2.11 Translating and starting a program

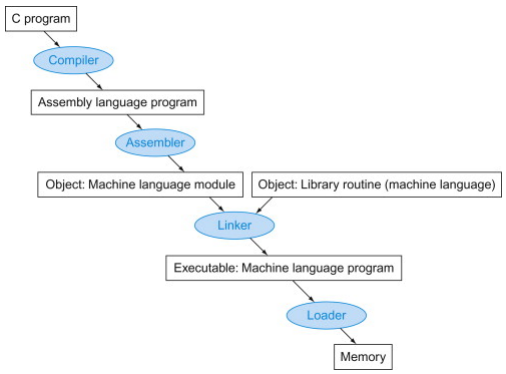
This section has been set as optional by your instructor.

This section describes the four steps in transforming a C program in a file from storage (disk or flash memory) into a program running on a computer. The following figure shows the translation hierarchy. Some systems combine these steps to reduce translation time, but programs go through these four logical phases. This section follows this translation hierarchy.

A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named `x.c`, assembly files are `x.s`, object files are named `x.o`, statically linked library routines are `x.a`, dynamically linked library routes are `x.so`, and executable files by default are called `a.out`. MS-DOS uses the suffixes `.C, .ASM, .OBJ, .LIB, .DLL,` and `.EXE` to the same effect.

PARTICIPATION ACTIVITY 2.11.1: A translation hierarchy for C (COD Figure 2.22).

Start 2x speed



Compiler

The compiler transforms the C program into an *assembly language program*, a symbolic form of what the machine understands. High-level language programs take many fewer lines of code than assembly language, so programmer productivity is much higher.

In 1975, many operating systems and assemblers were written in *assembly language* because memories were small and compilers were inefficient. The million-fold increase in memory capacity per single DRAM chip has reduced program size concerns, and optimizing compilers today can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

Assembly language: A symbolic language that can be translated into binary machine language.

Assembler

Since assembly language is an interface to higher-level software, the assembler can also treat common variations of machine language instructions as if they were instructions in their own right. The hardware need not implement these instructions; however, their appearance in assembly language simplifies translation and programming. Such instructions are called *pseudoinstructions*.

Pseudoinstruction: A common variation of assembly language instructions often treated as if it were an instruction in its own right.

As mentioned above, the LEGv8 hardware makes sure that register `XZR (X31)` always has the value 0. That is, whenever register `XZR` is used, it supplies a 0, and the programmer cannot change the value of register `XZR`. Register `XZR` is used to create the assembly language instruction that copies the contents of one register to another. Thus, the LEGv8 assembler accepts the following instruction even though it is not found in the LEGv8 machine language:

```
MOV X9, X10         // register X9 gets register X10
```

The assembler converts this assembly language instruction into the machine language equivalent of the following instruction:

```
ORR X9, XZR, X10    // register X9 gets 0 OR register X10
```

The LEGv8 assembler also converts `CMP` (compare) into a subtract instruction that sets the condition codes and has `XZR` as the destination. Thus

```
CMP X9, X10         // compare X9 to X10 and set condition codes
```

becomes

```
SUBS XZR, X9, X10   // use X9 - X10 to set condition codes
```

It also converts branches to faraway locations into two branches. As mentioned above, the LEGv8 assembler allows large constants to be loaded into a register despite the limited size of the immediate instructions. Thus, the assembler can accept *load address* (`LDA`) and turn it

into the necessary instruction sequence. Finally, it can simplify the instruction set by determining which variation of an instruction the programmer wants. For example, the LEGv8 assembler does not require the programmer to specify the immediate version of the instruction when using a constant for arithmetic and logical instructions; it just generates the proper opcode. Thus

```
AND X9, X10, #15    // register X9 gets X10 AND 15
```

becomes

```
ANDI X9, X10, #15   // register X9 gets X10 AND 15
```

We include the "I" on the instructions to remind the reader that this instruction produces a different opcode in a different instruction format than the `AND` instruction with no immediate operands.

In summary, pseudoinstructions give LEGv8 a richer set of assembly language instructions than those implemented by the hardware. If you are going to write assembly programs, use pseudoinstructions to simplify your task. To understand the LEGv8 architecture and be sure to get best performance, however, study the real LEGv8 instructions found in COD Figures 2.1 (LEGv8 assembly language revealed in this chapter) and 2.20 (LEGv8 instruction encoding).

Assemblers will also accept numbers in a variety of bases. In addition to binary and decimal, they usually accept a base that is more succinct than binary yet converts easily to a bit pattern. LEGv8 assemblers use hexadecimal.

Such features are convenient, but the primary task of an assembler is assembly into machine code. The assembler turns the assembly language program into an *object file*, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels. Assemblers keep track of labels used in branches and data transfer instructions in a *symbol table*. As you might expect, the table contains pairs of symbols and addresses.

