Example 24

An Integer Square Root Algorithm

The C algorithm shown in Fig. 2.8 performs an integer square root of the input *a* as shown in Table 2.1. Note from Table 2.1 that the difference between successive squares, *delta*, is just the sequence of odd numbers. Also note that the *while* loop is executed as long as *square* is less than or *equal* to *a*. Thus, the value of the square root, *delta*/2 − 1, occurs in the row following the *square* value in Table 2.1.

```c
unsigned long sqrt(unsigned long a){
  unsigned long square = 1;
  unsigned long delta = 3;
  while(square <= a){
     square = square + delta;
     delta = delta + 2;
  }
     return (delta/2 - 1);
}
```

Figure 2.8 Integer square root algorithm

Table 2.1
Illustrating the algorithm in Fig. 2.8

n	square = n^2	delta	delta/2-1
0	0		
1	1	3	
2	4	5	1
3	9	7	2
4	16	9	3
5	25	11	4
6	36	13	5
7	49	15	6
8	64	17	7
9	81	19	8
10	100	21	9
11	121	23	10
12	144	25	11
13	169	27	12
14	196	29	13
15	225	31	14
16	256	33	15
17	289		

Our goal is to implement this algorithm in hardware using the same method that we used in Example 23. The datapath for this square root algorithm is shown in Fig. 2.9. We will limit the input value *a*[7:0] to eight bits so we can connect them to the eight switches *sw*[7:0]. Thus, the register *aReg* will be an 8-bit register. From Table 2.1 the maximum value of *delta* will be 33. This suggests that the size of the *delta* register should be 6 bits. However, when we are done the final value of *delta* is divided by 2 and then 1 is subtracted from this result. If the value is 33 but we only use a 5-bit register for *delta* then the value in *delta* will be 1. When we divide this by 2 we will get 0, and when we subtract 1 from 0 we get 11111, the lower four bits of which are equal to 15 – the correct result. Thus, we can get by with a 5-bit register for *delta*. However, from Table 2.1 the maximum value of *square* in the algorithm will be 256, and therefore the *square* register must be 9 bits wide. The maximum square root value will be 15 and thus the output will be stored in a 4-bit register.

In addition to the four registers *aReg*, *sqReg*, *delReg*, and *outReg*, this datapath contains four combinational modules. The output, *lteflg*, of the <= module will be 1 if *square* (the output of *sqReg*) is less than or equal to *a* (the output of *aReg*).
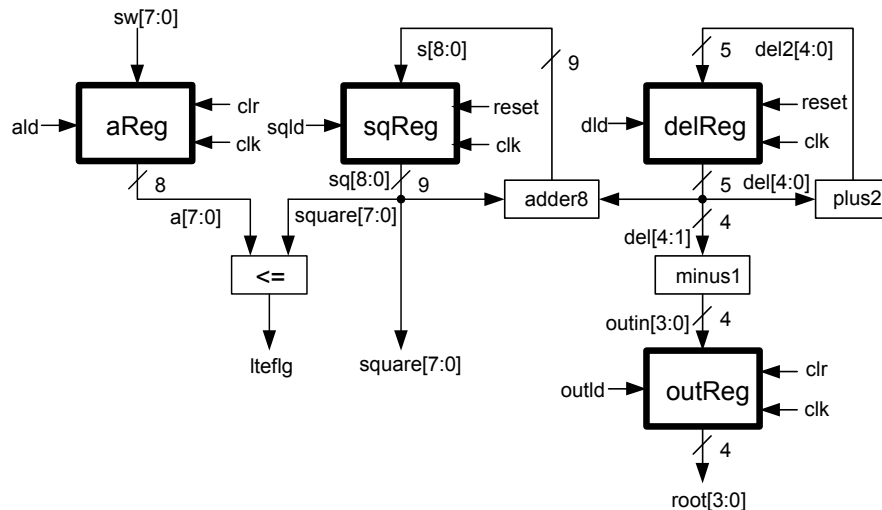

Figure 2.9 Datapath for square root algorithm

The Verilog program in Listing 2.5 implements the complete datapath shown in Fig. 2.9. The four registers have different bus widths and are initialized to different values. The generic register described in the Verilog file *regr2.v* shown in Listing 2.6 is used to instantiate all four registers in Listing 2.5. The two parameters *BIT0* and *BIT1* are used to set the lower two bits of the generic register on reset. Note in Listing 2.5 that the value of *sqReg* is initialized to 1 and the value of *delReg* is initialized to 3. In Listing 2.5 the combinational modules *adder8*, *plus2*, and *minus1* are implemented with *assign* statements, and the module *lte* is implemented by a separate *always* block.

The output signal *lteflg* in Fig. 2.9 is sent to the control unit. The control unit will then provide the datapath with the load signals for all of the registers. The design of the control unit will be described next.

