XSver: Sequential Verilog Examples

Early digital designers and many designers through the 1980s wrote out state tables by hand and built corresponding circuits using the synthesis methods that we described in Section 7.4. However, hardly anyone does that anymore. Nowadays, most state tables are specified with a hardware description language (HDL) such as ABEL, VHDL, or Verilog. The HDL compiler then performs the equivalent of our synthesis methods and realizes the specified machine in a PLD, CPLD, FPGA, ASIC, or other target technology.

This section has Verilog state-machine and other sequential-circuit design examples for targeting to CPLDs, FPGAs, or ASICs. Some examples illustrate the decisions that are made when partitioning a design into multiple modules.

As we explained in Section 7.13, the basic Verilog language features that we introduced in Section 5.4, including `always` blocks, are just about all that is needed to model sequential-circuit behavior. Unlike ABEL, Verilog does not provide any special language elements for modeling state machines. Instead, most designers use a combination of existing "standard" features—most notably parameter definitions and case statements—to write state-machine descriptions. We'll use this method in the examples in this section.

XSver.1 T-Bird Tail Lights

We described and designed a "T-bird tail-lights" state machine in Section 7.5. Just for fun, the T-bird's rear end is reproduced below, in Figure XSver-1. The tail-light flashing sequence is shown in Figure XSver-2.

Table XSver-1 is an equivalent Verilog program for the T-bird tail-lights machine. The state transitions in this machine are defined exactly the same as in the state diagram of Figure 7-58 on page 575. The machine uses an output-coded state assignment, taking advantage of the fact that the tail-light output values are different in each state. In this example, we have integrated the flip-flop and next-state behaviors into a single `always` block.

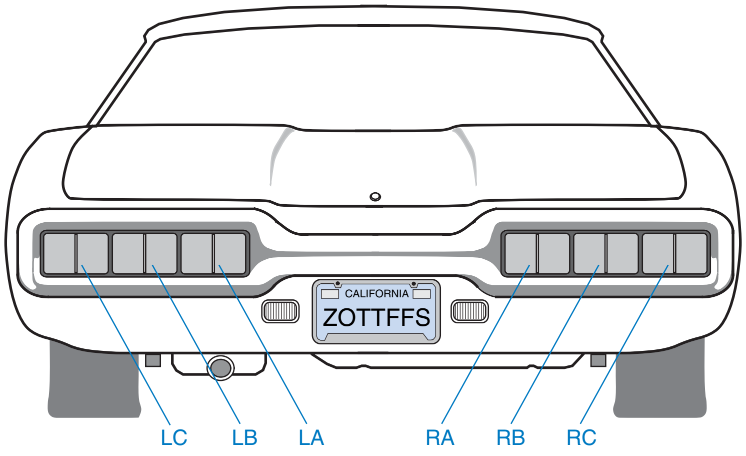

Figure XSver-1
T-bird tail lights.

Table XSver-1 Verilog module for the T-bird tail-lights machine.

```
module Vrtbird ( CLOCK, RESET, LEFT, RIGHT, HAZ, LIGHTS);
  input CLOCK, RESET, LEFT, RIGHT, HAZ;
  output [1:6] LIGHTS;
  reg [1:6] LIGHTS;

  parameter IDLE = 6'b000000,  // Output-coded state assignment
            L3   = 6'b111000,  //   and tail-light patterns
            L2   = 6'b110000,
            L1   = 6'b100000,
            R1   = 6'b000001,
            R2   = 6'b000011,
            R3   = 6'b000111,
            LR3  = 6'b111111;

  always @ (posedge CLOCK)  // State memory (with sync. reset)
    if (RESET) LIGHTS <= IDLE; else
      case (LIGHTS)         // and next-state logic.
        IDLE : if       (HAZ || (LEFT && RIGHT)) LIGHTS <= LR3;
               else if (LEFT)                    LIGHTS <= L1;
               else if (RIGHT)                   LIGHTS <= R1;
               else LIGHTS <= IDLE;
        L1   : if (HAZ) LIGHTS <= LR3; else LIGHTS <= L2;
        L2   : if (HAZ) LIGHTS <= LR3; else LIGHTS <= L3;
        L3   : LIGHTS <= IDLE;
        R1   : if (HAZ) LIGHTS <= LR3; else LIGHTS <= R2;
        R2   : if (HAZ) LIGHTS <= LR3; else LIGHTS <= R3;
        R3   : LIGHTS <= IDLE;
        LR3  : LIGHTS <= IDLE;
        default : ;
      endcase
endmodule
```

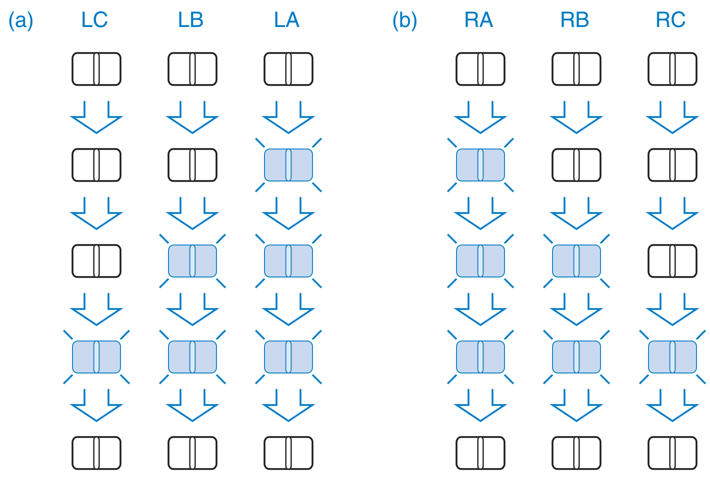

Figure XSver-2 Flashing sequence for T-bird tail lights: (a) left turn; (b) right turn.

