

Compiler, Assembler, Linker

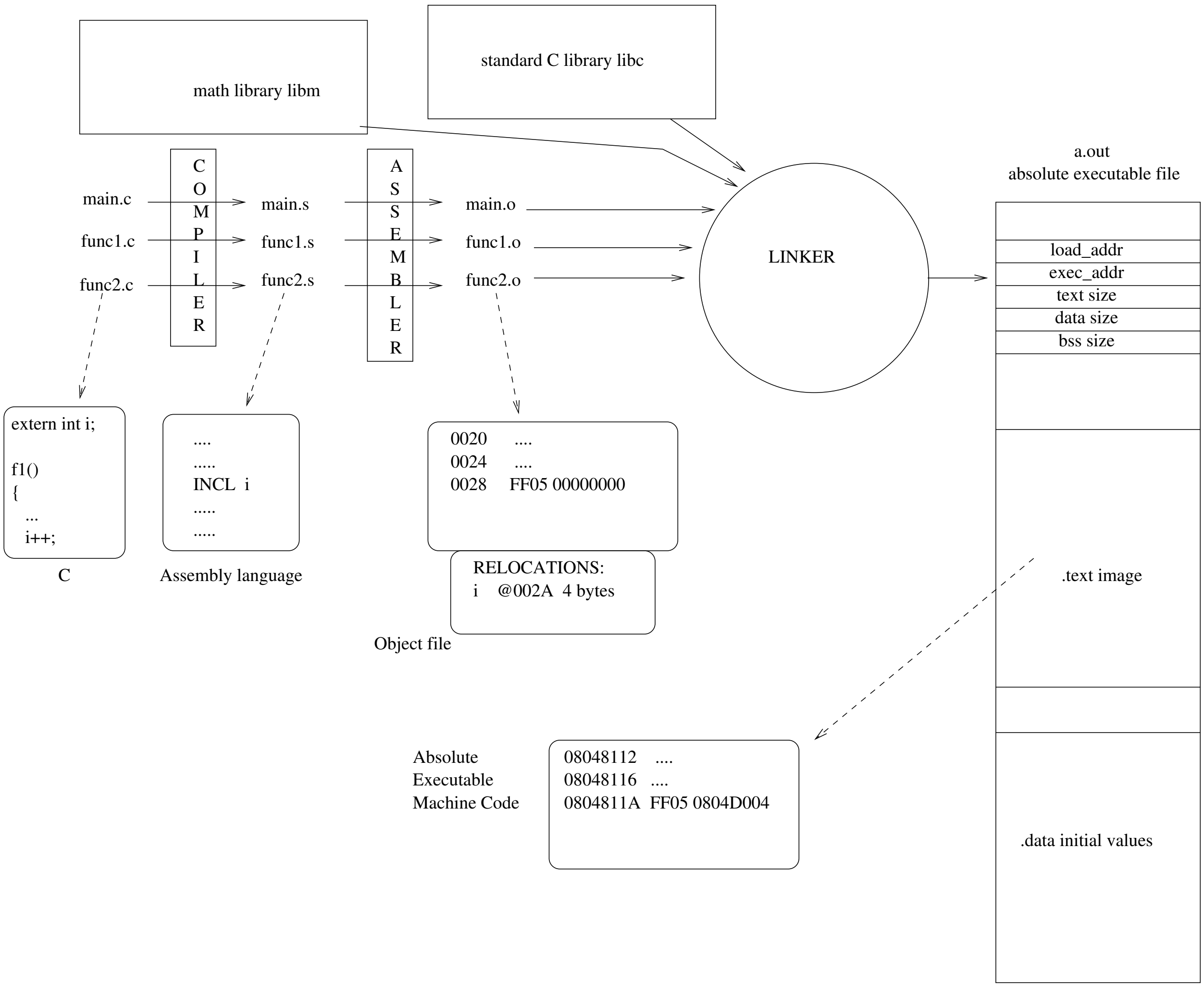
This course concerns itself with the compiler proper, which takes a high-level source language such as the example used in this course, C, and translates it into an assembly language. In the UNIX programming framework, assembly language files traditionally have a `.s` suffix, and are processed by the assembler, called `as`.

The result of assembly is a **relocatable object file**, which has a `.o` suffix. It is typical with larger projects to divide the code among multiple source files. When part of the project is changed, only the source file or files containing this change need to be re-compiled and re-assembled. The result is then linked with the object files previously produced, which have not been affected by the change.

This model allows intermixing of modules from different source languages (although issues of run-time calling convention creep in, e.g. the nested visibility of procedures and local variables in a language such as Pascal vs the more flat global/local model of C) or modules which are written directly in assembly language. The latter is commonly found in operating systems kernel code or in specialized graphics/multimedia applications which are looking to exploit machine-specific optimizations or specialities such as the SSE instructions under X86.

Furthermore, most programming environments expect a standard library of functions/procedures, such as the Standard C library (printf, etc.), and often a programmer will want to make use of a specialized library (e.g. in C the floating point functions are in the `m` library). The UNIX programming framework implements libraries as a single file which is an collection of `.o` files in a format called `ar` (this is an archive format similar in concept to `tar` or `zip`).

It is the linker program, called `ld` in UNIX (because an older synonym for the linker is the `loader`) which collects the various object files, including those referenced through libraries, and combines them into a monolithic, coherent, executable file.

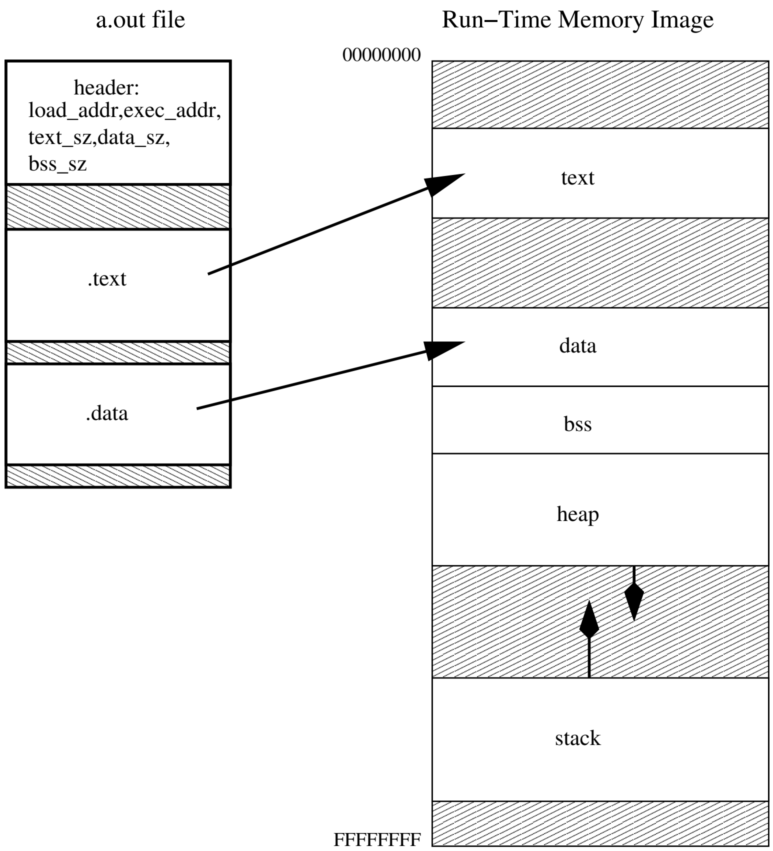


Therefore, compiling a C program into an executable file is a multi-stage process involving several tools. When one simply runs:

```
cc test.c
```

The `cc` command transparently executes the compiler, assembler and linker, resulting in an executable file called `a.out` (if there are no fatal errors in compilation). The "a" in "a.out" stands for "Absolute", meaning that all symbolic references have been resolved. It is possible to arrest the compiler wrapper at different stages, to see the assembly or object file that results, as we have seen in class. It is also of course possible to assign a different name to the `a.out` file.

An a.out file contains pure machine code that can be directly executed by the processor. The executable file ("a.out") contains everything that the operating system needs to create a program's initial memory configuration and begin execution. The header of the a.out identifies it to the operating system as an executable file and specifies the processor architecture on which the machine code instructions will run. A program compiled for an x86 architecture can not be executed, for example, on a SPARC processor. The header also gives the size of the text, data and bss memory requirements, as explained below.



The **bss** section contains all un-initialized global variables. The C language specification states that all such variables, lacking an explicit initializer, must be initialized by the operating system or C run-time environment to 0.

```
int j=2;
int k;
main()
{
 int l;
          /*...*/
}
```

In the example above, `j` has an explicit initializer, and will be in the data segment. The value of the initializer will be found in the .data section of the a.out. `k` is uninitialized. Therefore it will reside in the bss segment of memory and will have an initial value of 0. The ISO C standard says that "If an object that has static storage duration is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a NULL pointer constant". This is satisfied by filling every byte of the bss region with 0, and is performed by the operating system. The variable `l` has automatic storage scope, and therefore will be found on the stack (or, if the compiler is set for heavy optimization and a pointer to `l` is never taken, it may live in a register). Automatic variables are not 0-initialized.

