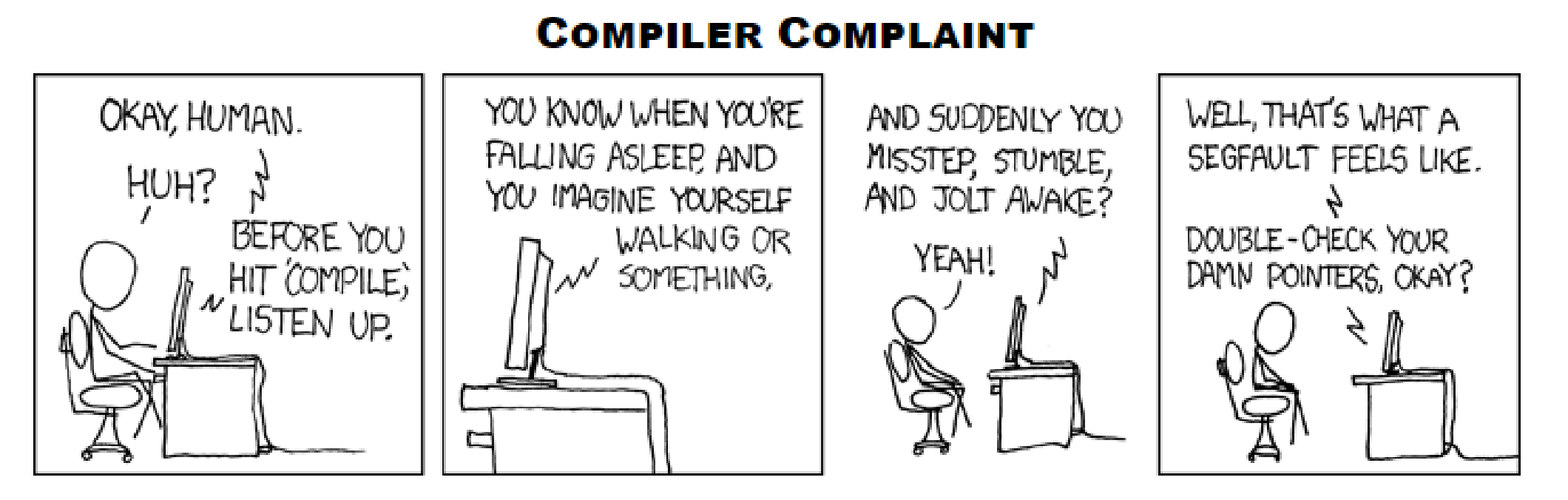
CHAPTER 7 – PART I
X86-64 INSTRUCTION SET OVERVIEW



CPSC 240-09
John Overton

More Notes…

- OK, I said that most quizzes, the mid term and final will be from my slides.

  Well, not exactly true! READ THE TEXT BOOK!

  (Please?)

- Midterm will be Thursday, March 28th, 2019
- Pop quiz next Tuesday, February 5, 2019. Questions will be from everything we've talked about from the first two weeks.
- In many of the slides I talk about 64-bit mode. This is the normal mode of operation for 64-bit operating systems, like Tuffix (we can use the 64-bit registers!) And, as I have said before, all of our exercises, experiments, and future homework/project assignments will use 64-bit mode. I have also called this long mode in previous slides and talks.

BUT FIRST, SOME THOUGHTS ABOUT A PREVIOUS TOPIC…

Questions…

- Little endian: In a program, if we were using a dd pseudo command to define a double word data variable, and in our program the initial number value was 156285_{10}, which is $0002627D_{16}$, what would be in memory (in hex)?

Review: Program Format – A line of code

- Generally, a source line is of the format:

```
label:      instruction  operands    ; comment
```

- For **instruction,** we could have any of these:
  - Actual instructions (these symbolically represent x86 machine code instructions),
  - Pseudo instructions (they tell the assembler something, but don't actually generate machine code)
  - Directives (similar to pseudo instructions)
  - Macro name

- A label is optional or could be on a line by itself
- An instruction may not have any operands

Program Format

- ; used to indicate the start of comments for the rest of the line of code.

- To specify a number in hex, precede the hex number with an 0x like 0x7f (127 in decimal). Octal numbers would be followed by a q like 377q (equivalent of 127 in decimal).

