ASSEMBLY LANGUAGE FUNDAMENTALS

- Assembly language statements are either *directives* or *instructions*

- Instructions are executable statements. They are translated by the assembler into machine instructions. Ex:

```
CALL MySub ;transfer of control
MOV  AX, 5 ;data transfer
```

- Directives tells the assembler how to generate machine code and allocate storage. Ex:

```
count db 50 ;creates 1 byte of
             storage initialized to 50
```

Template for Assembly Language Programs

```
.386
.model  Flat
include Cs266.inc

.data
  ... ;data allocation directives here
  ⋮

.code
  main:
    ... ;program instructions here
    ⋮
    Ret
  end
```

- .386: Directive to accept all instructions of 386 and previous processors (use .586 to assemble Pentium specific instructions)

- `main`: Label of the entry point of the program (first instruction to execute)

- `end`: Directive that marks the end of the program

- `ret`: Instruction that returns the control to the caller (here Win32 console)

- Macros to perform I/O are included in Cs266.inc

The `FLAT` Memory Model

- The `.model flat` directive tells the assembler to generate code that will run in protected mode and in 32-bit mode

- Also asks the assembler to do whatever is needed in order that code, stack and data share the same 32-bit memory segment

  All the segment registers will be loaded with the correct values at load time and do not need to be changed by the programmer

- Only the offset part of a virtual address becomes relevant

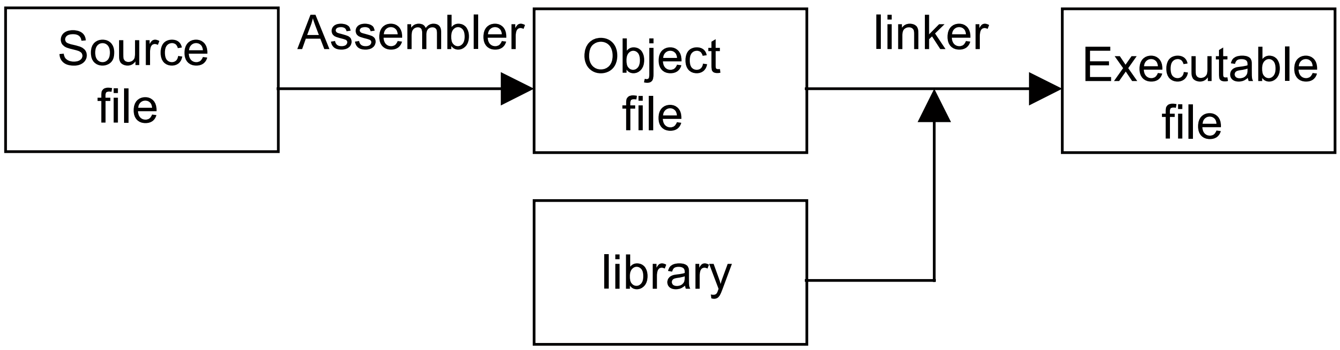
  Each data byte (or instruction) is referred to only by a 32-bit offset address

- The directives `.data` and `.code` mark the beginning of the data and code segments. They are used only for protection

  1. `.data` is a read and write segment

  2. `.code` is a read-only segment

Producing an Executable File



1. The *assembler* produces an object file from the assembly language source

2. The object file contains machine language code with some external and relocatable addresses that will be resolved by the linker. Their values are undetermined at that stage

3. The *linker* extract object modules (compiled procedures) from a library and links them with the object file to produce the executable file

4. The addresses in the executable file are all resolved but they are still virtual addresses

Using Borland's BCC32

- All these steps are performed with the command:

```
bcc32 -v hello.asm
```

- The bcc32 command calls TASM32 to assemble and produce an object file

- It then calls ILINK32 to link this object file with the C/C++ library functions and Win32 functions used by the program to produce the executable file hello.exe

- The -v option produces full debugging information

- See my home page for all the information you need

Names and Variables

- A *name* identifies either a:
  1. Variable
  2. Label
  3. Constant
  4. Keyword (assembler-reserved word

- The first character must be a letter or '@', '_', '$' or '?'. Subsequent characters can include digits

