7.1 REPRESENTATION OF FLOATING-POINT NUMBERS

$N = F \times 2^E$

Examples of floating-point numbers using a 4-bit fraction and 4-bit exponent:

F = 0.101 E = 0101 $N = 5/8 \times 2^5$
F = 1.011 E = 1011 $N = -5/8 \times 2^{-5}$
F = 1.000 E = 1000 $N = -1 \times 2^{-8}$

Normalization

Unnormalized:	F = 0.0101	E = 0011	$N = 5/16 \times 2^3 = 5/2$
Normalized:	F = 0.101	E = 0010	$N = 5/8 \times 2^2 = 5/2$
Unnormalized:	F = 1.11011	E = 1100	$N = -5/32 \times 2^{-4} = -5 \times 2^{-9}$
(shift F left)	F = 1.1011	E = 1011	$N = -5/16 \times 2^{-5} = -5 \times 2^{-9}$
Normalized	F = 1.011	E = 1010	$N = -5/8 \times 2^{-6} = -5 \times 2^{-9}$

Representation of Zero

F = 0.000, E = 1000 or 0.000×2^{-8}

Figure 7-1 Flowchart for Floating-Point Multiplication

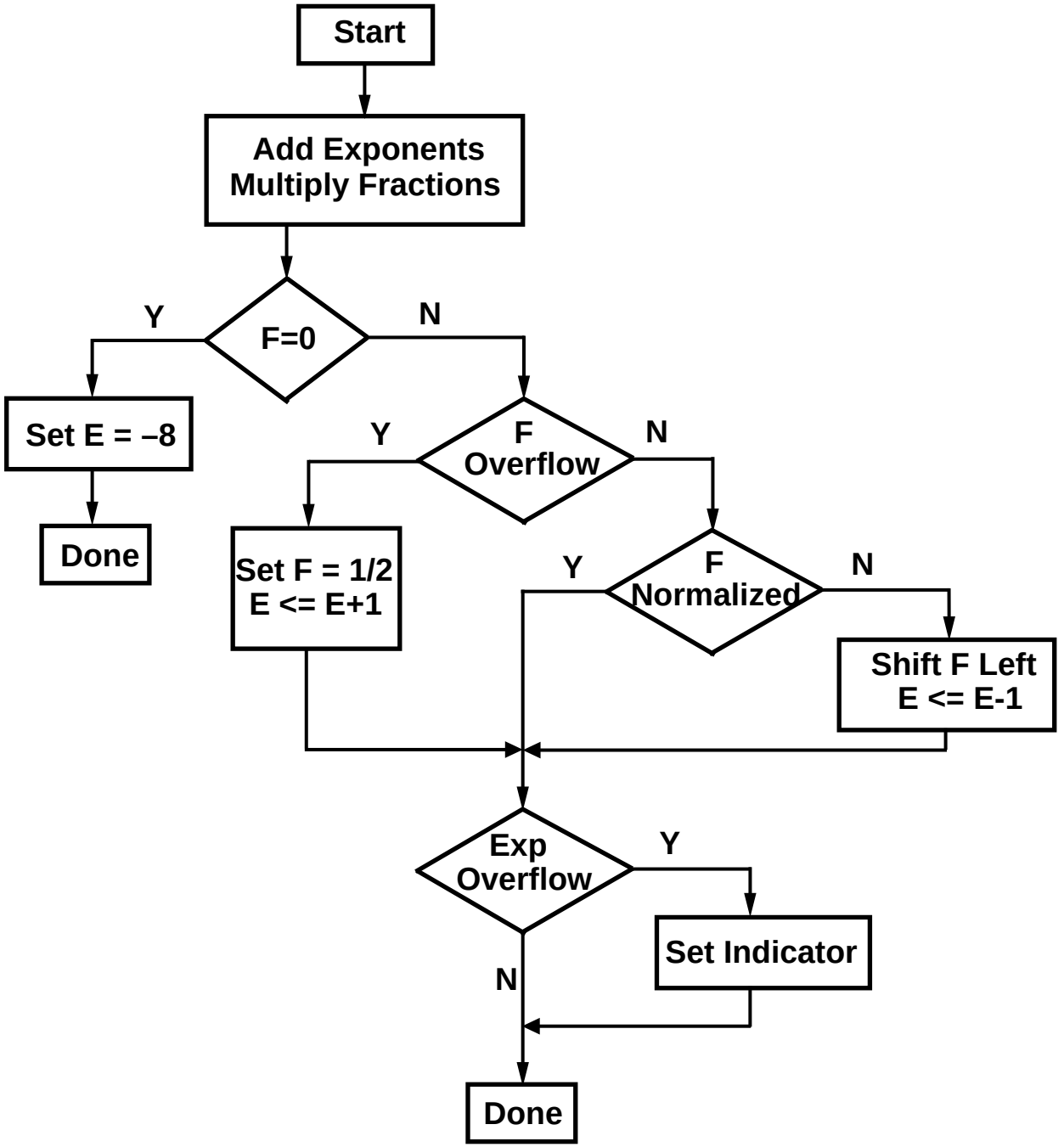


Figure 7-2(a) Exponent Adder and Fraction Multiplier

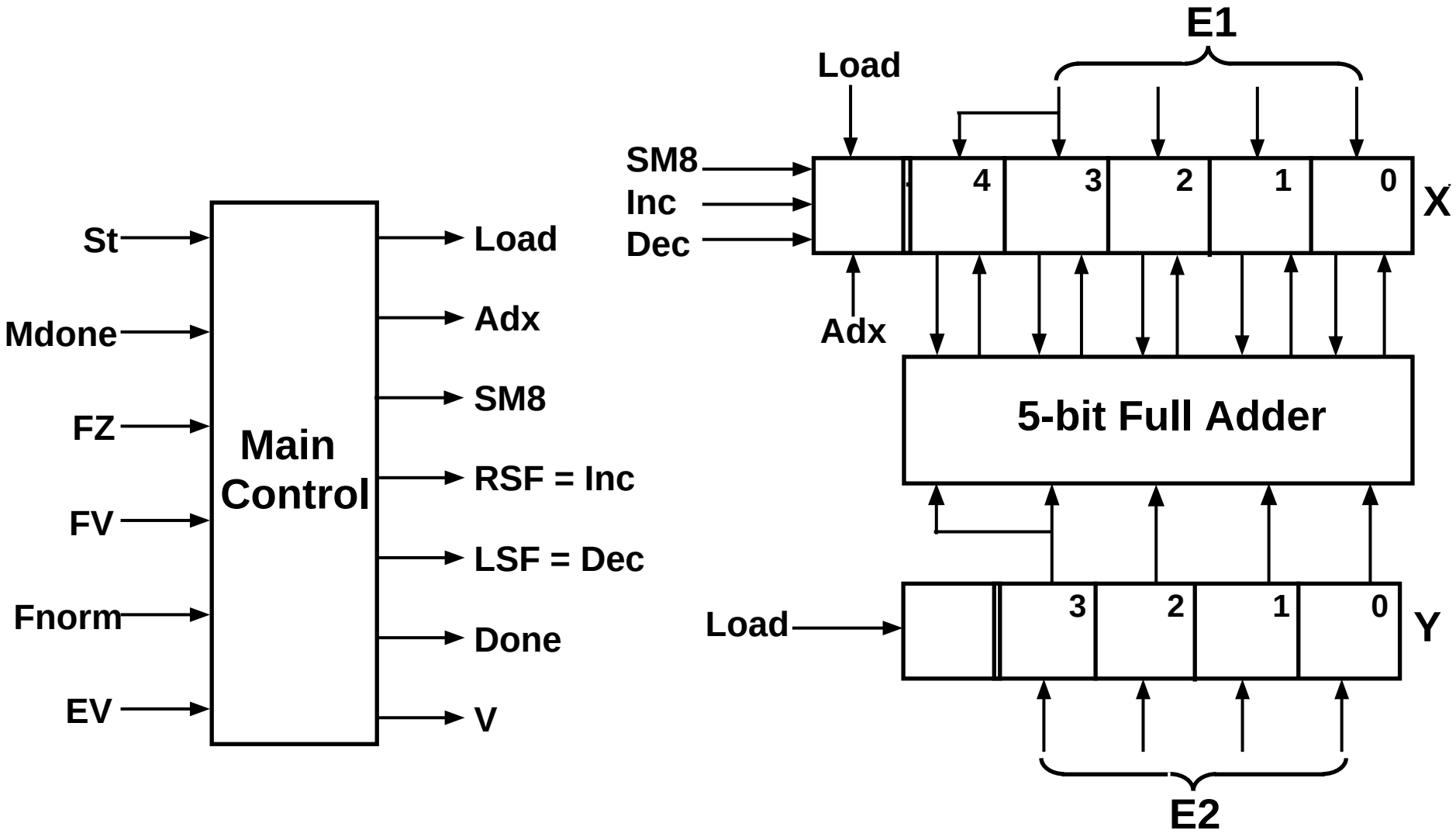


Figure 7-2(b) Exponent Adder and Fraction Multiplier

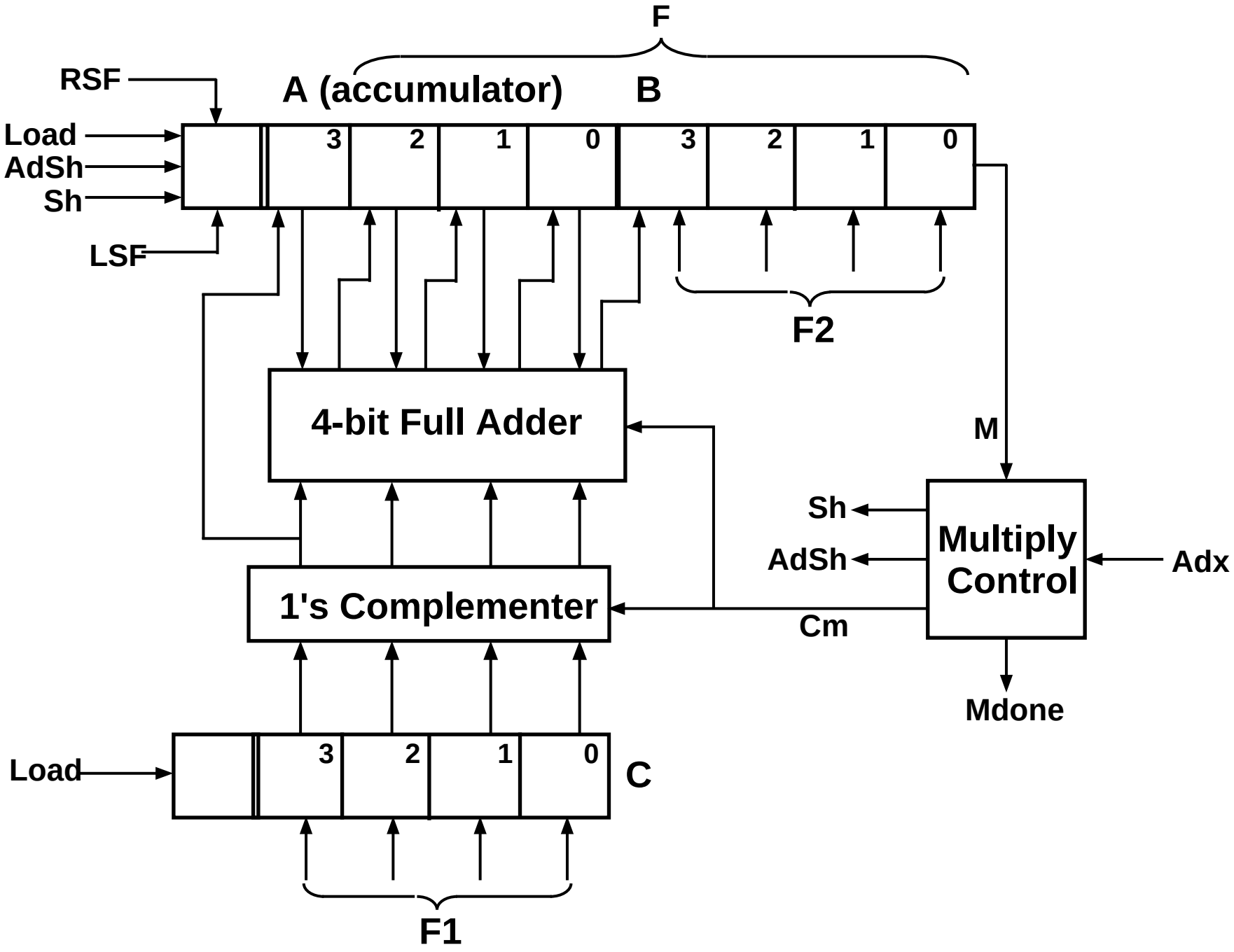


Figure 7-3 SM Chart for floating-Point Multiplication

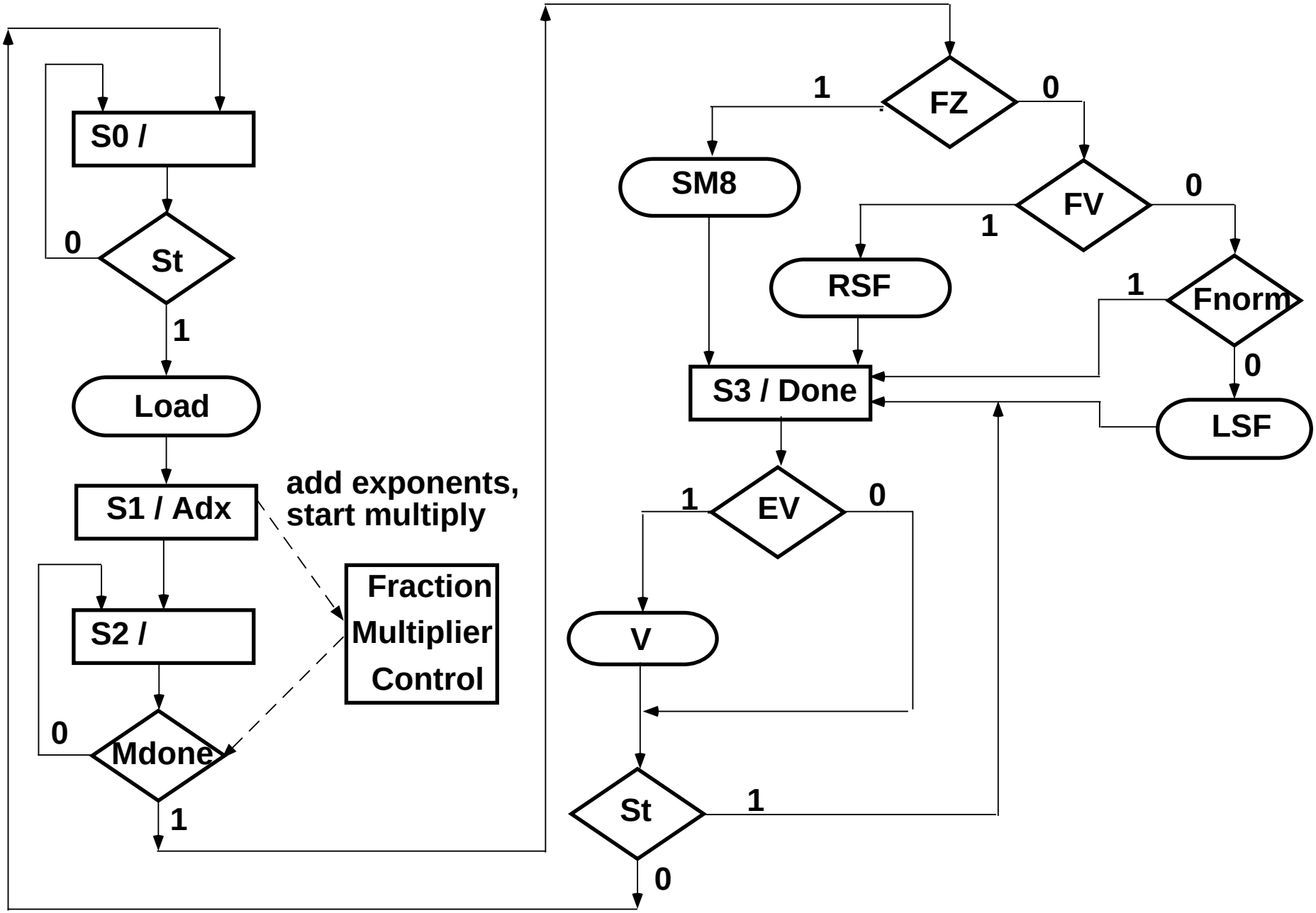


Figure 7-4 State Graph for Multiplier Control

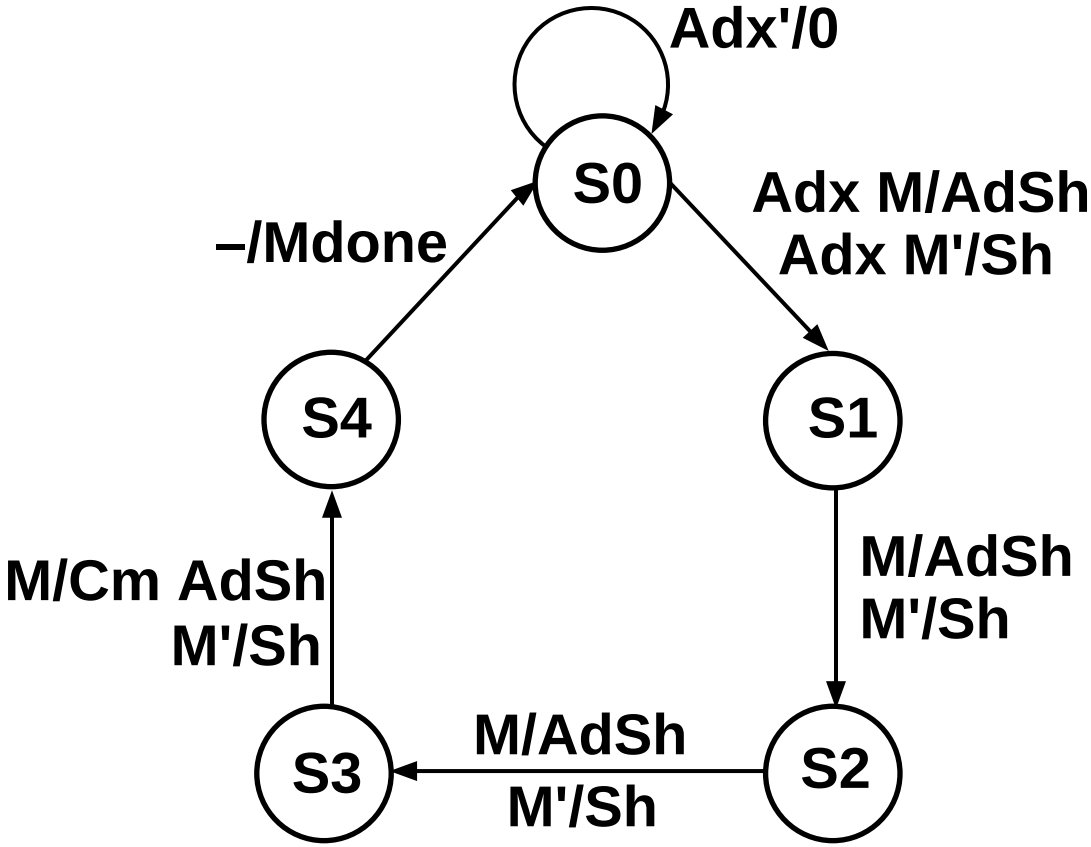


Figure 7-5(a) VHDL Code for Floating-Point Multiplier

```
library BITLIB;
use BITLIB.bit_pack.all;
entity FMUL is
