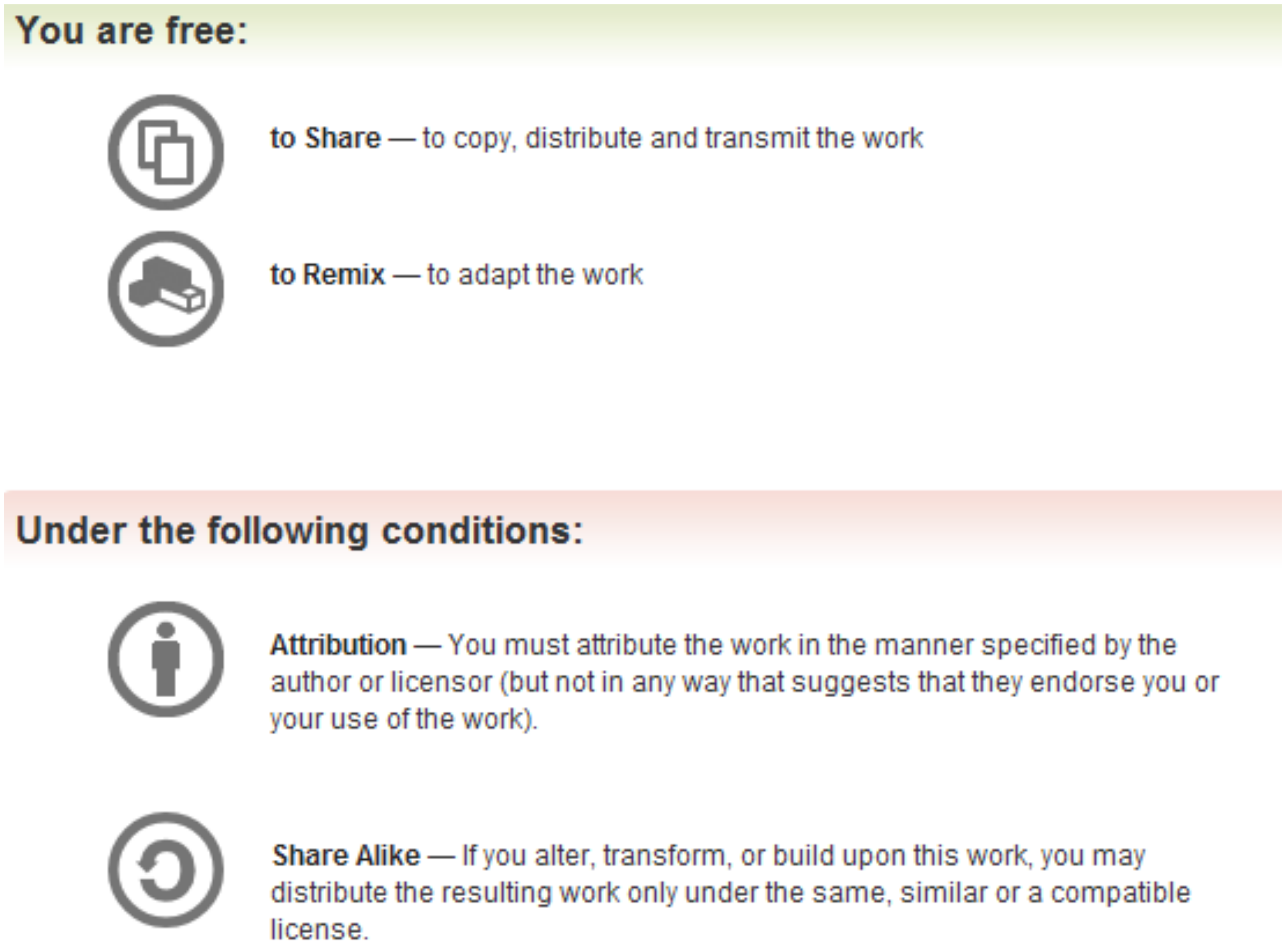
Intro x86 Part 4: Inline Assembly, Read The Fun Manual, Choose Your Own Adventure

Xeno Kovah – 2009/2010

xkovah at gmail

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

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



Inline assembly

- Inline assembly is a way to include assembly directly in a C/C++ file. However, the syntax will differ between compilers and assemblers.
- There are times when you actually have to code asm in order to do something from a C/C++ file.
  - Very common in OS or driver design, because there are many aspects of hardware which can only be accessed with special instructions
  - In crypto you might want access to the "rol/ror - rotate left/right" instructions which don't have corresponding C syntax like shifts do
- Or maybe you just want full control over the code being generated for optimization purposes
  - Keep in mind the compiler may still optimize your inline asm
- Also it's a great way to simply experiment with instructions
  - Though getting the syntax right for the desired instructions is sometimes annoying

VisualStudio inline assembly

- VisualStudio syntax - intel-syntax
- __asm{ instructions separated by \n};
  - That's two underscores at the beginning
  - Don't even need a semicolon after it, but I put them there since it makes the auto-indent work correctly

```
__asm{          mov eax, [esp+0x4]
                cmp eax, 0xdeadbeef
                je myLabel
                xor eax, eax
myLabel:        mov bl, al
};
```

VisualStudio inline assembly 2

- Syntax using C variables is the same, just put the variable in place of a register name for instance. (The assembler will substitute the correct address for the variable.)
- http://msdn.microsoft.com/en-us/library/4ks26t93(VS.80).aspx

```
int myVar;
//value into C variable from register
__asm {mov myVar, eax};
//value into register from C variable
__asm {mov eax, myVar};
```

GCC inline assembly

- GCC syntax - AT&T syntax
- asm(“instructions separated by \n”);
  - **DO** need a semicolon after close parentheses

```
asm("movl 0x4(%esp), %eax\n"
        "cmp $0xdeadbeef,%eax\n"
        "je myLabel\n"
        "xor %eax, %eax\n"
        "myLabel: movw %bx, %ax"
);
```

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

Book starting on p. 365

GCC inline assembly 2

- Syntax using C variables (aka “extended asm”):

```
asm ( assembler template
: output operands                   /* optional */
: input operands                    /* optional */
: list of clobbered registers       /* optional */
);
```

```
int myVar;
//value into C variable from register
asm ("movl %%eax, %0" : "=r" (myVar) );
//value into register from C variable
asm ("movl %0, %%eax" : : "r" (myVar) );
```

_emit and .byte

- Once you learn about opcodes later on, you can even specify exactly the instructions you want to use by using the “_emit” or “.byte” keywords, to place specific bytes into the code.
- Those bytes can then be interpreted as instructions or data
- This is sometimes useful if you can’t figure out the inline asm syntax for the instruction you want to use, but you know its opcodes (either from seeing them elsewhere, or by reading the manual)
- Examples:
  - __asm{_emit 0x55} is __asm{push ebp}
  - __asm{_emit 0x89};__asm{_emit 0xE5} is __asm{mov ebp, esp}
  - asm(“.byte 0x55”); is asm(“push %ebp”);
  - asm(".byte 0x89 ; .byte 0xE5"); is asm(“mov %esp, %ebp”);

Guess what?
I have repeatedly mislead 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)
- Already downloaded to the Manuals folder
- We'll only be looking at Vol. 2a & 2b in this class

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/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

