COMPUTE MODELS AND SYSTEM ARCHITECTURES

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Field-programmable gate array (FPGA) and reconfigurable architectures provide enormous raw computing power and tremendous flexibility. How do we best exploit this opportunity and bring it to bear on particular computing tasks? When we do take advantage of the flexibility, and how do we ensure correctness? How do we preserve and reuse our designs as technology continues to advance? The raw size and flexibility of today's devices and systems make these questions daunting to consider and intractable to approach in an undisciplined manner. In this chapter, we review models and organizational styles for large-scale, highly parallel computing resources and emphasize how they can be used in the organization of reconfigurable computers.

A modern FPGA has hundreds of thousands of independently configured bit-processing units and hundreds of memories. Today's multi-FPGA systems and future single-chip FPGAs raise these numbers to millions of bit-processing units and thousands or tens of thousands of memories. Furthermore, configurable interconnect allows us to arrange these resources in almost any manner. This gives us the power to adapt the computation to a particular task. Now that we have that power, what do we do with it?

Developing large software applications is a known hard problem, and managing resources and computations in highly parallel systems is, notoriously, even harder. Without care, our parallel computations may behave differently on each execution, producing nondeterministic results, some of which may be erroneous, and some executions may lead to deadlock. Unconstrained, the additional flexibility that comes with parallelism increases the complexity of application development and verification.

Considering both the limits of the human mind and the desire to achieve reasonably low time-to-solution periods, we cannot afford to custom-tailor each 4-LUT and each memory. With industry producing new devices according to Moore's Law, we cannot afford to design for 100,000 4-LUTs one year, discard the design, and then redesign for 200,000 4-LUTs three years later when the next part becomes available. Nor can we afford to reason about the interaction of every individual 4-LUT with every other—a number of interactions that grows quadratically with resource count.

The good news is that, while there is almost unbounded freedom in how we might solve problems, there are a small number of high-level organizational strategies that suffice to describe and efficiently implement most computing tasks. To bridge the semantic gap between applications and FPGA resources, we should think about two abstractions:

- *Compute models*—high-level models of the flow of computation in an application, useful for capturing parallelism and reasoning about correctness of implementations.
- *System architectures*—high-level strategies for organizing resources, managing the parallelism in the implementation, and facilitating optimization and design scaling.

Within each system architecture, there remains considerable flexibility to tailor the computing resources to the particular task, exploiting the flexibility of the architecture's reconfigurability. The compute model provides high-level constraints and guidance for conceptualizing the problem, reasoning about its correctness, and supporting manual and automated optimization. Chosen properly, the compute model naturally captures the parallelism of the application, making it easier to reason about its description and mapping.

A diversity of compute models and system architectures is needed to capture the diversity of natural organizations and implementations of tasks. Nonetheless, evidence to date suggests that there are only a modest number, perhaps tens of each, necessary to do this. Mismatches between the compute model and the task increase the complexity and awkwardness of the design and limit scalability. However, a good designers will be aware of the variety of compute models and system architectures and judiciously select the ones that naturally match her problem.

For decades, software engineers have faced the problem of managing complexity in large, highly concurrent software systems. *Software architectures* [1] were developed as one of the organizational tools to manage the complexity and to guide the design of these systems. The *system architectures* identified here are a deliberate expansion and adaptation of software architecture for reconfigurable computing, and many of the challenges are identical. However, the additional flexibility of reconfigurable architectures opens up design options and tradeoffs not typically present in the conventional multiprocessor systems for which software architectures have been traditionally targeted.

The two main sections in this chapter introduce, respectively, compute models and system architectures relevant to reconfigurable computing. For the reader approaching these topics for the first time, it may make sense to read the introductory sections, giving the detailed sections only a cursory review, for a high-level understanding of why we need a variety of models and architectures. As one delves further into reconfigurable designs or has a particular application in mind to solve, the in-depth sections can serve as a reference guide and provide deeper consideration of the merits and suitability of each approach.

5.1 COMPUTE MODELS

Figure 5.1 provides a taxonomy of the major compute models discussed and refined in this chapter. The leftmost branch is a set of models organized around the flow of data between operators; in these we think about the computation as a graph of computational operators and we reason about the correctness and assembly of operation in terms of data arrival at the operators, the function performed on the data, and the result produced and forwarded to other operators. The rightmost branch is a set of models organized around synchronous steps for the entire machine; here we think about the computation as a sequence of, perhaps parallel, operations performing transformation to global state.

At the top of the figure is a generic multi-threaded model or, formally, a model such as Hoare's Communicating Sequential Processes (CSP) [2]. All of the models below can be seen as refinements and stylizations on it. The multi-threaded model gives little guidance to the programmer on how to organize and design programs. Consequently, each of the refinements takes a stronger stand on how computation and parallelism are organized and how we manage synchronization. In many cases the refined models come with greater opportunities for optimization and stronger verification guarantees.

As we will see, system architectures are typically built on some of the same distinctions identified here in compute models (e.g., sequential control versus dataflow). However, there is not necessarily a one-to-one matching between the compute model used for capturing and reasoning about the application and the system architecture used for implementation. For example, modern superscalar microprocessors efficiently execute sequential instructions streams using dataflow techniques (e.g., Tomasulo [3]), and digital signal processors (DSPs) execute synchronous dataflow graphs as a sequence of instructions.

5.1.1 Challenges

When approaching a problem, we want to know how to implement the desired computation correctly, with the least effort, while exploiting the available

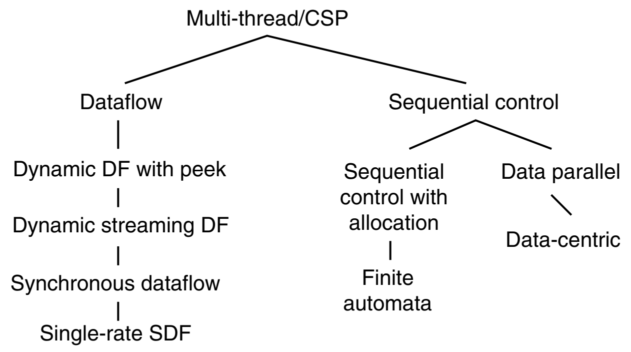

FIGURE 5.1 ■ Overview of compute models.

hardware capabilities on current and future machines. We can decompose this into specific challenges in selecting a compute model and developing the implementation:

- How do we think about composing the application?
- How does the compute model naturally lead to efficient, spatial solutions?
- How does the compute model support design (de)composition?
- How do we conceptualize parallelism?
- How do we trade area for time in the compute model?
- How do we reason about correctness?
- How do we deal with technology effects and adapt to technology changes?
- How does the compute model provide or guarantee determinacy?
- How do we avoid deadlock?
- What can we compute?
- How complex is it to optimize or validate properties of the application?

The first thing the compute model gives us is a way to think about the application. For example: Should we think of the application as a sequence of operations that need to be performed (sequential control)? As applying an operation to a set of independent data items (data parallel)? As a set of transformations on a data sequence (streaming dataflow)? To the extent these questions provide a natural way to describe the application, they make it easier to compose the application, identify the natural parallelism, and reason about correctness and transformations. Ideally, the compute model acts as part of the bridge between the application and the reconfigurable platform, providing a modest semantic gap between the application and the system architecture. The system architecture then brings to bear a large set of knowledge, accumulated across many applications, about how to efficiently bridge the gap between the compute model and the reconfigurable platform.

Reconfigurable platforms are most efficient when we can arrange for each resource to do the same thing over and over, and keep most of the resources active doing exactly the work needed for the computation. The compute model should allow us to capture computations that can exploit this; further, it should encourage the developer to express applications in a manner amenable to this kind of computation. The restrictions and stylizations in some compute models may limit the freedom in expressing an algorithm. However, limiting expressive freedom that would lead to poor reconfigurable solutions is one of the ways that a good compute model provides assistance and guidance. If the limitations make a solution hard to express, that can be good guidance that the solution approach is not well suited to a reconfigurable platform or that the compute model is not the natural choice for the task.

A good compute model helps us decompose a problem into components that can be designed and validated independently. This helps avoid the quadratic explosion in complexity arising from potentially interacting resources, and even avoid the linear cost necessary if we had to program each resource independently. Sequential control models may focus on sequences of subroutines; dataflow models, on composition of functions; and streaming dataflow models, on hierarchical operator graphs.

Important to identify early is where the parallelism exists in an application. Is it between data items (data parallelism), between coarse-grained tasks (task parallelism), between operators in a task (instruction-level parallelism), or within low-level arithmetic and binary operations (bit-level parallelism)? Identifying and exposing these opportunities assists area-time tradeoffs: On small, economical platforms, we can tune a task for the modest area at the expense of longer runtime, while on larger platforms we might exploit the additional area to reduce compute time. Parallelism shows up implicitly or explicitly in each compute model, and a good match in parallelism will facilitate successful application scaling.

One of the most important tools provided by each compute model is a way to reason about correctness, which ultimately facilitates scaling, implementation adaptation, and optimization because it defines what transformations are possible without impacting correctness. In a sequential control model we identify the visible state on each step and reason about the changes in it; in a streaming dataflow model we reason about the output sequence of a computational graph.

With rapidly advancing technology, the size, speed, and energy of computing primitives (e.g., gates, wires, memories) are changing continually as they move from platform to platform. Sometimes they move together, with compute, interconnect, and memory speeds all growing uniformly smaller. Often, however, they change at different rates. As vendors have optimized memory for density and logic for speed, relative speeds have diverged, and, as we reach into the deep submicron regime, interconnect scales more slowly than compute. As a result, simply moving an old design to a new platform is unlikely to optimally exploit it. With increasing interconnect delays, perhaps the design needs more pipelining to distant locations; with slower memories, perhaps it needs more parallel memory blocks servicing a compute block. The compute model helps us understand the transformations permissible for the design, which may point to techniques the system architecture can employ for tolerating changes in constituent delays. Stall signals, for example, allow sequential control to slow down only when uncommon operations run at slower speeds than the scaled speed of the rest of the logic; data presence (see Data presence subsection of Section 5.2.1) allows streaming dataflow computations to tolerate variable delays within and between operators.

