X86 ASSEMBLY

MALWARE ANALYSIS AND INCIDENT FORENSICS

M.Sc. in Cyber Security

MALWARE ANALYSIS

M.Sc. in Engineering in Computer Science

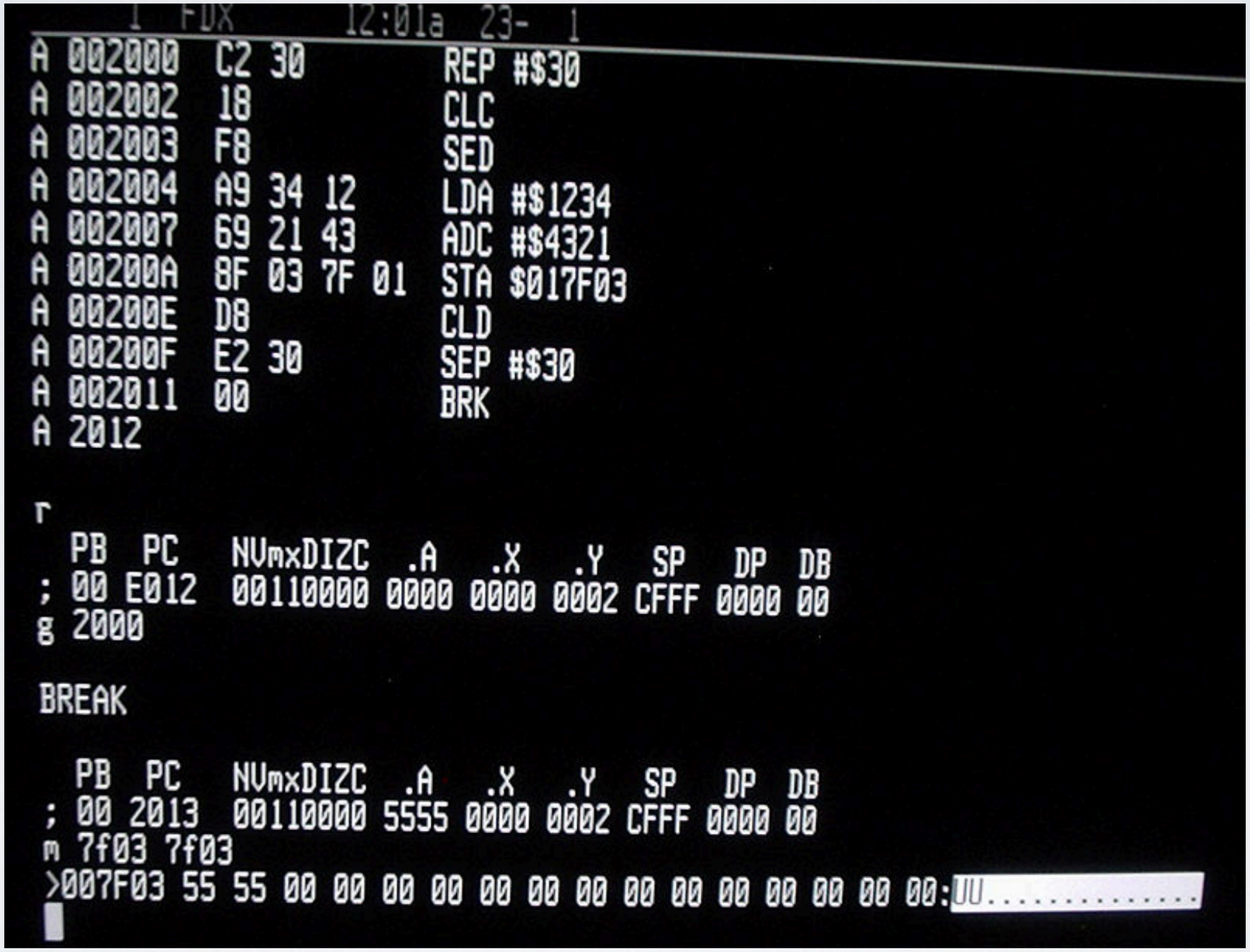
A.Y. 2025/2026

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

ANALYZING BINARY CODE

Daily routine for malware analysts 🧐



ANALYZING BINARY CODE

Reverse engineering of binary code is difficult.

Thankfully, only some portions of a malware sample really require it

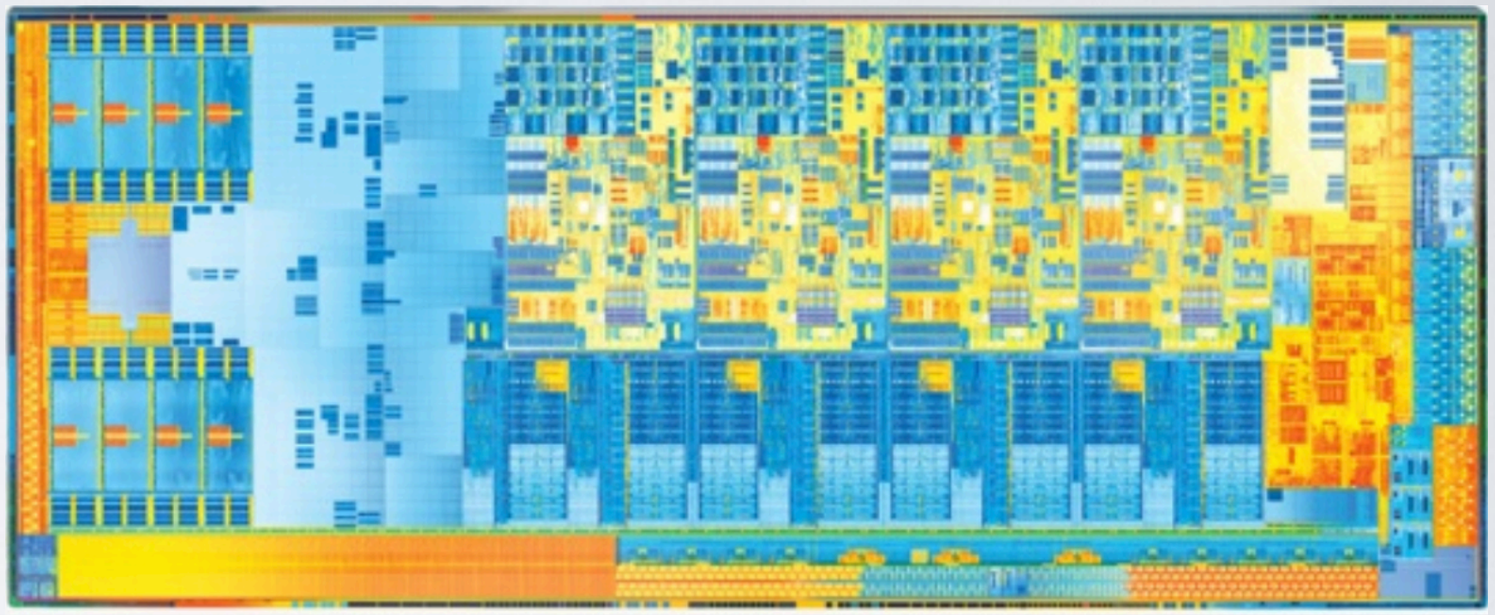
FACT: malicious code ≠ compiler-generated code

