CHAPTER 22

# INSTANCE-SPECIFIC DESIGN

Oliver Pell, Wayne Luk
*Department of Computing*
*Imperial College, London*

This chapter covers instance-specific design, an optimization technique involving effective exploitation of information specific to an instance of a generic design description. Here we introduce different types of instance-specific designs with examples. We then describe partial evaluation, a systematic method for producing instance-specific designs that can be automated. Our treatment covers the application of partial evaluation to hardware design in general, and to field-programmable gate arrays (FPGAs) in particular.

## 22.1 INSTANCE-SPECIFIC DESIGN

FPGAs are an effective way to implement designs in computationally intensive datapath-orientated applications such as cryptography, digital signal processing, and network processing. The main alternative implementation technologies in these application areas are general-purpose processors, digital signal processors, and application-specific integrated circuits (ASICs).

ASICs are integrated circuits designed to implement a single application directly in fixed hardware. Because they are specialized to a single application, they can be very efficient, with reduced resource usage and power consumption over processor-based software implementations. Reconfigurable logic offers similar advantages over general-purpose processors. However, the overhead of providing general-purpose logic and routing resources means that FPGA-based systems typically provide lower density and performance than ASICs. Still, reconfigurable logic can provide a level of specialization beyond what is possible for an ASIC: optimizing circuits not just for a particular problem but for a particular *instance* of it. For example, an encryption application can create custom FPGA mappings every time a new password is given, allowing any password to be supported yet providing very highly optimized circuitry.

The basic concept of instance-specific design is to optimize a circuit for a particular computation. This can allow a reduction in area and/or an increase in processing speed by sacrificing the flexibility of the circuit. It is important to distinguish between the FPGA itself, which is inherently flexible and can be reconfigured to suit any application by loading a new bitstream, and the current configuration of the chip, which may have a certain level of flexibility in processing its inputs.

One common way of achieving instance-specific designs automatically is constant folding (Section 22.2.3), which involves propagating static input values through a circuit to eliminate unnecessary logic. Thus, in our encryption example, an exclusive-or (XOR) gate with one input driven by a password bit can be replaced with a wire or an inverter because the value of that bit is known for each specific password.

To produce an instance-specific design, one first needs a means of providing a particular instance for a given design. In the previous encryption example, if all the passwords are known at design time, an instance-specific design specialized for each password can be produced, say by constant propagation followed by the usual tools such as placement (Chapter 14), routing (Chapter 17), and bitstream generation (Chapter 19).

At runtime, a processor is often used to control the configuration of the FPGA by the appropriate bitstream at the right moment to support a particular password. However, if the passwords are known only at runtime, then the designer has to decide whether the benefits of having instance-specific designs outweigh the time to produce them, since, for instance, current place and route tools often take a long time to complete and their use is usually not recommended at runtime. Fortunately for some applications, differences between instances are so small that they can be generated realistically using runtime partial evaluation (Section 22.2).

The ability to implement specialized designs, while at the same time providing flexibility by allowing different specialized designs to be loaded onto a device, can make reconfigurable logic more effective at implementing some applications than what is possible with ASICs. For other applications, performance improvements from optimizing designs to a particular problem instance can help shift the price/performance ratio away from ASICs and toward FPGAs. Specializing a Data Encryption Standard (DES) crypto-processor, for example, can save 60 percent in area, while replacing general multipliers with constant coefficient versions can save area and lead to speedups of two to four times. Instance-specific designs can also consume lower power. Bit-width optimization of digital filters, for example, has been shown to reduce power consumption by up to 98 percent [2].

Changing an instance-specific design at runtime is generally much slower than changing the inputs of a general circuit, because a new (or partial) configuration must be loaded. Because this may take many tens or hundreds of milliseconds, it is important to carefully choose how a design is specialized.

## 22.1.1 Taxonomy

**Types of instance-specific optimizations**

We can divide the different approaches to optimizing a design for a particular problem instance into three main categories. Table 22.1 lists some examples of the different categories used.

*Constant folding* Constant folding is the process of eliminating unnecessary logic that computes functions with some inputs that never change or that

**TABLE 22.1** ■ Examples of the uses of instance-specific designs

| | Purpose | Example use | Impact |
|---|---|---|---|
| **Constant folding** | Optimize logic for static inputs | Key-specific DES | 60% area reduction |
| **Function adaptation** | Optimize for desired quality of result | Accuracy-guaranteed bit-width optimization [4] | 26% area reduction, 12% latency reduction |
| **Architecture adaptation** | Achieve a specified performance, area, or power target | Custom instruction processors [3] | 72% decrease in runtime for 3% more area |

change only rarely. This logic can be specialized to increase performance and reduce area. Examples of circuits that can benefit from constant folding will be seen later, and a more detailed description of the technique can be found in Section 22.2.3.

*Function adaptation* Function adaptation is the process of altering a circuit's function to achieve a specific quality of result. Typically this involves varying the number of bits used to represent data values or switching between floating-point and fixed-point arithmetic functions. It can also involve adding or removing parts of processing units that affect accuracy—for example, adding or removing stages from a CORDIC circuit. Word-length optimization can be treated automatically (Chapter 23), modifying a circuit's area to meet particular accuracy constraints.

*Architecture adaptation* Architecture adaptation alters the way in which a circuit computes a result while keeping the overall function the same. This can entail introducing additional parallelism to increase speed, serializing existing parallel processing units to save area, or refining processing capabilities to exploit some expected characteristics of the input data. Custom instruction processors (see Figure 22.4 later) are one example of the latter type of architecture adaptation.

## 22.1.2 Approaches

Instance-specific circuits can be produced either by specializing a general-purpose circuit or by starting directly from a "template" that must be instantiated for a particular problem instance before use, as shown in Figure 22.1. Specialization has the advantage that it can often be performed automatically, using techniques such as partial evaluation (Section 22.2). The template approach probably requires the manual design of a template circuit substantially different from the general-purpose architecture, but it can possibly provide a greater level of optimization than what is possible through specializing a general-purpose circuit. It can also offer the advantage that the hardware compilation process may need to be