- A programmer chosen name must be different from an assembler reserved word

  Avoid using '@' as the first character since many keywords start with it

- A *variable* is a symbolic name for a location in memory that was allocated by a data allocation directive

```
Count db 50; allocates 1 byte for variable Count
```

- A *label* is a name given to an instruction. It must be followed by ':'

```
main:
  MOV  EAX, 5
  XOR  EAX, EBX
  JUMP main
```

- When called from bcc32, the TASM32 assembler is case sensitive for user-defined words, but case insensitive for the assembler reserved words

Integer, Character and String Constants

1. Integer constants are made of numerical digits with, possibly, a sign and a suffix.

   - −23 (negative integer, base 10 is default)
   - 1101b (binary number)
   - 1011 (decimal number)
   - 0A7Ch (hexadecimal number)
   - A7Ch (this is the name of a variable; an hexadecimal number must start with a decimal digit)

2. Character and string constants are any sequence of characters enclosed either in single or double quotation marks. Embedded quotes are permitted.

   - 'A'
   - 'ABC'
   - "Hello World!"
   - "123" (this is a string, not a number)
   - "This isn't a test"
   - 'Say "Hello" to him'

Data Allocation Directives

- The DB (define byte) directive allocates storage for one or more byte values

```
[variable name] DB initval [, initval]
```

Each initializer can be any constant.

```
Var1 DB 10, 32, 41    ;allocates 3 bytes
Var2 DB 0Ah, 20h, 'A' ;same values as above
```

- A question mark (?) in the initializer leaves the initial value of the variable undefined.

```
Var3 DB ? ;the initial value for Var3 is undefined
```

- Everything after the ';' is a comment

- A string is stored as a sequence of characters

```
StrVar1 DB "ABCD
StrVar2 DB 'A', 'B', 'C', 'D' ;same values as above
StrVar3 DB 41h, 42h, 43h, 44h ;same values again
```

Data Allocation Directives (Continued)

- The (offset) address of a variable is the address of its first byte.

```
.data
  Var1 DB "ABC"
  Var2 DB "DEFG"
```

If the above data segment starts at address 0 then

1. Address of `Var1` is 0
2. Address of `'A'` is 0
3. Address of `'B'` is 1
4. Address of `C'` is 2
5. Address of `Var2` is 3
6. . . . Address of `'G'` is 6

- DW (define word) allocates a sequence of words

```
Var3 DW 1234h, 5678 ;allocates 2 words
```

- Intel's x86 are little *endian* processors: the lowest order byte (of a word or double-word) is always stored at the lowest address

- If `Var3` (above) is located at address 0, then

1. Address: 0 1 2 3
2. Values: 34h 12h 78h 56h

Data Allocation Directives (Continued)

- `DD` (define double-word) allocates a sequence of double-words

```
Var1 DD 12345678h ;allocates 1 double-word
```

If `Var1` is located at address 0 then

1. Address: 0 1 2 3
2. Values: 78h 56h 34h 12h

- If a value fits into a byte, it'll be stored in the lowest order byte available

```
Var2 DW 'A'
```

The value will be stored as

1. Address: 0 1
2. Values: 41h 00h

- The `DUP` operator duplicates storage values

```
Var1 DB 100 DUP(?)     ;allocate 100 uninitialized bytes
Var2 DB 3   DUP("Ho") ;allocates 6 bytes: "HoHoHo"
```

DUP can be nested

```
Var3 DB 2 DUP('a', 2 DUP('b')) ;allocates 6 bytes: 'abbabb'
```

- `DUP` must be used with data allocation directives only

Constants

- We can use the equal-sign (=) directive or the `EQU` directive to give a name to a constant

```
Cst1 = 1;  ;this is a constant
Cst2 EQU 2 ;also a constant
```

- The `EQU` and = directives are equivalent
- The assembler does not allocate storage to a constant (in contrast with data allocation directives)
- It merely substitutes, at assembly time, the value of the constant at each occurrence of the assigned name
- A *constant expression* involves the standard operators used in HLLs: `+`, `-`, `*`, `/`. Ex: the constant expression below is evaluated at assembly time and given a name at assembly time

```
Cst3 = (-3 * 8) + 2
```

