette,
- 4 bpp: 16 colors,
- 2 bpp: 4 colors,
- 1 bpp: monochromatic.

Exemplary applications of MMX

Pixel data carries directly information about its color as the triple of Red, Green and Blue components. Hence this information can be directly passed to the analog to digital converter (DAC) and next to the graphic array analog output. With 8 bits for each of RGB components we have about 16 millions of colors to be displayed.

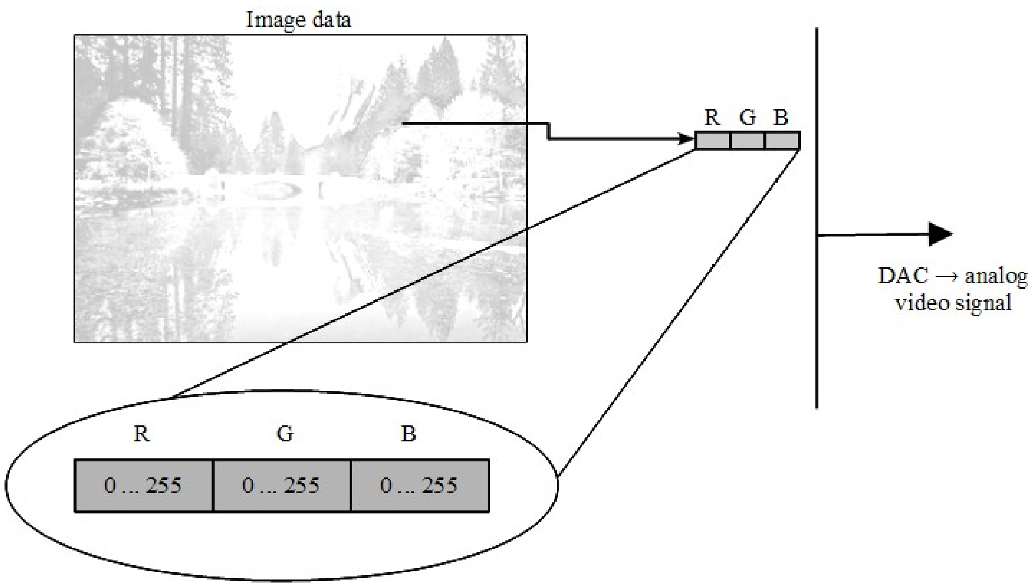


Fig. 6.3. Image representation without color palette

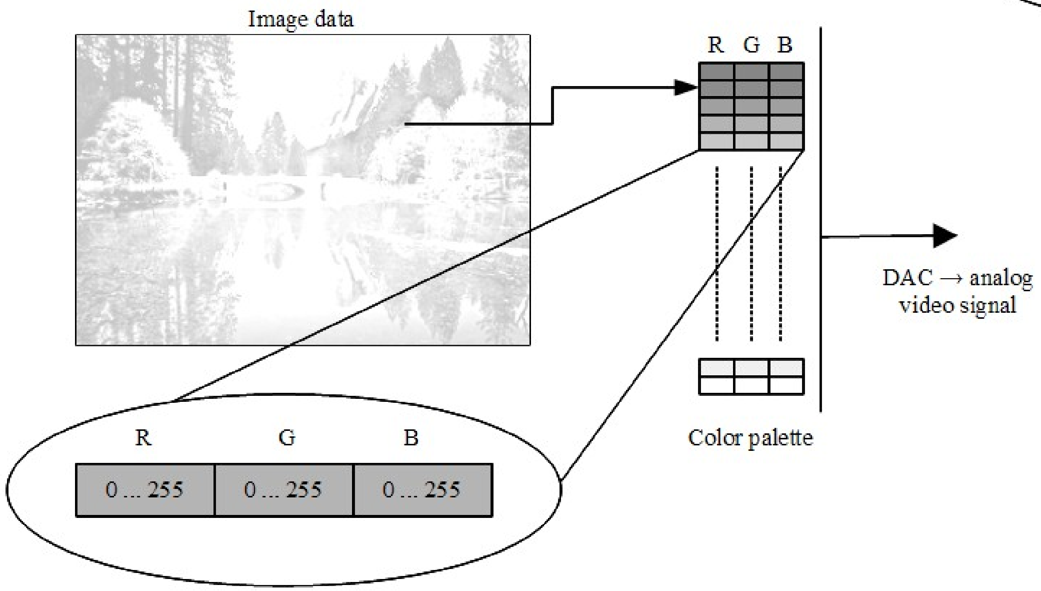


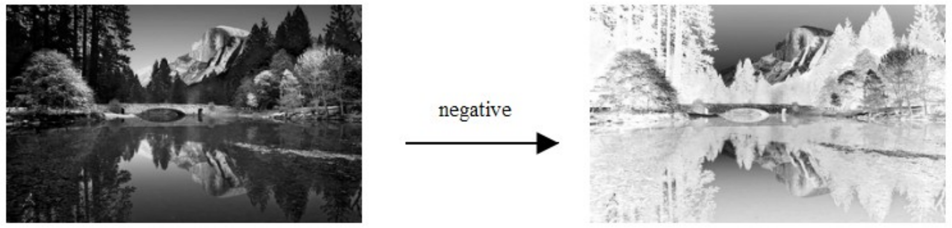
Fig. 6.4. Image representation with color palette

For palette images pixel color RGB components are taken from special table (palette). Index into that table is hold by pixel data. Hence pixel data is not directly passed to DAC but indirectly with use of palette table.

Exemplary applications of MMX

In presented exemplary applications we operate on grayscale images with 256-entry palette, colors ordered in ascending index order. Hence presented codes can be easily applied to color 24-bit images to each of RGB component layers separately.

2. Negative of an image
This is the simplest operation on image.
It is enough to apply **not** operation to each of pixel data.



```
.data
    (...)
    neg_mask    dq  -1
    (...)

.code
        (...)
        mov ebx, buffer_with_pixel_data
        mov ecx, (number_of_pixels)/8
        movq mm1, neg_mask
n0:     movq mm0, [ebx]
        pandn mm0, mm1
        movq [ebx], mm0
        add ebx, 8
        loop n0
