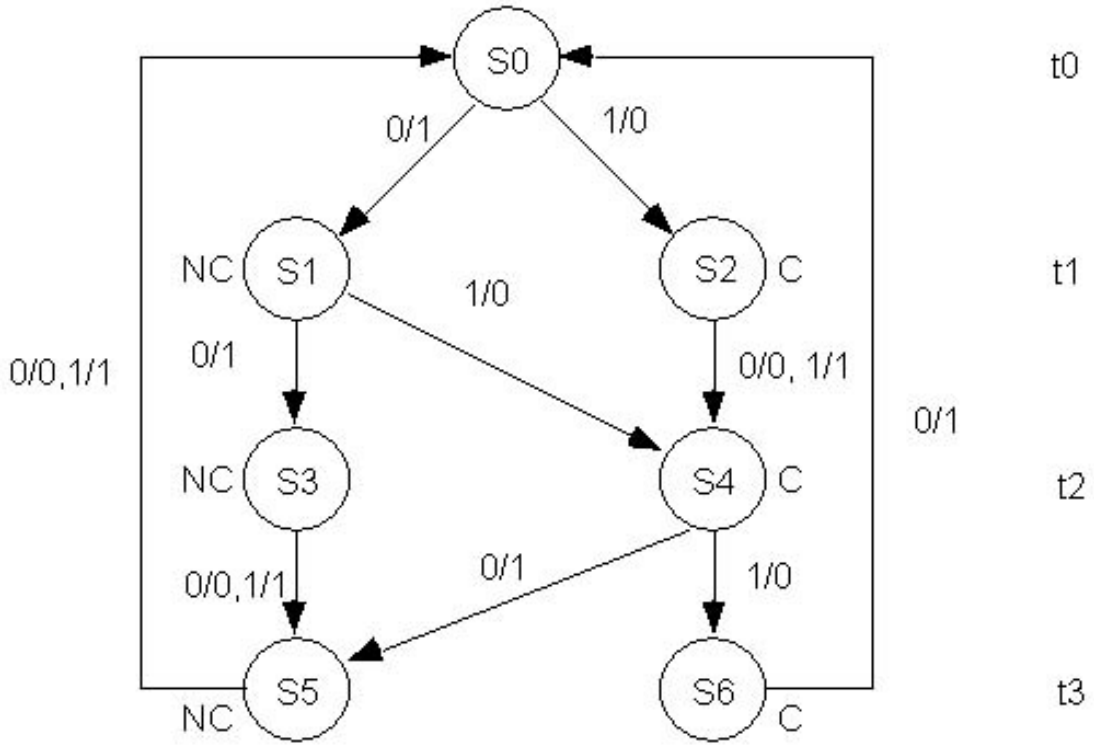
7.0 Sequential Machine VHDL Implementation

7.1 Design a BCD digit to excess-3-coded decimal digit.

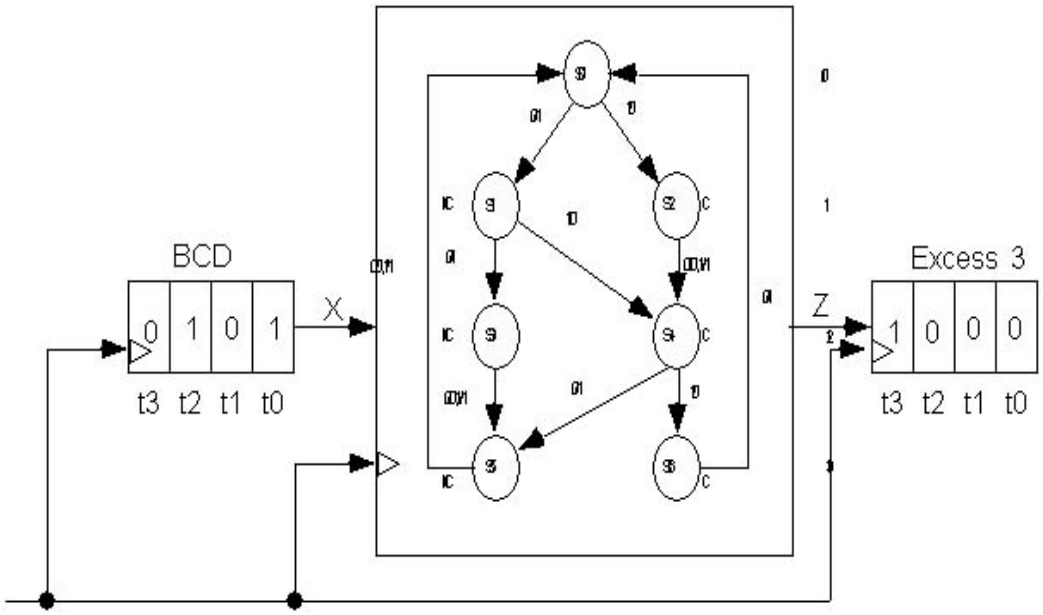
The excess-3 code is formed by adding 0011 to the BCD digit. The converter, however, will be implemented serially, starting with the least significant bit.

BCD				Excess-3			
t3	t2	t1	t0	t3	t2	t1	t0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0



At time to, we add 1 to the least significant bit, so if X=0, Z=1 (no carry), and if if X=1, Z=0 (carry=1). That is, if S0 is the reset state, S1 indicates no carry after the first addition, and S2 indicates a carry of 1. At t1 we add 1 to the next bit, so if there is no carry from the first addition (state S1), X=0 gives Z=0+1+0=1 and no carry (state S3), and X=1 gives Z=1+1+0=0 and a carry (state S4). If there is a carry from the first addition (state S2), then X=0 gives Z=0+1+1=0 and a carry (S4), and X=1 gives Z=1+1+1=1 and a carry (S4). At t2, 0 is added to X, and a transition to s5 (no carry) and S6 are determined in a similar manner. At t3, 0 is again added to X, and the network reset to S0. The input x=1 at s6 is not allowed because of overflow.

	NS		Z	
PS	X=0	X=1	X=0	X=1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	-	1	-




```
LIBRARY IEEE, ARITHMETIC;
USE IEEE.STD_LOGIC_1164.ALL;
USE ARITHMETIC.STD_LOGIC_ARITH.ALL;

ENTITY sm IS
        PORT(x, clk : IN BIT; z : OUT BIT);
END sm2;
```

```
ARCHITECTURE table OF sm IS
        TYPE s_type IS (s0, s1, s2, s3, s4, s5, s6);
        SIGNAL state, nxtstate : s_type;
BEGIN
        PROCESS(state,x)          --Combinational Network
        BEGIN
                CASE state IS
                        WHEN so =>
                                IF x='0' THEN z<='1' ; nxtstate<=s1;
                                ELSE z<='0';nxtstate<=s2; END IF;
                        WHEN s1 =>
                                IF x='0' THEN z<='1'; nxtstate<=s3;
                                ELSE z<='0';nxtstate<=s4; END IF;
                        WHEN s2 =>
                                IF x='0' THEN z<='0'; nxtstate<=s4;
                                ELSE z<='1';nxtstate<=s4; END IF;
                        WHEN s3 =>
                                IF x='0' THEN z<='0'; nxtstate<=s5;
                                ELSE z<='1';nxtstate<=s5; END IF;
                        WHEN s4 =>
                                IF x='0' THEN z<='1'; nxtstate<=s5;
                                ELSE z<='0';nxtstate<=s6; END IF;
                        WHEN s5 =>
                                IF x='0' THEN z<='0'; nxtstate<=s0;
                                ELSE z<='1';nxtstate<=s0; END IF;
                        WHEN s6 =>
                                IF x='0' THEN z<='1'; nxtstate<=s0;
                                ELSE z<='0';nxtstate<=s0; END IF;
                        WHEN others =>
                                z<='0';nxtstate<=s0;
                END CASE;
        END PROCESS;

        PROCESS(clk)              --state register
        BEGIN
                IF clk='1' AND clk'EVENT THEN
                        state <= nxtstate;
                END IF;
        END PROCESS;
END table;
```

7.2 STATE ASSIGNMENT

State Assignment Rules:

I. States that have the same next state (NS) for a given input should be given adjacent assignments.
II. States that are the next states of the same state should be given adjacent assignments.
III. States that have the same output for a given input should be given adjacent assignments.

