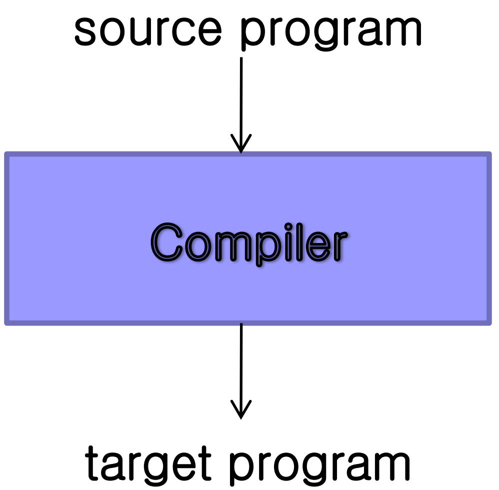
Introduction

- Read Chapter 1

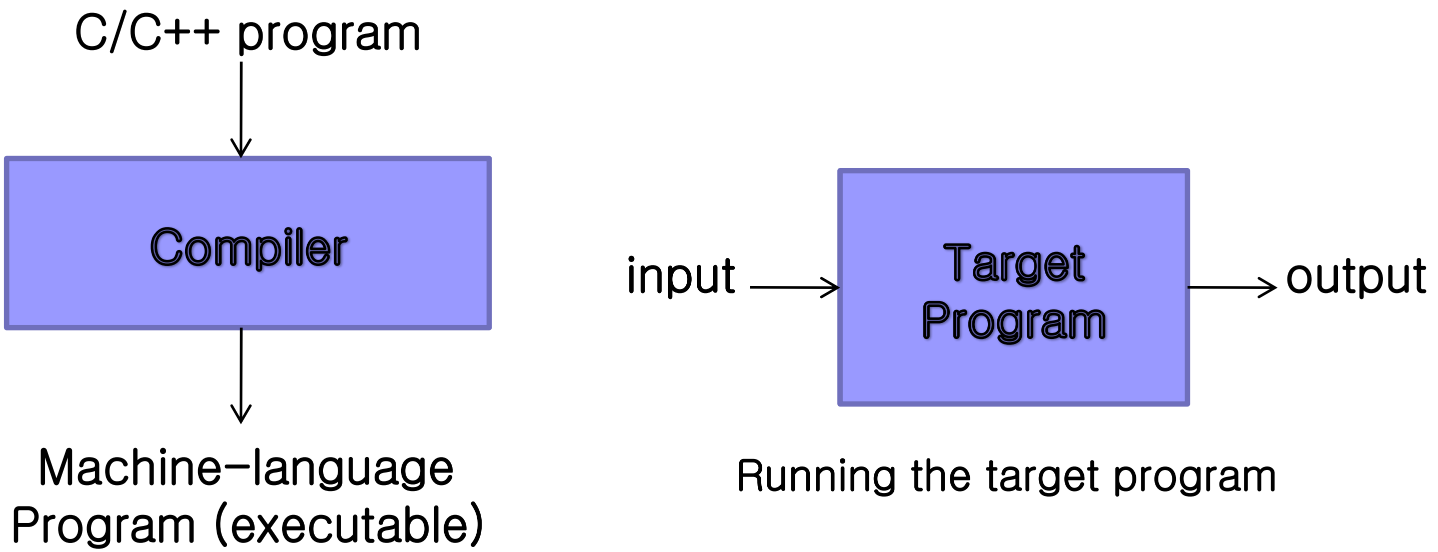
What is a Compiler?

