Notes on p86 Assembly Language and Assembling

A **program** is a collection of instructions and data that is loaded into a computer system's memory such that:

IF program execution begins at the (intended) first instruction to be executed
THEN the computer system will accomplish it's objective

A programming language is a notation for describing programs. The syntax and semantics of a programming language are constrained to narrow the interpretation of syntactic statements in describing programs. Constraining the language is necessary to allow the practical construction of tools like interpreters, compilers and assemblers. (Less constrained languages, like English, are too robust for this purpose – it is too hard to build automated tools (e.g. compilers) that can convert general English language statements into executable collections of instructions and data (programs).)

An assembly language is a programming language that permits the description of programs at the level of the computer system's state variables (i.e. in terms of a programming model that consists of processor registers, the processor's instruction set, memory cells, and I/O ports). The language has a syntax and a semantics. The **syntax** consists of the permitted atomic symbols and the set of grammar rules that specify how syntactically correct statements are formed using symbols. The **semantics** is a set of rules for interpreting syntactically correct statements in terms of the programming model of the computer system. A language's syntax gives *form* to descriptions, while the language's semantics gives *meaning*.

NOTE: Satisfying the grammar rules does not mean that the described program will accomplish its objective! Being syntactically correct only means that the description can be interpreted as a program according to the semantics. The process of deciding whether a program satisfies its objective is a more difficult problem than merely writing syntactically correct programs – the terms *verification* and *validation* are often used for this process.

The p86 assembly language, called p86ASM, is designed to describe programs for systems based on the p86 processor.

Character Set: The syntax of any language is based on a set of characters that can appear in statements. p86ASM permits the following character set (it consists of the displayable 7-bit ASCII character set):

blank space
; : ' " { } [ ] \ | = + – < > , . / ? ~ @ # $ %
^ & * ( ) A .. Z a .. z 0 .. 9

Statement: A statement is a (possibly empty) sequence of characters appearing on a **single** line of text such that the grammar rules are satisfied. This is different from many high-level languages, which may allow a statement to span multiple lines of text. p86ASM requires that an end-of-line indicator, such as carriage return/line feed, terminates each line of text. Text editors append end-of-line indicators automatically, so programmers do not have to worry about including them. Using the end-of-line as a statement terminator is different from high-level languages like C++ which often use a character like ";" as an explicit statement terminator. Blank statements (i.e. lines containing no displayable characters except possibly blank spaces) have no semantic meaning in the language and can be ignored.

When forming statements, certain characters are used as **separators** that break the statement into syntactically distinct parts called **tokens** (or **words**). For example, a blank space is often used as a separator. A separator is used to provide a break between two tokens, but is not considered to be part of either token. The grammar of a language may permit different characters to act as separators under different conditions. Separators allow a statement to be **parsed** into tokens (*parsing* breaks a statement into its constituent parts – in this case, the parts are tokens/words).

The grammar rules specify: the permitted forms of tokens, which characters are separators under various conditions, how syntactically correct statements can be formed by a sequence of tokens and separators, and how a syntactically correct program can be formed by sequence of statements.

Blank Space: Blank spaces are used to help to make programs easier for people to read. Blank spaces can be ignored as being meaningless, except:
1. if the spaces appear in a string constant (discussed later), or
2. where the grammar requires at least one space to separate tokens in a statement.

Comments: p86ASM assumes that all comments are intended to help humans to understand programs; and therefore, comments have no semantic meaning in p86ASM. Comments begin with ";" (semi-colon) and continue to the end of the line.

A Software Engineering Quandary: If comments don't contribute anything to the execution of a program – why bother with comments? This quandary often leads "closet hackers" to focus on executable statements while spewing documentation-free programs. Professional developers acknowledge that software development involves more than just writing statements that execute!

Symbolic Names: Symbolic names are sequences of characters that **begin with an alphabetic character** (A .. Z, or a .. z), and then contain only characters from the following:

A .. Z a .. z 0 .. 9 _

(the list consists of alphanumeric characters plus the "_" underscore character – the list does not include the blank space!)

p86ASM is case insensitive. (p86ASM does not differentiate between upper and lower case.) Therefore, the following names are all considered to be equivalent: mov, Mov MoV, MOv, MOV, mOV, moV, mOv)

Symbolic names are classified as either **reserved words**, or **user-defined symbols**. User-defined symbols are either labels or constants (both are discussed later).