- A constant can be defined in terms of another constant

```
Cst4 EQU (Cst3 + 2) / 2
```

Exercise #1

- Suppose that the following data segment starts at address 0

```
.data
  Var1 DW  1, 2
  Var2 DW  6ABCh
  Cst1 EQU 232
  Var3 DB 'ABCD'
```

Find the address of

1. Variable `Var1`
2. Variable `Var2`
3. Variable `Var3`
4. Character `'C'`

Data Transfer Instructions

- `MOV Destination, Source` → transfers the content of the source operand to the destination operand. This changes the content of `Destination` only. Also, both operands must be of the same size

- An operand can be either *direct* or *indirect*

- Direct operands (this chapter) are either

  1. Immediate (constant): called `Imm`
  2. Register: called `Reg`
  3. Memory variable (with displacement): called `Mem`

- Indirect operands are used for indirect addressing

- `MOV` restrictions

  1. Source and destination cannot both be `Mem`
  2. Destination operand cannot be `Imm`
  3. `EIP` cannot be an operand

Data Transfer Instructions (Continued)

- The type of an operand is given by its size. Hence both operands of `MOV` must be of the same type

- *Type checking* is done by the assembler

- The type assigned to a `Mem` operand is given by its data allocation directive

- The type assigned to a `Reg` operand is given by its register size

- An `Imm` source operand of `MOV` must fit into the size of the destination operand

- Examples of `MOV` usage

```
MOV BH,   255              ;8-bit operands
MOV AL,   256              ;Error: constant too large
MOV BX,   WordVar1         ;16-bit operands
MOV BX,   ByteVar1         ;Error: size mismatch
MOV EDX,  DoubleWordVar1   ;32-bit operands
MOV CX,   BL               ;Error: size mismatch
MOV Var1, Var2             ;Error: Mem-to-Mem
```

`MOVZX`: Move with Zero Extend

- `MOVZX Destination, Source` → moves the content of the source operand into a destination of larger size. High order part of `Destination` is filled with 0's

- `Imm` operands are not allowed

- Destination type must be *strictly* larger than source type

- Example

```
MOV   BH,  80h ;BH = 80h
MOVZX AH,  BH  ;Illegal:  size mismatch
MOVZX AX,  BH  ;AX = 0080h
MOVZX ECX, AX  ;ECX = 00000080h
```

- Notice that if the signed value in the source operand is negative, then `MOVZX` will not preserve the sign

```
MOV   BH, 80h ;BH = 80h is negative
MOVZX AX, BH  ;AX = 0080h is positive
```

MOVSX: Move with Sign Extend

- MOVSX Destination, Source → preserves the sign of the source operand. High order part of Destination is filled with the sign of Source

  The sign extension of a negative number is . . . 111111

  The sign extension of a positive number is . . . 000000

  Example

```
MOV   BH, 80h ;BH = 80h is negative
MOVSX AX, BH  ;AX = FF80h is positive
              ;FFh is the sign extension of 80h
MOVSX BL, 7Ah ;BL = 7Ah is positive
MOVSX AX, BL  ;AX = 007Ah is positive
              ;00h is the sign extension of 7Ah
```

- MOVSX preserves the signed value whereas MOVZX preserves the unsigned value

- Imm operands are not allowed and destination type must be *strictly* larger than source type

Data Transfer Instructions (Continued)

- We can add a displacement to a memory operand to access a memory value without a name

```
.data
  ArrB DB 10h, 20h
  ArrW DW 1234h, 5678h
```

  `ArrB+1` points to the second byte of `ArrB` and `ArrW+2` points to the third byte of `ArrW`

```
MOV AL, ArrB   ;AL = 10h
MOV AL, ArrB+1 ;AL = 20h
MOV AX, ArrW+2 ;AX = 5678h
MOV AX, ArrW+1 ;AX = 7812h
               ;Little endian convention!
MOV AX, ArrW-2 ;AX = 2010h
               ;negative displacement allowed
```

