The Target Environment

In units 1-4 we’ve seen how the compiler understands "what it says" and "what it means," along with methods of representing that meaning. In Unit 5 we looked at one way of expressing "how to do it" through abstract machines (one-address stack code and quads). While it is possible to write a "compiler" proper which stops at quads, and allows other tools to take it to the finish line (indeed, that is the approach taken in the Clang/LLVM framework), it is helpful to understand how we get to the final, target code environment. We’ll take it to assembly language, looking at X86 as an example.

Before we begin, it is crucial to understand that "The Target Environment" goes beyond just the CPU and its instructions. All targets have what is known as the **Applications Binary Interface** (ABI) which sets the "rules of the field" for how binary applications (those distributed in native binary machine code form, i.e. EXE or a.out files) interface with the operating system, with the system libraries, and with other binary code.

Overview of the compiler/assembler/linker toolchain

Although some compilers take the source language directly to machine code, most utilize the system assembler. Therefore, a C compiler will transform `.c` files into assembly language files with a `.s` suffix. The system assembler is traditionally called `as`.

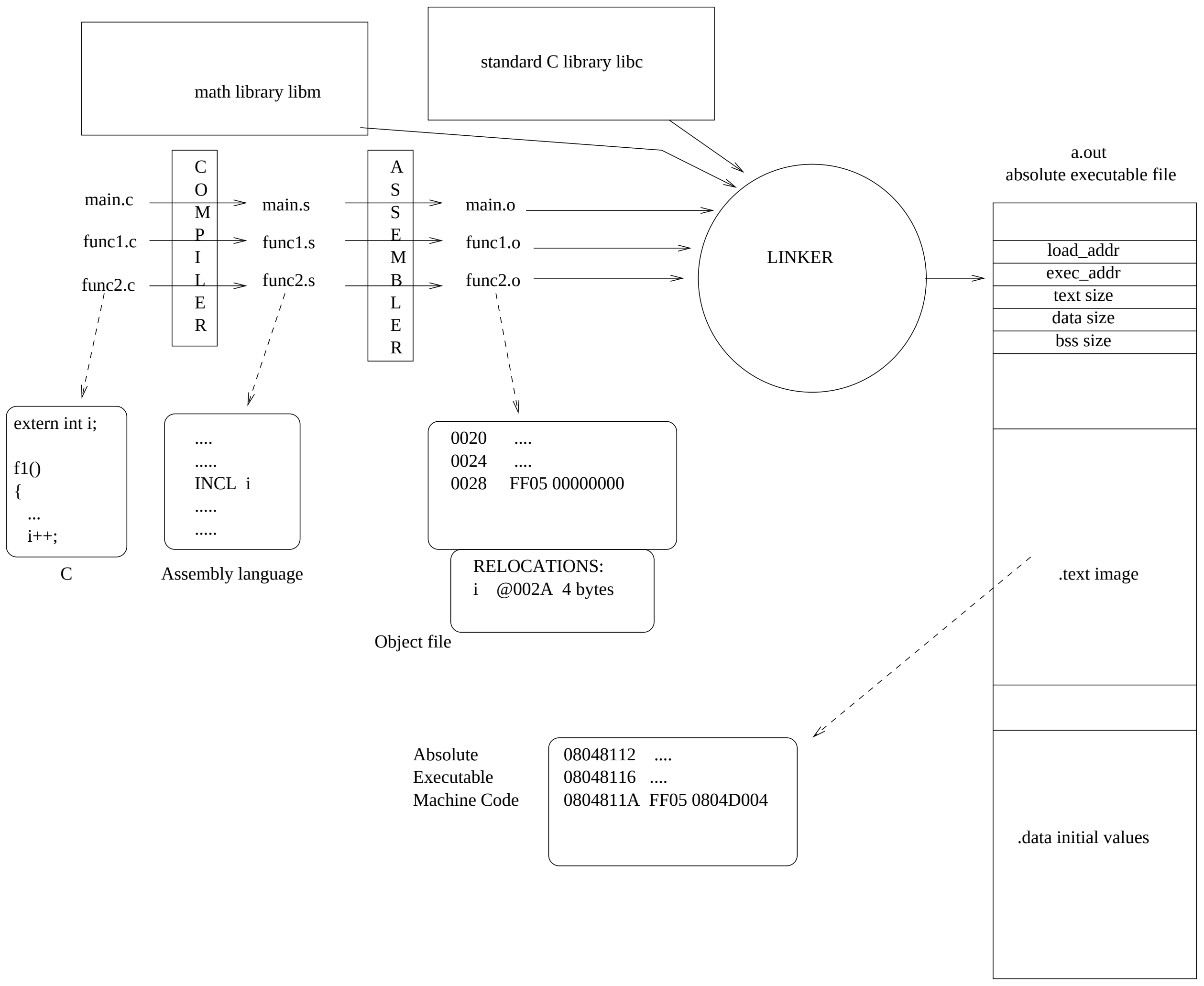
The assembler takes the `.s` file and creates a **relocatable object file**, which has a `.o` suffix. It is typical with larger projects to divide the code among multiple source files. When part of the project is changed, only the source file or files containing this change need to be re-compiled and 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 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). In this unit, we’ll see two different styles of libraries: **static libraries** which have a `.a` suffix, and **dynamic / shared libraries** which have a `.so` suffix.

Finally the linker program takes the various object files and libraries and combines them into a monolithic, coherent, executable file. At this time, addresses that were previously expressed as symbols are given specific run-time addresses. The resulting executable output by the linker is called, by default, `a.out` where the "a" refers to the Absolute address assignment to symbols.

The linker is called `ld` on UNIX systems. This harkens back to some very old terminology where the linker was often called a "loader" because it produced the "load deck" (of punched cards) which was the executable program



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 the `a.out` (if there are no fatal errors in compilation). 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.

Before we continue further with the assembler/linker part of the toolchain, let's get an in-depth understanding of a particular architecture. We'll look at X86.

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 of the form `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. Instructions have either zero operands, a single operands, two operands, or three operands. After the operands, a comment can be placed, started with a # character. This comment is ignored by the assembler. A line started by a # is also ignored as a comment.

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 current value of the "cursor" can be referenced within any assembly language expression by using a single dot (period) character.

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.

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 descendant 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.

X86 is, for the most part, a 2-address architecture. Instructions have just two operands. Therefore for instructions such as ADD, one of the operands acts as both one of the source operands and the destination.

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 first 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`. It is possible (although rare) to refer to the least significant 32 bits of these registers by the names %r8d, %r9d, etc.