Dynamic Memory is memory which is allocated from the operating system while the program is running, rather than at program load time. Dynamic memory is not part of the C language, but rather part of the run-time library, and is allocated using the `malloc` function, which in turn makes an operating system call. As such, dynamic memory is not the concern of the C compiler writer. In many other languages, dynamic memory allocation is specified as part of the core language.

Linker Symbols

The linker is independent of any particular high-level language. It keeps track of symbols, which are similar to identifiers in C, but their type is very low-level oriented. A symbol has a name (namespace limitations vary widely by system, but all UNIX systems allow at least 32 characters from a set at least as broad as C identifiers), a type (e.g. absolute or relative, as discussed below), a section (data, text, etc.), a size (e.g. 32 or 64 bits), and a value (the value may be undefined at certain points along the way, e.g. a reference to an external function in a `.o` file). There are only two linker symbol scopes: local (to that `.o` file) and global.

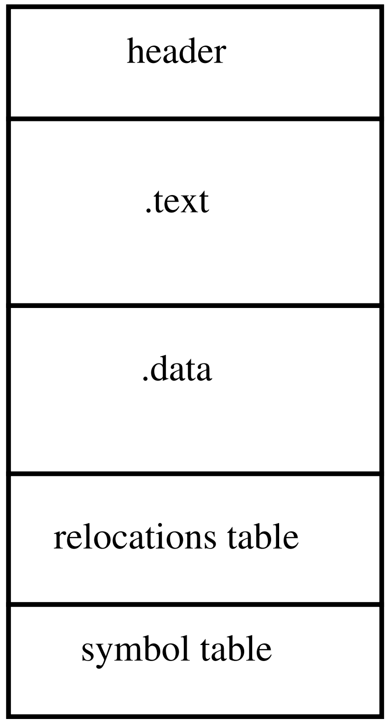
Here are some things in the C language that never appear in an object file (except for debugging purposes) and do not concern the linker:

• C language type specifiers (such as "p is a pointer to a pointer to a function returning int and taking two int arguments")
• C language structure, union and enum definitions and typedef names.
• `goto` labels
• Internal labels that may be generated by the compiler for loops, switch statements, if statements, etc. These labels will appear in the assembly language, but they are local symbols and will not be visible to the linker.
• automatic (local) variables. An important exception is a local variable that is declared with the `static` storage class. This variable has local scope, i.e. its name is not visible outside of the curly-braced block in which it is declared, but it lives in the same neighborhood as global variables.

Inside the object file

A relocatable oject file (`.o` file) contains a header and one or more sections:

Object File



A series of examples will now illustrate what goes into those sections. But first, a brief detour:

UNIX-style assembly syntax

The Intel documentation uses the Intel standard assembly language syntax, but the UNIX assembler `as` follows a different convention (which is consistent across different processors). In the UNIX syntax, an identifier is unambiguously an assembler symbol. To reference a register, its name is prefixed with a percent sign, e.g. %eax. To use a symbol as an immediate value, the dollar sign is used as a prefix. Otherwise the symbol means the contents of that address. Register indirect addressing modes are indicated by brackets or parentheses. UNIX assembly instructions are opcode src1,src2,dst for 3-address instructions or opcode src,dst for 2-address. (Note that Intel syntax is dst,src). We will use the UNIX syntax in these notes. Another name for this is the "AT&T" syntax, after the original authors of UNIX

In the UNIX/AT&T syntax, where a particular opcode can be performed at different precisions, that opcode receives a letter suffix: b,w,l,q for 8,16,32 and 64-bit operations respectively. In assembly language, each instruction is represented by a line having 4 fields which are delimited by whitespace. The first field is an optional label, which associates a symbolic name with the address of the instruction. The label may also appear on a line by itself. In either form, the label must have a colon following it. The next field is the opcode, followed by the operands field. After the operands, a comment can be placed, started with a # character. This comment is ignored by the assembler.

Some opcodes are not real machine language opcodes that the processor understands, but pseudo-opcodes, or directives, that are recognized by the assembler. Such pseudo-opcodes begin with a dot.

As the assembler digests its input, it maintains a group of location counters or "cursors". There is one cursor for each of the object file sections. The assembler also maintains a notion of which section it is putting output into. Initially, all of these cursors are initialized to 0 and the assembler is in the `.text` section. As we switch back and forth between sections with directives like `.data` and `.text`, the assembler remembers, via the location counter cursor, where it left off.

The presence of a label associates that symbol with the current section and location. E.g.:

```
label1:
        pushl   %ebp
```

The linker symbol `label1` will now have the value of the address of the `pushl` opcode, therefore

```
        jmp     label1
```

would have the desired effect of jumping to that point in the code.

Example: defining and referencing variable instances

Let's see what is actually in an object file and how it gets there:

```
/* f1.c */
extern int i;                   /* Referencing instance */

f()
{
        i=2;
}

/* f2.c */
int i;                          /* Defining instance */
f2()
{
        i=1;
        f();
}
```

In `f1.c`, the `extern` storage class for variable *i* tells the compiler that this variable is external to that `.c` file. Therefore, the compiler does not complain when it does not see a declaration that variable. Instead, it knows that `i` is a global variable, and should be accessed by using an absolute addressing mode. Neither the compiler nor the assembler knows what that actual address will be. That is not decided until the linker puts all the object files together and assigns addresses to symbols. Therefore, the assembler must leave a "place-holder" in the object file. Likewise, in f2.o, there will be a reference to the symbol `f`.

There is a difference between an extern storage class which is created with an explicit extern keyword, and one which is implicit for a global scope declaration lacking a specific storage class keyword. The former is used to inform the compiler of the data type of the identifier, and does not result in the emission of assembly language. The latter causes an assembly language directive to be output, as described later in this unit, and creates the defining instance of the symbol. It is good practice to use the `extern` keyword in header files which define the "public" global variables of a module, and to have exactly one place, in a .c file, not a .h file, where the defining instance (lacking the extern keyword) is placed. Because header files wind up getting included in each `.c` file that is compiled separately, having more than one defining instance could result in a redefinition error at link time.

Now let us examine the assembly language files produced by the C code above (using gcc under Linux on an x86 system).

```
**** f1.s
         .file     "f1.c"              #For proper error reporting
         .text                         #Place output into .text section
         .globl f            #f is a globally visible symbol
        .type    f,@function           # and represents a function addr
f:
        pushl %ebp           #These instructions set up
        movl %esp,%ebp                 #the stack frame pointer
        movl $2,i
        ret
        .size    f,.-f                 #Calculate the size of function f

**** f2.s
         .file     "f2.c"
        .comm   i,4,4                  #i is common block sym, 4 bytes long, align 4
         .text
         .globl f2
        .type    f2,@function
f2:
        pushl %ebp           #These instructions set up
        movl %esp,%ebp                 #the stack frame pointer
         subl $8,%ebp                  #useless operation, no local vars
         movl $1,i
        call f
        leave                          #Restore frame pointer
        ret
        .size    f2,.-f2
```

Looking at f1, the assembler directive `.text` tells the assembler that it is assembling opcodes to go in the .text section of the object file. The `.globl` directive will cause the assembler to export the associated symbol as a defining instance. `.type` is used to pass along information into the object file as to the type of the symbol. Please note that it has nothing to do with the C language notion of type. Symbol types may either be functions or variables. The linker is able to catch gross errors such as if `f` were defined as a variable in `f1.c` instead of a function. The `.size` directive calculates the size of the function by subtracting the value of the symbol representing the first instruction of the function (e.g. `main`) from the special assembler symbol . (period character), which represents the current byte output position. Note that the CALL to function `f` is done symbolically, as is the assignment into global variable `i`. The symbol `i` is *referenced* in f1 but it is defined in f2. The linker will stitch all of this together.

In f2, note the `.comm` assembler pseudo-opcode. It creates a defining instance of the symbol `i`, specifying its size and alignment restriction in bytes. Another name for the `bss` section is the `common block`. Because there is no initializer, it is not an error for multiple defining instances of the variable `i` to appear during linking, as long as they are all uninitialized and have the same size and alignment. It is also acceptable for there to be at most one initialized symbol (DATA) with the same name as one or more COMMON symbols. The term COMMON BLOCK is very old, dating to the FORTRAN days.

After gcc runs f1.s and f2.s each through the assembler (as), the resulting files f1.o and f2.o are **object files**. Object files are similar to an a.out file, however all addresses are relative. In addition, the object file will have a section known as the **symbol table** containing an entry for every symbol that is either defined or referenced in the file, and another section called the **relocations table**, described below.