        (...)
end
```

Here we have some exemplary code. As long as there is no not operation alone we use pandn instruction with mask composed of 1s.

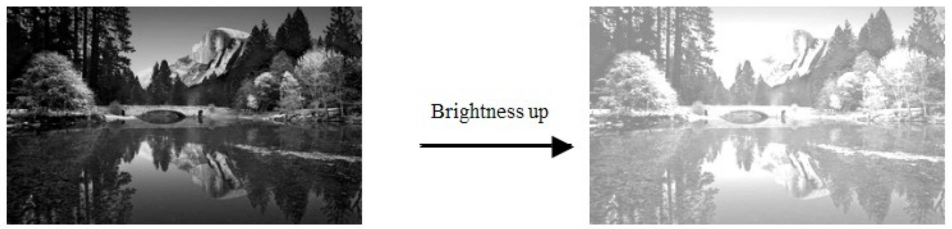
With this simple operation the advantage of using MMX instructions is not so evident. The negative can be also executed pretty effectively with the aid of general 32-bit registers where we would operate on 4 pixels at the same time.

In this case MMX would give up only to twofold reduction of computational time.

Exemplary applications of MMX

3. Change of image brightness

This operation is also very simple and requires addition (subtraction) of some value from image pixel data points.



The following code performs brightness increase operation.

```
.data
    (...)
    bup_val db 8 dup(32)
    (...)

.code
        (...)
        mov ebx, buffer_with_pixel_data
        mov ecx, (number_of_pixels)/8
        movq mm1, bup_val
bu0:    movq mm0, [ebx]
        paddusb mm0, mm1
        movq [ebx], mm0
        add ebx, 8
        loop n0
        (...)
end
```

In this case we take advantage of paddusb instruction which performs addition bytewise with saturation.

It should be noted that although brightness increase operation is very simple the advantage of MMX instructions is evident. Their strength lays in saturation feature which while programming with basic instructions would be replaced with cmp instruction and all the calculations would have to be performed bytewise, not on 8 bytes as with MMX.

Hence the reduction of calculations is up to eight times in this case.

