Program Analysis

Lecture 03: *Machine Language II*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Moodle info
  - http://moodle.ruhr-uni-bochum.de/course/view.php?id=689 - `PA2011`
- Introduction to MASM, OllyDbg and IDA Pro at 10:30am
- Open positions for student helpers (*SHKs*)
- Next Wednesday: talk by @vxradius and @matrosov
  - See http://www.nds.rub.de/teaching/lectures/471/

Topics of Last Week

- Machine language / assembler
- Hardware basics
  - Von Neumann vs. Harvard
  - RISC vs. CISC
  - Load-Store architectures
- Intel x86 architecture
  - Registers

x86 Registers

Important x86 Registers

General Purpose Register

Register	Purpose
`eax`	Accumulator
`ebx`	Base Address
`ecx`	Counter
`edx`	Data
`esi`	Source
`edi`	Destination

Special Purpose Register

Register	Purpose
`esp`	Stack Pointer
`ebp`	Base Pointer
`eip`	Instruction Pointer
`eflags`	Status Flags

Important x86 Registers

General Purpose Register

Register	Purpose
`eax`	Accumulator
`ebx`	Base Address
`ecx`	Counter

Special Purpose Register

Register	Purpose
`esp`	Stack Pointer
`ebp`	Base Pointer
`eip`	Instruction Pointer

This are just the basic registers, many more registers (e.g., FPU, MMX, and SSE) exist, but they are typically not interesting for us.

General Purpose Registers

Register	Typical purpose
`eax`	Accumulator
`ebx`	Base address for addressing
`ecx`	Counter for loops, index
`edx`	I/O data, double-precision operations
`esi`	Memory address for string source
`edi`	Memory address for string destination

General Purpose Registers

Register	Typical purpose
`eax`	Accumulator
`ebx`	Base address for addressing

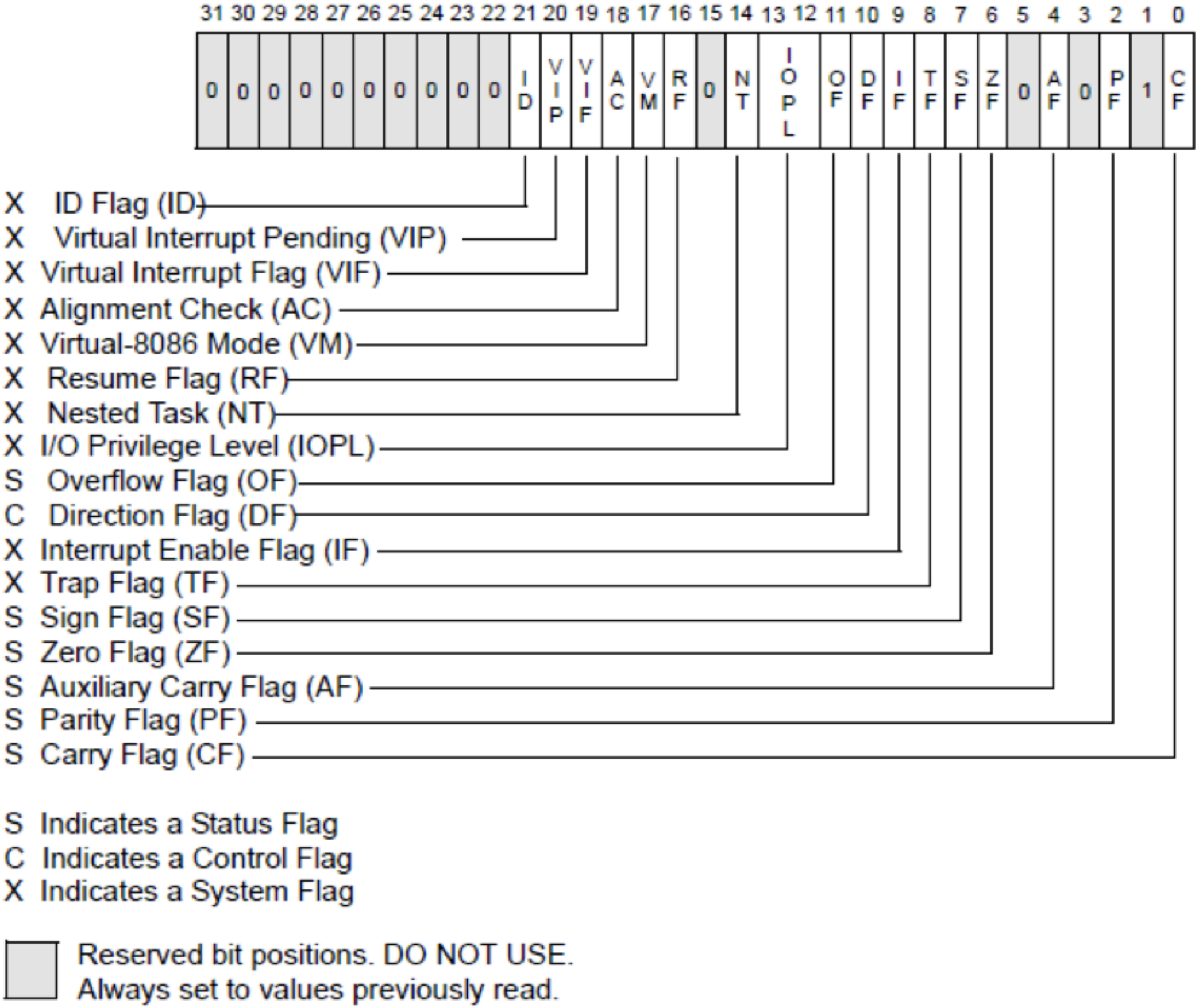
Note: This is just a convention, compilers and programmers can use the registers arbitrarily. However, several instructions expect the operand in specific registers, for example:

```
loopz _jumpdest
```

Special Purpose Registers

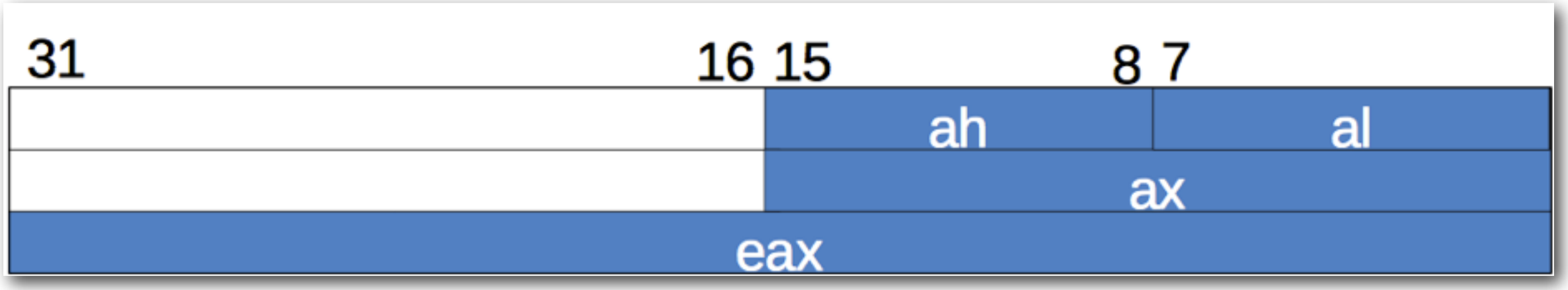
Register	Typical purpose
`esp`	Points to the next free stack element (**decreases**)
`ebp`	Points to current stack frame
`eip`	Points to current instruction
`eflags`	Different flags, typically used for conditional jumps (see next slide)