The object file for UNIX systems typically contains six distinct pieces:

- The *object file header* describes the size and position of the other pieces of the object file.
- The *text segment* contains the machine language code.
- The *static data segment* contains data allocated for the life of the program. (UNIX allows programs to use both *static data*, which is allocated throughout the program, and *dynamic data*, which can grow or shrink as needed by the program.)
- The *relocation information* identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
- The *symbol table* contains the remaining labels that are not defined, such as external references.
- The *debugging information* contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

The next subsection shows how to attach such routines that have already been assembled, such as library routines.

Symbol table: A table that matches names of labels to the addresses of the memory words that instructions occupy.

Elaboration

Similar to the case of `ADD` and `ADDI` mentioned above, the full ARMv8 instruction set does not use `ANDI` when one of the operands is an immediate; it just uses `AND`, and lets the assembler pick the proper opcode. For teaching purposes, LEGv8 again distinguishes the two cases with different mnemonics.

PARTICIPATION ACTIVITY 2.11.2: Compiler and assembler.

1) A compiler converts a high-level program, such as a C program, into an assembly language program.
   - ○ True
   - ○ False

2) An assembly language program is a series of 0s and 1s that a machine understands.
   - ○ True
   - ○ False

3) An assembler is used to convert an assembly language program to a machine language program.
   - ○ True
   - ○ False

4) An instruction at address 985 has a label SUM. The assembler might put the following in a symbol table:
   SUM: 985
   - ○ True
   - ○ False

5) An instruction jumps to label SUM. If a symbol table has SUM's address as 985, then the jump will be to address 985.
   - ○ True
   - ○ False

6) An object file is output by a compiler,

and is a combination of machine language instructions, data, and information needed to place the instructions properly in memory.

- ○ True
- ○ False

Linker

What we have presented so far suggests that a single change to one line of one procedure requires compiling and assembling the whole program. Complete retranslation is a terrible waste of computing resources. This repetition is particularly wasteful for standard library routines, because programmers would be compiling and assembling routines that by definition almost never change. An alternative is to compile and assemble each procedure independently, so that a change to one line would require compiling and assembling only one procedure. This alternative requires a new systems program, called a *link editor* or *linker*, which takes all the independently assembled machine language programs and "stitches" them together. The reason a linker is useful is that it is much faster to patch code than it is to recompile and reassemble.

Linker: Also called ***link editor***. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

The linker uses the relocation information and symbol table in each object module to resolve all undefined labels. Such references occur in branch instructions and data addresses, so the job of this program is much like that of an editor: it finds the old addresses and replaces them with the new addresses. Editing is the origin of the name "link editor," or linker for short.

If all external references are resolved, the linker next determines the memory locations each module will occupy. Recall that COD Figure 2.14 (The LEGv8 memory allocation for program and data) shows the LEGv8 convention for allocation of program and data to memory. Since the files were assembled in isolation, the assembler could not know where a module's instructions and data would be placed relative to other modules. When the linker places a module in memory, all *absolute* references, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.

The linker produces an *executable file* that can be run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references. It is possible to have partially linked files, such as library routines, that still have unresolved addresses and hence result in object files.

Executable file: A functional program in the format of an object file that contains no unresolved references. It can contain symbol tables and debugging information. A "stripped executable" does not contain that information. Relocation information may be included for the loader.

Example 2.11.1: Linking object files.

Link the two object files below. Show updated addresses of the first few instructions of the completed executable file. We show the instructions in assembly language just to make the example understandable; in reality, the instructions would be numbers.

Note that in the object files we have highlighted the addresses and symbols that must be updated in the link process: the instructions that refer to the addresses of procedures `A` and `B` and the instructions that refer to the addresses of data doublewords `X` and `Y`.

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	LDUR X0, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	LDUR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	STUR X1, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	STUR	Y
	4	BL	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Answer

Procedure `A` needs to find the address for the variable labeled `X` to put in the load instruction and to find the address of procedure `B` to place in the `BL` instruction. Procedure `B` needs the address of the variable labeled `Y` for the store instruction and the address of procedure `A` for its `BL` instruction.

The text segment starts at address `0000 0000 0040 0000`$_{hex}$ and the data segment at `0000 0000 1000 0000`$_{hex}$. The text of procedure `A` is placed at the first address and its data at the second. The object file header for procedure `A` says that its text is `100`$_{hex}$ bytes and its data is `20`$_{hex}$ bytes, so the starting address for procedure `B` text is `40 0100`$_{hex}$, and its data starts at `1000 0020`$_{hex}$.

