

Basic Assembly Language

ICS312 - Spring 2009 Machine-Level and Systems Programming

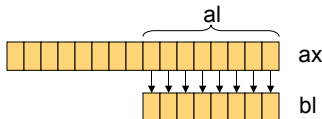
Henri Casanova (henric@hawaii.edu)

Size of Data

- Remember that labels merely declare an address in the data segment, and do not specify any size
- Size of data is inferred based on the source or destination register
 - `mov eax, [L]` ; loads 32 bits
 - `mov al, [L]` ; loads 8 bits
 - `mov [L], eax` ; stores 32 bits
 - `mov [L], ax` ; stores 16 bits
- This is why it's really important to know the names of the x86 registers

Size Reduction

- Sometimes one needs to decrease the data size
- For instance, you have a 4-byte integer, but you need to use it as a 2-byte integer for something
- We simply use the registers: when moving a quantity from an X-bit register to a Y-bit register ($Y < X$), the highest (X-Y) bits are simply removed
- Example:
 - `mov ax, [L]` ; loads 16 bits in ax
 - `mov bl, ax` ; takes the lower 8 bits of ax and puts them in bl
 - Equivalent to "mov bl, al"



Size Reduction

- Of course, when doing a size reduction, one loses information
- So the "conversion" may not work
- Example:
 - `mov ax, 000A2h` ; ax = 162 decimal
 - `mov bl, ax;` ; bl = 162 decimal
 - Decimal 162 is *encodable* on 8 bits
- Example:
 - `mov ax, 00101h` ; ax = 257 decimal
 - `mov bl, ax;` ; bl = 1 decimal
 - Decimal 257 is *not encodable* on 8 bits

Size Reduction and Sign

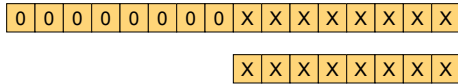
- Consider a 2-byte quantity: FFF4
- If we interpret this quantity as **unsigned** it is decimal 65,524
 - Remember that the computer does not know whether the content of registers/memory corresponds to signed or unsigned quantities
 - Once again it's the responsibility of the programmer to do the right thing
- In this case size reduction "does not work", meaning that reduction to a 1-byte quantity will not be interpreted as decimal 65,524, but instead as decimal 244 (F4h)
- If instead FFF4 is a **signed** quantity (using 2's complement), then it corresponds to -000C (000B + 1), that is to decimal -12
- In this case, size reduction works!

Size Reduction and Sign

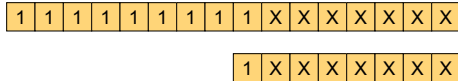
- This does not mean that size reduction always works for signed quantities
- For instance, consider FF32h, which is a negative number equal to -00CEh, that is, decimal -206
- A size reduction into a 1-byte quantity leads to 32h, which is decimal +50!
- Note that -206 is not encodable on 1 byte
 - The range of signed 1-byte quantities is between decimal -128 and decimal +127
- So, size reduction may work or not work for signed or unsigned quantities!

Two Rules to Remember

- For **unsigned numbers**: size reduction works if all removed bits are 0



- For **signed numbers**: size reduction works if all removed bits are all 0's or all removed bits are all 1's, AND if the highest bit not removed is equal to the removed bits
 - This highest remaining bit is the new sign bit, and thus must be the same as the original sign bit



Size Increase

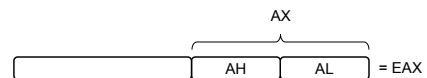
- Size increase for unsigned quantities is simple: just add 0s to the left of it
- Size increase for signed quantities requires sign extension: the **sign bit must be extended**, that is, replicated
- Consider the signed 1-byte number 5A. This is a positive number (decimal 90), and so its 2-byte version would be 005A
- Consider the signed 1-byte number 8A. This is a negative number (decimal -118), and so its 2-byte version would be FF8A

Unsigned size increase

- Say we want to size increase an unsigned 1-byte number to be a 2-byte unsigned number
- This can be done in a few easy steps, for instance:
 - Put the 1-byte number into al
 - Set all bits of ah to 0
 - Access the number as ax
- Example
 - mov al, 0EDh
 - mov ah, 0
 - move ..., ax

Unsigned size increase

- How about increasing the size of a 2-byte quantity to 4 byte?
- This cannot be done in the same manner because there is no way to access the 16 highest bit of register eax separately!



