with the macro body. A procedure call is a machine instruction that is inserted into the object program and that will later be executed to call the procedure. Figure 7-4 compares macro calls with procedure calls.

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Figure 7-4. Comparison of macro calls with procedure calls.

Conceptually, it is best to think of the assembly process as taking place in two passes. On pass one, all the macro definitions are saved and the macro calls expanded. On pass two, the resulting text is processed as though it was in the original program. In this view, the source program is read in and is then transformed into another program from which all macro definitions have been removed, and in which all macro calls have been replaced by their bodies. The resulting output, an assembly language program containing no macros at all, is then fed into the assembler.

It is important to keep in mind that a program is a string of characters including letters, digits, spaces, punctuation marks, and "carriage returns" (change to a new line). Macro expansion consists of replacing certain substrings of this string with other character strings. A macro facility is a technique for manipulating character strings, without regard to their meaning.

7.2.2 Macros with Parameters

The macro facility previously described can be used to shorten source programs in which precisely the same sequence of instructions occurs repeatedly. Frequently, however, a program contains several sequences of instructions that are almost but not quite identical, as illustrated in Fig. 7-5(a). Here the first sequence exchanges P and Q, and the second sequence exchanges R and S.

Macro assemblers handle the case of nearly identical sequences by allowing macro definitions to provide **formal parameters** and by allowing macro calls to supply **actual parameters**. When a macro is expanded, each formal parameter

```
MOV   EAX,P
MOV   EBX,Q
MOV   Q,EAX
MOV   P,EBX

MOV   EAX,R
MOV   EBX,S
MOV   S,EAX
MOV   R,EBX
```

(a)

```
CHANGE   MACRO P1, P2
         MOV EAX,P1
         MOV EBX,P2
         MOV P2,EAX
         MOV P1,EBX
         ENDM

         CHANGE P, Q

         CHANGE R, S
```

(b)

Figure 7-5. Nearly identical sequences of statements. (a) Without a macro. (b) With a macro.

appearing in the macro body is replaced by the corresponding actual parameter. The actual parameters are placed in the operand field of the macro call. Figure 7-5(b) shows the program of Fig. 7-5(a) rewritten using a macro with two parameters. The symbols *P1* and *P2* are the formal parameters. Each occurrence of *P1* within a macro body is replaced by the first actual parameter when the macro is expanded. Similarly, *P2* is replaced by the second actual parameter. In the macro call

```
CHANGE P, Q
```

P is the first actual parameter and *Q* is the second actual parameter. Thus the executable programs produced by both parts of Fig. 7-5 are identical. They contain precisely the same instructions with the same operands.

7.2.3 Advanced Features

Most macro processors have a whole raft of advanced features to make life easier for the assembly language programmer. In this section we will take a look at a few of MASM's advanced features. One problem that occurs with all assemblers that support macros is label duplication. Suppose that a macro contains a conditional branch instruction and a label that is branched to. If the macro is called two or more times, the label will be duplicated, causing an assembly error. One solution is to have the programmer supply a different label on each call as a parameter. A different solution (used by MASM) is to allow a label to be declared LOCAL, with the assembler automatically generating a different label on each expansion of the macro. Some other assemblers have a rule that numeric labels are automatically local.

MASM and most other assemblers allow macros to be defined within other macros. This feature is most useful in combination with conditional assembly.

Typically, the same macro is defined in both parts of an IF statement, like this:

```
M1   MACRO
     IF WORDSIZE GT 16
M2        MACRO
          ...
          ENDM
     ELSE
M2        MACRO
     ...
          ENDM
     ENDIF
          ENDM
```

Either way, the macro *M2* will be defined, but the definition will depend on whether the program is being assembled on a 16-bit machine or a 32-bit machine. If *M1* is not called, *M2* will not be defined at all.

Finally, macros can call other macros, including themselves. If a macro is recursive, that is, it calls itself, it must pass itself a parameter that is changed on each expansion and the macro must test the parameter and terminate the recursion when it reaches a certain value. Otherwise the assembler can be put into an infinite loop. If this happens, the assembler must be killed explicitly by the user.

7.2.4 Implementation of a Macro Facility in an Assembler

To implement a macro facility, an assembler must be able to perform two functions: save macro definitions and expand macro calls. We will examine these now.

The assembler must maintain a table of all macro names and, along with each name, a pointer to its stored definition so that it can be retrieved when needed. Some assemblers have a separate table for macro names and some have a combined opcode table in which all machine instructions, pseudoinstructions, and macro names are kept.

When a macro definition is encountered, a table entry is made giving the name of the macro, the number of formal parameters, and a pointer to another table—the macro definition table—where the macro body will be kept. A list of the formal parameters is also constructed at this time for use in processing the definition. The macro body is then read and stored in the macro definition table. Formal parameters occurring within the body are indicated by some special symbol. For example, the internal representation of the macro definition of *CHANGE* with semicolon as "carriage return" and ampersand as the formal parameter symbol might be:

```
MOV EAX,&P1; MOV EBX,&P2; MOV &P2,EAX; MOV &P1,EBX;
```

Within the macro definition table the macro body is simply a character string.

During pass one of the assembly, opcodes are looked up and macros expanded. Whenever a macro definition is encountered, it is stored in the macro table. When a macro is called, the assembler temporarily stops reading input from the input device and starts reading from the stored macro body instead. Formal parameters extracted from the stored macro body are replaced by the actual parameters provided in the call. The presence of an ampersand in front of the formal parameters makes it easy for the assembler to recognize them.

7.3 THE ASSEMBLY PROCESS

In the following sections we will briefly describe how an assembler works. Although each machine has a different assembly language, the assembly process is sufficiently similar on different machines that it is possible to describe it in general.

7.3.1 Two-Pass Assemblers

Because an assembly language program consists of a series of one-line statements, it might at first seem natural to have an assembler that read one statement, then translated it to machine language, and finally output the generated machine language onto a file, along with the corresponding piece of the listing, if any, onto another file. This process would then be repeated until the whole program had been translated. Unfortunately, this strategy does not work.

Consider the situation where the first statement is a branch to *L*. The assembler cannot assemble this statement until it knows the address of statement *L*. Statement *L* may be near the end of the program, making it impossible for the assembler to find the address without first reading almost the entire program. This difficulty is called the **forward reference problem**, because a symbol, *L*, has been used before it has been defined; that is, a reference has been made to a symbol whose definition will only occur later.

Forward references can be handled in two ways. First, the assembler may in fact read the source program twice. Each reading of the source program is called a **pass**; any translator that reads the input program twice is called a **two-pass translator**. On pass one, the definitions of symbols, including statement labels, are collected and stored in a table. By the time the second pass begins, the values of all symbols are known; thus no forward reference remains and each statement can be read, assembled, and output. Although this approach requires an extra pass over the input, it is conceptually simple.

The second approach consists of reading the assembly program once, converting it to an intermediate form, and storing this intermediate form in a table in memory. Then a second pass is made over the table instead of over the source program. If there is enough memory (or virtual memory), this approach saves I/O time. If a listing is to be produced, then the entire source statement, including all

the comments, has to be saved. If no listing is needed, then the intermediate form can be reduced to the bare essentials.

Either way, another task of pass one is to save all macro definitions and expand the calls as they are encountered. Thus defining the symbols and expanding the macros are generally combined into one pass.

7.3.2 Pass One

The principal function of pass one is to build up a table called the **symbol table**, containing the values of all symbols. A symbol is either a label or a value that is assigned a symbolic name by means of a pseudoinstruction such as

```
BUFSIZE EQU 8192
```

In assigning a value to a symbol in the label field of an instruction, the assembler must know what address that instruction will have during execution of the program. To keep track of the execution-time address of the instruction being assembled, the assembler maintains a variable during assembly, known as the **ILC** (**Instruction Location Counter**). This variable is set to 0 at the beginning of pass one and incremented by the instruction length for each instruction processed, as shown in Fig. 7-6. This example is for the x86.

Label	Opcode	Operands	Comments	Length	ILC
MARIA:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
ROBERTA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I * I	2	117
	IMUL	EBX, EBX	EBX = J * J	3	119
	IMUL	ECX, ECX	ECX = K * K	3	122
MARILYN:	ADD	EAX, EBX	EAX = I * I + J * J	2	125
	ADD	EAX, ECX	EAX = I * I + J * J + K * K	2	127
STEPHANY:	JMP	DONE	branch to DONE	5	129

Figure 7-6. The instruction location counter (ILC) keeps track of the address where the instructions will be loaded in memory. In this example, the statements prior to MARIA occupy 100 bytes.

Pass one of most assemblers uses at least three internal tables: the symbol table, the pseudoinstruction table, and the opcode table. If needed, a literal table is also kept. The symbol table has one entry for each symbol, as illustrated in Fig. 7-7. Symbols are defined either by using them as labels or by explicit definition (e.g., EQU). Each symbol table entry contains the symbol itself (or a pointer to it), its numerical value, and sometimes other information. This additional information may include

1. The length of data field associated with symbol.
2. The relocation bits. (Does the symbol change value if the program is loaded at a different address than the assembler assumed?)
3. Whether or not the symbol is to be accessible outside the procedure.

Symbol	Value	Other information
MARIA	100	
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Figure 7-7. A symbol table for the program of Fig. 7-6.

The opcode table contains at least one entry for each symbolic opcode (mnemonic) in the assembly language. Figure 7-8 shows part of an opcode table. Each entry contains the symbolic opcode, two operands, the opcode's numerical value, the instruction length, and a type number that separates the opcodes into groups depending on the number and kind of operands.

Opcode	First operand	Second operand	Hex opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Figure 7-8. A few excerpts from the opcode table for an x86 assembler.

As an example, consider the opcode ADD. If an ADD instruction contains EAX as the first operand and a 32-bit constant (immed32) as the second one, then opcode 0x05 is used and the instruction length is 5 bytes. (Constants that can be expressed in 8 or 16 bits use different opcodes, not shown.) If ADD is used with two registers as operands, the instruction is 2 bytes, with opcode 0x01. The (arbitrary) instruction class 19 would be given to all opcode-operand combinations that follow the same rules and should be processed the same way as ADD with two register operands. The instruction class effectively designates a procedure within the assembler that is called to process all instructions of a given type.

Some assemblers allow programmers to write instructions using immediate addressing even though no corresponding target language instruction exists. Such "pseudoimmediate" instructions are handled as follows. The assembler allocates

memory for the immediate operand at the end of the program and generates an instruction that references it. For instance, the IBM 360 mainframe and its successors have no immediate instructions. Nevertheless, programmers may write

```
L 14,=F'5'
```

to load register 14 with a full word constant 5. In this manner, the programmer avoids explicitly writing a pseudoinstruction to allocate a word initialized to 5, giving it a label, and then using that label in the L instruction. Constants for which the assembler automatically reserves memory are called **literals**. In addition to saving the programmer a little writing, literals improve the readability of a program by making the value of the constant apparent in the source statement. Pass one of the assembler must build a table of all literals used in the program. All three of our example computers have immediate instructions, so their assemblers do not provide literals. Immediate instructions are quite common nowadays, but formerly they were unusual. It is likely that the widespread use of literals made it clear to machine designers that immediate addressing was a good idea. If literals are needed, a literal table is maintained during assembly, with a new entry made each time a literal is encountered. After the first pass, this table is sorted and duplicates removed.

Figure 7-9 shows a procedure that could serve as a basis for pass one of an assembler. The style of programming is noteworthy in itself. The procedure names have been chosen to give a good indication of what the procedures do. Most important, Fig. 7-9 represents an outline of pass one which, although not complete, forms a good starting point. It is short enough to be easily understood and it makes clear what the next step must be—namely, to write the procedures used in it.

Some of these procedures will be relatively short, such as *check_for_symbol*, which just returns the symbol as a character string if there is one and null if there is not. Other procedures, such as *get_length_of_type1* and *get_length_of_type2*, may be longer and may call other procedures. In general, the number of types will not be two, of course, but will depend on the language being assembled and how many types of instructions it has.

Structuring programs in this way has other advantages in addition to ease of programming. If the assembler is being written by a group of people, the various procedures can be parceled out among the programmers. All the (nasty) details of getting the input are hidden away in *read_next_line*. If they should change—for example, due to an operating system change—only one subsidiary procedure is affected, and no changes are needed to the *pass_one* procedure itself.

As it reads the program, pass one of the assembler has to parse each line to find the opcode (e.g., ADD), look up its type (basically, the pattern of operands), and compute the instruction's length. This information is also needed on the second pass, so it is possible to write it out explicitly to eliminate the need to parse the line from scratch next time. However, rewriting the input file causes more I/O to

```
public static void pass_one( ) {
   // This procedure is an outline of pass one of a simple assembler.
   boolean more_input = true;                  // flag that stops pass one
   String line, symbol, literal, opcode;       // fields of the instruction
   int location_counter, length, value, type;  // misc. variables
   final int END_STATEMENT = –2;               // signals end of input

   location_counter = 0;                       // assemble first instruction at 0
   initialize_tables( );                       // general initialization

   while (more_input) {                        // more_input set to false by END
      line = read_next_line( );                // get a line of input
      length = 0;                              // # bytes in the instruction
      type = 0;                                // which type (format) is the instruction

      if (line_is_not_comment(line)) {
         symbol = check_for_symbol(line);      // is this line labeled?
         if (symbol != null)                   // if it is, record symbol and value
            enter_new_symbol(symbol, location_counter);
         literal = check_for_literal(line);    // does line contain a literal?
         if (literal != null)                  // if it does, enter it in table
            enter_new_literal(literal);

         // Now determine the opcode type.  –1 means illegal opcode.
         opcode = extract_opcode(line);        // locate opcode mnemonic
         type = search_opcode_table(opcode); // find format, e.g. OP REG1,REG2
         if (type < 0)                         // if not an opcode, is it a pseudoinstruction?
            type = search_pseudo_table(opcode);
         switch(type) {                        // determine the length of this instruction
            case 1: length = get_length_of_type1(line);  break;
            case 2: length = get_length_of_type2(line);  break;
            // other cases here
         }
      }

      write_temp_file(type, opcode, length, line);    // useful info for pass two
      location_counter = location_counter + length;       // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         rewind_temp_for_pass_two( );          // like rewinding the temp file
         sort_literal_table( );                // and sorting the literal table
         remove_redundant_literals( );         // and removing duplicates from it
      }
   }
}
```

Figure 7-9. Pass one of a simple assembler.

occur. Whether it is better to do more I/O to eliminate parsing or less I/O and more parsing depends on the relative speed of the CPU and disk, the efficiency of the file system, and other factors. In this example we will write out a temporary file containing the type, opcode, length, and actual input line. It is this line that pass two reads instead of the raw input file.

