for this data type by using the indirect addressing mode. The operands in the instructions using this mode are actually pointers. In an instruction using a pointer, the operand is essentially an address and must be treated as such.

5.4.2 Address Modes

We saw in Chapter 4 that the 12 bits in the operand field of a MARIE instruction can be interpreted in two different ways: the 12 bits represent either the memory address of the operand or a pointer to a physical memory address. These 12 bits can be interpreted in many other ways, thus providing us with several different *addressing modes*. Addressing modes allow us to specify where the instruction operands are located. An addressing mode can specify a constant, a register, or a location in memory. Certain modes allow shorter addresses and some allow us to determine the location of the actual operand, often called the *effective address* of the operand, dynamically. We now investigate the most basic addressing modes.

Immediate Addressing

Immediate addressing is so-named because the value to be referenced immediately follows the operation code in the instruction. That is to say, the data to be operated on is part of the instruction. For example, if the addressing mode of the operand is immediate and the instruction is `Load 008`, the numeric value 8 is loaded into the AC. The 12 bits of the operand field do not specify an address—they specify the actual operand the instruction requires. Immediate addressing is very fast because the value to be loaded is included in the instruction. However, because the value to be loaded is fixed at compile time it is not very flexible.

Direct Addressing

Direct addressing is so-named because the value to be referenced is obtained by specifying its memory address directly in the instruction. For example, if the addressing mode of the operand is direct and the instruction is `Load 008`, the data value found at memory address 008 is loaded into the AC. Direct addressing is typically quite fast because, although the value to be loaded is not included in the instruction, it is quickly accessible. It is also much more flexible than immediate addressing because the value to be loaded is whatever is found at the given address, which may be variable.

Register Addressing

In *register addressing*, a register, instead of memory, is used to specify the operand. This is very similar to direct addressing, except that instead of a memory address, the address field contains a register reference. The contents of that register are used as the operand.

Indirect Addressing

Indirect addressing is a very powerful addressing mode that provides an exceptional level of flexibility. In this mode, the bits in the address field specify a memory address that is to be used as a pointer. The effective address of the operand is found by going to this memory address. For example, if the addressing mode of the operand is indirect and the instruction is `Load 008`, the data value found at memory address 008 is actually the effective address of the desired operand. Suppose we find the value 2A0 stored in location 008. 2A0 is the "real" address of the value we want. The value found at location 2A0 is then loaded into the AC.

In a variation on this scheme, the operand bits specify a register instead of a memory address. This mode, known as *register indirect addressing*, works exactly the same way as indirect addressing mode, except it uses a register instead of a memory address to point to the data. For example, if the instruction is `Load R1` and we are using register indirect addressing mode, we would find the effective address of the desired operand in R1.

Indexed and Based Addressing

In *indexed addressing* mode, an index register (either explicitly or implicitly designated) is used to store an offset (or displacement), which is added to the operand, resulting in the effective address of the data. For example, if the operand X of the instruction `Load X` is to be addressed using indexed addressing, assuming R1 is the index register and holds the value 1, the effective address of the operand is actually $X + 1$. *Based addressing* mode is similar, except a base address register, rather than an index register, is used. In theory, the difference between these two modes is in how they are used, not how the operands are computed. An index register holds an index that is used as an offset, relative to the address given in the address field of the instruction. A base register holds a base address, where the address field represents a displacement from this base. These two addressing modes are quite useful for accessing array elements as well as characters in strings. In fact, most assembly languages provide special index registers that are implied in many string operations. Depending on the instruction-set design, general-purpose registers may also be used in this mode.

Stack Addressing

If *stack addressing mode* is used, the operand is assumed to be on the stack. We have already seen how this works in Section 5.2.4.

Additional Addressing Modes

Many variations on the above schemes exist. For example, some machines have *indirect indexed addressing*, which uses both indirect and indexed addressing at the same time. There is also *base/offset addressing*, which adds an offset to a specific base register and then adds this to the specified operand, resulting in the effective address of the actual operand to be used in the instruction. There are also *auto-increment* and *auto-decrement* modes. These modes automatically increment or decrement the register used, thus reducing the code size, which can be extremely important in applications such as embedded systems. *Self-relative addressing* computes the address of the operand as an offset from the current instruction. Additional modes exist; however, familiarity with immediate, direct,

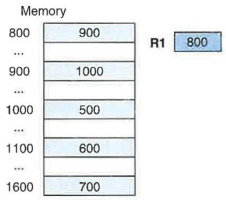


FIGURE 5.3 Contents of Memory When `Load 800` Is Executed

Mode	Value Loaded into AC
Immediate	800
Direct	900
Indirect	1000
Indexed	700

TABLE 5.1 Results of Using Various Addressing Modes on Memory in Figure 5.2

register, indirect, indexed, and stack addressing modes goes a long way in understanding any addressing mode you may encounter.

Let's look at an example to illustrate these various modes. Suppose we have the instruction `Load 800`, and the memory and register R1 shown in Figure 5.3.

Applying the various addressing modes to the operand field containing the 800, and assuming R1 is implied in the indexed addressing mode, the value actually loaded into AC is seen in Table 5.1.

The instruction `Load R1`, using register addressing mode, loads an 800 into the accumulator, and using register indirect addressing mode, loads a 900 into the accumulator.

We summarize the addressing modes in Table 5.2.

The various addressing modes allow us to specify a much larger range of locations than if we were limited to using one or two modes. As always, there are trade-offs. We sacrifice simplicity in address calculation and limited memory references for flexibility and increased address range.

5.5 INSTRUCTION-LEVEL PIPELINING

By now you should be reasonably familiar with the fetch-decode-execute cycle presented in Chapter 4. Conceptually, each pulse of the computer's clock is used to control one step in the sequence, but sometimes additional pulses can be used

Addressing Mode	To Find Operand
Immediate	Operand value present in the instruction
Direct	Effective address of operand in address field
Register	Operand value located in register
Indirect	Address field points to address of the actual operand
Register Indirect	Register contains address of actual operand
Indexed or Based	Effective address of operand generated by adding value in address field to contents of a register
Stack	Operand located on stack

TABLE 5.2 A Summary of the Basic Addressing Modes

to control smaller details within one step. Some CPUs break the fetch-decode-execute cycle down into smaller steps, where some of these smaller steps can be performed in parallel. This overlapping speeds up execution. This method, used by all current CPUs, is known as *pipelining*.

Suppose the fetch-decode-execute cycle were broken into the following "mini-steps":

1. Fetch instruction
2. Decode opcode
3. Calculate effective address of operands
4. Fetch operands
5. Execute instruction
6. Store result

Pipelining is analogous to an automobile assembly line. Each step in a computer pipeline completes a part of an instruction. Like the automobile assembly line, different steps are completing different parts of different instructions in parallel. Each of the steps is called a *pipeline stage*. The stages are connected to form a pipe. Instructions enter at one end, progress through the various stages, and exit at the other end. The goal is to balance the time taken by each pipeline stage (i.e., more or less the same as the time taken by any other pipeline stage). If the stages are not balanced in time, after awhile, faster stages will be waiting on slower ones. To see an example of this imbalance in real life, consider the stages of doing laundry. If you have only one washer and one dryer, you usually end up waiting on the dryer. If you consider washing as the first stage and drying as the next, you can see that the longer drying stage causes clothes to pile up between the two stages. If you add folding clothes as a third stage, you soon realize that this stage would consistently be waiting on the other, slower stages.

Figure 5.4 provides an illustration of computer pipelining with overlapping stages. We see each clock cycle and each stage for each instruction (where S1 represents the fetch, S2 represents the decode, S3 is the calculate state, S4 is the operand fetch, S5 is the execution, and S6 is the store).

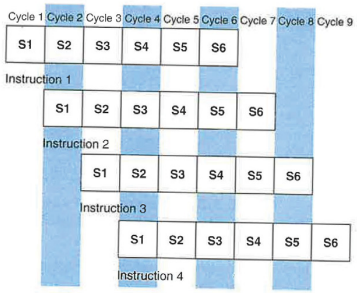


FIGURE 5.4 Four Instructions Going through a 6-Stage Pipeline

We see from Figure 5.4 that once instruction 1 has been fetched and is in the process of being decoded, we can start the fetch on instruction 2. When instruction 1 is fetching operands, and instruction 2 is being decoded, we can start the fetch on instruction 3. Notice these events can occur in parallel, very much like an automobile assembly line.

Suppose we have a k-stage pipeline. Assume the clock cycle time is t_p, that is, it takes t_p time per stage. Assume also we have n instructions (often called *tasks*) to process. Task 1 (T_1) requires $k \times t_p$ time to complete. The remaining $n - 1$ tasks emerge from the pipeline one per cycle, which implies a total time for these tasks of $(n - 1)t_p$. Therefore, to complete n tasks using a k-stage pipeline requires:

$$(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$$

or $k + (n - 1)$ clock cycles.

Let's calculate the speedup we gain using a pipeline. Without a pipeline, the time required is nt_n cycles, where $t_n = k \times t_p$. Therefore, the speedup (time without a pipeline divided by the time using a pipeline) is:

$$\text{speedup } S = \frac{nt_n}{(k + n - 1)t_p}$$

If we take the limit of this as n approaches infinity, we see that $(k + n - 1)$ approaches n, which results in a theoretical speedup of:

$$\text{Speedup} = \frac{k \times t_p}{t_p} = k$$

The theoretical speedup, k, is the number of stages in the pipeline.

