

# Sécurité des logiciels

## Assembly language, part 2

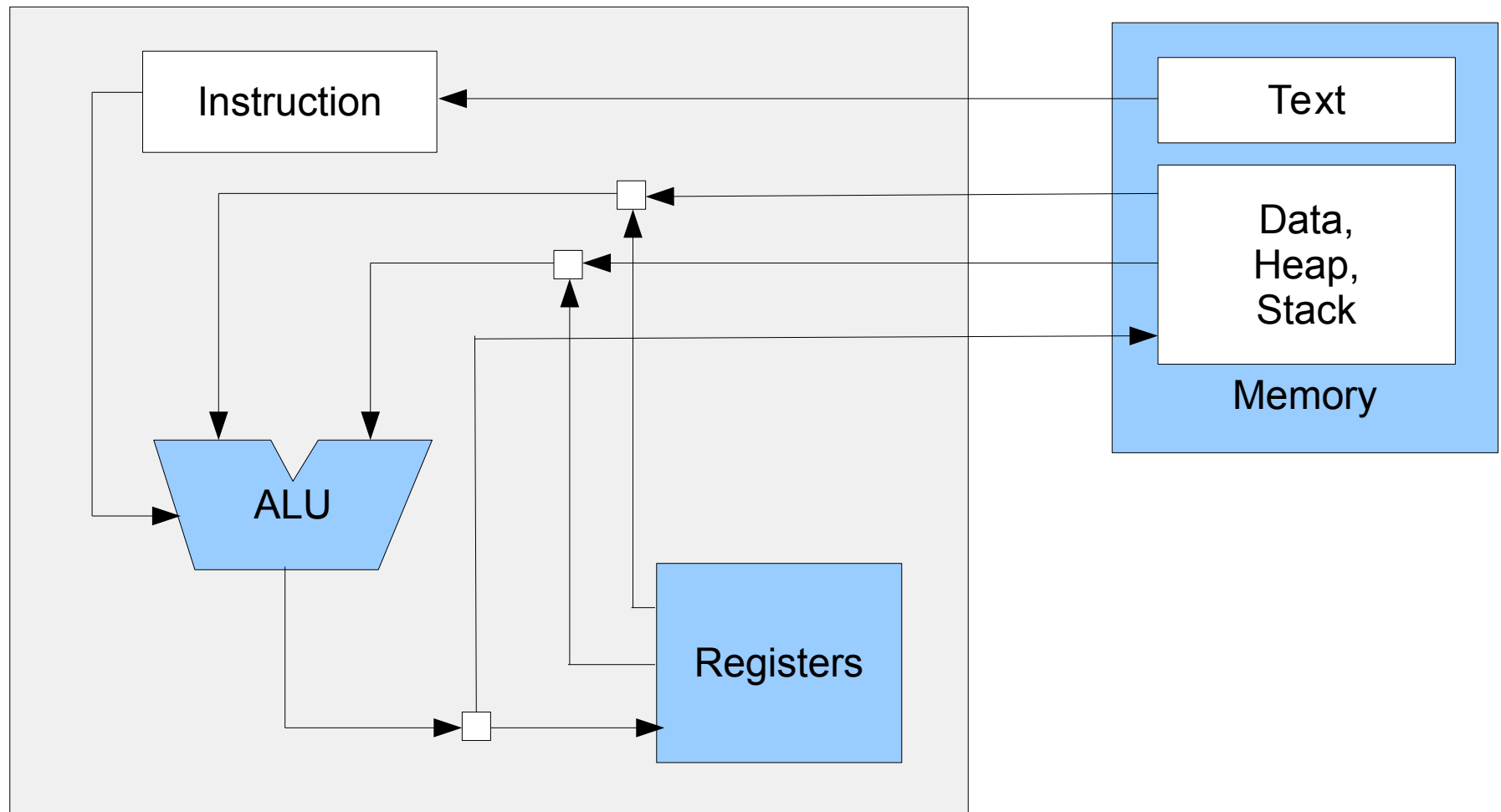
Samuel Thibault <[samuel.thibault@u-bordeaux.fr](mailto:samuel.thibault@u-bordeaux.fr)>

Pieces from Emmanuel Fleury <[emmanuel.fleury@u-bordeaux.fr](mailto:emmanuel.fleury@u-bordeaux.fr)>

CC-BY-NC-SA

# Execution model

# Execution model



# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```

# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

0x00:	mov \$1, A	←	IP:	0x00
0x05:	mov \$2, B		A:	0x00
0x0a:	mov \$3, C		B:	0x00
0x0f:	add A, B		C:	0x00
0x10:	cmp B, C		D:	0x00
0x11:	jz zero			
0x12:	mov \$41, A			
0x17:	jmp end			
0x18:	zero:			
0x18:	mov \$42, A			
0x1d:	end:			

# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```



```
IP:  0x05
A:   0x01
B:   0x00
C:   0x00
D:   0x00
```

# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```



```
IP:  0x0a
A:    0x01
B:    0x02
C:    0x00
D:    0x00
```

# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

0x00: mov \$1, A

0x05: mov \$2, B

0x0a: mov \$3, C

0x0f: add A, B

0x10: cmp B, C

0x11: jz zero

0x12: mov \$41, A

0x17: jmp end

0x18: zero:

0x18: mov \$42, A

0x1d: end:

IP: 0x0f

A: 0x01

B: 0x02

C: 0x03

D: 0x00





# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

0x00: mov \$1, A

0x05: mov \$2, B

0x0a: mov \$3, C

0x0f: add A, B

0x10: cmp B, C

0x11: jz zero

0x12: mov \$41, A

0x17: jmp end

0x18: zero:

0x18: mov \$42, A

0x1d: end:

IP: 0x10

A: 0x01

B: 0x03

C: 0x03

D: 0x00



# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```



```
IP:  0x11
A:    0x01
B:    0x03
C:    0x03
D:    0x00
Z
```

# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```

```
IP: 0x18
A:  0x01
B:  0x03
C:  0x03
D:  0x00
Z
```



# Execution model

## A simple example

- IP: Instruction Pointer (aka PC, Program Counter)
- A, B, C, D: registers

```
0x00:  mov $1, A
0x05:  mov $2, B
0x0a:  mov $3, C
0x0f:  add A, B
0x10:  cmp B, C
0x11:  jz  zero
0x12:  mov $41, A
0x17:  jmp end
0x18:  zero:
0x18:  mov $42, A
0x1d:  end:
```

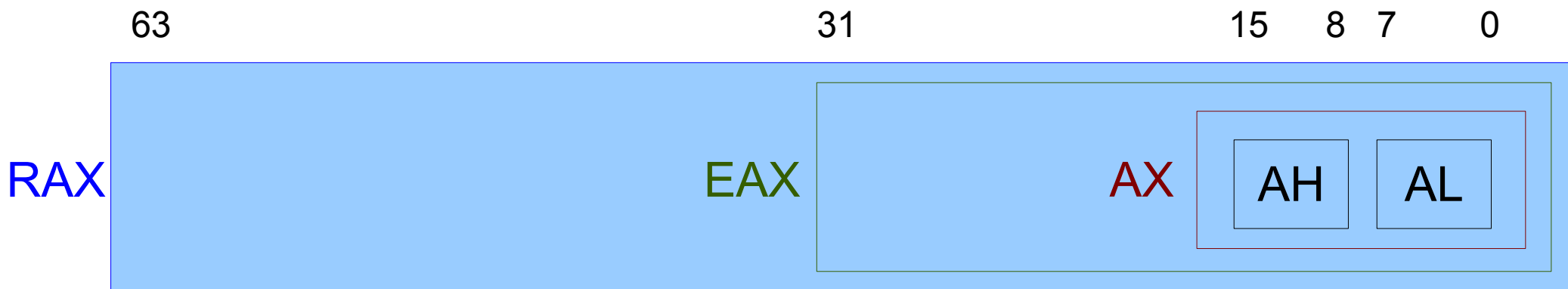
```
IP: 0x1d
A:  0x2a
B:  0x03
C:  0x03
D:  0x00
Z
```



# Registers

# Various register sizes

- Originally called A, B, C, D (8bits)
- Then called AX, BX, CX, DX (16bits) (plus AL, AH, etc.)
- Then called EAX, EBX, ECX, EDX (32bits)
- Then called RAX, RBX, RCX, RDX (64bits)
- Bets on the 128bit name?