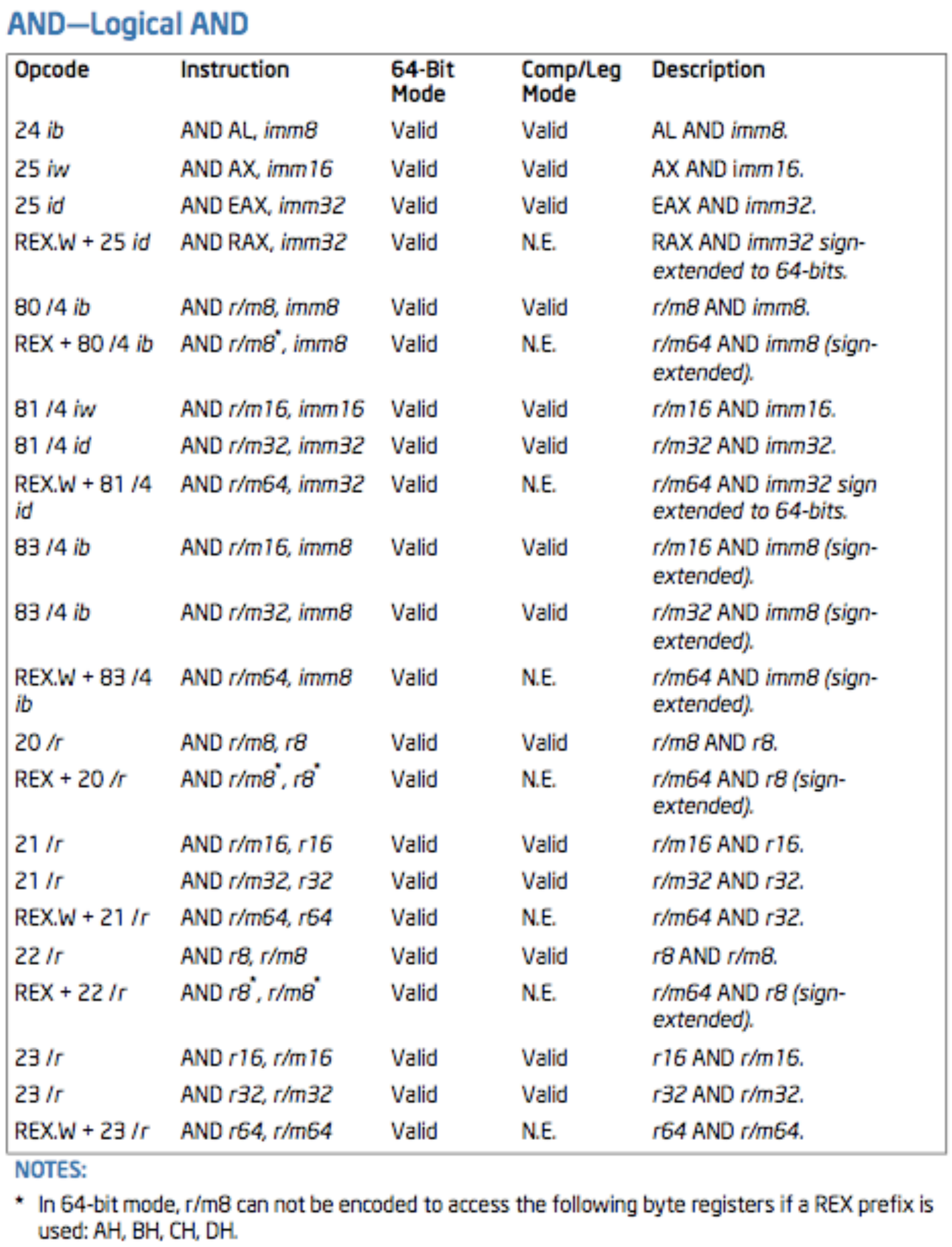
and al, bl

	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:

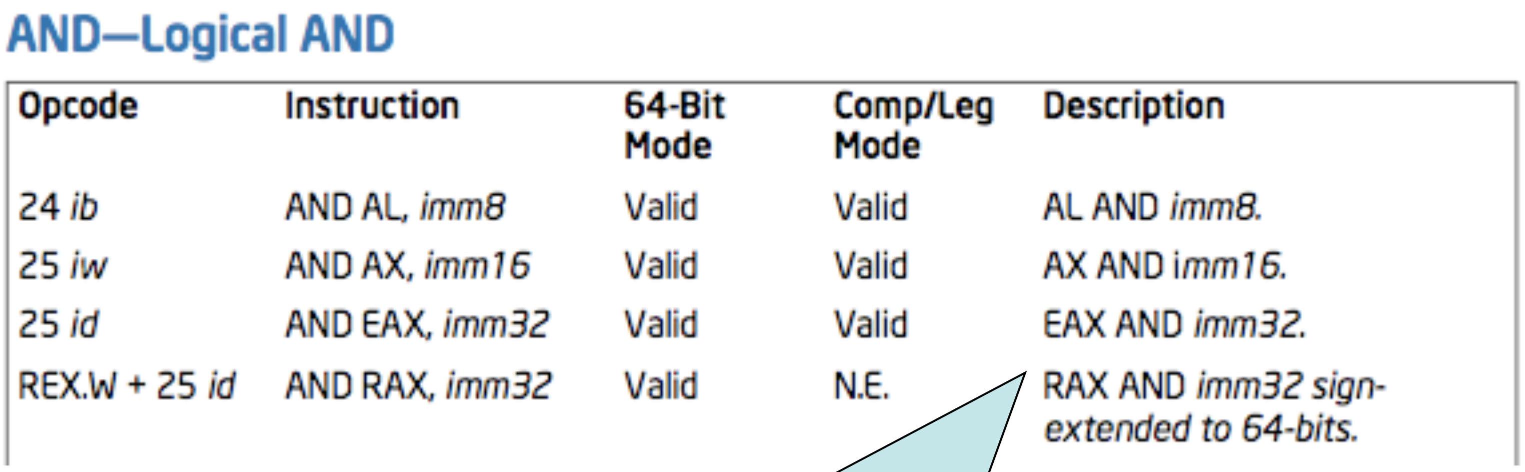


AND—Logical AND

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

NOTES:

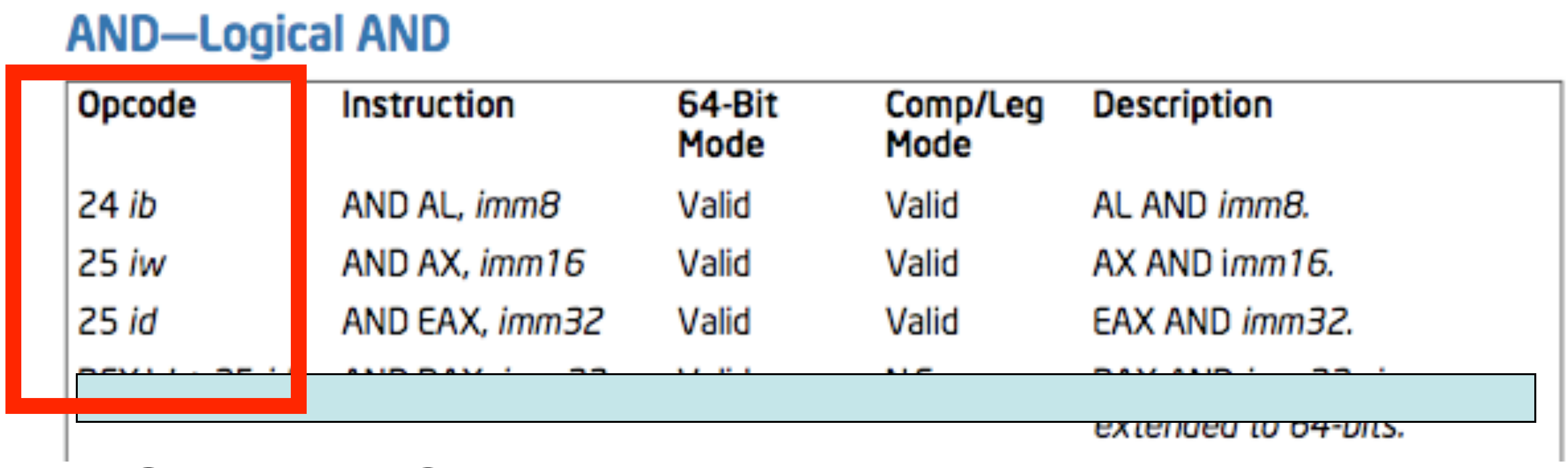
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.



AND—Logical AND

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

Ignore this line. Register names beginning with R refer to 64bit registers, and are not relevant for this class

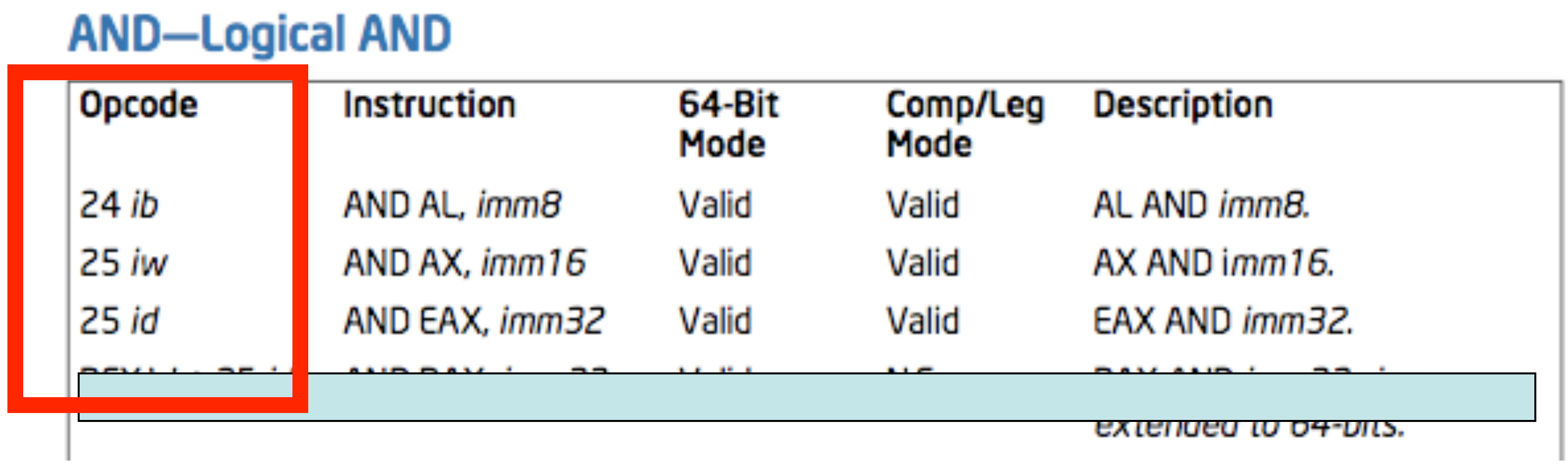


AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] 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.