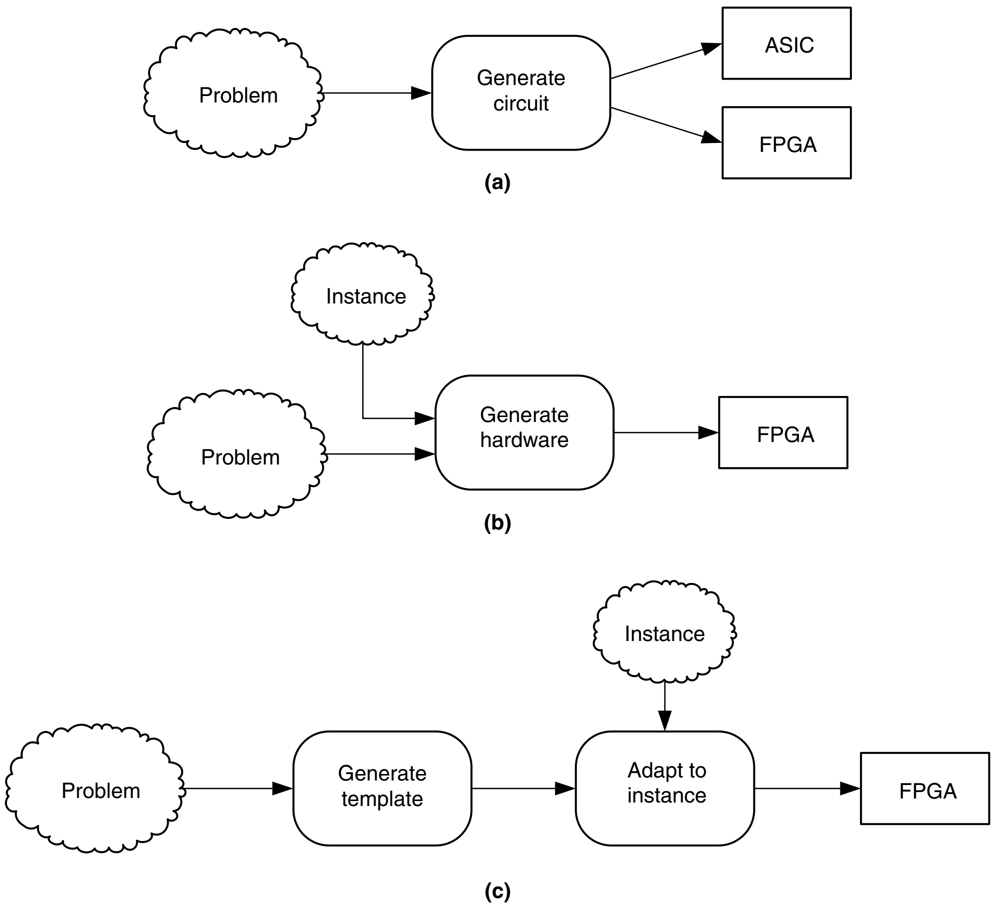


**FIGURE 22.1** ■ General-purpose hardware (a) can be implemented using FPGAs or ASICs. Instance information (b) can be incorporated at hardware generation to produce a specialized circuit. “Template” hardware (c) can be generated and then instantiated for particular problem instances. The reason for the differences between (b) and (c) are that, in (b) the time-consuming process of hardware compilation must be executed for each instance while in (c) hardware compilation may only need to be run once, after which the final circuit bitstream can be amended.

executed only once, with instance-specific information being annotated directly into the bitstream.

In both cases, one or more instance-specific designs will be produced that can be converted into bitstreams through the FPGA design flow (see chapters in Part III). The appropriate bitstream can then be used to configure an FPGA, usually under the control of a general-purpose processor; during the reconfiguration process the FPGA will usually not be able to process data, although some partially reconfigurable devices can support the reconfiguration of some of its resources, while some of its other resources stay operational.

### 22.1.3 Examples of Instance-specific Designs

The benefits of instance-specific design can be illustrated by considering a few examples of its use. In this section we present three examples of specialization by constant folding into an existing design, and two examples of architecture adaptation.

**Constant coefficient multipliers**

If using standard logic cells, multipliers are relatively expensive to implement on FPGAs. A standard combinational multiplier ANDs each bit of input B with all bits of input A (to perform the multiply by 0/1); an adder is then used to sum together the partial products. When one coefficient of the multiplication is constant, however, the required area can be reduced dramatically. The AND functions are unnecessary because multiplying by a fixed 0 or 1 is trivial, and the adders can be eliminated for bits of B that are 0 (and thus have a partial product of 0). Constant coefficient multiplication is a useful operation in many signal-processing applications.

Finite impulse response (FIR) filters contain a set of multiply–add cells that multiply the value of the input signal across a number of cycles with filter coefficients and then sum these values. The multiplier coefficients are properties of the filter and do not change with the input data, but only need adjusting when different filter properties are required. Thus, the generic multipliers in a FIR filter circuit can often be replaced by smaller constant coefficient multipliers. (see Figure 22.2).

Another application that requires multipliers with constant coefficients is conversion from RGB to YUV video signals. This is a matrix multiplication operation where one matrix is constant, allowing specialized multipliers to be used.

**Key-specific crypto-processors**

Cryptographic algorithms are often designed for efficient implementation in both hardware and software. Block ciphers, such as DES and its successor Advanced Encryption Standard (AES), have regular algorithmic structures consisting of simple operations, such as XOR and bit permutation, that are efficiently implemented in hardware.

The DES algorithm consists of 16 "rounds," or processing stages, that can be pipelined for parallel operation. Blocks of 64-bit data are input to the array along with a 56-bit key and processed through each round, with the same key required to decrypt the data at the other end of the communication channel. A single DES round is illustrated in Figure 22.3.

In typical operation it is likely that a crypto-processor is used to process large blocks of data with the same key—for example, when transferring data between a single sender and receiver in a network or encrypting a large file to be saved to disk. It is therefore expected that, in contrast to the data input, the key value will change very slowly.

The shaded area of Figure 22.3 is key generator circuitry that generates the round key from the master key and then uses it as an input to a set of 2-input XOR functions across the data bits.

