Generating Code and Running Programs

Stephen A. Edwards

Columbia University

Fall 2011
Part I

The Compilation Process
A Long K’s Journey into Byte†

Compiler front end

Source code

↓

Parser/Semantic Analysis

AST

↓

Intermediate code generation

Compiler back end

Intermediate code

↓

Optimization

Assembly Code

Assembler

↓

Assemble

Relocatable Object Code

Linker

↓

Link

Executable

Loader

↓

Relocate

In-memory image

†Apologies to O’Neill
The front end focuses on *analysis*: 

- Lexical analysis
- Parsing
- Static semantic checking
- AST generation

The back end focuses on *synthesis*: 

- Translation of the AST into intermediate code
- Optimization
- Generation of assembly code
Portable Compilers

Building a compiler a large undertaking; most try to leverage it by making it portable.

Diagram:
- C
- C++
- FORTRAN
- Objective C
- Ada-95
- Pascal
- MIPS
- SPARC
- x86
- Alpha
- 68k
- PPC
Portable Compilers

Building a compiler a large undertaking; most try to leverage it by making it portable.

Language-specific Frontends

Processor-specific Backends
Part II

Intermediate Representations/Formats
```java
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

```java
# javap -c Gcd

Method int gcd(int, int)
0 goto 19

3 iload_1 // Push a
4 iload_2 // Push b
5 if_icmple 15 // if a <= b goto 15

8 iload_1 // Push a
9 iload_2 // Push b
10 isub // a - b
11 istore_1 // Store new a
12 goto 19

15 iload_2 // Push b
16 iload_1 // Push a
17 isub // b - a
18 istore_2 // Store new b

19 iload_1 // Push a
20 iload_2 // Push b
21 if_icmpne 3 // if a != b goto 3

24 iload_1 // Push a
25 ireturn // Return a
```
Stack-Based IRs

Advantages:

- Trivial translation of expressions
- Trivial interpreters
- No problems with exhausting registers
- Often compact

Disadvantages:

- Semantic gap between stack operations and modern register machines
- Hard to see what communicates with what
- Difficult representation for optimization
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
Register-Based IRs

*Most common type of IR*

Advantages:
- Better representation for register machines
- Dataflow is usually clear

Disadvantages:
- Slightly harder to synthesize from code
- Less compact
- More complicated to interpret
Part III

Introduction to Optimization
int gcd(int a, int b) {
    while (a != b) {
        if (a < b) b -= a;
        else a -= b;
    }
    return a;
}
Typical Optimizations

- Folding constant expressions
  \(1 + 3 \rightarrow 4\)

- Removing dead code
  ```
  if (0) {...} \rightarrow \text{nothing}
  ```

- Moving variables from memory to registers
  ```
  ld [%fp+68], %i1
  sub %i0, %i1, %i0 \rightarrow sub %o1, %o0, %o1
  st %i0, [%fp+72]
  ```

- Removing unnecessary data movement

- Filling branch delay slots (Pipelined RISC processors)

- Common subexpression elimination
Machine-Dependent vs. -Independent Optimization

No matter what the machine is, folding constants and eliminating dead code is always a good idea.

\[ a = c + 5 + 3; \]
\[ \text{if } (0 + 3) \{ \]
\[ \quad b = c + 8; \quad \rightarrow \quad b = a = c + 8; \]
\[ \} \]

However, many optimizations are processor-specific:

- Register allocation depends on how many registers the machine has
- Not all processors have branch delay slots to fill
- Each processor’s pipeline is a little different
Basic Blocks

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a < b) b -= a;
        else a -= b;
    }
    return a;
}
```

The statements in a basic block all run if the first one does.

Starts with a statement following a conditional branch or is a branch target.

Usually ends with a control-transfer statement.
Control-Flow Graphs

A CFG illustrates the flow of control among basic blocks.

A:
sne t, a, b
bz E, t
slt t, a, b
bnz B, t
sub b, b, a
jmp C

B:
sub a, a, b
jmp C

C:
jmp A

E:
ret a

from A

A:
sne t, a, b
bz E, t
slt t, a, b
bnz B, t
sub b, b, a
jmp C

B:
sub a, a, b
jmp C

C:
jmp A

ult E:
ret a
Part IV

Assembly Code and Assemblers
Assembly Code

Most compilers produce assembly code: easy to debug.

! gcd on the SPARC

gcd:

```
  cmp %o0, %o1
  be  .LL8
  nop
  .LL9:
    ble,a .LL2
    sub %o1, %o0, %o1
    sub %o0, %o1, %o0
  .LL2:
    cmp %o0, %o1
    bne .LL9
    nop
  .LL8:
    retl
  nop
```

No operation
Role of an Assembler

Translate opcodes + operand into byte codes

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction code</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 80A20009</td>
<td>cmp    %o0, %o1</td>
<td></td>
</tr>
<tr>
<td>0004 02800008</td>
<td>be     .LL8</td>
<td></td>
</tr>
<tr>
<td>0008 01000000</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000c 24800003</td>
<td>ble,a   .LL2</td>
<td></td>
</tr>
<tr>
<td>0010 92224008</td>
<td>sub    %o1, %o0, %o1</td>
<td></td>
</tr>
<tr>
<td>0014 90220009</td>
<td>sub    %o0, %o1, %o0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018 80A20009</td>
<td>cmp    %o0, %o1</td>
<td></td>
</tr>
<tr>
<td>001c 12BFFFFFC</td>
<td>bne    .LL9</td>
<td></td>
</tr>
<tr>
<td>0020 01000000</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0024 81C3E008</td>
<td>retl</td>
<td></td>
</tr>
<tr>
<td>0028 01000000</td>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>
Encoding Example

```
sub   %o1, %o0, %o1
```

Encoding of “SUB” on the SPARC:

<table>
<thead>
<tr>
<th></th>
<th>rd</th>
<th>000100</th>
<th>rs1</th>
<th>0</th>
<th>reserved</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>29</td>
<td>24</td>
<td>18</td>
<td>13</td>
<td>12</td>
<td>4</td>
</tr>
</tbody>
</table>

rd = %o1 = 01001
rs1 = %o1 = 01001
rs2 = %o0 = 00100

10 01001 000100 01001 0 00000000 01000
1001 0010 0010 0010 0100 0000 0000 1000

= 0x92228004
Role of an Assembler

Transforming symbolic addresses to concrete ones.

Example: Calculating PC-relative branch offsets.

000c 24800003  ble, a .LL2
0010 92224008  sub  %o1, %o0, %o1
0014 90220009  sub  %o0, %o1, %o0

.LL2:
0018 80A20009  cmp  %o0, %o1
Role of an Assembler

Most assemblers are “two-pass” because they can’t calculate everything in a single pass through the code.

```assembly
.LL9:
  000c  24800003  ble,a .LL2
  0010  92224008  sub  %o1, %o0, %o1
  0014  90220009  sub  %o0, %o1, %o0

.LL2:
  0018  80A20009  cmp  %o0, %o1
  001c  12BFFFFFC  bne .LL9
```

Don’t know offset of LL2

Know offset of LL9
Role of an Assembler

Constant data needs to be aligned.

```c
char a[] = "Hello";
int b[3] = { 5, 6, 7 };
```

- `.section ".data"` ! “This is data”
- `.global a` ! “Let other files see a
- `.type a,#object` ! “a is a variable”
- `.size a,6` ! “six bytes long”

```
a:
0000 48656C6C .asciz "Hello" ! zero-terminated ASCII
```

Bytes added to ensure alignment

```
0006 0000 .global b
.align 4
.type b,#object
.size b,12
```

```
b:
0008 00000005 .uaword 5
000c 00000006 .uaword 6
0010 00000007 .uaword 7
```
Role of an Assembler

The MIPS has pseudoinstructions:

“Load the immediate value 0x12345abc into register 14:”

```
li $14, 0x12345abc
```

expands to

```
lui $14, 0x1234
ori $14, 0x5abc
```

“Load the upper 16 bits, then OR in the lower 16”

MIPS instructions have 16-bit immediate values at most

RISC philosophy: small instructions for common case
Part V

Optimization: Register Allocation
Optimization: Register Allocation

Where to put temporary results? The easiest is to put everything on the stack.

```c
int bar(int g, int h, int i,
      int j, int k, int l)
{
    int a, b, c, d, e, f;
    a = foo(g);
    b = foo(h);
    c = foo(i);
    d = foo(j);
    e = foo(k);
    f = foo(l);
    return a + b + c + d + e + f;
}
```
Quick Review of the x86 Architecture

Eight “general-purpose” 32-bit registers:
eax ebx ecx edx ebp esi edi esp

esp is the stack pointer
ebp is the base (frame) pointer

addl %eax, %edx  eax + edx → edx

Base-pointer-relative addressing:

movl 20(%ebp), %eax  Load word at ebp+20 into eax
Unoptimized GCC on the x86

movl 24(%ebp),%eax   % Get k
pushl %eax   % Push argument
call foo   % e = foo(k);
addl $4,%esp   % Make room for e
movl %eax,%eax   % Does nothing
movl %eax,-20(%ebp)   % Save return value on stack

movl 28(%ebp),%eax   % Get l
pushl %eax   % Push argument
call foo   % f = foo(l);
addl $4,%esp   % Make room for f
movl %eax,%eax   % Does nothing
movl %eax,-24(%ebp)   % Save return value on stack

movl -20(%ebp),%eax   % Get f
movl -24(%ebp),%edx   % Get e
addl %edx,%eax   % e + f
movl %eax,%edx   % Accumulate in edx
addl -16(%ebp),%edx   % d + (e+f)
movl %edx,%eax   % Accumulate in edx

int bar(int g, int h, int i, int j, int k, int l)
{
    int a, b, c, d, e, f;
    a = foo(g);
    b = foo(h);
    c = foo(i);
    d = foo(j);
    e = foo(k);
    f = foo(l);
    return a + b + c + d + e + f;
}
Optimized GCC on the x86

```assembly
movl 20(%ebp),%edx  % Get j
pushl %edx          % Push argument
call foo            % d = foo(j);
movl %eax,%esi     % save d in esi

movl 24(%ebp),%edx  % Get k
pushl %edx          % Push argument
call foo            % e = foo(k);
movl %eax,%ebx     % save e in ebx

movl 28(%ebp),%edx  % Get l
pushl %edx          % Push argument
call foo            % f = foo(l);
addl %ebx,%eax      % e + f
addl %esi,%eax      % d + (e+f)
```

```c
int bar(int g, int h, int i, int j, int k, int l)
{
    int a, b, c, d, e, f;
    a = foo(g);
    b = foo(h);
    c = foo(i);
    d = foo(j);
    e = foo(k);
    f = foo(l);
    return a + b + c + d + e + f;
}
```
Unoptimized vs. Optimized

```assembly
movl 20(%ebp),%edx
pushl %edx
call foo
movl %eax,%esi

movl 24(%ebp),%eax
pushl %eax
call foo
addl $4,%esp
movl %eax,%eax
movl %eax,-20(%ebp)

movl 28(%ebp),%eax
pushl %eax
call foo
addl $4,%esp
movl %eax,%eax
movl %eax,-24(%ebp)

movl -20(%ebp),%eax
movl -24(%ebp),%edx
addl %edx,%eax
movl %eax,%edx
addl -16(%ebp),%edx
addl %esi,%eax
movl %edx,%eax
```

```c
int bar(int g, int h, int i, int j, int k, int l)
{
    int a, b, c, d, e, f;
    a = foo(g);
    b = foo(h);
    c = foo(i);
    d = foo(j);
    e = foo(k);
    f = foo(l);
    return a + b + c + d + e + f;
}
```
Part VI

Separate Compilation and Linking
Separate Compilation and Linking

Compiler
- foo.c -> foo.s -> foo.o
- bar.c -> bar.s -> bar.o

Assembler
- foo.c -> foo.s
- bar.c -> bar.s

Linker
- foo.o
- bar.o

Archiver
- printf.o
- fopen.o
- malloc.o

ld
- foo

cc
- foo.c
- bar.c

as
- foo.s
- bar.s

ar
- libc.a
Linking

Goal of the linker is to combine the disparate pieces of the program into a coherent whole.

file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}
```

file2.c:

```c
#include <stdio.h>
extern char a[];
static char b[6];

void bar() {
    strcpy(b, a);
    baz(b);
}
```

libc.a:

```c
int printf(char *s, ...)
{
    /* ... */
}

char *
strcpy(char *d, char *s)
{
    /* ... */
}
```
Linking

Goal of the linker is to combine the disparate pieces of the program into a coherent whole.

file1.c:
```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();
int main() {
    bar();
}
void baz(char *s) {
    printf("%s", s);
}
```

file2.c:
```c
#include <stdio.h>
extern char a[];
static char b[6];
void bar() {
    strcpy(b, a);
    baz(b);
}
```

libc.a:
```c
int printf(char *s, ...)
{
    /* ... */
}
char *
strcpy(char *d, char *s)
{
    /* ... */
}
```
Linking

file1.o
- a="Hello"
- main()
- baz()

file2.o
- char b[6]
- bar()
Linking

**a.out**
- .text segment
  - main()
  - baz()
- .data segment
  - a="Hello"
  - bar()
- .bss segment
  - char b[6]

**file1.o**
- a="Hello"
- main()
- baz()
- bar()

**file2.o**
- char b[6]
- bar()

`.text` Code of program
`.data` Initialized data
`.bss` Uninitialized data
“Block Started by Symbol”
Object Files

Relocatable: Many need to be pasted together. Final in-memory address of code not known when program is compiled

Object files contain

- imported symbols (unresolved “external” symbols)
- relocation information (what needs to change)
- exported symbols (what other files may refer to)
file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}
```
# objdump -x file1.o

Sections:

<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>VMA</th>
<th>LMA</th>
<th>Offset</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>038</td>
<td>0</td>
<td>0</td>
<td>034</td>
<td>2**2</td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>008</td>
<td>0</td>
<td>0</td>
<td>070</td>
<td>2**3</td>
</tr>
<tr>
<td>2</td>
<td>.bss</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>078</td>
<td>2**0</td>
</tr>
<tr>
<td>3</td>
<td>.rodata</td>
<td>008</td>
<td>0</td>
<td>0</td>
<td>078</td>
<td>2**3</td>
</tr>
</tbody>
</table>

SYMBOL TABLE:

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Section</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>g</td>
<td>.data</td>
<td>a</td>
<td>006</td>
</tr>
<tr>
<td>0000</td>
<td>g</td>
<td>.text</td>
<td>main</td>
<td>014</td>
</tr>
<tr>
<td>0000</td>
<td><em>UND</em></td>
<td></td>
<td>bar</td>
<td>000</td>
</tr>
<tr>
<td>0014</td>
<td>g</td>
<td>.text</td>
<td>baz</td>
<td>024</td>
</tr>
<tr>
<td>0000</td>
<td><em>UND</em></td>
<td></td>
<td>printf</td>
<td>000</td>
</tr>
</tbody>
</table>

RELOCATION RECORDS FOR [.text]:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004</td>
<td>R_SPARC_WDISP30</td>
<td>bar</td>
</tr>
<tr>
<td>001c</td>
<td>R_SPARC_HI22</td>
<td>.rodata</td>
</tr>
<tr>
<td>0020</td>
<td>R_SPARC_LO10</td>
<td>.rodata</td>
</tr>
<tr>
<td>0028</td>
<td>R_SPARC_WDISP30</td>
<td>printf</td>
</tr>
</tbody>
</table>