In the following, will mostly use 32bit names (eax etc.)

# Various register sizes

## Instruction suffixes

- `movb $1, %al`      `# 8bits`
- `movb $1, %ah`      `# 8bits, high`
- `movw $1, %ax`      `# 16bits`
- `movl $1, %eax`      `# 32bits`
- `movq $1, %rax`      `# 64bits`

# Various registers

## General-purpose

- RAX
- RBX
- RCX (counter)
- RDX

## Indexing

- RSI (source)
- RDI (destination)

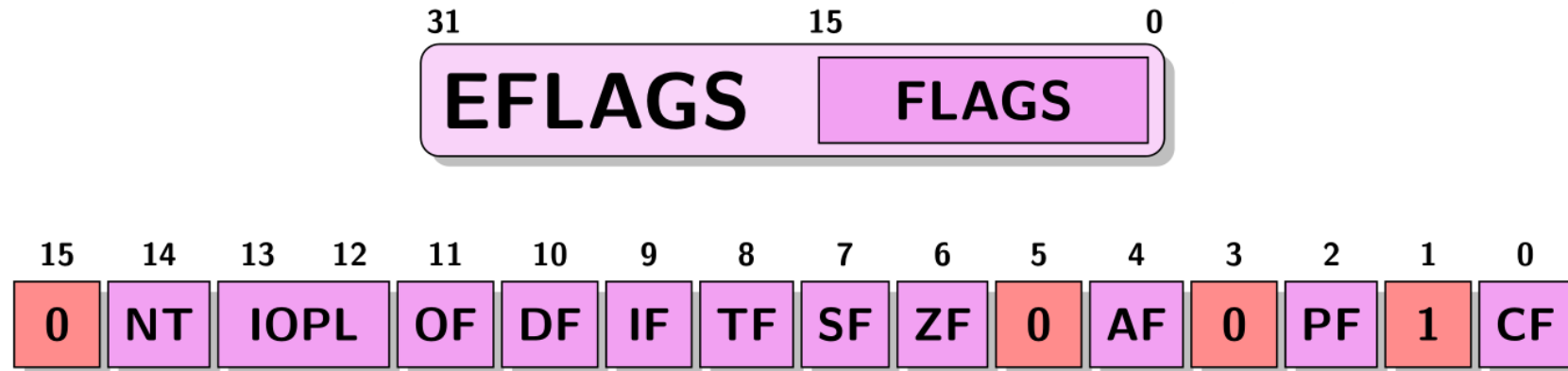
## New with 64bit

- R8...R15

More to come, but later

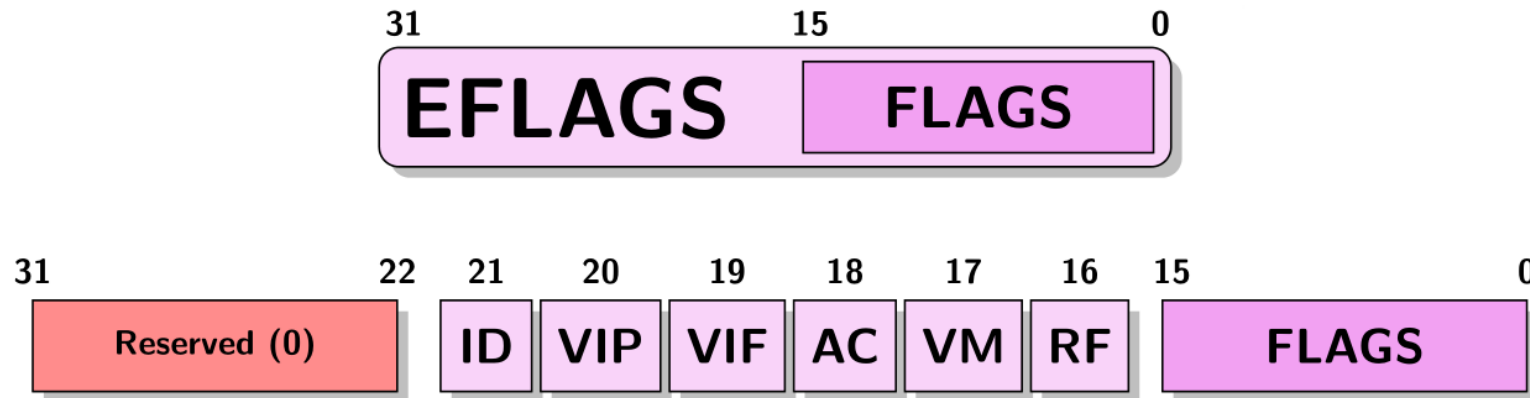


# Result eflags



- **CF** (**stat**): Carry flag;  
(left-most bit of the result)
- **PF** (**stat**): Parity flag;  
(right-most bit of the result)
- **AF** (**stat**): Adjust flag;
- **ZF** (**stat**): Zero flag;  
(set if result is zero)
- **SF** (**stat**): Sign flag;  
(most-significant bit of the result)
- **OF** (**stat**): Overflow flag;  
(set if an overflow occurs)
- **DF** (**ctrl**): Direction flag;  
(set the reading direction of a string)
- **IF** (**sys**): Interrupt enabled flag;
- **TF** (**sys**): Trap flag;
- **IOPL** (**sys**): I/O privilege level;  
(ring number currently in use)
- **NT** (**sys**): Nested task flag;

# Result eflags (2)



- **RF (sys)**: Resume flag;  
(Set CPU's response to debug exceptions)
- **VM (sys)**: Virtual 8086 mode;
- **AC (sys)**: Alignment check;
- **VIF (sys)**: Virtual interrupt flag;
- **VIP (sys)**: Virtual interrupt pending;  
(Set if an interrupt is pending)
- **ID (sys)**: CPUID flag;  
(Ability to use the CPUID instruction)



# Instructions

# Additive instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
add	src	dst	$\text{src} + \text{dst} \rightarrow \text{dst}$	OF, SF, ZF, AF, CF, PF
sub	src	dst	$\text{dst} - \text{src} \rightarrow \text{dst}$	
inc	dst		$\text{dst} + 1 \rightarrow \text{dst}$	
dec	dst		$\text{dst} - 1 \rightarrow \text{dst}$	
neg	dst		$-\text{dst} \rightarrow \text{dst}$	OF, SF, ZF, AF, PF

```
.globl _start
```

```
_start:
```

```
    movl $15, %eax    # %eax = 15
```

```
    subl $7, %eax     # %eax = %eax - 7
```

```
    addl $30, %eax    # %eax = %eax + 30
```

```
    decl %eax         # %eax = %eax - 1
```

# Multiplicative instructions

Mnemonic	Operand	Operation	Touched Flags
mul	src	%eax * src → %edx:%eax (unsigned)	CF, OF
imul	src	%eax * src → %edx:%eax (signed)	
div	dst	%edx:%eax / src → quot:%eax, rest:%edx (unsigned)	OF, SF, ZF, AF, CP, PF
idiv	dst	%edx:%eax / src → quot:%eax, rest:%edx (signed)	

```
.globl _start

_start:
    movl $8, %eax
    movl $0, %edx
    movl $2, %ebx
    divl %ebx      # %eax = %edx.%eax / %ebx
    addl $3, %eax  # %eax = %eax + 3
    movl $-15, %ebx
    imull %ebx     # %eax = -15 * %eax
```

