Chapter 1

Architecting Speed

Sophisticated tool optimizations are often not good enough to meet most design constraints if an arbitrary coding style is used. This chapter discusses the first of three primary physical characteristics of a digital design: speed. This chapter also discusses methods for architectural optimization in an FPGA.

There are three primary definitions of speed depending on the context of the problem: throughput, latency, and timing. In the context of processing data in an FPGA, throughput refers to the amount of data that is processed per clock cycle. A common metric for throughput is bits per second. Latency refers to the time between data input and processed data output. The typical metric for latency will be time or clock cycles. Timing refers to the logic delays between sequential elements. When we say a design does not "meet timing," we mean that the delay of the critical path, that is, the largest delay between flip-flops (composed of combinatorial delay, clk-to-out delay, routing delay, setup timing, clock skew, and so on) is greater than the target clock period. The standard metrics for timing are clock period and frequency.

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

- High-throughput architectures for maximizing the number of bits per second that can be processed by the design.
- Low-latency architectures for minimizing the delay from the input of a module to the output.
- Timing optimizations to reduce the combinatorial delay of the critical path.
  - Adding register layers to divide combinatorial logic structures.
  - Parallel structures for separating sequentially executed operations into parallel operations.
  - Flattening logic structures specific to priority encoded signals.
  - Register balancing to redistribute combinatorial logic around pipelined registers.
  - Reordering paths to divert operations in a critical path to a noncritical path.

Advanced FPGA Design. By Steve Kilts

1.1 HIGH THROUGHPUT

A high-throughput design is one that is concerned with the steady-state data rate but less concerned about the time any specific piece of data requires to propagate through the design (latency). The idea with a high-throughput design is the same idea Ford came up with to manufacture automobiles in great quantities: an assembly line. In the world of digital design where data is processed, we refer to this under a more abstract term: pipeline.

A pipelined design conceptually works very similar to an assembly line in that the raw material or data input enters the front end, is passed through various stages of manipulation and processing, and then exits as a finished product or data output. The beauty of a pipelined design is that new data can begin processing before the prior data has finished, much like cars are processed on an assembly line. Pipelines are used in nearly all very-high-performance devices, and the variety of specific architectures is unlimited. Examples include CPU instruction sets, network protocol stacks, encryption engines, and so on.

From an algorithmic perspective, an important concept in a pipelined design is that of "unrolling the loop." As an example, consider the following piece of code that would most likely be used in a software implementation for finding the third power of X. Note that the term "software" here refers to code that is targeted at a set of procedural instructions that will be executed on a microprocessor.

```
XPower = 1;
for (i=0;i < 3; i++)
  XPower = X * XPower;
```

Note that the above code is an iterative algorithm. The same variables and addresses are accessed until the computation is complete. There is no use for parallelism because a microprocessor only executes one instruction at a time (for the purpose of argument, just consider a single core processor). A similar implementation can be created in hardware. Consider the following Verilog implementation of the same algorithm (output scaling not considered):

```
module power3(
  output [7:0] XPower,
  output       finished,
  input  [7:0] X,
  input        clk, start); // the duration of start is a
                               single clock
  reg    [7:0] ncount;
  reg    [7:0] XPower;
  assign finished = (ncount == 0);
  always@(posedge clk)
    if(start) begin
      XPower <= X;
      ncount <= 2;
    end
```

```
    else if(!finished) begin
      ncount <= ncount - 1;
      XPower <= XPower * X;
    end
endmodule
```

In the above example, the same register and computational resources are reused until the computation is finished as shown in Figure 1.1.

