Compiler Code Generation

B.W.Stuck

30 November 2023

Outline

- Introduction
- An Example
- Code Generation
    - Input/Output
    - Basics
    - Instruction Selection
    - Runtime Code Generation
- Example Revisited
- Closing Remarks

An Example in Python: What Is the 27th Fibonacci Number?

- The simplest way to print Fibonacci numbers is using a while loop in Python.
- We initialize two variables, a and b, with 0 and 1,, representing the series' starting numbers.
- Inside the while loop, we print the current term and update the variables by adding them.
- This continues recursively to generate the sequence
- We will implement this in other ways, then look at how many distinct Apollo Guidance Computer instructions are needed to implement each of these

An Example in Python
What Is the 27th Fibonacci Number?

- a, b = 0, 1
- n = 27
- while b < n:
-     print(b)
-     a, b = b, a+b
- Number of AGC distinct assembly language instructions to implement this=7

An Example in Python
Backtracking to Find 27th Fibonacci Number

- def fib(n, a=0, b=1):
-     if n == 0:
-         return a
-     return fib(n-1, b, a+b)
- print(fib(27))
- Number of AGC distinct assembly language instructions to implement this=8

An Example in Python
Using Recursion to Find 27th Fibonacci Number

- def fib(n):
-     if n <= 1:
-         return n
-     else:
-         return fib(n-1) + fib(n-2)
- print(fib(27))
- Number of AGC distinct assembly language instructions to implement this=8

Using Dynamic Programming to Speed Up Recursion Calculation of 27th Fibonacci Number

- We can optimize the recursive solution using dynamic programming and memoization techniques. The basic idea is to store already computed terms in a lookup table. Before adding any term, we check if it exists in the lookup table. This avoids recomputing the words and makes the algorithm faster.
- memo = {0:0, 1:1}
- def fib_dynamic(n):
-     if n in memo:
-         return memo[n]
-     memo[n] = fib_dynamic(n-1) + fib_dynamic(n-2)
-     return memo[n]
- print(fib_dynamic(6))
- Number of AGC distinct assembly language instructions to implement this=9

Using Python LRU Cache to Improve Performance for 27th Fibonacci Number

- The Python lru_cache decorator can cache and reuse previously computed Fibonacci terms. This also prevents redundant calculations.
- from functools import lru_cache
- @lru_cache(maxsize=1000)
- def fib(n):
-     if n == 0:
-         return 0
-     elif n == 1:
-         return 1
-     else:
-         return fib(n-1) + fib(n-2)
- print(fib(5))
- Number of AGC distinct assembly language instructions to implement this=9

Comparing Fibonacci Algorithms

- The various algorithms have their pros and cons for generating the Fibonacci sequence.
- The loop method is the simplest to code but becomes slow for significant inputs.
- Recursion provides elegant code but has redundant function calls.
- Dynamic programming improves recursion by storing results.
- Caching further boosts efficiency by reusing prior computation.
- Recursion and dynamic programming follow the mathematical definition closely. Caching works best for iterative programs by caching previous results.
- The optimal algorithm depends on the use case, input size, and code complexity requirements. A mix of techniques can be used as well.
- Source: https://www.simplilearn.com/tutorials/python-tutorial/fibonacci-series#:~:text=The%20simplest%20way%20to%20print,recursively%20to%20generate%20the%20sequence.

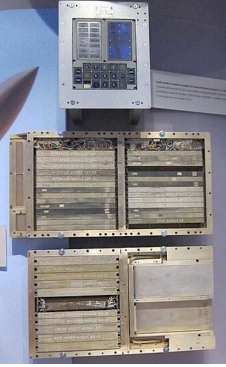
Apollo Guidance Computer AGC

- The **Apollo Guidance Computer** (**AGC**) was a digital computer produced for the Apollo program that was installed on board each Apollo command module (CM) and Apollo Lunar Module (LM).
- The AGC provided computation and electronic interfaces for guidance, navigation, and control of the spacecraft.[3] The AGC was the first computer based on silicon integrated circuits.
- The computer's performance was comparable to the first generation of home computers from the late 1970s, such as the Apple II, TRS-80, and Commodore PET.[4]

Apollo Guidance Computer AGC

- The AGC has a 16-bit word length, with 15 data bits and one parity bit. Most of the software on the AGC is stored in a special read-only memory known as core rope memory, fashioned by weaving wires through and around magnetic cores, though a small amount of read/write core memory is available.
- Astronauts communicated with the AGC using a numeric display and keyboard called the DSKY (for "display and keyboard", pronounced "DIS-kee"). The AGC and its DSKY user interface were developed in the early 1960s for the Apollo program by the MIT Instrumentation Laboratory and first flew in 1966.