When the END pseudoinstruction is read, pass one is over. The symbol table and literal tables can be sorted at this point if needed. The sorted literal table can be checked for duplicate entries, which can be removed.

7.3.3 Pass Two

The function of pass two is to generate the object program and possibly print the assembly listing. In addition, pass two must output certain information needed by the linker for linking up procedures assembled at different times into a single executable file. Figure 7-10 shows a sketch of a procedure for pass two.

```
public static void pass_two( ) {
   // This procedure is an outline of pass two of a simple assembler.
   boolean more_input = true;                  // flag that stops pass two
   String line, opcode;                        // fields of the instruction
   int location_counter, length, type;         // misc. variables
   final int END_STATEMENT = –2;               // signals end of input
   final int MAX_CODE = 16;                    // max bytes of code per instruction
   byte code[ ] = new byte[MAX_CODE];          // holds generated code per instruction

   location_counter = 0;                       // assemble first instruction at 0

   while (more_input) {                        // more_input set to false by END
      type = read_type( );                     // get type field of next line
      opcode = read_opcode( );                 // get opcode field of next line
      length = read_length( );                 // get length field of next line
      line = read_line( );                     // get the actual line of input

      if (type != 0) {                         // type 0 is for comment lines
         switch(type) {                        // generate the output code
            case 1: eval_type1(opcode, length, line, code);  break;
            case 2: eval_type2(opcode, length, line, code);  break;
            // other cases here
         }
      }

      write_output(code);                      // write the binary code
      write_listing(code, line);               // print one line on the listing
      location_counter = location_counter + length;      // update loc_ctr
      if (type == END_STATEMENT) {             // are we done with input?
         more_input = false;                   // if so, perform housekeeping tasks
         finish_up( );                         // odds and ends
      }
   }
}
```

Figure 7-10. Pass two of a simple assembler.

The operation of pass two is more-or-less similar to that of pass one: it reads the lines one at a time and processes them one at a time. Since we have written the type, opcode, and length at the start of each line (on the temporary file), all of these

are read in to save some parsing. The main work of the code generation is done by the procedures *eval_type1*, *eval_type2*, and so on. Each one handles a particular pattern, such as an opcode and two register operands. It generates the binary code for the instruction and returns it in *code*. Then it is written out. More likely, *write_output* just buffers the accumulated binary code and writes the file to disk in large chunks to reduce disk traffic.

The original source statement and the object code generated from it (in hexadecimal) can then be printed or put into a buffer for later printing. After the ILC has been adjusted, the next statement is fetched.

Up until now it has been assumed that the source program does not contain any errors. Anyone who has ever written a program, in any language, knows how realistic that assumption is. Some of the common errors are as follows:

1. A symbol has been used but not defined.
2. A symbol has been defined more than once.
3. The name in the opcode field is not a legal opcode.
4. An opcode is not supplied with enough operands.
5. An opcode is supplied with too many operands.
6. A number contains an invalid character like 143G6.
7. Illegal register use (e.g., a branch to a register).
8. The END statement is missing.

Programmers are quite ingenious at thinking up new kinds of errors to make. Undefined symbol errors are frequently caused by typing errors, so a clever assembler could try to figure out which of the defined symbols most resembles the undefined one and use that instead. Little can be done about correcting most other errors. The best thing for the assembler to do with an errant statement is to print an error message and try to continue assembly.

7.3.4 The Symbol Table

During pass one of the assembly process, the assembler accumulates information about symbols and their values that must be stored in the symbol table for lookup during pass two. Several different ways are available for organizing the symbol table. We will briefly describe some of them below. All of them attempt to simulate an **associative memory**, which conceptually is a set of (symbol, value) pairs. Given the symbol, the associative memory must produce the value.

The simplest implementation technique is indeed to implement the symbol table as an array of pairs, the first element of which is (or points to) the symbol and the second of which is (or points to) the value. Given a symbol to look up, the

symbol table routine just searches the table linearly until it finds a match. This method is easy to program but is slow, because, on the average, half the table will have to be searched on each lookup.

Another way to organize the symbol table is to sort it on the symbols and use the **binary search** algorithm to look up a symbol. This algorithm works by comparing the middle entry in the table to the symbol. If the symbol comes before the middle entry alphabetically, the symbol must be located in the first half of the table. If the symbol comes after the middle entry, it must be in the second half of the table. If the symbol is equal to the middle entry, the search terminates.

Assuming that the middle entry is not equal to the symbol sought, we at least know which half of the table to look for it in. Binary search can now be applied to the correct half, which yields either a match, or the correct quarter of the table. Applying the algorithm recursively, a table of size n entries can be searched in about $\log_2 n$ attempts. Obviously, this way is much faster than searching linearly, but it requires maintaining the table in sorted order.

A completely different way of simulating an associative memory is a technique known as **hash coding** or **hashing**. This approach requires having a "hash" function that maps symbols onto integers in the range 0 to $k - 1$. One possible function is to multiply the ASCII codes of the characters in the symbols together, ignoring overflow, and taking the result modulo k or dividing it by a prime number. In fact, almost any function of the input that gives a uniform distribution of the hash values will do. Symbols can be stored by having a table consisting of k **buckets** numbered 0 to $k - 1$. All the (symbol, value) pairs whose symbol hashes to i are stored on a linked list pointed to by slot i in the hash table. With n symbols and k slots in the hash table, the average list will have length n/k. By choosing k approximately equal to n, symbols can be located with only about one lookup on the average. By adjusting k we can reduce table size at the expense of slower lookups. Hash coding is illustrated in Fig. 7-11.

7.4 LINKING AND LOADING

Most programs consist of more than one procedure. Compilers and assemblers generally translate one procedure at a time and put the translated output on disk. Before the program can be run, all the translated procedures must be found and linked together properly. If virtual memory is not available, the linked program must be explicitly loaded into main memory as well. Programs that perform these functions are called by various names, including **linker**, **linking loader**, and **linkage editor**. The complete translation of a source program requires two steps, as shown in Fig. 7-12:

1. Compilation or assembly of the source files.
2. Linking of the object modules.

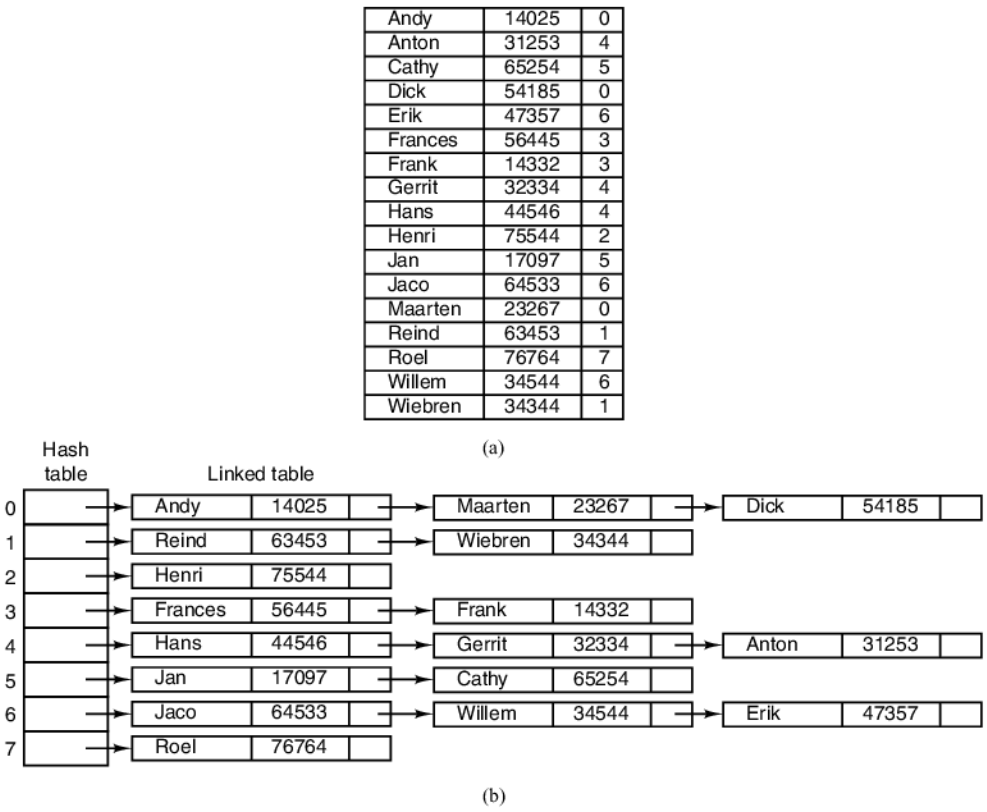

Figure 7-11. Hash coding. (a) Symbols, values, and the hash codes derived from the symbols. (b) Eight-entry hash table with linked lists of symbols and values.

The first step is performed by the compiler or assembler and the second one is performed by the linker.

The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation. The linking process, however, does not represent a change of level, since both the linker's input and the linker's output are programs for the same virtual machine. The linker's function is to collect procedures translated separately and link them together to be run as a unit called an **executable binary program**. On Windows systems, the object modules have extension *.obj* and the executable binary programs have extension *.exe*. On UNIX, the object modules have extension *.o*; executable binary programs have no extension.

Compilers and assemblers translate each source file separately for a very good reason. If a compiler or assembler were to read a series of source procedures and immediately produce a ready-to-run machine language program, changing one

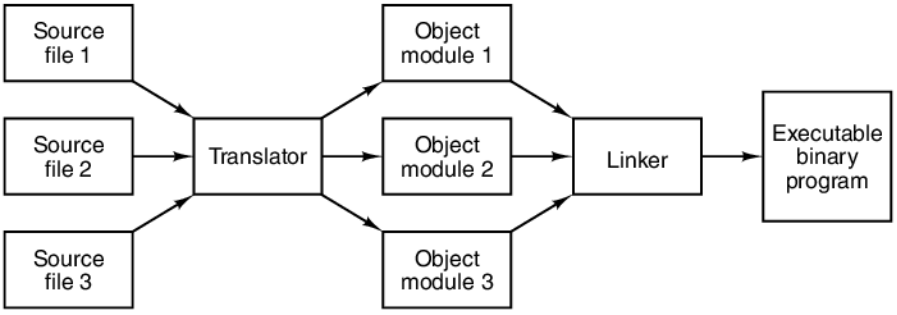

Figure 7-12. Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.

statement in one source procedure would require all the procedures to be retranslated.

If the separate-object-module technique of Fig. 7-12 is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again. Linking is usually much faster than translating, however; thus the two-step process of translating and linking can save a great deal of time during the development of a program. This gain is especially important for programs with hundreds or thousands of modules.

7.4.1 Tasks Performed by the Linker

At the start of pass one of the assembly process, the instruction location counter is set to 0. This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution. Figure 7-13 shows four object modules for a generic machine. In this example, each module begins with a BRANCH instruction to a MOVE instruction within the module.

In order to run the program, the linker brings the object modules from the disk into main memory to form the image of the executable binary program, as shown in Fig. 7-14(a). The idea is to make an exact image of the executable program's virtual address space inside the linker and position all the object modules at their correct locations. If there is not enough (virtual) memory to form the image, a disk file can be used. Typically, a small section of memory starting at address zero is used for interrupt vectors, communication with the operating system, catching uninitialized pointers, or other purposes, so programs often start above 0. In this figure we have (arbitrarily) started programs at address 100.

The program of Fig. 7-14(a), although loaded into the image of the executable binary file, is not yet ready for execution. Consider what would happen if execution began with the instruction at the beginning of module *A*. The program would not branch to the MOVE instruction as it should, because that instruction is now at

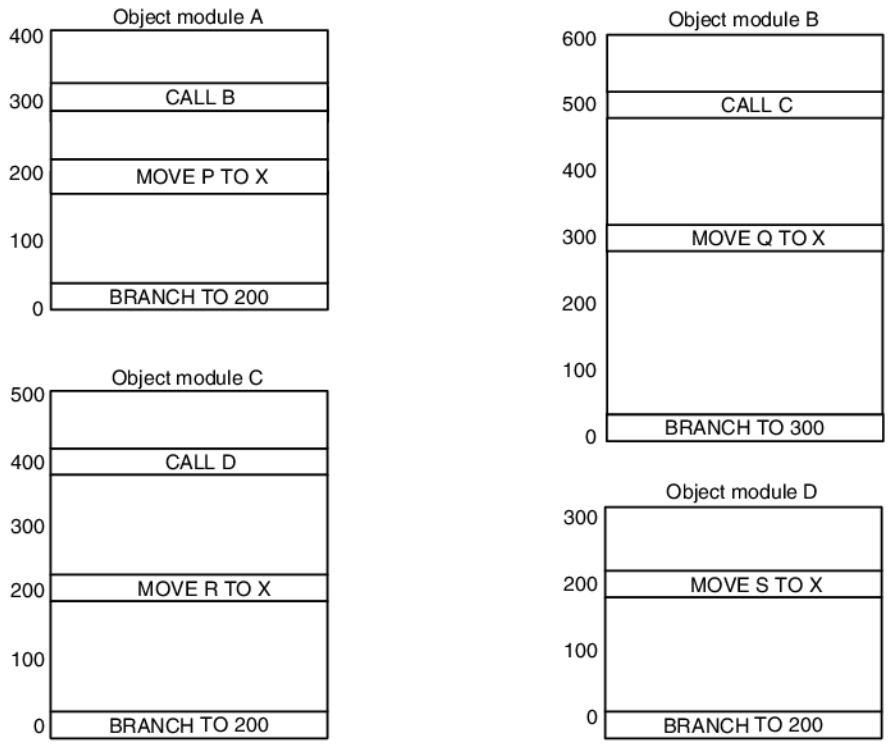

Figure 7-13. Each module has its own address space, starting at 0.

