Chapter 2

Architecting Area

This chapter discusses the second of three primary physical characteristics of a digital design: area. Here we also discuss methods for architectural area optimization in an FPGA.

We will discuss area reduction based on choosing the correct topology. Topology refers to the higher-level organization of the design and is not device specific. Circuit-level reduction as performed by the synthesis and layout tools refers to the minimization of the number of gates in a subset of the design and may be device specific.

A topology that targets area is one that reuses the logic resources to the greatest extent possible, often at the expense of throughput (speed). Very often this requires a recursive data flow, where the output of one stage is fed back to the input for similar processing. This can be a simple loop that flows naturally with the algorithm or it may be that the logic reuse is complex and requires special controls. This section describes both techniques and describes the necessary consequences in terms of performance penalties.

During the course of this chapter, we will discuss the following topics in detail:

- Rolling up the pipeline to reuse logic resources in different stages of a computation.
- Controls to manage the reuse of logic when a natural flow does not exist.
- Sharing logic resources between different functional operations.
- The impact of reset on area optimization.
  Impact of FPGA resources that lack reset capability.
  Impact of FPGA resources that lack set capability.
  Impact of FPGA resources that lack asynchronous reset capability.
  Impact of RAM reset.
  Optimization using set/reset pins for logic implementation.

2.1 ROLLING UP THE PIPELINE

The method of "rolling up the pipeline" is the opposite operation to that described in the previous chapter to improve throughput by "unrolling the loop" to achieve maximum performance. When we unrolled the loop to create a pipeline, we also increased the area by requiring more resources to hold intermediate values and replicating computational structures that needed to run in parallel. Conversely, when we want to minimize the area of a design, we must perform these operations in reverse; that is, roll up the pipeline so that logic resources can be reused. Thus, this method should be used when optimizing highly pipelined designs with duplicate logic in the pipeline stages.

Rolling up the pipeline can optimize the area of pipelined designs with duplicated logic in the pipeline stages.

Consider the example of a fixed-point fractional multiplier. In this example, A is represented in normal integer format with the fixed point just to the right of the LSB, whereas the input B has a fixed point just to the left of the MSB. In other words, B scales A from 0 to 1.

```
module mult8(
 output [7:0]  product,
 input  [7:0]  A,
 input  [7:0]  B,
 input         clk);
 reg    [15:0] prod16;

 assign product = prod16[15:8];

 always @(posedge clk)
   prod16 <= A * B;

endmodule
```

With this implementation, a new product is generated on every clock. There isn't an obvious pipeline in this design as far as distinct sets of registers, but note that the multiplier itself is a fairly long chain of logic that is easily pipelined by adding intermediate register layers. It is this multiplier that we wish to "roll up." We will roll this up by performing the multiply with a series of shift and add operations as follows:

```
module mult8(
 output           done,
 output reg [7:0] product,
 input      [7:0] A,
 input      [7:0] B,
 input            clk,
 input            start);
 reg        [4:0] multcounter; // counter for number of
                                  shift/adds
```

```
reg        [7:0] shiftB; // shift register for B
reg        [7:0] shiftA; // shift register for A

wire adden; // enable addition

assign adden = shiftB[7] & !done;
assign done = multcounter[3];

always @(posedge clk) begin
  // increment   multiply counter for shift/add ops
  if(start)      multcounter <= 0;
  else if(!done) multcounter <= multcounter + 1;

  // shift register for B
  if(start) shiftB <= B;
  else shiftB[7:0] <= {shiftB[6:0], 1'b0};

  // shift register for A
  if(start) shiftA <= A;
  else shiftA[7:0] <= {shiftA[7], shiftA[7:1]};

  // calculate multiplication
  if(start)      product <= 0;
  else if(adden) product <= product + shiftA;
end
endmodule
```

The multiplier is thus architected with an accumulator that adds a shifted version of A depending on the bits of B as shown in Figure 2.1. Thus, we completely eliminate the logic tree necessary to generate a multiply within a single clock and replace it with a few shift registers and an adder. This is a very compact form of a multiplier but will now require 8 clocks to complete a multiplication. Also note that no special controls were necessary to sequence through this multiply operation. We simply relied on a counter to tell us when to stop the shift and add operations. The next section describes situations where this control is not so trivial.

