Assembly Language Programming

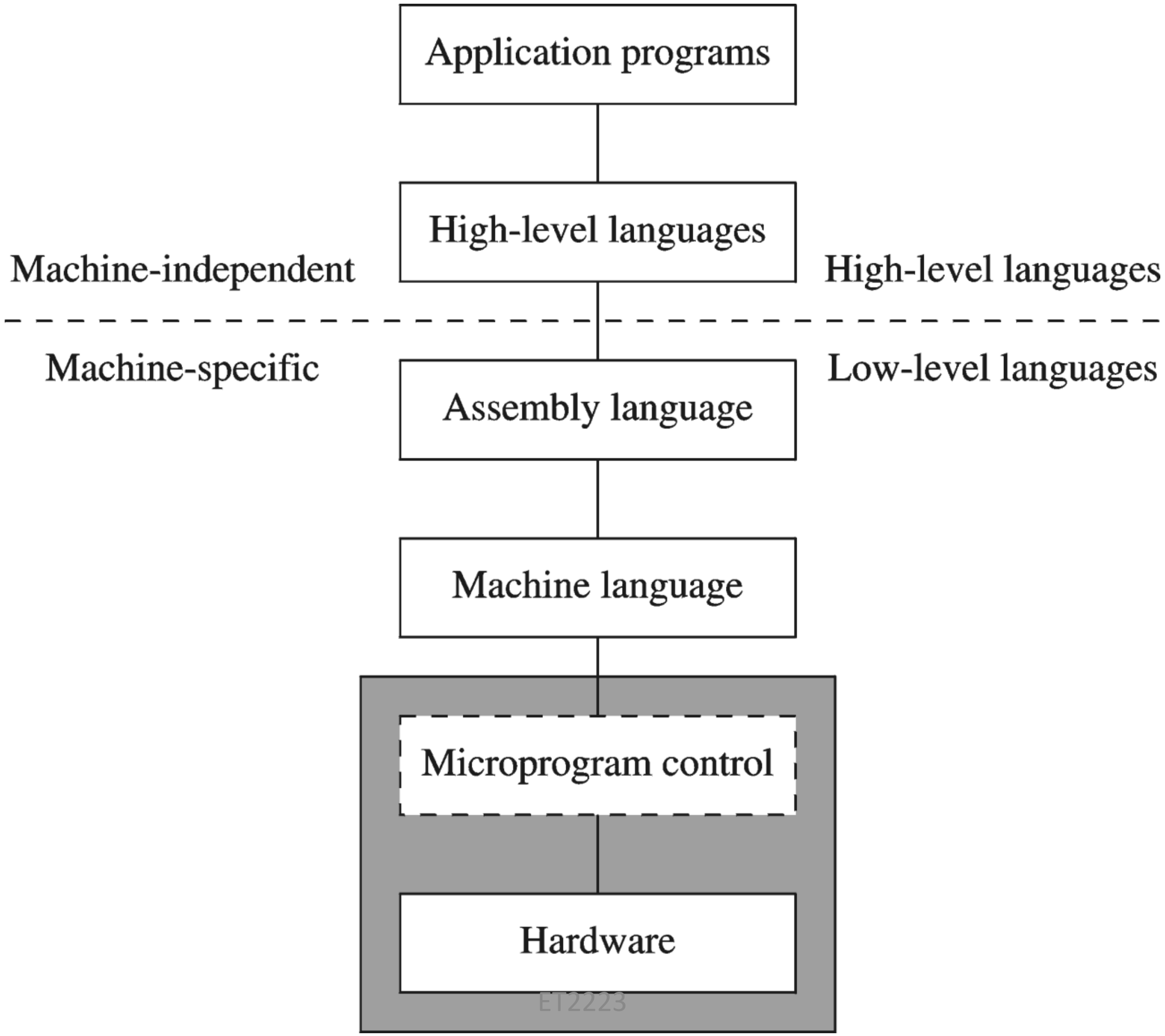
ET2223 Microprocessors, Microcontrollers, and Embedded Systems

Partially based on
Computer Organization & Assembly Language Programming by Dr Adnan Gutub
Assembly Language for Intel-Based Computers by Dr. Kip Irvine
Introduction to Computing Systems: From Bits and Gates to C and Beyond by Y. Patt and S. Patel

Some Important Questions to Ask

- What is Assembly Language?
- Why Learn Assembly Language?
- What is Machine Language?
- How is Assembly related to Machine Language?
- What is an Assembler?
- How is Assembly related to High-Level Language?
- Is Assembly Language portable?

