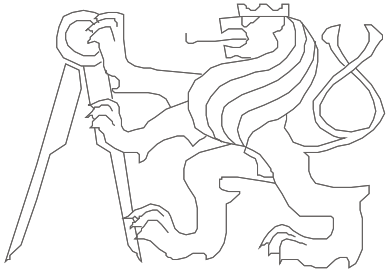
Computer Architectures

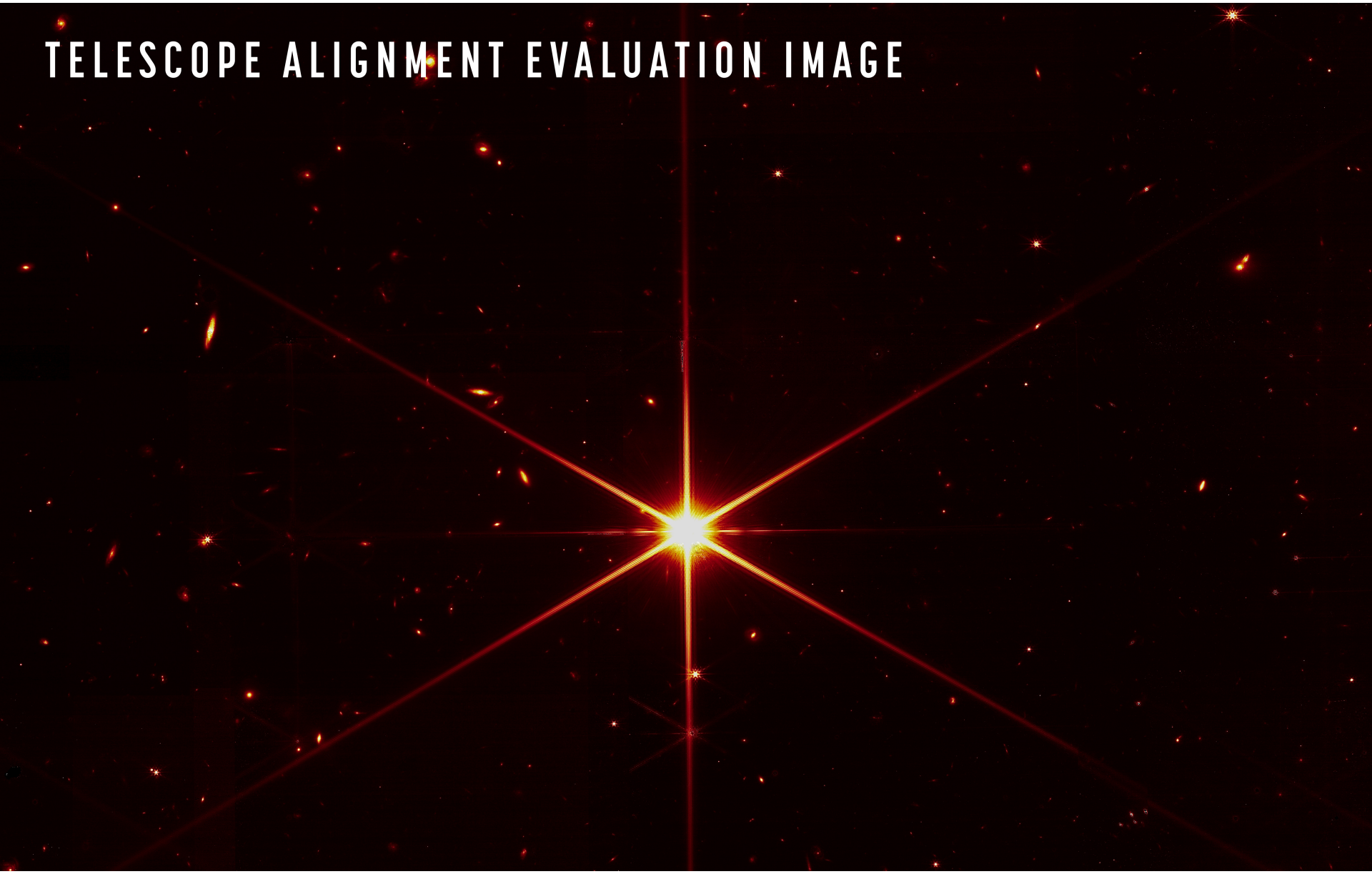
CISC – x86

Petr Stěpán, Pavel Píša



Czech Technical University in Prague, Faculty of Electrical Engineering

TELESCOPE ALIGNMENT EVALUATION IMAGE



History of x86 cpu family

8086 – A16 F20 (1978) first IBM PC (8088 - 1979)