Executable file header		
	Text size	300$_{hex}$
	Data size	50$_{hex}$
Text segment	Address	Instruction
	0000 0000 0040 0000$_{hex}$	LDUR X0, [X27,#0$_{hex}$]
	0000 0000 0040 0004$_{hex}$	BL 000 00FC$_{hex}$
	. . .	. . .
	0000 0000 0040 0100$_{hex}$	STUR X1, [X27,#20$_{hex}$]
	0000 0000 0040 0104$_{hex}$	BL 3FF FEFC$_{hex}$
	. . .	. . .
Data segment	Address	
	0000 0000 1000 0000$_{hex}$	(X)
	. . .	. . .
	0000 0000 1000 0020$_{hex}$	(Y)
	. . .	. . .

Now the linker updates the address fields of the instructions. It uses the instruction type field to know the format of the address to be edited. We have two types here:

1. The branch and link instructions use PC-relative addressing. Thus, for the `BL` at address `40 0004`$_{hex}$ to go to `40 0100`$_{hex}$ (the address of procedure `B`), it must put (`40 0100`$_{hex}$ - `40 0004`$_{hex}$) or `000 00FC`$_{hex}$ in its address field. Similarly, since `40 0000`$_{hex}$ is the address of procedure `A`, the `BL` at `40 0104`$_{hex}$ gets the negative number `3FF FEFC`$_{hex}$(`40 0000`$_{hex}$ - `40 0104`$_{hex}$) in its address field.
2. The load and store addresses are harder because they are relative to a base register. This example uses `X27` as the base register, assuming it is initialized to `0000 0000 1000 0000`$_{hex}$. To get the address `0000 0000 1000 0000`$_{hex}$ (the address of doubleword X), we place `0`$_{hex}$ in the address field of `LDUR` at address `40 0000`$_{hex}$. Similarly, we place `20`$_{hex}$ in the address field of `STUR` at address `40 0100`$_{hex}$ to get the address `0000 0000 1000 0020`$_{hex}$ (the address of doubleword `Y`).

Elaboration

Recall that LEGv8 instructions are word aligned, so `BL` drops the right two bits to increase the instruction's address range. Thus, it uses 26 bits to create a 28-bit byte address. Hence, the actual address in the lower 26 bits of the first `BL` instruction in this example is `000 003F`$_{hex}$, rather than `000 00FC`$_{hex}$.

Loader

Now that the executable file is on disk, the operating system reads it to memory and starts it. The *loader* follows these steps in UNIX systems:

1. Reads the executable file header to determine size of the text and data segments.
2. Creates an address space large enough for the text and data.
3. Copies the instructions and data from the executable file into memory.
4. Copies the parameters (if any) to the main program onto the stack.
5. Initializes the processor registers and sets the stack pointer to the first free location.
6. Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an `exit` system call.

Loader: A systems program that places an object program in main memory so that it is ready to execute.

PARTICIPATION ACTIVITY 2.11.3: Linkers and loaders.

1) A linker combines independently assembled machine language programs and then recompiles the programs.
   - ○ True
   - ○ False

2) An executable file is a program that can be run on the computer and has no unresolved references.
   - ○ True
   - ○ False

3) An executable file is executed after a loader places the file into main memory.
   - ○ True
   - ○ False

> "Virtually every problem in computer science can be solved by another level of indirection."
> *David Wheeler*

Dynamically linked libraries

The first part of this section describes the traditional approach to linking libraries before the program is run. Although this static approach is the fastest way to call library routines, it has a few disadvantages:

- The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
- It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program; for example, the standard C library is 2.5 MB.

These disadvantages lead to *dynamically linked libraries (DLLs)*, where the library routines are not linked and loaded until the program is run. Both the program and library routines keep extra information on the location of nonlocal procedures and their names. In the original version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.

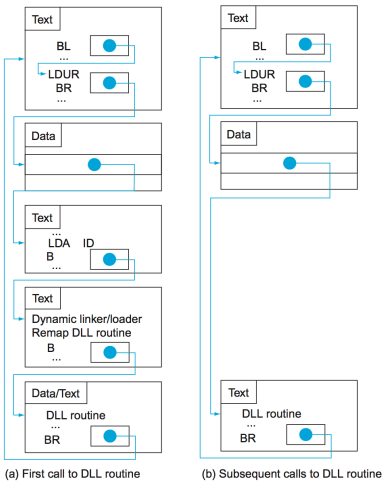
Dynamically linked libraries (***DLLs***): Library routines that are linked to a program during execution.

The downside of the initial version of DLLs was that it still linked all routines of the library that might be called, versus just those that are called during the running of the program. This observation led to the lazy procedure linkage version of DLLs, where each routine is linked only *after* it is called.

Like many innovations in our field, this trick relies on a level of indirection. The figure below shows the technique. It starts with the nonlocal routines calling a set of dummy routines at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect branch.

Figure 2.11.1: Dynamically linked library via lazy procedure linkage (COD Figure 2.23).

