# Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015

xeno@legbacore.com

## Architecture - CISC vs. RISC

- Intel is CISC - Complex Instruction Set Computer
  - Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  - Variable-length instructions, between 1 and 15 bytes long.
- Other major architectures are typically RISC - Reduced Instruction Set Computer
  - Typically more registers, less and fixed-size instructions
  - Examples: PowerPC, ARM, SPARC, MIPS

# Take a look, it’s in a book!

- Thanks to Dillon Beresford for sending concrete examples of the longest possible instructions, back when my slides said I was unsure on the max length:

“Longest x86 instruction is 15 bytes in 16-bit mode and 13 bytes in 32-bit mode:

[16-bit]

66 67 F0 3E 81 04 4E 01234567 89ABCDEF

add [ds:esi+ecx*2+0x67452301], 0xEFCDAB89

[32-bit]

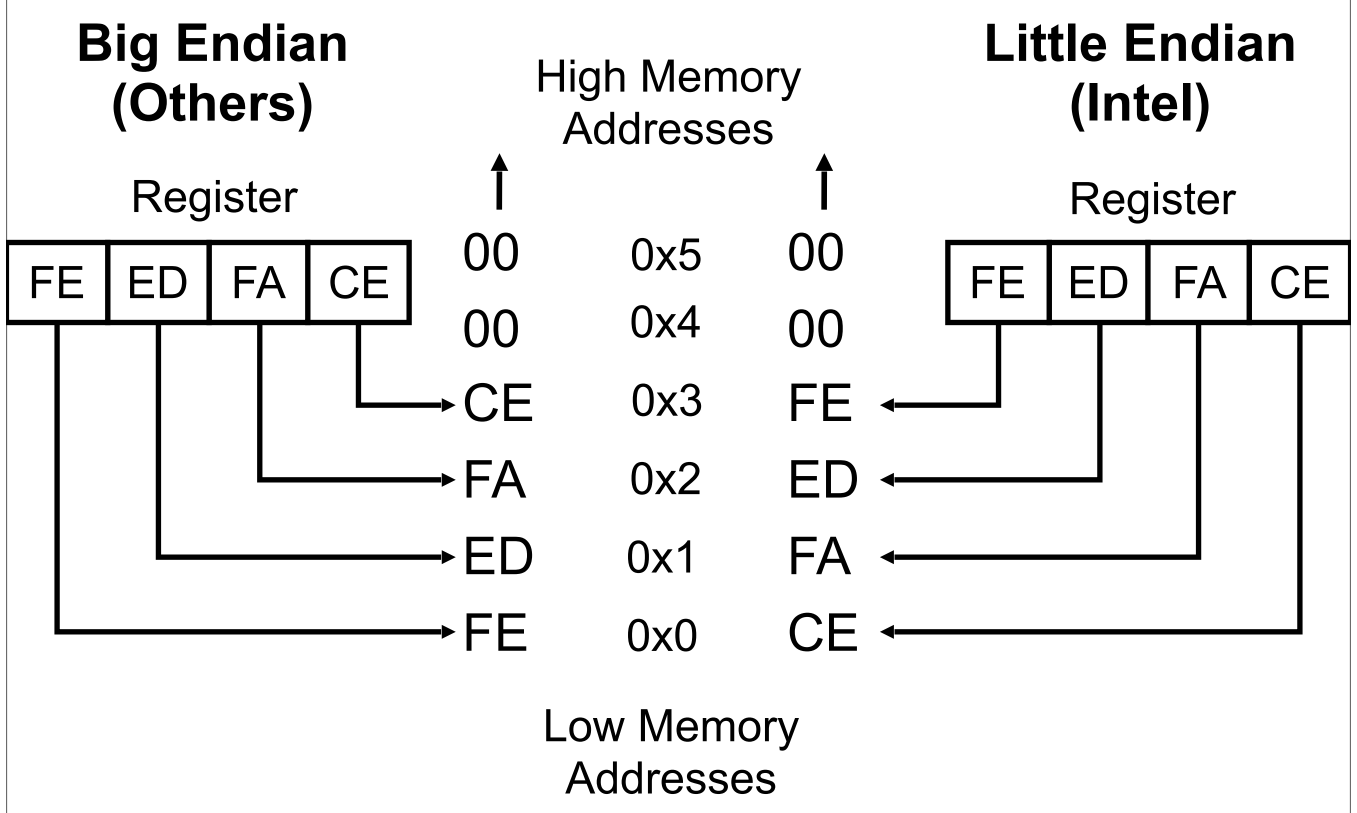
F0 3E 81 04 4E 01234567 89ABCDEF”

- If we get to the RTFM material by the end, you’ll be able to sort of read those ;)

