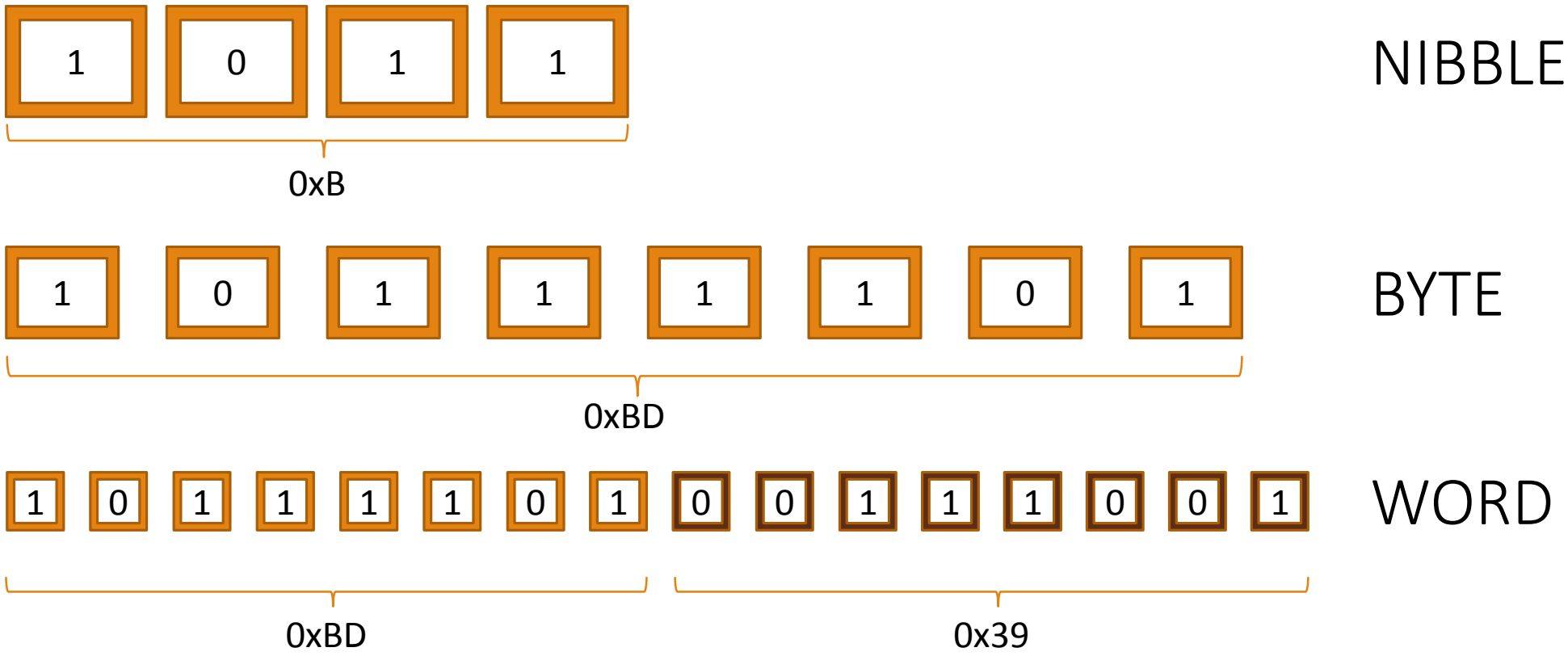
Static Analysis I

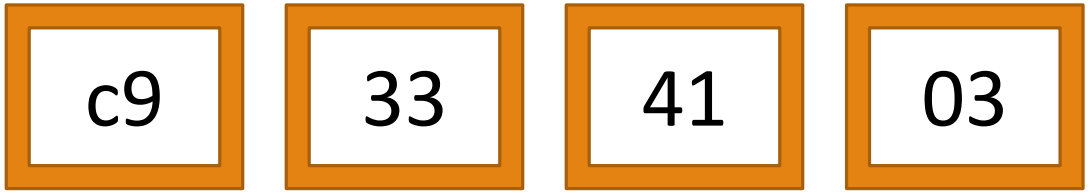
PAOLO PALUMBO, F-SECURE CORPORATION

Representing Data

Binary numbers



Endianness



Actual bytes

0x034133c9

Little Endian

0xc9334103

Big Endian

Strings

```
H  e  l  l  o     1  2  3  4
48 65 6C 6C 6F 20 31 32 33 34
```

ASCII

```
(BOM)  H     e     l     l     o
(ff fe) 48 00 65 00 6C 00 6C 00 6F 00
```

Unicode (UCS-2)

Where do string end?

ASCIIZ	H e l l o 48 65 6C 6C 6F 00
Null-terminated Unicode	H e l l o 48 00 65 00 6C 00 6C 00 6F 00 00 00
Pascal	H e l l o 05 48 65 6C 6C 6F
Delphi	H e l l o 05 00 00 00 48 65 6C 6C 6F

