Generating Code and Running Programs

Stephen A. Edwards

Columbia University

Summer 2014
Part I

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

Compiler front end

Source code
↓
Parser/Semantic Analysis
AST
↓
Intermediate code generation
IR
↓
Optimization

Compiler back end

Assembly Code
↓
Assemble
Relocatable Object Code
↓
Link
Executable
↓
Relocate
In-memory image

Assembler

Assembler

Linker

Loader

†Apologies to O’Neill
Compiler Frontends and Backends

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.
Portable Compilers

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

[Diagram showing IR at the center with branches to C, C++, FORTRAN, Objective C, Ada-95, Pascal, MIPS, SPARC, x86, Alpha, 68k, PPC. Language-specific_frontends: C, C++, FORTRAN, Objective C, Ada-95, Pascal. Processor-specific_backends: MIPS, SPARC, x86, Alpha, 68k, PPC.]
Part II

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

# 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;
}

gcd:
gcd._gcdTmp0:
    sne $vr1.s32 <- gcd.a,gcd.b
    seq $vr0.s32 <- $vr1.s32,0
    btrue $vr0.s32,gcd._gcdTmp1  // if!(a!=b) goto Tmp1

    sl $vr3.s32 <- gcd.b,gcd.a
    seq $vr2.s32 <- $vr3.s32,0
    btrue $vr2.s32,gcd._gcdTmp4  // if!(a<b) goto Tmp4

    mrk 2, 4  // Line number 4
    sub $vr4.s32 <- gcd.a,gcd.b
    mov gcd._gcdTmp2 <- $vr4.s32
    mov gcd.a <- gcd._gcdTmp2  // a=a-b
    jmp gcd._gcdTmp5

gcd._gcdTmp4:
    mrk 2, 6
    sub $vr5.s32 <- gcd.b,gcd.a
    mov gcd._gcdTmp3 <- $vr5.s32
    mov gcd.b <- gcd._gcdTmp3  // b=b-a

gcd._gcdTmp5:
    jmp gcd._gcdTmp0

gcd._gcdTmp1:
    mrk 2, 8
    ret gcd.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
```c
int gcd(int a, int b) {
    while (a != b) {
        if (a < b) b -= a;
        else a -= b;
    }
    return a;
}
```

<table>
<thead>
<tr>
<th>GCC on SPARC</th>
<th>GCC -O7 on SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc: save %sp, -112, %sp</td>
<td>gcc: cmp %o0, %o1</td>
</tr>
<tr>
<td>st %i0, [%fp+68]</td>
<td>be .LL8</td>
</tr>
<tr>
<td>st %i1, [%fp+72]</td>
<td>nop .LL8</td>
</tr>
<tr>
<td>.LL2: ld [%fp+68], %i1</td>
<td>.LL9: bge, a .LL2</td>
</tr>
<tr>
<td>ld [%fp+72], %i0</td>
<td>sub %o0, %o1, %o0</td>
</tr>
<tr>
<td>cmp %i1, %i0</td>
<td>sub %o1, %o0, %o1</td>
</tr>
<tr>
<td>bne .LL4</td>
<td>bne .LL9</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>b .LL3</td>
<td>.LL8: retl</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>.LL4: ld [%fp+68], %i1</td>
<td>.LL5: ld [%fp+68], %i0</td>
</tr>
<tr>
<td>ld [%fp+72], %i0</td>
<td>ld [%fp+72], %i1</td>
</tr>
<tr>
<td>cmp %i1, %i0</td>
<td>sub %i0, %i1, %i0</td>
</tr>
<tr>
<td>bge .LL5</td>
<td>st %i0, [%fp+72]</td>
</tr>
<tr>
<td>nop</td>
<td>b .LL2</td>
</tr>
<tr>
<td>ldl [%fp+72], %i0</td>
<td>nop .LL3</td>
</tr>
<tr>
<td>ld [%fp+68], %i1</td>
<td>b .LL2</td>
</tr>
<tr>
<td>sub %i0, %i1, %i0</td>
<td>nop .LL3</td>
</tr>
<tr>
<td>st %i0, [%fp+68]</td>
<td>.LL5: ld [%fp+68], %i0</td>
</tr>
<tr>
<td>b .LL2</td>
<td>ld [%fp+72], %i1</td>
</tr>
<tr>
<td>nop</td>
<td>sub %i0, %i1, %i0</td>
</tr>
<tr>
<td>.LL3: ld [%fp+68], %i0</td>
<td>st %i0, [%fp+68]</td>
</tr>
<tr>
<td>ret</td>
<td>b .LL2</td>
</tr>
<tr>
<td>restore</td>
<td>nop .LL3</td>
</tr>
</tbody>
</table>
Typical Optimizations

- Folding constant expressions
  \[1 + 3 \rightarrow 4\]

- Removing dead code
  \[\text{if (0) \{ \ldots \}} \rightarrow \text{nothing}\]

- Moving variables from memory to registers
  
  \[
  \text{ld} \ [%\text{fp}+68], \ %i1
  
  \text{sub} \ %i0, \ %i1, \ %i0 \rightarrow \text{sub} \ %o1, \ %o0, \ %o1
  
  \text{st} \ %i0, \ [%\text{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.

```c
a = c + 5 + 3;
if (0 + 3) {
    b = c + 8;
} → 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
Part IV

Assembly Code and Assemblers
Assembly Code

Most compilers produce assembly code: easy to debug.

! gcd on the SPARC  Comment

gcd:  Opcode
    cmp  %o0, %o1  Operand (a register)
    be  .LL8
    nop
    .LL9:  Label
        ble,a .LL2  Conditional branch to a label
        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>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 80A20009</td>
<td>gcd:</td>
<td>cmp %o0, %o1</td>
</tr>
<tr>
<td>0004 02800008</td>
<td></td>
<td>be .LL8</td>
</tr>
<tr>
<td>0008 01000000</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>.LL9:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000c 24800003</td>
<td></td>
<td>ble,a .LL2</td>
</tr>
<tr>
<td>0010 92224008</td>
<td></td>
<td>sub %o1, %o0, %o1</td>
</tr>
<tr>
<td>0014 90220009</td>
<td></td>
<td>sub %o0, %o1, %o0</td>
</tr>
<tr>
<td>.LL2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018 80A20009</td>
<td></td>
<td>cmp %o0, %o1</td>
</tr>
<tr>
<td>001c 12BFFFFFC</td>
<td></td>
<td>bne .LL9</td>
</tr>
<tr>
<td>0020 01000000</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>.LL8:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0024 81C3E008</td>
<td></td>
<td>retl</td>
</tr>
<tr>
<td>0028 01000000</td>
<td></td>
<td>nop</td>
</tr>
</tbody>
</table>
Encoding Example

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

Encoding of “SUB” on the SPARC:

```
<table>
<thead>
<tr>
<th>10</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.

.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
Role of an Assembler

Constant data needs to be aligned.

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

**Assembler directive**

```asm
.sect .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
6F00

Bytes added to ensure alignment

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

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
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
```

```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;
}
```
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
addl %ebx,%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
  - bar.c
  - cc
  - foo.s
  - bar.s
  - as

- Assembler
  - foo.o
  - bar.o

- Archiver
  - printf.o
  - fopen.o
  - malloc.o
  - printf.o
  - fopen.o
  - ar
  - libc.a

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

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

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

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

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

```
Linking file1.o
a="Hello"
main()
baz()

main()
baz()
bar()

.a.out
.text segment
main()
baz()
bar()

.data segment
a="Hello"

.bss segment
char b[6]

file2.o
char b[6]
bar()

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)
Object Files

file1.c:

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

int main() {
    bar();
}

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

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:
Idx Name  Size  VMA   LMA  Offset  Algn
  0 .text  038  0  0  034  2**2
  1 .data  008  0  0  070  2**3
  2 .bss  000  0  0  078  2**0
  3 .rodata  008  0  0  078  2**3

SYMBOL TABLE:
0000 g O .data 006 a
0000 g F .text 014 main
0000  *UND* 000 bar
0014 g F .text 024 baz
0000  *UND* 000 printf

RELOCATION RECORDS FOR [.text]:
OFFSET TYPE VALUE
0004 R_SPARC_WDISP30 bar
001c R_SPARC_HI22 .rodata
0020 R_SPARC_LO10 .rodata
0028 R_SPARC_WDISP30 printf
```
Object Files

file1.c:

```c
#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
  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

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

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

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

Code starting address changed

```
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

105f8 <main>:
  0: 9d e3 bf 90 save %sp, -112, %sp
  4: 40 00 00 0d call 10630 <bar>
    4: R_SPARC_WDISP30 bar
  8: 01 00 00 00 nop
   c: 81 c7 e0 08 ret
 10: 81 e8 00 00 restore

1060c <baz>:
  1060c: 9d e3 bf 90 save %sp, -112, %sp
  10610: f0 27 a0 44 st %i0, [ %fp + 0x44 ]
  10614: 11 00 00 41 sethi %hi(0x10400), %o0
  10618: 90 12 23 00 or %o0, 0x300, %o0
  10620: 40 00 40 62 call 207a8
  10624: 01 00 00 00 nop
  10628: 81 c7 e0 08 ret
  1062c: 81 e8 00 00 restore
```
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
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.

<table>
<thead>
<tr>
<th>Address 0:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>libXaw.a</td>
<td>libXaw.a</td>
</tr>
<tr>
<td></td>
<td>libX11.a</td>
<td>libX11.a</td>
</tr>
<tr>
<td>xeyes</td>
<td>xterm</td>
<td>xclock</td>
</tr>
</tbody>
</table>
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.

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>xterm</td>
<td>netscape</td>
<td></td>
</tr>
</tbody>
</table>

Address 0: xeyes
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>libXm.so</th>
<th>libXaw.so</th>
<th>libXaw.so</th>
</tr>
</thead>
<tbody>
<tr>
<td>libX11.so</td>
<td>libX11.so</td>
<td>libX11.so</td>
</tr>
<tr>
<td>xterm</td>
<td>netscape</td>
<td></td>
</tr>
<tr>
<td>xeyes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Address 0:  

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 each at a different address.
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
```