Given the same set of inputs, we might want our computation to produce the same outputs. That is, we often want our computation to be *deterministic*. Certainly, if the result of the computation differs each time it is performed, it becomes harder to debug our application or demonstrate its correctness. This can be a mild problem with sequential applications, where dependence on dynamic effects (e.g., dynamically allocated addresses) may change the program behavior; it becomes acute in concurrent systems. If there is variability in the relative timing of operations, the order of events can change, and without care this may result in different visible application behavior.

Further, as we scale to different hardware capacities, we may exploit different amounts of concurrency and deliberately change the order of primitive events. Nonetheless, we might want to guarantee that the application remains deterministic, providing the same results for any legal parallelism. Some compute models with limited constraints may not be able to guarantee such determinacy but place this burden on the individual programmer. Most, however, come with disciplines that the developer can use to provide determinism, and some come with a sufficient set of model restrictions to automatically guarantee it.

Still, sometimes we want or need nondeterminism to deal with variations in the outside world (e.g., waiting for human input) or with deliberate variations to avoid bad behavior (e.g., randomized algorithms). Sometimes, too, there are multiple "correct" results and it is efficient to allow the system to select any of them, perhaps in a way that looks nondeterministic to the application as a whole. The point here is that nondeterminism always adds complexity to construction and validation, so it should be used sparingly and with care [4]

When dealing with shared or limited resources or variable operations in concurrent systems, we must also watch out for *deadlock*; in other words, we must watch for cases where the system may enter a state that prevents it from making forward progress. Often deadlock occurs when we attempt to give exclusive access to resources in an application. If a set of tasks end up waiting for each other—that is, the task set has a dependent cycle waiting for resources—the tasks can become deadlocked and the application will never complete. This can happen in purely deterministic computations, but should be at least identified by reasonably testing if the paths through the code are largely data independent. However, if the paths are largely data dependent, and deadlock only occurs for certain data values, identifying it with *ad hoc* testing can be difficult. When resource allocation and sequencing are nondeterministic, avoiding deadlock can be even more tricky. For these reasons, it is necessary to carefully guarantee that none of the legal, nondeterministic choices leads to a deadlock situation.

Computational theory gives us a well-developed set of models for computation. The Church–Turing Thesis [5–7] suggests that there is a very robust class of computing models that are all equivalent to the Turing Machine or the Lambda Calculus model. In fact, most of the models discussed here are Turing Complete. However, some refinements, such as synchronous dataflow (see Synchronous dataflow subsection of Section 5.1.3) or finite-state sequential control (see Finite state subsection of Section 5.1.4) models, are specifically less powerful. As will be noted, these restricted models give up expressive power in order to gain more powerful optimization and analysis.

We want to be able to say that an application always has certain properties. Ideally, we can verify that our expression of the application is correct, or, more specifically, that our captured algorithm is deterministic or that it can never deadlock. Further, to facilitate automated optimization and area–time scaling, we must guarantee that any changes made to the implementation preserve determinism and freedom from deadlock. Thus, we are ultimately concerned with the computational tractability of verification and optimization. In Turing–Complete compute models, where anything is allowed, verification and general optimization can be *undecidable*; that is, without solving the *halting problem* it is not possible to analyze the design and say whether or not it is correct, determinate, or deadlock free. In more restricted models, verification or optimization may be *decidable* but *NP-hard*, meaning that we know of no polynomial time solutions to perform the optimization. And in even more restricted models, verification and optimization may be polynomial time. Consequently, we have a trade-off between the expressiveness and the strength of automation we can bring to bear on the problem, which suggests that the designer carefully select compute models that are expressive enough for her problem but not unnecessarily so.

5.1.2 Common Primitives

Two common primitives useful for defining and reasoning about compute models are *functions* and *objects*.

Function

A *function* is simply a deterministic, mathematical function that maps each finite input to a finite output:

$$Y = [y_0, y_1, \ldots, y_n] = f(X = [x_0, x_1, \ldots, x_m])$$

A function depends on no hidden state but only the input arguments to it, and it modifies no state values. Examples include addition, square root, and discrete-cosine transform (DCT). Functions can be composed, and the result is another function. For example:

$$y = (f \circ g)(x) = f(g(x))$$

Functions are interesting as a building block for several reasons:

- Functions are a useful formal primitive for defining computational models.
- Functional operations can be a tool or clue to parallelism—since functions do not modify state, they may be evaluated in parallel; evaluation of functions on different data can often be heavily pipelined.
- Functions can be a tool or guide to recurrent computations—those that show up regularly in the description of a computation are candidates for computational blocks that can be profitably implemented in spatial reconfigurable logic.

Transform or object

We can associate state with a function in order to create a common building block we can think of as a *transform,* or a primitive version of an *object*. In signal processing, we might think of a general transform as taking a sequence of inputs and computing outputs based on them as well as on some finite state from the previous output:

$$Y_i = f(X_i, Y_{i-1})$$

In an *object-oriented* model, we might think of the object, O, being the combination of state, $O.s$, and a function, $O.f$, with each invocation evaluating the function on the input and the state and returning an output and a new state value:

$$Y, O.s_i = O.f(X, O.s_{i-1})$$

Examples of transforms include accumulators, finite-impulse response filters (FIRs), infinite impulse response filters (IIRs), and linear-feedback shift registers (LFSRs).

This primitive object or transform is more powerful than a pure function, but the inclusion of state may restrict its freedom of usage and implementation. As described, the state is finite, and each object can be viewed as a finite automata. The model says that the sequential invocations of an object see the state from the previous invocation; this demands that we complete the function's evaluation before starting the next invocation—or, at least, that we provide an implementation that produces the same net output sequence and state updates as though we had done so. For simple functions (e.g., LFSRs) or those where the state can be maintained without computation (e.g., FIRs), we can still pipeline the operation heavily. However, for complex functions (e.g., IIRs) the state feedback may limit our ability to heavily pipeline the object.

Nonetheless, object state is owned by the object, so evaluation of an object affects no others. Consequently, distinct objects with a complete set of inputs can evaluate in parallel; they impact each other only by communicating values between them. Further, objects with the same function may be able to share the same hardware to create commonality. This is useful both for enabling area–time trade-offs and for keeping a spatial datapath active in repeatedly performing the same operations. If sequential dependencies within an operator limit pipelining and we have many objects of the same type, it may be possible to C-slow the function evaluation (Chapter 18) to use the same hardware to service multiple objects.

In rich object-oriented models, we may associate additional capabilities with objects. We will introduce some of these as we explore more powerful compute models in the following sections.

5.1.3 Dataflow

We begin our detailed discussion of compute models with the left branch in Figure 5.1. In these models, we reason about the computation based on the flow of data. Computations are performed by *operators*, which can be either functions or objects as defined previously. We connect the operators into a graph, linking the output data from one to the input data of another. When its inputs arrive,

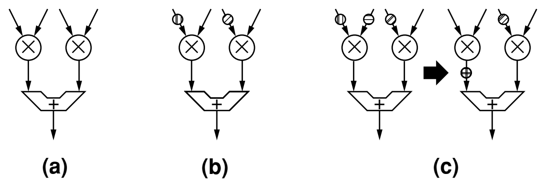

FIGURE 5.2 ■ Computation on a dataflow graph: (a) graph without inputs, (b) graph with partial inputs, and (c) arrival of matched input on left ×-operator allows it to evaluate and compute its output.

an operator can evaluate, produce its outputs, and send them to any operators connected to it (see Figure 5.2).

The dataflow graph exposes considerable parallelism and freedom in evaluation permitted to an implementation. The links capture the communication and dependence structure of the computation explicitly.

There is a large hierarchy of dataflow models with different flexibilities and challenges. For example, the simple models can be easily mapped to spatial, reconfigurable computation. The more flexible and powerful models are more complicated to implement efficiently, and make it difficult to guarantee correctness. However, for some applications, these more powerful models may be essential to efficiently describing and executing an application.

Single-rate synchronous dataflow

One of the most primitive dataflow models is that of a static graph of operators. The graph is created once, before the application executes, and persists unchanged throughout execution. In contrast, in the Streaming dataflow with allocation subsection (see page 102), we will consider models that allow the dataflow graph to change as part of the computation. We call the persistent edges between operators *streams* or *pipes*, as they deliver a sequence of values from a single producer to a single consumer, and we identify each value carried over these streams as a *token*. Such a graph of operators can itself be viewed as an operator, so this provides a model for composition of more powerful operators from more primitive functions and objects (see Figure 5.3). Computationally, this still provides the power of a finite automata, but the dataflow view is often a more natural way to describe, compose, and reason about the computation.

Synchronous dataflow

In single-rate synchronous dataflow, we assume that each transform operator takes in a single set of input tokens and produces a single set of output tokens. It is a simple generalization to allow the model to take in multiple tokens on a single stream link or to produce multiple tokens on an output stream link for one logical evaluation of the function. For example, a down-sample operator might read two inputs and only output one value, discarding every other input token. The number of inputs received from each input stream, or outputs produced on each output stream, can be different; for example, an operator might read two A tokens for every B token. However, as long as there are a constant number of

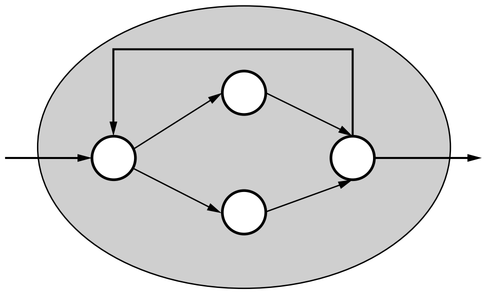

FIGURE 5.3 ■ A single-rate static dataflow graph.