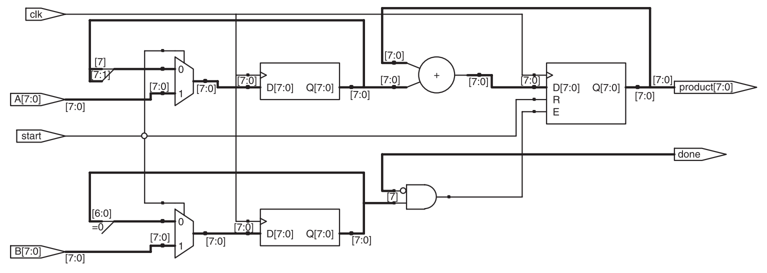

Figure 2.1 Shift/add multiplier.

2.2 CONTROL-BASED LOGIC REUSE

Sharing logic resources oftentimes requires special control circuitry to determine which elements are input to the particular structure. In the previous section, we described a multiplier that simply shifted the bits of each register, where each register was always dedicated to a particular input of the running adder. This had a natural data flow that lent itself well to logic reuse. In other applications, there are often more complex variations to the input of a resource, and certain controls may be necessary to reuse the logic.

Controls can be used to direct the reuse of logic when the shared logic is larger than the control logic.

To determine this variation, a state machine may be required as an additional input to the logic.

Consider the following example of a low-pass FIR filter represented by the equation:

$$Y = \text{coeffA} * X[0] + \text{coeffB} * X[1] + \text{coeffC} * X[2]$$

```
module lowpassfir(
 output reg [7:0] filtout,
 output reg       done,
 input            clk,
 input      [7:0] datain, // X[0]
 input            datavalid, // X[0] is valid
 input      [7:0] coeffA, coeffB; coeffC); // coeffs for
                                              low pass
                                              filter
 // define input/output samples
 reg        [7:0] X0, X1, X2;
 reg              multdonedelay;
 reg              multstart; // signal to multiplier to
                                begin computation
 reg        [7:0] multdat;
 reg        [7:0] multcoeff; // the registers that are
                                multiplied together
 reg        [2:0] state; // holds state for sequencing
                            through mults
 reg        [7:0] accum; // accumulates multiplier products
 reg                 clearaccum; // sets accum to zero
 reg        [7:0] accumsum;
 wire             multdone; // multiplier has completed
 wire       [7:0] multout; // multiplier product

 // shift-add multiplier for sample-coeff mults
 mult8 × 8 mult8 × 8(.clk(clk), .dat1(multdat),
      .dat2(multcoeff), .start(multstart),
      .done(multdone), .multout(multout));
```

```
    always @(posedge clk) begin
      multdonedelay <= multdone;

    // accumulates sample-coeff products
    accumsum <= accum + multout[7:0];

    // clearing and loading accumulator
    if(clearaccum)          accum <= 0;
    else if(multdonedelay) accum <= accumsum;
// do not process state machine if multiply is not done
case(state)
      0: begin
      // idle state
      if(datavalid) begin
       // if a new sample has arrived
       // shift samples
       X0     <= datain;
       X1     <= X0;
       X2     <= X1;
       multdat  <= datain;   // load mult
       multcoeff <= coeffA;
       multstart <= 1;
       clearaccum <= 1; // clear accum
       state   <= 1;
      end
      else begin
       multstart <= 0;
       clearaccum <= 0;
       done    <= 0;
      end
      end
      1: begin
      if(multdonedelay) begin
       // A*X[0] is done, load B*X[1]
       multdat  <= X1;
       multcoeff <= coeffB;
       multstart <= 1;
       state   <= 2;
      end
      else begin
       multstart <= 0;
       clearaccum <= 0;
       done    <= 0;
      end
      end
      2: begin
      if(multdonedelay) begin
       // B*X[1] is done, load C*X[2]
       multdat  <= X2;
```

```
       multcoeff <= coeffC;
       multstart <= 1;
       state   <= 3;
      end
      else begin
       multstart <= 0;
       clearaccum <= 0;
       done    <= 0;
      end
      end
      3: begin
      if(multdonedelay) begin
       // C*X[2] is done, load output
       filtout  <= accumsum;
       done    <= 1;
       state   <= 0;
      end
      else begin
       multstart <= 0;
       clearaccum <= 0;
       done    <= 0;
      end
      end
      default
       state   <= 0;
     endcase
   end
 endmodule
```

In this implementation, only a single multiplier and accumulator are used as can be seen in Figure 2.2. Additionally, a state machine is used to load coefficients and registered samples into the multiplier. The state machine operates on every combination of coefficients and samples: `coeffA*X[0]`, `coeffB*X[1]`, `and coeffC*X[2]`.