With this type of iterative implementation, no new computations can begin until the previous computation has completed. This iterative scheme is very similar to a software implementation. Also note that certain handshaking signals are required to indicate the beginning and completion of a computation. An external module must also use the handshaking to pass new data to the module and receive a completed calculation. The performance of this implementation is

Throughput = 8/3, or 2.7 bits/clock

Latency = 3 clocks

Timing = One multiplier delay in the critical path

Contrast this with a pipelined version of the same algorithm:

```
module power3(
  output reg [7:0] XPower,
  input            clk,
  input      [7:0] X
  );
  reg        [7:0] XPower1, XPower2;
  reg        [7:0] X1, X2;
  always @(posedge clk) begin
    // Pipeline stage 1
    X1      <= X;
    XPower1 <= X;
    // Pipeline stage 2
    X2      <= X1;
    XPower2 <= XPower1 * X1;
    // Pipeline stage 3
    XPower <= XPower2 * X2;
  end
endmodule
```

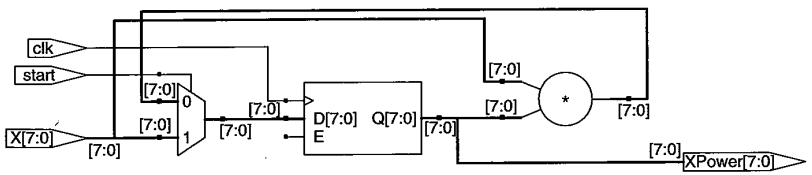

Figure 1.1 Iterative implementation

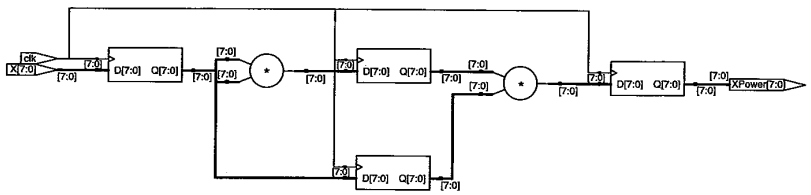

Figure 1.2 Pipelined implementation.

In the above implementation, the value of X is passed to both pipeline stages where independent resources compute the corresponding multiply operation. Note that while X is being used to calculate the final power of 3 in the second pipeline stage, the next value of X can be sent to the first pipeline stage as shown in Figure 1.2.

Both the final calculation of X^3 (XPower3 resources) and the first calculation of the next value of X (XPower2 resources) occur simultaneously. The performance of this design is

Throughput = 8/1, or 8 bits/clock

Latency = 3 clocks

Timing = One multiplier delay in the critical path

The throughput performance increased by a factor of 3 over the iterative implementation. In general, if an algorithm requiring n iterative loops is "unrolled," the pipelined implementation will exhibit a throughput performance increase of a factor of n. There was no penalty in terms of latency as the pipelined implementation still required 3 clocks to propagate the final computation. Likewise, there was no timing penalty as the critical path still contained only one multiplier.

Unrolling an iterative loop increases throughput.

The penalty to pay for unrolling loops such as this is an increase in area. The iterative implementation required a single register and multiplier (along with some control logic not shown in the diagram), whereas the pipelined implementation required a separate register for both X and XPower and a separate multiplier for every pipeline stage. Optimizations for area are discussed in the Chapter 2.

The penalty for unrolling an iterative loop is a proportional increase in area.

1.2 LOW LATENCY

A low-latency design is one that passes the data from the input to the output as quickly as possible by minimizing the intermediate processing delays. Oftentimes, a low-latency design will require parallelisms, removal of pipelining, and logical short cuts that may reduce the throughput or the max clock speed in a design.

Referring back to our power-of-3 example, there is no obvious latency optimization to be made to the iterative implementation as each successive multiply operation must be registered for the next operation. The pipelined implementation, however, has a clear path to reducing latency. Note that at each pipeline stage, the product of each multiply must wait until the next clock edge before it is propagated to the next stage. By removing the pipeline registers, we can minimize the input to output timing:

```
module power3(
  output [7:0] XPower,
  input  [7:0] X
  );
  reg    [7:0] XPower1, XPower2;
  reg    [7:0] X1, X2;

  assign XPower = XPower2 * X2;

  always @* begin
    X1      = X;
    XPower1 = X;
  end

  always @* begin
    X2      = X1;
    XPower2 = XPower1*X1;
  end
endmodule
```

In the above example, the registers were stripped out of the pipeline. Each stage is a combinatorial expression of the previous as shown in Figure 1.3.

