J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

PROGRAM ASSEMBLERS

Although most computer programs are now written in more abstract, higher-level, programming languages, it is virtually impossible to build a computer system without also having a human-readable low-level language for specification of individual machine instructions and the layout of objects in memory. The languages that allow this type of detailed specification are known as *assembly languages,* and the software that transforms an assembly language program into the corresponding raw bit patterns that the hardware can operate on is called an *assembler.*

There are many different assembly languages and assemblers. Each type of computer hardware has its own instruction set and memory access constraints, so each type of computer defines its own assembly language. For example, the assembly language used to program a Pentium is significantly different from that used to program a *SPARC*; they describe different types of instructions, registers, and the like. Despite these differences, nearly all assembly languages and assemblers have roughly the same basic structure because each serves the same purposes for the particular system it targets.

What is Assembly Language Used For?

Although assembly languages were once the most commonly used programming languages, this is no longer the case. The primary role of assembly languages is now to serve as the targets for sophisticated high-level language (*HLL*) compilers. Given these high-quality optimizing compilers, very few programmers will miss the experience of writing large amounts of assembly language code by hand. Instead, mixing a little bit of hand-written assembly code with a lot of assembly code automatically generated by an HLL compiler can yield tremendous benefits.

High-Level Languages. Like Fortran and C, HLLs offer programmers many advantages over assembly language. Perhaps the most significant of these is that an HLL programmer does not really need to know details about how the computer hardware will execute a program. A C programmer can reason about program behavior in terms of an imaginary virtual machine that directly executes C programs, with little or no specific knowledge of the actual computer hardware used. Issues like the choices of which machine instructions to use, which register or memory cells should hold each value, and how to optimize the code for that machine are all removed from the programmer's concern.

The fact that HLL programs are typically more abstract than assembly language code implies that HLL programs are generally easier to write, maintain, and port to other computer systems. That higher-level abstractions make code easier to write is obvious. For example, very few computers directly support arithmetic operations on complex numbers, but Fortran allows programmers to write, and think, in terms of operations on this type of data; similarly, a C programmer can directly express concepts like recursive algorithms operating on graphlike linked data structures in which each node contains a variety of different types of information about that node (i.e., each node is a C struct). These operations could be coded directly in assembly language, but the programs would be significantly longer, and the relationship between the program code and these higher-level

concepts would become less clear. This obfuscation of why specific operations were performed makes assembly language programs more difficult to maintain; bugs are more likely, and corrections or changes tend to be engineered by trial and error rather than smoothly propagated down from a specification of the new functionality. For the same reasons, porting HLL programs is easier, or at least more efficient, than porting assembly language programs; more precisely, porting involves simulating the architecture of one machine with another, and simulation overhead is higher for the lower-level, more intricate operations typical of assembly language.

Why Use Assembly Language? The preceding arguments against programming in assembly language do not make assembly language less important; assembly language has actually become more important. In order for an HLL compiler to generate machine code with efficiency comparable to that of carefully handwritten assembly language code, the compiler must use sophisticated analysis and optimization techniques. The result is that good compilers are usually remarkably complex programs. Although it would be possible for a compiler to generate machine code directly (this has been done in some compilers to improve compiler speed) instead of generating assembly language code, this would make the compiler still more complex and would further complicate the task of retargeting the compiler to generate code for another machine. Generating machine code also would make it far more difficult to examine the compiler's output for the purpose of debugging or improving the compiler. Without an assembly language and assembler, it even would be difficult to create the basic libraries and hardware device interfaces needed as support for compiled programs. Thus, as compilers have become more complex, assemblers have become more important as their targets.

That said, if you are not involved in building a compiler, why should you care about assembly language? The answer is that even though the bulk of your programming should favor HLLs over assembly language coding, there remain some things that HLL compilers either cannot do well or simply cannot do. By dealing directly with just a small amount of assembly language code, you can repair these shortcomings.

Some processors have instructions or addressing modes that are too complex, strange, or specialized for compilers to use efficiently. For example, the Texas Instruments TMS320 series of *DSP*s (digital signal processors) have instruction sets that are designed to make specific signal processing algorithms fast. One of the most important of these algorithms is the fast fourier transform (*FFT*), and one of the slowest portions of the FFT algorithm involves generating the addresses for the "butterfly" reference pattern. Given an FFT involving a power-of-two number of data points that are located in memory beginning at an address that is a multiple of that power of two, the TMS320C30 can directly generate these addresses by incrementing an auxiliary register using "indirect addressing with post-index add and bit-reversed modify" in which index register IR0 specifies a value that is half the number of points in the FFT (1). This addressing mode saves at least a couple of clock cycles for each address computation in FFT or related algorithms, and assembly language programmers can easily use it (once they have been briefed and shown a code example), but it is unlikely that, for example, a C compiler would ever be smart enough to recognize when it can profitably use this addressing mode. This should not prevent you from using a C compiler to generate most of your code for the TMS320C30; you could use assembly language just for the FFT routine, or you even could use assembly language just to paste in the few instructions and addressing modes that the C compiler would not have been clever enough to use.

Most modern processors also provide a range of hardware-dependent operations that portable HLLs cannot express, but that easily can be accessed with no more than a few assembly language instructions. These operations range from various types of privileged instructions that only should be used in systems software to operations that interact with portions of the hardware that are specific to your particular system. Examples of privileged instructions include interrupt handling and manipulation of protection or memory mapping hardware. Machine-specific operations include accesses to I/O (Input/Output) devices, performance monitoring registers with the processor, and even system configuration information like cache size or processor version number.

Finally, and hopefully least frequently, it also is useful to be able to modify the assembly code generated by an HLL compiler either to take advantage of optimizations that the compiler missed or to work around compiler bugs. Although compilers are much better than humans at consistently applying the optimizations

that they understand, the compiler can apply an optimization only if it can confirm that this optimization is safe, and overly conservative assumptions about worst-case behavior often make compilers fail to apply even some of the most straightforward optimizations. Humans are much better at recognizing that an optimization applies in a particular case. For example, many HLL compilers will disable a variety of optimizations for code that contains a store through a pointer variable (because few HLL compilers can accurately track pointer aliasing), but it might be trivially obvious to the programmer that the pointer does not really prevent any of the usual optimizations from being applied. Smart compilers are also notorious for making assumptions that sometimes result in "optimizing" code into a less-efficient form; a good example is that many loops typically execute their body zero times, but smart compilers often will generate code that moves computations out of the loop body based on the assumption that the loop body will be executed more times than code outside of the loop. There also may be minor flaws in how the compiler pieces together fragments of code. For example, the SPARC (2) does not have an integer divide instruction, so HLL compilers generate assembly code that calls a general-purpose subroutine for this operation; there may be no way other than assembly language coding for a programmer to force a particular integer divide to be implemented without a subroutine call.

In summary, use HLLs as much as you can and assembly language as little as possible. Assembly language programming is like using a sharp knife; a sharp knife is a very efficient tool, but not every job needs a knife and misusing a sharp knife can be very painful.