Let's look at an example. Suppose we have a 4-stage pipeline, where:

- S1 = fetch instruction
- S2 = decode and calculate effective address

- S3 = fetch operand
- S4 = execute instruction and store results

We must also assume the architecture provides a means to fetch data and instructions in parallel. This can be done with separate instruction and data paths; however, most memory systems do not allow this. Instead, they provide the operand in cache, which, in most cases, allows the instruction and operand to be fetched simultaneously. Suppose, also, that instruction I3 is a conditional branch statement that alters the execution sequence (so that instead of I4 running next, it transfers control to I8). This results in the pipeline operation shown in Figure 5.5.

Note that I4, I5, and I6 are fetched and proceed through various stages, but after the execution of I3 (the branch), I4, I5, and I6 are no longer needed. Only after time period 6, when the branch has executed, can the next instruction to be executed (I8) be fetched, after which, the pipe refills. From time periods 6 through 9, only one instruction has executed. In a perfect world, for each time period after the pipe originally fills, one instruction should flow out of the pipeline. However, we see in this example that this is not necessarily true.

Please note that not all instructions must go through each stage of the pipe. If an instruction has no operand, there is no need for stage 3. To simplify pipelining hardware and timing, all instructions proceed through all stages, whether necessary or not.

From our preceding discussion of speedup, it might appear that the more stages that exist in the pipeline, the faster everything will run. This is true to a point. There is a fixed overhead involved in moving data from memory to registers. The amount of control logic for the pipeline also increases in size proportional to the number of stages, thus slowing down total execution. In addition, there are several conditions that result in "pipeline conflicts," which keep us from reaching the goal of executing one instruction per clock cycle. These include:

- Resource conflicts
- Data dependencies
- Conditional branch statements

Resource conflicts are a major concern in instruction-level parallelism. For example, if one instruction is storing a value to memory while another is being fetched

Time Period →	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
(branch) 3			S1	S2	S3	S4							
4				S1	S2	S3							
5					S1	S2							
6						S1							
8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

FIGURE 5.5 Example Instruction Pipeline with Conditional Branch

from memory, both need access to memory. Typically this is resolved by allowing the instruction executing to continue, while forcing the instruction fetch to wait. Certain conflicts can also be resolved by providing two separate pathways: one for data coming from memory and another for instructions coming from memory.

Data dependencies arise when the result of one instruction, not yet available, is to be used as an operand to a following instruction. There are several ways to handle these types of pipeline conflicts. Special hardware can be added to detect instructions whose source operands are destinations for instructions further up the pipeline. This hardware can insert a brief delay (typically a no-op instruction that does nothing) into the pipeline, allowing enough time to pass to resolve the conflict. Specialized hardware can also be used to detect these conflicts and route data through special paths that exist between various stages of the pipeline. This reduces the time necessary for the instruction to access the required operand. Some architectures address this problem by letting the compiler resolve the conflict. Compilers have been designed that reorder instructions, resulting in a delay of loading any conflicting data but having no effect on the program logic or output.

Branch instructions allow us to alter the flow of execution in a program, which, in terms of pipelining, causes major problems. If instructions are fetched one per clock cycle, several can be fetched and even decoded before a preceding instruction, indicating a branch, is executed. Conditional branching is particularly difficult to deal with. Many architectures offer *branch prediction*, using logic to make the best guess as to which instructions will be needed next (essentially, they are predicting the outcome of a conditional branch). Compilers try to resolve branching issues by rearranging the machine code to cause a *delayed branch*. An attempt is made to reorder and insert useful instructions, but if that is not possible, no-op instructions are inserted to keep the pipeline full. Another approach used by some machines given a conditional branch is to start fetches on both paths of the branch and save them until the branch is actually executed, at which time the "true" execution path will be known.

In an effort to squeeze even more performance out of the chip, modern CPUs employ superscalar design (introduced in Chapter 4), which is one step beyond pipelining. Superscalar chips have multiple ALUs and issue more than one instruction in each clock cycle. The clock cycles per instruction can actually go below one. But the logic to keep track of hazards becomes even more complex; more logic is needed to schedule operations than to do them. But even with complex logic, it is hard to schedule parallel operations "on the fly."

The limits of dynamic scheduling have led machine designers to consider a very different architecture, explicitly parallel instruction computers (EPIC), exemplified by the Itanium architecture discussed in Chapter 4. EPIC machines have very large instructions (recall the instructions for the Itanium are 128 bits), which specify several operations to be done in parallel. Because of the parallelism inherent in the design, the EPIC instruction set is heavily compiler dependent (which means a user needs a sophisticated compiler to take advantage of the parallelism to gain significant performance advantages). The burden of scheduling

operations is shifted from the processor to the compiler, and much more time can be spent in developing a good schedule and analyzing potential pipeline conflicts.

To reduce the pipelining problems due to conditional branches, the IA-64 introduced *predicated* instructions. Comparison instructions set predicate bits, much like they set condition codes on the x86 machine (except that there are 64 predicate bits). Each operation specifies a predicate bit; it is executed only if the predicate bit equals 1. In practice, all operations are performed, but the result is stored into the register file only if the predicate bit equals 1. The result is that more instructions are executed, but we don't have to stall the pipeline waiting for a condition.

There are several levels of parallelism, varying from the simple to the more complex. All computers exploit parallelism to some degree. Instructions use words as operands (where words are typically 16, 32, or 64 bits in length), rather than acting on single bits at a time. More advanced types of parallelism require more specific and complex hardware and operating system support.

Although an in-depth study of parallelism is beyond the scope of this text, we would like to take a brief look at what we consider the two extremes of parallelism: program level parallelism (PLP) and instruction level parallelism (ILP). PLP actually allows parts of a program to run on more than one computer. This may sound simple, but it requires coding the algorithm correctly so that this parallelism is possible, in addition to providing careful synchronization between the various modules.

ILP involves the use of techniques to allow the execution of overlapping instructions. Essentially, we want to allow more than one instruction within a single program to execute concurrently. There are two kinds of ILP. The first type decomposes an instruction into stages and overlaps these stages. This is exactly what pipelining does. The second kind of ILP allows individual instructions to overlap (that is, instructions can be executed at the same time by the processor itself).

In addition to pipelined architectures, superscalar, superpipelining, and very long instruction word (VLIW) architectures exhibit ILP. Superscalar architectures (as you may recall from Chapter 4) perform multiple operations at the same time by employing parallel pipelines. Examples of superscalar architectures include IBM's PowerPC, Sun's UltraSparc, and DEC's Alpha. *Superpipelining* architectures combine superscalar concepts with pipelining, by dividing the pipeline stages into smaller pieces. The IA-64 architecture exhibits a *VLIW* architecture, which means each instruction can specify multiple scalar operations (the compiler puts multiple operations into a single instruction). Superscalar and VLIW machines fetch and execute more than one instruction per cycle.

5.6 REAL-WORLD EXAMPLES OF ISAs

Let's return to the two architectures we discussed in Chapter 4, Intel and MIPS, to see how the designers of these processors chose to deal with the issues introduced in this chapter: instruction formats, instruction types, number of operands,

addressing, and pipelining. We'll also introduce the Java Virtual Machine to illustrate how software can create an ISA abstraction that completely hides the real ISA of the machine.

5.6.1 Intel

Intel uses a little endian, two-address architecture, with variable-length instructions. Intel processors use a register-memory architecture, which means all instructions can operate on a memory location, but the other operand must be a register. This ISA allows variable-length operations, operating on data with lengths of 1, 2, or 4 bytes.

The 8086 through the 80486 are single-stage pipeline architectures. The architects reasoned that if one pipeline was good, two would be better. The Pentium had two parallel five-stage pipelines, called the U pipe and the V pipe, to execute instructions. Stages for these pipelines include Prefetch, Instruction Decode, Address Generation, Execute, and Write Back. To be effective, these pipelines must be kept filled, which requires instructions that can be issued in parallel. It is the compiler's responsibility to make sure this parallelism happens. The Pentium II increased the number of stages to 12, including Prefetch, Length Decode, Instruction Decode, Rename/Resource Allocation, UOP Scheduling/Dispatch, Execution, Write Back, and Retirement. Most of the new stages were added to address Intel's MMX technology, an extension to the architecture that handles multimedia data. The Pentium III increased the stages to 14, and the Pentium IV to 24. Additional stages (beyond those introduced in this chapter) included stages for determining the length of the instruction, stages for creating microoperations, and stages to "commit" the instruction (make sure it executes and the results become permanent). The Itanium contains only a 10-stage instruction pipeline.

Intel processors allow for the basic addressing modes introduced in this chapter, in addition to many combinations of those modes. The 8086 provided 17 different ways to access memory, most of which were variants of the basic modes. Intel's more current Pentium architectures include the same addressing modes as their predecessors, but also introduce new modes, mostly to help with maintaining backward compatibility. The IA-64 is surprisingly lacking in memory-addressing modes. It has only one: register indirect (with optional post-increment). This seems unusually limiting but follows the RISC philosophy. Addresses are calculated and stored in general-purpose registers. The more complex addressing modes require specialized hardware; by limiting the number of addressing modes, the IA-64 architecture minimizes the need for this specialized hardware.