- `XCHG Destination, Source` → swaps the contents of `Source` and `Destination`. Operands must be `Mem` or `Reg`, must have the same type, and cannot be both `Mem`

- To exchange the content of two `Mem` operands

```
MOV  AX,       WordVar1
XCHG WordVar2, AX
MOV  WordVar1, AX
```

Exercise #2

- Given the following data segment

```
.data
  A DW 1234h, -1
  B DD 55h, 66778899h
```

- Indicate if each of the following instructions is legal. If it is, indicate the value, in hexadecimal, of the destination operand immediately after the instruction is executed (please verify your answers with a debugger)

```
MOV EAX, A
MOV BX,  A+1
MOV BX,  A+2
MOV DX,  A+4
MOV CX,  B+1
MOV EDX, B+2
```

Arithmetic Instructions

- <u>`ADD Destination, Source`</u> $\longrightarrow$ adds the source to the destination

- <u>`SUB Destination, Source`</u> $\longrightarrow$ subtracts the source from destination.

- Result of `ADD` or `SUB` is stored in `Destination` and `Source` remains unchanged. Operands must have the same type and cannot be both `Mem`

- Recall: for `A - B`, the CPU performs `A + NEG(B)`

- `ADD` and `SUB` affect all the status flags of the `EFLAGS` register according to the result of the operation

  `ZF` (zero flag) $= 1 \leftrightsquigarrow$ result is 0

  `SF` (sign flag) $= 1 \leftrightsquigarrow$ MSB is 1

  `OF` (overflow flag) $= 1 \leftrightsquigarrow$ signed overflow

  `CF` (carry flag) $= 1 \leftrightsquigarrow$ unsigned overflow

  1. Signed overflow: out-of-range signed value

  2. Unsigned overflow: out-of-range unsigned value

More on Overflows

- *Signed (unsigned) overflow* occurs if and only if (iff) the signed (unsigned) value of the result does not fit into the destination.

  This happens iff the signed (unsigned) interpretation of the result is erroneous. It is signaled by `OF = 1` (`CF = 1`)

- Both types of overflow occur independently and are signaled separately by `OF` and `CF`

```
MOV AL, 0FFh
ADD AL, 1     ;AL=00h, OF=0, CF=1
MOV AL, 7Fh
ADD AL, 1     ;AL=80h, OF=1, CF=0
MOV AL, 80h
ADD AL, 80h   ;AL=00h, OF=1, CF=1
```

  Hence we can have either type of overflow or both at once

Overflow Examples

```
MOV AX, 4000h
ADD AX, AX    ;AX = 8000h
```

1. Unsigned Interpretation:
   $\leadsto$ unsigned result is correct, hence `CF` = 0

2. Signed Interpretation:
   $\leadsto$ we add two positive numbers: `4000h + 4000h`
   $\leadsto$ and obtain a negative number (!)
   $\leadsto$ signed result is incorrect, hence `OF` = 1

```
MOV AX, 8000h
SUB AX, 0FFFFh ;AX = 8001h
```

1. Unsigned Interpretation:
   $\leadsto$ we subtract a larger magnitude (`0FFFFh`) from a smaller magnitude (`8000h`)
   $\leadsto$ unsigned result is incorrect, hence `CF` = 1

2. Signed Interpretation:
   $\leadsto$ signed result is correct (`0FFFFh = -1`), hence `OF` = 0

```
MOV AH, 40h
SUB AH, 80h ;AX = C0h
```

1. Unsigned Interpretation:
   $\leadsto$ we subtract a larger magnitude (`80h`) from a smaller magnitude (`40h`)
   $\leadsto$ unsigned result is incorrect, hence `CF` = 1

2. Signed Interpretation:
   $\leadsto$we subtract the negative number `80h` (-128) from the positive number `40h` (64)
   $\leadsto$ and obtain a negative number (!)
   $\leadsto$ signed result is incorrect, hence `OF` = 1

Exercise #3

- For each of the following instructions, give the content (in hexadecimal) of the destination operand and the `CF` and `OF` flags immediately after the execution of the instruction (verify your answers with a debugger)

