Basic Assembly Language I
(Data Size)

ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)
Size of Data

- Labels merely declare an address in the data segment, and do not specify any data 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 some purpose
- We simply use the fact that we can access lower bits of some registers independently
- Example:
  - `mov ax, [L]` ; loads 16 bits in ax
  - `mov bl, al` ; takes the lower 8 bits of ax and puts them in bl
Size Reduction

- Of course, 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 (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!
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!
- 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!
  - There will always be “bad” cases.
Two Rules to Remember

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

  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 a X X X X X X X X

  a = 0 or 1
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`
- `mov    ..., 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 \texttt{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.
- Four “old” conversion instructions with implicit operands:
  - CBW (Convert Byte to Word): Sign extends AL into AX
  - CWD (Convert Word to Double): Sign extends AX into DX:AX
    - DX contains high bits, AX contains low bits
    - 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 MOVSX instruction:
  - Works just like MOVZX, but does sign extension
  - CBW equiv. to MOVSX ax, al
  - CWDE equiv. to MOVSX 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
                 ; (FFFFFFFFFA7)
```
Signed/Unsigned in C

- In the C (and C++) language (not in Java!) 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:
  
  ```c
  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
    ```c
    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” = “signed int”)

- We need to extend 1-byte values to 4-byte values

- This is done in C with a “cast”
  ```c
  int a = (int) uchar;  // the compiler will use MOVZAX to do this
  int b = (int) schar;  // the compiler will use MOVZX to do this
  ```
Signed/Unsigned in C

```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
  - “%u”: unsigned
- Arguments to printf are automatically size extended to 4-byte integers!
  - Unless you specify “short” as in “%hd” or “%hu”
- Good luck understanding this if you have never studied assembly at all...
- Let’s try this out (this is overkill, but if you understand it, then you understand much more than the average C developer!)
Let’s together try to understand what will be printed.....
Understanding printf

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

printf("%d %d\n", us, ss);
printf("%u %u\n", us, ss);
printf("%hd %hd\n", us, ss);
print("%hu %hu\n", us, ss);

259    -45
259    4294967251
259    -45
259    65491
What does this code print?

Or at least what’s the hex value of the decimal value it prints?
Example

unsigned short ushort;
signed char schar;
int integer;
schar = 0xAF;
integer = (int) schar;
integer++;
ushort = integer;
printf("ushort = %d\n",ushort);

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:
  
  ```c
  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:
  ```c
  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)
Conclusion

- If everything you do is Java, then these issues should never arise.
- But being aware of data sizes and of data size extension/reduction behaviors is important when doing low-level development:
  - Assembly, C, etc.
- Unfortunately, almost every developer at some point is confronted with data size issues and having studied a bit of assembly is really the way to remove mysteries.