Addressing Modes

There are a number of addressing modes which are used to specify where to find or put the operands of an instruction. 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, although typically one-address memory operands are ok (e.g. `incl xyz`). Consult the X86 documentation for each opcode to determine the allowable address mode combinations.

• 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 A symbol may also be used such as $a.

• Memory (Absolute): The absolute address of the operand is specified without a prefix. E.g. `movl $1,y` moves the immediate value 1 into the memory address which is associated with the linker symbol y. A numeric value could also be used but this is not typical, since memory addresses are determined by linker symbols.

• Register Indirect: The operand is at a memory location, the address of which is in a general-purpose register. This is similar to a pointer dereference in C. For example, the syntax `(%edx)` interprets the %edx register as the memory address of the operand.

• Scaled Register Indirect with Displacement: The X86 has a handy mode for accessing local variables, elements of a struct, or elements of an array. The syntax is `disp(%base,%index,scale)` . The address of the operand is computed as `addr=disp+base+(index*scale)`. The base and index may be any of the general-purpose registers with the odd exception that %esp is not allowed as an index. The displacement is a 32-bit value. The scale factor may be 1, 2, 4 or 8. Some of these parameters may be omitted, forming simpler addressing modes:

`disp(%base)` is a simple register indirect with offset and is commonly used for local variables.

`disp(,%index,scale)` skips the use of a base register. It could be used to access elements of a global array.

`(%base,%index,scale)` The displacement is omitted.

Below is an example of using these addressing modes:

```
//ary.c
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++;
}


         call     g                   #%eax contains return value now
         movl     $1, a(,%eax,4)      #a[i]=1.  Note no %base register in this addrmode
         movl     p, %edx
         movl     $2, (%edx,%eax,4)  #p[i]=2.  Note no displacement
         movl     px, %edx
```

```
          incl      4(%edx,%eax,8)               #Full addrmode, very optimized!
```

X86 Branches

Branch instructions in X86 use a program-counter-relative (PCR) addressing mode, although that is transparent to the assembly language programmer, because the assembler interprets the parameter as an absolute address, and translates to an 8, 16, or 32 bit program counter (%eip or %rip) relative offset as appropriate.

Conditional branches test the **condition codes** flags which are carried in the eflags register. There are often multiple aliases for a given condition code. The X86 condition codes (as used in the conditional jump instruction) are:

Opcode	Aka	Meaning	Usage
ja	jnbe	Jump if above/not below or equal	if (au>bu) UNSIGNED
jae	jnc/jnb	Jump if above or equal / no carry / not below	if (au>=bu) UNSIGNED
jb	jc/jnae	Jump if below / carry / not above or equal	if (au<bu) UNSIGNED
jbe	jna	Jump if below or equal / not above	if (au<=bu) UNSIGNED
je	jz	Jump if equal / zero	if (a==b) OR if (!a)
jne	jnz	Jump if not equal / not zero	if (a!=b) OR if(a)
jg	jnle	Jump if greater than / not less than or equal	if (a>b) SIGNED
jge	jnl	Jump if greater than or equal / not less than	if (a>=b) SIGNED
jl	jnge	Jump if less than / not greater than or equal	if (a<b) SIGNED
jle	jng	Jump if less than or equal / not greater than	if (a<=b) SIGNED
js	---	Jump if signed (negative)	if (a<0) SIGNED
jns	---	Jump if not signed (positive)	if (a>0) SIGNED

(Note: There are additional Jcc opcodes but they are of little use to the compiler. E.g. JPE is "Jump if Parity Even" which uses the obscure Parity Flag in the eflags register. The Overflow Flag can be tested directly with JO and JNO.)

There are several ways in which the condition codes are set. The most typical is with the `cmp` instruction. Confusingly, this subtracts the first operand (normally the src operand) from the second (normally the dst) but discards the result instead of writing to dst. The subtraction sets the condition codes flags.

```
//jmp1.c

int f()
{
extern int a,b,c;
          if (a<c) goto X;
          b=1;
X:
          c=2;
}
```

```
f:
         movl     c, %eax            # cmpl c,a not possible!
         cmpl     %eax, a
         jl       .L2
         movl     $1, b
 .L2:
         movl     $2, c
```

Note that the cmp is done in two steps, because memory,memory operands are generally not allowed in X86. Getting the "polarity" of compare and branch correct can be tricky!

Another means of setting condition codes is via the `test` instruction, which performs a bitwise AND operation between the two operands (both of which are considered source operands) and sets some of the condition codes (the sign and zero flags). This instruction can be used for bitwise testing or, if the two operands are the same, to test a single value.

```
//jmp.c
int f()
{
extern int a,b,c;
         if (a<0) goto X;
         b=1;
X:
         c=2;
}


f:
         movl     a, %eax
         testl    %eax, %eax
         js       .L2                #Jump if a was negative (signed)
         movl     $1, b
 .L2:                                #This corresponds to label X:
         movl     $2, c
         ret
```

Most arithmetic instructions such as `add` also set condition codes. An optimized compiler can take advantage of this:

```
//jmp2.c

int f()
{
extern int a,b,c;
         if (a+b<0) goto X;
         b=1;
X:
         c=2;
}
f:
         movl     b, %eax
         addl     a, %eax
         js       .L2
         movl     $1, b
 .L2:
         movl     $2, c
         ret
```

Note how no explicit comparison to 0 is used.

A wild and crazy conditional branch is jecx (jrcx in 64 bit mode) which branches if the %ecx/%rcx register is zero. A related instruction is `loop` which decrements the %ecx/%rcx register and branches if the result is not zero. Hey, they called it the "counter" register for a reason!

X86-32 Function Calling Convention

A function (procedure) "calling convention" describes how arguments are passed to functions, how values are returned, and the division of responsibility for the stack and register saves between the caller and the callee. Different calling conventions exist for different languages based on the semantics of the language. When it comes to C compilers for the X86-32 architecture, the "CDECL" convention is always used on Unix systems, and is typically used on Windows systems. This is the one we'll discuss.