Note: `eip` and `eflags` registers can only be written and read *indirectly*


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

Register Addressing

- All registers are 32bit long
- Lower parts can be addressed explictly



- Same for `ebx`/`ecx`/`edx` registers
- Only 16 bit available for `esi`/`edi`/`ebp`/`esp` registers

Subregisters

- Modification of subregisters does not influence **higher-order** registers
- Example:

```
eax = 12345678h      // initialize EAX
ror ax, 8            // rotate right
⇒ eax = 12347856h  // value preserved
```

- Subregister are available to enable backward compatibility to 8/16bit programs

Other Registers

- Floating points: Register stack with 8 elements (80bit)
- MMX: `mmx0...mmx7` (64bit)
- SSE: `xmm0 ... xmm7` (128bit)
- Diverse number of control register
  - MMU controls
  - Debugging
- ... (many more documented in *Intel Manuals*)

x86 Instruction Set

Instruction Set

- Many different instructions (CISC)
- Commonly only a subset of the complete instruction set is used

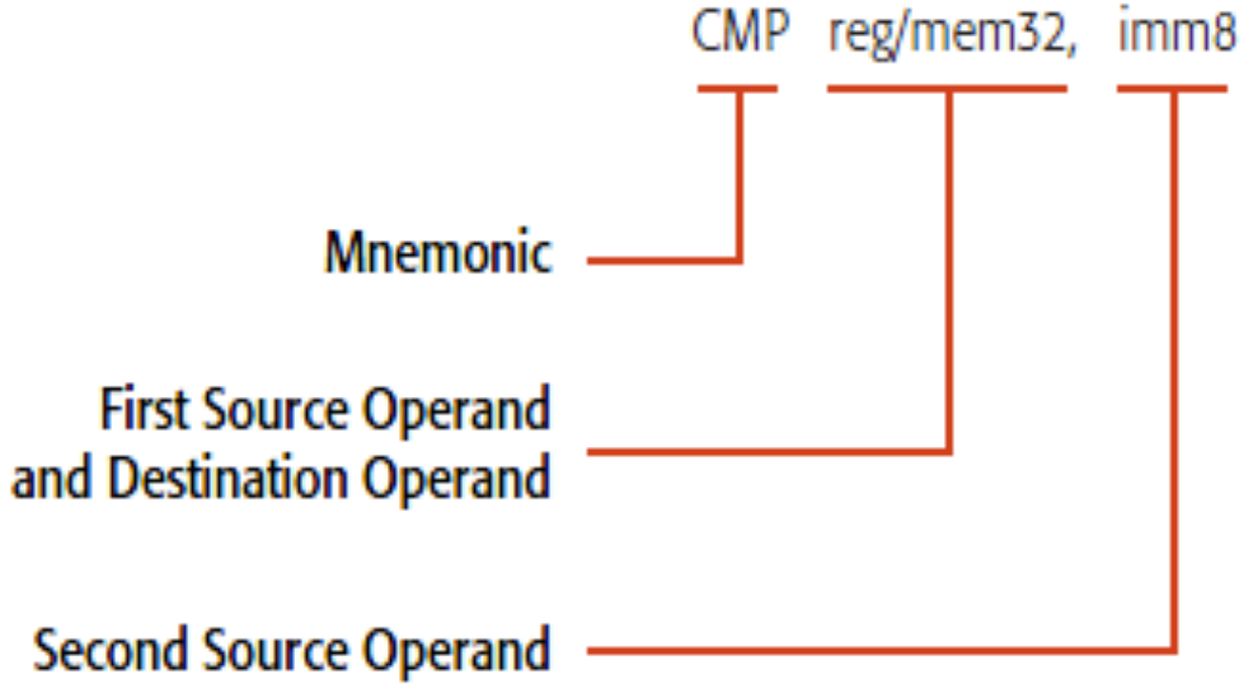
- Especially compiler often use only a small number of instructions
- Enables us to identify artifacts of compilers or to detect that a specific shellcode was presumably written by hand

Instruction Set

- Coarse classification
  - Memory access (read/write operations)
  - Arithmetic, logical and bitwise instructions
  - Subroutine instructions (call, return, ...)
  - Control transfer instructions (static jumps, conditional jumps)
  - String instructions (string comparison, ...)
  - ...

Instruction Layout

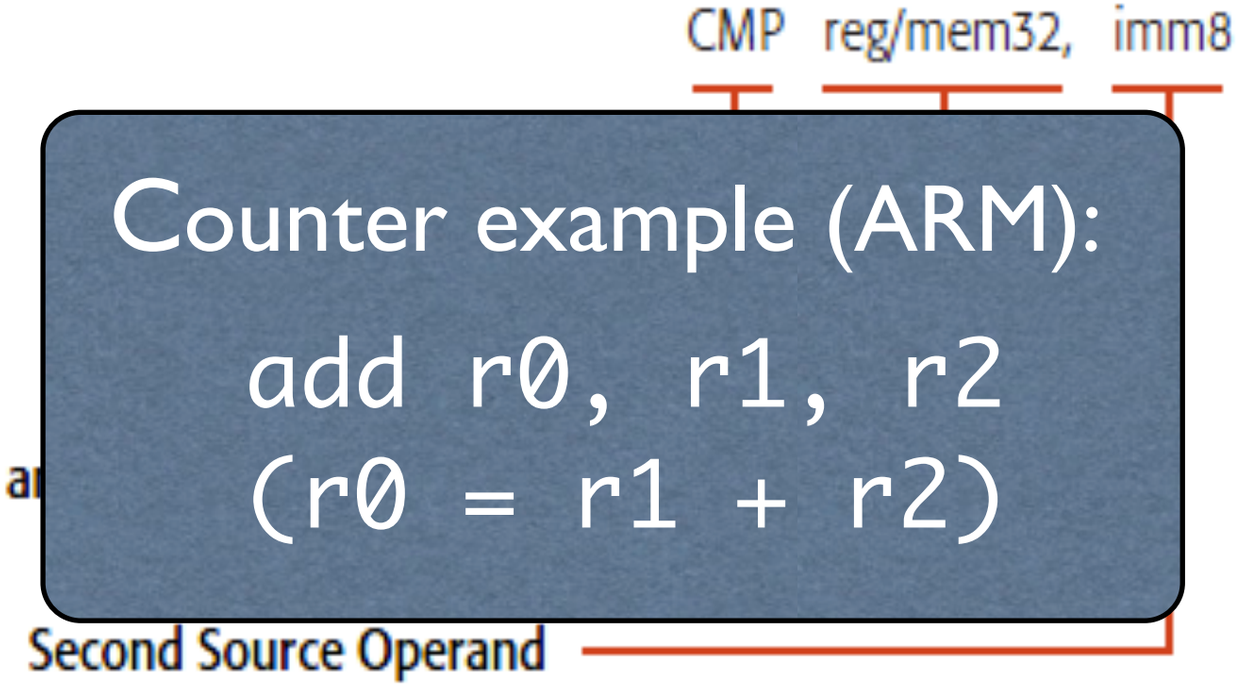
- Assembler instructions are displayed as *mnemonics* (German translation: *Gedächtnisstütze*, source is Greek *mnēmonikά - memory/Gedächtnis*)



