# Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

# Compilers and Assemblers

# Schedule

- **Today and Friday** – Compilers & Assemblers
- **Quiz 6** – Wednesday, April 11th
  - Input / Output (HW #16)
  - Operating Systems (HW #17)
  - Compilers & Assemblers (HW #17)
  - **Review the lecture notes before the quiz (not just the homework!)**
  - **Bring a Calculator**

# Homework #16

- Review HW #16
  - Amdahl's Law
  - Disk capacity / access time
    - Hard drive prefixes are powers of 10, not 2
  - SSD bottleneck – changing a byte!
  - SSD optimization - TRIM

# Quiz 5

- Return and Review Quiz 5

# Compilers & Assemblers

# Assembler

- You used the MARIE assembler this semester
  - **What was the input?**
    - Mnemonic instructions (assembly code)
  - **What was the output?**
    - Machine code (binary)

# Assembler

- Most assemblers do this translation in two passes over the source code
  - **Pass #1:** Partially assemble the code and build symbol table
  - **Pass #2:** Complete the instructions by replacing labels with the memory addresses stored in the symbol table
- You can do this by hand – it's that easy!
  - *See Homework #8 problem*

# Loader

- The **assembler** produces the binary machine code
- The **loader** (part of the operating system) copies the machine code from disk and places it in main memory
- **Are we ready to execute it?**
  - Not quite - there's a challenge!

# Memory Addresses

- Imagine a system *without* virtual memory
- The **operating system** wants to load and run two programs at once:
  - Program A will be placed at address 1000+
  - Program B will be placed at address 5000+
- **What if the assembly code for program B was hardwired to assume it started at address 0?**
  - The program would fail – we'd have to get the programmer to send us a new version written to run at address 5000…

# Relocatable Binary Code

- Obviously, that would be a huge pain…
- Solution? **Relocatable Code**
  - MARIE doesn't use this, but real systems do
- The assembler treats your program as if it started at memory address 0
  - But, when the operating system **loader** copies the binary code from disk into main memory (to execute it), it **modifies all your memory addresses**
  - New Mem Addr = Old Addr + Start Addr of Program