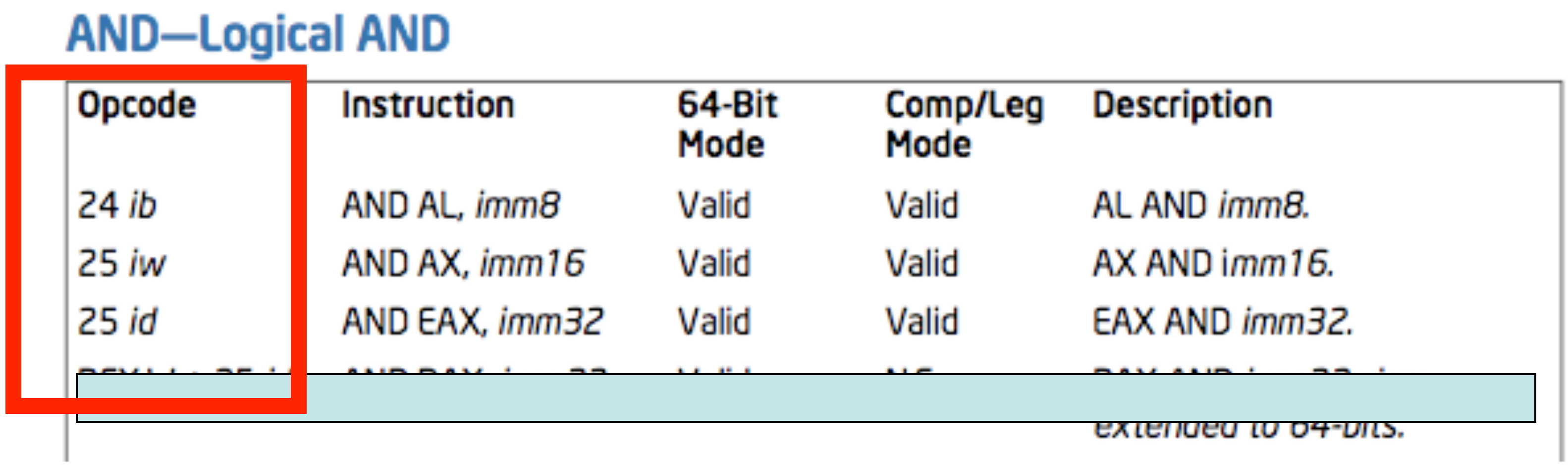
See Intel Vol. 2a section 3.1.1.1



AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
				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 32bit or 16bit mode. Because we're only considering 32bit mode in this class, the 4 bytes ("id" aka "imm32" aka "dword") following 0x25 will always be considered the operand to the instruction.



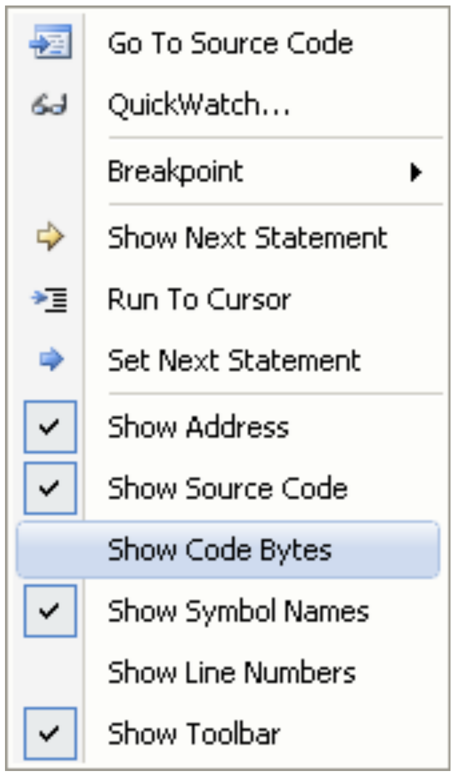
AND—Logical AND

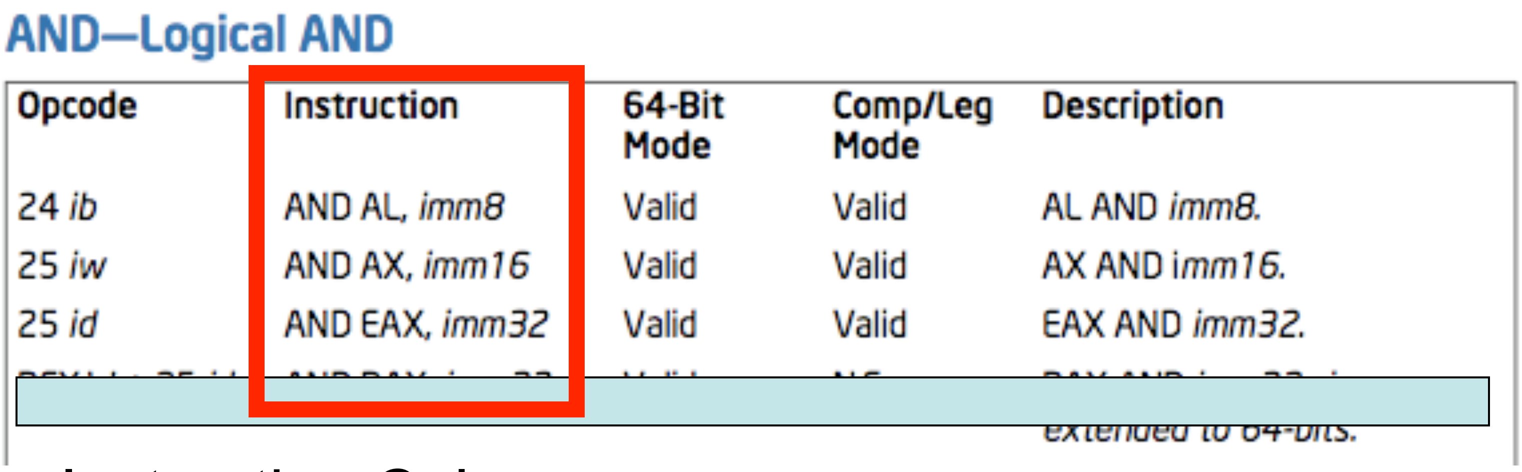
Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] extended to 64-bits.

- How to see the opcodes in VisualStudio:
- Seeing the exact opcode will

help confirm the exact version of an Instruction

(I couldn't find a decent way to do it in gdb besides using "x/<num>xb <addr>")



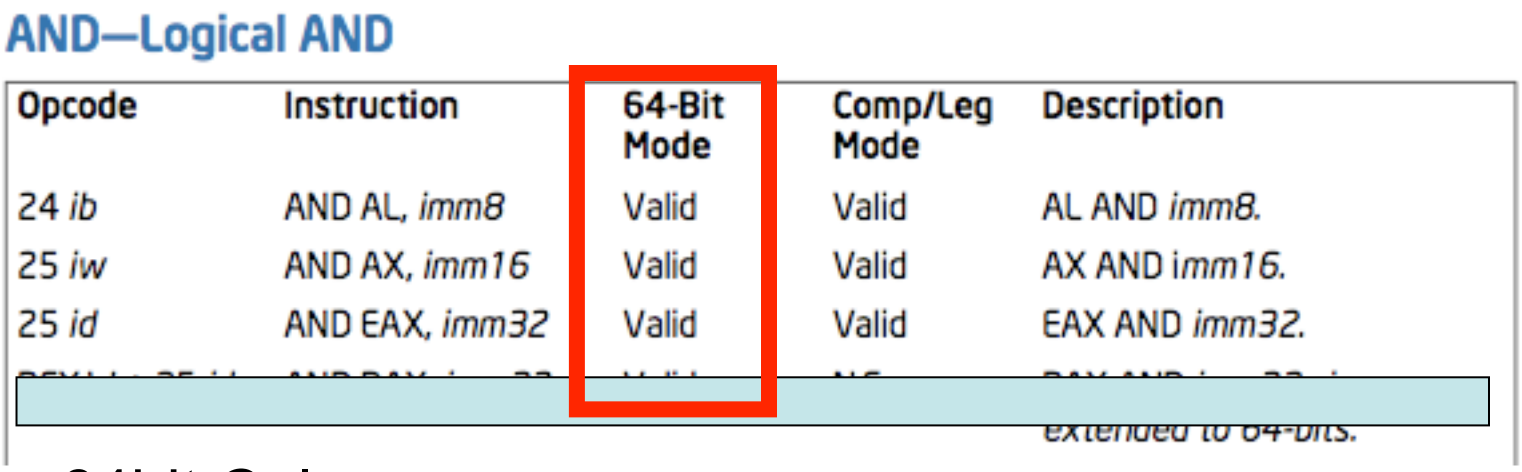


AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] 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/m32 format” which I’ve previously discussed

See Intel Vol. 2a section 3.1.1.2

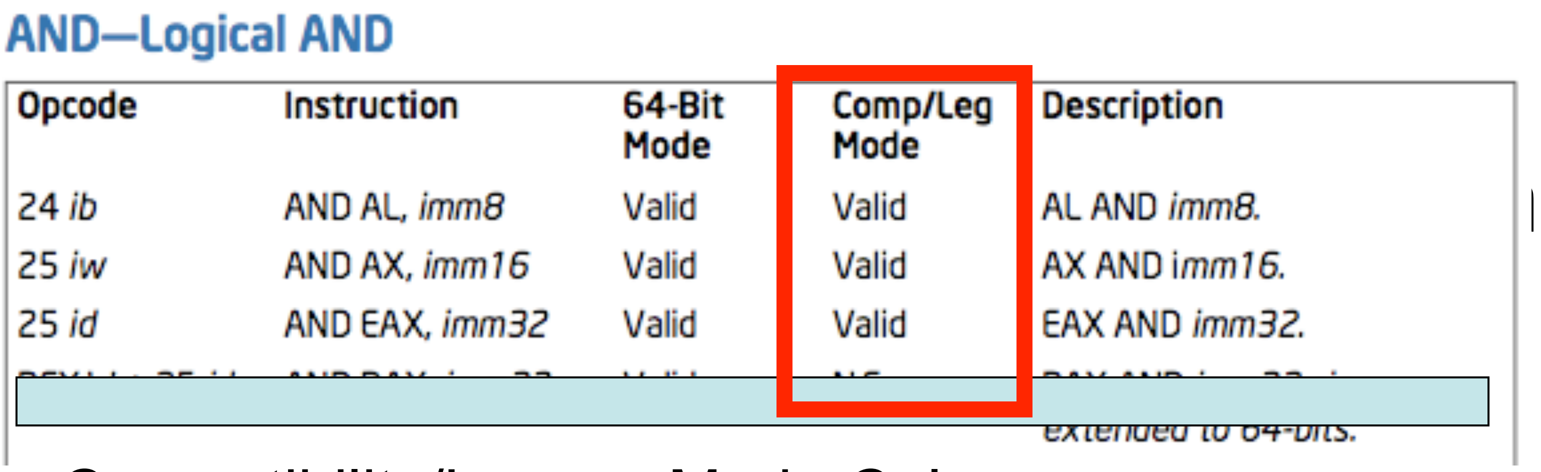


AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8.*
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16.*
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32.*
				extended to 64-bits.