Reserved Words: Reserved words have a predefined meaning (semantics). The meaning of reserved words cannot be redefined in programs. The following words are reserved in p86ASM:

Register Names:	AL, AH, BL, BH, CL, CH, DL, DH AX, BX, CX, DX, BP, SI, DI, SP (IP is not reserved!?)
Instruction Mnemonics:	see Instruction Reference for complete list (e.g. MOV, DIV, AND, JE, etc.)
Assembler Directives:	END, EQU, ORG

Instruction Statements: Instruction statements are interpreted as operations to be performed by the p86 processor (see the Instruction Reference for more details on the syntax and semantics of instructions). The operation to be performed is represented syntactically by the instruction mnemonic (e.g. MOV, JMP). When an instruction statement is encoded as a binary value, the mnemonic is converted into a series of bits referred to as the **opcode** (short for **operation code**). Instructions may have explicit and/or implicit operands. **Explicit operands** must be written as part of an instruction statement, and the specified operand addressing information gets assembled into the binary encoding of the instruction. Addressing modes allow the identification of the state variables (registers, memory, I/O ports) to be used as explicit operands. In some instructions, the p86 processor always uses specific state variables as **implicit operands**. There is no need to specify implicit operands in instruction statements, since the programmer has no control over the use of these operands. By choosing an instruction that uses implicit operands, the programmer must accept that the implicit operands will be used (in fact, it is usually the case that the instruction is chosen because it uses the implicit operands!). Similarly, there is no need to assemble implicit operand information into the binary encoding of instructions – implicit operands are used as part of the way the execution of the instruction was built. For example, the 16-bit DIV instruction:

```
        DIV    CX
```

divides the 32-bit quantity comprised of DX:AX (the 16-bit value from DX concatenated by the 16-bit value in AX) by the 16-bit contents of CX. In this case, the dividend

(DX:AX) is an implicit operand, and the divisor (CX) is an explicit operand. The binary value that represents the assembled instruction consists of bits used to encode the DIV opcode and the explicit divisor operand, but no bits are used to encode the implicit dividend operand.

Instruction statements must have an instruction mnemonic. The mnemonic may be followed by zero, one, or two explicit operands, depending on the instruction (see Instruction Reference for details). When operands are present, they must appear in the same statement (i.e. on the same line) as the instruction mnemonic, and the first operand must be separated from the mnemonic by at least one blank space character. When a second operand is present, it must be separated from the first operand by a "," (comma).

The syntax of operands depends on the **addressing modes** involved (see Instruction Reference for the addressing modes permitted for specific instructions):

- **register** mode: specify the operand using the **register name**
    - example:
      ```
      DIV   CX          ; CX is specified using register mode
      ```

- **immediate** mode: specify the (source) operand as a **constant** – constants are discussed below
    - example:
      ```
      MOV    AX, 1      ; 1 specified using immediate mode
      ```

- **direct** mode: specify the address of the operand as a constant, and enclose: **[address]**
    - example:
      ```
      X:  DW                ; declare 16-bit variable X
                            ;     memory declarations
          . . .             ;     are discussed below
          ADD    DX, [X]    ; variable X is specified using direct mode
      ```

    - **register indirect** mode: specify one of the registers: BX, BP, SI, DI (for memory operands) or DX (for I/O operands), and enclose **[register]**
        - example:
          ```
          X:  DW                ; declare 16-bit variable X
              . . .
              MOV    BX, X      ; BX := address of variable X
              ADD    DX, [BX]   ; BX is specified using register indirect mode
                                  (execution will access variable X)
          ```

    - **indexed** (indirect) mode: register indirect + a constant – for memory operands only – specify register (one of BX, BP, SI, DI) + constant, and enclose **[register + constant]**
        - example:
          ```
          X:  DW    ; suppose X is start of array of 16-bit variables
              . . .
              MOV    BX, X        ; BX := address of variable X
              ADD    DX, [BX + 4]   ; source operand is specified using
                                    ;   indexed mode
                                  (execution will access 3rd word of array X)
          ```

- **based** (indirect) mode: register indirect using 2 registers – for memory operands only – specify one of BX or BP + one of SI or DI, and enclose **[register + register] NOTE**: [SI + DI] and [BX + BP] are **NOT** legal register combinations
  - example:

```
X:  DW     ; suppose X is start of array of 16-bit variables
    . . .
    MOV   BX, X        ; BX := address of variable X
    MOV   SI, 6
    ADD   DX, [BX + SI]   ; source operand is specified
                          ;   using based mode
                  (execution will access 4th word of array X)
```