An Overview of Assembly

Most people have a vague notion that assembly language lurks somewhere in the ominous darkness beneath their HLL and compiler, but assembly languages are generally very clean and simple languages. Assembly languages try to provide direct access to all the features of the computer system and to do this in a way that is intuitive and predictable for someone who understands the computer's architecture.

To better understand the role of assemblers, it is useful to review how they are used within a process that transforms an HLL program into an executable machine code image in memory. The basic syntax of assembly languages is then briefly overviewed. A brief discussion of the impact of the RISC versus CISC controversy (3) on assembly language completes our overview of assembly.

Where Does the Assembler Fit In? The complete process of converting an HLL program to machine code being executed by a processor is complex enough to warrant a diagram. This process is depicted in Fig. 1.

In a typical programming system, most users write code in an HLL language. This HLL code is then compiled into assembly language. Much as a compiler processes the HLL code, the assembler converts the assembly language program into a lower-level form. In some cases, the output of the assembler may be little more than the raw binary instructions (machine code) and data that can be loaded into the system as a memory image and then executed by the hardware. Examples of such forms are the hexadecimal-ASCII encoded S records and Intel hex formats. These forms are commonly used when the program is being developed on one computer system but will be executed on a different, much simpler, computer—often a dedicated microcontroller such as the Motorola MC68HC11 (4). An assembler that generates code for a machine other than the one that the assembler runs on is called a cross assembler.

When the assembled code is intended for execution on a more complete computer system rather than a microcontroller, the assembler's output is usually a complex mix of binary instructions, data, and symbolic information about the names that the programmer used for functions and data. Generically, this type of output is referred to as an object module, with *COFF* (Common Object File Format) and the *ELF* (Executable and Linking Format) variant of COFF among the most popular file formats. Although the symbolic information in the object module can be used to aid in debugging, it also allows multiple pieces of a program to be assembled separately, in which case machine code within each module is not pure but still contains some symbolic references to names that are defined in other modules.

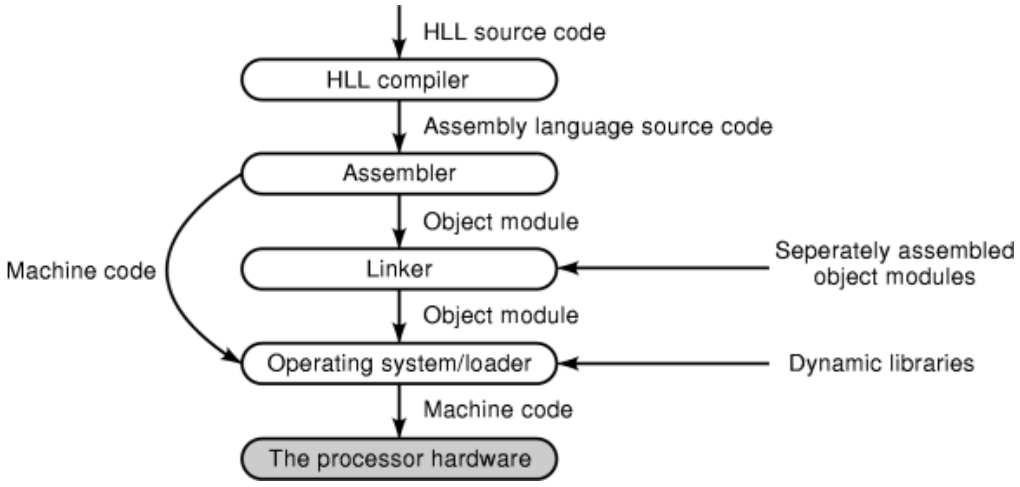

Fig. 1. Conversion of an HLL program to machine code.

A linker is a program that links references to symbolic names in one module to their definitions in another module. In some sense, this could be done just as well by passing all the assembly code through the assembler together so that only a single module is generated. Using a separate linker saves the overhead of reparsing the full library's assembly language source code every time a program that uses a routine from that library is assembled. It also allows libraries to be distributed in the form of modules rather than source code, which allows the library's source code to be kept somewhat more private. The output of the linker is again an object module.

When the object module is to be executed, a portion of the operating system called a loader is responsible for reading the contents of the object module and creating an appropriate executable image in memory. In most cases, the object module's machine code is complete, and all symbolic references have been resolved, so the loader's job is quite simple. However, in some cases there may still be unresolved symbolic references in the object module, in which case the operating system is required to link to other modules dynamically either at the time the program is loaded or when an unresolved symbolic reference is encountered as the program is running. Shared libraries and DLLs are both mechanisms for dynamic linking.

It is interesting to note that the symbolic information stored in an object module can be surprisingly complete, and the move toward formats like ELF is largely motivated by the desire to incorporate even more symbolic information. For example, ELF files can even maintain tree-structured symbolic information about C++ classes. A debugger can use this symbolic information to help disassemble, or reverse assemble, the machine code, at least generating assembly language mnemonics for instructions and perhaps going as far as determining the HLL source code construct that each instruction originally came from.

Thus, the role of the assembler is essentially to convert the instructions and data into their raw binary representations, usually also converting symbolic references into a standard format that can be processed by linkers, loaders, and debuggers.

Assembly Language Syntax. Although each different type of processor has its own assembly language, and some processors are even supported by several different assembly languages, nearly all assembly languages have the same general format. This very simple format is line-oriented, with each statement taking one line. There are only a few different types of statements; some specify instructions, others specify data, and still others are pseudo-operations that serve to control the assembly process.

Specifying an Instruction. When one thinks of specifying a program so that a computer can execute it, it is natural to focus immediately on how individual machine instructions are specified. Although this is only part of the problem, and different machines generally differ most dramatically in the sets of instructions that they support, the same basic syntax is used in nearly all assemblers.

For most assembly languages, each machine instruction in a program is specified by a separate line of assembly language code. Each different type of machine instruction is given a name—a neumonic—that can be used to identify the instruction. To specify a particular instruction, the corresponding neumonic, possibly followed by a comma-separated list of operands to that instruction, is given on a single line. For example, an IA32 instruction to add the constant value 601 to register %eax is

```
addl    $601, %eax
```

Specifying Data. Space for data values that are to be stored in fixed memory locations, as opposed to data dynamically allocated at runtime or allocated on the runtime stack, can be specified in a number of ways. For initialized variables, most assembly languages offer a variety of pseudo-operations that to encode values of the various types. There is also a way to reserve space without initializing it. Consider the C data declarations:

```
char a = 1;
short b = 3;
int c = 5;
```

For the IA32 Gnu ASsembler (*GAS*), this would be coded in assembly language like:

```
a:     # call this address a
.byte 1    # an 8-bit char initialized to 1
.align 2    # force address to be a multiple of 2
b:     # call this address b
.size b,2
.value 3    # a 16-bit short initialized to 3
.align 4    # force address to be a multiple of 4
c:     # call this address c
.size c,4
.value 5    # a 32-bit integer initialized to 5
.comm  datum,400,4 # reserve 400 uninitialized bytes
# aligned on a multiple of 4 address
# and call that address datum
```

In this example, the portions of each line after the # character are comments. Most of the assembly code is obvious; however, the .align pseudo-operations may not be.