Listing 2.5 SQRTpath.v

```verilog
// Example 24a: Square root datapath
module SQRTpath (
input wire clk ,
input wire reset ,
input wire ald ,
input wire sqld ,
input wire dld ,
input wire outld ,
input wire [7:0] sw ,
output reg lteflg ,
output wire [3:0] root
);
wire [7:0] a;
wire [8:0] sq, s;
wire [4:0] del, dp2;
wire [3:0] dm1;

assign s = sq + {4'b0000, del};      // adder8
assign dp2 = del + 2;                // plus2
assign dm1 = del[4:1] - 1;           // minus1

always @(*)
      begin
            if(sq <= {1'b0,a})
                  lteflg <= 1;
            else
                  lteflg <= 0;
      end

regr2 #(
      .N(8),
      .BIT0(0),
      .BIT1(0))
aReg (.load(ald),
      .clk(clk),
      .reset(reset),
      .d(sw),
      .q(a)
);

regr2 #(
      .N(9),
      .BIT0(1),
      .BIT1(0))
sqReg (.load(sqld),
      .clk(clk),
      .reset(reset),
      .d(s),
      .q(sq)
);
```

Listing 2.5 (cont.) SQRTpath.v

```
regr2 #(
      .N(5),
      .BIT0(1),
      .BIT1(1))
delReg (.load(dld),
      .clk(clk),
      .reset(reset),
      .d(dp2),
      .q(del)
);

regr2 #(
      .N(4),
      .BIT0(0),
      .BIT1(0))
outReg (.load(outld),
      .clk(clk),
      .reset(reset),
      .d(dm1),
      .q(root)
);

endmodule
```

Listing 2.6 regr2.v

```
//     Example 24b: N-bit register with reset and load
//  Resets to initial value of lowest 2 bits
module regr2
#(parameter N = 4,
parameter BIT0 = 1,
parameter BIT1 = 1)
 (input wire load ,
  input wire clk ,
  input wire reset ,
  input wire [N-1:0] d ,
  output reg [N-1:0] q
);

always @(posedge clk or posedge reset)
      if(reset == 1)
        begin
            q[N-1:2] <= 0;
            q[0] <= BIT0;
            q[1] <= BIT1;
        end
      else if(load == 1)
            q <= d;

endmodule
```

Square Root Control Unit

Listing 2.5 describes the datapath shown in Fig. 2.9. In this section we will design the control unit that will control this datapath as shown in Fig. 2.10. Note in this figure that the datapath sends the signal *lteflg* to the control unit, and the control unit sends the register load signals *ald*, *sqld*, *dld*, and *outld* to the datapath.

The control unit will be a *state machine* that defines the timing sequence of the algorithm. For this square root algorithm there are four states: *start*, *test*, *update*, and *done* as shown in Fig. 2.11. The program begins in the *start* state, and stays in this state until the *go* signal goes high. It then goes to the *test* state, which will test to see if *square* <= *a* is true or false. If it is true, then the program goes to the *update* state, which will update the values of *square* and *delta* in the *while* loop of Fig. 2.8. Otherwise, the program goes to the *done* state, which computes the return value and stays in this *done* state. Listing 2.7 is a Verilog program for the control unit in Fig. 2.10, which implements the state diagram shown in Fig. 2.11.
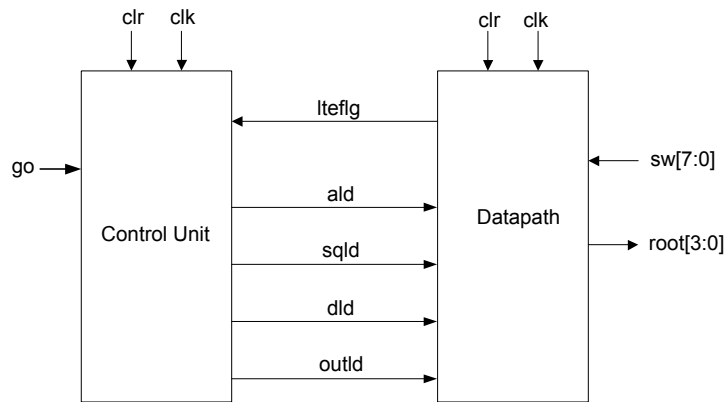