5.6.2 MIPS

The MIPS architecture (which originally stood for "Microprocessor without Interlocked Pipeline Stages") is a little endian, word-addressable, three-address, fixed-length ISA. This is a load and store architecture, which means only the load and store instructions can access memory. All other instructions must use registers for operands, which implies that this ISA needs a large register set. MIPS is

also limited to fixed-length operations (those that operate on data with the same number of bytes).

Some MIPS processors (such as the R2000 and R3000) have five-stage pipelines. The R4000 and R4400 have 8-stage superpipelines. The R10000 is quite interesting in that the number of stages in the pipeline depends on the functional unit through which the instruction must pass: there are five stages for integer instructions, six for load/store instructions, and seven for floating-point instructions. Both the MIPS 5000 and 10000 are superscalar.

MIPS has a straightforward ISA with five basic types of instructions: simple arithmetic (add, XOR, NAND, shift), data movement (load, store, move), control (branch, jump), multi-cycle (multiply, divide), and miscellaneous instructions (save PC, save register on condition). MIPS programmers can use immediate, register, direct, indirect register, base, and indexed addressing modes. However, the ISA itself provides for only one (base addressing). The remaining modes are provided by the assembler. The MIPS64 has two additional addressing modes for use in embedded systems optimizations.

The MIPS instructions in Chapter 4 had up to four fields: an opcode, two operand addresses, and one result address. Essentially three instruction formats are available: the I type (immediate), the R type (register), and the J type (jump).

R type instructions have a 6-bit opcode, a 5-bit source register, a 5-bit target register, a 5-bit shift amount, and a 6-bit function. I type instructions have a 6-bit operand, a 5-bit source register, a 5-bit target register or branch condition, and a 16-bit immediate branch displacement or address displacement. J type instructions have a 6-bit opcode and a 26-bit target address.

5.6.3 Java Virtual Machine

Java, a language that is becoming quite popular, is very interesting in that it is platform independent. This means that if you compile code on one architecture (say a Pentium) and you wish to run your program on a different architecture (say a Sun workstation), you can do so without modifying or even recompiling your code.

The Java compiler makes no assumptions about the underlying architecture of the machine on which the program will run, such as the number of registers, memory size, or I/O ports, when you first compile your code. After compilation, however, to execute your program, you will need a *Java Virtual Machine* (*JVM*) for the architecture on which your program will run. (A *virtual machine* is a software emulation of a real machine.) The JVM is essentially a "wrapper" that goes around the hardware architecture, and is very platform dependent. The JVM for a Pentium is different from the JVM for a Sun workstation, which is different from the JVM for a Macintosh, and so on. But once the JVM exists on a particular architecture, that JVM can execute any Java program compiled on any ISA platform. It is the JVM's responsibility to load, check, find, and execute bytecodes at run time. The JVM, although virtual, is a nice example of a well-designed ISA.

The JVM for a particular architecture is written in that architecture's native instruction set. It acts as an interpreter, taking Java bytecodes and interpreting them into explicit underlying machine instructions. *Bytecodes* are produced when

a Java program is compiled. These bytecodes then become input for the JVM. The JVM can be compared to a giant switch (or case) statement, analyzing one bytecode instruction at a time. Each bytecode instruction causes a jump to a specific block of code, which implements the given bytecode instruction.

This differs significantly from other high-level languages with which you may be familiar. For example, when you compile a C++ program, the object code produced is for that particular architecture. (Compiling a C++ program results in an assembly language program that is translated to machine code.) If you want to run your C++ program on a different platform, you must recompile it for the target architecture. Compiled languages are translated into runnable files of the binary machine code by the compiler. Once this code has been generated, it can be run only on the target architecture. Compiled languages typically exhibit excellent performance and give very good access to the operating system. Examples of compiled languages include C, C++, Ada, FORTRAN, and COBOL.

Some languages, such as LISP, PhP, Perl, Python, Tcl, and most BASIC languages, are interpreted. The source must be reinterpreted each time the program is run. The trade-off for the platform independence of interpreted languages is slower performance—usually by a factor of 100 times. (We will have more to say on this topic in Chapter 8.)

Languages that are a bit of both (compiled and interpreted) exist as well. These are often called *P-code languages*. The source code written in these languages is compiled into an intermediate form, called P-code, and the P-code is then interpreted. P-code languages typically execute from 5 to 10 times more slowly than compiled languages. Python, Perl, and Java are actually P-code languages, even though they are typically referred to as interpreted languages.

Figure 5.6 presents an overview of the Java programming environment.

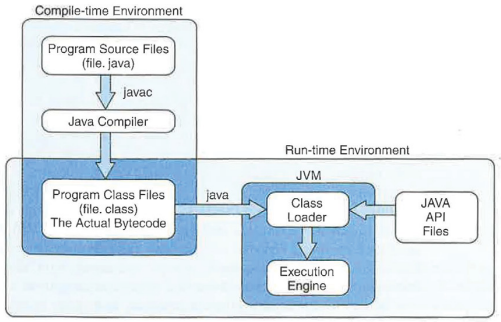


FIGURE 5.6 The Java Programming Environment

Perhaps more interesting than Java's platform independence, particularly in relationship to the topics covered in this chapter, is the fact that Java's bytecode is a stack-based language, partially composed of zero address instructions. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode itself indicates whether it is followed by operands and the form the operands (if any) take. Many of these instructions require zero operands.

Java uses two's complement to represent signed integers but does not allow for unsigned integers. Characters are coded using 16-bit Unicode. Java has four registers, which provide access to five different main memory regions. All references to memory are based on offsets from these registers; pointers or absolute memory addresses are never used. Because the JVM is a stack machine, no general registers are provided. This lack of general registers is detrimental to performance, as more memory references are generated. We are trading performance for portability.

Let's take a look at a short Java program and its corresponding bytecode. Example 5.2 shows a Java program that finds the maximum of two numbers.

EXAMPLE 5.2 Here is a Java program to find the maximum of two numbers.

```
public class Maximum {

  public static void main (String[] Args)
  { int X,Y,Z;
    X = Integer.parseInt(Args[0]);
    Y = Integer.parseInt(Args[1]);
    Z = Max(X,Y);
    System.out.println(Z);
  }

  public static int Max (int A, int B)
  { int C;
    if (A>B)C=A;
    else C=B;
    return C;
  }
}
```

After we compile this program (using `javac`), we can disassemble it to examine the bytecode, by issuing the following command:

```
javap -c Maximum
```

You should see the following:

```
Compiled from Maximum.java
public class Maximum extends java.lang.Object {
    public Maximum();
```

```
    public static void main(java.lang.String[]);
    public static int Max(int, int);
}

Method Maximum()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 aload_0
   1 iconst_0
   2 aaload
   3 invokestatic #2 <Method int parseInt(java.lang.String)>
   6 istore_1
   7 aload_0
   8 iconst_1
   9 aaload
  10 invokestatic #2 <Method int parseInt(java.lang.String)>
  13 istore_2
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Max(int, int)>
  19 istore_3
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
  24 invokevirtual #5 <Method void println(int)>
  27 return

Method int Max(int, int)
   0 iload_0
   1 iload_1
   2 if_icmple 10
   5 iload_0
   6 istore_2
   7 goto 12
  10 iload_1
  11 istore_2
  12 iload_2
  13 ireturn
```

Each line number represents an offset (or the number of bytes that an instruction is from the beginning of the current method). Notice that

```
Z = Max (X,Y);
```


gets compiled to the following bytecode:

```
14 iload_1
15 iload_2
16 invokestatic #3 <Method int Max(int, int)>
19 istore_3
```

It should be very obvious that Java bytecode is stack-based. For example, the `iadd` instruction pops two integers from the stack, adds them, and then pushes the result back to the stack. There is no such thing as "add r0, r1, f2" or "add AC, X". The `iload_1` (integer load) instruction also uses the stack by pushing slot 1 onto the stack (slot 1 in main contains *X*, so *X* is pushed onto the stack). *Y* is pushed onto the stack by instruction 15. The `invokestatic` instruction actually performs the `Max` method call. When the method has finished, the `istore_3` instruction pops the top element of the stack and stores it in *Z*.

We will explore the Java language and the JVM in more detail in Chapter 8.

CHAPTER SUMMARY

The core elements of an instruction set architecture include the memory model (word size and how the address space is split), registers, data types, instruction formats, addressing, and instruction types. Even though most computers today have general purpose register sets and specify operands by combinations of memory and register locations, instructions vary in size, type, format, and the number of operands allowed. Instructions also have strict requirements for the locations of these operands. Operands can be located on the stack, in registers, in memory, or a combination of the three.

Many decisions must be made when ISAs are designed. Larger instruction sets mandate longer instructions, which means a longer fetch and decode time. Instructions having a fixed length are easier to decode but can waste space. Expanding opcodes represent a compromise between the need for large instruction sets and the desire to have short instructions. Perhaps the most interesting debate is that of little versus big endian byte ordering.

There are three choices for internal storage in the CPU: stacks, an accumulator, or general purpose registers. Each has its advantages and disadvantages, which must be considered in the context of the proposed architecture's applications. The internal storage scheme has a direct impact on the instruction format, particularly the number of operands the instruction is allowed to reference. Stack architectures use zero operands, which fits well with RPN notation.