Alignment refers to the fact that, because data paths to memory are more than one bit wide, accessing a value that requires more than one data path width transmission is either unsupported by the hardware or slow (because multiple bus transactions are used). Most current processors use 32-bit wide data paths but allow memory to be indexed by addresses of 8-bit positions. In general, a data value is properly aligned if the byte-address is a multiple of the data value's size in bytes. Figure 2 clarifies these constraints.

Set and Equate. Most assembly languages actually provide at least two types of assembly-time symbols that differ primarily in their scoping rules. The best analogy in a conventional programming language is that symbols used one way behave like variables, whereas symbols used the other way are eternal constants. Consider this example in 8080 assembly language notation in which comments begin with ;:

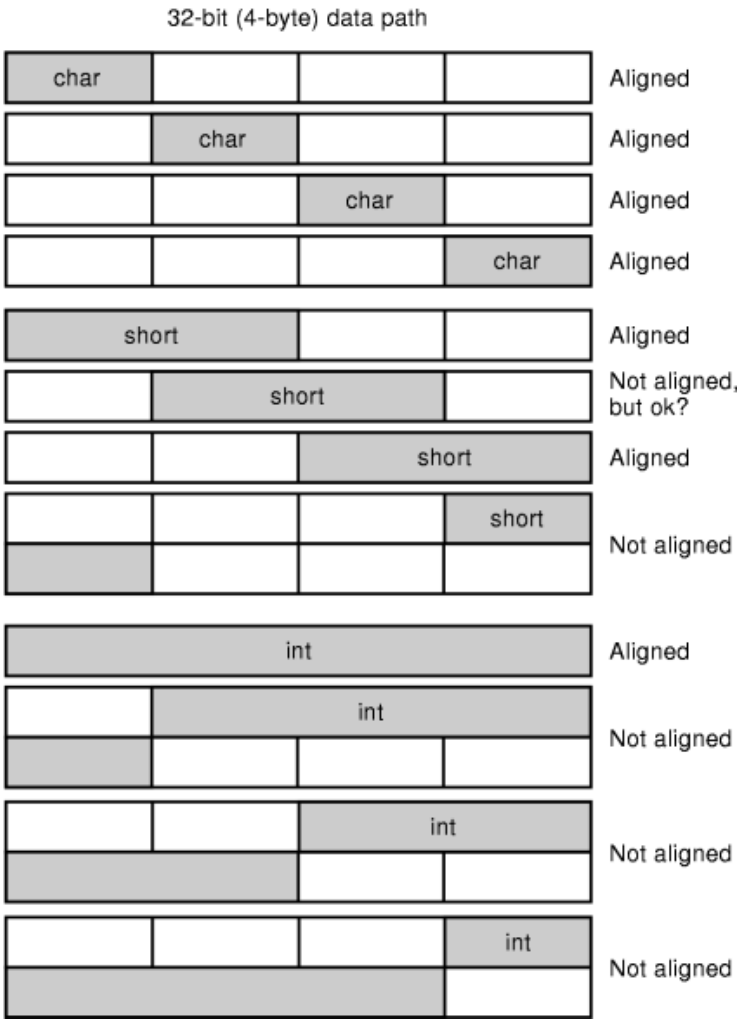

Fig. 2. Alignment constraints.

```
that  set  this+2
; here, this is 5 and that is 7
this  equ  5
that  set  this+1
; here, this is 5 and that is 6
a:
b    equ
; a and b both get the value of the location counter
```

The set directive can be viewed as establishing a temporary equivalence, much like assigning to a variable. A symbol set to a value holds that value only up to the point it is set to another value. In contrast, equ (EQUate) establishes a value for the symbol that will exist as a constant throughout the program, carried not just forward,

but also backward. It is useful to recognize that the *label:* syntax is really just a shorthand for equating to the current value of the location counter (i.e., the address of the next instruction).

In addition to these rules, it is now common for equated symbols to be local to a module unless explicitly declared as global using another directive.

Conditional Assembly. Conditional assembly directives are primarily used for one of two purposes: including routines from libraries and selecting the most appropriate of several alternative codings.

Although linkers are usually used to include the library routines that a program references, it is also possible to literally pass the entire library through the assembler immediately after the user program, conditionally assembling only the library routines that implement symbols referenced in the user's program. This is done with a conditional assembly directive that tests if a symbol has been referenced:

```
ifref afunc
afunc:
; code for this function...
; this code is not assembled unless
; afunc had been previously referenced
endif
```

Within a group of machines that share the same basic architectural model and assembly language, there are usually a number of significant differences relating to performance or existence of certain enhancements. For example, the IA32 architecture spans from the Intel 80386 to the latest Pentium II processors with essentially the same instruction set, but there are differences. One performance difference is that many instructions have changed their relative costs; an idivl (Integer DIVide Long) takes 46 clock cycles on both the 80386 and the Pentium, but an imull (Integer MULtiply Long) instruction could take as many as 41 clock cycles on an 80386 (5) and no more than 11 clock cycles on a Pentium (6). The instruction set has also been extended, not just with the *MMX* (MultiMedia eXtensions) that have been so highly advertised but also with new instructions that provide functions ranging from a wider selection of atomic memory access operations (helpful mostly in multiprocessor systems) to adding a multitude of user-accessible registers and counters for detailed performance analysis. There may even be alternative implementations for systems with identical processors, but different memory sizes, video or other I/O hardware, and the like. Assembling the appropriate version for a particular system could be done by something like:

```
if cache_size>512
; version of code for a big cache
else
; version of code for a small cache
endif
```

Macros. Macros provide a method for creating apparently higher-level instructions by giving a name to a parameterized sequence of assembly language operations. There used to be a wide range of powerful macro definition mechanisms used in assembly language programming, some built-into assemblers and others implemented by using separate preprocessors (e.g., using the C preprocessor or M4).

However, given the wide availability of good compilers that support inline assembly code, assembly language macros have become less useful. The problem with macros is one of performance; it is very difficult to write macros such that expanding two macros in sequence will not result in at least a little unnecessarily inefficient code. Optimizing compilers, on the other hand, are very adept at performing the analysis that, for example, will reallocate registers to avoid unnecessary register-to-register moves.

A Few Words About RISC and CISC. In any discussion of processors and instruction sets, and such discussion is unavoidable when the topic is assemblers, the terms RISC and CISC are unavoidable.

RISC stands for reduced instruction set computer. These processor designs reflect the fact that the simpler the instructions, the easier it is to build very fast hardware. Led by processors like the MIPS (7), even *DSP* chips like Analog Devices SHARC (8) are adopting the RISC philosophy. Because these simpler instructions tend to be more regular, more consistent in the sense that different operations tend to have the same methods for specifying operands, this simplification of the instruction set tends to make compiler optimization technology more effective. It also makes the assembly language constructs easier to understand and memorize. However, the more significant impact of this trend on assembly language programming is that using assembly language to directly encode higher-level, more abstract program concepts requires more of these simpler instructions.

The alternative to RISC is *CISC* (complex instruction set computer). Perhaps the ultimate example of a commercially produced CISC instruction set design was the Intel iAPX 432 (9). This processor was designed to execute high-level language statements as single instructions, with the hardware directly aware of data structures and their protected access mechanisms. An early marketing claim was that there was no assembler or, put another way, that the assembly language arguably was an HLL.