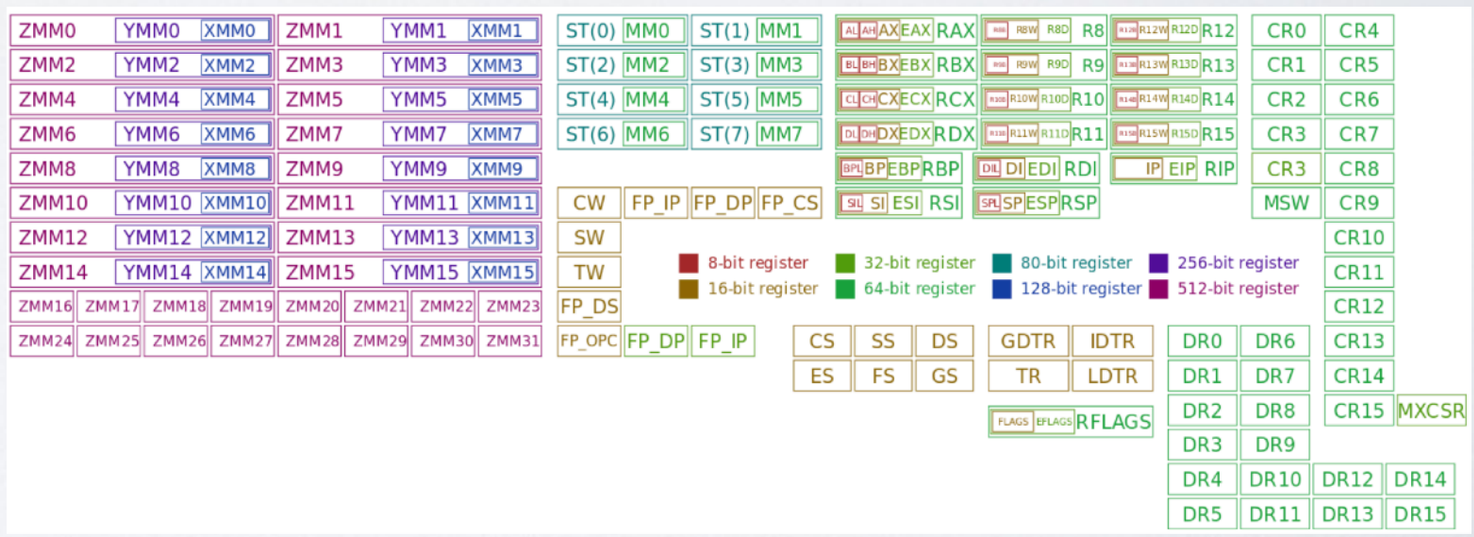
- hand-written sequences
- self-modifying code
- subtle semantics (e.g., jump in the middle of an instruction)
- obfuscation of control and data flows
- other decoys

DECOMPILATION?

- Decompilers (e.g., Ghidra, Hex-Rays) try to reconstruct a high-level C-like representation of a binary code fragment
- Process far from perfect
  - information loss during compilation (e.g., types, identifiers)
  - decompiling obfuscated code returns an obfuscated source
  - anti-analysis techniques
- Nevertheless, helpful for reverse-engineering professionals

CPU AND MEMORY BASICS



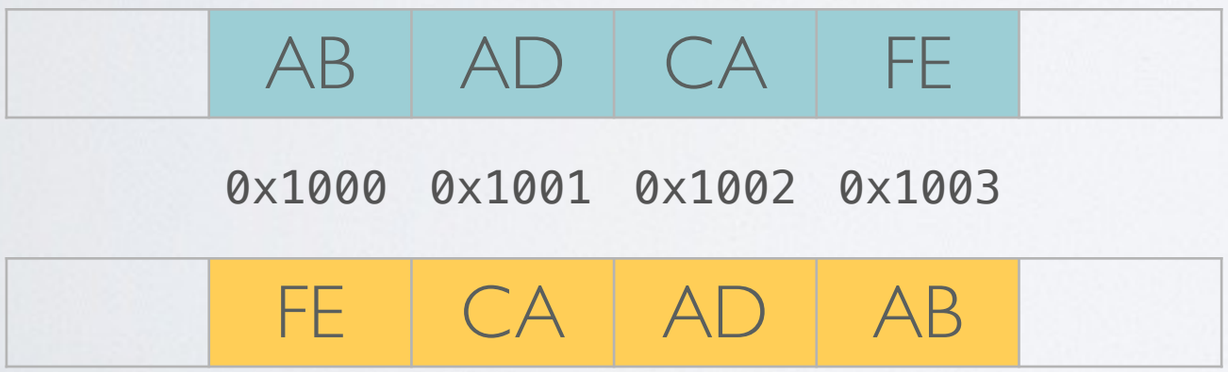


ISA AND X86

- **Instruction Set Architecture** (ISA) = abstract model describing what a programmer should know to program the machine
- ISA ≈ interface between software and hardware
  - Humans or compilers produce ISA-conformant **assembly** code, then an assembler encodes it in a **binary** format understood by hardware
- Intel defined the **x86** ISA for its 8086 processor (1976-1978)
  - Later extensions preserved backward compatibility
  - Today most malware is still 32-bit
  - x86 is also known as IA-32
  - x86-64 (x64 for short) is the 64-bit successor