      port (CLK, St: in bit; F1,E1,F2,E2: in bit_vector(3 downto 0);
            F: out bit_vector(6 downto 0); V, done:out bit);
end FMUL;
architecture FMULB of FMUL is
signal A, B, C: bit_vector(3 downto 0);                    -- fraction registers
signal X, Y: bit_vector(4 downto 0);                       -- exponent registers
signal Load, Adx, Mdone, SM8, RSF, LSF, NC: bit;  signal AdSh, Sh, Cm: bit;
signal PS1, NS1: integer range 0 to 3;                     -- present and next state
signal State, Nextstate: integer range 0 to 4;             -- mulitplier control state
alias M: bit is B(0);  constant one:bit_vector(4 downto 0):="00001";
constant neg_one:bit_vector(4 downto 0):="11111";
begin
main_control: process
      begin
      Load <= '0'; Adx <= '0';                             -- clear control signals
      SM8 <= '0'; RSF <= '0'; LSF <= '0';
      case PS1 is
            when 0 =>  done<='0'; V<='0';                  -- clear outputs
                  if St = '1' then Load <= '1'; NS1 <= 1; F <= "0000000";  end if;
            when 1 =>  Adx <= '1'; NS1 <= 2;
```

Figure 7-5(b) VHDL Code for Floating-Point Multiplier

```
            when 2 =>  if Mdone = '1' then                    -- wait for multiply
                        if A = "0000" then SM8 <= '1';        -- zero fraction
                        elsif A = "0100" and B = "0000" then  -- fraction overflow
                              RSF <= '1';                     -- shift AB right
                        elsif A(2) = A(1) then                -- test for unnormalized
                              LSF <= '1';                     -- shift AB left
                        end if;  NS1 <= 3;
                  end if;
            when 3 =>                                         --test for exp overflow
                  if X(4) /= X(3) then V <= '1'; else V <= '0'; end if;
                  done <= '1';
                  F <= A(2 downto 0) & B;                     --output fraction
                  if ST = '0' then NS1<=0; end if;
      end case;
      wait until rising_edge(CLK);
            PS1 <= NS1;
      wait for 0 ns;                                          --wait for state change
      end process main_control;
mul2c: process                                                --2's complement multiply
      begin
      AdSh <= '0'; Sh <= '0'; Cm <= '0';                      --clear control signals
      case State is
            when 0=>  Mdone <= '0';                           --start multiply
            if Adx='1' then
                  if M = '1' then AdSh <= '1'; else Sh <= '1'; end if;
                  Nextstate <= 1;
            end if;
```