Representing Code

CPU Architectures

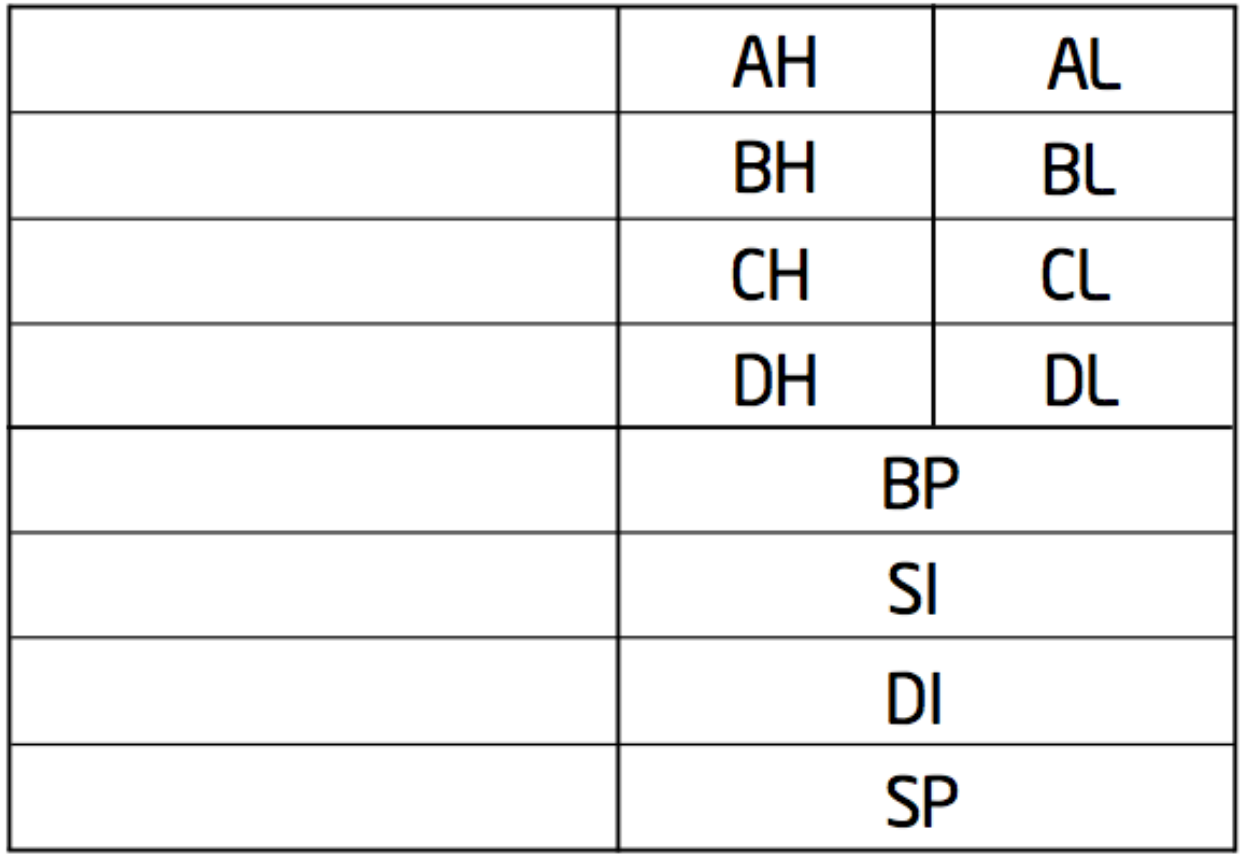
- CISC – Complex Instruction Set Computing
  - Emphasis on hardware and assembly programming
  - Example: X86
- RISC – Reduced Instruction Set Computing
  - Emphasis on software and high-level languages
  - Example: ARM

Introduction to x86

The i386, in short

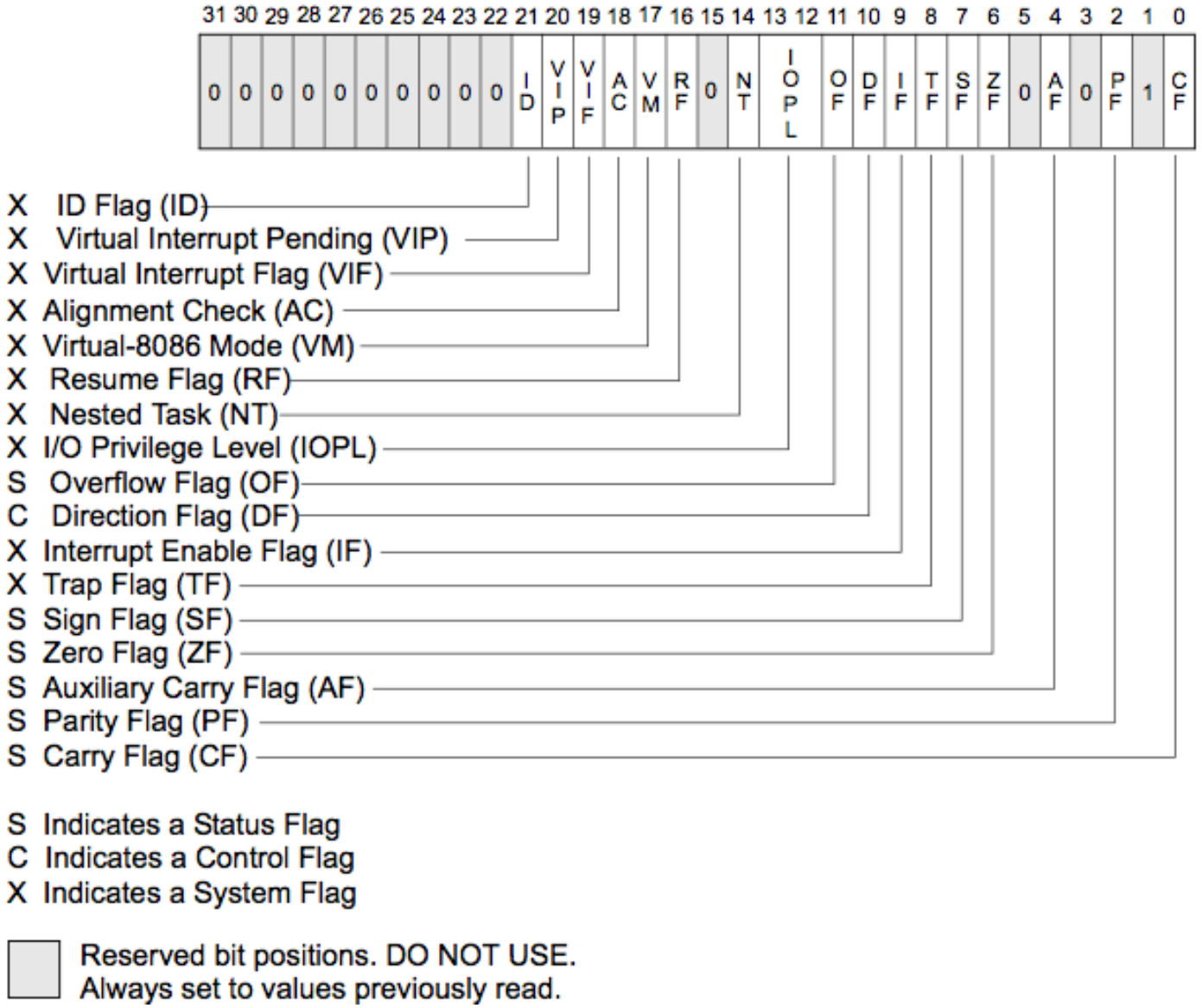
- Introduced in 1986
- Has a 32-bit word length
- Has 8 general-purpose registers
- Supports paging and virtual memory
- Addresses up to 4GiB of memory

