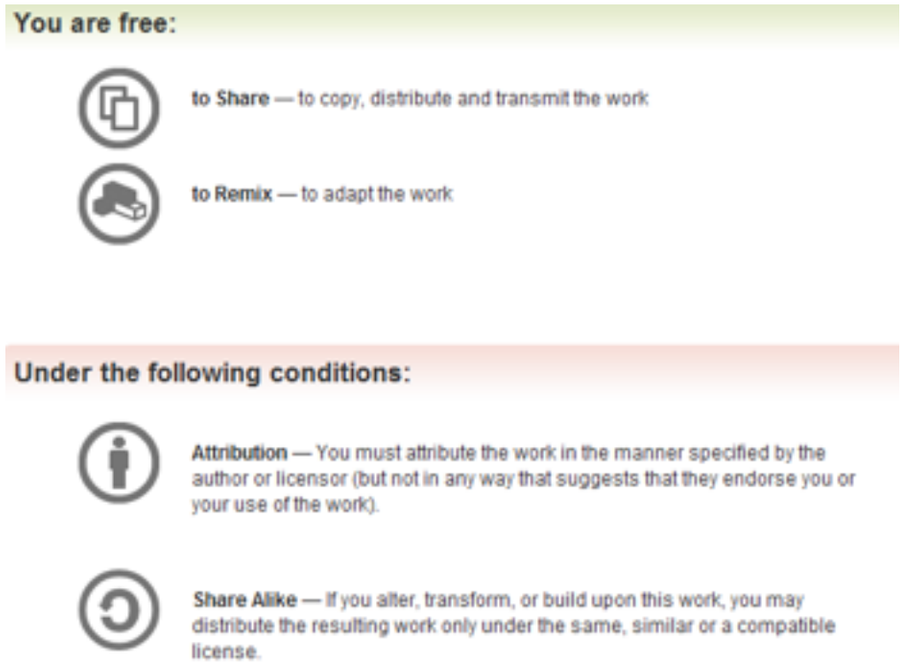
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014

xkovah at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

Guess what?
I have repeatedly misled you!

- Simplification is misleading
- Time to learn the *fascinating* truth…
- Time to RTFM!

Read The Fun Manuals

- http://www.intel.com/products/processor/manuals/
- Vol.1 is a summary of life, the universe, and everything about x86
- Vol. 2a & 2b explains all the instructions
- Vol. 3a & 3b are all the gory details for all the extra stuff they've added in over the years (MultiMedia eXtentions - MMX, Virtual Machine eXtentions - VMX, virtual memory, 16/64 bit modes, system management mode, etc)
- Reminder, we're using the pre-downloaded May 2012 version as the standardized reference throughout this class so we're all looking at the same information
- We'll only be looking at Vol. 2a & 2b in this class

Googling is fine to start with, but eventually you need to learn to read the manuals to get the details from the authoritative source

Interpreting the Instruction Reference Pages

- The correct way to interpret these pages is given in the Intel Manual 2a, section 3.1
- I will give yet another simplification
- Moral of the story is that you have to RTFM to RTFM ;)

Here's what I said:

AND - Logical AND

- Destination operand can be r/mX or register
- Source operand can be r/mX or register or immediate (No source *and* destination as r/mXs at the same time)

and al, bl

	00110011b (al - 0x33)
AND	01010101b (bl - 0x55)
result	00010001b (al - 0x11)

and al, 0x42

	00110011b (al - 0x33)
AND	01000010b (imm - 0x42)
result	00000010b (al - 0x02)

Here's what the manual says:

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.
80 /4 *ib*	AND *r/m8*, *imm8*	MR	Valid	Valid	*r/m8* AND *imm8*.
REX + 80 /4 *ib*	AND *r/m8**, *imm8*	MR	Valid	N.E.	*r/m8* AND *imm8*.
81 /4 *iw*	AND *r/m16*, *imm16*	MR	Valid	Valid	*r/m16* AND *imm16*.
81 /4 *id*	AND *r/m32*, *imm32*	MR	Valid	Valid	*r/m32* AND *imm32*.
REX.W + 81 /4 *id*	AND *r/m64*, *imm32*	MR	Valid	N.E.	*r/m64* AND *imm32* sign extended to 64-bits.
83 /4 *ib*	AND *r/m16*, *imm8*	MR	Valid	Valid	*r/m16* AND *imm8* (sign-extended).
83 /4 *ib*	AND *r/m32*, *imm8*	MR	Valid	Valid	*r/m32* AND *imm8* (sign-extended).
REX.W + 83 /4 *ib*	AND *r/m64*, *imm8*	MR	Valid	N.E.	*r/m64* AND *imm8* (sign-extended).
20 /r	AND *r/m8*, *r8*	MI	Valid	Valid	*r/m8* AND *r8*.
REX + 20 /r	AND *r/m8**, *r8**	MI	Valid	N.E.	*r/m64* AND *r8* (sign-extended).
21 /r	AND *r/m16*, *r16*	MI	Valid	Valid	*r/m16* AND *r16*.
21 /r	AND *r/m32*, *r32*	MI	Valid	Valid	*r/m32* AND *r32*.
REX.W + 21 /r	AND *r/m64*, *r64*	MI	Valid	N.E.	*r/m64* AND *r32*.
22 /r	AND *r8*, *r/m8*	I	Valid	Valid	*r8* AND *r/m8*.
REX + 22 /r	AND *r8**, *r/m8**	I	Valid	N.E.	*r/m64* AND *r8* (sign-extended).
23 /r	AND *r16*, *r/m16*	I	Valid	Valid	*r16* AND *r/m16*.
23 /r	AND *r32*, *r/m32*	I	Valid	Valid	*r32* AND *r/m32*.
REX.W + 23 /r	AND *r64*, *r/m64*	I	Valid	N.E.	*r64* AND *r/m64*.