- Therefore, there is an instruction called **movzx** (Zero eXtend), which takes two operands:
 - Destination: 16- or 32-bit register
 - Source: 8- or 16-bit register, or 1 byte of memory, or a word of memory
 - The destination must be larger than the source!

Using movzx

- movzx eax, ax ; extends ax into eax
- movzx eax, al ; extends al into eax
- movzx ax, al ; extends al into eax
- movzx ebx, ax ; extends ax into ebx
- movzx ebx, [L] ; gives a "size not specified" error
- movzx ebx, **byte** [L] ; extends 1-byte value at address L into ebx
- movzx eax, **word** [L] ; extends 2-byte value at address L into eax

Signed Size Increase

- There is no way to use mov or movzx instructions to increase the size of signed numbers, because of the needed sign extension
- Four "old" conversion instructions with implicit operands
 - CBW (Convert Byte to Word): Sign extends AL into AX
 - DX contains high bits, AX contains low bits
 - CWD (Convert Word to Double): Sign extends AX into DX:AX
 - a left-over instruction from the time of the 8086 that had no 32-bit registers
 - CWDE (Convert Word to Double word Extended): Sign extends AX into EAX
 - CDQ (Convert Double word to Quad word): Signs extends EAX into EDX:EAX (implicit operands)
 - EDX contains high bits, EAX contains low bits
 - This is really a 64-bit quantity (and we have no 64-bit register)
- The **much more popular MOVZX instruction**
 - Works just like MOVZX, but does sign extension
 - CBW equiv. to MOVZX ax, al
 - CWDE equiv. to MOVZX eax, ax

Example

```

mov al 0A7h      ; as a programmer, I view this
                 ; as a unsigned, 1-byte quantity
                 ; (decimal 167)
mov bl 0A7h      ; as a programmer, I view this
                 ; as a signed 1-byte
                 ; quantity (decimal -89)

movzx eax, al;   ; extend to a 4-byte value
                 ; (000000A7)
movsx ebx, bl;   ; extend to a 4-byte value
                 ; (FFFFFFA7)

```

In-class Exercise

- Consider the following code


```

mov     al, 0B2h
movsx   eax, al
mov     bx, eax
movzx   ebx, bx

```
- What's the final value of eax?
- What's the final value of ebx?

In-class Exercise

		EAX	EBX
mov	al, 0B2h	?? ?? ?? B2	?? ?? ?? ??
movsx	eax, al	FF FF FF B2	?? ?? ?? ??
mov	bx, eax	FF FF FF B2	?? ?? FF B2
movzx	ebx, bx	FF FF FF B2	00 00 FF B2

Signed/Unsigned in C

- In the C (and C++) language one can declare variables as signed or unsigned
 - Motivation: if I know that a variable never needs to be negative, I can extend its range by declaring it unsigned
 - Often one doesn't do this, and in fact one often uses 4-byte values (int) when 1-byte values would suffice
 - e.g., for loop counters
- Let's look at a small C-code example

Signed/Unsigned in C

- Declarations:


```

unsigned char    uchar = 0xFF;
signed char     schar = 0xFF;
// "char" defaults to "signed char"

```
- I declared these variables as 1-byte numbers because I know I don't need to store large numbers
 - Often used to store ASCII codes, but can be used for anything


```

char x;
for (x=0; x<30; x++) { ... }

```
- Let's say now that I have to call a function that requires a 4-byte int as argument (by default "int" is "signed int")
- We need to extend 1-byte values to 4-byte values
- This is done in C with a "cast"


```

int a = (int) uchar; // the compiler will use MOVZX to do this
int b = (int) schar; // the compiler will use MOCSX to do this

