12 Assembler and linker tools

The purpose of the assembler and linker tools is to automatically convert the statements, written in assembly language, into code that can be executed by the microprocessor. Without these tools, the programmer would find the task of building reliable programs a forbiddingly difficult task. There is an old saying that a good workman understands his tools; in this chapter we take a brief look at how the assembler and linker tools work.

A major factor in the process of creating reliable programs is that the programmer is able to identify each of the various functions of the program and write the code for each of the functions independently of the other functions. Thus the tools available to the programmer should accommodate the need to be able to write the code for a particular function as though it existed alone, and then to bring together, or **link**, all the functions of the program into the code for the complete program.

Figure 12.1 shows how the GDS program **development environment** accommodates the needs of a programmer who wishes to bring together three source code files in order to make an application program, BigProg. The programmer writes three source code files `*.asm`. She then converts each of the three source code files into three **relocatable files**, `*.rel`, by running the assembler tool three times, once for each source code file. She then uses the linker tool to link the three `*.rel` files into a single file, `BigProg.ihx`, which contains the executable code.

The process of generating executable code thus requires two steps: first, the assembly of all source files containing the various functions, second, the linking of the relocatable files to form executable code. (Even if the whole of the program is contained within just one source code file, the linking process is still required.)

12.1 How an assembler works

Assume that we are the designers of an assembler tool. We know, from Chapter 6, how to assemble source code manually so we shall base our simple design of the assembler tool along similar lines. When assembling a program manually, we made use of the table of all possible G80 instructions given in Appendix A. Thus, our assembler tool will contain a table containing the mnemonic for every G80 instruction together with the bytes of the machine code for the instruction. The basic design is that our assembler tool will read each line of the source code in turn, look up the mnemonic in the table, and write the required machine code to a file. Consider how our assembler will process the source code shown in Figure 12.2. The assembler directives in

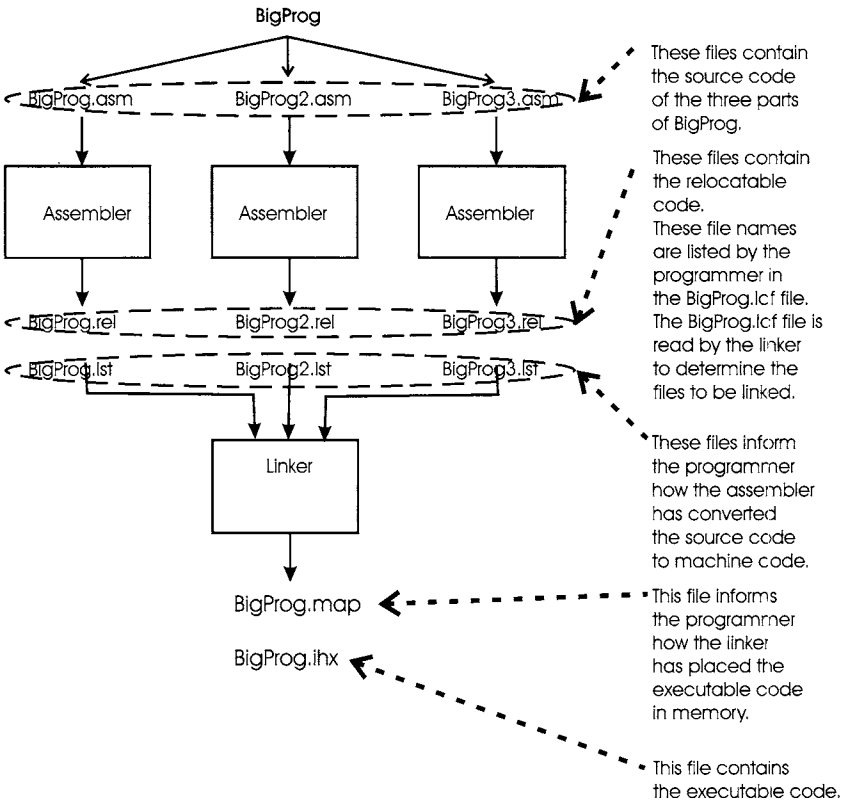

Figure 12.1 *Assembly and linking processes*

```
1          .seg CODE (abs)
2          .org 0x2000
3          ld a, (fred)
4          ld b, a
5          ld a, (jane)
6          halt
7     fred:
8          .db 7
9     jane:
10         .db 4
```

Figure 12.2 *Sample source code*

lines 1 and 2 indicate that the code is to be assembled to begin in memory location 0x2000.