- Define constants with the “equ” pseudo instruction (it’s like #define in C/C++)

Program Format

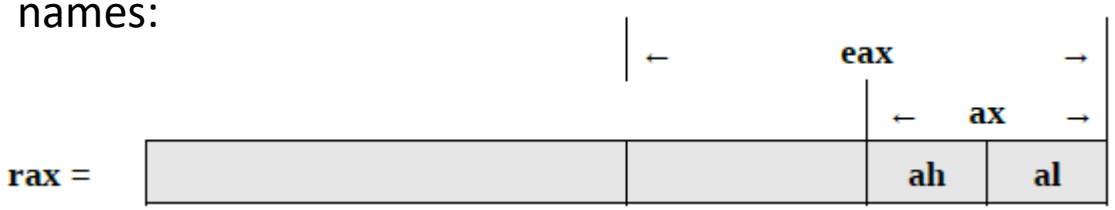
- Define data in your program with db and friends (db, dw, dd, dq)
- Use “sections” to define sections (what is the difference between the three types of sections?):
  - .text for code sections
  - .data for data sections
  - .bss for uninitialized data
- Reserve space in uninitialized data sections with resb, resw, resd or resq:

```
Var1:     resb   8   ;Var1 is 8 bytes in size and uninitialized (resq?)
```

Review of register names

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

For example, here is the RAX register and its other names:



Instructions we will learn in chapter 7:

- mov
- lea
- Narrowing/widening conversions with mov
- movzx (unsigned conversion)
- Signed conversions: movsx, movsxd, cbw, cwd, cwde, cdq, cdqe, cqo
- xlat (not in the book, but very, very helpful to know)
- Integer arithmetic instructions:
  add, inc, adc, sub, dec, mul, imul, div, idiv
- Logical operations: and, or, xor
- Shift operations: shl, shr, sal, sar, rol, ror
- Control instructions: jmp, cmp, je, jne, jl, jle, jg, jge, jb, jbe, ja, jae,
- Iteration: loop

General Notation for all instructions…

Operand Notation	Description
<reg>	Register operand. The operand must be a register.
<reg8>, <reg16>, <reg32>, <reg64>	Register operand with specific size requirement. For example, **reg8** means a byte sized register (e.g., **al**, **bl**, etc.) only and **reg32** means a double-word sized register (e.g., **eax**, **ebx**, etc.) only.
<dest>	Destination operand. The operand may be a register or memory. Since it is a destination operand, the contents will be overwritten with the new result (based on the specific instruction).
<src>	Source operand. Operand value is unchanged after the instruction.
<imm>	Immediate value. May be specified in decimal, hex, octal, or binary.
<mem>	Memory location. May be a variable name or an indirect reference (i.e., a memory address).
<op> or <operand>	Operand, register or memory.
<op8>, <op16>, <op32>, <op64>	Operand, register or memory, with specific size requirement. For example, **op8** means a byte sized operand only and **reg32** means a double-word sized operand only.
<label>	Program label.

mov instruction…

- Typically, data must be moved into a CPU register from RAM in order to be operated upon.
- Once the data in the register has been operated upon, then you can use another mov instruction to save the data back in memory.

- The general form of the move instruction is:

```
mov     <dest>, <src>
```

- In 64-bit mode, there are also 32-bit, 16-bit and 8-bit register names that are just some portion of their 64-bit registers.
- If the destination <dest> is a 8-bit or 16-bit register name, then the mov will leave the “high-order” portion of the 64-bit register alone (whatever was previously in the register will be left in that portion)
- In 64-bit mode, if the destination is a 32-bit register, then the high-order 32 bits will be set to zero.

mov instruction…

- Nasm gives us a way to describe how much to move, by using byte, word, dword or qword before the variable name. But, many times, Nasm will figure it out by looking at the variable size

