```
always @(CS1_ or Rd_)
begin
  if (!CS1_ && !Rd_)
    DataBufDir = 1'b0;  // drive CPU bus when ROM selected for read
  else
    DataBufDir = 1'b1;  // otherwise, always drive data to ROM
end
```

**FIGURE 10.7** Data buffer control logic.

on an alarm. The opening door can be detected using a switch connected to an input signal. When the CPU reads the status of this signal, it can determine whether the switch is open or closed. An alarm can be turned on when the CPU sets an output signal that enables an alarm circuit. Control and status registers must be implemented to enable the CPU to read and write I/O signals. In our continuing example, we assume an eight-bit data bus coming from the CPU and the need for eight input signals and eight output signals. Implementing registers varies according to whether the CPU bus is synchronous or asynchronous. Some older microprocessors use asynchronous buses requiring latches to be formed within the support logic. Figure 10.8 shows the implementation of two registers using the previously decoded IntSel signal in both synchronous and asynchronous styles. Again, the proper declarations for ports and variables are assumed.

An added level of address decoding is required here to ensure that the two registers are not accessed simultaneously. The register logic consists of two basic sections: the write logic and read logic. The write logic (required only for the control register that drives output signals) transfers the contents of the CPU data bus to the internal register when the register is addressed and the write enable is active. The ControlRegSel signal is implemented in a case statement but can be implemented in a variety of ways. More select signals will be added in coming examples. The asynchronous write logic infers a latch, because not all permutations of input qualifiers are represented by assignments. If Reset_ is high and the control register is not being selected for a write, there is no specified action. Therefore, memory is implied and, in the absence of a causal clock, a latch is inferred. The synchronous write logic is almost identical, but it references a clock that causes a flop inference. Reset is implemented to provide a known initial state. This is a good idea so that external logic that is driven by the control register can be safely designed with the assumption that operations begin at a known state. The known state is usually inactive so that peripherals do not start operating before the CPU finishes booting and can disable them.

The read logic consists of two sections: the output multiplexer and the output buffer control. The output multiplexer simply selects one of the available registers for reading. It is not necessary to qualify the multiplexer with any other logic, because a read will not actually take place unless the output buffer control logic sends the data to the CPU. Rather than preventing a latch in ReadData by assigning it a default value before the case construct, the Verilog keyword *default* is used as the final case enumeration to specify default operation. Either solution will work—it is a matter of preference and style over which to use in a given situation. Both read-only and writable registers are included in the read multiplexer logic. Strictly speaking, it is not mandatory to have writable register contents readable by the CPU, but this is a very good practice. Years ago, when logic was very expensive, it was not uncommon to find write-only registers. However, there is a substantial drawback to this approach: you can never be sure what the contents of the register are if you fail to keep track of the exact data that has already been written!

Implementing bidirectional signals in Verilog can be done with a continuous assignment that selects between driving an active variable or a high-impedance value, Z. The asynchronous read logic is very simple: whenever the internal registers are selected and read enable is active, the tri-state buffer is enabled, and the output of the multiplexer is driven onto the CPU data bus. At all other

```
always @(Addr[3:0] or StatusInput[7:0] or ControlReg[7:0] or IntSel)
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];
    default : ReadData[7:0] = 8´h0; // alternate means to prevent latch
  endcase

  ControlRegSel = 1´b0;  // default inactive value

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

// Option #1A: asynchronous read logic

assign CpuData[7:0] = (IntSel && !Rd_) ? ReadData[7:0] : 8´bz;

// Option #1B: synchronous read logic

always @(posedge CpuClk)
begin
  if (!Reset_)  // synchronous reset
    CpuDataOE <= 1´b0;
    // no need to reset ReadDataReg and possibly save some logic
  else begin
    CpuDataOE        <= IntSel && !Rd_; // all outputs are registered
    ReadDataReg[7:0] <= ReadData[7:0];
  end
end

assign CpuData[7:0] = CpuDataOE ? ReadDataReg[7:0] : 8´bz;

// Option #2A: asynchronous write logic

always @(ControlRegSel or CpuData[7:0] or Wr_ or Reset_)
begin
  if (!Reset_)
    ControlReg[7:0] = 8´h0;  // reset state is cleared
  else if (ControlRegSel && !Wr_)
    ControlReg[7:0] = CpuData[7:0];
  // missing else forces memory element: intentional latch!
end

// Option #2B: synchronous write logic

always @(posedge CpuClk)
begin
  if (!Reset_)  // synchronous reset
    ControlReg[7:0] <= 8´h0;
  else if (ControlRegSel && !Wr_)
    ControlReg[7:0] <= CpuData[7:0];
end
```

**FIGURE 10.8** Control/status register logic.