times, the data bus is held in a high-impedance state. This works as expected, because the value Z can be overridden by another assignment. In simulation, the other assignment may come from a test bench that emulates the CPU's operation. In synthesis, the software properly recognizes this arrangement as inferring a tri-state bus. The continuous assignment takes advantage of Verilog's conditional operator, ? : , which serves an if…else function. When the logical expression before the question mark is true, the value before the colon is used. Otherwise, the value after the colon is used. A bidirectional port is declared using the Verilog *inout* keyword in place of *input* or *output*. The synchronous version of the read logic is very similar to the asynchronous version, except that the outputs are first registered before being used in the tri-state assignment.

Interrupt control registers are implemented by support logic when the number of total interrupt sources in the system exceeds the CPU's interrupt handling capacity. Gathering multiple interrupts and presenting them to the CPU as a single interrupt signal can be as simple as logically ORing multiple interrupt signals together. A somewhat more complex scheme, but one with value, is where interrupts can be selectively masked by the CPU, and system-wide interrupt status is accumulated in a single register. This gives the CPU more control over how it gets interrupted and, when interrupted, provides a single register that can be read to determine the source of the interrupt. Such a scheme can be implemented with two registers and associated logic: a read/write interrupt mask register and a read-only interrupt status register. Each pair of bits in the mask and status registers corresponds to a single interrupt source. When an interrupt is active, the corresponding bit in the status register is active. However, only those interrupt sources whose mask bits have been cleared will result in a CPU interrupt. At reset, the mask register defaults to 0xFF to disable all interrupts. Figure 10.9 shows a Verilog implementation of interrupt control as an extension of the previous example. Synchronous registers are assumed here, but asynchronous logic is easily adapted.

Aside from the registers themselves, the main function of the interrupt control logic is implemented by the bit-wise ORing and reduction AND of the interrupt mask bits and the external interrupt signals. Verilog's | operator is a bit-wise OR function in contrast to the || logical OR function. When two equal-size vectors are bit-wise ORed, the result is a single vector of the same size wherein each bit is the OR of the corresponding pair of bits of the operands. This first step disables any active interrupts that are masked; per DeMorgan's law, an OR acts as an active-low AND function. The second step is a reduction AND as indicated by the unary & operator. Similar to OR, & is the bit-wise version of &&. Invoking & with a single operand makes it a reduction operator that ANDs together all bits of a vector and generates a single output bit. AND is used because the interrupt polarities are active-low. Therefore, if any one interrupt is asserted (low), the reduction AND function will generate a low output. Per DeMorgan's law once again, an AND acts as an active-low OR function.

Timers are often useful structures that can periodically interrupt the CPU to invoke a time-critical interrupt service routine, enable the CPU to determine elapsed time, or trigger some other event in hardware. It is common to find timers implemented in certain CPU products such as microcontrollers. Sometimes, however, it becomes necessary to implement a custom timer in support logic. One such example is presented here with a fixed prescaler and an eight-bit counter with an eight-bit configurable terminal count value. The prescaler is used to slow down the timer so that it can interrupt the CPU over longer periods of time. For the sake of discussion, let's assume that the CPU is running at a frequency of 10 MHz and that the timer granularity should be 1 ms. Therefore, the prescaler should count from 0 to 9,999 to generate a 1-ms tick when running with a 100-ns period. A Verilog implementation of such a timer is shown in Fig. 10.10 without the associated read/write logic already presented in detail. A 14-bit prescaler is necessary to represent numbers from 0 to 9,999. It is assumed that the terminal count register, TermCount, is implemented elsewhere as a general read/write register.

When the timer rolls over, it asserts TimerRollOver for one CpuClk cycle to trigger whatever logic is desired by the application. If the trigger event is a CPU interrupt, the logic should create a

```
always @(Addr[3:0] or StatusInput[7:0] or ControlReg[7:0] or IntSel
                  or ISR[7:0] or IMR[7:0])
begin
  case (Addr[3:0]) // read multiplexer
    4´h0    : ReadData[7:0] = StatusInput[7:0]; // external input pins
    4´h1    : ReadData[7:0] = ControlReg[7:0];
    4´h2    : ReadData[7:0] = ISR[7:0];  // interrupt status
    4´h3    : ReadData[7:0] = IMR[7:0];  // interrupt mask
    default : ReadData[7:0] = 8´h0; // alternate means to prevent latch
  endcase

  ControlRegSel  = 1´b0;  // default inactive value
  IMRSel         = 1´b0;

  case (Addr[3:0]) // select signal only needed for writeable registers
    4´h1 : ControlRegSel = IntSel;
    4´h3 : IMRSel        = IntSel;
  endcase
end

// Interrupt gathering

always @(IMR[7:0] or ExtInt_[7:0])
begin
  ISR[7:0] = ExtInt_[7:0]; // reflect status of external signals
  CpuInt_  = &(IMR[7:0] | ExtInt_[7:0]); // reduction AND
end

// Write logic

always @(posedge CpuClk)
begin
  if (!Reset_)
    IMR[7:0] <= 8´hff; // mask all interrupts on reset
  else if (IMRSel && !Wr_)
    IMR[7:0] <= CpuData[7:0];
end
```

FIGURE 10.9 Interrupt control logic.

“sticky” version of TimerRollOver that does not automatically get cleared by hardware. Rather, it is sticky because, once set by hardware, the bit will retain its state until explicitly cleared by software. This provides an arbitrarily long time for software to respond to the interrupt assertion, read the interrupt status register to detect the timer roll-over event, and then clear the sticky bit. Without the sticky bit, software would have no chance of catching a pulse that is only a single cycle in width.

10.3 CLOCK DOMAIN CROSSING

Some logic design tasks involve exchanging information between logic running on unrelated clocks. When multiple independent clock domains exist in a system, there is no guaranteed skew or phase relationship between the various clocks. Synchronous timing analysis dictates that a flop’s setup and hold times must be met to ensure reliable capture of the data presented to it. Yet it is impossible to guarantee proper setup and hold times when the source flop’s clock has no relationship to the destination flop’s clock.