```

Signed/Unsigned in C

```

unsigned char    uchar = 0xFF;
signed char     schar = 0xFF;
int              a = (int)uchar;
int              b = (int)schar;

printf("a = %d\n",a);
printf("b = %d\n",b);

```

- Prints out:
 - a = 255 (a = 0x000000FF)
 - b = -1 (b = 0xFFFFFFFF)

In-class Exercise

```
unsigned short    ushort; // 2-byte quantity
signed char      schar;
int              integer;

schar = 0xAF;
integer = (int) schar;
integer++;
ushort = integer;

printf("ushort = %d\n",ushort);
```

- What does this code print?
 - Or at least what's the hex value of the decimal value it prints?

In-class Exercise

```
unsigned short    ushort;
signed char      schar;
int              integer;

schar = 0xAF;

integer = (int) schar;

integer++;

ushort = integer;

printf("ushort = %d\n",ushort);
```

schar **AF**

integer **FF FF FF AF**

integer **FF FF FF B0**

ushort **FF B0**

prints 65456

More Signed/Unsigned in C

- On page 32 of the textbook there is an interesting example about the use of the `fgetc()` function
 - `fgetc` reads a 1-byte character from a file but returns it as a 4-byte quantity!
- This is a good example of how understanding low-level details is necessary to understand high-level constructs
- Let's go through the example...

The Trouble with `fgetc()`

- The `fgetc()` function in the standard C I/O library takes as argument a file opened for reading, and returns a character, i.e., an ASCII code
- This function is often used to read in all characters of the file
- The prototype of the function is:

```
int fgetc(FILE *)
```
- One may have expected for `fgetc()` to return a char rather than an int
- But if the end of the file is reached, `fgetc()` returns a special value called EOF (End Of File)
 - Typically defined to be -1 (`#define EOF -1`)
- So `fgetc` returns either
 - A character zero-extended into a 4-byte int (i.e., 000000xx)
 - Integer -1 (i.e., FFFFFFFF)

The Trouble with `fgetc()`

- Buggy code to compute the sum of ASCII codes in a text file:

```
char c;
while ( (c = fgetc(file)) != EOF) {
    sum += c;
}
```
- In this code we have mistakenly declared `c` as a char
- C being C (and not Java), it happily thinks we know we're doing and does a size-reduction of a 4-byte int into a 1-byte char when doing the assignment into `c`
- Let's say we just read in a character with ASCII code FF (decimal 255)
- `fgetc()` returned 000000FF, but it was truncated into 1-byte integer `c=FF`
 - FF is -1 in decimal
- So we then compare 1-byte value FF to 4-byte value FFFFFFFF
 - C allows comparing signed integer values of different byte sizes, for convenience, by internally sign-extending the shorter value
 - So FF is sign-extended into FFFFFFFF
- Therefore, the above code will "miss" all characters after ASCII code FF!
- Solution: declare `c` as an int (which may seem counter-intuitive)

Addition and Subtraction

- Two instructions used for additions and subtractions: **add** and **sub**
- **Both instructions can be used on a pair of signed numbers or on a pair of unsigned numbers**
 - One of the big advantages of 2's complement storage
 - No mixing of signed and unsigned numbers
- **IMPORTANT:** The CPU does not know whether numbers stored in registers are signed or unsigned!
 - You, the programmer, must keep your own interpretation of the number consistent throughout your program
 - The CPU will happily add whatever registers together using binary addition
- These two instructions each may set some bits of the FLAG register:
 - The **carry** bit
 - The **overflow** bit
 - The **zero** bit (=1 if the result is equal to zero)
 - The **sign** bit (=1 if the result is negative)

The Magic of 2's Complement

- I have two 1-byte values, A3 and 17, and I add them together:
 $A3 + 17 = BA$
- If my interpretation of the numbers is **unsigned**:
 - A3h = 163d
 - 17h = 23d
 - BAh = 186d
 - and indeed, $163d + 23d = 186d$
- If my interpretation of the numbers is **signed**:
 - A3h = -93d
 - 17h = 23d
 - BAh = -70d
 - and indeed, $-93d + 23d = -70d$
- So, as long as I stick to my interpretation, the binary addition does the right thing!!
 - Same thing for the subtraction
- This is why the computer does not need to know whether register contents are signed or unsigned

