

B35APO: Computer Architectures

Lecture 11. x86 Architecture

Pavel Píša

pisa@fel.cvut.cz

Petr Štěpán

stepan@fel.cvut.cz



18. června, 2025

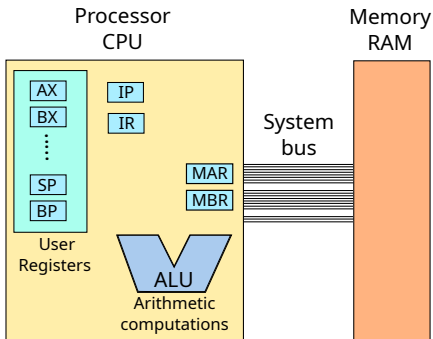
Outline

- 1 x86 History
- 2 x86 – Instruction Set
- 3 Floating-Point Unit FPU – x87
- 4 Multimedia/SIMD x86 Extensions – MMX
- 5 x86 SSE Extension

Processor – CPU

Basic features:

- data and address bus width
- number and size of internal registers
- control signal rate – frequency
- instruction set



x86/AMD64 – History

- x86 - CPU family, x is placeholder for x values - 0,1,2,3,4,5,6

8086 – R16 A20 (1978) the first IBM PC (8088 - 1979)

80286 – R16 A24 (1982) protected mode

80386 – R32 A32 (1985) paging

80486 – R32 A32 (1989) pipelining, FPU, cache

80586 – R32 A32 (1993) Pentium superscalar

- More detailed list –

https://en.wikibooks.org/wiki/X86_Assembly

80686 – R32 A36 (1995) Pentium Pro PAE, L2 cache, out-of-order & speculative exec

IA-64 – R64 A52 (2001) Itanium 64-bit version

AMD64 – R64 A40 (2003)

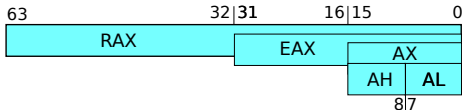
Athlon 64-bit version from AMD

Core2 – R64 A36 (2006) Intel 64 EM64T, SSSE3, μ op, virtualization

x86/AMD64 – Registers

User/General Purpose Registers

- All registers are 64/32/16/8 bit for backward compatibility
- integer registers for storing program values `eax`, `ebx`, `ecx`, `edx`
- specialized registers as memory pointers `esi`, `edi`, `ebp`
- `esp` – stack pointer - more details below
- AMD64/EM64T adds 8 additional registers `r8-r15`, in the form of `r8b` lowest byte, `r8w` lowest word (16 bits), `r8d` – lower 32 bits, `r8` – 64 bit register

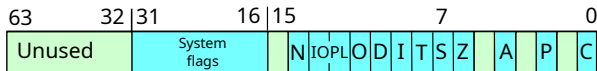


Control and Status Registers

- `IP/EIP/RIP` – instruction pointer – points to the current executed instruction
- `FLAGS/EFLAGS/RFLAGS` – program status word/flags registers

x86/AMD64 – FLAGS Register

RFLAGS register



C – Carry flag

P – Parity flag

Z – Zero flag

S – Sign flag

O – Overflow flag

A – Auxiliary flag (BCD)

I – Interrupt enable

T – Trap flag

IOPL – I/O privilege level

System Flags:

- VM – Virtual 8086 Mode
- VIF – Virtual Interrupt Flag
- VIP – Virtual Interrupt Pending

x86/AMD64 – Operating Modes

FLAGS register

- Only two protection rings/modes used – the base of the hardware protections
 - CPL0¹ = privileged (system) mode
 - processor can access hardware (I/O ports), manipulate paging state etc.
 - CPL3 = user (application) mode
 - privileged operations are not permitted
- Privileged operations
 - manipulation with system/core state (halt, reset, Interrupt Enable/Disable, whole Flags modifications, MMU registers changes and control instructions)
 - I/O port access instructions (in, out)
- Switching between modes and privileges
 - The old 16 real mode after system reset, start at address 0xffff:0000
 - Complex sequence to switch into 64-bit mode at CPL0 (system)
 - Switch to user (CPL3) mode – Flags registers change (popf or reti)
 - Switch to system (CPL0) mode – only by interrupt, including synchronous by int instruction