WORD SIZE AND ENDIANNESS

- A **word** is the natural data unit for a specific CPU design. For x86 processors, the word size is 32 bits
  - However, for backward compatibility, x86 instructions assume a **word** operand to be 16-bit long, while a **dword** operand is 32-bit long
  - IA32 common data types: byte, word, dword
- Memory always operates at byte level. **Endianness** specifies how multi-byte sequences are read from/written to memory



Say we want to store `dword 0xABADCAFE` *at address* `0x1000`*...*

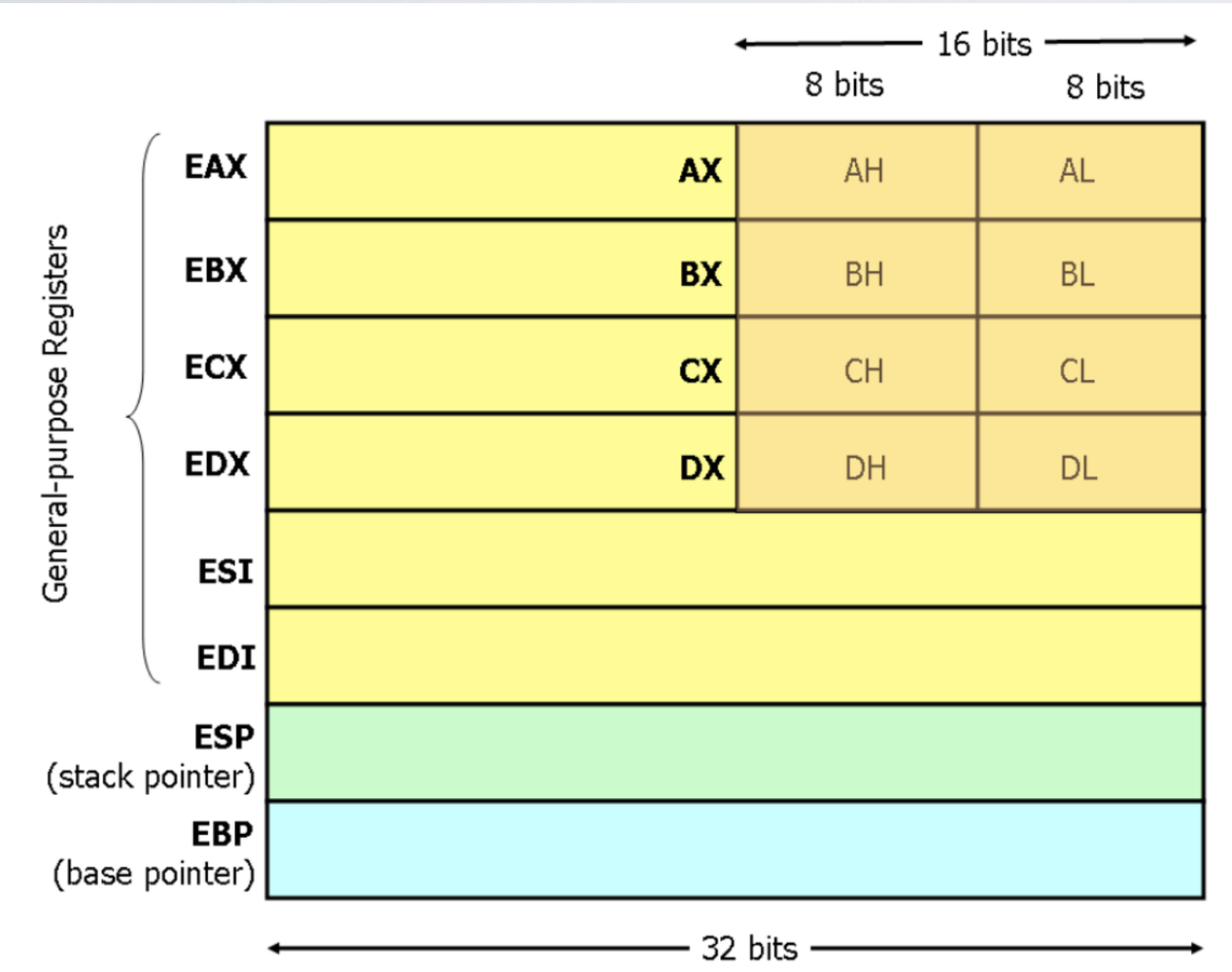
- **Big Endian**: most significant byte first
- **Little Endian**: least significant byte first

x86 uses Little Endian encoding

REGISTERS

- Coders use (many) variables. CPUs work on (few) **registers**
- Registers hold data with fast access time
  - *General-purpose registers* (GPRs) can store data or memory addresses
  - *Status register* holds truth values on the state of the processor after the last executed instruction (used, for example, in conditional computations)
  - *Program counter* holds the address of the instruction being executed
  - Many other registers
- x86 general-purpose registers are 32-bit wide

ACCESSING REGISTERS



Credits: Yale's CS421

8 general-purpose registers

- A, B, C, D, SI, DI
- ESP points to top of the stack
- EBP often set := ESP when a function begins (otherwise is available)

Prefix **E**AX indicates 32-bit size. When E is absent, only the least significant 16 bits (2 bytes) get manipulated. The two least significant bytes of A, B, C, and D can also be accessed individually

Program counter EIP and status register EFLAGS cannot be accessed directly

INSTRUCTION CYCLE

- Fetch-decode-execute cycle
  - *Fetch*: CPU reads instruction from address stored in EIP
  - *Decode:* control unit determines meaning of the instruction
  - *Execute:* carry out computation using ALU or move data
  - Then EIP advances to the next adjacent instruction, unless current instruction explicitly altered the control flow (e.g., with a jump)
- Instructions vary in size (1-15 bytes) and are stored consecutively in memory along with any **immediate operands** (i.e., data or addresses) they might use. Register operands are embedded instead in the **opcode** binary representation:

```
b8 01 00 00 00    mov eax, 0x1
bf 01 00 00 00    mov edi, 0x1
```