Instructions are classified into the following categories: data movement, arithmetic, Boolean, bit manipulation, I/O, transfer of control, and special

purpose. Some ISAs have many instructions in each category, others have very few in each category, and many have a mix of each.

The advances in memory technology, resulting in larger memories, have prompted the need for alternative addressing modes. The various addressing modes introduced included immediate, direct, indirect, register, indexed, and stack. Having these different modes provides flexibility and convenience for the programmer without changing the fundamental operations of the CPU.

Instruction-level pipelining is one example of instruction-level parallelism. It is a common but complex technique that can speed up the fetch-decode-execute cycle. With pipelining we can overlap the execution of instructions, thus executing multiple instructions in parallel. However, we also saw that the amount of parallelism can be limited by conflicts in the pipeline. Whereas pipelining performs different stages of multiple instructions at the same time, superscalar architectures allow us to perform multiple operations at the same time. Superpipelining, a combination of superscalar and pipelining, in addition to VLIW, was also briefly introduced. There are many types of parallelism, but at the computer organization and architecture level, we are really concerned mainly with ILP.

Intel and MIPS have interesting ISAs, as we have seen in this chapter as well as in Chapter 4. However, the Java Virtual Machine is a unique ISA, because the ISA is built in software, thus allowing Java programs to run on any machine that supports the JVM. Chapter 8 covers the JVM in great detail.

FURTHER READING

Instruction sets, addressing, and instruction formats are covered in detail in almost every computer architecture book. The Patterson and Hennessy book (1997) provides excellent coverage in these areas. Many books, such as Brey (2003), Messmer (1993), Abel (2001) and Jones (2001) are devoted to the Intel x86 architecture. For those interested in the Motorola 68000 series, we suggest Wray and Greenfield (1994) or Miller (1992).

Sohi (1990) gives a very nice discussion of instruction-level pipelining. Kaeli and Emma (1991) provide an interesting overview of how branching affects pipeline performance. For a nice history of pipelining, see Rau and Fisher (1993). To get a better idea of the limitations and problems with pipelining, see Wall (1993).

We investigated specific architectures in Chapter 4, but there are many important instruction set architectures worth mentioning. Atanasoff's ABC computer (Burks and Burks [1988], Von Neumann's EDVAC and Mauchly and Eckert's UNIVAC (Stern [1981] for information on both) had very simple instruction set architectures but required programming to be done in machine language. The Intel 8080 (a one-address machine) was the predecessor to the 80x86 family of chips introduced in Chapter 4. See Brey (2003) for a thorough and readable introduction to the Intel family of processors. Hauck (1968) provides good coverage of the Burroughs zero-address machine. Struble (1975) has a nice presentation of IBM's 360 family. Brunner (1991) gives details about DEC's VAX systems, which incorporated two-address architectures with more sophisticated instruction sets. SPARC (1994) provides a great overview of the SPARC architecture. Meyer and Downing (1991), Lindholm and Yellin, and Venner provide very interesting coverage of the JVM.

REFERENCES

Abel, Peter. *IBM PC Assembly Language and Programming*, 5th ed., Upper Saddle River, NJ: Prentice Hall, 2001.

Brey, B. *Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, and Pentium Pro Processor, Pentium II, Pentium III, and Pentium IV: Architecture, Programming, and Interfacing*, 6th ed., Englewood Cliffs, NJ: Prentice Hall, 2003.

Brunner, R.A. *VAX Architecture Reference Manual*, 2nd ed., Herndon, VA: Digital Press, 1991.

Burks, Alice, & Burks, Arthur. *The First Electronic Computer: The Atanasoff Story.* Ann Arbor, MI: University of Michigan Press, 1988.

Hauck, E. A., & Dent, B. A. "Burroughs B6500/B7500 Stack Mechanism," *Proceedings of AFIPS SJCC* (1968), Vol. 32, pp. 245–251.

Jones, William. *Assembly Language Programming for the IBM PC Family*, 3rd ed., El Granada, CA: Scott/Jones Publishing, 2001.

Kaeli, D., & Emma, P. "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns." *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

Lindholm, Tim, & Yellin, Frank. *The Java Virtual Machine Specification*. Online at java.sun.com/docs/books/vmspec/html/VMSpecTOC.cod.html.

Messmer, H. *The Indispensable PC Hardware Book*. Reading, MA: Addison-Wesley, 1993.

Meyer, J., & Downing, T. *Java Virtual Machine*. Sebastopol, CA: O'Reilly & Associates, 1991.

Miller, M. A. *The 6800 Family, Architecture Programming and Applications*, 2nd ed., Columbus, OH: Charles E. Merrill, 1992.

Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design, The Hardware/Software Interface*, 2nd ed., San Mateo, CA: Morgan Kaufmann, 1997.

Rau, B. Ramakrishna, & Fisher, Joseph A. "Instruction-Level Parallel Processing: History, Overview and Perspective." *Journal of Supercomputing 7* (1), Jan. 1993, pp. 9–50.

Sohi, G. "Instruction Issue Logic for High-Performance Interruptible, Multiple Functional Unit, Pipelined Computers." *IEEE Transactions on Computers*, March 1990.

SPARC International, Inc., *The SPARC Architecture Manual: Version 9*, Upper Saddle River, NJ: Prentice Hall, 1994.

Stallings, W. *Computer Organization and Architecture*, 5th ed., New York, NY: Macmillan Publishing Company, 2000.

Stern, Nancy. *From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers.* Herndon, VA: Digital Press, 1981.

Struble, G. W. *Assembler Language Programming: The IBM System/360 and 370*, 2nd ed., Reading, MA: Addison-Wesley, 1975.

Tanenbaum, Andrew. *Structured Computer Organization*, 4th ed., Upper Saddle River, NJ: Prentice Hall, 1999.

Venner, Bill. *Inside the Java Virtual Machine*. Online at www.artima.com.

Wall, David W. *Limits of Instruction-Level Parallelism*. DEC-WRL Research Report 93/6, Nov. 1993.

Wray, W. C., & Greenfield, J. D. *Using Microprocessors and Microcomputers, the Motorola Family*. Englewood Cliffs, NJ: Prentice Hall, 1994.

REVIEW OF ESSENTIAL TERMS AND CONCEPTS

1. Explain the difference between register-to-register, register-to-memory, and memory-to-memory instructions.
2. Several design decisions exist with regard to instruction sets. Name four and explain.
3. What is an expanding opcode?
4. If a byte-addressable machine with 32-bit words stores the hex value 98765432, indicate how this value would be stored on a little endian machine and on a big endian machine. Why does "endian-ness" matter?
5. We can design stack architectures, accumulator architectures, or general-purpose register architectures. Explain the differences between these choices and give some situations where one might be better than another.
6. How do memory-memory, register-memory, and load-store architectures differ? How are they the same?
7. What are the pros and cons of fixed-length and variable-length instructions? Which is currently more popular?
8. How does an architecture based on zero operands ever get any data values from memory?
9. Which is likely to be longer (have more instructions): a program written for a zero-address architecture, a program written for a one-address architecture, or a program written for a two-address architecture? Why?
10. Why might stack architectures represent arithmetic expressions in reverse Polish notation?
11. Name the seven types of data instructions and explain each.
12. What is an address mode?
13. Give examples of immediate, direct, register, indirect, register indirect, and indexed addressing.
14. How does indexed addressing differ from based addressing?
15. Why do we need so many different addressing modes?
16. Explain the concept behind pipelining.
17. What is the theoretical speedup for a 4-stage pipeline with a 20ns clock cycle if it is processing 100 tasks?
18. What are the pipeline conflicts that can cause a slowdown in the pipeline?
19. What are the two types of ILP and how do they differ?
20. Explain superscalar, superpipelining, and VLIW architectures.

21. List several ways in which the Intel and MIPS ISAs differ. Name several ways in which they are the same.
22. Explain Java bytecodes.
23. Give an example of a current stack-based architecture and a current GPR-based architecture. How do they differ?

EXERCISES

1. Assume you have a byte-addressable machine that uses 32-bit integers and you are storing the hex value 1234 at address 0:
   - ◆ **a)** Show how this is stored on a big endian machine.
   - ◆ **b)** Show how this is stored on a little endian machine.
   - **c)** If you wanted to increase the hex value to 123456, which byte assignment would be more efficient, big or little endian? Explain your answer.
2. Show how the following values would be stored by byte-addressable machines with 32-bit words, using little endian and then big endian format. Assume each value starts at address 10_{16}. Draw a diagram of memory for each, placing the appropriate values in the correct (and labeled) memory locations.
   - **a)** $456789A1_{16}$
   - **b)** $0000058A_{16}$
   - **c)** 14148888_{16}
- ◆ 3. The first two bytes of a 2M × 16 main memory have the following hex values:
   - Byte 0 is FE
   - Byte 1 is 01

   If these bytes hold a 16-bit two's complement integer, what is its actual decimal value if:
   - ◆ **a)** memory is big endian?
   - ◆ **b)** memory is little endian?
4. What kinds of problems do you think endian-ness can cause if you wished to transfer data from a big endian machine to a little endian machine? Explain.
- ◆ 5. The Population Studies Institute monitors the population of the United States. In 2000, this institute wrote a program to create files of the numbers representing populations of the various states, as well as the total population of the U.S. This program, which runs on a Motorola processor, projects the population based on various rules, such as the average number of births and deaths per year. The institute runs the program and then ships the output files to state agencies so the data values can be used as input into various applications. However, one Pennsylvania agency, running all Intel machines, encountered difficulties, as indicated by the following problem.