We immediately encounter a problem: when the instruction in line 3 is read, the assembler cannot form the binary code for the instruction. This is because it contains a reference to the memory location named `fred`, but the actual address value of `fred` is not known. Thus, in our design of the assembler tool, we adopt the strategy of first reading through all the lines in the source file to form the actual value of each of the symbols. We shall then read the source file a second time and use the actual values of the symbols to form the required code. Thus, we shall have a **two-pass** assembler.

12.1.1 First pass

To calculate the actual values of the symbolic addresses, we let the assembler program contain a variable called **LocationCounter**. When line 2 is read, the assembler will set LocationCounter to 0x2000. When line 3 is read, the assembler simply determines how many bytes there are in the instruction `ld a, (NN)`; it can do this by looking up the instruction in the instruction table stored within the assembler tool. Having determined that the instruction is 3 bytes long, the assembler adds 3 to the current value of

LocationCounter, making its value 0x2003. After reading and processing line 4, the assembler LocationCounter has the value 0x2004 since the `ld b, a` instruction is 1 byte long. Similarly, after line 5, LocationCounter has the value 0x2007, and after line 6, LocationCounter has the value 0x2008.

When the assembler reads line 7, it detects the symbol `fred` and saves `fred` and the current value of the LocationCounter (0x2008) in a table, the **Symbol Table**. After line 8, LocationCounter has the value 0x2009. When line 9 is read, the assembler detects the symbol `jane` and stores it together with its actual value, 0x2009, in the Symbol Table. After reading all the source code lines, the Symbol Table in the assembler has stored the information:

```
fred = 0x2008
jane = 0x2009
```

12.1.2 Second pass

During the second pass through the source code, the assembler again reads each line of the source code. This time it looks up the binary code for each of the instructions and, if the instruction contains a reference to a symbol, it looks up the actual value in the Symbol Table. For example, when line 3 is processed, the operation code for `ld a,(NN)` is looked up to get 0x3A. The symbol `fred` is then looked up in the Symbol Table to find 0x2008. So the machine code for this instruction is the 3 bytes 0x3A 0x08 0x20. Thus, after the second pass, the source code has been converted to machine code.

12.1.3 Practical assemblers

To make a more useful assembler tool, our basic design must be developed to accommodate the errors that programmers make when writing their source code. These mistakes may be simple typing errors or more serious errors such as defining a symbol more than once. When an error is detected, the assembler tool should produce messages to the programmer that help her to make the required corrections. The flowcharts, Figure 12.3, show the two passes of a simple assembler that informs the user of the most obvious programming errors.

The G80 assembler is actually somewhat more sophisticated than our simple design. One difference is that it does not contain a look-up table for every possible instruction; instead it makes use of the way the operation codes have been constructed by the microprocessor designers. For example, all the `ld register, register` operation codes take the form, in binary, 01 ddd sss where both ddd and sss are 3 bits that identify one of the eight registers, A, B,..L. Thus, once the assembler has detected an instruction of this type, it is able to **compute** the corresponding operation code rather than look it up in a table.