I. (S1, S2) , (S3, S4), (S5, S6) ;In col X=1, S1 and S2 both have S4 as NS; In col X=0, S3 and S4 have S5 as NS, S5 and S6 have S0 as NS.

II. (S1, S2), (S3, S4), (S5, S6) ;S1 and S2 are NS of S0; S3 and S4 are NS of S1; S5 and S6 are NS of S4.

III. (S0, S1, S4, S6), (S2, S3, S5) ;In col X=0, S0, S1, S4, and S6 have Z=1; In col X=1, S2, S3, and S5 have Z=1

	Q1	
Q2Q3	0	1
00	S0	S1
01		S2
11	S5	S3
10	S6	S4

```
LIBRARY IEEE, ARITHMETIC;
USE IEEE.STD_LOGIC_1164.ALL;
USE ARITHMETIC.STD_LOGIC_ARITH.ALL;

ENTITY sm IS
        PORT(x, clk : IN BIT; z : OUT BIT);
END sm2;

ARCHITECTURE table2 OF sm IS
        SUBTYPE s_type IS INTEGER RANGE 0 TO 7;
        SIGNAL state, nxtstate : s_type;
        CONSTANT s0 : s_type := 0;      --state assignment=Q1Q2Q3
        CONSTANT s1 : s_type := 4;
        CONSTANT s2 : s_type := 5;
        CONSTANT s3 : s_type := 7;
        CONSTANT s4 : s_type := 6;
        CONSTANT s5 : s_type := 3;
        CONSTANT s6 : s_type := 2;
BEGIN
        PROCESS(state,x)        --Combinational Network
        BEGIN
                CASE state IS
                        WHEN so =>
                                IF x='0' THEN z<='1' ; nxtstate<=s1;
                                ELSE z<='0';nxtstate<=s2; END IF;
                        WHEN s1 =>
                                IF x='0' THEN z<='1'; nxtstate<=s3;
                                ELSE z<='0';nxtstate<=s4; END IF;
                        WHEN s2 =>
                                IF x='0' THEN z<='0'; nxtstate<=s4;
                                ELSE z<='1';nxtstate<=s4; END IF;
                        WHEN s3 =>
                                IF x='0' THEN z<='0'; nxtstate<=s5;
                                ELSE z<='1';nxtstate<=s5; END IF;
                        WHEN s4 =>
                                IF x='0' THEN z<='1'; nxtstate<=s5;
                                ELSE z<='0';nxtstate<=s6; END IF;
                        WHEN s5 =>
```

```
                        IF x='0' THEN z<='0'; nxtstate<=s0;
                        ELSE z<='1';nxtstate<=s0; END IF;
                WHEN s6 =>
                        IF x='0' THEN z<='1'; nxtstate<=s0;
                        ELSE z<='0';nxtstate<=s0; END IF;
                WHEN others =>
                        z<='0';nxtstate<=s0;
        END CASE;
    END PROCESS;

    PROCESS(clk)            --state register
    BEGIN
        IF clk='1' AND clk'EVENT THEN
            state <= nxtstate;
        END IF;
    END PROCESS;
END table2;
```

7.3 HARDWARE SYNTHESIS

	NS=D1D2D3		Z	
PS=Q1Q2Q3	X=0	X=1	X=0	X=1
S0=000	S1=100	S2=101	1	0
S1=100	S3=111	S4=110	1	0
S2=101	S4=110	S4=110	0	1
S3=111	S5=011	S5=011	0	1
S4=110	S5=011	S6=010	1	0
S5=011	S0=000	S0=000	0	1
S6=010	S0=000	-=XXX	1	-=X
S7=001	-=XXX	-=XXX	-=X	-=X

Karnaugh Map Minimization

XQ1

Q2Q3	00	01	11	10
00	1	1	1	1
01	X	1	1	X
11				
10				X

D1=Q2’

XQ1

Q2Q3	00	01	11	10
00		1	1	
01	X	1	1	x
11		1	1	
10		1	1	x

D2=Q1

XQ1

Q2Q3	00	01	11	10
00		1		1
01	X			X
11		1	1	

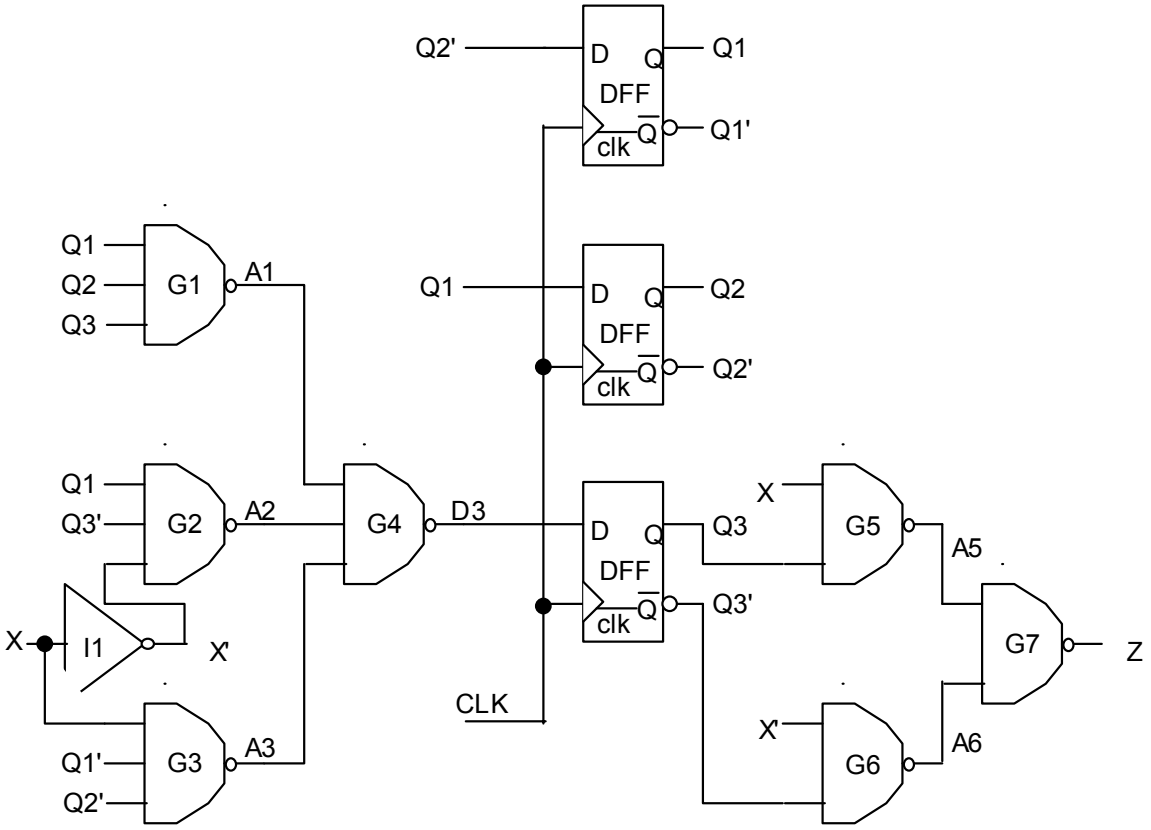
10		1		X

D3=Q1Q2Q3 +X'Q1Q3'+XQ1'Q2'