MEMORY ADDRESSING

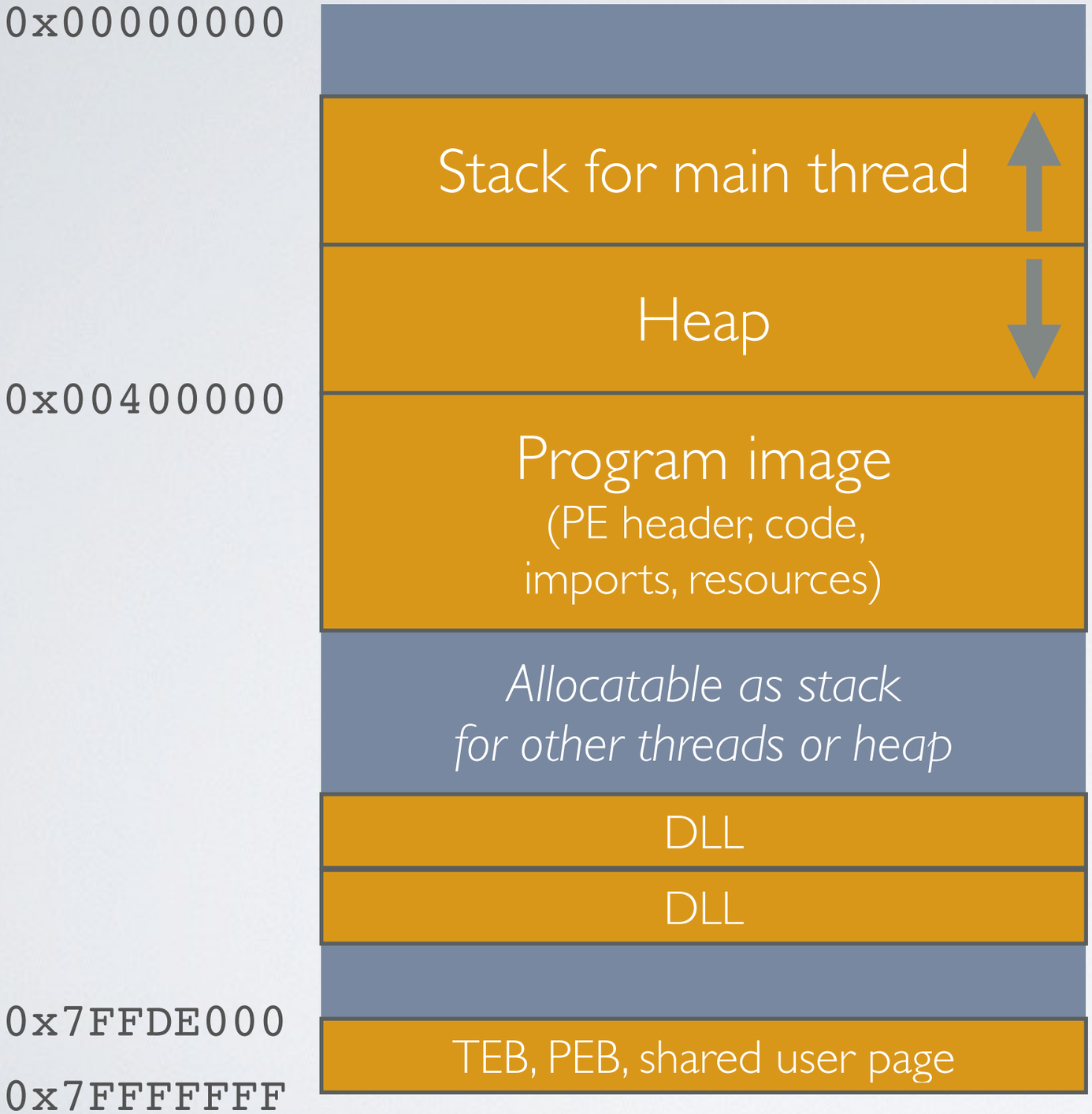
- **Addressing modes** provide a way to express the addresses of data to be read from/written to the main memory
- Expressions can take immediate operands, registers, or both
- Addressing modes enable operand combinations to compute complex expressions that depend on the program state
- Usually, an instruction can have at most one memory operand

COMMON ADDRESSING MODES

Some popular ways to specify data to be read from/written to the main memory. For the last two rows, the *offset* field is optional.

Mode	Intel syntax
Immediate	mov eax, [0x1000]
Register	mov eax, [esi]
Register + offset	mov eax, [esp-8]
Register*width + offset	mov eax, [ebx*4+0xff]
Base + Reg*width + offset	mov eax, [edx+ebx*4+8]

MEMORY MAP OF A PROCESS



Simplified view of a 32-bit Windows program

- lower 2GB for user space, upper 2GB (not shown) for kernel code
- program image = PE header + .text + .rdata +.data + .rsrc
- **stack grows to low addresses**
- DLLs have their own areas
- fixed addresses for first Thread Entry Block, Process Entry Block, and a special page that the kernel shares with userland

MEMORY PROTECTION

- **Paging** mechanisms
  - determine whether an address is valid
  - enforce read/write/execute permissions (granularity: regions)
  - pages in kernel memory are accessible only from kernel code
- Operating systems may enforce high-level security mechanisms to hinder memory vulnerability exploitation attacks
  - DEP (Data Execution Prevention)
  - ASLR (Address Space Layout Randomization)
  - Heap allocation randomization & other protections in Windows 10+

BASIC X86 INSTRUCTIONS

- Programs are made of three kinds of instructions
  - **Data movement**
  - **Arithmetic & logic**
  - **Program flow control**
- Data movement instructions are straightforward
  - **mov** dest, src
    Copy {register, memory content, immediate} to {register, memory location}
  - **push** src
    pop dest
    Used for stack manipulation (we will talk about them in a while…)