80286 – A16 F24 (1982) protected mode

80386 – A32 F32 (1985) paging

80486 – A32 F32 (1989) pipelining, FPU, cache

80586 – A32 F32 (1993) Pentium superscalar

80686 – A32 F36 (1995) Pentium Pro PAE, L2 cache, out-of-order & speculative exec

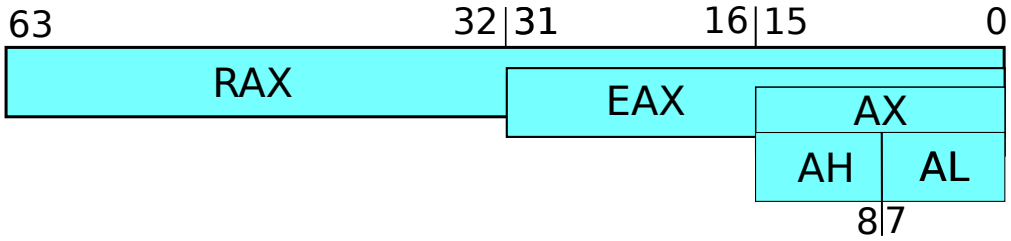
IA-64 – A52 F52 (2001) Itanium 64-bit version

AMD64 – A40 F40 (2003) Athlonn 64-bit version from AMD

Core2 – A36 F36 (2006) Intel 64 EM64T, SSSE3, μ op, virtualization

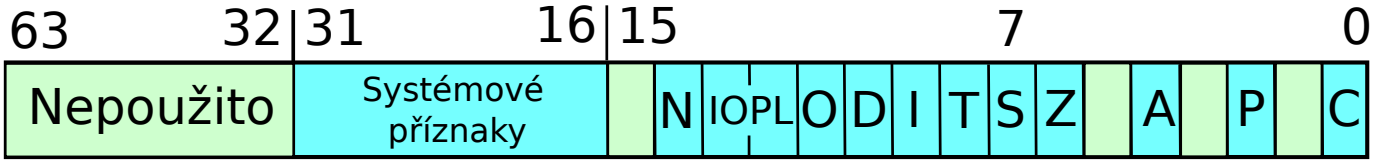
User Registers - x86/AMD64/EM64T

- All registers are 64/32/16/8 bits with respect to backward compatibility
- General user registers eax, ebx, ecx, edx
- Registers – pointers into memory esi, edi, ebp
- Stack pointer – esp, more details follows
- AMD64/EM64T add 8 general registers r8-r15, r8b – lowest byte, r8w lowest word (16 bits), r8d – lowest double word (32 bits), r8 – 64 bits register



- Control and status registers
- IP/EIP/RIP – instruction pointer – address of executed instruction
- FLAGS/EFLAGS/RFLAGS – status of processor

Register FLAGS



- Result of numeric operations
  - C -- Carry flag
  - P -- Parity flag
  - Z -- Zero flag
  - S -- Sign flag
  - O -- Overflow flag
  - A -- Auxiliary flag (BCD)
- System flags
  - I -- Interrupt enable
  - T -- Trap flag
  - IOPL -- I/O privilege level
  - VM -- Virtual 8086 Mode
  - VIF -- Virtual Interrupt Flag
  - VIP -- Virtual Interrupt Pending

x86/AMD64 instruction

Two different assembler syntax:

AT&T	Intel
movq from, to 64bits	mov to, from
movl from, to 32bits	
movw from, to 16bits	
movb from, to 8bits	
Register naming:	
%ax	ax
Values:	
$0xff, $4	0ffh, 4

x86/AMD64 instruction

Reference to/from memory:

AT&T	Intel
movl (%ecx), %eax	mov eax, [ecx]
movl 3(%ebx), %eax	mov eax, [ebx+3]
movl (%ebx, %exc, 2), %eax	mov eax, [ebx+ecx*2]
movl -0x20(%ebx, %ecx, 4), %eax	mov eax, [ebx+ecx*4-020h]