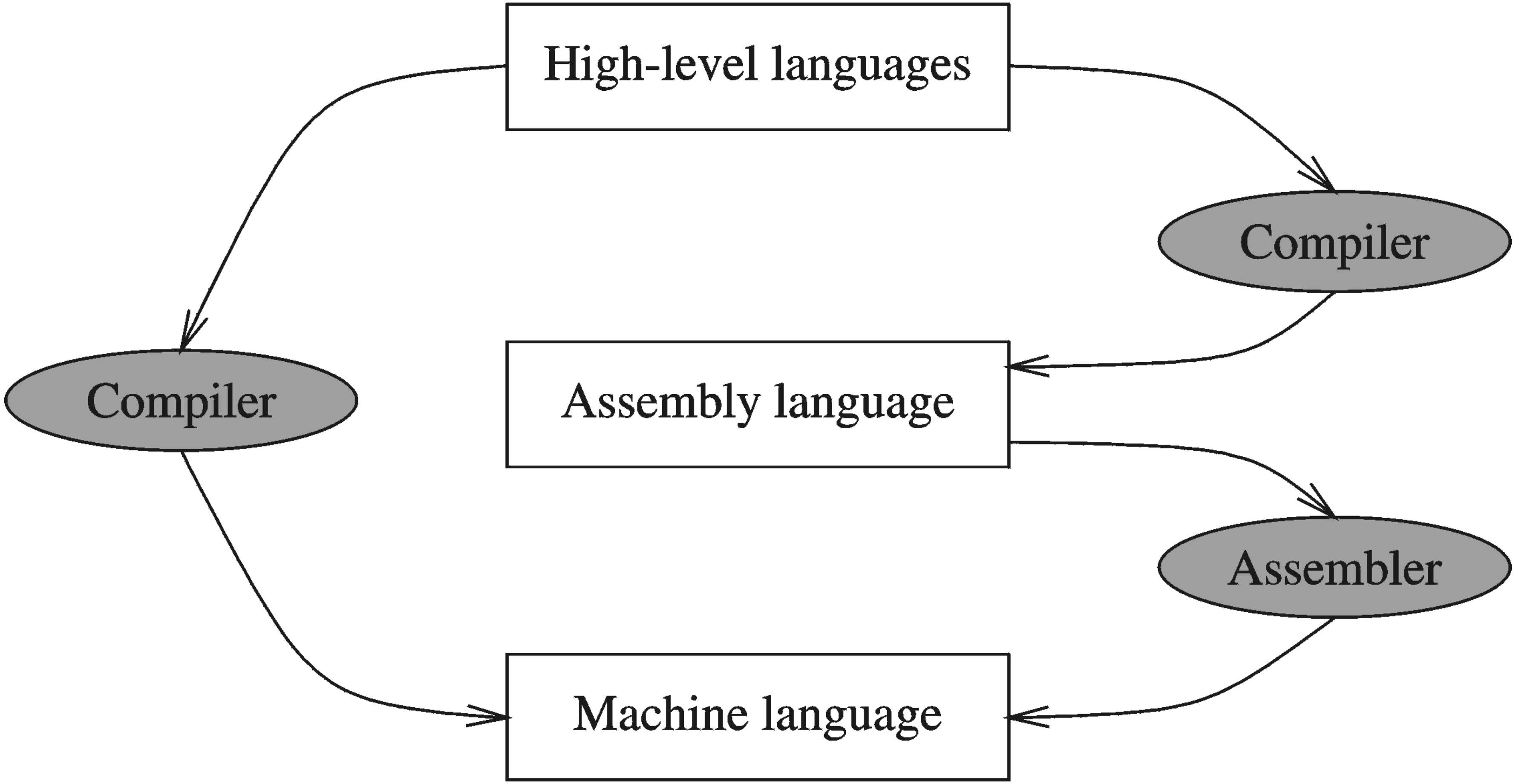
A Hierarchy of Languages



Assembly and Machine Language

- Machine language
  - Native to a processor: executed directly by hardware
  - Instructions consist of binary code: 1s and 0s
- Assembly language
  - A programming language that uses symbolic names to represent operations, registers and memory locations.
  - Slightly higher-level language
  - Readability of instructions is better than machine language
  - One-to-one correspondence with machine language instructions
- Assemblers translate assembly to machine code
- Compilers translate high-level programs to machine code
  - Either directly, or
  - Indirectly via an assembler

Compiler and Assembler



Instructions and Machine Language

- Each command of a program is called an instruction (it instructs the computer what to do).
- Computers only deal with binary data, hence the instructions must be in binary format (0s and 1s) .
- The set of all instructions (in binary form) makes up the computer's machine language.
- This is also referred to as the instruction set.

Instruction Fields

- Machine language instructions usually are made up of several fields. Each field specifies different information for the computer. The major two fields are:
- Opcode field which stands for operation code and it specifies the particular operation that is to be performed.
  - Each operation has its unique opcode.
- Operands fields which specify where to get the source and destination operands for the operation specified by the opcode.
  - The source/destination of operands can be a constant, the memory or one of the general-purpose registers.

Assembly vs. Machine Code

Instruction Address	Machine Code	Assembly Instruction
0005	B8 0001	MOV AX, 1
0008	B8 0002	MOV AX, 2
000B	B8 0003	MOV AX, 3
000E	B8 0004	MOV AX, 4
0011	BB 0001	MOV BX, 1
0014	B9 0001	MOV CX, 1
0017	BA 0001	MOV DX, 1
001A	8B C3	MOV AX, BX
001C	8B C1	MOV AX, CX
001E	8B C2	MOV AX, DX
0020	83 C0 01	ADD AX, 1
0023	83 C0 02	ADD AX, 2
0026	03 C3	ADD AX, BX
0028	03 C1	ADD AX, CX
002A	03 06 0000	ADD AX, i
002E	83 E8 01	SUB AX, 1
0031	2B C3	SUB AX, BX
0033	05 1234	ADD AX, 1234h

Translating Languages

English: D is assigned the sum of A times B plus 10.

High-Level Language: D = A * B + 10

A statement in a high-level language is translated typically into several machine-level instructions

Intel Assembly Language:

```
mov   eax, A
mul   B
add   eax, 10
mov   D, eax
```

Intel Machine Language:

```
A1 00404000
F7 25 00404004
83 C0 0A
A3 00404008
```

Mapping Between Assembly Language and HLL

- Translating HLL programs to machine language programs is not a one-to-one mapping
- A HLL instruction (usually called a statement) will be translated to one or more machine language instructions

Mapping between some C instructions and 8086 assembly language

Instruction Class	C	Assembly Language
Data Movement	a = 5	MOV a, 5

Arithmetic/Logic	b = a + 5	MOV ax, a
		ADD ax, 5
		MOV b, ax

Control Flow	goto LBL	JMP LBL

Example

- I = J + K

  Four-address format

  ```
  ADD  J, K, I, NEXT    ; I = J + K
                        ; next instruction in location NEXT
  ```

  Three-address format

  ```
  ADD  J, K, I          ; I = J + K
                        ; next instruction in PC
  ```

  Two-address format

  ```
  MOVE  J, I            ; I = J
  ADD   K, I            ; I = K + I
  ```

Example

- I = J + K

  One-address format

  ```
  LOAD   J          ; AC = J
  ADD    K          ; AC = J + K
  STORE  I          ; I = AC
  ```

  Zero-address format, postfix: I = JK+

  ```
  LOAD   J          ; push J onto stack
  LOAD   K          ; push K onto stack
  ADD               ; pop and add J and K, result on top
  STORE I           ; pop stack top to I
  ```

Advantages of High-Level Languages

- Program development is faster
  - High-level statements: fewer instructions to code
- Program maintenance is easier
  - For the same above reasons
- Programs are portable
  - Contain few machine-dependent details
    - Can be used with little or no modifications on different machines
  - Compiler translates to the target machine language
  - However, Assembly language programs are not portable

Why Learn Assembly Language?