The `objdump` command can be used to view parts of an object or a.out file, with command-line flags controlling what is dumped. For example, `objdump -h` dumps the header section, and `-t` dumps the symbol table. Let us view a disassembly of the .text section of `f1.o` (the listing below was re-formatted from the output of objdump -d):

```
Offset Opcodes                         Disassembly
0000   55                         pushl %ebp
0001   89E5                       movl %esp,%ebp
0003   C705 00000000 02000000     movl $2,0
000D   C3                         ret
```

If we were to look at the .text section in a hex dump, we'd see the string of bytes 55 89 E5 C7 05....etc. The `-d` option "disassembles" those bytes back to assembly language mnemonics. Note that the offset of the first instruction is 0. Obviously, this can not be a valid memory address. All offsets in object files are relative to the object file. It isn't until the linker kicks in that symbols gain absolute, usable addresses.

Next note that in the instruction beginning at offset 0003, the constant 2 is moved to memory address 0. We know from examining the corresponding assembly language input file that this is the instruction that moves 2 into variable `i`. The assembler has left a place-holder of 00000000 in the object file where the linker will have to fill in the actual 32-bit address of symbol `i` once that is known. C705 is the x86 machine language opcode for the MOVL instruction where the source addressing mode is Immediate and the destination mode is Absolute. The next 4 bytes are the destination address and the final 4 bytes of the instruction are the source operand (which is in Intel-style, or "little-endian" byte order).

Here is the dump of f2.o:

```
Offset Opcodes                              Disassembly
0000   55                           pushl %ebp
0001   89E5                         movl %esp,%ebp
0003   83EC08                               subl  $8,%esp
0006   C705 00000000 01000000       movl $1,0
0010   E8FCFFFFFF                   call 0x0E
0015   C9                           leave
0016   C3                           ret
```

Note that the placeholder in the CALL instruction is FFFFFFFC, but the disassembler decodes that as 000E. This is because the CALL instruction uses a Program Counter Relative addressing mode. The address to which execution jumps is the operand in the instruction added to the value of the Program Counter register *corresponding to the byte beyond the last byte of the CALL instruction*. In two's complement, FFFFFFFC is -4, therefore the CALL appears to be to the instruction at offset 000E. Of course, all of this is meaningless since it is just a placeholder that will be overwritten by the linker. Nonetheless, it reminds us that there are different **Relocation Types** depending on the addressing mode being used.

The symbol and relocation tables for the two object files will look something like this (`objdump -t -r`):

```
f1.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000005 R_386_32          i

SYMBOL TABLE:
00000000 g     F .text  0000000f f
00000000         *UND*  00000000 i



f2.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          i
```

```
00000011 R_386_PC32        f

SYMBOL TABLE:
00000004       O *COM*  00000004 i
00000000 g     F .text  00000017 f2
00000000         *UND*  00000000 f
```

The symbol table output is interpreted as follows: The first column is the value of the symbol (its relative or absolute address). [Note: for COMmon block symbols, the first column is the size] The second column contains flag characters (l=local, g=global, F=symbol is a function name, O=symbol is an object (variable) name, see man page for other flags). The third column is the section that the symbol belongs to (text,data,COMmon (bss) or UNDefined). The fourth column is either the size of the symbol if it is defined, or the alignment for undefined symbols.

Let us run this simple example through the linker. Of course, it is non-sensical, since without a main function this code will not actually run. (output below has been edited somewhat to remove irrelevant info:)

```
$ ld f1.o f2.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000008048094
$ objdump -t -d a.out

a.out:     file format elf32-i386

SYMBOL TABLE:
08048094 l    d  .text  00000000 .text
08049114 l    d  .bss   00000000 .bss
08048094 g     F .text  0000000f f
08049114 g     O .bss   00000004 i
080480a3 g     F .text  00000017 f2
00000000         *UND*  00000000 _start
08049114 g       .bss   00000000 __bss_start
08049114 g       .bss   00000000 _edata
08049118 g       .bss   00000000 _end

Disassembly of section .text:

08048094 <f>:
 8048094:       55                      push   %ebp
 8048095:       89 e5                   mov    %esp,%ebp
 8048097:       c7 05 14 91 04 08 02    movl   $0x2,0x8049114
 804809e:       00 00 00
 80480a1:       5d                      pop    %ebp
 80480a2:       c3                      ret

080480a3 <f2>:
 80480a3:       55                      push   %ebp
 80480a4:       89 e5                   mov    %esp,%ebp
 80480a6:       83 ec 08                sub    $0x8,%esp
 80480a9:       c7 05 14 91 04 08 01    movl   $0x1,0x8049114
```

```
 80480b0:       00 00 00
 80480b3:       e8 dc ff ff ff          call   8048094 <f>
 80480b8:       c9                      leave
 80480b9:       c3                      ret
```

We see that all symbolic references (relocation "holes") have now been filled in with absolute addresses. The text has been amalgamated into one contiguous section. Observe at address 80480B3 the CALL to function f has a machine-language operand of FFFFFFDC, because CALL uses a Program Counter (Instruction Pointer) Relative addressing mode. The program counter will be at address 80480B8 when the CALL is executed. FFFFDC+080480B8 = 08048094 (using two's complement arithmetic) and therefore gets us to the first opcode of function f. The "hole" is filled in with the value of the symbol, minus the location (program counter value) of the hole, and plus the original value of the hole. The bss symbol `i` has been given an absolute address of 08049114. Note that this disassembly is able to resolve symbols, because the symbol table is still in the a.out. Sometimes, e.g. for code obfuscation purposes, we don't want to include the symbol table. The `-s` option to gcc (which gets passed to ld) causes the symbol table to be stripped from the a.out

Address Constants

It is possible to create a compile-time constant which is the address of a linker symbol +/- an integer. This is explicitly permitted in the C standard, e.g.

```
extern int a[];

f()
{
int *p= &a[2];
}


Assembly Language:
f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    $a+8, -4(%ebp)
        leave
        ret

Disassembly:
00000000 <f>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 04                sub    $0x4,%esp
   6:   c7 45 fc 08 00 00 00    movl   $0x8,0xfffffffc(%ebp)
   d:   c9                      leave
   e:   c3                      ret
```

Why does this work? At compile time, the identifier `a` is equivalent to the address of the beginning of the array. Therefore, the address of `a[2]` is 8 bytes more. Because the absolute address of a is not known by the compiler, it must express this symbolically to the assembler as `$a+8`. The object file created by the assembler has the usual "hole" for the unknown value, but instead of being filled with a 0, it has the value of 8. When the linker assigns the address of the symbol `a` and fills in the relocations, it will add that number 8 to the symbol to create the proper address.

Initialized data

When global variables (including local static variables) have an initializer, that initializer value must wind up in the DATA section of the executable so it can be loaded with the program. Unlike automatic variables, these initializers only take effect prior to the program start and their value must be computable at compile time, at least in terms of a linker symbol +/- an optional offset.

```
int i=3;

Assembly:

        .data                 #Place bytes into the .data section now
        .align 4    #Align the next output byte on a 4-byte boundary
        .type   i,@object     #Declare i to be a variable
        .size   i,4           #Set size of i
i:
        .long   3             #Output 4 bytes
```

The `.globl` directive instructs the linker to make the symbol i globally visible during the linkage process. The `.data` directive switches the assembly target into the DATA section of the output (object) file. `.type` declares i to be a symbol associated with a variable, not a function. `align` causes the next output byte to be aligned on a 4-byte boundary, possibly skipping bytes (inserting padding) to do so. Finally, the label `i:` associates the next directive with the symbol i. That directive `.long` emits 4 bytes into the current section (DATA). There are similar directives for defining initializers of other simple data types. Complicated initializers, e.g. for structs or arrays, are built up with a succession of these simple directives.

Static variables with block/function scope

Local static variables require special treatment:

```
f()
{
static int a;
          a=1;
}

g()
```

```
{
static int a;
          a=2;
}

Assembly:
        .local  a.0
        .comm   a.0,4,4
          .text
f:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $1, a.0
        popl    %ebp
        ret
        .local  a.1
        .comm   a.1,4,4
          .text

g:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $2, a.1
        popl    %ebp
        ret
```

Because the linker does not know about C namespace and scope rules, the compiler must append unique tags to the identifier names to avoid conflict. The `.comm` directive by default creates a global symbol. The `.local` directive overrides that and prevents the symbol from being visible to other object files during the linkage process (although relocation references within the same file are satisfied by the symbol).