When the 32-bit unsigned integer $1D2F37E8_{16}$ (representing the overall U.S. population prediction for 2003) is used as input, and the agency's program simply outputs this input value, the U.S. population forecast for 2003 is far too large. Can you help this Pennsylvania agency by explaining what might be going wrong?

6. There are reasons for machine designers to want all instructions to be the same length. Why is this not a good idea on a stack machine?

◆ **7.** A computer has 32-bit instructions and 12-bit addresses. Suppose there are 250 2-address instructions. How many 1-address instructions can be formulated? Explain your answer.

8. Convert the following expressions from infix to reverse Polish (postfix) notation.

◆ **a)** X × Y + W × Z + V × U

b) W × X + W × (U × V + Z)

c) (W × (X + Y × (U × V)))/(U × (X + Y))

9. Convert the following expressions from reverse Polish notation to infix notation.

a) W X Y Z − + ×

b) U V W X Y Z + × + × +

c) X Y Z + V W − × Z + +

10. a) Write the following expression in postfix (reverse Polish) notation. Remember the rules of precedence for arithmetic operators!

$$X = \frac{A - B + C \times (D \times E - F)}{G + H \times K}$$

b) Write a program to evaluate the above arithmetic statement using a stack organized computer with zero-address instructions (so only `pop` and `push` can access memory).

11. a) In a computer instruction format, the instruction length is 11 bits and the size of an address field is 4 bits. Is it possible to have

5 2-address instructions

45 1-address instructions

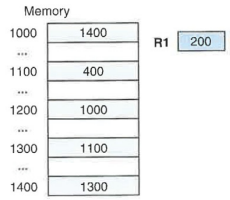
32 0-address instructions

using the format? Justify your answer.

b) Assume that a computer architect has already designed 6 two-address and 24 zero-address instructions using the instruction format given in Problem 11. What is the maximum number of one-address instructions that can be added to the instruction set?

12. What is the difference between using direct and indirect addressing? Give an example.

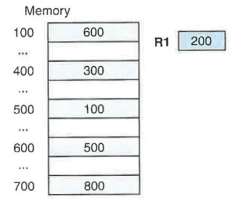
◆ **13.** Suppose we have the instruction `Load 1000`. Given that memory and register R1 contain the values below:



Assuming R1 is implied in the indexed addressing mode, determine the actual value loaded into the accumulator and fill in the table below:

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

14. Suppose we have the instruction `Load 500`. Given that memory and register R1 contain the values below:



Assuming R1 is implied in the indexed addressing mode, determine the actual value loaded into the accumulator and fill in the table below:

Mode	Value Loaded into AC
Immediate	
Direct	
Indirect	
Indexed	

15. A nonpipelined system takes 200ns to process a task. The same task can be processed in a 5-segment pipeline with a clock cycle of 40ns. Determine the speedup ratio of the pipeline for 200 tasks. What is the maximum speedup that could be achieved with the pipeline unit over the nonpipelined unit?
16. A nonpipeline system takes 100ns to process a task. The same task can be processed in a 5-stage pipeline with a clock cycle of 20ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the theoretical speedup that could be achieved with the pipeline system over a nonpipelined system?
17. Write code to implement the expression A = (B + C) × (D + E) on 3-, 2-, 1-, and 0-address machines. In accordance with programming language practice, computing the expression should not change the values of its operands.
◆ 18. A digital computer has a memory unit with 24 bits per word. The instruction set consists of 150 different operations. All instructions have an operation code part (opcode) and an address part (allowing for only one address). Each instruction is stored in one word of memory.
   ◆ a) How many bits are needed for the opcode?
   ◆ b) How many bits are left for the address part of the instruction?
   ◆ c) What is the maximum allowable size for memory?
   ◆ d) What is the largest unsigned binary number that can be accommodated in one word of memory?
19. The memory unit of a computer has 256K words of 32 bits each. The computer has an instruction format with 4 fields: an opcode field; a mode field to specify 1 of 7 addressing modes; a register address field to specify 1 of 60 registers; and a memory address field. Assume an instruction is 32 bits long. Answer the following:
   a) How large must the mode field be?
   b) How large must the register field be?
   c) How large must the address field be?
   d) How large is the opcode field?
20. Suppose an instruction takes four cycles to execute in a nonpipelined CPU: one cycle to fetch the instruction, one cycle to decode the instruction, one cycle to perform the ALU operation, and one cycle to store the result. In a CPU with a 4-stage pipeline, that instruction still takes four cycles to execute, so how can we say the pipeline speeds up the execution of the program?
*21. Pick an architecture (other than those covered in this chapter). Do research to find out how your architecture approaches the concepts introduced in this chapter, as was done for Intel, MIPS, and Java.

True or False.

1. Most computers typically fall into one of three types of CPU organization: (1) general register organization; (2) single accumulator organization; or (3) stack organization.
2. The advantage of zero-address instruction computers is that they have short programs; the disadvantage is that the instructions require many bits, making them very long.

RAM /abr./: Rarely Adequate Memory.
because the more memory a computer has, the faster it can produce error messages.

—Anonymous

640K [of memory] ought to be enough for anybody.

—Bill Gates

CHAPTER 6 Memory

6.1 INTRODUCTION

Most computers are built using the Von Neumann model, which is centered on memory. The programs that perform the processing are stored in memory. We examined a small 4 × 3-bit memory in Chapter 3 and we learned how to address memory in Chapters 4 and 5. We know memory is logically structured as a linear array of locations, with addresses from 0 to the maximum memory size the processor can address. In this chapter we examine the various types of memory and how each is part of the memory hierarchy system. We then look at cache memory (a special high-speed memory) and a method that utilizes memory to its fullest by means of virtual memory implemented via paging.

6.2 TYPES OF MEMORY

A common question many people ask is "why are there so many different types of computer memory?" The answer is that new technologies continue to be introduced in an attempt to match the improvements in CPU design—the speed of memory has to, somewhat, keep pace with the CPU, or the memory becomes a bottleneck. Although we have seen many improvements in CPUs over the past few years, improving main memory to keep pace with the CPU is actually not as critical because of the use of *cache memory*. Cache memory is a small, high-speed (and thus high-cost) type of memory that serves as a buffer for frequently accessed data. The additional expense of using very fast technologies for memory cannot always be justified because slower memories can often be "hidden" by

high-performance cache systems. However, before we discuss cache memory, we will explain the various memory technologies.

Even though a large number of memory technologies exist, there are only two basic types of memory: *RAM* (*random access memory*) and *ROM* (*read-only memory*). RAM is somewhat of a misnomer; a more appropriate name is read-write memory. RAM is the memory to which computer specifications refer; if you buy a computer with 128 megabytes of memory, it has 128MB of RAM. RAM is also the "main memory" we have continually referred to throughout this book. Often called primary memory, RAM is used to store programs and data that the computer needs when executing programs; but RAM is volatile, and loses this information once the power is turned off. There are two general types of chips used to build the bulk of RAM memory in today's computers: SRAM and DRAM (static and dynamic random access memory).

Dynamic RAM is constructed of tiny capacitors that leak electricity. DRAM requires a recharge every few milliseconds to maintain its data. Static RAM technology, in contrast, holds its contents as long as power is available. SRAM consists of circuits similar to the D flip-flops we studied in Chapter 3. SRAM is faster and much more expensive than DRAM; however, designers use DRAM because it is much denser (can store many bits per chip), uses less power, and generates less heat than SRAM. For these reasons, both technologies are often used in combination: DRAM for main memory and SRAM for cache. The basic operation of all DRAM memories is the same, but there are many flavors, including Multibank DRAM (MDRAM), Fast-Page Mode (FPM) DRAM, Extended Data Out (EDO) DRAM, Burst EDO DRAM (BEDO DRAM), Synchronous Dynamic Random Access Memory (SDRAM), Synchronous-Link (SL) DRAM, Double Data Rate (DDR) SDRAM, and Direct Rambus (DR) DRAM. The different types of SRAM include asynchronous SRAM, synchronous SRAM, and pipeline burst SRAM. For more information about these types of memory, refer to the references listed at the end of the chapter.

In addition to RAM, most computers contain a small amount of ROM (read-only memory) that stores critical information necessary to operate the system, such as the program necessary to boot the computer. ROM is not volatile and always retains its data. This type of memory is also used in embedded systems or any systems where the programming does not need to change. Many appliances, toys, and most automobiles use ROM chips to maintain information when the power is shut off. ROMs are also used extensively in calculators and peripheral devices such as laser printers, which store their fonts in ROMs. There are five basic different types of ROM: ROM, PROM, EPROM, EEPROM, and flash memory. *PROM (programmable read-only memory)* is a variation on ROM. PROMs can be programmed by the user with the appropriate equipment. Whereas ROMs are hardwired, PROMs have fuses that can be blown to program the chip. Once programmed, the data and instructions in PROM cannot be changed. *EPROM* (*erasable PROM*) is programmable with the added advantage of being reprogrammable (erasing an EPROM requires a special tool that emits ultraviolet light). To reprogram an EPROM, the entire chip must first be erased. *EEPROM* (*electrically erasable PROM*) removes many of the disadvantages of EPROM: no

special tools are required for erasure (this is performed by applying an electric field) and you can erase only portions of the chip, one byte at a time. *Flash memory* is essentially EEPROM with the added benefit that data can be written or erased in blocks, removing the one-byte-at-a-time limitation. This makes flash memory faster than EEPROM.