Overflow for Unsigned Operations

- There is an overflow with an unsigned operation (i.e., on unsigned quantities) if the carry bit is set
- If the carry bit is set, that means we'd need a larger quantity to hold the result
 - This also works for subtractions (instead of a carry, we have a "borrow", but it's still set in the carry bit)
- 1-byte Example (all in hex):
 - FF + 02 Carry is set (result would be 101h)
 - $255 + 2 > 255$
 - 01 - 02 Carry is set (result cannot be negative)
 - $1 - 2 < 0$
 - 8A - 0F Carry is not set (result is 7Bh)
 - $138 - 15 = 123$

In-Class Exercise

- Which of these operations set the Carry bit to 1? (presumably we care because we think of these as unsigned operations)
 - 0F12 + F212 (2-byte quantities)
 - 00E3 + F74F (2-byte quantities)
 - F1 - FA (1-byte quantities)
 - FB12 - A3AA (2-byte quantities)
 - A314 - B010 (2-byte quantities)
- Which of these operations set the Overflow bit to 1? (presumably we care because we think of these as signed operations)
 - 00E3 + FF4F (2-byte quantities)
 - F1 - 7A (1-byte quantities)

Overflow???

- Generally speaking, overflow occurs when the result of an arithmetic operation generates a result that's "out of range"
- This happens because a register has a **limited number of bits**, which means that our interpretation of a number comes with a **valid range**
- For instance
 - adding 1-byte unsigned quantity 240d to 1-byte unsigned quantity 100d will lead to an overflow because $340d > 255d$
 - subtracting 1-byte unsigned quantity 240d from 1-byte unsigned quantity 100d will lead to an overflow because $-140d < 0d$
 - adding 1-byte signed quantity 100d to 1-byte signed quantity 120d will lead to an overflow because $220d > 127d$
 - etc.
- Question: how do we detect overflow in a program?
 - Important otherwise we could be working with bogus numbers
- It depends on whether numbers are signed or unsigned...

Overflow for Signed Operations

- There is an overflow with a signed operation (i.e., on signed quantities) if the overflow bit is set
 - This bit is set when the sign of the result does not agree with the signs of the operands, which would be annoying for the programmer to check by hand
- 1-byte Example (all in hex, same as before):
 - FF + 02 Overflow is not set (result is 01h)
 - $-1 + 2 = +1$
 - 01 - 02 Overflow is not set (result is FFh)
 - $1 - 2 = -1$
 - 8A - 0F Overflow is set (result would be $< 80h$)
 - 8A is negative, and is equal to $-76h = -118d$
 - $-118 - 15 < -128$, and thus cannot be represented as a 1-byte signed quantity

In-Class Exercise

- Which of these operations set the Carry bit to 1?
 - 0F12
+ F212
= 10124 Carry bit is set
 - 00E3
+ F74F
= F832 Carry bit is not set
 - F1 - FA: F1 < FA Carry bit is set
 - FB12 - A3AA: FB12 > A3AA Carry bit is not set
 - A314 - B010: A314 < B010 Carry bit is set

In-Class Exercise

- Which of these operations set the Overflow bit to 1?
 - 00E3 + FF4F
 - 00E3 > 0, equal to decimal +251
 - FF4F < 0, 2's complement = 00B0+1 = B1, equal to decimal -177
 - +251 - 177 = 74
 - 2 byte unsigned numbers are in [-32,768, +32,767]
 - Overflow bit is not set
 - F1 - 7A
 - F1 < 0, 2's complement = 0E+1 = 0F, equal to decimal -15
 - 7A > 0, equal to 122
 - -15 - 122 = -137
 - 1-byte unsigned numbers are in [-128,+127]
 - Overflow bit is set