- 64bit Column
- Whether or not the opcode is valid in 64 bit code.
- Can be ignored for our purposes.

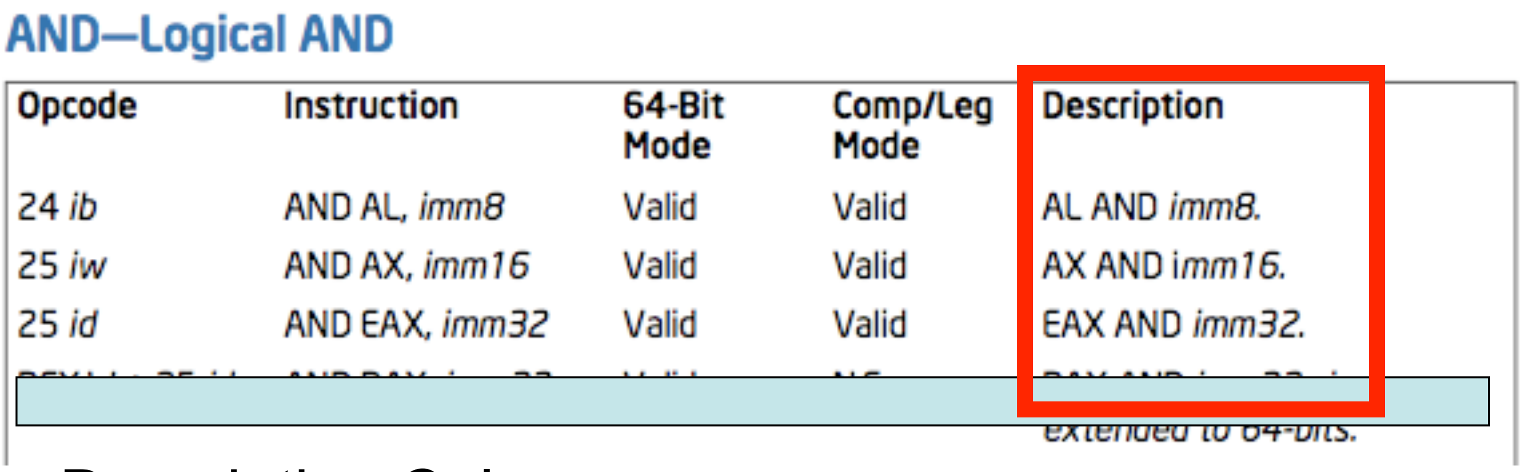
See Intel Vol. 2a section 3.1.1.3


AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] extended to 64-bits.

- Compatibility/Legacy Mode Column
- Whether or not the opcode is valid in 32/16 bit code.
  - For 64 bit instructions, the N.E. Indicates an Intel 64 instruction mnemonics/syntax that is not encodable”
- Can be ignored for our purposes.

See Intel Vol. 2a section 3.1.1.4

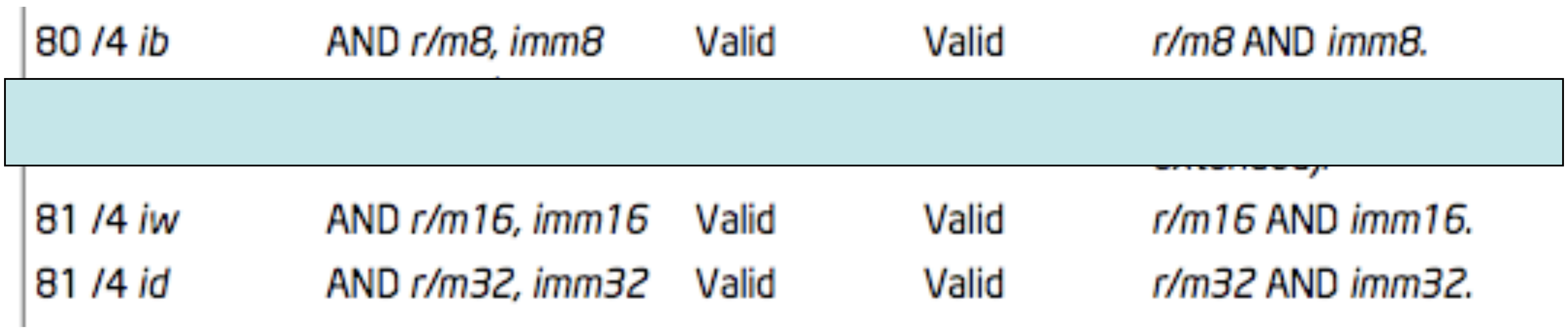


AND—Logical AND

Opcode	Instruction	64-Bit Mode	Comp/Leg Mode	Description
24 *ib*	AND AL, *imm8*	Valid	Valid	AL AND *imm8*.
25 *iw*	AND AX, *imm16*	Valid	Valid	AX AND *imm16*.
25 *id*	AND EAX, *imm32*	Valid	Valid	EAX AND *imm32*.
[illegible]	[illegible]	[illegible]	[illegible]	[illegible] 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 3.1.1.5



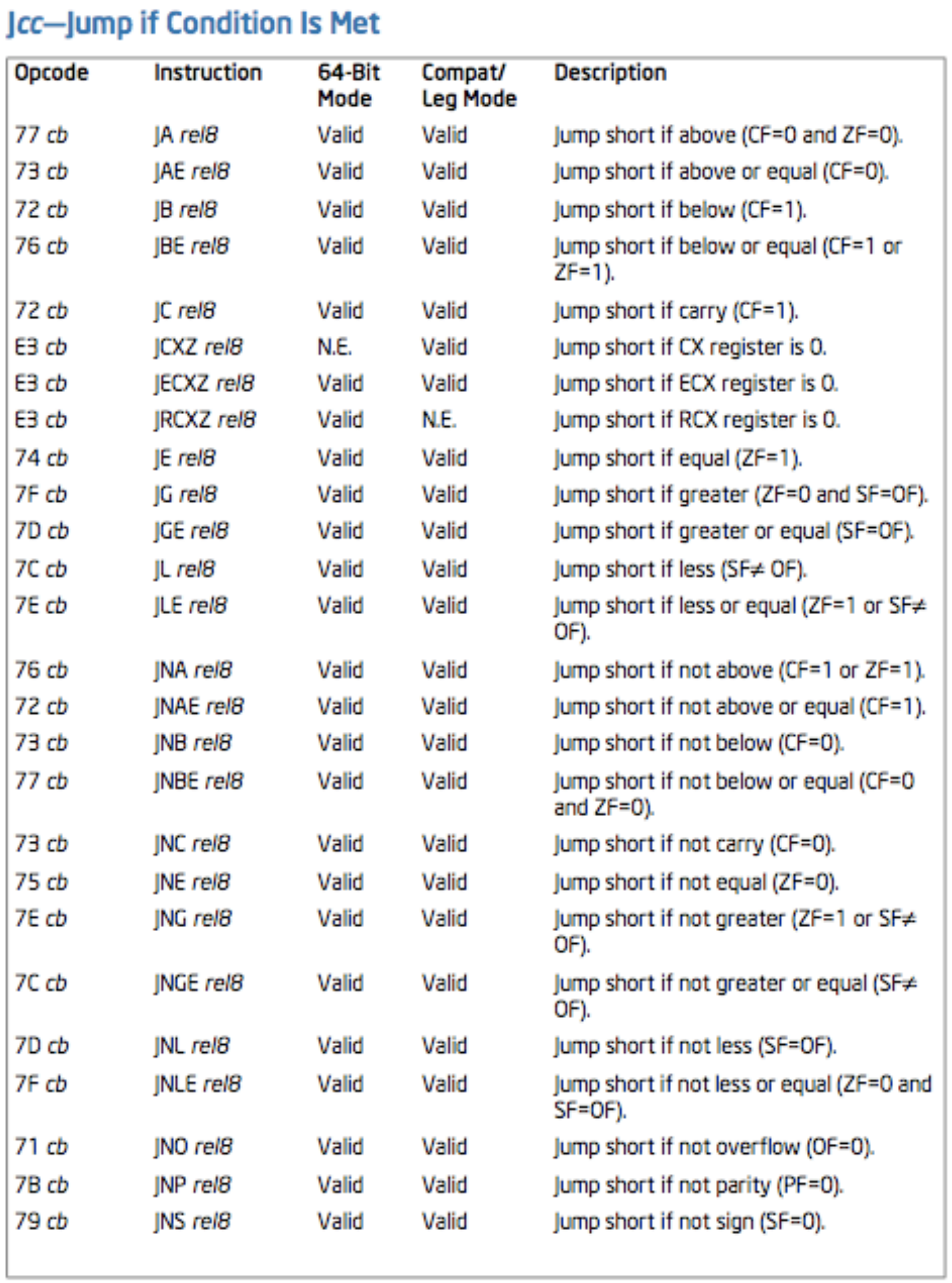
80 /4 *ib*	AND *r/m8, imm8*	Valid	Valid	*r/m8* AND *imm8.*
81 /4 *iw*	AND *r/m16, imm16*	Valid	Valid	*r/m16* AND *imm16.*
81 /4 *id*	AND *r/m32, imm32*	Valid	Valid	*r/m32* AND *imm32.*