General-Purpose Registers



16-bit	32-bit
AX	EAX
BX	EBX
CX	ECX
DX	EDX
	EBP
	ESI
	EDI
	ESP

EFLAGS register



Read on your own: common x86 instructions

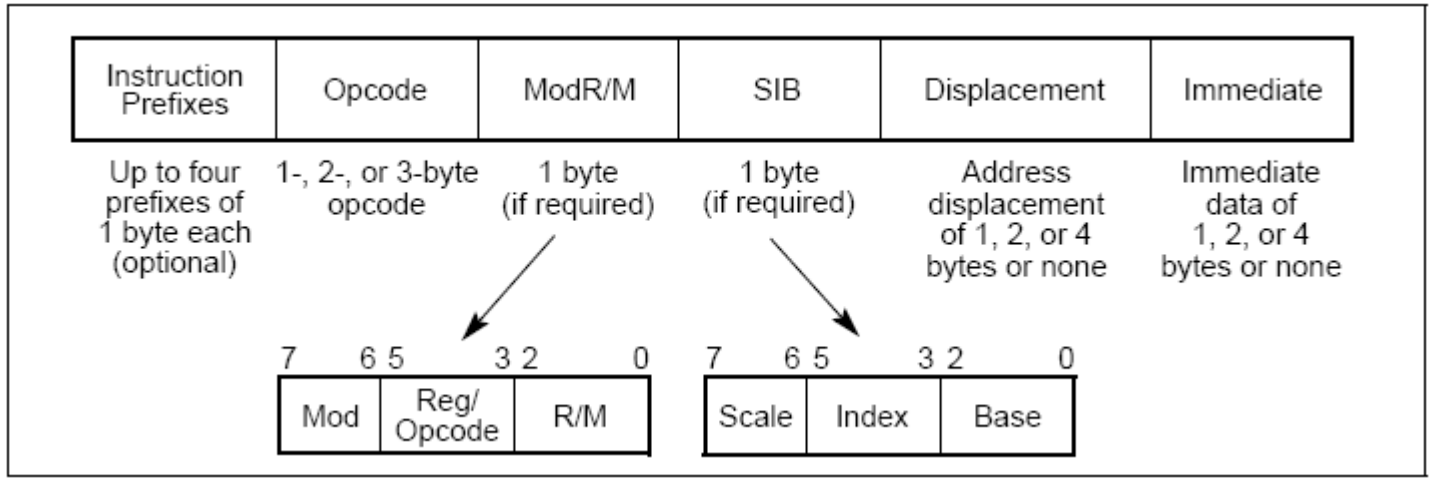
`MOV EAX, 1`	Move 1 to EAX
`ADD EDX, 5`	Add 5 to EDX
`SUB EBX, 2`	Subtract 2 from EBX
`AND ECX, 0`	Bit-wise AND 0 to ECX
`XOR EDX, 4`	Bit-wise eXclusive OR 4 to EDX
`SHL ECX, 6`	Shift ECX left by six
`ROR EBX, 3`	Bit-wise rotate EBX right by 3
`INC ECX`	Increment ECX

Read on your own: common x86 instructions (continued)

`JNZ label`	Jump if not zero (equal)
`JMP label`	Unconditional jump to label
`CALL func`	Call function
`RET`	Return from function
`LOOP label`	ECX--, Jump to label if not zero
`PUSH EAX`	Push EAX to stack
`POP EDI`	Pop EDI from stack

Intel x86 Instruction Format (32-bit)

- Variable-size complex instruction format (CISC)
- Instruction format: OPERATION, [OPERAND 1, [OPERAND 2, [OPERAND 3]]]
  - Operands can operate on register, memory address or immediate data



- A single instruction can be of any size between 1 and 15 bytes

Intel x86-64 Instruction Format (64-bit)

Legacy Prefixes	**REX Prefix**	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address dis-placement of 1, 2, or 4 bytes or none	Immediate data of 1, 2, or 4 bytes or none

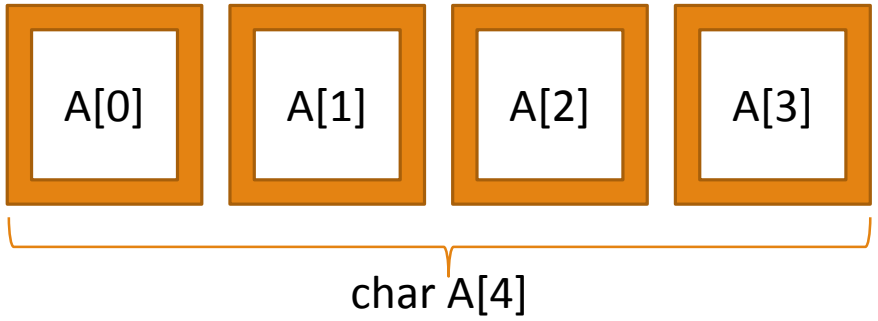
Reversing C

Basic data types

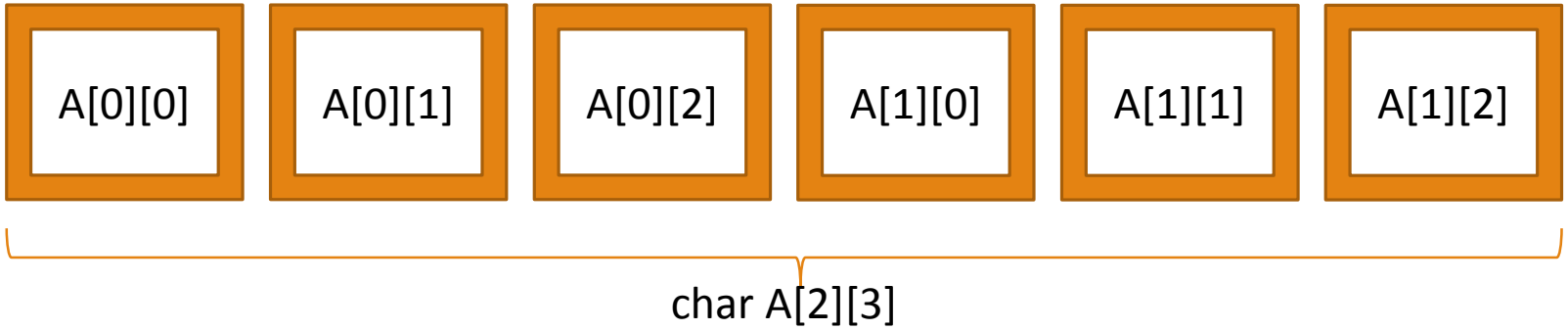
- char – 1 byte
- short – 2 bytes
- int – 4 bytes (platform word)
- long – 4 bytes
- float – 4 bytes floating point
- double – 8 bytes floating point