The reason this implementation required a state machine is because there was no natural flow to the recursive data as there was with the shift and add multiplier

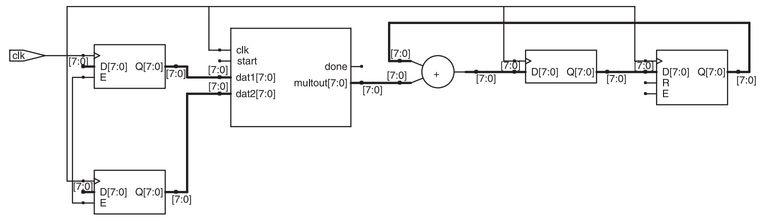

Figure 2.2 FIR with one MAC.

example. In this case, we had arbitrary registers that represented the inputs required to create a set of products. The most efficient way to sequence through the set of multiplier inputs was with a state machine.

2.3 RESOURCE SHARING

When we use the term *resource sharing*, we are not referring to the low-level optimizations performed by FPGA place and route tools (this is discussed in later chapters). Instead, we are referring to higher-level architectural resource sharing where different resources are shared across different functional boundaries. This type of resource sharing should be used whenever there are functional blocks that can be used in other areas of the design or even in different modules.

A simple example of resource sharing is with system counters. Many designs use multiple counters for timers, sequencers, state machines, and so forth. Oftentimes, these counters can be pulled to a higher level in the hierarchy and distributed to multiple functional units. For instance, consider modules A and B. Each of these modules uses counters for a different reason. Module A uses the counter to

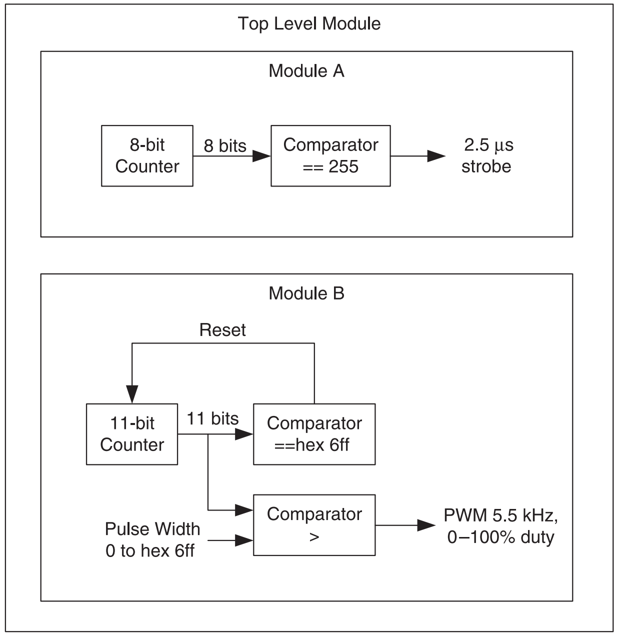

Figure 2.3 Separated counters.

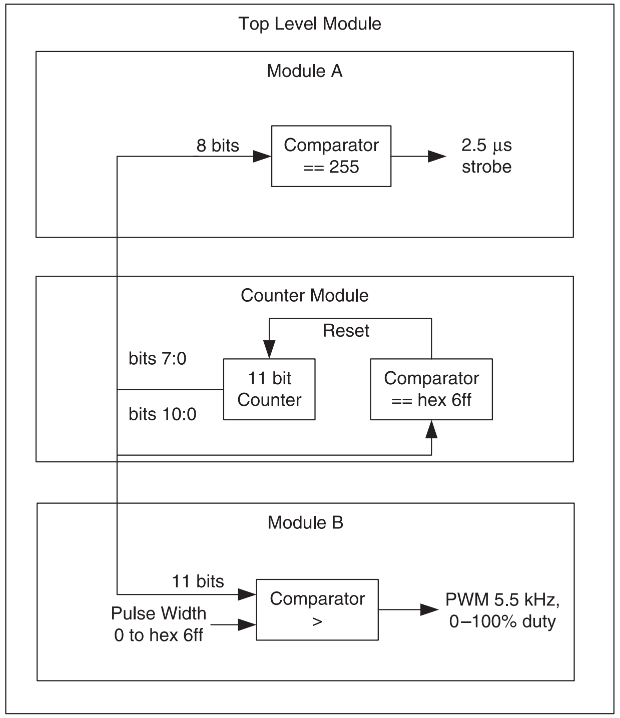

Figure 2.4 Shared counter.

flag an operation every 256 clocks (at 100 MHz, this would correspond with a trigger every 2.56 μs). Module B uses a counter to generate a PWM (Pulse Width Modulated) pulse of varying duty cycle with a fixed frequency of 5.5 kHz (with a 100-MHz system clock, this would correspond with a period of hex 700 clocks).

Each module in Figure 2.3 performs a completely independent operation. The counters in each module also have completely different characteristics. In module A, the counter is 8 bits, free running, and rolls over automatically. In module B, the counter is 11 bits and resets at a predefined value (1666). Nonetheless, these counters can easily be merged into a global timer and used independently by modules A and B as shown in Figure 2.4.

Here we were able to create a global 11-bit counter that satisfied the requirement of both module A and module B.

For compact designs where area is the primary requirement, search for resources that have similar counterparts in other modules that can be brought to a global point in the hierarchy and shared between multiple functional areas.

2.4 IMPACT OF RESET ON AREA

A common misconception is that the reset structures are always implemented in a purely global sense and have little effect on design size. The fact is that there are a number of considerations to take into account relative to area when designing a reset structure and a corresponding number of penalties to pay for a suboptimal design.

The first effect on area has to do with the insistence on defining a global set/reset condition for every flip-flop. Although this may seem like good design practice, it can often lead to a larger and slower design. The reason for this is because certain functions can be optimized according to the fine-grain architecture of the FPGA, but bringing a reset into every synchronous element can cause the synthesis and mapping tools to push the logic into a coarser implementation.

An improper reset strategy can create an unnecessarily large design and inhibit certain area optimizations.

The next sections describe a number of different scenarios where the reset can play a significant role in the speed/area characteristics and how to optimize accordingly.

2.4.1 Resources Without Reset

This section describes the impact that a global reset will have on FPGA resources that do not have reset available. Consider the following example of a simple shift register:

IMPLEMENTATION 1 : *Synchronous Reset*

```
always @(posedge iClk)
 if(!iReset) sr <= 0;
 else sr        <= {sr[14:0], iDat};
```

IMPLEMENTATION 2 : *No Reset*

```
always @(posedge iClk)
 sr <= {sr[14:0], iDat};
```

The differences between the above two implementations may seem trivial. In one case, the flip-flops have resets defined to be logic-0, whereas in the other implementation, the flip-flops do not have a defined reset state. The key here is that if we wish to take advantage of built-in shift-register resources available in the FPGA, we will need to code it such that there is a direct mapping. If we were targeting a Xilinx device, the synthesis tool would recognize that the shift-register SRL16 could be used to implement the shift register as shown in Figure 2.5.

Note that no resets are defined for the SRL16 device. If resets are defined in our design, then the SRL16 unit could not be used as there are no reset control

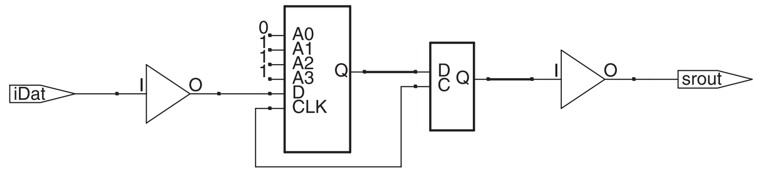

Figure 2.5 Shift register implemented with SRL16 element.

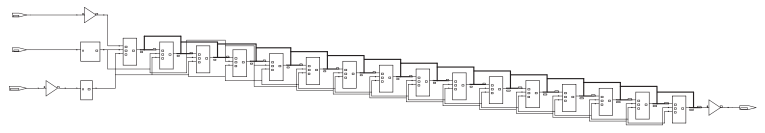

Figure 2.6 Shift register implemented with flip-flops.

Table 2.1 Resource Utilization for Shift Register Implementations

Implementation	Slices slice	Flip-flops
Resets defined	9	16
No resets defined	1	1

signals to the resource. The shift register would be implemented as discrete flip-flops as shown in Figure 2.6. The difference is drastic as summarized in Table 2.1.

An optimized FPGA resource will not be used if an incompatible reset is assigned to it. The function will be implemented with generic elements and will occupy more area.

By removing the reset signals, we were able to reduce 9 slices and 16 slice flip-flops to a single slice and single slice flip-flop. This corresponds with an optimally compact and high-speed shift-register implementation.

2.4.2 Resources Without Set

Similar to the problem raised in the previous section, some internal resources lack any type of set capability. An example is that of an 8×8 multiplier:

```
module mult8(
  output reg [15:0] oDat,
  input       iReset, iClk,
  input   [7:0] iDat1, iDat2,
  );
```

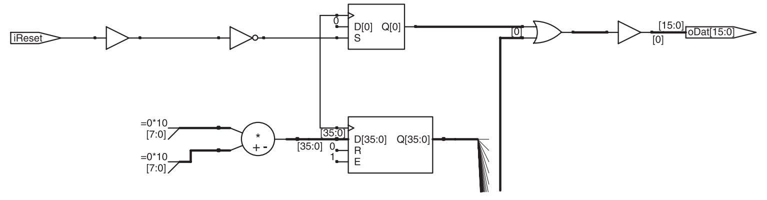

Figure 2.7 Set implemented with external logic.

Table 2.2 Resource Utilization for Set and Reset Implementations

Implementation	Slices slice	Flip-flops	LUTs	Mult16
Reset	9	16	1	1
Set	1	1	1	1

```
    always @(posedge iClk)
      if(!iReset) oDat <= 16'hffff;
      else    oDat <= iDat1 * iDat2;
   endmodule
```

Again, the only variation to the above code will be the reset condition. Unlike the shift-register example, the multiplier resources in most FPGAs have built-in reset resources. They do not, however, typically have set resources. If the set functionality as described above (16'hffff instead of simply 0) is required, the circuit illustrated in Figure 2.7 will be implemented.

Here an additional gate for each output is required to set the output when the reset is active. The reset on the multiplier, in this case, will go unused. The resource usage between the set and reset implementations is shown in Table 2.2.

By changing the multiplier set to a reset operation, we are able to reduce 9 slices and 16 slice flip-flops to a single slice and single slice flip-flop. This corresponds with an optimally compact and high-speed multiplier implementation.

2.4.3 Resources Without Asynchronous Reset

Many new high-performance FPGAs provide built-in multifunction modules that have general applicability to a wide range of applications. Typically, these resources have some sort of reset functionality but are constrained relative to the type of reset topology. Here we will look at Xilinx-specific multiply–accumulate modules for DSP (Digital Signal Processing) applications. The internal structure of a built-in DSP is typically not flexible to varying reset strategies.

DSPs and other multifunction resources are typically not flexible to varying reset strategies.

Consider the following code for a multiply and accumulate operation:

```
module dspckt(
 output reg [15:0] oDat,
 input       iReset, iClk,
 input      [7:0] iDat1, iDat2);
 reg        [15:0] multfactor;

 always @(posedge iClk or negedge iReset)
  if(!iReset) begin
   multfactor <= 0;
   oDat       <= 0;
  end
  else begin
   multfactor <= (iDat1 * iDat2);
   oDat       <= multfactor + oDat;
 end
endmodule
```

The above code defines a multiply–accumulate function with asynchronous resets. The DSP structures inside a Xilinx Virtex-4 device, for example, have only synchronous reset capabilities as shown in Figure 2.8.

The reset signal here is fed directly into the reset pin of the MAC core. To implement an asynchronous reset as shown in the above code example, on the other hand, the synthesis tool must create additional logic outside of the DSP core.

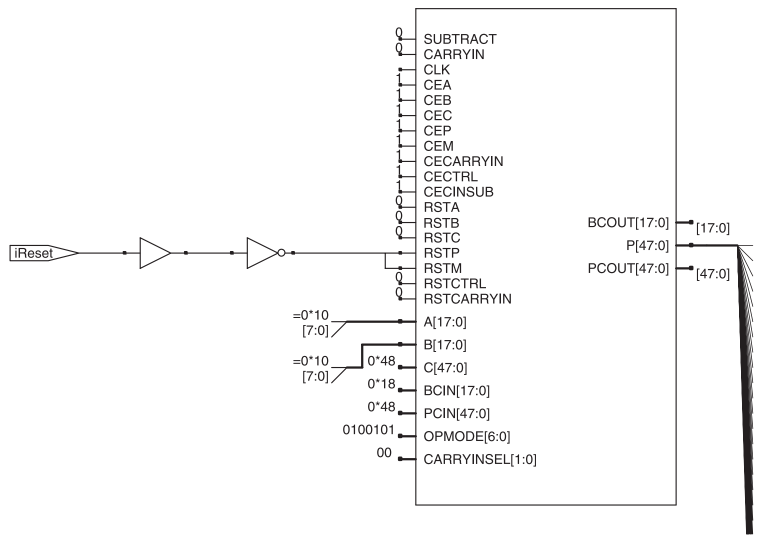

Figure 2.8 Xilinx DSP block with synchronous reset.

Table 2.3 Resource Utilization for Synchronous and Asynchronous Resets

Architecture	Slices	Flip-flops	LUTs	DSPs
Async Reset	17	32	16	1
Sync Reset	0	0	0	1

Comparing this to a similar structure using synchronous resets, we are able to obtain the results shown in Table 2.3.

When the synchronous reset was used, the synthesis tool was able to use the DSP core available in the FPGA device. By using a different reset than what was available on this device, however, a significant amount of logic was created around it to implement the asynchronous reset.

2.4.4 Resetting RAM

There are reset resources in many built-in RAM (Random Access Memory) resources for FPGAs, but similar to the DSP resource described in the previous sections, often only synchronous resets are available. Attempting to implement an asynchronous reset on a RAM module can be catastrophic to area optimization because there are not smaller elements that can be optimally used to construct a RAM (like a multiplier and an adder can be stitched together to form a MAC module) other than smaller RAM resources, nor can the synthesis tool easily add a few gates to the output to emulate this functionality.

Resetting RAM is usually poor design practice, particularly if the reset is asynchronous.

Consider the following code:

```
module resetckt(
 output reg [15:0] oDat,
 input       iReset, iClk, iWrEn,
 input      [7:0] iAddr, oAddr,
 input      [15:0] iDat);
 reg        [15:0] memdat [0:255];

 always @(posedge iClk or negedge iReset)
 if(!iReset)
  oDat          <= 0;
 else begin
  if(iWrEn)
    memdat[iAddr] <= iDat;

  oDat          <= memdat[oAddr];
 end

endmodule
```

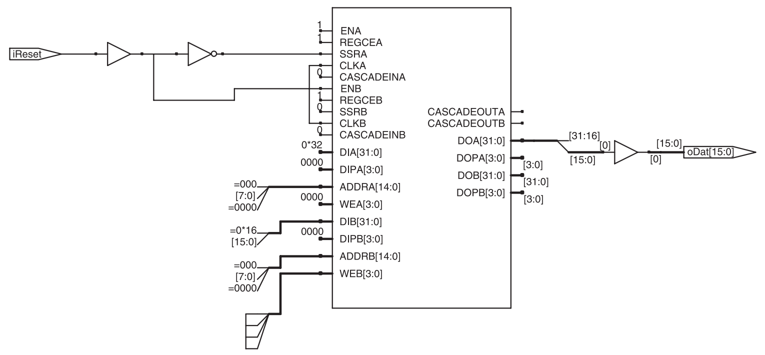

Figure 2.9 Xilinx BRAM with synchronous reset.

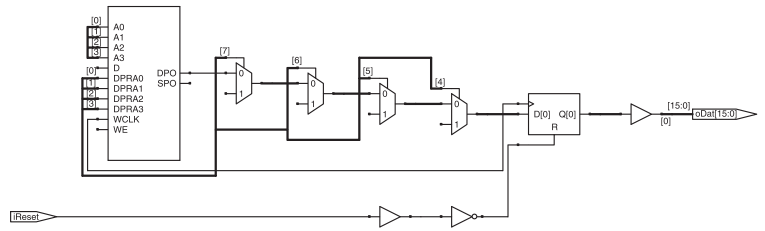

Figure 2.10 Xilinx BRAM with asynchronous reset logic.

Again, the only variation we will consider in the above code is the type of reset: synchronous versus asynchronous. In Xilinx Virtex-4 devices, for example, BRAM (Block RAM) elements have synchronous resets only. Therefore, with a synchronous reset, the synthesis tool will be able to implement this code with a single BRAM element as shown in Figure 2.9.

However, if we attempt to implement the same RAM with an asynchronous reset as shown in the code example above, the synthesis tool will be forced to create a RAM module with smaller distributed RAM blocks, additional decode logic to create the appropriate-size RAM, and additional logic to implement the asynchronous reset as partially shown in Figure 2.10. The final implementation differences are staggering as shown in Table 2.4.