- For x86, one source operand is always the destination

Instruction Layout

- Assembler instructions are displayed as *mnemonics* (German translation: *Gedächtnisstütze*, source is Greek *mnēmonikά - memory/Gedächtnis*)



- For x86, one source operand is always the destination

Syntax

- We use *Intel Syntax* in this class
  - Target operand **always** on the left
  - Example: `sub eax,ecx` means `eax = eax – ecx`
  - Size of operand is not explicitly mentioned, if it is clear from context: `mov eax, [1234]` (32bit)
  - Numbers are decimal. Hexadecimal is explicit with "h" at the end (`10h = 16`)

Syntax

- **Caution**: there is another convetion - *AT&T Syntax*
  - For example used by GNU asssembler as default
    - Example `subl %eax, %ecx`

      means `ecx = ecx - eax`
- We *always* use Intel syntax in this class

Syntax

	Intel Syntax	AT&T Syntax
Parameter order	Destination before source	Source before destination
Parameter size	Derived from register used (e.g., `eax`, `al`)	Mnemonics have size as suffix (e.g., l = dword)
Prefixes	Automatically detected	Constants with $, registers with %
Effective address	In [square brackets] `mov eax, dword [ebx + ecx*4 + mem_location]`	DISP(Base, Index, Scale) `movl mem_location(%ebx, %ecx,4), %eax`

Source: http://en.wikipedia.org/wiki/Intel_syntax#Syntax

Intel Syntax

```
push_loop:
 mov  [ebp - 4], ecx
 mov  edx, [ebp + 8]
 xor  eax, eax
 mov  al, byte [ebx + esi]
 push eax
 push edx
 call printf
 add  esp, 8
mov  ecx, [ebp - 4]
inc  esi
loop push_loop
push newline
call printf
add  esp, 4
mov  esp, ebp
pop  ebp
ret
```

AT&T Syntax

```
push_loop:
 movl  %ecx, -4(%ebp)
 movl  8(%ebp), %edx
 xorl  %eax, %eax
 movb  (%ebx, %esi, 1), %al
 pushl %eax
 pushl %edx
 call  printf
 addl  $8, %esp
movl  -4(%ebp), %ecx
incl  %esi
loop  push_loop
pushl $newline
call  printf
addl  $4, %esp
movl  %ebp, %esp
popl  %ebp
ret
```

Examples

Data transfer: mov	
`mov eax, ecx`	eax = ecx
`mov edx, [1234]`	edx = content of memory address 1234

Stack operations: push , pop	
`push ecx`	push value of ecx to stack and substract 4 from esp
`pop esi`	add 4 to esp and copy top of stack to esi

Examples

Arithmetic operations: `add`, `sub`, `mul`, `rol`, `shl`, ...	
`add eax, ecx`	eax = eax + ecx
`sub edx, ebx`	edx = edx - ebx
`mul ecx`	eax = eax * ecx
`div ebx`	eax = eax / ebx, edx = eax % ebx
`rol eax, 12`	Rotate eax 12 bit to left
`shl eax, 1`	Shift left (= multiply eax by 2)

Examples

Comparison/jumps: `cmp`, `jmp`, `jXX`	
`jmp 2342`	Set eip to 2342
`cmp eax, ecx`	Compare content of eax and ecx and set eflags accordingly
`jz 1234`	Set eip to 1234 if ZF (Zero Flag) is set (*jump zero*)
`jae 5678`	Set eip to 5678 if CF (Carry Flag) is not set (*jump above or equal*)

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF) = 0	Above/not below or equal
JAE/JNB	CF = 0	Above or equal/not below
JB/JNAE	CF = 1	Below/not above or equal
JBE/JNA	(CF or ZF) = 1	Below or equal/not above
JC	CF = 1	Carry
JE/JZ	ZF = 1	Equal/zero
JNC	CF = 0	Not carry
JNE/JNZ	ZF = 0	Not equal/not zero
JNP/JPO	PF = 0	Not parity/parity odd
JP/JPE	PF = 1	Parity/parity even
JCXZ	CX = 0	Register CX is zero
JECXZ	ECX = 0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) = 0	Greater/not less or equal
JGE/JNL	(SF xor OF) = 0	Greater or equal/not less
JL/JNGE	(SF xor OF) = 1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF) = 1	Less or equal/not greater
JNO	OF = 0	Not overflow
JNS	SF = 0	Not sign (non-negative)
JO	OF = 1	Overflow
JS	SF = 1	Sign (negative)

Examples

If (v >= 13) { … } else { … }	for (int i = 0; i < 10; i++) { … }

`  cmp [v], 13`
`  jb  _else`
`  …  ; if clause`
`  jmp _end`
`_else:`
`  …  ; else clause`
`_end:`
	`  mov ecx, 0`
	`_loop:`
	`  cmp ecx, 10`
	`  jae _end`
	`  …`
	`  add ecx, 1`
	`  jmp _loop`
	`_end:`

Examples

Subroutines: `call`, `ret` (*next lecture*)	
`call 2342`	Call function at address 2342, push address of next instruction to stack
`ret`	Return from subroutine, use the instruction pushed previously onto the stack (pop it from stack)

Example Programs


```
mov eax, [x]
mul 5682
shr eax, 16
add eax, 120948
mov [var], eax
```

Example Programs

var = (x * 5682) >> 16 + 120948

```
mov eax, [x]
mul 5682
shr eax, 16
add eax, 120948
mov [var], eax
```

Example Programs

var = (x * 5682) >> 16 + 120948

```
mov eax, [x]
mul 5682
shr eax, 16
add eax, 120948
mov [var], eax
```

```
  mov eax, 12345678 ; pointer to string
  mov ecx, 0
loop:
  cmp byte ptr[eax+ecx], 0
  jz  end
  inc ecx
  jmp loop
end:
```

Example Programs

var = (x * 5682) >> 16 + 120948

```
mov eax, [x]
mul 5682
shr eax, 16
add eax, 120948
mov [var], eax
```

strlen()

```
  mov eax, 12345678 ; pointer to string
  mov ecx, 0
loop:
  cmp byte ptr[eax+ecx], 0
  jz  end
  inc ecx
  jmp loop
end:
```

