Data Size

ICS312

Machine-Level and Systems Programming

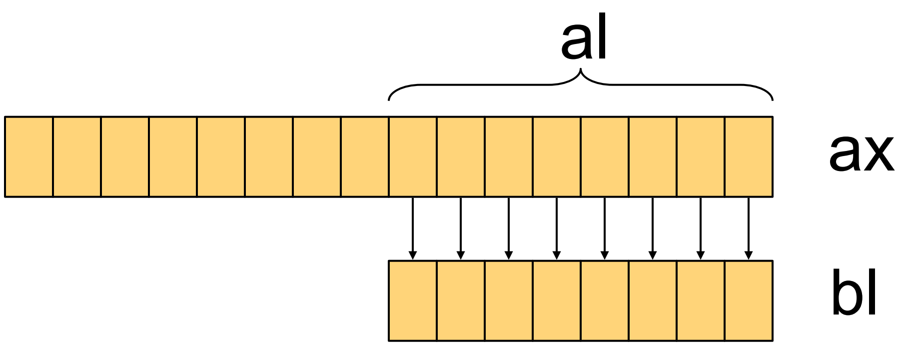
Henri Casanova (henric@hawaii.edu)

Size of Data

- In .data and .bss segments, a label merely declares an address
- No data size that would be used by the program is enforced in the declaration
  - 2 dw's can be later used as 4 db's or 1 dd
- Instead, the 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 (aka Type Narrowing)

- Sometimes one needs to decrease the data size
- For instance, you have a 4-byte integer, but you needs to use it as a 1-byte integer for some purpose
  - e.g., you did a **`read_int`**, but you know the number is between 0 and 128 and is in fact a 1-byte ASCII code
- We can simply use the the fact that we can access lower bits of some registers independently
- Example:
  - mov ax, [L] ; load 16 bits in ax
  - mov bl, al ; take the lower 8 bits of ax and puts them in bl



Size Reduction (aka Type Narrowing)

- When doing a size reduction, one loses information
- So the “conversion to integers” may or may not work
- Example that “works”:
  - mov ax, 000A2h ; ax = 162 decimal
  - mov bl, al; ; bl = 162 decimal
  - Decimal 162 is *encodable* on 8 bits (because it’s < 256)
- Example that “doesn’t work”:
  - mov ax, 00101h ; ax = 257 decimal
  - mov bl, al; ; bl = 1 decimal
  - Decimal 257 is *not encodable* on 8 bits because > 255

Size Reduction and Sign

- Consider a 2-byte quantity: FFF4
- If we interpret this quantity as unsigned it is decimal 65,524
  - 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, using the right instructions (more on this later)
- In this case size reduction "does not work", meaning that reduction to a 1-byte quantity will not be interpreted as decimal 65,524 (which is way over 255!), 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!
  - 1-byte value F4 is decimal value -12

Size Reduction and Sign

- The previous examples do **not** mean that size reduction always works for signed quantities
- For instance, consider signed 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!
- This is because -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!
- In other words, there will always be “bad” cases

High-Level Languages

- All that we said in the previous slides applies to high-level languages
- For instance, in C/C++

```
#include <stdio.h>
int main()  {

    int   a = 65535;     // 4-byte
    short b = a;         // 2-byte
    printf("%d\n",b);    // prints "-1"

    int   x = -50000;    // 4-byte
    short y = x;         // 2-byte
    printf("%d\n",y);    // prints  "15536"

}
```

- No compiler warning

High-Level Languages (2)

- Other languages are a bit more conservative, e.g., Java

```
public class Foo {
    public static void main(String[]  args) {
        int a = 65535;
        short b =  a; // Does not compile
                    // (incompatible types: possible lossy
                    // conversion from int to short)

        short c = (short)a;    // Compiles
        System.out.println(c); //  prints “-1"
    }
}
```

- Other languages automatically adjust data size based on values, e.g., Python3

Two Rules to Remember

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

0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X

X	X	X	X	X	X	X	X

- 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

a	a	a	a	a	a	a	a	a	X	X	X	X	X	X	X

a = 0 or 1

a	X	X	X	X	X	X	X

Size Increase (aka Type Widening)

- Size increase for **unsigned** quantities is simple: just add 0's 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

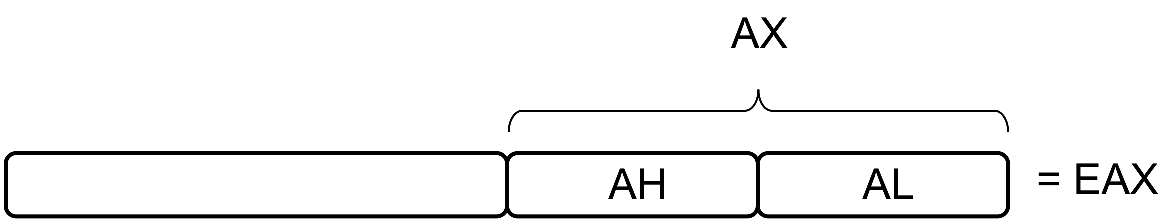
Unsigned size increase

- 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
mov      ..., ax   ; =00ED
```

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 in memory, or 1 word in memory
  - The destination must be larger than the source!

Using movzx

- movzx eax, ax ; zero extends ax into eax
- movzx eax, al ; zero extends al into eax
- movzx ax, al ; zero extends al into ax
- movzx ebx, ax ; zero extends ax into ebx
- movzx ebx, [L] ; leads to a "size not specified" error
- movzx ebx, **byte** [L] ; zero extends 1-byte value at address L into ebx
- movzx eax, **word** [L] ; zero 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
  - Sometimes we want to add 0's (like movzx), but sometimes we want to add 1's
- For this reason, we have a new instruction: movsx
  - Works just like MOVZX, but does sign extension
- Let's see an example..

Example