  1. `ADD AX, BX` when

     `AX` contains 8000h and

     `BX` contains FFFFh

  2. `SUB AL, BL` when

     `AL` contains 00h and

     `BL` contains 80h

  3. `ADD AH, BH` when

     `AH` contains 2Fh and

     `BH` contains 52h

  4. `SUB AX, BX` when

     `AX` contains 0001h and

     `BX` contains FFFFh

- <u>`INC Destination`</u> → adds 1 to a single `Mem` or `Reg` operand

- <u>`DEC Destination`</u> → subtracts 1 from a single `Mem` or `Reg` operand

- Both instructions affect all status flags, except `CF`. Ex: if `CF = OF = 0` initially, then

```
MOV BH, 0FFh ;         CF=0, OF=0
INC BH       ;BH=00h, CF=0, OF=0
MOV BH, 7Fh  ;         CF=0, OF=0
INC BH       ;BH=80h, CF=0, OF=1
```

- <u>`NEG Destination`</u> →performs the two's complement of its single `Mem` or `Reg` operand

  `CF` = 0 ⇝ the result is 0
  `OF` = 1 ⇝ there is a signed overflow

```
MOV AX, -5
NEG AX          ;CF=1, OF=0
MOV AX, 8000h
NEG AX          ;CF=1, OF=1 signed overflow!
```

Input/Output on the Win32 Console

- Our programs will communicate with the user via the Win32 concole (the MS-DOS box)

  1. Input is done on the keyboard
  2. Output is done on the screen

- Modern OS like Windows forbids user programs to interact directly with I/O hardware

  User programs can only perform I/O operations via system calls

- For simplicity, our programs will perform I/O operations by using macros that are provided in Cs266.inc file

  1. These macros call C library functions like `printf()` which, in turn, call the Win32 API
  2. Hence, these I/O operations will be slow but simple to use and easy to migrate to another OS

- We will examine the mechanisms involved in I/O operations later in the course

Character Output Macro

- PUTCH Source → prints on the screen the character of the operand's ASCII code. Where `Source` must be a 32-bit operand, that is either `Imm`, `Reg32` or `Mem32`. The cursor will advance one position after printing the character

```
.data
  Wrd DW 41h
  Drd DD 61h
.code
  PUTCH Wrd       ;error: 16 bit operand
  PUTCH Drd       ;'a' is written on screen
  PUTCH 'b'       ;'b' is written on screen
  MOV   EAX, 'c'
  PUTCH EAX       ;'c' is written on screen
  PUTCH AX        ;error: 16-bit operand
```

- `PUTCH` macro calls the `putchar()` function from the C library. Hence

  The number 10 = 0Ah will direct the cursor to the start of the next line (the *newline character* in C). So the `<CR>` and `<LF>` functions are both performed on the screen

```
PUTCH 10 ;moves the cursor to the
         ;start of the next line
```

String Output and Integer Output Macros

- PUTSTR Source → prints a string. Where Source must be a Mem operand

- PUTSTR calls the C library's printf("%s",). Hence

  1. The number 10 = 0Ah will move the cursor to the start of the next line

  2. The string must be a *null terminating* string. The last character must have ASCII code 0h

```
.data
  Msg DB "hello", 0Ah, "world", 0h
.code
  PUTSTR Msg ;prints 'hello' on one line, and
             ;prints 'world on the next line
```

- PUTINT Source → prints the signed value of an integer. Where Source must be a Imm, Reg32 or Mem32 operand

```
.data
  Wrd DW 243
  Drd DD -266
.code
  PUTINT Wrd                  ;error: 16 bit operand
  PUTINT Drd                  ;-266 is written on screen
  PUTINT -1                   ;-1 is written on screen
  MOV    EAX, 0FFFFFFFFh
  PUTINT EAX                  ;-1 is written on screen
  PUTINT AX                   ;error: 16-bit operand
```

Character Input Macro

- GETCH → reads one or more characters on the keyboard

- This macro calls C library's `getchar()`. So it uses a memory buffer called the *input buffer*. Upon execution of `GETCH`, the input buffer is first examined