- Accessibility to system hardware
  - Assembly Language is useful for implementing system software
  - Also useful for small embedded system applications
- Space and Time efficiency
  - Understanding sources of program inefficiency
  - Tuning program performance
  - Writing compact code
- Writing assembly programs gives the computer designer the needed deep understanding of the instruction set and how to design one
- To be able to write compilers for HLLs, we need to be expert with the machine language. Assembly programming provides this experience

Assembly vs. High-Level Languages

Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

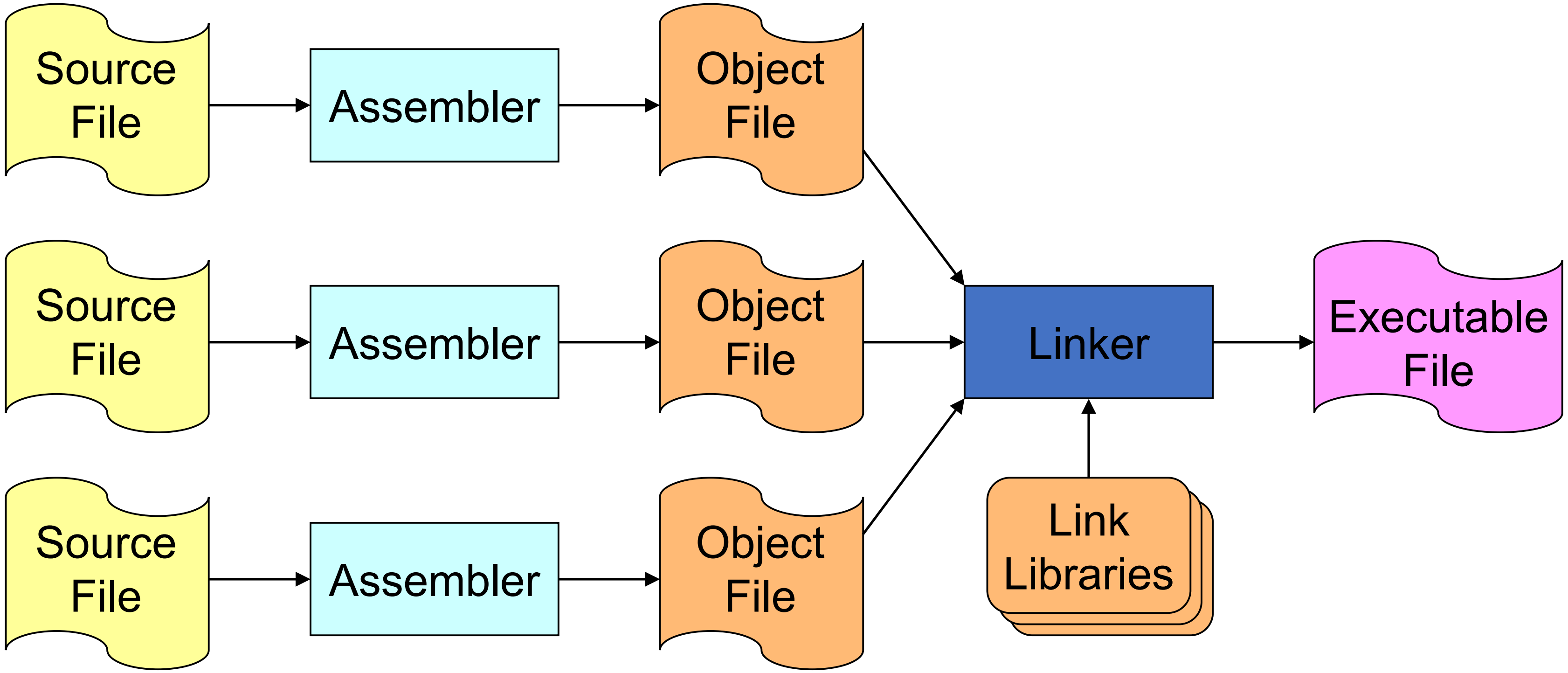
Assembler

- Software tools are needed for editing, assembling, linking, and debugging assembly language programs
- An assembler is a program that converts source-code programs written in assembly language into object files in machine language
- Popular assemblers have emerged over the years for the Intel family of processors. These include …
  - TASM (Turbo Assembler from Borland)
  - NASM (Netwide Assembler for both Windows and Linux), and
  - GNU assembler distributed by the free software foundation

Linker and Link Libraries

- You need a linker program to produce executable files
- It combines your program's object file created by the assembler with other object files and link libraries, and produces a single executable program
- LINK32.EXE is the linker program provided with the MASM distribution for linking 32-bit programs
- We will also use a link library for input and output
- Called Irvine32.lib developed by Kip Irvine
  - Works in Win32 console mode under MS-Windows