# Shift/rotate instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
shl	cnt	dst	dst << cnt → dst (unsigned)	CF, OF
shr	cnt	dst	dst >> cnt → dst (unsigned)	
sal	cnt	dst	src << cnt → dst (signed)	
sar	cnt	dst	dst >> cnt → dst (signed)	
rol	cnt	dst	left rotate 'dst' of 'cnt' bits	
ror	cnt	dst	right rotate 'dst' of 'cnt' bits	

## Multiplication by $2^7$

```
.globl _start

_start:
    shll $7, %eax    # %eax * 2 ^ 7
    ret
```

# Bitwise instructions

Mnemonic	Operand	Operand	Operation	Touched Flags
and	src	dst	src & dst → dst	SF, ZF, PF
or	src	dst	src   dst → dst	
xor	src	dst	src ^ dst → dst	
test	op_1	op_2	op_1 & op_2 (result discarded)	
not	dst		~dst → dst	
cmp	op_1	op_2	op_2 - op_1 (result discarded)	OF, SF, ZF, AF, CF, PF

```
.globl _start

_start:
    movl $8, %eax
    andl $9, %eax
    notl %eax
L0:
    cmp $8, %eax # %eax == 8
    jz L0        # Jump to L0 if ZF==0
```

# Jump instructions

Mnemonic (unsigned / signed)	Operand	Operation
jmp	addr	Jump to 'addr' (unconditional jump)
ja = jnbe / jg = jnle	addr	Jump to 'addr' if <b>a</b> bove = <b>n</b> ot <b>b</b> elow or <b>e</b> qual / <b>g</b> reater = <b>n</b> ot <b>l</b> ess or <b>e</b> qual
jae = jnb = jnc / jge = jnl	addr	Jump to 'addr' if <b>a</b> bove or <b>e</b> qual = <b>n</b> ot <b>b</b> elow = <b>n</b> o <b>c</b> arry / greater or equal = not less
jna = jbe / jng = jle	addr	Jump to 'addr' if <b>n</b> ot <b>a</b> bove = <b>b</b> elow or <b>e</b> qual / <b>n</b> ot <b>g</b> reater = <b>l</b> ess or <b>e</b> qual
jnae = jb = jc / jnge = jl	addr	Jump to 'addr' if <b>n</b> ot <b>a</b> bove or <b>e</b> qual = <b>b</b> elow = <b>c</b> arry / <b>n</b> ot <b>g</b> reater or <b>e</b> qual = <b>l</b> ess
je = jz	addr	Jump to 'addr' if <b>e</b> qual / <b>z</b> ero (ZF = 1)
jne = jnz	addr	Jump to 'addr' if <b>n</b> ot <b>e</b> qual / <b>n</b> on- <b>z</b> ero (ZF = 0)
js / jns	addr	Jump to 'addr' if <b>s</b> igned (SF = 1) / <b>n</b> ot <b>s</b> igned (SF = 0)



# Jump instructions (2)

```
.globl _start

_start:
    movl $8, %ebx
    cmpl %eax, %ebx    # Compute %ebx - %eax
    jle L0             # If result ≤ 0 go to L0
    incl %ebx          # Increment %ebx
    ret

L0:
    decl %ebx          # Decrement %ebx
    ret
```

# Jump instructions (3)

## Small loop

```
.globl main
main:
    movl $0, %eax
L0:
    #...
    addl $1, %eax      # count one loop done
    cmpl %ebx, %eax    # compute %eax - %ebx
    jl  L0             # If < 0, %eax < %ebx, goto L0
end:
    ret
```

# Miscellaneous instructions

Mnemonic	Operand	Operation
loop	addr	decrement %ecx, jump to 'addr' if %ecx > 0
loope / loopz	addr	decrement %ecx, jump to 'addr' if %ecx > 0 and ZF = 1
loopnz / loopnz	addr	decrement %ecx, jump to 'addr' if %ecx > 0 and ZF = 0
nop (0x90)		No Operation



How about memory?

# Memory locations

## Different types of operands

- Registers

```
movl %eax,%ebx
```

- Immediates

```
movl $1, %eax      # constant
```

```
movl $0xa, %eax    # constant
```

```
movl $a, %eax       # address of label/symbol
```

- Direct memory locations

```
movl 1, %eax        # !!
```

```
movl a, %eax        # reference to label/symbol
```

# Memory locations

## Indirect memory locations

<code>movl (%ebx), %eax</code>	<code># address ebx</code>
<code>movl 4(%ebx), %eax</code>	<code># address ebx+4</code>
<code>movl -4(%ebx), %eax</code>	<code># address ebx-4</code>
<code>movl t(%ebx), %eax</code>	<code># address t+ebx</code>
<code>movl (%ebx,%ecx), %eax</code>	<code># address ebx+ecx</code>
<code>movl (%ebx,%ecx,2), %eax</code>	<code># address ebx+ecx*2</code>
<code>movl t(%ebx,%ecx,4), %eax</code>	<code># address t+ebx+ecx*4</code>
<code>movl t(,%ecx,2), %eax</code>	<code># address t+ecx*2</code>
<code>...</code>	

# Move instructions

Mnemonic	Operand	Operand	Operation
mov	src	dst	src → dst
xchg	src	dst	src ↔ dst
lea	addr	reg	addr → reg

## Load Effective Address (lea)

lea calculates its src operand as in the mov instruction, but rather than loading the contents of that address into dest, it loads the address itself.

It can be used for calculating addresses, but also for general-purpose unsigned integer arithmetic (with the caveat and possible benefit that FLAGS is untouched).

```
leal 8(,%eax,4), %eax    # Multiply eax by 4 and add 8
leal (%edx,%eax,2), %eax  # Multiply eax by 2 and add edx
```

# Move instructions

Mnemonic	Operand	Operand	Operation
movs	-	-	(%esi) → (%edi)
lods	-	-	(%esi) → %eax
stos	-	-	%eax → (%edi)
cmps	-	-	(%edi) - (%esi)

+ index increment! (or decrement if DF = 1)

Prefix	Repetition
rep	while %ecx > 0
repe / repz	while %ecx > 0 and ZF = 1
repne / repnz	while %ecx > 0 and ZF = 0

+ ecx decrement!

```
movl $var,%edi
movl $123,%ecx
cld
rep stosb
```



# Move instructions

Mnemonic	Operand	Operand	Operation
cmovxy	src	dst	src → dst if xy

```
# if (a < b) x = 1
```

```
cmpl    %eax, %ebx  
cmovgel $1, x
```