¹Current privilege level

Outline

- 1 x86 History
- 2 x86 – Instruction Set**
- 3 Floating-Point Unit FPU – x87
- 4 Multimedia/SIMD x86 Extensions – MMX
- 5 x86 SSE Extension

x86/AMD64 Instructions

Data transfers register to register and immediate to register
(Two different syntaxes are commonly used for x86 assembler code.)

AT&T

`movq source 64b, dest`

`movl source 32b, dest`

`movw source 16b, dest`

`movb source 8b, dest`

registers

`%ax`

immediate `$`, hex `0x`

Intel

`mov dest, source`

only `ax`

number, hex postfix `h`

`movl $0xff, %ebx` `mov ebx, 0ffh`

If the source width is n-ener than destination, two variants exists:

- `movsX` – sign extension, the MSB is expanded to additional bits
- `movzX` – zero extension, the destination additional bits are zeroed

x86/AMD64 – Instructions

Data transfers to/from memory (memory pointers / register indirect)

AT&T

```
movl (%ecx),%eax
```

```
movl 3(%ebx), %eax
```

```
movl (%ebx, %ecx, 0x2), %eax
```

```
movl -0x20(%ebx, %ecx, 0x4), %eax
```

Intel

```
mov eax, [ecx]
```

```
mov eax, [ebx+3]
```

```
mov eax, [ebx+ecx*2h]
```

```
mov eax, [ebx+ecx*4h-20h]
```

- effective address has 4 components: $base + index * scale + offset$
- the *scale* possible values are 1,2,4,8
- directly maps to array of structures access: *base* points to array start, $index * scale$ specifies which entry/index and *offset*, which structure field is accessed.
- it is not required that all four address components are used each time

x86/AMD64 – Repeat Prefix and String Operations

Instructions for string/bloc operations - REP prefix for repeat over array

- repeat while `ecx>0`:
 - operations `(%esi), (%edi)`
 - `esi += d*operand_size`
 - `edi += d*operand_size`
 - `ecx --`
- string operations `movs, cmps, lods, stos, scas, ins, outs`
- `d` - direction specification values `+1`, or `-1`
- REP repeat `ecx` specified times
- REPE/REPNE repeat max `ecx` times while condition is true
 - operation `cmps` stops if there is/not difference between `[edi]` and `[esi]` memory locations
 - operation `scas` stops if there is/not difference between `[edi]` and `eax` register value

x86/AMD64 – Repeat Example – Fill Array

Set each array element to the -1 value:

```
int array[128];
for (int i=0; i<128; i++) {
    array[i]=-1;
}
```

translated:

```
mov    array, %edi    ; Set edi register to points to the array start
mov    $128, %ecx    ; Setup number of repetitions
mov    $-1, %eax     ; Set the value to be stored
rep    stosd         ; Fill the whole array
```

x86/AMD64 – Zero Terminated String Length

Find end of the zero terminated string:

```
char str[128];  
int i;  
for (i=0; i<128; i++) {  
    if (str[i]==0)  
        break;  
}
```

translated:

```
mov    str, %edi    ; Set EDI to point to start of the string  
mov    $128, %ecx   ; Setup maximal number of repetitions  
mov    $0, %eax     ; Set register for the search value (0)  
repne scasb        ; Scan the str until 0 is located
```

x86/AMD64 – Arithmetic Operations

Arithmetic – AT&T syntax

the *X* placeholder for next operations allowed values are b, w, l, q

The source (*src*) is applied through specified operation to the value at destination (*dst*) and result is stored to the same destination