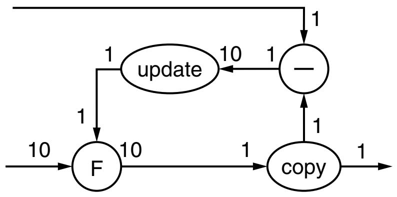

FIGURE 5.4 ■ A multirate dataflow graph.

tokens consumed from each input stream and tokens produced on each output stream on each such evaluation, the model retains the same power as before, but it now allows us to efficiently express multirate streaming applications; that is, some loops in the dataflow graph can operate at much lower frequency than others.

An inner loop might execute on every input to the graph, while an outer loop might perform updates only once every 10 inputs as shown in Figure 5.4. The numbers on the operator I/Os in Figure 5.4 indicate the rate of I/O consumption or production. The update module produces a single output every 10 tokens; the F function consumes a single input from update every tenth data input and output token; and the copy and subtract units each produce a single set of output tokens for each set of input tokens.

This is the Synchronous Dataflow (SDF) model [8], and it retains the same computational power of a finite automata. However, it allows multirate designs to be expressed more efficiently, explicitly identifying the relative operating rates of each of the computational functions in the graph. An implementation can use this information when provisioning operators and scheduling the sharing of physical resources. The computation is completely deterministic, and it is possible to automatically identify when operator rates are mismatched, leading to deadlock, and to automatically identify any buffering necessary during execution [9].

Dynamic streaming dataflow

Synchronous dataflow retains analysis simplicity because there is no data dependence in the consumption or production of tokens. Every evaluation of an object consumes and produces the same number of tokens regardless of the data.

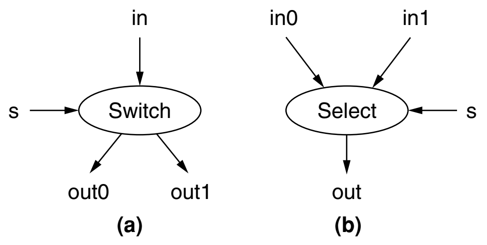

FIGURE 5.5 ■ The dynamic dataflow primitives—*switch* (a) and *select* (b).

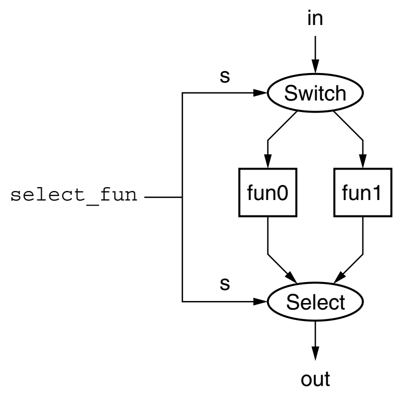

FIGURE 5.6 ■ Data-driven function selection in the dataflow model.

A more general model allows the production of input and output tokens to depend on the object state or the values of the inputs. We can fully capture this additional power by adding the *switch* and *select* operators, shown in Figure 5.5, to a persistent object graph.

In the figure, these two operators are data dependent, producing data on only one output, or consuming the inputs selectively, based on an input value. Equivalently, this can be captured by generalizing the notion of an object to allow its state to determine the token consumption and production actually performed on each evaluation. This allows us to efficiently deal with data-dependent cases, such as the following:

- Performing different operations based on the data (Figure 5.6).
- Varying the rate of the output relative to the input, such as in a compressor or decompresser (e.g., Huffman encoder).
- Iterating a computation a variable number of times to yield convergence (e.g., Newton–Raphson method for finding roots of equations).

In some cases these operations can be data independent, but only at the expense of more work (e.g., evaluating both functions in Figure 5.6 and then discarding one result). However, if no constant bound can be placed on the iterations (e.g., the number of cycles required for convergence), data dependence is a necessity, not just an efficiency optimization.

The addition of data-dependent operators changes the power of the streaming dataflow, making it more difficult to analyze statically. The computation remains deterministic, but data-dependent production and consumption rates on operators necessitate reasoning about the streams as first-in-first-out (FIFO) token buffers. The addition of unbounded buffers between operators is sufficient to make the model Turing Complete, and it is no longer always possible to determine the FIFO buffers' required capacity. If the implementation buffer capacity is too small, the application may artificially deadlock. This demands either that the developer identify the necessary buffer size to avoid deadlock for each application or that the implementation provide dynamic support to allow arbitrary buffer expansion at runtime [10].

Dynamic Streaming Dataflow with Peeks

So far, we have demanded that the object evaluate based on a valid set of input tokens. In the data-dependent case, we allowed the value of the present tokens to determine which other tokens were consumed. We can further allow the operator to perform an action or modify state based on the absence of a token; that is, we can allow it to *peek* to see if an input is present. For example, a merge unit might have two inputs and forward either token to its output whenever there is some input present. As the merge unit example suggests, this creates new freedom for efficient evaluation but also introduces nondeterminism. The operator can now behave differently based on the arrival timing of its inputs. The data-dependent streaming dataflow model discussed earlier only introduced concern about deadlock but remained deterministic. The Dynamic Streaming Dataflow with Peeks model forces the developer to manage determinacy.

Streaming dataflow with allocation

The parallelism in the application is, in general, data dependent. Consequently, it can be useful for the operator dataflow graph to evolve on the basis of the data in a computation. In a telecommunications application, the number and type (e.g., voice, data) of connections change over time. Each channel has its own noise characteristics, perhaps requiring filter complexity (e.g., number of taps, length of echo cancellation) different from the others'. To accommodate these changes in the computational demand of an application over time, we must change the dataflow operator graph. We could force the graph construction to a different compute model and stay with graph evaluation as one of the models reviewed earlier. Alternately, we must expand the compute model with the ability to create new operators and link them into the graph.

The key addition now is for our operators to be able to perform instantiation (e.g., `new`) of operators and streams and to be able to connect them. Even if operators remain finite state, instantiation provides the ability to create unbounded state by growing the object graph to unbounded size, with arbitrary data structures implemented as subgraphs. This provides a more efficient path to achieving Turing Completeness than the unbounded buffers in dynamic streaming dataflow.

While powerful, dynamic allocation means that the logical graph is changing during execution, and with dynamically changing computational graphs, it is no longer possible to optimize, schedule, place, and route them before execution. As a result, dynamic allocation gives us a model for the application to change during execution that can exploit the capabilities of a reconfigurable computing platform. However, it can also force a need for reconfiguration during execution, so allocation should be used with care. If it is infrequent, and allocated objects are long-lived, the cost of runtime management and reconfiguration can be amortized out over long usage periods.

General dataflow

Once we add allocation of operators, the model becomes powerful enough to be used as general dataflow computation. Some dataflow models do not treat operators or links as persistent (e.g., Arvind and Nikhil [11] and Culler et al. [12]). Rather, the dataflow is instantiated during a function or object call, used once, and then it is disposed. This does not change the model, but it does change the relative rate of allocation versus dataflow usage in a significant way. On typical reconfigurable platforms, dataflow construction is expensive, making it more difficult to efficiently map models that dispose of and reconstruct dataflow. For efficient execution on a reconfigurable platform, the compiler must discover opportunities to create dataflow operator graphs and reuse them across many invocations.

5.1.4 Sequential Control

The most widely used models for capturing and reasoning about algorithms are based on some form of sequential operation, including popular programming languages (e.g., C, Java, Fortran), control structures for hardware (finite-state machines), and formal models of computation (Deterministic Finite Automata, Sequential Turing Machines). The basic idea behind these models is that computations are defined as a sequence of primitive operations performed on some data state. The primitive operations define how state is transformed, including the state that determines which primitive operation(s) to execute next. Simple, concrete embodiments of this include sequential Instruction Set Architecture (ISA) processor models [13], but the state transforms can be much larger, may be coarse grained, and may include substantial parallelism on each sequential step.

Sequential control allows us to decompose a problem into simple, primitive operations. One thing happens at a time, making it relatively easy to reason about what each operation can do to the state.

Execution where only one primitive operation occurs at a time does not take full advantage of spatial reconfigurable architectures, leaving almost all the hardware idle as operations are sequentialized. Coarse-grained sequential operations that perform complex functions on large amounts of data may provide sufficient parallelism to match the reconfigurable hardware. While strict sequentialization of operations defines the intended results in the model, careful analysis can often reconstruct a data dependence graph (Chapter 7), essentially the dataflow graph (see Section 5.1.3), to allow several operations to proceed in parallel and at the same time maintaining the sequential model semantics. Still, care must be taken in the sequential expression to avoid introducing false dependencies that inhibit parallelism. In general, the sequential expression can be a poor match for the parallel capabilities, and sequential models tend to lead the designer away from good reconfigurable implementations. There are, however, characteristics of our computations that sequential control may capture well at a high level.

- Data-dependent calculations are naturally captured with branching. Sequential control here allows us to express the selection of the computation we need to perform on the data.
- Phased computations where the algorithm does widely different things at different times may also be captured well with sequential control. If each phase requires widely different computation, spatially supporting them all at once may leave much of the reconfigurable hardware idle during the calculation. Transitions between phases gives us a way of expressing and identifying points in the program where it may be useful to reconfigure the hardware for the different portions of the task, instantiating only the relevant hardware for each phase.

Finite state
The simplest models of sequential control operate with a finite amount of state and are computationally equivalent to finite automata. Given this, verification of optimized computations can be performed in polynomial time with state reachability [14].

Sequential control with allocation
In more powerful models of sequential computation, we allow operations that allocate additional memory (e.g.,`malloc`,`new`). Coupled with data-dependent branching, this allows the computation to allocate an unbounded amount of state, making the model Turing Complete, which in turn means that we cannot generally prove a bound on the amount of memory the application may require to run to completion.

Single memory pool
As noted earlier (see Section 5.1.2), because of an object's internal state we must carefully sequence the operations on it. We can think of each logical memory pool in a sequential model as an object with state so every operation on a single memory can be dependent on every other. If static analysis cannot prove that two users of the memory operator modify disjoint state in the memory, the operations must be sequentialized to preserve sequential correctness. In single-memory compute models, such as the C programming language or a traditional ISA execution environment, all memory operations must be sequentialized. This

sequentialization significantly limits the parallelism a compiler can extract from a single-thread, single-memory compute model. Consequently, large C programs that have not been carefully written to avoid these dependencies can be difficult or impossible to parallelize. Nonetheless, aggressive compilers can sometimes succeed in decomposing the monolithic memory into disjoint memory pools (e.g., Babb et al. [15]).