Less extreme examples of higher-level language constructs encoded by single CISC instructions are still common. For example, most high-level languages incorporate the concept of strings. A string is simply an array of character-valued memory cells that can be collectively viewed as a sequence of characters. Thus, basic string operations include making a new copy of a string, comparing two strings, and scanning one string for a nested occurrence of another string. Before RISC, many CISC processor designs directly implemented these kinds of operations in hardware. For example, the Zilog Z80 microprocessor, which was used in many early personal computers including the Radio Shack TRS-80, provides an instruction that can make a copy of an entire string, LDIR (LoaD, Increment, and Repeat) (10). To copy a BC-character string whose address is HL register pair into a string buffer whose address is in DE, one would use the single Z80 assembly language instruction:

```
LDIR    ; memory[DE] = memory[HL]
; DE = DE + 1
; HL = HL + 1
; BC = BC − 1
; repeat until   BC == 0
```

The IBM System/370 family of processors included similar instructions called MVC (MoVe Character) and MVCL (MoVe Character Long, which allows a longer than 256-character string to be copied). The microprocessors used in modern personal computers (*PC*s) are members of the Intel ×86 family (more precisely *IA32*, Intel Architecture 32-bit), which supports a CISC instruction set including a REP (repeat) prefix that generalizes this type of string operation. RISC processors generally do not implement such high-level operations as single instructions.

Writing large-scale software systems in assembly language has not fallen out of favor because CISC instructions were not as abstract as HLL constructs but rather because each instruction set embodies a fixed set of abstractions that are not portable between machines. For example, although the Zilog Z80, IBM System/370, and Intel IA32 families of processors all support string copy instructions, they are not one-for-one interchangeable replacements for each other, nor are their assembly language versions similar enough to allow an assembler to translate one into another trivially.

In summary, there is probably less to be gained over HLL compilers by writing assembly language code for RISC architectures than for CISC. However, the CISC instructions that offer the best speedups are easy to recognize and are easy to apply by using very small segments of assembly language code with a HLL.

Using Assembly Language

There are really three different ways in which assembly language may be used: writing code from scratch, examining and modifying code generated by an HLL compiler, or writing small segments of inline assembly code to be embedded within a HLL program.

Writing Code from Scratch. One of the best features of assembly language is that it allows the programmer total freedom in how the machine is used; however, it is difficult to use code in multiple programs unless it follows some standards for its interface. In assembly languages, the primary interface is the subroutine/function call and return mechanism.

HLL Call Semantics. In HLLs, there are two basic kinds of calls, subroutine calls and function calls. A subroutine, or procedure, call is a statement that has the effect of executing the statements defined within the subroutine and resuming execution. Function calls are similar; however, they return a value and are, hence, valid expressions instead of statements. Some languages, most notably C, allow the usage to determine the type of call: function or subroutine. In this case, a value is always returned, but it is ignored when code is called as a subroutine.

Both subroutines and functions can be "passed" arguments. These arguments can be transmitted in several different ways: as global data, by value, by reference, by name, or by using one of many variations on these techniques.

Passing data to a function by having the function directly reference globally visible storage cells [usually fixed memory locations, but sometimes reserved registers (11)] is a very simple technique. However, it requires that the function and its callers agree on where data will be placed, making it more difficult to reuse a function written for one program within another program. Furthermore, by making more data structures globally visible, the probability of accidental name conflicts or other similar bugs is significantly increased.

All the other techniques are based on the idea of having the caller place information about the arguments directly into what will become the called routine's local storage space. When data are passed by their individual values, each of the arguments is evaluated prior to performing the call, and the resulting values are copied into this space. This is the technique used in most modern languages, including Pascal, C, C++, and Java. In contrast, data passed by reference does not get copied, but a descriptor (often the address of each datum) is passed to the subroutine or function so that it can then directly access the data. This technique is more efficient than call by value when large data structures like arrays are being passed, making it an appropriate choice for Fortran and for Pascal var parameters. C does not directly support call by reference but can simulate it by explicitly passing values that are pointers. Given that call by value copies values and call by reference copies addresses, one begins to wonder what is left to be copied for call by name. Call by name is usually implemented by passing descriptors that are actually the addresses of "thunks" of code that evaluate each argument. This technique was developed for Algol and has been used for little else, but much of the current trend toward object-oriented programming can be seen as evolving toward these semantics. There are also variations of these semantics, like copy-in/copy-out, that copy values into the called routine like call by value, but update the original variables to the values from the called routine's copies when the called routine returns.

Another issue, orthogonal to the choice of argument passing method, is whether recursive calls are allowed. Recursion is the ability of a function to call itself, either directly or indirectly. Most early machine designs encouraged call mechanisms that would not directly support recursion; for example, the DEC PDP7 (12) JMS (JuMp to Subroutine) instruction placed the return address in the word just before the first instruction of the subroutine, and this return address would be incorrectly overwritten if a recursive call were attempted. In one form or another, a stacklike data structure is needed to hold the return address and information about the arguments if recursion is to be supported. The IBM 360 family (13,14) used a linked list of save areas to implement a stack, but most modern processors simply use an array that extends downward from an appropriately high memory address.

As interesting as these various semantics are, the basic reality is that if you want to write assembly language routines that can be called from either C or Fortran, you must use call by address. If they will be called exclusively from C, you can use call by value. In either case, you will probably want to use a method that supports recursive calls because both these languages do. In fact, because you will probably want to use the existing C and/or Fortran compilers, the choice of how to manage arguments was probably made for you when these compilers were written.

Call Stack Frames. Given that nearly all compilers for modern HLLs use stacks to implement recursive subroutine and function calls, it is useful to consider how data are arranged on the stack.

The data associated with a particular invocation of a subroutine or function is called a stack frame. Although the exact structure of a stack frame varies from machine to machine, the following information is generally found within a frame:

- The return address.
- The information about the arguments. In the case of call-by-value languages like C, these are literally copies of the values of the expressions used as parameters within the call. If there is more than one argument, some systems push the arguments in right-to-left order; others push the arguments in left-to-right order. It makes little difference, provided that the same order is used consistently throughout each system.
- Space reserved for local "automatic" variables. If any local variables are declared within the routine, space for these is generally allocated, but not necessarily initialized, within the stack frame.
- Copies of the previous values of any registers that may have been in use or may be reused within this routine. Even though nearly all systems save old register contents in the stack frame and then restore the old values when the function returns, there are many variations. Does the caller save the values of registers or does the called routine save register values? Are all registers saved (some processors have register-block store and load instructions to facilitate this), or are just the ones whose values matter saved? Does the called routine restore registers just before returning, or does the caller restore these values after the routine has returned? These choices are somewhat arbitrary, but the style selected must be consistently applied.
- Space for intermediate values from computations that require more registers than were available. Because modern processor designs tend to have quite a few registers, there tend to be relatively few such stack temporaries. The exceptions are machines with stack-oriented instruction sets, such as Java byte code, that may use a number of stack temporaries because they do not use registers for temporary values.
- Something that identifies where the stack frame began. Often, a register will be reserved as a frame pointer. When a call is made, the previous frame pointer value is pushed onto the stack, and the frame pointer register is made to point at the stack position that holds the previous frame pointer. In this way, the frame pointers form a linked list running back through all the frames on the stack; thus, even if the top of the stack becomes corrupt, the frame pointer chain can be used to provide a trace of where each nested call came from, which can be very useful for debugging. Of course, if the frame is not corrupted and the frame size is known, then no frame pointer is needed.
- For a function, a place to put the return value. Most often, this value is not kept in the stack frame but, instead, is moved into a particular register just before returning. Reserving space in the stack frame for the return value remains a common alternative.