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- Opcode Column
- Represents the literal byte value(s) which correspond to the given instruction
- In this case, if you were to see a 0x24 followed by a byte or 0x25 followed by 4 bytes, you would know they were specific forms of the AND instruction.
  - Subject to correct interpretation under x86's multi-byte opcodes as discussed later.

See Intel Vol. 2a section 3.1.1.1 ("Opcode Column in the Instruction Summary Table")

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- If it was 0x25, how would you know whether it should be followed by 2 bytes (imm16) or 4 bytes (imm32)? Because the same single opcode byte is used for both, the length of the operand depends on if the processor is in 16-bit, 32-bit, or 64-bit mode. Each mode has a default operand size (i.e. the size of the value).
- For 64-bit mode, the default operand size is 32-bits for most instructions and the default address size is 64-bits
- This means the default interpretation will usually be the ones with the r/m32, r32, imm32, or in this case a specific register like EAX

There are many instructions which are “overloaded” with equivalent 16 bit and 32 bit versions shown in the manual.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32 sign-extended to 64-bits.*

- In some cases the operand or address size can be overridden with special prefix bytes that come before the regular instruction opcode
- There are REX prefixes, address size prefixes, and operand size prefixes
- Will not go into detail for all of them, but the REX.W byte shown in this example (0x48) will cause the instruction to use 64-bit operands if in 64-bit mode (rather than 32-bit operands)
- Therefore, to encode this instruction to use 64-bit operands (RAX in this case), the code would have byte sequence 0x48 0x25 …

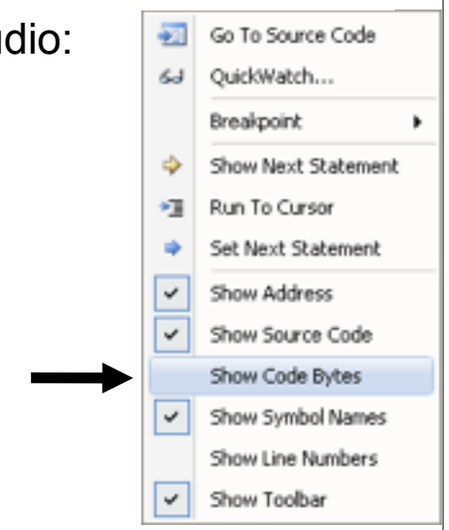
There are many instructions which are “overloaded” with equivalent 16 bit and 32 bit versions shown in the manual.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- How to see the opcodes in VisualStudio:
- Seeing the exact opcode will help confirm the exact version of an Instruction

(to show bytes in gdb, use: disassemble/r optionally passing an address to disassemble at)



AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- Instruction Column
- The human-readable mnemonic which is used to represent the instruction.
- This will frequently contain special encodings such as the “r/mX format” which I’ve previously discussed

See Intel Vol. 2a section 3.1.1.3 (Instruction Column in the Opcode Summary Table)

AND—Logical AND

Should be I, fixed in latest

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8.*
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16.*
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32.*
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32 sign-extended to 64-bits.*

Should be RI, fixed in latest

- Operand Encoding Column
- This column was added in more recent manuals. I would find it more useful it there weren't so many errors :-/

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Should allow for imm8/16/32, not fixed in latest

See Intel Vol. 2a section 3.1.1.4

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32 sign-extended to 64-bits*.

- 64bit Column
- Whether or not the opcode is valid in 64 bit mode.

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- Compatibility/Legacy Mode Column
- Whether or not the opcode is valid in 32/16 bit code.
  - The N.E. Indicates an an instruction encoding which is only encodable in 64-bit mode

See Intel Vol. 2a section 3.1.1.5
"64/32-bit Mode Column in the Instruction Summary Table"

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	RM	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	RM	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	RM	Valid	Valid	EAX AND *imm32*.
REX.W + 25 *id*	AND RAX, *imm32*	RM	Valid	N.E.	RAX AND *imm32* sign-extended to 64-bits.