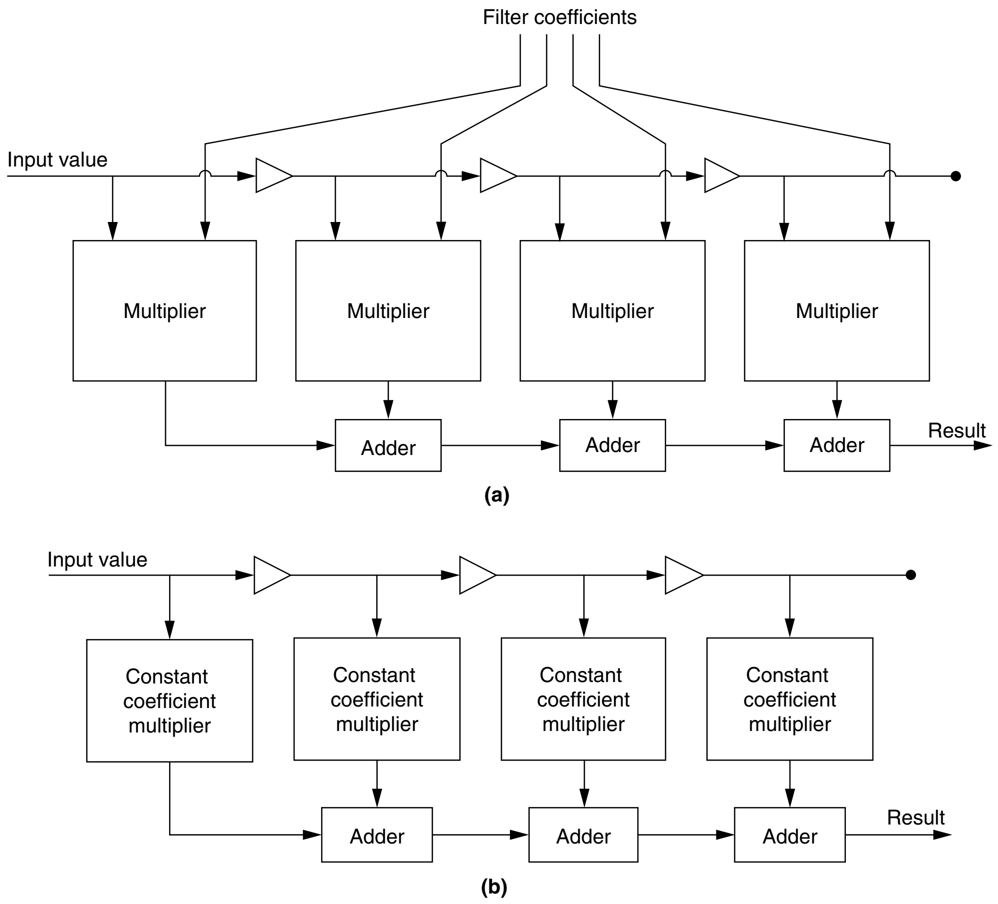


**FIGURE 22.2** ■ FIR filters utilizing (a) general multipliers with variable filter coefficients and (b) instance-specific multipliers specialized to filter coefficients.

When the key value is known, the key generation circuitry can be eliminated and the XOR functions replaced with either wires or inverters [5]. In fact, these inverters can be merged into the substitution stage, eliminating the inverter logic as well [11]. Key-specific crypto-processors can exhibit much higher throughput than general versions, even outperforming ASIC implementations. Area savings are also significant—a relatively simple specialization of a placed DES description can yield area savings of 60 percent when implemented on a Xilinx Virtex FPGA [9].

### Network intrusion detection

Network Intrusion Detection Systems (NIDS) perform deep packet inspection on network packets to identify malicious attacks. Normally, these systems are implemented in software, but on high-speed networks software alone is often unable to process all traffic at the full data rate.

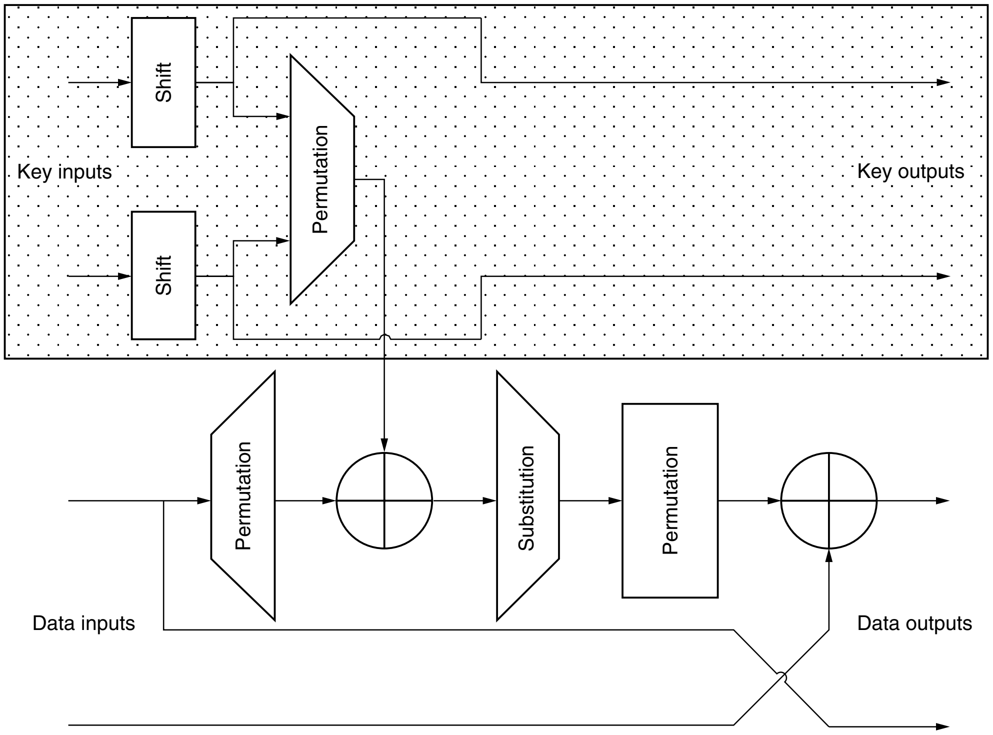


**FIGURE 22.3** ■ A single round of a DES circuit. The shaded area contains key expansion circuitry that can be eliminated in a key-specific DES circuit, allowing the XOR function to be optimized.

The SNORT open source NDIS (see *http://www.snort.org*) uses a rule-based language to detect abnormal network activities. It contains thousands of rules, more than 80 percent of which contain signatures that must be matched against packet contents. Eighty percent of the CPU time for SNORT is consumed by this string-matching task [6]. String matching can be done efficiently in hardware and in particular can be easily optimized for particular search strings. While network data might be expected to arrive at high speed, the rule set changes much more slowly, so string-matching circuitry on FPGAs can be customized to match particular signatures. Section 22.2.5 illustrates in more detail how an instance-specific pattern matcher can be constructed. Further information about instance-specific designs for SAT solving applications can be found in Chapter 29.

### Customizable instruction processors

General-purpose instruction processors are very flexible computational devices. Application-specific instruction processors, in contrast, have been customized to perform particularly well in a particular application area. This is a form of architecture adaptation that can improve performance for particular problem instances while maintaining the flexibility of the overall system.

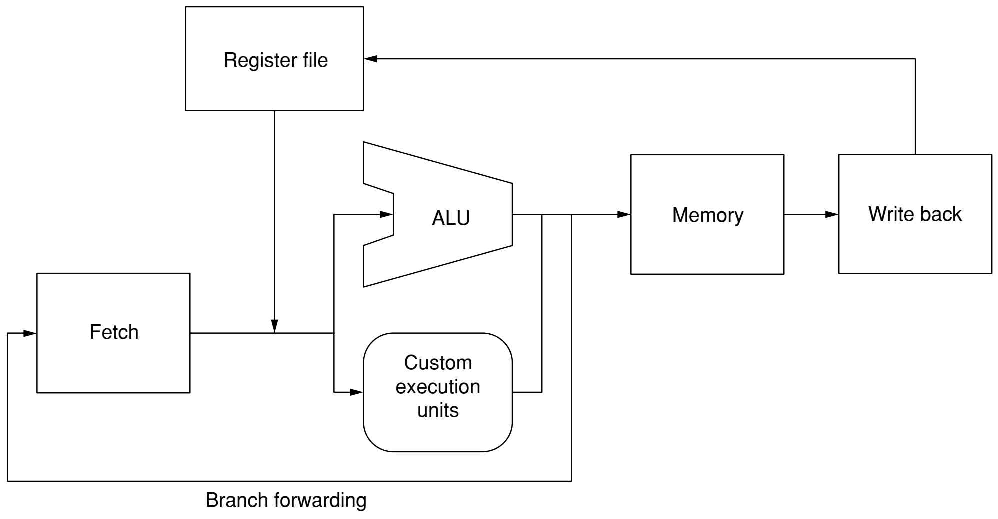


**FIGURE 22.4** ■ A simplified architecture of a custom instruction processor. The standard arithmetic and logic operations are augmented by custom execution units that can accelerate particular applications.