6.3 THE MEMORY HIERARCHY

One of the most important considerations in understanding the performance capabilities of a modern processor is the memory hierarchy. Unfortunately, as we have seen, not all memory is created equal, and some types are far less efficient and thus cheaper than others. To deal with this disparity, today's computer systems use a combination of memory types to provide the best performance at the best cost. This approach is called *hierarchical memory*. As a rule, the faster memory is, the more expensive it is per bit of storage. By using a hierarchy of memories, each with different access speeds and storage capacities, a computer system can exhibit performance above what would be possible without a combination of the various types. The base types that normally constitute the hierarchical memory system include registers, cache, main memory, and secondary memory.

Today's computers each have a small amount of very high-speed memory, called a *cache*, where data from frequently used memory locations may be temporarily stored. This cache is connected to a much larger *main memory*, which is typically a medium-speed memory. This memory is complemented by a very large secondary memory, composed of a hard disk and various removable media. By using such a hierarchical scheme, one can improve the effective access speed of the memory, using only a small number of fast (and expensive) chips. This allows designers to create a computer with acceptable performance at a reasonable cost.

We classify memory based on its "distance" from the processor, with distance measured by the number of machine cycles required for access. The closer memory is to the processor, the faster it should be. As memory gets further from the main processor, we can afford longer access times. Thus, slower technologies are used for these memories, and faster technologies are used for memories closer to the CPU. The better the technology, the faster and more expensive the memory becomes. Thus, faster memories tend to be smaller than slower ones, due to cost.

The following terminology is used when referring to this memory hierarchy:

- *Hit*—The requested data resides in a given level of memory (typically, we are concerned with the hit rate only for upper levels of memory).
- *Miss*—The requested data is not found in the given level of memory.
- *Hit rate*—The percentage of memory accesses found in a given level of memory.
- *Miss rate*—The percentage of memory accesses not found in a given level of memory. Note: Miss Rate = 1 − Hit Rate.
- *Hit time*—The time required to access the requested information in a given level of memory.

- *Miss penalty*—The time required to process a miss, which includes replacing a block in an upper level of memory, plus the additional time to deliver the requested data to the processor. (The time to process a miss is typically significantly larger than the time to process a hit.)

The memory hierarchy is illustrated in Figure 6.1. This is drawn as a pyramid to help indicate the relative sizes of these various memories. Memories closer to the top tend to be smaller in size. However, these smaller memories have better performance and thus a higher cost (per bit) than memories found lower in the pyramid. The numbers given to the left of the pyramid indicate typical access times.

For any given data, the processor sends its request to the fastest, smallest partition of memory (typically cache, because registers tend to be more special purpose). If the data is found in cache, it can be loaded quickly into the CPU. If it is not resident in cache, the request is forwarded to the next lower level of the hierarchy, and this search process begins again. If the data is found at this level, the whole block in which the data resides is transferred into cache. If the data is not found at this level, the request is forwarded to the next lower level, and so on. The key idea is that when the lower (slower, larger, and cheaper) levels of the hierarchy respond to a request from higher levels for the content of location X, they also send, at the same time, the data located at addresses $X + 1$, $X + 2$, . . . , thus returning an entire block of data to the higher-level memory. The hope is that this extra data will be referenced in the near future, which, in most cases, it is. The memory hierarchy is functional because programs tend to exhibit a property known as *locality*, which often allows the processor to access the data returned for addresses $X + 1$, $X + 2$, and so on. Thus, although there is one miss to, say

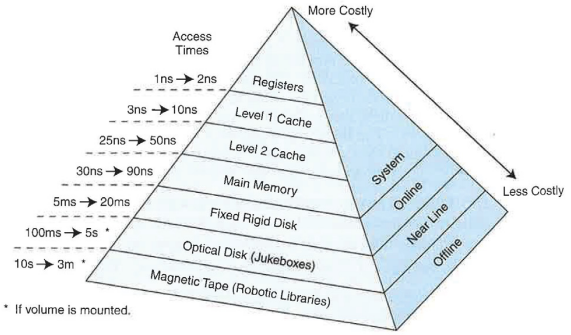


FIGURE 6.1 The Memory Hierarchy

cache, for X, there may be several hits in cache on the newly retrieved block afterward, due to locality.

6.3.1 Locality of Reference

In practice, processors tend to access memory in a very patterned way. For example, in the absence of branches, the PC in MARIE is incremented by one after each instruction fetch. Thus, if memory location X is accessed at time t, there is a high probability that memory location $X + 1$ will also be accessed in the near future. This clustering of memory references into groups is an example of *locality of reference*. This locality can be exploited by implementing the memory as a hierarchy; when a miss is processed, instead of simply transferring the requested data to a higher level, the entire block containing the data is transferred. Because of locality of reference, it is likely that the additional data in the block will be needed in the near future, and if so, this data can be loaded quickly from the faster memory.

There are three basic forms of locality:

- *Temporal locality*—Recently accessed items tend to be accessed again in the near future.
- *Spatial locality*—Accesses tend to be clustered in the address space (for example, as in arrays or loops).
- *Sequential locality*—Instructions tend to be accessed sequentially.

The locality principle provides the opportunity for a system to use a small amount of very fast memory to effectively accelerate the majority of memory accesses. Typically, only a small amount of the entire memory space is being accessed at any given time, and values in that space are being accessed repeatedly. Therefore, we can copy those values from a slower memory to a smaller but faster memory that resides higher in the hierarchy. This results in a memory system that can store a large amount of information in a large but low-cost memory, yet provide nearly the same access speeds that would result from using very fast but expensive memory.

6.4 CACHE MEMORY

A computer processor is very fast and is constantly reading information from memory, which means it often has to wait for the information to arrive, because the memory access times are slower than the processor speed. A cache memory is a small, temporary, but fast memory that the processor uses for information it is likely to need again in the very near future.

Noncomputer examples of caching are all around us. Keeping them in mind will help you to understand computer memory caching. Think of a homeowner with a very large tool chest in the garage. Suppose you are this homeowner and have a home improvement project to work on in the basement. You know this project will require drills, wrenches, hammers, a tape measure, several types of saws, and many different types and sizes of screwdrivers. The first thing you want to do is measure and then cut some wood. You run out to the garage, grab

the tape measure from a huge tool storage chest, run down to the basement, measure the wood, run back out to the garage, leave the tape measure, grab the saw, and then return to the basement with the saw and cut the wood. Now you decide to bolt some pieces of wood together. So you run to the garage, grab the drill set, go back down to the basement, drill the holes to put the bolts through, go back to the garage, leave the drill set, grab one wrench, go back to the basement, find out the wrench is the wrong size, go back to the tool chest in the garage, grab another wrench, run back downstairs . . . wait! Would you really work this way? No! Being a reasonable person, you think to yourself "If I need one wrench, I will probably need another one of a different size soon anyway, so why not just grab the whole set of wrenches?" Taking this one step further, you reason "Once I am done with one certain tool, there is a good chance I will need another soon, so why not just pack up a small toolbox and take it to the basement?" This way, you keep the tools you need close at hand, so access is faster. You have just cached some tools for easy access and quick use! The tools you are less likely to use remain stored in a location that is further away and requires more time to access. This is all that cache memory does: It stores data that has been accessed and data that might be accessed by the CPU in a faster, closer memory.

Another cache analogy is found in grocery shopping. You seldom, if ever, go to the grocery store to buy one single item. You buy any items you require immediately in addition to items you will most likely use in the future. The grocery store is similar to main memory, and your home is the cache. As another example, consider how many of us carry around an entire phone book. Most of us have a small address book instead. We enter the names and numbers of people we tend to call more frequently; looking a number up in our address book is much quicker than finding a phone book, locating the name, and then getting the number. We tend to have the address book close at hand, whereas the phone book is probably located in our home, hidden in an end table or bookcase somewhere. The phone book is something we do not use frequently, so we can afford to store it in a little more out of the way location. Comparing the size of our address book to the telephone book, we see that the address book "memory" is much smaller than that of a telephone book. But the probability is very high that when we make a call, it is to someone in our address book.

Students doing research offer another commonplace cache example. Suppose you are writing a paper on quantum computing. Would you go to the library, check out one book, return home, get the necessary information from that book, go back to the library, check out another book, return home, and so on? No, you would go to the library and check out all the books you might need and bring them all home. The library is analogous to main memory, and your home is, again, similar to cache.

And as a last example, consider how one of your authors uses her office. Any materials she does not need (or has not used for a period of more than six months) get filed away in a large set of filing cabinets. However, frequently used "data" remain piled on her desk, close at hand, and easy (sometimes) to find. If she needs something from a file, she more than likely pulls the entire file, not simply one or two papers from the folder. The entire file is then added to the pile on her

desk. The filing cabinets are her "main memory" and her desk (with its many unorganized-looking piles) is the cache.