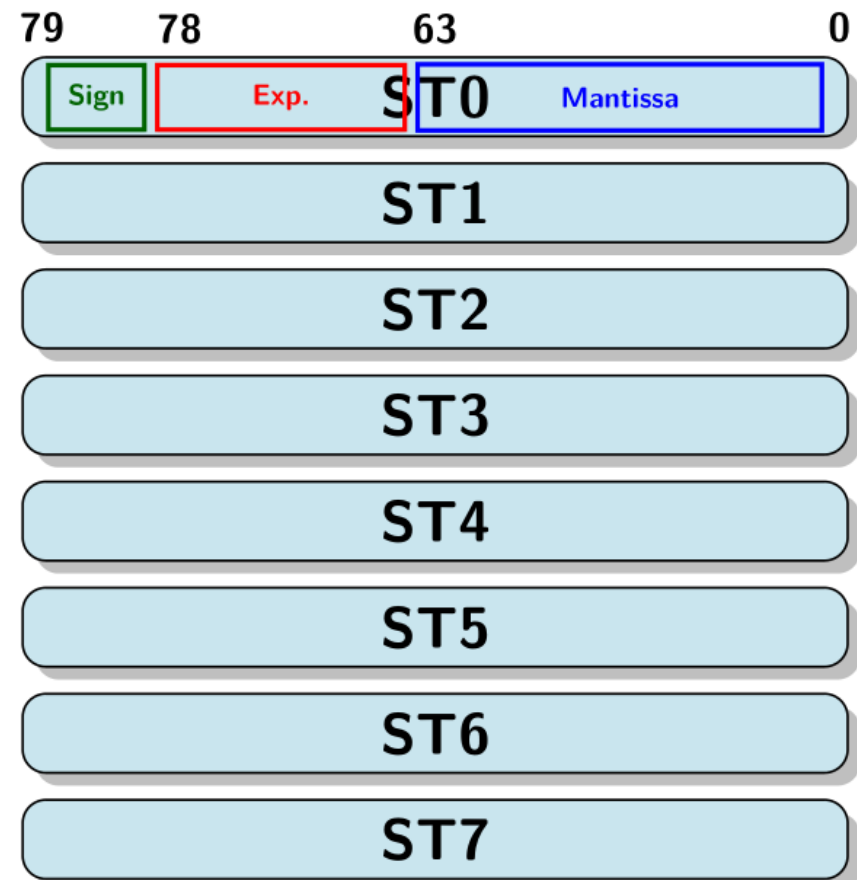
```
# compute %ebx-%eax  
# if > 0, set x to 1
```



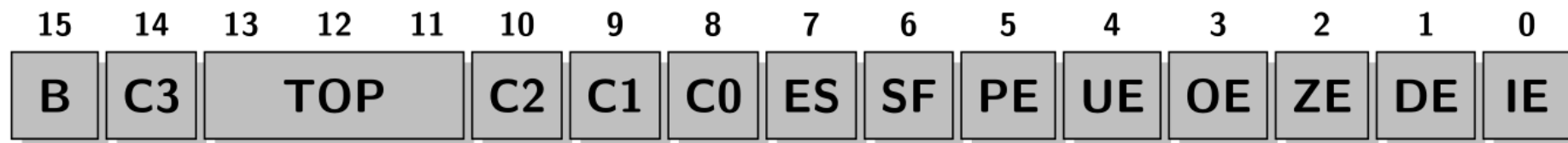
# Floating-point computation

# x87 floating point

- Float is 32bit wide
- Double is 64bit wide
- Processor computes with 80bits!
  - 1bit sign
  - 15bits exponent
  - 64bits mantissa
- Registers accessed as stack
  - No direct access
    - `push / pop`
  - MMX fixed that, but 64bit



# x87 floating point status (stw)



## Exception Flags (bits 0-5)

- **IE**: Invalid Operation Exception
- **DE**: Denormalized Operand Exception
- **FE**: Zero Divide Exception
- **OE**: Overflow Exception
- **UE**: Underflow Exception
- **PE**: Precision Exception

## Other Flags (bits 6-15)

- **B**: FPU Busy
- **TOP**: Top of Stack Pointer
- **C0, C1, C2, C3**: Condition Code
- **ES**: Error Summary Status
- **SF**: Stack Fault

# x87 instructions

Mnemonic	Operand	Operation	Notes
finit	–	–	Initialize FPU
fincstp	–	$ST + 1 \rightarrow ST$	Increase FPU stack pointer
fdecstp	–	$ST - 1 \rightarrow ST$	Decrease FPU stack pointer
ffree	st(i)	$0 \rightarrow st(i)$	Free st(i)
fldz	–	$0 \rightarrow st(0)$	Load zero
fld1	–	$1 \rightarrow st(0)$	Load one
fldpi	–	$\pi \rightarrow st(0)$	Load $\pi$
fld	addr	$(addr) \rightarrow st(0)$	Load float
	st(i)	$st(i) \rightarrow st(0)$	
fild	addr	$(addr) \rightarrow st(0)$	Load integer
fst	addr	$st(0) \rightarrow (addr)$	Store float to memory
fxch	st(i)	$st(i) \leftrightarrow st(0)$	Exchange content of st(0) and st(i)

# x87 instructions (2)

Mnemonic	Operand	Operation	Notes
<b>faddp</b>	—	$st(1) + st(0) \rightarrow st(0)$	Float addition
<b>fadd</b>	addr	$(addr) + st(0) \rightarrow st(0)$	
	addr, addr	$(addr) + (addr) \rightarrow st(0)$	
<b>fsubp</b>	—	$st(1) - st(0) \rightarrow st(0)$	Float subtraction
<b>fsub</b>	addr	$(addr) - st(0) \rightarrow st(0)$	
	addr, addr	$(addr) - (addr) \rightarrow st(0)$	
<b>fmlp</b>	—	$st(1) \times st(0) \rightarrow st(0)$	Float multiplication
<b>fmul</b>	addr	$(addr) \times st(0) \rightarrow st(0)$	
	addr, addr	$(addr) \times (addr) \rightarrow st(0)$	
<b>fdivp</b>	—	$st(1) \div st(0) \rightarrow st(0)$	Float division
<b>fdiv</b>	addr	$(addr) \div st(0) \rightarrow st(0)$	
	addr, addr	$(addr) \div (addr) \rightarrow st(0)$	
<b>fchs</b>	—	$-st(0) \rightarrow st(0)$	Change sign
<b>fabs</b>	—	$ st(0)  \rightarrow st(0)$	Absolute value
<b>fsqrt</b>	—	$\sqrt{st(0)} \rightarrow st(0)$	Square root
<b>fsin</b>	—	$\sin(st(0)) \rightarrow st(0)$	Sinus
<b>fcos</b>	—	$\cos(st(0)) \rightarrow st(0)$	Cosine

# x87 instructions (3)

Mnemonic	Operand	Operation	Notes
fcom	–	Compares st(0) and st(1)	C0=( <i>st</i> (0) < <i>src</i> ), C3=( <i>st</i> (0) == <i>src</i> )
	st( <i>i</i> )	Compares st(0) and st( <i>i</i> )	
	addr	Compares st(0) and (addr)	
fcomi	st( <i>i</i> )	Compares st(0) and st( <i>i</i> )	Set EFLAGS (not STW)
fcmbv	st( <i>i</i> )	if (CF=1) <i>st</i> ( <i>i</i> ) → <i>st</i> (0)	Move if below
fcmove	st( <i>i</i> )	if (ZF=1) <i>st</i> ( <i>i</i> ) → <i>st</i> (0)	Move if equal
fcmbve	st( <i>i</i> )	if (CF=1)&(ZF=1) <i>st</i> ( <i>i</i> ) → <i>st</i> (0)	Move if below or equal

# x87 instructions example

```
_start:
    movl    $1024, %eax # push the integer (1024) to analyze
    bsrl    %eax, %eax # bit scan reverse (smallest non zero index)
    inc     %eax       # take the 0th index into account
    pushl   %eax       # save the result on the stack

    fldl    (%esp)      # load to the FPU stack (st(0))
    fldlg2             # load log10(2) on the FPU stack
    fmulp    %st, %st(1) # %st(0) * %st(1) -> %st(0)

    # Set the FPU control word (%stcw) to 'round-up' (default: 'round-down')
    fstcw    -2(%esp)    # save the old FPU control word
    movw     -2(%esp), %ax # store the FPU control word in %ax
    andw     $0xf3ff, %ax # remove everything else
    orw      $0x0800, %ax # set the 'round-up' bit
    movw     %ax, -4(%esp) # store the value back to the stack
    fldcw    -4(%esp)    # set the FPU control word with the proper value

    frndint                     # round-up

    fldcw    -2(%esp)    # restore the old FPU control word

    fistpl   (%esp)      # load the final result to the stack
    popl     %eax        # set the return value to be our result

    leave                     # clean the stack-frame
```