5.1.5 Data Parallel

Some applications are naturally captured as performing identical transformations on a set of independent data items. For example, we may need to perform the same color–space conversion to every pixel in an image, or perform the same match test to every data item in a database. Even though we could express such a task as a sequential loop over all the data items, it is often difficult for a compiler to prove the independence of each data item transform, and it can be tricky for the developer to identify which loop operations allow independent computation. Therefore, it is often useful to have an explicitly data parallel model that allows us to reason about and express algorithms as a sequence of transformations on aggregate datasets.

Once the desired computation is captured as a sequence of independent, identical, potentially parallelizable operations, we have considerable freedom in implementation for area–time tradeoffs. The computation can be rendered spatially and kept active as a heavily pipelined vector unit (see Vector coprocessors subsection of Section 5.2.4). Additional, parallel units can be allocated as the dataset demands and the platform permits.

The model typically remains sequential at the core and can suffer from artificial parallelism limits based on the provided sequential model. In particular, it may be hard to determine cases where multiple, independent data parallel operations can occur simultaneously. Although the parallelism on a single operation is limited by the size of the aggregate data item, the data parallel model does give general high-level guidance to the developer that often trends in the right direction for efficient spatial realizations.

5.1.6 Data-centric

In the streaming dataflow model, the designer thinks of the application as a transformation graph with data generally flowing through operators with fixed state. For some applications, such as physical simulations, it makes sense to turn that around and think about the operators and their state as the primary data structure, and reason about the computation as transformations on the operator state. For a network flow problem, we might construct the graph for the network; each operator maintains state to represent the flow through its links and the accumulating overflow at the node, and each operator sends tokens over the edges between operators to reroute flow. At each sequential step we may allow each operator to process a set of inputs and send a set of outputs. At a high level the operation is data parallel, with each operator performing its node update operation; however, locally the computation may be data and state dependent. High-level data parallel instructions to the operators can sequence phases of the computation (e.g., preflow and push phases in network flow).

Applications that regularly visit many nodes on large graphs of data are a natural source of parallelism. Even if the nodes are not identical, there are usually only a small number of different node types, providing an opportunity for sharing of spatial operators. Without strict dataflow communication ordering, additional disciplines may be necessary to maintain determinacy. Efficient execution may require load balancing and sharing if graph nodes have low or widely varying activity factors.

5.1.7 Multi-threaded

A widely used model for parallelism is multi-threading or some form of CSP [2]. Basically the model is a collection of sequential control processes with communication links between them, either as direct communication edges or as shared memory. Multi-threading is a very general model and, in fact, any of the models presented so far could be seen as subsets of it.

The problem with multi-threading is that it is too general and powerful to provide guidance for application development and correct implementation. It permits the expression of solutions that are difficult to reason about, and it provides little guidance on good solutions and guaranteeing determinism [4]. How should the application be divided into threads? How do the threads synchronize with each other? How do we guarantee determinism and avoid deadlock? In our streaming dataflow model, we think of each thread, the operators, as transforms on the data flowing through them, and we synchronize based on token flow; in our data parallel model, we think of each thread as a separate data item and update each in lockstep; in our data-centric model, we think of each thread as an active object in the graph, performing updates on barrier-synchronized steps.

When faced with applications that demand more power than is available in a more restricted model, we should think about the power actually necessary for our application and the extent to which we can define a restricted discipline for using the multi-threaded model that answers the questions the model does not answer for us. What do our threads and operators represent? What is the synchronization discipline? What is our basis for reasoning about determinacy, deadlock, and correctness?

5.1.8 Other Compute Models

The compute models reviewed here are by no means exhaustive. From the start, we want to emphasize the need to consider multiple models and choose the one most natural for the application. The set just described are useful in reasoning about the architectures and applications developed in this book and may be most helpful for reasoning about reconfigurable applications. Nonetheless, as we master these models and encounter applications that match poorly with them, we should look for others that further ease the conceptualization of an application. (For other summaries of compute models see Lee and Sangiovanni-Vincentelli [16] and Lee and Neuendorffer [17].)

5.2 SYSTEM ARCHITECTURES

Whereas the compute model helped us understand the natural composition and parallelism in the application, the system architecture deals primarily with how we organize the implementation. As noted (introduction to Section 5.1), applications in a compute model may be mapped to any of several system architectures. The choice of architecture will depend on technology costs and resource availability compared to the application resource and performance requirements. For example, a platform that is very small compared to the size of the task drives serialization in the implementation, which may favor sequential control. Even here, though, we have important decisions to make about the level at which the sequential control is exercised (e.g., coarse-grained phasing) (see Phased reconfiguration manager subsection of Section 5.2.2) versus cycle-by-cycle sequencing (see FSMD, VLIW datapath control, and Processor subsections of Section 5.2.2).

Figure 5.7 is an overview of the system architectures, and their variants, covered in this section. To help the designer easily identify those that may be relevant to his or her specific problem, we open the description of each one by identifying the major problem or challenge it addresses.

5.2.1 Streaming Dataflow

We best exploit a reconfigurable platform when we can spatially arrange specialized computational pipelines and keep them each actively working on useful computation at a high cycle rate. How do we organize computations that can exploit this efficient use and arrange for data to feed the pipelines?

In the simplest case, we can use one of the streaming dataflow compute models (Section 5.1.3) directly as a guide for system implementation; that is,

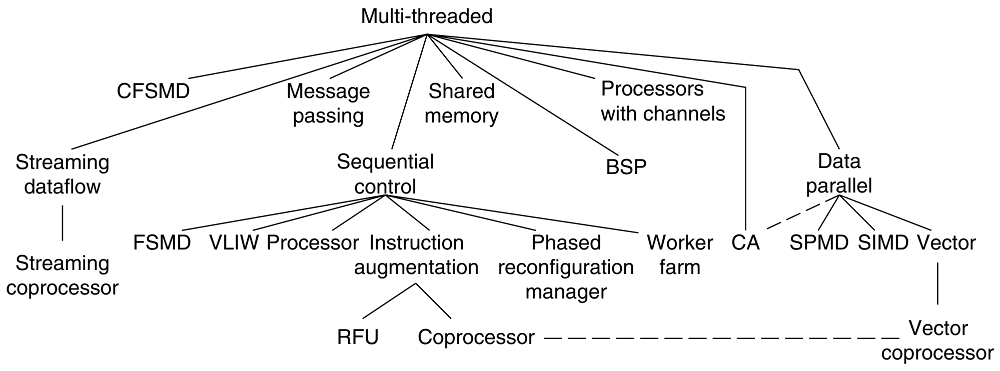

FIGURE 5.7 ■ Overview of system architectures.

we can map each operator to its own physical datapath and interconnect them all via configured interconnect. The efficiency of spatial pipelines on FPGAs and reconfigurable architectures makes this attractive. Further, the streaming model shows where in the detailed, cycle-by-cycle behavior of operations we have the implementation freedom to adapt to target platform delays. This architecture is known as *Pipe and Filter* in the literature [1]. Chapters 8 and 9 describe applications and programming that use it.

In the remainder of this section, we highlight four detailed techniques that are often useful in implementing streaming dataflow architectures.

Data presence

Direct connections of pipelined datapaths may pose challenges to guaranteeing the proper streaming dataflow semantics, offering efficient implementations, or allowing composition. These challenges include:

- Configured interconnect paths between operator datapaths may be long and can vary on the basis of platform, implementation technology, and operator placement. Long interconnect paths may limit the speed of operation.
- Different implementations of an operator may operate at different rates, and we want to be able to interchange these implementations without redesigning the implementations for all of the operators that interact with this operator.
- In dynamic dataflow models, an operator may not be able to consume an input, or produce an output, on every cycle of operation.

To promote easy and efficient operator composition, we can associate a "data present" signal with each data item. We design the physical functional units so that they can stall while waiting for the required inputs to be present. This decouples the clock cycle for interconnect and compute from the logical alignment of data, allowing us to pipeline the datapaths and the interconnect paths between them without changing the meaning of our computation. In many cases, we need to treat the interconnect paths as FIFO queues between operators; further, we can use back-pressure to indicate when a stream link between operators is full and so the upstream operator must wait before producing additional results.

The discipline makes the implementation of an operator independent of the implementation of others with which it communicates, allowing each to run at its desired clock rate even as all of them are composed together to build a larger system. This permits a variety of composite implementations:

- Operators and interconnect can all be designed to a single target clock frequency.
- Operators may run on separate clocks that are based on a common base frequency.
- Operators and interconnect may run fully asynchronously, handshaking locally.

- Operators may use a Globally Asynchronous, Locally Synchronous (GALS) model, with local operator clocks and asynchronous handshaking between operators.

It is still necessary to pay attention to the length of logical cycles in the original streaming dataflow graph; a loop in the graph may force sequential evaluation of all the graph's operators. Even though we can physically pipeline the operators and the links, the logical alignment of data may force the operators to effectively operate at lower rates, leaving the datapaths and interconnect inactive on most cycles. Such dependencies may motivate sequential sharing of operators or the resources inside them.

Datapath sharing

Ultimately, we must fit our entire dataflow graph onto our physical platform. For efficiency, we hope all of the hardware allocated to the dataflow graph is put to productive use on each cycle. Following are specific scenarios we may need to address:

- The substrate may not be large enough to hold the entire dataflow graph spatially.
- Multirate dataflow graphs may leave some operators idle while others are busy.
- Cyclic dependencies in the dataflow graph may make it impossible to keep all the operators active simultaneously.

To use the datapath hardware efficiently in cases such as these, it is often useful to share a physical datapath among multiple operators. In the simplest case, we share identical operators so that the datapath remains the same, only adding the unique state associated with each of the operators. In more complicated cases, we might generalize the datapath so that it can implement two or more types of operators.