General reference to memory can have 4 items:
base+index*scale+shift
Scale can have value only 1,2,4,8

Can be used as index into array of structures:
Base – start of the array
index*scale – select item from array
Shift – select item from structure

x86 instruction for strings

Instructions for strings – prefix REP, repeat for arrays of values

- Repeat until ecx>0:

  Op (%esi), (%edi)

  esi+=d*operand_size

  edi+=d*operand_size

  ecx--

- Operation can be movs, cmps, lods, stos, scas, ins, outs
- d is direction 1, or -1
- REP repeat until ecx>0
- REPE/REPNE repeat until ecx>0 and comparision fits/not fits
  - Operation cmps fits if (%edi)==(%esi)
  - Operation scas fits if (%edi)==%eax

x86 instruction for strings

- Example: set all members of array to -1:

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

- In assembly x86:

```
mov   array, %edi ; set edi to beginning of array
mov   $128,  %ecx ; set repeat count
mov   $-1,   %eax ; set value to store
rep stosd         ; fill whole array
```

x86 instruction for strings

- Find end of the string:

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

- In assembly x86:

```
mov   array, %edi ; set edi to beginning of array
mov   $128,  %ecx ; set repeat count
mov   $0,    %eax ; set value to store
rep scasb         ; scan str and find value 0
```

Instruction x86

Arithmethic operations:

Addq $0x05,%rax	rax = rax+5
Subl -4(%ebp), %eax	Eax = eax - mem[ebp-4]
Subl %eax, -4(%ebp)	mem[epb-4] = mem[ebp-4]-eax

Other operations (X define size of operands

andX	Bitwise and
orX	Bitwise or
xorX	Bitwise xor
mulX	Multiplication of numbers without sign
divX	Division of numbers without sign
imulX	Multiplication of numbers with sign
idivX	Division of numbers with sign

Instruction x86

Unary operations:

Incl %eax	eax++
Decw (%ebx)	mem[ebx]--
Shlb $3, %al	Al = al << 3
Shrb $1, %bl	bl=11000001 after op. bl=01100000
Sarb $1, %bl	bl=11000001 after op. bl=11100000
Rorb $1,%bl	bl=11000001 after op. bl=11100000
Rolb $1, %bl	bl=11000001 after op. bl=10000011
Rcrb $1, %bl	bl=11000001 after op. bl=C1100000, C=1
Rclb $1, %bl	bl=11000001 after op. bl=1000001C, C=1

Instruction x86

Before conditional jump:

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)

Conditional jump:

je	Jmp if test or cmp operands are equal
jne	Jmp if .. not equal
jg/ja	Jmp if a1>a2 (signed/unsigned version)
jge/jae	Jmp if a1>=a2 (signed/unsigned version)
jl/jb	Jmp if a1<a2 (signed/unsigned version)
jle/jbe	Jmp if a1<=a2 (signed/unsigned version)
jz/jnz	Jmp if Z flag is set/unset
jo/jno	Jmp if O flag (overflow) is set/unset
jc/jnc	Jmp if C flag is set/unset

CISC – RISC comparison

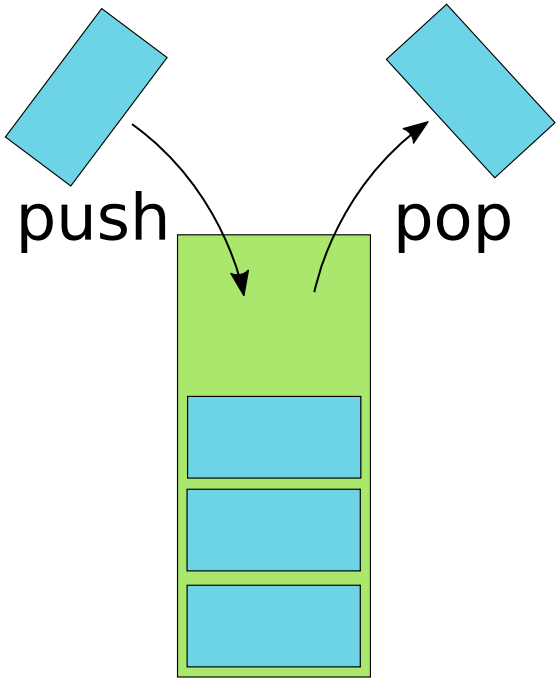
- CISC programs are smaller:

```
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
                             jne  t4, zero, l1
```