ARITHMETIC INSTRUCTIONS

- These instructions are relevant for code analysis for many reasons
  - computing offsets and function addresses
  - stack pointer modification
  - used also for data movement or to update EFLAGS

add dest, src	dest += src
sub dest, src	dest -= src
inc dest	++dest
dec dest	—dest
not dest	1's complement
neg dest	2's complement

LOGIC INSTRUCTIONS

- Common instances
  - zero-ing registers: `xor eax, eax`
  - extracting bits and checking conditions
  - obfuscation
- Beware that these operations are **bit-wise**. High-level logical operations such as && and || have a very different semantics

and dest, src	bitwise AND
or dest, src	bitwise OR
xor dest, src	bitwise XOR

ROTATE/SHIFT INSTRUCTIONS

Frequently encountered in bit manipulations:

shr dest, src **shl** dest, src	Unsigned shift (right/left)
sar dest, src **sal** dest, src	Signed shift
ror dest, src **rol** dest, src	Rotate
rcr dest, src **rcl** dest, src	Rotate with carry

(when src is absent, a default of 1 is assumed)

PROGRAM FLOW CONTROL

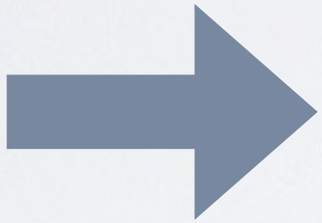
- Control flow can be implemented in three ways
  - **Unconditional branch**: EIP is replaced with some desired address
  - **Conditional branch**: EIP is replaced with some desired address depending on the value of one or more bits from EFLAGS
  - **Function calls and returns**: special type of unconditional branches
- Two primitives needed:
  - a way to specify the target of a branch (static or dynamic)
  - a way to evaluate a condition on the program state
- Code locations in assembly can be annotated with labels

UNCONDITIONAL BRANCHING

An unconditional branch can take as destination:

- an **offset**, specified as a relative offset from the current EIP value or as an absolute offset (from the base of the current code segment…)
- or an **absolute address**, provided as a register or memory operand

```
jmp 0x4
jmp L
sub eax, 0
jmp eax
L:
add eax, 0
jmp [eax]
```



```
0:   e9 00 00 00 00          jmp    5 <_main+0x5>
5:   eb 05                   jmp    c <L>
7:   83 e8 00                sub    eax,0x0
a:   ff e0                   jmp    eax
0000000c <L>:
c:   83 c0 00                add    eax,0x0
f:   ff 20                   jmp    DWORD PTR [eax]
```

Absolute addresses are straightforward, offsets may be not 🙂 [7+5=c above]

CONDITIONAL BRANCHING

- Conditional branches evaluate conditions over selected bits of the EFLAGS register. Common cases:
  - **CF** (carry flag), meant for errors in unsigned arithmetics
  - **OF** (overflow flag), meant for errors in signed arithmetics
  - **SF** (sign flag), set when the result of an operation is negative
  - **ZF** (zero flag), set when the result of an operation is zero
- We can use the **cmp** instruction to compare two values
  - `cmp dest, src` computes the difference dest-src without modifying dest, and sets the flags above in a way that enables many useful comparisons…

CONDITIONAL BRANCHING

- A **jcc** instruction checks the condition specified by **cc** suffix and jumps to the given offset accordingly (no absolute addresses)
- Say we just executed `cmp D, S`:

	Jump to L if	Negated version
je L ; **jz** L	D**==**S	[D**!=**S] jne L ; jnz L
jg L ; **jnle** L	D**>**S	[D**<=**S] jng L; jle L
jge L ; **jnl** L	D**>=**S	[D**<**S] jnge L; jl L
ja L ; **jnbe** L	> with unsigned operands	jna L ; jbe L
jae L ; **jnb** L	>= with unsigned operands	jnae L ; jb L

OTHER COMPARISONS

- Another common comparison instruction is **test**
  - it computes the bitwise AND of its operands without altering them
  - say we want to check whether register D holds zero as value: **`test D, D`** will set the zero flag iff. D==0, then **jz** can be used
- But malware won't use only cmp and test
  - Say, you can decrement a value with `dec` or `sub` and then check if the instruction set the zero flag with `jz` 🤧 (also optimizing compilers do that!)

INTEL VS. AT&T SYNTAX

x86 assembly has two main syntax types: AT&T and Intel. Intel syntax is dominant in the Windows world. Relevant differences:

	Intel	AT&T
Parameter order	Destination, then source	Source, then destination
Parameter size	Derived from register that is used	Suffix (b, w, l)
Sigils	Autodetect	$ for immediates, % for register names
Addresses	[expression] ; add byte/word/dword when ambiguous	offset(base, idx, width)

FUNCTIONS AND STACK FRAMES

- **Function** ≈ unit of code that controls register values and its portion of stack independently of other units
  - if a code unit calls another, the latter should not clobber registers in use
  - CPUs come with few general-purpose registers: one can *spill* (i.e., save) some values on the stack and fetch them later in the execution as needed
- The stack is (conceptually) divided into **frames**, one per currently active function. A frame usually includes:
  - the arguments for the function invocation
  - local function variables and other storage
  - return address for the call (i.e., where to resume execution upon function exit)