```
mov al, 0A7h        ; as a programmer, I view this
                    ; as an 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, ax
movzx    ebx, bx
```

- What’s the final value of eax?
- What’s the final value of ebx?

(poll)

In-class Exercise Solution

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

Signed/Unsigned in C

- In C/C++ one can declare variables as signed or unsigned
  - In Java you don't have unsigned data types, but there are methods that treat signed data types as unsigned
    - which a LOT of people hate with a passion, with pretty good reasons
    - the rationale is likely that Java should be "easy for the average developer"
- Why would I like a language that supports signed/unsigned?
  - 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, which wastes bytes, and thus CPU cycles
  - When dealing with various binary data formats, it's really convenient to know exactly what the data means and manipulate it without extra bits
  - If I make a mistake (like setting an unsigned variable to a negative value) I want to compiler to complain!
- Let's look at a small C-code example

Signed/Unsigned in C/C++

- Declarations:

```
unsigned char   uchar = 0xFF;
signed char     schar = 0xFF; // "char"="signed char"
```

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

```
for (char 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" = "signed int" in C/C++)
- 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 MOVSX 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);
```

- What does this program print?

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)

printf in C

- So, by declaring variables as “signed” or “unsigned” you define which of movsx or movzx will be used when you have a cast in C
- Printf can print signed or unsigned interpretation of numbers, regardless of how they were declared:
  - “%d”: signed decimal
  - “%u”: unsigned decimal
- Arguments to printf are automatically extended to 4-byte integers! (using movzx or movsx internally)
  - Unless you specify “short” as in “%hd” or “%hu”
- Good luck understanding this if you have never studied assembly at all...
- Let’s try a simple example

Understanding printf

```
unsigned short        us = 259;  // 0x0103
signed short          ss = -45;  // 0xFFD3

printf("%d %d\n",us, ss);
printf("%u %u\n",us, ss);
```

- Let’s together try to understand what will be printed before we look at the answer…

Understanding printf

```
unsigned short        us = 259;  // 0x0103
signed short          ss = -45;  // 0xFFD3

printf("%d %d\n",us, ss);
printf("%u %u\n",us, ss);
```

259 -45

259 4294967251

A “kitchen sink” example

```
unsigned short      ushort;    // 2-byte quantity
signed   char       schar;     // 1-byte quantity
int                 integer;   // 4-byte quantity

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

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

- What does this code print?
  - Or what’s the hex value of the value it prints?
- Let’s do this together…

A “kitchen sink” example

```
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		

Because printf doesn’t specify “h” ushort is size augmented to 4-bytes using movzx (because declared as unsigned): 00 00 FF B0
The number is then printed as a signed integer (“%d”): 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 can be 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), or
  - 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 thinks we know what 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, "ÿ")
- 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
    - int x=-1; char y=-1; // (x == y) returns TRUE
  - So FF is sign-extended into FFFFFFFF
- Therefore, the above code will "miss" all characters after ASCII code FF and mistake them for an end of file
- Solution: declare c as an int (which may seem counter-intuitive)

Example Type Widening Bug

- If you search around, you'll find bug reports about type widening pretty frequently
- For instance, https://unspecified.wordpress.com/2011/08/08/integer-conversions-in-c/
- Last paragraph is particularly illuminating
  - There's an implicit type widening of a signed char, that then can add a bunch of 1's when the intent was to always add a bunch of 0's
- This bug is for a popular password encryption library, which weakens its security
  - "This can result in passwords being even easier to crack than expected. This is due to a char signedness bug in crypt_blowfish."

Should you care?

- It all depends of what kind of work you do and what kind of software you deal with
  - Some codes will have stuff like that all over with signed/ unsigned declarations and casts galore
  - Some codes will have none of that ever
- If all you do is JavaScript Web app development, you likely will rarely care
- If you do lower-level development, you may care every single day
  - Or rather, if you don't know all this, your life will be very difficult
- Overall, it's pretty rare to completely avoid it for your entire life
  - Often due to binary data formats used all over the place

Conclusion

- Being aware of data sizes and of data size extension/reduction behaviors is important when doing low-level development
  - Assembly, C, C++, etc.
- Unfortunately, almost every developer at some point is confronted with data size issues and having studied a bit of assembly is the only way to solve mysteries
  - Important to know that a cast isn’t magical, and can do the "wrong” thing