GCC: From C to Executable

by
W. E. Cohen
(wcohen@redhat.com)
GCC Engineer
Red Hat, Inc.

Abstract

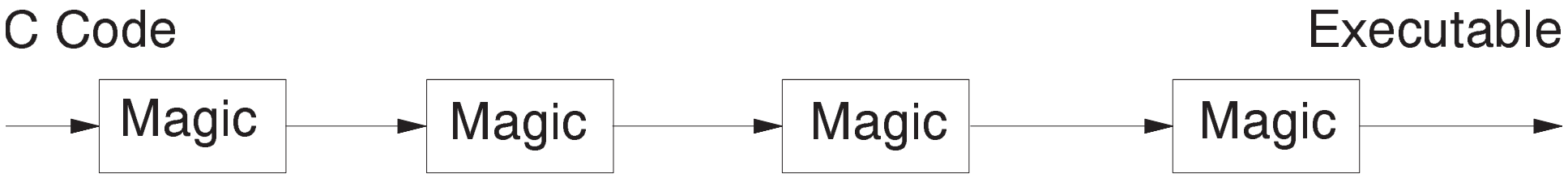
- The GNU C Compiler (GCC) enables operating systems and application programs written in high-level languages such as Fortran and C to be ported to numerous computer architectures. GCC, and compilers in general, are often viewed as large, incomprehensible software systems. This talk will divide the GCC compiler into smaller, more comprehensible components. Examples will illustrate the steps GCC takes to convert a high-level program written in C into executable machine code.

Introduction

- Many view compilers as magical, incomprehensible programs
- What does a compiler really do?
- How does a compiler convert C (or other languages) into executable?

Compiler Block Diagram

Revised Compiler Block Diagram



C Model vs. Real Machine

- Arbitrary variable names
- Rules for name scoping
- Everything in memory
- Fancy operations, e.g. floating-point (FP) arith.
- Model uniform across computers

- All numerical addresses
- No address scoping
- Different places to store data, e.g. registers
- Restrictions on storage location and operations
- Limited arithmetic operations, e.g. no FP
- Significant differences between processors, e.g. PowerPC vs. x86

Simple Example Program

- Program:

```
/* hello.c, a simple example program */
 #define GREETING "Hello, Everyone!\n"

 main()
 {
    printf(GREETING);
 }
```

- Compilation:

```
gcc hello.c
```

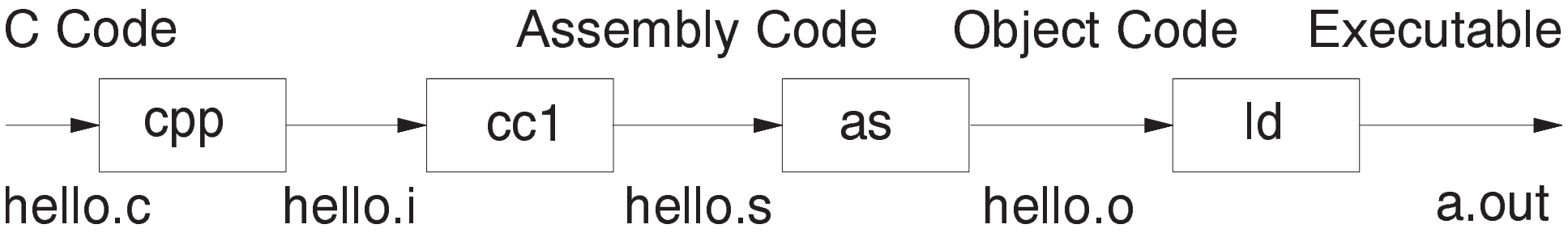
- Execution:

```
[cohen@univega lilax]$ a.out
 Hello, Everyone!
```

GCC: A Set of Programs

- Need to manage complexity
- Divide the compiler in more reasonable sized parts
- Driver program gcc calls "right" program

Program	Job
cpp	C preprocessor
cc1	C compiler
as	Assembler
ld	Linker



C Preprocessor

- Strips out comments
- Performs textual substitution
- Preprocessor Output:

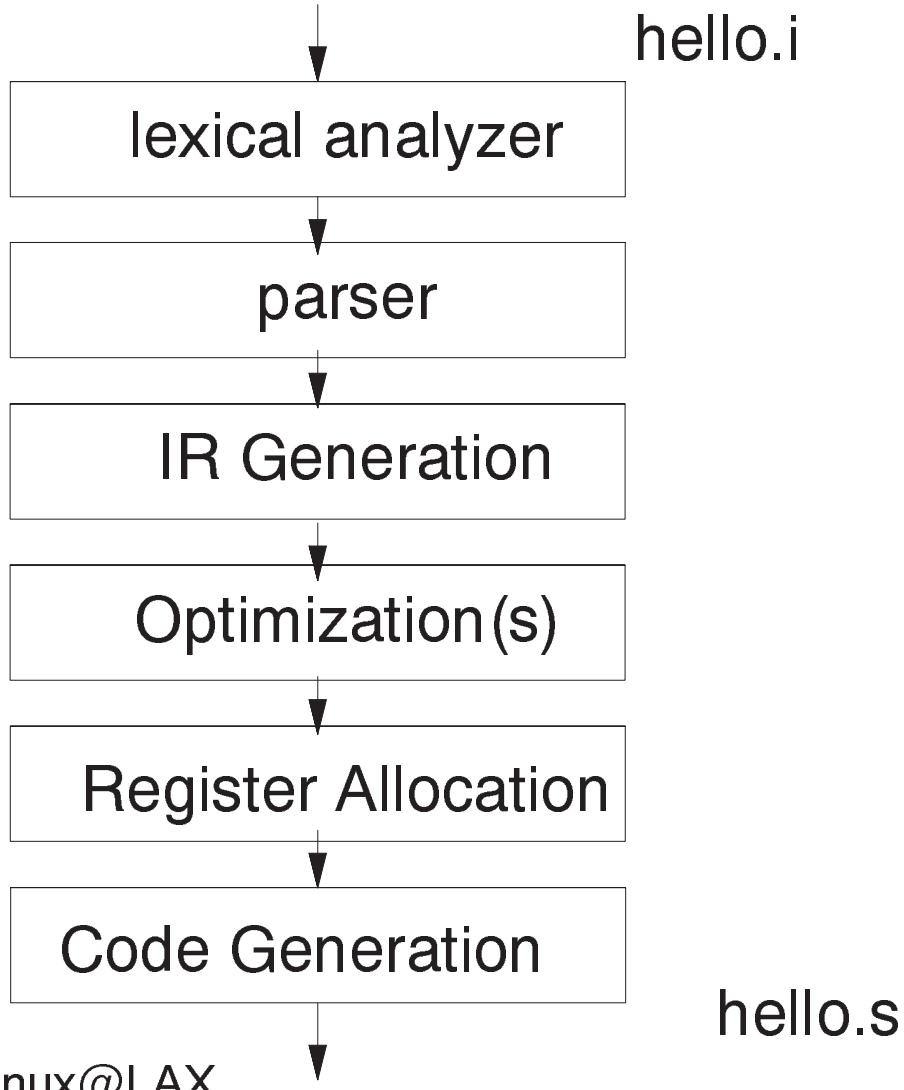
```
# 1 "hello.c"



main()
{
  printf("Hello, Everyone!\n");
}
```

CC1

- Translates preprocessed C into assembly language
- Broken into a smaller steps



Lexical Analyzer

- Inconvenient to handle individual characters, e.g. 3.14159 as a number not seven characters
- Groups individual characters into tokens

``#1 \"hello.c\``	CPP_EQ
``main"	CPP_NAME
``("	CPP_OPEN_PAREN
``)"	CPP_CLOSE_PAREN
``{"	CPP_OPEN_BRACE
``printf"	CPP_NAME
``("	CPP_OPEN_PAREN
``\``Hello, Everyone!\n\""	CPP_STRING
``)"	CPP_CLOSE_PAREN
``;"	CPP_SEMICOLON
``}"	CPP_CLOSE_BRACE

Parser

- Groups tokens into structures, e.g. expressions, statements, and functions
- Finds syntax errors
- Generates straightforward Intermediate Representation (IR)

Intermediate Representation

- Called Register Transfer Language (RTL)
- Based on registers
- Describes operations, e.g. additions, subtractions, and data movement
- Used to pass information between phases of compiler

Example's Initial Intermediate Representation

```
;; Function main
(note 2 0 3 NOTE_INSN_DELETED 0)
(note 3 2 6 NOTE_INSN_FUNCTION_BEG 0)
(note 6 3 9 0x401562a0 NOTE_INSN_BLOCK_BEG 0)
(insn 9 6 11 (parallel[
            (set (reg:SI 7 esp)
                (plus:SI (reg:SI 7 esp)
                    (const_int -12 [0xfffffff4])))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))
(insn 11 9 12 (set (mem/f:SI (pre_dec:SI (reg:SI 7 esp)) 0)
        (symbol_ref:SI ("*.LC0"))) -1 (nil)
    (nil))
(call_insn 12 11 14 (set (reg:SI 0 eax)
        (call (mem:QI (symbol_ref:SI ("printf")) 0)
            (const_int 16 [0x10]))) -1 (nil)
    (nil)
    (nil))
(insn 14 12 16 (parallel[
            (set (reg:SI 7 esp)
                (plus:SI (reg:SI 7 esp)
                    (const_int 16 [0x10])))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (nil))
(note 16 14 17 0x401562a0 NOTE_INSN_BLOCK_END 0)
(note 17 16 19 NOTE_INSN_FUNCTION_END 0)
(insn 19 17 20 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(code_label 20 19 21 2 "" "" [0 uses])
(insn 21 20 0 (use (reg/i:SI 0 eax)) -1 (nil)
    (nil))
```

Conversion of RTL into Assembly Language

- Phases to convert initial RTL to assembly:
  - hello.c.00.rtl initial RTL generated
  - hello.c.02.jump
  - hello.c.04.addressof
  - hello.c.10.cfg
  - hello.c.12.life
  - hello.c.17.lreg
  - hello.c.18.greg
  - hello.c.19.flow2
  - hello.c.28.stack
  - hello.s instructions selected

Jump Phase

- Removes:
  - Jumps to jumps
  - Labels that are not referenced in the code

Addressof Phase

- Finds register lifetime information:
  - First use of register
  - Last use of register
- Data useful in later phases

Conversion of RTL into Assembly Language

- Phases to convert initial RTL to assembly:
  - hello.c.00.rtl initial RTL generated
  - hello.c.02.jump
  - hello.c.04.addressof
  - hello.c.10.cfg
  - hello.c.12.life
  - hello.c.17.lreg
  - hello.c.18.greg
  - hello.c.19.flow2
  - hello.c.28.stack
  - hello.s instructions selected

Control Flow Graph (CFG) Phase

- Finds Basic Blocks
- What is a basic block? A group of instructions that have a label at the beginning and either a label or jump at the end. Thus, if one instruction in a basic block is executed, all the instructions in the basic block are executed
- Finds what registers hold useful values at beginning and end of each basic block

Local and Global Register Allocation Phase

- Register allocation maps registers in intermediate representation to real processor registers
- Local register allocation picks registers to store information in a basic block
- Global Register allocation picks register to store information across basic blocks

Conversion of RTL into Assembly Language

- Phases to convert initial RTL to assembly:
  - hello.c.00.rtl initial RTL generated
  - hello.c.02.jump
  - hello.c.04.addressof
  - hello.c.10.cfg
  - hello.c.12.life
  - hello.c.17.lreg
  - hello.c.18.greg
  - hello.c.19.flow2
  - hello.c.28.stack
  - hello.s instructions selected

Stack Phase

- IA-32 (x86) specific phase in GCC
- Pass fixes code to better match the stack-oriented floating point hardware on IA-32

Code Generation

- Machine description (i386.md) contains description of available instructions
- Template that matches insn with call to printf:

```
(define_insn "*call_value_0"
  [(set (match_operand 0 "" "")
   (call (mem:QI (match_operand:SI 1 "constant_call_address_operand" ""))
         (match_operand:SI 2 "" "")))]
  ""
  "*
{
  if (SIBLING_CALL_P (insn))
    return \"jmp\n %P1\";
  else
    return \"call\\t%P1\";
}"
  [(set_attr "type" "callv")])
```

Output of CC1

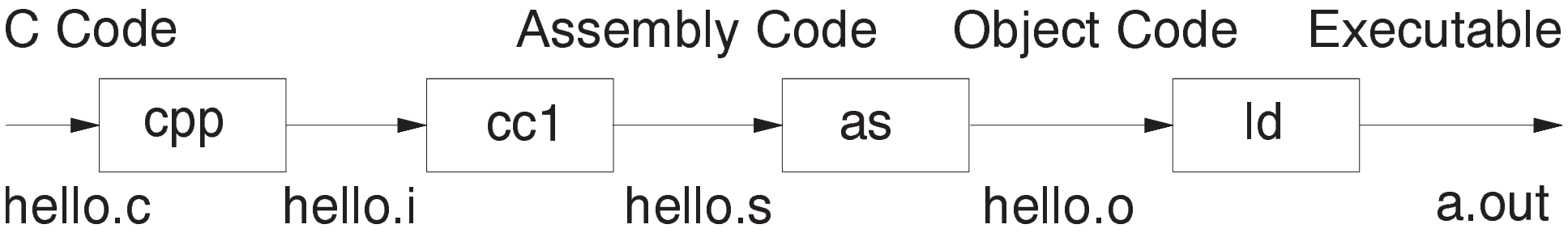
- Assembly language for the the appropriate processor

```
    .file   "hello.c"
    .version "01.01"
gcc2_compiled.:
        .section .rodata
.LC0:
    .string  "Hello, Everyone!\n"
.text
    .align 4
.globl main
    .type    main,@function
main:
    pushl   %ebp
    movl%esp, %ebp
    subl$8, %esp
    subl$12, %esp
    pushl   $.LC0
    callprintf
    addl$16, %esp
    leave
    ret
.Lfe1:
    .size    main,.Lfe1-main
    .ident   "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.0)"
```

GCC: A Set of Programs

- Need to manage complexity
- Divide the compiler in more reasonable sized parts
- Driver program gcc calls "right" program

Program	Job
cpp	C preprocessor
cc1	C compiler
as	Assembler
ld	Linker



AS Assembler

- Converts assembly language into object files
- Cannot run object files, they aren't complete
  - Need to fix up addresses, e.g. function printf
  - Need start up code (code that runs before main)

```
hello.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0: 55                                push   %ebp
   1: 89 e5                             mov    %esp,%ebp
   3: 83 ec 08                          sub    $0x8,%esp
   6: 83 ec 0c                          sub    $0xc,%esp
   9: 68 00 00 00 00                    push   $0x0
   e: e8 fc ff ff ff                    call   f <main+0xf>
  13: 83 c4 10                          add    $0x10,%esp
  16: c9                                leave
  17: c3                                ret
```

LD Linker

- Fixes unresolved addresses, e.g. calls
- Puts in startup code (crt0.o)
- Creates executable file
- File named a.out by default

Optimization

- By default GCC generates straightforward code
- Significant improvements in code quality can be obtained
- By default optimization is off, turned on with "-O2"
- Optimizations performed at the RTL level:
  - Avoid redundant computations
  - Eliminate dead code
  - Reduce number of slow memory references
  - Make better use of registers in processor
- Optimizations performed at the instruction level
  - Replace sequences of instructions with better idoms
  - Reorder instructions to keep processor busier

Debugging Information

- The debugger (gdb) needs information to map machine code back to original program
- By default none generated, turn on with "-g"
  With debug option .s file will have .stabn and .stabs lines with additional information
- Information maps:
  - code addresses to lines in source code
  - data addresses to variables
  - register to variable
  - data in memory into meaningful data, e.g. float, double, and struct

"Home Experiments"

- Find out what the compiler is really doing:
  gcc -v hello.c
- See data CC1 passes between different steps of compilation:
  gcc -save-temps -dap hello.c
- Where to get current GCC source code:
  http://gcc.gnu.org/