300. In fact, all memory reference instructions will fail for the same reason. Clearly something has to be done.

This problem, called the **relocation problem**, occurs because each object module in Fig. 7-13 represents a separate address space. On a machine with a segmented address space, such as the x86, theoretically each object module could have its own address space by being placed in its own segment. However, OS/2 was the only operating system for the x86 that supported this concept. All versions of Windows and UNIX support only one linear address space, so all the object modules must be merged together into a single address space.

Furthermore, the procedure call instructions in Fig. 7-14(a) will not work either. At address 400, the programmer had intended to call object module *B*, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction. The address of object module *B* is not known until linking time. This problem is called the **external reference** problem. Both of these problems can be solved in a simple way by the linker.

The linker merges the separate address spaces of the object modules into a single linear address space in the following steps:

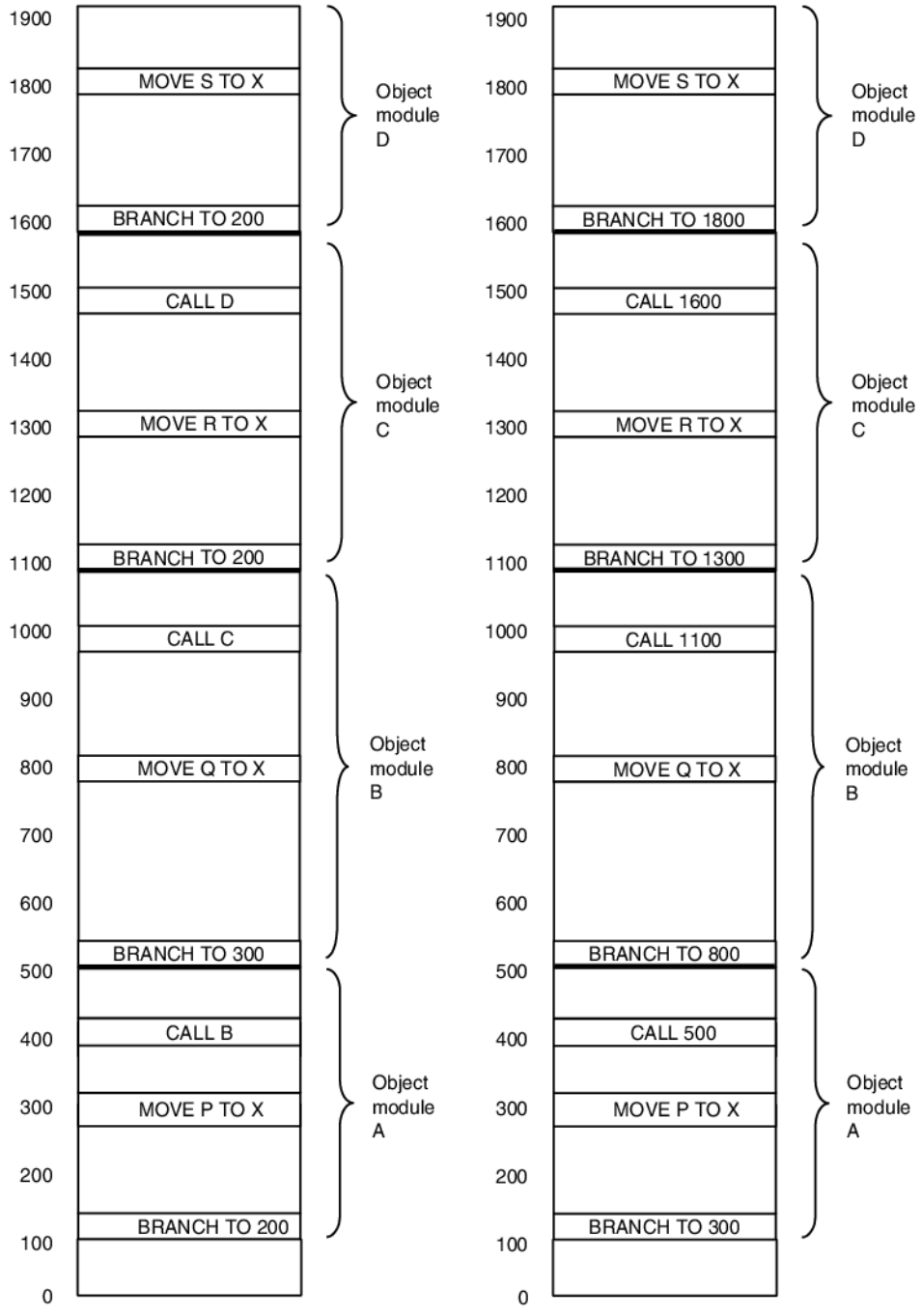

Figure 7-14. (a) The object modules of Fig. 7-13 after being positioned in the binary image but before being relocated and linked. (b) The same object modules after linking and after relocation has been performed.

1. It constructs a table of all the object modules and their lengths.
2. Based on this table, it assigns a base address to each object module.
3. It finds all the instructions that reference memory and adds to each a **relocation constant** equal to the starting address of its module.
4. It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

The object module table constructed in step 1 is shown for the modules of Fig. 7-14 below. It gives the name, length, and starting address of each module.

Module	Length	Starting address
A	400	100
B	600	500
C	500	1100
D	300	1600

Figure 7-14(b) shows how the address space of Fig. 7-14(a) looks after the linker has performed these steps.

7.4.2 Structure of an Object Module

Object modules often contain six parts, as shown in Fig. 7-15. The first part contains the name of the module, certain information needed by the linker, such as the lengths of the various parts of the module, and sometimes the assembly date.

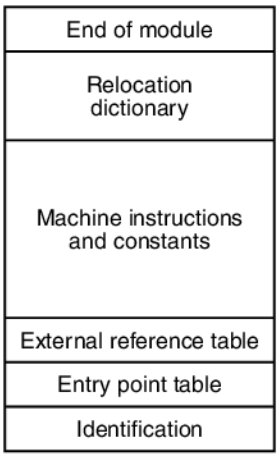

Figure 7-15. The internal structure of an object module produced by a translator. The *Identification* field comes first.

The second part of the object module is a list of the symbols defined in the module that other modules may reference, together with their values. For example, if the module consists of a procedure named *bigbug*, the entry point table will

contain the character string "bigbug" followed by the address to which it corresponds. The assembly language programmer indicates which symbols are to be declared as **entry points** by using a pseudoinstruction such as PUBLIC in Fig. 7-2.

The third part of the object module consists of a list of the symbols that are used in the module but which are defined in other modules, along with a list of which machine instructions use which symbols. The linker needs the latter list in order to be able to insert the correct addresses into the instructions that use external symbols. A procedure can call other independently translated procedures by declaring the names of the called procedures to be external. The assembly language programmer indicates which symbols are to be declared as **external symbols** by using a pseudoinstruction such as EXTERN in Fig. 7-2. On some computers entry points and external references are combined into one table.

The fourth part of the object module is the assembled code and constants. This part of the object module is the only one that will be loaded into memory to be executed. The other five parts will be used by the linker to help it do its work and then discarded before execution begins.

The fifth part of the object module is the relocation dictionary. As shown in Fig. 7-14, instructions that contain memory addresses must have a relocation constant added. Since the linker has no way of telling by inspection which of the data words in part four contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table. The information may take the form of a bit table, with 1 bit per potentially relocatable address, or an explicit list of addresses to be relocated.

The sixth part is an end-of-module mark, perhaps a checksum to catch errors made while reading the module, and the address at which to begin execution.

Most linkers require two passes. On pass one the linker reads all the object modules and builds up a table of module names and lengths, and a global symbol table consisting of all entry points and external references. On pass two the object modules are read, relocated, and linked one module at a time.

7.4.3 Binding Time and Dynamic Relocation

In a multiprogramming system, a program can be read into main memory, run for a little while, written to disk, and then read back into main memory to be run again. In a large system, with many programs, it is difficult to ensure that a program is read back into the same locations every time.

Figure 7-16 shows what would happen if the already relocated program of Fig. 7-14(b) were reloaded at address 400 instead of address 100 where the linker put it originally. All the memory addresses are incorrect; moreover, the relocation information has long since been discarded. Even if the relocation information were still available, the cost of having to relocate all the addresses every time the program was swapped would be too high.

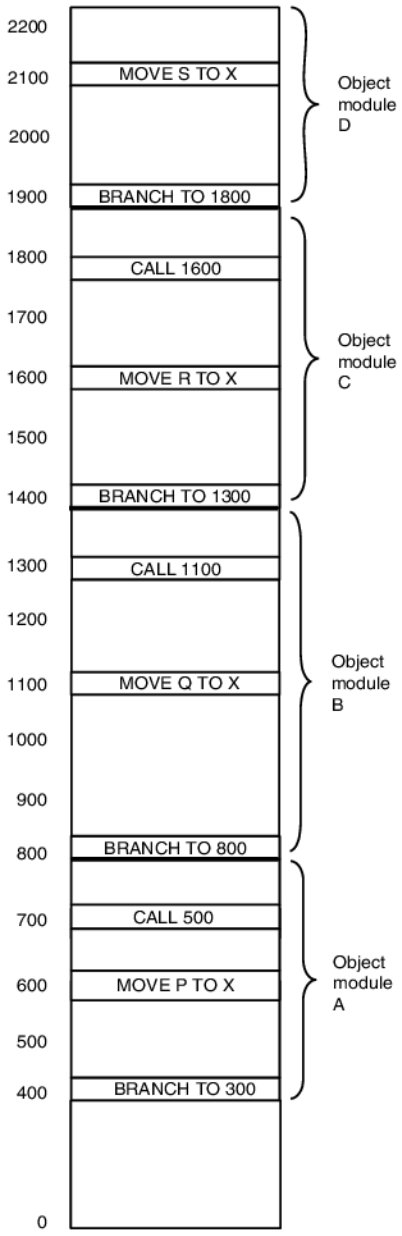

Figure 7-16. The relocated binary program of Fig. 7-14(b) moved up 300 addresses. Many instructions now refer to an incorrect memory address.

The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed. When a program is written it contains symbolic names for memory addresses, for example, `BR L`. The time at which the actual main memory address corresponding to *L* is determined is called the **binding time**. At least six possibilities for the binding time exist:

1. When the program is written.
2. When the program is translated.
3. When the program is linked but before it is loaded.
4. When the program is loaded.
5. When a base register used for addressing is loaded.
6. When the instruction containing the address is executed.

If an instruction containing a memory address is moved after binding, it will be incorrect (assuming that the object referred to has also been moved). If the translator produces an executable binary as output, the binding has occurred at translation time, and the program must be run at the address at which the translator expected it to be run at. The linking method described in the preceding section binds symbolic names to absolute addresses during linking, which is why moving programs after linking fails, as shown in Fig. 7-16.

Two related issues are involved here. First, there is the question of when symbolic names are bound to virtual addresses. Second, there is a question of when virtual addresses are bound to physical addresses. Only when both operations have taken place is binding complete. When the linker merges the separate address spaces of the object modules into a single linear address space, it is, in fact, creating a virtual address space. The relocation and linking serve to bind symbolic names onto specific virtual addresses. This observation is true whether or not virtual memory is being used.

Assume for the moment that the address space of Fig. 7-14(b) is paged. It is clear that the virtual addresses corresponding to the symbolic names *A*, *B*, *C*, and *D* have already been determined, even though their physical main memory addresses will depend on the contents of the page table at the time they are used. An executable binary program is really a binding of symbolic names to virtual addresses.

Any mechanism that allows the mapping of virtual addresses onto physical memory addresses to be changed easily will facilitate moving programs around in main memory, even after they have been bound to a virtual address space. One such mechanism is paging. After a program has been moved in main memory, only its page table need be changed, not the program itself.

A second mechanism is the use of a runtime relocation register. The CDC 6600 and its successors had such a register. On machines using this relocation

technique, the register always points to the physical address of the start of the current program. All memory addresses have the contents of the relocation register added to them by the hardware before being sent to the memory. The entire relocation process is transparent to the user programs. They do not even know that it is occurring. When a program is moved, the operating system must update the relocation register. This mechanism is less general than paging because the entire program must be moved as a unit (unless there are separate code and data relocation registers, as on the Intel 8088, in which case it has to be moved as two units).

A third mechanism is possible on machines that can refer to memory relative to the program counter. Many branch instructions are relative to the program counter, which helps. Whenever a program is moved in main memory only the program counter need be updated. A program, all of whose memory references are either relative to the program counter or absolute (e.g., to I/O device registers at absolute addresses) is said to be **position independent**. A position-independent procedure can be placed anywhere within the virtual address space without the need for relocation.

7.4.4 Dynamic Linking

The linking strategy discussed in Sec. 7.4.1 has the property that all procedures that a program might call are linked before the program can begin execution. On a computer with virtual memory, completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory. Many programs have procedures that are called only under unusual circumstances. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.

A more flexible way to link separately compiled procedures is to link each procedure at the time it is first called. This process is known as **dynamic linking**. It was pioneered by MULTICS whose implementation is in some ways still unsurpassed. In the next sections we will look at dynamic linking in several systems.

Dynamic Linking in MULTICS

In the MULTICS form of dynamic linking, associated with each program is a segment, called the **linkage segment**, which contains one block of information for each procedure that might be called. This block of information starts with a word reserved for the virtual address of the procedure and it is followed by the procedure name, which is stored as a character string.

When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block, as shown in Fig. 7-17(a). The compiler fills this word with either an invalid address or a special bit pattern that forces a trap.