- **based-indexed** mode: based + index constant – for memory operands only – specify one of BX or BP + one of SI or DI + constant, and enclose **[register + register + constant]**
- example:

```
X:  DW     ; suppose X is start of array of 16-bit variables
    . . .
    MOV   BX, X        ; BX := address of variable X
    MOV   SI, 5
    ADD    DX, [BX + SI+1]   ; source operand is specified
                             ;    using based mode
             (execution will access 4th word of array X)
```

Instructions must have **compatible operand sizes**. When at least one register is specified as an operand in an instruction, then the instruction's operand size is determined by the size of the specified register. If more then one register is specified, then the size of the registers must be the same. Some examples:

```
    MOV   AX , 3
```

the destination is a 16-bit register, so the assembler converts the constant 3 into the 16-bit value 0003H.

```
    MOV   AL , 1234H
```

the destination is an 8-bit register, so the assembler converts the constant 1234H into an 8-bit value by keeping only the least significant byte, i.e. 1234H is shortened to 34H.

```
    MOV   BH , CL
```

both the source and destination registers are of the same size (8 bits).

When an instruction does not include a register as a specified operand, then there may be ambiguity in the operand size. Immediate source values (constants) can be expanded or shortened by the assembler as needed, and therefore, the expression of a constant value cannot be used to determine operand size. Furthermore, memory operands are specified using direct of indirect addressing modes. These modes involve specifying a single 16-bit address. If the operand is an 8-bit operand, then the specified address is interpreted as referring to a single byte of memory. If the operand is a 16-bit operand, then the specified address is interpreted as referring to two consecutive bytes of memory (the addressed byte and the next sequential byte – remember little endian!). Specifying a single address (by itself) is ambiguous as to whether it is referring to one or two bytes. Example:

```
        MOV   [ BX ] ,  1
```

in the instruction intended to move the 8-bit constant 1 to a byte, or the 16-bit constant 1 to a word?

The size of a memory operand can be stated unambiguously by adding **BYTE PTR** or **WORD PTR** to qualify whether the supplied address should be interpreted as referring to a byte or word of memory. Some examples:

```
        MOV   BYTE PTR [ BX ] , 0
```

the destination is clearly identified as an 8-bit memory location, so the assembler converts the constant 0 into an 8-bit value.

```
        MOV   WORD PTR [ 012ADH ] , 10
```

the destination is clearly identified as a 16-bit word, so the assembler converts the constant 10 in a 16-bit value (000AH). When the instruction is executed, the 16-bit value is stored little endian with the value 0AH being written (to the location) at address 012ADH and the value 00H being written at address 012AEH.

Memory Declarations: Recall that a program is a collection of instructions and data. Instruction statements are used to describe instructions, and memory declaration statements are used to describe data variables. A description of data must account for both the size (i.e. the amount of memory needed to store the data) and possibly an initial value. p86ASM allows two sizes of memory declarations:

DB	declares a byte (reserves one byte of memory)
DW	declares a 16-bit word (reserves two consecutive bytes of memory)

An initialisation value may be included as an operand in a memory declaration statement. If the operand is present, it must:

1. follow the size declaration on the same line,
2. be separated from the size declaration by at least one blank space, and
3. be stated as a constant (see below, in particular, note the case when the operand is given as a string constant).

A memory declaration statement is interpreted as reserving the specified number of bytes of memory. If an initialisation value is specified, the statement specifies the values that the reserved memory should be initialised to hold when the program is loaded. NOTE: the declaration causes the reserved memory to be initialised only once, and that occurs when the program is loaded for execution.

Labels: Labels are user-defined symbols that are used to represent constant (static) addresses symbolically (instead of stating binary values). As a program is being written, it is often necessary for instructions to access data stored in memory (data transfer and data manipulation instructions), or to direct execution control to non-sequential locations (control flow instructions). The exact memory location of a variable or target address of a jump instruction depends on the position of assembled statements in the final program. This situation creates a dilemma for programmers – they must refer to addresses as constants in the instructions they write (e.g. for direct addressing purposes), but the exact addresses to use may not be known until the program description is converted into a program. The dilemma is resolved by using labels to identify memory locations symbolically (i.e. by a logical name rather than an exact address), without concern for exact addresses. The assembler (and possibly the linker and/or loader) converts the symbolic labels into explicit addresses when the exact addresses to be used are assigned. In addition to solving the dilemma above, the careful choice of user-defined labels can increase the (human) readability of a program.