Figure 22.4 illustrates the architecture of a simple custom instruction processor that has standard arithmetic and logic functions implemented by a standard ALU. These functions can be supported by additional custom execution units to accelerate particular applications. The automatic identification of instructions that can benefit from the custom execution units is a topic of active research [1]. Further information about partitioning sequential and parallel programs for software and hardware execution can be found in Chapter 26.

## 22.2 PARTIAL EVALUATION

Partial evaluation is a process that automates specialization in software or hardware. In both cases the motivation is the same: to produce a design that runs faster than the original. In software, partial evaluation can be thought of as a combination of constant folding, loop unrolling, function inlining, and interprocedural analyses; in hardware, constant folding is mainly used as an optimization method.

Partial evaluation is accomplished by detecting fragments of hardware that depend exclusively on variables with fixed values and then optimizing the hardware logic to reduce its area or even eliminate it totally from the design by precomputing the result.

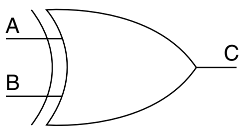
### 22.2.1 Motivation

Partial evaluation can simplify logic, and thus reduce area and increase performance. Figure 22.5 illustrates its impact on a 2-input XOR function. When both inputs are dynamic, the logical function must be implemented; however, when one input is known, a partial evaluator can simplify the circuit. If one input is fixed high, the XOR functions as an inverter and so can be replaced by a 1-input NOT gate; if the input is fixed low, the XOR serves as a wire and the logic can be completely eliminated.

Constant folding propagates constants through a circuit and can substantially simplify logic functions. This can both reduce area (by allowing functions to be implemented using fewer LUTs) and increase performance (by reducing the number of logic levels between registers).

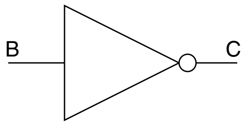
In this chapter we highlight two related uses of partial evaluation for circuits. The first, at the beginning of Section 22.2.4, optimizes generic circuit descriptions for improved performance. That is, circuits are described using clear and easily maintainable but nonoptimal design patterns, which are then automatically optimized during synthesis. The second, in the middle of Section 22.2.4, specializes general circuits when some inputs are static, such as constant coefficient arithmetic.

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



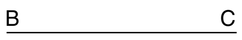
(a)

| B | C |
|---|---|
| 0 | 1 |
| 1 | 0 |



(b)

| B | C |
|---|---|
| 0 | 0 |
| 1 | 1 |



(c)

**FIGURE 22.5** ■ Partial evaluation of an XOR gate. (a) A 2-input XOR function can be specialized, when input A is to become static: (b) an inverter when A is true or (c) a wire when A is false.

### 22.2.2 Process of Specialization

Consider a general circuit $C$ producing output $R$, whose inputs are partitioned into two sets $S$ and $D$.

$$R = C(S,D)$$

This circuit can be specialized for a particular set of $S$ inputs such that it computes the same result for all possible inputs $D$:

$$R = C_{S=X}(D)$$

A partial evaluator is an algorithm that, when supplied with values for the set of inputs $S$ and the circuit $C$, produces a specialized circuit $C_{S=X}$.

$$C_{S=X} = \mathbf{P}(C,S,X)$$

where $S$ is the set of static inputs that are known at compile time, and $D$ is the set of dynamic inputs. The importance of partial evaluation is that the specialized circuit computes precisely the same result as the original circuit, though it may require less hardware to do so.

Relating this framework to the XOR gate example, $R = \mathrm{XOR}(A,B)$, with $S = \{A\}$ and $D = \{B\}$, the two possible simplified functions can be described as

$$R = \mathrm{XOR}_{A=X}(B)$$

for the two possible values of $A$.

$$\mathrm{XOR}_{A=0} = \mathrm{P}(\mathrm{XOR},A,0) = \mathrm{NOT}(B)$$
$$\mathrm{XOR}_{A=1} = \mathrm{P}(\mathrm{XOR},A,1) = B$$

### 22.2.3 Partial Evaluation in Practice

**Constant folding in logical expressions**

Partial evaluation of logic is well understood and has been used to simplify circuit logic for many years. Figure 22.6 gives a simple partial evaluation function, $\mathrm{P}(S)[[X]]$, for optimizing Boolean logic expressions expressed using *not*, *and*, and *or* connectives. The function is parameterized by a set $S$ of pairs mapping static variables to their values and a Boolean expression $X$ represented as a tree.

The function is defined recursively on the structure of Boolean expressions. Cases (1), (2), and (3) are base conditions, indicating that partial evaluation of the Boolean constants True and False always has no effect, and partial evaluation of a variable $a$ returns either the constant value of that variable (if it is contained within the static inputs) or the variable name if it is not static (i.e., remains dynamic).

Case (4) defines partial evaluation of a single-input *not* function. If the subexpression evaluates to logical truth or falsity, this is inverted by the conditional

```
(1) P(S)[[True]]     =  True

(2) P(S)[[ False ]]  =  False

(3) P(S)[[ a ]]      =  if a ∈ dom(S) then P(S)[[ S(a) ]] else a

(4) P(S)[[ ¬ x ]]    =  Let y = P(S)[[ x ]]
                        If y == True then False
                        Else if y == False then True
                        Else ¬ y

(5) P(S)[[ x & y ]]  =  Let x′ = P(S)[[ x ]]
                        Let y′ = P(S)[[ y ]]
                        if(x′ == False || y′ == False) then False
                        Else if x′ == True then y′
                        Else if y′ == True Then x′
                        Else x′ & y′

(6) P(S)[[ x+y ]]    =  Let x′ = P(S)[[ x ]]
                        Let y′ = P(S)[[ y ]]
                        If(x′ == True || y′ == True) then True
                        Else if x′ == False then y′
                        Else if y′ == False then x′
                        Else x+y
```

**FIGURE 22.6** ■ A partial evaluation algorithm for simplifying Boolean logic expressions.

check. Otherwise, the partially evaluated subexpression is returned with the *not* operation.

Cases (5) and (6) define partial evaluation of 2-input *and* and *or* functions. The process is the same: Simplify the subexpressions, precompute the function result if possible, and, if not, return the function with simplified arguments.