Apollo Guidance Computer Registers



- The AGC had four 16-bit registers for general computational use, called the central registers:
- A: The accumulator, for general computation
- Z: The program counter – the address of the next instruction to be executed
- Q: The remainder from the DV instruction, and the return address after TC instructions
- LP: The lower product after MP instructions
- There were also four locations in core memory, at addresses 20–23, dubbed editing locations because whatever was stored there would emerge shifted or rotated by one bit position, except for one that shifted right seven bit positions, to extract one of the seven-bit interpretive op. codes that were packed two to a word.

Apollo Guidance Computer Instruction Set

- The instruction format used 3 bits for opcode, and 12 bits for address. Block I had 11 instructions: TC, CCS, INDEX, XCH, CS, TS, AD, and MASK (basic), and SU, MP, and DV (extra).
- The first eight, called basic instructions, were directly accessed by the 3-bit op. code.
- The final three were denoted as extracode instructions because they were accessed by performing a special type of TC instruction (called EXTEND) immediately before the instruction.
- Total number of instructions=33
- Total storage=2,048 words erasable magnetic storage + 36,864 words read only core rope memory (16 bit word: 15 bits data, 1 bit odd parity)=77.824 kilobytes total storage

Apollo Guidance Computer Interpreter

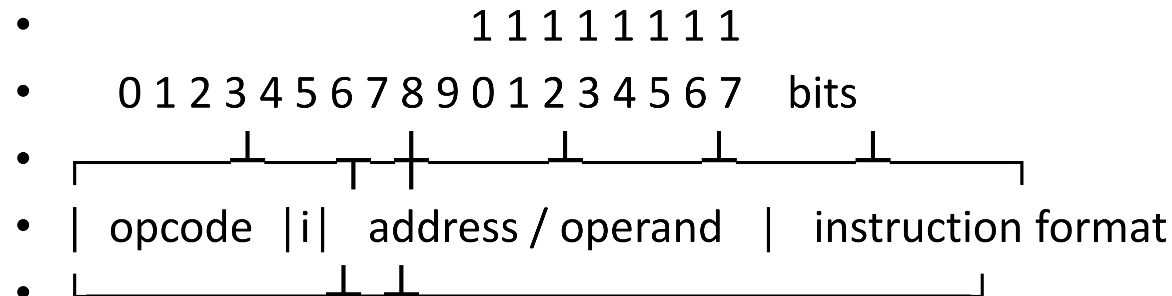
- The AGC had a sophisticated software interpreter, developed by the MIT Instrumentation Laboratory, that implemented a virtual machine with more complex and capable pseudo-instructions than the native AGC.
- These instructions simplified the navigational programs. Interpreted code, which featured double precision trigonometric, scalar and vector arithmetic (16 and 24-bit), even an MXV (matrix × vector) instruction, could be mixed with native AGC code.
- While the execution time of the pseudo-instructions was increased (due to the need to interpret these instructions at runtime) the interpreter provided many more instructions than AGC natively supported and the memory requirements were much lower than in the case of adding these instructions to the AGC native language which would require additional memory built into the computer (at that time the memory capacity was very expensive). The average pseudo-instruction required about 24 ms to execute. The assembler, named YUL for an early prototype Christmas Computer,[enforced proper transitions between native and interpreted code.

Example: System 360 Instruction Set Classes

- Branch Instructions, which jump from one part of a program to another
- Data Transfer Instructions, which move data from one part of memory (including registers)
- Control Flow Instructions, which make changes in the scope of execution of a program
- Arithmetic Instructions, which do computations
- Logic Instructions, which make comparisons
- Shift and Rotate Instructions, which move bits right or left,
- Privileged Instructions, used by the operating system or other programs having special permission, primarily to do input/output or to control the operations of the machine, and
- Other Instructions, instructions not otherwise classified

Example: DEC PDP1 Instruction Set

- The PDP-1 (Digital Equipment Corporation Programmed Data Processor – 1) uses the following instruction format:



- The standard memory of 4k 18-bit words is in the address space from 00000 to 77777 (octal). Additional memory is addressible either by an extended address mode or by bank switching. There are 6 program flags and 6 sense switches. The console test word is a normal 18-bit register.
- The PDP-1 uses 1's complement to represent negative numbers. There are two user addressible register, the accumulator (AC) and the I/O register (IO), which might be combined to a single 36 bit register for shifts and rotates. Further, there is an overflow flag (set in instructions add and sub), a program counter (PC), and an internal memory buffer (MB). Addresses and values are usually referred to in octal format (since memory words, addresses, and any parameters are multiples of 3 and a nibble of 3 bits is forming a single octal digit).