- Translator from one language (source language) to another language (target language)
  - Input a program in one language
  - Output an equivalent program in another language
  - One important role is to report any errors in the source program



Programming Lang. Compilers

Compile source program and run target program



Interpreter (Emulator)

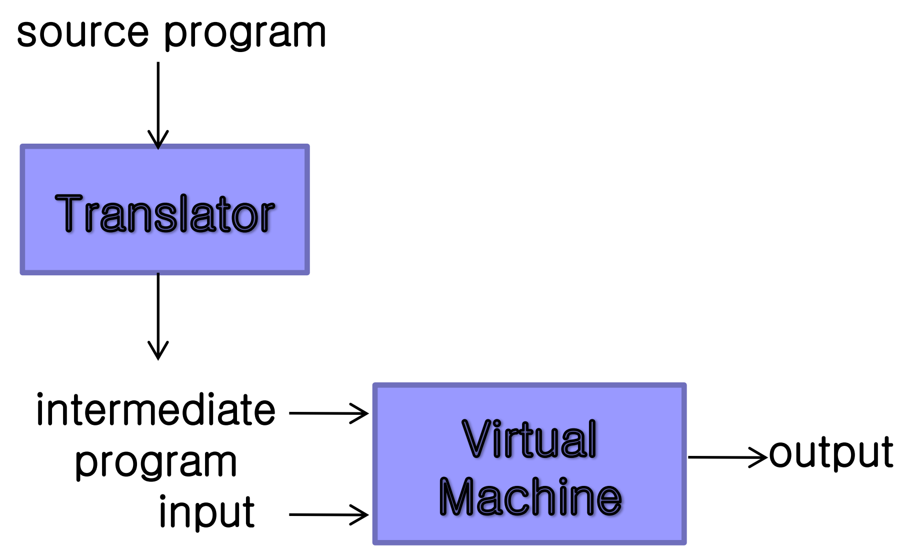
- Another form of running program
  - Instead of producing a target program with translation, directly execute the source program

An interpreter

  - Slower than machine program, but faster to develop and better handling of errors

A hybrid compiler

- Combines compilation and interpretation
  - Java program is compiled to *bytecode*, which is then interpreted on the virtual machine
    - Better portability
    - Mainstream these days
    - JavaScript, Python, Ruby, …