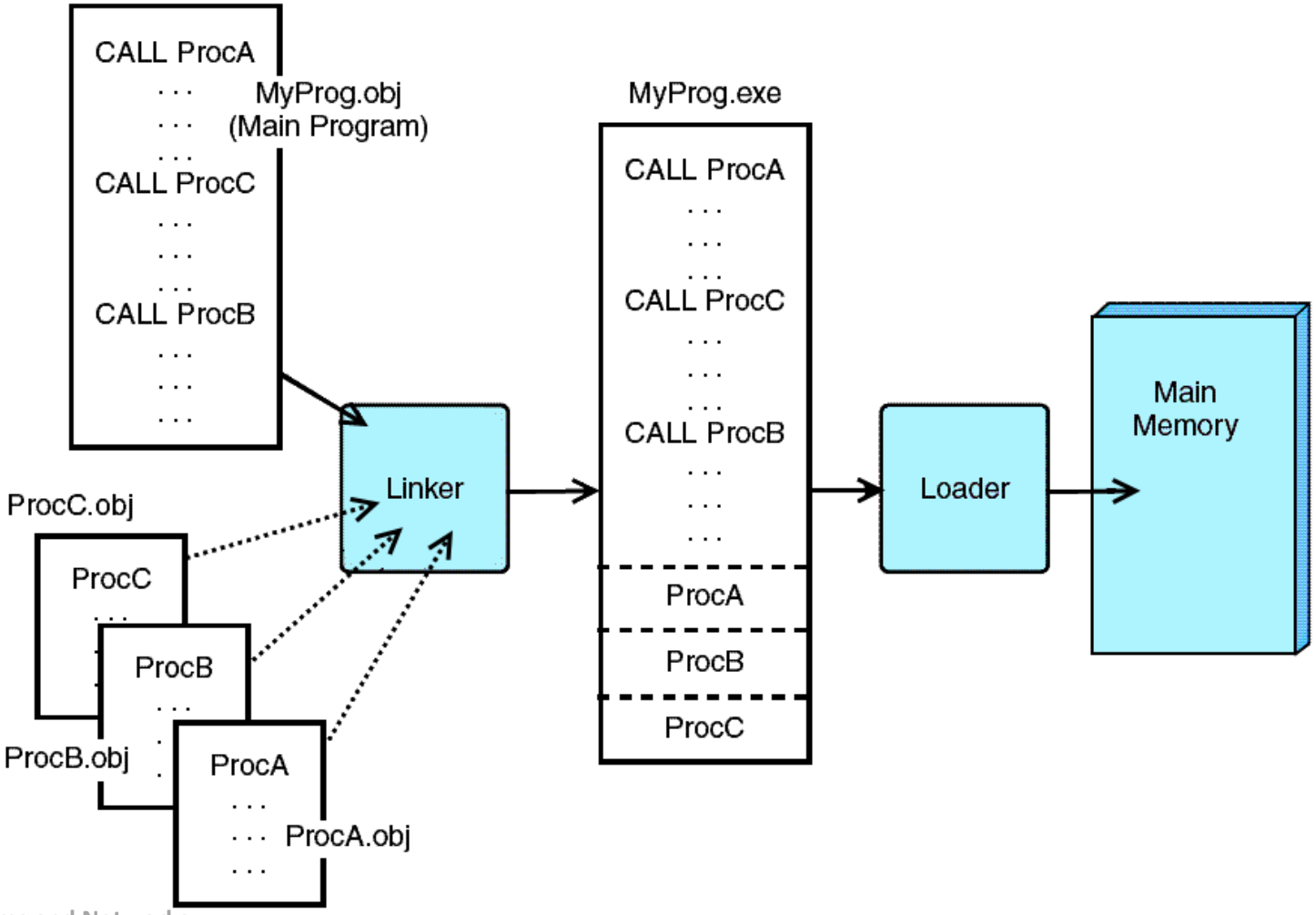
# Binary Code

- Three different types of binary code
  - **Absolute** code – operand addresses are fixed
    - This is how MARIE works
    - Suitable for device and operating system code only
  - **Relocatable** code – code that can be copied to any memory address, but must be **modified** before executing
    - Operand addresses are relative to where the operating system chooses to load the program (i.e. offset from a base address)
    - The **loader must adjust operands** when loading the program
      - Or, special registers in CPU provide base address
  - **Position Independent Code** – code that can be copied to any memory address and run **without modification**

# Linker

- Real programs are typically written with **multiple source files** and many subroutines
  - Each file is assembled separately
  - But we need some way to join everything together into a single executable file
- This is the job of the **linker** (aka "link editor")
  - Input – many files with binary machine code
  - Output – single file with all of the necessary binary machine code
- Linker also uses two passes:
  - Pass #1: Creates a symbol table
  - Pass #2: Resolve references to the values in the symbol table