IDLE MUSINGS

In Verilog state machines, it's not necessary to make an explicit assignment of a next state if it's the same state that the machine is already in. In the execution of an `always` block, a Verilog variable (such as a `reg`) keeps its value if no assignment is made to it. Thus, in Table XSver-1, the final "`else`" clause in the `IDLE` state could be omitted, with no effect on the machine's behavior.

Separately, the robustness of the state machine in Table XSver-1 could be improved by replacing the null statement in the "`default`" case with a transition to the `IDLE` state. As it turns out, this also yields a smaller realization—only 24 product terms versus 38 when the module is targeted to the Xilinx XC9500 CPLD family.

XSver.2 The Guessing Game

A "guessing-game" machine was defined in Section 7.7.1 starting on page 580, with the following description:

> Design a clocked synchronous state machine with four inputs, G1–G4, that are connected to pushbuttons. The machine has four outputs, L1–L4, connected to lamps or LEDs located near the like-numbered pushbuttons. There is also an ERR output connected to a red lamp. In normal operation, the L1–L4 outputs display a 1-out-of-4 pattern. At each clock tick, the pattern is rotated by one position; the clock frequency is about 4 Hz.
>
> Guesses are made by pressing a pushbutton, which asserts an input Gi. When any Gi input is asserted, the ERR output is asserted if the "wrong" pushbutton was pressed, that is, if the Gi input detected at the clock tick does not have the same number as the lamp output that was asserted before the clock tick. Once a guess has been made, play stops and the ERR output maintains the same value for one or more clock ticks until the Gi input is negated, then play resumes.

As we discussed in Section 7.7.1, the machine requires six states—four in which a corresponding lamp is on, and two for when play is stopped after either a good or a bad pushbutton push. A Verilog module for the guessing game is shown in Table XSver-2 on the next page. This version also includes a `RESET` input that forces the game to a known starting state.

LOGICAL CODING STYLES

Several different coding styles are possible for logical expressions. If we need an expression that is true when 1-bit variable `G1` is 1, we can write "`G1==1'b1`". But remember that a logical expression returns `1'b1` if true, and `1'b0` if false. So, we can instead write simply "`G1`" instead of "`G1==1'b1`". Similarly, to test for 0, we can write "`~G1`" instead of "`G1==1'b0`". We use the compact style in Table XSver-2.

We also exploit the 1/0 encoding of true/false in this module's output logic. We assign "`L1=(Sreg==S1)`" to get a 1-bit result which is 1 only in the `S1` state. The alternative coding style is clunkier: "`if (Sreg==S1) L1=1'b1; else L1=1'b0`".

Table XSver-2 Verilog module for the guessing-game machine.

```
module Vrggame ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  input CLOCK, RESET, G1, G2, G3, G4;
  output L1, L2, L3, L4, ERR;
  reg L1, L2, L3, L4, ERR;
  reg [2:0] Sreg, Snext;    // State register and next state
  parameter S1   = 3'b001,  // State encodings for 4 running states,
            S2   = 3'b010,
            S3   = 3'b011,
            S4   = 3'b100,
            SOK  = 3'b101,  // OK state, and
            SERR = 3'b110;  // error state

  always @ (posedge CLOCK)  // Create state memory with sync reset
    if (RESET) Sreg <= SOK; else Sreg <= Snext;

  always @ (G1 or G2 or G3 or G4 or Sreg)   // Next-state logic
    case (Sreg)
      S1   : if (G2 || G3 || G4) Snext = SERR;
             else if (G1)        Snext = SOK;
             else                Snext = S2;
      S2   : if (G1 || G3 || G4) Snext = SERR;
             else if (G2)        Snext = SOK;
             else                Snext = S3;
      S3   : if (G1 || G2 || G4) Snext = SERR;
             else if (G3)        Snext = SOK;
             else                Snext = S4;
      S4   : if (G1 || G2 || G3) Snext = SERR;
             else if (G4)        Snext = SOK;
             else                Snext = S1;
      SOK : if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SOK;
      SERR: if (~G1 & ~G2 & ~G3 & ~G4) Snext = S1; else Snext = SERR;
      default : Snext = S1;
    endcase

  always @ (Sreg) begin      // Output logic
    L1  = (Sreg == S1);
    L2  = (Sreg == S2);
    L3  = (Sreg == S3);
    L4  = (Sreg == S4);
    ERR = (Sreg == SERR);
  end
endmodule
```

The module is pretty much a straightforward translation of the original state diagram in Figure 7-60 on page 581. The parameter definitions at the beginning of the module establish the state encoding, a simple binary sequence using three bits. At reset the machine goes to the `SOK` state, and the `default` case sends it to the `S1` state if it should somehow enter one of the two unused states.

For this state machine, it is also possible to use an output-coded state assignment, using just the lamp and error output signals that are already required. Verilog does not have a convenient mechanism for grouping together the module's existing output signals and using them for state, but we can still achieve the desired effect with the changes shown in Table XSver-3. Here, `Sreg` and `Snext` are increased to five bits each. The state definitions in the parameter declaration are changed, and we use a comment to document the correspondence between outputs and `Sreg` bits. Finaly, we change each of the output assignment statements to pick off the appropriate bit instead of fully decoding the state.

