Source Code and Compilation in Brief

Small Amount of Chapter 1

COMP2401 F25

Connor Hillen (Lecturer, He/Him)
connorhillen@cunet.carleton.ca

Learning Outcomes

Students who engage in this lecture and complete any required readings, assignments, and other practice materials should be prepared to demonstrate the following on exams and projects:

- **Use** the gcc compiler to compile a C program to produce an executable.
- **Implement** a C program that uses basic syntax, including variables, operators, expressions, and control flow statements.
- **Navigate** a *nix-like OS and its filesystem through a shell-based command line interface. Create and edit files in this environment.

Required Reading

Chapter 1 of the course notes. Make sure to **engage** in the notes: Get the Virtual Machine running, take time to learn about the new terms, try experimenting with the shell and running sample code.

Maybe ask Copilot for practice problems, helpful exercises, flashcard style questions, or questions to determine if you have any misunderstandings.

Required Reading

Some learning outcomes for the readings (helpful context for an AI, too!):

- **Navigate** a *nix-like OS and its filesystem through a shell-based command line interface. Create and edit files in this environment.
- **Implement** a C program that uses basic syntax, including variables, operators, expressions, and control flow statements.
- **Distinguish** the different types of primitive data types in C, including char, int, float, double, and void. **Explain** the differences between them.
- **Implement** a C program that uses the basic input/output functions, including printf and scanf.
- **Use** the gcc compiler to compile a C program to produce an executable.
- **Explain** the role of the virtual memory address space in a C program and how it distinguishes from hardware memory addresses.

Computing: A Brief History

Exploring how and why programs execute how they do

Computing History

Some great optional reading from the *Computer History Museum*:
https://www.computerhistory.org/timeline/computers/

Early 1940's

- ENIAC: Programmed using switches and patch cables.
- Programming was electrical engineering.
- **Debugging:** Is the cable plugged in all the way? Did a transistor burn out? Did a bug eat through a cable?