Arrays

- One dimensional arrays



- Multi dimensional arrays



```
char val[2][3][2] = {{{'0','1'},{'2','3'},{'4','5'}}, {{'6','7'},{'8','9'},{'a','b'}}};
```

```
mov     [ebp+val], '0'
mov     [ebp+val+1], '1'
mov     [ebp+val+2], '2'
mov     [ebp+val+3], '3'
mov     [ebp+val+4], '4'
mov     [ebp+val+5], '5'
mov     [ebp+val+6], '6'
mov     [ebp+val+7], '7'
mov     [ebp+val+8], '8'
mov     [ebp+val+9], '9'
mov     [ebp+val+0Ah], 'a'
mov     [ebp+val+0Bh], 'b'
```

Structures and Unions

```
struct {
    unsigned int   id;
    unsigned short age;
    char name[16];
} record;
```

Memory allocated for all members combined:

sizeof(record) = 24

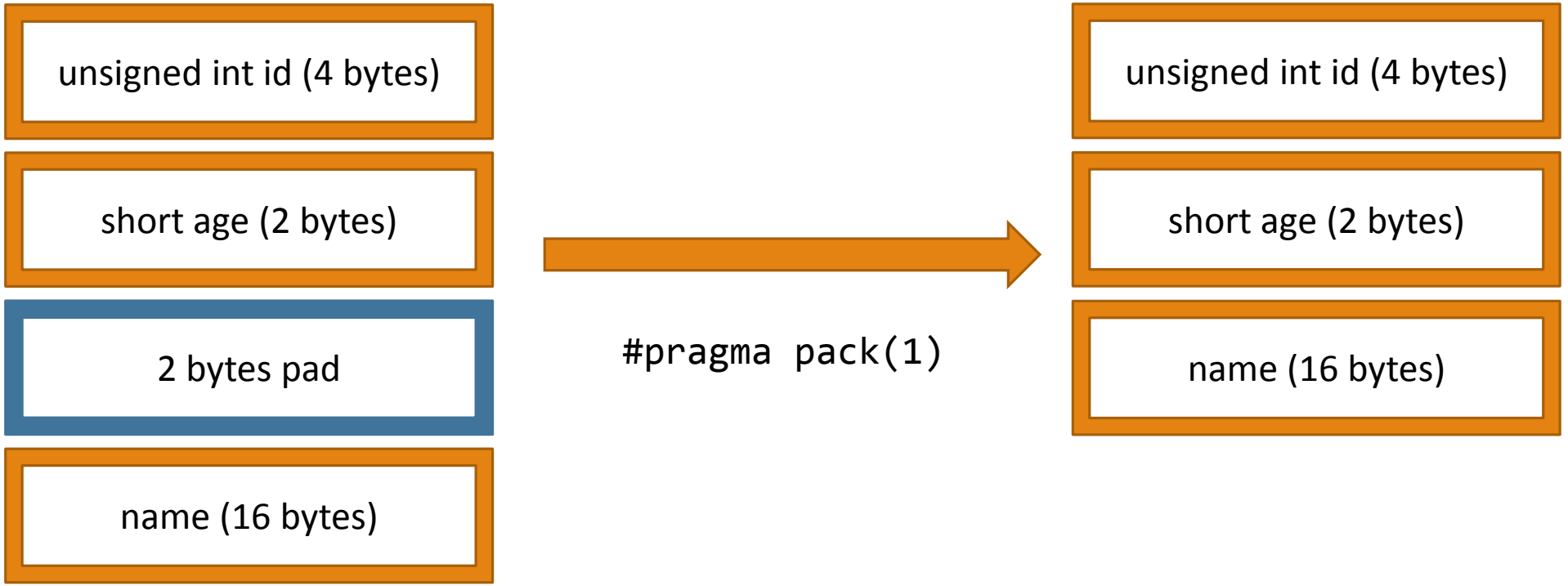
```
union foo {
  int  one;
  char two;
  };
```

Memory allocated for largest member only:

sizeof(foo) = 4

Structure alignment

- Data structure are aligned to machine word size
  - The `#pragma pack()` directive controls the alignment



Calling conventions: stdcall

```
int __stdcall foobar(int x, int y, char *foo, char *bar);
```

```
mov     eax, [ebp+bar]
push    eax                 ; bar
mov     ecx, [ebp+foo]
push    ecx                 ; foo
mov     edx, [ebp+y]
push    edx                 ; y
mov     eax, [ebp+x]
push    eax                 ; x
call    foobar
mov     [ebp+res], eax
```

```
foobar
```

```
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn    10h
endp
```

Calling conventions: cdecl

```
int __cdecl foobar(int x, int y, char *foo, char *bar);
```

```
mov     eax, [ebp+bar]
push    eax                     ; bar
mov     ecx, [ebp+foo]
push    ecx                     ; foo
mov     edx, [ebp+y]
push    edx                     ; y
mov     eax, [ebp+x]
push    eax                     ; x
call    foobar
add     esp, 10h
mov     [ebp+res], eax
```

```
                pop     edi
                pop     esi
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
foobar          endp
```

Calling conventions: fastcall

```
int __fastcall foobar(int x, int y, char *foo, char *bar);
```

```
mov     eax, [ebp+bar]
push    eax                     ; bar
mov     ecx, [ebp+foo]
push    ecx                     ; foo
mov     edx, [ebp+y]            ; y
mov     ecx, [ebp+x]            ; x
call    foobar
mov     [ebp+res], eax
```

```
        pop     edi
        pop     esi
        pop     ebx
        mov     esp, ebp
        pop     ebp
        retn    8
foobar  endp
```

Calling conventions: thiscall

```
class Foo
{
public:
    int Bar(int x, int y, char *foo, char *bar);

};
```

```
mov     eax, [ebp+bar]
push    eax             ; bar
mov     ecx, [ebp+foo]
push    ecx             ; foo
mov     edx, [ebp+y]
push    edx             ; y
mov     eax, [ebp+x]
push    eax             ; x
lea     ecx, [ebp+f]    ; this
call    Foo__Bar
```

```
Foo__Bar        pop     edi
                pop     esi
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn    10h
                endp
```

Parameter Passing

Visual Studio 2010 | Other Versions ▾ | 3 out of 4 rated this helpful - Rate this topic

The first four integer arguments will be passed in registers. Integer values will be passed (in order left to right) in RCX, RDX, R8, and R9. Arguments five and higher will passed onto the stack. All arguments are right justified in registers. This is done so the callee can ignore the upper bits of the register if need be and can access only the portion of the register necessary.