In this CDECL X86-32 convention, 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:

```
//call-stack.c
f1()
{
        f2(2,3);
}

f2(int b,int c)
{
int a;
        a++;
        b--;
        c+=4;
        return 1;
}

f1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp               #two arg slots + padding
        movl    $2, (%esp)              #put arg onto stack
        movl    $3, 4(%esp)             #second arg
        call    f2
        subl    $24, %esp               #restore stack pointer
        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
        addl    $4,12(%ebp)             #add 4 to param c
        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.

It is preferred that the stack pointer be aligned on a 16-byte boundary while the function is executing. This will cause the compiler to emit seemingly spurious `subl $something,%esp` instructions, and/or to subtract a greater value from the esp during the function prologue than necessary to cover the local variables.

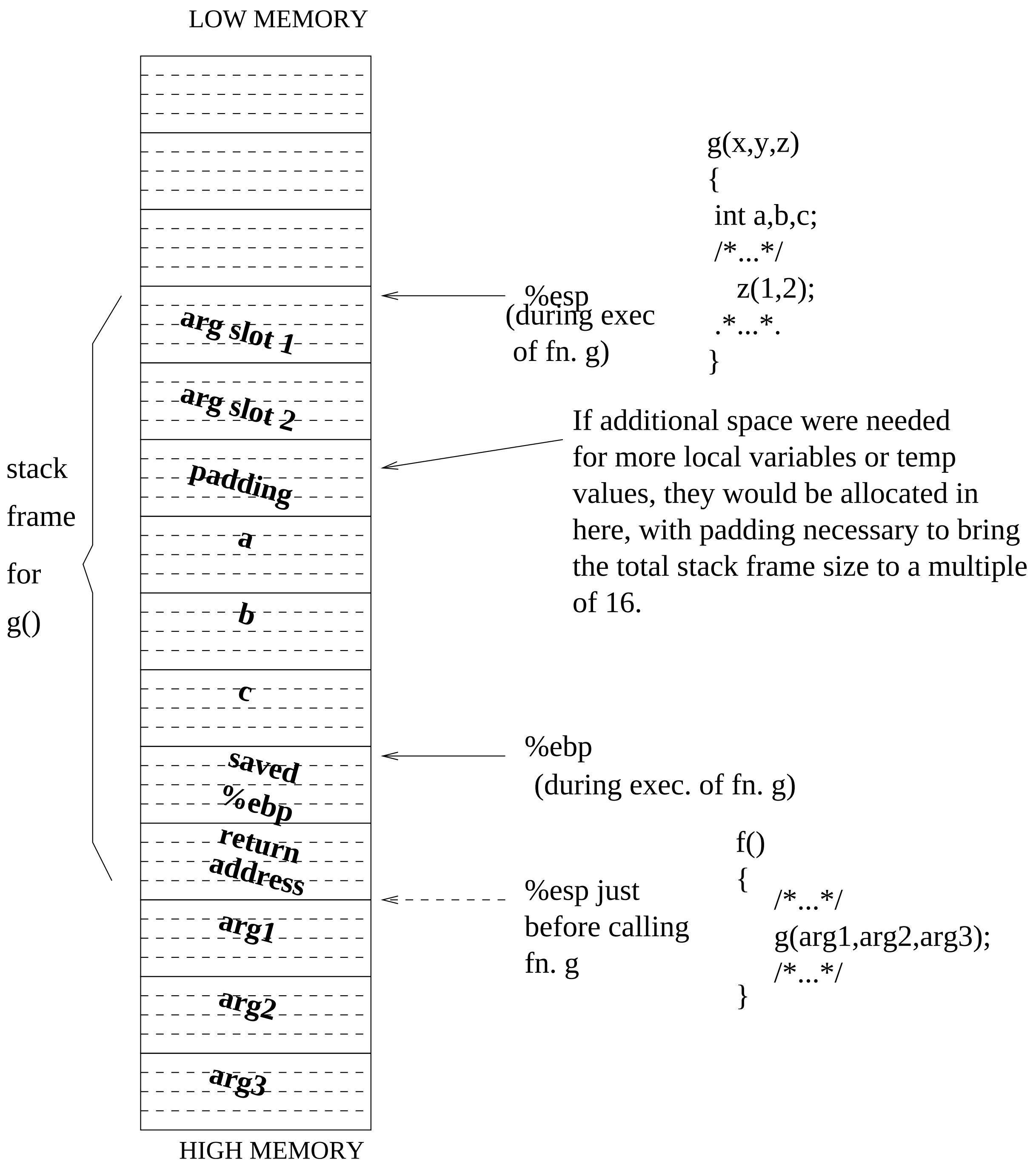
When a function call is made, arguments can be pushed on the stack in right-to-left order, using the `pushl` instruction. After the `CALL` instruction, an `addl $X,%esp` would be needed to adjust the stack pointer and reverse the effects of the previous pushes. Alternatively, one could determine during 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 some compilers prefer, 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 (see later in this unit), we would see pushes of those registers on entry and corresponding pops on exit.



```
g(x,y,z)
{
 int a,b,c;
 /*...*/
   z(1,2);
 .*...*.
}
```

If additional space were needed for more local variables or temp values, they would be allocated in here, with padding necessary to bring the total stack frame size to a multiple of 16.

```
f()
{   /*...*/
    g(arg1,arg2,arg3);
    /*...*/
}
```

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.

Passing a struct

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. Here is the X86-32 code

```
#struct-arg.s
          pushl   s1+4
          pushl   s1
          call    f2
          addl    $8, %esp   !adjust the stack back
```

Passing a big struct, using REP and MOVS

X86 has this wonderful feature known as the REP prefix.

```
struct BS {
          int a;
          int b[100];
          int c;
} bs;

f()
{
 void bff(struct BS);
          bff(bs);
}



#32-bit mode example
f:
          pushl    %edi                 #preserve Callee-save register
          movl     $102, %ecx           #count register
          pushl    %esi                 #preserve Callee-save register
          movl     $bs, %esi #source address of string operation
          subl     $420, %esp           #create room on the stack
          movl     %esp, %edi           #destination of string operation
          rep movsl          #copy struct to stack
          call     bff
          addl     $420, %esp           #fix stack
          popl     %esi                 #restore Callee-save register
          popl     %edi                 #restore Callee-save register
          ret
```

The `movsl` (Move String Longword) instruction moves a 4-byte value (a longword in

AT&T notation or a DoubleWord in Intel notation) from (%esi) to (%edi) and then increments both registers by 4. (For extra fun, set the Direction Flag DF in %eflags to 1 and it will decrement). By using the REP prefix ahead of MOVSL, we will repeat the MOVSL by %ecx times. This handy sequence allows memcpy to be implemented with just a few opcodes. However, the esi, edi, and ecx registers (or the rsi, rdi, rcx in 64 bit mode) must be used for this purpose.

Returning a struct

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 {int a,b;} s1;

struct s rs();

int fs()
{
 extern int a;
         a=rs().b; // Implemented as rs(&temp);
}

fs:
         pushl    %ebp                         #Usual prologue
         movl     %esp, %ebp                   # ""
         subl     $16, %esp                    #Make room for phantom var
         leal     -8(%ebp), %eax               #Address of phantom var
         pushl    %eax                         #Phantom arg
         call     rs
         movl     -4(%ebp), %eax               #access .b of returned struct
         movl     %eax, a                      #move to global var
         leave
         ret
```

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.