When we share operators, we need to identify which data inputs are associated with which logical operator. This can be simply orchestrated by scheduling and pipelining for static-rate operators, but for dynamic operators and variable implementation delays, it may be necessary to further tag the data with information that identifies the logical operator for which it is destined.

Streaming coprocessors

With extreme variation in operator frequencies, large numbers of operators, and very small platforms, operator sharing may not be sufficient to provide an efficient solution. Here, even allocating a single datapath for a particular hardware type may leave the datapath highly underutilized or it may still demand more area than the platform provides.

In these more extreme cases, it is often useful to schedule the low-rate operators onto an embedded or attached processor (see Processor subsection of Section 5.2.2). By augmenting the processor with streaming instructions, processor-mapped operators can communicate efficiently with streaming dataflow. Data destined for active operators can be forwarded spatially, while data intended for inactive operators can be queued in memory. Data presence allows the processor tasks to operate without knowing the size of the reconfigurable platform or the residency of operators. Data presence on stream reads by the processor can be used like a memory stall, tolerating varying implementation delay on the reconfigurable platform or triggering an operator swap, similar to a thread swap on an I/O or virtual memory page miss.

Interconnect sharing
In spatial computations, interconnect often consumes a substantial portion of the hardware area and can often be a performance bottleneck. Consequently, we should always be concerned about using the interconnect efficiently. A direct, configured connection between a source and a sink can be inefficient when

- The link between operators is used infrequently because of a slow datapath or a low-rate operator relative to the rest of the computation.
- Because of dynamic data dependence, the communication rate on many links is highly variable.

To optimize interconnect in these cases it may be possible to reduce the interconnect requirements on these interconnect links by sharing them. Links can be shared in a variety of ways, including shared bus, pipelined ring, and network-on-a-chip. These can be statically scheduled in data-independent cases and in data-dependent cases with low communication variability, or dynamically managed when the data-dependence produces high variability.

5.2.2 Sequential Control

While sequential control is familiar and heavily used for highly sequential machines and algorithms, it is most interesting to us as a way to organize synchronization and control of a large set of spatially parallel operators, particularly when

- The compute task is too large to fit spatially onto the available computing resources, so we must share the resources in time.
- Data dependencies result in low utilization of the datapath, so we can share resources to produce a smaller design with little or no impact on compute time.

Even when we start with a dataflow or data-centric computation, it may be useful to control the implementation, or parts of it, in sequential manner; this is especially true when we share spatial operators in time to economize on space.

A common idiom is to

1. Start with the computation data dependence graph (e.g., Figure 5.8 (a) or Figure 5.2) based on the description in the compute model.
2. Identify a base set of datapath elements that can implement all the operators in the computation graph.

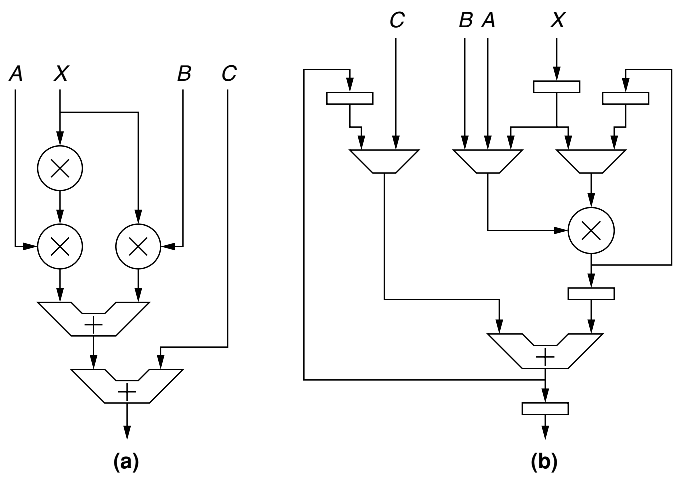

FIGURE 5.8 ■ A dataflow graph for $y = Ax^2 + Bx + C$ with three multiplies and two adds (a); a shared datapath (b) with a single multiplier and adder with state registers and multiplexors.

3. Schedule the operators in the compute graph onto the datapath elements.
4. Add data storage and interconnect to hold intermediate operator state and forward data between the locations where producing and consuming operations are performed.

In the simplest case, we might allocate a single datapath element for every operator in the compute graph. While there is no sharing in this case, it may still be necessary to control when the elements should sample their inputs and produce outputs. This can be done in a purely dataflow manner as suggested in the Data presence subsection of Section 5.2.1; however, for modest blocks in a single clocking domain with predictable datapath timing, it can be more efficient to centrally control the operators, sending control signals to each datapath element from a central control unit.

In the more general case, we have fewer datapath elements than operators and must orchestrate the sharing of those elements and interconnect. Intermediate values in the original computational graph that are not consumed in the cycle immediately following production, or immediately after being routed from the source to the destination, are stored temporarily in memories (see Figure 5.8). Object state that persists through the computation must be stored in memory or registers and routed to the associated datapath when the operator has its turn to use the datapath.

Within this paradigm, the key piece of freedom is the selection of the base datapath elements and the assignment of operators to them. This selection is where we can exploit area–time tradeoffs, allocating more spatial datapath elements as we have more area available and want to reduce the time for computation; it is also where we have opportunities to instantiate highly specialized operators that are matched to the needs of a particular task (e.g., Chapter 22).

The design community has identified a number of stylized forms for sequential control over the years. In the remainder of this section, we highlight a number of organizations and note when they may be useful for managing reconfigurable resources.

FSMD

Once we have selected the operators, assigned them to datapath elements, and scheduled the operations, we still need some way to implement the central control that manages resource sharing and orchestrates the routing of intermediate data among datapath elements.

One common way to support this control is to build a finite-state machine (FSM) that controls the operation of the datapath; this is called a Finite-State Machine with Datapath (FSMD) [18]. The FSM controller can assert the various controls (e.g., multiplexer selections, load or read/write enables, datapath operation selection) on each cycle and provide cycle-by-cycle sequencing of them (see Figure 5.9). Further, the FSM can take inputs from the datapath and, based on their data, branch to different control sequences.

A data-dependent operator might be internally implemented as an FSMD, with the state transitions in the FSM controlling the input consumption and output production (see Dynamic streaming dataflow subsection of Section 5.1.3, or Section 5.1.6).

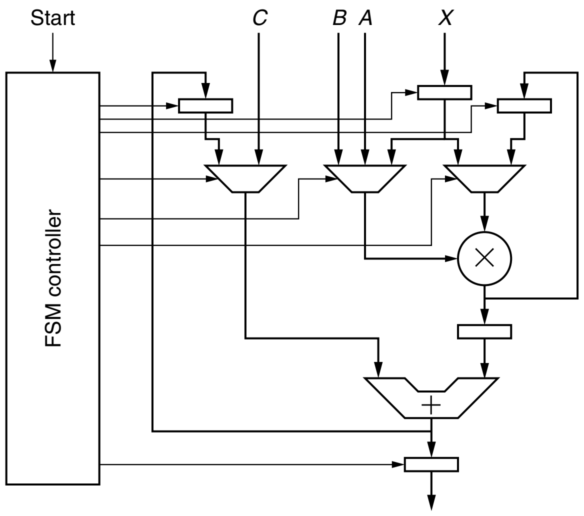

FIGURE 5.9 ■ FSMD for a single multiply and add datapath for quadratic equation evaluation.

VLIW datapath control

While we can build a custom FSMD for each application, the FSMD form does not, itself, provide disciplined organizations for state storage and data routing, nor does it suggest any organizing principles for managing the control of each datapath. As a result:

- With heavy sharing there is a proliferation of intermediate state that needs to be managed.
- With many datapath operators, state memories, and switched interconnect, there is a proliferation of control signals that must be distributed to these compute, memory, and interconnect elements.
- For generality, robustness to change, and the opportunity to deploy the datapath for multiple tasks, it may be useful to be able to change the control sequencing without rebuilding the entire controller.

One stylization for sequential control is the Very Long Instruction Word (VLIW) model, which in its most primitive form is closely related to Horizontal Microcode [19]. In VLIW we start with the collection of datapath elements as before. These can be homogeneous or heterogeneous and provisioned according to the needs of the task. We then add one or more memory banks to hold inputs to each datapath element, and we add switched interconnect between the datapath elements and the memories. The controls to the memories, datapath elements, and interconnect become the long instruction word, to which we allocate a wide memory, perhaps distributing it with the memory cells and memory outputs local to the compute, interconnect, and memory elements they control (see Figure 5.10). To issue an "instruction" (see Chapter 36), the controller sends a single instruction address to the wide memory, and the memory output tells every datapath element, memory, and interconnect switch how it

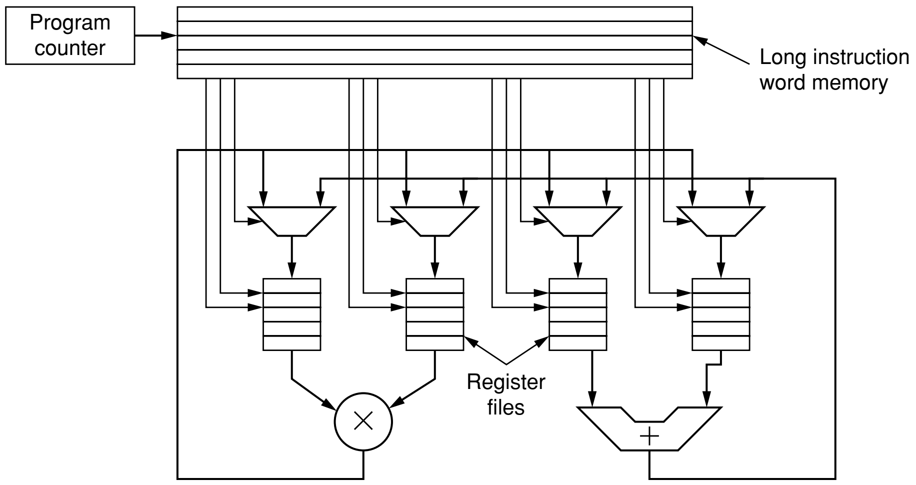

FIGURE 5.10 ■ VLIW-style control of a single multiply and add datapath.

should be configured on that cycle of operation. Typically, the datapath is configured to send one or a few bits back to the controller that can be used to select the next instruction address to allow data-dependent branching.