As an example, consider the application of this algorithm to the simplification of the XOR function in Figure 22.5. XOR can be described in terms of basic Boolean operators as

$$a \text{ xor } b = (a \& \neg b) + (\neg a \ \& \ b)$$

Partially evaluating when $a$ is asserted, the function is executed:

$$\text{(i)} \quad P(\{a \rightarrow \text{True}\})[[(a \ \& \ \neg b) + (\neg a \ \& \ b)]]$$

Case (6) for simplifying logical-or is used, and the two subexpressions are partially evaluated separately:

$$\text{(ii)} \quad P(\{a \rightarrow \text{True}\})[[a \& \neg b]]$$
$$\text{(iii)} \quad P(\{a \rightarrow \text{True}\})[[\neg a \& b]]$$

Both (ii) and (iii) are partially evaluated by the case for logical-and. For (ii) the two subexpressions are first evaluated as

$$\text{(iv)} \quad P(\{a \rightarrow \text{True}\})[[a]] = \text{True}$$
$$\text{(v)} \quad P(\{a \rightarrow \text{True}\})[[\neg b]] = \neg b$$

In (iv), the variable $a$ is within the static inputs $S$ and thus is simplified to True, while $\neg b$ is unchanged because it does not contain $a$. The results from partially evaluating (iii) are similar:

$$\text{(vi)}\quad P(\{a \rightarrow \text{True}\})[[\neg a]] = P(\{a \rightarrow \text{True}\})[[\neg\text{True}]] = \text{False}$$

$$\text{(vii)}\quad P(\{a \rightarrow \text{True}\})[[b]] = b$$

Equipped with the simplified subexpressions, the expression $a \ \& \ \neg b$ is simplified to $\neg b$ and the expression $\neg a \ \& \ b$ is simplified to False. At the top level this gives a logical-or: $\neg b + \text{False}$:

$$\text{(viii)}\quad P(\{a \rightarrow \text{True}\})[[\neg b + \text{False}]] = \neg b$$

The XOR function reduces to a single inverter; if supplied with $\{a \rightarrow \text{False}\}$ the partial evaluation function instead returns just $b$, indicating the simple wire. This is consistent with the truth tables in Figure 22.5.

The partial evaluation function just given is quite simple and does not capture all possible optimizations. For example, the logic function $a + \neg a$ always evaluates to True, regardless of the value of $a$; however, this expression will not be simplified by this function.

**Unnecessary logic removal**

Another optimization that can be carried out during partial evaluation is removal of dead logic in a design, which does not affect any output and thus is unnecessary. This is a very important optimization because it allows generic hardware blocks computing many functions to be used in designs, with unused functions pruned during synthesis.

As an algorithmic process, logic removal is quite simple and can be formulated in a number of different ways. One of the simplest is to identify each gate whose output is unconnected and eliminate it. By recursively applying this rule we can eliminate acyclic dead logic.

### 22.2.4 Partial Evaluation of a Multiplier

**Optimizing a simple description**

Figure 22.7 shows a shift–add circuit designed for a Xilinx architecture to compute the 3-bit multiplication of two 3-bit inputs. This circuit appears semi-regular, with $x$ and $y$ inputs propagating horizontally and vertically through a triangular array of processing cells. Each processing cell has common features; however, it contains slightly different logic depending on its position in the array.

Creating and maintaining a circuit description that contains and correctly connects the different types of cell is quite complicated. A simpler approach is to exploit the regularity to describe the circuit as an array of a single type of cell that is then partially evaluated during synthesis to produce the circuit in Figure 22.7.

The general cell of the multiplier can be described as shown in Figure 22.8. This cell implements a multiplication operation for 1 bit of $x$ and 1 bit of $y$,

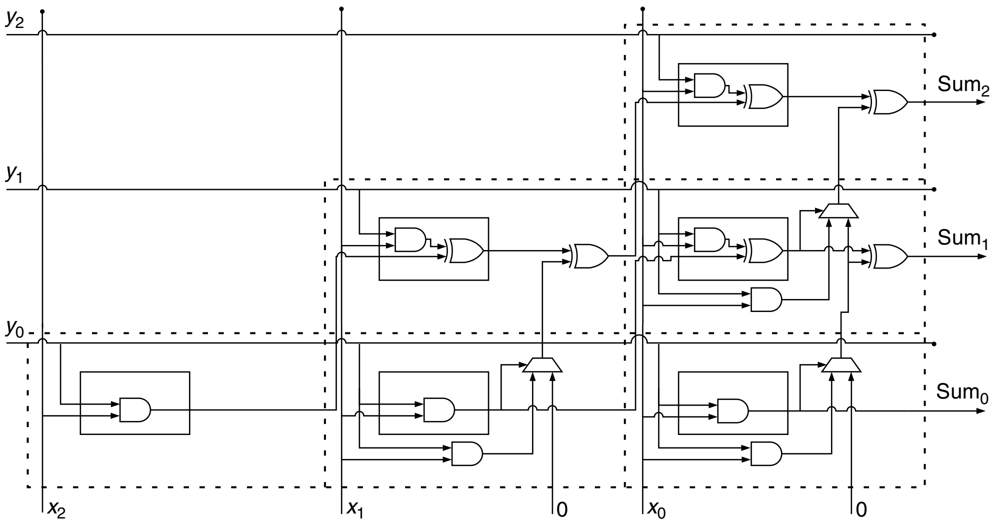


**FIGURE 22.7** ■ A shift–add multiplier circuit that takes two 3-bit inputs and produces a 3-bit output.

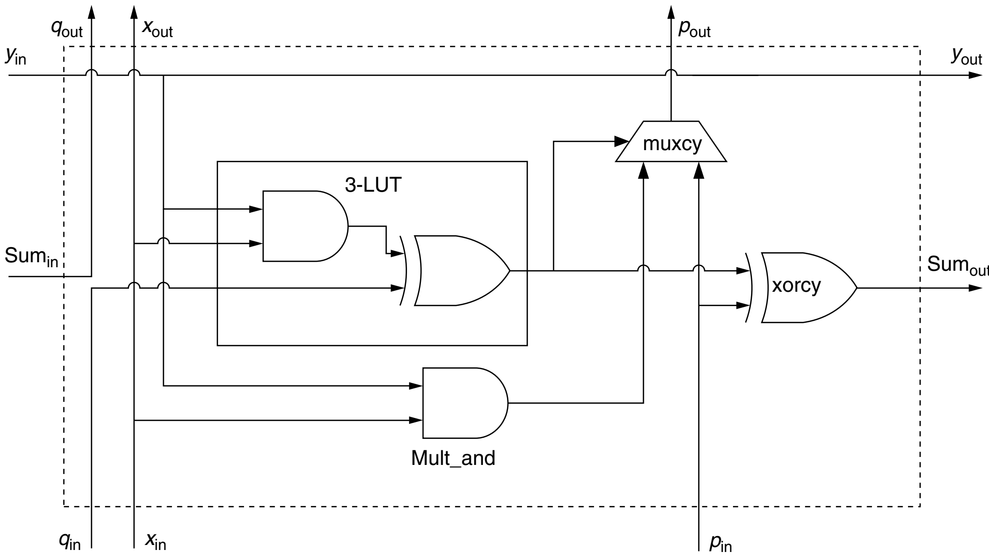


**FIGURE 22.8** ■ This cell design can be replicated in a grid arrangement to create a multiplier.

producing sum and carry-out bits, and can be arranged in a grid to generate a multiplication circuit identical in function to that shown in Figure 22.8. These cells can be implemented densely on Xilinx architectures by using the specialized `mult_and`, `xorcy`, and `muxcy` components in each slice.