Labels are used in two ways: definitions and references. A **label definition** consists of a user-defined symbol followed immediately by a ":" (colon). When present in a statement, a definition must appear as the first non-blank characters in the line. A definition may appear by itself as a complete statement. If a definition is followed in a statement by either an instruction or a memory declaration, then the definition must be separated from the remainder of the statement by at least one blank space.

A label definition is interpreted to mean the address of the next reserved memory location (i.e. the address of the first byte of the next instruction, or the address of the first byte of the next location reserved by a memory declaration). In the following example, the labels A, B and C are all interpreted to mean the address of the first byte of the MOV instruction (i.e. the labels all have the same meaning):

```
A:
B:
C:      MOV   AX, 1
```

A **label reference** occurs when the label (without the ":") appears as an operand in some other statement. A label reference is interpreted to mean the constant address value for which the label has been defined. For example, BINGO and B4 are labels:

```
BINGO:        DW     ; label definition – defines the address of a memory word
        . . .
        MOV   BX, BINGO   ; label reference – load BX with address of word
        MOV   CX, [BINGO] ; label reference – load CX with contents of word
B4:                  ; label definition – defines the address of JMP instruction
        JMP   B4          ; label reference – jump to target
```

NOTE: The **target operands of control flow instructions** are specified using a label reference. If necessary, p86ASM calculates the relative offset to the label when the instruction is assembled. This eliminates the need for programmers to calculate relative offsets. ☺

NOTE: Continuing with the above example, the instructions:

```
MOV   [ B4 ] , AX
JMP   BINGO
```

are both “legal”, but are not really useful (?) The MOV instruction is overwriting part of the jump instruction (at B4) with data – this is called “self-modifying code”. ☹ The JMP instruction will cause the processor to fetch the next instruction from the memory location containing data (at BINGO) – this is sometimes called “executing data”. ☹ Self-modifying code and executing data are unfortunate artefacts of the stored program concept (program and data stored in the same memory). Although the instructions are syntactically permitted and have semantic meaning, neither of these practices are supported by the simulator (the simulator assumes that any attempts to execute these sorts of instructions are an unplanned mistake and the simulator stops program execution).

Constants: A constant is a static value that may be stated as: a numeric constant, a string constant, a label reference, a user-defined symbolic constant, or a static expression. A static value is one that can be determined when the program is assembled (as opposed to a dynamic value, which depends on state variable values at a point in program execution).

Numeric constants may be given in decimal, hexadecimal or binary forms. All numeric constants must begin with a decimal digit (0..9). The default is decimal numeric constants. A decimal numeric constant is a word consisting of characters in the range 0..9. A binary numeric constant is a word consisting of characters 0 and 1, and ends with either “B” or “b”. A hexadecimal numeric constant is a word consisting of characters in the range 0..9, A..F and a..f, and ends with either “H” or “h”. Examples:

```
        1234   (decimal)
        0AH    (hexadecimal)
        1111B  (binary)
```

Only decimal constants may be stated as negative values by prefixing the constant with "-". Example:

```
        -1
```

NOTE: **Hexadecimal constants** must start with a digit in 0..9. If the first digit is less that A (hex), then this is not a problem; however, if the first digit is in A..F, then the word representing the constant might be confused with a label, mnemonic, or reserved word. Therefore, hexadecimal constants that start with a digit in A..F should have a leading 0 (zero) placed in front of the first digit (adding a leading 0 does not change the value). Instead of writing: B2H, the constant must be written 0B2H. To show the ambiguity of leaving off the leading zero, consider the following code fragment:

```
        MOV  CL, CH  ; what does the programmer mean?  CH → hex constant or register?
  BAH:               ; a legal label definition
        MOV  BX, BAH ; what does the programmer mean? BAH → hex constant or address?
```

String constants are interpreted as sequences of 7-bit ASCII encoded bytes. A string constant starts and ends with “ ’ ” (single quote), and the constant is interpreted as the sequence of ASCII encoded bytes represented by the sequence of characters appearing between the quotes. Any blank spaces appearing in the string are part of the string. For example: ’Hi Mom’ is a 6-character string representing a sequence of 6 bytes of ASCII-encoded information. A single character is specified as a one-character string, e.g. ’A’. The use of the displayable character “ ’ ” (single quote) to delimit string constants creates a problem when it is desired to include “ ’ ” (single quote) as one of the data characters in the string constant. For example, the second “ ’ ” in:

’the computer’s memory’

is intended to be part of the string constant, not the end of the constant. To allow the single quote delimiter to appear as data, the language assumes that every appearance of two successive “ ’ ” characters in a string constant should be interpreted as a single “ ’ ” data character. Therefore, the above example would be written:

’the computer’’s memory’

String constants in memory declarations: String constants can be used as the initialisation value in a **DB** memory declaration statement. The statement is interpreted as reserving enough bytes of memory to hold the entire constant, and the bytes are initialised to hold the ASCII encoded values in the order they appear in the string. For example:

```
DB    ’Hi Mom’
```

would reserved 6 bytes of memory and would initialise the bytes to the hexadecimal values:

```
48 69 20 4D 6F 6D
```

The above declaration is equivalent to the following, but the following is more awkward to use:

```
DB    ’H’
DB    ’i’
DB    ’ ’
DB    ’M’
DB    ’o’
DB    ’m’
```

The following declaration reserves one byte of memory and initialises it to 27H (representing the “’” single quote character):

```
DB    ’’’’
```

NOTE: The use of *string* in this p86ASM discussion is NOT the same as the use of ***string*** in **high-level language** discussions. In high-level languages, strings usually include a terminating character in addition to the ASCII characters. For example, **C++ uses null-terminated strings**, in which the ASCII-encoded bytes representing the characters in the string are followed by the null-terminator byte 00H. This is why the character arrays used to hold C++ strings must be one character (byte) larger than the character count in the body of the string. For example, a null-terminated string can be implemented easily in p86ASM:

```
DB    ’Hi Mom’          ; char’s in body of string, followed by .....
DB    00H               ; null-terminator byte
```

Expressions: [under construction – not currently supported] arithmetic expressions

Location Counter: [under construction – not currently supported]

Assembler Directives: Assembler directives are statements that influence how the assembler converts the program description into an OBJ file, and may instruct the assembler to insert information into the OBJ file for use by tools that come later in the development process (e.g. the linker and loader).

END

The **END** directive tells the assembler to stop reading lines from the source (.ASM) file. Once the END statement is encountered, any remaining lines in the source file are ignored. The END statement has one operand. The operand must be a label reference, and the label reference is interpreted as specifying the program's start address (i.e. the address of the intended first instruction to be executed). For example:

```
            . . .               ; some statements here, possibly including instructions
StartHere:  MOV ...     ; instruction intended to be first instruction executed
            . . .               ; more statements here, possibly including instructions
            END     StartHere   ; end of statements & declaration of execution start address
```

If no start address is specified in the END directive, then the assembler issues an error.

Symbolic Constants: EQU

The use of numeric constants can sometimes decrease the intuitive readability of a program, and in these cases, the constants are often referred to as "magic numbers". For example, I/O addresses like the display port address 04E9H are required to make programs work, but a reader may find the number awkward – the number does not say anything about the purpose of the port being addressed. The EQU directive allows symbolic constants to be defined. A symbolic constant is a programmer-defined name that is used throughout the program to represent a constant value. During assembly, the assembler replaces every occurrence of the name with the associated constant.

For example, the display port address might be defined symbolically as:

```
            DisplayPort     EQU     04E9H
```

The symbolic name "DisplayPort" can then be used to improve the readability of a program:

```
                        MOV     DX, DisplayPort
```

The general syntax of the EQU statement is:

```
symbolic-name   EQU    numeric_constant
```

where symbolic-name is a symbolic name (see previous definition). Note that there is no ":" in the statement – the statement is not defining a label (see previous definition). The current version of the assembler only supports numeric constants as EQU operands.

The use of symbolic constants has an important software engineering advantage in addition to improving readability. Symbolic names often help to simplify program modification. If a program is written in terms of particular magic numbers, then the program is hard-coded to include the numbers at every point where they are relevant. Now suppose that the program is to be modified in a way that some of the magic numbers change. If symbolic constants are used, then only the relevant EQU statements need to be modified. If symbolic constants are not used, then every point at which the magic numbers are used must be found and modified.

ORG

The **ORG** directive instructs the assembler to change the location at which memory is being allocated (i.e. change the location counter value). The ORG statement has one operand, and that operand must be a numeric constant representing the desired address at which assembly should continue. An ORG statement may only advance the location counter value, it cannot change the location counter to a value less than the value at the point where the assembler encounters the ORG statement.

If the specified address is less than the current location counter value, then the assembler issues an error.

The Assembly Process

The assembler converts ASM files into OBJ files. The results of the conversion, including any error descriptions, are written to LST files. The following discussion of the assembly process has some similarities to the execution of programs, but please remember, execution happens when the program has been loaded (by a loader) into memory and is executed by the processor. The assembly of the ASM file must happen *before* the program can be executed!