When VLIW was first introduced for general-purpose processing (e.g., Ellis [20]), the datapath elements used were generic (e.g., ALUs, FPUs, load/store units), of modest size, and fairly homogeneous. With FPGAs and reconfigurable architectures, we have the opportunity to select the datapath elements based on the task, make them highly specialized, and potentially even make them fairly coarse grained (e.g., a DCT step, motion estimation step, or AES encryption step).

Processor

The FSMD and, to some extent, VLIW control both assume that there are common datapaths that can be shared, and both allow multiple, concurrent operations to exploit the spatial parallelism available on an FPGA or reconfigurable device. However, for some computations our premium may be space saving rather than operation performance. That is, overall system performance may depend on this operator fitting onto the platform and being performed infrequently, but the time the operator takes may have little impact on it.

A conventional, sequential processor or microcontroller with a single arithmetic logic unit (ALU) is the extreme end of sharing, where we

1. Allocate a single, universal datapath element.
2. Decompose all operators into sequences of operations on this primitive datapath element.
3. Provide state storage for all intermediates between the cycle of production and the cycle of consumption, including storage for all object state.
4. Define a narrow instruction to control the datapath element and state storage.
5. Provide a sequencer and branch unit to sequence the instructions on the datapath in a potentially data-dependent manner.

Because this allocates minimal area to computation and interconnect, the total area for the computation can be very compact; however, compactness comes at the expense of most of the resources going to control, instruction, and state management. As a result, only a tiny fraction of the consumed computational resources go directly to implementing the application (see Chapter 36).

If heavy serialization to economize area is what we need for an entire task, a dedicated processor is certainly more efficient than a processor configured on top of an FPGA. Nonetheless, there are a few scenarios where a processor configured on top of an FPGA might be reasonable. Such scenarios would typically exploit the flexibility of either building a particularly specialized and lightweight processor for a specific task and/or embedding one in a flexible and highly integrated manner alongside a much larger computation implemented using a more spatial implementation architecture.

When we have multirate computations (e.g., Synchronous Dataflow model subsection of Section 5.1.3), some operators may execute at much lower rates than others. To balance the system and achieve maximum application performance in a limited area, we typically allocate space to operators in proportion to the fraction of the total computation they perform. As a result, we may end up with some very infrequent operators that are needed to complete the task but can afford to operate very slowly. If there is a dedicated, attached processor, perhaps these operators can be run there; if not, or if the flexibility to place the processor datapath for this operator local to other computations is important, it may be worthwhile to implement the operator as a configured processor.

Instruction augmentation

For resource sharing, a sequential controller is often necessary to direct the use of specialized datapaths. Sometimes this takes the form of a mix of irregular, low-throughput tasks that do not need to be executed quickly along with some very regular computations that are critical to performance. Manifestations of this need include:

- We need to sequence a modest amount of FPGA or reconfigurable logic.
- The computation contains a few operations that account for most of the time, embedded in a large amount of irregular tasks necessary to define the complete computation.

A processor is an efficient, programmable, and well-understood sequential controller. Consequently, it is often useful as the base design for a sequential controller. This is common enough that many platforms provide a dedicated processor attached to an FPGA or reconfigurable array (Chapter 2). It is also useful enough that this may be one of the motivations to employ a custom, configured processor.

One way to provide the coupling between the processor and the FPGA array is to treat the functions provided by the FPGA as additional instructions that augment the processor's base instructions. The processor's execution model of issuing instructions and expecting them to be performed in sequence remains intact, but the set of instructions it can issue are enlarged by the configured array. The FPGA instructions can potentially be very powerful, performing the equivalent of hundreds of base processor instructions in a single invocation. This can be particularly effective when a few such powerful instructions can cover the bulk of the execution time in the task. The processor serves as the application glue, sequencing these dominant operations and orchestrating the movement of data to connect them.

Functional Unit model One way to implement instruction augmentation is to provide a reconfigurable functional unit (RFU) (e.g., Razdan and Smith [21], Hauck et al. [22], and the Tightly coupled RPF and processor subsection of Section 2.2.2); that is, we treat the reconfigurable array just like any other functional unit

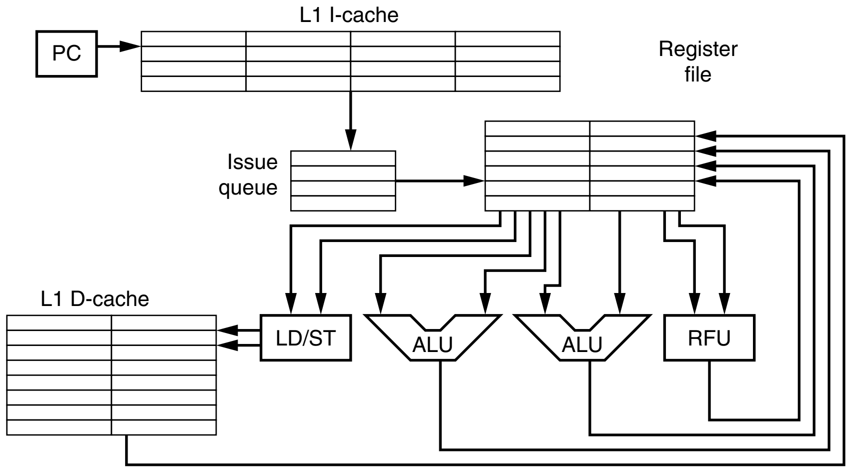

FIGURE 5.11 ■ A super-scalar processor with an RFU.

in the processor (see Figures 2.12 and 5.11). The processor issues instructions to it, feeding it data from the register file, and the array returns the result to a register. Normal processor issue and scoreboarding mechanisms can be used to accommodate variable delay in the array operation. The Functional Unit model may be particularly useful in specializing a configured processor to a particular application, where the custom functional units each perform a single function. It can also be used for coupling a custom processor to a reconfigurable array. One variant is to allocate a set of opcodes in the instruction for the reconfigurable function unit so that the processor instruction can call out different array operations.

The Functional Unit model is easily integrated into a conventional processor pipeline. However, it provides limited I/O between the processor and the array and demands that the reconfigurable operation be a function, preserving no internal state. This potentially limits the use of the array, by preventing the allocation of large, coarse-grained operations on it.

Coprocessor model Another way to implement instruction augmentation is to treat the reconfigurable array as a coprocessor (e.g., Callahan et al. [23]—see Figure 5.12), with the processor performing explicit data moves to and from it and directing it to perform specific operations. The coprocessor model allows the array to hold its state and places data close to it. This makes it possible to push larger portions of the computation onto the array, only communicating data back to the processor at large operation boundaries. The I/O to a single operation can be sequenced over several cycles, which allows greater flexibility in operator granularity.

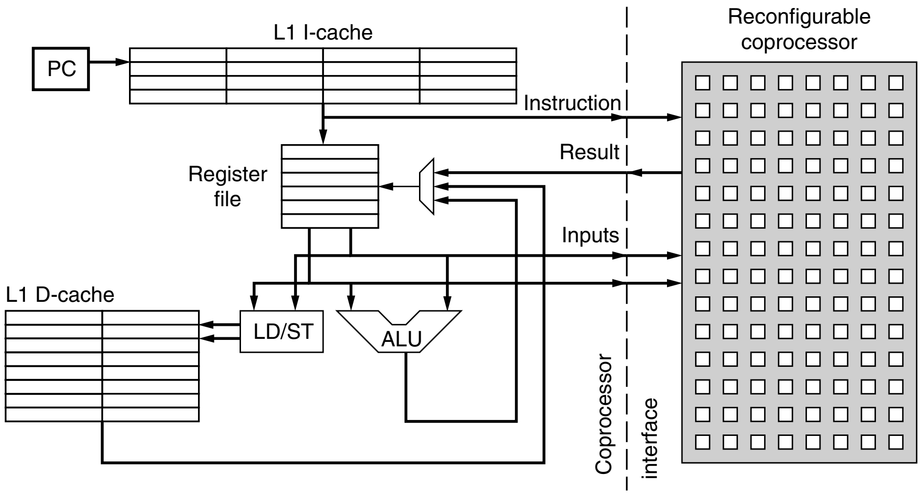

FIGURE 5.12 ■ A scalar processor with a reconfigurable coprocessor.

Phased reconfiguration manager

In the preceding sections, we shared the FPGA or the reconfigurable array resources in time in a fine-grained manner by scheduling operators on a cycle-by-cycle basis onto the datapath elements. This works when we have common operator types that permit sharing, or when we can generalize the datapath element to support many operators. In order to realize this we added additional circuitry to the design to flexibly route data between the datapath elements and to sequence the sharing. These additional resources did not contribute computation to the original task and so were pure overhead. However, since our hardware is reconfigurable, in some cases it is possible to reconfigure it and perform this sharing at a coarser granularity with less overhead. Since reconfiguration is often slow, this is viable only when we can arrange for the array to be used for a long period of time in a single configuration, such as when tasks operate in phases, performing distinct computations for long times. For this to be useful the "long time" in a configuration should be long compared to the time required to perform the reconfiguration (see Section 4.2 and Chapter 9).

In these cases, sequentialization is very coarse grained. We can nonetheless still think of the sequencing as a sequential control application, with each state potentially representing a different configuration of the array. The sequential controller monitors the execution to detect the end of the phase, implements configuration, and may even perform state-dependent branching. Sequential control can be realized with many of the architectures previously discussed (e.g., FSM, processor, instruction augmentation).