Strings

```
char ary[]="ABCDEF";

char *p="WXYZ";

main()
{
          *p=10;
}
```

The declaration of `ary` creates a 7-byte array of characters and initializes it to contain the characters in the string "ABCDEF". The declaration of `p` creates an area of memory that is initialized with the address of another area of memory which holds the string WXYZ. String constants are placed in a section of the object file known as `.rodata`. This section is typically loaded into memory as part of the text region, but that is operating-system dependent. On UNIX systems, since the text region is not writable, string constants are immutable, and `p[1]=0` will cause a run-time exception.

```
        .data                                   #enter .data section
        .type   ary,@object
        .size   ary,7
ary:
        .string "ABCDEC"            #emit sequence of bytes (includes NUL)
        .section .rodata            #enter .rodata section
 .LC0:                                          #private label for string constant
        .string "WXYZ"
        .data                                   #back to .data section
        .align 4
        .type   p,@object
        .size   p,4                             #reserve 4 bytes starting at p
p:
        .long   .LC0                            #initialize with addr of string const
        .text
        .type   main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    p, %eax
        movb    $10, (%eax)
        leave
        ret
```

What ld does

The first task of `ld` is to take inventory of all of the `.o` files being presented to it. It loads in the symbol tables from all of the object files to create a unified symbol table for the entire program. There is only one global namespace for all linker symbols. This might be considered a deficiency. Let's say we have a module `foo.c` that defines a function called `calculate`. If we attempt to incorporate that module into a program written by someone else, they may have also made a function called `calculate`. It is, after all, a common name.

If there is more than one defining instance of a symbol, i.e. if a symbol is *multiply-defined*, this is generally a fatal error. Consider what would happen if a programmer accidentally included two versions of function `f` above in two different `.c` files. When `f` is called somewhere in the program, which version should be called? (However, recall the common block exception above for uninitialized variables)

To ameliorate this problem of flat global linker namespace, a convention exists that one should prepend to one's global variable and function names a reasonably unique prefix. Therefore, we might call our function `foo_calculate`. This is less likely to conflict with another name. It isn't a perfect solution, but it works fairly well in reality.

In languages such as C++ with more complex namespace models, the compiler engages in what is called **symbol name mangling** to create unique linker symbols.

Global variable and function names that are intended to remain private to the `.c` file in which they are declared should be protected with the `static` storage class (see below). `static` symbols still require the assistance of the linker to be relocated. However, the use of `static` causes the compiler and the assembler to flag that symbol as a LOCAL symbol. The linker will then enter the symbol into a private namespace just for the corresponding object file, and the symbol will never conflict with symbols from other object files.

There are rare cases where it is useful to deliberately redefine a symbol. For instance, we may need to change how a piece of code, available only in library or object file form, calls another function. This all falls under the heading of "wild and crazy ld tricks" and will not be discussed further. Just remember that any duplicate symbol definitions are generally wrong.

Frequently symbols have a defining instance, but they are never actually referenced. For example, the programmer may write a function, but never call it, or declare a global variable, yet never use it in an expression. The linker doesn't care about this.

However, it cares deeply about the opposite case: a symbol that is referenced (by appearing in a relocations table) but which has no defining instance. Such a situation makes it impossible for the linker to complete its task of creating an absolute executable file with no dangling references. Therefore, it will stop with a fatal error, reporting the undefined symbol and the object file or files in which it is referenced.

Once all of the symbol definitions and references are resolved, `ld` will assemble the `a.out` file by concatenating all of the text sections of all of the object files, forming the single `.text` section of the `a.out`. In doing so, the text sections are relocated. An instruction which was at offset 10 in a particular object file's text section may now be at offset 1034 in the `a.out`.

Furthermore, `ld` has a concept of the `load address` of the program, i.e. the memory address at which the first byte of the `.text` section of the `a.out` will be loaded. This knowledge is part of configuring `ld` for a particular operating system and processor type, and is not normally something that the programmer needs to worry about. The load address is also placed in the `a.out` file so the operating system is sure to load the program at the address for which it was linked. Knowing the size of each object file's `.text` section, `ld` can calculate the absolute address that it will occupy in the final image, and can thus complete the symbol table, assigning absolute values to each text symbol.

A similar process is undertaken for the `data` and `bss` sections. `ld` assigns an absolute address to each `data` or `bss` symbol, based on the configured starting address of the `data` and `bss` memory regions. In the case of `data` symbols, the initializers contained in the individual object files are concatenated, forming the `.data` section of the `a.out`, which will thus contain the byte-for-byte image of what that section will look like when the program starts. For `bss` symbols, there are obviously no initializers. `ld` keeps track of the total number of `bss` bytes needed, and places that information into the `a.out` header so the operating system can allocate the memory when the program is loaded. During this process, each `data` or `bss` symbol is assigned an absolute address in the symbol table.

Now `ld` concatenates the various object files, processing the relocation records, replacing each "placeholder" with the actual, absolute address that the associated symbol is now known to have. In the case of Program-Counter-Relative relocation types, the proper offset is calculated with respect to the absolute address of the placeholder in question.

Static Libraries

When you write a C program, you expect certain functions, such as `printf`, to be available. These are supplied in the form of a **library**. A library is basically a collection of `.o` files, organized together under a common wrapper format called a `.a` file. It is similar to a "tar" or "zip" archive (although no compression is provided).

Convention is that a library *ZZZ* is contained in the library file: `libZZZ.a`. Using the `-l` option to `cc` tells the `cc` program to ask the linker to link with the specified library. For example: `cc myprog.c -lm` asks for the system library file `libm.a` (the math library) to be additionally linked. By default, `cc` always links the standard C library `libc.a`. These libraries are located in a system directory, typically `/usr/lib`.

Although a library embodies a collection of object files, the behavior when linking to a library is slightly different than if one just linked in all of the object files individually. In the latter case, the `.text` and `.data` sections of each object file would wind up in the `a.out`, regardless of whether or not anything in a particular object file was actually used. With a library, `ld` builds a symbol table of all the object files in the library, and then only selects those object files that are actually needed for inclusion into the executable.

However, it is important to realize that static libraries are not generally used anymore in modern, general-purpose operating systems. The issue is with software maintenance. Let us say that an important library, such as `libc` the standard C library, has a security vulnerability. With static libraries, it would be necessary to re-compile or re-link every executable on the system against a new version of the library. This means either shipping the `.o` version of each executable, and/or the source code, and hoping that the required compiler/linker toolchain is in place. Therefore, dynamic libraries are normally used for system libraries.

Dynamic Linking

When dynamic (shared) libraries are used, there are two parts to the linkage process. At compile time, `ld` links against the dynamic library (which has a `.so` extension in UNIX) for the purpose of learning which symbols are defined by it. However, none of the code or data initializers from the dynamic library are actually included in the a.out file. Instead, `ld` records which dynamic libraries were linked against. This information is placed into one of the auxiliary sections of the `a.out` file.

At execution time, the second phase takes place before the `main` gets invoked. A small helper program called `ld.so` is loaded into the process address space by the kernel and executed. I.e. the start address of the program is not `main` and not even the `__start` standard C library startup function (covered in ECE357). Instead, it is the start address of the dynamic library loader.

On Linux systems, the kernel maps this `ld.so` loader code into a convenient place in the process address space, and sets up the stack so that the list of required shared libraries, and other required information, is present. The dynamic loader finds each of the required libraries, by looking at a sequence of directories similar to the `$PATH` shell variable for finding executables. In this case, the environment variable `LD_LIBRARY_PATH` contains this list of directories. There is also a pre-defined list which is hard-coded into `ld.so`, and additional search places which can be hard-coded into the `a.out` at link time. Each library must be located, if not, a fatal error occurs and `ld.so` causes the process to exit. For each library, the dynamic loader reads its header and then uses `mmap` to create text, data and bss regions for the library.

Since the actual libraries used at run-time to satisfy the requirements are not known at link-time, this means we don't know when linking what the actual, absolute memory addresses will be of any code or data that are in the shared libraries. Thus some additional magic is required to link the static and the dynamic portions of the executable, and to compile and link the shared library itself.

From the standpoint of the static portion of the file, we are concerned with accessing (1) functions that are defined in the shared library and (2) global variables that are exported by the shared library. With regard to both points, at the time of compilation (from C to assembly) of the static portions, we don't know if the external references to code and data will, when ultimately satisfied by the linker, match up to other static portions (i.e. static `.a` libraries or other loose `.o` files) or to dynamic libraries (`.so`). Therefore, the compiler can't do anything special, and has to access these global functions and variables in the same way.

For global variables (exported by the library), this means that they are assigned a static address at link time, and are accessed by absolute memory address. We should note here that exporting global variables from a library to the users of the library is a deprecated practice, because it is not inherently thread-safe. And, in fact, to make multi-threaded programming work correctly with the classic UNIX API, things such as `errno` are actually macros which return thread-specific versions of the variables.

For functions exported by the library, the user of the library (static code) is going to emit `call` instructions (X86 opcode, but generally every architecture has a call opcode) which reference an absolute memory address. But, the exact absolute memory address of the referenced function is not known at link time. Therefore, for text symbols that match up to a shared library at link time, the linker inserts some special magic. It creates what is known as the **Procedure Linkage Table** (PLT). This is a series of `jmp` instructions, but the target of the jump is not filled in until run-time. So, for example, when the static user of the dynamic C library calls `printf`, that symbolic reference is satisfied by a PLT stub such as

```
printf:   jmp       printf@LIBC
```

and the relocations table associated with the a.out contains an entry to cause the address ultimately assigned to **printf** at run-time to be poked into that slot in the PLT. We therefore see that calls to shared library functions incur a slight additional overhead of the double jump.