As a more specific example, consider the stack frame structure used by GCC for the IA32 architecture, as shown in Fig. 3.

The function argument values are pushed onto the stack first, in reverse order. Next, the return address is pushed onto the stack as a side-effect of the caller executing a call instruction.

The first thing done by the called routine is to push the value of the frame pointer and make the new frame pointer value be the address in the stack at which the old value was saved. By convention, the IA32

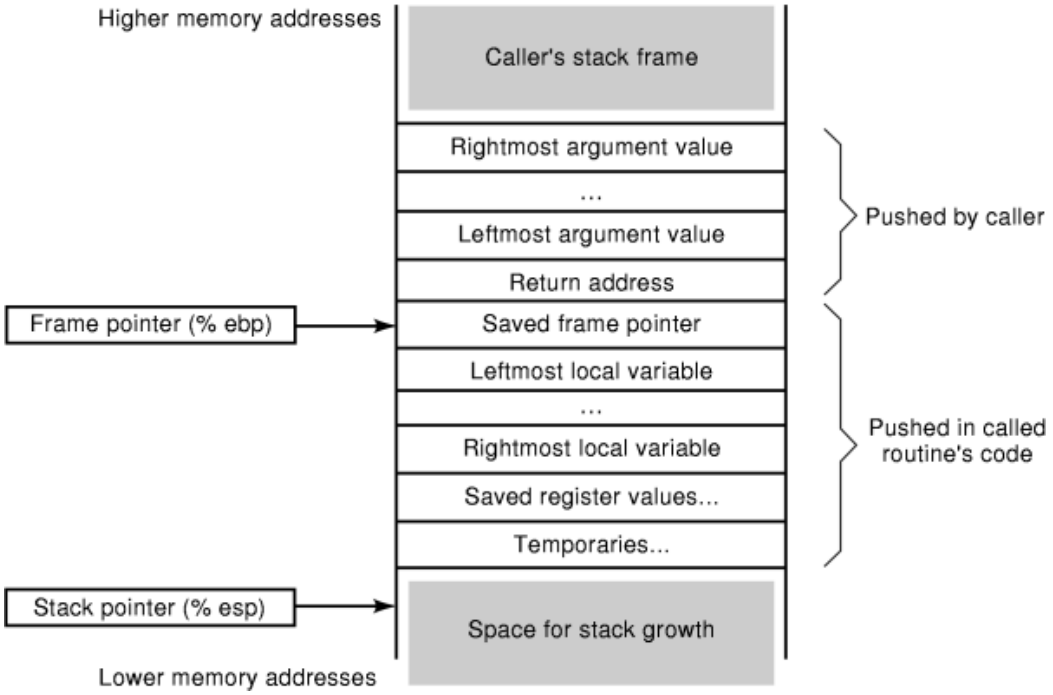

Fig. 3. GCC stack frame structure.

architecture register that is used as the frame pointer is %ebp. Next, space is allocated for local variables in the order that they are declared. No matter how many local variables are allocated, they are allocated using a single instruction that simply decrements the stack pointer, %esp, by the total number of bytes of space required. The C language does not require local variables to be initialized to any particular value, and simply decrementing the stack pointer does not perform any initialization. Notice that both argument values and local variables can be accessed by simple constant offsets from the frame pointer; positive offsets for argument values, negative offsets for local variables. The next issue is the saving of registers; IA32 GCC uses a callee-saves/callee-restores policy because this makes it simple for the compiler to selectively push only the old values of registers that the called function will actually reuse. If there are not enough registers for some computation, temporaries can be freely allocated on the top of the stack.

When the called function is ready to return to the caller, the return value is first placed in the %eax register, which the caller had reserved for this purpose. Next, the called function adjusts the stack pointer upward so that the saved register values are on top of the stack and restores these old register values. The leave instruction restores the old frame pointer value. Finally, a return instruction removes the return address from the stack and jumps to that address.

Finally, back in the caller, we still have the argument values from the call on top of the stack. Because the caller put them there, it knows precisely how many to remove and does this by incrementing the stack pointer by the appropriate amount. Everything is now as it was before the call, except that the function's return value is now in %eax.

Compiler-Generated Assembly Language Code. One of the best ways to become familiar with a particular assembly language is to use a compiler to generate assembly language versions of some simple HLL code. This is generally very easy to do. For example, although most people think that cc is the C compiler on a typical UNIX system, it is usually just a little driver program that recognizes command-line arguments,

invoking the C preprocessor, the C compiler, the assembler, and the linker and removing any intermediate files. To compile a C program called file.c without assembling or linking, the command cc −S file.c will generally yield an assembly language version of your program in a file called file.s. This was the approach used in the following example.

The C function given here as an example uses a modified version of Euclid's method to find the greatest common divisor of two integers:

```
int
gcd(int x, int y)
{
register int t;
t = y;
if (x > 0) {
t = (t % x);
t = gcd(t, x);
}
return(t);
}
```

This function is a good example in that it is very simple yet illustrates recursive function calling, control flow (the if statement), and some simple arithmetic (the modulus operation, %). The following (slightly edited) code was generated by *GCC* the (Gnu C Compiler) for an IA32 (6) system running Linux:

```
gcc2_compiled.:
.text
.align 4
.globl gcd
 .type gcd,@function
gcd:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 12(%ebp),%ebx
cmpl $0,8(%ebp)
jle .L4
movl %ebx,%eax
cltd
idivl 8(%ebp)
movl %edx,%ebx
movl 8(%ebp),%eax
pushl %eax
pushl %ebx
call gcd
addl $8,%esp
movl %eax,%ebx
.L4:
movl %ebx,%eax
jmp .L1
.align 4
.L1:
movl −4(%ebp),%ebx
```

```
leave
ret
.Lfe1:
.size gcd,.Lfe1-gcd
```

The names followed by : characters are labels whose values will be the memory addresses corresponding to those points in the machine code, in assembler parlance, the value of the location counter at that point. The label gcd is a globally visible label marking the start of this function's code; the .globl and .type directives specify this. Directives, which generally begin with . in this assembly language, are not really machine instructions but rather a way of giving the assembler some information that it will use in assembling the program.

The actual assembly language instructions are given indented, one per line. The first word of each of these is the instruction mnemonic, an intuitive name identifying the type of operation that the instruction will perform [e.g., movl MOVes a copy of a Long (32-bit) integer value into another storage cell]. The operands to the instruction, if any, follow the mnemonic as a comma-separated list, and the operand that gets the result, if any, is the rightmost one. There are various different ways to identify operands, and each method uses a slightly different notation. The immediate constant zero is denoted by $0; registers are referenced with a % character in front of their names; a memory location indexed by a register plus a constant offset is written as *offset*(*%register*). There are no explicit references to the C function's variables x, y, and t because x and y are stored on the stack and accessed as 8(%ebp) and 12(%ebp), respectively, whereas the compiler used %ebx to hold the value of t.