# SSE

## XMM registers

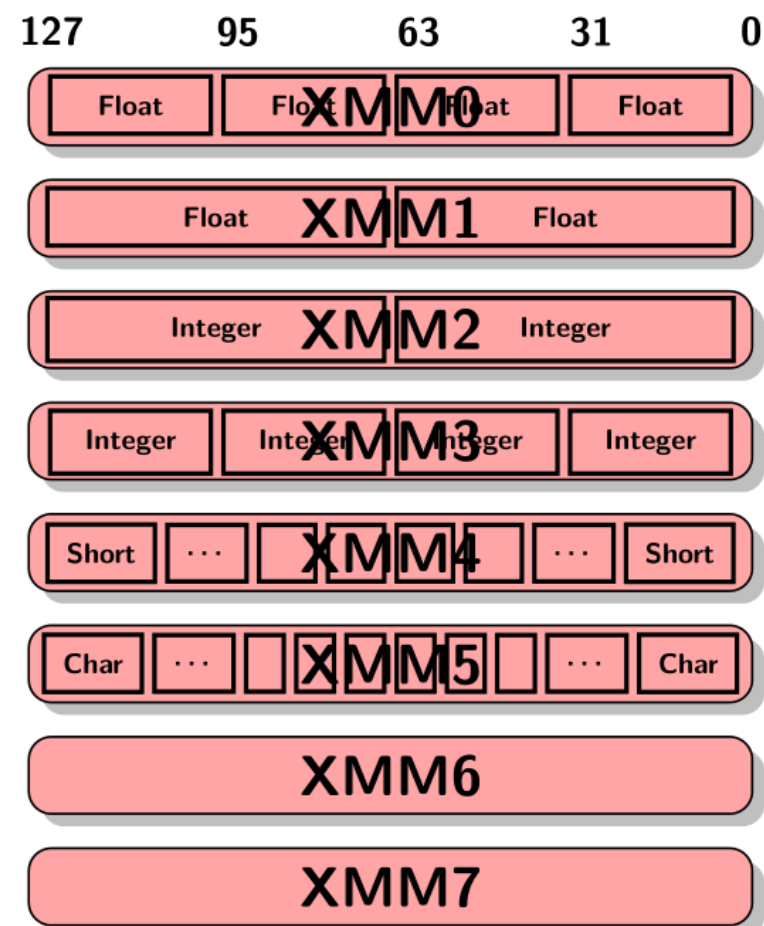
- 128 bits
- Can pack small arrays of data