Floating-point and double-precision arguments are passed in XMM0 – XMM3 (up to 4) with the integer slot (RCX, RDX, R8, and R9) that would normally be used for that cardinal slot being ignored (see example) and vice versa.

__m128 types, arrays and strings are never passed by immediate value but rather a pointer will be passed to memory allocated by the caller. Structs/unions of size 8, 16, 32, or 64 bits and __m64 will be passed as if they were integers of the same size. Structs/unions other than these sizes will be passed as a pointer to memory allocated by the caller. For these aggregate types passed as a pointer (including __m128), the caller-allocated temporary memory will be 16-byte aligned.

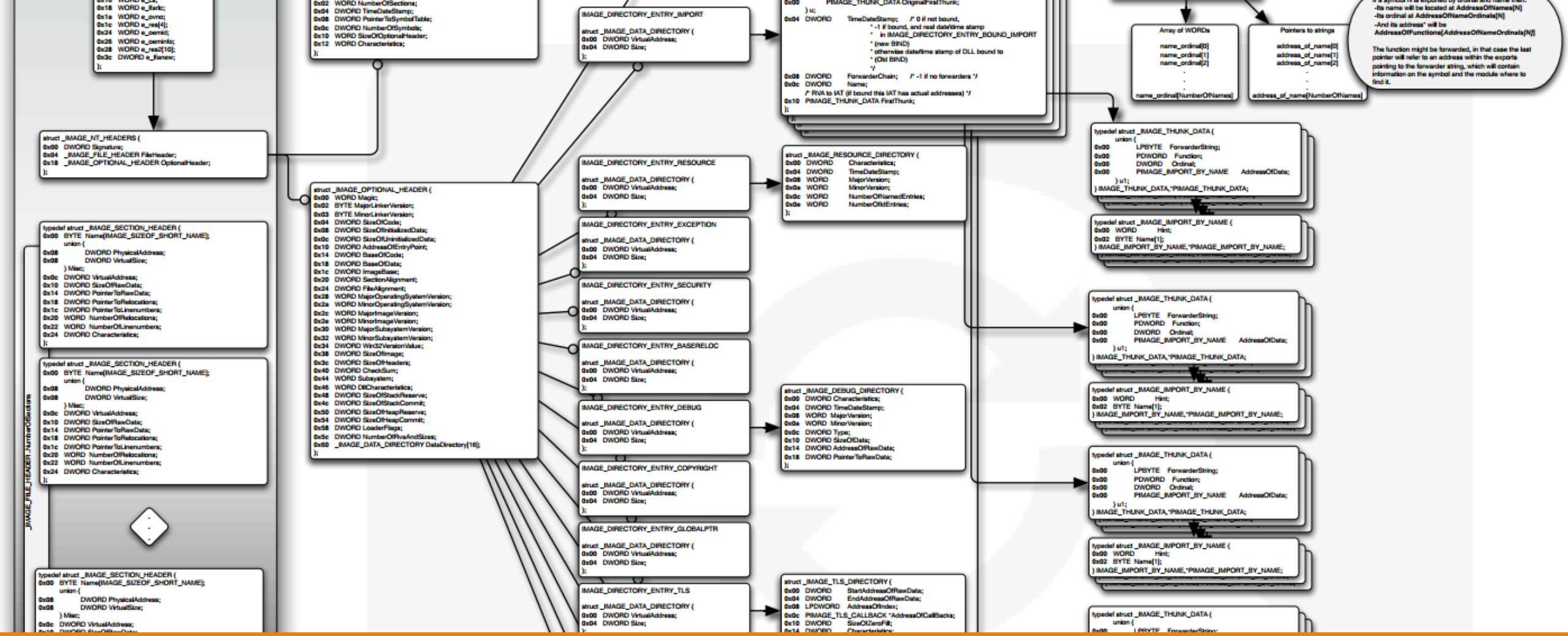
Intrinsic functions that do not allocate stack space and do not call other functions can use other volatile registers to pass additional register arguments because there is a tight binding between the compiler and the intrinsic function implementation. This is a further opportunity for improving performance.

The callee has the responsibility of dumping the register parameters into their shadow space if needed.

x64 __fastcall calling convention

x64 __fastcall parameter passing summary

Parameter type	How passed
Floating point	First 4 parameters – XMM0 through XMM3. Others passed on stack.
Integer	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (8, 16, 32, or 64 bits) and __m64	First 4 parameters – RCX, RDX, R8, R9. Others passed on stack.
Aggregates (other)	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9
__m128	By pointer. First 4 parameters passed as pointers in RCX, RDX, R8, and R9



PE/COFF

Introduction to PE/COFF

PE stands for Portable Executable

Microsoft introduced PE in Windows NT 3.1

It originates from Unix COFF

Features dynamic linking, symbol exporting/importing

Can contain Intel, Alpha, MIPS and even .NET MSIL binary code

64-bit version is called PE32+

Microsoft PE and COFF Specification

Updated: February 6, 2013

This specification describes the structure of executable (image) files and object files under the Windows family of operating systems. These files are referred to as Portable Executable (PE) and Common Object File Format (COFF) files, respectively.

Note: This document is provided to aid in the development of tools and applications for Windows but is not guaranteed to be a complete specification in all respects. Microsoft reserves the right to alter this document without notice.