Because parameters come into a function in registers, we can't take their address directly. If a pointer to a parameter is needed, its value is copied to a phantom local variable slot at entry to the function.

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

Calling variadic functions X86-64

In this course we are not exploring floating point, nor some of the advanced features of X86, such as vector operation instructions. An odd thing which you might see when compiling code for modern X86-64 processors is the following:

```
int f()
{
          /* ... */
        /* No prototype for g */
        g(1);
}

f:
#... elided
          movl      $1,%edi             #first parameter
          xorl      %eax,%eax           #set eax to 0
          call      g
```

Why is the compiler setting eax to 0 before calling a function that has no prototype? The X86-64 ABI says that when calling a variadic function (or a function with unknown arguments) the EAX register contains the number of vector registers which are used in the argument list. It is EAX not RAX because this number fits easily within 32 bits. We aren't calling the function g with any floating point values, but just in case g could accept these, we need to set EAX accordingly. With a prototype in effect such as `void g(int);` this behavior is not seen, nor is it seen on X86-32.

CMOVcc and SETcc

X86 has a SETcc opcode which is effective when a relational operator is evaluated in an rvalue context:

```
//setcc.c
int f()
{
        extern int a,b;
        return a>b;
}

f:
        movl    b, %eax
        cmpl    %eax, a                #We can't just compare memory,memory
        setg    %al          #Set LSB of EAX to 1 if a>%eax
        movzbl  %al, %eax    #zero-extend to 32 bits
        ret
```

The `movzbl` opcode extends an 8-bit unsigned value to 32 bits by filling the more significant bits with 0. A similar `movsbl` opcode does sign extension.

The CMOVcc (Conditional move) opcode can sometimes help with ternary operators:

```
int f()
{
         extern int a,b;
         return a>b?5:200;
}

f:
         movl     b, %eax
         movl     $200, %edx        #Optimizer re-ordered instructions!
         cmpl     %eax, a           #Condition codes set here
         movl     $5, %eax          #MOV doesn't affect CC
         cmovle   %edx, %eax        #Move 200 to eax if a<=b
         ret
```

Both SETCC and CMOVcc use the same condition code names as Jump.

X86-64 32/64 conversions

The X86-64 architecture was designed to be compatible with X86-32. This can result in some confusing and counter-intuitive things! For example, the instruction `movq $1,%rax` is actually still a 32-bit instruction, despite the "q" suffix. The q tells us that the destination will be 64 bits wide, and thus we are selecting the entire RAX register. But the immediate source operand is still just 32 bits.

So is the result sign-extended? Unfortunately, the answer is "it depends" :(When the destination is the full 64-bit register, and the source operand is 32 bits (such as a 32-bit immediate value), the source is sign-extended before being operated upon. But when the destination register is 32 bits (specified with %eXX) then the upper 32 bits of that register are filled with 0 bits.

This behavior is not consistent with the 8 and 16 bit behavior, which harkens back to the old 8086 through 80286 processors. For example, `movw    $2,%ax` only affects the least significant 16 bits of the eax/rax register and does neither 0-filling nor sign extension!

```
//bigarg
int bigarg()
{
 void fff(long long);
 void ggg(long long);
        fff(-3);
        ggg(0xABCDEF0011223344);
}
```

```
bigarg:
        movq    $-3, %rdi
        call    fff
        movabsq $-6066930339431697596, %rdi
        call    ggg

0000000000000000 <bigarg>:
   0:   48 c7 c7 fd ff ff ff    mov    $0xfffffffffffffffd,%rdi
   7:   e8 00 00 00 00          callq  10 <bigarg+0x10>
  0c:   48 bf 44 33 22 11 00    movabs $0xabcdef0011223344,%rdi
  13:   ef cd ab
  16:   e8 00 00 00 00          callq  1f <bigarg+0x1f>
```

In the call of fff, the argument -3 fits within a 32-bit immediate value, so the C7C7 opcode (move 32 bits immediate->register) is used, with the "REX" prefix byte of 48, which selects the "R" version of the DI register and causes sign extension of the 32-bit immediate source operand. But in the call of ggg, that value doesn’t fit in 32 bits. We need to go to the movabsq opcode 48BF, which moves a 64 bit immediate value into a register. (The GNU X86-64 assembler automatically chooses a movabs opcode if the supplied immediate value doesn’t fit into 32 bits)

X86-64 Global Variables

Unfortunately the X86-64 architecture makes it awkward to use 64-bit absolute memory addresses. There are very few instructions where a memory operand can be specified directly with a 64-bit absolute address. This results in two different approaches or "models." First, the more common "small memory model" approach:

```
//bigglob.c
extern int i;

f()
{
         i=2;
}

f:
        pushq   %rbp                            #Prologue, save base pointer
        movq    %rsp, %rbp            #Set new base pointer
        movl    $2, i(%rip)           #Program Counter Relative mode
        leave
        ret
```

The operand `i(%rip)` is using the RIP-relative addressing mode. This allows a 32 bit (not 64) displacement relative to the %rip register. There is no corresponding EIP-relative mode in X86-32, however. There will be a 32-bit "hole" in the `movl` opcode and the relocation type will be R_X86_64_PC32 which tells us that this relocation is for a 64-bit symbol (i) but relative to the program counter (%rip) value at that point, and limited to 32 bits. This relocation is similar to the R_386_PC32 relocation which we will see later in this unit.

The small memory model thus has the limitation that statically-linked code and data must fall within the same contiguous +/-2GB memory region at run time (this limitation does not apply to shared libraries or dynamically allocated program memory).

The small model is the default, as it is quite rare for the static text, data, and bss objects to fail to fit within 4GB! But, to use a "large" memory model where code and data may be anywhere within the 64-bit address space, invoke gcc with the -mcmodel=large option. Different opcodes are now used:

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

Now GCC uses the movabsq opcode to move a 64-bit immediate value into the 64-bit register %rax (other registers might be chosen depending on the register allocation situation at that point in the code), and then uses the simple register indirect addressing mode to reference the memory operand. The symbol $i is 64-bit, and the relocation type is R_X86_64_64.