- Looking at some other forms, we now see those "r/m32" 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 :)
- All you *really* need to know is that any time you see a r/m8 or r/m32, 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	64-Bit Mode	Compat/ Leg Mode	Description
77 *cb*	JA *rel8*	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	Valid	Valid	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	Valid	Valid	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	Valid	Valid	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	N.E.	Valid	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	Valid	Valid	Jump short if ECX register is 0.
E3 *cb*	JRCXZ *rel8*	Valid	N.E.	Jump short if RCX register is 0.
74 *cb*	JE *rel8*	Valid	Valid	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	Valid	Valid	Jump short if greater or equal (SF=OF).
7C *cb*	JL *rel8*	Valid	Valid	Jump short if less (SF≠ OF).
7E *cb*	JLE *rel8*	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 *cb*	JNA *rel8*	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 *cb*	JNAE *rel8*	Valid	Valid	Jump short if not above or equal (CF=1).
73 *cb*	JNB *rel8*	Valid	Valid	Jump short if not below (CF=0).
77 *cb*	JNBE *rel8*	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 *cb*	JNC *rel8*	Valid	Valid	Jump short if not carry (CF=0).
75 *cb*	JNE *rel8*	Valid	Valid	Jump short if not equal (ZF=0).
7E *cb*	JNG *rel8*	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C *cb*	JNGE *rel8*	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D *cb*	JNL *rel8*	Valid	Valid	Jump short if not less (SF=OF).
7F *cb*	JNLE *rel8*	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 *cb*	JNO *rel8*	Valid	Valid	Jump short if not overflow (OF=0).
7B *cb*	JNP *rel8*	Valid	Valid	Jump short if not parity (PF=0).
79 *cb*	JNS *rel8*	Valid	Valid	Jump short if not sign (SF=0).

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

How about looking at the manual when a new instruction is encountered?

```
//Example6.c
int main(){
        unsigned int a, b, c;
        a = 0x40;
        b = a * 8;
        c = b / 16;
        return c;
}
```

$\longrightarrow$

```
//Example6-mod.c
int main(){
        int a, b, c;
        a = 0x40;
        b = a * 8;
        c = b / 16;
        return c;
}
```

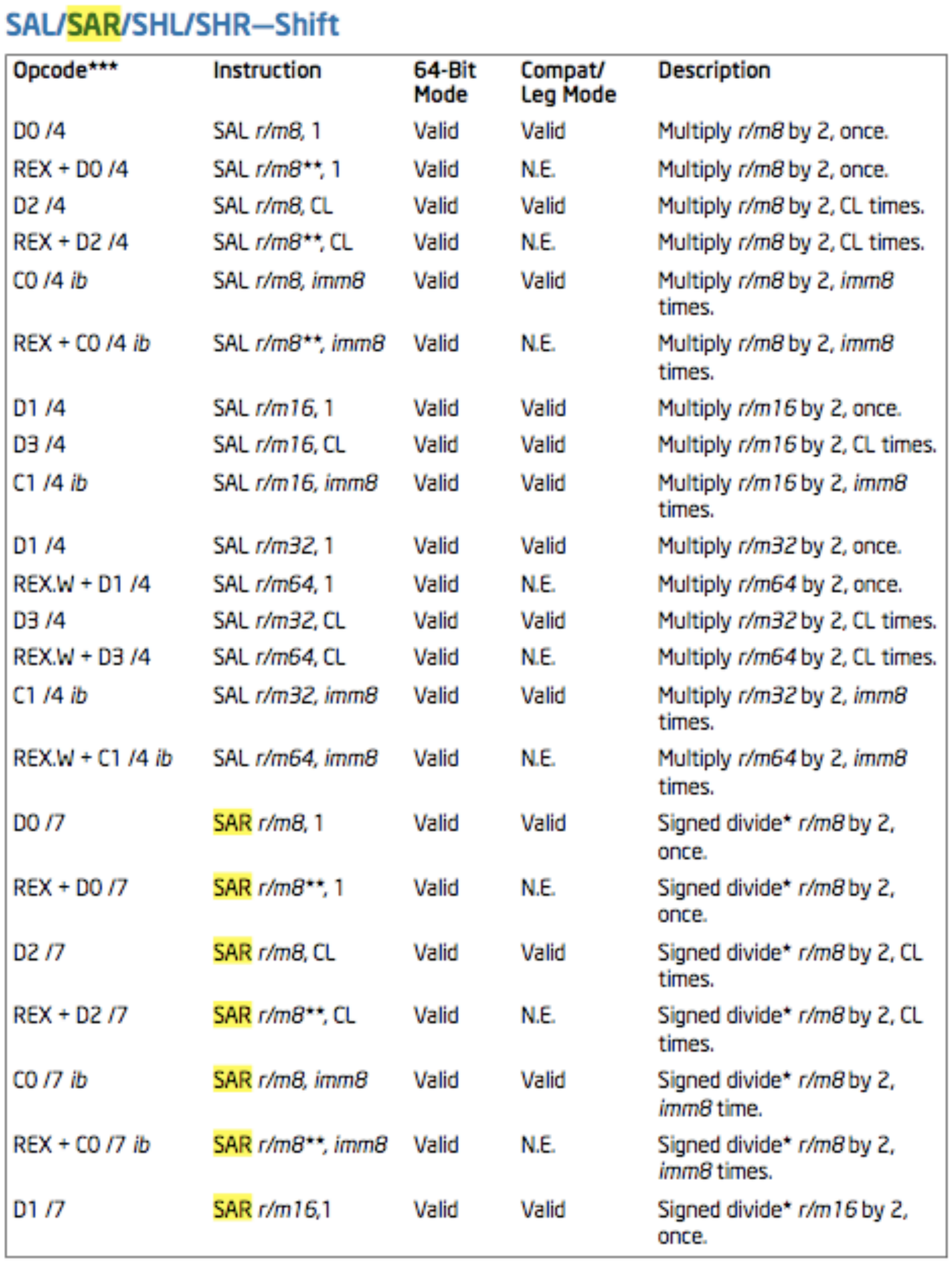
```
08048344 <main>: //Example6
 8048344:       lea    0x4(%esp),%ecx
 8048348:       and    $0xfffffff0,%esp
 804834b:       pushl  -0x4(%ecx)
 804834e:       push   %ebp
 804834f:       mov    %esp,%ebp
 8048351:       push   %ecx
 8048352:       sub    $0x10,%esp
 8048355:       movl   $0x40,-0x8(%ebp)
 804835c:       mov    -0x8(%ebp),%eax
 804835f:       shl    $0x3,%eax
 8048362:       mov    %eax,-0xc(%ebp)
 8048365:       mov    -0xc(%ebp),%eax
 8048368:       shr    $0x4,%eax
 804836b:       mov    %eax,-0x10(%ebp)
 804836e:       mov    -0x10(%ebp),%eax
 8048371:       add    $0x10,%esp
 8048374:       pop    %ecx
 8048375:       pop    %ebp
 8048376:       lea    -0x4(%ecx),%esp
 8048379:       ret
```

changed

```
08048344 <main>: //Example6-mod
 8048344:       lea    0x4(%esp),%ecx
 8048348:       and    $0xfffffff0,%esp
 804834b:       pushl  -0x4(%ecx)
 804834e:       push   %ebp
 804834f:       mov    %esp,%ebp
 8048351:       push   %ecx
 8048352:       sub    $0x10,%esp
 8048355:       movl   $0x40,-0x8(%ebp)
 804835c:       mov    -0x8(%ebp),%eax
 804835f:       shl    $0x3,%eax
 8048362:       mov    %eax,-0xc(%ebp)
 8048365:       mov    -0xc(%ebp),%edx
 8048368:       mov    %edx,%eax
 804836a:       sar    $0x1f,%eax
 804836d:       shr    $0x1c,%eax
 8048370:       add    %edx,%eax
 8048372:       sar    $0x4,%eax
 8048375:       mov    %eax,-0x10(%ebp)
 8048378:       mov    -0x10(%ebp),%eax
 804837b:       add    $0x10,%esp
 804837e:       pop    %ecx
 804837f:       pop    %ebp
 8048380:       lea    -0x4(%ecx),%esp
 8048383:       ret
```

Compiled and disassembled on Linux

Why? Cause VS added an extra, distracting, instruction



SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL *r/m8*, 1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SAL *r/m8***, 1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SAL *r/m8*, CL	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SAL *r/m8***, CL	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8*, *imm8*	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SAL *r/m8***, *imm8*	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*, 1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SAL *r/m16*, CL	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16*, *imm8*	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*, 1	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SAL *r/m64*, 1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SAL *r/m32*, CL	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SAL *r/m64*, CL	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SAL *r/m32*, *imm8*	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SAL *r/m64*, *imm8*	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /7	SAR *r/m8*, 1	Valid	Valid	Signed divide* *r/m8* by 2, once.
REX + D0 /7	SAR *r/m8***, 1	Valid	N.E.	Signed divide* *r/m8* by 2, once.
D2 /7	SAR *r/m8*, CL	Valid	Valid	Signed divide* *r/m8* by 2, CL times.
REX + D2 /7	SAR *r/m8***, CL	Valid	N.E.	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8*, *imm8*	Valid	Valid	Signed divide* *r/m8* by 2, *imm8* time.
REX + C0 /7 *ib*	SAR *r/m8***, *imm8*	Valid	N.E.	Signed divide* *r/m8* by 2, *imm8* times.
D1 /7	SAR *r/m16*,1	Valid	Valid	Signed divide* *r/m16* by 2, once.