file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}
```
Object Files

#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}

# objdump -d file1.o
0000 <main>:
    0: 9d e3 bf 90 save %sp, -112, %sp
    4: 40 00 00 00 call 4 <main+0x4>
        4: R_SPARC_WDISP30 bar
    8: 01 00 00 00 nop
    c: 81 c7 e0 08 ret
10: 81 e8 00 00 restore

0014 <baz>:
    14: 9d e3 bf 90 save %sp, -112, %sp
    18: f0 27 a0 44 st %i0, [ %fp + 0x44 ]
    1c: 11 00 00 00 sethi %hi(0), %o0
        1c: R_SPARC_HI22 .rodata
    20: 90 12 20 00 mov %o0, %o0
        20: R_SPARC_LO10 .rodata
    24: d2 07 a0 44 ld [ %fp + 0x44 ], %o1
    28: 40 00 00 00 call 28 <baz+0x14>
        28: R_SPARC_WDISP30 printf
    2c: 01 00 00 00 nop
    30: 81 c7 e0 08 ret
    34: 81 e8 00 00 restore
Before and After Linking

- Combine object files
- Relocate each function's code
- Resolve previously unresolved symbols

```c
int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}
```
Linking Resolves Symbols

file1.c:

```c
#include <stdio.h>
char a[] = "Hello";
extern void bar();

int main() {
    bar();
}

void baz(char *s) {
    printf("%s", s);
}
```

file2.c:

```c
#include <stdio.h>
extern char a[];
static char b[6];

void bar() {
    strcpy(b, a);
    baz(b);
}
```
Part VII

Shared Libraries and Dynamic Linking
Shared Libraries and Dynamic Linking

The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

Under a *static linking* model, each executable using a library gets a copy of that library’s code.

Address 0:

```
libXaw.a
libX11.a
xeyes
```

```
xterm
```

```
libXaw.a
libX11.a
xclock
```
The 1980s GUI/WIMP revolution required many large libraries (the Athena widgets, Motif, etc.)

Under a *static linking* model, each executable using a library gets a copy of that library's code.

Address 0:

```
libXaw.a
libX11.a
xterm
```

Wasteful: running many GUI programs at once fills memory with nearly identical copies of each library.

Something had to be done: another level of indirection.
Shared Libraries: First Attempt

Most code makes assumptions about its location.

First solution (early Unix System V R3) required each shared library to be located at a unique address:

<table>
<thead>
<tr>
<th>libXaw.so</th>
<th>libXaw.so</th>
<th>libXm.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>libX11.so</td>
<td>libX11.so</td>
<td>libX11.so</td>
</tr>
<tr>
<td>xeyes</td>
<td>xterm</td>
<td>netscape</td>
</tr>
</tbody>
</table>
Shared Libraries: First Attempt

Most code makes assumptions about its location.

First solution (early Unix System V R3) required each shared library to be located at a unique address:

| Address 0: | xeyes | | libXm.so |
| libXaw.so | libXaw.so | | |
| libX11.so | libX11.so | libX11.so |
| xterm | netscape | |

Obvious disadvantage: must ensure each new shared library located at a new address.

Works fine if there are only a few libraries; tended to discourage their use.
Problem fundamentally is that each program may need to see different libraries \textit{each at a different address}.
Position-Independent Code

Solution: Require the code for libraries to be position-independent. Make it so they can run anywhere in memory.

As always, add another level of indirection:

- All branching is PC-relative
- All data must be addressed relative to a base register.
- All branching to and from this code must go through a jump table.
Position-Independent Code for bar()

Normal unlinked code

```assembly
save %sp, -112, %sp
sethi %hi(0), %o0
    R_SPARC_HI22 .bss
mov %o0, %o0
    R_SPARC_LO10 .bss
sethi %hi(0), %o1
    R_SPARC_HI22 a
mov %o1, %o1
    R_SPARC_LO10 a
call 14
    R_SPARC_WDISP30 strcpy
nop
sethi %hi(0), %o0
    R_SPARC_HI22 .bss
mov %o0, %o0
    R_SPARC_LO10 .bss
call 24
    R_SPARC_WDISP30 baz
nop
ret
restore
```

gcc -fpic -shared

```assembly
save %sp, -112, %sp
sethi %hi(0x10000), %l7
call 8e0 ! add PC to %l7
add %l7, 0x198, %l7
ld [ %l7 + 0x20 ], %o0
ld [ %l7 + 0x24 ], %o1
    Actually just a stub
call 10a24 ! strcpy
nop
ld [ %l7 + 0x20 ], %o0
    call is PC-relative
call 10a3c ! baz
nop
ret
restore
```