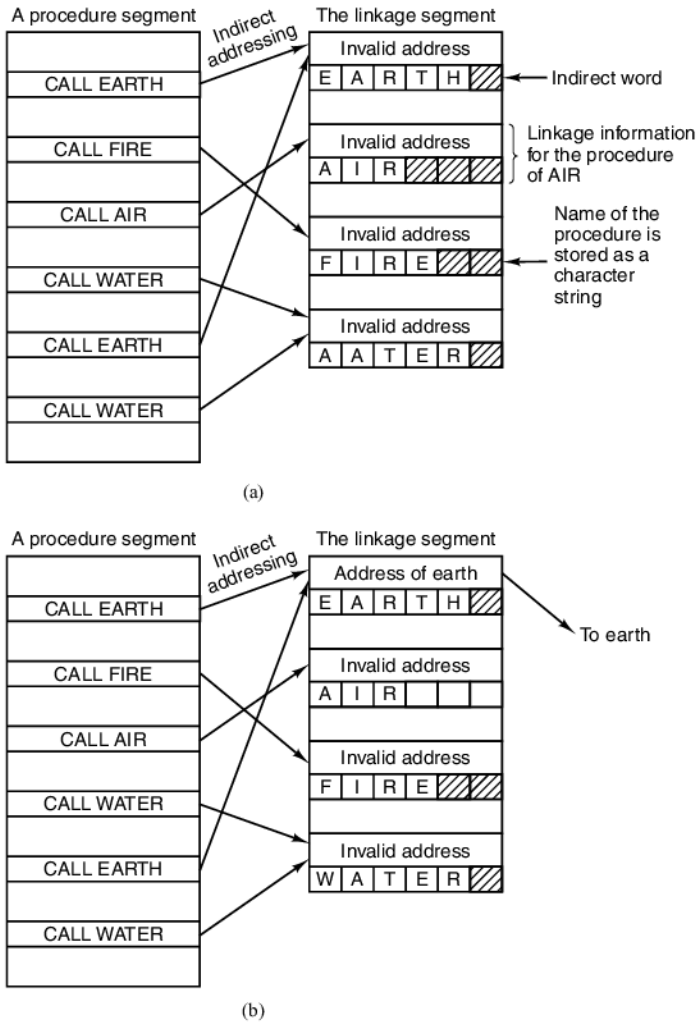

Figure 7-17. Dynamic linking. (a) Before *EARTH* is called. (b) After *EARTH* has been called and linked.

When a procedure in a different segment is called, the attempt to address the invalid word indirectly causes a trap to the dynamic linker. The linker then finds the character string in the word following the invalid address and searches the user's file directory for a compiled procedure with this name. That procedure is then assigned a virtual address, usually in its own private segment, and this virtual address overwrites the invalid address in the linkage segment, as indicated in Fig. 7-17(b). Next, the instruction causing the linkage fault is re-executed, allowing the program to continue from the place it was before the trap.

All subsequent references to that procedure will be executed without causing a linkage fault, for the indirect word now contains a valid virtual address. Consequently, the dynamic linker is invoked only the first time a procedure is called.

Dynamic Linking in Windows

All versions of the Windows operating system support dynamic linking and rely heavily on it. Dynamic linking uses a special file format called a **DLL** (**Dynamic Link Library**). DLLs can contain procedures, data, or both. They are commonly used to allow two or more processes to share library procedures or data. Many DLLs have extension *.dll*, but other extensions are also in use, including *.drv* (for driver libraries) and *.fon* (for font libraries).

The most common form of a DLL is a library consisting of a collection of procedures that can be loaded into memory and accessed by multiple processes at the same time. Figure 7-18 illustrates two processes sharing a DLL file that contains four procedures, *A*, *B*, *C*, and *D*. Program 1 uses procedure *A*; program 2 uses procedure *C*, although they could equally well have used the same procedure.

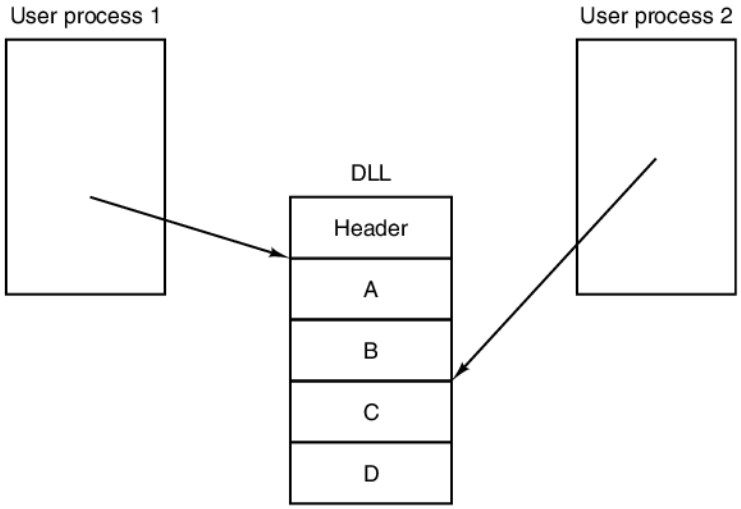

Figure 7-18. Use of a DLL file by two processes.

A DLL is constructed by the linker from a collection of input files. In fact, building a DLL file is very much like building an executable binary program, except that a special flag is given to the linker to tell it to make a DLL. DLLs are commonly constructed from collections of library procedures that are likely to be needed by multiple processes. The interface procedures to the Windows system call library and large graphics libraries are common examples of DLLs. The advantage o using DLLs is saving space in memory and on disk. If some common li-

brary were statically bound to each program using it, it would appear in many executable binaries on the disk and in memory, wasting space. With DLLs, each library appears only once on disk and once in memory.

In addition to saving space, this approach makes it easy to update library procedures, even after the programs using them have been compiled and linked. For commercial software packages, where the users rarely have the source code, using DLLs means that the software vendor can fix bugs in the libraries by just distributing new DLL files over the Internet, without requiring any changes to the main program binaries.

The main difference between a DLL and an executable binary is that a DLL cannot be started and run on its own (because it has no main program). It also has different information in its header. In addition, the DLL as a whole has several extra procedures not related to the procedures in the library. For example, there is one procedure that is automatically called whenever a new process is bound to the DLL and another one that is automatically called whenever a process is unbound from it. These procedures can allocate and deallocate memory or manage other resources needed by the DLL.

There are two ways for a program to bind to a DLL. In the first way, called **implicit linking**, the user's program is statically linked with a special file called an **import library** that is generated by a utility program that extracts certain information from the DLL. The import library provides the glue that allows the user program to access the DLL. A user program can be linked with multiple import libraries. When a program using implicit linking is loaded into memory for execution, Windows examines it to see which DLLs it uses and checks to see if all of them are already in memory. Those that are not in memory are loaded immediately (but not necessarily in their entirety, since they are paged). Some changes are then made to the data structures in the import libraries so the called procedures can be located, somewhat analogous to the changes shown in Fig. 7-17. They also have to be mapped into the program's virtual address space. At this point, the user program is ready to run and can call the procedures in the DLLs as though they had been statically bound with it.

The alternative to implicit linking is (not surprisingly) **explicit linking**. This approach does not require import libraries and does not cause the DLLs to be loaded at the same time the user program is. Instead, the user program makes an explicit call at run time to bind to a DLL, then makes additional calls to get the addresses of procedures it needs. Once these have been found, it can call the procedures. When it is all done, it makes a final call to unbind from the DLL. When the last process unbinds from a DLL, the DLL can be removed from memory.

It is important to realize that a procedure in a DLL does not have any identity of its own (as a thread or process does). It runs in the called's thread and uses the called's stack for its local variables. It can have process-specific static data (as well as shared data) and otherwise behaves the same as a statically-linked procedure. The only essential difference is how the binding to it is performed.

Dynamic Linking in UNIX

The UNIX system has a mechanism similar in essence to DLLs in Windows. It is called a **shared library**. Like a DLL file, a shared library is an archive file containing multiple procedures or data modules that are present in memory at run time and can be bound to multiple processes at the same time. The standard C library and much of the networking code are shared libraries.

UNIX supports only implicit linking, so a shared library consist of two parts: a **host library**, which is statically linked with the executable file, and a **target library**, which is called at run time. While the details differ, the concepts are essentially the same as with DLLs.

7.5 SUMMARY

Although most programs can and should be written in a high-level language, occasional situations exist in which assembly language is needed, at least in part. Programs for resource-poor computers such as smart cards and embedded processors in small consumer devices like clock radios are potential candidates. An assembly language program is a symbolic representation for some underlying machine language program. It is translated to the machine language by a program called an assembler.

When extremely fast execution is critical to the success of some application, a better approach than writing everything in assembly language is to first write the whole program in a high-level language, then measure where it is spending its time, and finally rewrite only those portions of the program that are heavily used. In practice, a small fraction of the code is usually responsible for a large fraction of the execution time.

Many assemblers have a macro facility that allows the programmer to give commonly used code sequences symbolic names for subsequent inclusion. Usually, these macros can be parameterized in a straightforward way. Macros are implemented by a kind of literal string-processing algorithm.

Most assemblers are two pass. Pass one is devoted to building up a symbol table for labels, literals, and explicitly declared identifiers. The symbols can either be kept unsorted and then searched linearly, first sorted and then searched using binary search, or hashed. If symbols do not need to be deleted during pass one, hashing is usually the best method. Pass two does the code generation. Some pseudoinstructions are carried out on pass one and some on pass two.

Independently-assembled programs can be linked together to form an executable binary program that can be run. This work is done by the linker. Its primary tasks are relocation and binding of names. Dynamic linking is a technique in which certain procedures are not linked until they are actually called. Windows DLLs and UNIX shared libraries use dynamic linking.

PROBLEMS

1. For a certain program, 2% of the code accounts for 50% of the execution time. Compare the following three strategies with respect to programming time and execution time. Assume that it would take 100 man-months to write it in C, and that assembly code is 10 times slower to write and four times more efficient.

a. Entire program in C.
b. Entire program in assembler.
c. First all in C, then the key 2% rewritten in assembler.

2. Do the considerations that hold for two-pass assemblers also hold for compilers?

a. Assume that the compilers produce object modules, not assembly code.
b. Assume that the compilers produce symbolic assembly language.

3. Most assemblers for the x86 have the destination address as the first operand and the source address as the second operand. What problems would have to be solved to do it the other way?

4. Can the following program be assembled in two passes? EQU is a pseudoinstruction that equates the label to the expression in the operand field.

```
P EQU Q
Q EQU R
R EQU S
S EQU 4
```

5. The Dirtcheap Software Company is planning to produce an assembler for a computer with a 48-bit word. To keep costs down, the project manager, Dr. Scrooge, has decided to limit the length of allowed symbols so that each symbol can be stored in a single word. Scrooge has declared that symbols may consist only of letters, except the letter Q, which is forbidden (to demonstrate their concern for efficiency to the customers). What is the maximum length of a symbol? Describe your encoding scheme.

6. What is the difference between an instruction and a pseudoinstruction?

7. What is the difference between the instruction location counter and the program counter, if any? After all, both keep track of the next instruction in a program.

8. Show the symbol table after the following x86 statements have been encountered. The first statement is assigned to address 1000.

```
EVEREST:    POP BX          (1 BYTE)
K2:         PUSH BP         (1 BYTE)
WHITNEY:    MOV BP,SP       (2 BYTES)
MCKINLEY:   PUSH X          (3 BYTES)
FUJI:       PUSH SI         (1 BYTE)
KIBO:       SUB SI,300      (3 BYTES)
```

9. Can you envision circumstances in which an assembly language permits a label to be the same as an opcode (e.g., *MOV* as a label)? Discuss.

10. Show the steps needed to look up Ann Arbor using binary search on the following list: Ann Arbor, Berkeley, Cambridge, Eugene, Madison, New Haven, Palo Alto, Pasadena, Santa Cruz, Stony Brook, Westwood, and Yellow Springs. When computing the middle element of a list with an even number of elements, use the element just after the middle index.

11. Is it possible to use binary search on a table whose size is prime?

12. Compute the hash code for each of the following symbols by adding up the letters (A = 1, B = 2, etc.) and taking the result modulo the hash table size. The hash table has 19 slots, numbered 0 to 18.

els, jan, jelle, maaike

Does each symbol generate a unique hash value? If not, how can the collision problem be dealt with?

13. The hash coding method described in the text links all the entries having the same hash code together on a linked list. An alternative method is to have only a single *n*-slot table, with each table slot having room for one key and its value (or pointers to them). If the hashing algorithm generates a slot that is already full, a second hashing algorithm is used to try again. If that one is also full, another is used, and so on, until an empty is found. If the fraction of the slots that are full is *R*, how many probes will be needed, on the average, to enter a new symbol?

14. As technology progresses, it may one day be possible to put thousands of identical CPUs on a chip, each CPU having a few words of local memory. If all CPUs can read and write three shared registers, how can an associative memory be implemented?

15. The x86 has a segmented architecture, with multiple independent segments. An assembler for this machine might well have a pseudoinstruction `SEG N` that would direct the assembler to place subsequent code and data in segment *N*. Does this scheme have any influence on the ILC?

16. Programs often link to multiple DLLs. Would it not be more efficient just to put all the procedures in one big DLL and then link to it?

17. Can a DLL be mapped into two process' virtual address spaces at different virtual addresses? If so, what problems arise? Can they be solved? If not, what can be done to eliminate them?

18. One way to do (static) linking is as follows. Before scanning the library, the linker builds a list of procedures needed, that is, names defined as `EXTERN` in the modules being linked. Then the linker goes through the library linearly, extracting every procedure that is in the list of names needed. Does this scheme work? If not, why not and how can it be remedied?

19. Can a register be used as the actual parameter in a macro call? How about a constant? Why or why not?

20. You are to implement a macro assembler. For esthetic reasons, your boss has decided that macro definitions need not precede their calls. What implications does this decision have on the implementation?

21. Think of a way to put a macro assembler into an infinite loop.

22. A linker reads five modules, whose lengths are 200, 800, 600, 500, and 700 words, respectively. If they are loaded in that order, what are the relocation constants?

23. Write a symbol table package consisting of two routines: *enter(symbol, value)* and *lookup(symbol, value)*. The former enters new symbols in the table and the latter looks them up. Use some form of hash coding.

24. Repeat the previous problem, only this time instead of using a hash table, after the last symbol is entered, sort the table and use a binary-lookup algorithm to find symbols.

25. Write a simple assembler for the Mic-1 computer of Chap. 4. In addition to handling the machine instructions, provide a facility for assigning constants to symbols at assembly time, and a way to assemble a constant into a machine word. These should be pseudoinstructions, of course.

26. Add a simple macro facility to the assembler of the preceding problem.

8

PARALLEL COMPUTER ARCHITECTURES

Although computers keep getting faster, the demands placed on them are increasing at least as fast. Astronomers want to simulate the entire history of the universe, from the big bang until the show is over. Pharmaceutical engineers would love to be able to design medicines for specific diseases on their computer instead of having to sacrifice legions of rats. Aircraft designers could come up with more fuel-efficient products if computers could do all the work, without the need for constructing physical wind-tunnel prototypes. In short, for many users, however much computing power is available, especially in science, engineering, and industry, it is never enough.