Stack

- Stack is a data structure:
- Data structure Last In First Out – LIFO
- Two operations:
  - Push – insert data into structure
  - Pop – remove data from structure



Stack in x86

- Stack implementation in x86:
  - Special register sp – stack pointer
  - Push – decrease sp by size of opernad and store value of operand to memory pointed by sp
  - Pop – read value from memory pointed by sp and increase sp by size of operand
- Instructions:

pushl %eax	Store vlaue of eax on stack
popw %bx	Read value of word from stack to bx
pushf	Store flags on stack
popf	Restore flags from stack
pusha	Store all user registers on stack
popa	Restore all user registers from stack

Stack in x86

- Stack usage:
  - Return address of function
    - instruction call = jmp + push eip;
    - Instruction ret = pop eip
  - Arguments of function
  - Local variable of functions
- Problems of stack:
  - Stack is limited
    - Limited size for local variables
    - Possible problems with recursion
    - Control of stack overflow

Stack in function implementation

- Function intro:

```
push %ebp        ; save old value EBP
mov  %esp, %ebp  ; set new value EBP to ESP
sub  $12, %esp   ; make space for local variables
```

- First local variable have address -4(%ebp), second -8(%ebp) ...
- First function parameter have address 8(%ebp), next 12(%ebp) ...
- Function end:

```
mov  %ebp, %esp  ; set ESP to old value
pop  %ebp        ; get old value of EBP
ret              ; return from function
```

- New instruction leave
  - mov %ebp, %esp
  - pop %ebp

Compiler optimization

- Compiler select fastest and shortes instructions
  - Xor %rbx, %rbx = movq $0, %rbx
    - Instruction xor is coded by two bytes, instruction movq contains 8 bytes of 0, the whole instruction has 10 bytes
    - Xor is better with respect to instruction cache
  - Lea – load effective address
    - Compute address in format base+index*scale+shift
    - Lea is not using ALU, it use special limited ALU in decode stage
    - Lea -12(%esp), %esp = sub $12, %esp
    - Lea is not using standard ALU, the result is computed in decode phase

FPU coprocessor - x87

Special coprocesor for floting point arithmetic:

- It supprots single-32 bits, double-64 bits, extended-80bits floating point numbers and also exotic formats like BCD
- It has its own 8 registers for 80 bits numbers
- Registers are organized like stack (push, pop), but it enables direct access (0-7)
- Every operation works with top of the stack and one another register or value
- Originally special chip, from 486 on-die – one big chip
- Supports all IEEE-754 operations:
- fadd, fsub, fmul, fdiv, fsqrt, fcmp, fsin, ...

FPU coprocessor - x87

Basic operations load and stores real values to/from FPU internal registers:

- fld – load real value from memory to register – push
- fst – store real value from register to memory
- fstp – store real value from register to memory and remove register from stack – pop

Basic operations load and stores integer values to/from FPU internal registers:

- fild - load integer value from memory to register – push
- fist - store integer value from register to memory
- fistp – store integer value from register to memory and remove register from stack – pop
- fisttp – store rounded value to integer from register to memory and remove register from stack – pop

FPU coprocessor - x87

Possible arguments of operations with real numbers will be demonstrated on addition (ST(0) is stack top, ST(1) is register under the stack top):

- fadd float/double – add value from memory to ST(0) and result save in ST(0)
- fiadd short/int – add integer value from memory to ST(0) and result save in ST(0)
- fadd ST(0), ST(i) – add values from ST(0) and ST(i) and result store into ST(0)
- fadd ST(i), ST(0) – add values from ST(0) and ST(i) and result store into ST(i)
- faddp ST(i), ST(0) – add values from ST(0) and ST(i) and result store into ST(i) and remove ST(0) from stack = do pop
- faddp – add values from ST(0) and ST(1) and result store into ST(1) and remove ST(0) from stack

FPU coprocessor - x87

Operations SUB a DIV have reverze form with different operand order:

- fsub ST(0), ST(i) – result of ST(0) - ST(i) save into ST(0)
- fsubr ST(0), ST(i) – result of ST(i) - ST(0) save into ST(0)