Book p. 225

SAR - Shift Arithmetic Right

- Can be explicitly used with the C “>>” operator, if the operands are signed
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It divides the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Each bit shifted off the right side is place in CF.

sar ecx, 2

	10110011b (ecx - 0xB3)
result	11101100b (ecx - 0xEC)

!=

shr ecx, 2

	10110011b (ecx - 0xB3)
result	00101100b (ecx - 0x2C)

sar ecx, 2

	00110011b (ecx - 0x33)
result	00001100b (ecx - 0x0C)

==

shr ecx, 2

	00110011b (ecx - 0x33)
result	00001100b (ecx - 0x0C)

Discussion

```
; semi-colons are comments
mov    -0xc(%ebp),%edx    ; edx == “b”
mov    %edx,%eax          ; eax == edx
sar    $0x1f,%eax         ; If the most significant bit of %eax was 1
                          ; when this happened, %eax == 0xFFFFFFFF,
                          ; else %eax == 0
shr    $0x1c,%eax         ; if %eax was 0, it’s still 0, else if %eax
                          ; was 0xFFFFFFFF the least significant four
                          ; bits of %eax are set (i.e. 0xF)
add    %edx,%eax          ; Add 0xF or 0 to the value to be shifted
sar    $0x4,%eax          ; Now perform the expected shift
```

- But why add something to the least signficant bits when it's just going to get shifted away?
- It turns out the 0xF (four ones) is only because it's a 4 bit shift. And everything which gets shifted off the right side gets shifted into the Carry Flag (CF). Thus it's guaranteeing that when the sequence of operations is done, that CF == 1 if and only if the original number was signed (MSB == 1)7.
- If we change the C code to b / 32, and hence a 5 bit shift, the `shr $0x1c,%eax` turns into `shr $0x1b,%eax`, and the `sar $0x4,%eax` turns to `sar $0x5,%eax`
- If you analyze Example6-mod.c with VisualStudio, it does the same thing, but it uses different instructions to do it.

Discussion: variable-length opcodes

- Any given sequence of bytes can be interpreted in different ways, depending on where the CPU starts executing it from
- This has many subtle implications, but it seems to get abused the most in the security domain
- Examples: inability to validate intended instructions, return-oriented-programming, code obfuscation and polymorphic/self-modifying code
- In comparison, RISC architectures typically have fixed instruction sizes, which must be on aligned boundaries, and thus makes disassembly much simpler

Variable-length opcode decoding example

```
(gdb) x/5i $eip
0x8048385 <main+17>:    movl   $0x8048460,(%esp)
0x804838c <main+24>:    call   0x80482d4 <puts@plt>
0x8048391 <main+29>:    mov    $0x1234,%eax
0x8048396 <main+34>:    add    $0x4,%esp
0x8048399 <main+37>:    pop    %ecx
(gdb) x/5i $eip+1
0x8048386 <main+18>:    add    $0x24,%al
0x8048388 <main+20>:    pusha
0x8048389 <main+21>:    test   %al,(%eax,%ecx,1)
0x804838c <main+24>:    call   0x80482d4 <puts@plt>
0x8048391 <main+29>:    mov    $0x1234,%eax
(gdb) x/5i $eip+2
0x8048387 <main+19>:    and    $0x60,%al
0x8048389 <main+21>:    test   %al,(%eax,%ecx,1)
0x804838c <main+24>:    call   0x80482d4 <puts@plt>
0x8048391 <main+29>:    mov    $0x1234,%eax
0x8048396 <main+34>:    add    $0x4,%esp
```

```
(gdb) x/5i $eip+6
0x804838b <main+23>:    or     %ch,%al
0x804838d <main+25>:    inc    %ebx
0x804838e <main+26>:    (bad)
0x804838f <main+27>:    (bad)
0x8048390 <main+28>:    (bad)
```

```
(gdb) x/xb 0x804838e
0x804838e <main+26>:    0xff
(no instruction starts with 0xFF)
```

Questions about anything in the class? Stuff you’d like me to go over again?

Choose your own adventure

- Effects of compiler optimization/security/ debugging options? Goto p35
- Dissect the binary bomb? Goto p39
- Messing with a disassembler? Goto p41
- Mystery box! Goto p52
- Why twos compliment? Goto p

Effects of Compiler Options

Our standard build

```
//Example8.c
int main(){
        char buf[40];
        buf[39] = 42;
        return 0xb100d;
}
```

```
main:
00401010  push       ebp
00401011  mov        ebp,esp
00401013  sub        esp,28h
00401016  mov        byte ptr [ebp-1],2Ah
0040101A  mov        eax,0B100Dh
0040101F  mov        esp,ebp
00401021  pop        ebp
00401022  ret
```

Effects of Compiler Options 2

/O1 (minimum size) or
/O2 (maximum speed)

```
main:
0040100F  mov        eax,0B100Dh
00401014  ret
```

Debug information format
Disabled (viewed from WinDbg) or
/Z7 (C7 Compatible)
(no change)

```
main:
00401010push    ebp
00401011mov     ebp,esp
00401013 sub    esp,28h
00401016 mov     byte ptr [ebp-1],2Ah
0040101amov     eax,0B100Dh
0040101f mov    esp,ebp
00401021pop    ebp
00401022ret
```

/Zi -> /ZI (Program database for edit & continue)

```
main:
00411250  push       ebp
00411251  mov        ebp,esp
00411253  sub        esp,68h
00411256  push       ebx
00411257  push       esi
00411258  push       edi
00411259  mov        byte ptr [ebp-1],2Ah
0041125D  mov        eax,0B100Dh
00411262  pop        edi
00411263  pop        esi
00411264  pop        ebx
00411265  mov        esp,ebp
00411267  pop        ebp
00411268  ret
```

Effects of Compiler Options 3

/GS - Buffer Security Check (default enabled nowadays)
aka “stack cookies” (MS term)
aka “stack canaries” (original research term)

```
main:
00401010  push      ebp
00401011  mov       ebp,esp
00401013  sub       esp,2Ch
00401016  mov       eax,dword ptr [___security_cookie (405000h)]
0040101B  xor       eax,ebp
0040101D  mov       dword ptr [ebp-4],eax
00401020  mov       byte ptr [ebp-5],2Ah
00401024  mov       eax,0B100Dh
00401029  mov       ecx,dword ptr [ebp-4]
0040102C  xor       ecx,ebp
0040102E  call      __security_check_cookie (401040h)
00401033  mov       esp,ebp
00401035  pop       ebp
00401036  ret
```

Effects of source options

/O1 optimization when the volatile keyword is present

```
int main(){
        volatile char buf[40];
        buf[39] = 42;
        return 0xb100d;
}
```

```
main:
00401010  sub     esp,28h
00401013  mov     byte ptr [esp+27h],2Ah
00401018  mov     eax,0B100Dh
0040101D  add     esp,28h
00401020  ret
```

This is a trick I picked up from a 2009 Defcon presentation
http://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf
He also talked a little bit about control flow flattening which is covered in an academic paper in the "Messing with the disassembler" adventure. Goto page 41.

Bomb lab

- From CMU architecture class - http://csapp.cs.cmu.edu/public/labs.html
- Goal is to reverse engineer multiple phases to determine the program's desired input
- Create a text file with answers, one per line, named "answers"
- gdb -x myCmds bomb
- run with "r < answers"
- Should add/remove breakpoints on the different phases as you go along

GDB/Bomb Lab Cheat Sheet

- Christian Arllen found this, and it has many more example of gdb syntax, as well as some help for if you get stuck on the lab

- http://condor.depaul.edu/~jriely/csc373fall2010/extras/mygdbnotes.txt
- (get it on google cache while you can, because it's gone now)

