Topics I
Fundamentals
Topic I: Outline

• Part I: Compiler Fundamentals
  • An Overview on Compiler Design
  • Compiler Front-End and IR
  • Middle-End: Analysis and Optimizations
  • Back-End: Code Generation and Optimization
Foundations for Compiler Design

1. Processor architecture design flow
2. Compiler structure and design flow
3. Code generation design flow
Why Study Compilers?

• Influences on programming language design
• Influences on computer design
• Compiling techniques are useful for software development
  — Parsing techniques are often used
  — Learn practical data structures and algorithms
  — Basis for many tools such as text formatters, structure editors, silicon compilers, design verification tools,…

Writing a compiler requires an understanding of almost all important CS subfields
Architecture Models

Vector architectures, SIMD
Instructional Level Parallelism (ILP)
  superscalar
  VLIW
Multithreaded Architectures
Chip multiprocessing (CMP, multi-core, many-core, etc.)
GPGPUs
Reconfigurable Architectures

What is the impact of these ideas on compilers?
Instruction Set Architecture Design (Microarchitecture Design-I)

System Level Design

System Level Simulator

Processor Architecture Design Flow Diagram

Instruction Set Simulator

IR Level Design

Compiler Design

Compiler Design

Code Optimizer

Code Generator

Toolchain
- Intel VTune™
- IBM Performance Evaluator

Debugger

Arch./Compiler and System Software Design Toolset

RTL Level Design (Microarchitecture Design II)

RTL Level Simulator

Switch Level Simulator

Circuit Level Simulator

HDL (VHDL or Verilog)

Hardware Design

Switch Level Design

Circuit Level Design

Arch./Compiler and System Software Design Toolset
What does a Compiler do Anyway?
What does a Compiler do Anyway? (Cont’d)
What does a Compiler do Anyway? (Cont’d)
What does a Compiler do Anyway? (Cont’d)

Compiler: .c → .o

Linker: .o → .exe
What does a Compiler do Anyway? (Cont’d)
The Whole Compilation Chain

Compiler → .c → .o → Linker → .exe → .so → Loader → Running Program
Inside a Compiler

Front-End
- Scanning
- Parsing
- Semantic Analysis

Middle-End
- Optimization 1
- Optimization N

Back-End
- Select
- Schedule
- Allocate
A Quick Look at the Front-End: The Lexer

- Makes sure that every single “word” in the programming language is well-formed
- Outputs tokens which describe to what category each word in a given program belongs to
- Think of it as a “spell checker”
A Quick Look at the Front-End: The Parser

- Takes tokens as input
- Makes sure the tokens are inserted in a valid sequence
- Think of it as a “grammatical checker”

The lexer and the parser make sure there the input program is correct with respect to the formal semantics of the language used by the programmer
A Quick Look at the Front-End: The Semantic Analyser

- Context-sensitive analysis (or semantic analysis) checks that the output of the lexer and parser has *meaning*.
  - E.g. “This house is very clever” vs “This student is very clever” → both are grammatically correct, only one has meaning
- It is useful at several levels:
  - Correctness can be further ensured
  - Can ensure safety through type-checking
  - Can provide the middle-end and back-end with useful information w.r.t. certain expressions
A Good Compiler Infrastructure Needed – A modern View

Good IR

Front end

Interprocedural Analysis and Optimization

Loop Nest Optimization and Parallelization

Global (Scalar) Optimization

Middle-End

Backend

Code Generation
Middle-End Optimization

- Flow Analysis
  - Control flow analysis
  - Dataflow analysis
- Global scalar optimization
- Loop nest optimization
- Advanced topics:
  - Static Single Assignment form (SSA)
  - Application of SSA to scalar optimization
Backend Optimization (I)

- Instruction selection
- Instruction scheduling
- Register allocation
- Others
Backend Optimization (II)

- Loop optimization and scheduling
- Software pipelining