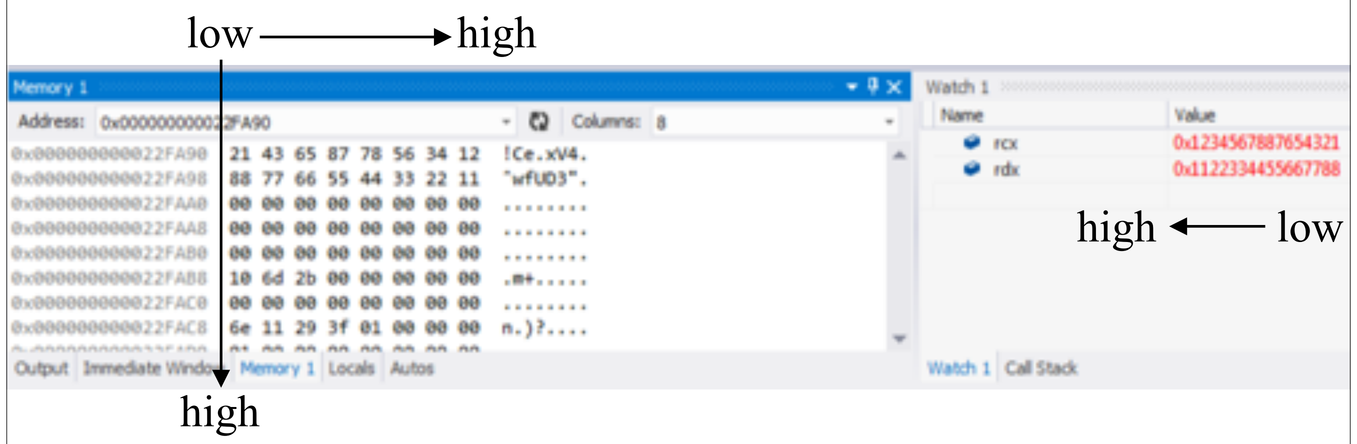
## Architecture - Endian

- Endianness comes from Jonathan Swift's *Gulliver's Travels*. It doesn't matter which way you eat your eggs
- Little Endian - 0x12345678 stored in RAM "little end" first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Many larger RISC systems (PowerPC, SPARC, MIPS, Motorola 68k) started as Big Endian but can now be configured as either (Bi-Endian). ARM started out Little Endian and now is Bi-Endian

# Endianess pictures



# How you'll probably usually see endianness expressed:



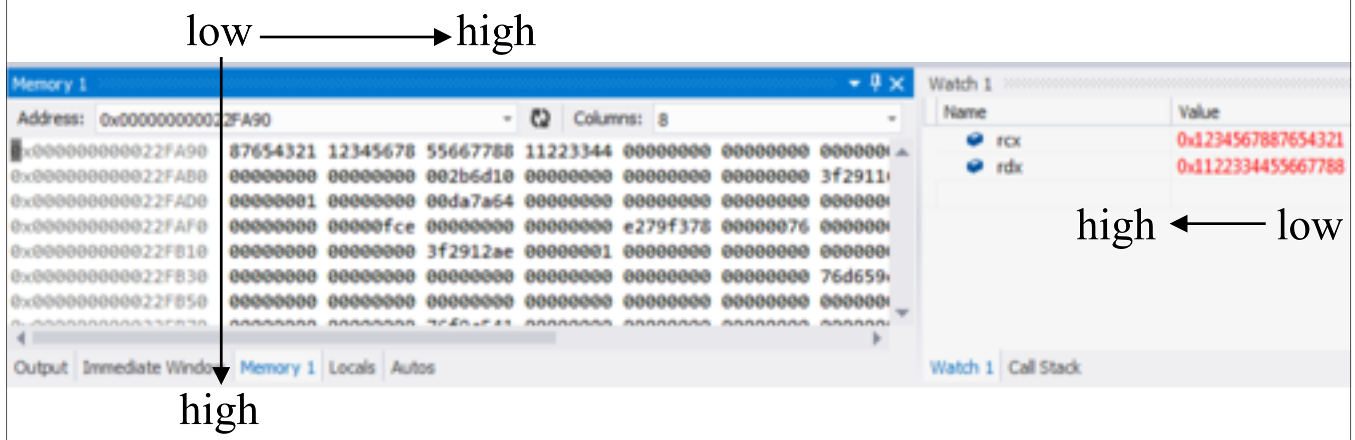
```
0x000000000022FA90  21 43 65 87 78 56 34 12  !Ce.xV4.
0x000000000022FA98  88 77 66 55 44 33 22 11  "wfUD3".
0x000000000022FAA0  00 00 00 00 00 00 00 00  ........
0x000000000022FAA8  00 00 00 00 00 00 00 00  ........
0x000000000022FAB0  00 00 00 00 00 00 00 00  ........
0x000000000022FAB8  10 6d 2b 00 00 00 00 00  .m+.....
0x000000000022FAC0  00 00 00 00 00 00 00 00  ........
0x000000000022FAC8  6e 11 29 3f 01 00 00 00  n.)?....
```

| Name | Value |
|---|---|
| rcx | 0x1234567887654321 |
| rdx | 0x1122334455667788 |

Memory dump windows are typically shown in typical English writing left-to-right, top-to-bottom form, with the upper left being the lowest address