- Description Column
- Simple description of the action performed by the instruction
- Typically this just conveys the flavor of the instruction, but the majority of the details are in the main description text

See Intel Vol. 2a section 3.1.1.7
“Description Column in the Instruction Summary Table”

80 /4 *ib*	AND *r/m8, imm8*	Valid	Valid	*r/m8* AND *imm8*.
REX + 80 /4 *ib*	AND *r/m8**, *imm8*	Valid	N.E.	*r/m64* AND *imm8 (sign-extended).*
81 /4 *iw*	AND *r/m16, imm16*	Valid	Valid	*r/m16* AND *imm16*.
81 /4 *id*	AND *r/m32, imm32*	Valid	Valid	*r/m32* AND *imm32*.

- Looking at some other forms, we now see those “r/mX” things I told you about
- We know that for instance it can start with an 0x80, and end with a byte, but what’s that /4?
- Unfortunately the explanation goes into too much detail for this class. Generally the only people who need to know it are people who want to write disassemblers. But I still put it in the Intermediate x86 class :)
- The main thing you need to know is that any time you see a r/mX, it can be either a register or memory value.

AND Details

- Description
  - "Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

    This instruction can be used with a LOCK prefix to allow the it to be executed atomically."
- Flags effected
  - "The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined."

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 cb	JA *rel8*	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 cb	JAE *rel8*	D	Valid	Valid	Jump short if above or equal (CF=0).
72 cb	JB *rel8*	D	Valid	Valid	Jump short if below (CF=1).
76 cb	JBE *rel8*	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 cb	JC *rel8*	D	Valid	Valid	Jump short if carry (CF=1).
E3 cb	JCXZ *rel8*	D	N.E.	Valid	Jump short if CX register is 0.
E3 cb	JECXZ *rel8*	D	Valid	Valid	Jump short if ECX register is 0.
E3 cb	JRCXZ *rel8*	D	Valid	N.E.	Jump short if RCX register is 0.
74 cb	JE *rel8*	D	Valid	Valid	Jump short if equal (ZF=1).
7F cb	JG *rel8*	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D cb	JGE *rel8*	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C cb	JL *rel8*	D	Valid	Valid	Jump short if less (SF≠ OF).
7E cb	JLE *rel8*	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 cb	JNA *rel8*	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 cb	JNAE *rel8*	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 cb	JNB *rel8*	D	Valid	Valid	Jump short if not below (CF=0).
77 cb	JNBE *rel8*	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 cb	JNC *rel8*	D	Valid	Valid	Jump short if not carry (CF=0).
75 cb	JNE *rel8*	D	Valid	Valid	Jump short if not equal (ZF=0).
7E cb	JNG *rel8*	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).

Jcc Revisited

- If you look closely, you will see that there are multiple mnemonics for the same opcodes
- 0x77 = JA - Jump Above
- 0x77 = JNBE - Jump Not Below or Equal
- 0x74 = JE / JZ - Jump Equal / Zero
- Which mnemonic is displayed is disassembler-dependent

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL *r/m8**	M	Valid	Valid	AX← AL * *r/m* byte.
F7 /5	IMUL *r/m16*	M	Valid	Valid	DX:AX ← AX * *r/m* word.
F7 /5	IMUL *r/m32*	M	Valid	Valid	EDX:EAX ← EAX * *r/m32*.
REX.W + F7 /5	IMUL *r/m64*	M	Valid	N.E.	RDX:RAX ← RAX * *r/m64*.
0F AF /r	IMUL *r16, r/m16*	RM	Valid	Valid	word register ← word register * *r/m16*.
0F AF /r	IMUL *r32, r/m32*	RM	Valid	Valid	doubleword register ← doubleword register * *r/m32*.
REX.W + 0F AF /r	IMUL *r64, r/m64*	RM	Valid	N.E.	Quadword register ← Quadword register * *r/m64*.
6B /r *ib*	IMUL *r16, r/m16, imm8*	RMI	Valid	Valid	word register ← *r/m16* * sign-extended immediate byte.
6B /r *ib*	IMUL *r32, r/m32, imm8*	RMI	Valid	Valid	doubleword register ← *r/m32* * sign-extended immediate byte.
REX.W + 6B /r *ib*	IMUL *r64, r/m64, imm8*	RMI	Valid	N.E.	Quadword register ← *r/m64* * sign-extended immediate byte.
69 /r *iw*	IMUL *r16, r/m16, imm16*	RMI	Valid	Valid	word register ← *r/m16* * immediate word.
69 /r *id*	IMUL *r32, r/m32, imm32*	RMI	Valid	Valid	doubleword register ← *r/m32* * immediate doubleword.
REX.W + 69 /r *id*	IMUL *r64, r/m64, imm32*	RMI	Valid	N.E.	Quadword register ← *r/m64* * immediate doubleword.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA

IMUL Revisited

- Scavenger hunt: for “extra credit” (i.e. getting credited in the slides ;)) find me another “basic” instruction, that’s not part of a special add-on instruction set (like VMX, SMX, MMX, SSE*, AES, AVX, etc) and isn’t a floating point instruction, which uses >= 3 operands
- hint: if you see a “CPUID feature flag” column, it means it’s a special instruction set

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA