delays. Throughput can be enhanced in a serial manner by trying to execute a desired function faster. If each function is executed at a faster clock frequency, more functions can be executed in a given time period. An alternative parallel approach can be taken whereby multiple functions are executed simultaneously, thereby improving performance over time. These two approaches can be complementary in practice. Different logic implementations make use of serial and parallel enhancement techniques in the proportions and manners that are best suited to the application at hand.

A logic function is represented by a set of Boolean equations that are then implemented as discrete gates. During one clock cycle, the inputs to the equations are presented to a collection of gates via a set of input flops, and the results are clocked into output flops on the next rising edge. The propagation delays of the gates and their interconnecting wires largely determine the shortest clock period at which the logic function can reliably operate.

Pipelining, called *superpipelining* when taken to an extreme, is a classic serial throughput enhancement technique. Pipelining is the process of breaking a Boolean equation into several smaller equations and then calculating the partial results during sequential clock cycles. Smaller equations require fewer gates, which have a shorter total propagation delay relative to the complete equation. The shorter propagation delay enables the logic to run faster. Instead of calculating the complete result in a single 40 ns cycle, for example, the result may be calculated in four successive cycles of 10 ns each. At first glance, it may not seem that anything has been gained, because the calculation still takes 40 ns to complete. The power of pipelining is that different stages in the pipeline are operating on different calculations each cycle. Using an example of an adder that is pipelined across four cycles, partial sums are calculated at each stage and then passed to the next stage. Once a partial sum is passed to the next stage, the current stage is free to calculate the partial sum of a completely new addition operation. Therefore, a four-stage pipelined adder takes four cycles to produce a result, but it can work on four separate calculations simultaneously, yielding an average throughput of one calculation every cycle—a four-times throughput improvement.

Pipelining does not come for free, because additional logic must be created to handle the complexity of tracking partial results and merging them into successively more complete results. Pipelining a 32-bit unsigned integer adder can be done as shown in Fig. 7.8 by adding eight bits at a time and then passing the eight-bit sum and carry bit up to the next stage. From a Boolean equation perspective, each stage only incurs the complexity of an 8-bit adder instead of a 32-bit adder, enabling it to run faster. An array of pipeline registers is necessary to hold the partial sums that have been calculated by previous stages and the as-yet-to-be-calculated portions of the operands. The addition results ripple through the pipeline on each rising clock edge and are accumulated into a final 32-bit result as operand bytes are consumed by the adders. There is no feedback in this pipelined adder, meaning that, once a set of operands passes through a stage, that stage no longer has any involvement in the operation and can be reused to begin or continue a new operation.

Pipelining increases the overall throughput of a logic block but does not usually decrease the calculation latency. High-performance microprocessors often take advantage of pipelining to varying degrees. Some microprocessors implement superpipelining whereby a simple RISC instruction may have a latency of a dozen or more clock cycles. This high degree of pipelining allows the microprocessor to execute an average of one instruction each clock cycle, which becomes very powerful at operating frequencies measured in hundreds of megahertz and beyond.

Superpipelining a microprocessor introduces complexities that arise from the interactions between consecutive instructions. One instruction may contain an operand that is calculated by the previous instruction. If not handled correctly, this common circumstance can result in the wrong value being used in a subsequent instruction or a loss of performance where the pipeline is frequently stalled to allow one instruction to complete before continuing with others. Branches can also cause havoc with a superpipelined architecture, because the decision to take a conditional branch may nullify the few instructions that have already been loaded into the pipeline and partially executed. De-

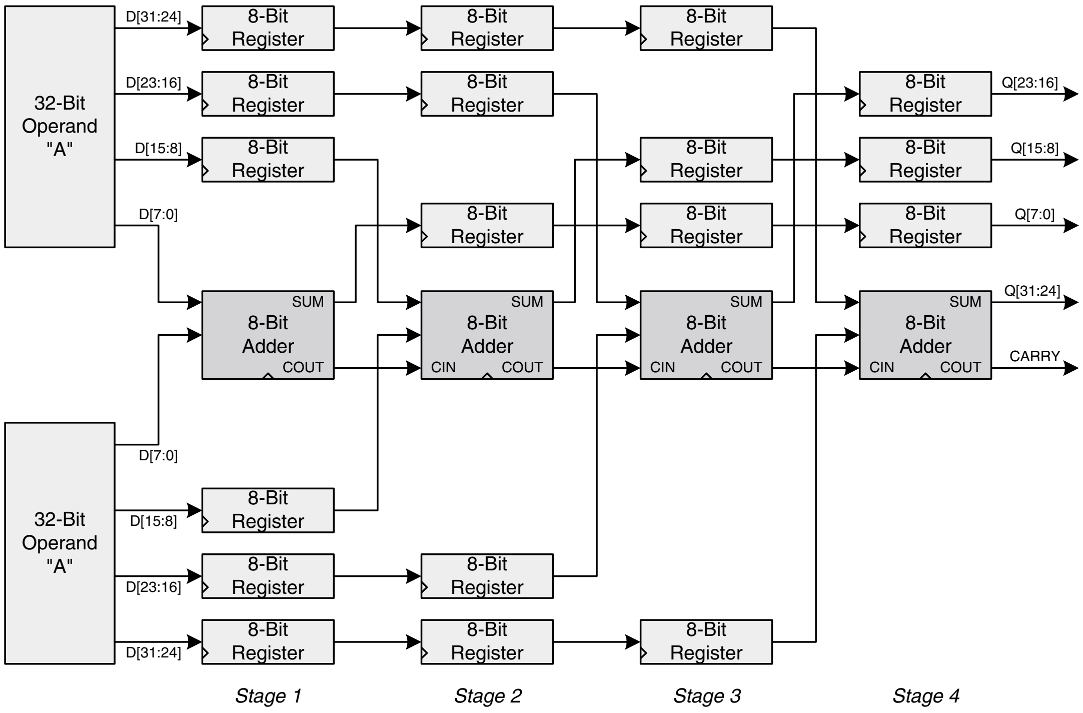

FIGURE 7.8 Four-stage pipelined adder.

pending on how the microprocessor is designed, various state information that has already been modified by these partially executed instructions may have to be rolled back as if the instructions were never fetched. Branches can therefore cause the pipeline to be flushed, reducing the throughput of the microprocessor, because there will be a gap in time during which new instructions advance through the pipeline stages and finally emerge at the output.

Traditional microprocessor architecture specifies that instructions are executed serially in the order explicitly defined by the programmer. Microprocessor designers have long observed that, within a given sequence of instructions, small sets of instructions can be executed in parallel without changing the result that would be obtained had they been executed in the traditional serial manner. *Superscalar* microprocessor architecture has emerged as a means to execute multiple instructions simultaneously within a single microprocessor that is operating on a normal sequence of instructions. A superscalar architecture contains multiple independent execution units, some of which may be identical, that are organized and replicated according to statistical studies of which instructions are executed more often and how easily they can be made parallel without excessive restrictions and dependencies. Arithmetic execution units are prime targets for replication, because calculations with floating-point numbers and large integers require substantial logic and time to fully complete. A superscalar microprocessor may contain two integer ALUs and separate FPUs for floating-point addition and multiplication operations. Floating-point operations are the most complex instructions that many microprocessors execute, and they tend to have long latencies. Most floating-point applications contain a mix of addition and multiplication operations, making them well suited to an architecture with individual FPUs that each specialize in one type of operation.