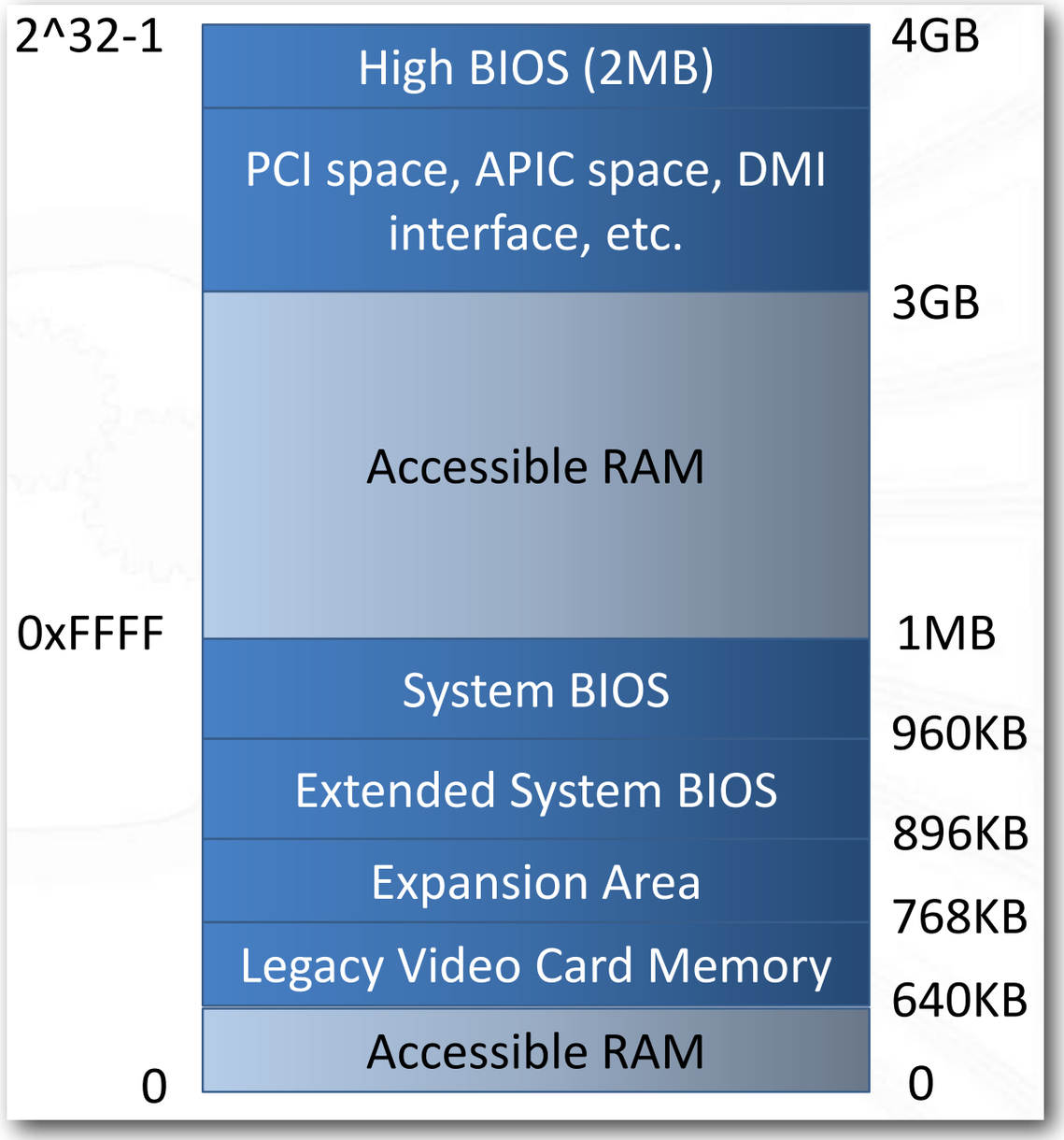
x86 Memory Access

Memory

- Linear, continuous address space
- Smallest addressable unit: 1 byte
- Divided in *physical*, *linear* and *virtual* address space
  - Not covered in detail in this class
- 32bit ⇒ at most 4 GB can be addressed
- Typically not the whole address space is allocated

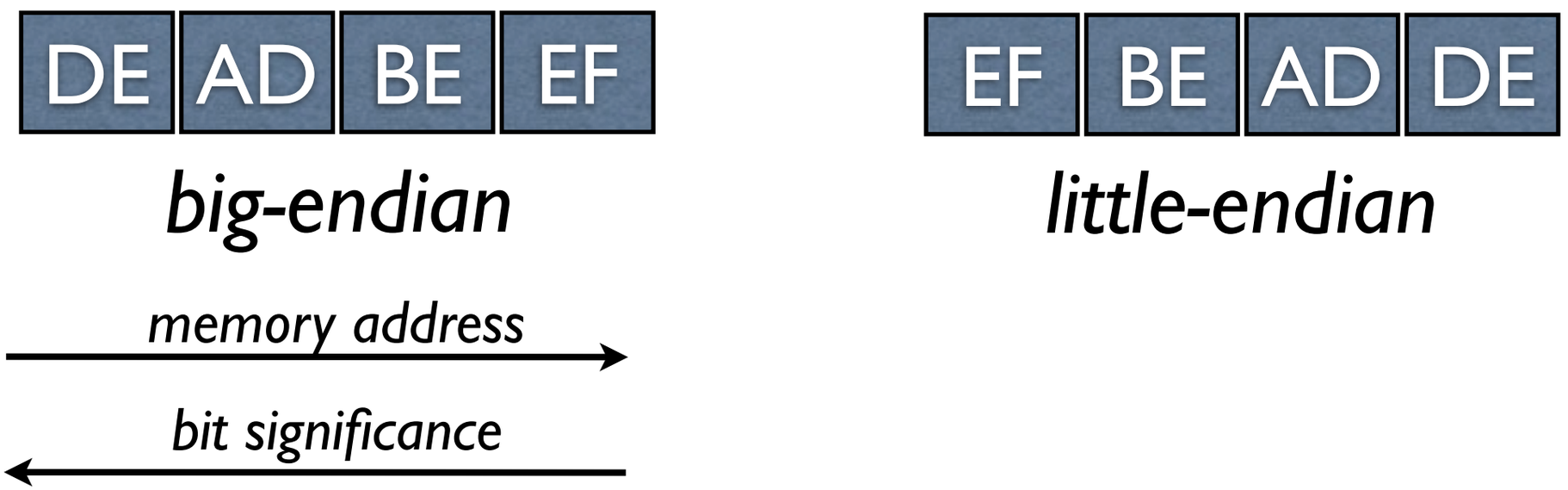
Physical Memory

- Initialized by BIOS
- *Interrupt Vector Table* (*IVT*) at address 0 (see later slides)
- In *real mode* only access to first 640 KB