Unary functions like sin, cos:

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

Logarithmic functions - calculation y * $\log_2$ x:

- fyl2x - ST(1) replace by value ST(1)*(log_2 ST(0)) and remove register ST(0) from stack – pop

Load constant values:

- fldz/fld1 - load 0.0/1.0 to the stack top – operation push
- fldpi/fldl2e - load Π / ($\log_2$ e) to the stack top – operation push

FPU coprocessor - x87

Example how to calculate 1.1*2.2+sin(3.3):

```
fldl   adr_1.1  ; load 1.1 from memory
fmull  adr_2.2  ; multiply ST(0) with value from
                  memory
fldl   adr_3.3  ; load 3.3 from memory
fsinl           ; compute sin from 3.3
faddp           ; add ST(0) and ST(1), pop ST(0)
fstp   adr_vysl ; store result into memory
```

FPU coprocessor – x87 Extension MMX

SIMD - Single Instruction Multiple Data – one type of operation parallel on more data

MMX - MultiMedia eXtension (sometimes expleined as Multiple Math eXtension)

MMX is using same registers as FPU x87, cannot be used parallel FPU and MMX

64-bits registr ST(0)..ST(7) can have following representation

- B - 8x byte
- W - 4x short int
- D - 2x int

Operations:

- Arithmetic – addition, subtraction, multiplication
- Logic - and, or, rotation, comparison
- Conversion - pack, move between registers

FPU for RISC V

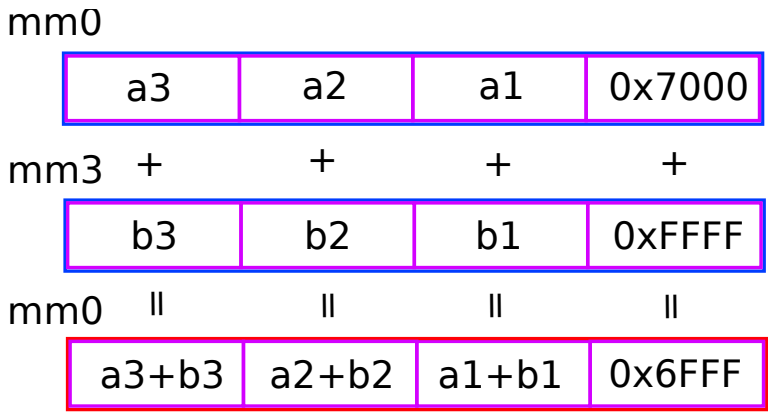
- Instruction extension RV64F – float, RV64D – double
- 32 internal registers with 32bit or 64bits
- New instruction for load and store – flw, fsw (fld, fsd)
- New instruction for:
  - fadd.s, fsub.s, fmul.s, fdiv.s (*.d for double precision)
  - fadd.s F[rd]=F[rs1]+F[rs2]
  - fsqrt.s – square root F[rd] = sqrt(F[rs1])
  - fmadd.s – multiply and add, F[rd]=F[rs1]*F[rs2]+F[rs3]
  - fmsub.s – multiply and sub, F[rd]=F[rs1]*F[rs2]-F[rs3]
  - fmin.s – F[rd] = (F[rs1]<F[rs2]) ? F[rs1] : F[rs2]
  - conversion between float, double, integer