Although clock rates are continually rising, circuit speed cannot be increased indefinitely. The speed of light is already a major problem for designers of high-end computers, and the prospects of getting electrons and photons to move faster are dim. Heat-dissipation issues are turning supercomputers into state-of-the-art air conditioners. Finally, as transistor sizes continue to shrink, at some point each transistor will have so few atoms in it that quantum-mechanical effects (e.g., the Heisenberg uncertainty principle) may become a major problem.

Therefore, in order to be able to handle larger and larger problems, computer architects are turning increasingly to parallel computers. While it may not be possible to build a computer with one CPU and a cycle time of 0.001 nsec, it may well be possible to build one with 1000 CPUs each with a cycle time of 1 nsec. Although the latter design uses slower CPUs, its total computing capacity is theoretically the same. Herein lies the hope.

Parallelism can be introduced at various levels. At the lowest level, it can be added to the CPU chip, through pipelining and superscalar designs with multiple functional units. It can also be added by having very long instruction words with implicit parallelism. Special features can be added to a CPU to allow it to handle multiple threads of control at once. Finally, multiple CPUs can be put together on the same chip. Together, these features can pick up perhaps a factor of 10 in performance over purely sequential designs.

At the next level, extra CPU boards with additional processing capacity can be added to a system. Usually, these plug-in CPUs have specialized functions, such as network packet processing, multimedia processing, or cryptography. For specialized applications, they can also gain a factor of perhaps 5 to 10.

However, to win a factor of a hundred or a thousand or a million, it is necessary to replicate entire CPUs and to make them all work together efficiently. This idea leads to large multiprocessors and multicomputers (cluster computers). Needless to say, hooking up thousands of processors into a big system leads to its own problems that need to be solved.

Finally, it is now possible to lash together entire organizations over the Internet to form very loosely coupled compute grids. These systems are only starting to emerge, but have interesting potential for the future.

When two CPUs or processing elements are close together, have a high bandwidth and low delay between them, and are computationally intimate, they are said to be **tightly coupled**. Conversely, when they are far apart, have a low bandwidth and high delay and are computationally remote, they are said to be **loosely coupled**. In this chapter we will look at the design principles for these various forms of parallelism and study a variety of examples. We will start with the most tightly coupled systems, those that use on-chip parallelism, and gradually move to more and more loosely coupled systems, ending with a few words on grid computing. This spectrum is crudely illustrated in Fig. 8-1.

The whole issue of parallelism, from one end of the spectrum to the other, is a hot topic of research. Accordingly, many references are given in this chapter, primarily to recent papers on the subject. Many conferences and journals publish papers on the subject as well and the literature is growing rapidly.

8.1 ON-CHIP PARALELLISM

One way to increase the throughput of a chip is to have it do more things at the same time. In other words, exploit parallelism. In this section, we will look at some of the ways of achieving speed-up through parallelism at the chip level, including instruction-level parallelism, multithreading, and putting more than one CPU on the chip. These techniques are quite different, but each helps in its own way. In all cases the idea is to get more activity going at the same time.

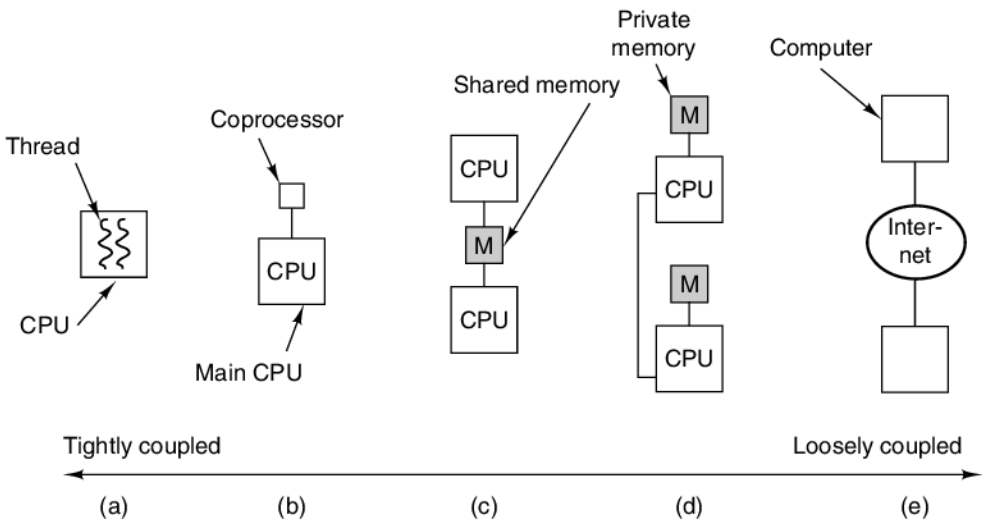

Figure 8-1. (a) On-chip parallelism. (b) A coprocessor. (c) A multiprocessor. (d) A multicomputer. (e) A grid.

8.1.1 Instruction-Level Parallelism

At the lowest level, one way to achieve parallelism is to issue multiple instructions per clock cycle. Multiple-issue CPUs come in two varieties: superscalar processors and VLIW processors. We have touched on both earlier in the book, but it may be useful to review this material here.

We have seen superscalar CPUs before (e.g., Fig. 2-5). In the most common configuration, at a certain point in the pipeline an instruction is ready to be executed. Superscalar CPUs are capable of issuing multiple instructions to the execution units in a single clock cycle. The number of instructions actually issued depends on both the processor design and current circumstances. The hardware determines the maximum number that can be issued, usually two to six instructions. However, if an instruction needs a functional unit that is not available or a result that has not yet been computed, the instruction will not be issued.

The other form of instruction-level parallelism is found in **VLIW** (**Very Long Instruction Word**) processors. In the original form, VLIW machines indeed had long words containing instructions that used multiple functional units. Consider, for example, the pipeline of Fig. 8-2(a), where the machine has five functional units and can perform two integer operations, one floating-point operation, one load, and one store simultaneously. A VLIW instruction for this machine would contain five opcodes and five pairs of operands, one opcode and operand pair per functional unit. With 6 bits per opcode, 5 bits per register, and 32 bits per memory address, instructions could easily be 134 bits—quite long indeed.

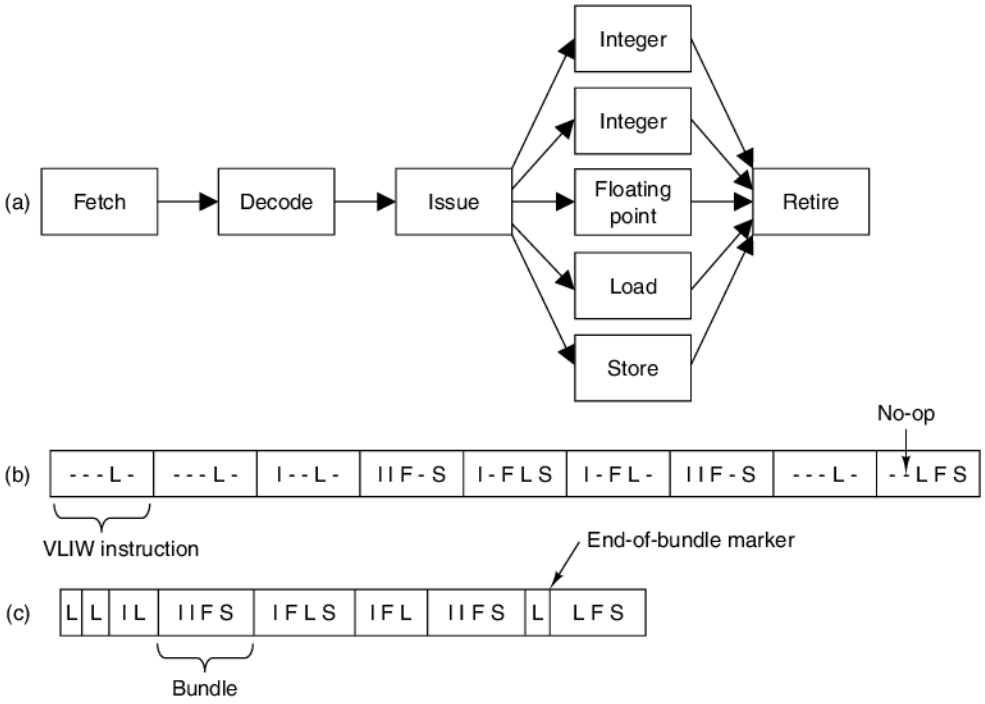

Figure 8-2. (a) A CPU pipeline. (b) A sequence of VLIW instructions. (c) An instruction stream with bundles marked.

However, this design proved too rigid because not every instruction was able to utilize every functional unit, leading to many useless NO-OPs used as filler, as illustrated in Fig. 8-2(b). Consequently, modern VLIW machines have a way of marking a bundle of instructions as belonging together, for example with an "end-of-bundle" bit, as shown in Fig. 8-2(c). The processor can then fetch the entire bundle and issue it all at once. It is up to the compiler to prepare bundles of compatible instructions.

In effect, VLIW shifts the burden of determining which instructions can be issued together from run time to compile time. Not only does this choice make the hardware simpler and faster, but since an optimizing compiler can run for a long time if need be, better bundles can be assembled than what the hardware could do at run time. Of course, such a radical change in CPU architecture will be difficult to introduce, as demonstrated by the slow acceptance of the Itanium except for niche applications.

It is worth noting in passing that instruction-level parallelism is not the only form of low-level parallelism. Another is memory-level parallelism, in which multiple memory operations are in flight at the same time (Chou et al., 2004).

The TriMedia VLIW CPU

We studied one example of a VLIW CPU, the Itanium-2, in Chap. 5. Let us now look at a very different VLIW processor, the **TriMedia**, designed by Philips, the Dutch electronics company that also invented the audio CD and CD-ROM. The TriMedia is intended for use as an embedded processor in image-, audio-, and video-intensive applications such as CD, DVD, and MP3 players, CD and DVD recorders, interactive TV sets, digital cameras, camcorders, and so on. Given these application areas, it is not surprising that it differs considerably from the Itanium-2, which is a general-purpose CPU intended for high-end servers.

The TriMedia is a true VLIW processor with every instruction holding as many as five **operations**. Under completely optimal conditions, every clock cycle one instruction is started and the five operations are issued. The clock runs at 266 MHz or 300 MHz, but since five operations per cycle can be issued, the effective clock speed is as much as five times higher. In the discussion below, we will focus on the TM3260 implementation of the TriMedia; other versions differ in minor ways from it.

A typical instruction is illustrated in Fig. 8-3. The instructions vary from standard 8-, 16-, and 32-bit integer instructions through IEEE 754 floating-point instructions to parallel multimedia instructions. As a consequence of the five issues per cycle and the parallel multimedia instructions, the TriMedia is fast enough to decode streaming DV from a camcorder at full size and full frame rate in software.

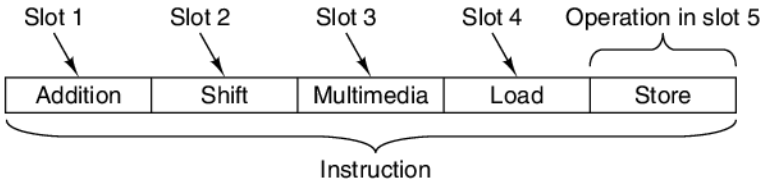

Figure 8-3. A typical TriMedia instruction, showing five possible operations.

The TriMedia has a byte-oriented memory, with the I/O registers mapped into the memory space. Half words (16 bits) and full words (32 bits) must be aligned on their natural boundaries. It can run either as little endian or big endian, depending on a PSW bit that the operating system can set. This bit affects only the way load operations and store operations transfer between memory and registers. The CPU contains a split 8-way set-associative cache, with a 64-byte line size for both the instruction cache and the data cache. The instruction cache is 64 KB; the data cache is 16 KB.

There are 128 general-purpose 32-bit registers. Register `R0` is hardwired to 0. Register `R1` is hardwired to 1. Attempting to change either one gives the CPU a heart attack. The remaining 126 registers are all functionally equivalent and can be used for any purpose. In addition, four special-purpose, 32-bit registers also exist.

These are the program counter, program status word, and two registers that relate to interrupts. Finally, a 64-bit register counts the number of CPU cycles since the CPU was last reset. At 300 MHz, it takes nearly 2000 years for the counter to wrap around.

The Trimedia TM3260 has 11 different functional units for doing arithmetic, logical, and control flow operations (as well as one for cache control that we will not discuss here). They are listed in Fig. 8-4. The first two columns name the unit and give a brief description of what it does. The third column tells how many hardware copies of the unit exist. The fourth column gives the latency, that is, how many clock cycles it takes to complete. In this context, it is worth nothing that all the functional units except the FP square-root/divide unit are pipelined. The latency given in the table tells how long before the result of an operation is available, but a new operation can be initiated every cycle. Thus, for example, each of three consecutive instructions can hold two load operations, resulting in six loads in various stages of execution at the same time.

Finally, the last five columns show which instruction slots can be used for which functional unit. For example, floating-point compare operations must appear only in the third slot of an instruction

Unit	Description	#	Lat.	1	2	3	4	5
Constant	Immediate operations	5	1	x	x	x	x	x
Integer ALU	32-Bit arithmetic, Boolean ops	5	1	x	x	x	x	x
Shifter	Multibit shifts	2	1	x	x	x	x	x
Load/Store	Memory operations	2	3				x	x
Int/FP MUL	32-Bit integer and FP multiplies	2	3		x	x		
FP ALU	FP arithmetic	2	3	x			x	
FP compare	FP compares	1	1			x		
FP sqrt/div	FP division and square root	1	17		x			
Branch	Control flow	3	3		x	x	x	
DSP ALU	Dual 16-bit, quad 8-bit multimedia arithmetic	2	3	x		x		x
DSP MUL	Dual 16-bit, quad 8-bit multimedia multiplies	2	3		x	x		

Figure 8-4. The TM3260 functional units, their quantity, latency, and which instruction slots they can use.

The constant unit is used for immediate operations, such as loading a number stored in the operation itself into a register. The integer ALU does addition, subtraction, the usual Boolean operations, and pack/unpack operations. The shifter can shift a register in either direction a specified number of bits.

The load/store unit fetches memory words into registers and writes them back. The TriMedia is basically an augmented RISC CPU, so normal operations operate on registers and the load/store unit is used to access memory. Transfers can be 8, 16, or 32 bits. Arithmetic and logical instructions do not access memory.