Assemble and Link Process



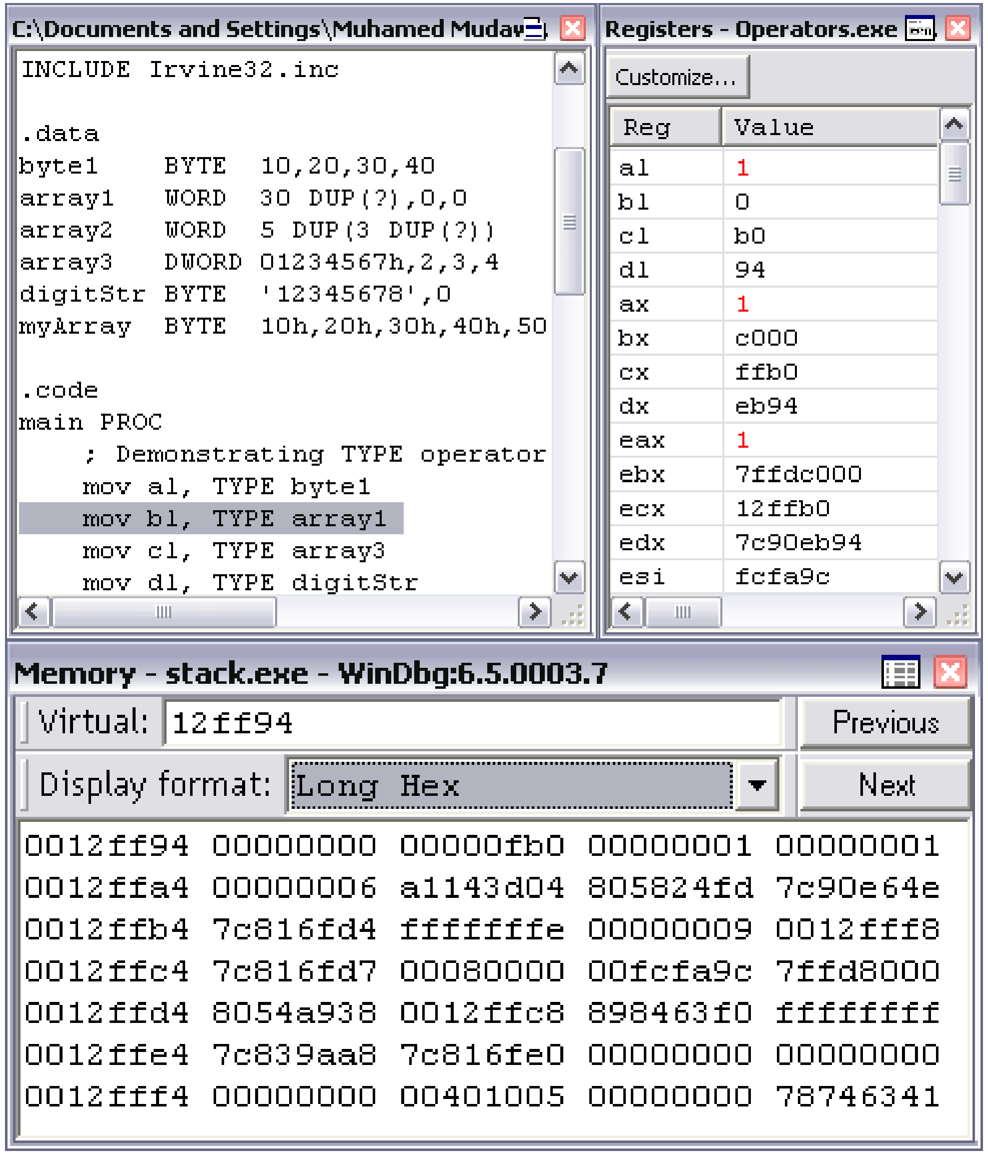
A project may consist of multiple source files