We now come to the issue of the dynamic portion of the code. This comprises `.o` files that have been compiled with the special `-fpic` (position independent code) flag, and then linked with `gcc -shared`. This instructs the compiler that the intended run-time environment is that of dynamically-loaded code. Therefore, there are no absolute memory references whatsoever. Any references that are satisfied within a given `.o` file, such as static global variables, are handled by taking an offset from the program counter. Since the sizes of the text, data and bss region of that given shared library are known at the time that the shared library is built, the linker can insert the appropriate program-counter-relative offsets at this time.

In order to access global variables which are also visible to the static portion of the code, another magic table is required, known as the **Global Offset Table** (GOT). We don't want to force `ld.so` to have to back-patch every possible location of a reference to a global variable at run-time. Instead, the GOT is an array of pointers. At link time, since the absolute memory addresses of the globals are known, the linker populates this table. Then at run time, dynamic code accesses the global variables by first finding the GOT, then loading the address of the correct variable from the appropriate slot in the GOT, and then finally dereferencing the pointer. Thus there is a performance penalty.

Alignment and Padding

Different architectures impose different alignment restrictions. For example, on a SPARC processor, accesses to ints/longs must be at memory addresses which are a multiple of 4. Violation results in a fatal run-time exception. While the X86 architecture is more forgiving, better performance is obtained if 4-byte accesses are aligned on 4-byte boundaries.

When laying out a structure and assigning offsets to its members, the compiler must use a recursive algorithm which is based on a lookup table giving the size and alignment constraints of the scalar data types. The compiler uses two golden rules: 1) The offset of any member must be a multiple of the alignment restriction of that member. E.g. if the member is an int, the offset must be a multiple of 4. 2) The total size of the structure is such that were the structure in an array of such structures, each element of the array would begin at an offset which satisfies the most restrictive member of the structure.

The compiler maintains an offset counter, which is initialized to 0 for each structure definition. 0 is clearly a multiple of any alignment boundary, and therefore rule #1 is satisfied at the start. After inserting a member, the offset counter is advanced by the sizeof that member. Then before the next member is inserted, the offset counter is rounded up to the next alignment boundary based on the constraints of that next member. E.g. in the following :

```
struct example {
        int a;
        char b;
        int c;
        char d;
};
```

Assuming that ints are 4 bytes long and are aligned on 4 byte boundaries, and that chars are 1 byte long and have no alignment constraint: Member a is clearly given offset 0. Member b is given offset 4. The offset counter is now 5, but member c requires an offset which is a multiple of 4. Therefore the counter is rounded up and c is given offset 8. The space between b and c is wasted and is called **padding**.

(*Aside: on gcc compilers, it is possible to use the directive __packed__ to force the compiler to lay out the structure without padding. This is often used to make a structure match a physical device mapped into memory, or the format of a network protocol.*)

When the last member is laid out, the compiler determines the worst-case alignment restriction among all the members. The offset counter is then rounded up (if necessary) to satisfy that worst-case constraint. The `sizeof` the structure type is this rounded-up value. This takes care of Rule #2 above, and by satisfying that rule, Rule #1 will automatically be satisfied for all members of all elements of an array of structures.

So, element d is given the offset 12, but in order to satisfy rule #2, padding must be inserted at the end, after element d. Since the worst-case alignment restriction is that of the int (4 byte boundary) 3 bytes of padding are added, making the total size of the structure 16 bytes.

The algorithm above can be applied recursively for structures which contain aggregate data types (structures, unions and arrays) as members.

One can write a simple test program to determine the sizeof and alignment of basic data types:

```
#define S(t) struct {char a;t test;} s_##t

typedef int * ptr;
typedef long long longlong ;
typedef long double longdouble;
S(char);
```

```
S(short);
S(int);
S(long);
S(longlong);
S(float);
S(double);
S(longdouble);
S(ptr);

main()
{
#define AS(t) printf("%s size %d alignment %d",#t,\
          (int)sizeof(s_##t.test),\
          (int)((char *)&s_##t.test - (char *)&s_##t))
AS(char);
AS(short);
AS(int);
AS(long);
AS(longlong);
AS(float);
AS(double);
AS(longdouble);
AS(ptr);
}
```

Broad vs narrow compilers

There are two different approaches to compiler design. One attempts to craft an optimal compiler for a specific architecture or narrow range of architectures. The other attempts to make a general-purpose compiler which works consistently across a broad range of architectures. The gcc project is an example of the latter, employing the front-end, IR, optimizer, code generation architecture described throughout this course.

The advantage of the former approach is it tends to produce a more optimal output, because it can advance certain decisions further forward in the compilation process, not having to worry about re-targetability.

The advantage of the latter approach is of course re-use of compiler code. This is not just a matter of laziness. The effort which goes in to verifying the correctness of the various phases of the compiler is then reaped for many different targets and several different input languages.

In designing such a broad compiler, finding an appropriate IR is challenging because of the wide variety of target architectures. It is not just a matter of renaming opcodes. There are substantial philosophical differences among processors.

CISC vs RISC

One of the basic distinctions is Complex Instruction Set vs Reduced Instruction Set design. The CISC approach is older, having evolved from the days of hand-crafted assembly. It tends to offer many, complicated instructions and addressing modes, often with quirky restrictions or optimizations which make sense to a human programmer but are difficult to express to an optimizing compiler. In contrast, RISC was an approach that developed out of research into how high-level languages were being treated by compilers. It was designed with fewer, simpler instructions and addressing modes. Whereas CISC instructions tend to be of variable size and execution time, RISC instructions tend to be fixed size and constant time. This makes it easier for a compiler to generate good assembly code, but makes it more awkward for a human programmer. We will look at one example of each approach.

The Intel X86 architecture

X86 refers broadly to a family of Intel (and compatible) microprocessors manufactured in the last 25 years or so. It is also called the X86 architecture by Intel. The first 32-bit X86 processor was the 80386. X86-64 is a 64-bit extension to X86. Intel's is a CISC architecture which is a direct linear descendent of the very first microprocessor, the 4004 (a 4-bit product).

There are many who find the X86 architecture to be a dinosaur, and a badly designed one at that, which should have long ago become extinct. However, IBM's choice of it for its first personal computer sealed its fate as the most popular processor architecture.

The X86-64 architecture extends the 32-bit X86 to use 64-bit registers, while retaining backwards compatibility with 32-bit X86 code.

In the X86-32 model, with 32-bit registers, ints, longs and pointers are all 32 bits. On the 64-bit model, with registers now widened to 64 bits, there are different data type width models. The one which is used on UNIX systems is sometimes called "LP-64", meaning longs and pointers are 64-bit, but ints remain 32.

Below is a summary of the X86/X86-64 architecture The reader is detoured to the official reference manuals for full details.

X86 Register Model

When referring to X86 registers, their size is implied by a prefix. For example, there is a 32-bit register called EAX. The least significant 16 bits of that register are called AX. It is possible to refer to the least significant byte as AL and the next most significant byte as AH. In the 64-bit X86-64 instruction set, the 64-bit version of EAX would be called RAX. We will consider the 32-bit model.

The register model of X86 is convoluted and archaic, making efficient register allocation and instruction selection a challenge. The following general-purpose registers are typically used for holding temporary values, general integer computation, etc.
• %eax: The "accumulator". Many instructions use %eax as an implied operand.
• %ebx: The "base register" (not to be confused with %ebp).
• %ecx: The "counter register".
• %edx: The "data register".
• %esi: Source register for string operations
• %edi: Destination register for string operations

The following special registers are used for control flow:
• %eip: The "instruction pointer", aka the Program Counter. At the time of instruction execution, %eip contains the address of the next instruction to be fetched. A branch instruction modifies %eip and causes the next instruction to be fetched from that new address.
• %esp: The stack pointer.
• %ebp: Typically used in the C / assembly language convention for the stack frame "base pointer". Aka the "frame pointer".
• %eflags: The flags register. It contains the condition code flags (carry, parity, BCD adjust, zero, signed, overflow) as well as a number of flags and control bits which are generally used only by the operating system (e.g. the Interupt Enable Flag).

The X86 addressing scheme is based on an obsolete concept known as "segment/offset" addressing. In all modern operating systems, program addresses are linear, and the segmentation is basically ignored. The register model contains the registers %cs, %ds, %ss which are initialized by the operating system and should not be touched. They are what enable code, data and stack accesses to work. Additional segment registers %es, %fs and %gs are general-purpose and, because a linear addressing model is being used, could be employed as general-purpose scratch registers, subject to some restrictions as to which registers may appear in which instructions. However, both the %fs and %gs registers are used by the operating system and the standard library, and should be avoided. In addition, they are 16-bit registers so their utility as general-purpose registers is dubious.