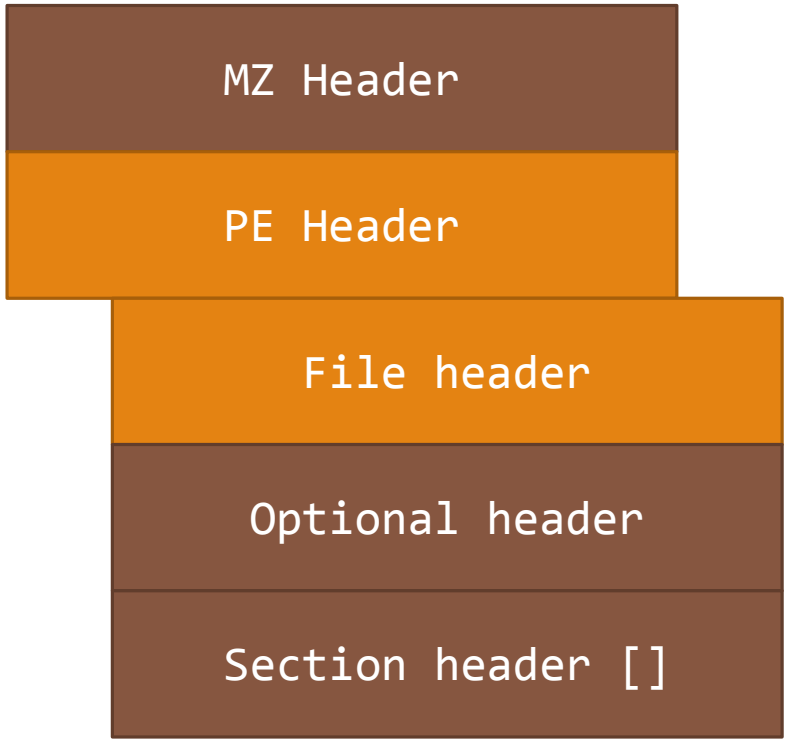
This revision of the Microsoft Portable Executable and Common Object File Format Specification replaces Revision 6.0 of this specification.

Get the PE file format specification here!

MZ header

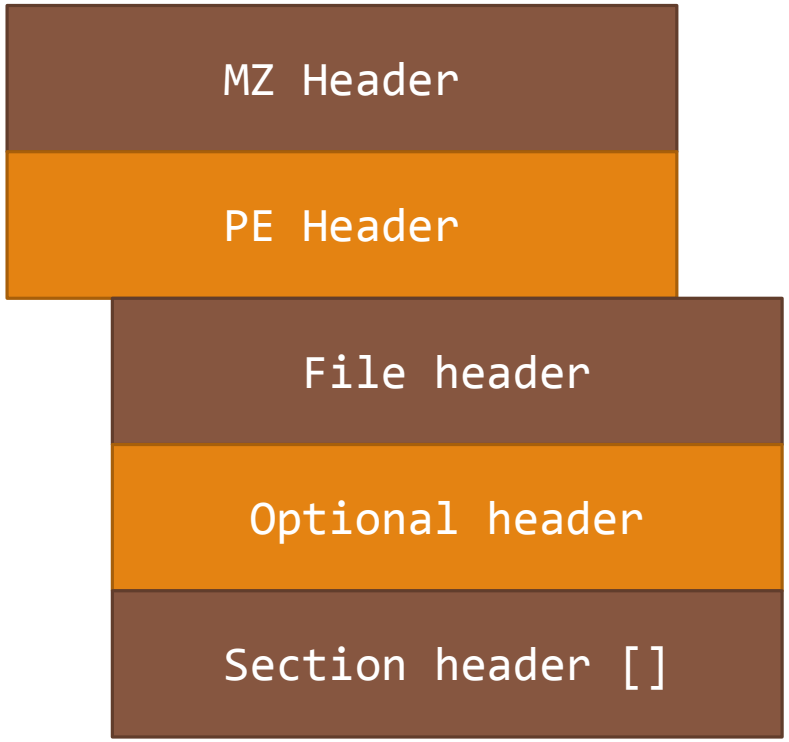

```
struct _IMAGE_DOS_HEADER {
0x00   WORD e_magic;
0x02   WORD e_cblp;
0x04   WORD e_cp;
0x06   WORD e_crlc;
0x08   WORD e_cparhdr;
0x0a   WORD e_minalloc;
0x0c   WORD e_maxalloc;
0x0e   WORD e_ss;
0x10   WORD e_sp;
0x12   WORD e_csum;
0x14   WORD e_ip;
0x16   WORD e_cs;
0x18   WORD e_lfarlc;
0x1a   WORD e_ovno;
0x1c   WORD e_res[4];
0x24   WORD e_oemid;
0x26   WORD e_oeminfo;
0x28   WORD e_res2[10];
0x3c   DWORD e_lfanew;
};
```

PE header



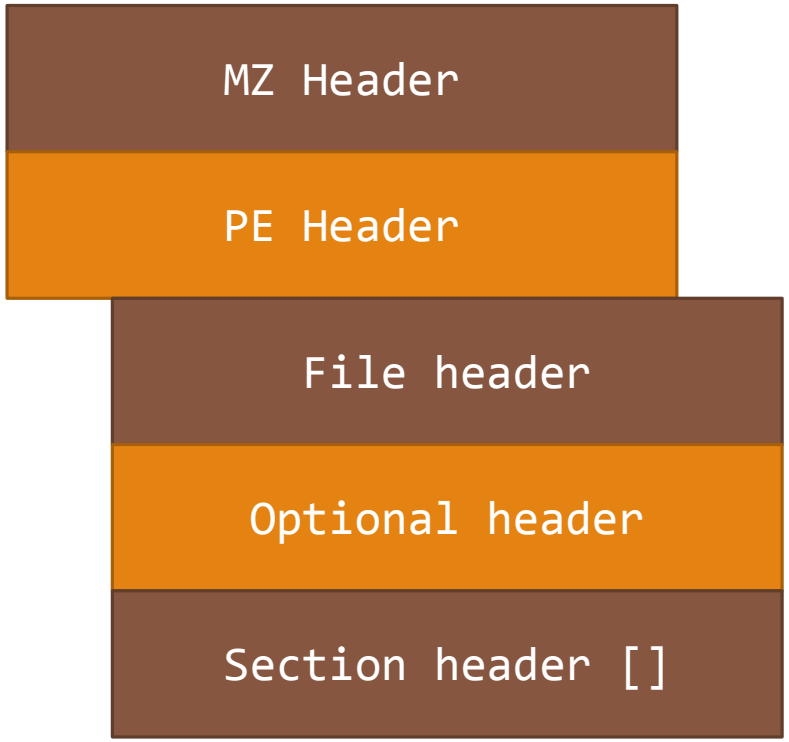

```
struct _IMAGE_FILE_HEADER {
0x00   WORD Machine;
0x02   WORD NumberOfSections;
0x04   DWORD TimeDateStamp;
0x08   DWORD PointerToSymbolTable;
0x0c   DWORD NumberOfSymbols;
0x10   WORD SizeOfOptionalHeader;
0x12   WORD Characteristics;
};
```