## AVX2: YMM registers

- 256 bits

## AVX512: ZMM registers

- 512 bits



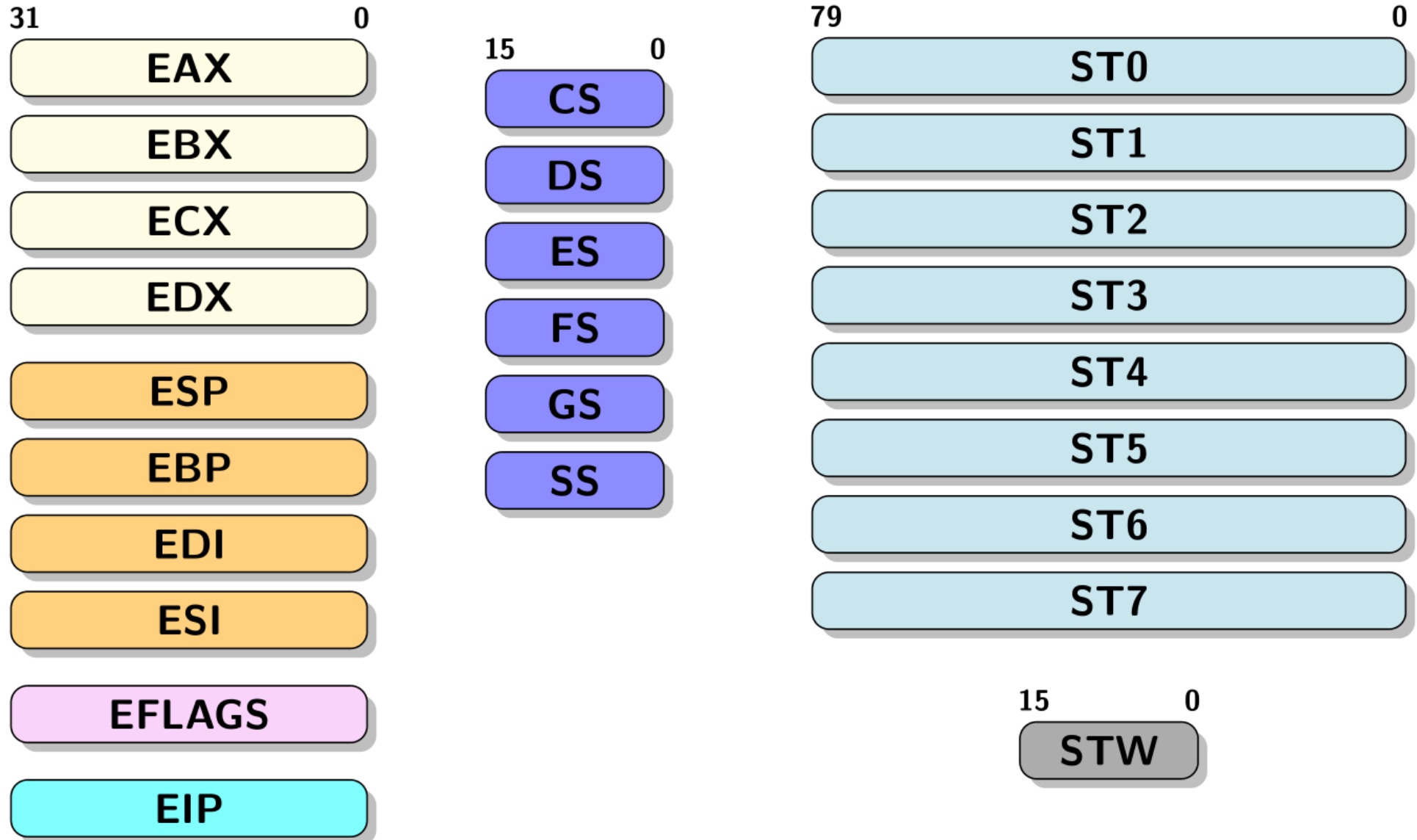
# SSE instructions

## SSE Instructions (Single-precision Floats)

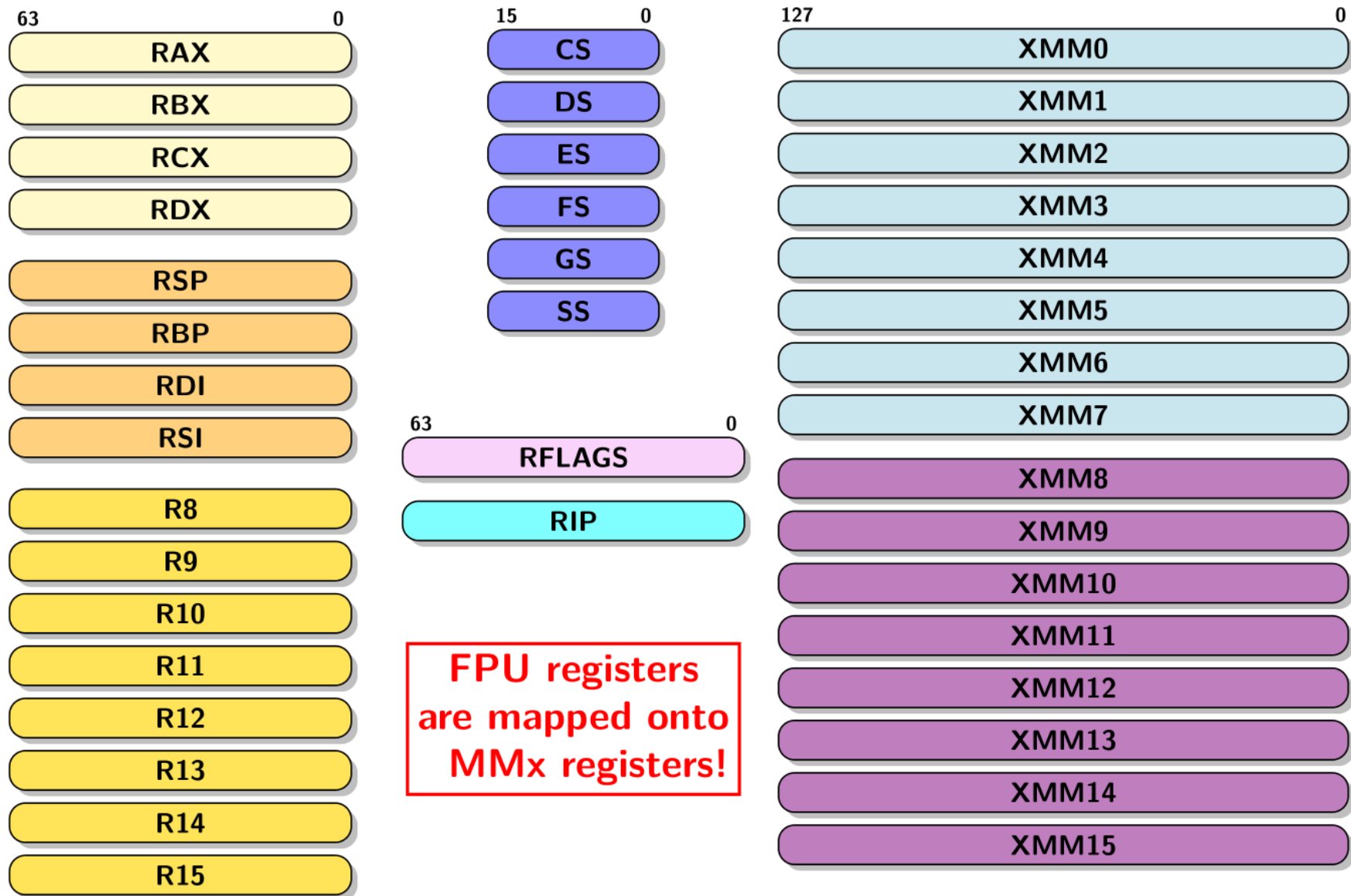
Mnemonic	Operand	Operand	Operation
<code>movss</code>	src (addr/xmm)	dst (xmm)	src $\rightarrow$ dst
<code>addss</code>	src (addr/xmm)	dst (xmm)	src + dst $\rightarrow$ dst
<code>subss</code>	src (addr/xmm)	dst (xmm)	src - dst $\rightarrow$ dst
<code>mulss</code>	src (addr/xmm)	dst (xmm)	src $\times$ dst $\rightarrow$ dst
<code>divss</code>	src (addr/xmm)	dst (xmm)	src $\div$ dst $\rightarrow$ dst

# Registers recap

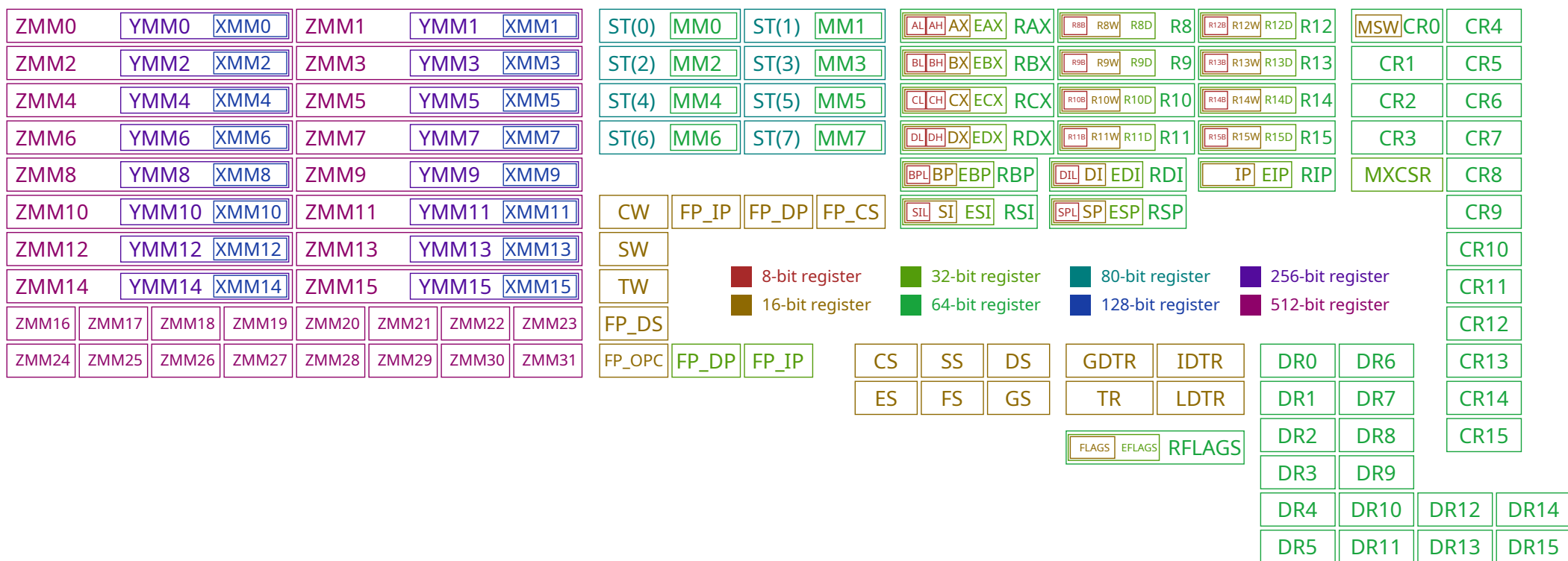
# Legacy 32bit registers



# 64bit registers



# ALL 64bit registers with avx 512



From wikipedia X86 page, CC-BY-SA 3.0



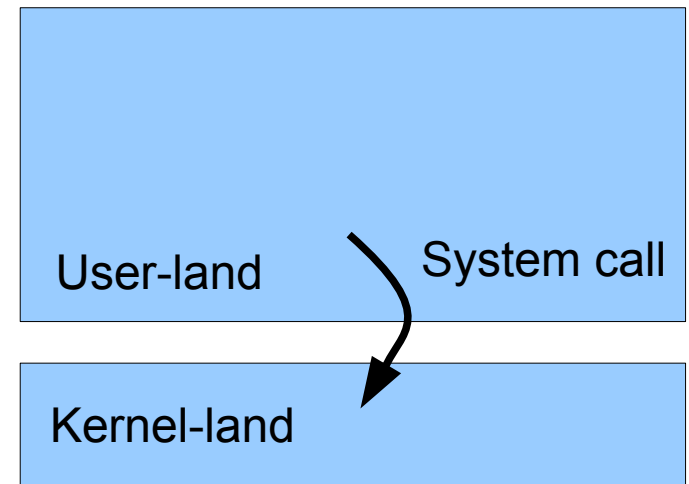
How to enter the kernel?