Improperly resetting a RAM can have a catastrophic impact on the area.

Table 2.4 Resource Utilization for BRAM with Synchronous and Asynchronous Resets

Implementation	Slices slice	Flip-flops	4 Input LUTs	BRAMs
Asynchronous reset	3415	4112	2388	0
Synchronous reset	0	0	0	1

2.4.5 Utilizing Set/Reset Flip-Flop Pins

Most FPGA vendors have a variety of flip-flop elements available in any given device, and given a particular logic function, the synthesis tool can often use the set and reset pins to implement aspects of the logic and reduce the burden on the look-up tables. For instance, consider Figure 2.11. In this case, the synthesis tool may choose to implement the logic using the set pin on a flip-flop as shown in Figure 2.12. This eliminates gates and increases the speed of the data path. Likewise, consider a logic function of the form illustrated in Figure 2.13. The AND gate can be eliminated by running the input signal to the reset pin of the flip-flop as shown in Figure 2.14.

The primary reason synthesis tools are prevented from performing this class of optimizations is related to the reset strategy. Any constraints on the reset will not only use available set/reset pins but will also limit the number of library elements to choose from.

Using set and reset can prevent certain combinatorial logic optimizations.

For instance, consider the following implementation in a Xilinx Spartan-3 device:

```
module setreset(
 output reg oDat,
 input      iReset, iClk,
 input      iDat1, iDat2);

 always @(posedge iClk or negedge iReset)
  if(!iReset)
   oDat <= 0;
  else
   oDat <= iDat1 | iDat2;
endmodule
```

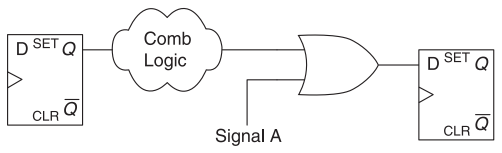

Figure 2.11 Simple synchronous logic with OR gate.

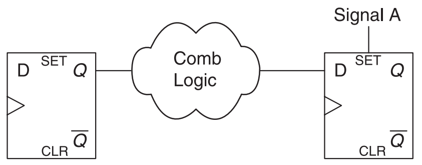

Figure 2.12 OR gate implemented with set pin.

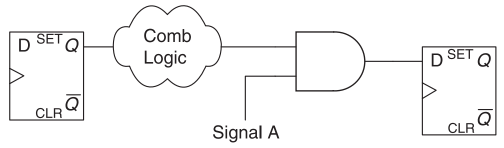

Figure 2.13 Simple synchronous logic with AND gate.

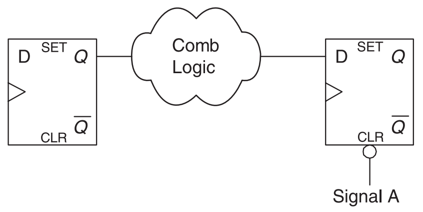

Figure 2.14 AND gate implemented with CLR pin.

In the code example above, an external reset signal is used to reset the state of the flip-flop. This is represented in Figure 2.15.

As can be seen in Figure 2.15, a resetable flip-flop was used for the asynchronous reset capability, and the logic function (OR gate) was implemented in discrete logic. As an alternative, if we remove the reset but implement the same logic function, our design will be optimized as shown in Figure 2.16.

In this implementation, the synthesis tool was able to use the FDS element (flip-flop with a synchronous set and reset) and use the set pin for the OR operation. Thus, by allowing the synthesis tool to choose a flip-flop with a synchronous set, we are able to implement this function with zero logic elements.

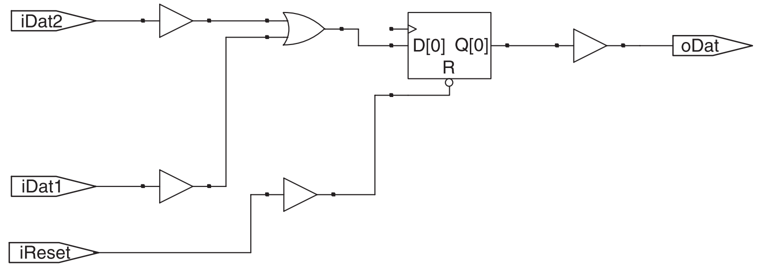

Figure 2.15 Simple asynchronous reset.

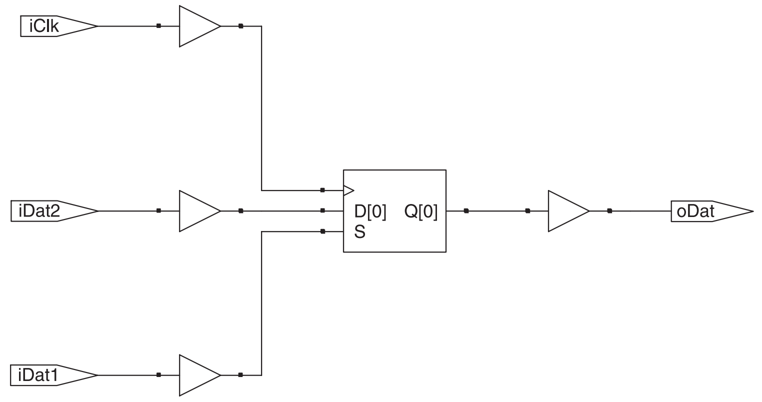

Figure 2.16 Optimization without reset.

We can take this one step further by using both synchronous set and reset signals. If we have a logic equation to evaluate in the form of

oDat <= !iDat3 & (iDat1 | iDat2)

we can code this in such a way that both the synchronous set and reset resources are used:

```
module setreset (
 output reg oDat,
 input iClk,
 input iDat1, iDat2, iDat3);

 always @(posedge iClk)
  if(iDat3)
  oDat <= 0;
  else if(iDat1)
   oDat <= 1;
  else
   oDat <= iDat2;

endmodule
```

Here, the iDat3 input takes priority similar to the reset pin on the associated flip-flops. Thus, this logic function can be implemented as shown in Figure 2.17.

In this circuit, we have three logical operations (invert, AND, and OR) all implemented with a single flip-flop and zero LUTs. Because these optimizations

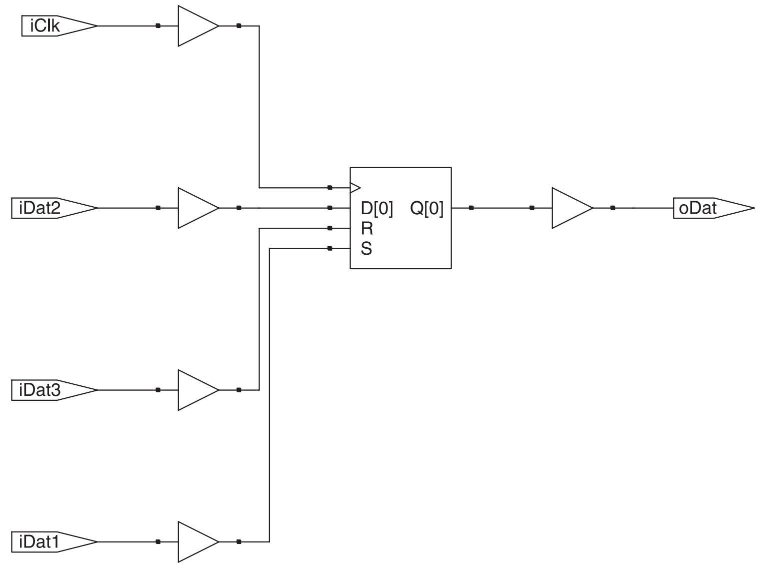

Figure 2.17 Optimization using both set and reset pins.

are not always known at the time the design is architected, avoid using set or reset whenever possible when area is the key consideration.

Avoid using set or reset whenever possible when area is the key consideration.

2.5 SUMMARY OF KEY POINTS

- Rolling up the pipeline can optimize the area of pipelined designs with duplicated logic in the pipeline stages.
- Controls can be used to direct the reuse of logic when the shared logic is larger than the control logic.
- For compact designs where area is the primary requirement, search for resources that have similar counterparts in other modules that can be brought to a global point in the hierarchy and shared between multiple functional areas.
- An improper reset strategy can create an unnecessarily large design and inhibit certain area optimizations.
- An optimized FPGA resource will not be used if an incompatible reset is assigned to it. The function will be implemented with generic elements and will occupy more area.

- DSPs and other multifunction resources are typically not flexible to varying reset strategies.
- Improperly resetting a RAM can have a catastrophic impact on the area.
- Using set and reset can prevent certain combinatorial logic optimizations.
- Avoid using set or reset whenever possible when area is the key consideration.