The performance of this design is

Throughput = 8 bits/clock (assuming one new input per clock)

Latency = Between one and two multiplier delays, 0 clocks

Timing = Two multiplier delays in the critical path

By removing the pipeline registers, we have reduced the latency of this design below a single clock cycle.

Latency can be reduced by removing pipeline registers.

The penalty is clearly in the timing. Previous implementations could theoretically run the system clock period close to the delay of a single multiplier, but in the

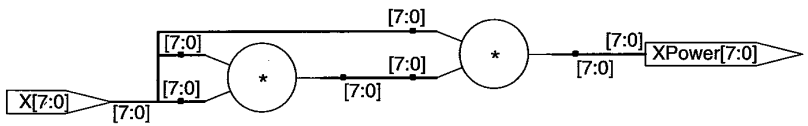

Figure 1.3 Low-latency implementation.

low-latency implementation, the clock period must be at least two multiplier delays (depending on the implementation) plus any external logic in the critical path.

The penalty for removing pipeline registers is an increase in combinatorial delay between registers.

1.3 TIMING

Timing refers to the clock speed of a design. The maximum delay between any two sequential elements in a design will determine the max clock speed. The idea of clock speed exists on a lower level of abstraction than the speed/area trade-offs discussed elsewhere in this chapter as clock speed in general is not directly related to these topologies, although trade-offs within these architectures will certainly have an impact on timing. For example, one cannot know whether a pipelined topology will run faster than an iterative without knowing the details of the implementation. The maximum speed, or maximum frequency, can be defined according to the straightforward and well-known maximum-frequency equation (ignoring clock-to-clock jitter):

Equation 1.1 Maximum Frequency

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}} \quad (1.1)$$

where F_{max} is maximum allowable frequency for clock; $T_{clk\text{-}q}$ is time from clock arrival until data arrives at Q; T_{logic} is propagation delay through logic between flip-flops; $T_{routing}$ is routing delay between flip-flops; T_{setup} is minimum time data must arrive at D before the next rising edge of clock (setup time); and T_{skew} is propagation delay of clock between the launch flip-flop and the capture flip-flop.

The next sections describes various methods and trade-offs required to improve timing performance.

1.3.1 Add Register Layers

The first strategy for architectural timing improvements is to add intermediate layers of registers to the critical path. This technique should be used in highly pipelined designs where an additional clock cycle latency does not violate the design specifications, and the overall functionality will not be affected by the further addition of registers.

For instance, assume the architecture for the following FIR (Finite Impulse Response) implementation does not meet timing:

```
module fir(
  output [7:0] Y,
  input  [7:0] A, B, C, X,
  input        clk,
```

```
  input        validsample);
  reg    [7:0] X1, X2, Y;

  always @(posedge clk)
    if(validsample) begin
      X1 <= X;
      X2 <= X1;
      Y <= A* X+B* X1+C* X2;
    end
endmodule
```

Architecturally, all multiply/add operations occur in one clock cycle as shown in Figure 1.4.

In other words, the critical path of one multiplier and one adder is greater than the minimum clock period requirement. Assuming the latency requirement is not fixed at 1 clock, we can further pipeline this design by adding extra registers intermediate to the multipliers. The first layer is easy: just add a pipeline layer between the multipliers and the adder:

```
module fir(
  output [7:0] Y,
  input  [7:0] A, B, C, X,
  input        clk,
  input        validsample);
  reg    [7:0] X1, X2, Y;
  reg    [7:0] prod1, prod2, prod3;

  always @ (posedge clk) begin
    if(validsample) begin
      X1    <= X;
      X2    <= X1;
      prod1 <= A * X;
      prod2 <= B * X1;
      prod3 <= C * X2;
    end
    Y <= prod1 + prod2 + prod3;
  end
endmodule
```