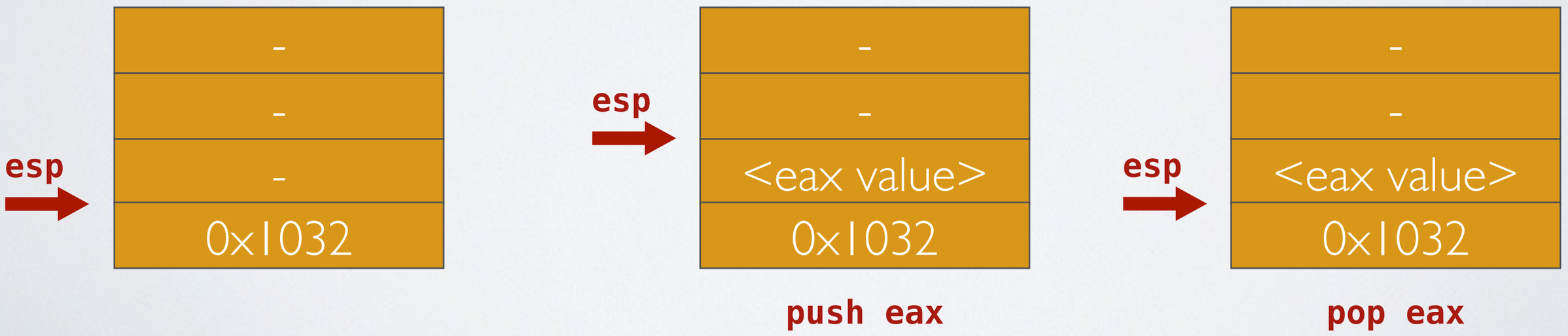
LOCAL STORAGE

- Suppose we don't have free registers left and we would like to add to EDX the difference between EBX and ECX:

```
mov TMP, eax
mov eax, ebx
sub eax, ecx
add edx, eax
mov eax, TMP
```

```
push eax
mov eax, ebx
sub eax, ecx
add edx, eax
pop eax
```

TMP ≈ temporary location for saving and restoring data currently in EAX, so that we can "borrow" EAX for a while



PUSH AND POP INSTRUCTIONS

- **push src** moves the top of the stack up by 4 bytes, then copies the content of the operand (immediate or register) to the address pointed by the updated ESP
- **pop dest** copies the 4 bytes currently pointed by ESP to the dest register, then moves the top of the stack down by 4 bytes

- push/pop operations can also be realized via sub/add + mov:

```
push ebp
xor ecx, ecx
pop edx
```

```
sub esp, 4
mov [esp], ebp
xor ecx, ecx
mov edx, [esp]
add esp, 4
```

USING THE BASE POINTER

- When entering a function that makes use of local variables, register EBP can be used to reference their locations via fixed offsets

```
mov ebp, esp
sub esp, 8
mov [ebp-8], 0x1111
mov [ebp-4], 0xabadcafe
...
mov eax, [ebp-8]
```

Accessing local variables with ESP is like chasing a moving target. However, if code logic uses EBP differently, dereferencing ESP+offset is the only option left…

CALL AND RET INSTRUCTIONS

- A code piece may invoke a function using the **call** instruction
- Unlike jumps, before the control transfer takes place, it stores on the stack the address of the instruction that follows it
  - This address is called the return address for the call
  - The callee can fetch it using **ret** and resume execution in the caller: this instruction will populate EIP with the address read from the top of the stack
- Beware: *functions exists only at a logical level*
  - Code is laid out in memory without any separators
  - Adversaries may scramble their layout, making it hard for analysts to identify function boundaries!

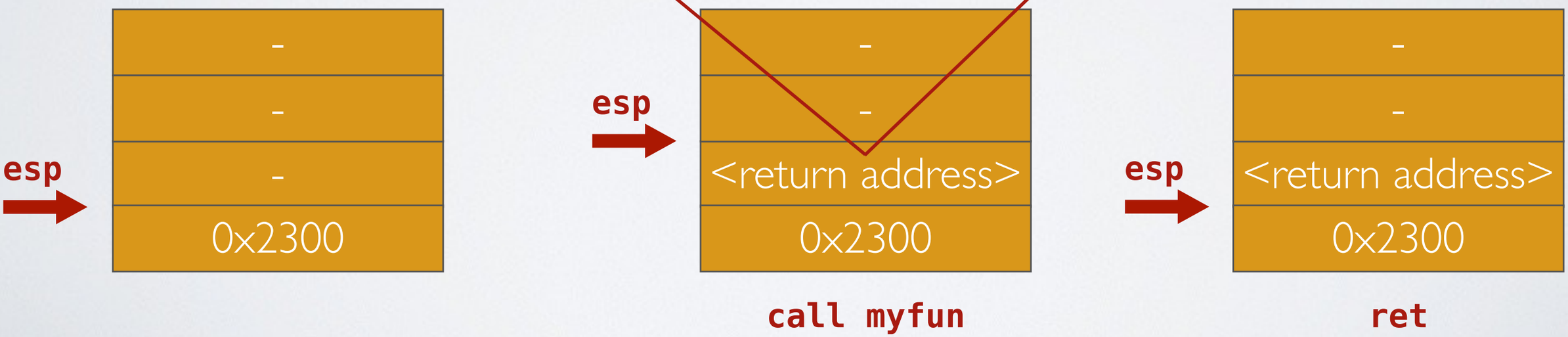
FUNCTION CALL: AN EXAMPLE

```
push eax
add eax, 1
call myfun
sub eax, 1
```

```
myfun:
mov ebp, esp
add esp, 8
…
sub esp, 8
ret
```

```
push eax
add eax, 1
call myfun
sub eax, 1
```

eip



CALLING CONVENTIONS

- (At least) two questions left to answer:
  - How are register values preserved across function invocations?
  - How are parameters being passed to functions?
- **Calling conventions** regulate:
  - how parameters are passed (via registers/stack) and in which order
  - which registers the callee must preserve for the caller
  - who performs ESP realignment upon return
- Malware may use custom calling conventions for its functions, but has to follow standard ones to interact with Windows APIs