As an aside, the X86-64 architecture can do movabsq absmem,%rax where absmem is a 64-bit absolute address, but only the rax register is supported as the destination!

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 Usage 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
			%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:

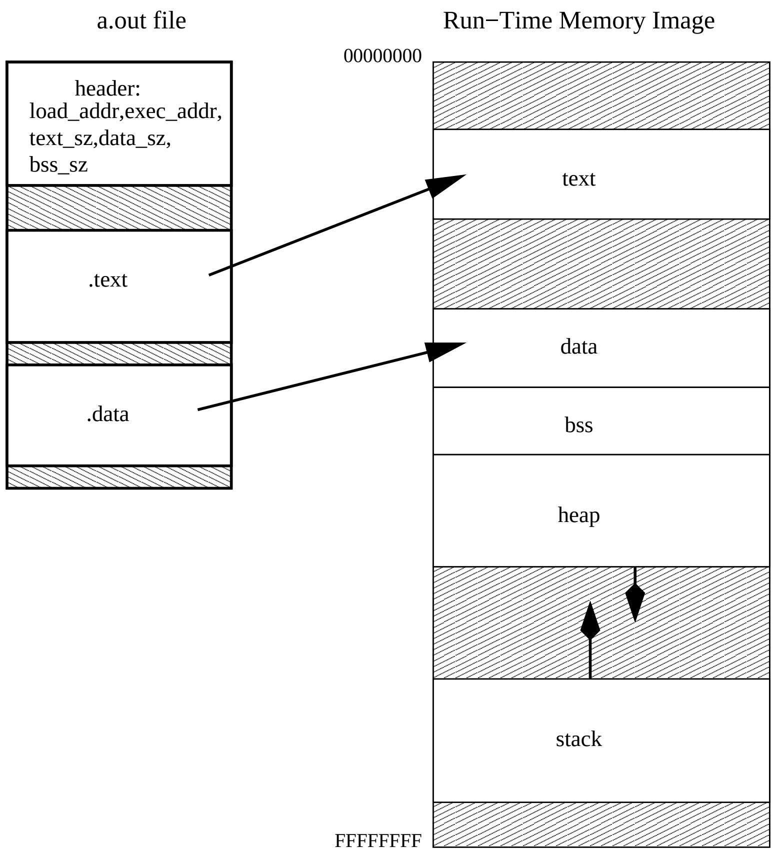
	X86-32		X86-64		
type	size	align	size	align	
char	1	1	1	1	
short	2	2	2	2	
int	4	4	4	4	
long	4	4	8	8	
long long	8	4	8	8	
pointer	4	4	8	8	
float	4	4	4	4	
double	8	4	8	8	
long double	12	4	16	16	(see note)

Aside: The long double type in C corresponds to the double-extended precision floating point type in X86. In both 32 and 64 bit modes, this is actually an 80-bit format. The compiler rounds this up to the nearest 4 bytes (32 bit mode) or 8 bytes (64 bit mode) when computing sizeof

This concludes, for now, our X86 tour. We'll next consider how the linker works.

The UNIX run-time environment

The information below is summarized from Operating Systems. In UNIX systems, addresses are virtual and each running program has its own virtual address space. There are four basic regions of memory: text, data, bss, stack, each with its own characteristics. Additional regions can be created dynamically by the program. These are used for dynamically-allocated variables and to facilitate dynamically loaded libraries ("shared" libraries). The header of the `a.out` file gives the size of the text and data regions and the required size of the static portion of the bss region.



The **bss** region 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. The **data** region contains all initialized global variables. The term "global" here refers to storage scope, not lexical scope.

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

In the example above, `j` has an explicit initializer, and will be in the data region. 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 region 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. refers to a function vs a variable), 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 (but these are replaced by other symbols)

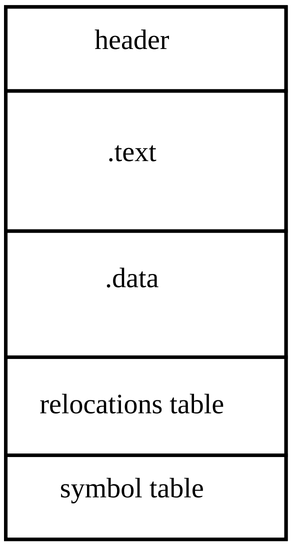
• Internal labels that may be generated by the compiler for loops, switch statements, if statements, etc. These labels will appear in the assembly language, but they are local symbols and will not be visible to the linker, similar to goto labels.
• automatic (local) variables. An important exception 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

There have been many different formats used for relocatable object files and executable files on UNIX systems over the years. On Linux systems today, the ELF (Extensible Linker Format) is the de-facto standard. We can use the `readelf` tool to explore an ELF file. This, combined with `objdump`, solves most linker questions.

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.

Operation of the Linker

The first task of `ld` is to take inventory of all of the `.o` files being presented to it (including additional .o files that are contained in static libraries.) ld 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 namespace for all global 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. (However, see the "common block" exception described later for uninitialized variables) 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?

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 **linker symbol 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. Look into "weak symbols" for more information, and 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 put together 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`. A similar procedure is performed for all the .data sections. ld is guided by a *load map*, specific to the target architecture environment, which defines where each section would be loaded at run time. For example, on X86-32 Linux, the text region of memory will be at 0x08048000. We can therefore say that we have "relocated" each section of each object file to a specific absolute address. Once this is done, absolute values are assigned to each symbol. For example, if a symbol `xyz` has an offset 0x0E within the .text section of the first object file, and that text section will be loaded at 0x08048000, then it will now have a value of 0x0804800E.

`ld` makes sure to record within the header of the a.out the information about where the text, data, and bss regions are intended to be loaded, along with their sizes. This allows the kernel to load the executable at the proper addresses.

The last step of `ld` is to "patch" all the relocation records, replacing the "holes" representing symbolic memory addresses with their absolute values. So the a.out file consists of pure executable code, and all the kernel needs to do is mmap it into memory and jump to the address in the text section of the first opcode.