Assembler translates each source file separately into an object file

Linker links all object files together with link libraries

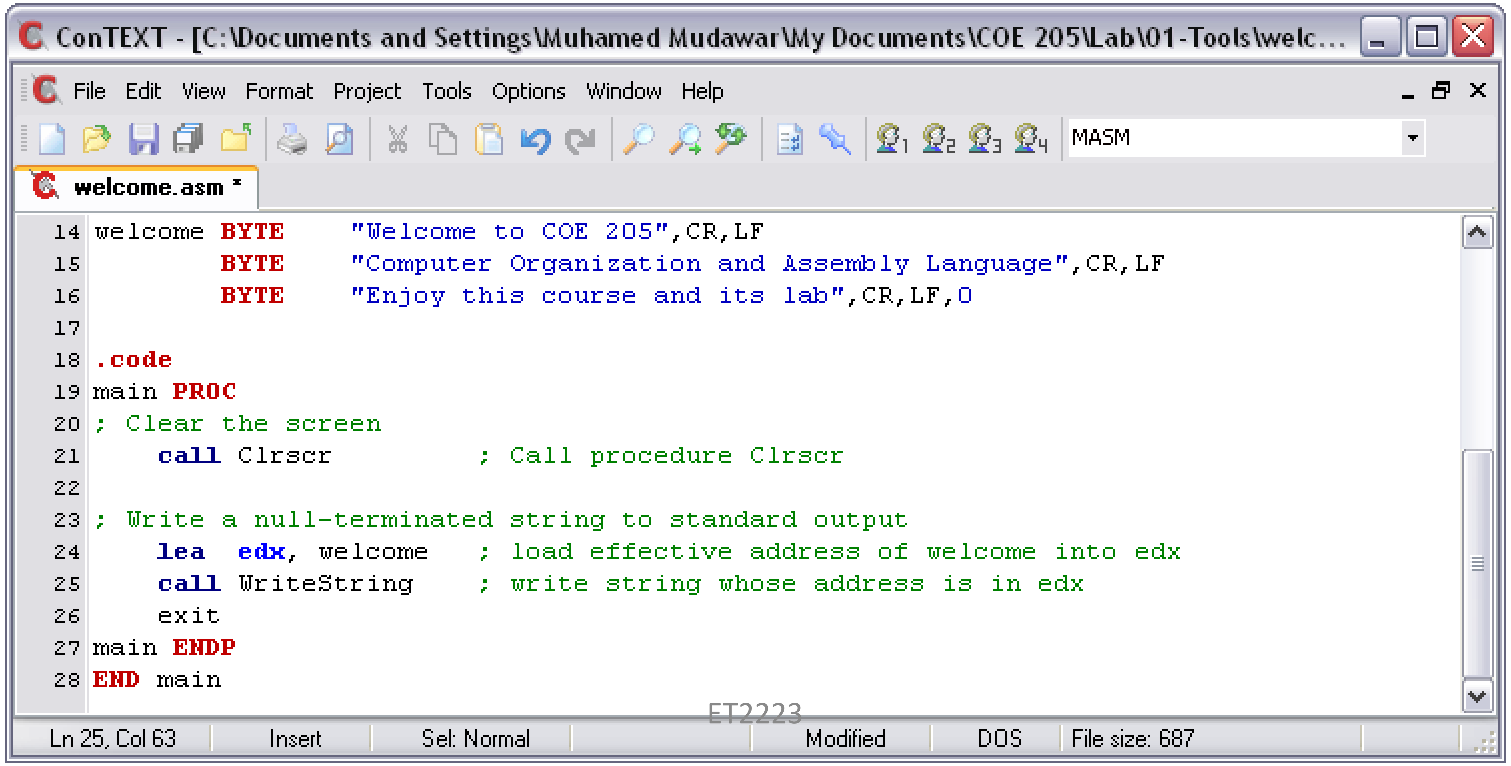
Debugger

- Allows you to trace the execution of a program
- Allows you to view code, memory, registers, etc.
- Example: 32-bit Windows debugger

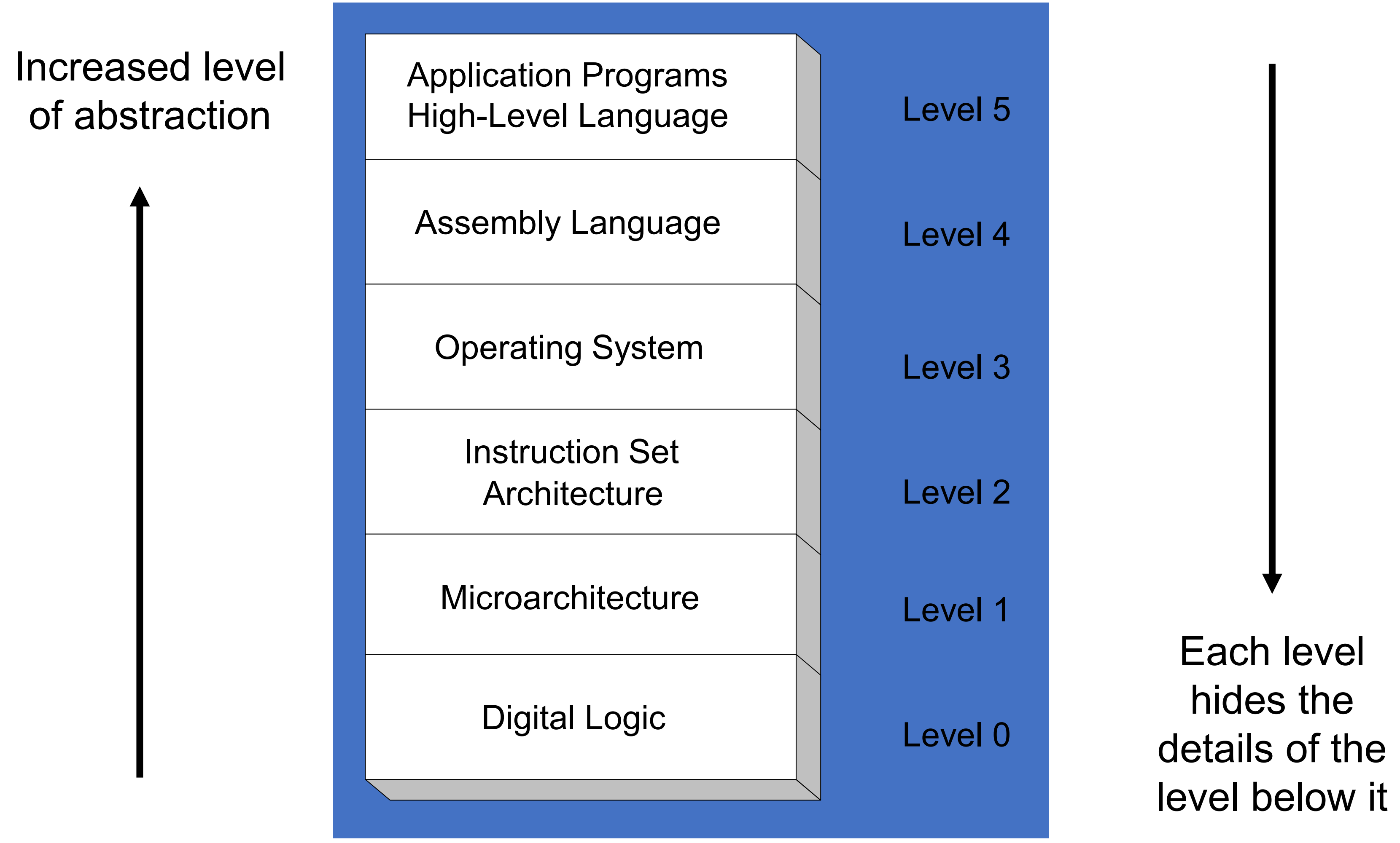


Editor

- Allows you to create assembly language source files
- Some editors provide syntax highlighting features and can be customized as a programming environment



Programmer's View of a Computer System



Programmer's View of a Computer System

- Application Programs (Level 5)
  - Written in high-level programming languages
  - Such as Java, C++, Pascal, Visual Basic . . .
  - Programs compile into assembly language level (Level 4)
- Assembly Language (Level 4)
  - Instruction mnemonics are used
  - Have one-to-one correspondence to machine language
  - Calls functions written at the operating system level (Level 3)
  - Programs are translated into machine language (Level 2)
- Operating System (Level 3)
  - Provides services to level 4 and 5 programs
  - Translated to run at the machine instruction level (Level 2)

Programmer’s View of a Computer System

- Instruction Set Architecture (Level 2)
  - Specifies how a processor functions
  - Machine instructions, registers, and memory are exposed
  - Machine language is executed by Level 1 (microarchitecture)
- Microarchitecture (Level 1)
  - Controls the execution of machine instructions (Level 2)
  - Implemented by digital logic (Level 0)
- Digital Logic (Level 0)
  - Implements the microarchitecture
  - Uses digital logic gates
  - Logic gates are implemented using transistors