MMX - Operations

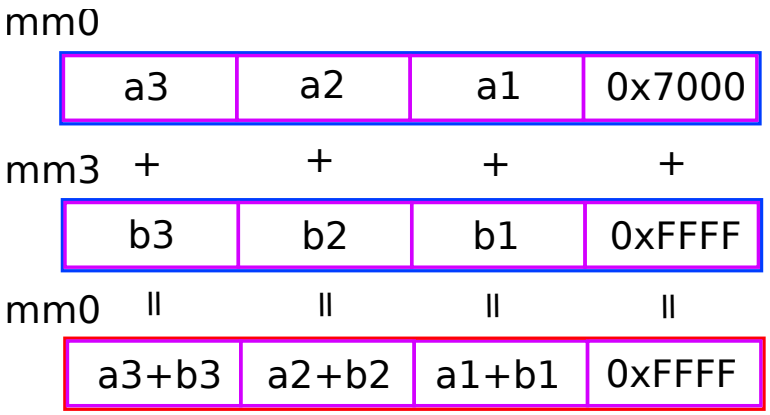
PADDW – packed add

mm0..mm7 4x words

Parallel add

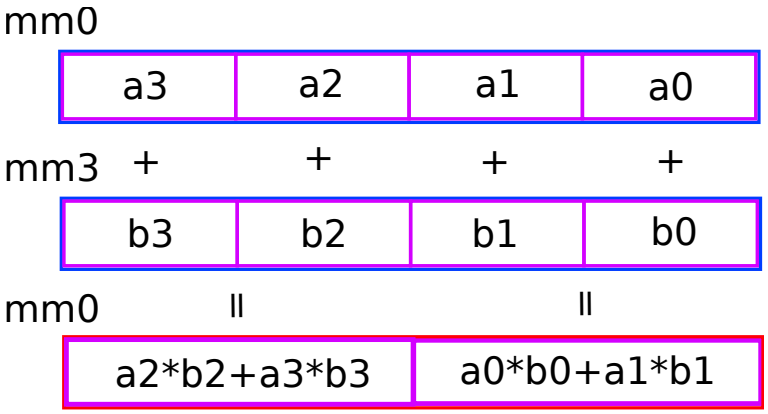


PADDUSW – packed add with saturation

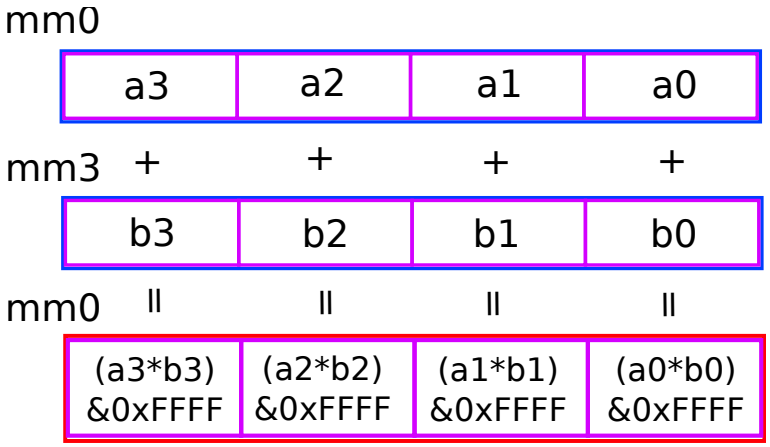


MMX - Operations

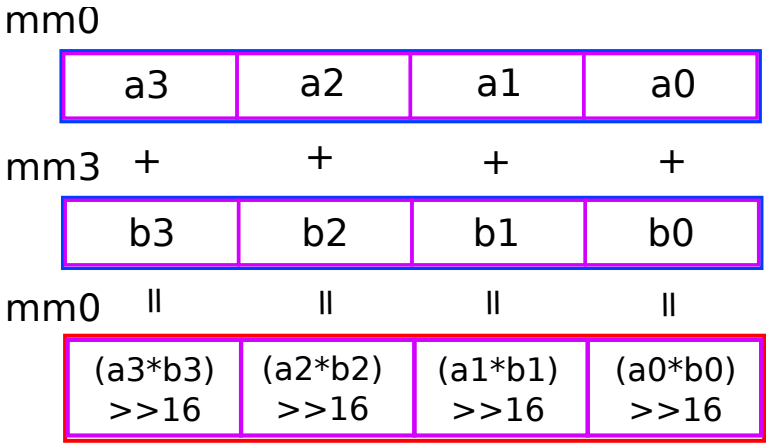
PMADDWD – multiply and add



PMULLW – multiply (low part)