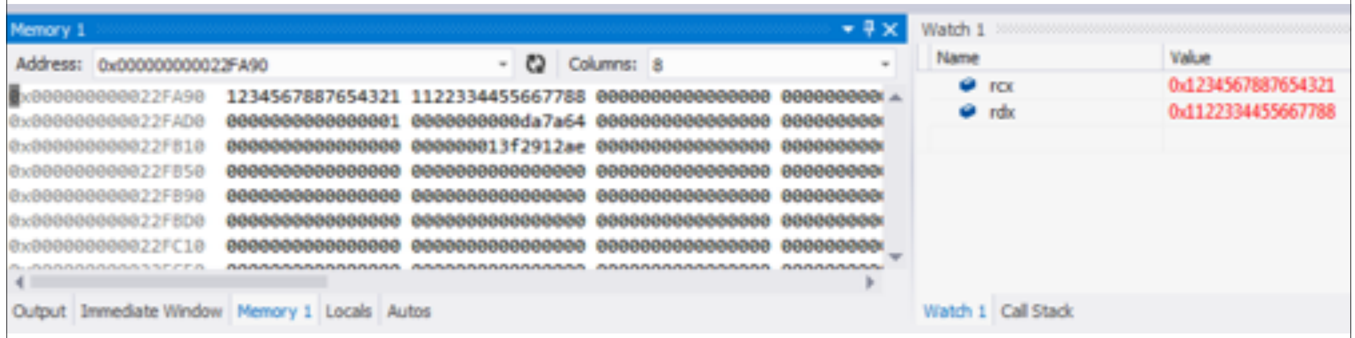
Register view windows always show the registers in big endian order

# But if you change the display size…



```
Memory 1
Address: 0x000000000022FA90          Columns: 8
0x000000000022FA90  87654321 12345678 55667788 11223344 00000000 00000000 000000
0x000000000022FAB0  00000000 00000000 002b6d10 00000000 00000000 00000000 3f2911
0x000000000022FAD0  00000001 00000000 00da7a64 00000000 00000000 00000000 000000
0x000000000022FAF0  00000000 00000fce 00000000 00000000 e279f378 00000076 000000
0x000000000022FB10  00000000 00000000 3f2912ae 00000001 00000000 00000000 000000
0x000000000022FB30  00000000 00000000 00000000 00000000 00000000 00000000 76d659
0x000000000022FB50  00000000 00000000 00000000 00000000 00000000 00000000 000000
```

```
Watch 1
Name    Value
rcx     0x1234567887654321
rdx     0x1122334455667788
```



```
Memory 1
Address: 0x000000000022FA90          Columns: 8
0x000000000022FA90  1234567887654321 1122334455667788 0000000000000000 000000000
0x000000000022FAD0  0000000000000001 0000000000da7a64 0000000000000000 000000000
0x000000000022FB10  0000000000000000 000000013f2912ae 0000000000000000 000000000
0x000000000022FB50  0000000000000000 0000000000000000 0000000000000000 000000000
0x000000000022FB90  0000000000000000 0000000000000000 0000000000000000 000000000
0x000000000022FBD0  0000000000000000 0000000000000000 0000000000000000 000000000
0x000000000022FC10  0000000000000000 0000000000000000 0000000000000000 000000000
```

```
Watch 1
Name    Value
rcx     0x1234567887654321
rdx     0x1122334455667788
```

If you start asking the debugger to display things, 2, 4, or 8 bytes at a time, it will typically take those chunks and display them each big endian order

## Architecture - Registers

- Registers are small memory storage areas built into the processor (still volatile memory)
- 16 “general purpose” registers + the instruction pointer which points at the next instruction to execute
  - But two of the 16 are not that general
- On x86-32, aka IA32 registers are 32 bits long
- On x86-64, aka IA32e they’re 64 bits

*Book Chapter 5*

## Architecture - Register Conventions 1

These are Intel's suggestions to compiler developers (and assembly handcoders). Registers don't have to be used these ways, but if you see them being used like this, you'll know why. But I simplified some descriptions. I also color coded as **GREEN** for the ones which we will actually see in *this* class (as opposed to future ones), and **RED** for not.

- **RAX** – Stores function return values
- **RBX** – Base pointer to the data section
- **RCX** – Counter for string and loop operations
- **RDX** – I/O pointer