XQ1

Q2Q3	00	01	11	10
00	1	1		
01	X		1	X
11			1	1
10	1	1		X

Z=X'Q3'+XQ3



7.3.1 VHDL Behavioral Modelling of Minimized Hardware

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY sm IS
        PORT(x, clk : IN BIT; z : OUT BIT);
END sm1;
```

```
ARCHITECTURE equation OF sm IS
        SIGNAL q1, q2, q3 : BIT;
BEGIN
        PROCESS(clk)
        BEGIN
                IF (clk='1' AND clk'EVENT) THEN
                        q1 <=NOT q2;
                        q2 <= q1;
                        q3 <= (q1 AND q2 AND q3) OR ((NOT x) AND q1 AND (NOT q3)) OR
                                (x AND (NOT q1) AND (NOT q2));
                END IF;
        END PROCESS;
        z <= ((NOT x) AND (NOT q3)) OR (x AND q3)
END equation;
```

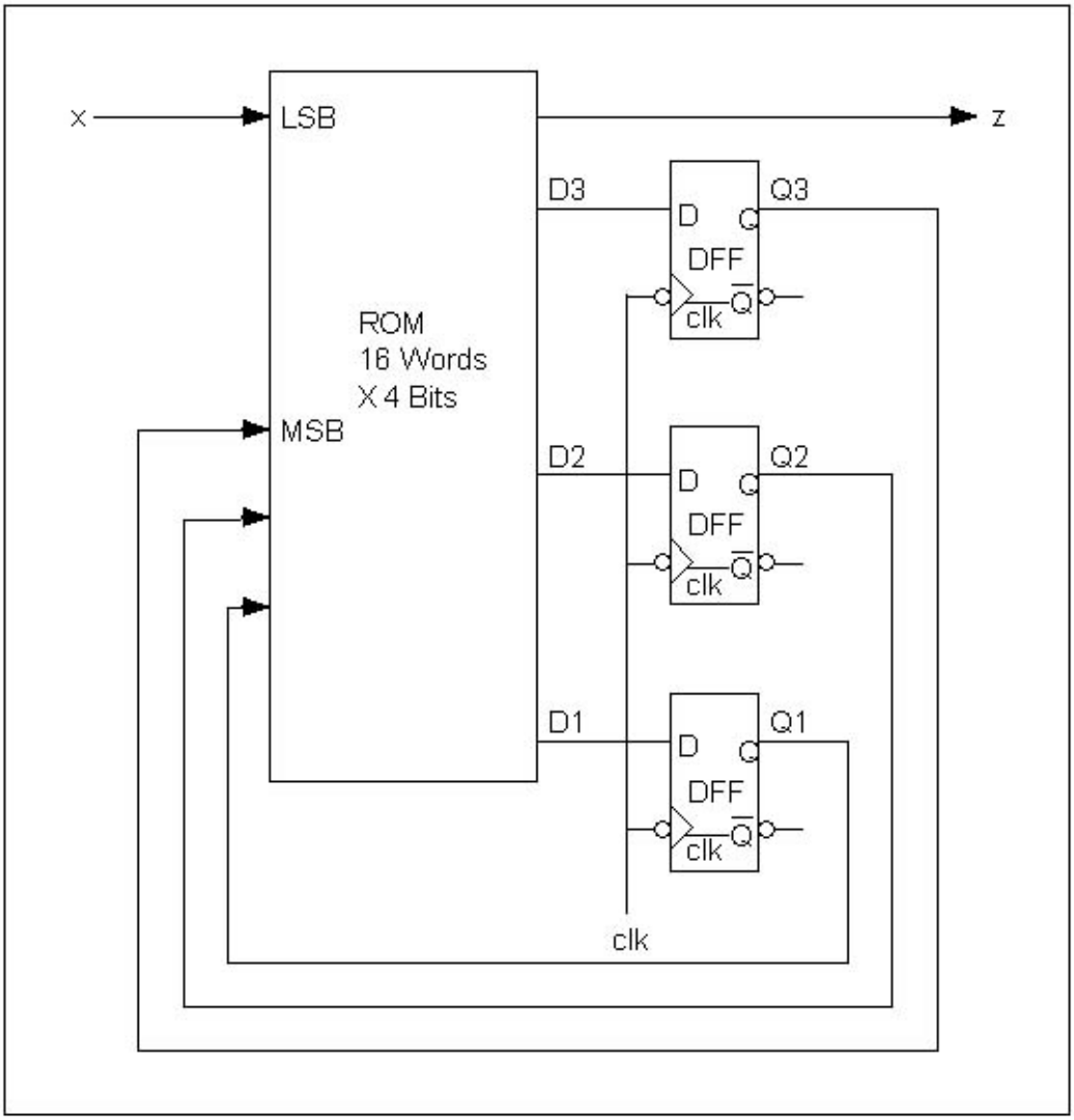
10.3.2 VHDL Structural Modelling of the Minimized Hardware

```
PACKAGE bit_pack IS

        COMPONENT dff
                PORT(d, clk: IN BIT; q: OUT BIT; qn: OUT BIT);
        END COMPONENT;

        COMPONENT nand2 IS
                PORT(i1, i2 :IN BIT; o:OUT BIT);
        END COMPONENT;

        COMPONENT nand3 IS
                PORT(i1, i2, i3:IN BIT; o:OUT BIT);
        END COMPNENT;

        COMPONENT inv IS
                PORT(i:IN BIT; o:OUT BIT);
        COMPONENT;
END bit_pack;

PACKAGE BODY bit_pack IS

        ENTITY dff IS
                PORT(d, clk:IN BIT; q:OUT BIT; qn:OUT BIT:='1');
        END dff;

        ARCHITECTURE arch_dff OF dff IS
                SIGNAL value: BIT;
        BEGIN
                PROCESS(clk)
                BEGIN
                        IF(clk='0' AND clk'EVENT) THEN
                                value <= d;
                        END IF;
                END PROCESS;
                q <= value;
                qn <= NOT value;
        END;
```

```
        ENTITY nand2 IS
                PORT(i1, i2:IN BIT; o:OUT BIT);
        END nand2;

        ARCHITECTURE arch_nand2 OF nand2 IS
        BEGIN
                o <= NOT (i1 AND i2);
        END arch_nand2;

        ENTITY nand3 IS
                PORT(i1, i2, i3:IN BIT; o:OUT BIT);
        END nand3;

        ARCHITECTURE arch_nand3 OF nand3 IS
        BEGIN
                o <= NOT (i1 AND i2 AND i3);
        END arch_nand3;

        ENTITY inv IS
                PORT(i:IN BIT; o:OUT BIT);
        END inv;