WINDOWS CONVENTIONS

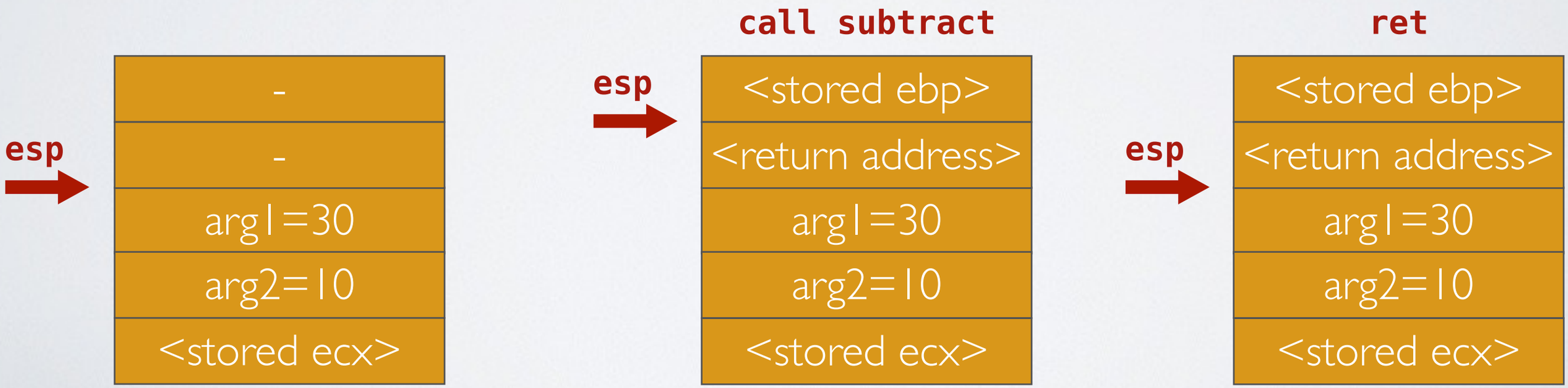
- Different conventions can be used in the same program
- **cdecl** (C programs)
  - arguments pushed on the stack, right-to-left order
  - caller saves EAX, ECX, and EDX when in use (i.e., free for callee, which has to preserve instead EBP, EBX, EDI, and ESI whenever it needs them)
  - caller adjusts ESP after return
  - 32-bit **return value** stored in EAX
- **stdcall** (Win32 API)
  - arguments, caller/callee-save registers and return value as in cdecl
  - callee adjusts the stack pointer upon return
    - **ret N** instruction is used, where N indicates to add N+4 bytes to ESP *after* EIP is updated

EXAMPLE: CDECL

```
mov ecx, 20
...
push ecx
push 10
push 30
call subtract
add esp, 8
pop ecx
cmp eax, ecx
jnz ErrorLabel
```

```
subtract:
push ebp
mov ebp, esp
mov eax, [ebp+8]
mov edx, [ebp+12]
sub eax, edx
pop ebp
ret
```

```
int __cdecl__
subtract(int arg1,
int arg2) {
  return arg1-arg2;
}
```

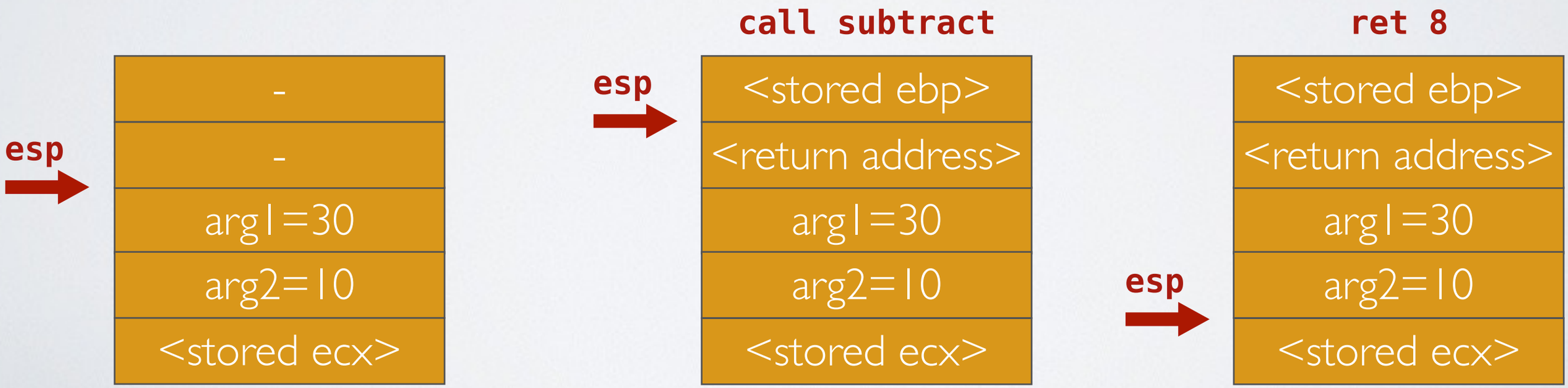


EXAMPLE: STDCALL

```
mov ecx, 20
...
push ecx
push 10
push 30
call subtract
add esp, 8
pop ecx
cmp eax, ecx
jnz ErrorLabel
```

```
subtract:
push ebp
mov ebp, esp
mov eax, [ebp+8]
mov edx, [ebp+12]
sub eax, edx
pop ebp
ret 8
```

```
int __stdcall__
subtract(int arg1,
int arg2) {
  return arg1-arg2;
}
```



CALLER-SAVE VS CALLEE-SAVE

- **Caller-save** registers are pushed to the stack **before a call**, if the caller is going to use their values after the call
  - typically EAX, ECX, EDX *(cdecl, stdcall)*
- **Callee-save** registers are pushed to the stack in the **prologue** of the callee when it needs to use them
  - typically EDI, ESI, EBX *(cdecl, stdcall)*
- Compilers often prefer callee-save registers for performance: when register pressure gets high, also caller-save registers are used. But in custom code we can find any sort of assortment…

EBP - ENTER AND LEAVE

- When used as base pointer a function prologue may set EBP with:

```
push ebp
mov ebp, esp
```

  This can be done with a single instruction: `enter`
  (optional immediate subtracted from ESP to make room for local variables)

- To leave a function, ESP must point to the caller's return address.
  EBP may come to the rescue in a function epilogue:

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

  This can be done with a single instruction: `leave`

LEA INSTRUCTION

- LEA (Load Effective Address) evaluates an expression that defines a memory address and writes it to a destination register
  - data movement and ALU instructions dereference a given address and operate on the content of the pointed location
  - LEA simply computes an address!
- Two uses
  - address calculation (pointer arithmetic)
  - arithmetic expressions

LEA: AN EXAMPLE

- Suppose EBX=0x1000, and 0xABADCAFE is stored at 0x1004
  - `mov eax, [ebx+4]` will copy 0xABADCAFE to register EAX
  - `lea eax, [ebx+4]` will copy 0x1004 to register EAX
- Operands may also not be addresses
  - Suppose EAX=10, EBX=4 and we have to compute C=A+2*B-1
    `lea ecx, [eax+2*ebx-1]` will do the trick with a single instruction!
  - Constraints: width for multiplication has to be 1, 2 or 4