# Entering the kernel

- User programs run in userland
- Need to interact with the world

## System call

- $\sim$  = function call
- But change address space
- Thus special instruction
- `int $0x80`
  - System call number in `%eax`
  - Parameters in `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`
  - Return value in `%eax`
- `Syscall`
  - System call number in `%rax`
  - Parameters in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
  - Return value in `%rax`





# Entering the kernel

```
.globl main
main:
    movl $1, %eax # System call number
    int $0x80     # Calling the OS
    ret
```

# 32bit system calls

eax	Name	ebx	ecx	edx	esi	edi
0x01	sys_exit	int	—	—	—	—
0x02	sys_fork	—	—	—	—	—
0x03	sys_read	unsigned int	char*	size_t	—	—
0x04	sys_write	unsigned int	const char*	size_t	—	—
0x05	sys_open	const char*	int	int	—	—
0x06	sys_close	unsigned int	—	—	—	—
0x07	sys_waitpid	pid_t	unsigned int	int	—	—
0x08	sys_create	const char*	int	—	—	—
0x09	sys_link	const char*	const char*	—	—	—
0x0a	sys_unlink	const char*	—	—	—	—
0x0b	sys_execve	const char*	char**	char**	—	—
0x0c	sys_chdir	const char*	—	—	—	—
0x0d	sys_time	int*	—	—	—	—
0x0e	sys_mknod	const char*	mode_t	dev_t	—	—
0x0f	sys_chmod	const char*	mode_t	—	—	—
...	...	...	...	...	...	...

See in `/usr/include/*/asm/unistd_32.h`

# 64bit system calls

rax	Name	rdi	rsi	rdx	r10	r8	r9
0x00	sys_read	unsigned int	char*	size_t	—	—	—
0x01	sys_write	unsigned int	const char*	size_t	—	—	—
0x02	sys_open	const char*	int	int	—	—	—
0x03	sys_close	unsigned int	—	—	—	—	—
0x04	sys_stat	const char*	struct stat*	—	—	—	—
0x05	sys_fstat	unsigned int	struct stat*	—	—	—	—
0x06	sys_lstat	fconst char*	struct stat*	int	—	—	—
0x07	sys_poll	struct poll_fd	unsigned int	long	—	—	—
0x08	sys_lseek	unsigned int	off_t	unsigned int	—	—	—
...	...	...	...	...	...	...	...

See in `/usr/include/*/asm/unistd_64.h`

# Hello, world!

```
.data # Data section
msg:
    .asciz "Hello World!\n" # String
    len = . - msg          # String length

.text # Text section
.globl main # Export entry point to ELF linker

main: # Write the string to stdout
    movl $len, %edx # 3rd argument: string length
    movl $msg, %ecx # 2nd argument: pointer to string
    movl $1, %ebx # 1st argument: file handler (stdout)
    movl $4, %eax # System call number (sys_write)
    int $0x80 # Kernel call
    # And exit
    movl $0, %ebx # 1st argument: exit code
    movl $1, %eax # System call number (sys_exit)
    int $0x80 # Kernel call
```

**!! AT&T vs Intel !!**

# AT&T vs Intel

	AT&T Syntax	Intel Syntax
Community	UNIX	Microsoft
Direction of operands	<p><b>from left to right</b></p> <p>First operand is 'source' and second operand is 'destination'.</p> <pre>Instr.  src,    dest mov      (%ecx), %eax</pre>	<p><b>from right to left</b></p> <p>First operand is 'destination' and second one is 'source'.</p> <pre>Instr.  dest,    src mov      eax,    [ecx]</pre>
Addressing Memory	<p>Addresses are enclosed in parenthesis ('(', ')') and given by the formula:  <code>offset(base, index, scale)</code></p> <pre>movl (%ebx),          %eax movl 3(%ebx),          %eax movl 0x20(%ebx),       %eax addl (%ebx,%ecx,0x2),  %eax leal (%ebx,%ecx),      %eax subl -0x20(%ebx,%ecx,0x4), %eax</pre>	<p>Addresses are enclosed in brackets('[', ']') and given by the formula:  <code>[base+index*scale+offset]</code></p> <pre>mov eax, [ebx] mov eax, [ebx+3] mov eax, [ebx+20h] add eax, [ebx+ecx*2h] lea eax, [ebx+ecx] sub eax, [ebx+ecx*4h-20h]</pre>

# AT&T vs Intel (2)

	AT&T Syntax	Intel Syntax
Data types	<ul style="list-style-type: none"><li>● Registers: '%eax'</li><li>● Concatenation: '%eax:%ebx'</li><li>● Immediate values: '\$1'</li><li>● Decimal: '10' (or '0d10')</li><li>● Hexadecimal: '0x10'</li><li>● Operand on bytes: 'movb'</li><li>● Operand on words: 'movw'</li><li>● Operand on longs: 'movl'</li></ul> <pre>movl \$1,      %eax movl \$0xff,   %ebx int  \$0x80 movb %bl,     %al movw %bx,     %ax movl %ebx,    %eax movl (%ebx),  %eax</pre>	<ul style="list-style-type: none"><li>● Registers: 'eax'</li><li>● Concatenation: 'eax:ebx'</li><li>● Immediate values: '1'</li><li>● Decimal: '10' (or '10d')</li><li>● Hexadecimal: '10h'</li><li>● Address of bytes: 'byte ptr'</li><li>● Address of words: 'word ptr'</li><li>● Address of longs: 'dword ptr'</li></ul> <pre>mov eax, 1 mov ebx, 0ffh int 80h mov al, bl mov ax, bx mov eax, ebx mov eax, dword ptr [ebx]</pre>

