Compilers

Topic: Introduction to Compilers

Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper's CS412 at Rice University

Programming Abstractions

We can program a microprocessor using

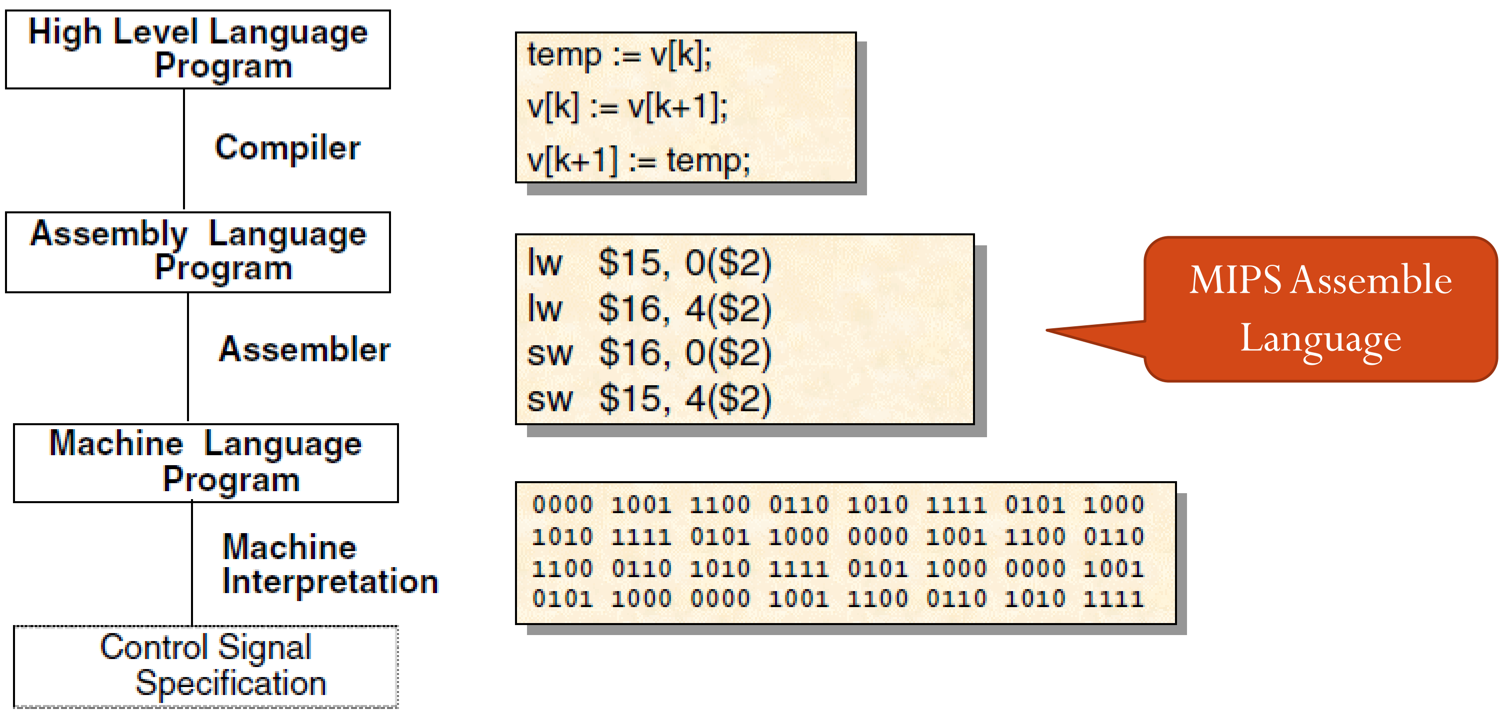
a) Instruction opcodes (also called Machine Code)

b) Assembly language

c) High level programming languages

- The level of abstraction increases from Top to Bottom.
- As the level of abstraction increases, ease of programmability also increases! → But some one has to implement the abstraction for us efficiently!
- Hmm, but we may lose the fine-grained control over the underlying hardware?