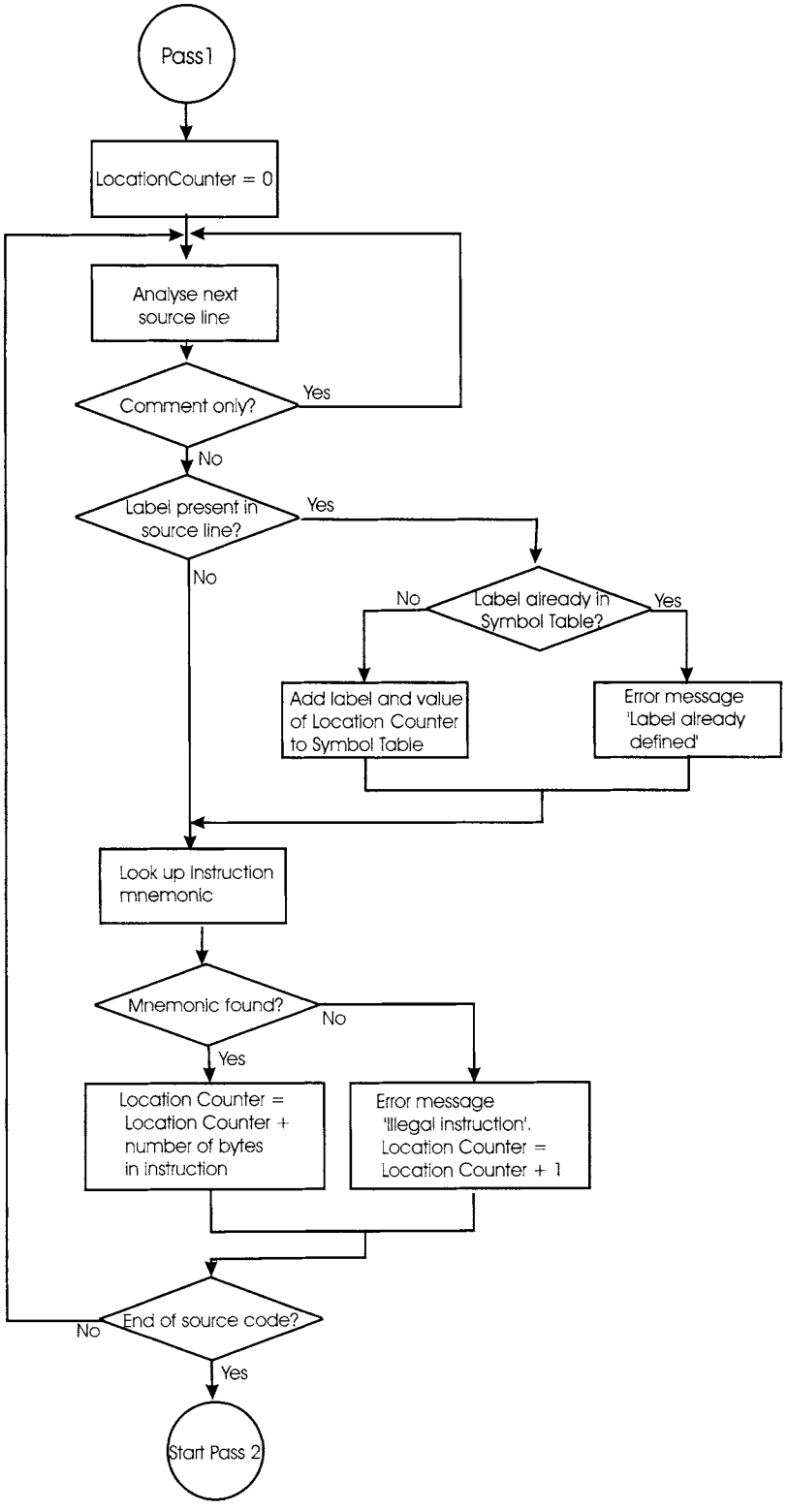

Figure 12.3(a) *First pass of a simple assembler*

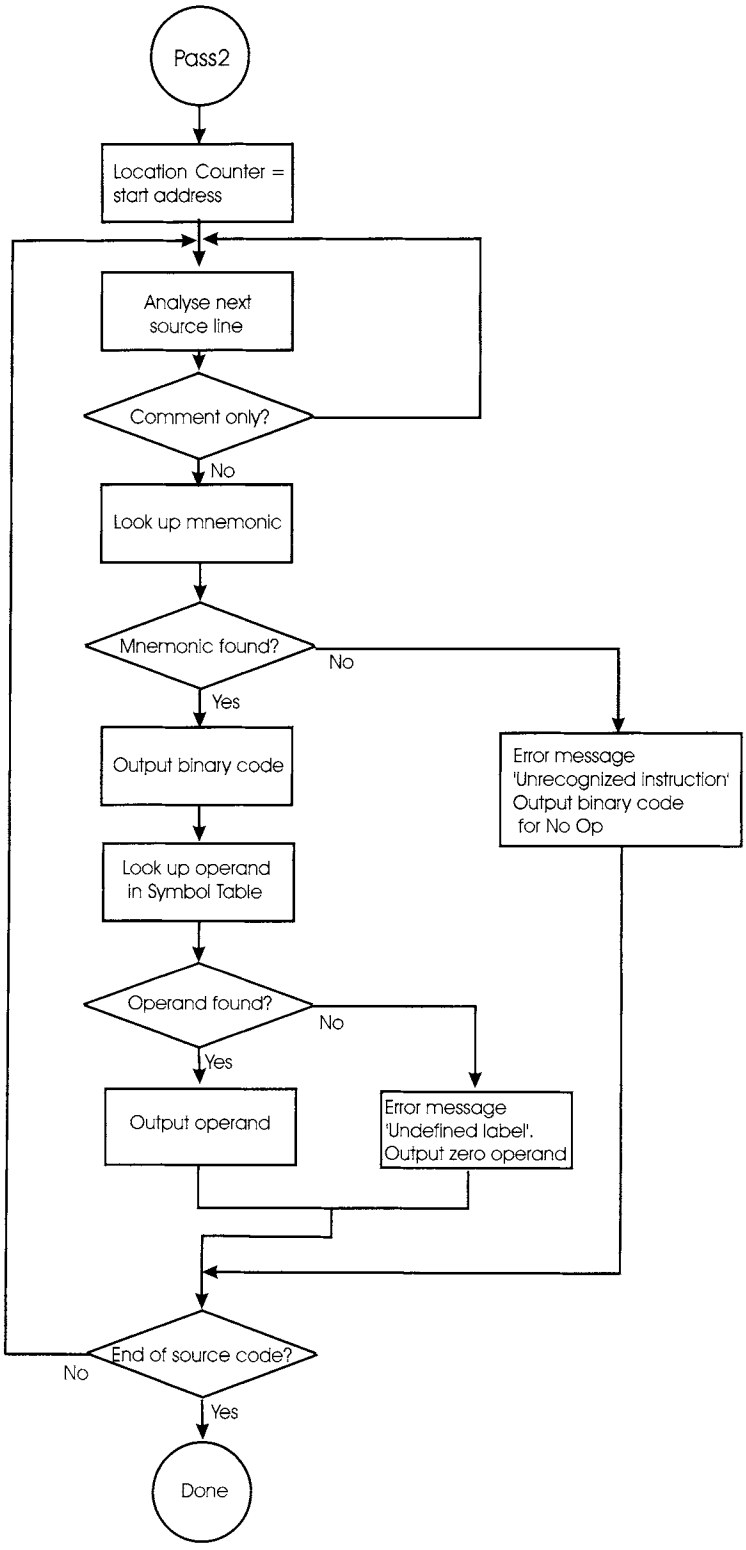

Figure 12.3(b) *Second pass of a simple assembler*

12.1.4 Relocatable segments

In an embedded system, the program code is stored in ROM and data is stored in RAM. It will assist the programmer if she can separate these different uses of the computer memory space within different program **segments** or **areas** within her program. Thus, the programmer may specify that all program instructions are placed in a segment called, say, CODE while all data storage is placed in a segment called, say, DATA. Each individual source code file may then contain one or both segments. (If the programmer does not specify a segment, the G80 assembler assumes that the code is in a relocatable segment.)

For example, Figure 12.4(a) shows three source code files. Source file `BigProg.asm` contains program instructions in segment CODE and the data to which these instructions refer are contained within segment DATA. File `BigProg.asm` contains only program instructions in segment CODE and `BigProg.asm` contains the two segments, CODE and DATA. The intention is to produce one file, which, when loaded into memory, appears as shown in Figure 12.4(b). Here it is assumed that the computer memory contains ROM at 0000 to 7FFF and RAM at 8000 to FFFF. Note that all the CODE segments, A, C, and D, are placed in ROM, while DATA segments B and E are placed in RAM. The assembler must therefore generate files that can be combined in this way. The combining is effected by the linker tool. We discuss the use of the G80 linker next.