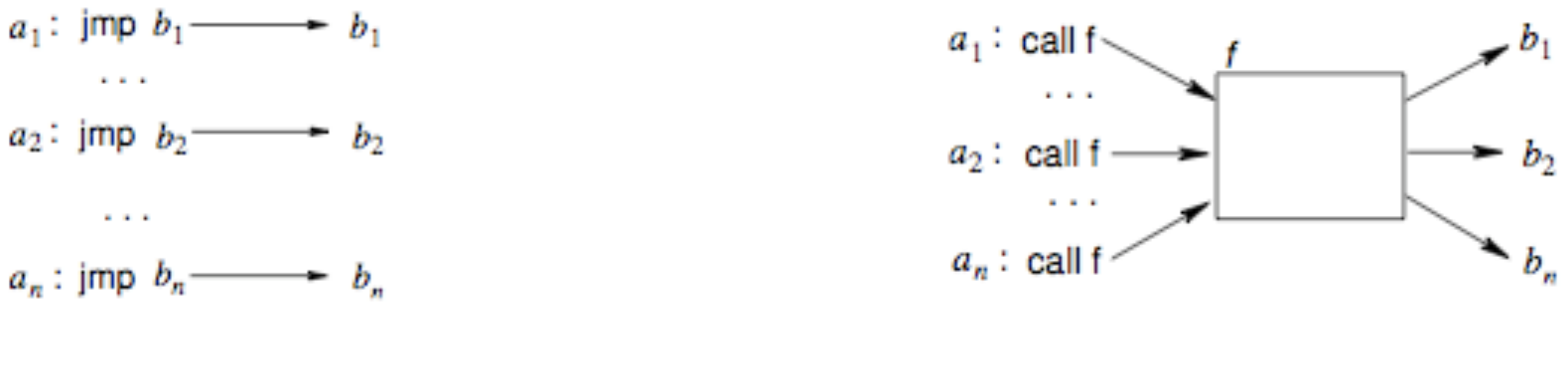
Messing with a disassembler

- Obfuscation of Executable Code to Improve Resistance to Static Disassembly - Linn & Debray
  - http://www.cs.arizona.edu/solar/papers/CCS2003.pdf
  - Linear sweep vs. recursive traversal disassembly
  - Also discusses and measures the "self-repairing" nature of x86 disassembly which we saw earlier
- Confusing linear sweep (objdump) by inserting junk bytes after unconditional jumps.
  - Could be literally unconditional "jmp"
  - Could be a jcc, which must always be true, like "xor eax, eax" and then "jz <addr>"
  - Have to do this multiple times because of the self-repairing disassembly

Messing with disassembler 2

- Confusing recursive traversal
  - 3.4.1: Branch functions. All jmps turned into a call to a specific function.
  - 3.4.2: Call conversion. Branch functions + the junk byte technique which messed with linear sweep.
  - 3.4.3: Opaque predicates. Create ostensibly conditional jumps which will in fact always follow only one path. The disassembler doesn’t have the smarts to determine this.
  - 3.4.5: Jump table spoofing. Exploits the fact that the disassembler may try to estimate the size of the jump table based on a constraint. The trick is to add a jump table which will never be reached.

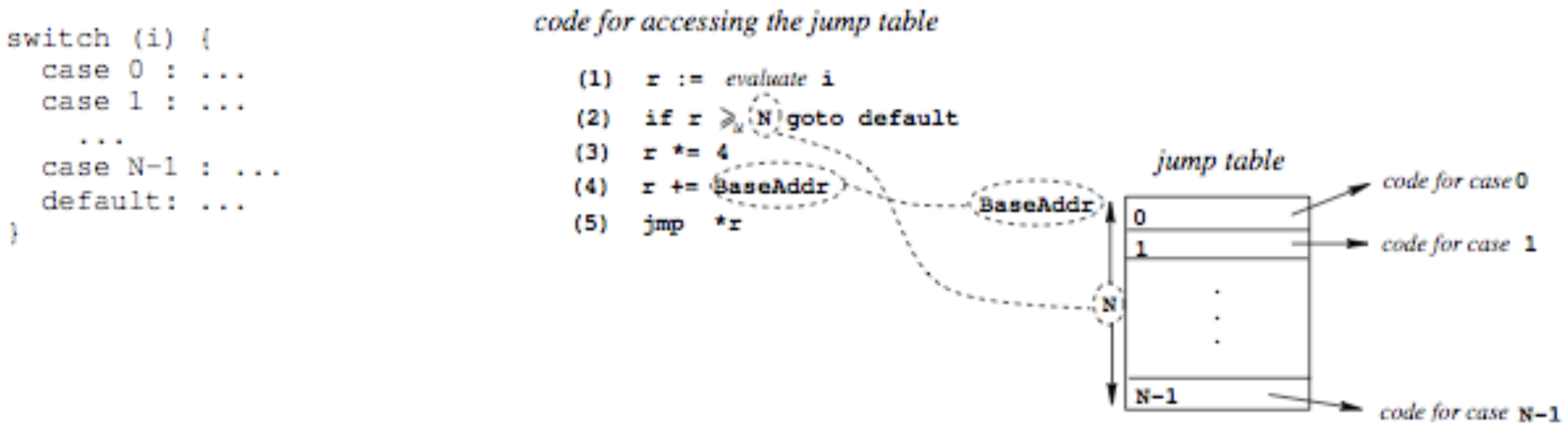
Branch Functions Visualized



(a) Original code (b) Code using a branch function

Figure 5: Branch functions

Jump table visualized



```
switch (i) {
  case 0 : ...
  case 1 : ...
    ...
  case N-1 : ...
  default: ...
}
```

code for accessing the jump table

```
(1)  r := evaluate i
(2)  if r $\geqslant_u$ N goto default
(3)  r *= 4
(4)  r += BaseAddr
(5)  jmp  *r
```

(a) Source code

(b) Implementation using a jump table

Figure 3: A example of a C `switch` statement and its implementation using a jump table

Addressing Linn & Debray obfuscations

- Static Disassembly of Obfuscated Binaries - Kruegel *et al.*
  - http://www.cs.ucsb.edu/~chris/research/doc/usenix04_disasm.pdf
  - Attempt to improve on the state of the art in disassembling, to deal with the Linn & Debray obfuscations
  - I don't know if there are any disassemblers which try to use these improved disassembly methods (objdump and IDA definitely don't). Confirmed with Kruegel that he's not aware of anywhere that uses the improvements either.

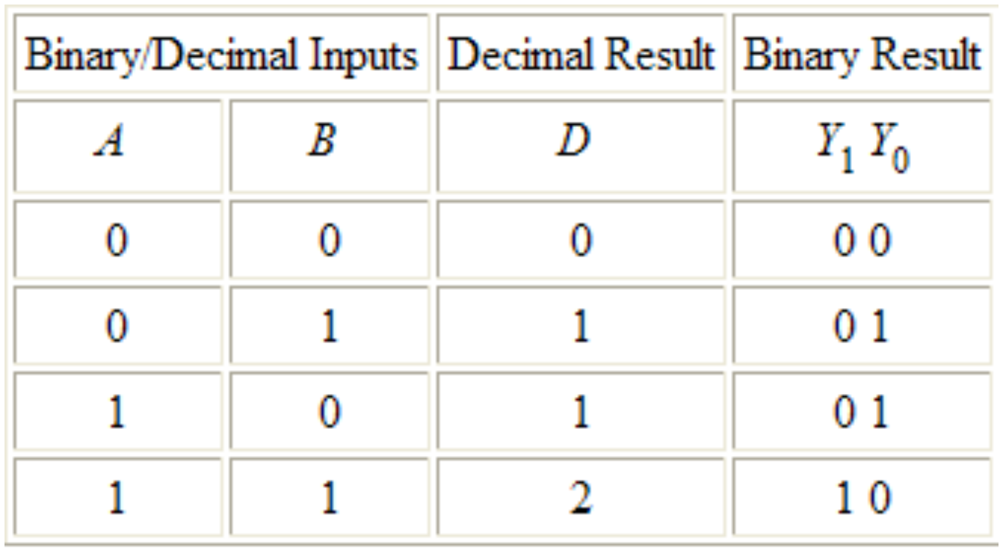
Digression – Why Two's Compliment?

- Alternative methods of representing negative numbers (signed magnitude, or just ones compliment), as well as their problems presented on page 166-167 of the book.
  - Note to self: show on board quick
- The benefit of two's compliment is due to having only one representation of zero, and being able to reuse the same hardware for addition/subtraction
- Dave Keppler suggested expanding on this

Why Two’s Compliment? 2

Carry

$$\begin{array}{rr}
1 & \\
 & 5d \\
+ & 6d \\
\hline
 & 11d
\end{array}
\qquad\qquad
\begin{array}{rr}
1 & \\
 & 1b \\
+ & 1b \\
\hline
 & 10b
\end{array}$$

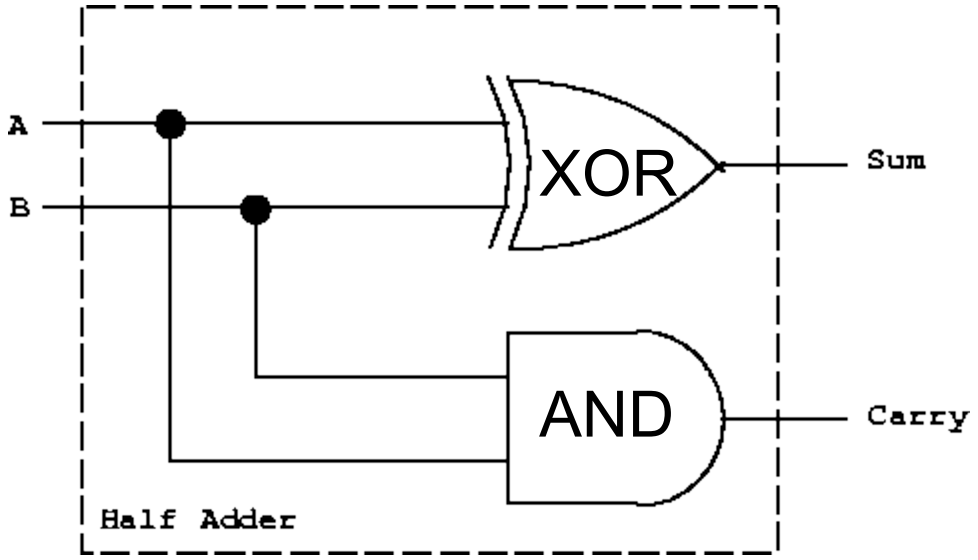


Binary/Decimal Inputs		Decimal Result	Binary Result
A	B	D	$Y_1\ Y_0$
0	0	0	0 0
0	1	1	0 1
1	0	1	0 1
1	1	2	1 0