There are many additional registers in the X86 model, but they are either used by the operating system only, or are for instructions that are beyond the scope of this introduction, such as floating point, MMX, and SSE instructions.

On X86-64, there are additional general-purpose registers `%r8 - %r15`.

Addressing Modes

There are a number of addressing modes which are used to specify where to find or put the operands of an instruction:
• Register Direct: Specify the register name with a % prefix, e.g. %eax.
• Immediate: The immediate value must be prefixed with the dollar sign, e.g. $1
• Memory Absolute: The absolute address of the operand is specified without a prefix qualifier. E.g. `movl $1,y` moves the immediate value 1 into the memory address which is associated with the linker symbol y.
• Base-index (Register Indirect with offset): The X86 has a handy mode for accessing elements of an array. The syntax is `disp(%base,%index,scale)` . The address of the operand is computed as `addr=base+index*scale+disp`. The base and index may be any of the general-purpose registers (eax, ebx, ecx, edx, ebp, esi, edi, esp (not allowed as the index)). The displacement is a 32-bit absolute address. The scale factor may be 1, 2, 4 or 8. Some of these parameters may be omitted, forming simpler addressing modes. E.g. in `movl $1, (%eax)` the eax register contains a pointer to a memory location, into which the immediate value 1 is moved.

```
int a[10];
int *p;
struct X {int a,b;} *px;

f()
{
 int i;
          i=g();
          a[i]=1;
          p[i]=2;
          px[i].b++;
}


f:
          pushl     %ebp
          movl      %esp, %ebp
          subl      $8, %esp
          call      g
          movl      $1, %edx
          movl      %edx, a(,%eax,4)    #a[i]=1
          movl      p, %edx
          movl      $2, (%edx,%eax,4)   #p[i]=2
          movl      px, %edx
          incl      4(%edx,%eax,8)                #wow, very optimized!
          movl      %ebp, %esp
          popl      %ebp
          ret
          .size     f,.Lfe1-f
          .comm     a,40,32
```

```
        .comm    p,4,4
        .comm    px,4,4
```

X86 is generally a 2-address architecture, meaning that one of the operands is both a source and a destination. There are many combinations of src/dst addressing modes including some odd restrictions. Generally speaking, most opcodes allow register/register, register/immediate, register/memory or immediate/memory combinations. Memory/memory is generally not allowed.

X86/32 Function Calling Convention

We will discuss what the Intel documentation calls the CDECL convention for procedure calling, as that is what is used in the C/UNIX world. Other calling conventions do exist. In the X86/UNIX architecture, all arguments to a function are pushed on the stack, and the return value is returned in the %eax register. If the return value is 64 bits (long long), it is returned in the register pair %edx:%eax, with the %edx being the most significant 32 bits. Floating point return values are returned in the floating point registers (floating point operations are not discussed in these notes). Returning a struct requires a special convention, discussed below.

Recall that %esp is the stack pointer, and the stack grows towards low memory. The PUSH instruction predecrements the stack pointer, then writes the value to (%esp). Likewise, POP reads from (%esp) and then postincrements %esp. Arguments in C are pushed to the stack in right-to-left order. Therefore, just before issuing the CALL instruction, the leftmost argument is on the top of the stack. This convention allows variadic functions to work properly. The callee does not need to know in advance (at compile time) the exact number of arguments which will be pushed. It is able to retrieve the arguments left-to-right by positive offsets from %esp.

The CALL instruction pushes the value of %eip, thus on entry to a function (%esp) contains the address of the instruction to which control should return (i.e. the instruction after the CALL). The first thing any function does is set up its local stack frame. Let's look at an example:

```
f1()
{
        f2(2);
}

f2(int b)
{
int a;
        a++;
        b--;
        return 1;
}

f1:
        pushl   %ebp
```

```
        movl    %esp, %ebp
        subl    $8, %esp                !one arg slot + one padding slot
        movl    $2, (%esp)              !put arg onto stack
        call    f2
        leave
        ret

f2:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp               !extra space for alignment
        incl    -4(%ebp)                !access local var a
        decl    8(%ebp)                 !access param b
        movl    $1,%eax                 !return value
        leave
        ret
```

The %ebp register is the frame pointer, and will be used to access both local variables and parameters. Its value must be preserved so the first action is to save it on the stack. Then the stack pointer is decremented to create room for local variables. In our example, function g has one local variable which takes up 4 bytes. The %ebp contains the value of the stack pointer after saving the old %ebp. Therefore 4(%ebp) is the return address, (%ebp) is the saved %ebp, and the first parameter is 8(%ebp). Parameters will be at positive offsets from %ebp and local variables will be at negative offsets. Generally speaking, the local variables mentioned first in a function will have the lowest memory address (i.e. highest negative offset from %ebp), but that behavior is not guaranteed.

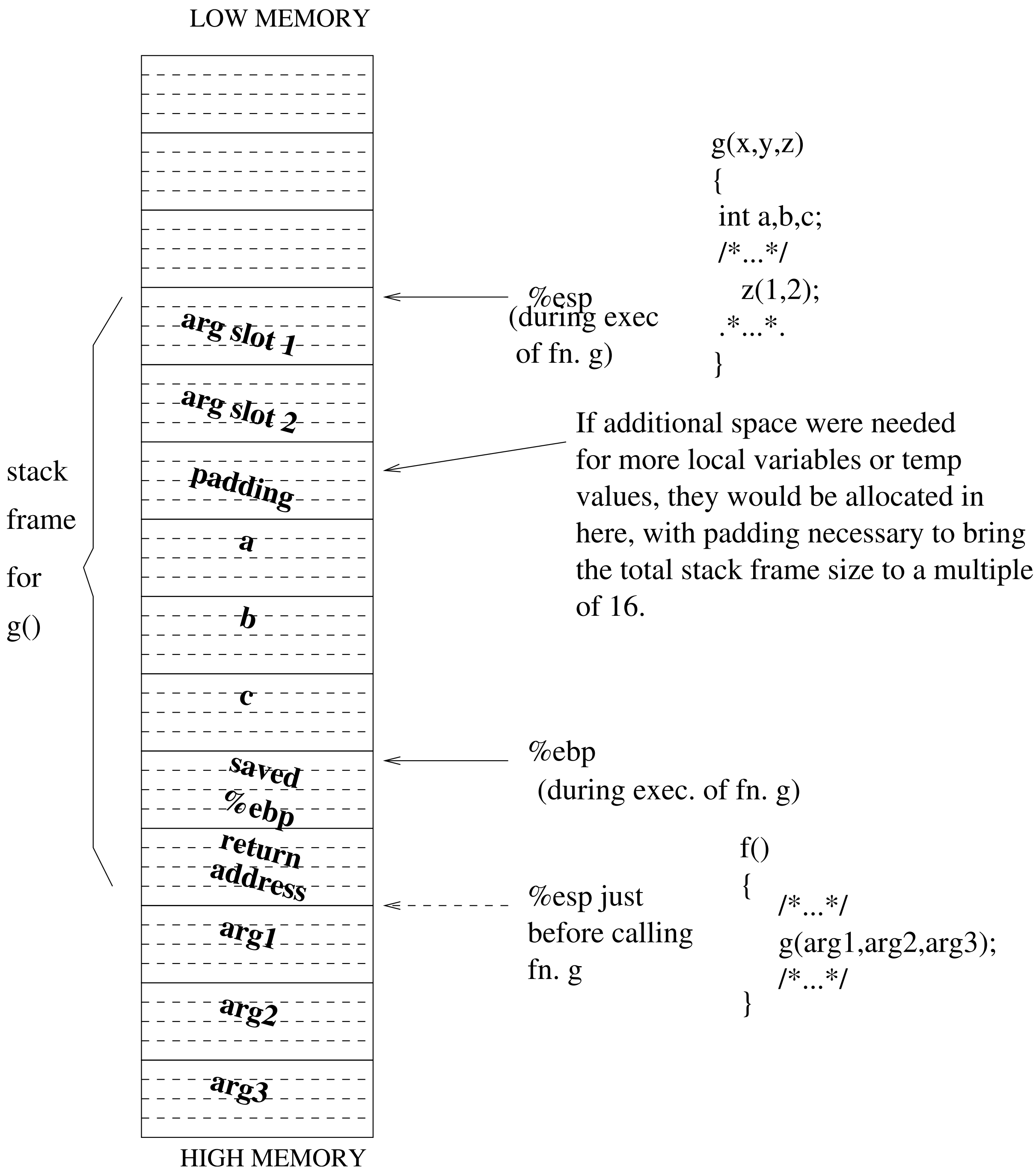
When a function call is made, arguments can be pushed on the stack in right-to-left order, using the `pushl` instruction. After the `CALL` instruction, an `addl $X,%esp` would be needed to adjust the stack pointer and reverse the effects of the previous pushes. Alternatively, one could determine during code generation which function call (within the function being generated) has the highest number of arguments. The number of bytes thus required for passing arguments can be added to the total local stack frame size, as if these "argument slots" were hidden local variables. Then the arguments can be passed via `movl OFFSET(%esp)`, in any order desired, and there is no need to adjust the stack pointer after the call. This is the approach that gcc takes, and that is seen in the asm code above.