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                 #for stack alignment
        movl $1,i
        call f
        leave                        #Restore frame pointer
        ret
        .size    f2,.-f2
```

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 mark the associated symbol as globally visible to all other object files in the linkage. `.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 Memory 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 0x11
0015   C9                          leave
0016   C3                          ret
```

Note that the placeholder in the CALL instruction is FFFFFFFC, but the disassembler decodes that as 0x0011. 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*, 0x0015. In two's complement, FFFFFFFC is -4, therefore the CALL appears to be to the instruction at offset 0011. 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 unresolved 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. We see now why the assembler inserted a -4 (FFFFFFFC) into the hole. The linker doesn't actually understand assembly/machine language. It is strictly a byte-for-byte program. When it patches a R_386_PC32 hole, it doesn't know that the CALL opcode will add the operand to the value of the program counter beyond the entire 5-byte CALL instruction. As far as the linker is concerned, it is patching a hole at offset 0x11 from the start of f2.o's .text section. But by calculating the difference between this location and the branch target for the PCR mode, the linker would come up with a value which is too high by 4, because it is working from offset 0x11 (the first byte of the hole) whereas at run time the CPU will be working relative to offset 0x15. To fix this, the assembler inserted a -4 into the hole for the PCR relocation mode.

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

The entrypoint address, where execution will begin, was not specified explicitly (we'd use the -e option to ld for that) so it defaults to the first opcode of the .text section, which is our function f. On Linux X86-32 ABI, the text region of memory starts at 0x08048000. However, symbol f is at 0x08048094. The linker puts the text section of the a.out file right after the header. But mmap has to be on a page (4096 byte) boundary. So if we were to examine memory at run time just above the first text opcode, we'd see the ELF header!

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:

          .globl i
        .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
```

Again, `.globl` directive instructs the linker to make the symbol i globally visible during the linkage process. By default, symbols are local to this object file. 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;             // Hmmmmmm
}
```

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=10` will cause a run-time exception.

```
         .data                                 #enter .data section
         .type    ary,@object
         .size    ary,7
ary:
         .string  "ABCDEF"           #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
```

The `.rodata` section may also be used for global variables that are declared with the `const` qualifier.

Assembly Language Directives

The following is a quick recap of the assembly language directives (pseudo-opcodes) we’ve seen with GNU as. It is not intended as a complete reference and as always, please consult the authoritative documentation for more information.

Directive	Meaning
.file *"filename"*	Debugging comment, indicates source origin
.file *lineno "filename"*	Debugging comment, indicates source origin
.text	Switch output to .text section of object file
.data	Switch output to .data section of object file
.section *name*	Switch to named section
.size *sym, size*	Define the storage size allocated to symbol sym
.globl *sym*	Make sym globally visible
.local *sym*	Make sym not global (local)
.comm *sym,size,align*	Declare a common block object (BSS variable)
.org *loc*	Move the cursor (.) to the specified value loc
.align *n*	Adjust the cursor (.) upwards to nearest n-byte boundary
.type *sym, @type*	Define the type (@object or @function) of symbol sym
.byte *n*	Emit a single byte n into the output
.word *n*	Emit a two-byte integer n into the output
.long *n*	Emit a four-byte integer n into the output
.quad *n*	Emit an eight-byte integer n into the output
.fill *repeat, size, n*	Emit repeat copies of value n which is size bytes long
.zero *count*	Emit 0 byte * count times
.string *"string*"	Emit a sequence of characters including NUL terminator

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, Overview

In "modern" operating systems (since ca. 2000), dynamic linking is used almost exclusively for calling the system libraries. When dynamic linking is in use, there are two distinct groups of regions in the program's address space at run time:

The **static part** consists of the traditional text, data, and bss regions, as already discussed. They reside at "static" addresses, meaning that whenever the program is executed, a given text, data, or bss symbol in the static part always has the same binary address.

The **shared objects** are attached to the address space by the dynamic linker, just before the program starts executing. Each dynamic object can have text, data, and bss symbols, but their addresses are not known when the program is linked into an a.out. Rather, each of these dynamic objects (or "shared libraries") is like a mini a.out that gets loaded at an unpredictable address. Even though we know which shared libraries will need to be loaded to make the program run, we can't know at link time what their addresses are, in part because we don't know exactly which version of each library we are going to see at run-time.

In dynamically linked programs, the stack region behaves the same way. There is just one stack region (or, in a multi-threaded program, one stack region for each thread)

Dynamic Linking, Link-Time behavior

Shared libraries are built from individual .c and .o files, just like static libraries, but they need to be built very differently. When using gcc, the flag `-fpic` must be used when building each `.o` file from source. This instructs the compiler to use Position Independent Code (PIC) techniques, eschewing absolute memory address modes (see further discussion below). The set of .o files can be put into a shared library using gcc, e.g. `gcc -shared -o libXYZ.so obj1.o obj2.o etc.o` The `.so` suffix (shared object) is used to denote the shared library.

A program which wants to use a shared library simply links against it, the same way it would link against a traditional library with a `.a` suffix. However, the linker applies different rules to link the static and shared parts together and create an `a.out` file that will work correctly at run time. Such an executable will be marked as "dynamically linked" and this causes very different behavior when it is exec'd. A dynamic executable carries with it additional sections known as the Dynamic Symbol Table and the Dynamic Relocations Table. These will guide the run-time resolution of dynamic symbols by the dynamic loader:

The dynamic loader ld.so

On Linux systems using the ELF format for object files, libraries, and executables, dynamically linked executables are executed by means of a "helper program" known as `ld.so`. The actual pathname will be system specific, e.g. `/lib64/ld-linux-x86-64.so.2` The ELF header of the a.out mentions this ld.so executable as the "binary interpreter." This is a feature which the Linux kernel provides wherein it will automatically load and execute the helper program to complete the dynamic linking. The following steps are taken by the kernel, and then by ld.so:

1) The a.out file is located by the kernel and its sections (text, data, bss) are "loaded" (mmap'd) according to the section map in the ELF header of the a.out. The entrypoint (program counter) address in the text region is noted.

2) The requested binary interpreter (ld.so) is located (if not found, it is a fatal error and execve returns an error code) and its text, data, bss regions are mapped to dynamically-assigned addresses, typically at the high end of memory.

3) A stack region is created by the kernel. The argument vector, environment, and another data structure known as the "auxiliary vector" are created on the stack. The latter is for the guidance of ld.so.