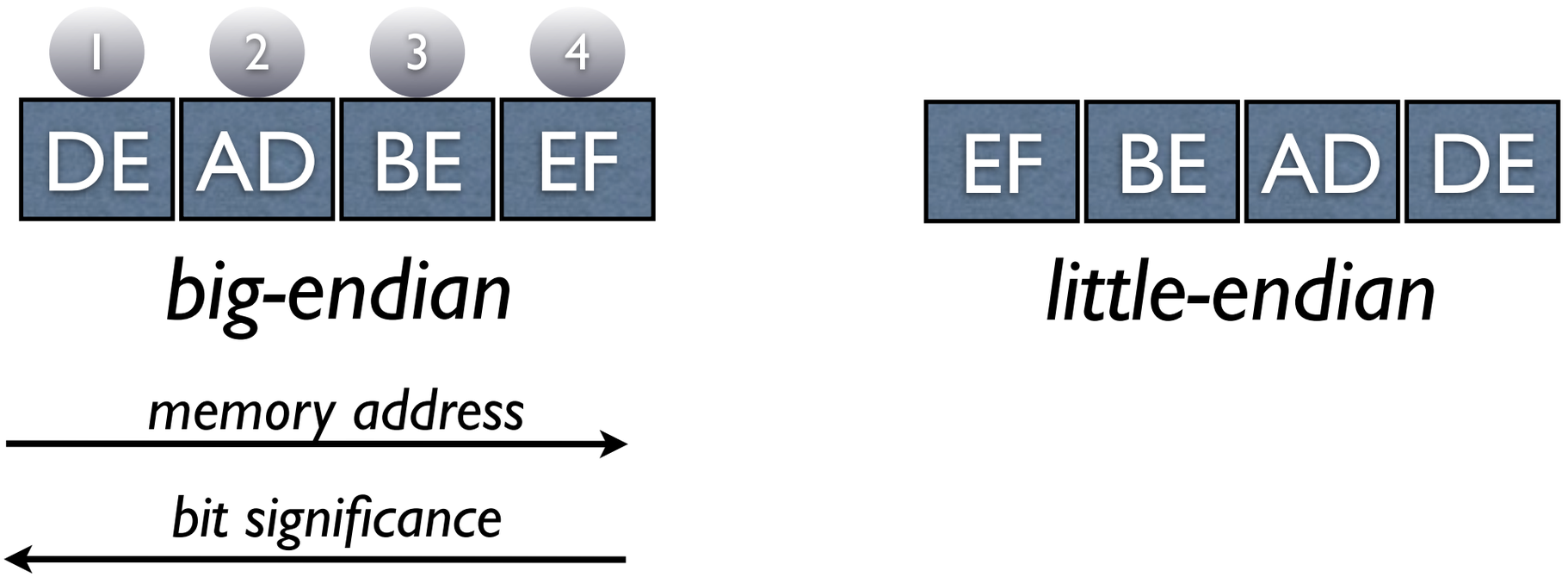
Endianness

- There are two different ways to address memory, *little-endian* and *big-endian*
- Basically it describes the way memory is read/written
- Example: Store value `DEADBEEF` (4 byte)



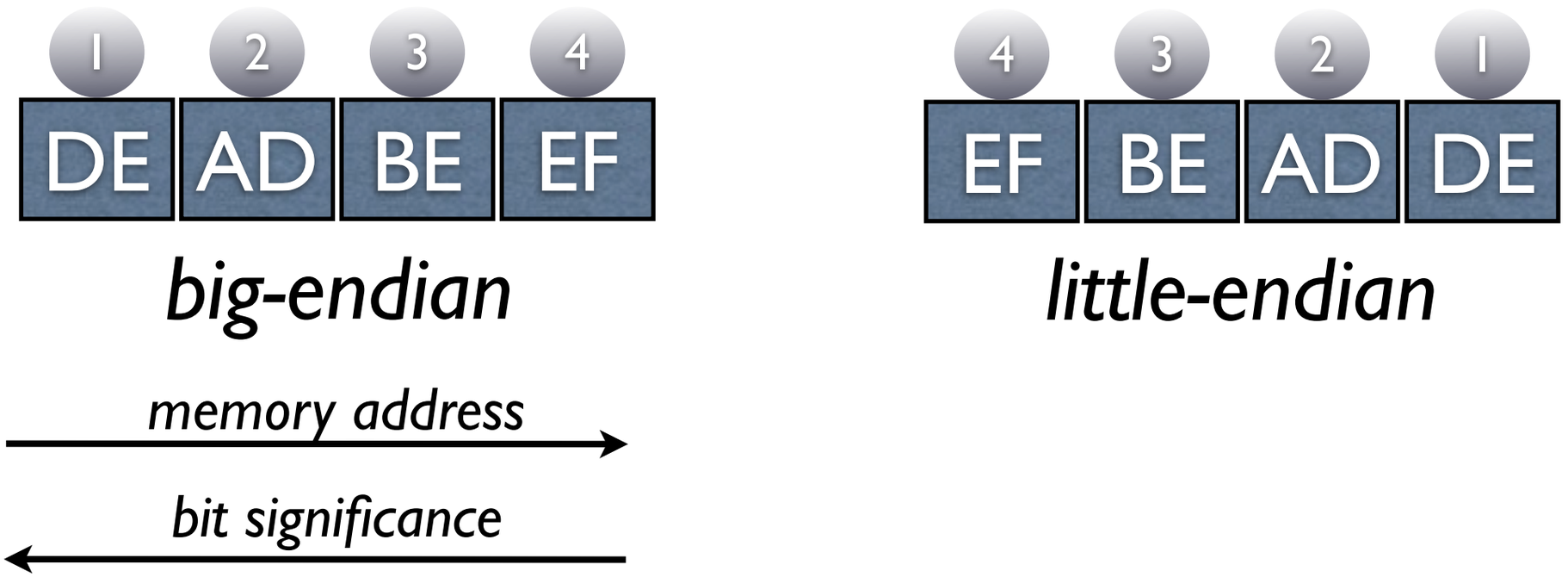
Endianness

- There are two different ways to address memory, *little-endian* and *big-endian*
- Basically it describes the way memory is read/written
- Example: Store value `DEADBEEF` (4 byte)