<code>addq \$0x05,%rax</code>	<code>rax = rax + 5</code>
<code>subl -4(%ebp), %eax</code>	<code>eax = eax - mem(ebp-4)</code>
<code>subl %eax, -4(%ebp)</code>	<code>mem(ebp-4) = mem(ebp-4)-eax</code>
<code>andX src, dst</code>	bitwise and
<code>orX src, dst</code>	bitwise or
<code>xorX src, dst</code>	bitwise xor (fast register clear)
<code>mulX multiplier</code>	multiply <code>eax</code> by unsigned value → <code>edx:eax</code>
<code>divX divisor</code>	divide <code>edx:eax</code> by unsigned value
<code>imulX multiplier</code>	multiply <code>eax</code> by signed value → <code>edx:eax</code>
<code>idivX divisor</code>	divide <code>edx:eax</code> by signed value

x86/AMD64 – Single Operand Functions

Arithmetic with single operand – AT&T syntax
operation

<code>incl %eax</code>	<code>eax = eax + 1</code>
<code>decw (%ebx)</code>	<code>mem(ebx) = mem(ebx)-1</code>
<code>shlb \$3, %al</code>	<code>al = al<<3</code>
<code>shrb \$1, %bl</code>	<code>bl=11000000, po bl=01100000</code>
<code>sarb \$1, %bl</code>	<code>bl=11000000, po bl=11100000</code>
<code>rorX, rolX</code>	rotate left/right by n bits.
<code>rcrX, rclX</code>	rotate left/right by n bits with C – carry flag

x86/AMD64 – Flow Control

Conditional branches

`test a1, a2` `tmp = a1 AND a2, Z tmp=0, C tmp<0`

`cmp a1, a2` `tmp = a1-a2, Z tmp=0, C tmp<0`

next `jump/branch` instructions can be used then

`jmp taget` unconditional jump, equivalent to `%eip=taget`

`je taget` `jmp equal – jump when equal (Z=1)`

`jne taget` `jmp not equal – jump when not equal (Z=0)`

`jg/ja taget` `jmp greater/above – jump when $a1 > a2$ (sign/unsig)`

`jge/jae taget` `jump if $a1 \geq a2$ (sign/unsig)`

`jl/jb taget` `jmp less/below – jump when $a1 < a2$ (sign/unsig)`

`jle/jbe taget` `jump if $a1 \leq a2$ (sign/unsig)`

`jz/jnz taget` `jump if Z=1/0`

`jo/jno taget` `jump when O (overflow) = 1/0`

Quiz

Is the difference for program sizes for CISC (x86) and RISC (RISC-V)?

- A no difference, approximately the same
- B CISC programs are longer
- C RISC programs are longer

Comparison of CISC vs. RISC Program

CISC programs are usually shorter if compressed (16-bit) encoding is not used on RISC side:

```
incl 10(%ecx)  -      lw  t2, 10(t1)
                  addi t2, t2, 1
                  sw  t2, 10(t1)
```

```
rep movs      - l1:  lw  t3, 0(t1)
                  sw  t3, 0(t2)
                  addi t1, t1, 4
                  addi t2, t2, 4
                  addi t4, t4, -1
                  bne  t4, zero, l1
```

On the other hand, larger number of GPR registers and return address storage in the register instead on the stack leads to lower number of accesses to memory which is advantage even if memory is cached.

Quiz

Are CISC (x86) programs faster than RISC (RISC V) programs?