Upon leaving a function, the LEAVE instruction is used, which performs two operations: %ebp is moved into %esp, thus restoring the stack pointer to its value just after the base pointer save on entry, then %ebp is popped from the stack. I.e. LEAVE is equivalent to

```
movl     %ebp,%esp
popl     %ebp
```

Now everything is restored, and the RET instruction pops the return address from the stack and resumes execution in the caller.

If the compiler chose to use any registers which are callee-saves, we would see pushes of those registers on entry and corresponding pops on exit.



Note that classic C widened char and short before passing. C89/C99 does not do this if there is a prototype for the called function, e.g. void f(char,char). However, the caller still passes the arguments as 32 bit values, with the more significant bits zeroed (for unsigned) or sign-extended. Floats are passed as 32-bit, doubles are 64-bit.

When passing whole structs as arguments, they are pushed onto the stack in reverse order. So:

```
 struct s {int a,b;} s1;
```

```
 fn(s1);
```

is the same as

```
fn(s1.b,s1.a);
```

Since the leftmost parameter winds up at the lowest memory address, this preserves the addressability of the entire structure on the stack in the correct order.

If a function returns a structure, the CALLER must allocate an area on its stack frame to hold the return value. Then it passes a hidden first parameter which is the address of this area.

```
  struct s *g()
  {

  }

  s1=g();
```

is the same as

```
  struct s temp;
          g(&temp);
```

X86-64 Function Calling

Under the 64 bit architecture, the first 6 integer arguments are passed in registers, rather than on the stack. Arguments are placed in left-to-right order in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9. If there are additional arguments, they are put on the stack right-to-left, i.e. with the right-most argument at the highest memory address, just like X86-32. If structs are passed as arguments, they are always placed on the stack. The integer return value is in the %rax register.

This hybrid register/memory argument passing model introduces some complexity with variadic functions, aka <`stdarg.h`>. GCC implements `stdarg` as a compiler built-in.

X86-64 Global Variables

There is an odd limitation in the X86-64 instruction set: the absolute addressing mode is not supported for 64-bit addresses. To access a memory operand, a register indirect addressing mode must be used.

```
extern int i;

f()
{
          i=2;
}

f:
        pushq   %rbp                                    #Prologue, save base pointer
```

```
movq    %rsp, %rbp             #Set new base pointer
subq       $32, %rsp           #Create stack frame
movl    $2, i(%rip)            #Program Counter Relative mode
leave
ret
```

There will be a 32-bit "hole" in the `movl` opcode which will be a program counter relative relocation type (similar to the example of the CALL opcode earlier in this unit). At link time, when the address of symbol i has been resolved, this hole will be filled with the i's address, minus the address of the hole itself.

This introduces a limitation that code and data must fall within the same contiguous +/-2GB memory region at run time. gcc calls this the "small" memory model and it is the default option. To use a "large" memory model where code and data may be anyplace within the 64-bit address space, invoke gcc with the `-cmodel=large` option. Different opcodes are now used:

```
movabsq    $i, %rax            #Move 64 bit immediate value to rax
movl    $2, (%rax)             #Register indirect
```

Caller/Callee saves

It is the case for any architecture and operating system that there is a function calling "convention" which specifies how arguments are passed and returned, and how registers may be used. This convention dictates which of the registers are expected to survive a function call, and which ones may be used as "scratch" registers, and are therefore expected to be volatile across function calls. Another way of saying this is there are caller-saved registers (the scratch registers.. if the caller wants to keep a value in there through a function call it must explicitly save it) and callee-saved registers (if a function wants to use one of these registers it must explicitly save it on entry and restore it before returning).

In the X86 architecture under UNIX, the %eax,%ecx,and %edx registers are scratch registers (caller-saves). You will find that the compiler tends to put short-lived values in these registers. Of course the %eflags register is also expected to be modified by a function call. The %ebx,%edi,%esi and %es [CAUTION: this is a 16-bit register] registers are callee-saved. The compiler may use these for longer-lived values (such as local variables which are assigned to a register for all or part of the function to improve speed). However, if one of these registers is used by the compiler, it must emit code to push it on the stack on entry, and pop it on return.

On X86-64, the caller-save (scratch) registers are %rax,%rcx,%rdx,%rsi,%rdi, and %r8-%r11, while the callee-save (long-term) registers are %rbx, %r12-%r15. Note that %rsi and %rdi are caller-save on 64 bit, whereas they were callee-save on 32-bit. This is because they are used for argument passing on 64 bit.

X86 General-Purpose Register Summary

-32 reg	Role	Notes	-64 reg	Role	Notes
%eax	Scratch	fn retval	%rax	Scratch	fn retval
%ebx	Long-term		%rbx	Long-term	
%ecx	Scratch		%rcx	Scratch	arg #4
%edx	Scratch	longlong ret	%rdx	Scratch	arg #3
%edi	Long-term		%rdi	Scratch	arg #1
%esi	Long-term		%rsi	Scratch	arg #2
%es	Long-term		N/A		
			%r8	Scratch	arg #5
			%r9	Scratch	arg #6
			%r10	Scratch	
			%r11	Scratch	
			%r12	Long-term	
			%r13	Long-term	
			%r14	Long-term	
			%r15	Long-term	

X86 Alignment

The X86 architecture does not generally impose alignment restrictions on either instruction or data fetches (in fact, with single-byte opcodes, instructions are rarely aligned). The processor will simply handle misaligned accesses by performing multiple fetches from memory and shifting the bits around. However, this can introduce inefficiency. E.g. on a 32-bit system, fetching a 4-byte int at an address which is not a multiple of 4 results in two memory accesses, whereas it is a single address if it is aligned. Therefore, the compiler will introduce padding to keep things aligned:

```
                    X86-32                X86-64
type               size      align               size      align
char               1         1                   1         1
short              2         2                   2         2
int                4         4                   4         4
long               4         4                   8         8
long long          8         4                   8         8
pointer            4         4                   8         8
float              4         4                   4         4
double             8         4                   8         8
long double        12        4                   16        8
```

Furthermore, for performance reasons having to do with cache lines, it is advantageous to have the stack frame aligned to a 16-byte boundary. I.e. during the execution of a function, `%esp` is a multiple of 16. On entry to the function, the compiler emits code to create enough stack frame area not only for local and temporary variables, but also for passing arguments to any functions which are called from within the function in question, plus padding if necessary to bring the total stack frame size to a multiple of 16. The compiler then uses `movl ARG,SLOT(%esp)` to place the arguments on the stack, rather than `pushl`. This means that upon return from the called function, there is no need to adjust the stack pointer.

APPENDIX: SPARC Architecture

As an appendix, we'll take a look at an example of RISC architecture. These notes were prepared for the SPARC architecture, which at one time enjoyed significant market share among Sun Microsystems workstations and servers.

SPARC, as with most RISC designs, is shockingly different from X86. Almost all instructions are 3-address, with src1, src2 and dst. There is an emphasis on register-to-register operations, with limited memory addressing modes. Instructions execute in constant time (one clock cycle), and are of fixed length (32 bits), which improves pipeline performance.

Register Model

There are 32 registers, r0-r31. These are broken into four groups:

```
r0-r7     == g0-g7           Global registers (g0 is /dev/null)
r8-r15    == o0-o7           Outs
r16-r23   == l0-l7           Locals
r24-r31   == i0-i7           Ins
```

The names %i0, etc. are aliases for %r24, etc.

Addressing Modes

SPARC has a limited number of addressing modes. Almost all instructions are 3-address, with two explicit source operands and one explicit destination (contrast with the 2-address X86 with an implicit operand). Exceptions to this are instructions where it doesn't make sense to have 3 operands. Any of the operands may be a register. Because there are 32 addressable registers, 5 bits are required to specify a register. Most instructions also allow just one of the source operands to be a 13-bit immediate value. We will see how 32-bit immediate values are handled later. Most instructions do not allow direct access to memory operands; they must go through a register using special load/store instructions.

Register Windows

One of the most interesting aspects of the register model is register windowing. The register window is the set of 24 registers r8-r31. The processor contains a "register file" which is a large array of registers. The global registers are always available, but the other registers are accessed through the 24-register window, which moves over the register file based on a hidden register (manipulated by the operating system) called CWP (current window pointer).