PE header, continued




```
struct _IMAGE_OPTIONAL_HEADER {
0x00   WORD Magic;
0x02   BYTE MajorLinkerVersion;
0x03   BYTE MinorLinkerVersion;
0x04   DWORD SizeOfCode;
0x08   DWORD SizeOfInitializedData;
0x0c   DWORD SizeOfUninitializedData;
0x10   DWORD AddressOfEntryPoint;
0x14   DWORD BaseOfCode;
0x18   DWORD BaseOfData;
0x1c   DWORD ImageBase;
0x20   DWORD SectionAlignment;
0x24   DWORD FileAlignment;
0x28   WORD MajorOperatingSystemVersion;
0x2a   WORD MinorOperatingSystemVersion;
0x2c   WORD MajorImageVersion;
0x2e   WORD MinorImageVersion;
0x30   WORD MajorSubsystemVersion;
0x32   WORD MinorSubsystemVersion;
0x34   DWORD Win32VersionValue;
0x38   DWORD SizeOfImage;
0x3c   DWORD SizeOfHeaders;
0x40   DWORD CheckSum;
0x44   WORD Subsystem;
0x46   WORD DllCharacteristics;
0x48   DWORD SizeOfStackReserve;
0x4c   DWORD SizeOfStackCommit;
0x50   DWORD SizeOfHeapReserve;
0x54   DWORD SizeOfHeapCommit;
0x58   DWORD LoaderFlags;
0x5c   DWORD NumberOfRvaAndSizes;
0x60   _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

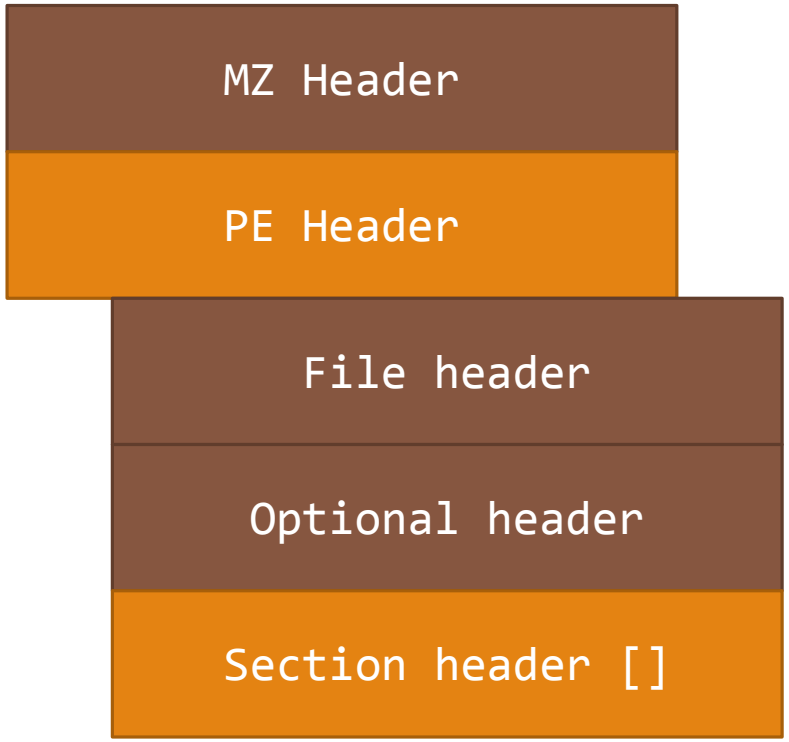
PE header, continued



IMAGE_DIRECTORY_ENTRY_EXPORT
IMAGE_DIRECTORY_ENTRY_IMPORT
IMAGE_DIRECTORY_ENTRY_RESOURCE
IMAGE_DIRECTORY_ENTRY_EXCEPTION
IMAGE_DIRECTORY_ENTRY_SECURITY
IMAGE_DIRECTORY_ENTRY_BASERELOC
IMAGE_DIRECTORY_ENTRY_DEBUG
IMAGE_DIRECTORY_ENTRY_COPYRIGHT
IMAGE_DIRECTORY_ENTRY_GLOBALPTR
IMAGE_DIRECTORY_ENTRY_TLS
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
IMAGE_DIRECTORY_ENTRY_IAT
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR

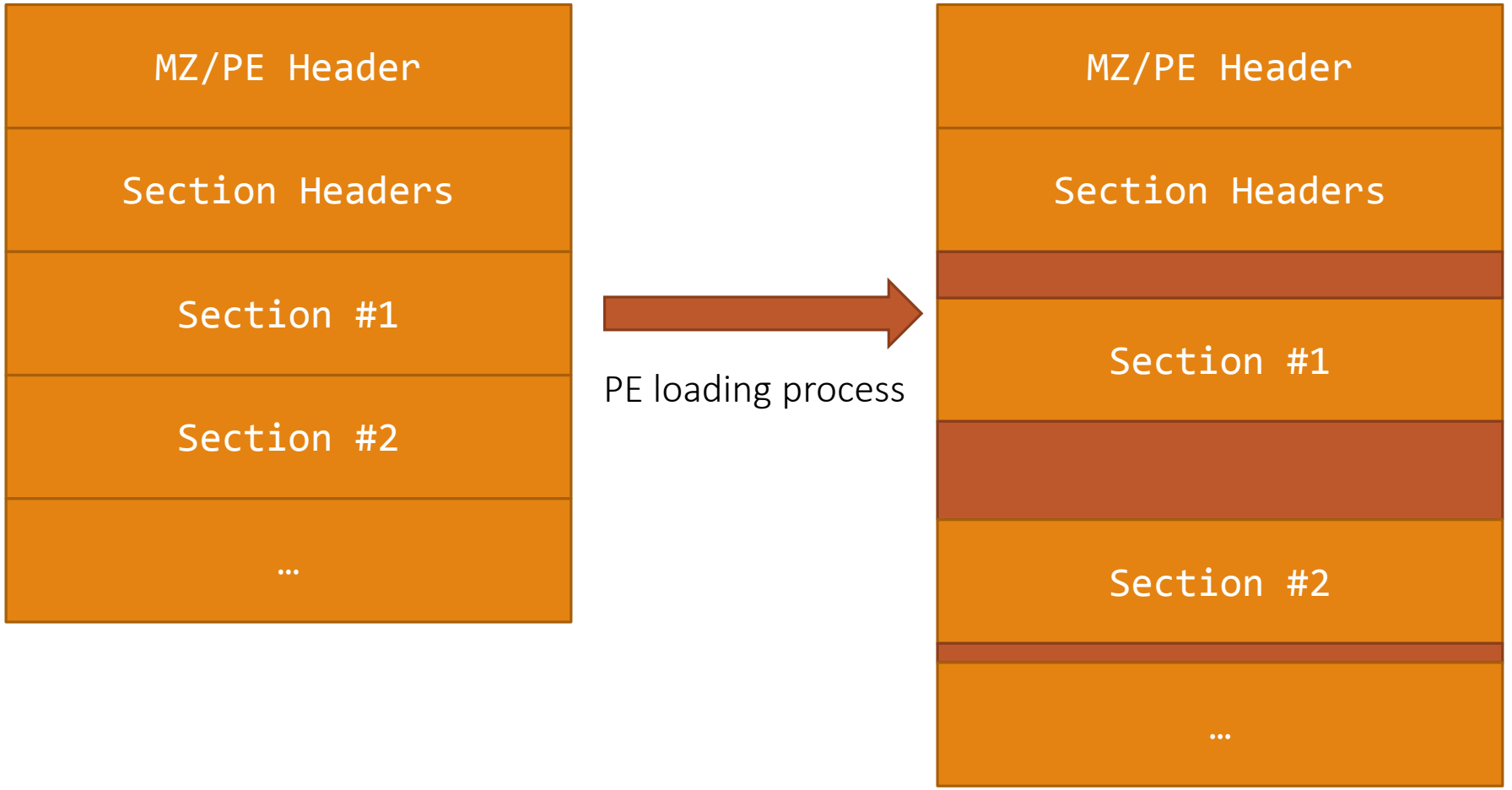
```
struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