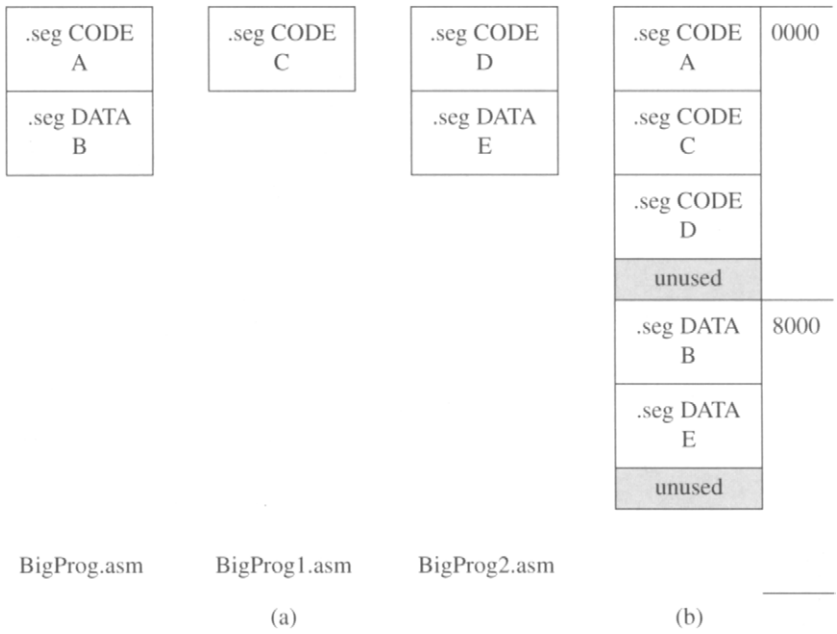

Figure 12.4 (a) *Three source files with segments;* (b) *use of memory*

12.2 Linker

The programmer's project is to produce the 0s and 1s that, when loaded into the memory of the microprocessor, will perform the required functions. In making the project the programmer will usually write the source code of the functions in separate source code files, `*.asm`. The assembler tool converts each of the source code files into a relocatable file, `*.rel`. The purpose of the linker tool is to join, or link, these relocatable files into a single file that can be loaded into the memory of the computer.

In the programs we have considered until now, all the code has been in one source file. Unless instructed otherwise, the linker tool locates the code at memory address 0000. But, given two or more source code files to link, where is the linker to locate the separate pieces of code? We see how this is handled in the following examples.

12.2.1 Link example 1 – single segment

In this example, the linker locates two pieces of code in sequential memory locations. We will write two source code files and place the code in a single relocatable segment. Both source code files are then assembled independently to produce two `*.rel` files. In producing the `*.rel` files, the assembler tool will assemble each of the `*.asm` files as though it begins at memory location 0000. The linker tool will then concatenate[1] the two `*.rel` files according to the order specified in the linker command file, `*.lcf`. That is, the linker tool will read the `*.lcf` file to obtain the name of a relocatable file, and locate that file immediately after the preceding one.

Both the source files in this example, `LinkExample1_FileA.asm` and `LinkExample1_FileB.asm`, contain the `.seg MYCODE (rel)` directive. This indicates that the code in both files is to be contained within a relocatable segment named MYCODE.

Here is the assembly of `LinkExample1_FileA.asm`:

```
                    1   ; LinkExample1_FileA.asm
                    2   .seg MYCODE (rel)
0000 3E 02          3   ld a, 2
0002 06 04          4   ld b, 4
0004 0E 06          5   ld c, 6
```

Here is the assembly of `LinkExample1_FileB.asm`:

```
                    1   ;LinkExample1_FileB.asm
                    2   seg MYCODE (rel)
0000 48             3   ld c, b
0002 47             4   ld b, a
0004 79             5   d a, c
```

Here is the linker command file, `LinkExample1_FileA.lcf`; this instructs the linker to concatenate the two sources in the order shown:

[1] Join end to end.

```
C:\G80UserProgs\LinkExample1_FileA.rel
C:\G80UserProgs\LinkExample1_FileB.rel
```

Type the two files `LinkExample1_FileA.asm` and `LinkExample1_FileB.asm` and assemble both of them independently. With `LinkExample1_FileA` displayed in GDS (`LinkExample1_FileB.lcf` is not used), edit `LinkExample1_FileA.lcf` to make it as shown above, then run the linker tool. Now run the simulator and click on the Disassembly tab[2]. This shows the code that has been produced by the linker: observe that it is as though the following single source file had been written:

```
0000 3E 02      ld a, 2
0002 06 04      ld b, 4
0004 0E 06      ld c, 6
0006 48         ld c, b
0007 47         ld b, a
0008 79         ld a, c
```

The code in `LinkExample1_A.rel` is 6 bytes long and has been located to memory locations 0000 to 0005 inclusive. The code in `LinkExample1_B.rel` is 3 bytes long and has been located to the next memory locations 0006 to 0008 inclusive.

12.2.2 Link example 2 – multiple segments

Usually a source file will refer to data stored in RAM. Let us assume that the computer stores the program code in ROM chips beginning at location 0000 and has RAM beginning at 8000. We will write the program instructions in a relocatable segment named CODE and place data in another relocatable segment named RAM. We shall then use the linker to locate segment CODE at memory address 0000 and segment RAM at memory location 8000.

Here is the assembly listing of `LinkExample2_FileA.asm`:

```
                1   ;LinkExample2_FileA.asm
                2             .seg CODE (rel)
0000 3A`01`00   3             ld a, (sue)
0003 2F         4             cpl
0004 3A`00`00   5             ld a, (fred)
                6   ;
                7             .seg RAM (rel)
0000            8   fred:     .ds 1
0001            9   sue:      .ds 1
```

[2] The G80 disassembler performs the opposite function to the assembler; that is, it converts the 0s and 1s stored in the G80 memory into instruction mnemonics. It does not produce symbolic addresses, instead it shows the absolute address.

Here is the assembly listing of `LinkExample2_FileB.asm`:

```
                    1   ;LinkExample2_FileB.asm
                    2               .seg CODE (rel)
0000 4F             3               ld c, a
0001 32`00`00       4               ld (yoko), a
                    5   ;
                    6               .seg RAM (rel)
0000                7   yoko:       .ds 1
```

Here is the linker command file, `LinkExample2_FileA.lcf`.

```
-b CODE = 0x0000
-b RAM = 0x8000
C:\G80UserProgs\LinkExample2_FileA.rel
C:\G80UserProgs\LinkExample2_FileB.rel
```

The first two lines specify the base addresses of the two segments. (The first line is not strictly necessary since if the base address of a segment is not specified, the linker uses the value 0000.) The programmer must maintain the linker command file manually; GDS helps by automatically writing the basis of the file when a source code file is first assembled.

Use the disassembly function in the simulator to check that the resulting linked code is as though the following, single, source file had been written:

```
0000 3A 01 80        ld a, (sue)
0003 2F              cpl
0004 3A 00 80        ld a, (fred)
0007 4F              ld c, a
0008 32 02 80        ld (yoko), a
                ;
                     .org 0x8000
8000            fred:     .ds 1
8001            sue:      .ds 1
8002            yoko:     .ds 1
```

Observe that the code in segments having the same name has been concatenated. That is, the 4 bytes in the CODE segment of `LinkExample2_FileB` have been located immediately after the 7 bytes in the CODE segment of `LinkExample2_FileA`. In addition, the 1 byte in the RAM segment of `LinkExample2_FileB` has been located immediately after the 2 bytes in the RAM segment of `LinkExample2_FileA`.

12.2.3 Link example 3 – global variables

Often two or more source files will refer to the same data that is stored in RAM. In this example, memory location `fred` is referenced in two source files. In order to make `fred` known to both files, we declare it as a **global** variable, using the assembler directive, `.globl`.