        ARCHITECTURE arch_inv OF inv IS
        BEGIN
                o <= NOT i;
        END arch_inv;

END bit_pack;

LIBRARY IEEE
USE IEEE.STD_LOGIC_1164.ALL:
USE WORK.bit_pack.ALL;

ENTITY sm IS
        PORT(x, clk: IN BIT; z:OUT BIT);
END sm3;

ARCHITECTURE structure OF sm IS
FOR I1 : inv USE ENTITY WORK.inv(arch_inv);
FOR G1, G2, G3, G4: nand3 USE ENTITY WORK.nand3(arch_nand3);
FOR G5, G6, G7: nand2 USE ENTITY WORK.nand2(arch_nand2);
FOR FF1, FF2, FF3: dff USE ENTITY WORK.dff(arch_dff);
        SIGNAL a1, a2, a3, a5, a6, d3: BIT:='0';
        SIGNAL q1, q2, q3 : BIT:='0';
        SIGNAL q1n, q2n, q3n xn :BIT :='1';
BEGIN
        I1: inv PORT MAP(x, xn);
        G1: nand3 PORT MAP(q1, q2, q3, a1);
        G2: nand3 PORT MAP(q1, q3n, xn, a2);
        G3: nand3 PORT MAP(x, q1n, q2n, a3);
        G4: nand3 PORT MAP(a1, a2, a3, d3);
        FF1: dff PORT MAP(q2n, clk, q1, q1n);
        FF2: dff PORT MAP(q1, clk, q2, q2n);
        FF3: dff PORT MAP(d3, clk, q3, q3n);
        G5: nand2 PORT MAP(x, q3, a5);
        G6: nand2 PORT MAP(xn, q3n, a6);
```

```
        G7: nand2 PORT MAP(a5, a6, z);
END structure;
```

7.4 Implementation Using ROM



State	Q1Q2Q3
S0	000
S1	100
S2	101
S3	111
S4	110
S5	011
S6	010
S7	001

MSB LSB

Q1	Q2	Q3	Q4=X	D1	D2	D3	Z
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

```
LIBRARY IEEE
USE IEEE.STD_LOGIC_1164.ALL:

ENTITY sm IS
        PORT (x, clk: IN BIT; z: OUT BIT);
END rom_sm

ARCHITECTURE rom OF sm IS
        SIGNAL Q, D : BIT_VECTOR(1 TO 3) :=”000”;
        TYPE rom IS ARRAY (0 TO 15) OF BIT_VECTOR( 3 DOWNTO 0);
        CONSTANT rom_tab: rom := (“1001”, “1010”, “0000”, “0000”,
                                   “0001”, “0000”, “0000”, “0001””
                                  “1111”, “1100”, “1100”, “1101””
                                  “0111”, “0100”, “0110”, “0111”);
BEGIN
        PROCESS (Q, x)
                romval: BIT_VECTOR(3 DOWNTO 0);
        BEGIN
                romval :=rom_tab(TO_INTEGER(‘0’&Q&x,0));
                D<=romval(3 DOWNTO 1);
                z <= romval(0);
        END PROCESS;

        PROCESS(clk)
        BEGIN
                IF clk=’1’ THEN Q<=D; END IF;
        END PROCESS;
END rom;
```

NOTE: ROM can not be implemented using VHDL code. ROM is assumed to be available as a separate design entity, available externally.

7.5 Binary Multiplier [1]

Design an arithmetic circuit that multiplies two fixed-point binary numbers in sign-magnitude representation. The product obtained from the multiplication of two binary numbers whose magnitudes consist of k bits each can be up to 2k bits long. The sign of each number occupies one additional bit.

Multiplication of two fixed-point binary numbers in sign-magnitude is done with paper and pencil by successive additions and shifting. This process is best illustrated with a numerical example. Let us multiply the two binary numbers 10111 and 10011:

```
    23          10111   multiplicand
        X
    19          10011   multiplier
 _________      _______
```

[1] Digital Logic and Computer Design, M. Mano, Ch10., pp435-443.