Table XSver-3 Changes to Verilog guessing-game module for output-coded state assignment.

```
module Vrggameoc ( CLOCK, RESET, G1, G2, G3, G4, L1, L2, L3, L4, ERR );
  ...
  reg [1:5] Sreg, Snext;      // State register and next state
  // The five state bits correspond to outputs L1, L2, L3, L4, ERR
  parameter S1   = 5'b10000,  // Output coded state assignment
            S2   = 5'b01000,  //  for 4 running states,
            S3   = 5'b00100,
            S4   = 5'b00010,
            SOK  = 5'b00000,  // OK state, and
            SERR = 5'b00001;  // error state
  ...
  always @ (Sreg) begin   // Output logic
    L1  = Sreg[1];
    L2  = Sreg[2];
    L3  = Sreg[3];
    L4  = Sreg[4];
    ERR = Sreg[5];
  end
endmodule
```

OUTPUT-CODED PERFORMANCE

It's interesting to compare the performance of the original and output-coded versions of the guessing-game state machine. I targeted both modules to the Xilinx XC9500 CPLD family using Xilinx ISE tools. The initial results surprised me, because they were almost exactly the same! Upon further investigation, I discovered that the ISE synthesizer's "FSM (finite-state-machine) extraction" algorithm was turned on. This facility tries to discover state machines within the designer's code, and then applies its own ideas about what might be a good state encoding. It applied almost the same ideas to both modules.

When I turned off "FSM extraction" for the output-coded module, the results were quite different. Both versions fit in an XC9536-5 CPLD; but the original state machine needed only 17 product terms, while the output-coded version was much larger, with 27 product terms. On the other hand, the output-coded version was much faster, with a minimum clock period of 8.3 versus 10.8 ns, and very short clock-to-output delay, as expected—only 4.0 versus 11.5 ns.

XSver.3 Reinventing Traffic-Light Controllers

Our next example is from the world of cars and traffic. Traffic-light controllers in California, especially in the fair city of Sunnyvale, are carefully designed to *maximize* the waiting time of cars at intersections. An infrequently used intersection (one that would have no more than a "yield" sign if it were in Chicago) has the sensors and signals shown in Figure XSver-3 below. The state machine that controls the traffic signals uses a 1-Hz clock and a timer and has four inputs:

NSCAR	Asserted when a car on the north-south road is over either sensor on either side of the intersection.
EWCAR	Asserted when a car on the east-west road is over either sensor on either side of the intersection.
TMLONG	Asserted if more than five minutes has elapsed since the timer started; remains asserted until the timer is reset.
TMSHORT	Asserted if more than five seconds has elapsed since the timer started; remains asserted until the timer is reset.

The state machine has seven outputs:

NSRED, NSYELLOW, NSGREEN	Control the north-south lights.
EWRED, EWYELLOW, EWGREEN	Control the east-west lights.
TMRESET	When asserted, resets the timer and negates TMSHORT and TMLONG. The timer starts timing when TMRESET is negated.

Figure XSver-3 Traffic sensors and signals at an intersection in Sunnyvale, California.