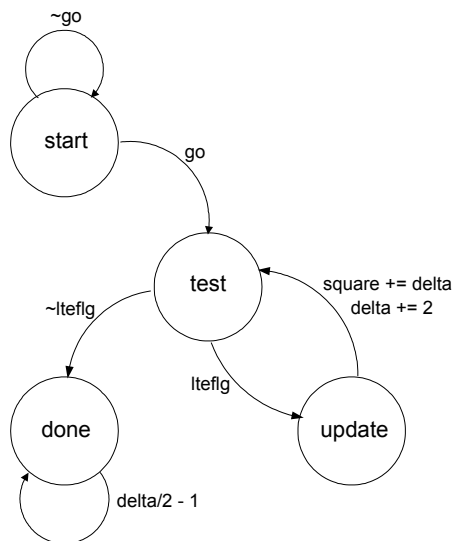Figure 2.10 Top-level logic diagram for square root algorithm

Figure 2.11 State diagram for square root algorithm

Note that we have implemented the state machine in Listing 2.7 as a Moore machine of the type shown in Fig. 1.37 with three *always* blocks: the sequential state register block and the two combinational blocks *C*1 and *C*2. Note how the *C*1 block finds the next state by directly implementing the state diagram in Fig. 2.11 with a *case* statement. The output block *C*2 also uses a *case* statement to set the register load signals to the proper values for each state. Verify that these load signals are appropriate for implementing the square root algorithm.

Listing 2.7 SQRTctrl.v

```verilog
// Example 24c: Square root control
module SQRTctrl (
input wire clk ,
input wire clr ,
input wire lteflg ,
input wire go ,
output reg ald ,
output reg sqld ,
output reg dld ,
output reg outld
);
reg[1:0] present_state, next_state;
parameter start = 2'b00, test =2'b01, update = 2'b10,
          done = 2'b11; // states

// State registers
always @(posedge clk or posedge clr)
  begin
      if (clr == 1)
             present_state <= start;
      else
             present_state <= next_state;
  end

// C1 module
always @(*)
  begin
    case(present_state)
            start: if(go == 1)
                      next_state = test;
                   else
                      next_state = start;
            test: if(lteflg == 1)
                      next_state = update;
                   else
                      next_state = done;
            update: next_state = test;
            done: next_state = done;
       default next_state = start;
    endcase
  end
```

Listing 2.7 (cont.) SQRTctrl.v

```
// C2 module
 always @(*)
   begin
       ald = 0; sqld = 0;
       dld = 0; outld = 0;
     case(present_state)
            start: ald = 1;
            test: ;
            update:
              begin
                  sqld = 1; dld = 1;
              end
            done: outld = 1;
            default ;
     endcase
   end

endmodule
```

The top-level design shown in Fig. 2.10 can be implemented in Verilog by simply instantiating the datapath and control unit as shown in Listing 2.8. Note that we have included an output signal *done* that goes high when the *done* state is entered. A simulation of this program that computes the square root of 64 is shown in Fig. 2.12.
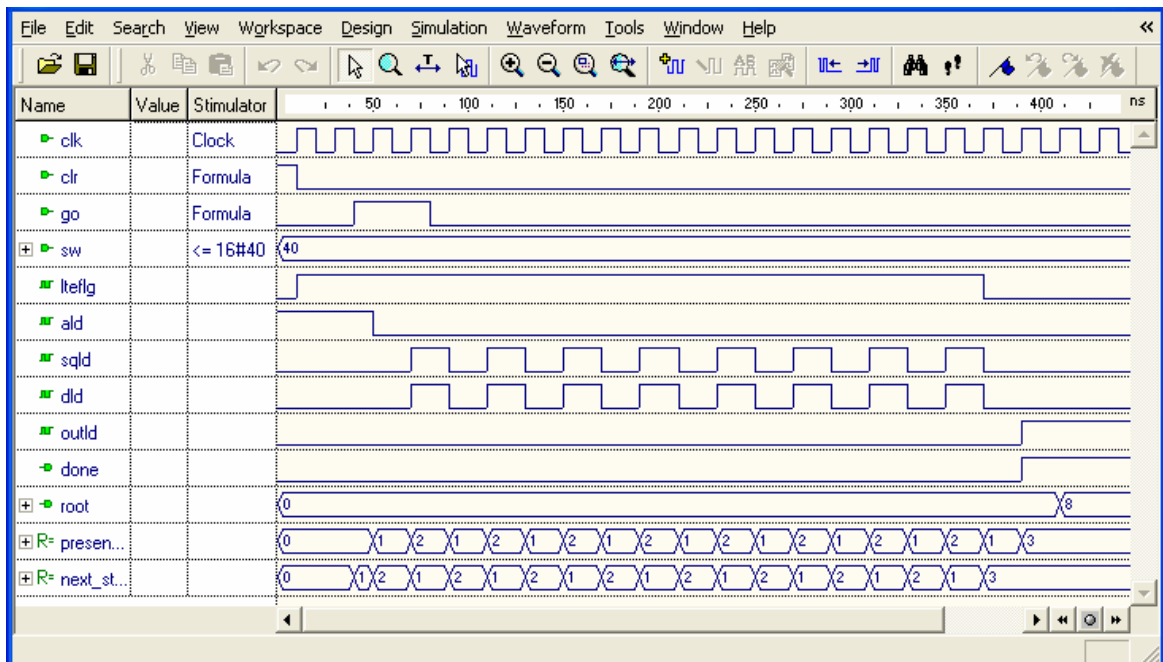