Worker farm

Sometimes we may have a set of dependent operations where each one runs for a large and variable amount of time. For example, Unix/Linux `make` rules specify

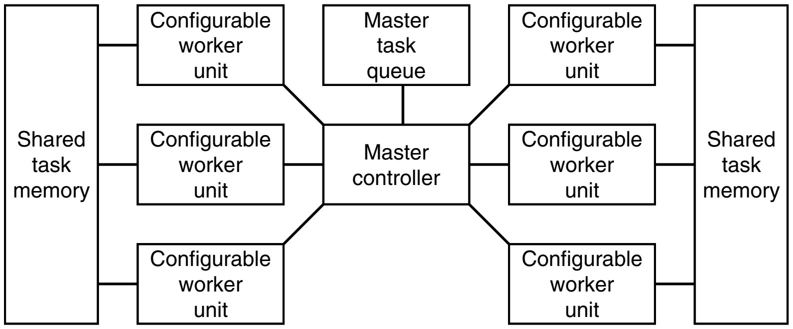

FIGURE 5.13 ■ A worker farm.

a coarse-grained, dependent task set, and clustered multiprocessors exploit this parallelism with parallel `make` utilities such as `pmake` [24]. The variable runtime means that predetermined assignments of operations to hardware resources can be very inefficient.

We can exploit the reconfigurable hardware in these cases by organizing resources as a set of *workers*, which actually process jobs, and a central *manager*, which is responsible for assigning operations to them, potentially coordinating data movement and reconfiguration (see Figure 5.13). Here, the manager might

1. Maintain a queue of ready tasks.
2. Issue the first ready task to execute on a free worker.
3. Continue issuing tasks to workers until there are no free workers.
4. Wait for one or more workers to signal completion.
5. As tasks complete, put any tasks they enable on the ready queue.
6. Loop back to step 2.

Operations are enabled in dataflow form as they are completed. If the tasks are largely homogeneous or taken from a small set of types, the workers may be identical or taken from a small set of datapath configurations. If they are long running and highly heterogeneous, it may make sense to reconfigure them to each task; when the reconfigurable array supports it, this might include partial reconfiguration (see Section 4.2.3) of the array to customize each worker for its next task.

5.2.3 Bulk Synchronous Parallelism

In our sequential control architectures, we had a central controller telling every datapath element what to do. This guaranteed that the datapath elements moved forward in a synchronized manner. However, if the work required by each datapath is highly data dependent, a centralized locus of control may become inefficient. Consequently, we often allow the local datapathsto have independent control but still want to guarantee that they remain synchronized at some coarser granularity. In particular, we might want to ensure that one set of tasks completes before another begins.

Bulk Synchronous Parallelism (BSP) [25] can be seen as a variant that keeps the synchronization centralized but distributes the datapath sequencing. In BSP, independent units of computation progress independently, with the local computations punctuated by periodic barrier synchronization events. Each local computation announces when it reaches the barrier and waits for a global acknowledgment that all local tasks have reached it before proceeding.

The barrier is an efficient technique for supporting data-dependent, time-variable operations in each task while still providing strong synchronization guarantees. An alternate would be to statically determine the length of each epoch and have local tasks that complete their epoch early wait until the static epoch duration completes. If the runtime of each task varies widely based on data or potential resource contention, the static bound necessary to guarantee correctness may be excessively long compared to the common case local task completion time.

Further, if the local tasks are Turing Complete, it may not be possible to even identify such a static upper bound on the timing between barriers. The expense of the barrier is that it requires $\Omega(\log(N))$ time to perform the synchronization in the ideal case, where wire delays are negligible, and $O(\sqrt{N})$ or $O(\sqrt[3]{N})$ time in realistic 2-space or 3-space physical implementations for the barrier to complete. This suggests efficient operation only when the computational work between barriers is at least as large as this barrier synchronization time.

A BSP architecture can be appropriate for implementing data-centric computations (Section 5.1.6). Often objects communicate over their connected graph links. For many applications it is useful to guarantee that each object processes one round of method invocations before starting the next round. Barriers between rounds allow the operator to know when it has received all the invocations associated with a single round and can safely advance [26].

5.2.4 Data Parallel

As the Data Parallel Compute model suggests, sometimes computation can be organized as a set of computations applied, mostly independently, to a large set of data (see Section 5.1.5). This gives us both parallelism and regularity that a reconfigurable implementation can exploit. We want to be able to use this parallelism in a scalable manner, allocating more or less hardware as the platform permits.

A number of stylized architectures support data parallel computations and can be tuned for varying amounts of parallelism. The remainder of this section highlights three architectures and one technique for interfacing and controlling data parallel computation with more general computation.

Single program, multiple data

Although it is sometimes useful to apply the same basic operations to each component piece of data, these operations can be highly data dependent and can benefit from independent, local control. However, even though they are locally independent, it may be useful to guarantee that a set of operations on the data completes before continuing with the next operation set.

SPMD (single program, multiple data) is an organizational structure that follows the high-level Data Parallel model with minimum stylization within each data parallel task. Essentially, we have a collection of independent threads or control units that happen to be performing the same operation on different datasets. Individual independent threads can, themselves, be implemented as one of the system architectures described here. They are typically synchronized periodically in BSP fashion (Section 5.2.3).

Single-instruction multiple data

Control and instructions for a datapath can become expensive. Thus, if the data dependence for data parallel operations can be kept low, it is beneficial to share instructions and control across a large set of datapaths.

SIMD (single-instruction multiple data) architectures control the hardware operations on a cycle-by-cycle basis similarly to our sequential control architectures (Section 5.2.2). However, instead of a heterogeneous set of datapath elements, each potentially receiving unique operations, a single, common instruction is delivered to all of them. Each element has its own data and performs the sequence of instructions on it. Communication between datapath elements is also supported with common instructions to orchestrate data movement.

SIMD architectures can be more compact per processing element than VLIW architectures, because they do not need to store separate instructions for each compute, memory, or interconnect block. However, since SIMD architectures force all datapath elements to perform the same operation simultaneously, the SIMD datapath elements are efficiently utilized only on much more stylized and limited computations (see Chapter 36).

Chapter 10 describes a particular SIMD system in more detail, including an approach to SIMD compilation for FPGAs.

Vector

The motivation for vector architectures is similar to that for SIMD: When operations are sufficiently regular and data independent, they admit implementations that economize on resources by sharing instructions and associated control. Vector architectures particularly exploit the fact that datapath operations often have long latencies and can be pipelined so that calculations on many, independent data items can reuse the datapath at high throughput.

In a vector organization, a sequential controller issues data parallel instructions across a logical dataset. Here, we think of supplying vectors of component data, rather than individual words, as our inputs and outputs of instructions. The instructions perform operations similarly to a sequential processor on the pairwise components of vector inputs. Rather than the data living with the datapath elements, as is typical in SIMD, the vector data is normally kept in central memory banks and vector register files and is routed to the datapath elements. The vector instructions then specify where to find vector inputs and where to return vector results. The data parallel operation on these vectors can be performed in sequence on a highly pipelined vector functional unit, in parallel on a set of parallel functional units, or as a sequentialized set of parallel batches based on the area allocated.

On reconfigurable platforms, we can construct highly specialized vector functional units for each task. Thus, a vector control unit can be augmented with specialized vector pipelines just as a processor can be augmented with configurable instructions in an Instruction Augmentation architecture (see Instruction augmentation subsection of Section 5.2.2). Here we are operating on vectors of data rather than on individual scalar data elements. As with other models, we can identify the coarse-grained, data parallel operations required in the task and allocate a suitable set of functional units for them. The vector control unit then issues instructions to perform the data routing and sequencing to connect the operations running on the vector functional units (see Figure 5.14).

Vector coprocessors

As noted earlier, we often have a mix of irregular computations and more regular stylized computations (see Instruction augmentation subsection of Section 5.2.2). This is certainly true when exploiting highly stylized, data parallel computations using vector or SIMD architectures.

The Coprocessor model (see Coprocessor model subsection of Section 5.2.2) provides one stylized way to add configurable vector units to a base processor architecture (e.g., Wawrzynek et al. [27] and Jacob and Chow [28]). Here, the vector operations become coprocessor instructions. The processor can remain scalar, with normal instructions and register files, with the configurable vector unit maintaining all the vector states local to the configurable array. The vector

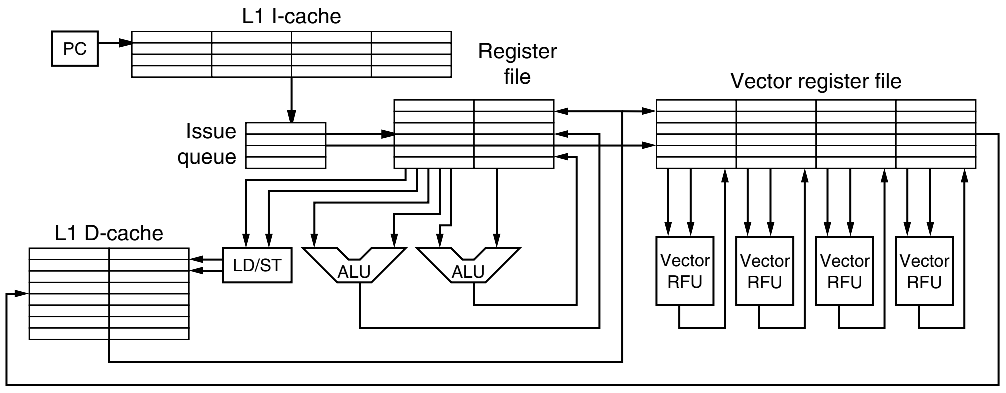

FIGURE 5.14 ■ A super-scalar processor with vector functional units.

coprocessor can keep multiple vector operations in flight, using scoreboarding on memory or vector registers to enforce sequential semantics on the sequentially issued vector operations.

5.2.5 Cellular Automata

Although spatial computation organizations offer great parallelism, they also demand that the spatially distributed datapaths communicate with each other. For large computations, the physical latency between distant operations can be large; further, the worst-case, cross-chip latencies actually grow relative to cycle rates as technology scales. Considerable, nonlocal traffic can slow the computation both because of round-trip latencies and because of limited available cross-chip bandwidth.