4) The kernel starts execution at the entrypoint of ld.so (NOT the _start entrypoint of the program itself)

5) ld.so has full access to the a.out's ELF headers since the file is mapped into memory. ld.so determines which shared libraries are required, finds each one, and mmaps each into memory at addresses which are determined by the kernel (and not fixed). Each shared object has text, possibly initialized data (these are mapped from the .so file), and additionally any private bss areas needed by the library are created by ld.so. Unlike the mapping of the static a.out and ld.so itself, which is done by the kernel, all of step 5 is done in user mode, by ld.so

One can use the environment variable LD_DEBUG=all to debug ld.so's searching for and loading of dynamic objects. Alternatively, one can see this during a `strace` of the program. ld.so searches for the needed shared libraries according to a specific sequence:

i) The RPATH parameter of the a.out file contains a list of pathnames to search for libraries, similar to $PATH. This is set with the ld flag -rpath=paths and to pass that flag to ld via gcc, use e.g. `gcc {..other options...} -Wl,-rpath=/usr/local/mylibdir`

ii) The environment variable LD_LIBRARY_PATH contains a similar list of pathnames to search. For security reasons, this variable is ignored for setuid executables.

iii) The RUNPATH parameter of the a.out is similar to RPATH, but is consulted after LD_LIBRARY_PATH. Refer to the gcc documentation for more details.

iv) Locations of recently resolved shared libraries are cached in `/etc/ld.so.cache`

v) A list of default, trusted places to look for shared libaries is hard-coded into the ld.so binary (e.g. `/lib64`, `/usr/lib64`, `/lib/x86_64-linux-gnu`)

6) ld.so looks at which symbols each shared object "needs," and which it defines. It also examines the static symbol table for symbol definitions. At the conclusion of shared object loading, we now know the actual run-time address of all symbols.

7) ld.so now performs the **dynamic relocations**, patching the actual address of those symbols requring this into the appropriate GOT slots (see below). On many modern "hardened" Linux systems, after this is done, the GOT section is turned readonly by ld.so using the mprotect system call. This is for cybersecurity reasons, to reduce the risk exposure of a GOT slot being overwritten due to a security exploit.

8) Finally, ld.so jumps to the actual entrypoint of the program. If the program is a normal C program, this entrypoint is the `_start` function located in the static .text portion of the libcrt static library (which is linked with any C program). _start in turn winds up calling `__libc_start_main` in the C library, which does the initialization of the standard library, and *finally* (really, this time), calls `main`, as described in the ECE357 material.

The PLT and GOT

Two critical data structures are employed to allow the dynamic parts of the program to access the static parts and vice versa. These are the Procedure Linkage Table (PLT) and the Global Offset Table (GOT). Our discussion will focus on the Linux/X86 implementation only.

Let’s say a function in the static text wants to call a function which will ultimately come from a dynamic library. The static part of the program must be able to function unaware of the dynamic nature. Therefore we need to use a regular CALL opcode which specifies the actual address of the function being called (although expressed relative to the eip/rip register). However, at static linkage time, we don't actually know this address.

One approach could be that we leave dynamic relocation "holes" in the text, and require that ld.so dynamically patch every place in the text where we call, e.g., `printf`. This approach is not desirable for several reasons. First, it would require a lot of work and slow down the initial execution of the program. It would also induce page faults all over the text region. Lastly, we'd like to leave the text region as readonly at all times.

Therefore, at static link time, each call/jump reference to a symbol which is being supplied from a dynamic library causes the linker to create a slot in the PLT for that symbol. The PLT slot contains a jump to the actual function (e.g. printf). While it would seem logical that ld.so should just patch each PLT slot with the appropriate dynamically-loaded function address, the situation is actually more complicated on X86/Linux. The reasons are far too complicated to get into during this introduction.

So a typical PLT slot generated by the linker whenever printf is needed by the program would look like this:

```
printf@plt:
          jmpq      *printf@GLIBC(%rip)
```

Note the use of symbols which are local to a specific section using the @ syntax. This PLT slot uses an indirect jump. The processor fetches the contents of the memory address which is symbolically printf@GLIBC. This is an example of a Global Offset Table (GOT) slot. Each GOT slot is the size of a pointer (8 bytes in the X86-64 architecture) and contains the actual run-time address of the corresponding symbol. `printf@GLIBC` is symbolically the address of the GOT slot for the printf symbol in the GLIBC library.

The PLT method must also be employed when a shared object wishes to call a function defined in the static portion of the program, since the compiler and linker have no idea where that function will be when the shared object itself is being compiled and linked.

Global variables defined in shared objects

If a shared library has global variables which it wants to be visible to the static part of the program, this introduces some new problems. We can't have these variables live in the data or bss regions associated with the shared library itself, because the static text needs to access them unaware of the fact that their address will be dynamic. Therefore, these variables will be allocated a place in the bss region (even if they are initialized globals) associated with the static program. The static text accesses them like any global variable, using a memory absolute addressing mode (X86-32 and X64-64 large model) or RIP-relative mode (X64-64 small model).

But how does the shared library access its own global variables? It must use the GOT.

Each such variable (symbol) is given a GOT slot by the linker at static link time. Then ld.so patches the run-time address of that variable into the GOT. The shared library uses a GOT-relative addressing:

```
/* This is how we’d access extern int i; from a shared object */
/* This example is i=10; */
movq      i@GOTPCREL(%rip), %rax
movq      $10,(%rax)
```

This code references the GOT slot of variable i as a RIP-relative address. It loads the actual address of i from that GOT slot, places it in rax, and then uses the register indirect mode to finally access variable i.

Initialized globals in a shared library

What if the global variable exported by a shared library has an initializer? Again, this variable must actually live in static portion of the program. However, it can’t live in the static .data section, because the initializer value is really part of the shared library. We can’t pretend to know that at static link time. It can’t live in the .data section of the shared object, because the static code needs to be able to access it without knowledge of its dynamic nature.

Therefore, the variable is actually located in the bss region of the static portion of the program, while its initializer is in the data section of the shared library. The associated symbol is marked as a "COPY" type of dynamic relocation. ld.so sees this and, in addition to patching the variable’s address into the GOT, loads the initializer value from the library’s data region and copies it to the actual location of the variable. Whew!