PE header, continued




```
typedef struct _IMAGE_SECTION_HEADER {
0x00    BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
0x08            DWORD PhysicalAddress;
0x08            DWORD VirtualSize;
        } Misc;
0x0c    DWORD VirtualAddress;
0x10    DWORD SizeOfRawData;
0x14    DWORD PointerToRawData;
0x18    DWORD PointerToRelocations;
0x1c    DWORD PointerToLinenumbers;
0x20    WORD  NumberOfRelocations;
0x22    WORD  NumberOfLinenumbers;
0x24    DWORD Characteristics;
};
```

PE Loading in a nutshell



Importing external symbols

Symbols (functions/data) can be imported from external DLLs

The loader will load external DLLs automatically

All the dependencies are loaded as well

DLLs will be loaded only once

External addresses are written to the Import Address Table (IAT)

Exporting symbols

Symbols can be exported with ordinals, names or both

Ordinals are simple index numbers of symbols

Name is a full ASCII name of the exported symbol

Exports can be forwarded to another DLL

- Forwarded symbol's address points to a name in the exports section

Resources

Resources in PE are similar to an archive

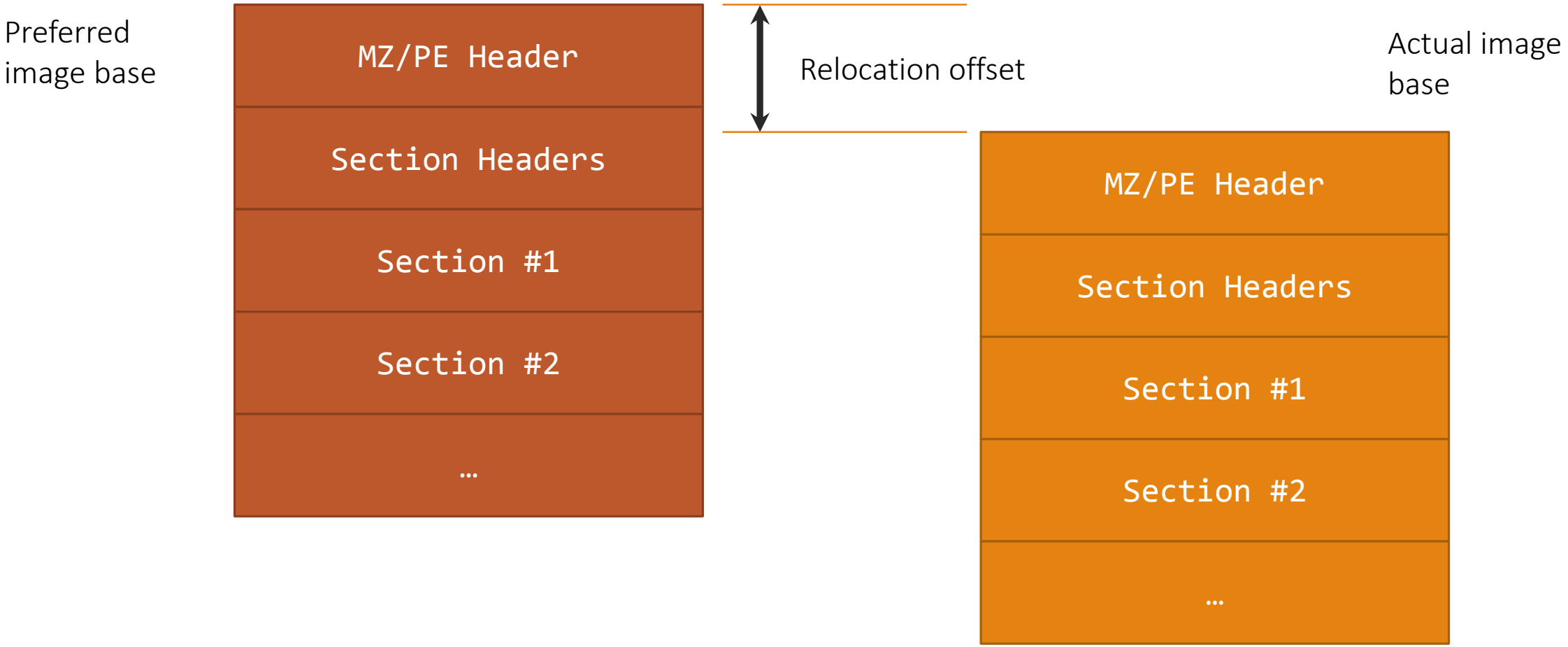
Resource files can be organised into directory trees

The data structure is quite complex but there are tools to handle it

Most common resources:

- Icons
- Version information
- GUI resources

Base Relocation



Resources

Intel Developer Manuals:

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Microsoft PE format specification:

- http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx