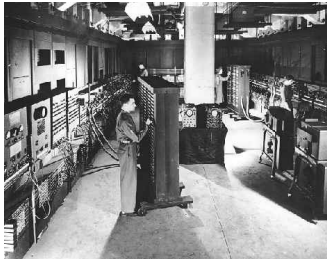


Generating Code and Running Programs

Stephen A. Edwards

Columbia University

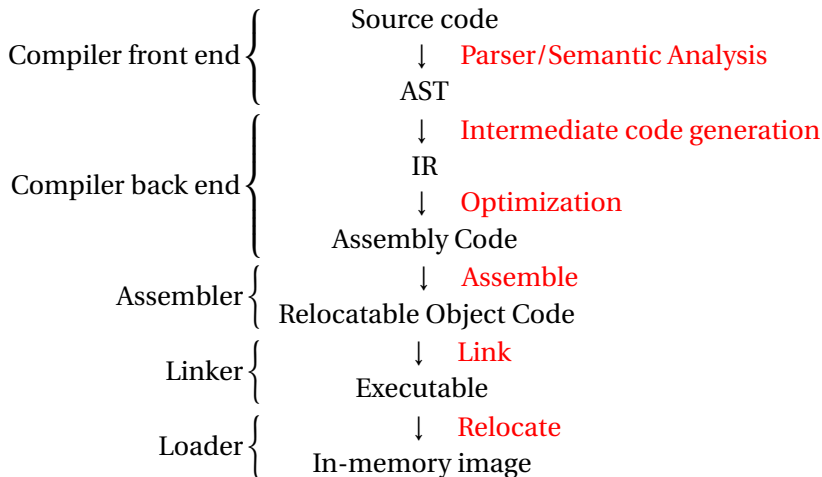
Summer 2014



Part I

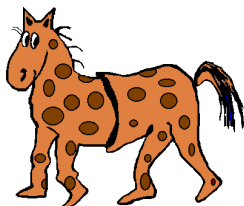
The Compilation Process

A Long K's Journey into Byte[†]



[†]Apologies to O'Neill

Compiler Frontends and Backends



The front end focuses on *analysis*:

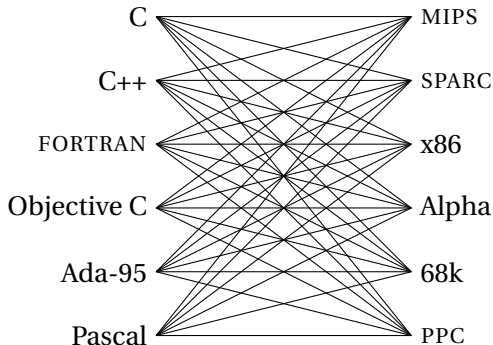
- ▶ Lexical analysis
- ▶ Parsing
- ▶ Static semantic checking
- ▶ AST generation

The back end focuses on *synthesis*:

- ▶ Translation of the AST into intermediate code
- ▶ Optimization
- ▶ Generation of assembly code

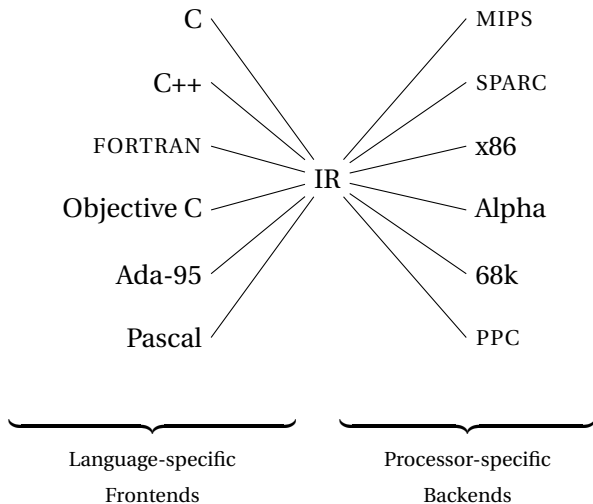
Portable Compilers

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.



Part II

Intermediate Representations/Formats

Stack-Based IR: Java Bytecode

```
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

Register-Based IR: Mach SUIF

```
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

Optimization In Action

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



GCC on SPARC

```
gcd: save %sp, -112, %sp  
      st %i0, [%fp+68]  
      st %i1, [%fp+72]  
.LL2: ld [%fp+68], %i1  
      ld [%fp+72], %i0  
      cmp %i1, %i0  
      bne .LL4  
      nop  
      b .LL3  
      nop  
.LL4: ld [%fp+68], %i1  
      ld [%fp+72], %i0  
      cmp %i1, %i0  
      bge .LL5  
      nop  
      ld [%fp+72], %i0  
      ld [%fp+68], %i1  
      sub %i0, %i1, %i0  
      st %i0, [%fp+72]  
      b .LL2  
      nop  
.LL5: ld [%fp+68], %i0  
      ld [%fp+72], %i1  
      sub %i0, %i1, %i0  
      st %i0, [%fp+68]  
      b .LL2  
      nop  
.LL3: ld [%fp+68], %i0  
      ret  
      restore
```