Assembly Language

Human-Readable Machine Language

- Computers like ones and zeros…

```
0001110010000110
```

- Humans like symbols…

```
ADD R6,R2,R6   ; increment index reg.
```

- Assembler is a program that turns symbols into machine instructions.
  - ISA-specific:
    close correspondence between symbols and instruction set
    - mnemonics for opcodes
    - labels for memory locations
  - additional operations for allocating storage and initializing data

An Assembly Language Program

```
• ;
• ; Program to multiply a number by the constant 6
• ;
•       .ORIG  x3050
•       LD     R1, SIX
•       LD     R2, NUMBER
•       AND    R3, R3, #0    ; Clear R3.  It will
•                            ; contain the product.
• ; The inner loop
• ;
• AGAIN ADD    R3, R3, R2
•       ADD    R1, R1, #-1   ; R1 keeps track of
•       BRp    AGAIN         ; the iteration.
• ;
•       HALT
• ;
• NUMBER.BLKW  1
• SIX   .FILL  x0006
• ;
•       .END
```

LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assember directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with “;”) are also ignored.
- An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```

optional *mandatory*

Opcodes and Operands

- Opcodes
  - reserved symbols that correspond to LC-3 instructions
  - listed in Appendix A
    - ex: `ADD`, `AND`, `LD`, `LDR`, …
- Operands
  - registers -- specified by Rn, where n is the register number
  - numbers -- indicated by # (decimal) or x (hex)
  - label -- symbolic name of memory location
  - separated by comma
  - number, order, and type correspond to instruction format
    - ex:

```
ADD  R1,R1,R3
ADD  R1,R1,#3
LD   R6,NUMBER
BRz  LOOP
```

Types of Opcodes

- Arithmetic, logical
  - add, sub, mult
  - and, or
  - Cmp
- Memory load/store
  - ld, st
- Control transfer
  - jmp
  - bne
- Complex
  - movs

Operands

- Each operand taken from a particular addressing mode:
- Examples:

Register	add r1, r2, r3
Immediate	add r1, r2, 10
Indirect	mov r1, (r2)
Offset	mov r1, 10(r3)
PC Relative	beq 100

- Reflect processor data pathways

Labels and Comments

- Label
  - placed at the beginning of the line
  - assigns a symbolic name to the address corresponding to line
    - ex:

```
LOOP   ADD  R1,R1,#-1
       BRp  LOOP
```

- Comment
  - anything after a semicolon is a comment
  - ignored by assembler
  - used by humans to document/understand programs
  - tips for useful comments:
    - avoid restating the obvious, as “decrement R1”
    - provide additional insight, as in “accumulate product in R6”
    - use comments to separate pieces of program

Assembler Directives

- Pseudo-operations
  - do not refer to operations executed by program
  - used by assembler
  - look like instruction, but “opcode” starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Trap Codes

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

Code	*Equivalent*	*Description*
`HALT`	`TRAP x25`	**Halt execution and print message to console.**
`IN`	`TRAP x23`	**Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].**
`OUT`	`TRAP x21`	**Write one character (in R0[7:0]) to console.**
`GETC`	`TRAP x20`	**Read one character from keyboard. Character stored in R0[7:0].**
`PUTS`	`TRAP x22`	**Write null-terminated string to console. Address of string is in R0.**