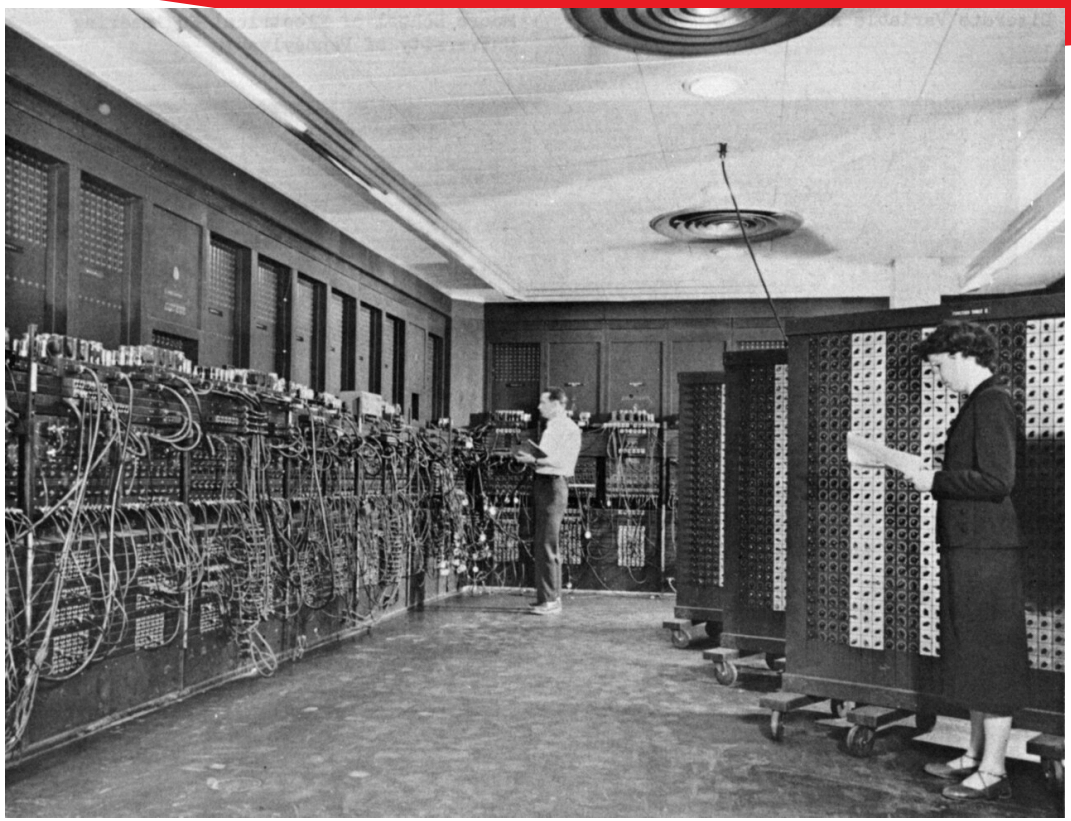


Figure 1: Glenn Beck & Betty Snyder Programming ENIAC for US Army

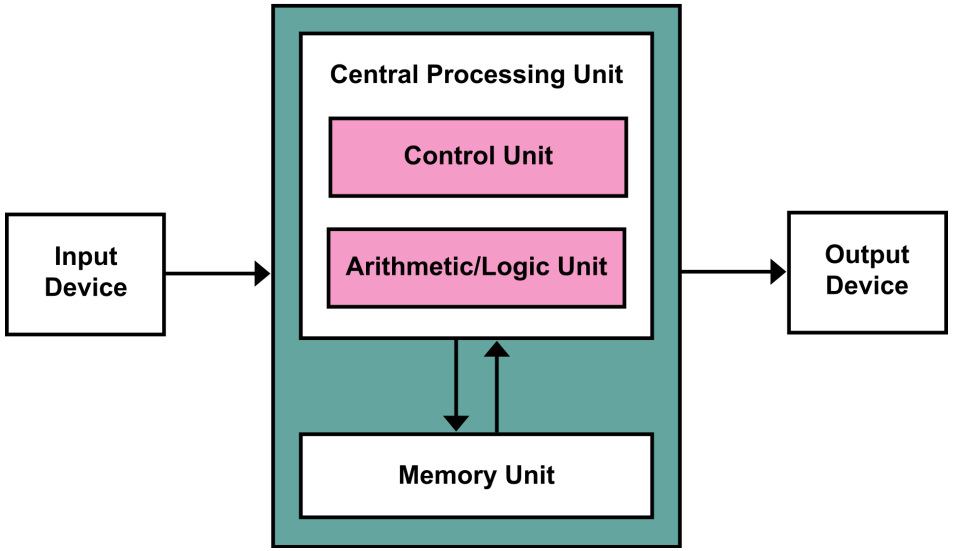


Figure 2: von Neumann Architecture

Late 40′s - 50′s:

- **Memory** holds program + instructions as a sequence of numbers
- **CU** loads instructions from memory, decode them, coordinate
- **ALU** Logic and arithmetic calculations
- **Registers** store small amounts of book keeping data
- Video Overview of **EDSAC**, Earliest “Stored Program” Computer:
  - YouTube
  - Timestamped

Assembly

EDSAC was meant to be used by students at Cambridge, so they didn't want students writing directly in binary or constantly looking up numbers in tables.

No difference between "data" and "instructions". self-modifying instructions were common to save memory.

- `A5S` means "add `memory[5]` to the `A` register"
- `E5S` means "if `A >= 0` jump to `memory[5]`"
- `G5S` means "go to `memory[5]` if `A < 0`"
- `I5S` (effectively) means "Store next character on the paper tape input into `memory[5]`"
- More instructions, and the A register (accumulator) was momre complex than letting on…

Subroutines

Let's make it easier for students!

- Write a small piece of functionality to reuse, e.g., "Divide"
- When writing a new program, copy the subroutine punched holes to the end
- To "Call" the subroutine:
  - Copy a "jump to here + 1" instruction to the end of the subroutine's instructions (the return address)
  - Jump to the subroutine

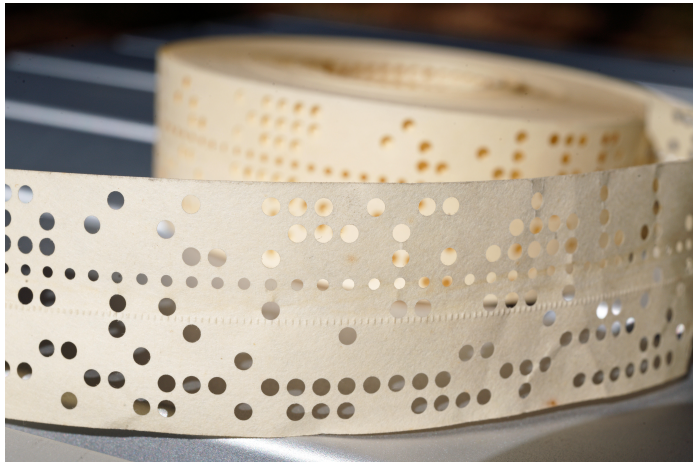


Figure 3: Punched Tape

Programming in English

What if programs could be written using English words?