The purpose of the OBJ file is to describe the program as a binary image to be loaded into memory for execution (with the possible exception of details that can only be added by other tools like the linker or loader – there are no details of this type in this course, but these sorts of details will arise in 203!). Since a program consists of instructions and data (memory variables), the assembler must decide which memory locations will be used for particular instructions and data. The decision to use a particular memory location to hold a particular program artefact is referred to as **allocating the memory location** to that artefact. In addition to allocating memory, the assembler must determine the binary values that should be stored in the locations. Once each instruction and data declaration has been allocated sufficient memory locations, and the contents of each allocated location have been determined, the assembler writes these decisions to the OBJ file. The OBJ file records the address and contents of each location allocated to the program. The intention is that the program can be loaded easily into memory by reading the OBJ file and storing the specified values in the specified locations. The resulting description is often called the binary image because there is no need to add additional information prior to loading and executing the program.

Assembly is carried out in two passes through the ASM file. A **pass** consists of starting at the beginning of the file and processing all of the lines sequentially until an END statement is encountered. During the first pass, the assembler checks syntax, allocates memory, and builds the symbol table. The second pass is used to construct the binary image for each statement. Information is written to the OBJ file only if both passes are completed successfully. The results of the process are written to the LST file regardless of how many errors are encountered.

1st Pass

The first pass is used to check the syntax of the program statements, to allocate the memory needed for each statement, and to build the symbol table. Syntax checking is a prerequisite first step, since a syntactically incorrect statement is meaningless to the assembler. The allocation of memory to each statement is fundamental to the assembly process and determines the locations in memory where the image of each statement should (eventually) be loaded for execution. The symbol table records the values of user-defined symbols. This information must be tabulated during the first pass so that the values of all symbols are known before starting the second pass. The first pass does not involve the generation of any binary images – this is done during the second pass.

The first pass is accomplished by processing each line of the ASM file sequentially. After reading a line, the assembler then answers the following questions about the statement present in the line. Although the questions are presented separately below, assemblers tend to integrate their resolution, and failure at any point usually results in an error and stops further analysis of the statement:

- Is it syntactically correct? This includes parsing the statement into parts (like operation and operands) and deciding whether the statement satisfies the grammar rules.
- Is the statement a memory declaration, instruction, or directive? Any other statement must be either a blank line, or a comment – both of which are ignored by the assembler. EQU and ORG directives are processed during the first pass (discussed

below). The END directive terminates each pass; however, the operand of the END directive is processed during the second pass (discussed below).

- Does the line include a symbol definition (label definition or EQU directive)? If yes, then the value of the symbol is determined and recorded in the symbol table (discussed below).
- How much (if any) memory will be needed for the image of the statement?
- Which memory locations will be used for the image? The location counter is used to manage the allocation of memory (discussed below).

The assembler assumes that **memory allocation begins at address 0000H** (i.e. the first byte to be allocated to a program artefact will be the byte at address 0000H). The assembler allocates memory sequentially to the statements as they are encountered in the ASM file. This means that sequential statements in the ASM file will be allocated to sequential memory locations. This is good, because the inherent operation of the execution cycle assumes that sequential instructions have been allocated to sequential memory locations.

The assembler uses the **location counter** to manage memory allocation. The location counter is a variable (local to the assembler program) that records the address of the next memory location to be allocated in the assembly process. When the assembler begins execution, it initialises the location counter to 0000H. When allocating a byte of memory for some purpose, the assembler uses the location counter value as the address of the byte of memory to be used, and then the location counter is incremented to reference the next sequential memory location. The location counter will be represented using “ **$** ”.

Suppose that the first line of a program is:

```
        DB      3
```

The line is intended to reserve one byte of memory for the program, and to initialise the byte to the value 3 when the program is loaded. Since this is the first statement in the program, $ = 0000H when the line is read during the first pass. The assembler determines that the statement is reserving one byte of memory (DB), so the byte at address 0000H (the current $ value) is allocated for that statement. The assembler then increments the location counter, i.e. $ = 0001H, so that the next byte allocated to the program will be at address 0001H. Note that the image associated with the value 3 is not generated during this pass (binary images are generated during the second pass!). Since there is no other information relevant to the first pass, the assembler then continues to the next line of the ASM file.