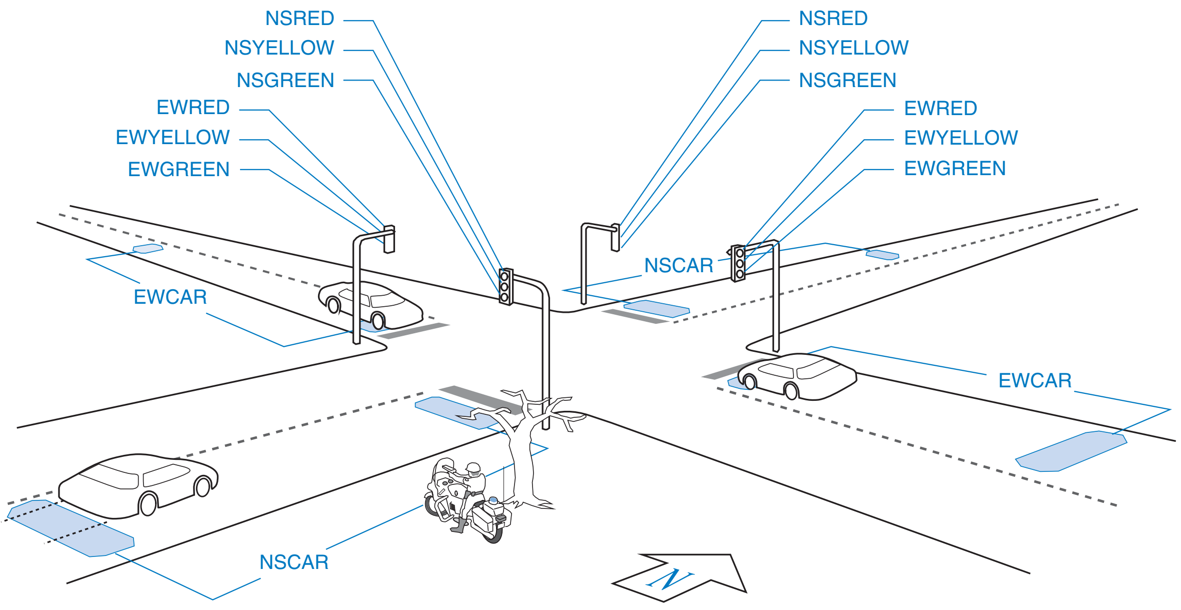

Table XSver-4 Verilog module for Sunnyvale traffic-light controller.

```
module Vrsvale ( CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG,
                 OVERRIDE, FLASHCLK, NSRED, NSYELLOW, NSGREEN,
                 EWRED, EWYELLOW, EWGREEN, TMRESET );
  input CLOCK, RESET, NSCAR, EWCAR, TMSHORT, TMLONG, OVERRIDE, FLASHCLK;
  output NSRED, NSYELLOW, NSGREEN, EWRED, EWYELLOW, EWGREEN, TMRESET;
  reg [2:0] Sreg, Snext;       // State register and next state
  parameter NSGO    = 3'b000,  // State encodings
            NSWAIT  = 3'b001,
            NSWAIT2 = 3'b010,
            NSDELAY = 3'b011,
            EWGO    = 3'b100,
            EWWAIT  = 3'b101,
            EWWAIT2 = 3'b110,
            EWDELAY = 3'b111;

  always @ (posedge CLOCK)   // Create state memory with sync reset
    if (RESET) Sreg <= NSDELAY; else Sreg <= Snext;

  always @ (NSCAR or EWCAR or TMSHORT or TMLONG)  // Next-state logic.
    case (Sreg)
      NSGO :                                      // North-south green.
        if      (~TMSHORT)        Snext = NSGO;   // Minimum 5 seconds.
        else if (TMLONG)          Snext = NSWAIT; // Maximum 5 minutes.
        else if ( EWCAR & ~NSCAR) Snext = NSGO;   // Make EW car wait.
        else if ( EWCAR &  NSCAR) Snext = NSWAIT; // Thrash if cars both ways.
        else if (~EWCAR &  NSCAR) Snext = NSWAIT; // New NS car? Make it stop!
        else                      Snext = NSGO;   // No one coming, stay as is.
      NSWAIT  : Snext = NSWAIT2;                  // Yellow light,
      NSWAIT2 : Snext = NSDELAY;                  //   two ticks for safety.
      NSDELAY : Snext = EWGO;                     // Red both ways for safety.
      EWGO    :                                   // East-west green.
        if      (~TMSHORT)        Snext = EWGO;   // Same behavior as above.
        else if (TMLONG)          Snext = EWWAIT;
        else if ( NSCAR & ~EWCAR) Snext = EWGO;
        else if ( NSCAR &  EWCAR) Snext = EWWAIT;
        else if (~NSCAR &  EWCAR) Snext = EWWAIT;
        else                      Snext = EWGO;
      EWWAIT  : Snext = EWWAIT2;
      EWWAIT2 : Snext = EWDELAY;
      EWDELAY : Snext = NSGO;
      default : Snext = NSDELAY;                  // "Reset" state.
    endcase
```

A typical, municipally approved algorithm for controlling the traffic lights is embedded in the Verilog module of Table XSver-4. This algorithm produces two frequently seen behaviors of “smart” traffic lights. At night, when traffic is light, it holds a car stopped at the light for up to five minutes, unless a car approaches on the cross street, in which case it stops the cross traffic and lets the

Table XSver-5 (continued)

```
  assign TMRESET  = (Sreg==NSWAIT2 || Sreg==EWWAIT2);
  assign NSRED    = (OVERRIDE) ?  FLASHCLK  :
                         (Sreg!=NSGO && Sreg!=NSWAIT && Sreg!=NSWAIT2);
  assign NSYELLOW = (OVERRIDE) ?  0   : (Sreg==NSWAIT || Sreg==NSWAIT2);
  assign NSGREEN  = (OVERRIDE) ?  0   : (Sreg==NSGO);
  assign EWRED    = (OVERRIDE) ?  FLASHCLK  :
                         (Sreg!=EWGO && Sreg!=EWWAIT && Sreg!=EWWAIT2);
  assign EWYELLOW = (OVERRIDE) ?  0   : (Sreg==EWWAIT || Sreg==EWWAIT2);
  assign EWGREEN  = (OVERRIDE) ?  0   : (Sreg==EWGO);
endmodule
```

waiting car go. (The "early warning" sensor is far enough back to change the lights before the approaching car reaches the intersection.) During the day, when traffic is heavy and there are always cars waiting in both directions, it cycles the lights every five seconds, thus minimizing the utilization of the intersection and maximizing everyone's waiting time, thereby creating a public outcry for more taxes to fix the problem.

While writing the program, we took the opportunity to add two inputs that weren't in the original specification. The OVERRIDE input may be asserted by the police to disable the controller and put the signals into a flashing-red mode at a rate determined by the FLASHCLK input. This allows them to manually clear up the traffic snarls created by this wonderful invention.

Table XSver-4 uses a simple binary encoding of states. An output-coded state assignment can be made as shown in Table XSver-5. Many of the states can

Table XSver-6 Changes to Sunnyvale traffic-lights machine for output-coded state assignment.

```
reg [1:7] Sreg, Snext;       // State register and next state
  // bit positions of output-coded assignment: [1] NSRED, [2] NSYELLOW, [3] NSGREEN,
  //                                    [4] EWRED, [5] EWYELLOW, [6] EWGREEN, [7] EXTRA
  parameter NSGO    = 7'b0011000,  // State encodings
            NSWAIT  = 7'b0101000,
            NSWAIT2 = 7'b0101001,
            NSDELAY = 7'b1001000,
            EWGO    = 7'b1000010,
            EWWAIT  = 7'b1000100,
            EWWAIT2 = 7'b1000101,
            EWDELAY = 7'b1001001;
...
                                   // Output logic.
  assign TMRESET  = (Sreg==NSWAIT2 || Sreg==EWWAIT2);
  assign NSRED    = Sreg[1];
  assign NSYELLOW = Sreg[2];
  assign NSGREEN  = Sreg[3];
  assign EWRED    = Sreg[4];
  assign EWYELLOW = Sreg[5];
  assign EWGREEN  = Sreg[6];
```

be identified by a unique combination of light-output values. But there are three pairs of states that are not distinguishable by looking at the lights alone: (`NSWAIT`, `NSWAIT2`), (`EWWAIT`, `EWWAIT2`), and (`NSDELAY`, `EWDELAY`). We can handle these by adding one more state variable, `Sreg[7]` or "`EXTRA`", that has different values for the two states in each pair.

Note that the output-coded module in Table XSver-5 does not have the `OVERRIDE` and `FLASHCLK` functionality of Table XSver-4; providing this feature while keeping an output-coded state assignment requires a different approach (see Exercise XSver.9).

XSver.4 A Synchronous System Design Example

This subsection presents a representative example of a synchronous system design in Verilog. The example is a *shift-and-add multiplier* for unsigned 8-bit integers, which produces a 16-bit product using the algorithm of Section 2.8. Besides illustrating synchronous design (using a single clock), this example also shows the hierarchical possibilities of design with Verilog.

shift-and-add multiplier

As shown in Figure XSver-4, the multiplier design has five individual modules nested three levels deep. The top level is broken into a datapath module `MPYdata` and a control-unit module `MPYctrl`. The control unit contains both a state machine `MPYsm` and a counter `MPYcntr`. Before you look at any details, it's important to understand the basic data-unit registers and functions that are used to perform an 8-bit multiplication, as shown in Figure XSver-5:

MPY/LPROD	A shift register initially stores the multiplier, and accumulates the low-order bits of the product as the algorithm is executed.
HPROD	A register that is initially cleared, and accumulates the high-order bits of the product as the algorithm is executed.
MCND	A register that stores the multiplicand throughout the algorithm.
F	A combinational function equal to the 9-bit sum of HPROD and MCND if the low-order bit of MPY/LPROD is 1, and equal to HPROD (extended to 9 bits) otherwise.

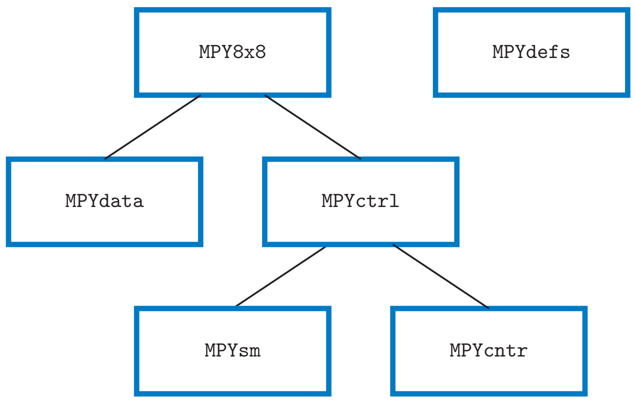

Figure XSver-4 Verilog modules and "`include`" file used in the shift-and-add multiplier.

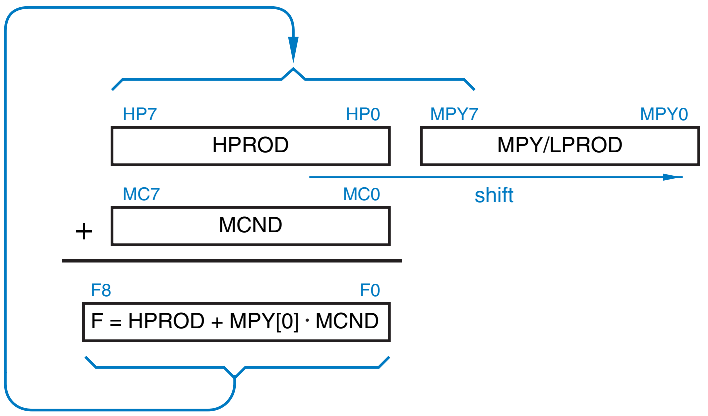

Figure XSver-5 Registers and functions used by the shift-and-add multiplication algorithm.

The MPY/LPROD shift register serves a dual purpose, holding both yet-to-be-tested multiplier bits (on the right) and unchanging product bits (on the left) as the algorithm is executed. At each step it shifts right one bit, discarding the multiplier bit that was just tested, moving the next multiplier bit to be tested to the rightmost position, and loading into the leftmost position one more product bit that will not change for the rest of the algorithm.

The Verilog `MPY8x8` module for the multiplier system has the following inputs and outputs:

CLOCK	A single clock signal for the state machine and registers.
RESET	A reset signal to clear the registers and put the state machine into its starting state before the system begins operation.
INP[7:0]	An 8-bit input bus for the multiplicand and multiplier to be loaded into registers in two clock ticks at the beginning of a multiplication.
PROD[15:0]	A 16-bit output bus that will contain the product at the end of a multiplication.
START	An input that is asserted prior to a rising clock edge to begin a multiplication. START must be negated before asserting it again will start a new multiplication.
DONE	An output that is asserted when the multiplication is done and PROD[15:0] is valid.

A timing diagram for the multiplier system is shown in Figure XSver-6. The first six waveforms show the input/output behavior and how a multiplication takes place in 10 or more clock periods as described below:

1. START is asserted. The multiplicand is placed on the INP bus and is loaded into the MCND register at the end of this clock period.
2. The multiplier is placed on the INP bus and is loaded into the MPY register at the end of the clock period.

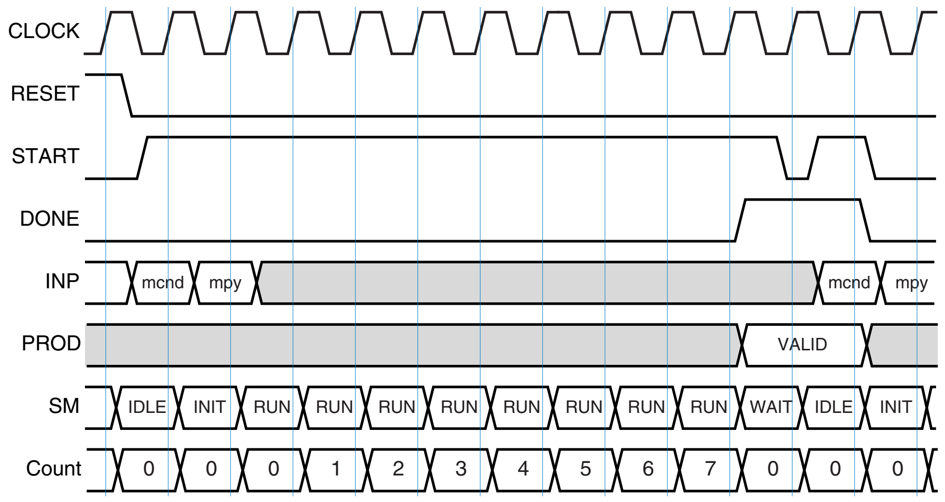

Figure XSver-6 Timing diagram for multiplier system.

3–10. One shift-and-add step is performed at each of the next eight clock ticks. Immediately following the eighth clock tick, DONE should be asserted and the 16-bit product should be available on PROD[15:0]. A new multiplication can also be started during this clock tick, but it *may* be started later.

To begin the Verilog design we create parameter definitions in an “include” file `MPYdefs.v` as shown in Table XSver-6; this is “`include`’d” by all of the modules. By changing the parameters in this file, the designer can change the state encodings for the state machine and can also create a multiplier of any desired width; we will continue to use an operand width of 8 for this example.

Our control unit `MPYcntrl` for running the multiplication is based on a decomposed state machine, as introduced in Section 7.8. A top-level state machine `MPYsm` controls the overall operation, while a counter `MPYcntr` counts the eight shift-and-add steps. These three Verilog modules are shown in tables on the next two pages.

```
parameter IDLE  = 2'b00,  // State-machine states
          INIT  = 2'b01,
          RUN   = 2'b10,
          WAIT  = 2'b11,
          SMmsb = 1,      // SM size [SMmsb:SMlsb]
          SMlsb = 0;

parameter MPYwidth = 8,            // Operand width
          MPYmsb   = MPYwidth-1,   // Index of operand MSB
          PRODmsb  = 2*MPYwidth-1, // Index of product MSB
          MaxCnt   = MPYmsb,       // Number of shift-and-add steps
          CNTRmsb  = 2;            // Step-counter size [CNTRmsb:0]
```

Table XSver-7 Definitions “include” file `MPYdefs.v` for shift-and-add multiplier.

Table XSver-8 Verilog control-unit module `MPYctrl`.

```
module MPYctrl ( RESET, CLK, START, DONE, SM );
`include "MPYdefs.v"
  input RESET, CLK, START;
  output DONE;
  reg DONE;
  output [SMmsb:SMlsb] SM;

  wire MAX;
  wire [SMmsb:SMlsb] SMi;

  MPYsm   U1 ( .RESET(RESET), .CLK(CLK), .START(START), .MAX(MAX), .SM(SMi) );
  MPYcntr U2 ( .RESET(RESET), .CLK(CLK), .SM(SMi), .MAX(MAX) );

  always @ (posedge CLK) // implement DONE output function
    if (RESET) DONE <= 1'b0;
    else if ( ((SMi==RUN) && MAX) || (SMi==WAIT) ) DONE <= 1'b1;
    else DONE <= 1'b0;

  assign SM = SMi;       // Output copy of SM state, visible to other entities
endmodule
```

Table XSver-9 Verilog state-machine module `MPYsm`.

```
module MPYsm ( RESET, CLK, START, MAX, SM );
`include "MPYdefs.v"
  input RESET, CLK, START, MAX;
  output [1:0] SM;
  reg [1:0] Sreg, Snext;

  always @ (posedge CLK) // state memory (w/ sync. reset)
    if (RESET) Sreg <= IDLE;
    else Sreg <= Snext;

  always @ (START or MAX or Sreg) // next-state logic
    case (Sreg)
      IDLE : if (START)               Snext <= INIT;
             else                     Snext <= IDLE;
      INIT :                          Snext <= RUN;
      RUN  : if (MAX && ~START)       Snext <= IDLE;
             else if (MAX && START)   Snext <= WAIT;
             else                     Snext <= RUN;
      WAIT : if (~START)              Snext <= IDLE;
             else                     Snext <= WAIT;
      default :                       Snext <= IDLE;
    endcase

  assign SM = Sreg;  // Copy state to module output
endmodule
```

Table XSver-10 Verilog counter module MPYcntr.

```
module MPYcntr ( RESET, CLK, SM, MAX );
`include "MPYdefs.v"
  input RESET, CLK;
  input [SMmsb:SMlsb] SM;
  output MAX;
  reg [CNTRmsb:0] Count;

  always @ (posedge CLK)
    if (RESET) Count <= 0;
    else if (SM==RUN) Count <= (Count + 1);
    else Count <= 0;

  assign MAX = (Count == MaxCnt);
endmodule
```

As shown in Table XSver-7, the control unit MPYctrl instantiates the state machine and the counter, and also has a small always block to implement the DONE output function, which requires an edge-triggered register. Notice how input signals RESET, CLK, and START simply "flow through" MPYctrl and become inputs of MPYsm and MPYcntr. Also notice how a local signal, SMi, is declared to receive the state from MPYsm and deliver it both to SMcntr and the output of MPYctrl.

The MPYsm state machine has four states for multiplier control. Multiplication begins when START is asserted. The machine goes to the INIT state and then the RUN state, and stays in the RUN state until the MAX input, produced by the MPYcntr module, is asserted after eight clock ticks. Then it goes to the IDLE or the WAIT state, depending on whether or not START has been negated yet.

The MPYcntr module counts from 0 to MaxCnt (MPYwidth-1) when the state machine is in the RUN state. The state-machine states and counter values during an 8-bit multiplication sequence were shown in the last two waveforms in Figure XSver-6.

The multiplier data path logic is defined in the MPYdata module, shown in Table XSver-10. This module declares local registers MPY, MCND, and HPROD. Besides the RESET, CLK, and INP inputs and the PROD output, which you would naturally need for the data path, this module also has START and the state-machine state SM as inputs. These are needed to determine when to load the MPY and MCND registers, and when to update the partial product (in the RUN state).

The last statement in the MPYdata module produces the PROD output as a combinational concatenation of the HPROD and MPY registers. Note the use of concatenation to pad the addends to nine bits in the addition operation that assigns a value to F.

Finally, the MPY8x8 entity in Table XSver-11 instantiates the data-path and control-unit entities to create the multiplier system. Besides the main system inputs and outputs, it declares only one local signal SM to convey the state-machine state from the control unit to the data path.

Table XSver-11 Verilog data-path module `MPYdata`.

```
module MPYdata (RESET, CLK, START, INP, SM, PROD );
`include "MPYdefs.v"
  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  input [SMmsb:SMlsb] SM;
  output [PRODmsb:0] PROD;
  reg [MPYmsb:0] MPY, MCND, HPROD;
  wire [MPYmsb+1:0] F;

  always @ (posedge CLK) // implement registers
    if (RESET)         // clear registers on reset
      begin MPY  <= 0; MCND <= 0; HPROD <= 0; end
    else if ((SM==IDLE) && START)  // load MCND, clear HPROD
      begin MCND <= INP; HPROD <= 0; end
    else if (SM==INIT) MPY <= INP; // load MPY
    else if (SM==RUN) begin        // shift registers
      MPY <= {F[0], MPY[MPYmsb:1]};
      HPROD <= F[(MPYmsb+1):1];  end

  assign F = (MPY[0]) ? ({1'b0,HPROD}+{1'b0,MCND}) : {1'b0, HPROD};
  assign PROD = {HPROD, MPY};
endmodule
```

Table XSver-12 Verilog top-level multiplier module `MPY8x8`.

```
module MPY8x8 (RESET, CLK, START, INP, DONE, PROD );
`include "MPYdefs.v"
  input RESET, CLK, START;
  input [MPYmsb:0] INP;
  output DONE;
  output [PRODmsb:0] PROD;
  wire [SMmsb:SMlsb] SM;

  MPYdata U1 ( .RESET(RESET), .CLK(CLK), .START(START), .INP(INP),
                      .SM(SM), .PROD(PROD) );
  MPYctrl U2 ( .RESET(RESET), .CLK(CLK), .START(START), .DONE(DONE), .SM(SM) );
endmodule
```

A test bench can be written for the multiplier as shown in Table XSver-12. Its `always` block creates a free-running clock with a 10-ns period. Its `initial` block does most of the work, using a nested `for` loop to perform multiplication of all possible pairs of 8-bit numbers, taking ten clock ticks for each pair.

After multiplying each pair of numbers, the test bench compares the circuit's result (`PROD`) with a result calculated by the simulator, and prints an error message and stops the simulation if there is a mismatch. The error message includes the current simulated time and the current values of `ii`, `jj`, and `PROD`, as well as the expected product.

Table XSver-13 VHDL test bench for shift-and-add multiplier.

```
module MPY8x8_tb ();
`include "MPYdefs.v"
  reg Tclk, RST, START;
  wire DONE;
  reg [MPYmsb:0] INP;
  wire [PRODmsb:0] PROD;
  integer ii, jj, cnt;

  MPY8x8 UUT( .CLK(Tclk), .RESET(RST), .START(START), .INP(INP),
              .DONE(DONE), .PROD(PROD) );               // instantiate UUT

  always begin    // create free-running test clock with 10 ns period
    #5 Tclk = 0;  // 5 ns high
    #5 Tclk = 1;  // 5 ns low
  end

  initial begin    // What to do starting at time 0
    RST = 1; START = 0; INP = 0; // Initial inputs
    #15;                         // Wait 15 ns,
    RST = 0;                     // then apply inputs and check outputs.
    for (ii=0; ii<=2**MPYwidth-1; ii=ii+1)  // Try all 256x256 combinations
      for (jj=0; jj<=2**MPYwidth-1; jj=jj+1) begin
        START = 1; INP = ii;
        #10;                     // Wait for 10 ns
        START = 0; INP = jj;
        #10;                     // Wait for 10 ns
        for (cnt=0; cnt<=MPYwidth-1; cnt=cnt+1)
          #10;                   // Shift-and-add MPYwidth times
        if (PROD != ii*jj) begin // Display and stop on error
          $display($time," Error, ii(%d) * jj(%d), expected %d(%b), got %d(%b)",
                   ii, jj, ii*jj, ii*jj, PROD, PROD); $stop(1); end;
      end
    $stop(1);                    // end test
  end
endmodule
```

Exercises

XSver.1 Design a clocked synchronous state machine that checks parity on a serial byte-data line with timing similar to Figure XSbb-3 in Section XSbb.1. The circuit should have three inputs, RESET, SYNC, and DATA, in addition to CLOCK, and one Moore-type output, ERROR. The ERROR output should be asserted if any DATA byte received since reset had odd parity. Using Verilog, devise a state machine that does the job using no more than four states. Include comments to describe each state's meaning and use. Write a Verilog test bench that checks your machine for proper operation by applying three bytes in succession with even, odd, and even parity.

XSver.2 Enhance the state machine in the preceding exercise by also asserting ERROR if SYNC was not asserted within eight clock ticks after the machine starts up after reset, or if any SYNC pulses fail to be exactly eight clock ticks apart. Augment the test bench to check for proper machine operation in the newly defined conditions.

XSver.3 Using Verilog, design a clocked synchronous state machine with two inputs, INIT and X, and one Moore-type output Z. As long as INIT is asserted, Z is continuously 0. Once INIT is negated, Z should remain 0 until X has been 0 for two successive ticks and 1 for two successive ticks, regardless of the order of occurrence. Then Z should go to 1 and remain 1 until INIT is asserted again. Write a Verilog test bench that checks your design for proper operation. (*Hint:* No more than ten states are required.)

XSver.4 Using Verilog, design a parallel-to-serial conversion circuit with eight 2.048-Mbps, 32-channel serial links and a single 2.048-MHz, 8-bit, parallel data bus that carries 256 bytes per frame. Each serial link should have the frame format defined in Figure XSbb-3 in Section XSbb.1. Each serial data line SDATAi should have its own sync signal SYNCi; the sync pulses should be staggered so that SYNCi + 1 has a pulse one tick after SYNCi. Show the timing of the parallel bus and the serial links, and write a table or formula that shows which parallel-bus timeslots are transmitted on which serial links and timeslots. Create a test bench that checks your design by applying at least one frame's worth of sequential parallel data (256 bytes, 0 through 255) to your circuit, and checking for the corresponding data on the serial data lines.

XSver.5 In the same environment as the preceding exercise, design a serial-to-parallel conversion circuit that converts eight serial data lines into a parallel 8-bit data bus. Create a test bench that connects the SDATAi outputs of the preceding exercise with the SDATAi inputs of this one. It should check the two circuits together by applying random parallel data bytes to the inputs of the first and looking for them at the output of the second, a certain number of clock ticks later.

XSver.6 Redesign the T-bird tail-lights machine of Section XSver.1 to include parking-light and brake-light functions. When the BRAKE input is asserted, all of the lights should go on immediately, and stay on until BRAKE is negated, independent of any other function. When the PARK input is asserted, each lamp is turned on at 50% brightness at all times when it would otherwise be off. This is achieved by driving the lamp with a 100-Hz signal DIMCLK with a 50% duty cycle. Partition the Verilog design into as many entities as you feel are appropriate, but target the design to a single CPLD or FPGA. Also, write a short description of how your system works.

XSver.7 The operation of the guessing game in Section XSver.2 is very predictable; it's easy for a player to learn the rate at which the lights change and always hit the button at the right time. The game is more fun if the rate of change is more variable.

Modify the Verilog guessing-game module of Table XSver-2 so that in states `S1`–`S4`, the state machine advances only if a new input, `SEN`, is asserted. (`SEN` is intended to be hooked up to a pseudorandom bit-stream generator.) Button pushes should be recognized whether or not `SEN` is asserted.

Also, add another `always` block to the program to provide a pseudorandom bit-stream generator using an 8-bit LFSR. After how many clock ticks does the bit sequence repeat? What is the maximum number of 0s that occur in a row? What is the maximum number of 1s? How can you double these numbers?

XSver.8 Modify the behavior of the Sunnyvale traffic-light-controller state machine in Table XSver-4 to have more reasonable behavior, the kind you'd like to see for traffic lights in your own home town.

XSver.9 Modify the Sunnyvale traffic-lights state machine of Table XSver-4 to use an output-coded state assignment *and* still provide an `OVERRIDE` input that can be asserted to make the red lights flash at a 0.5 Hz rate. (*Hint:* Eliminate the `FLASHCLK` input, create one new state, and modify the next-state logic.)

XSver.10 Using Verilog, design a data unit and a control-unit state machine for multiplying 8-bit two's-complement numbers using the algorithm discussed in Section 2.8.

XSver.11 Using Verilog, design a data unit and control-unit state machine for dividing 8-bit unsigned numbers using the shift-and-subtract algorithm discussed in Section 2.9.