- A Yes
- B No
- C It depends
- D Every today superscalar high performance architecture uses RISC style instructions processing internally

Stack for Calls, Arguments, Save and Local Variables

Stack:

- LIFO data structure

Implementation:

- dedicated *SP* register - points to the stack top
- before every push (data save) operation, *SP* is decremented by operand size, for every pop (register restore) operation, the *SP* is incremented by same size

<code>pushl %eax</code>	save <code>eax</code> onto stack top
<code>popw %bx</code>	restore 16-bit <code>bx</code> register and increments <i>SP</i> by 2
<code>pushf/popf</code>	save/restore <code>EFLAGS</code> register
<code>pusha/popa</code>	save/restore all user registers

- push operation saves data into stack
 - `sub $size, sp`
 - `mov %eax, 0(sp)`
- pop restores data from the stack
 - `mov 0(sp), %eax`
 - `add $size, sp`
- it is possible to access stack with offsets same as for RISC-V:

- `sub $16, sp`
- `mov %eax, 0(sp)`
- `mov %ebx, 4(sp)`
- `mov %ecx, 8(sp)`
- `mov %edx, 12(sp)`

Quiz

Which program is executed faster by the superscalar architecture implementation:

A

```
push %eax
push %ebx
push %ecx
push %edx
```

B

```
sub $16, sp
mov %eax, 0(sp)
mov %ebx, 4(sp)
mov %ecx, 8(sp)
mov %edx, 12(sp)
```

- A Both take similar time
- B A is faster to execute than B
- C B is faster to execute than A
- D Cannot be determined

x86 Function Calling Convention for 32-bit "cdecl"

- The caller saves all parameters on the stack
- The order of saving on the stack is from the last parameter to the first

function calling

`call adr` equivalent to `push %eip, jmp adr`

return from the function

`ret` equivalent to `pop %eip`

- The function return/result value will be stored in the `eax` register.
- The registers `ebp`, `ebx`, `esi`, `edi` must be preserved by the function. If the function has use for them, the original value must be saved on the stack.

x86 Function Calling – Function Stack Frame

The `ebp` register points to the stack where previous `ebp` of caller function was stored

Function prologue example

```
push  %ebp           ; Caller EBP value is stored to stack
mov   %esp, %ebp    ; Remember actual stack position in EBP
sub   $12, %esp     ; Reserve 12 bytes for local variables,
                   ; registers save and or call arguemnts
```

The first local variable or save would be at $-4(\%ebp)$, next $-8(\%ebp)$

The first argument at address $8(\%ebp)$, next $12(\%ebp)$

Function epilogue – finalization:

```
mov   %ebp, %esp    ; Restore stack top to the original position
pop   %ebp          ; Restore original EBP register value
ret   ; Return from the function
```

The cobined instruction for restore stack state at return: `leave` is equivalent for:

```
mov   %ebp, %esp
pop   %ebp
```

x86 Function Calling – Example

Function calling:

```
t = addfour(1, 2, 3, 4);
```

Compiled x86 machine code

```
push $4
push $3
push $2
push $1
call 10e2 <addfour>
add $0x10, %esp
```

Function prologue

```
push %ebp
mov %esp, %ebp
push %ebx
sub $0x10, %esp
```

Access to local variables:

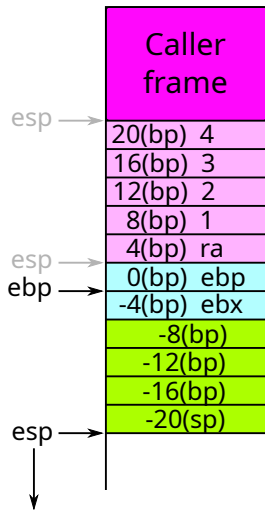
```
mov %eax, -4(%ebp)
```

The first and the second argument access:

```
mov 0x8(%ebp), %edx
mov 0xc(%ebp), %eax
```

Function epilogue – finalization:

```
leave
ret
```



AMD64 Function Calling Convention – ELF/Linux

- Up to 6 integer arguments are passed in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` and or up to 8 floating point arguments are passed in `ymm0-7`
- The rest of the arguments is passed by stack. The `al` is required to hold number of `ymmX` used for variadic functions or functions without prototype calls.
- The function's return value will be stored in the `rax` and `rdx` registers.
- The `rbp`, `rbx`, `r12-r15` registers must be preserved by the function. If the function wants to use them, the original value must be saved to the stack.
- Red Zone – a zone 128 bytes from the `rsp` pointer that must not be changed by the interrupt handler. This zone allows this memory to be used for temporary variables without moving the `rsp` pointer. Of course, calling the function changes this zone.