Levels of Abstraction



Source: Hennessy & Patterson

Assembly Language or a High Level PL

AssemblyLanguage	High Level Programming Languages
❑ Hard to Program.	❑ Easy to program.
❑ Low productivity.	❑ High productivity.
❑ Hard to read.	❑ Easy to read.
❑ Difficult to debug.	❑ Easy to debug.
❑ Code not portable to new platforms.	❑ Relatively easy to port to new platforms.
❑ Better performance?	❑ Lower performance?
.	.
.	.

Good compilers can produce assemble code with the same or even better performance than the equivalent program directly written in Assembly Language.

Algorithms, Data Structures and Programs

Algorithms + Data Structures = Programs

Niklaus Wirth

Programming Languages

A Programming Language provides

Key Idea: We should be able to realize these abstractions through the ISA of a given processor!

- Data Abstractions
  - int, float, bool etc. data types.
  - Mechanism for hierarchal composition of new Data Abstractions
    - Structures, Arrays, Unions etc.
- Data Processing Abstractions
  - Arithmetic and Boolean Operations, String Operations etc.
- Control Abstractions
  - while, if, for constructs etc.
- Object Oriented Language Abstractions: Classes, Polymorphism, …..

Programming Language Abstractions

- A Compiler acts as a Bridge between Programming Language Abstractions and Computer Architecture
- Any Programming Language Abstraction implicitly requires a strategy for efficient implementation from the underlying compiler/runtime system.

Question: What principles guide the design of Language Abstractions?

Compilers

A Compiler translates a program in a source language to an equivalent program in a target language

Typically

- Source Languages - C, C++, ADA, FORTRAN etc.
- Target languages - Instruction set of some microprocessor

An assembler translates assemble language programs in to object code.

Compilers and Computer Architecture

Compilers should

- Efficiently use the architectural features provided by the underlying microprocessors
- Hide drawbacks in the architectural design of the microprocessors
- Compilers and runtime systems are key to exploiting the raw computing power provided by the past and emerging Computing Systems.

Desirable Properties of a Compiler

- Speed
- Space
- Feedback
- Debugging
- Compile-Time Efficiency

Compilers and Computer Architecture

Parallelism Granularities

- Instruction Level Parallelism – Simple Pipelined Processors, Superscalar processors, VLIW processors
- Processor Level Parallelism – Mulitcore architectures, Hyperthreading architectures etc.

Compilers and runtime systems play a vital role in utilizing the parallelism provided by the underlying computing system.

Reducing the Price of Abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses for them
- We invent ways to make them efficient
- Programming is the way we realize these inventions
- Well written compilers make abstraction affordable
- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
  — Don't expect bubblesort to become quicksort

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

-John Backus

Interpreters

An interpreter executes the operations specified by a source program on its inputs.

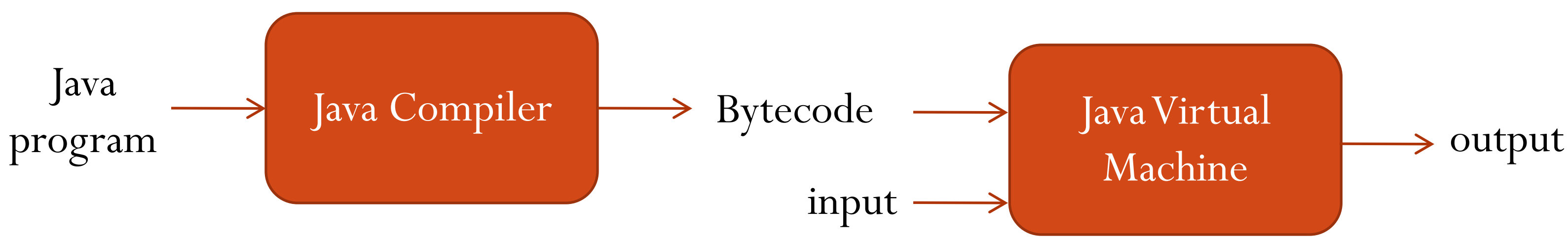
Examples – Scheme, Perl, Python, Shell scripts, PDF readers, JavaScript, Java(?)