Overflow is your Responsibility

- The processor merely computes bits and puts them into the destination location as if everything were fine, and it's your responsibility to check the overflow!
- Let's look at two examples
 - An unsigned arithmetic example
 - A signed arithmetic example
- Note that we will see later how to "check" the Carry bit and the Overflow bit in the FLAGS register

Unsigned Overflow

On web site as
ics312_overflow_unsigned.asm

```

mov     al, 0F0h      ; al = F0h
mov     bl, 0A3h     ; bl = A3h
add     al, bl        ; al = al + bl
movzx   eax, al      ; increase size for printing
call    print_int    ; print al as an integer
  
```

- As a programmer we decided to do some computation with **unsigned values**
- We put value F0h in al (unsigned F0h is decimal 240)
- We put value A3h in bl (unsigned A3h is decimal 163)
- We add them together
- The "true" result should be decimal 240+163 = 403, which cannot be encoded on 8 bits (should be < 255)
- But the processor just goes ahead: F0 + A3 = 193h, and then drops the leftmost bits to truncate to a 1-byte value to get 93h!
- Therefore, when we call print_int, we print the decimal value 00000093, that is: 147!
- This is obviously wrong, and we can tell (or will be able to shortly) because the carry bit is in fact set to 1
- **Note that this is all correct if we assume signed values and replace movzx by movsx, but then our initial interpretation of the two values is different**

Signed Overflow

On web site as
ics312_overflow_signed.asm

```

mov     al, 09Ah     ; al = 9Ah
mov     bl, 073h     ; bl = 73h
sub     al, bl        ; al = al - bl
movsx   eax, al      ; increase size for printing
call    print_int    ; print al as an integer
  
```

- As a programmer we decided to do some computation with **signed values**
- We put value 9Ah in al (signed 9Ah is decimal -102)
- We put value 73h in bl (signed 73h is decimal +115)
- We subtract bl from al
- The "true" result should be decimal -102 - 115 = -217, which cannot be encoded on 8 bits (should be >= -128)
- But the processor just goes ahead: 9A - 73 = 27h
- Therefore, when we call print_int, we print the decimal value 00000027, that is: 39!
- This is obviously wrong, and we can tell (or will be able to shortly) because the overflow bit is in fact set to 1
- **Note that this is all correct if we assume unsigned values and replace movsx by movzx, but then our initial interpretation of the two values is different**

In-Class Exercise

```

mov     al, 0E1h
mov     bl, 0A2h
add     al, bl
movzx   eax, al
call    print_int
movsx   eax, bl
call    print_int
  
```

- What does this program print?

In-Class Exercise

```

mov     al, 0E1h
mov     bl, 0A2h
add     al, bl
movzx   eax, al
call    print_int
movsx   eax, bl
call    print_int
  
```

AL E1 BL A2

In-Class Exercise

```

mov    al, 0E1h
mov    bl, 0A2h
add    al, bl
movzx  eax, al
call   print_int
movsx  eax, bl
call   print_int
    
```

AL **83** BL **A2**

E1
+ A2
= 183

In-Class Exercise

```

mov    al, 0E1h
mov    bl, 0A2h
add    al, bl
movzx  eax, al
call   print_int
movsx  eax, bl
call   print_int
    
```

EAX **00 00 00 83** BL **A2**

E1
+ A2
= 183

In-Class Exercise

```

mov    al, 0E1h
mov    bl, 0A2h
add    al, bl
movzx  eax, al
call   print_int
movsx  eax, bl
call   print_int
    
```

EAX **00 00 00 83** BL **A2**

E1 prints out: 131
+ A2
= 183

In-Class Exercise

```

mov    al, 0E1h
mov    bl, 0A2h
add    al, bl
movzx  eax, al
call   print_int
movsx  eax, bl
call   print_int
    
```

EAX **FF FF FF A2** BL **A2**

E1
+ A2
= 183

In-Class Exercise

```

mov    al, 0E1h
mov    bl, 0A2h
add    al, bl
movzx  eax, al
call   print_int
movsx  eax, bl
call   print_int
    
```