Continuing with the example, suppose that the second line of the program is:

```
        DW      10
```

This statement reserves a word (two bytes) of memory. At the time that the line was encountered, the location counter has the value 0001H (the byte at 0000H was allocated to the previous DB statement). The assembler determines that the image of the statement will require two bytes, so the assembler: allocates the byte at 0001H (the current $ value) as the first byte and then increments the location counter ($ = 0002H); then allocates the byte at 0002H (the current $ value) as the second byte and then increments the location counter ($ = 0003H). As with the DB statement, the generation of the image to be loaded into the allocated bytes is deferred to the second pass.

Suppose that the example continues with several lines (omitted) after which the location counter has the value 0016H (i.e. $ = 0016H). Now suppose that the next line is:

```
        MOV   AL, 'H'
```

The assembler determines that the encoding of this instruction (i.e. the binary image) will require 3 bytes of memory (instruction encoding will be discussed later). The assembler will allocate three bytes for the MOV instruction starting at the current location counter value (i.e. the bytes at 0016H, 0017H and 0018H) and will adjust the location counter to have the value 0019H. The binary image for the instruction will not be generated until the second pass.

The exception to sequential memory allocation occurs when an **ORG** directive is encountered. The ORG directive is used to force the assembler to continue allocating memory from a particular address. The statement has one operand, and that operand must be a numeric constant indicating the address at which allocation should continue. The assembler processes ORG statements as they are encountered during the first pass. When an ORG statement is encountered, the assembler assigns the location counter the value specified in the statement. This ensures that the next byte to be allocated will be at the specified address.

Statements may define symbolic labels and symbolic constants. These definitions associate values with user-defined symbolic names. During the first pass, the assembler determines the values to associate with the symbolic names, and records the associations in the symbol table. The **symbol table** is a data structure local to the assembler. The symbol table entries are *(symbol, value)* pairs. The information accumulated in the symbol table during the first pass is then used during the second pass to build binary images that involve references to the symbols.

Recall from earlier: Symbolic label definitions (*symbolic-name:*) are used to identify the address of the next byte of memory allocated after the definition. Once defined, the name can be used elsewhere in the program to refer to the address (**remember this**: the value of a label is an address!). Labels are used to identify the memory locations where variables are stored, and to identify the targets of control flow instructions. Label definitions are handled easily by the assembler. Since the location counter contains the address of the next memory location to be allocated, the assembler simply associates the current value of the location counter with the defined label.

For example, suppose $ = 000AH when the assembler encounters the line:

```
        start:
```

The statement consists only of the definition of the label "start". When processing the line during the first pass, the assembler would save the pair (start, 000AH) in the symbol table.

Symbolic constants are defined using EQU statements. During the first pass, the assembler evaluates the constant specified in the statement and saves the relevant information in the symbol table. For example, suppose the assembler encountered the line:

```
        display_port    EQU   04E9H
```

The assembler would process the line by saving the pair (display_port, 04E9H) in the symbol table.

Note that the processing of the symbolic definitions does not involve allocating memory! The value of the location counter is not changed by these definitions – the definitions result in additional information being written to the symbol table.

When an **END** statement is encountered, the assembler terminates the first pass through the ASM file. If there were no errors during the first pass, then all required memory locations have been allocated to statements, and the symbol table contains the values of all symbolic names. The assembler then uses the results of the first pass to build the binary image for each statement during the second pass.

2nd Pass

The second pass is used to build the binary image of the program. The assembler uses the information generated during the first pass to determine the binary value that should be contained in each memory location allocated to the program. The allocation of specific locations to particular statements was accomplished in the first pass. The assembler must now determine the initial values to load into the allocated memory variables, and the encodings of all instructions.

The initial value for a memory variable is an optional operand in a memory declaration statement. If an initialisation value is given, then the assembler must determine the binary representation of the value. The assembler will truncate values to fit the memory allocated to the variable. For example, suppose the following line was to be assembled:

```
        DB      0FF1H
```

The statement reserves one byte (8-bits), yet the initialisation value has 12 significant bits (i.e. the initialisation value is larger than the declared variable size). The assembler truncates the supplied initialisation value by keeping only the least significant bits needed to fill the allocated memory. In this example, the assembler would truncate the initialisation value (0FF1H) to 0F1H.

If an initialisation value is specified for a word variable, then the value is stored in little endian format in the two bytes allocated to the variable. For example, suppose the locations at address 1234H and 1235H have been allocated for the statement:

```
        DW     1
```

The statement would be assembled into two consecutive bytes:

```
        1234H  →  01H      ; stored little endian!
        1235H  →  00H
```

If no initialisation value is specified for a variable, then the programmer does not care what initial value the variable is given (presumes that the programmer will initialise the variable dynamically prior to using it in the program). The assembler is free to handle this case in any way it might find convenient. Some assembler have default initial values that are used (as is the case for the assembler in this course), others do not (we will see a case of this in 94.203!).