Computers don’t understand English, though! What a silly notion.

Lt. Grace Hopper, Ph.d, mathemetician, programmer, insisted it would be much easier to write programs in English at a time when computers were only though to be capable of arithmetic.

Figure 4: Commodore Grace M. Hopper, USNR Official portrait photograph.

The First Compilers (c. 1958)

Grace developed the first compilers to convert mathematical notation into machine code, then continued to develop the first English-like programming language, **FLOW-MATIC**.

It separated the data from the instructions.

Each instruction mapped to **templates** of machine code instructions.

```
  (0)  INPUT INVENTORY FILE-A PRICE FILE-B ;
OUTPUT PRICED-INV FILE-C UNPRICED-INV
     FILE-D ; HSP D .
 (1)  COMPARE PRODUCT-NO (A) WITH PRODUCT-NO
(B) ; IF GREATER GO TO OPERATION 10 ;
     IF EQUAL GO TO OPERATION 5 ; OTHERWISE
GO TO OPERATION 2 .
 (2)  TRANSFER A TO D .
 (3)  WRITE-ITEM D .
 (4)  JUMP TO OPERATION 8 .
 (5)  TRANSFER A TO C .
 (6)  MOVE UNIT-PRICE (B) TO UNIT-PRICE (C) .
 (7)  WRITE-ITEM C .
 (8)  READ-ITEM A ; IF END OF DATA GO TO
OPERATION 14 .
 (9)  JUMP TO OPERATION 1 .
(10)  READ-ITEM B ; IF END OF DATA GO TO
OPERATION 12 .
(11)  JUMP TO OPERATION 1 .
(12)  SET OPERATION 9 TO GO TO OPERATION 2 .
...
```

Business Data Processing

Many languages (COBOL, FORTRAN) popped up to make it easier to conduct business, process data records, generate reports.

Programming for business people, **not** scientists.

Abstraction: Avoid thinking about the system that's running the program.

C and Unix (c. 1970)

What if a programming language could be used to drive the system itself, rather than working in assembly?

- Ken Thompson and Dennis Ritchie developed Unix, a simple operating system allowing multiple users and programs to work with multiple stored programs, in assembly
- They later created the C programming language to write Unix more easily.
- Intentionally **close** to assembly and avoiding high-level abstractions to keep it appropriate for systems programming

C and Unix (c. 1970)

What if a programming language could be used to drive the system itself, rather than working in assembly?

- Ken Thompson and Dennis Ritchie developed Unix, a simple operating system allowing multiple users and programs to work with multiple stored programs, in assembly
- They later created the C programming language to write Unix more easily.
- Intentionally **close** to assembly and avoiding high-level abstractions to keep it appropriate for systems programming
- Small programs that could be called and reused by other programs

Modern Compiling of C Code

How does C compile and execute?