Contrast that IA32 assembly language code with a version compiled by the Sun C compiler for a SPARC (2) running the Solaris operating system:

```
.section ``.text'',#alloc,#execinstr
.align 8
.skip 16
 .global gcd
 .type gcd,2
gcd:
 save %sp,−104,%sp
 st %i1,[%fp+72]
st %i0,[%fp+68]
 ld [%fp+72],%l0
mov %l0,%i5
 ld [%fp+68],%l0
 cmp %l0,0
 ble .L17
 nop
 ld [%fp+68],%l0
 mov %i5,%o0
 mov %l0,%o1
 call .rem
 nop
 mov %o0,%l0
 mov %l0,%i5
 ld [%fp+68],%l0
 mov %i5,%o0
 mov %l0,%o1
 call gcd
 nop
```

```
mov %o0,%l0
mov %l0,%i5
.L17:
st %i5,[%fp-4]
ba .L13
nop
.L13:
ld [%fp-4],%l0
mov %l0,%i0
jmp %i7+8
restore
.size gcd,(.-gcd)
```

Although the notation is not identical, and the instruction sets are quite different, there is a striking similarity between these two (and most other) assembly languages. Obvious differences include the use of different mnemonics and the [%fp+*offset*] notation for indexing memory. You may also have noted that this code is slightly longer; this is because the SPARC is a RISC architecture, whereas the IA32 is CISC. For the SPARC, memory accesses occur only in ld (LoaD) or st (STore) instructions, whereas the IA32 can directly access memory even in instructions like cmpl (CoMPare Long). The IA32 also benefits from the complex instruction idivl (Integer DIVide Long), which the SPARC code simulates by calling .rem (a subroutine that computes the remainder for an integer divide). The nops (Null OPerations) in the SPARC code are also a RISC artifact, required to avoid a potential problem in pipelined execution of instructions that modify the program counter.

Note that assembly language programs, even those generated by HLL compilers, are still ordinary text files. Thus, one can hand-optimize the assembly code generated by a compiler using an ordinary text editor. For example, if the gcd function is important to the performance of my program overall, I might want to replace the SPARC code's call to the .rem subroutine with carefully hand-written divide code. After making such changes, most C compiler drivers allow you to assemble and link simply by giving a command like cc file.s; you do not even need to invoke the assembler explicitly. Keep in mind, however, that any changes you make to this assembly language code will be lost if you edit and recompile the HLL source program.

Using Inline Assembly Code in an HLL Program. Given that most programs can and should be written primarily in HLLs like C, the question becomes one of how to insert a few small segments of assembly language code into your HLL program. There are two basic techniques:

- Write and separately assemble your code as pure assembly language functions. This technique offers the advantage that it isolates the assembly code from the HLL code, making it easy to substitute a different function when, for example, the program is ported to a different machine. However, there may be significant overhead in executing the HLL call/return interface code.
- Write your assembly language instructions using the inline assembly mechanism of the HLL compiler. Because the HLL compiler can be aware of the assembly code, this allows the usual HLL call/return interface to be avoided. However, the inline assembly code might require additional information so that the compiler can generate an appropriate interface between the HLL and assembly code.

As a general rule, inline assembly code is more efficient and thus should be used whenever possible.

The syntax used for inline assembly code is, of course, dependent on the target machine's assembly language, but it also varies somewhat with different compilers. The Gnu C Compiler is one of the most widely available and most commonly used compilers that has good support for inline assembly code. Here is a brief introduction to using GCC with inline assembly code for IA32 processors.

In some very time-critical code, it may be necessary to ensure that no external interrupts could distract the processor from the current computation. HLLs like C do not provide any direct means of controlling external interrupts, but assembly language does. For IA32 processors, the ordinary assembly code would be

```
cli;   disable interrupt processing
;non-interruptible code goes here
sti;   enable interrupt processing
```

Neither of these instructions has any operands or side effects that might interfere with the code generated by the C compiler for HLL constructs, so the simplest form of inline assembly syntax can be used. There is no need for HLL call/return sequences; it is sufficient to place each of these assembly language instructions precisely where it is needed within the C program. GCC uses __asm__ to introduce inline assembly code and __volatile__is used to indicate that the code should not be moved or otherwise altered by the compiler's optimizer. Thus, the example is written as

```
__asm__ __volatile__ (``cli'');
/* non-interruptible code goes here */
__asm__ __volatile__ (``sti'');
```

A more complex example of inline assembly code would be one or more assembly language instructions that have operands and yield side effects that may interact with the compiler-generated code.

Whereas most processors allow only I/O (input and output) devices to be accessed by mapping them into memory addresses and then using load and store operations, the IA32 architecture additionally provides instructions that can access I/O devices using an independent I/O port address space. Systems-oriented HLLs like C and Ada have constructs that allow direct access to memory-mapped I/O devices, but not to IA32 I/O ports. The following C code uses GCC's IA32 inline assembly facilities to input a byte from an I/O port and then to output that same value to the port at the next address:

```
unsigned short ioport;
unsigned char datum;
 /* Input the byte datum from I/O port ioport */
 __asm__ __volatile__ (``inb %w1,%b0'',
 :``='' (datum)
 :``d'' (ioport)
 );
 /* Increment ioport to the next I/O port address */
 ioport = ioport + 1;
 /* Output the byte datum to I/O port ioport */
 __asm__ __volatile__ (``outb %b0,%w1'',
 : /* nothing written */
 :``a'' (datum), ``d'' (ioport)
 );
```

The actual instructions generated for these two inline assembly code fragments might be inb %dx,%al and outb %al,%dx, neither of which was literally specified in the inline assembly statements. The %w1 specifies a register holding a word (16-bit short) value that was the second register specified (i.e., "d" (ioport)). Similarly, %b0 specifies a register holding a byte (8-bit char) value that was the first register specified (i.e., "a" (datum)). The :"=a" (datum) portion informs GCC that the register selected by a, which happens to be a particular register but could instead have been a register class from which GCC would have automatically selected a

register, will have its value modified by the inline assembly code and that this result should be seen in the C code as the datum of the variable datum. In the same way, :"d" (ioport) specifies the register that is to be used for this operand to the assembly code and that this register should be initialized with the value of the C variable ioport. In summary, you are telling the compiler quite a bit more than just what assembly instruction to use, but the reward is a zero-overhead interface between your assembly code and the code generated for your C constructs.

Although most programs will need no more than a few inline assembly operations, and many programs need none at all, this ability to use the full instruction set of the machine can be critical in systems-oriented or performance-critical applications. For example, operating systems will need access to various special "privileged mode" instructions that allow manipulation of interrupts and other I/O device interface features, as well as manipulation of the page table and other protected hardware mechanisms. Performance-critical tasks may need access to timers and special processor "performance counters"; they may also need to use special instructions that the compiler does not understand how to generate, such as the MMX (MultiMedia eXtension instructions) that were recently added to the IA32 architecture.