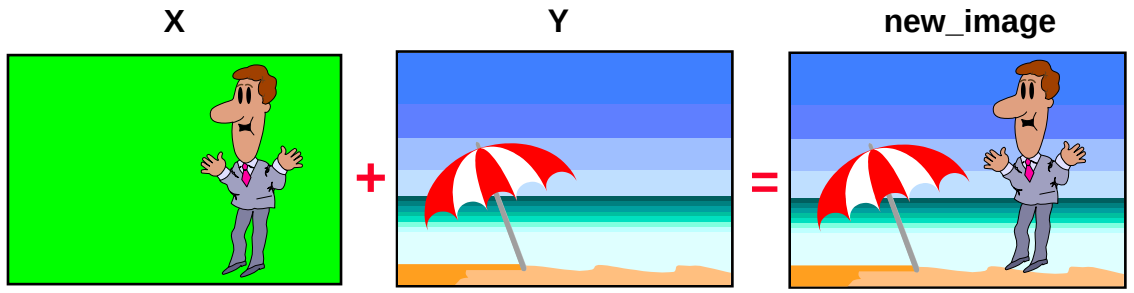


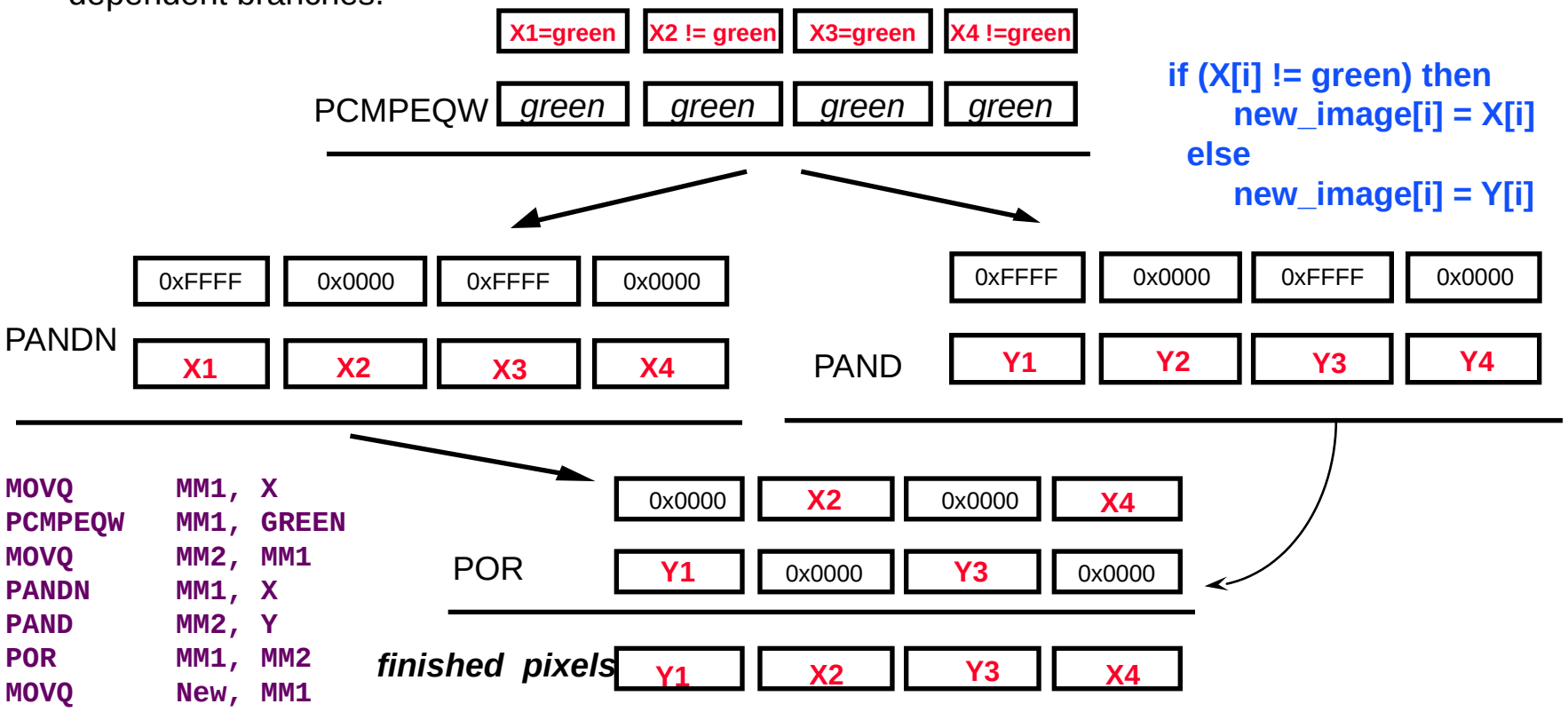
PMULHW – multiply (high part)