Partial evaluation can automatically produce the optimized multiplication circuitry from the initial regular description. The four components within each cell each have their own logical formula. In the case of `mult_and`, `xorcy`, and `muxcy`, no simplification is possible unless we can totally eliminate these functions, because these are fixed resources on the device, compared with the LUT, which can flexibly implement any 4-input function.

The logic of the standard cell can be represented as

$$\begin{aligned} LUT_{\text{out}} &= (Y_{\text{in}}\ \&\ X_{\text{in}})\,\text{xor}\,Q_{\text{in}} = (\neg\,(Y_{\text{in}}\ \&\ X_{\text{in}})\ \&\ Q_{\text{in}}) + ((Y_{\text{in}}\ \&\ X_{\text{in}})\ \&\neg Q_{\text{in}}) \\ AND_{\text{out}} &= (Y_{\text{in}}\ \&\ X_{\text{in}}) \\ P_{\text{out}} &= (LUT_{\text{out}}\ \&\ P_{\text{in}}) + (\neg\,LUT_{\text{out}}\ \&\ AND_{\text{out}}) \\ SUM_{\text{out}} &= (\neg\,LUT_{\text{out}}\ \&\ P_{\text{in}}) + (LUT_{\text{out}}\ \&\ \neg\,P_{\text{in}}) \end{aligned}$$

This logic can be simplified by two operations: removing unconnected logic and constant folding to optimize the logic that remains. Removal of disconnected logic transforms the grid into the triangular array, while constant folding can be performed by the partial evaluation function introduced in Figure 22.6.

For example, for the cells along the bottom in Figure 22.8, inputs $Q_{\text{in}}$ and $P_{\text{in}}$ are all zero. This allows the LUT contents to be optimized by

$$\begin{aligned} LUT_{\text{out}}{}' = {} & P(\{Q_{\text{in}} \rightarrow \text{False}, SUM_{\text{in}} \rightarrow \text{False}, P_{\text{in}} \rightarrow \text{False}\}) \\ & [[(\neg(Y_{\text{in}}\ \&\ X_{\text{in}})\ \&\ Q_{\text{in}}) + ((Y_{\text{in}}\ \&\ X_{\text{in}})\ \&\neg\ Q_{\text{in}})]] = (Y_{\text{in}}\ \&\ X_{\text{in}}) \end{aligned}$$

The function attempts to partially evaluate both branches of the OR expression. On the left branch, $\neg(Y_{\text{in}}\ \&\ X_{\text{in}})$ cannot be further optimized and so is left intact; however, $Q_{\text{in}}$ is known to be false, so the entire left branch must be false and thus is eliminated. On the right branch, $\neg Q_{\text{in}}$ is evaluated to true and eliminated from the expression, leaving $(Y_{\text{in}}\ \&\ X_{\text{in}})$ as the simplified function for the LUT contents.

$AND_{\text{out}}$ cannot be simplified because both $Y_{\text{in}}$ and $X_{\text{in}}$ are unknown. Neither can $P_{\text{out}}$ because, although it can be partially optimized (because $P_{\text{in}}$ is false), it is a fixed component available on the FPGA that cannot be simplified. Partial evaluation of $SUM_{\text{out}}$ does succeed in eliminating logic:

$$\begin{aligned} SUM_{\text{out}}{}' = {} & P(\{Q_{\text{in}} \rightarrow \text{False}, SUM_{\text{in}} \rightarrow \text{False}, P_{\text{in}} \rightarrow \text{False}\}) \\ & [[(\neg LUT_{\text{out}}\ \&\ P_{\text{in}}) + (LUT_{\text{out}}\ \&\ \neg P_{\text{in}})]] = LUT_{\text{out}} \end{aligned}$$

The result of this partial evaluation is that the bottom cells of the multiplier are optimized to remove the unnecessary `xorcy` component and to simplify the 3-input LUT function into a basic 2-input AND function.

### Functional specialization for constant inputs

If some of the input values to the multiplication circuit are known statically, we can apply constant folding to eliminate further logic. For example, assume that $x_1$ is static and always zero. Partially evaluating the cell logic under the new assumption that $\{X_{\text{in}} \rightarrow \text{False}\}$ we find that the entire cell can be eliminated and replaced with pure routing. The simplified cell is shown in Figure 22.9.

Because a single bit of the *x* input is shared with an entire column of the multiplier, this specialized cell can be used for the full column, replacing all the logic with routing, as shown in Figure 22.10; this arrangement in turn allows optimizations to be applied to the second LUT in the final column to eliminate the XOR function (not shown in the figure so that the routing can be seen).