Interpreters

Interpretation approaches

- Interpret the source program directly
- Convert the source program into an intermediate representation (IR) and then interpret the IR.

Examples of IR: Abstract Syntax Trees (ASTs), Byte Code



Interpreters vs Compilers

Interpreters	Compilers
❑ Programs written in interpreted languages are easily portable.	❑ Every program needs to be recompiled for a new platform.
❑ Interpreter has run time information about the program. So better error diagnostics and performance optimization.	❑Compiler has only static information about the program. So may not be able to catch some runtime errors.
❑ Lower performance.	❑Better performance.
……	…

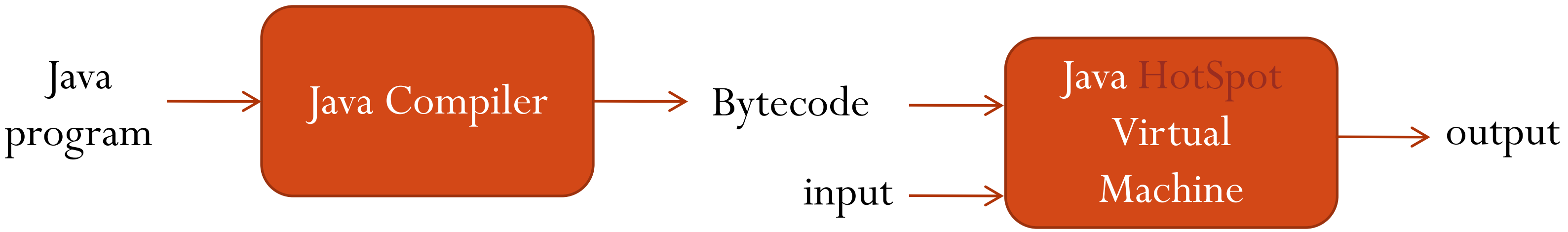
Focus of this Course is on Compilers

Just-in-Time Compilers

Java HotSpot Virtual machine compiles

- frequently called methods and
- methods containing loops that loop a lot

into native code at runtime



Pure Interpretation versus JIT

```
public class sqrsum {
    public static void main(String args[]) throws NumberFormatException, IOException {

        int a = Integer.parseInt(args[0]);
        long result=0;
        for(int i=1; i<= a; ++i)
            result+= sqrsum(i);
       System.out.println("sqrsum of the numbers is: " + result);
    }
    static int sqrsum(int b)    {
     return b*b;
    }
}
```

Using Sun's Java HotSpot VM, for a = 10000000 (10^7)

- Pure Intrepretation: 0.37 seconds
- With JIT: 0.07 seconds

Back to Compilers

Why Study Compilers?

- Compiler construction poses challenging and interesting problems:
  - Compilers must process large inputs, perform complex algorithms, but also run quickly
  - Compilers have primary responsibility for run-time performance
  - Compilers are responsible for making it acceptable to use the full power of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
    - Compilers must hide that complexity from the programmer
- A successful compiler requires mastery of the many complex interactions between its constituent parts

Why Study Compilers?

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms, Heuristic search techniques
Algorithms	Graph algorithms, union-find and Dynamic programming
Theory	DFAs & PDAs, pattern matching, Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management, Instruction set use

What makes Compilers Interesting?

- Excellent application of theory to practice.
  - Front end – Lexical analysis, parsing etc.
  - Back end – Application of Lattice Theory to Data Flow Analysis
- Practical algorithms (algorithms which makes sense?)
- Almost all problems in the Compiler backend are NP-Complete
  - Well, how are we going to solve these hard problems?
- Application of ideas from graph theory, linear programming etc.

Why Does this Matter Today?

In the last 3 years, most processors have gone multicore

- The era of clock-speed improvements is drawing to an end
  - Faster clock speeds mean higher power (n^2 effect)
  - Smaller wires mean higher resistance for on-chip wires
- For the near term, performance improvement will come from placing multiple copies of the processor (core) on a single die
  - Classic programs, written in old languages, are not well suited to capitalize on this kind of multiprocessor parallelism
    - Parallel languages, some kinds of OO systems, functional languages
  - Parallel programs require sophisticated compilers