Stack Purposes

Stack:

- parameters for the function (current and called)
- return address, next instruction to continue after function call is finished
- local variables of the function
 - the stack is usually small
 - limited size of local variables
 - be careful with recursion - it is better to avoid recursion

Quiz

Consider program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Where is allocated space for `i` local variable?

- A On stack for both RISC-V and x86.
- B Allocated from heap on RISC-V and on the stack for x86.
- C In the `.data` or `.bss` section for both RISC-V and x86.
- D In the register for both RISC-V and x86.

Quiz

Consider program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Location for y variable when compiled without optimization?

- A On the stack or in the register for both RISC-V and x86.
- B Dynamically allocates on heap for both RISC-V and x86
- C In the .data or .bss section for both RISC-V and x86.
- D cannot be determined for RISC-V neither x86.

Quiz

Consider program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Location of x variable for compilation without optimization?

- A RISC-V and x86: on stack
- B RISC-V: in register; x86: on stack
- C RISC-V: in register and then on stack; x86: on stack
- D RISC-V: on stack; x86: in register
- E RISC-V: on stack; x86: in register and then on stack

Quiz

Consider program:

```

int factorial (int x) {
    int y;
    if (x<=1) {
        return 1;
    } else {
        y = factorial (x-1)
        return y*x;
    }
}

int i;
int main() {
    i=10;
    i=factorial (i);
    return i;
}

```

Where is stored return address for return from the `factorial` function?

- A RISC-V and x86: on stack
- B RISC-V: in register; x86: on stack
- C RISC-V: in register and then on stack; x86: on stack
- D RISC-V: on stack; x86: in register
- E RISC-V: on stack; x86: in register and then on stack

x86/AMD64 Instruction Set

The complexity of the assembler

- An algorithm can be translated into assembler in various ways
- Machine translation is sometimes very confusing
 - e.g. `mov 0x12345, %esi; mov %esi, %ebx` instead of `mov 0x12345, %ebx`
- Different methods work at different speeds and are of different lengths and are different in clarity
 - `xor %ebx, %ebx` is the same as `mov $0, %ebx`
 - `lea address, register` – load effective address – sets the value of computed effective address to the specified register
 - `lea -12(%esp), %esp` is the same as `sub $12, %esp`
 - `lea` has more advances when used, it does not block the ALU unit usually (however, for example, Atom has address computation slower than the ALU).

Outline

- 1 x86 History
- 2 x86 – Instruction Set
- 3 Floating-Point Unit FPU – x87**
- 4 Multimedia/SIMD x86 Extensions – MMX
- 5 x86 SSE Extension

FPU Coprocessor – x87

Special part of the processor for computation with real numbers

- Supports single-32, double-64, extended-80 and exotic BCD formats
- Contains 8 own registers of 80 bits each
- Registers are organized in a stack (push, pop), but also allow direct access (0-7) relative to the stack top
- Each operation works with the top of the stack and one other register, or value
- Originally a separate processor, since 486 on-die – on a single chip
- Supports all IEEE-754 operations:
 - fadd, fsub, fmul, fdiv, fsqrt, fcmp, fsin, ...

x87 FPU Operations – Load/Store

Basic operations are used to load/store a real number from/to registers:

- fld - loads a value from memory to the register stack – push
- fst - stores a value from a register to memory without pop
- fstp - stores a value from a register to memory and pops

Basic operations for storing an integer from/to registers:

- fild - loads an integer from memory to the register stack – push
- fist - stores an integer from a register to memory without pop
- fistp - stores an integer from a register to memory and pops
- fisttp - stores a rounded integer from a register to memory and pops

x87 FPU Operations – Computations

The addition operation is used as an example (other operations have the same form, ST(0) is the top of the stack, ST(1) the value below it, etc.):