# Compiling / disassembling



# Compiling with gcc

## With libc

```
# Example with libc
.globl main

main:
    movl $20, %eax
    ret
```

## Without libc

```
# Example with no libc
.globl _start

_start:
    movl $20, %eax
    # No 'ret' in _start!!!
```

## Build the binary

```
gcc -m32 -static -o ex1 asm.s
gcc -m64 -static -o ex1 asm.s
```

## Run the binary

```
#> ./ex1
#> echo $?
```

## Build the binary

```
gcc -m32 -static -nostdlib -o ex1 asm.s
gcc -m64 -static -nostdlib -o ex1 asm.s
```

## Run the binary

```
#> ./ex1
#> echo $?
```

# Compiling with as / ld

## Without libc

```
# Example with no libc
.globl _start

_start:
    movl $20, %eax
    # No 'ret' in _start!!!
```

### Build the binary (32)

```
as --32 -o ex1.o asm.s
ld -m elf_i386 -o ex1 ex1.o
```

### Run the binary

```
#> ./ex1
#> echo $?
```

### Build the binary (64)

```
as --64 -o ex1.o asm.s
ld -m elf_x86_64 -o ex1 ex1.o
```

### Run the binary

```
#> ./ex1
#> echo $?
```

# Gdb in textmode

```
#> gdb ./ite
...
Reading symbols from ./ite...done.
(gdb) break _start
Breakpoint 1 at 0x175: file ite.s, line 5.
(gdb) run
Starting program: ./ite

Breakpoint 1, _start () at ite.s:5
5          movl $8, %ebx
(gdb) disas
Dump of assembler code for function _start:
=> 0x56555175 <+0>:      mov     $0x8,%ebx
    0x5655517a <+5>:      cmp     %eax,%ebx
    0x5655517c <+7>:      jle     0x56555180 <L0>
    0x5655517e <+9>:      inc     %ebx
    0x5655517f <+10>:     ret
End of assembler dump.
(gdb) disas L0
Dump of assembler code for function L0:
    0x56555180 <+0>:      dec     %ebx
    0x56555181 <+1>:      ret
End of assembler dump.
(gdb) nexti
6          cmpl %eax, %ebx
(gdb) stepi
7          jle L0
(gdb) si
8          incl %ebx
(gdb) disas
Dump of assembler code for function _start:
    0x56555175 <+0>:      mov     $0x8,%ebx
    0x5655517a <+5>:      cmp     %eax,%ebx
    0x5655517c <+7>:      jle     0x56555180 <L0>
=> 0x5655517e <+9>:      inc     %ebx
    0x5655517f <+10>:     ret
End of assembler dump.
```

```
(gdb) backtrace
#0 _start () at ite.s:8
(gdb) print /x $eax
$1 = 0xf7ffd918
(gdb) set $eax = 0
(gdb) info reg
eax             0x0             0
ecx             0xffffd134      -11980
edx             0xf7fe88b0      -134313808
ebx             0x8             8
esp             0xffffd130      0xffffd130
ebp             0x0             0x0
esi             0xffffd13c      -11972
edi             0x56555175      1448431989
eip             0x5655517e      0x5655517e <_start+9>
eflags         0x207           [ CF PF IF ]
...
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: ./ite

Breakpoint 1, _start () at ite.s:5
5          movl $8, %ebx
(gdb) set $eax = 10
(gdb) si
6          cmpl %eax, %ebx
(gdb) si
7          jle L0
(gdb) si
11         decl %ebx
(gdb) disas
Dump of assembler code for function L0:
=> 0x56555180 <+0>:      dec     %ebx
    0x56555181 <+1>:      ret
End of assembler dump.
(gdb)
```

# Gdb in tui mode

- Control-x 2 several times

```
rip      0x55555555005      0x55555555005 < start+5>
rax      0x1c              28
rbx      0x8               8
rcx      0x0              0
rdx      0x7ffff7fe2180    140737354015104
rsi      0x7ffff7ffe720    140737354131232
rdi      0x7ffff7ffe180    140737354129792
rbp      0x0              0x0
rsp      0x7fffffffdd180    0x7fffffffdd180
r8       0x0              0
r9       0x0              0
r10      0x555555554000    93824992231424
r11      0x206            518

test.s
B+ 3      movl $8, %ebx
>4      cmpl %eax,%ebx
5        jle L0
6        incl %ebx
7        ret
8
9        L0:
10       decl %ebx
11       ret

native process 2174023 In: _start L4 PC: 0x55555555005
(gdb) b _start
Breakpoint 1 at 0x1000: file test.s, line 3.
(gdb) si
The program is not being run.
(gdb) r
Starting program: /home/samy/enseignement/SSI/ScuLang/cours4/test

Breakpoint 1, _start () at test.s:3
(gdb) si
(gdb)
```

# Disassembling with objdump

```
#> objdump -d ite
```

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175 <_start>:
```

175:	bb 08 00 00 00	mov	\$0x8,%ebx
17a:	39 c3	cmp	%eax,%ebx
17c:	7e 02	jle	180 <L0>
17e:	43	inc	%ebx
17f:	c3	ret	

```
00000180 <L0>:
```

180:	4b	dec	%ebx
181:	c3	ret	

# Disassembling with objdump

```
#> objdump -d ite
```

Instruction addresses

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175 <_start>:
```

175:	bb 08 00 00 00	mov	\$0x8,%ebx
17a:	39 c3	cmp	%eax,%ebx
17c:	7e 02	jle	180 <L0>
17e:	43	inc	%ebx
17f:	c3	ret	

```
00000180 <L0>:
```

180:	4b	dec	%ebx
181:	c3	ret	

# Disassembling with objdump

```
#> objdump -d ite
```

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175 <_start>:
```

175:	bb 08 00 00 00
17a:	39 c3
17c:	7e 02
17e:	43
17f:	c3

```
mov    $0x8,%ebx
cmp    %eax,%ebx
jle    180 <L0>
inc    %ebx
ret
```

```
00000180 <L0>:
```

180:	4b
181:	c3

```
dec    %ebx
ret
```

Instruction addresses

Instruction opcodes

# Disassembling with objdump

```
#> objdump -d ite
```

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175 <_start>:
```

175:	bb 08 00 00 00
17a:	39 c3
17c:	7e 02
17e:	43
17f:	c3

mov	\$0x8,%ebx
cmp	%eax,%ebx
jle	180 <L0>
inc	%ebx
ret	

```
00000180 <L0>:
```

180:	4b	dec	%ebx
181:	c3	ret	

Instruction addresses

Instruction opcodes

Instruction mnemonics



# Disassembling with objdump

```
#> objdump -d ite
```

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175 <_start>:
```

175:	bb 08 00 00 00
17a:	39 c3
17c:	7e 02
17e:	43
17f:	c3

mov
cmp
jle
inc
ret

\$0x8,%ebx
%eax,%ebx
180 <L0>
%ebx

```
00000180 <L0>:
```

180:	4b
181:	c3

dec	%ebx
ret	

Instruction addresses

Instruction opcodes

Instruction mnemonics

Instruction operands

# Disassembling with objdump

```
#> objdump -d ite
```

```
ite: file format elf32-i386
```

```
Disassembly of section .text:
```

```
00000175: <_start>:
175:    bb 08 00 00 00    mov     $0x8,%ebx
17a:    39 c3              cmp     %eax,%ebx
17c:    7e 02              jle     180 <L0>
17e:    43                inc     %ebx
17f:    c3                ret

00000180: <L0>:
180:    4b                dec     %ebx
181:    c3                ret
```

Symbols

# More with objdump

- Disassemble functions  
`objdump -d test`
- Disassemble everything  
`objdump -D test`
- Show headers  
`objdump -x test`
- Show symbols  
`objdump -t test`
- Show dynamically-loaded symbols  
`objdump -T test`
- ... and more...
- Also, readelf  
`readelf -a test`

# Disassembling with radare2

```
#> r2 ./ite
```

```
[0x00000175]> aaa
```

```
[x] Analyze all flags starting with sym. and entry0 (aa)
```

```
[x] Analyze len bytes of instructions for references (aar)
```

```
[x] Analyze function calls (aac)
```

```
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
```

```
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
```

```
[0x00000175]> pdf
```

```
    ;-- section_end..dynstr:
```

```
    ;-- section..text:
```

```
    ;-- _start:
```

```
/ (fcn) entry0 13
```

```
|  entry0 ();
```

```
|          0x00000175  bb08000000  mov ebx, 8    ; [6] va=0x175 sz=13 rwx=--r-x .text
```

```
|          0x0000017a  39c3          cmp ebx, eax
```

```
|          ,=< 0x0000017c  7e02          jle loc.L0
```

```
|          | 0x0000017e  43           inc ebx
```

```
|          | 0x0000017f  c3           ret
```

```
|          `-> ;-- L0:
```

```
|          | ; JMP XREF from 0x0000017c (entry0)
```

```
|          `-> 0x00000180  4b          dec ebx
```

```
\          0x00000181  c3          ret
```

```
[0x00000175]>
```

## pdf: Print Disassembly Functions

The principle of Radare2 is to use one-letter commands that can be combined into more advanced commands. For example, 'pd' (Print Disassembly) is a family of commands among which can be found the 'pdf' command. To get the list of all commands of the family, just do: 'pd?'

# References (1)



Intel Corporation.

*Intel 64 and IA-32 Architectures Optimization Reference Manual*,  
April 2012.



Intel Corporation.

*Intel 64 and IA-32 Architectures Software Developer's Manual*,  
August 2012.



Randall Hyde.

*Write Great Code: Understanding the Machine*, volume 1.  
NoStarch Press, 2004.



Randall Hyde.

*Write Great Code: Thinking Low-Level, Writing High-Level*,  
volume 2.  
NoStarch Press, 2006.

# References (2)



Randall Hyde.

*The Art of Assembly Language.*

Number ISBN: 978-1-59327-207-4. NoStarch Press, second edition, 2010.



Jon Larimer.

Intro to x64 reversing.

Talk at SummerCon'2011, NYC, USA, 2011.