Shared Library Global variables contained therein

Variables declared within a shared library with global storage scope but local linkage scope are meant for the library’s own use, and do not need to be accessed by the static part (or by other libraries). These will be located within the data or bss section of the shared library. However, since their absolute address will not be known when the shared library is linked into the `.so` file, the compiler must access these using PIC:

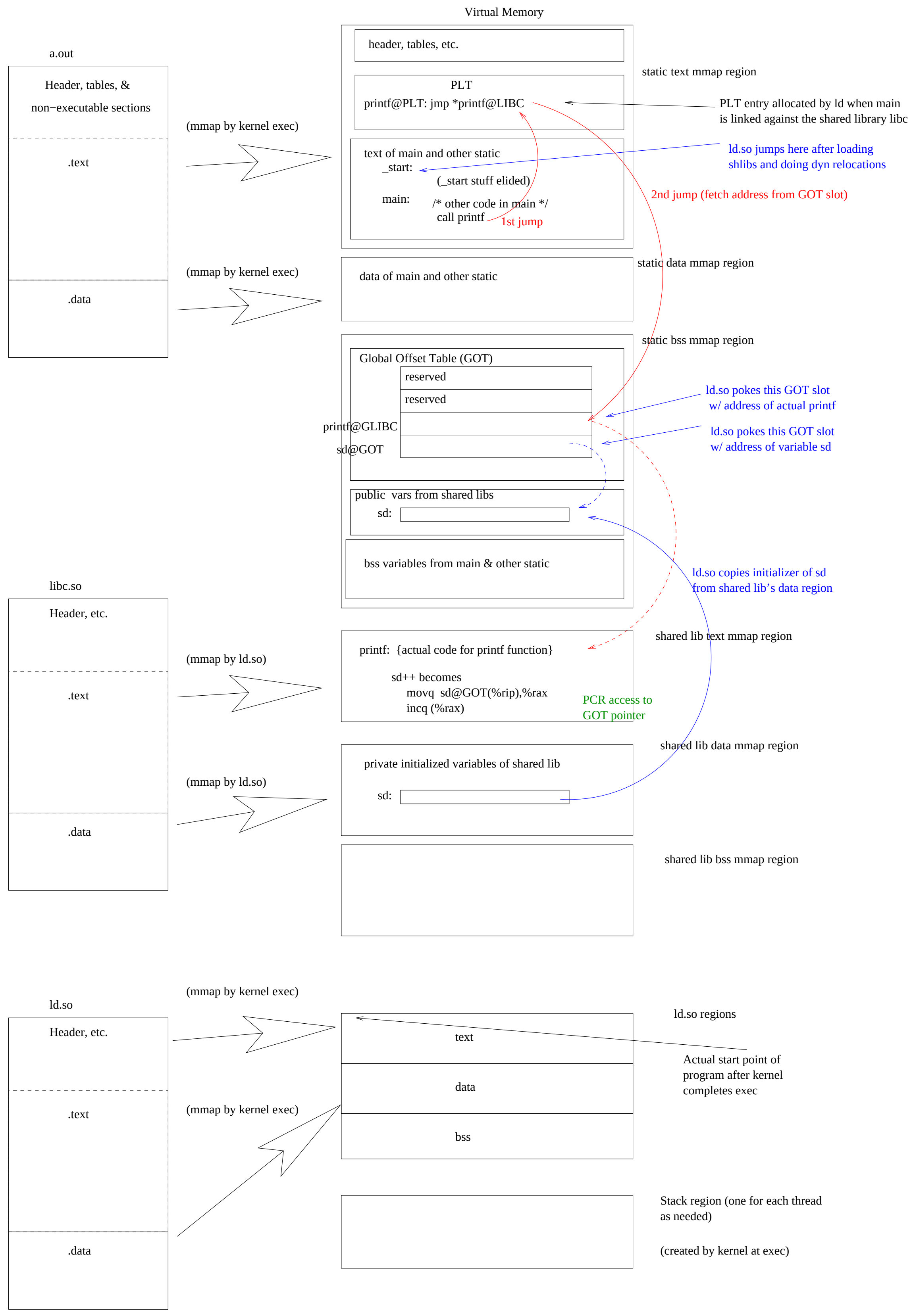
```
/* This is code within a shared library, compiled with -fpic */.
static int ss;

f()
{
          ss=10;
}

movl      $10, ss(%rip)
```

The address of the `ss` is represented relative to the %rip at that point in the code. This relocation will be taken care of when the `.so` file is linked, because at that time we know the relative address between the opcode in the shared object's text, and the variable in the shared object's data or bss region. The symbol `ss` will NOT have a GOT slot and will not be subject to dynamic relocation by ld.so.

Note that this RIP-relative mode looks just the same as when static code needs to access variables in X86-64 small memory model. The student is invited to explore what happens in X86-32 or X86-64 large model!


Virtual Memory
a.out
Header, tables, &
non−executable sections
.text
.data
(mmap by kernel exec)
(mmap by kernel exec)
header, tables, etc.
static text mmap region
PLT
printf@PLT: jmp *printf@LIBC
PLT entry allocated by ld when main
is linked against the shared library libc
text of main and other static
_start:
(_start stuff elided)
main:
/* other code in main */
call printf
1st jump
ld.so jumps here after loading
shlibs and doing dyn relocations
2nd jump (fetch address from GOT slot)
data of main and other static
static data mmap region
static bss mmap region
Global Offset Table (GOT)
reserved
reserved
printf@GLIBC
sd@GOT
ld.so pokes this GOT slot
w/ address of actual printf
ld.so pokes this GOT slot
w/ address of variable sd
public vars from shared libs
sd:
bss variables from main & other static
ld.so copies initializer of sd
from shared lib's data region
libc.so
Header, etc.
.text
.data
(mmap by ld.so)
(mmap by ld.so)
printf: {actual code for printf function}
sd++ becomes
movq sd@GOT(%rip),%rax
incq (%rax)
PCR access to
GOT pointer
shared lib text mmap region
shared lib data mmap region
private initialized variables of shared lib
sd:
shared lib bss mmap region
ld.so
Header, etc.
.text
.data
(mmap by kernel exec)
(mmap by kernel exec)
text
data
bss
ld.so regions
Actual start point of
program after kernel
completes exec
Stack region (one for each thread
as needed)
(created by kernel at exec)

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 {   // Assume ints are 4@4
         int a;             // offset 0
         char b;            // offset 4
         int c;             // offset 8
         char d;            // offset 12
};                 // sizeof(struct example)==16
```

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\n ",#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);
}
```

Starting in C99, the C language provides this functionality through the `__alignof` unary operator. It functions similarly to the `sizeof` operator in that it can take either an actual expression, or an abstract type name. An interesting case arises:

```
x=__alignof(char [i++]);
```

Here we have an abstract type name which is a variably modified array type. With the `sizeof` operator, we saw in Unit 4 that the expression `i++` actually gets evaluated at run-time. But with `__alignof` the alignment of an array type does not depend on the number of elements in the array (it is just the alignment of the base type). Therefore the size control expression *does not* get evaluated here!

More caution is advised with this operator. In C11, it changed from `__alignof` with a double underscore (GCC allows the variant `__alignof__`) to `_Alignof` (single underscore). Furthermore, both GCC and Clang give inconsistent results between the two versions of this operator when applied to some architectures, including X86-32!

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.

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.