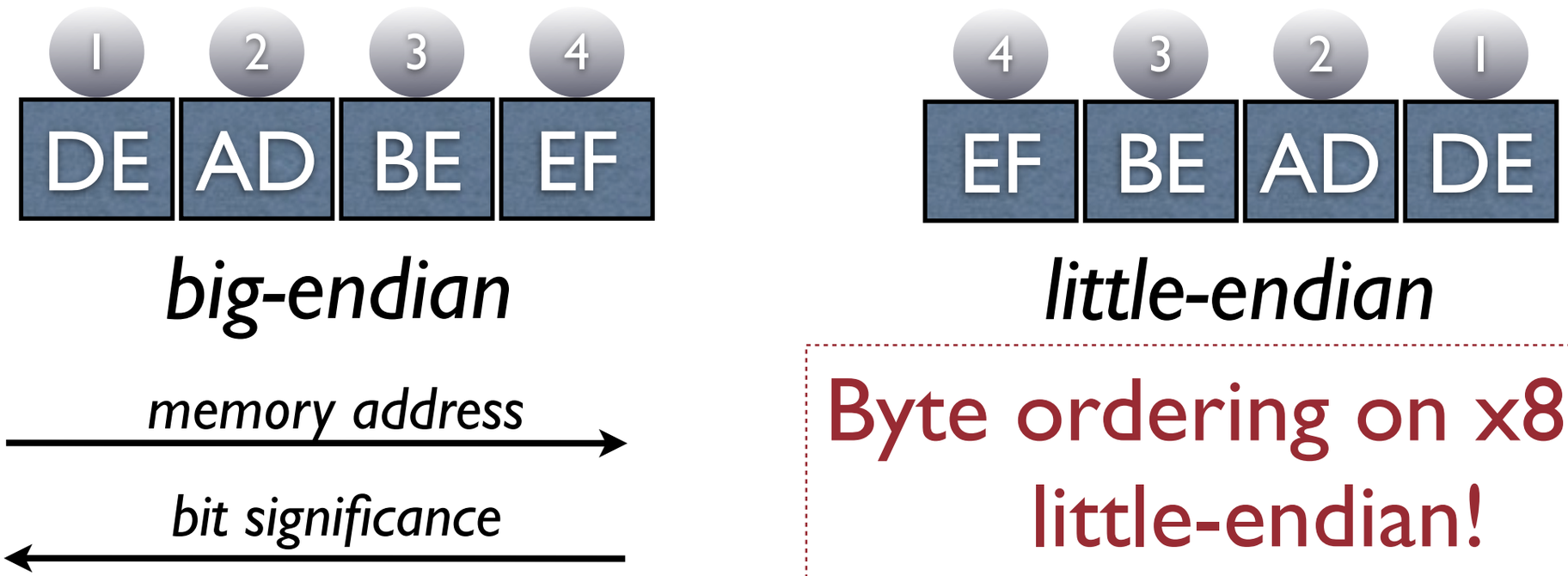
Endianness

- There are two different ways to address memory, *little-endian* and *big-endian*
- Basically it describes the way memory is read/written
- Example: Store value `DEADBEEF` (4 byte)



Endianness

- There are two different ways to address memory, *little-endian* and *big-endian*
- Basically it describes the way memory is read/written
- Example: Store value `DEADBEEF` (4 byte)



Byte ordering on x86 is little-endian!

Example: `mov var, 12345678h`

Before

Arbeitsspeicher 1

Adresse: 0x012E7164 | Spalten: Automatisch

```
0x012E7164  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 30 e9 1d  ................0é.
0x012E7177  68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  h..................
0x012E718A  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
0x012E719D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
0x012E71B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Überwachen 1

Name	Wert	Typ
&var	0x012e7164 int var	int *
var	0	int

Auto | Lokal | Threads | Module | Überwachen 1

After

Arbeitsspeicher 1

Adresse: 0x012E7164 | Spalten: Automatisch

```
0x012E7164  78 56 34 12 00 00 00 00 00 00 00 00 00 00 00 00 30 e9 2f  xV4.............0é/
0x012E7177  68 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  h..................
0x012E718A  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
0x012E719D  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
0x012E71B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...................
```

Überwachen 1

Name	Wert	Typ
&var	0x012e7164 int var	int *
var	305419896	int

Auto | Lokal | Threads | Module | Überwachen 1

Addressing

- Four different types of memory access
  - *Immediate*, *register*, *memory*, and *offset*
- Convention is to depict memory access in **[***square brackets***]**
- One instruction can perform **at most** one memory access
  - Remember what *load-store architecture* means

Examples

Instruction	Type	Explanation
`mov eax, 1337`	Immediate	Load constant into eax
`mov eax, ebx`	Register	Load ebx into eax
`mov eax, [2342]`	Memory	Load content from static memory address
`mov esi, [edi]`	Offset	Load content from memory address via pointer in register

`mov eax, [ebx + ecx]`
`mov eax, [ebx + 1234]`
`mov eax, [ebx + ecx*4 - 1337]`
	Offset	Other variants of the same construct

Examples

Instruction	Type	Explanation
`mov eax, 1337`	Immediate	Load constant into eax
`mov eax, ebx`	Register	Load ebx into eax
`mov eax, [2342]`		from static memory
`mov esi, [edi]`		from memory address register

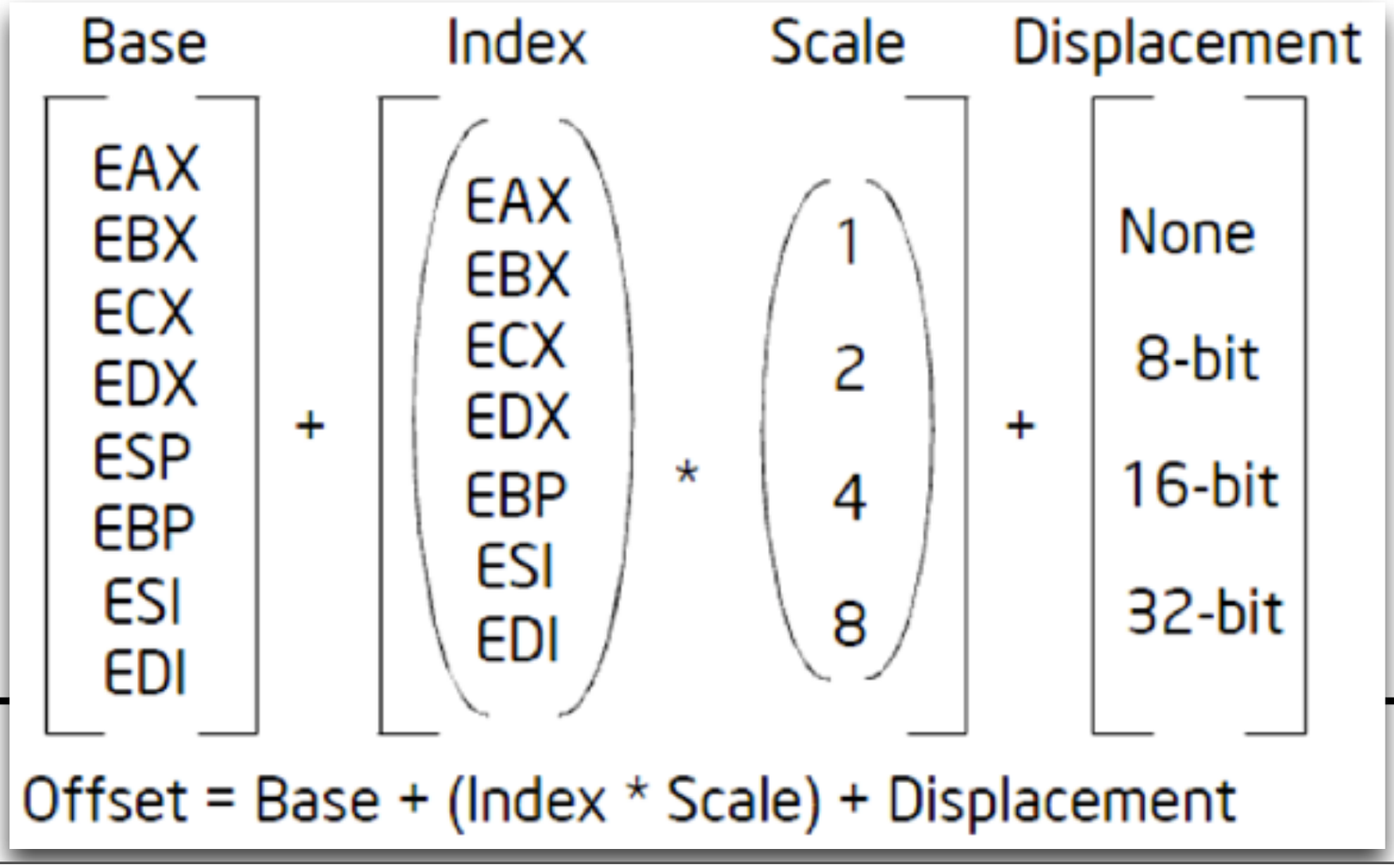
`mov eax, [ebx + ecx]`
`mov eax, [ebx + 1234]`
`mov eax, [ebx + ecx*4 - 1337]`
	Offset	Other variants of the same construct

Not possible!

`mov [esi], [edi]`

Addressing

- Actual memory destination when using offsets is called *effective address*
- Computed based on base (`reg`), index * scale (`reg` * `1,2,3,4`), and displacement (32bit)



Example I

Addition of two variables