GCC -O7 on SPARC

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

Typical Optimizations

- ▶ Folding constant expressions

$1+3 \rightarrow 4$

- ▶ Removing dead code

`if (0) { ... } \rightarrow 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;  
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



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

lower
→

```
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  
C: jmp A  
E: ret a
```

split
→

```
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  
C: jmp A  
E: ret 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.

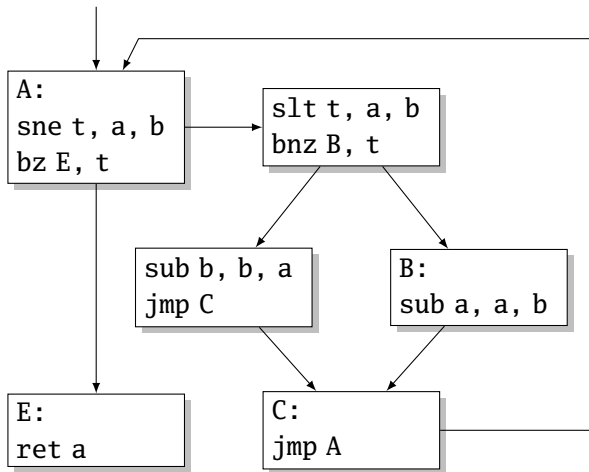
A:
sne t, a, b
bz E, t

slt t, a, b
bnz B, t

sub b, b, a
jmp C

B:
sub a, a, b

C:
jmp 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

Address	Instruction code
0000 80A20009	gcd: cmp %o0, %o1
0004 02800008	be .LL8
0008 01000000	nop
	.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 12BFFFFC	bne .LL9
0020 01000000	nop
	.LL8:
0024 81C3E008	retl
0028 01000000	nop

Encoding Example

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

Encoding of “SUB” on the SPARC:

10	rd	000100	rs1	0	reserved	rs2
31	29	24	18	13	12	4

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.

LL2 is 3 words away

```
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 12BFFFFC    bne   .LL9
```

Don't know offset of LL2

Know offset of LL9

Role of an Assembler

Constant data needs to be aligned.

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

Assembler directive

```
.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  
      6F00  
      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

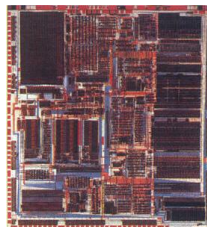


Optimization: Register Allocation

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

```
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

```
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)
```

```
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

```

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

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

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

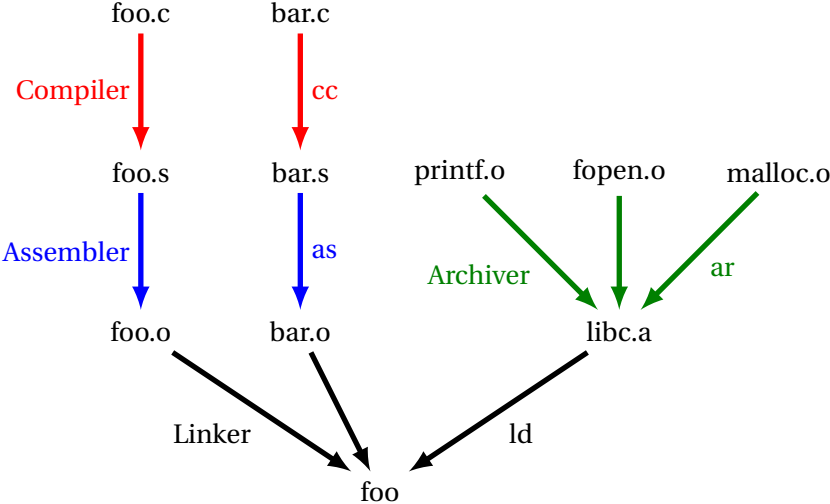
movl -20(%ebp),%eax
movl -24(%ebp),%edx
addl %edx,%eax                       addl %ebx,%eax
movl %eax,%edx
addl -16(%ebp),%edx                 addl %esi,%eax
movl %edx,%eax
```

```
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



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();
}

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



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();
}
```

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

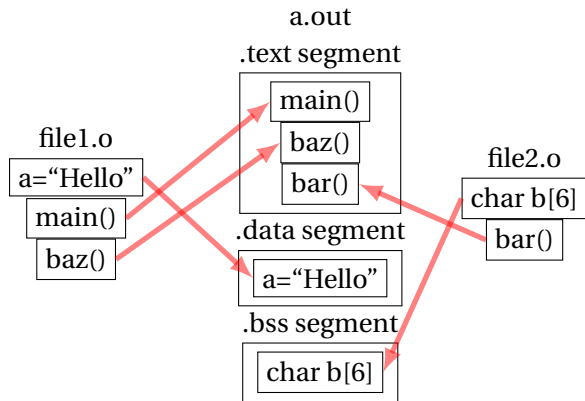
baz()

file2.o

char b[6]

bar()

Linking



.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)

Object Files

file1.c:

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

int main() {
    bar();
}

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

exported symbols

imported symbols

Object Files

file1.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	0	.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_L010	.rodata
0028	R_SPARC_WDISP30	printf

Object Files

file1.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
   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

```
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 Unresolved symbol
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>:
105f8: 9d e3 bf 90 save %sp, -112, %sp
105fc: 40 00 00 0d call 10630 <bar>
10600: 01 00 00 00 nop
10604: 81 c7 e0 08 ret
10608: 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
1061c: d2 07 a0 44 ld [ %fp + 0x44 ], %o1
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:

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

int main() {
    bar();
}

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

```
105f8 <main>:
105f8: 9d e3 bf 90 save %sp, -112, %sp
105fc: 40 00 00 0d call 10630 <bar>
10600: 01 00 00 00 nop
10604: 81 c7 e0 08 ret
10608: 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 ! "%s"
1061c: d2 07 a0 44 ld [ %fp + 0x44 ], %o1
10620: 40 00 40 62 call 207a8 ! printf
10624: 01 00 00 00 nop
10628: 81 c7 e0 08 ret
1062c: 81 e8 00 00 restore
```

file2.c:

```
#include <stdio.h>
extern char a[];

static char b[6];

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

```
10630 <bar>:
10630: 9d e3 bf 90 save %sp, -112, %sp
10634: 11 00 00 82 sethi %hi(0x20800), %o0
10638: 90 12 20 a8 or %o0, 0xa8, %o0 ! 208a8 <b>
1063c: 13 00 00 81 sethi %hi(0x20400), %o1
10640: 92 12 63 18 or %o1, 0x318, %o1 ! 20718 <a>
10644: 40 00 40 4d call 20778 ! strcpy
10648: 01 00 00 00 nop
1064c: 11 00 00 82 sethi %hi(0x20800), %o0
10650: 90 12 20 a8 or %o0, 0xa8, %o0 ! 208a8 <b>
10654: 7f ff ff ee call 1060c <baz>
10658: 01 00 00 00 nop
1065c: 81 c7 e0 08 ret
10660: 81 e8 00 00 restore
10664: 81 c3 e0 08 retl
10668: ae 03 c0 17 add %o7, %l7, %l7
```

Part VII

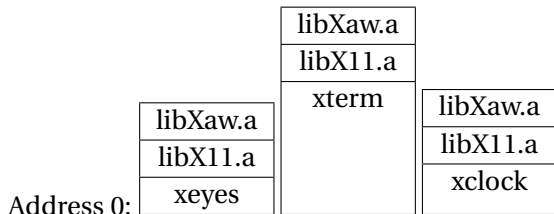
Shared Libraries and Dynamic Linking



Shared Libraries and Dynamic Linking

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

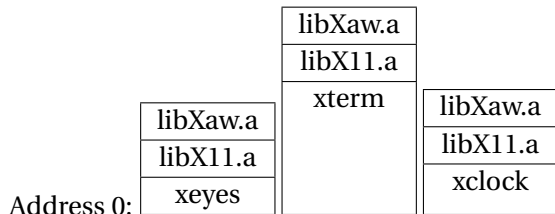
Under a *static linking* model, each executable using a library gets a copy of that library's code.



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.



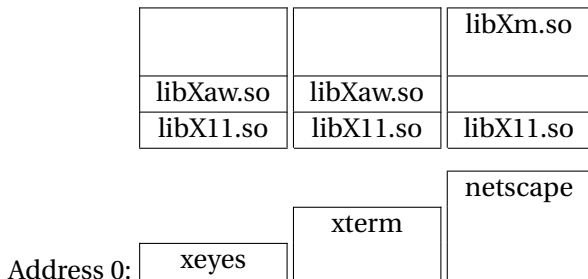
Wasteful: running many GUI programs at once fills memory with **nearly identical** copies of each library.

Something had to be done: another level of indirection.

Shared Libraries: First Attempt

Most code makes assumptions about its location.

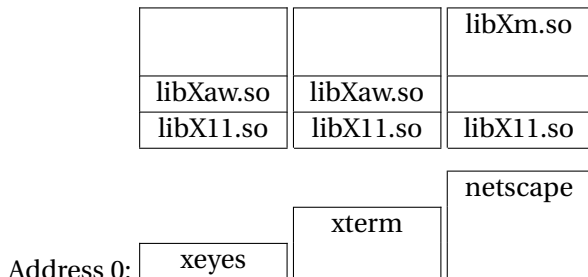
First solution (early Unix System V R3) required each shared library to be located at a unique address:



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:

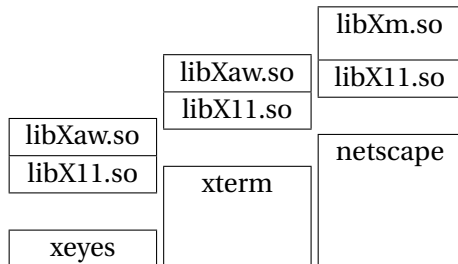


Obvious disadvantage: must ensure each new shared library located at a new address.

Works fine if there are only a few libraries; tended to discourage their use.

Shared Libraries

Problem fundamentally is that each program may need to see different libraries **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

```
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

```
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
```