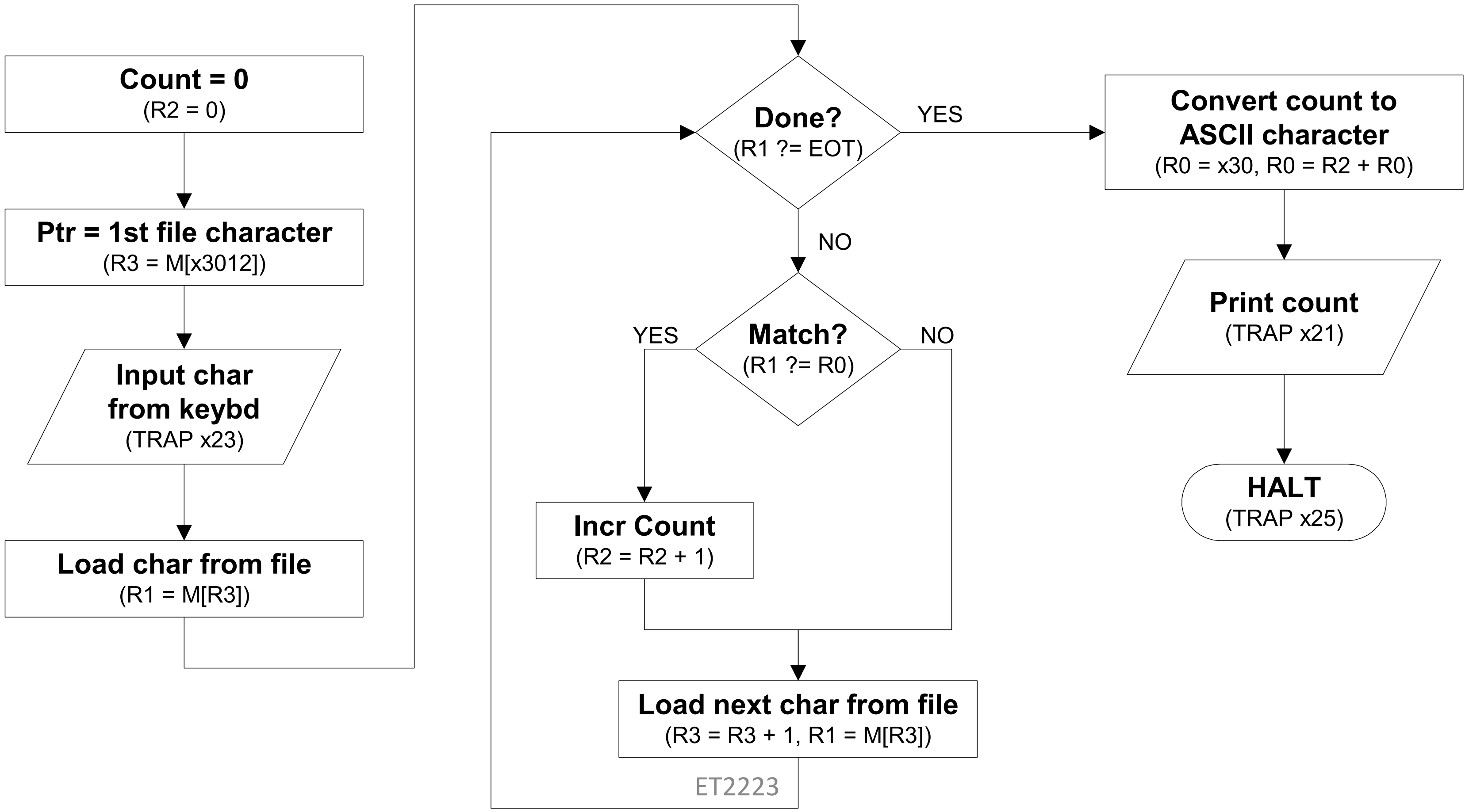
Style Guidelines

- Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author's name, date, etc., and purpose of program.
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
    - Mixed upper and lower case for readability.
    - ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.
7. Each line must fit on the page -- no wraparound or truncations.
    - Long statements split in aesthetically pleasing manner.

Sample Program

- Count the occurrences of a character in a file.



Char Count in Assembly Language (1 of 3)

```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG   x3000
        AND     R2, R2, #0      ; R2 is counter, initially 0
        LD      R3, PTR         ; R3 is pointer to characters
        GETC                    ; R0 gets character input
        LDR     R1, R3, #0      ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD     R4, R1, #-4     ; Test for EOT (ASCII x04)
        BRz     OUTPUT          ; If done, prepare the output
```

Char Count in Assembly Language (2 of 3)

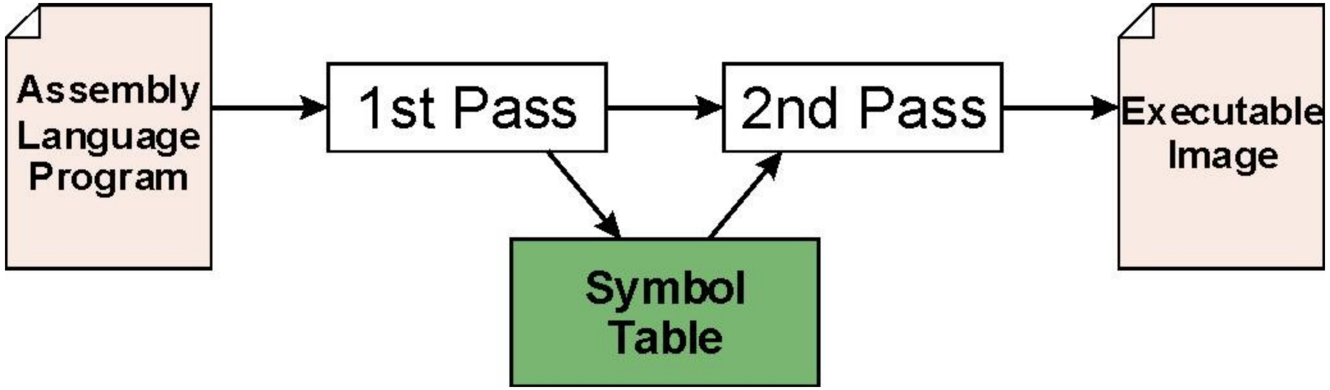
```
• ;
• ; Test character for match.  If a match, increment count.
• ;
•       NOT    R1, R1
•       ADD    R1, R1, R0  ; If match, R1 = xFFFF
•       NOT    R1, R1      ; If match, R1 = x0000
•       BRnp   GETCHAR     ; If no match, do not increment
•       ADD    R2, R2, #1
• ;
• ; Get next character from file.
• ;
• GETCHAR       ADD        R3, R3, #1    ; Point to next character.
•       LDR    R1, R3, #0  ; R1 gets next char to test
•       BRnzp  TEST
• ;
• ; Output the count.
• ;
• OUTPUT LD     R0, ASCII   ; Load the ASCII template
•       ADD    R0, R0, R2  ; Covert binary count to ASCII
•       OUT                ; ASCII code in R0 is displayed.
•       HALT               ; Halt machine
```

Char Count in Assembly Language (3 of 3)

```
;
; Storage for pointer and ASCII template
;
ASCII  .FILL   x0030
PTR    .FILL   x4000
       .END
```

Assembly Process

- Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.



- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

First Pass: Constructing the Symbol Table

1. Find the `.ORIG` statement,
   which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.

2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - NOTE: If statement is `.BLKW` or `.STRINGZ`,
        increment LC by the number of words allocated.