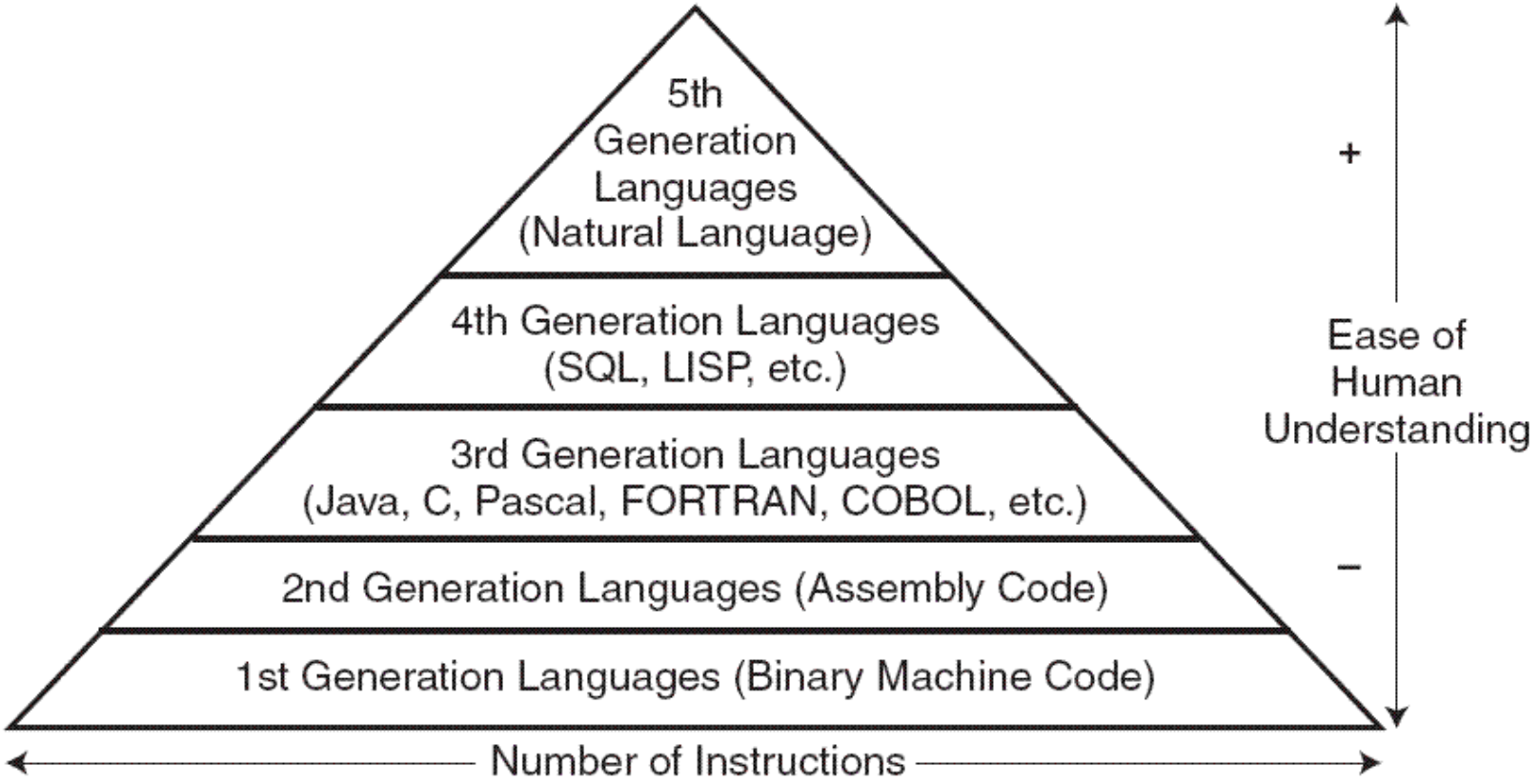
# Linker + Loader



# Dynamic Linking

- Regular linking happens at compile time (last step to produce executable file)
- **Dynamic linking** is when the linker runs when the program is loaded (or even later – when the program is running!)
  - External modules are loaded from from dynamic link libraries (DLLs)
  - Dynamic linking makes program modules smaller, but carries the risk that the programmer may not have control over the DLL

# Language Levels




Remember that the computer can understand only the 1st GL!

# Language Levels

- Each language generation presents problem solving tools that are:
  - Closer to how people think
  - Farther away from how the machine implements the solution
- Assembly code
  - **Why would I <u>want</u> (or need) to use assembly code?**
  - **Why would I <u>not want</u> to use assembly code?**
- **Compilers** bridge the semantic gap between the higher level language and the machine's binary instructions