Figure 7-5(c) VHDL Code for Floating-Point Multiplier

```
        when 1 | 2  =>                                          --add/shift state
        if M = '1' then AdSh <= '1'; else Sh <= '1'; end if;
        Nextstate <= State + 1;
        when 3  =>
        if M = '1' then Cm <= '1'; AdSh <= '1'; else Sh <='1'; end if;
        Nextstate <= 4;
        when 4 =>  Mdone <= '1';  Nextstate <= 0;
end case;
wait until rising_edge(CLK);
State <= Nextstate;
wait for 0 ns;                                                  --wait for state change
end process mul2c;

update: process                                                 --update registers
variable addout: bit_vector(4 downto 0);
begin
wait until rising_edge(CLK);
if Cm = '0' then addout := add4(A,C,'0');
        else addout := add4(A, not C,'1'); end if;              --add 2's comp. of C
if Load = '1' then  X <= E1(3)&E1; Y <= E2(3)&E2;
        A <= "0000"; B <= F2; C <= F1; end if;
if ADX = '1' then addvec(X,Y,'0',X,NC,5); end if;
if SM8 = '1' then  X <= "11000"; end if;
if RSF = '1' then  A <= '0'&A(3 downto 1);
        B <= A(0)&B(3 downto 1);
        addvec(X,one,'0',X,NC,5); end if;                       -- increment X
```

Figure 7-5(d) VHDL Code for Floating-Point Multiplier

```
        if LSF = '1' then
            A <= A(2 downto 0)&B(3); B <= B(2 downto 0)&'0';
            addvec(X,neg_one,'0',X,NC,5); end if;              -- decrement X
        if AdSh = '1' then
            A <= (C(3) xor Cm) & addout(3 downto 1);           -- load shifted adder
            B <= addout(0) & B(3 downto 1); end if;            -- output into A & B
        if Sh = '1' then
            A <= A(3) & A(3 downto 1);                         -- right shift A & B
            B <= A(0) & B(3 downto 1);                         -- with sign extend
        end if;
        end process update;
end FMULB;
```

Figure 7-6(a) Test Data for Floating-Point Multiply

```
list f x f1 e1 f2 e2 v done
force f1 0111 0, 1001 200, 1000 400, 0000 600, 0111 800
force e1 0001 0, 1001 200, 0111 400, 1000 600, 0111 800
force f2 0111 0, 1001 200, 1000 400, 0000 600, 1001 800
force e2 1000 0, 0001 200, 1001 400, 1000 600, 0001 800
force st 1 0, 0 20, 1 200, 0 220, 1 400, 0 420, 1 600, 0 620, 1 800, 0 820
force clk 0 0, 1 10 -repeat 20
run 1000
```

Figure 7-6(b) Simulation Results for Floating-Point Multiply

```
ns  delta       f     x   f1   e1   f2   e2  v done
  0    +0 0000000 00000 0000 0000 0000 0000  0 0
  0    +1 0000000 00000 0111 0001 0111 1000  0 0 (0.111 x 2^1) x (0.111 x 2^-8)
 10    +1 0000000 00001 0111 0001 0111 1000  0 0
 30    +1 0000000 11001 0111 0001 0111 1000  0 0
150    +2 0110001 11001 0111 0001 0111 1000  0 1    = 0.110001 x 2^-7
170    +2 0000000 11001 0111 0001 0111 1000  0 0
200    +0 0000000 11001 1001 1001 1001 0001  0 0 (1.001 x 2^-7) x (1.001 x 2^1)
250    +1 0000000 11010 1001 1001 1001 0001  0 0
370    +2 0110001 11010 1001 1001 1001 0001  0 1    = 0.110001 x 2^-6
390    +2 0000000 11010 1001 1001 1001 0001  0 0
400    +0 0000000 11010 1000 0111 1000 1001  0 0 (1.000 x 2^7) x (1.000 x 2^-7)
430    +1 0000000 00111 1000 0111 1000 1001  0 0
450    +1 0000000 00000 1000 0111 1000 1001  0 0
570    +1 0000000 00001 1000 0111 1000 1001  0 0
570    +2 0100000 00001 1000 0111 1000 1001  0 1    = 0.100000 x 2^1
590    +2 0000000 00001 1000 0111 1000 1001  0 0
600    +0 0000000 00001 0000 1000 0000 1000  0 0 (0.000 x 2^-8) x (0.000 x 2^-8)
630    +1 0000000 11000 0000 1000 0000 1000  0 0
650    +1 0000000 10000 0000 1000 0000 1000  0 0
770    +1 0000000 11000 0000 1000 0000 1000  0 0
770    +2 0000000 11000 0000 1000 0000 1000  0 1    = 0.0000000 x 2^-8
790    +2 0000000 11000 0000 1000 0000 1000  0 0
800    +0 0000000 11000 0111 0111 1001 0001  0 0 (0.111 x 2^7) x (1.001 x 2^1)
830    +1 0000000 00111 0111 0111 1001 0001  0 0
850    +1 0000000 01000 0111 0111 1001 0001  0 0
970    +2 1001111 01000 0111 0111 1001 0001  1 1    = 1.001111 x 2^8 (overflow)
990    +2 0000000 01000 0111 0111 1001 0001  0 0
```