Whenever a function is entered, a SAVE instruction is executed, which moves the CWP ahead by 16 slots. Since the register window is 24 slots, this creates an 8-slot overlap, with the result that registers %o0-%o7 in the caller refer to the same register as %i0-%i7 in the callee. Likewise, when the function returns, the RESTORE instruction moves the window back, and the values that were in %i0-%i7 in the callee are accessible as %o0-%o7 in the caller.

This fact is used to great advantage to allow function calling and return without ever having to touch the stack memory. %o0-%o5 are used to pass the first 6 parameters of a function, with %o0 being the leftmost. (If there are more than 6 parameters, the extra ones are pushed on the stack. Studies of many lines of C code showed this happens in less than 2% of functions).

The CALL instruction stores its own address in register %o7 in the caller window, which becomes %i7 in the callee window (after the callee executes the SAVE instruction.) The RET instruction, which is executed after a RESTORE, jumps back to %i7+8, i.e. the next instruction after the CALL (and the delay slot instruction, see below). Register %o6 is aliased to %sp, the stack pointer. In the callee, %i6 is then automatically the old stack pointer, and is aliased to %fp (the frame pointer). The SAVE instruction, in addition to moving the register window, can be used to decrement %sp (in the callee's new window) to create the local stack frame. The RESTORE instruction, by virtue of its window rollback, restores the old %sp and %fp implicitly.

The number of registers in the file is of course finite, and typically a small number, e.g. 32 register windows. When nesting reaches this static limit, the SAVE instruction has no place to move the register window, because it would collide with the oldest window. This results in a "spill trap", which the operating system handles. Executing a FLUSHW instruction flushes the current register window to the top of the stack. This instruction can also be executed by ordinary programs. E.g. a debugger needs to gain access to the complete set of register windows. It must execute a FLUSHW on behalf of the program under observation so those values become visible on the stack.

Because a function can not predict when such a register window spill will happen, it must reserve a chunk of space at the top of the stack so there will be a place to spill its register window if needed. This spill area is added to the stack frame size requirements.

It follows from the register window mechanics that the compiler always has the %l0-%l7 local registers available. It does not have to worry about their being overwritten by a called function, nor does it need to save and restore them explicitly. The global registers

%g1-%g7 are considered scratch registers, i.e. caller-saved registers. It is expected that the g registers will be destroyed across function boundaries. Likewise the %o0-%o5 registers can be used as extra scratch registers when they are not actively being used to pass function arguments.

Register g0

The %g0 register always reads as 0, and writes to it have no effect. Since the number of operands is fixed, it is useful when it is desired to discard the result of an operation, or when an immediate 0 is needed but it isn't convenient to use the 13 bit immediate mode.

Accessing Memory

There are only two ordinary instructions which access memory, LD and ST (LOAD/STORE). The value to be moved is contained in a register. The address in memory to be accessed is also contained in a register. Since LD and ST have only 2 operands, there is room within the 32-bit instruction to allow a 13-bit immediate value (offset) to be added to the register specifying the memory address. While this restrictiveness surrounding memory access may seem inefficient, recall that the RISC philosophy is to reduce the instruction set to its bare essentials. On a CISC machine such as X86 with its indirect scaled offset plus displacement addressing mode, the processor takes just as many steps to perform the address calculation, they are just hidden in internal registers. In addition, by isolating memory access to specific instructions, compiler optimizations are actually easier.

```
f()
{
int a;
        a++;
}

f:
        save %sp,-120,%sp    !120 bytes for stack frame
        ld [%fp-20],%o1              !Load a into register o1
        add %o1,1,%o0                !Not optimized, could have stored back in o1
        mov %o0,%o1
        st %o1,[%fp-20]              !Store result back in local variable
          ret
          restore                    !Delay slot, see comment in text
```

Loading 32-bit constants

The structure of the 32-bit instruction words does not lend itself to getting 32-bit (or 64-bit for that matter) constants into registers. It was found from program analysis that most integer constants used in C programs are small, and therefore the provision of a 13-bit immediate field, while strange, satisfies most cases.

However, to access a global variable requires loading an absolute 32 bit value into a register (or 64 bits if in the 64 bit model). This requires two instructions in SPARC. The SETHI instruction is a special case. It specifies a destination register and 22 bits of constant, which are placed in the most significant bits of the register. Following this, an OR with a 13-bit immediate as one source, and the register as the other source and destination, effects the 32-bit constant load. It may also be possible to use the 13-bit offset addressing mode to accomplish the same effect:

```
f()
{
extern int a,b;
        a=b;
}


f:
        save %sp,-112,%sp
        sethi %hi(a),%o0
        sethi %hi(b),%o1
        ld [%o1+%lo(b)],%o2
        st %o2,[%o0+%lo(a)]
```

Note the unusual syntax %hi(symbol). This is not a register at all, but merely an assembly-language macro that evaluates the most significant 22 bit of its value. Likewise %lo is the least significant 13 bits. Also note the use of the register+offset indirect addressing mode in the LD and ST instructions.

When executing under the 64-bit model, the code to access a global variable is even hairier:

```
        sethi   %hh(a), %g1
        sethi   %lm(a), %g4
        or      %g1, %hm(a), %g1
        sllx    %g1, 32, %g1
        add     %g1, %g4, %g1
        or      %g1, %lo(a), %g5
        sethi   %hh(b), %g1
        sethi   %lm(b), %g4
        or      %g1, %hm(b), %g1
        sllx    %g1, 32, %g1
        add     %g1, %g4, %g1
        or      %g1, %lo(b), %g1
        ld      [%g1], %g1
        st      %g1, [%g5]
```

Control Flow Instructions

It is possible to jump to an absolute address which is contained within a register.

However, most flow control instructions utilize a program counter relative addressing mode. The conditional branch instructions contain a displacement which is between 16 and 22 bits (depending on the form of the instruction). This displacement is interpreted as a signed number of 4-byte words. Since branches in C occur within a function, the branch target is usually close by, and this amount of displacement is more than adequate. CALL instructions, which are used to call another function, have 30 bits of displacement. This does impose a limit on how the address space of a program is laid out. Functions can not be more than 2**31 bytes away from each other in either direction.

The Delay Slot

When a SPARC processor is evaluating a branch instruction, the instruction which is located physically at the next memory address after that branch instruction has, because of the pipeline, already been fetched and decoded. Under CISC architecture, that instruction would be thrown away if the branch is taken, but SPARC (and many other RISC architectures) take advantage of it (it helps that all instructions are the same length and execution time) and execute this "delay slot" instruction, regardless of whether the branch is taken or not. The delay slot instruction is executed as the branch target is being fetched and decoded, and takes effect before that target instruction. This can lead to code which, when represented linearly, is difficult to follow:

```
f(a,b)
{
    if (g()>5) return 3; else return 4;
}

f:
        save %sp,-112,%sp
        call g,0
        mov 3,%i0
        cmp %o0,5
        ble,a .LL5
        mov 4,%i0
          ret                         !same as jmpl       %i7+8,%g0
          restore
```

Following the CALL to function g, the MOV instruction is executed in the delay slot. Because it takes effect before the first instruction of the target g, the %i0 still refers to the register in f’s window. Likewise, the MOV 4,%i0 takes place in the delay slot of the conditional branch instruction. The ,a following the branch opcode ble means that the delay slot instruction is anulled if the branch is not taken (normally the delay slot instruction is always executed). So if the return value from function g is less than or equal to 5, then the correct return value from f (4) is already in the %i0 register when the RET instruction is executed. If g() is greater than 5, then the correct value 3 was previously in the %i0 register. Also note the use of the RESTORE instruction in the delay slot. Because it completes before the transfer of control back to the caller of f, the register window is properly restored. If there is no useful work to be done in a delay slot, it must be filled with a NOP instruction.

Note that the above example is still not optimal, because the compiler was targetting the earlier SPARC V8 instruction set. The SPARC V9 instruction set includes conditional move instructions, which can often eliminate branches entirely:

```
f:
        save    %sp, -112, %sp
        call    g, 0
        mov     3, %i0
        cmp     %o0, 5
        return  %i7+8
        movle   %icc, 4, %o0
```

Here the MOVLE instruction conditionally moves the value 4 into %o0 if the condition codes indicate less than or equal (the %icc identifies this as a 32-bit operation), and this instruction executes in the delay slot of the RETURN instruction (which includes RESTORE semantics). Because the register window rollback has already taken place during the RETURN instruction, register %o0 in the MOVLE instruction refers to the caller's %o0, i.e. the return value slot.

Delay Slot Annul and Branch Prediction

When a BR branch opcode has the ,a suffix, the delay slot is annulled if the branch is **not taken**. This allows the compiler to have a better chance of finding a useful instruction to move into the delay slot, which will only be executed when the branch is taken.

The compiler can also add a branch prediction suffix to the opcode. The ",pt" suffix means that the branch is likely to be taken, while ",pn" means it is likely to be not taken. Branch prediction provides a helpful hint to the hardware in modern processors where several instructions may be in play at any given time.