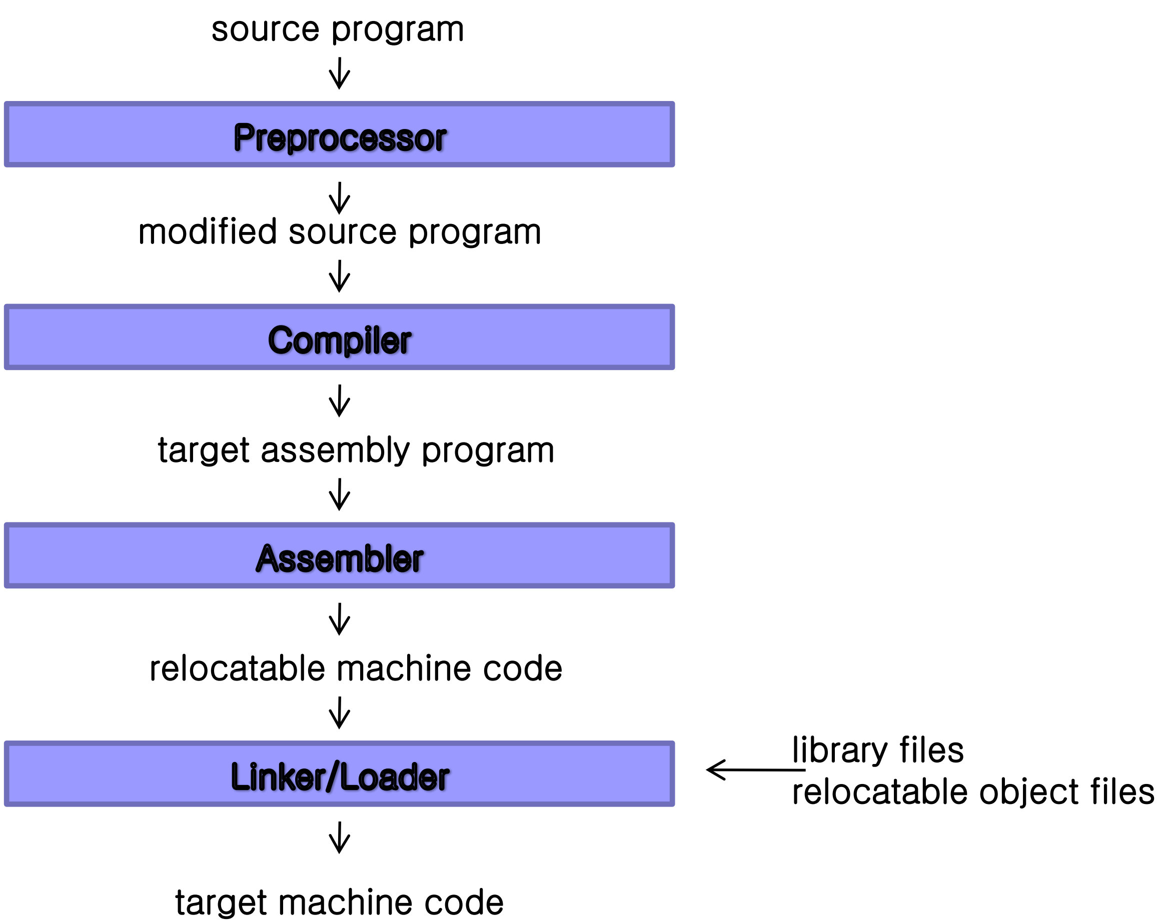


A hybrid compiler

Other Usage of Compilers

- While compilers most prevalently participate in programming language translation, other form of compiler technology has also been utilized
  - Compiler-compilers:
    lex: regular expressions → scanner (lexer)
    yacc: language grammars → parser
  - Text processing: LaTex, Tex, troff
  - Database query processors
    Predicates → commands to search the DB
  - Silicon compilers: Circuit spec → VLSI layouts
- The goal of every compiler is correct and efficient translation

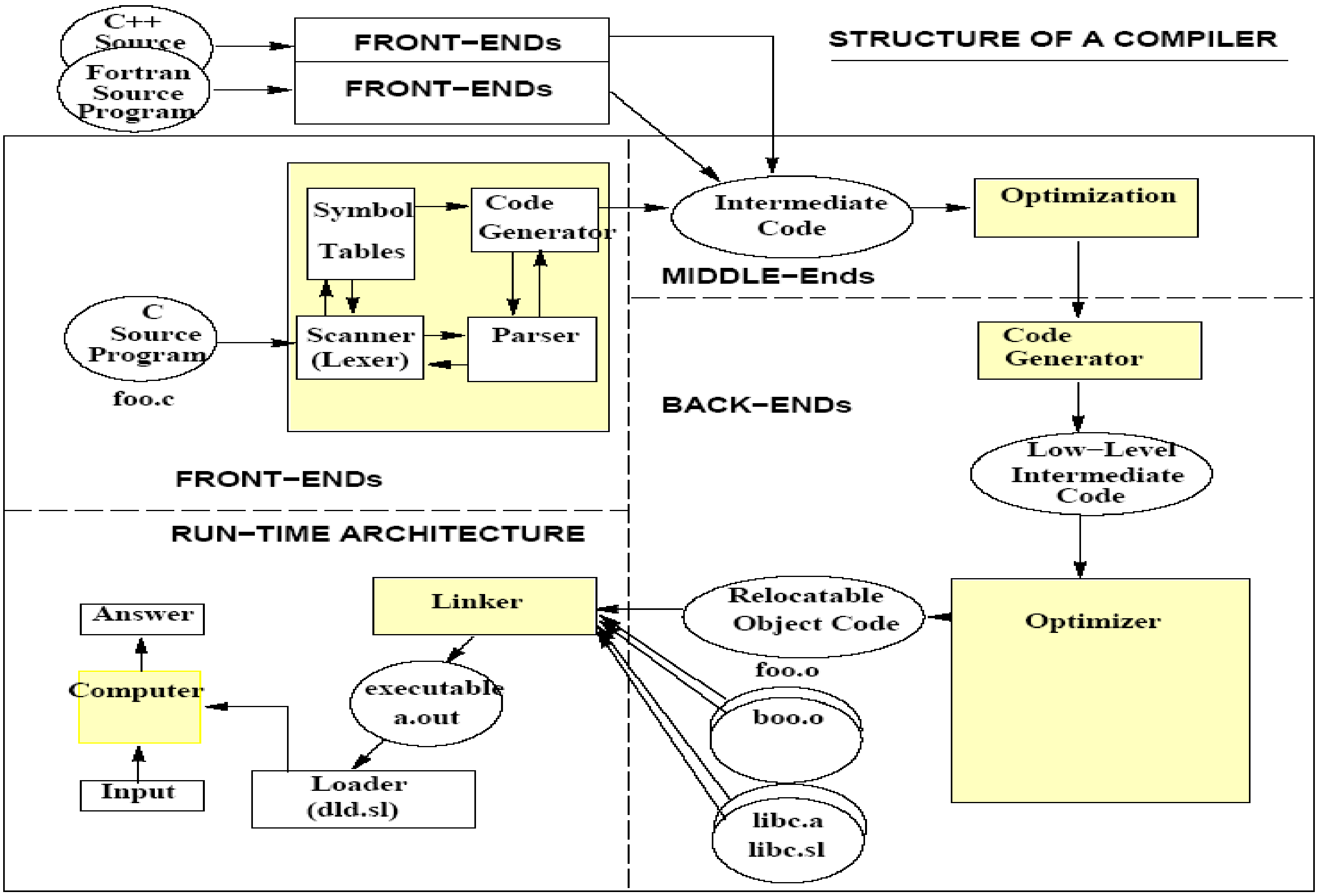
Language Processing System



Structure of Modern Compilers

- Requires the analysis of the source language and the synthesis of the target language
  - Analysis: Front-end
    - Lexical, syntactic, semantic with symbol table
  - Synthesis: Back-end
    - Intermediate code (representation) generation
      - E.g., P-code (Pascal), U-code, bytecode, parse tree,…
    - machine-independent optimization
    - Machine code generation and optimization
  - Runtime architecture
    - Linking, loading, shared libraries

Structure of Modern Compilers



Two Viewpoints of Compilers

- Compilers interact both with programming languages and with processor architectures
- Therefore, compilers affect
  - Programming language (PL) design
  - Processor architecture (ISA) design

Compilers and PL Design

- PL feature and compiler techniques
  - Virtual methods (C++, Java): dispatch table
  - Non-locals (Pascal): static links
  - Automatic memory deallocation (Lisp, Java): garbage collection (GC)
  - Call-by-name (Algol): thunks

  Static links, GC, thunks are expensive: not in C

Compilers and ISA

- Old wisdom
  - CPUs have CISC ISA and compiler tries to generate CISC code
- Current wisdom
  - CPUs provide orthogonal RISC ISA and the compiler (optimizer) make the best use of these instructions for better performance
- It is not easy to generate complex CISC instructions; e.g.,

```
int A[10]; for for (i = 1; i < 10; i++) X += A[i];
```

  - VAX ISA has a CISC instruction to get the address of A[i]
    `Index(A,i,low,high)`: if (low<=i<= high) return (A+4*i) else error;
  - RISC ISA will do the same using simple additions/multiplications
- RISC H/W is simpler without complex instructions, while optimizing compiler generates high-performance code

Compiler Optimizations

- Compiler Optimization
  - Transform a computation to an equivalent but better computation
  - Not actually optimal

What Can an Optimizer Do?

- Execution time of a program is decided by
  - Instruction count (# of instructions executed)
  - CPI (Average # of cycles/instruction)
  - Cycle time of the machine
- Compiler can reduce the first two items

How?

- Reduce the # of instructions in the code
- Replace expensive instructions with simpler ones (e.g., replace multiply by add or shift)
- Reduce cache misses (both instruction and data accesses)
- Grouping independent instructions for parallel execution (for superscalar or EPIC)

- Sometimes reducing the size of object code (e.g., for DSPs or embedded microcontrollers)

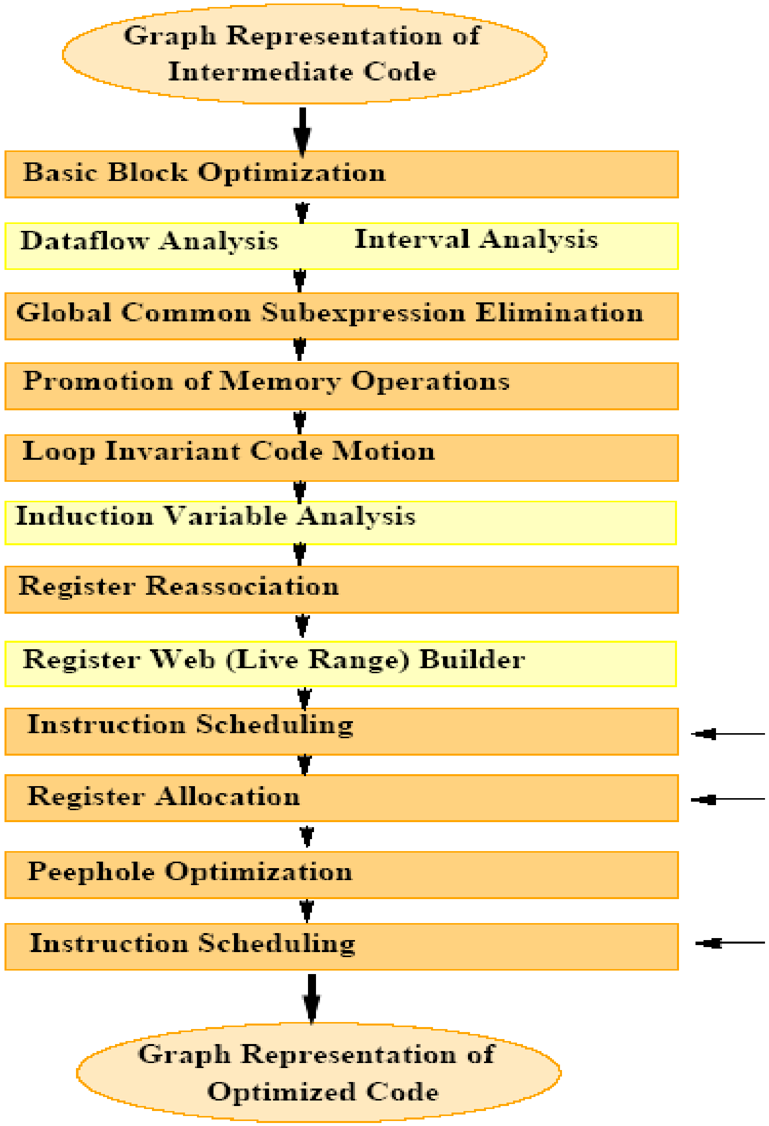
Why Optimizations Interesting ?

- Seriously affects computer performance
  - Overall performance of a program is determined by H/W performance and by quality of its code
  - H/W is fixed once it is released while compiler optimizations keep improving the performance (e.g., SPEC numbers)
  - Many architectural features are primarily controlled by compiler
    - e.g., prefetch instructions, EPIC, non-blocking caches, ...
- An example of a large software system
  - Problem solving: find common cases, formulate mathematically, develop algorithm, implement, evaluate on real data
  - Software engineering Issues
    - Hard to maintain and debug (why? Compiler output is code)

Structure of Modern Optimizers

- Phase-by-phase structure
  - Better code as phases proceed
  - Phase ordering problem
  - Register allocation is most time consuming
    - Based on graph coloring which is NP-complete
- Optimization levels
  - -O1: basic optimizations only
  - -O2 (which is -O): stable optimizations
  - -Ox (x>2): aggressive but not always stable

Structure of Optimizing Compilers



What can Optimizations do for You?