```
mov eax, [1337]
mov ecx, [1234]
add eax, ecx
mov [1337], eax
```

Alternative implementation

```
mov eax, 1337
mov ecx, [1234]
add [eax], ecx
```

Example I

Addition of two variables

```
mov eax, [1337]
mov ecx, [1234]
add eax, ecx
mov [1337], eax
```

Alternative implementation

```
mov eax, 1337
mov ecx, [1234]
add [eax], ecx
```

Memory access is possible in each instruction, not only for `mov` (no load-store architecture!)

Example II

int a[256]; return a[i];

```
mov ebx, offset a
mov ecx, [i]
mov eax, [ebx+ecx*4]
```

Example II

int a[256]; return a[i];

```
mov ebx, offset a
mov ecx, [i]
mov eax, [ebx+ecx*4]
```

Complex example

```
mov eax, [eax+ecx*4+12345678h]
```

Example II

int a[256]; return a[i];

```
mov ebx, offset a
mov ecx, [i]
mov eax, [ebx+ecx*4]
```

Complex example

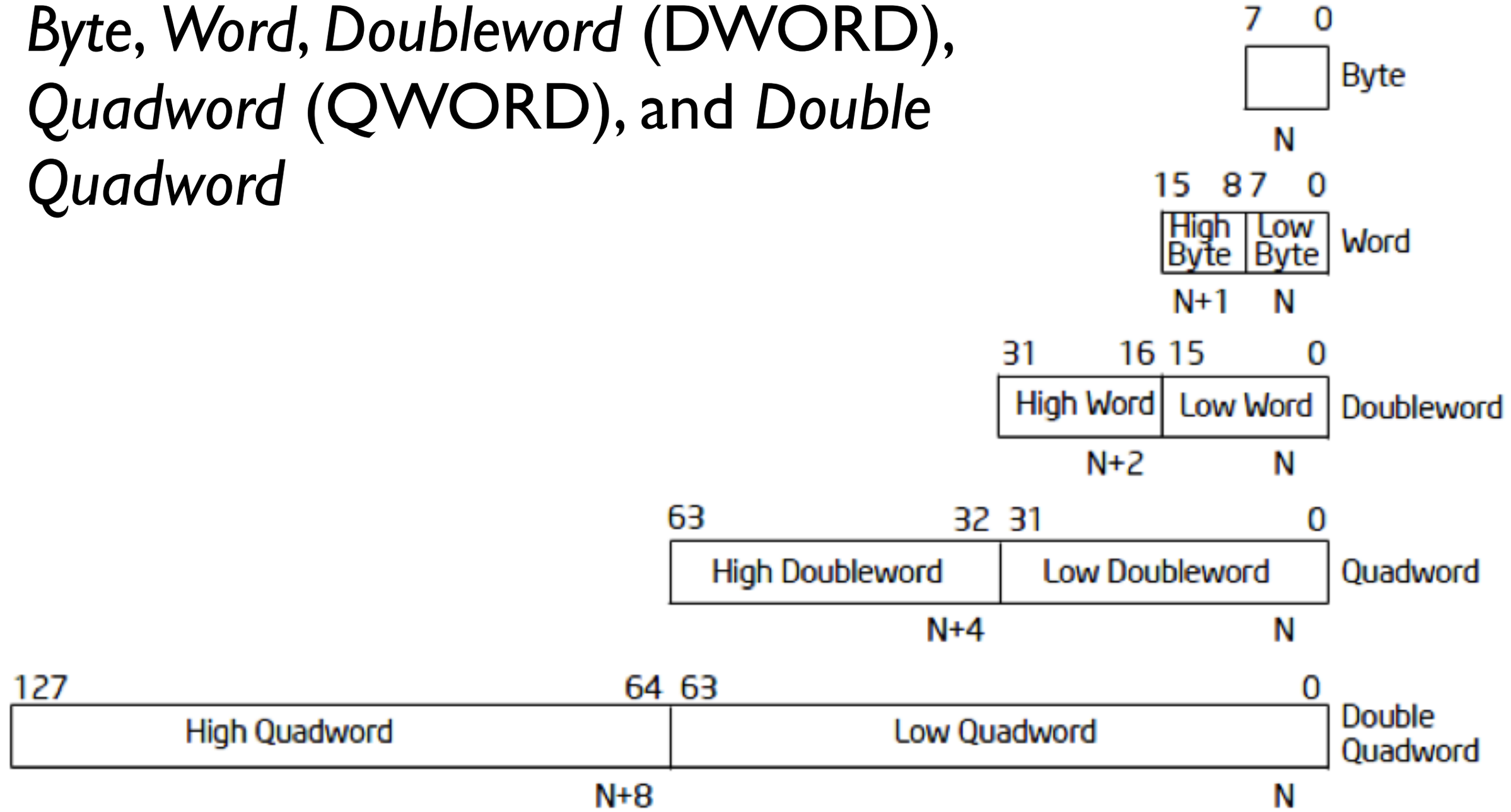
```
mov eax, [eax+ecx*4+12345678h]
```

Load effective address