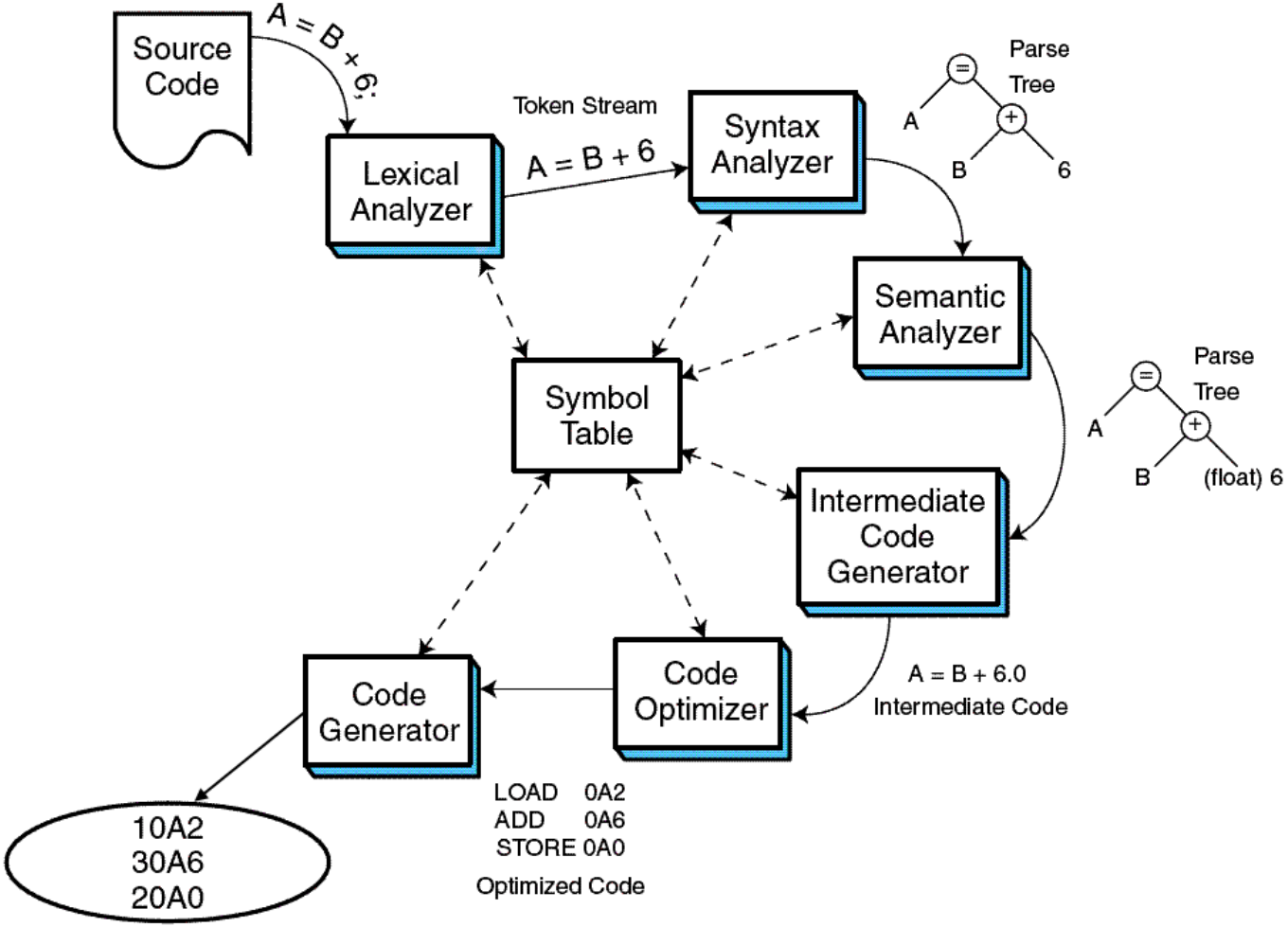
# Compiler Operation

- Compilers are much more complicated than assemblers/linkers
- Translation process takes 6 steps
- The first three steps are source code analysis:
  1. **Lexical analysis** extracts tokens, e.g., reserved words and variables
  2. **Syntax analysis** (parsing) checks statement construction
  3. **Semantic analysis** checks data types and the validity of operators