Here is the assembly listing of `LinkExample3_FileA.asm`:

```
                    1   ; LinkExample3_FileA.asm
                    2         .seg CODE (rel)
0000 3E 2A          3         ld a, 42
0002 32`00`00       4         ld (fred), a
                    5   ;
                    6         .globl fred
                    7         .seg DATA (rel)
0000                8   fred: .ds 1
```

Here is the assembly listing of `LinkExample3_FileB.asm`:

```
                    1   ;LinkExample3_FileB.asm
                    2         .globl fred
                    3         .seg CODE (rel)
0000 AF             4         xor a
0001 3A`00`00       5         ld a, (fred)
                    ;Note: Location fred is defined in
                              FileA.asm
```

The linker command file, `LinkExample3_FileA.lcf`, is:

```
-b CODE = 0x0000
-b DATA = 0x8000
C:\G80UserProgs\LinkExample3_FileA.rel
C:\G80UserProgs\LinkExample3_FileB.rel
```

The resulting linked code is as though a single source file had been written:

```
                          .org 0x0000
0000 3E 2A                 ld a, 42
0002 32 00 80              ld (fred), a
0005 AF                    xor a
0006 3A 00 80              ld a, (fred)
                        ;
                           .org 0x8000
8000                fred:  .ds 1
```

12.3 Intel format file

The G80 linker tool combines the various relocatable files into a file named `*.ihx`. The format of this file conforms to a standard for 8-bit microprocessors defined by Intel Corp. In practice, this file is read by a device[3], which writes the 0s and 1s into a read-only memory chip. This chip is then plugged into the computer circuit board so placing the program code into the computer memory[4]. When you invoke the G80 simulator tool, the `*.ihx` file is automatically loaded into the G80 memory.

[3] Such devices are called PROM programmers.

[4] Some microcontroller chips combine a microprocessor and ROM (and RAM and ports) on a single chip. In this case, the PROM programming device writes the `*.ihx` file to the ROM in the microcontroller.

12.4 High-level languages

Writing programs in assembly language is an expensive process. The programmer must be familiar with the physical architecture of the microprocessor and with its instruction set. The programmer must then exercise her ingenuity to write code that uses efficient use of the microprocessor characteristics, and include many comments in order to make the intentions of the code clear. All this amounts to a considerable, and costly, effort. To reduce this effort, John Backus and his colleagues at IBM produced a programming language called FORTRAN (Formula Translator) in 1954 to 1957. This **high-level language** allows programs to be written without the need to understand the details of the microprocessor. The language is translated into assembly code using a **compiler** program. Many other high-level languages and their compilers followed, notably COBOL (Commercial and Business Orientated Language), BASIC (Beginners All-purpose Symbolic Instruction Code), Pascal, Ada, and C. For many years, C has been popular and manufacturers of modern day microprocessors often arrange for a C compiler to be available to produce code for their hardware products. Usually, these compilers run on a desktop PC. Because they run on one computer yet produce code for another microprocessor, they are called **cross-compilers**.

The aim of the programmer is to produce the binary code that will be executed by the microprocessor to produce the required outcome. When a high-level language is used, the programmer has to rely on the compiler to make best use of the architecture of the microprocessor. Thus, the compiler and the microprocessor should be regarded as a combination that implements the intentions of the programmer as described by the high-level language. Since the compiler produces relocatable files that are subsequently linked, a programmer may still choose to write some parts of her program in assembly language (or another high-level language) and link them with those produced by the compiler program. This is called **mixed-language programming**.

12.5 Problems

1 Change the order of the files specified in `LinkExample1_FileA.lcf` and run the linker. Using the disassembler tool in the simulator, observe that the order of the two pieces of code has been changed.

2 Change the base address of segment RAM in `LinkExample2_FileA.lcf` to `0x1000` and run the linker. Using the disassembler tool in the simulator, observe that the data is now located at 0x1000.

3 Modify the code for program `X5Sub.asm`, so that the main routine is in a file named `X5Main.asm` and the X5 subroutine is in a file named `X5.asm`. Build the complete program and test that it produces the correct result.

4 Modify the code for programs `String1.asm` and `CntChar.asm`, so that the main routine is in a file named `TestCntChar.asm` and the `CntChar` subroutine is in a file named `CntCharSub.asm`. Build the complete program and test that it produces the correct result.