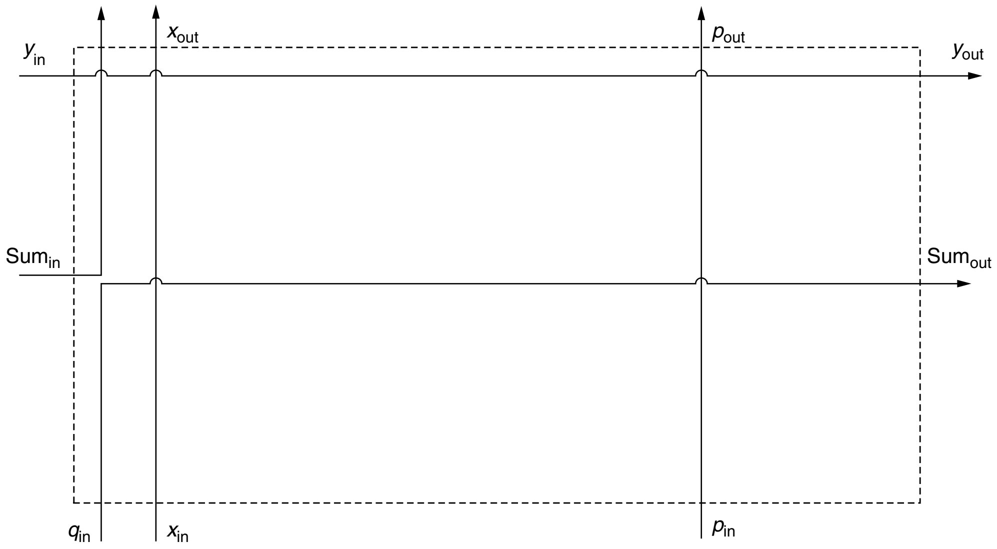


**FIGURE 22.9** ■ The impact of partial evaluation on multiplier cell logic when $X_{in}$ = False.

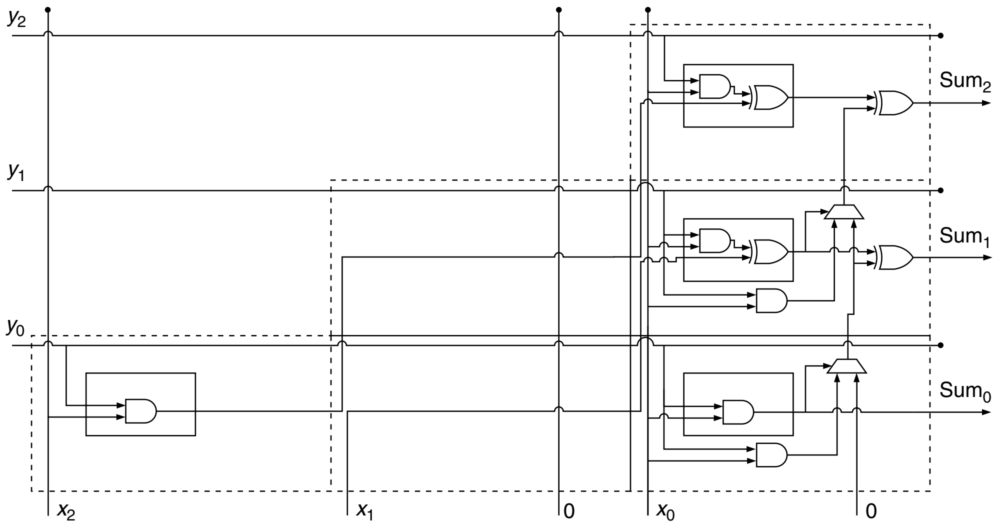


**FIGURE 22.10** ■ Multiplier circuit specialized by eliminating the center column when $x_i$ is always zero.

When an $x$ value is known to be true, partial evaluation can still carry out some optimizations. However, it does not offer the significant advantages that result when $x$ is false. The LUT can again be optimized to a 2-input function and the `mult_and` component can be eliminated. This is not very significant, however—the `mult_and` component is already present on the device, so no area is saved, and it is utilized in parallel with the (slower) LUT so there is also no performance gain.

**Geometric specialization**
High-performance FPGA designs often include layout information to produce good placements with low routing delays (see Chapter 17). Specialization of placed designs may lead to nonoptimal results if the placement is not updated to reflect eliminated logic. Automatic placement is not affected, since partial evaluation is usually carried out at the synthesis stage prior to placement and routing. However, when hand-placed designs are specialized, the effect can be to introduce unnecessary delays by failing to compact components. These gaps can also prevent effective use of freed logic because it is fragmented among other components. To ensure a good placement of specialized designs it is necessary to optimize placement information, compacting the circuit. This can be achieved in a framework that allows partial evaluation prior to placement position generation [8] or by describing circuit layouts in a way that adapts when the circuit is specialized [12].

## 22.2.5 Partial Evaluation at Runtime

Pattern matching is a relatively simple operation that can be performed efficiently in hardware. It is useful in a range of fields but is of particular interest in networking for inspecting the contents of data packets.

Figure 22.11 illustrates a simple general pattern matcher made up of a repeating bit-level matcher cell. Each cell contains a pattern and a mask value, which can be loaded separately from the data to be matched. Input data is streamed in 1 bit per cycle; if the mask value for a particular bit position is set, the cell for that position checks the current data value against the bit pattern.

The pattern matcher requires one LUT and three registers for each bit in the data pattern. However, it is likely that the pattern and mask values will change much more slowly than the data input, so it is reasonable to investigate the potential for partial evaluation to optimize this circuit for fixed patterns.

When the pattern and mask are fixed, the registers storing their values can be eliminated and the logic in the LUTs can be optimized. Figure 22.12 shows how the pattern matcher can be optimized for a pattern of “10X1” (the third pattern bit is a “don’t care,” as specified by the mask of “1101”). This circuit uses fewer registers and three LUTs rather than four. The significance of this particular way of optimizing is that the pattern matcher’s structure has mostly been maintained and thus this specialization can be carried out at runtime.

Changes to the mask require routing changes—complex, though far from impossible at runtime; however, the pattern to be matched can be changed merely by updating the LUT contents.

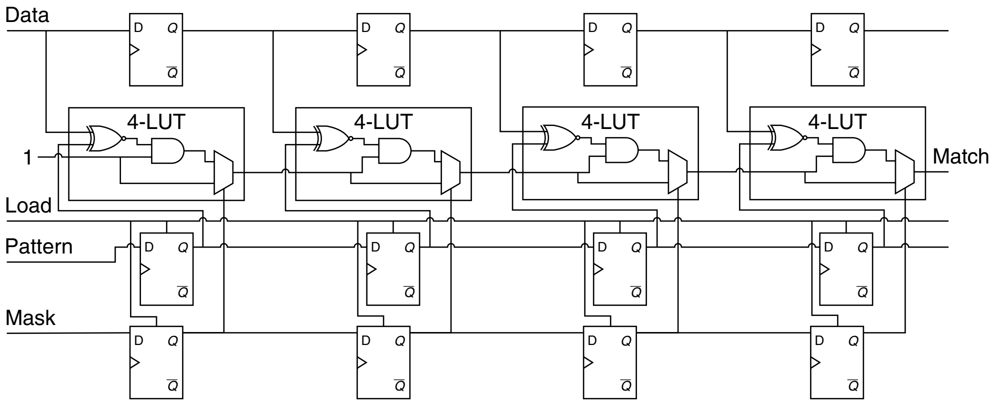


**FIGURE 22.11** ■ A general bit-level pattern matcher, shown for 4-bit patterns. The pattern matcher circuit is controlled by a pattern and a mask, which can be loaded by asserting the load signal. If the mask bit is set for a particular position, the matcher will attempt to detect a match between the pattern bit and the data bit.

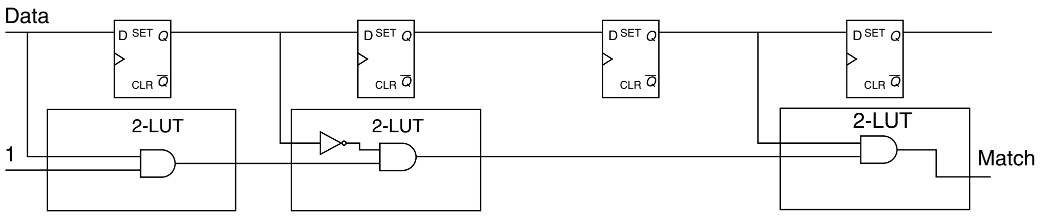


**FIGURE 22.12** ■ An instance-specific pattern matcher optimized for a mask of `1101` and pattern of `10X1` requires only three LUTs and four registers.