3. Stop when `.END` statement is reached.

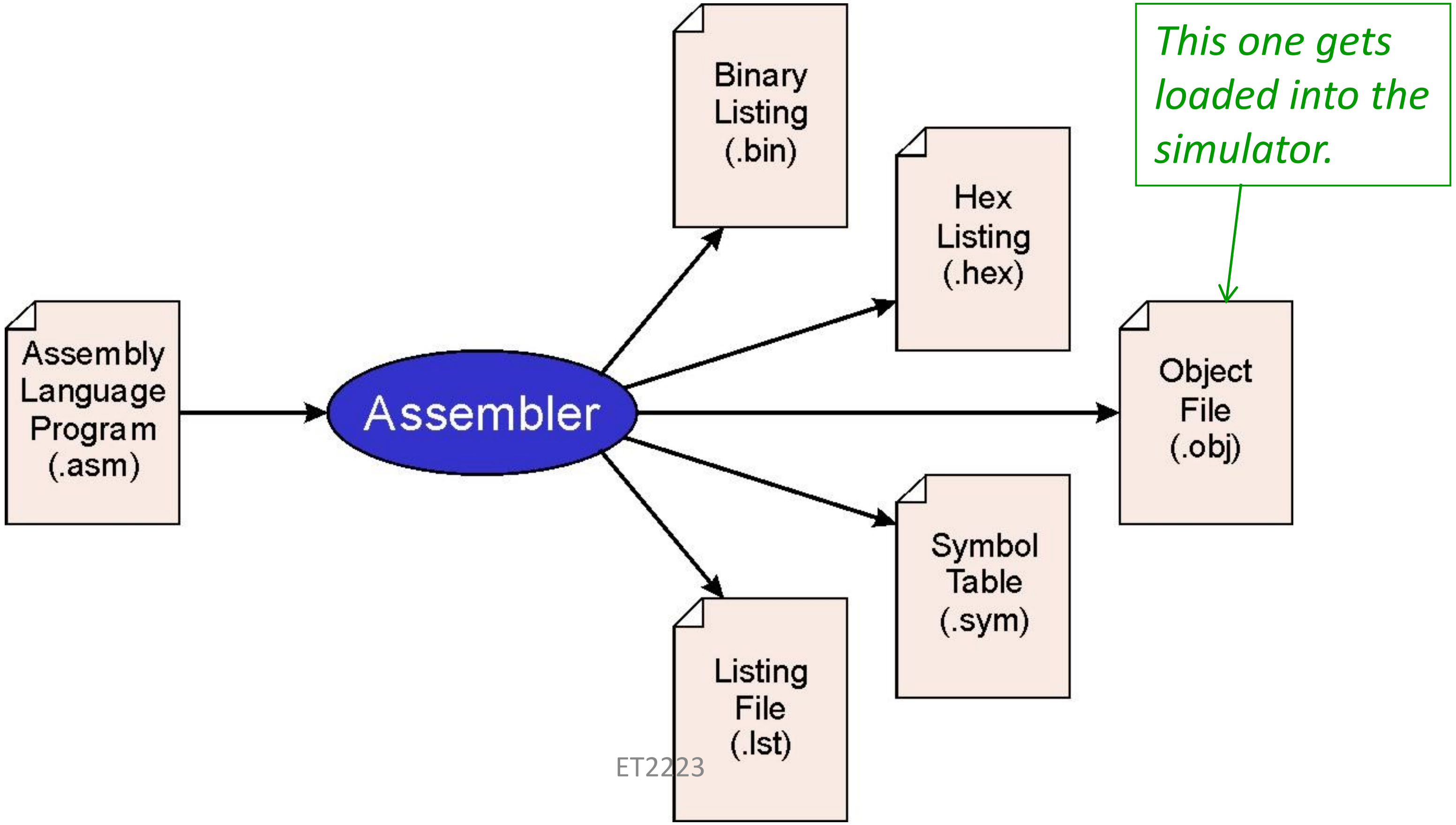
- NOTE: A line that contains only a comment is considered an empty line.

Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.
- Potential problems:
  - Improper number or type of arguments
    - ex:
      ```
      NOT     R1,#7
      ADD     R1,R2
      ADD     R3,R3,NUMBER
      ```
  - Immediate argument too large
    - ex:
      ```
      ADD     R1,R2,#1023
      ```
  - Address (associated with label) more than 256 from instruction
    - can’t use PC-relative addressing mode

LC-3 Assembler

- Using “assemble” (Unix) or LC3Edit (Windows),generates several different output files.



Object File Format

- LC-3 object file contains
  - Starting address (location where program must be loaded), followed by…
  - Machine instructions
- Example
  - Beginning of “count character” object file looks like this:

```
0011000000000000    <-- .ORIG x3000
0101010010100000    <-- AND R2, R2, #0
0010011000010001    <-- LD R3, PTR
1111000000100011    <-- TRAP x23
.
.
.
```

Multiple Object Files

- An object file is not necessarily a complete program.
  - system-provided library routines
  - code blocks written by multiple developers
- For LC-3 simulator, can load multiple object files into memory, then start executing at a desired address.
  - system routines, such as keyboard input, are loaded automatically
    - loaded into “system memory,” below x3000
    - user code should be loaded between x3000 and xFDFF
  - each object file includes a starting address
  - be careful not to load overlapping object files

Linking and Loading

- *Loading* is the process of copying an executable image into memory.
  - more sophisticated loaders are able to *relocate* images to fit into available memory
  - must readjust branch targets, load/store addresses
- *Linking* is the process of resolving symbols between independent object files.
  - suppose we define a symbol in one module, and want to use it in another
  - some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
  - linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

Types of Assembly Languages

- Assembly language closely tied to processor architecture
- At least four main types:
  - CISC: Complex Instruction-Set Computer
  - RISC: Reduced Instruction-Set Computer
  - DSP: Digital Signal Processor
  - VLIW: Very Long Instruction Word

CISC Assembly Language

- Developed when people wrote assembly language
- Complicated, often specialized instructions with many effects
- Examples from x86 architecture
  - String move
  - Procedure enter, leave
- Many, complicated addressing modes
- So complicated, often executed by a little program (microcode)

RISC Assembly Language

- Response to growing use of compilers
- Easier-to-target, uniform instruction sets
- "Make the most common operations as fast as possible"
- Load-store architecture:
  - Arithmetic only performed on registers
  - Memory load/store instructions for memory-register transfers
- Designed to be pipelined

DSP Assembly Language

- Digital signal processors designed specifically for signal processing algorithms
- Lots of regular arithmetic on vectors
- Often written by hand
- Irregular architectures to save power, area
- Substantial instruction-level parallelism

VLIW Assembly Language

- Response to growing desire for instruction-level parallelism
- Using more transistors cheaper than running them faster
- Many parallel ALUs
- Objective: keep them all busy all the time
- Heavily pipelined
- More regular instruction set
- Very difficult to program by hand
- Looks like parallel RISC instructions