- If the buffer is empty, then `GETCH` waits for the user to enter an input line (a sequence of char ended by `<CR>`)

  1. Each character that the user enters (at the keyboard) is copied into the buffer

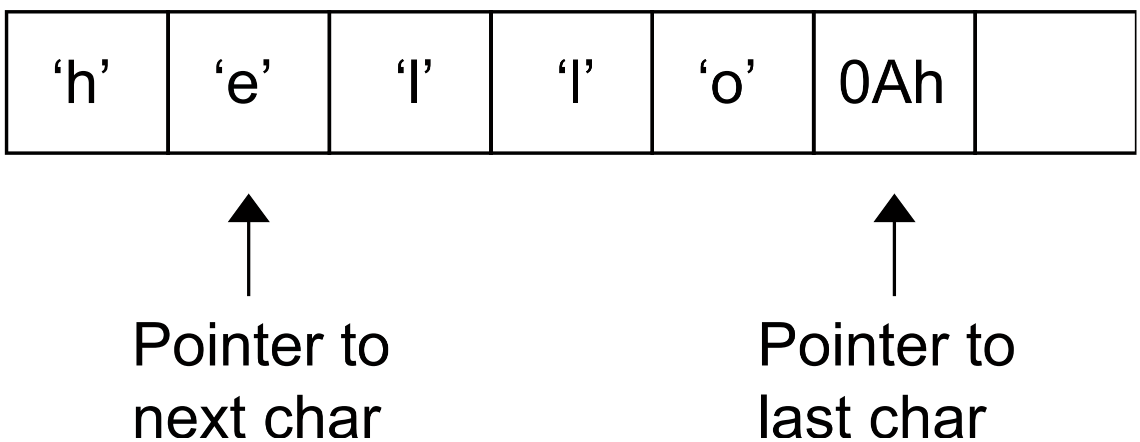
  2. When the user enters `<CR>`: the cursor moves to the next line, the value 0Ah is stored in the buffer and the control is passed to the instruction following `GETCH`

  3. The ASCII code of the first character entered on the keyboard will be stored in `AL`. The remaining bits of `EAX` are filled with 0's

     ```
     MOV   EAX, -1
     GETCH Drd      ;EAX = 41h
                    ;if the user first hits 'A'
     ```

Character Input Macro (Continued)

- Ex: Suppose that the buffer is initially empty and, upon execution of `GETCH`, the user enters `"hello"+<CR>` on the keyboard. Then, when the control returns to the instruction following `GETCH`, `EAX` contains 068h (='h') and the input buffer looks like this



- If the buffer is not empty when `GETCH` is executed, then `EAX` will be loaded with the ASCII code of the next character in the buffer and the pointer to the next character will increase by one

- The buffer is empty only when the pointer to the next character points beyond the last character (i.e. 0Ah)

- The user is prompted only when the buffer is empty

Character Input Macro (Example)

```
.386
.model  Flat
include Cs266.inc
.code
  main:
    PUTCH '?'
    PUTCH 10
    GETCH
    PUTCH EAX
    GETCH
    PUTCH EAX
    GETCH
    PUTCH EAX
    Ret
  end
```

- Try to understand this program: It first prints
  `"?"`
  and moves the cursor to the next line awaiting user input

- When the user enters `"abcdef"+<CR>`, the program displays (before exiting)
  `abc`

- But if, instead, the user enters `"a"+<CR>`, the program displays
  `a`
  and the cursor moves to the next line awaiting user input. If the user then enters `"bcdef"+<CR>`, the program prints on the next line (before exiting)
  `b`

Character Input Macro (Example)

```
.386
.model  Flat
include Cs266.inc
.data
  Msg1 DB "Enter a lower case letter:", 0
  Msg2 DB 'In upper case it is:'
  Char DB ?, 0
.code
  main:
    PUTSTR Msg1
    GETCH           ;letter in EAX and goto next line
    SUB    AL, 20h  ;converts to upper case letter
    MOV    Char, AL
    PUTSTR Msg2
    Ret
  end
```