```
`define COMPARE_VALUE 7'd98  // N-2 due to pipeline delay

// counter logic with pipelined comparison logic

always @(posedge Clk)
begin
  if (ClrCount) begin
    CountDone  <= 1'b0;
    Count[6:0] <= 8'h1;
  end
  else if (IncCount) begin
    CountDone  <= Count[6:0] == `COMPARE_VALUE;
    Count[6:0] <= Count[6:0] + 1;
  end
end

// Partial FSM logic

...
    `WAIT_COUNT :
       if (Event && CountDone) // N-1 CountDone flag qualified with
         NextState[3:0] = `FINISH;  // Event, indicating Nth event
       else if (Event)
         IncCount = 1'b1;
         ...
       end
...
```

FIGURE 10.21 Pipelined count comparison logic.

```
    `COMPLEX_STATE :
       if ( (BytesFound[15:0] + BytesOffset[7:0]) >
                  BytesThreshold[15:0] )
         NextState[3:0] = `MORE_MATH;
         ...
       else
         ...
       end
...
```

FIGURE 10.22 Arithmetic expression in FSM branch condition.

process. Despite this complexity, there are situations in which this is the only way of obtaining the desired performance.

Pipelining is perhaps most tricky when the logic involved contains a feedback path through the FSM. This feedback path can be of the type discussed previously whereby the FSM is detecting an event, incrementing a counter, and then relying on that count value to take further action. There are other situations in which no feedback path exists. An FSM may be called on to process an incoming data stream over which it has no direct control. As in the earlier pattern matching example, the FSM may need to look for certain data values and take action if they are located. In Fig. 10.17, the FSM directly evaluated the eight-bit data stream in searching for the values 0x01, 0x02, 0x03, and 0x04. This logic can be optimized by inserting pattern-matching flops between the data input port and the FSM as shown in Fig. 10.23. In doing so, the FSM no longer needs to evaluate the entire eight-bit vector but instead can test a single flop for each pattern.

```
// pipelined pattern matching flops

always @(posedge Clk)
begin
  if (!Reset_) begin
    Data01 <= 1´b0;
    Data02 <= 1´b0;
    Data03 <= 1´b0;
    Data04 <= 1´b0;
  end
  else begin
    Data01 <= Data[7:0] == 8´h01;
    Data02 <= Data[7:0] == 8´h02;
    Data03 <= Data[7:0] == 8´h03;
    Data04 <= Data[7:0] == 8´h04;
  end
end

always @(State[2:0] or Trigger or Data01 or Data02 or Data03 or Data04)
begin
  NextState[2:0] = State[2:0];
  NextMatch      = 1´b0;

  case (State[2:0])

    `IDLE :
       if (Trigger && Data01) // Instead of Data[7:0] == 8´h01
         NextState[2:0] = `WAIT02;
...
```

FIGURE 10.23 Pattern-matching logic with pipelining.

When pipeline flops are inserted into a data path, care must be taken to apply a consistent delay to the entire data path. This example does not use the data path for any purpose other than pattern matching; therefore, there is no need to perform further operations on the data once it is checked for the relevant patterns. If, however, the logic performed pattern matching and then manipulated the data based on that matching (e.g., recognize the start of a data packet and then store the packet into a memory), it would be critical to keep the pipelined pattern-matching flops coincident with the data that they represent. Failure to do so would result in detecting the pattern late with respect to the data stream, thereby missing the packet's initial data. The data path can be delayed along with the pipelined logic by passing it through a register also. If the previous example did process the data, logic reading `DataPipe[7:0] <= Data[7:0]` could be placed into the same always block as the pattern matching logic. Subsequent references to Data[7:0] would be replaced by references to DataPipe[7:0]. This way, when Data[7:0] equals 0x01, Data01 will be set on the next cycle, and DataPipe[7:0] will be loaded with the value 0x01 on the next cycle as well.