- `fadd float/double` - add the contents of memory to ST(0) and store the result in ST(0)
- `fiadd short/int` - add an integer from memory to ST(0) and store the result in ST(0)
- `fadd ST(0), ST(i)` - add the contents of ST(0) and ST(i) and store the result in ST(0)
- `fadd ST(i), ST(0)` - add the contents of ST(i) and ST(0) and store the result in ST(i)
- `faddp ST(i), ST(0)` - add the contents of ST(i) and ST(0) and store the result in ST(i) and perform a pop operation (delete the value of ST(0))
- `faddp` - add the contents of ST(1) and ST(0) and store the result in ST(1) and perform a pop operation (delete the value of ST(0))

x87 FPU Operations – More Computational Instructions

The SUB and DIV operations also have a reverse form, i.e., the order of the operands is reversed (in all versions, both with memory and with registers):

- fsub ST(0), ST(i) - the result of ST(0) - store ST(i) in ST(0)
- fsubr ST(0), ST(i) - the result of ST(i) - store ST(0) in ST(0)

Unary functions sin, cos:

- fsin/fcos - replace ST(0) with the value sin/cos(ST(0))

Logarithm - calculation $y \cdot \log_2 x$:

- fyl2x - replace ST(1) with the value $ST(1) * (\log_2 ST(0))$ and do a pop

Loading constants:

- fldz/fld1 - loads 0.0/1.0 on the stack
- fldpi/fldl2e - loads $\pi/\log_2 e$ on the stack

x87 FPU Code Example

Example of the expression computation $1.1 * 2.2 + \sin(3.3)$:

```
fldl    adr_1.1 ; Load the first operand
fmull   adr_2.2 ; Multiply it with the second one
fldl    adr_3.3 ; Load the third operand
fsinl                   ; Compute sine value from the third operand
faddp                   ; Add two registers and store result on the stack
fstp    dst_addr ; Store result in memory
```

Floating-Point Unit FPU – RISC-V

- Two extensions RV64F – float, RV64D – double
- 32 internal registers, either 32 bit wide for float only or 64 bits for float and double support
- New load and store instructions – flw, fsw (fld, fsd)
- New instructions for operations:
 - fadd.s, fsub.s, fmul.s, fdiv.s (*.d for double)
 - fadd.s $F[rd]=F[rs1]+F[rs2]$
 - fsqrt.s – square root – $F[rd] = \text{sqrt}(F[rs1])$
 - fmadd.s – multiply and add/accumulate, $F[rd]=F[rs1]*F[rs2]+F[rs3]$
 - fmsub.s – multiply and subtract, $F[rd]=F[rs1]*F[rs2]-F[rs3]$
 - fmin.s – $F[rd] = (F[rs1]<F[rs2]) ? F[rs1] : F[rs2]$
 - operations for conversion between integer values and float, double

Outline

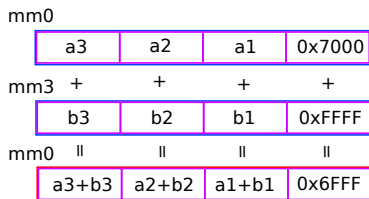
- 1 x86 History
- 2 x86 – Instruction Set
- 3 Floating-Point Unit FPU – x87
- 4 Multimedia/SIMD x86 Extensions – MMX**
- 5 x86 SSE Extension

SIMD - MMX

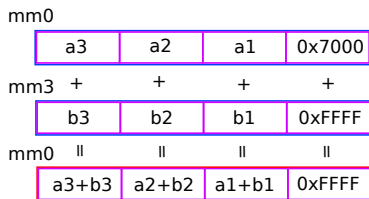
- SIMD - Single Instruction Multiple Data - execution of one type of instruction on multiple data at once
- MMX - MultiMedia eXtension (sometimes explained as Multiple Math eXtension)
- They use the same registers as the x87 FPU, so they cannot be used simultaneously
- A 64-bit register can operate in the following modes:
 - B - 8× byte
 - W - 4× short int
 - D - 2× int
- Operations:
 - Arithmetic - addition, subtraction, multiplication
 - Logical - and, or, rotation, comparison
 - Conversion - pack, transfers between registers