(a) Steps for the first time a call is made to the DLL routine. (b) The steps to find the routine, remap it, and link it are skipped on subsequent calls. As we will see in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy), the operating system may avoid copying the desired routine by remapping it using virtual memory management.



The first time the library routine is called, the program calls the dummy entry and follows the indirect branch. It points to code that puts a number in a register to identify the desired library routine and then branches to the dynamic linker/loader. The linker/loader finds the wanted routine, remaps it, and changes the address in the indirect branch location to point to that routine. It then branches to it. When the routine completes, it returns to the original calling site. Thereafter, the call to the library routine branches indirectly to the routine without the extra hops.

In summary, DLLs require additional space for the information needed for dynamic linking, but do not require that whole libraries be copied or linked. They pay a good deal of overhead the first time a routine is called, but only a single indirect branch thereafter. Note that the return from the library pays no extra overhead. Microsoft's Windows relies extensively on dynamically linked libraries, and it is also the default when executing programs on UNIX systems today.

PARTICIPATION ACTIVITY 2.11.4: Dynamically linked libraries.

1) A compiler and assembler using dynamically linked libraries (DLL) create a complete object program that can be loaded and run.
   - ○ True
   - ○ False

2) A DLL approach results in a larger object program.
   - ○ True
   - ○ False

3) A DLL approach better supports library

updates.

- ○ True
- ○ False

4) A DLL approach results in faster-running programs.

- ○ True
- ○ False

5) In a lazy procedure linkage approach to DLL's, each routine is only loaded if a call to the routine exists in the object program.

- ○ True
- ○ False

Starting a Java program

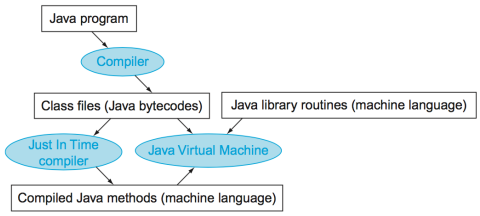
The discussion above captures the traditional model of executing a program, where the emphasis is on fast execution time for a program targeted to a specific instruction set architecture, or even a particular implementation of that architecture. Indeed, it is possible to execute Java programs just like C. Java was invented with a different set of goals, however. One was to run safely on any computer, even if it might slow execution time.

The figure below shows the typical translation and execution steps for Java. Rather than compile to the assembly language of a target computer, Java is compiled first to instructions that are easy to interpret: the *Java bytecode* instruction set. This instruction set is designed to be close to the Java language so that this compilation step is trivial. Virtually no optimizations are performed. Like the C compiler, the Java compiler checks the types of data and produces the proper operation for each type. Java programs are distributed in the binary version of these bytecodes.

Java bytecode: Instruction from an instruction set designed to interpret Java programs.

Figure 2.11.2: A translation hierarchy for Java (COD Figure 2.24).

A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the *Java Virtual Machine* (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.



A software interpreter, called a *Java Virtual Machine* (JVM), can execute Java bytecodes. An interpreter is a program that simulates an instruction set architecture. For example, the ARMv8 simulator used with this book is an interpreter. There is no need for a separate assembly step since either the translation is so simple that the compiler fills in the addresses or JVM finds them at runtime.

Java Virtual Machine (***JVM***) : The program that interprets Java bytecodes.

The upside of interpretation is portability. The availability of software Java virtual machines meant that most people could write and run Java programs shortly after Java was announced. Today, Java virtual machines are found in billions of devices, in everything from cell phones to Internet browsers.

The downside of interpretation is lower performance. The incredible advances in performance of the 1980s and 1990s made interpretation viable for many important applications, but the factor of 10 slowdown when compared to traditionally compiled C programs made Java unattractive for some applications.

To preserve portability and improve execution speed, the next phase of Java's development was compilers that translated *while* the program was running. Such *Just In Time compilers (JIT)* typically profile the running program to find where the "hot" methods are and then compile them into the native instruction set on which the virtual machine is running. The compiled portion is saved for the next time the program is run, so that it can run faster each time it is run. This balance of interpretation and compilation evolves over time, so that frequently run Java programs suffer little of the overhead of interpretation.

Just In Time compiler (***JIT***) : The name commonly given to a compiler that operates at runtime, translating the interpreted code segments into the native code of the computer.

As computers get faster so that compilers can do more, and as researchers invent betters ways to compile Java on the fly, the performance gap between Java and C or C++ is closing. COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) goes into much greater depth on the implementation of Java, Java bytecodes, JVM, and JIT compilers.

PARTICIPATION ACTIVITY 2.11.5: Check yourself: Advantages of an interpreter over a translator.

1) Which of the advantages of an interpreter over a translator was likely the most important for the designers of Java?

- ○ Ease of writing an interpreter

- Higher performance
- Smaller object code
- Machine independence

Provide feedback on this section