Assembler Technology

Having discussed how assembly languages are structured and how they are used, our concern now shifts to the internal operation of the assembler itself.

Parsing assembly language syntax is trivial using modern techniques, and the transformation of each assembly language statement into the corresponding instruction bit pattern is generally straightforward. However, fundamental problems arise when the bit pattern encoding an operation is to be generated before the assembler has seen the definitions of all components of that statement.

A forward reference occurs whenever a symbol is referenced lexically before it has been defined. Such references are common in assembly language programs, most often in the form of forward branches. For nearly all modern processors, an HLL if statement typically yields assembly language code with a forward branch that skips over the instructions of the then clause if the given condition is false. The result is a conditional forward branch like

```
; evaluate if condition...
 L0: brz L1-L0 ; if 0, skip then clause
 ; instructions for the then clause...
L1:
```

It is a simple enough matter for the assembler to generate the appropriate opcode bit pattern immediately upon reading the assembly language mnemonic brz, a conditional branch-if-zero operation. However, the machine coding of the branch is not complete until an appropriate offset to the location L1 has been added to the coding of the opcode. At the time that the assembler first reads brz L1-L0, it knows the value of L0 but has not yet seen the definition of L1 and, thus cannot generate the operand bit pattern corresponding to the value of L1-L0.

This is not a trivial problem. In fact, HLLs like C and Pascal are very carefully designed so that the compiler can output assembly code without ever needing to use a symbol's definition lexically before the definition of that symbol appears. Of course, the compiler-generated assembly language code may contain many instances of the type of forward reference described, and the assembler will need to resolve these references in order to generate complete machine code. The several different ways in which these forward references can be handled are the foundation of assembler technology.

Backpatching. Backpatching is conceptually the simplest way to resolve a forward reference. As the assembler reads the program, each statement is directly translated to the corresponding machine code bit pattern. When the assembler encounters a reference to an as yet undefined symbol, it can simply leave an appropriately sized gap in the binary output. The location of this gap and the symbolic value that should be encoded there (in this case, L1-L0) are recorded in an internal table of forward references.

When the assembler encounters the definition of a symbol, it notes this symbol–value pair so that any future reference to this symbol immediately can be translated into the appropriate binary representation; this effortlessly handles backward references. However, the newly defined symbol may have been forward referenced, so it also is necessary for the assembler to check the internal table for forward references that can now be resolved to specific values. For each forward reference that now can be resolved, the appropriate bit pattern is computed and then "patched" back into the correct spot in the output binary code stream.

Unfortunately, the patch operations may require random access to the entire output. This can be implemented using seek operations on the binary output file, but such random access can easily result in disk thrashing and poor performance. Hence, this simple technique is usually not very efficient unless the patches can be made on output buffered in memory.

To avoid thrashing when output is sent to disk, the assembler can instead delay the patching until after the end of the input program has been reached and the entire output has been generated. At that time, the complete set of patches needed is known. Rather than performing these patches as each symbol is defined, the complete set of patches to be made can be sorted into increasing address order and then applied in that order using a second pass over the binary output code.

What happens if some values are not known even after the entire source program has been processed? The answer is that this cannot happen if the assembler is to generate pure machine code directly, but often this would occur because the as-yet-undefined symbols appear in separately assembled modules that are expected to be linked to this code. The solution is for the table of unresolved symbolic references to be included in the assembler's output so that the linker can effectively do the patching across separately assembled modules. Note that the linker will also need to know where absolute addresses appear in the binary machine code because it will need to adjust these addresses to appropriately reflect the base addresses it selects for each module worth of machine code being linked together. This patching and relocation are the only functions performed by a typical linker; thus, the primary advantage in using a linker is that the overhead of reparsing the assembly code for commonly used library modules can be avoided.

Span-Dependent and Other Value-Dependent Instructions. Although backpatching can be a very effective method for resolving forward references, it can be used only if the assembler can know the appropriate way to encode the instruction, and the size of the hole to leave for the as-yet-undefined datum, without knowing anything about the forward-referenced value. For some assembly languages, this property can be achieved only by generating potentially inefficient binary code.

The simple forward-reference example given earlier used a branch instruction, but many processors actually have both branch and jump instructions. The subtle distinction is that a jump instruction typically specifies an absolute address to jump to, whereas a branch usually specifies a small offset from the current address. Thus, a jump can go anywhere, but branches have smaller codings, are often faster to execute, and can be used only if the target address is nearby.

A compiler generating assembly code for an HLL construct cannot easily know if a target address would be in range for a branch so it must always generate the assembly language instruction that is safe for the general case, namely jump. To avoid this inefficiency, many assembly languages allow a single mnemonic and assembly language syntax to be used for both jump and branch, with the assembler automatically determining which instruction to use by examining the span between the instruction and the target address. These types of assembly language pseudo-instructions are called span-dependent instructions because their encoding depends on the span. Many assembly languages, including the DEC PDP-11 (15) (and the similar assembly languages used by many Motorola processors), provide this type of span-dependent instruction. Even the IA32 provides

both jump and branch (jump relative) instructions, with either 8-bit or 32-bit signed relative offsets for the branch (6).

Branch/jump operations are by far the most common value-dependent coding problems, but they are not the only instructions with this property. For example, the basic concept of loading a constant value into a register should logically be accomplished by a single assembly language operation, but there are actually many different ways to load a value, and which is most efficient generally depends on what that value is. A trivial example is that loading the value 0 into a register can be done by a CLR (clear) instruction (15) or XOR (exclusive or) of the register with itself (6), either of which yields shorter binary code than loading an immediate value of 0. The Motorola 68000 (16) MOVE and MOVEQ operations can both be used to load an immediate value into a register, but MOVEQ allows only a signed 8-bit immediate value. In some instruction sets, there are a variety of different-length immediate values directly supported rather than just two sizes. For immediate values that are not functions of addresses assigned, the compiler can determine the best coding and output the appropriate assembly language instruction; however, loading values that are functions of addresses (e.g., initializing pointer variables) can be optimized by the assembler only.

For any type of value-dependent coding problem that contains only backward references, an assembler can trivially select the correct coding. However, resolving forward references using backpatching would require all value-dependent codings that contain forward references to assume the most general, least efficient, coding.

Multiple-Pass Resolution. In multiple-pass resolution, forward references are resolved by making multiple passes reading the input program until all the forward-referenced values are known. There are two fundamentally different kinds of passes:

- the final pass (often called Pass 2) and
- all other passes (often called Pass 1—although there may be many occurrences of Pass 1).

In the final pass, the values for all forward references are known because they were determined in previous passes and remembered in a symbol table; hence, code can be generated in sequence. The earlier passes do not generate any code at all but merely keep track of how much code would be generated so that the forward referenced values can be determined and stored for use in later passes.

Parsing the assembly language source file multiple times is not as complicated as it first sounds. The same parser can be used for all passes. The assembler simply rewinds the input file before each pass.