The multiply unit handles both integer and floating-point multiplications. The next three units handle floating-point additions/subtractions, compares, and square roots and divisions, respectively.

Branch operations are executed by the branch unit. There is a fixed 3-cycle delay after a branch, so the three instructions (up to 15 operations) following a branch are always executed, even for unconditional branches.

Finally, we come to the two multimedia units, which handle the special multimedia operations. The DSP in the name of the functional unit refers to **Digital Signal Processor**. which the multimedia operations effectively replace. We will describe the multimedia operations briefly below. One noteworthy feature is that they all use **saturated arithmetic** instead of two's complement arithmetic used by the integer operations. When an operation produces a result that cannot be expressed due to overflow, instead of generating an exception or giving a garbage result, the closest valid number is used. For example, with 8-bit unsigned numbers, adding 130 and 130 gives 255.

Because not every operation can appear in every slot, often an instruction does not contain all five potential operations. When a slot is not used, it is compacted to minimize the amount of space wasted. Operations that are present occupy 26, 34, or 42 bits. Depending on the number of operations actually present, TriMedia instructions vary from 2 to 28 bytes, including some fixed overhead.

The TriMedia does not make run-time checks to see whether the operations in an instruction are compatible. If they are not, it just executes them anyway and gets the wrong answer. Leaving the check out was a deliberate decision to save time and transistors. The Core i7 does do run-time checking to make sure all the superscalar operations are compatible, but at a huge cost in complexity, time, and transistors. The TriMedia avoids this expense by putting the burden of scheduling on the compiler, which has all the time in the world to carefully optimize the placement of operations in instruction words. On the other hand, if an operation needs a functional unit that is not available, the instruction will stall until it becomes available.

As in the Itanium-2, TriMedia operations are predicated. Each operation (with two minor exceptions) specifies a register that is tested before the operation is executed. If the low-order bit of the register is set, the operation is executed; otherwise, it is skipped. Each of the (up to) five operations is individually predicated. An example of a predicated operation is

```
IF R2 IADD R4, R5 –> R8
```

which tests R2 and, if the low-order bit is 1, adds R4 to R5 and stores the result in R8. An operation can be made unconditional by using R1 (which is always 1) as the predicate register. Using R0 (which is always 0) makes it a no-op.

The TriMedia multimedia operations can be grouped into the 15 groups listed in Fig. 8-5. Many of the operations involve clipping, which specifies an operand

and a range and forces the operand into the range, using the lowest or highest values for operands that fall outside it. Clipping can be done on 8-, 16-, or 32-bit operands. For example, when clipping is performed with a range of 0 to 255 on 40 and 340, the clipped results are 40 and 255, respectively. The clip group performs clip operations.

Group	Description
Clip	Clip 4 bytes or 2 half words
DSP absolute value	Clipped, signed, absolute value
DSP add	Clipped signed addition
DSP subtract	Clipped signed subtraction
DSP multiply	Clipped signed multiplication
Min, max	Get minimum or maximum of four byte pairs
Compare	Bytewise compare of two registers
Shift	Shift a pair of 16-bit operands
Sum of products	Signed sum of 8- or 16-bit products
Merge, pack, swap	Byte and half word manipulation
Byte quad averages	Unsigned byte-wise quad averaging
Byte averages	Unsigned byte-wise average of four elements
Byte multiplies	Unsigned 8-bit multiply
Motion estimation	Unsigned sum of absolute values of signed 8-bit diffs
Miscellaneous	Other arithmetic operations

Figure 8-5. The major groups of TriMedia custom operations.

The next four groups in Fig. 8-5 perform the indicated operation on operands of various sizes, clipping the results into a specific range. The min, max group examines two registers and for each byte finds the smallest or largest value. Similarly, the compare group regards two registers as four pairs of bytes and compares each pair.

Multimedia operations are rarely performed on 32-bit integers because most images are composed of RGB pixels with 8-bit values for each of the red, green, and blue colors. When an image is being processed (e.g., compressed), it is normally represented by three components, one for each color (RGB space) or a logically equivalent form (YUV space, discussed later in this chapter). Either way, a lot of processing is done on rectangular arrays containing 8-bit unsigned integers.

The TriMedia has a large number of operations specifically designed for processing arrays of 8-bit unsigned integers efficiently. As a simple example, consider the upper left-hand corner of an array of 8-bit values stored in (big-endian) memory as illustrated in Fig. 8-6(a). The 4×4 block shown in the corner contains 16 8-bit values labeled *A* through *P*. Suppose, for example, that the image needs to be transposed, to produce Fig. 8-6(b). How can this task be achieved?

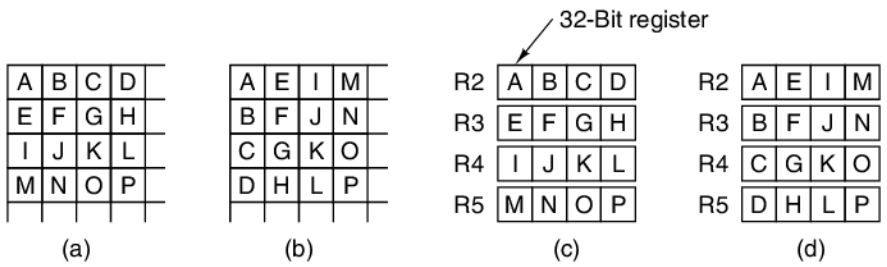

Figure 8-6. (a) An array of 8-bit elements. (b) The transposed array. (c) The original array fetched into four registers. (d) The transposed array in four registers.

One way to do the transposition is to use 12 operations that each load a byte into a different register, followed by 12 more operations that each store a byte in its correct location. (*Note*: the four bytes on the diagonal do not move in the transposition.) The problem with this approach is that it requires 24 (long and slow) operations that reference memory.

An alternative approach is to start with four operations that each load a word into four different registers, R2 through R5, as shown in Fig. 8-6(c). Then the four output words are assembled by masking and shifting operations to achieve the desired output, as shown in Fig. 8-6(d). Finally, these words are stored in memory. Although this way of doing it reduces the number of memory references from 24 to 8, the masking and shifting is expensive due to the many operations required to extract and insert each byte in the correct position.

The TriMedia provides a better solution than either of these. It begins by fetching the four words into registers. However, instead of building the output using masking and shifting, special operations that extract and insert bytes within registers are used to build the output. The result is that with eight memory references and eight of these special multimedia operations, the transposition can be accomplished. The code first contains an operation with two load operations in slots 4 and 5, respectively, to load words into R2 and R3, followed by another such operation to load R4 and R5.

The instructions holding these operations can use slots 1, 2, and 3 for other purposes. When all the words have been loaded, the eight special multimedia operations can be packed into two instructions to build the output, followed by two operations to store them. Only six instructions are needed, and 14 of the 30 slots in these instructions are available for other operations. In effect, the entire job can be done with the effective equivalent of about three or so instructions. Other multimedia operations are also efficient. Between these powerful operations and the five issue slots per instruction, the TriMedia is highly efficient at doing the kinds of calculations needed in multimedia processing.

8.1.2 On-Chip Multithreading

All modern, pipelined CPUs have an inherent problem: when a memory reference misses the level 1 and level 2 caches, there is a long wait until the requested word (and its associated cache line) are loaded into the cache, so the pipeline stalls. One approach to dealing with this situation, called **on-chip multithreading**, allows the CPU to manage multiple threads of control at the same time in an attempt to mask these stalls. In short, if thread 1 is blocked, the CPU still has a chance of running thread 2 in order to keep the hardware fully occupied.

Although the basic idea is fairly simple, multiple variants exist, which we will now examine. The first approach, called **fine-grained multithreading**, is illustrated in Fig. 8-7 for a CPU with the ability to issue one instruction per clock cycle. In Fig. 8-7(a)–(c), we see three threads, *A*, *B*, and *C*, for 12 machine cycles. During the first cycle, thread *A* executes instruction *A1*. This instruction completes in one cycle, so in the second cycle instruction *A2* is started. Unfortunately, this instruction misses on the level 1 cache so two cycles are wasted while the word needed is fetched from the level 2 cache. The thread continues in cycle 5. Similarly, threads *B* and *C* also stall occasionally as well, as illustrated in the figure. In this model if an instruction stalls, subsequent instructions cannot be issued. Of course, with a more sophisticated scoreboard, sometimes new instructions can still be issued, but we will ignore that possibility in this discussion.

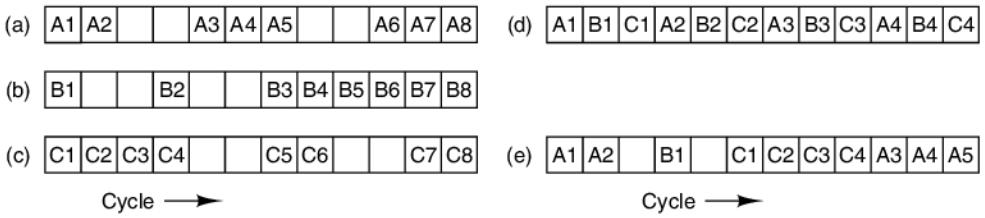

Figure 8-7. (a)–(c) Three threads. The empty boxes indicate that the thread has stalled waiting for memory. (d) Fine-grained multithreading. (e) Coarse-grained multithreading.

Fine-grained multithreading masks the stalls by running the threads round robin, with a different thread in consecutive cycles, as shown in Fig. 8-7(d). By the time the fourth cycle comes up, the memory operation initiated in *A1* has completed, so instruction *A2* can be run, even if it needs the result of *A1*. In this case the maximum stall is two cycles, so with three threads the stalled operation always completes in time. If a memory stall took four cycles, we would need four threads to insure continuous operation, and so on.

Since different threads have nothing to do with one another, each one needs its own set of registers. When an instruction is issued, a pointer to its register set has to be included along with the instruction so that if a register is referenced, the

hardware will know which register set to use. Consequently, the maximum number of threads that can be run at once is fixed at the time the chip is designed.

Memory operations are not the only reason for stalling. Sometimes an instruction needs a result computed by a previous instruction that is not yet complete. Sometimes an instruction cannot start because it follows a conditional branch whose direction is not yet known. In general, if the pipeline has k stages but there are at least k threads to run round robin, there will never be more than one instruction per thread in the pipeline at any moment, so no conflicts can occur. In this situation, the CPU can run at full speed, never stalling.

Of course, there may not be as many threads available as there are pipeline stages, so some designers prefer a different approach, known as **coarse-grained multithreading**, illustrated in Fig. 8-7(e). Here thread *A* starts and continues to issue instructions until it stalls, wasting one cycle. At that point a switch occurs and *B1* is executed. Since the first instruction of thread *B* stalls, another thread switch happens and *C1* is executed in cycle 6. Since a cycle is lost whenever an instruction stalls, coarse-grained multithreading is potentially less efficient than fine-grained multithreading, but it has the big advantage that many fewer threads are needed to keep the CPU busy. In situations with an insufficient number of active threads, to be sure of finding a runnable one, coarse-grained multithreading works better.

Although we have depicted coarse-grained multithreading as doing thread switches on a stall, that is not the only option. Another possibility is to switch immediately on any instruction that might cause a stall, such as a load, store, or branch, before even finding out whether it does cause a stall. This allows a switch to occur earlier (as soon as the instruction is decoded), and may make it possible to avoid dead cycles. In effect, it is saying: "Run until there might be a problem, then switch just in case." Doing so makes coarse-grained multithreading somewhat more like fine-grained multithreading with its frequent switches.

No matter which kind of multithreading is used, some way is needed to keep track of which operation belongs to which thread. With fine-grained multithreading, the only serious possibility is to attach a thread identifier to each operation, so that as it moves through the pipeline, its identity is clear. With coarse-grained multithreading, another possibility exists: when switching threads, let the pipeline clear and only then start the next thread. In that way, only one thread at a time is in the pipeline and its identity is never in doubt. Of course, letting the pipeline run dry on a thread switch makes sense only if thread switches take place at intervals very much longer than the time it takes to empty the pipeline.

So far we have assumed that the CPU can issue only one instruction per cycle. As we have seen, however, modern CPUs can issue multiple instructions per cycle. In Fig. 8-8 we assume the CPU can issue two instructions per clock cycle, but we maintain the rule that when an instruction stalls, subsequent instructions cannot be issued. In Fig. 8-8(a) we see how fine-grained multithreading works with a dual-issue superscalar CPU. For thread *A*, the first two instructions can be issued in the

first cycle, but for thread *B* we immediately hit a problem in the next cycle, so only one instruction can be issued, and so on.

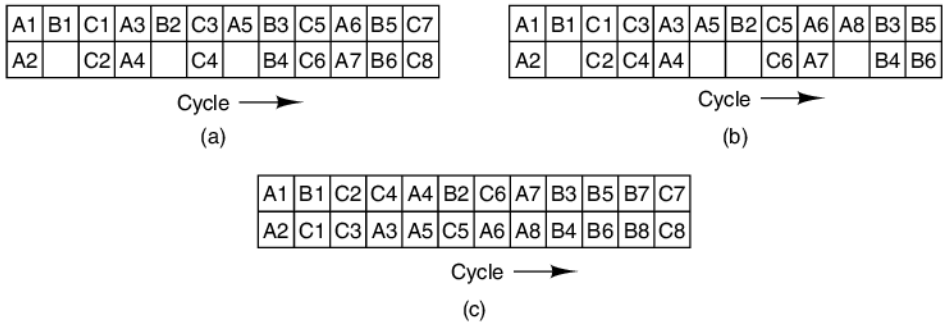

Figure 8-8. Multithreading with a dual-issue superscalar CPU. (a) Fine-grained multithreading. (b) Coarse-grained multithreading. (c) Simultaneous multithreading.

In Fig. 8-8(b), we see how coarse-grained multithreading works with a dual-issue CPU, but now with a static scheduler that does not introduce a dead cycle after an instruction that stalls. Basically, the threads are run in turn, with the CPU issuing two instructions per thread until it hits one that stalls, at which point it switches to the next thread at the start of the next cycle.

With superscalar CPUs, a third possible way of doing multithreading is available, called **simultaneous multithreading** and illustrated in Fig. 8-8(c). This approach can be seen as a refinement to coarse-grained multithreading, in which a single thread is allowed to issue two instructions per cycle as long as it can, but when it stalls, instructions are immediately taken from the next thread in sequence to keep the CPU fully occupied. Simultaneous multithreading can also help keep all the functional units busy. When an instruction cannot be started because a functional unit it needs is occupied, an instruction from a different thread can be chosen instead. In this figure, we are assuming that *B8* stalls in cycle 11, so *C7* is started in cycle 12.