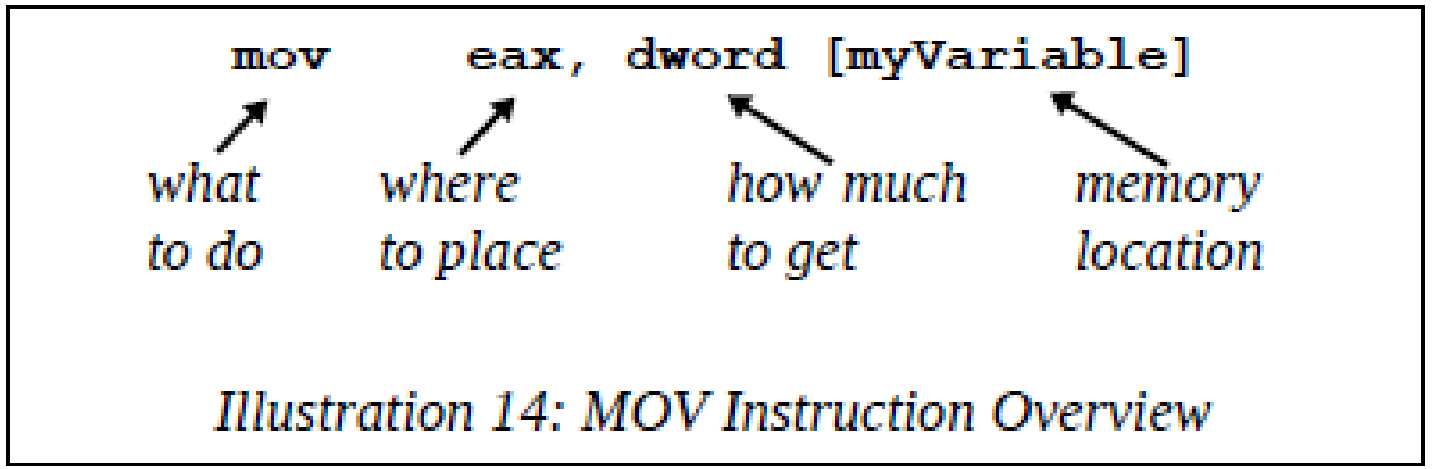


Illustration 14: MOV Instruction Overview

- The term “immediate” refers to some number actually in the instruction

```
mov    dword  [dVar], 27        ; move the number 27 to the variable dVar which is in memory
```

Since both <src> and <dst> can not both be variables in memory, if you want to move data from one variable to another variable, you need to do it in 2 instructions:

```
mov   eax, dword [dvar1]
mov   dword [dvar2], eax
```

mov instruction…

- For some instructions, the explicit type specification (that is, bute, word, dword, qword) can be omitted as the other operand will clearly define the size. Good programming practice is to include the type specification
- If we want to access memory variables, we need to surround the variable name with brackets ([]'s). Brackets around a variable name tell nasm that we want to take the data at that variable's address (after all, variables are just a label that refer to a memory address.)

lea – Load effective address

- You can load the address of a variable by using the lea instruction. (this is a pointer, right?)

```
lea     rax, [VarName]
```

- You can then use the address in a register to refer to that variable:

```
mov     eax, dword [rax]
```

- Does the <dest> register need to be a 64-bit register to hold an address?

Instruction	**Explanation**
`lea <reg64>, <mem>`	Place address of <**mem**> into **reg64**.
Examples:	`lea rcx, byte [bvar]` `lea rsi, dword [dVar]`

Narrowing Conversions…

- This has to do with converting (downsizing) a value from, say a quad-word to a double word, or a double word to a word or a word to a byte.
- By using the register names that are a portion of the 64-bit register, you can take just the lower portion of the value in that 64-bit register. For example, if dVal is a double word and has 50 in it:

```
mov    rax, dword [dVal]
mov   byte [bVal], al
```

Widening Conversions - unsigned…

- There are two different cases: unsigned and signed.
- For an unsigned widening conversion, if we are using a similar technique using registers, then we may need to make sure the higher-order bits are zero'd out (they would never be ones because it would always be positive)
- For example, to convert from byte to double word, you could do this:

```
mov     rbx,  0                 ; clear all of rbx to zero
mov     bl, byte [bVar]         ; load byte value that you want to widen
mov     dword [dVar], rbx
```

- You can also use movzx to mov data to a register

Instruction	Explanation

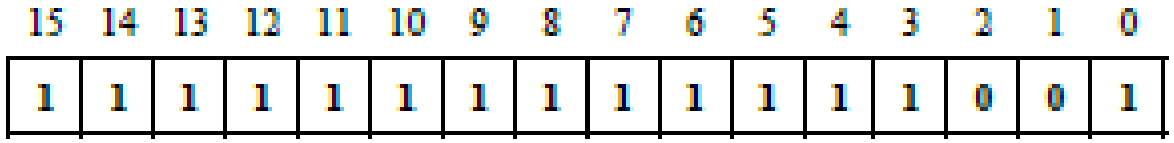
`movzx <dest>, <src>`
 	
`movzx <reg16>, <op8>`
`movzx <reg32>, <op8>`
`movzx <reg32>, <op16>`
`movzx <reg64>, <op8>`
`movzx <reg64>, <op16>`
	Unsigned widening conversion.
	Note 1, both operands cannot be memory.
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.