Source Code

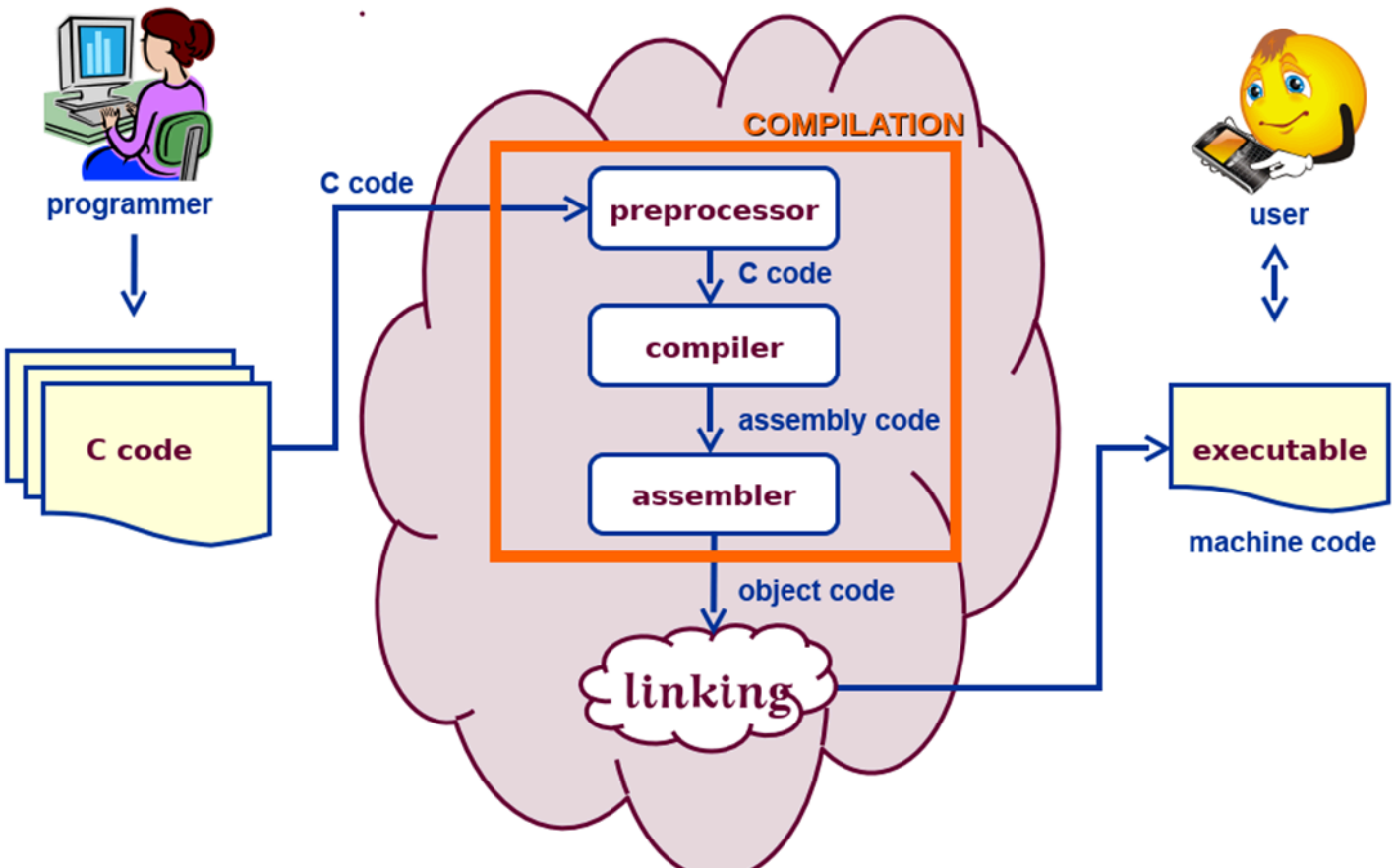
Like FLOW-MATIC, C source code is a readable format that ultimately maps to templates of machine code instructions.

Source Code Goals:

- Concisely express the programmer's intentions
- Easily translate into machine code
- Assist in avoiding errors

Ask:

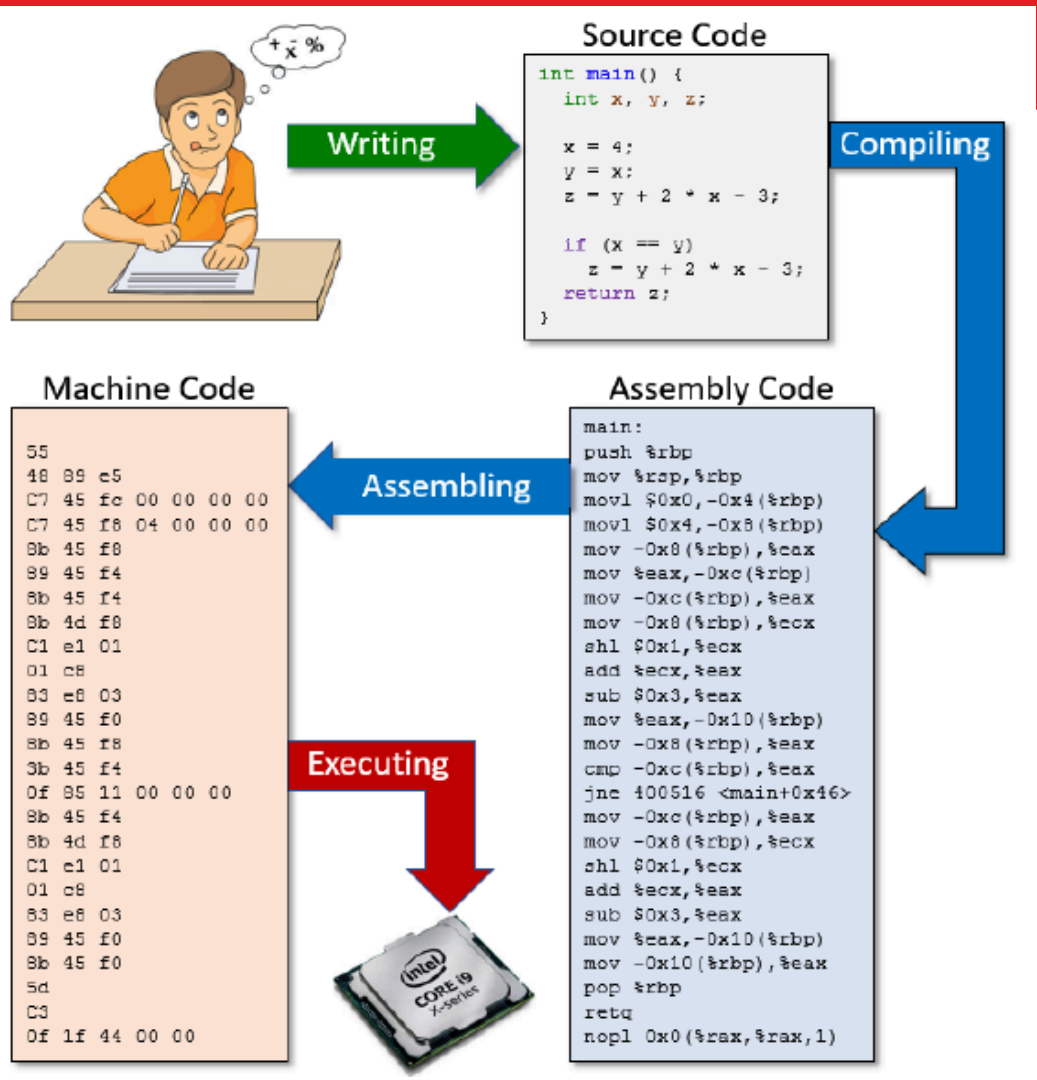
- Why is this part of the language?
- What does it translate to?
- What problem is this solving?


programmer
C code
C code
COMPILATION
preprocessor
C code
compiler
assembly code
assembler
object code
linking
user
executable
machine code

Compiling & Linking

gcc: GNU Compiler Collection (Formerly GNU C Compiler)

- One C source file *compiles* into one **object file**
- Object files contain machine code + partial memory layout information
- Linker **links** object files into one cohesive set of data and instructions - a **binary, executable** file.
- Refer to the whole process as **building**



Example

We'll look more at building and linking much later in the course, but for now, let's try to compile and execute some C code.

Compared to Java & Python

- Java and Python are **not** meant to be systems programming languages; they build on the backs of C and C++ to avoid needing to work in assembly
- **Python** is an **interpreted** language:
  - A compiled executable, Python, reads the Python script and executes it.
  - **Can** be compiled to Python bytecode which is interpreted by Python
- **Java** is similar, but it compiles to bytecode that runs on the Java Virtual Machine (JVM), which interprets the bytecode at runtime.
- Specifically designed to be agnostic to what system is running it!

C is Direct, Simple

C is a **simple** language that can be **difficult** to work in.

There is little “magic” in C — it is very explicit about what it is doing.

Its simplicity requires that you understand the level of control that it offers you.

The Old is Still Relevant

The concepts of the 1940s and 1950s are still the core of how computers work today and are the foundation of all of the abstractions our programming languages build off of.

Virtual memory allows the program to pretend that it's the only program running, working on a **private** address space, but the operating system hides the complexity of modern memory and process management.

Instead of directly writing return addresses to our instructions, we have the **function call stack**.

Instead of punched tape and vacuum tubes, we have integrated circuits and microprocessors.

Learning Outcomes

Students who engage in this lecture and complete any required readings, assignments, and other practice materials should be prepared to demonstrate the following on exams and projects:

- **Use** the gcc compiler to compile a C program to produce an executable.
- **Implement** a C program that uses basic syntax, including variables, operators, expressions, and control flow statements.
- **Navigate** a *nix-like OS and its filesystem through a shell-based command line interface. Create and edit files in this environment.

Credits

- Punched tape image: By Jud McCranie - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=116588518
- von Neumann image: User Kapooht - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:Von_Neumann_Architecture.svg
- FLOW-MATIC code: https://en.wikipedia.org/wiki/FLOW-MATIC