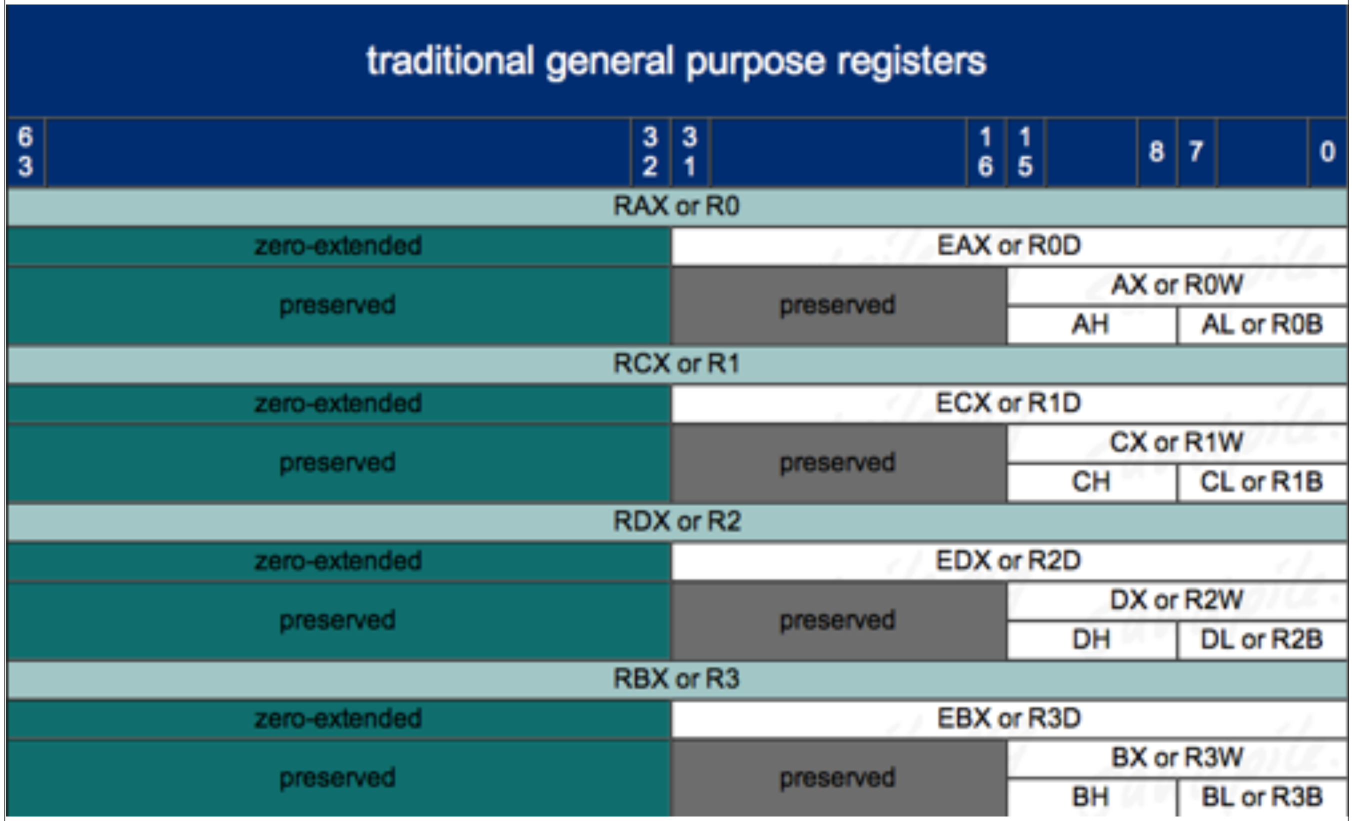
Intel Arch v1 Section 3.4.1 - General-Purpose Registers, page 3-11
Also MS's conventions: http://msdn.microsoft.com/en-us/library/9z1stfyw.aspx

http://msdn.microsoft.com/en-us/library/9z1stfyw.aspx

# Architecture - Registers Conventions 2

- **RSI** – Source pointer for string operations
- **RDI** – Destination pointer for string operations
- **RSP** – Stack top pointer
- **RBP** – Stack frame base pointer
- **RIP** - Pointer to next instruction to execute (“instruction pointer”)