# Compiler Operation

- The last three compiler steps are **synthesis** phases:
  4. **Intermediate code generation** creates three address code to facilitate optimization and translation
  5. **Optimization** creates assembly code while taking into account architectural features that can make the code efficient
  6. **Code generation** creates binary code from the optimized assembly code
- We write these steps as separate modules
  - Benefit: Compilers can be written for various CPU architectures by rewriting only the last two modules

# Compiler Operation



# Interpreter

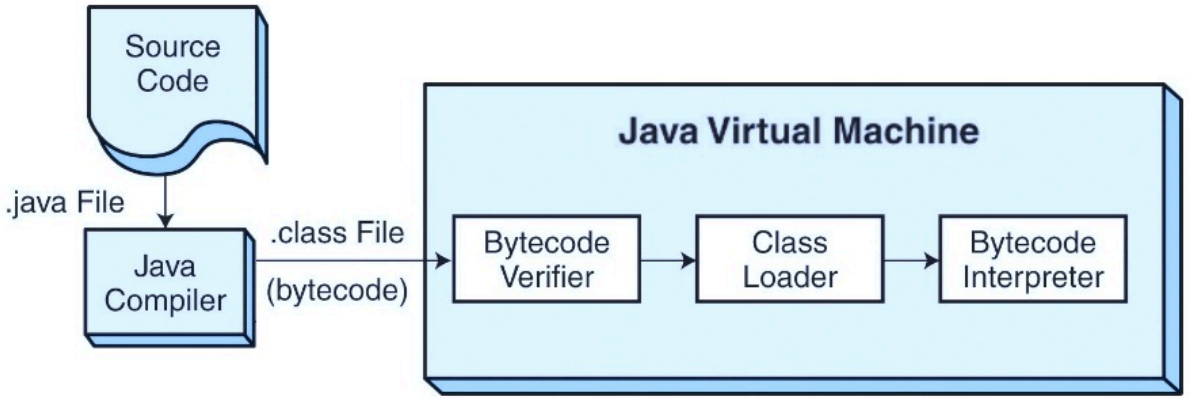
- A compiler processes all the source code and produces a binary executable *first*. Then, the executable is run.
- **Interpreters** produce executable code from source code in real time (i.e. while the program is running)
- **Pros?**
  - Don't have to wait for entire program to compile to test a part of it
  - Portability – the program is distributed as the source code, and can run on any machine architecture that has an interpreter
- **Cons?**
  - Performance – the compiler runs once, but the interpreter runs every time the program is executed

# Java: All of the Above

- Java exemplifies many of the concepts that we have discussed in this chapter
- Java programs (classes) execute within a virtual machine, the Java Virtual Machine (JVM)
  - This allows Java programs to run on any platform for which a virtual machine environment has been written
- Java is both a compiled and an interpreted language
  - The output of the compilation process is an assembly-like intermediate code (bytecode)
  - This bytecode is interpreted by the JVM

# Java: All of the Above

- The JVM is an operating system in miniature
  - It loads programs, links them, starts execution threads, manages program resources, and deallocates resources when the programs terminate



# Java: All of the Above

- At execution time, a Java Virtual Machine must be running on the host system
- It loads and executes the bytecode class file
- While loading the class file, the JVM verifies the integrity of the bytecode
- The loader then performs a number of run-time checks as it places the bytecode in memory
- The loader invokes the bytecode interpreter

# Java: All of the Above

- The bytecode interpreter:
  - Run a linker over the bytecode instructions and asks the loader to supply all referenced classes and system binaries if they are not already loaded
  - Creates and initializes the main stack frame and local variables
  - Creates and starts execution thread(s)
  - Manages heap storage by deallocating unused storage while the threads are executing **(garbage collection)**
  - Deallocates resources of terminated threads
  - Upon program termination, kills any remaining threads and terminates the JVM

# Java: All of the Above

- Because the JVM does so much as it loads and executes its bytecode, it can't match the performance of a compiled language
  - A **Just-In-Time (JIT) compiler** can help
    - Compiles blocks of to native machine code, and saves it for future reuse
- Benefits of using an interpreter?
  - Class files can be created on one machine architecture and executed on a completely different machine architecture
  - "Write once, run anywhere" model