Cache memory works on the same basic principles as the preceding examples by copying frequently used data into the cache rather than requiring an access to main memory to retrieve the data. Cache can be as unorganized as your author's desk or as organized as your address book. Either way, however, the data must be accessible (locatable). Cache memory in a computer differs from our real-life examples in one important way: The computer really has no way to know, *a priori*, what data is most likely to be accessed, so it uses the locality principle and transfers an entire block from main memory into cache whenever it has to make a main memory access. If the probability of using something else in that block is high, then transferring the entire block saves on access time. The cache location for this new block depends on two things: the cache mapping policy (discussed in the next section) and the cache size (which affects whether there is room for the new block).

The size of cache memory can vary enormously. A typical personal computer's level 2 (L2) cache is 256K or 512K. Level 1 (L1) cache is smaller, typically 8K or 16K. L1 cache resides on the processor, whereas L2 cache resides between the CPU and main memory. L1 cache is, therefore, faster than L2 cache. The relationship between L1 and L2 cache can be illustrated using our grocery store example: If the store is main memory, you could consider your refrigerator the L2 cache, and the actual dinner table the L1 cache.

The purpose of cache is to speed up memory accesses by storing recently used data closer to the CPU, instead of storing it in main memory. Although cache is not as large as main memory, it is considerably faster. Whereas main memory is typically composed of DRAM with, say, a 60ns access time, cache is typically composed of SRAM, providing faster access with a much shorter cycle time than DRAM (a typical cache access time is 10ns). Cache does not need to be very large to perform well. A general rule of thumb is to make cache small enough so that the overall average cost per bit is close to that of main memory, but large enough to be beneficial. Because this fast memory is quite expensive, it is not feasible to use the technology found in cache memory to build all of main memory.

What makes cache "special"? Cache is not accessed by address; it is accessed by content. For this reason, cache is sometimes called *content addressable memory* or *CAM*. Under most cache mapping schemes, the cache entries must be checked or searched to see if the value being requested is stored in cache. To simplify this process of locating the desired data, various cache mapping algorithms are used.

6.4.1 Cache Mapping Schemes

For cache to be functional, it must store useful data. However, this data becomes useless if the CPU can't find it. When accessing data or instructions, the CPU first generates a main memory address. If the data has been copied to cache, the address of the data in cache is not the same as the main memory address. For example, data located at main memory address 2E3 could be located in the very

first location in cache. How, then, does the CPU locate data when it has been copied into cache? The CPU uses a specific mapping scheme that "converts" the main memory address into a cache location.

This address conversion is done by giving special significance to the bits in the main memory address. We first divide the bits into distinct groups we call *fields*. Depending on the mapping scheme, we may have two or three fields. How we use these fields depends on the particular mapping scheme being used. The mapping scheme determines where the data is placed when it is originally copied into cache and also provides a method for the CPU to find previously copied data when searching cache.

Before we discuss these mapping schemes, it is important to understand how data is copied into cache. Main memory and cache are both divided into the same size blocks (the size of these blocks varies). When a memory address is generated, cache is searched first to see if the required word exists there. When the requested word is not found in cache, the entire main memory block in which the word resides is loaded into cache. As previously mentioned, this scheme is successful because of the principle of locality—if a word was just referenced, there is a good chance words in the same general vicinity will soon be referenced as well. Therefore, one missed word often results in several found words. For example, when you are in the basement and you first need tools, you have a "miss" and must go to the garage. If you gather up a set of tools that you might need and return to the basement, you hope that you'll have several "hits" while working on your home improvement project and don't have to make many more trips to the garage. Because accessing a cache word (a tool already in the basement) is faster than accessing a main memory word (going to the garage yet again!), cache memory speeds up the overall access time.

So, how do we use fields in the main memory address? One field of the main memory address points us to a location in cache in which the data resides if it is resident in cache (this is called a *cache hit*), or where it is to be placed if it is not resident (which is called a *cache miss*). (This is slightly different for associative mapped cache, which we discuss shortly.) The cache block referenced is then checked to see if it is valid. This is done by associating a *valid bit* with each cache block. A valid bit of 0 means the cache block is not valid (we have a cache miss) and we must access main memory. A valid bit of 1 means it is valid (we may have a cache hit but we need to complete one more step before we know for sure). We then compare the tag in the cache block to the *tag field* of our address. (The tag is a special group of bits derived from the main memory address that is stored with its corresponding block in cache.) If the tags are the same, then we have found the desired cache block (we have a cache hit). At this point we need to locate the desired word in the block; this can be done using a different portion of the main memory address called the *word field*. All cache mapping schemes require a word field; however, the remaining fields are determined by the mapping scheme. We discuss the three main cache mapping schemes on the next page.

Direct Mapped Cache

Direct mapped cache assigns cache mappings using a modular approach. Because there are more main memory blocks than there are cache blocks, it should be clear that main memory blocks compete for cache locations. Direct mapping maps block X of main memory to block Y of cache, mod N, where N is the total number of blocks in cache. For example, if cache contains 10 blocks, then main memory block 0 maps to cache block 0, main memory block 1 maps to cache block 1, . . . , main memory block 9 maps to cache block 9, and main memory block 10 maps to cache block 0. This is illustrated in Figure 6.2. Thus, main memory blocks 0 and 10 (and 20, 30, and so on) all compete for cache block 0.

You may be wondering, if main memory blocks 0 and 10 both map to cache block 0, how does the CPU know which block actually resides in cache block 0 at any given time? The answer is that each block is copied to cache and identified

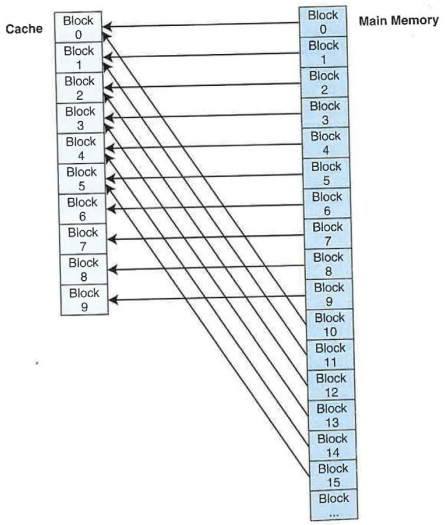


FIGURE 6.2 Direct Mapping of Main Memory Blocks to Cache Blocks

Block	Tag	Data	Valid
0	00000000	words A, B, C,...	1
1	11110101	words L, M, N,...	1
2	----------		0
3	----------		0

FIGURE 6.3 A Closer Look at Cache

by the tag previously described. If we take a closer look at cache, we see that it stores more than just that data copied from main memory, as indicated in Figure 6.3. In this figure, there are two valid cache blocks. Block 0 contains multiple words from main memory, identified using the tag "00000000". Block 1 contains words identified using tag "11110101". The other two cache blocks are not valid.

To perform direct mapping, the binary main memory address is partitioned into the fields shown in Figure 6.4.

The size of each field depends on the physical characteristics of main memory and cache. The *word* field (sometimes called the *offset* field) uniquely identifies a word from a specific block; therefore, it must contain the appropriate number of bits to do this. This is also true of the *block* field—it must select a unique block of cache. The *tag* field is whatever is left over. When a block of main memory is copied to cache, this tag is stored with the block and uniquely identifies this block. The total of all three fields must, of course, add up to the number of bits in a main memory address.

Consider the following example: Assume memory consists of 2^{14} words, cache has 16 blocks, and each block has 8 words. From this we determine that memory has $\frac{2^{14}}{2^3} = 2^{11}$ blocks. We know that each main memory address requires 14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the word field (we need 3 bits to uniquely identify one of 8 words in a block). We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits. The remaining 7 bits make up the tag field. The fields with sizes are illustrated in Figure 6.5.

As mentioned previously, the tag for each block is stored with that block in the cache. In this example, because main memory blocks 0 and 16 both map to cache block 0, the tag field would allow the system to differentiate between block

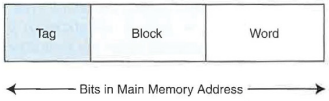


FIGURE 6.4 The Format of a Main Memory Address Using Direct Mapping

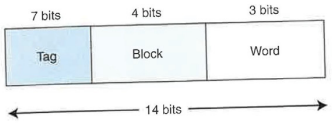


FIGURE 6.5 The Main Memory Address Format for Our Example

0 and block 16. The binary addresses in block 0 differ from those in block 16 in the upper leftmost 7 bits, so the tags are different and unique.

To see how these addresses differ, let's look at a smaller, simpler example. Suppose we have a system using direct mapping with 16 words of main memory divided into 8 blocks (so each block has 2 words). Assume the cache is 4 blocks in size (for a total of 8 words). Table 6.1 shows how the main memory blocks map to cache.

We know:

- A main memory address has 4 bits (because there are 2^4 or 16 words in main memory).
- This 4-bit main memory address is divided into three fields: The word field is 1 bit (we need only 1 bit to differentiate between the two words in a block); the block field is 2 bits (we have 4 blocks in main memory and need 2 bits to uniquely identify each block); and the tag field has 1 bit (this is all that is left over).

The main memory address is divided into the fields shown in Figure 6.6.