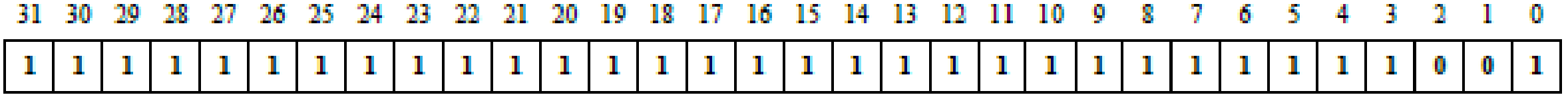
Examples:	`movzx cx, byte [bVar]`
	`movzx dx, al`
	`movzx ebx, word [wVar]`
	`movzx ebx, cx`
	`movzx rbx, cl`
	`movzx rbx, cx`

Widening Conversions - signed…

- For widening signed variables, we need to extend the sign. Why?
- Generally, we take the sign position from the value that we are going to extend.
  - If it is zero (positive number), then all of the bits in the upper bit positions of the destination need to be set to zero as well.
  - If it is a one (negative number), then all of the bits in the upper bit positions of the destination need to be set to a one as well.

So, for example, if we had a word that contained -7 (hexadecimal 0xFFF9), in the destination field, the high-order bits (positions 16 to 31) will need to be set to ones (this is called extending the sign)





Widening Conversions - signed…

- Fortunately, we have some instructions to do that. We actually have old instructions and new instructions. The old instructions used “hard-coded” operands.

Here are the new instructions:

Instruction	Explanation

`movsx  <dest>, <src>`
 	
`movsx  <reg16>, <op8>`
`movsx  <reg32>, <op8>`
`movsx  <reg32>, <op16>`
`movsx  <reg64>, <op8>`
`movsx  <reg64>, <op16>`
`movsxd  <reg64>, <op32>`
	Signed widening conversion (via sign extension).
	Note 1, both operands cannot be memory.
	Note 2, destination operands cannot be an immediate.
	Note 3, immediate values not allowed.
	Note 4, special instruction (*movsxd*) required for 32-bit to 64-bit signed extension.

Examples:	`movsx  cx, byte [bVar]`
	`movsx  dx, al`
	`movsx  ebx, word [wVar]`
	`movsx  ebx, cx`
	`movsxd  rbx, dword [dVar]`

Widening Conversions - signed…

Here are the old instructions (actually, cdqe and cqo are new and use 64-bit registers). These all use 'hard-wired' registers

Instruction	Explanation
cbw	Convert byte in **al** into word in **ax**. *Note*, only works for **al** to **ax** register.
Examples:	cbw
cwd	Convert word in **ax** into double-word in **dx:ax**. *Note*, only works for **ax** to **dx:ax** registers.
Examples:	cwd
cwde	Convert word in **ax** into double-word in **eax**. *Note*, only works for **ax** to **eax** register.
Examples:	cwde
cdq	Convert double-word in **eax** into quadword in **edx:eax**. *Note*, only works for **eax** to **edx:eax** registers.
Examples:	cdq
cdqe	Convert double-word in **eax** into quadword in **rax**. *Note*, only works for **rax** register.
Examples:	cdqe
cqo	Convert quadword in **rax** into word in double-quadword in **rdx:rax**. *Note*, only works for **rax** to **rdx:rax** registers.
Examples:	cqo

Building up your library

- We stand on the shoulders of engineers who have come before us (and share their libraries with us)
- In our homework assignments, we will want to build our own libraries
- However, for some of our assignments, we may be able to use libraries that are given to us (by our teacher, or later on, we will use the C standard library)

The translate instruction: xlat

- Can be used to find a character in a table, given an offset into the table
- It uses ‘hard-coded’ registers:
  - rbx must point to the table
  - al has the offset into the table
  - The result will be placed from the table into al
- An example of this might be to translate a hex number into its asci representation
- This is where the lea instruction comes in handy, to load rbx with the address of the table (point to the table)

```
xtable:   db    “0123456789ABCDEF”

          lea   rbx, [xtable]
          xlat
```

Today’s experiment:

- Write some code to translate a hex number into its asci equivalent and write it to the console.
- Then, write out a carriage return to the console (so the previous write to the console is not so messy on the screen)
- Use our hello world program as a guide line for writing out things to the console (hello.asm)