- Let's see an example: a bubble sort program

```
#define N 100
main ()
{
 int A[N], i, j;
 int temp;
 for (i = N-1; i >= 0; i--)
   for (j = 0; j < i; j++)
   {
   if (A[j] > A[j+1]) {
         temp = A[j];
         A[j] = A[j+1];
         A[j+1] = temp;
   }
 }
}
```

– We compiled with/without optimizations for the PA-RISC CPU
  - `cc -S bubblesort.c`
  - `cc -O -S bubblesort.c`

```
LDI      99,%r1
STW      %r1,-48(%r30)   ; 99->i
LDW      -48(%r30),%r31
COMIBF,<=,N 0,%r31,$002; i>=0 ?
$003
STW      %r0,-44(%r30)   ; 0->j
LDW      -44(%r30),%r19
LDW      -48(%r30),%r20
COMBF,<,N %r19,%r20,$001;j<i ?
$006
LDW      -44(%r30),%r21
LDO      1(%r21),%r22    ; j+1
LDO      -448(%r30),%r1 ; &A
LDW      -44(%r30),%r31 ; j
LDWX,S   %r31(%r1),%r19 ; A[j]
LDO      -448(%r30),%r20; &A
LDWX,S   %r22(%r20),%r21; A[j+1]
COMB,<=,N %r19,%r21,$004;A[j]<A[j+1]
LDO      -448(%r30),%r22 ;&A
LDW      -44(%r30),%r1    ; j
LDWX,S   %r1(%r22),%r31   ; A[j]
STW      %r31,-40(%r30) ;A[j]->temp
LDW      -44(%r30),%r19
LDO      1(%r19),%r20     ; j+1
LDO      -448(%r30),%r21 ; &A
```

```
|   LDWX,S  %r20(%r21),%r22; A[j+1]
|   LDW     -44(%r30),%r1   ; j
|   LDO     -448(%r30),%r31; &A
|   SH2ADD  %r1,%r31,%r19   ; A[j]
|   STWS    %r22,0(%r19);A[j+1]->A[j]
|   LDW     -44(%r30),%r20;
|   LDO     1(%r20),%r21
|   LDW     -40(%r30),%r22
|   LDO     -448(%r30),%r1
|   SH2ADD  %r21,%r1,%r31
|   STWS    %r22,0(%r31);temp->A[j+1]
|$004
|   LDW     -44(%r30),%r19 ; j
|   LDO     1(%r19),%r20    ; j++
|   STW     %r20,-44(%r30)
|   LDW     -44(%r30),%r21
|   LDW     -48(%r30),%r22 ; i
|   COMB,<  %r21,%r22,$006  ; j<i ?
|   NOP
|$001
|   LDW     -48(%r30),%r1   ; i
|   LDO     -1(%r1),%r31    ; i--
|   STW     %r31,-48(%r30)  ;
|   LDW     -48(%r30),%r19
|   COMIB,<=  0,%r19,$003  ; i>=0 ?
```

Optimized Assembly Code

```
    LDI        99,%r31
$003
    COMBF,<,N          %r0,%r31,$001
    LDO        -444(%r30),%r23
    SUBI       0,%r31,%r24
$006
    LDWS       -4(%r23),%r25
    LDWS,MA 4(%r23),%r26
    COMB,<=,N          %r25,%r26,$007
    STWS       %r26,-8(%r23)
    STWS       %r25,-4(%r23)
$007
    ADDIB,<,N          1,%r24,$006+4
    LDWS       -4(%r23),%r25
$001
    ADDIBF,<           -1,%r31,$003
    NOP
$002
```

- Compare the Number of Instructions in the Loop!

What you can get from this class?

- Understanding of compilation technology
- Make yourself familiar with
  - lex and yacc
  - compilation tools
  - gcc tool set
- Understanding code optimizations, virtual machine technology, garbage collection