REP INSTRUCTIONS

- Special instructions for manipulating bytes in buffers
  - `cmpsb` to compare bytes from two buffers
  - `stosb` to initialize all bytes in a buffer with same value
  - `movsb` to copy one buffer into another
  - `scasb` to search a byte in a buffer
- They are preceded by a `REP` prefix: it repeats the operation until ECX == 0 or an additional condition on ZF is verified
  - EDI contains the address of the first buffer (destination), ESI contains the address of the second buffer (source), and AL contains the byte to set/search
- We can implement memcmp, memset, memcpy and memchr from the C standard library using just the instructions above

HANDS-ON & BONUS PARTS

GEARING UP YOUR MACHINE

- We will use **IDA Freeware**, which is pre-installed in the VM for this course
- If you have your own VM, get IDA Freeware 8.3 or higher from the Hex Rays website (quick link: https://out7.hex-rays.com/files/idafree83_windows.exe)
- In that case, you may also need the following packages:
  - Get 64-bit MSVCRT from https://aka.ms/vs/17/release/vc_redist.x64.exe
  - (Optional) get 32-bit MSVCRT too https://aka.ms/vs/17/release/vc_redist.x86.exe

CALLS\CDECL.EXE

- Take a look at the C source used for generating this executable
- What are the addresses of the main(), test(), and mysum() functions?
- Function test()
  - What is the purpose of the `sub esp, 18h` instruction?
  - Why the compiler pushes ESI and EBX on the stack after the prologue?
  - What is the role of the `lea eax, [esi+ebx]` instruction?
  - What is the role of the instructions involving `[ebp-0c]`?
- Function mysum()
  - Can you tell why the compiler emitted only `pop ebp` instead of a `leave` (or an equivalent `mov esp, ebp; pop ebp` sequence) as function epilogue?

CALLS\POINTERS.EXE

- Take a look at the C source used for generating this executable
- Function main()
  - How do you take the address of a local variable?
  - How do you take the address of a global variable?
  - What can you tell by the looks of the two printed addresses?
- Function mysum()
  - How is the pointer dereferencing performed?

CALLS\STDCALL.EXE

- Take a look at the C source used for generating this executable
- What are the addresses of the main(), test(), and mysum() functions?
- What can you speculate about the calling conventions that test() and mysum() follow by just looking at their assembly code?
- Function test()
  - What is the purpose of the `sub esp, 28h` instruction?
  - What about the subsequent `sub esp, 0Ch` instruction?
  - What is the lowest value ESP can reach during the execution? (Hint: take a close look at all the components that may alter it…)
  - Can you think of better ways to optimize stack allocation to use fewer bytes?

MORE WINDOWS CONVENTIONS

Two less popular conventions

- **fastcall**
  - first two arguments (left-to-right) in ECX and EDX, others on stack (right-to-left)
  - caller saves EAX, ECX, and EDX when necessary
  - callee adjusts stack pointer on return (as in stdcall)
- **thiscall** (Visual C++ for non-static class member functions)
  - object pointer **this** is passed in ECX, arguments on stack (right-to-left)
  - if the number of arguments is fixed, callee adjusts stack pointer on return (as in stdcall); when is variable, caller adjusts stack pointer (as in cdecl)

REP*.EXE

- We will take a look at instructions that compactly encode common control flow patterns
- Knowing popular usage patterns can speed up your code analysis
- For REP bear in mind that (where this applies) EDI holds a destination address, ESI a source address, and AL the byte of interest. ECX is used as counter.
- At a glance:
  - memcmp() to compare bytes using `rep cmbp`
  - memcpy() to copy buffers using `rep movsb`
  - memchr() to look up occurrences using `rep scasb`
  - memset() to initialize buffers using `rep stosb`

LOOP INSTRUCTION

- Loops are normally encoded using jump instructions. However, a special **`loop`** instruction is available (compilers avoid it for efficiency)
- ECX is used as accumulator: loop first decrements it, then if the value is not zero it jumps to the specified label

```
mov ecx, 10
L: add eax, 1
loop L
```

- For complex conditions, **`loop`** can be combined with a ZF check:
  - **`loope`** jumps when ECX != 0 && ZF = 1
  - **`loopne`** jumps when ECX != 0 && ZF = 0

ANTI-DISASSEMBLY

- antidisassembly.exe hinders the disassembly process two steps
  - it computes a displacement for the instruction pointer by first leaking its value using the `call $+5` pattern (we will meet it again in a few weeks…)
  - it jumps to the computed address, whereas the disassembler may erroneously believe it to be part of an instruction starting a few bytes earlier *(put in other words, the disassembler chose a "wrong" initial position to look for valid instructions)*
- Using a debugger clearly helps, but we can handle this one statically:
  - `call $+5` pushes on the stack the address of the subsequent `pop eax` instruction
  - this address is written to register EAX and is equal to 401468 (hexadecimal)
  - the target of the jump is 401468 + 0A = 401472. Execution will continue there!
  - older versions of the IDA disassembler interpret it as part of the instruction starting at 401470 *(=> undefine such instruction and instruct IDA that some code starts instead at 401472; once the code is displayed, you can also create a function there)*

MORE ANTI-DISASSEMBLY

- smc.exe takes antidisassembly.exe to the next level
  - it leaks the instruction pointer value using the `call $+5` pattern and uses it to compute two addresses that get stored in EAX and EDX
  - the two take part in a loop that swaps memory contents *(bear in mind the code called VirtualProtect to allow code edits)* for endianness affecting 4 bytes at a time
  - when it reaches the `nop` instruction, something odd is about to happen!
- By using a debugger (or 📄&✏️ if you prefer 🤯) we can tell that:
  - the `pop ebp; ret` sequence is gone
  - the two subsequent bytes *(hint: undefine the dword 1400EB02h at address 401488)* `02 eb` after the swap became a `jmp $+2` placed next to the `nop`
  - this will jump past the swapped `ret ; pop ebp` sequence and do the real gig!
  - the code that gets executed is the same that we extracted for antidisassembly.exe