```
lea eax, [ecx+edx]
```

Data Sizes

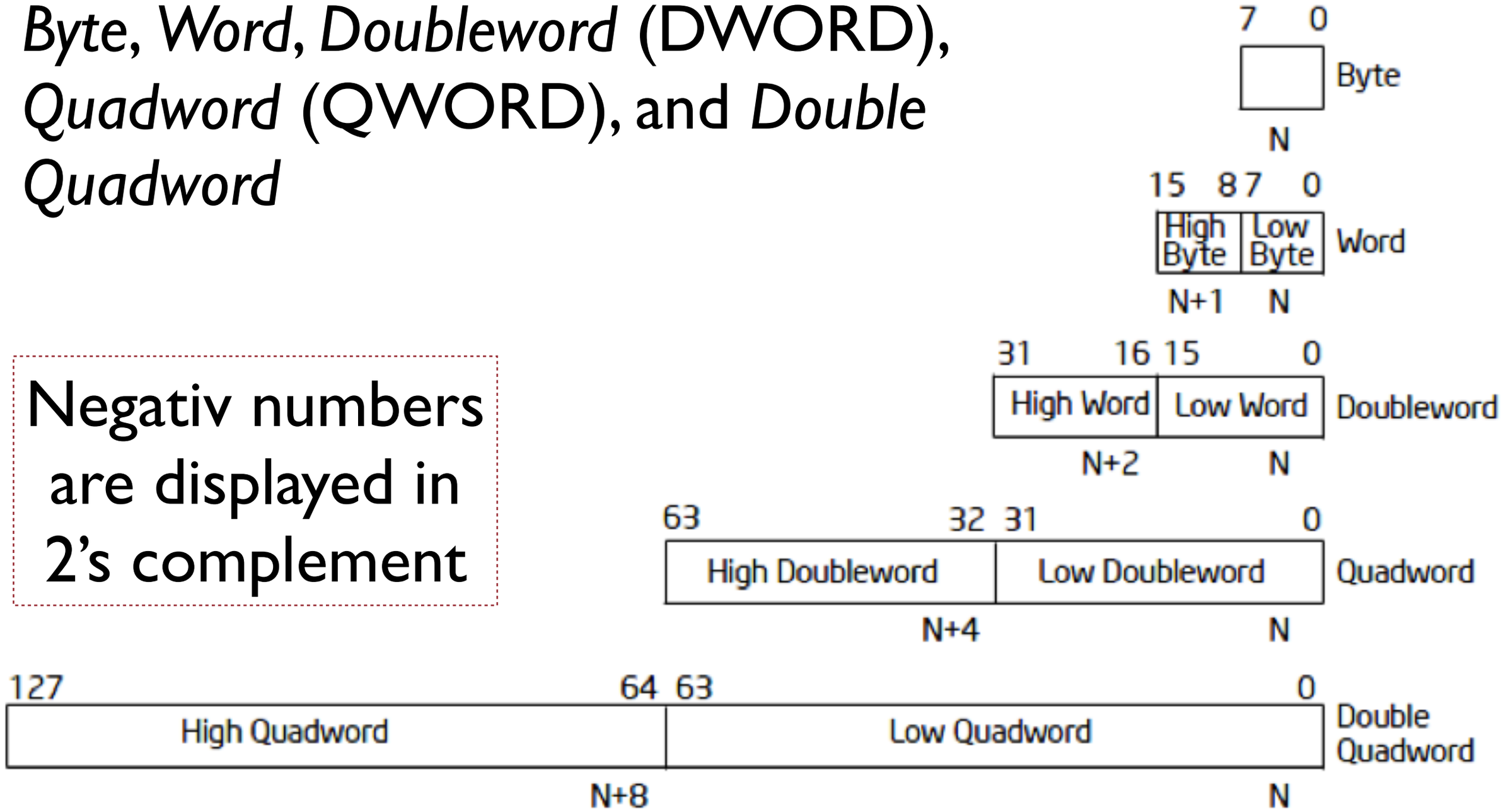
Byte, *Word*, *Doubleword* (DWORD), *Quadword* (QWORD), and *Double Quadword*



Data Sizes

Byte, *Word*, *Doubleword* (DWORD), *Quadword* (QWORD), and *Double Quadword*

Negativ numbers are displayed in 2's complement



Privileges

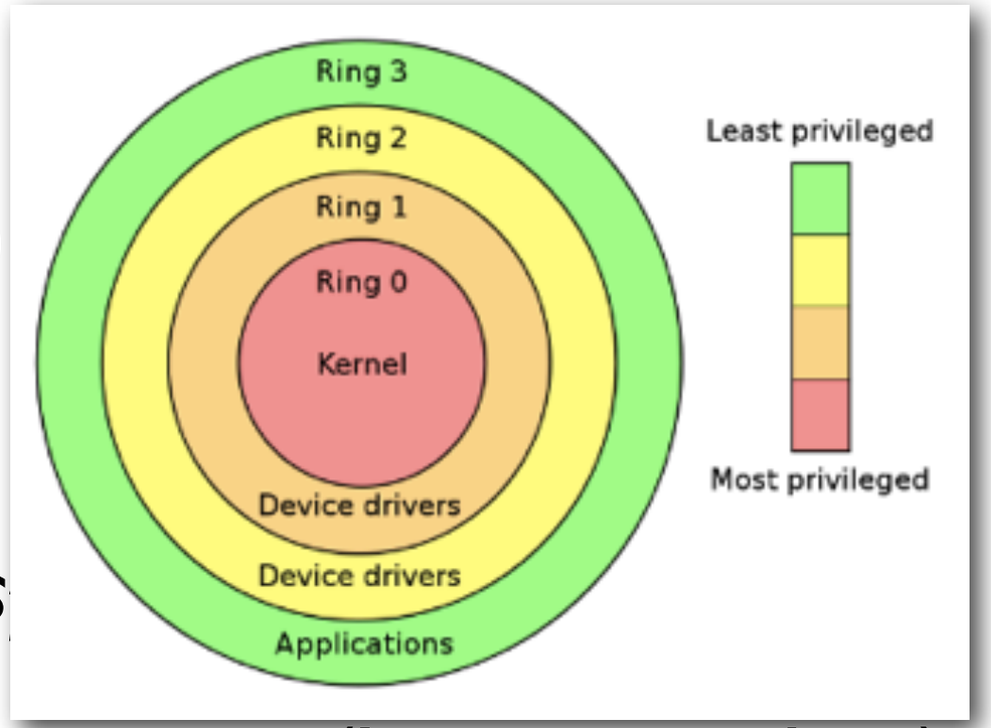
- There are several privilege levels, so called *rings*
- Each instruction is executed in a specific ring
- Some instructions are only valid in certain rings
- Memory access can be based on privilege system
  - Access to certain memory areas only possible in certain rings

Privileges

- There are four rings
  - But effectively only ring 0 and 3 are used
- **Ring 0**: Kernel and drivers (*kernel space*)
- **Ring 3**: Applications (*user space*)
- Current ring is saved in `cs` register (lower two bits)
- Transfer in other rings via *system calls*, *interrupts* and *exceptions*

Privileges

- There are four rings
  - But effectively only ring 0
- **Ring 0**: Kernel and drivers
- **Ring 3**: Applications (*user s*
- Current ring is saved in `cs` register (lower two bits)
- Transfer in other rings via *system calls*, *interrupts* and *exceptions*



Interrupts

- Exceptions are often simply called interrupts, handling of both is identical
- Interrupts and exceptions are handled by specific handlers that take care of appropriate actions
- Central data structure: *Interrupt Vector Table* (*IVT*)
- The IVT keeps a pointer to the specific handler for each type of interrupt

Interrupt Vector Table

Vector	Description
0	Integer Divide-by-Zero exception
1	Debug exception
2	Non-maskable interrupt
3	breakpoint exception (INT 3)
4	overflow exception (INTO instruction)
5	Bound-range exception (BOUND instruction)
6	Invalid opcode exception
7	Device not available exception
...	...
13	General protection exception
14	Page fault exception
...	...
20 – 255	Not specified, can be used by OS

Questions?

Contact:
Prof. Thorsten Holz
thorsten.holz@rub.de
@thorstenholz on Twitter

More information:
http://syssec.rub.de
http://moodle.rub.de

Sources

- Lecture *Software Reverse Engineering* at University of Mannheim, spring term 2010 (Ralf Hund, Carsten Willems and Felix Freiling)