Figure 7-7 Bus Structure for Floating-Point Multiplier

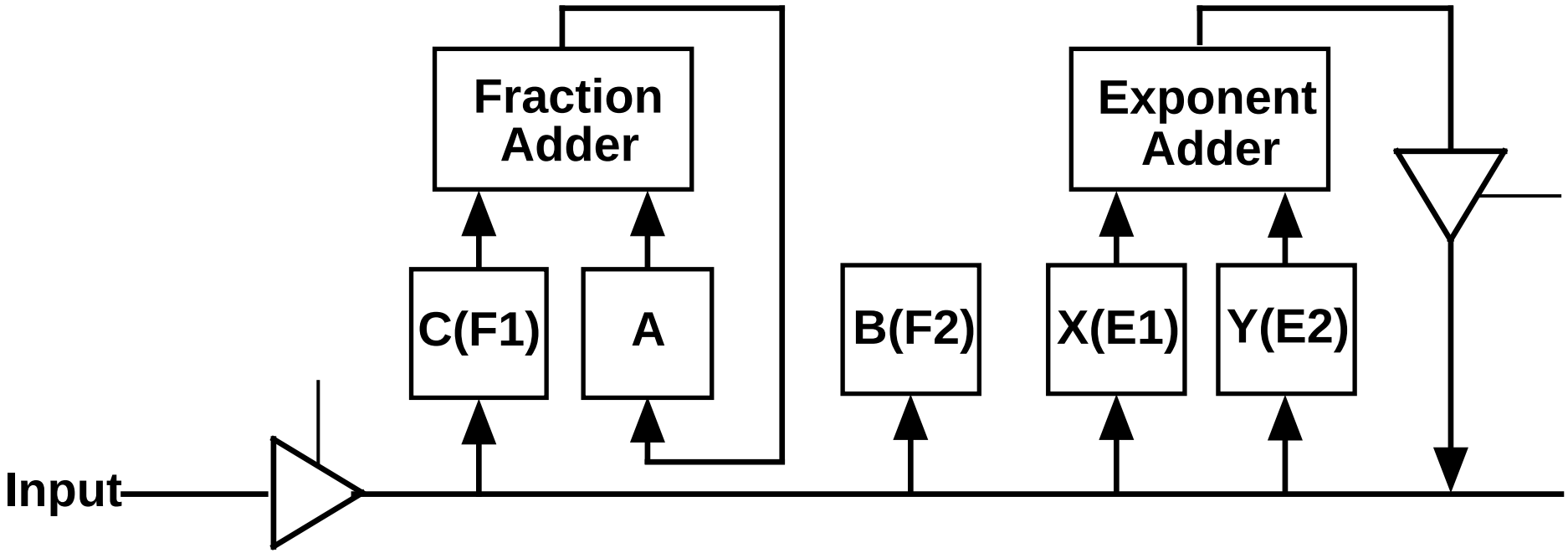


Figure 7-8(a) Top-level Schematic for Floating-Point Multiplier

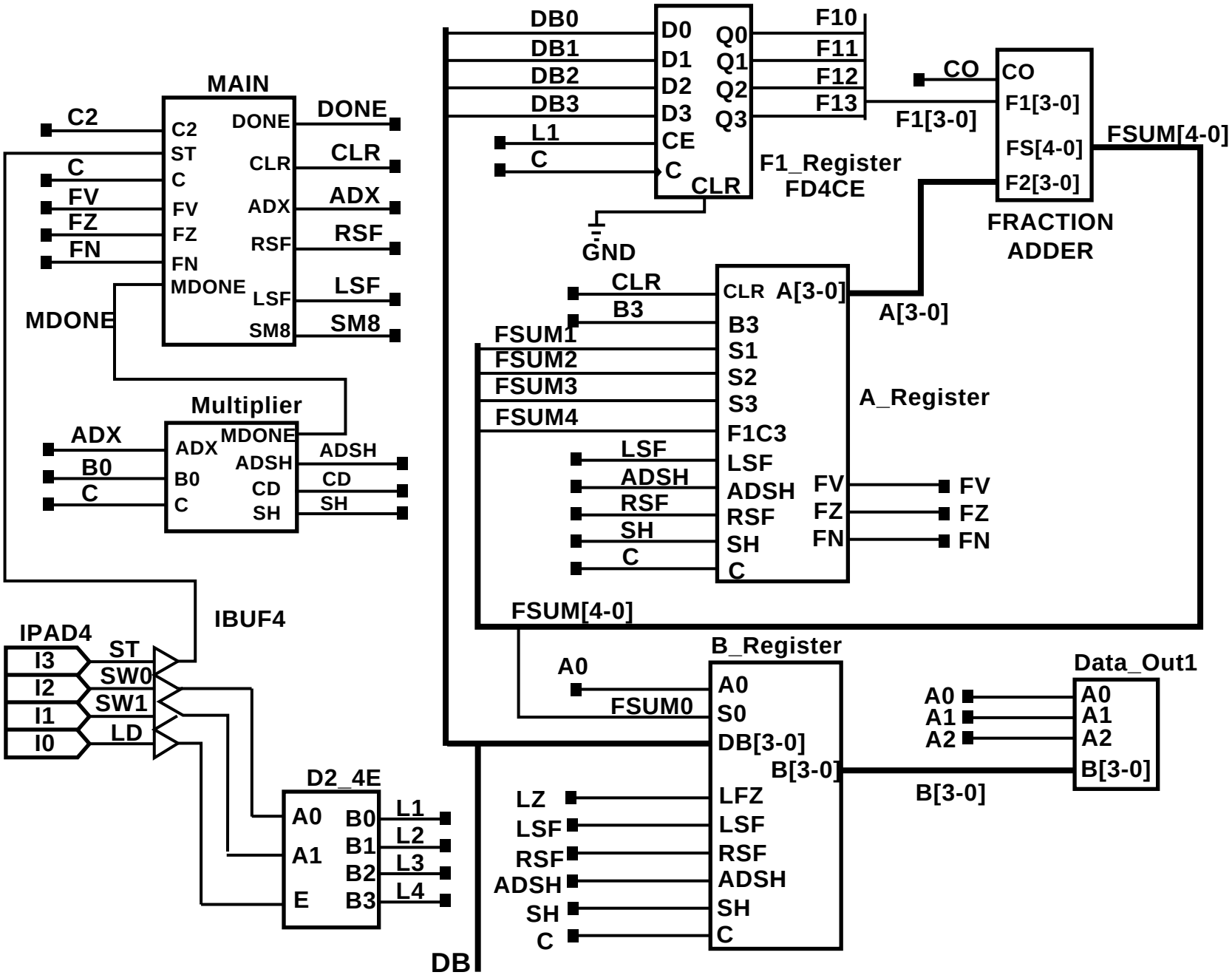


Figure 7-8(b) Top-level Schematic for Floating-Point Multiplier

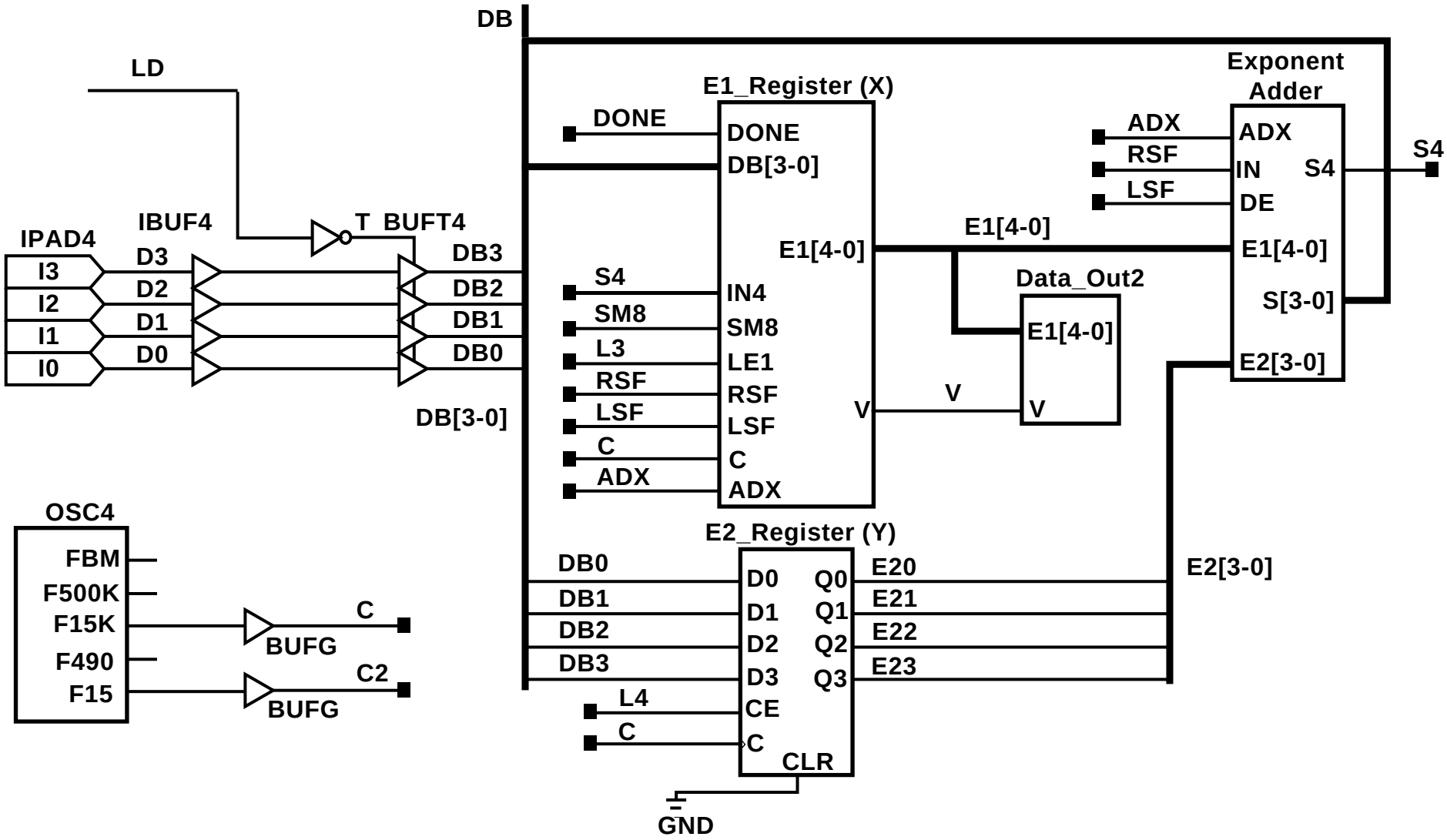


Figure 7-9 Main Control for Floating-Point Multiplier

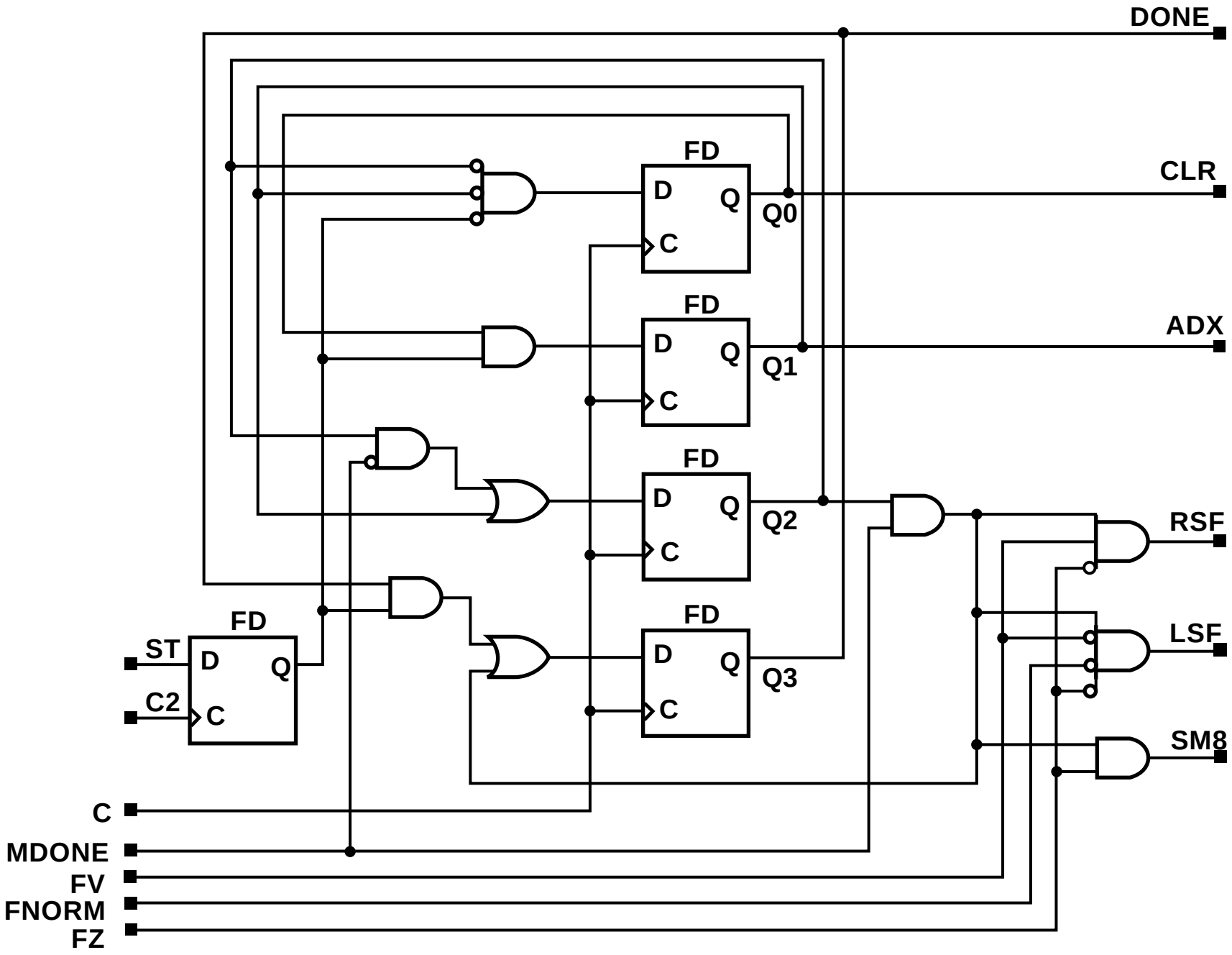


Figure 7-10 Multiplier Control

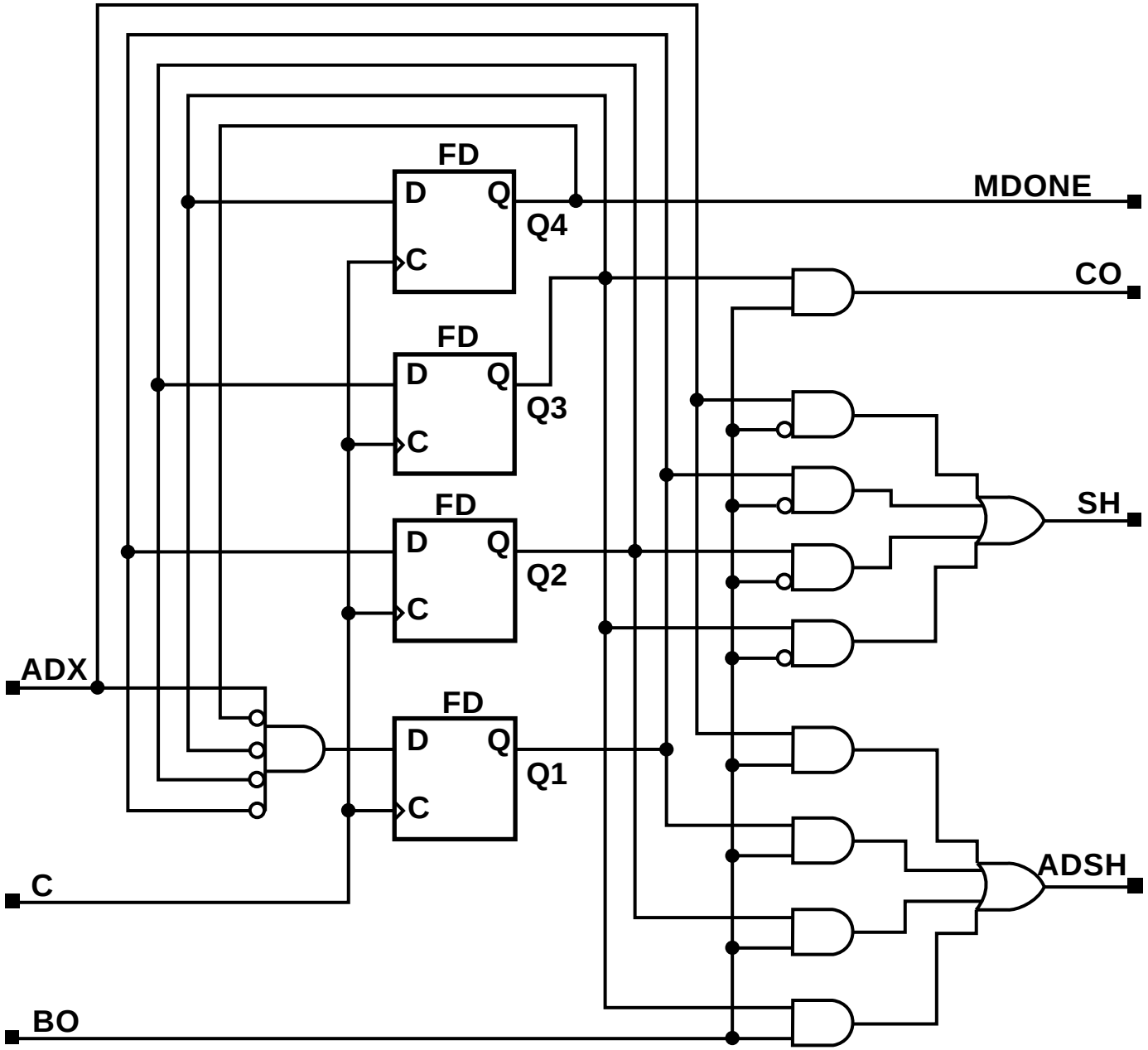