Main Memory	Maps To	Cache
Block 0 (addresses 0, 1)	→	Block 0
Block 1 (addresses 2, 3)	→	Block 1
Block 2 (addresses 4, 5)	→	Block 2
Block 3 (addresses 6, 7)	→	Block 3
Block 4 (addresses 8, 9)	→	Block 0
Block 5 (addresses 10, 11)	→	Block 1
Block 6 (addresses 12, 13)	→	Block 2
Block 7 (addresses 14, 15)	→	Block 3

TABLE 6.1 An Example of Main Memory Mapped to Cache

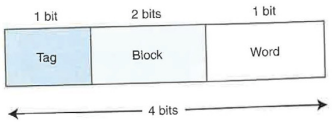


FIGURE 6.6 The Main Memory Address Format for a 16-Word Memory

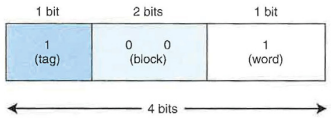


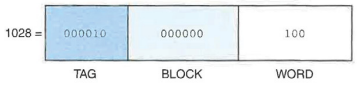
FIGURE 6.7 The Main Memory Address 9 = 1001_2 Split into Fields

Suppose we generate the main memory address 9. We can see from the mapping listing above that address 9 is in main memory block 4 and should map to cache block 0 (which means the contents of main memory block 4 should be copied into cache block 0). The computer, however, uses the actual main memory address to determine the cache mapping block. This address, in binary, is represented in Figure 6.7.

When the CPU generates this address, it first takes the block field bits 00 and uses these to direct it to the proper block in cache. 00 indicates that cache block 0 should be checked. If the cache block is valid, it then compares the tag field value of 1 (in the main memory address) to the tag associated with cache block 0. If the cache tag is 1, then block 4 currently resides in cache block 0. If the tag is 0, then block 0 from main memory is located in block 0 of cache. (To see this, compare main memory address $9 = 1001_2$, which is in block 4, to main memory address $1 = 0001_2$, which is in block 0. These two addresses differ only in the leftmost bit, which is the bit used as the tag by the cache.) Assuming the tags match, which means that block 4 from main memory (with addresses 8 and 9) resides in cache block 0, the word field value of 1 is used to select one of the two words residing in the block. Because the bit is 1, we select the word with offset 1, which results in retrieving the data copied from main memory address 9.

Let's do one more example in this context. Suppose the CPU now generates address $4 = 0100_2$. The middle two bits (10) direct the search to cache block 2. If the block is valid, the leftmost tag bit (0) would be compared to the tag bit stored with the cache block. If they match, the first word in that block (of offset 0) would be returned to the CPU. To make sure you understand this process, perform a similar exercise with the main memory address $12 = 1100_2$.

Let's move on to a larger example. Suppose we have a system using 15-bit main memory addresses and 64 blocks of cache. If each block contains 8 words, we know that the main memory 15-bit address is divided into a 3-bit word field, a 6-bit block field, and a 6-bit tag field. If the CPU generates the main memory address:



it would look in block 0 of cache, and if it finds a tag of 000010, the word at offset 4 in this block would be returned to the CPU.

Fully Associative Cache

Direct mapped cache is not as expensive as other caches because the mapping scheme does not require any searching. Each main memory block has a specific location to which it maps in cache; when a main memory address is converted to a cache address, the CPU knows exactly where to look in the cache for that memory block by simply examining the bits in the block field. This is similar to your address book: The pages often have an alphabetic index, so if you are searching for "Joe Smith," you would look under the "s" tab.

Instead of specifying a unique location for each main memory block, we can look at the opposite extreme: allowing a main memory block to be placed anywhere in cache. The only way to find a block mapped this way is to search all of cache. (This is similar to your author's desk!) This requires the entire cache to be built from *associative memory* so it can be searched in parallel. That is, a single search must compare the requested tag to *all* tags in cache to determine whether the desired data block is present in cache. Associative memory requires special hardware to allow associative searching, and is, thus, quite expensive.

Using associative mapping, the main memory address is partitioned into two pieces, the tag and the word. For example, using our previous memory configuration with 2^{14} words, a cache with 16 blocks, and blocks of 8 words, we see from Figure 6.8 that the word field is still 3 bits, but now the tag field is 11 bits. This tag must be stored with each block in cache. When the cache is searched for a specific main memory block, the tag field of the main memory address is compared to all the valid tag fields in cache; if a match is found, the block is found. (Remember, the tag uniquely identifies a main memory block.) If there is no match, we have a cache miss and the block must be transferred from main memory.

With direct mapping, if a block already occupies the cache location where a new block must be placed, the block currently in cache is removed (it is written back to main memory if it has been modified or simply overwritten if it has not been changed). With fully associative mapping, when cache is full, we need a replacement algorithm to decide which block we wish to throw out of cache (we call this our *victim block*). A simple first-in, first-out algorithm would work, as

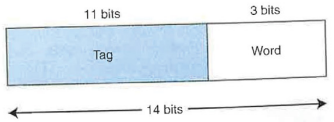


FIGURE 6.8 The Main Memory Address Format for Associative Mapping

would a least-recently used algorithm. There are many replacement algorithms that can be used; these are discussed shortly.

Set Associative Cache

Owing to its speed and complexity, associative cache is very expensive. Although direct mapping is inexpensive, it is very restrictive. To see how direct mapping limits cache usage, suppose we are running a program on the architecture described in our previous examples. Suppose the program is using block 0, then block 16, then 0, then 16, and so on as it executes instructions. Blocks 0 and 16 both map to the same location, which means the program would repeatedly throw out 0 to bring in 16, then throw out 16 to bring in 0, even though there are additional blocks in cache not being used. Fully associative cache remedies this problem by allowing a block from main memory to be placed anywhere. However, it requires a larger tag to be stored with the block (which results in a larger cache) in addition to requiring special hardware for searching of all blocks in cache simultaneously (which implies a more expensive cache). We need a scheme somewhere in the middle.

The third mapping scheme we introduce is *N-way set associative cache mapping*, a combination of these two approaches. This scheme is similar to direct mapped cache, in that we use the address to map the block to a certain cache location. The important difference is that instead of mapping to a single cache block, an address maps to a *set* of several cache blocks. All sets in cache must be the same size. This size can vary from cache to cache. For example, in a 2-way set associative cache, there are two cache blocks per set, as seen in Figure 6.9. In this figure, we see that set 0 contains two blocks, one that is valid and holds the data A, B, C, . . . , and another that is not valid. The same is true for Set 1. Set 2 and Set 3 can also hold two blocks, but currently, only the second block is valid in each set. In an 8-way set associative cache, there are 8 cache blocks per set. Direct mapped cache is a special case of N-way set associative cache mapping where the set size is one.

In set associative cache mapping, the main memory address is partitioned into three pieces: the tag field, the set field, and the word field. The tag and word fields assume the same roles as before; the set field indicates into which cache set the main memory block maps. Suppose we are using 2-way set associative mapping with a main memory of 2^{14} words, a cache with 16 blocks, where each block contains 8 words. If cache consists of a total of 16 blocks, and each set has 2

Set	Tag	Block 0 of set	Valid	Tag	Block 1 of set	Valid
0	00000000	Words A, B, C, . . .	1	----------		0
1	11110101	Words L, M, N, . . .	1	----------		0
2	----------		0	10111011	P, Q, R, . . .	1
3	----------		0	11111100	T, U, V, . . .	1

FIGURE 6.9 A Two-Way Set Associative Cache



FIGURE 6.10 Format for Set Associative Mapping

blocks, then there are 8 sets in cache. Therefore, the set field is 3 bits, the word field is 3 bits, and the tag field is 8 bits. This is illustrated in Figure 6.10.

6.4.2 Replacement Policies

In a direct-mapped cache, if there is contention for a cache block, there is only one possible action: The existing block is kicked out of cache to make room for the new block. This process is called *replacement*. With direct mapping, there is no need for a replacement policy because the location for each new block is predetermined. However, with fully associative cache and set associative cache, we need a replacement algorithm to determine the "victim" block to be removed from cache. When using fully associative cache, there are K possible cache locations (where K is the number of blocks in cache) to which a given main memory block may map. With N-way set associative mapping, a block can map to any of N different blocks within a given set. How do we determine which block in cache should be replaced? The algorithm for determining replacement is called the *replacement policy*.

There are several popular replacement policies. One that is not practical but that can be used as a benchmark by which to measure all others is the *optimal* algorithm. We like to keep values in cache that will be needed again soon, and throw out blocks that won't be needed again, or that won't be needed for some time. An algorithm that could look into the future to determine the precise blocks to keep or eject based on these two criteria would be best. This is what the optimal algorithm does. We want to replace the block that will not be used for the longest period of time in the future. For example, if the choice for the victim block is between block 0 and block 1, and block 0 will be used again in 5 seconds, whereas block 1 will not be used again for 10 seconds, we would throw out block 1. From a practical standpoint, we can't look into the future—but we can run a program and then rerun it, so we effectively *do* know the future. We can then apply the optimal algorithm on the second run. The optimal algorithm guarantees the lowest possible miss rate. Because we cannot see the future on every single program we run, the optimal algorithm is used only as a metric to determine how good or bad another algorithm is. The closer an algorithm performs to the optimal algorithm, the better.

We need algorithms that best approximate the optimal algorithm. We have several options. For example, we might consider temporal locality. We might guess that any value that has not been used recently is unlikely to be needed again