For more information about multithreading, see Gebhart et al. (2011) and Wing-kei et al. (2011).

Hyperthreading on the Core i7

Having looked at multithreading in the abstract, let us now consider a practical example: the Core i7. In the early 2000s, processors such as the Pentium 4 were not delivering the performance boosts that Intel needed to keep up sales. After the Pentium 4 was already in production, the architects at Intel looked for various ways to speed it up without changing the programmers' interface, something that would never have been accepted. Five ways quickly popped up:

1. Increasing the clock speed.
2. Putting two CPUs on a chip.
3. Adding functional units.
4. Making the pipeline longer.
5. Using multithreading.

An obvious way to improve performance is to increase the clock speed without changing anything else. Doing this is relatively straightforward and well understood, so each new chip that comes out is generally slightly faster than its predecessor. Unfortunately, a faster clock also has two main drawbacks that limit how great an increase can be tolerated. First, a faster clock uses more energy, which is a huge problem for notebook computers and other battery-powered devices. Second, the extra energy input means the chip gets hotter and there is more heat to dissipate.

Putting two CPUs on a chip is relatively straightforward, but it comes close to doubling the chip area if each one has its own caches and thus reduces the number of chips per wafer by a factor of two, which essentially doubles the unit manufacturing cost. If the two chips share a common cache as big as the original one, the chip area is not doubled, but cache size per CPU is halved, cutting into performance. Also, while high-end server applications can often fully utilize multiple CPUs, not all desktop applications have enough inherent parallelism to warrant two full CPUs.

Adding additional functional units is also fairly easy, but it is important to get the balance right. Having 10 ALUs does little good if the chip is incapable of feeding instructions into the pipeline fast enough to keep them all busy.

A longer pipeline with more stages, each doing a smaller piece of work in a shorter time period increases performance but also increases the negative effects of branch mispredictions, cache misses, interrupts, and other factors that interrupt normal pipeline flow. Furthermore, to take full advantage of a longer pipeline, the clock speed has to be increased, which means more energy is consumed and more heat is produced.

Finally, multithreading can be added. Its value is in having a second thread utilize hardware that would otherwise have lain fallow. After some experimentation, it became clear that a 5% increase in chip area for multithreading support gave a 25% performance gain in many applications, making this a good choice. Intel's first multithreaded CPU was the Xeon in 2002, but multithreading was later added to the Pentium 4, starting with the 3.06-GHz version and continuing with faster versions of the Pentium processor, including the Core i7. Intel calls the implementation of multithreading used in its processors **hyperthreading**.

The basic idea is to allow two threads (or possibly processes, since the CPU cannot tell what is a thread and what is a process) to run at once. To the operating

system, a hyperthreaded Core i7 chip looks like a dual processor in which both CPUs happen to share a common cache and main memory. The operating system schedules the threads independently. If two applications are running at the same time, the operating system can run each one at the same time. For example, if a mail daemon is sending or receiving email in the background while a user is interacting with some program in the foreground, the daemon and the user program can be run in parallel, as though there were two CPUs available.

Application software that has been designed to run as multiple threads can use both virtual CPUs. For example, video editing programs usually allow users to specify certain filters to apply to each frame in some range. These filters can modify the brightness, contrast, color balance, or other properties of each frame. The program can then assign one CPU to process the even-numbered frames and the other to process the odd-numbered frames. The two can then run in parallel.

Since the two threads share all the hardware resources, a strategy is needed to manage the sharing. Intel identified four useful strategies for resource sharing in conjunction with hyperthreading: resource duplication, partitioned resources, threshold sharing, and full sharing. We will now touch on each of these in turn.

To start with, some resources are duplicated just for threading. For example, since each thread has its own flow of control, a second program counter had to be added. The table that maps the architectural registers (`EAX`, `EBX`, etc.) onto the physical registers also had to be duplicated, as did the interrupt controller, since the threads can be independently interrupted.

Next we have **partitioned resource sharing**, in which the hardware resources are rigidly divided between the threads. For example, if the CPU has a queue between two functional pipeline stages, half the slots could be dedicated to thread 1 and the other half to thread 2. Partitioning is easy to accomplish, has no overhead, and keeps the threads out of each other's hair. If all the resources are partitioned, we effectively have two separate CPUs. On the down side, it can easily happen that at some point one thread is not using some of its resources that the other one wants but is forbidden from accessing. As a consequence, resources that could have been used productively lie idle.

The opposite of partitioned sharing is **full resource sharing**. When this scheme is used, either thread can acquire any resources it needs, first come, first served. However, imagine a fast thread consisting primarily of additions and subtractions and a slow thread consisting primarily of multiplications and divisions. If instructions are fetched from memory faster than multiplications and divisions can be carried out, the backlog of instructions fetched for the slow thread and queued but not yet fed into the pipeline will grow in time.

Eventually, this backlog will occupy the entire instruction queue, bringing the fast thread to a halt for lack of space in the instruction queue. Full resource sharing solves the problem of a resource lying idle while another thread wants it, but creates a new problem of one thread potentially hogging so many resources that it slows the other one down or stops it altogether.

An intermediate scheme is **threshold sharing**, in which a thread can acquire resources dynamically (no fixed partitioning) but only up to some maximum. For resources that are replicated, this approach allows flexibility without the danger that one thread will starve due to its inability to acquire any of the resource. If, for example, no thread can acquire more than 3/4 of the instruction queue, no matter what the slow thread does, the fast thread will be able to run. The Core i7 hyperthreading uses different sharing strategies for different resources in an attempt to address the various problems alluded to above. Duplication is used for resources that each thread requires all the time, such as the program counter, register map, and interrupt controller. Duplicating these resources increases the chip area by only 5%, a modest price to pay for multithreading. Resources available in such abundance that there is no danger of one thread capturing them all, such as cache lines, are fully shared in a dynamic way. On the other hand, resources that control the operation of the pipeline, such as the various queues within the pipeline, are partitioned, giving each thread half of the slots. The main pipeline of the Sandy Bridge microarchitecture used in the Core i7 is illustrated in Fig. 8-9, with the white and gray boxes indicating how the resources are allocated between the white and gray threads.

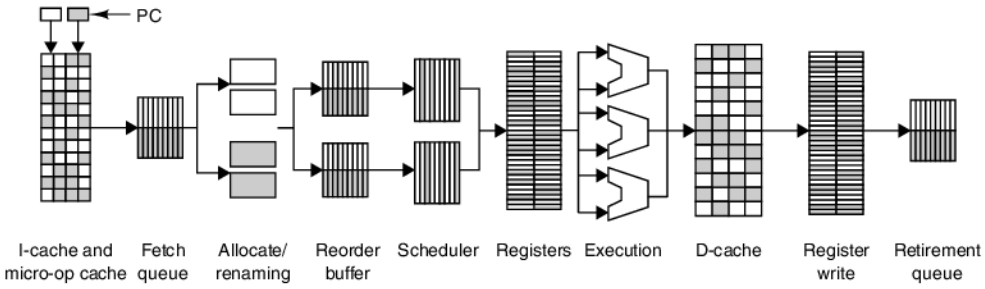

Figure 8-9. Resource sharing between threads in the Core i7 microarchitecture.

In this figure we can see that all the queues are partitioned, with half the slots in each queue reserved for each thread. In this one, neither thread can choke off the other. The register allocator and renamer is also partitioned. The scheduler is dynamically shared, but with a threshold, to prevent either thread from claiming all of the slots. The remaining pipeline stages are fully shared.

All is not sweetness and light with multithreading, however. There is also a downside. While partitioning is cheap, dynamic sharing of any resource, especially with a limit on how much a thread can take, requires bookkeeping at run time to monitor usage. In addition, situations can arise in which programs work much worse with multithreading than without it. For example, imagine two threads that each need 3/4 of the cache to function well. Run separately, each one works fine and has few (expensive) cache misses. Run together, each one has numerous cache misses and the net result may be far worse than without multithreading.

More information about multithreading and its implementation inside Intel processors is given in Gerber and Binstock (2004) and Gepner et al. (2011).

8.1.3 Single-Chip Multiprocessors

While multithreading provides significant performance gains at modest cost, for some applications a much larger performance gain is needed. To get this gain, multiprocessor chips are being developed. Two areas where these chips, which contain two or more CPUs, are of interest are high-end servers and in consumer electronics. We will now briefly touch on each of them.

Homogeneous Multiprocessors on a Chip

With advances in VLSI technology, it is now possible to put two or more powerful CPUs on a single chip. Since these CPUs often share the same level 2 cache and main memory, they qualify as a multiprocessor, as discussed in Chap. 2. A typical application area is a large Web server farm consisting of many servers. By putting two CPUs in the same box, sharing not only memory but also disks and network interfaces, the performance of the server can often be doubled without doubling the cost (because even at twice the price, the CPU chip is only a fraction of the total system cost).

For small-scale single-chip multiprocessors, two designs are prevalent. In the first one, shown in Fig. 8-10(a), there is really only one chip, but it has a second pipeline, potentially doubling the instruction execution rate. In the second one, shown in Fig. 8-10(b), there are separate cores on the chip, each containing a full CPU. A **core** is a large circuit, such as a CPU, I/O controller, or cache, that can be placed on a chip in a modular way, usually next to other cores.

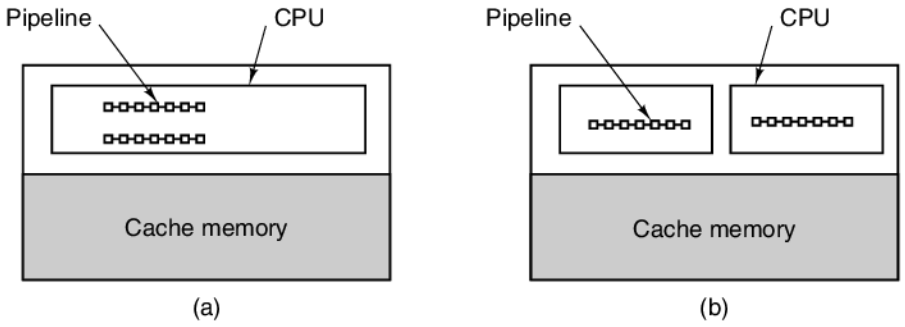

Figure 8-10. Single-chip multiprocessors. (a) A dual-pipeline chip. (b) A chip with two cores.

The former design allows resources, such as functional units, to be shared between the processors, thus allowing one CPU to use resources the other does not

need. On the other hand, this approach requires redesigning the chip and it does not scale well much above two CPUs. In contrast, just putting two or more CPU cores on the same chip is relatively easy to do.

We will discuss multiprocessors later in the chapter. While that discussion is somewhat focused on multiprocessors built from single-CPU chips, much of it applies to multi-CPU chips as well.

The Core i7 Single-Chip Multiprocessor

The Core i7 CPU is a single-chip multiprocessor that is manufactured with four or more cores on a single silicon die. The high-level organization of a Core i7 processor is illustrated in Fig. 8-11.

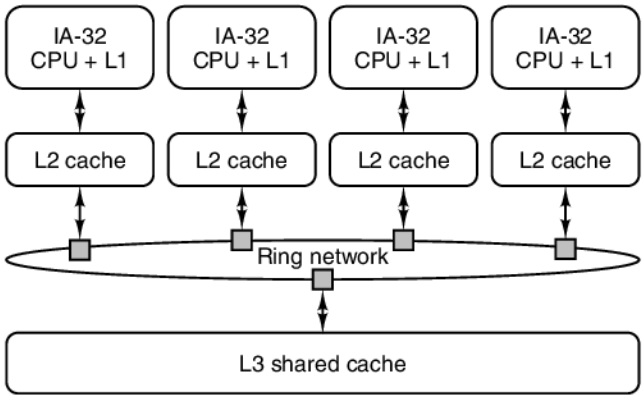

Figure 8-11. Single-chip multiprocessor architecture of the Core i7.

Each processor in the Core i7 has its own private L1 instruction and data caches, plus its own private L2 unified cache. The processors are connected to the private caches with dedicated point-to-point connections. The next level of the memory hierarchy is the shared and unified L3 data cache.

The L2 caches connect to the L3 shared cache using a **ring network**. When a communication request enters the ring network, it is forwarded to the next node on the network, where it is checked to see if it has reached its destination node. This process continues from node to node on the ring until the destination node is found or the request arrives at its source again (in which case the destination does not exist). The advantage of a ring network is it is a cheap way to get high bandwidth, at the cost of increased latency as requests hop from node to node. The Core i7 ring network serves two primary purposes. First, it provides a way to move memory and I/O requests between the caches and processors. Second, it implements the checks necessary to ensure that each processor is always seeing a coherent view of memory. We will learn more about these coherence checks later in this chapter.

Heterogeneous Multiprocessors on a Chip

A completely different application area calling for single-chip multiprocessors is embedded systems, especially in audiovisual consumer electronics, such as television sets, DVD players, camcorders, game consoles, cell phones, and so on. These systems have demanding performance requirements and tight constraints. Although these devices look different, more and more of them are simply small computers, with one or more CPUs, memories, I/O controllers, and some custom I/O devices. A cell phone, for example, is merely a PC with a CPU, memory, dwarf keyboard, microphone, loudspeaker, and a wireless network connection in a small package.

Consider, as an example, a portable DVD player. The computer inside has to handle the following functions:

1. Control of a cheap, unreliable servomechanism for head tracking.
2. Analog-to-digital conversion.
3. Error correction.
4. Decryption and digital rights management.
5. MPEG-2 video decompression.
6. Audio decompression.
7. Encoding the output for NTSC, PAL, or SECAM television sets.

This work must be done subject to stringent real-time, quality-of-service, energy, heat-dissipation, size, weight, and price constraints.