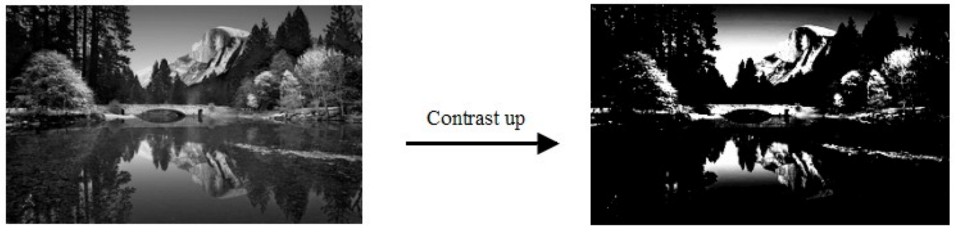
For brightness decrease we would use psubusb instead.

Exemplary applications of MMX

4. Change of image contrast

In this case all pixels must be multiplied by some value (in our example 2). However not to change the image brightness we must also subtract from each pixel the mean value of all pixels (when we multiply by 2, in general case the mean scaled by (a-1) where a is the contrast multiplier).

```
.data
        (...)
        cup_v0 dw 4 dup(2)
        cup_v1 dq 0
        cup_v2 dq 0
        (...)
.code
        (...)
        mov ebx, buf
        mov ecx, number_of_pixels
        xor eax, eax
        xor edx, edx
cu0:    mov dl, [ebx]
        add eax, edx
        inc ebx
        loop cu0
        xor edx, edx
        mov ecx, number_of_pixels
        div ecx
        xor edi, edi
        mov ecx, 4
        xor edi, edi
cu1:    mov [cup_v1+edi], ax
        add edi, 2
        loop cu1
        mov ebx, buf
        mov ecx, (number_of_pixels)/8
```



```
        movq mm7, cup_v2
        movq mm6, cup_v0
        movq mm5, cup_v1
cu2:    movq mm0, [ebx]
        movq mm1, mm0
        punpckhbw mm0, mm7
        punpcklbw mm1, mm7
        pmullw mm0, mm6
        pmullw mm1, mm6
        psubusw mm0, mm5
        psubusw mm1, mm5
        packuswb mm1, mm0
        movq [ebx], mm1
        add ebx, 8
        loop cu2
        (...)
end
```

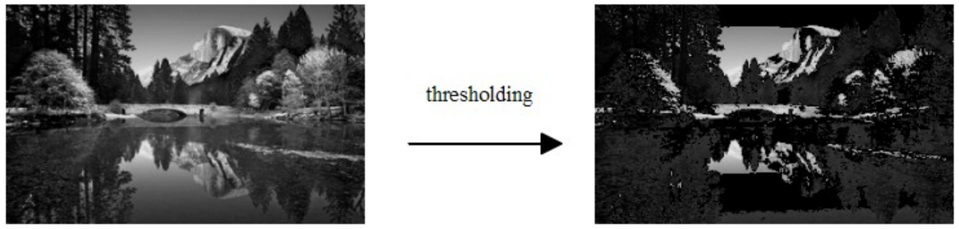
In this example the mean value of all pixels was calculated with the aid of standard instructions. The first part of presented code is devoted to this operation.

There are also some difficulties due to the fact that there is no bitwise multiplication in the set of MMX instructions.

Exemplary applications of MMX

5. Image thresholding

In thresholding simply pixel values lying below some assumed value are zeroed.



```
.data
        (...)
        thr_v0 db 8 dup(0)
        thr_v1 dq -1
        (...)
.code
        (...)
        mov ebx, buf
        mov ecx, (number_of_pixels)/8
        movq mm3, thr_v1
        movq mm2, thr_v0
tu0:    movq mm0, [ebx]
        movq mm1, mm0
        pcmpgtb mm1, mm2
        pandn mm1, mm3
        pand mm0, mm1
        movq [ebx], mm0
        add ebx, 8
        loop tu0
        (...)
end
```

In this example masking with pand instruction is performed. The mask for that operation is calculated with the aid of comparison pcmpgtb instruction and than negated.