What Is Code Generation?

- In computing, code generation is part of the process chain of a compiler and converts intermediate representation of source code into a form (e.g., machine code) that can be readily executed by the target system.
- Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the completed processing performed by another optimization.
- This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the backend) needs to change from target to target.

Code Generation Input/Output

- The input to the code generator typically consists of a parse tree or an abstract syntax tree.
- The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three-address code.
- Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program.
  - For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.

Code Generation Basics

- Instruction selection: which instructions to use.
- Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
- Register allocation: the allocation of variables to processor registers
- Debug data generation if required so the code can be debugged.

Instruction Set Implementation Tradeoffs

- The design of instruction sets is a complex issue. There were two stages in history for the microprocessor. The first was the CISC (Complex Instruction Set Computer), which had many different instructions.
- In the 1970s, however, places like IBM did research and found that many instructions in the set could be eliminated. The result was the RISC (Reduced Instruction Set Computer), an architecture that uses a smaller set of instructions.
- A simpler instruction set may offer the potential for higher speeds, reduced processor size, and reduced power consumption.
- However, a more complex set may optimize common operations, improve memory and cache efficiency, or simplify programming.

Instruction Set Architecture

- In computer science, an instruction set architecture (ISA) is an abstract model of a computer. A device that executes instructions described by that ISA, such as a central processing unit (CPU), is called an implementation.
- In general, an ISA defines the supported instructions, data types, registers, the hardware support for managing main memory, fundamental features (such as the memory consistency, addressing modes, virtual memory), and the input/output model of a family of implementations of the ISA.
- An ISA specifies the behavior of machine code running on implementations of that ISA in a fashion that does not depend on the characteristics of that implementation, providing binary compatibility between implementations. This enables multiple implementations of an ISA that differ in characteristics such as performance, physical size, and monetary cost (among other things), but that are capable of running the same machine code, so that a lower-performance, lower-cost machine can be replaced with a higher-cost, higher-performance machine without having to replace software. It also enables the evolution of the microarchitectures of the implementations of that ISA, so that a newer, higher-performance implementation of an ISA can run software that runs on previous generations of implementations.

Instruction Selection

- Instruction selection is typically carried out by doing a recursive postorder traversal on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.
- In a compiler that uses an intermediate language, there may be two instruction selection stages—one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine.
- This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding opcodes. However, if the compiler is actually a language translator (for example, one that converts Java to C++), then the second code-generation phase may involve building a tree from the linear intermediate code.

Runtime Code Generation

- In instances where code generation takes place during runtime, such as in just-in-time compilation (JIT), it becomes crucial for the entire process to be efficient in terms of both space and time.
- For instance, when interpreting regular expressions and generating code on-the-fly, a non-deterministic finite state machine is frequently preferred over a deterministic one.
- This preference arises because the former can typically be generated more swiftly and requires less memory space. Despite the tendency to produce less efficient code, JIT code generation can leverage profiling information that is accessible only during runtime.

Related Concepts

- The fundamental task of taking input in one language and producing output in a non-trivially different language can be understood in terms of the core transformational operations of formal language theory. Consequently, some techniques that were originally developed for use in compilers have come to be employed in other ways as well.
- For example, YACC (Yet Another Compiler-Compiler) takes input in Backus–Naur form and converts it to a parser in C. Though it was originally created for automatic generation of a parser for a compiler, yacc is also often used to automate writing code that needs to be modified each time specifications are changed.
- Many integrated development environments (IDEs) support some form of automatic source-code generation, often using algorithms in common with compiler code generators, although commonly less complicated.

An Example Revisited

- To implement the example requires 7 out of 33 different machine instructions actually available
- This suggests that only a small subset of machine instructions will determine performance (e.g., lines of source code compiled per second, lines of source code executed per second)

Reflection

- In general, a syntax and semantic analyzer tries to retrieve the structure of the program from the source code, while a code generator uses this structural information (e.g., data types) to produce code.
- In other words, the former **adds** information while the latter **loses** some of the information.
- One consequence of this information loss is that reflection becomes difficult or even impossible. To counter this problem, code generators often embed syntactic and semantic information in addition to the code necessary for execution.