MMC – Conditional Select / Branch Removal

- Packed Comparison (PCMPCC) and the logical instructions enable conditional select operations in parallel and without data dependent branches.






```
MOVQ      MM1, X
PCMPEQW   MM1, GREEN
MOVQ      MM2, MM1
PANDN     MM1, X
PAND      MM2, Y
POR       MM1, MM2
MOVQ      New, MM1
```


Source: Dr. Konstantin Levit-Gurevich, Intel Israel

MMX - Example

Aplication of mask for fusion of two images:

```
unsigned char mask[size],
 obr1[size], obr2[size];

if (mask[i]==0) {
  new_img[i] = obr1[i];
} else {
  new_img[i] = obr2[i];
}
```

MMX implementation

8 pixels in one run

```
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!

- 3Dnow! Extension of MMX
- Extension 3Dnow! added computation with real numbers in old registers mm0-mm7
- It enables to store in one register 2 floating point numbers with 32 bits
- It extends addition, subtraction, multiplicaton and division of real numbers in parallel
- It adds conversion from real number to integer and back, avaraging of 8bits and 16bits integer numbers
- It adds comparison of real numbers and finding minimal a maximal values

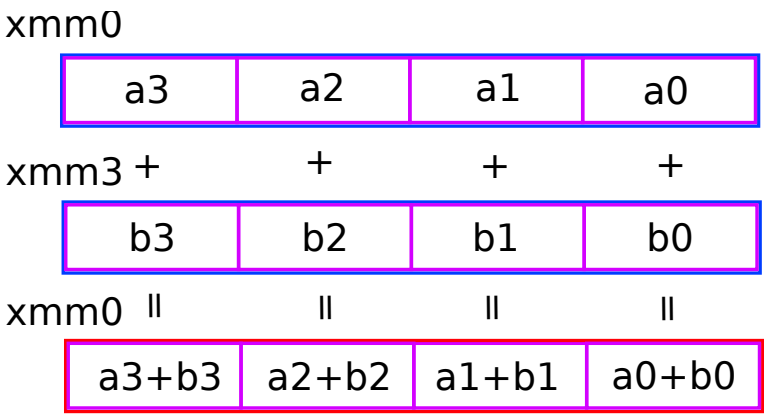
SSE – Streaming SIMD Extension

- SSE next SIMD extension
- New registers xmm0-xmm7,
- It enables to use FPU and SSE in parallel
- Each register has 128-bits
- Register can be used as:
  - 4x float – 32-bits FP
  - 2x double – 64-bits FP
- Extension of integer operation to new xmm 128-bits registers

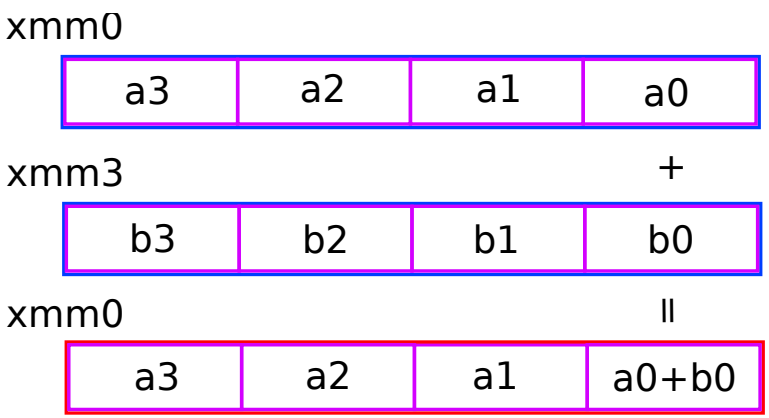
SSE – Instructions

Two type of instructions:

- packet suffix -ps,



- scalar suffix -ss



SSE – Instructions

Basic packed instructions:

- Load and store xmm register from memory
- Aritmethmetic operations with float: add, sub, mul, div, rcp, sqrt, max, min, rsqrt
- Bitwise operations: and, or, xor, andn
- Comparison: cmp, comi, ucomi

Scalar operation: addss, subss, mulss, divss

SSE – Extension

New versions of SSE:

- SSE2 – added new 144 new instructions
- SSE3 – added new 13 new instructions
- SSSE3 – added new 16 new instructions
- SSE4 – added new 47 new instructions
- SSE4.2 – added new 170 new instructions