Table taken from
http://thalia.spec.gmu.edu/~pparis/classes/notes_101/node110.html

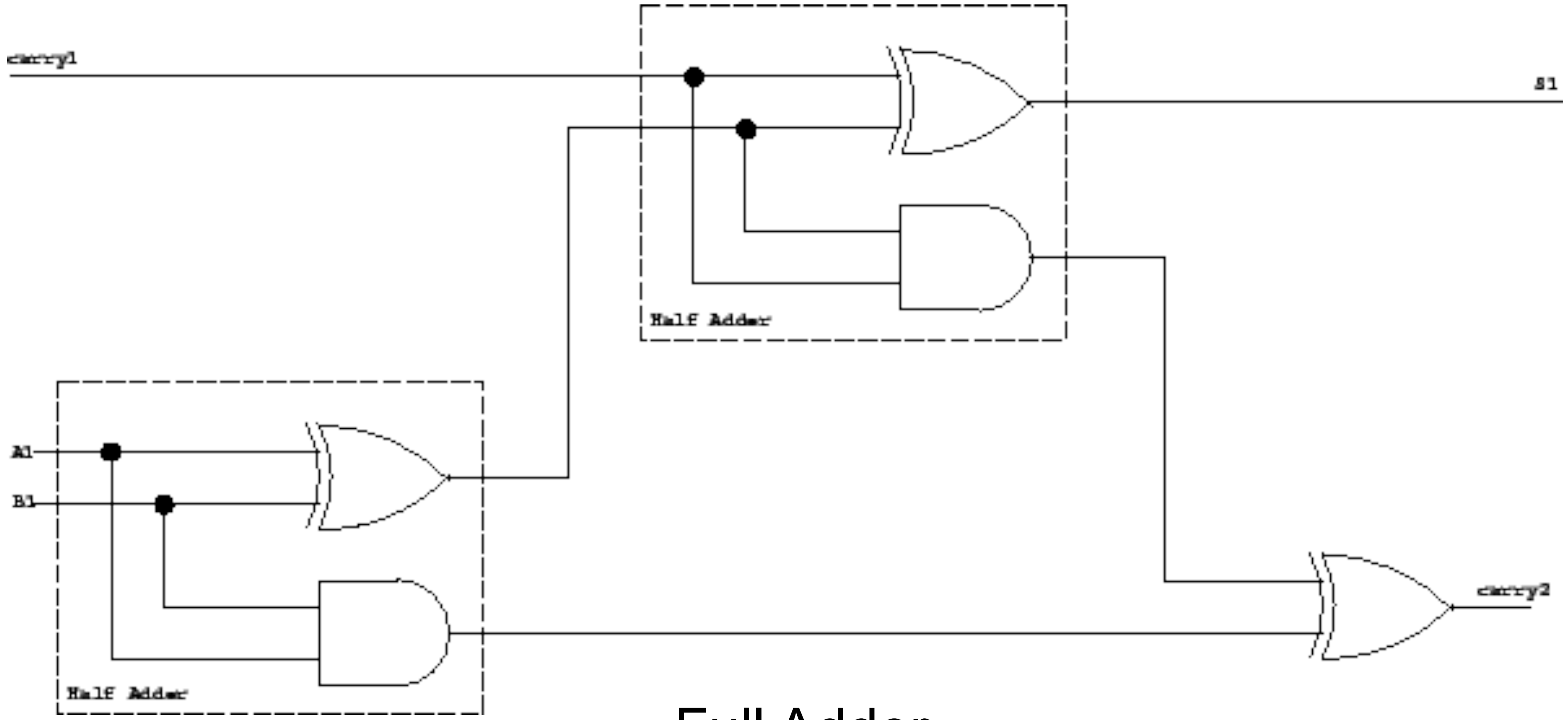
Why Two’s Compliment? 3

A half adder circuit suffices for one bit addition



Picture taken from
http://thalia.spec.gmu.edu/~pparis/classes/notes_101/node110.html

Why Two's Compliment? 4



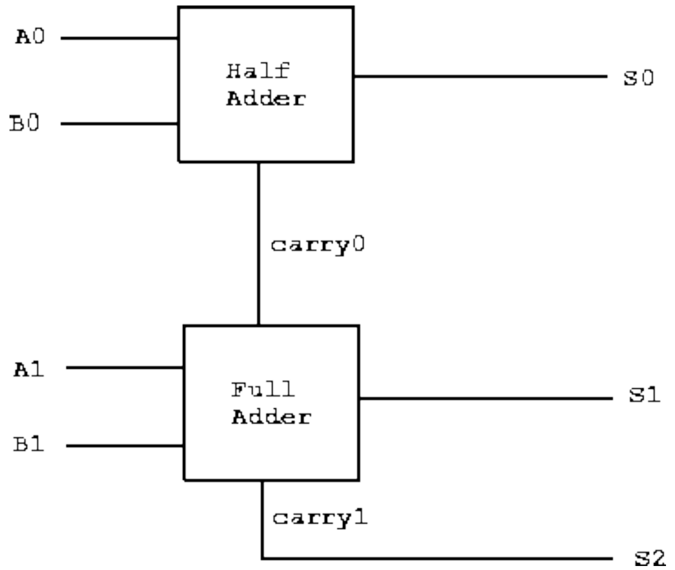
Full Adder

You can't just chain the one bit half adders together to get multi-bit adders. To see why, see the truth table at the link.

Picture taken from
http://thalia.spec.gmu.edu/~pparis/classes/notes_101/node111.html

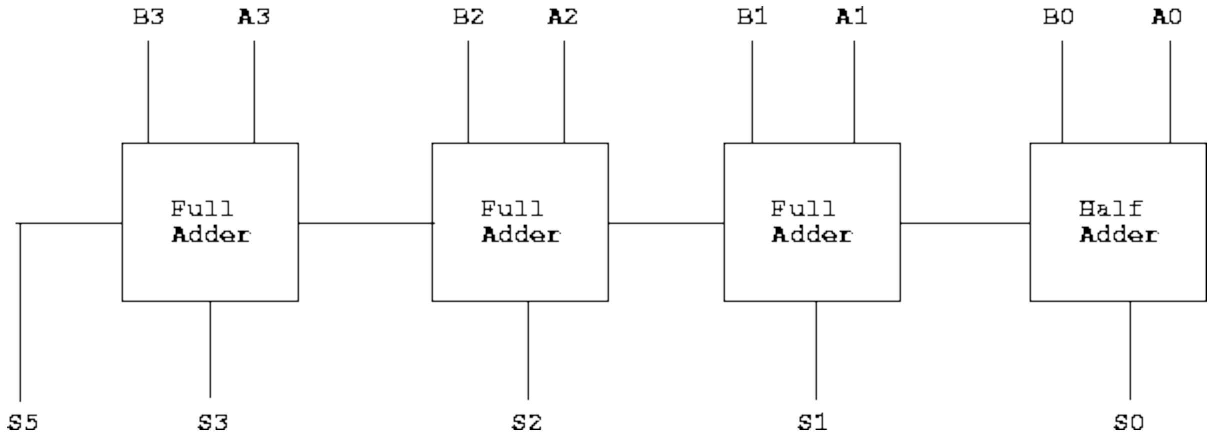
Why Two's Compliment? 5

2 bit adder



Note: we start with a half adder because a full adder would need a carry input at the start. However, if we wanted to use this for subtraction we could use a full adder to start. More on this on next slide.

4 bit adder
(continue to make n bit adder)



Pictures taken from
http://thalia.spec.gmu.edu/~pparis/classes/notes_101/node112.html
http://thalia.spec.gmu.edu/~pparis/classes/notes_101/node113.html

Why Two’s Compliment? 6

- So you have these physical adder circuits in the Arithmetic Logic Unit (ALU), and you can feed both add and subtract to the same circuit. But for this to work, you need to start with a full adder, and then run one the one subtract operand bits through not gates, and then set carry to one on the first full adder.
- Keppler’s example of x-y == x+(-y)
  - Cause it was right there in my email and I’m lazy ;)

```
  00001010   00001010 (10d) ==  00001010 (10d)
+ 00000101  -00000101 (5d)     +11111011 (-5d)
----------  ---------          ---------
  00001111   00000101         1 00000101
```

What’s in the mystery box!?

?

Wrap up - instructions

- Learned around 26 instructions and variations
- About half are just math or logic operations
- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT
- SHR/SHL/SAR/SAL
- IMUL/DIV
- REP STOS, REP MOV
- LEAVE

Wrap up

- Learned about the basic hardware registers and how they're used
- Learned about how the stack is used
- Saw how C code translates to assembly
- Learned basic usage of compilers, disassemblers, and debuggers so that assembly can easily be explored
- Learned about Intel vs AT&T asm syntax
- Learned how to RTFM

The shape of things to come

- How does a system map a limited amount of physical memory to a seemingly unlimited amount of virtual memory?
- How does debugging actually work? How can malware detect your debugger and alter its behavior?
- How is “user space” actually separated from “kernel space”? I’ve heard there’s “rings”, but where are these fabled rings actually at?
- What if I want to talk to hardware beyond the CPU?

That's all Folks!