# Architecture - Registers – 8/16/32/64 bit addressing 1



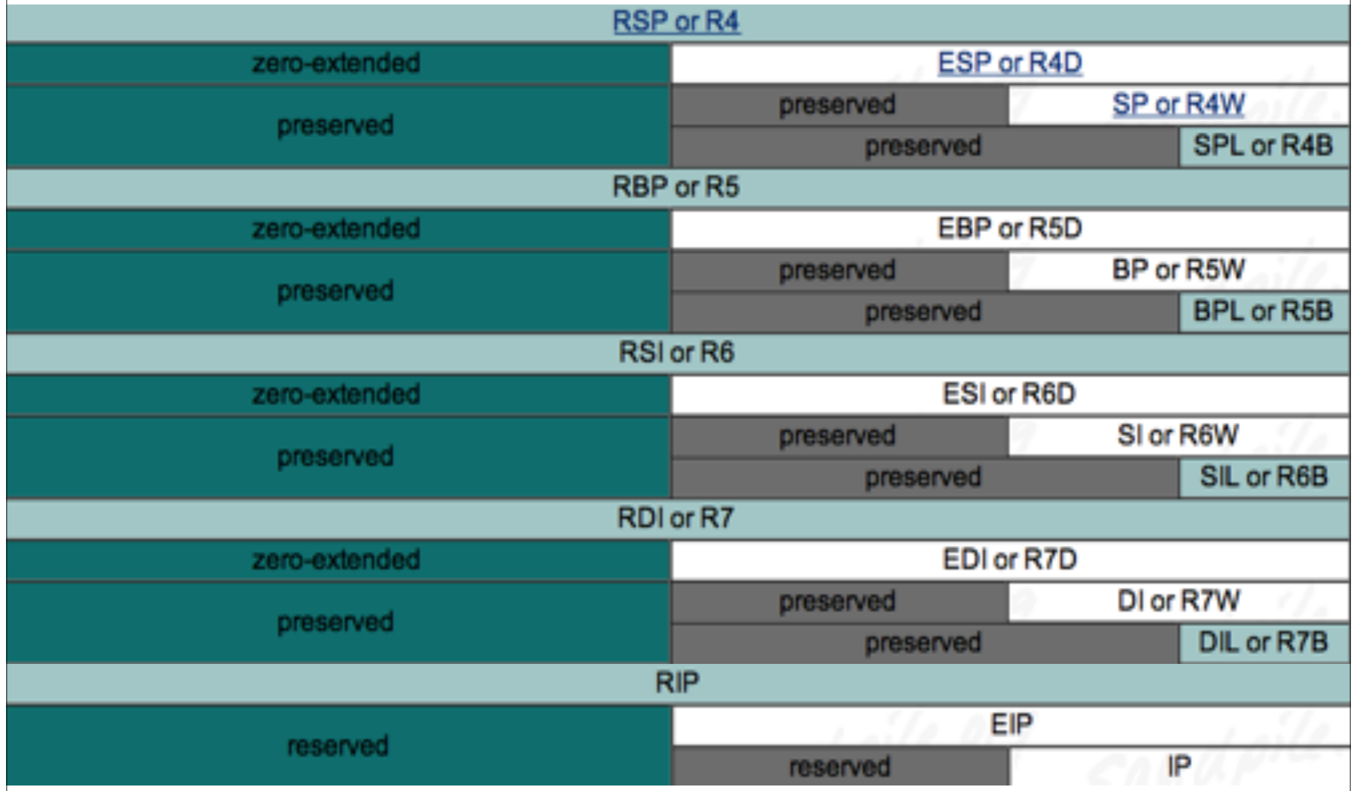
traditional general purpose registers

| 63 32 | 31 16 | 15 8 | 7 0 |
|---|---|---|---|
| RAX or R0 | | | |
| zero-extended | EAX or R0D | | |
| preserved | preserved | AX or R0W | |
| | | AH | AL or R0B |
| RCX or R1 | | | |
| zero-extended | ECX or R1D | | |
| preserved | preserved | CX or R1W | |
| | | CH | CL or R1B |
| RDX or R2 | | | |
| zero-extended | EDX or R2D | | |
| preserved | preserved | DX or R2W | |
| | | DH | DL or R2B |
| RBX or R3 | | | |
| zero-extended | EBX or R3D | | |
| preserved | preserved | BX or R3W | |
| | | BH | BL or R3B |

http://www.sandpile.org/x86/gpr.htm

# Architecture - Registers – 8/16/32/64 bit addressing 2

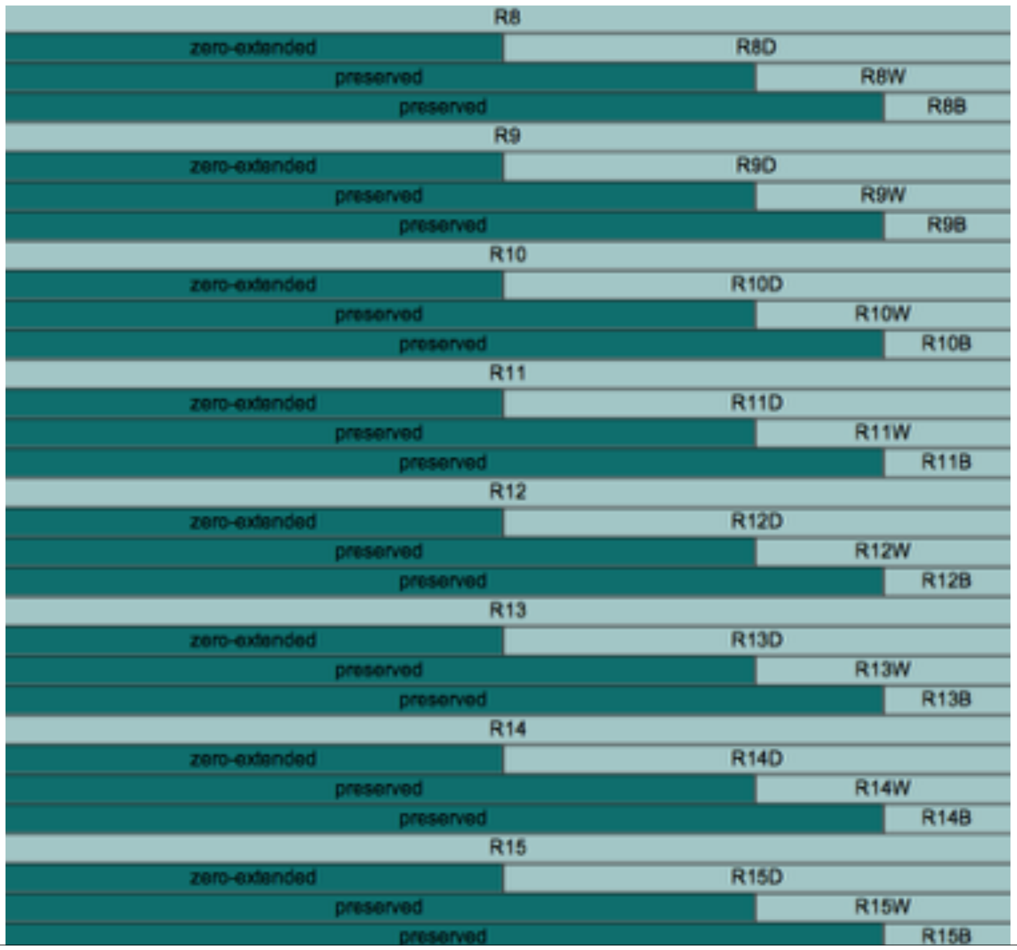
(note: we didn't previously have low-byte access to *SP, *BP, *SI, or *DI!)



| | | | |
|---|---|---|---|
| RSP or R4 | | | |
| zero-extended | ESP or R4D | | |
| preserved | preserved | SP or R4W | |
| | preserved | | SPL or R4B |
| RBP or R5 | | | |
| zero-extended | EBP or R5D | | |
| preserved | preserved | BP or R5W | |
| | preserved | | BPL or R5B |
| RSI or R6 | | | |
| zero-extended | ESI or R6D | | |
| preserved | preserved | SI or R6W | |
| | preserved | | SIL or R6B |
| RDI or R7 | | | |
| zero-extended | EDI or R7D | | |
| preserved | preserved | DI or R7W | |
| | preserved | | DIL or R7B |
| RIP | | | |
| reserved | EIP | | |
| | reserved | IP | |

http://www.sandpile.org/x86/gpr.htm

## Architecture - Registers – 8/16/32/64 bit addressing 3



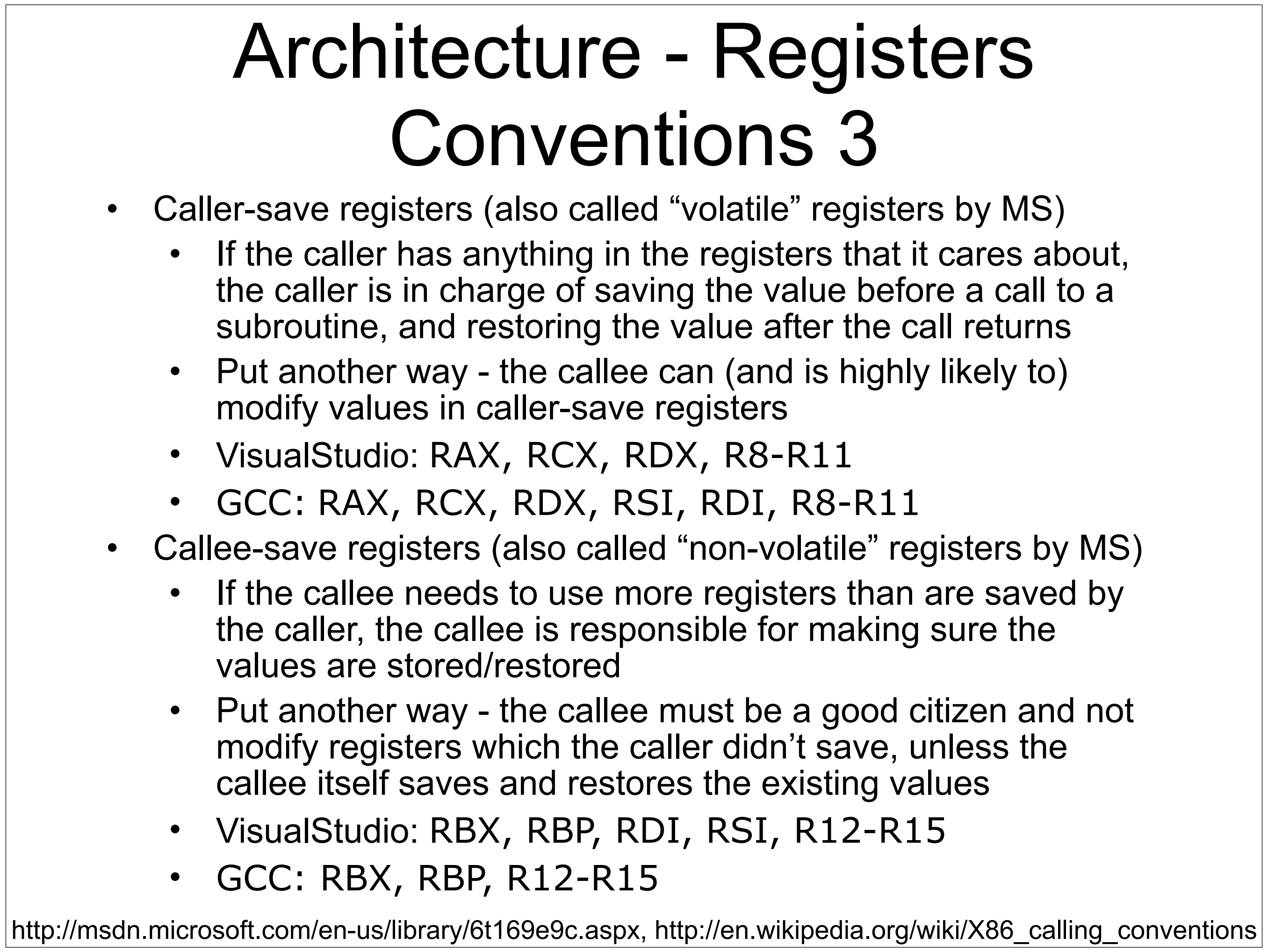
http://www.sandpile.org/x86/gpr.htm

note to scornwell: maybe a simple game here like:

“DIL is the 8 least-significant-bit access for which 64 bit register?” (RDI)

“What is the word-sized access register for R14 called?” (R14W)

“What is the 16 bit access for R14 called?” (R14W)



## Architecture - Registers Conventions 3

- Caller-save registers (also called “volatile” registers by MS)
  - If the caller has anything in the registers that it cares about, the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns
  - Put another way - the callee can (and is highly likely to) modify values in caller-save registers
  - VisualStudio: RAX, RCX, RDX, R8-R11
  - GCC: RAX, RCX, RDX, RSI, RDI, R8-R11
- Callee-save registers (also called “non-volatile” registers by MS)
  - If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored/restored
  - Put another way - the callee must be a good citizen and not modify registers which the caller didn’t save, unless the callee itself saves and restores the existing values
  - VisualStudio: RBX, RBP, RDI, RSI, R12-R15
  - GCC: RBX, RBP, R12-R15

http://msdn.microsoft.com/en-us/library/6t169e9c.aspx, http://en.wikipedia.org/wiki/X86_calling_conventions

http://msdn.microsoft.com/en-us/library/6t169e9c.aspx

http://msdn.microsoft.com/en-us/library/9z1stfyw.aspx

## Architecture - RFLAGS

- "In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS."
- RFLAGS register holds many single bit flags. Will only ask you to remember the following for now.
  - Zero Flag (ZF) - Set if the result of some instruction is zero; cleared otherwise.
  - Sign Flag (SF) - Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)

Intel Vol 1 Sec 3.4.3.1 - page 3-22 - May 2012 manuals

🟦 = Intro x86-64

🟩 = Intermediate x86-64

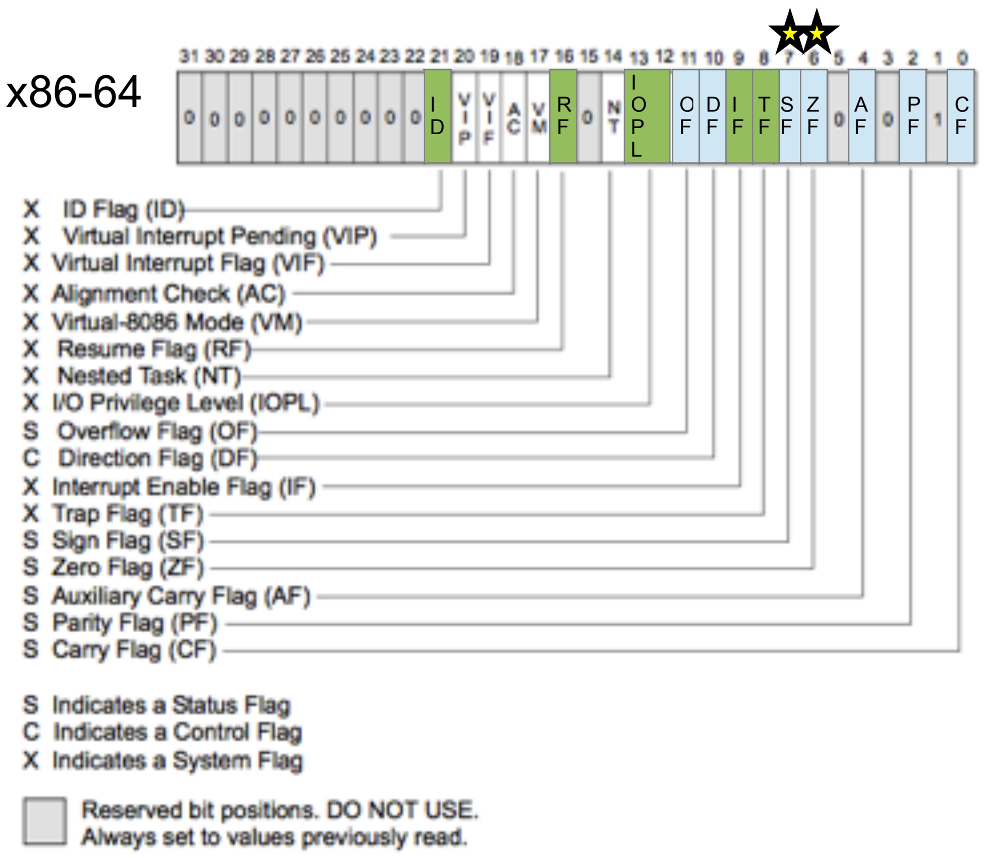
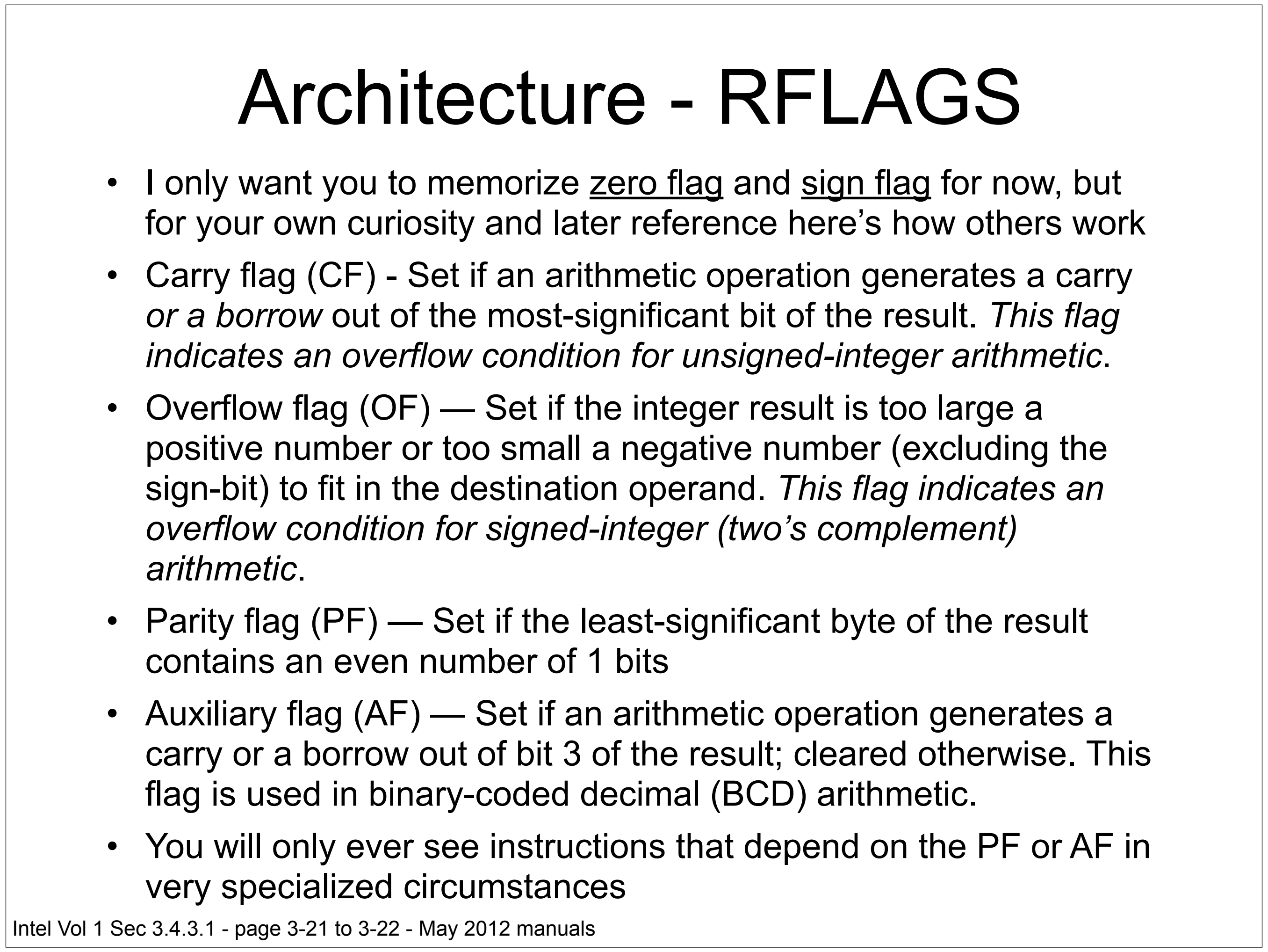

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume 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.

**Figure 3-8. EFLAGS Register**



## Architecture - RFLAGS

- I only want you to memorize zero flag and sign flag for now, but for your own curiosity and later reference here's how others work
- Carry flag (CF) - Set if an arithmetic operation generates a carry *or a borrow* out of the most-significant bit of the result. *This flag indicates an overflow condition for unsigned-integer arithmetic*.
- Overflow flag (OF) — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand. *This flag indicates an overflow condition for signed-integer (two's complement) arithmetic*.
- Parity flag (PF) — Set if the least-significant byte of the result contains an even number of 1 bits
- Auxiliary flag (AF) — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
- You will only ever see instructions that depend on the PF or AF in very specialized circumstances

Intel Vol 1 Sec 3.4.3.1 - page 3-21 to 3-22 - May 2012 manuals

Note to scornwell: In one-step, three-step, we can teach PF and AF-based jumps in the later advanced rounds. It should start with SF/ZF-based ones, then move on to CF/OF-based ones, and then eventually PF/AF-based ones

1

## Your first x86-64 instruction: NOP

- NOP - No Operation! No registers, no values, no nothin'!
- Just there to pad/align bytes, or to delay time
- Bad guys use it to make simple exploits more reliable. But that's another class ;)
- OpenSecurityTraining.info/Exploits1.html

## Extra! Extra! Late-breaking NOP news!

- Amaze those who know x86 by citing this interesting bit of trivia:
- “The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.”
  - I had never looked in the manual for NOP apparently :)
- Every other person I had told this to had never heard it either.
- Thanks to Jon Erickson for cluing me in to this.
- XCHG instruction is not officially in this class. But if I hadn’t just told you what it does, I bet you would have guessed right anyway.

## Instructions we now know (1)

- NOP