```
                  10111
                 10111
                00000
               00000
              10111
          +  _________
437       110110101   product
```

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from previous number. Finally, the numbers are added; their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is plus. If they are unlike, the sign of the product is minus.

When the above process is implemented in a digital machine, it is convenient to change the process slightly. First, instead of providing digital circuits to store and add simultaneously as many binary numbers as there are 1's in the multiplier, it is convenient to provide circuits for the summation of only two binary numbers and successively accumulate the partial products in a register. Seconds, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value. Assume the signs and numbers are assigned to the following registers:

B[7..0] = multiplicand
Bs = sign of multiplicand
Q[7..0] = multiplier (initially), and LSByte of partial product
Qs = sign of multiplier
A[7..0] = MSByte of partial product
As = sign of the product
E = stores the output carry
P = counter, initially set to the number of bits in the multiplier (=8).
Pz=1, if P=0
=0, if P<>0

The previous example is repeated with the given registers:

B[4..0]=10111

Event	Q0(LSB of Q)	E	A[4..0]	Q[4..0]	P[3..0]
1	INIT	0	00000	10011	0101
2	1	0	00000+10111=10111	10011	0100
		0	01011	11001	0100
3	1	1	01011+10111=00010	11001	0011
		0	10001	01100	0011
4	0	0	10001	01100	0010
		0	01000	10110	0010
5	0	0	01000	10110	0001
		0	00100	01011	0001
6	1	0	00100+10111=11011	01011	0000
		0	01101	10101	0000

Algorithm

Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs. The multiplication process is initiated when the start signal, St=1. The two signs are compared by means of an Xor gate. If the two signs are alike, the Xor operation produces a 0 which is transferred to As to give a plus for the product. If the signs are unlike, a 1 is transferred to As to give a negative sign for the product. Registers A and E are cleared and the sequence counter P is set to a binary number k, which is equal to the number of bits in the multiplier.

Next we enter a loop that keeps forming the partial products. The multiplier bit in Q0 is checked, and if it is equal to 1, the multiplicand in B is added to the present partial product in A. Any carry from the addition is transferred to E. The partial product in A is left unchanged if Q0=0. The P counter is decremented by 1 regardless of the value of Q0. Register A, Q, and E are then shifted once to the right to obtain a new partial product. This shift operation is symbolized in the flowchart in compact form with the statement:

AQ <= shr EAQ, E<=0

EAQ is a composite register made up of registers E, A, and Q.

The value in the P counter is checked after the formation of each partial product. If P is not 0, the process is repeated and new partial product is formed. The process stops after the kth partial product when P=0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in A and Q, with A holding the most significant bits and Q holding the least significant bits. The sign of the product is in As.

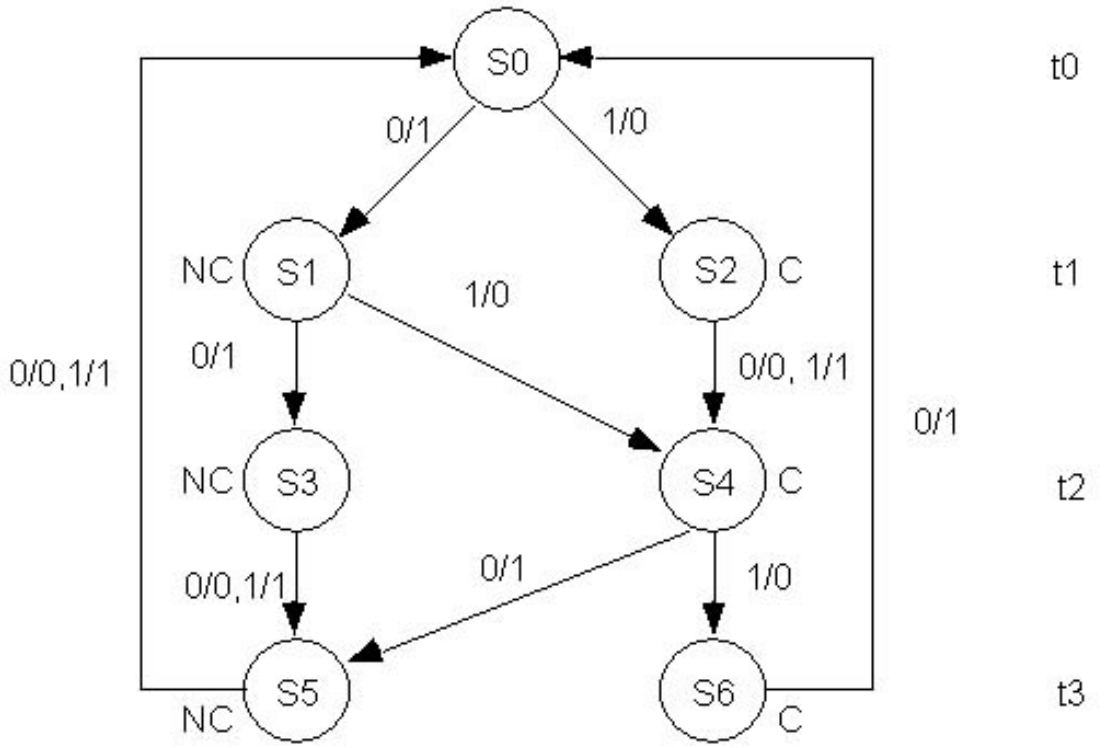


Figure 1. Binary multiplier flowchart

The conversion from a flowchart to a state diagram is not unique. The flowchart may be considered a preliminary formulation of the algorithm. The control state diagram, together with a list of microoperations, is more precise since it takes into consideration the hardware constraints of the system.

The control has four states, and the register-transfer operations for each state are listed next to it. Control stays in the initial state T0 until St becomes 1. It then goes to state T1 to initialize register A,E, and P and to form the sign of the product. Control then goes to state T2. In this state, register P is decremented and the contents of B are added to A if Q0=1; otherwise, A is left unchanged. The two control functions at time T2 are:

Q0T2: A<=A+B, E<=Cout
T2: P<=P-1

The second statement is always executed when T2=1. The first statement is executed at time T2 only if Q0=1. Thus, a status variable Q1 can be included with a timing variable to form a control function. Note it is convenient to decrement P at state T2 so that its new value can be checked at state T3.

Control goes to T3 after T2. At state T3, the composite register EAQ is shifted to the right and the contents of P are checked for zeros. The binary variable Pz is 1 if the P register contains all 0's ; otherwise Pz is 0. If Pz=1, the operation is terminated and control goes to the initial state. If Pz=0, control goes to state T2 to form a new partial product. Note that P refers to the contents of the register, whereas Pz is a binary variable.

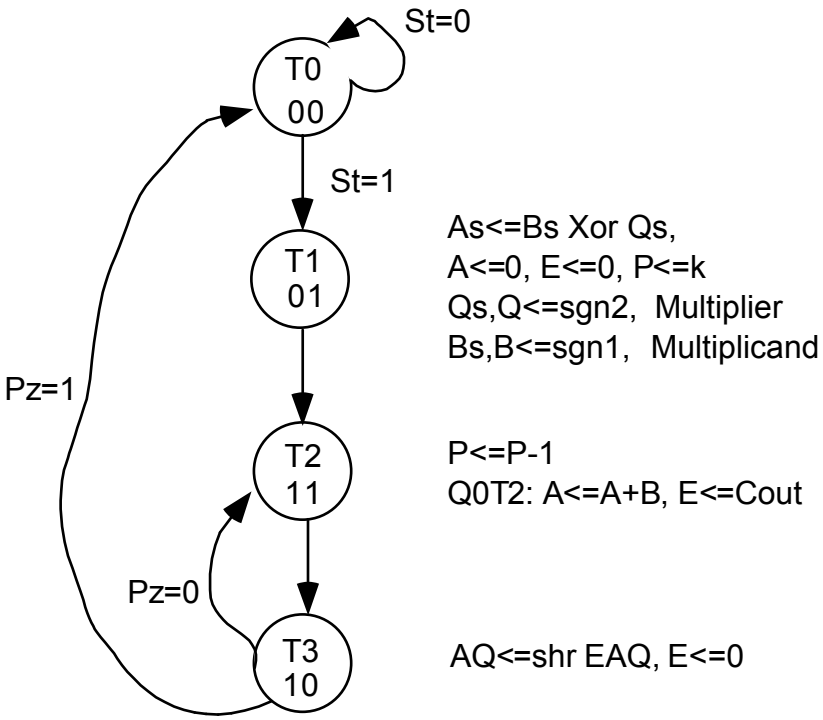


Figure 2. Binary multiplier state diagram

```
LIBRARY IEEE, ARITHMETIC;
USE IEEE.STD_LOGIC_1164.ALL;
USE ARITHMETIC.STD_LOGIC_ARITH.ALL;

ENTITY mul_ctr IS
        PORT(St, Pz, clk : IN BIT; T: OUT BIT_VECTOR(0 TO 3));
END mul_ctr;

ARCHITECTURE table OF mul_ctr IS
        SUBTYPE s_type IS INTEGER RANGE 0 TO 3;
        SIGNAL state, nxtstate : s_type;
        CONSTANT s0 : s_type := 0;        --state assignment=Q1Q2
        CONSTANT s1 : s_type := 1;
        CONSTANT s2 : s_type := 3;
        CONSTANT s3 : s_type := 2;
BEGIN
        PROCESS(state, St, Pz)    --Combinational Network
        BEGIN
                CASE state IS
                        WHEN s0 =>
                                IF St='0' THEN nxtstate<=s0;
                                ELSE nxtstate<=s1; END IF;
                        WHEN s1 =>
                                nxtstate <=s2;
                        WHEN s2 =>
                                nxtstate <=s3;
                        WHEN s3 =>
                                IF Pz='0' THEN nxtstate<=s2;
                                ELSE nxtstate<=s0; END IF;
                        WHEN others =>
                                nxtstate<=s0;
                END CASE;
        END PROCESS;

        PROCESS(clk)              --state register
        BEGIN
                IF clk='1' AND clk'EVENT THEN
                        state <= nxtstate;
                END IF;
        END PROCESS;