The difficult question is how many passes are needed? Clearly, to achieve the same quality of code (instruction selection) that would be achieved using backpatching, just two passes are needed. However, this does not optimally select the shortest possible form for value-dependent assembly language operations containing forward references. Consider the following assembly code:

```
L0:   jbr   L2
 ; X bytes worth of code...
 L1:   jbr   L3
 ; Y bytes worth of code...
 L2:
 ; Z bytes worth of code...
 L3:
```

If X, Y, and Z are large enough, then a single pass can determine that both jbr operations will need to be coded in the long form, as jump instructions. Likewise, if X, Y, and Z are small enough, a single can also suffice to determine that both jbr operations can use the short form and be coded as branch instructions. However, closer examination reveals that the span between first jbr and L2: is actually X + Y + the size of the coding of the second jbr. Thus, it is possible that X + Y + the size of a branch would make L2: close enough for the

first jbr to be coded as a branch, whereas X + Y + the size of a jump would require the first jbr to be coded as a jump. The thing that makes this interesting is that the assembler does not know the size of the second jbr until it is too late; thus, if the assembler guessed incorrectly for the coding of the first jbr, a second Pass 1 will be needed to determine the optimal instruction codings.

For simple symbolic references (which are all that most compilers will generate), the worst-case number of Pass 1s needed actually approaches *n* for an *n*-instruction program—fortunately, that type of code structure is very rare, and no more than a few Pass 1s are required even for very large programs. The need for another Pass 1 can be detected by noting that at least one value-dependent operation changed its preferred coding. A Pass 1 in which no change occurred signals that we are ready for Pass 2.

Notice that this rule is consistent with our earlier statement that the use of a single Pass 1 is sufficient if all forward-referencing value-dependent instructions are always given the longest form. In fact, if the assembler begins with the assumption that all instructions take the longest form and then shortens the forms as Pass 1s determine that it is safe to do so, we can safely stop after any number of Pass 1s and still have a correct, but perhaps suboptimal, coding of the program. Many assemblers take advantage of this property to impose a fixed limit on the number of passes that will be made. The bad news is that opportunistically shortening forms can yield a suboptimal solution even when the number of Pass 1s is not limited; consider the following:

```
 L0:    jbr    L2
; X bytes worth of code...
L1:    jbr    L0
 ; Y bytes worth of code...
L2:
```

Here, with appropriate choices for X and Y, the choices for the lengths of the jbrs have two stable states: both can take the short form, but if either is assumed long, then both must be long. Opportunistically shortening forms finds the long–long solution; only starting with the short forms and lengthening forms as needed will find the short–short solution. The scary thing about lengthening forms is that the assembler has not found a workable solution until it has found the optimal solution—additional Pass 1s are not optional but required.

An excellent solution was devised by T. G. Szymanski (17). It is equivalent to the multipass lengthening algorithm, but it uses only a single Pass 1. In Pass 1, it not only builds the symbol table entries but also constructs a directed graph in which each node represents a particular span-dependent statement in the assembly language program and is labeled with the minimum possible span for that statement. For each node, an arc is drawn to every other node that could increase that node's span. The graph is processed such that if any node is labeled with a span that requires a longer form, then all nodes pointing at that node have their spans increased appropriately. Pass 2 generates code in the usual way, using the graph to determine which form to code for each span-dependent instruction.

Conclusion and Topics for Further Research

This article has briefly covered why assembly languages are still important, the characteristics of assemblers and assembly languages, how to program in assembly language most effectively, and the basic technology used within assemblers.

If you are only interested in a particular assembly language, there are many sources of good information freely available. You might think that searching the World Wide Web would yield good results, but, unless you care only about IA32 assembly language, searching for "assembly language" is likely to give you many references to assembly languages for processors other than the one you care about. A better starting place is to contact the company that makes the particular processor that you are interested in. The programmer's reference manual for most processors is freely available via the World Wide Web, and this document generally gives a detailed description of the instruction set and the official assembly language.

Assemblers for most processors also can be obtained for free via the World Wide Web. However, keep in mind that the notation used in many of these assemblers is not always a perfect match for the official assembly language notation. For example, the GAS supports assembly languages for a number of different architectures, but in doing so it normalizes their notations to be more similar to each other than the official notations were. In many ways, the increased similarity of the assembly languages is beneficial, but it can also be very frustrating when every example you type in from the processor's manual results in a syntax error.

If you are more interested in building assemblers than in using them, although the basic technology used in assemblers has been stable for many years, there are a few assembler-related research topics actively being pursued by various research groups:

- How can a formal specification of the instruction set be mechanically converted into an efficient assembler? Although a number of assembler systems support multiple assembly languages, including one built by the author of this article (*ASA*, the ASsembler Adaptable), none of these systems fully automates the process of building a new assembler. The New Jersey Machine-Code Toolkit (18) is one of the most advanced systems toward this goal.
- Automatic translation of programs from one machine language to another. A number of researchers have recently focussed on this problem, most often to port IA32 programs to various RISC architectures.
- Optimizing linkers and loaders. Although the traditional linker is constrained to simply relocate or patch address references, some work is now being pursued in which a variety of compiler optimizations would be applied across object modules at link time.

BIBLIOGRAPHY

1. Digital Signal Processor Products, *TMS320C3x User's Guide*, Revision A, Texas Instruments, 1990.
2. *Sun-4 Assembly Language Reference Manual*, Revision A, Sun Microsystems, 1990.
3. S. Heath *Microprocessor Architectures: RISC, CISC and DSP*, 2nd ed., Oxford, UK: Newnes, 1995.
4. Motorola, *HC11—M68HC11 Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
5. *Introduction to the 80386 including the 80386 Data Sheet*, Santa Clara, CA: Intel Corp., 1986.
6. *Pentium Family User's Manual, Architecture and Programming Manual*, Vol. 3, Mt. Prospect, IL: Intel, 1994.
7. J. Heinrich *MIPS R4000 User's Manual*, Englewood Cliffs, NJ: Prentice Hall, 1993.
8. *ADSP-21020/21010 User's Manual*, 2nd ed., Norwood, MA: Analog Devices, 1995.
9. P. Tyner *iAPX 432 General Data Processor Architecture Reference Manual*, Santa Clara, CA: Intel, 1981.
10. W. Barden, Jr. *TRS-80 Assembly-Language Programming*, Fort Worth, TX: Radio Shack, 1979.
11. M. Johnson *Am29000 User's Manual*, Sunnyvale, CA: Advanced Micro Devices, 1990.
12. *PDP-7 Symbolic Assembler Programming Manual*, Maynard, MA: Digital Equipment, 1965.
13. P. Abel *Programming Assembler Language*, Reston, VA: Reston, 1979.
14. G. W. Struble *Assembler Language Programming: The IBM System/360 and 370*, 2nd ed., Reading, MA: Addison-Wesley, 1975.
15. A. Gill *Machine and Assembly Language Programming of the PDP-11*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
16. Motorola, *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, 1984.
17. T. G. Szymanski Assembling code for machines with span-dependent instructions, *Commun. ACM*, **21** (4): 300–308, 1978.
18. N. Ramsey M. Fernandez The New Jersey machine-code toolkit, *Proc. 1995 USENIX Tech. Conf.*, New Orleans, LA, 1995, pp. 289–302.

HENRY DIETZ
Purdue University