Figure 7-11 A Register

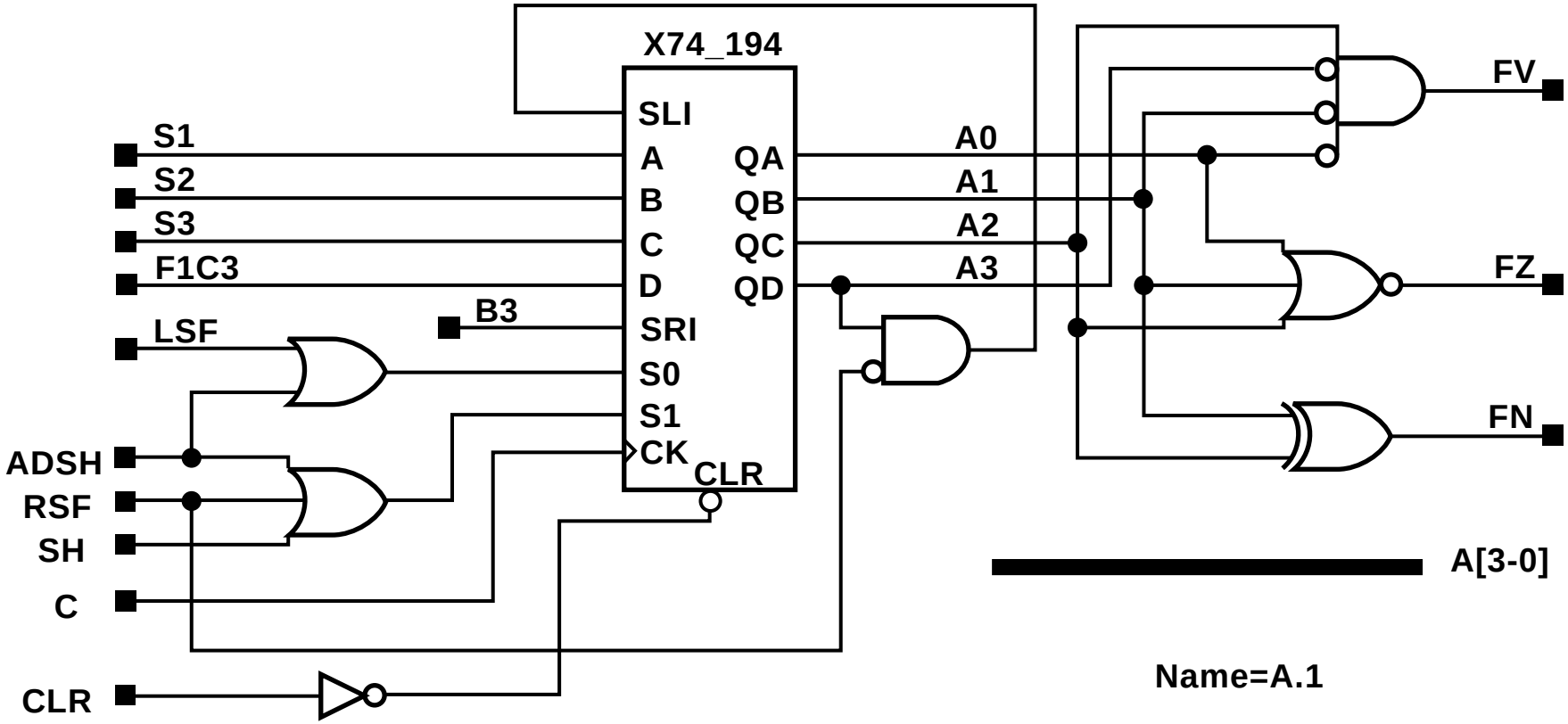


Figure 7-12 E1 Register

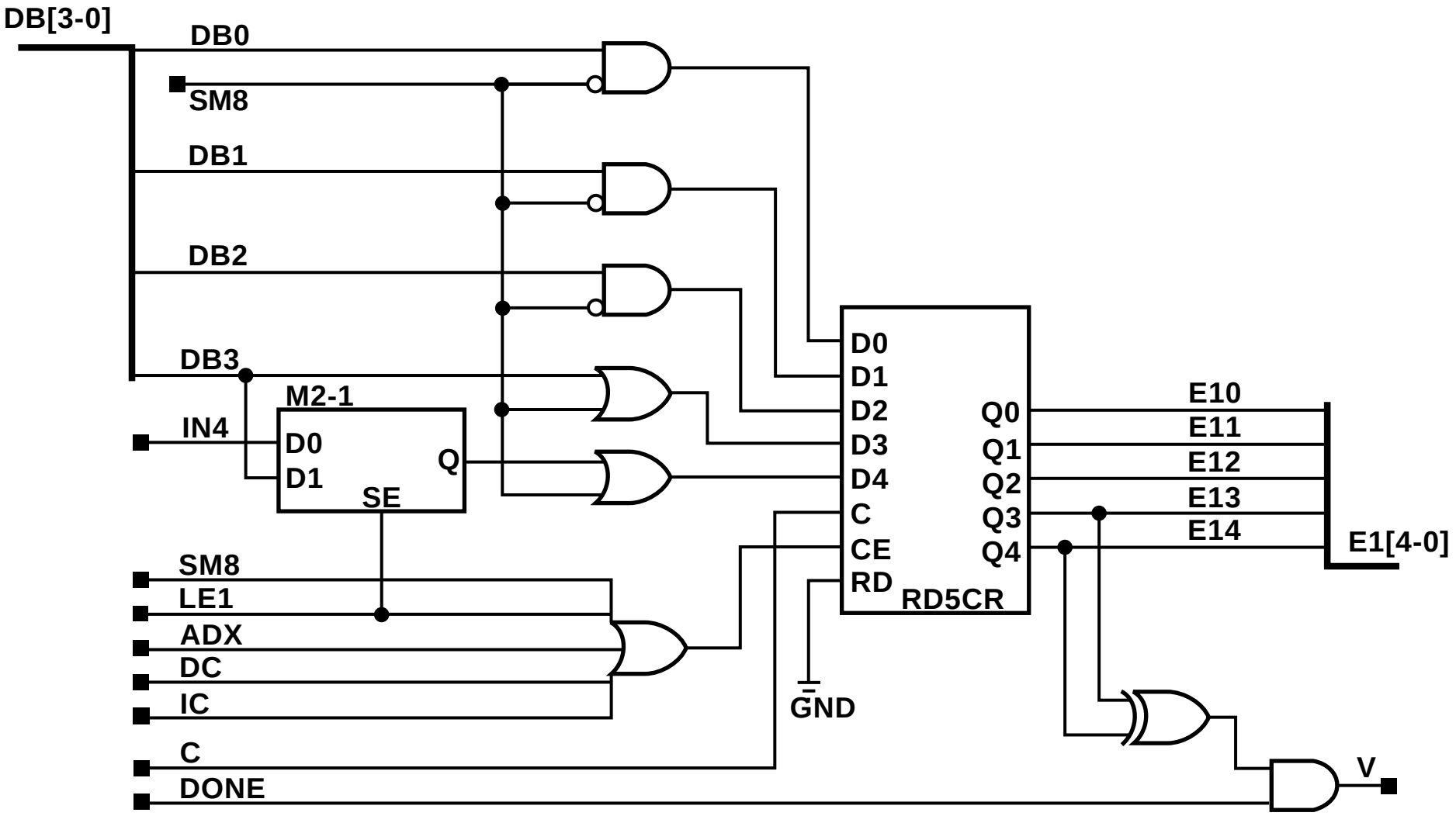
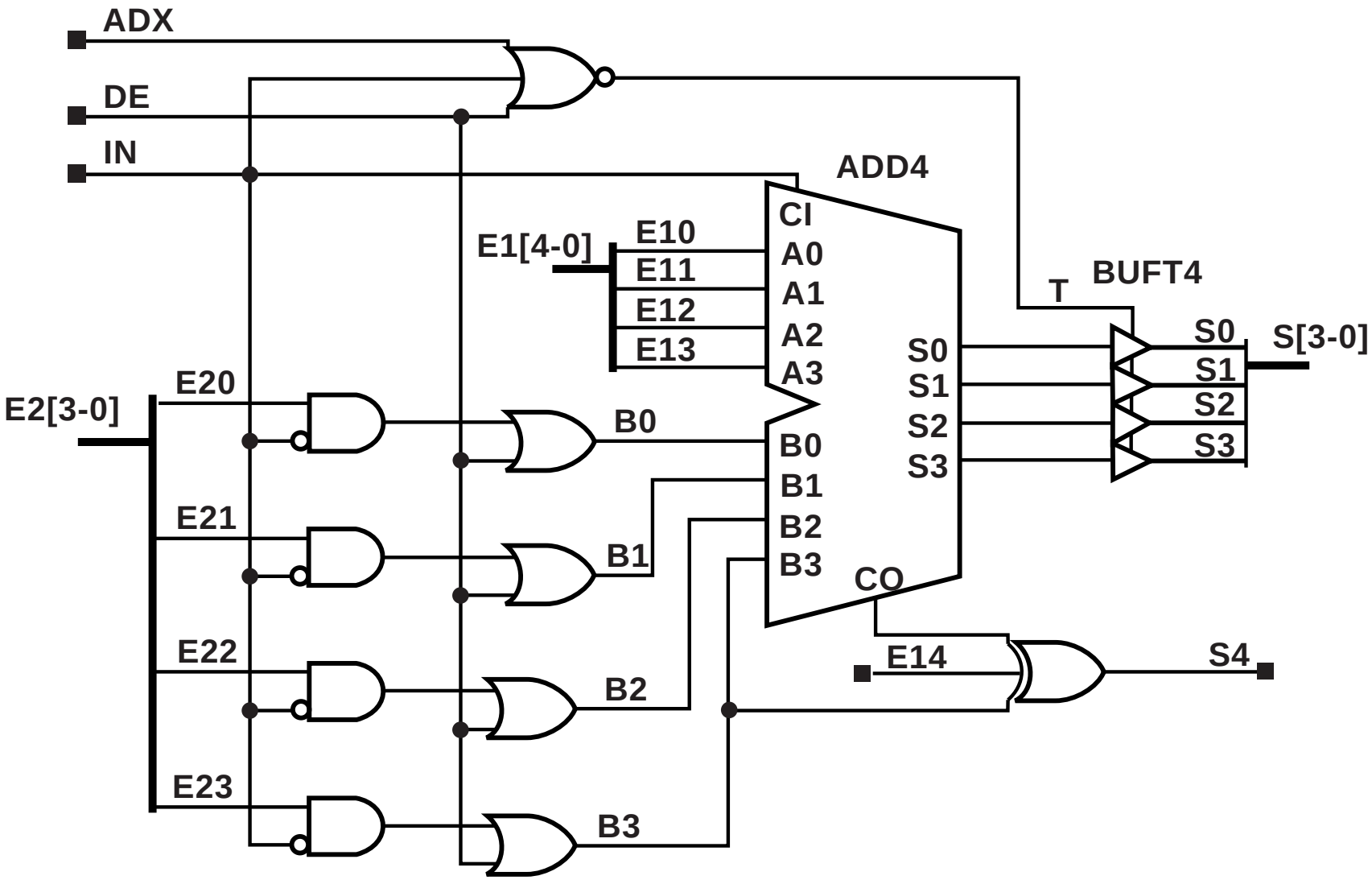


Figure 7-13 Exponent Adder



FLOATING POINT ADDITION *(from page 259)*

$$(F_1 \times 2^{E_1}) + (F_2 \times 2^{E_2}) = F \times 2^{E}$$

Example 1:

Add	$F_1 \times 2^{E_1} = 0.111 \times 2^5$ and $F_2 \times 2^{E_2} = 0.101 \times 2^3$
Unnormalize	$0.101 \times 2^3 = 0.0101 \times 2^4 = 0.00101 \times 2^5$
Add fractions	$(0.111 \times 2^5) + (0.00101 \times 2^5) = 01.00001 \times 2^5$
Fix Overflow	$F \times 2^E = 0.100001 \times 2^6$

Example 2: $(1.100 \times 2^{-2}) + (0.100 \times 2^{-1})$

$= (1.110 \times 2^{-1}) + (0.100 \times 2^{-1})$ (after shifting F_1)

$= 0.010 \times 2^{-1}$ (result of adding fractions is unnormalized)

$= 0.100 \times 2^{-2}$ (normalized by shifting left and subtracting one from exponent)

FLOATING POINT ADDITION SUMMARY *(from page 260)*

1) If exponents are not equal, shift the fraction with the smallest exponent right and add 1 to its exponent; repeat until the exponents are equal.

2) Add the fractions.

3) (a) If fraction overflow occurs, shift right and add 1 to the exponent to correct the overflow.
(b) If the fraction is unnormalized, shift left and subtract 1 from exponent until the fraction is normalized.
(c) If the fraction is 0, set the exponent to the appropriate value.

4) Check for exponent overflow.