        PROCESS(state)            --output decoder
        BEGIN
                CASE state IS
                        WHEN s0 => T<="1000";
                        WHEN s1 => T <="0100";
                        WHEN s2 => T <="0010";
                        WHEN s3 => T <="0001";
                        WHEN others => T<="1000";
                END CASE;
        END PROCESS;
```

END table;

NS=D1D2D3

	PS=Q1Q2	Input=St Pz	NS=D1D2
T0	00	0X	00
T0	00	1X	01
T1	01	XX	11
T2	11	XX	10
T3	10	X0	11
T3	10	X1	00

StPz

Q1Q2	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

D1=Q2 + Q1Q2'Pz'

StPz

Q1Q2	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	0	0	0	0
10	1	0	0	1

D2=Q1'Q2 + Q1'St +Q1Q2'Pz'

7.6 Binary Division

Two's Complement[2]

The 2's complement of a four-bit number A is 10000 – A. But 10000 is five bits, too big for a four-bit system. The five bits 10000 can be written as 1111 + 1. Now the 2's complement conversion can be stated as 2C(A)=1111+1-A. Since, in four-bit words 1111-A = A' is the one's complement of A. The two's complement formula can be rewritten as 2C(A) = A'+1. In this notation 2C(2C(A))=A.

[2] Digital Design from Zero to One, J. D. Daniels, Ch10, pp 532-533.

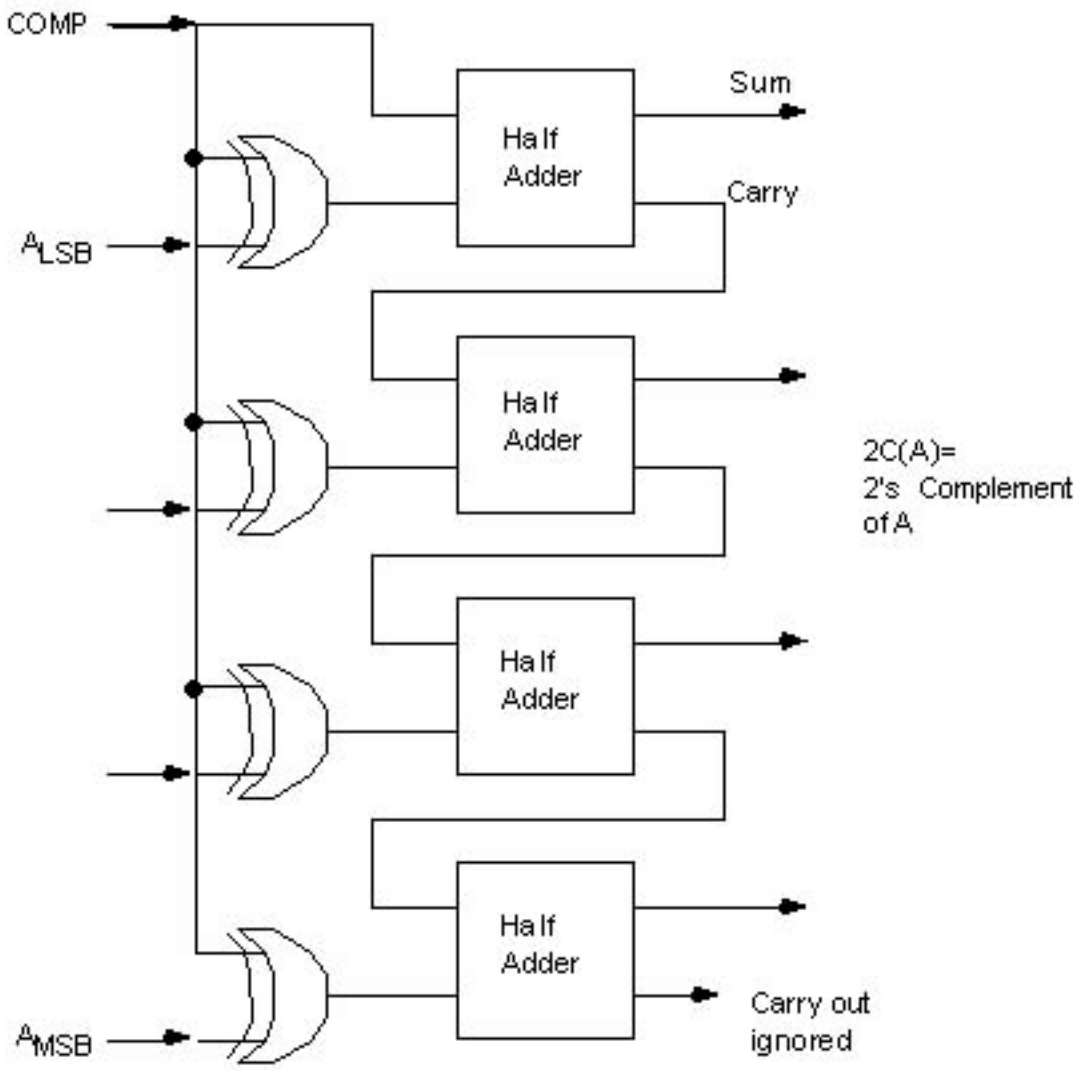


Figure 1. Two's Complement Implementation

