lates either to a circuit or a procedure. Therefore, each instruction should perform a unique function without duplicating any other instruction. Some people refer to this characteristic as **orthogonality**. In actuality, orthogonality goes one step further. Not only must the instructions be independent, but the instruction set must be consistent. For example, orthogonality addresses the degree to which operands and addressing modes are uniformly (and consistently) available with various operations. This means the addressing modes of the operands must be independent from the operands (addressing modes are discussed in detail in Section 5.4.2). Under orthogonality, the operand/opcode relationship cannot be restricted (there are no special registers for particular instructions). In addition, an instruction set with a multiply command and no divide instruction would not be orthogonal. Therefore, orthogonality encompasses both independence and consistency in the instruction set. An orthogonal instruction set makes writing a language compiler much easier; however, orthogonal instruction sets typically have quite long instruction words (the operand fields are long due to the consistency requirement), which translates to larger programs and more memory use.

5.4 ADDRESSING

Although addressing is an instruction design issue and is technically part of the instruction format, there are so many issues involved with addressing that it merits its own section. We now present the two most important of these addressing issues: the types of data that can be addressed and the various addressing modes. We cover only the fundamental addressing modes; more specialized modes are built using the basic modes in this section.

5.4.1 Data Types

Before we look at how data is addressed, we will briefly mention the various types of data an instruction can access. There must be hardware support for a particular data type if the instruction is to reference that type. In Chapter 2 we discussed data types, including numbers and characters. Numeric data consist of integers and floating-point values. Integers can be signed or unsigned and can be declared in various lengths. For example, in C++ integers can be *short* (16 bits), *int* (the word size of the given architecture), or *long* (32 bits). Floating-point numbers have lengths of 32, 64, or 128 bits. It is not uncommon for ISAs to have special instructions to deal with numeric data of varying lengths, as we have seen earlier. For example, there might be a `MOVE` for 16-bit integers and a different `MOVE` for 32-bit integers.

Nonnumeric data types consist of strings, Booleans, and pointers. String instructions typically include operations such as copy, move, search, or modify. Boolean operations include AND, OR, XOR, and NOT. Pointers are actually addresses in memory. Even though they are, in reality, numeric in nature, pointers are treated differently than integers and floating-point numbers. MARIE allows 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 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 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.

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 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.

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.

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.

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, 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.

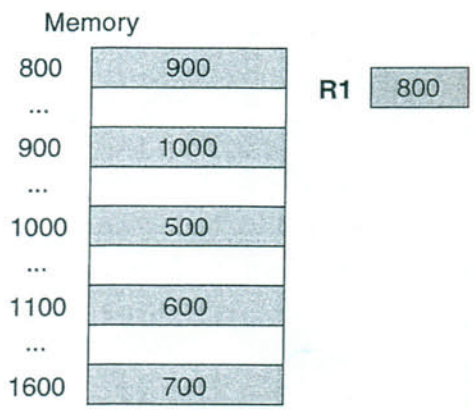

FIGURE 5.3 Contents of Memory When `Load 800` Is Executed

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 5.3

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

How does the computer know which addressing mode is supposed to be used for a particular operand? We have already seen one way to deal with this issue. In MARIE, there are two `JUMP` instructions—a `JUMP` and a `JUMPI`. There are also two add instructions—an `ADD` and an `ADDI`. The instruction itself contains information the computer uses to determine the appropriate addressing mode. Many languages have multiple versions of the same instruction, where each variation indicates a different addressing mode and/or a different data size.

Encoding the address mode in the opcode itself works well if there is a small number of addressing modes. However, if there are many addressing modes, it is better to use a separate address specifier, a field in the instruction with bits to indicate which addressing mode is to be applied to the operands in the instruction.

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 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).

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:

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

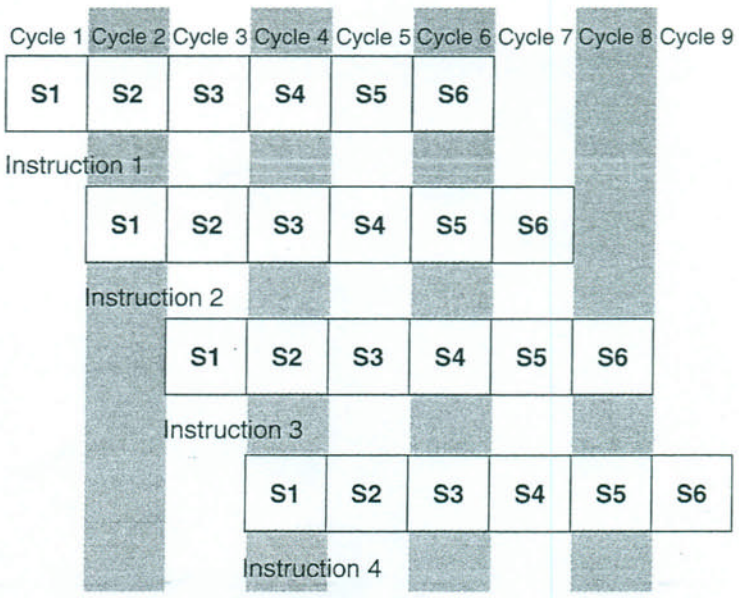

FIGURE 5.4 Four Instructions Going through a 6-Stage Pipeline

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

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