CD, DVD, and Blu-ray disks contain a long spiral containing the information, as illustrated in Fig. 2-25 (for a CD). In this section we will discuss DVDs since they are still more common than Blu-ray disks, but Blu-ray disks are very similar to DVDs except they use MPEG-4 instead of MPEG-2 for encoding. With all optical media, the read head must accurately track the spiral as the disk rotates. The price is kept low by using a relatively simple mechanical design and tight control over the head position in software. The signal coming off the head is an analog signal, which must be converted to digital form before being processed. After it has been digitized, heavy error correction is required because DVDs are pressed and contain many errors, which must corrected in software. The video is compressed using the MPEG-2 international standard, which requires complex (Fourier-transform-like) computations for decompression. Audio is compressed using a psycho-acoustic model, which also requires sophisticated calculations for decompression. Finally, audio and video have to be rendered in a suitable form for output to NTSC, PAL, or SECAM television sets, depending on the country to which the DVD player is shipped. It should come as no surprise that doing all this work in real time in software on a cheap general-purpose CPU is not possible.

What is needed is a heterogeneous multiprocessor containing multiple cores, each specialized for one particular task. An example DVD player is given in Fig. 8-12.

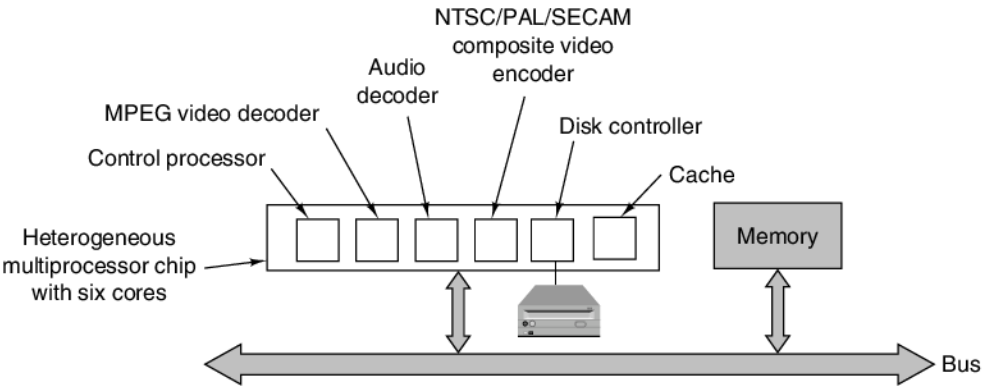

Figure 8-12. The logical structure of a simple DVD player contains a heterogeneous multiprocessor containing multiple cores for different functions.

The functions of the cores in Fig. 8-12 are all different, with each being carefully designed to be extremely good at what it does for the lowest possible price. For example, DVD video is compressed using a scheme known as **MPEG-2** (after the **Motion Picture Experts Group** that invented it). It consists of dividing each frame up into blocks of pixels and doing a complex transformation on each one. A frame can consist entirely of transformed blocks or it can specify that a certain block is the same as one found in the previous frame but located at an offset of $(\Delta x, \Delta y)$ from its current position except with a couple of pixels changed. Doing this calculation in software is extremely slow, but it is possible to build an MPEG-2 decoding engine that can do it in hardware quite rapidly. Similarly, audio decoding and reencoding the composite audio-video signal to conform to one of the world's television standards can be done better by dedicated hardware processors. These observations quickly lead to heterogeneous multiprocessor chips containing multiple cores specifically designed for audio-visual applications. However, because the control processor is a general-purpose programmable CPU, the multiprocessor chip can also be used in other, similar applications, such as a DVD recorder.

Another device requiring a heterogeneous multiprocessor is the engine inside an advanced cell phone. Current phones sometimes have still cameras, video cameras, game machines, web browsers, email readers, and digital satellite radio receivers, using either cell-phone technology (CDMA or GSM, depending on the country) or wireless Internet (IEEE 802.11, also called WiFi) built in; future ones may include all of these. As devices take on more and more functionality, with watches becoming GPS-based maps and eyeglasses becoming radios, the need for heterogeneous multiprocessors will only increase.

Fairly soon, large chips will have tens of billions of transistors. Such chips are far too large to design one gate and one wire at a time. The human effort required would render the chips obsolete by the time they were finished. The only feasible approach is to use cores (essentially libraries) containing fairly large subassemblies and to place and interconnect them on the chip as needed. Designers then have to determine which CPU core to use for the control processor and which special-purpose processors to throw in to help it. Putting more of the burden on software running on the control processor makes the system slower but yields a smaller (and cheaper) chip. Having multiple special-purpose processors for audio and video processing takes up chip area, increasing the cost, but produces higher performance at a lower clock rate, which means lower power consumption and less heat dissipation. Thus chip designers increasingly contend with these macroscopic trade-offs rather than worrying about where to place each transistor.

Audiovisual applications are very data intensive. Huge amounts of data have to be processed quickly, so typically 50% to 75% of the chip area is devoted to memory in one form or another, and the amount is rising. The design issues here are numerous. How many levels of cache should be used? Should the cache(s) be split or unified? How big should each cache be? How fast should each be? Should some actual memory go on the chip, too? Should it be SRAM or SDRAM? The answers to each of these questions have major implications for the performance, energy consumption, and heat dissipation of the chip.

Besides design of the processors and memory system, another issue of considerable consequence is the communication system—how do all the cores communicate with each other? For small systems, a single bus will usually do the trick, but for larger ones it rapidly becomes a bottleneck. Often the problem can be solved by going to multiple buses or possibly a ring from core to core. In the latter case, arbitration is handled by passing a small packet called a **token** around the ring. To transmit, a core must first capture the token. When it is done, it puts the token back on the ring so it can continue circulating. This protocol prevents collisions on the ring.

As an example of an on-chip interconnect, look at the IBM **CoreConnect**, illustrated in Fig. 8-13. It is an architecture for connecting cores on a single-chip heterogeneous multiprocessor, especially complete system-on-a-chip designs. In a sense, CoreConnect is to one-chip multiprocessors what the PCI bus was to the Pentium—the glue that holds all the parts together. (With modern Core i7 systems, PCIe is the glue, but it is a point-to-point network without a shared bus like PCI.) However, unlike the PCI bus, CoreConnect was designed without any requirements to be backward compatible with legacy equipment or protocols and without the constraints of board-level buses, such as limits on the number of pins the edge connector can have.

CoreConnect consists of three buses. The **processor bus** is a high-speed, synchronous, pipelined bus with 32, 64, or 128 data lines clocked at 66, 133, or 183 MHz. The maximum throughput is thus 23.4 Gbps (vs. 4.2 Gbps for the PCI bus).

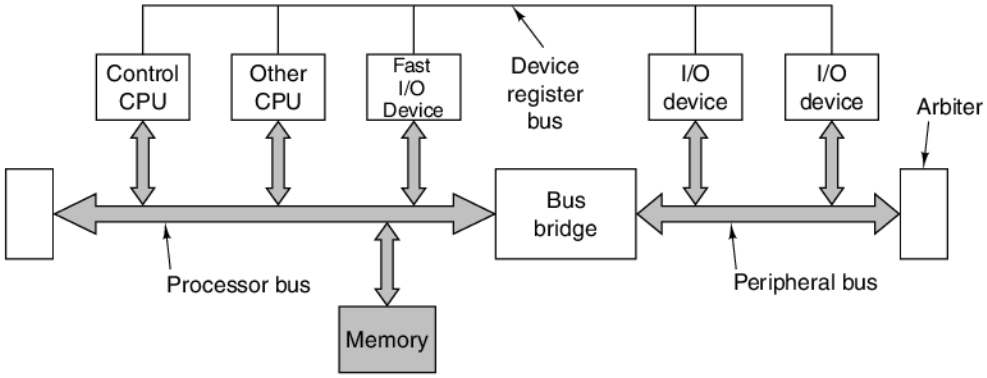

Figure 8-13. An example of the IBM CoreConnect architecture.

The pipelining features allow cores to request the bus while a transfer is going on, and allow different cores to use different lines at the same time, similar to the PCI bus. The processor bus is optimized for short block transfers. It is intended to connect fast cores, such as CPUs, MPEG-2 decoders, high-speed networks, and similar items.

Stretching the processor bus over the entire chip would reduce its performance, so a second bus is present for low-speed I/O devices, such as UARTs, timers, USB controllers, serial I/O devices, and so forth. This **peripheral bus** has been designed to simplify interfacing 8-, 16-, and 32-bit peripherals to it using no more than a few hundred gates. It, too, is a synchronous bus, with a maximum throughput of 300 Mbps. The two buses are connected by a bridge, not unlike the bridges that were used to connect the PCI and ISA buses in PCs until the ISA bus was phased out a number of years ago.

The third bus is the **device register bus**, a very low-speed, asynchronous, handshaking bus used to allow the processors to access the device registers of all the peripherals in order to control the corresponding devices. It is intended for infrequent transfers of only a few bytes at a time.

By providing a standard on-chip bus, interface, and framework, IBM hopes to create a miniature version of the PCI world, in which many manufacturers produce processors and controllers that plug together easily. One difference, however, is that in the PCI world the manufacturers produce and sell actual boards that PC vendors and end users buy. In the CoreConnect world, third parties design cores but do not manufacture them. Instead, they license them as intellectual property to consumer electronics and other companies, which then design custom heterogeneous multiprocessor chips based on their own and licensed third-party cores. Since manufacturing such large and complex chips requires a massive investment

in fabrication facilities, in most cases the consumer electronics company just does the design, subcontracting the chip manufacturing out to a semiconductor vendor. Cores exist for numerous CPUs (ARM, MIPS, PowerPC, etc.) as well as for MPEG decoders, digital signal processors, and all the standard I/O controllers.

The IBM CoreConnect is not the only popular on-chip bus on the market. The **AMBA** (**Advanced Microcontroller Bus Architecture**) is also widely used to connect ARM CPUs to other CPUs and I/O devices (Flynn, 1997). Other, somewhat less popular on-chip buses are the **VCI** (**Virtual Component Interconnect**) and **OCP-IP** (**Open Core Protocol-International Partnership**), which are also competing for market share (Bhakthavatchalu et al., 2010). On-chip buses are only the start; people are now putting complete networks on a chip (Ahmadinia and Shahrabi, 2011).

With chip manufacturers having increasing difficulty in raising clock frequencies due to heat-dissipation problems, single-chip multiprocessors are a very hot topic. More information can be found in Gupta et al. (2010), Herrero et al. (2010), and Mishra et al. (2011).

8.2 COPROCESSORS

Having examined some ways of achieving on-chip parallelism, let us now move up a step and look at how the computer can be speeded up by adding a second, specialized processor. These **coprocessors** come in many varieties, from small to large. On the IBM 360 mainframes and all of their successors, independent I/O channels exist for doing input/output. Similarly, the CDC 6600 had 10 independent processors for doing I/O. Graphics and floating-point arithmetic are other areas where coprocessors have been used. Even a DMA chip can be seen as a coprocessor. In some cases, the CPU gives the coprocessor an instruction or set of instructions and tells it to execute them; in other cases, the coprocessor is more independent and runs pretty much on its own.

Physically, coprocessors can range from a separate cabinet (the 360 I/O channels) to a plug-in board (network processors) to an area on the main chip (floating-point). In all cases, what distinguishes them is the fact that some other processor is the main processor and the coprocessors are there to help it. We will now examine three areas where speed-ups are possible: network processing, multimedia, and cryptography.

8.2.1 Network Processors

Most computers nowadays are connected to a network or to the Internet. As a result of technological progress in network hardware, networks are now so fast that it has become increasingly difficult to process all the incoming and outgoing data in software. As a consequence, special network processors have been developed to

handle the traffic, and many high-end computers now have one of these processors. In this section we will first give a brief introduction to networking and then discuss how network processors work.

Introduction to Networking

Computer networks come in two general types: **local-area networks**, or **LAN**s, which connect multiple computers within a building or campus, and **wide-area networks** or **WAN**s, which connect computers spread over a large geographic area. The most popular LAN is called **Ethernet**. The original Ethernet consisted of a fat cable into which a wire coming from each computer was forcibly inserted using what was euphemistically referred to a **vampire tap**. Modern Ethernets have the computers attached to a central switch, as illustrated in the right-hand portion of Fig. 8-14. The original Ethernet crawled along at 3 Mbps, but the first commercial version was 10 Mbps. It was eventually replaced by fast Ethernet at 100 Mbps and then by gigabit Ethernet at 1 Gbps. A 10-gigabit Ethernet is already on the market and a 40-gigabit Ethernet is in the pipeline.

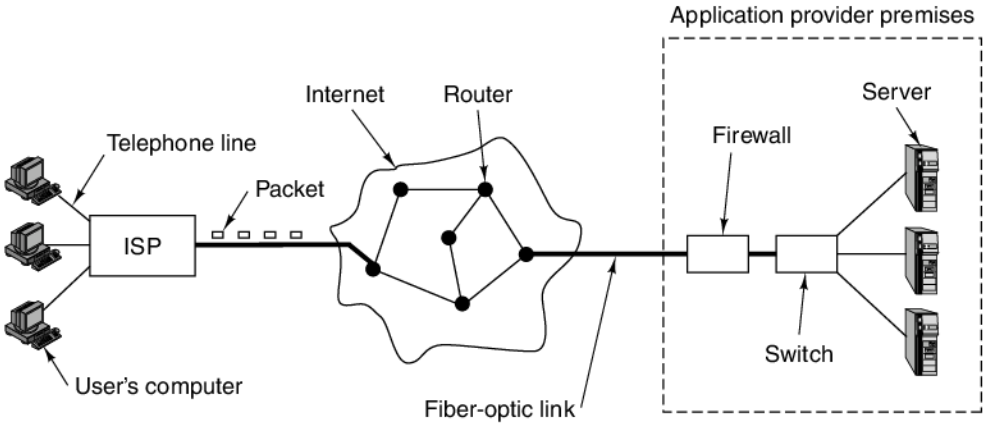

Figure 8-14. How users are connected to servers on the Internet.

WANs are organized differently. They consist of specialized computers called **routers** connected by wires or optical fibers, as shown in the middle of Fig. 8-14. Chunks of data called **packets**, typically 64 to about 1500 bytes, are moved from the source machine through one or more routers until they reach their destination. At each hop, a packet is stored in the router's memory and then forwarded to the next router along the path as soon as the needed transmission line is available. This technique is called **store-and-forward packet switching**.

Although many people think of the Internet as a single WAN, technically it is a collection of many WANs connected together. However, for our purposes, that distinction is not important. Figure 8-14 gives a bird's-eye view of the Internet from the perspective of a home user. The user's computer is typically connected to a