### 22.2.6 FPGA-specific Concerns

**LUT mapping**

Recall the pattern matcher example from the previous section, where we showed one partial evaluation of the circuit for a particular pattern. In this case partial evaluation significantly simplified the contents of each LUT, from a 4-input function to a much simpler 2-input function.

It is important that, in contrast to ASICs, there is often no performance advantage to be gained by reducing the complexity of logic functions in an FPGA unless the number of LUTs required to implement those functions is reduced. The propagation delay of a LUT is independent of the function it implements; thus, there is no gain in reducing a 4-input function to a 2-input function within the same LUT (although it does allow routing resources to be freed for other uses).

For runtime specialization, it may be desirable to maintain much of the original circuit structure. However, when partial evaluation is carried out at compile time it should be performed before logic is mapped to LUTs, giving more scope for improvements in circuit area and performance. Figure 22.13 shows that the

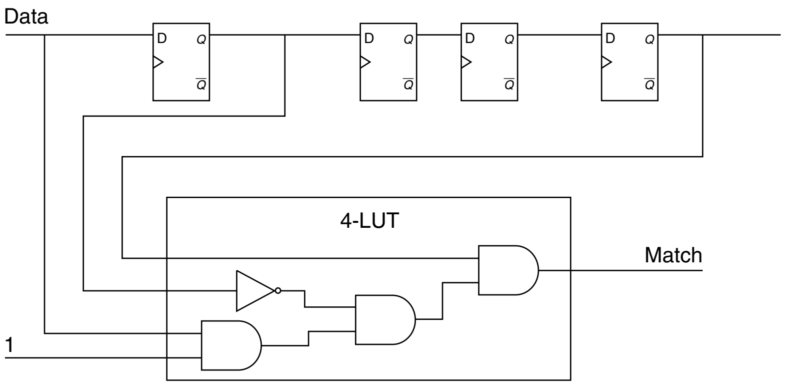


**FIGURE 22.13** ■ The instance-specific pattern matcher from Figure 22.12 can be implemented using a single 4-LUT rather than three 2-LUTs.

specialized pattern matcher can indeed be implemented using one 4-LUT rather than three 2-LUTs, with higher performance and lower area requirements than the version partially evaluated at runtime.

In fact, the static 1-input can also be eliminated from this LUT; however, it has been left to indicate that this LUT structure can be used as part of a chain in a larger pattern matcher.

## Static resources

As alluded to in the multiplier example, the existence of specific resources on an FPGA in addition to LUTs, such as carry chain logic, poses a problem for automatic partial evaluation algorithms. Not only can this logic not be simplified (for example, the `xorcy` gate cannot be replaced with an inverter), in some cases it cannot be eliminated at all because of routing constraints (carry signals must propagate through `muxcy` multiplexers, for example, regardless of necessity).

Furthermore, it is often important to maintain use of the dedicated carry chain, even though significantly simpler logic could perhaps be generated after partial evaluation, because the carry chain is designed to propagate carry signals very quickly—and much faster than the general routing fabric.

## Verification of runtime specialization

Dynamic specialization at runtime poses additional verification problems over and above verification of an original design. While a circuit may have been verified through extensive simulation or formal methods prior to synthesis, when it is specialized at runtime it is possible for new errors to be introduced.

To avoid this it is necessary to ensure that the algorithms that apply partial evaluation at runtime have themselves been verified. Formal proof is an appropriate methodology for this problem, since it is necessary to check a generic property of the algorithm applied to all circuits rather than any particular specialization operation.

Although formal verification has been applied to partial evaluation algorithms for specialization of FPGA circuits [7, 14], it remains a relatively unexplored area.

## 22.3 SUMMARY

This chapter described instance-specific design, which offers the opportunity to exploit the reconfigurable nature of FPGAs to improve performance by tailoring circuits to particular problem instances. It can be broadly categorized into three techniques: constant folding, which can be applied when some inputs are static; function adaptation, which alters the function of circuitry to produce a certain quality of result; and architecture adaptation, in which the circuit architecture is adapted without affecting its functional behavior.

The level of automation that can be applied varies among these approaches. Constant folding can often be carried out automatically using partial evaluation techniques. Function adaptation can be performed by varying bit widths and arithmetic methods in parameterized IP cores. Tools, such as Quartz (for low-level design) [12] or ASC (for stream architectures) [10], can produce highly parameterized circuit cores where design parameters can be traded off against each other to achieve the desired requirements in area, speed, and power consumption. Architecture adaptation, such as adding additional processing units to instruction processors, is typically much less automated. The designer must create separate implementations of the different architectures, optimizing each of them somewhat independently.

### References

[1] K. Atasu, R. Dimond, O. Mencer, W. Luk, C. Özturan, G. Dündar. Optimizing instruction-set extensible processors under data bandwidth constraints. *Proceedings of Design, Automation and Test in Europe Conference*, 2007.

[2] G. A. Constantinides. Perturbation analysis for word-length optimization. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[3] R. Dimond, O. Mencer, W. Luk. Application-specific customisation of multi-threaded soft processors. *IEE Proceedings on Computers and Digital Techniques*, May 2006.

[4] D. Lee, A. Abdul Gaffar, R.C.C. Cheung, O. Mencer, W. Luk, G. A. Constantinides. Accuracy guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 2006.

[5] J. Leonard, W. Magione-Smith. A case study of partially evaluated hardware circuits: Key-specific DES. *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, 1997.

[6] E. P. Markatos, S. Antonatos, M. Polychronakis, K. G. Anagnostakis. Exclusion-based signature matching for intrusion detection. *Proceedings of IASTED International Conference on Communication and Computer Networks*, 2002.

[7] S. McKeever, W. Luk. Provably-correct hardware compilation tools based on pass separation techniques. *Formal Aspects of Computing*, June 2006.

[8] S. McKeever, W. Luk, A. Derbyshire. Towards verifying parametrised hardware libraries with relative placement information. *Proceedings of the 36th IEEE Hawaii International Conference on System Sciences*, 2003.

[9] S. McKeever, W. Luk, A. Derbyshire. Compiling hardware descriptions with relative placement information for parameterised libraries. *Proceedings of International Conference on Formal Methods in Computer-Aided Design*, LNCS 2517, 2002.

[10] O. Mencer. ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design*, August 2006.

[11] C. Patterson. High performance DES encryption in Virtex FPGAs using JBits. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[12] O. Pell, W. Luk. Compiling higher-order polymorphic hardware descriptions into parametrised VHDL libraries with flexible placement information. *Proceedings of the International Workshop on Field-Programmable Logic and Applications*, 2006.

[13] O. Pell, W. Luk. Quartz: A framework for correct and efficient reconfigurable design. *Proceedings of the International Conference on Reconfigurable Computing and FPGAs,* 2005.

[14] K. W. Susanto, T. Melham. Formally analyzed dynamic synthesis of hardware. *Journal of Supercomputing* 19(1), 2001.