EAX **FF FF FF A2** BL **A2**

FFFFFFA2 is a negative number prints out: -94
2's complement: (0000005D+1) = 5E
= decimal 94

Multiplication

- There are two instructions to perform multiplications
- Multiplying **unsigned** numbers: **mul**
- Multiplying **signed** numbers: **imul**
- Why do we need two different instructions?
- Consider the multiplication of FF by FF
 - If we assume unsigned quantities, this is $255 * 255 = 65035 = FE0Bh$
 - If we assume signed quantities, this is $-1 * -1 = 1 = 0001h$

The mul Instruction

- The size of the result of the multiplication is sometimes twice larger than the size of the operands
 - Multiplications just leads to much bigger numbers than additions
 - At most the result will be twice the size of the operands ($255 * 255 = 65,025$, which is encodable on 2 bytes)
- The oldest form of multiplication is the "mul" instruction, which produce a result twice the size of its operand
 - mul <register or memory reference>
 - If the operand is a byte, then it is multiplied by AL and the result is stored in (16-bit) AX
 - If the operand is 16-bit, it is multiplied by AX and stored in (32-bit) DX:AX
 - There used to be no 32-bit registers
 - If the operand is 32-bit, it is multiplied by EAX and the result is stored in (64-bit) EDX:EAX
 - We don't have 64-bit registers on a 32-bit architecture

The imul instruction

- Imul, which is used for signed numbers has three formats:
 - imul src
 - imul dst, src1
 - imul dst, src1, src2
- The different combinations are shown in Table 2.2 in the text book
- This table uses the typical way in which one specifies operands:
 - reg16: a 16-bit register
 - reg32: a 32-bit register
 - immed8: an 8-bit immediate operand (i.e., a number)
 - mem16: a word of memory
 - etc.
- Let's look at the table

The imul instruction

Will not overflow (although the overflow bit may be set)

dst	src1	src2	action
	reg/mem8		AX = AL * src1
	reg/mem16		DX:AX = AX * src1
	reg/mem32		EDX:EAX = EAX * src1
reg16	reg/mem16		dst *= src1
reg32	reg/mem32		dst *= src1
reg16	immed8		dst *= immed8
reg32	immed8		dst *= immed8
reg16	immed16		dst *= immed16
reg32	immed32		dst *= immed32
reg16	reg/mem16	immed8	dst = src1*src2
reg32	reg/mem32	immed8	dst = src1*src2
reg16	reg/mem16	immed16	dst = src1*src2
reg32	reg/mem32	immed32	dst = src1*src2

Division

- Two instruction:
 - div for unsigned quantities
 - idiv for signed quantities
- They perform **integer division**
 - e.g.: $19 / 4$ produces quotient = 4 remainder = 3
- Only one format for both:
 - div/idiv src
- If src is an 8-bit quantity:
 - AX is divided by src
 - quotient stored into AL
 - remainder stored into AH
- If src is a 16-bit quantity:
 - DX:AX is divided by src
 - quotient stored into AX
 - remainder stored into DX

Division

- If src is a 32-bit quantity:
 - EDX:EAX is divided by src
 - quotient stored into EAX
 - remainder stored into EDX
- Warning: it's very common for programmers to forget initializing DX or EDX before the division

Negation

- There is a convenient instruction to negate an operand: **neg**
- It simply computes the 2's complement of a quantity
- Works on 8-bit, 16-bit, or 32-bit quantities
 - either in registers or in memory
- We'll ignore the content of Section 2.1.5 in the textbook

Example Program in Textbook

- Section 2.1.4 shows a sample program that uses all the arithmetic operations we just saw
- There is nothing particularly difficult about it, especially because overflows are not handled (so the numbers entered had better be “small”)
- One interesting point: One cannot divide by an immediate value and must use a register
- Make sure you go through this example and understand how it works
 - You may want to run it as well

Homework #3

- HW #3 to be posted shortly

In-class Quiz

- Quiz #4 will be on this set of slides
 - Basic Assembly Language
- The quiz will be on ...