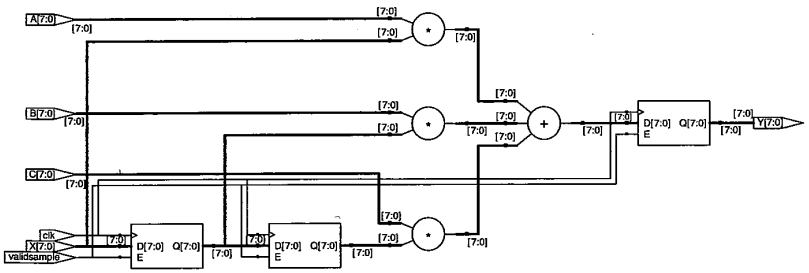

Figure 1.4 MAC with long path

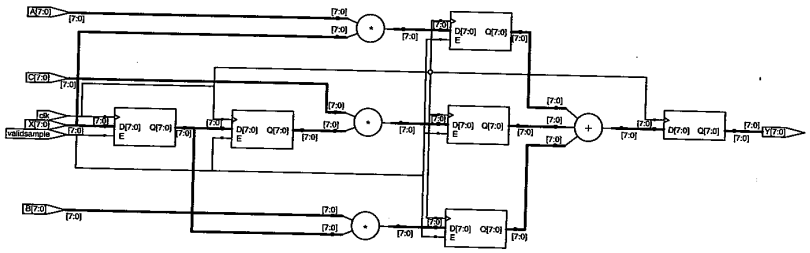

Figure 1.5 Pipeline registers added.

In the above example, the adder was separated from the multipliers with a pipeline stage as shown in Figure 1.5.

Multipliers are good candidates for pipelining because the calculations can easily be broken up into stages. Additional pipelining is possible by breaking the multipliers and adders up into stages that can be individually registered.

> Adding register layers improves timing by dividing the critical path into two paths of smaller delay.

Various implementations of these functions are covered in other chapters, but once the architecture has been broken up into stages, additional pipelining is as straightforward as the above example.

1.3.2 Parallel Structures

The second strategy for architectural timing improvements is to reorganize the critical path such that logic structures are implemented in parallel. This technique should be used whenever a function that currently evaluates through a serial string of logic can be broken up and evaluated in parallel. For instance, assume that the standard pipelined power-of-3 design discussed in previous sections does not meet timing. To create parallel structures, we can break the multipliers into independent operations and then recombine them. For instance, an 8-bit binary multiplier can be represented by nibbles A and B:

$$X = \{A, B\},$$

where A is the most significant nibble and B is the least significant.

Because the multiplicand is equal to the multiplier in our power-of-3 example, the multiply operation can be reorganized as follows:

$$X * X = \{A, B\} * \{A, B\} = \{(A * A), (2 * A * B), (B * B)\};$$

This reduces our problem to a series of 4-bit multiplications and then recombining the products. This can be implemented with the following module:

```
module power3(
  output [7:0] XPower,
```

```
input  [7:0] X,
input        clk);
reg    [7:0] XPower1;
// partial product registers
reg    [3:0] XPower2_ppAA, XPower2_ppAB, XPower2_ppBB;
reg    [3:0] XPower3_ppAA, XPower3_ppAB, XPower3_ppBB;
reg    [7:0] X1, X2;
wire   [7:0] XPower2;

// nibbles for partial products (A is MS nibble, B is LS
                                 nibble)
wire   [3:0] XPower1_A = XPower1[7:4];
wire   [3:0] XPower1_B = XPower1[3:0];
wire   [3:0] X1_A      = X1[7:4];
wire   [3:0] X1_B      = X1[3:0];
wire   [3:0] XPower2_A = XPower2[7:4];
wire   [3:0] XPower2_B = XPower2[3:0];
wire   [3:0] X2_A      = X2[7:4];
wire   [3:0] X2_B      = X2[3:0];

// assemble partial products
assign XPower2         = (XPower2_ppAA << 8)+
                         (2*XPower2_ppAB << 4)+
                          XPower2_ppBB;
assign XPower          = (XPower3_ppAA << 8)+
                         (2*XPower3_ppAB << 4)+
                         XPower3_ppBB;

always @(posedge clk) begin
  // Pipeline stage 1
  X1           <= X;
  XPower1      <= X;

  // Pipeline stage 2
  X2           <= X1;
  // create partial products
  XPower2_ppAA <= XPower1_A * X1_A;
  XPower2_ppAB <= XPower1_A * X1_B;
  XPower2_ppBB <= XPower1_B * X1_B;

  // Pipeline stage 3
  // create partial products
  XPower3_ppAA <= XPower2_A * X2_A;
  XPower3_ppAB <= XPower2_A * X2_B;
  XPower3_ppBB <= XPower2_B * X2_B;
end
endmodule
```

This design does not take into consideration any overflow issues, but it serves to illustrate the point. The multiplier was broken down into smaller functions that could be operated on independently as shown in Figure 1.6.

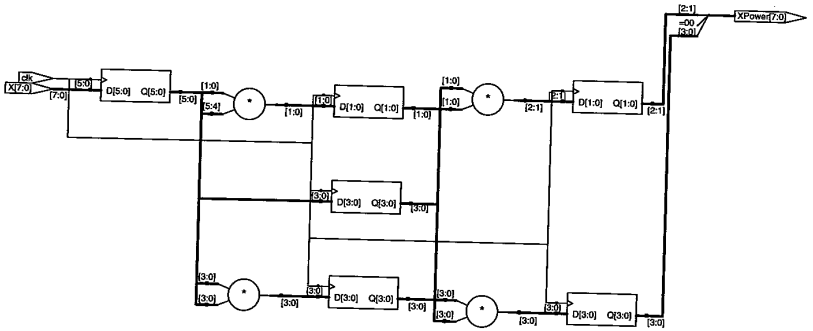

Figure 1.6 Multiplier with separated stages.

By breaking the multiply operation down into smaller operations that can execute in parallel, the maximum delay is reduced to the longest delay through any of the substructures.

> Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the path delay to the longest of the substuctures.

1.3.3 Flatten Logic Structures

The third strategy for architectural timing improvements is to flatten logic structures. This is closely related to the idea of parallel structures defined in the previous section but applies specifically to logic that is chained due to priority encoding. Typically, synthesis and layout tools are smart enough to duplicate logic to reduce fanout, but they are not smart enough to break up logic structures that are coded in a serial fashion, nor do they have enough information relating to the priority requirements of the design. For instance, consider the following control signals coming from an address decode that are used to write four registers:

```
module regwrite(
  output reg [3:0] rout,
  input            clk, in,
  input      [3:0] ctrl);

  always @(posedge clk)
    if(ctrl[0])      rout[0] <= in;
    else if(ctrl[1]) rout[1] <= in;
    else if(ctrl[2]) rout[2] <= in;
    else if(ctrl[3]) rout[3] <= in;
endmodule
```

In the above example, each of the control signals are coded with a priority relative to the other control signals. This type of priority encoding is implemented as shown in Figure 1.7.

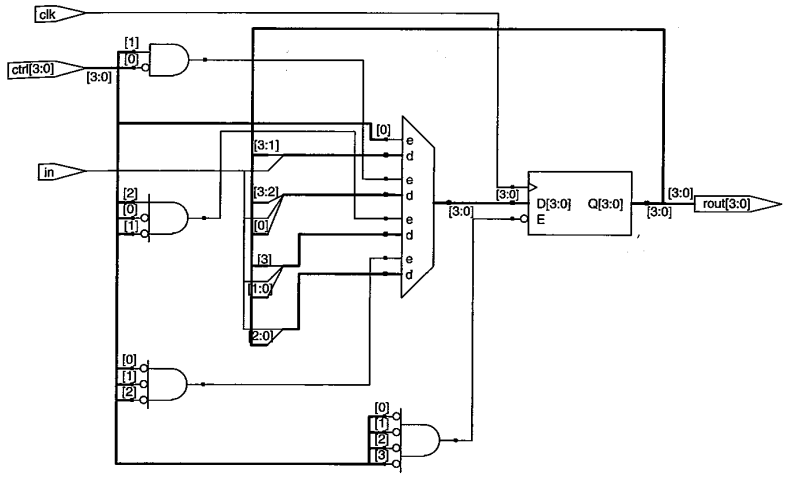

Figure 1.7 Priority encoding.

If the control lines are strobes from an address decoder in another module, then each strobe is mutually exclusive to the others as they all represent a unique address. However, here we have coded this as if it were a priority decision. Due to the nature of the control signals, the above code will operate exactly as if it were coded in a parallel fashion, but it is unlikely the synthesis tool will be smart enough to recognize that, particularly if the address decode takes place behind another layer of registers.

To remove the priority and thereby flatten the logic, we can code this module as shown below:

```
module regwrite(
  output reg [3:0] rout,
  input            clk, in,
  input      [3:0] ctrl);

  always @(posedge clk) begin
    if(ctrl[0]) rout[0] <= in;
    if(ctrl[1]) rout[1] <= in;
    if(ctrl[2]) rout[2] <= in;
    if(ctrl[3]) rout[3] <= in;
  end
endmodule
```

As can be seen in the gate-level implementation, no priority logic is used as shown in Figure 1.8. Each of the control signals acts independently and controls its corresponding rout bits independently.

By removing priority encodings where they are not needed, the logic structure is flattened and the path delay is reduced.

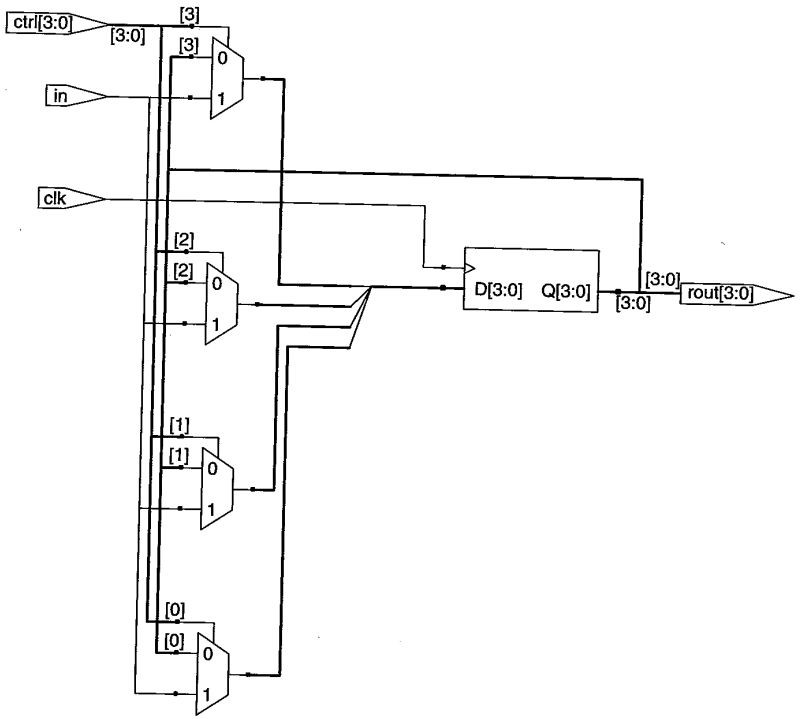

Figure 1.8 No priority encoding.

1.3.4 Register Balancing

The fourth strategy is called register balancing. Conceptually, the idea is to redistribute logic evenly between registers to minimize the worst-case delay between any two registers. This technique should be used whenever logic is highly imbalanced between the critical path and an adjacent path. Because the clock speed is limited by only the worst-case path, it may only take one small change to successfully rebalance the critical logic.

Many synthesis tools also have an optimization called register balancing. This feature will essentially recognize specific structures and reposition registers around logic in a predetermined fashion. This can be useful for common structures such as large multipliers but is limited and will not change your logic nor recognize custom functionality. Depending on the technology, it may require more expensive synthesis tools to implement. Thus, it is very important to understand this concept and have the ability to redistribute logic in custom logic structures.

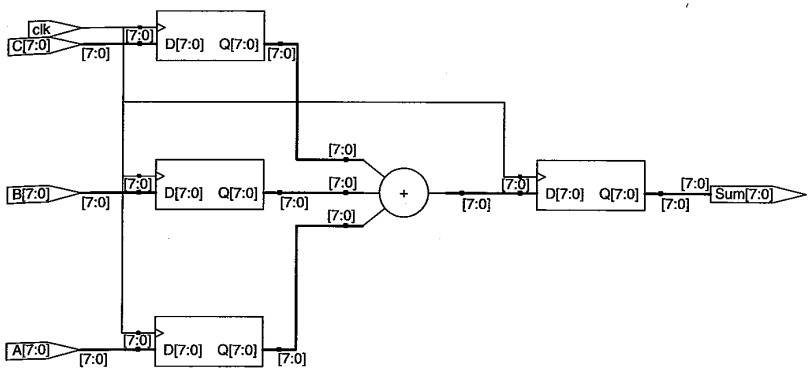

Figure 1.9 Registered adder.

Note the following code for an adder that adds three 8-bit inputs:

```
module adder(
  output reg [7:0] Sum,
  input      [7:0] A, B, C,
  input            clk);
  reg        [7:0] rA, rB, rC;

  always @(posedge clk) begin
    rA   <= A;
    rB   <= B;
    rC   <= C;
    Sum  <= rA + rB + rC;
  end
endmodule
```

The first register stage consists of rA, rB, and rC, and the second stage consists of Sum. The logic between stages 1 and 2 is the adder for all inputs, whereas the logic between the input and the first register stage contains no logic (assume the outputs feeding this module are registered) as shown in Figure 1.9.

If the critical path is defined through the adder, some of the logic in the critical path can be moved back a stage, thereby balancing the logic load between the two register stages. Consider the following modification where one of the add operations is moved back a stage:

```
module adder(
  output reg [7:0] Sum,
  input      [7:0] A, B, C,
  input            clk);
  reg        [7:0] rABSum, rC;
```



```
  always @(posedge clk) begin
    rABSum <= A + B;
    rC     <= C;
    Sum    <= rABSum + rC;
  end
endmodule
```

We have now moved one of the add operations back one stage between the input and the first register stage. This balances the logic between the pipeline stages and reduces the critical path as shown in Figure 1.10.

> Register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.

1.3.5 Reorder Paths

The fifth strategy is to reorder the paths in the data flow to minimize the critical path. This technique should be used whenever multiple paths combine with the critical path, and the combined path can be reordered such that the critical path can be moved closer to the destination register. With this strategy, we will only be concerned with the logic paths between any given set of registers. Consider the following module:

```
module randomlogic(
  output reg [7:0] Out,
  input      [7:0] A, B, C,
  input            clk,
  input            Cond1, Cond2);
  always @(posedge clk)
    if(Cond1)
    Out <= A;
  else if(Cond2 && (C < 8))
    Out <= B;
  else
    Out <= C;
endmodule
```

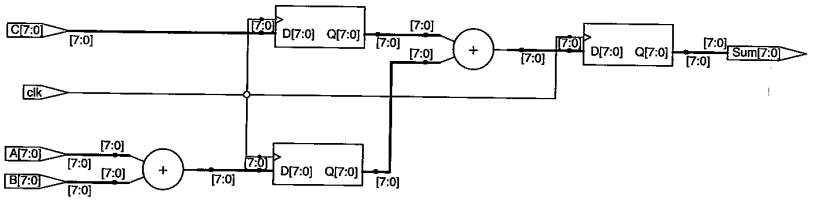

Figure 1.10 Registers balanced.

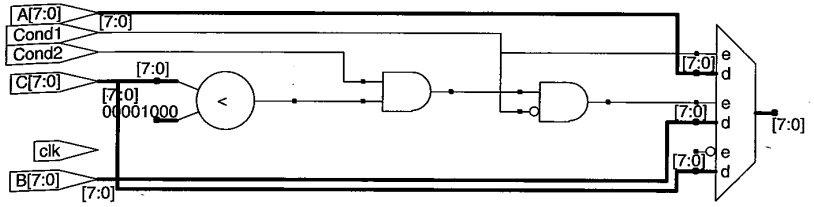

Figure 1.11 Long critical path.

In this case, let us assume the critical path is between C and Out and consists of a comparator in series with two gates before reaching the decision mux. This is shown in Figure 1.11. Assuming the conditions are not mutually exclusive, we can modify the code to reorder the long delay of the comparitor:

```
module randomlogic(
  output reg [7:0] Out,
  input      [7:0] A, B, C,
  input            clk,
  input            Cond1, Cond2);
  wire CondB = (Cond2 & !Cond1);

  always @(posedge clk)
    if(CondB && (C < 8))
      Out <= B;
    else if(Cond1)
      Out <= A;
    else
      Out <= C;
endmodule
```

By reorganizing the code, we have moved one of the gates out of the critical path in series with the comparator as shown in Figure 1.12. Thus, by paying careful

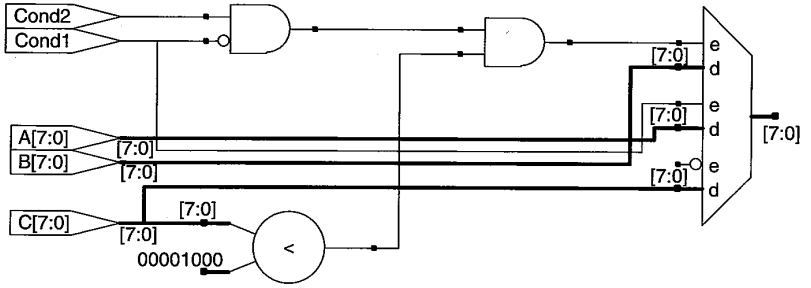

Figure 1.12 Logic reordered to reduce critical path.

attention to exactly how a particular function is coded, we can have a direct impact on timing performance.

> Timing can be improved by reordering paths that are combined with the critical path in such a way that some of the critical path logic is placed closer to the destination register.

1.4 SUMMARY OF KEY POINTS

- A high-throughput architecture is one that maximizes the number of bits per second that can be processed by a design.
- Unrolling an iterative loop increases throughput.
- The penalty for unrolling an iterative loop is a proportional increase in area.
- A low-latency architecture is one that minimizes the delay from the input of a module to the output.
- Latency can be reduced by removing pipeline registers.
- The penalty for removing pipeline registers is an increase in combinatorial delay between registers.
- Timing refers to the clock speed of a design. A design meets timing when the maximum delay between any two sequential elements is smaller than the minimum clock period.
- Adding register layers improves timing by dividing the critical path into two paths of smaller delay.
- Separating a logic function into a number of smaller functions that can be evaluated in parallel reduces the path delay to the longest of the substructures.
- By removing priority encodings where they are not needed, the logic structure is flattened, and the path delay is reduced.
- Register balancing improves timing by moving combinatorial logic from the critical path to an adjacent path.
- Timing can be improved by reordering paths that are combined with the critical path in such a way that some of the critical path logic is placed closer to the destination register.