Figure 2.12 Simulation of the square root program in Listing 2.8

Listing 2.8 SQRT.v

```verilog
// Example 24d: Integer Square Root
module sqrt (
input wire clk ,
input wire clr ,
input wire go ,
input wire [7:0] sw ,
output wire done ,
output wire [3:0] root
);
wire lteflg, ald, sqld, dld, outld;

assign done = outld;

SQRTctrl sqrt1 (.clk(clk),
      .clr(clr),
      .lteflg(lteflg),
      .go(go),
      .ald(ald),
      .sqld(sqld),
      .dld(dld),
      .outld(outld)
);

SQRTpath sqrt2 (.clk(clk),
      .reset(clr),
      .ald(ald),
      .sqld(sqld),
      .dld(dld),
      .outld(outld),
      .sw(sw),
      .lteflg(lteflg),
      .root(root)
);

endmodule
```

To test the square root algorithm on the Nexys-2 board we will use the top-level design shown in Fig. 2.13. When *btn*[3] is pressed to reset the circuit the decimal value of the eight switch settings is displayed on the 7-segment display. We use the 8-bit binary-to-BCD program in Listing 1.14 of Example 12 to do this conversion. When *btn*[0] is pressed the *sqrt* component computes the square root of the switch settings, and when the calculation is complete the *done* signal switches the multiplexer so that the decimal value of the square root is displayed on the 7-segment display. Try it.
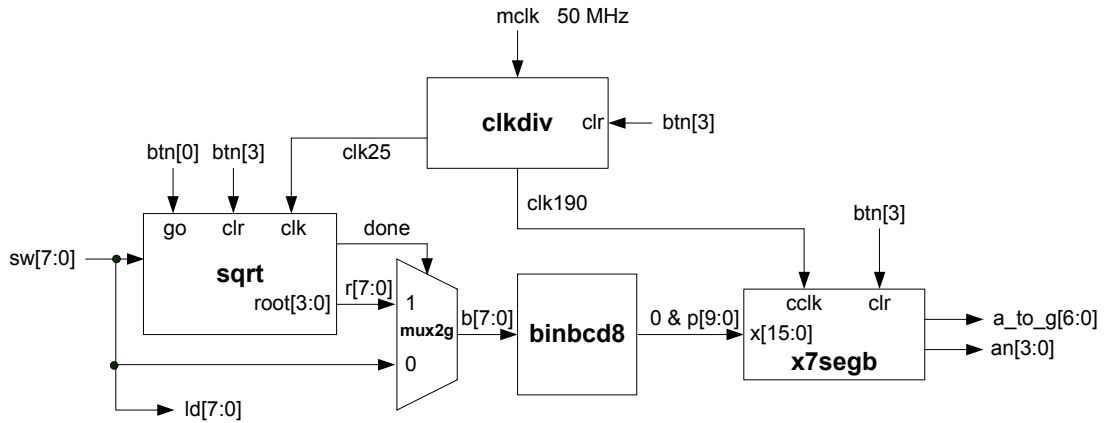
Figure 2.13 Top-level design for testing square root algorithm

Listing 2.9 sqrt_top.v

```verilog
// Example 24e: sqrt_top
module sqrt_top (
input wire mclk ,
input wire [3:0] btn ,
input wire [7:0] sw ,
output wire [7:0] ld ,
output wire dp ,
output wire [6:0] a_to_g ,
output wire [3:0] an
);
wire clk25, clk190, clr, done;
wire [15:0] x;
wire [9:0] p;
wire [3:0] root;
wire [7:0] b, r;

assign clr = btn[3];
assign r = {4'b0000,root};
assign x = {6'b000000,p};
assign ld = sw;

clkdiv U1 (.mclk(mclk),
      .clr(clr),
      .clk190(clk190),
      .clk25(clk25)
);

sqrt U2 (.clk(clk25),
      .clr(clr),
      .go(btn[0]),
      .sw(sw),
      .done(done),
      .root(root)
);
```

Listing 2.9 (cont.) sqrt_top.v

```verilog
mux2g #(
      .N(8))
U3 (.a(sw),
      .b(r),
      .s(done),
      .y(b)
);

binbcd8 U4 (.b(b),
      .p(p)
);

x7segb U5 (.x(x),
      .cclk(clk190),
      .clr(clr),
      .a_to_g(a_to_g),
      .an(an),
      .dp(dp)
);

endmodule
```