MMX Operations – Parallel Additions

PADDW - add packed word (4 × 16-bit) integers



PADDUSW - saturated addition (limit instead of (modulo) overflow)



MMX Operations – Multiply

PMADDWD - packed multiply and paired add

mm0

a3	a2	a1	a0
----	----	----	----

mm3 + + + +

b3	b2	b1	b0
----	----	----	----

mm0 || ||

a2*b2+a3*b3		a0*b0+a1*b1	
-------------	--	-------------	--

PMULLW - multiply store low
word of the result

mm0

a3	a2	a1	a0
----	----	----	----

mm3 + + + +

b3	b2	b1	b0
----	----	----	----

mm0 || || || ||

(a3*b3) &0xFFFF	(a2*b2) &0xFFFF	(a1*b1) &0xFFFF	(a0*b0) &0xFFFF
--------------------	--------------------	--------------------	--------------------

PMULHW - multiply store high
word of the result

mm0

a3	a2	a1	a0
----	----	----	----

mm3 + + + +

b3	b2	b1	b0
----	----	----	----

mm0 || || || ||

(a3*b3) >>16	(a2*b2) >>16	(a1*b1) >>16	(a0*b0) >>16
-----------------	-----------------	-----------------	-----------------

MMX – Example

Combine masked image with background:

```
unsigned char mask[size],
  img1[size ], img2[size ];
if (mask[i]==0) {
  new_img[i] = img1[i];
} else {
  new_img[i] = img2[i];
}
```

MMX implementation for 8 8-bit pixels in once

```
movq  mask_ptr, %mm0
pcmpeqb %mm0, 0
movq  %mm0, %mm1
pand  %mm1, obr1_ptr
pandn %mm0, obr2_ptr
por   %mm0, %mm1
movq  %mm0, new_img_ptr
```

3Dnow! Extension of MMX

- The 3Dnow! extension added real number operations to the existing registers `mm0-mm7`.
- Only allows pack two 32-bit/single real numbers in one register
- Adds conversion of integers to real numbers and back, also using averaging of 8-bit and 16-bit integers
- Addition, subtraction, multiplication, division of packed (pair of) real numbers
- Comparing real numbers and finding minima and maxima

Outline

- 1 x86 History
- 2 x86 – Instruction Set
- 3 Floating-Point Unit FPU – x87
- 4 Multimedia/SIMD x86 Extensions – MMX
- 5 x86 SSE Extension**

x86 SSE Extension – SIMD

- SSE - Streaming SIMD Extension
- new registers `xmm0-xmm7`
- 128-bit wide each, next packed types can be used
- 4× float - 32-bit floating point number
- 2× double - 64-bit floating point number
- extension of MMX integer operations for 128-bit SSE registers

x86 SSE Instructions

- Operations: packed suffix -ps, scalar suffix -ss
- Load/store from/to memory: mov
- Floating point math operations: add, sub, mul, div, rcp, sqrt, max, min, rsqrt
- Logic operations on binary representation: and, or, xor, andn
- Compare with mask set as the result: cmp, comi, ucomi
- Scalar operations: addss, subss, mulss, divss

Further x86 SSE Extensions

Extension evolution in chronological order:

- SSE2 - 144 new instructions added
- SSE3 - 13 yet additional intructions
- SSSE3 - 16 additional intructions
- SSE4 - 47 additional intructions
- SSE4.2 - 170 additional intructions
- AVX - Advanced Vector Extensions
- AVX2 - widen to 256 bits (YMMx registers)
- AVX-512 - 512 bits support (ZMMx registers)