Cellular automata (CA) suggest a pattern for organizing computations as a line (one dimension), mesh (two dimensions), or cube (three dimensions) of regular operators with nearest-neighbor communication (see Figure 5.15). The operators run logically in lockstep, sampling the state of adjacent operators and updating their own. The regularity of identical operators makes it easy to scale to larger spatial designs. Moreover, nearest-neighbor communication eases layout and guarantees that communication does not limit overall design performance. A CA can be seen as a very stylized data-centric (see Section 5.1.6) computation in which the parallel operators have a restricted, regular communication pattern.

The restriction for nearest-neighbor communication may seem extreme, but it naturally shows up in many physical world simulations. Because physical interactions are also primarily nearest neighbor, the topology of the physical problem often maps directly to that of a regular CA. Examples of physical simulations include discrete-time solutions to wave, diffusion, Navier–Stokes, or Maxwell's equations (see Chapter 32). Perhaps the simplest and most well-known CA is Conway's game of "Life" [29]. It is even possible to implement CAD optimizations,

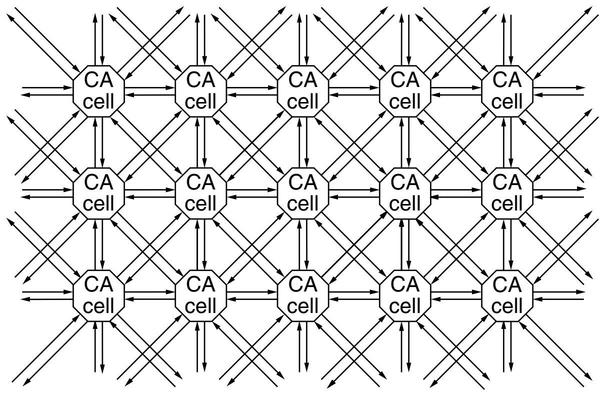

FIGURE 5.15 ■ Two-dimensional cellular automata.

such as placement, using CAs (e.g., Wrighton and DeHon [30]; also mentioned briefly in Chapters 9 and 20).

Folded CA

CAs can be highly efficient, but the size of the fully spatial design depends on the size of the problem. Thus, for large problem sizes the fully spatial CA can be too large for an affordable reconfigurable platform.

Since the CA is based on an array of identical operators and regular communication, it can be efficiently folded onto smaller physical platforms (e.g., Margolus [31] and Kobori et al. [32]). At one virtualization extreme we can build a single, physical CA cell processor and stream through the state of the virtual cells in series, using the single physical cell to implement all cells. The access pattern for the data is regular and predetermined, allowing efficient use of memory bandwidth. Plus, all the data communications are local, which means that we can readily program data buffering so that all data is available at the cell as needed on a single pass through the memory. We can also implement a single row (or column) of the CA and scan through one row (or column) at a time. In fact, we can choose just about any number of physical cells to implement—up to half the number in the logical array—and achieve a linear speedup of the computation. Chapter 32 describes a particular, folded mapping for a finite-difference, time-domain solution of Maxwell's equations.

5.2.6 Multi-threaded

The architectures discussed previously all place restrictions and stylizations on the computation to allow efficient implementation. Nonetheless, particular restrictions may not match with the needs of some applications and, consequently, may not provide the most efficient implementation support.

As suggested with compute models, multi-threading can be seen as the most general organization (Section 5.1.7). It provides great expressiveness, but at the cost of little guidance to the designer in how to exploit that expressiveness and guarantee correctness of implementation. This expressiveness can also make it expensive to support the full generality of the model on reconfigurable hardware.

In the remainder of this section, we review some common multi-threaded organizations and the benefits and caveats they entail.

Communicating FSMs with datapaths

Earlier we noted that an FSMD is a stylized way to control the operation of a datapath (see FSMD subsection of Section 5.2.2). For very large designs, a central controller may become a performance bottleneck for the following reasons:

- Central control may lead to unnecessary state explosion in a central controller.
- Sending control signals across a large system to a central controller and distributing control back from it may result in long latencies and slow operating rates.

One alternative to a single controller is decomposing the system into a number of independent FSMDs that communicate with each other. In this way, in addition to its own datapath controls, the FSM controller for each FSMD now contains inputs and outputs to one or more of the other FSMDs through which it coordinates synchronization. Thus, each FSM controller can be simpler and faster than the single, monolithic controller, and each can branch independently. However, the designer must be careful to manage the coordination of the FSMs so that they do not deadlock or otherwise transition into inconsistent states.

Technically, a composition of finite automata is still a finite automata, and it is possible to compute the composite automata in order to prove properties of the composite system. The state space of the composed automata can be as large as the product set of the state space of the individual automata. In some cases this state explosion can become intractably large for practical verification.

Processors with channels

In the Processor subsection of Section 5.2.2, we saw the motivation for an operator or several operators to run on a processor or, more likely, a processor controller with a specialized datapath. For similar reasons that motivate the communicating finite-state machines with datapaths (CFSMD) described in the previous section, it may not make sense to centrally control this collection of processors.

Here, too, we decompose the computation into a collection of augmented processor datapaths that coordinate with each other through direct links. Special instructions allow the processor to poll information from input channels and place information on output channels.

Message passing

When we connect processors or FSMs with communication channels, we often find it inefficient to commit dedicated, point-to-point links.

- The data rate on point-to-point channels between processors, operators, or FSMDs can often be too low to merit a dedicated channel.
- Dedicating point-to-point channels between processors, operators, or FSMDs can be too expensive for an implementation.
- Individual units of control may only need to communicate infrequently.

Rather than keep a channel open at all times, operators can share a common communication infrastructure (e.g., bus, pipelined ring, network-on-a-chip) and send their coordination information tagged with the identity or location of the recipient—in other words, send messages.

Shared memory

Multiple operators cooperating on a task may need infrequent access to a large set of shared state. When we exploit parallelism, these operators may be running on different parts of a physical platform yet need to access shared data pools.

When possible, it is best to give ownership of state to a single operator and let it provide coordinated access to it. This approach avoids a host of synchronization problems that can make parallel execution particularly troublesome. If the state is small and infrequently changed, and when several operators need regular access to it, it can make sense to allow each to have its own copy and allow coordination operations to change the data across them. However, when the state is large and infrequently accessed, such as with a large database, it is sometimes efficient to allow multiple physical processing elements to access a single, shared memory pool to avoid replicating the data and to allow data communication to be deferred until it is needed. This can allow each processing element to extract just the information it needs without burdening the others with knowing which information will be needed by which processing element.

In the general case, we may share memory pools between small sets of physical processing elements. Unlike with homogeneous multiprocessors, there is generally little reason to have a single, large, shared memory pool across all the processing elements in a reconfigurable computer. The configurability of our reconfigurable designs allows us to limit sharing based on the shape of communications in the application.

5.2.7 Hierarchical Composition

In this chapter we described most system architectures as homogeneous entities. However, in general we can consider them each as levels in a hierarchy. For example, it may make sense to use FSMD (see FSMD subsection of Section 5.2.2) or vector coprocessor (see Vector coprocessors subsection of Section 5.2.4) nodes to implement the dataflow operators in a streaming dataflow system architecture (Section 5.2.1). Further, to model and coordinate changes in the composition of the dataflow network over time, it may make sense to model each of the dataflow configurations as a state in a very coarse-grained FSM (see Phased reconfiguration manager subsection of Section 5.2.2). With a variety of system architectures, rich implementation options within each, and their hierarchical compositions, we have a broad and powerful set of techniques to exploit the flexibility in reconfigurable computing platforms.

References

[1] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
[2] C. A. R. Hoare. *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, 1985.
[3] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 1967.
[4] E. A. Lee. The problem with threads. *IEEE Computer* 36(5), May 2006.
[5] S. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society* 53(1), 1943.
[6] A. M. Turing. On computable numbers, with an application to the entscheidungs problem. *Proceedings of the London Mathematical Society* 42(2), 1937.
[7] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics* 58, 1936.

[8] E. A. Lee, D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE* 75(9), September 1987.

[9] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee. *Software Synthesis from Dataflow Graphs* (Synchronous Dataflow chapter), Kluwer Academic, 1996.

[10] T. M. Parks. *Bounded Scheduling of Process Networks*, UCB/ERLl95-105, University of California at Berkeley, 1995.

[11] Arvind, R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers* 39(3), March 1990.

[12] D. E. Culler, S. C. Goldstein, K. E. Schauser, T. von Eicken. TAM—a compiler-controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, June 1993.

[13] J. Hennessy, D. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2002.

[14] S. Devadas, Hi-K. T. Ma, R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 7(6), June 1988.

[15] J. Babb, M. Rinard, C. A. Moriz, W. Lee, M. Frank, R. Barua, S. Amarasinghe. Parallelizing applications into silicon. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[16] E. Lee, A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17(12), December 1998.

[17] E. Lee, S. Neuendorffer. Concurrent models of computation for embedded software. *IEEE Proceedings—Computers and Digital Techniques* 152(2), March 2005.

[18] D. Gajski, L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers* 11(4), 1994.

[19] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers* 30(7), 1981.

[20] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, 1986.

[21] R. Razdan, M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994.

[22] S. Hauck, T. Fry, M. Hosler, J. Kao. The chimaera reconfigurable functional unit. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.

[23] T. Callahan, J. Hauser, J. Wawrzynek. The Garp architecture and C compiler. *IEEE Computer* 33(4), April 2000.

[24] E. H. Baalbergen. Design and implementation of parallel make. *Computing Systems* 1(2), 1988.

[25] L. G. Valliant. A bridging model for parallel computation. *Communications of the ACM* 33(8), August 1990.

[26] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. Knight, Jr., A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.

[27] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, March 1996.

[28] J. A. Jacob, P. Chow. Memory interfacing and instruction specification for reconfigurable processors. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 1999.

[29] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American* 223, October 1970.

[30] M. Wrighton, A. DeHon. Hardware-assisted simulated annealing with application for fast FPGA placement. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, February 2003.

[31] N. Margolus. An FPGA architecture for DRAM-based systolic computations. *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[32] T. Kobori, T. Maruyama, T. Hoshino. A cellular automata system with FPGA. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.