Figure 1 shows a combinatorial circuit that will either pass A through unchanged or take the 2's complement of A. If the control line COMP is HI, the output of the circuit is the 2's complement of the number A on the data lines. The XOR gates invert the bits of A when COMP is HI. When COMP is HI, 1 is added to the input of the half adder. Therefore, when COMP is HI, the output is 2C(A). If COMP is LO, the input to the half adder is not inverted and 0 is added to the input. The truth table is shown below:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0

1	1	0	1

Division Algorithm

The division is based on repeated subtraction. That is, the denominator is repeatedly subtracted from the numerator until the difference is less than the denominator. The number of times the denominator can be subtracted from the numerator is the quotient, and the difference which is less than the denominator is the remainder.

The division algorithm of two unsigned numbers is shown in Figure 2. At the initial state INIT, the D and N registers are initialized to the denominator and numerator respectively; the multiplexer select line is cleared, s<=0, to select the numerator N. Upon receiving the start signal ST, the initial remainder R is set to N, by latching the accumulator ACC, and the quotient Q is initialized to zero by clearing the counter C<=0. Then the multiplexer select input is set to 1, s<=1, to select the current remainder R in the subsequent repeated subtraction operation. The denominator D is compared to the remainder R, if the D<=R the denominator is subtracted from the current remainder, R=ACC<=R-D and the counter is incremented by one, Q=C<=C+1. This operation is repeated until D>R, the operation stop and go back to the INIT state, the accumulator ACC contains the remainder R and the counter C contains the quotient Q.

To illustrate this algorithm, the division problem N/D=22/5 will be used as an example.

Condition	R	Q
INIT	22	0
D>R =5>22 N	R=22-5=17	1
D>R =5>17 N	R=17-5=12	2
D>R = 5>12 N	R=12-5=7	3
D>R = 5>7 N	R=7-5=2	4
D>R = 5>2 Y	STOP(INIT)	STOP(INIT)

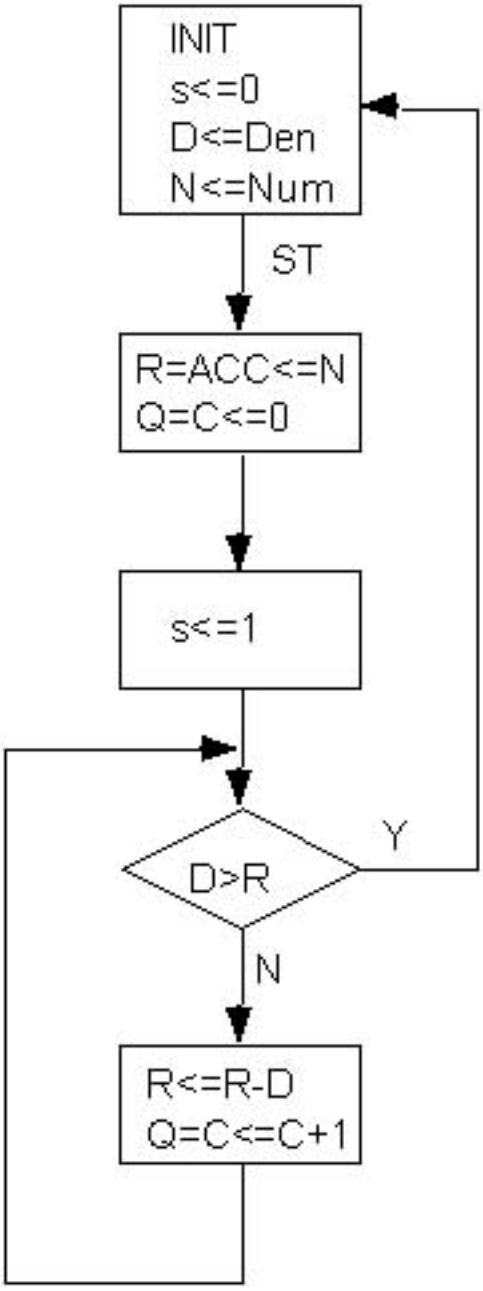


Figure 2. The division algorithm flowchart.

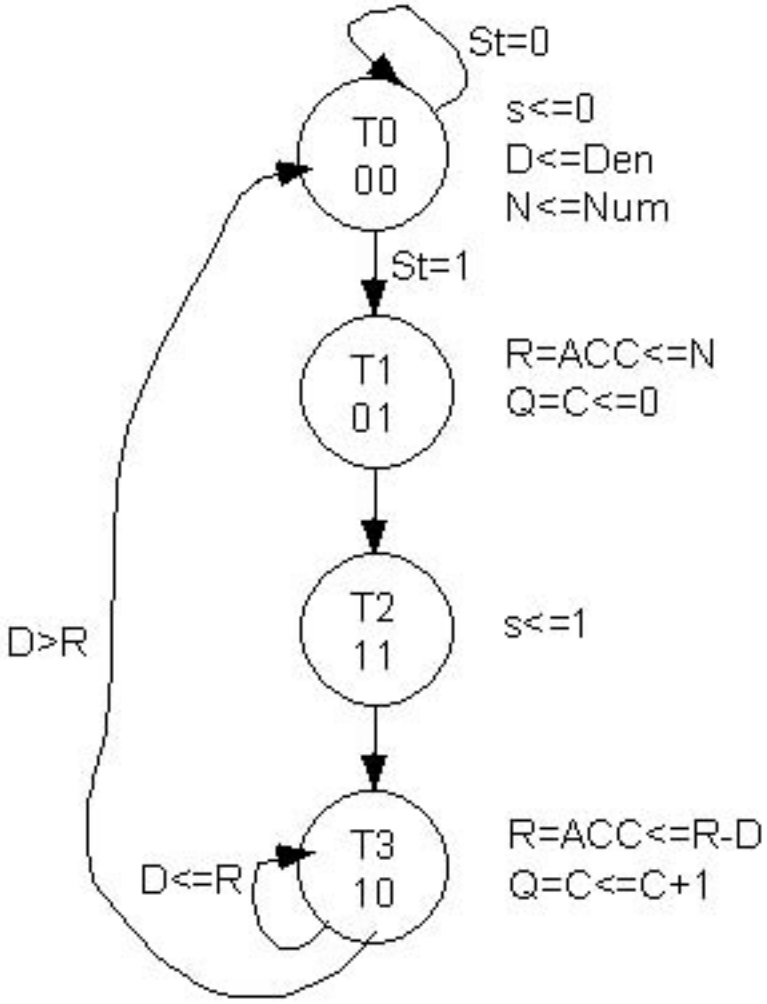


Figure 3. Division algorithm state diagram.

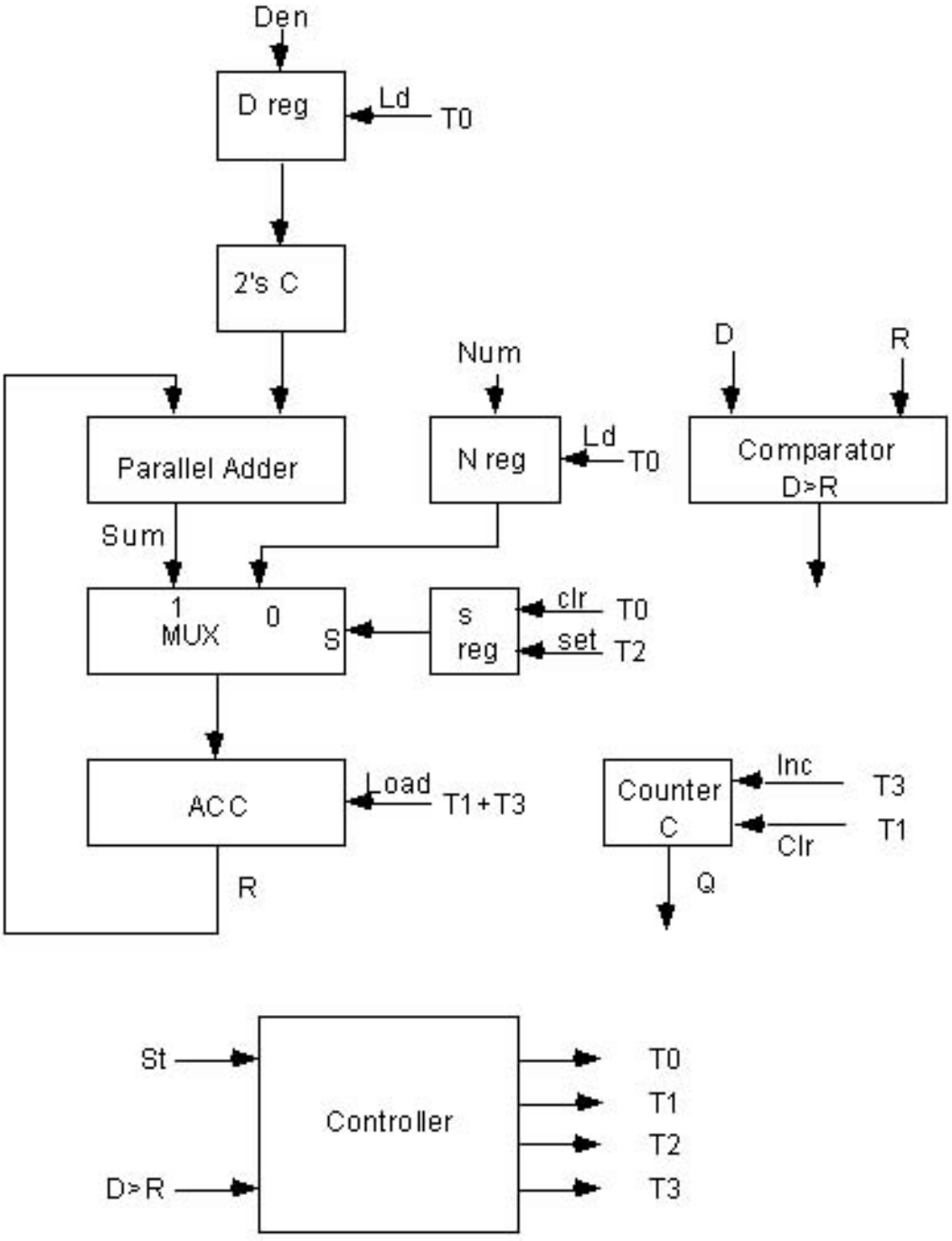


Figure 4. Division Algorithm Structural Implementation.

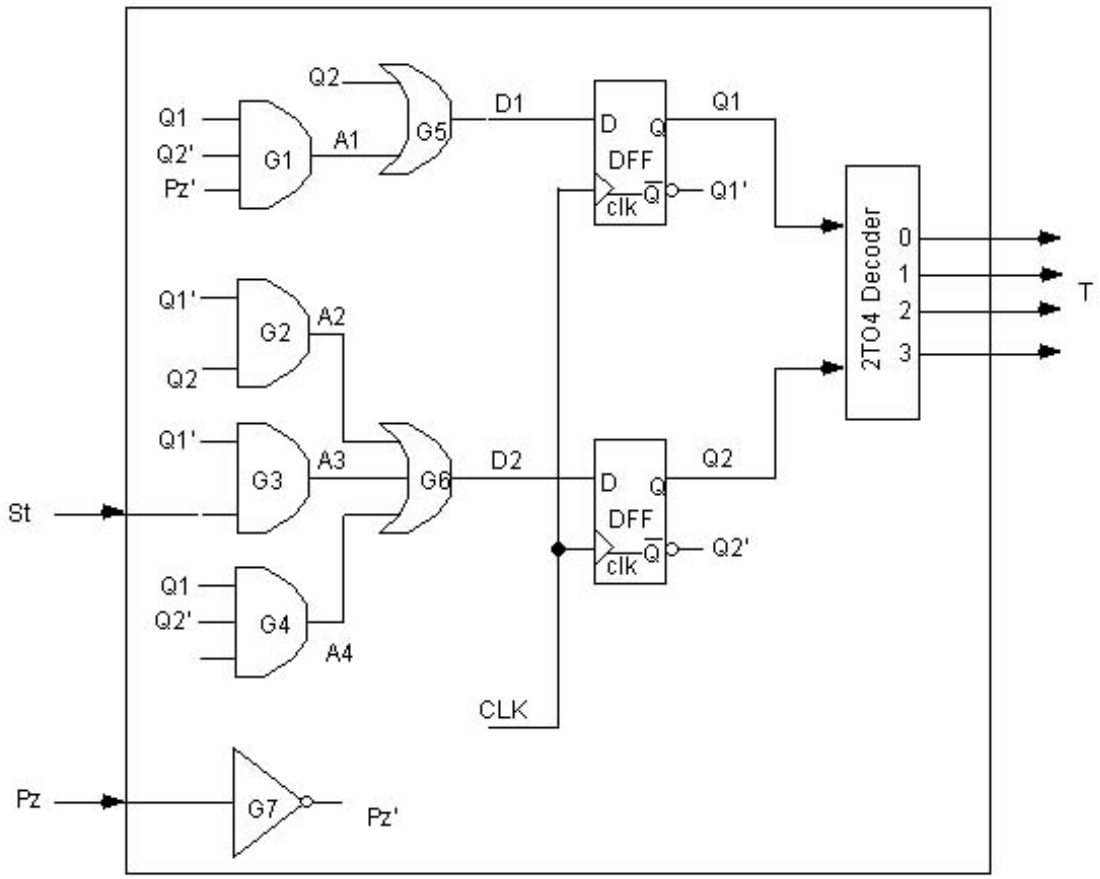
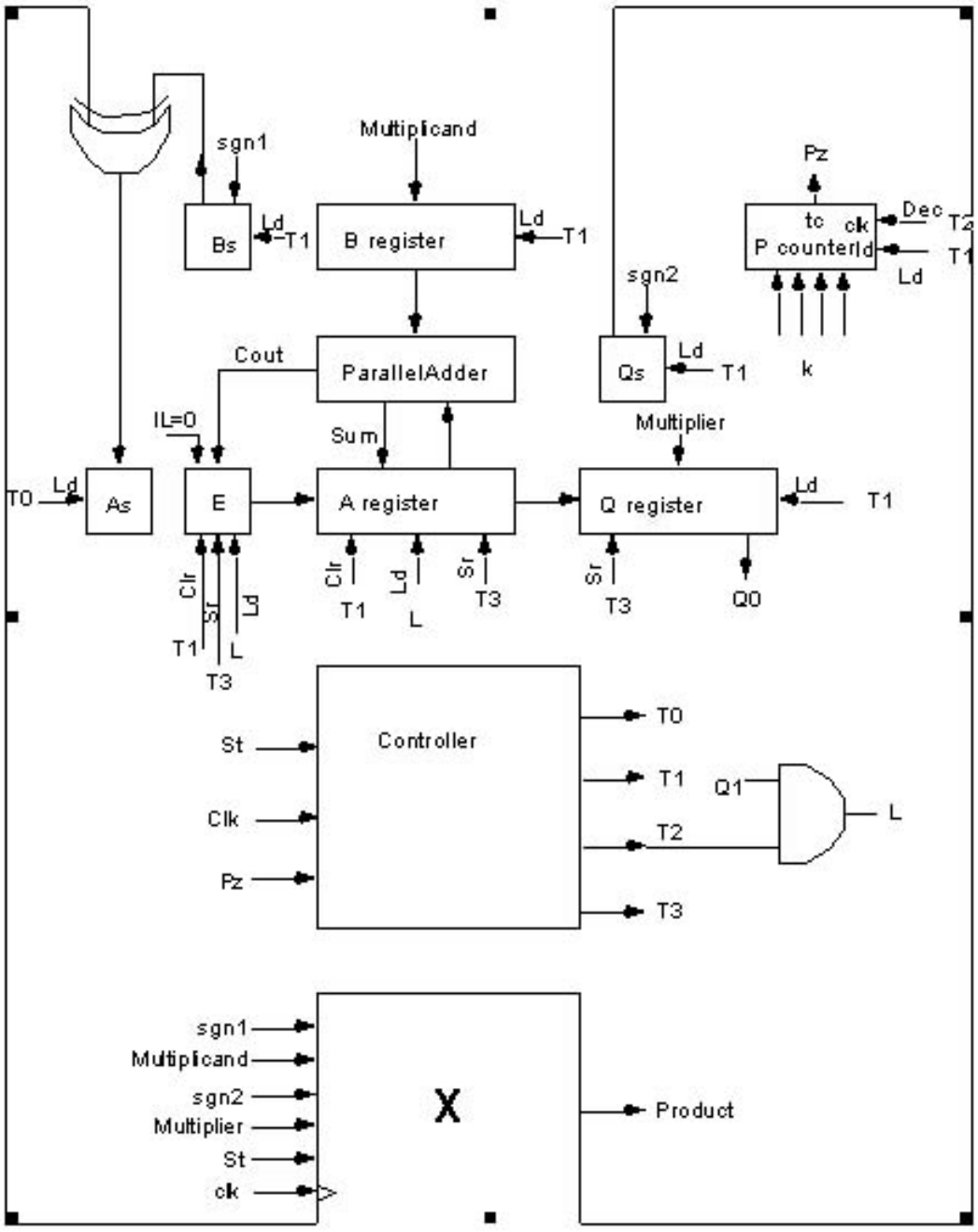


Figure 3. Binary multiplier controller schematic diagram



NOTE: T0 is used to latch As.

Figure 4. Binary multiplier schematic diagram