It is important to note that initialisation values are loaded into memory only when the program is loaded initially for execution. If a program modifies the value of the variable, then the initial value is lost and can be restored only if the program re-initialises the variable dynamically (i.e. during execution).

It is possible that a variable might be initialised to contain a value that has been defined symbolically. In such cases, the symbol value recorded in the symbol table is used as the initialisation vale. In the following example, "Lx." has been added to simplify references to the lines:

```
L1.    ArraySize      EQU  8
L2.    ArrayXLoc:     DW   ArrayXStart   ; start address of array X
L3.    ArrayXSize     DW   ArraySize     ; size of array X
L4.    ArrayXStart:   DW            ; first word of arrayX
                      . . .         ;   more words of arrayX
```

Consider the first pass: Assume that the location counter value is 1000H when L1 is encountered. The constant value associated with the symbol ArraySize is saved to the symbol table (ArraySize, 8), and since the statement does not reserve any memory, $ = 1000H after processing the statement. Processing the label definition in the next line (L2) results in (ArrayXLoc, 1000H) being saved in the symbol table. While processing the DW memory declaration, the location counter is adjusted to 1002H. Processing the label definition in the next line (L3) results in (ArrayXSize, 1002H) being saved in the symbol table. While processing the DW memory declaration, the location counter is adjusted to 1004H. Processing the label definition in the next line (L4) results in (ArrayXStart, 1004H) being saved in the symbol table. While processing the DW memory declaration, the location counter is adjusted to 1006H.

Now consider the second pass: L1 does not result in the generation of any binary image. The word allocated in L2 is specified to (initially) contain the value of the symbolic label ArrayXStart. The assembler retrieves the value of the symbol (1004H) from the symbol table, and the resulting binary image for the two reserved bytes would be 04H and 10H (little endian!). The word allocated in L3 is specified to (initially) contain the value of the symbolic constant ArraySize, and the assembler retrieves the value of the symbol (8) from the symbol table and expands the value to a 16-bit form. As a result, the binary image for the two reserved bytes word would be 08H and 00H (little endian!).

The second pass must also build the binary image of instructions. The details of instruction encoding will be discussed later in the course. At this point, it is sufficient to appreciate that the assembler constructs the image of each instruction using encoding tables and the information in the symbol table. Any time a symbol appears as a reference in an instruction operand, the value of the symbol is obtained from the symbol table (as was the case for resolving symbolic references in memory initialisation values).

Example: MOV DX, 04E9H
The encoding of the instruction is 4 bytes: C7 C2 E9 04 (hex values)
The first two bytes (C7 C2) identify the MOV instruction, and specify the source as a 16-bit immediate value and the destination as the DX register. The actual 16-bit immediate source value is encoded in the last two bytes (E9 04 – little endian!)

The generation of the image of some instructions requires the assembler to perform calculations. For example, conditional jump instructions require the assembler to calculate an 8-bit relative offset. Conditional jump instructions always encode as a 2-byte instruction with the first byte encoding the operation (jump condition) and the second byte encoding the relative (signed) offset to the target. The target of the jump is specified as a label reference and the assembler handles the calculation of the relative offset.

Consider the following example of assembled code:

0014	72 06	JC Got1
0016	C6 C0 30	MOV AL , '0'
0019	E9 03 00	JMP DUMP
001C		Got1:
001C	C6 C0 31	MOV AL , '1'
001F		DUMP:
001F	EE	OUT [DX] , AL

The first column represents the (hex) value of the location counter when the statement was encountered (and hence represents the address of the first location allocated to the statement). The second column represents the binary image (hex) of each encoded instruction as consecutive memory values. The third column represents the program text that was assembled (minus the preceding lines that resulted in the location counter reaching the value 0014). To determine the relative offset to the target of the JC instruction, the assembler calculates the address of the target (Got1, address = 001C) minus the address of the byte after the JC instruction (0016) – i.e. 001C – 0016 – to arrive at the offset value 06 that has been encoded as the second byte of the instruction. Similarly the 16-bit offset assembled into the JMP instruction is 0003 (stored little endian in the instruction!) – the offset was obtained by calculating the target address (DUMP, address = 001F) minus the address of the byte after the JMP instruction (001C).

Make sure you understand the relationship between the calculation of the offset by the assembler, and the way that jump instructions are executed by the processor.