Binary Coded Decimal (BCD) Division by Shift and Subtract

07 September 2006 Application Note DKAN0003A

Introduction

This application note discusses a technique to divide binary coded decimal (BCD) numbers in hardware. BCD division is easily achievable by repeatedly adding the divisor to itself and counting the iterations required for the sum to equal the dividend. The resulting count yields the quotient, while the difference between the sum and dividend provides the remainder. However, this technique crawls when the dividend is orders of magnitude larger than the divisor.

BCD division by shift and subtract offers an alternative method appropriate for this situation. Based on long division, it is more difficult to implement, but it tremendously improves the computation time needed to divide a small number into a large one. This makes it suitable for applications requiring regular calculations of this sort (e.g. averaging small sets of large numbers).

Besides a discussion of the concepts, this document provides an open source VHDL example implementation of BCD division by shift and subtract, applicable to CPLDs and FPGAs. This example divides 2-digit (or less) divisors into 6-digit (or less) dividends. It is easily modified to fit other dividend/divisor size requirements.

Application

Concept

As alluded to above, BCD division by shift and subtract mimics long division. The flow chart in Figure 1 describes the procedure. As in long division, left-align the divisor below the dividend. Compare the divisor to the dividend digits directly above it to determine the most significant quotient digit. Once the divisor is greater than the dividend digits above it, the divisor is shifted to the right one digit. Comparing the divisor to the new dividend digits above produces the next quotient digit. This process repeats until the divisor is larger than the remaining dividend. The quotient is complete, and the residual dividend equals the remainder.

To help illustrate the concept, the procedure is applied in the example below. This technique uses only addition, subtraction, comparisons, and shifting, making it straightforward to implement in programmable logic.

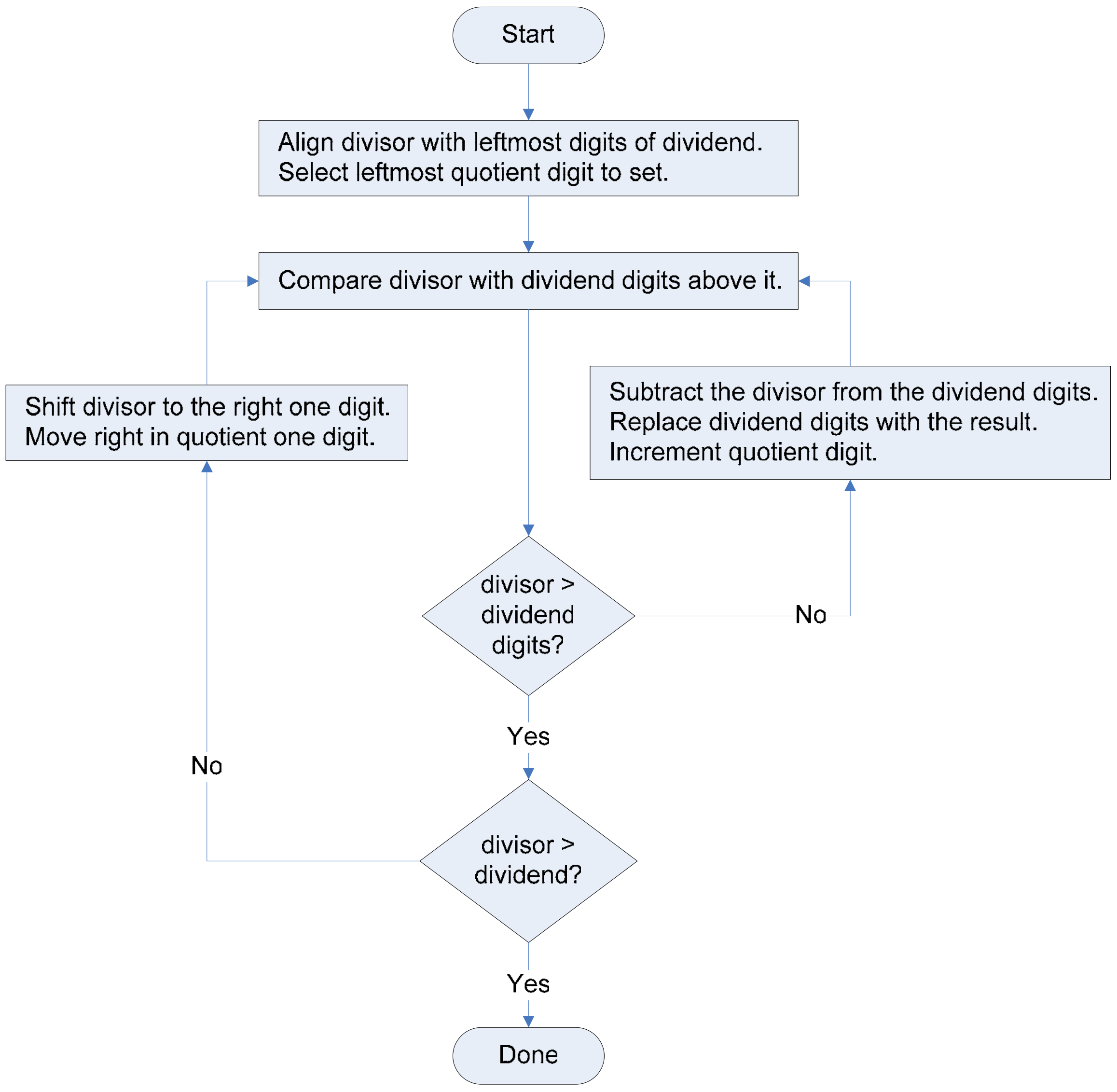
Start
Align divisor with leftmost digits of dividend.
Select leftmost quotient digit to set.
Compare divisor with dividend digits above it.
Shift divisor to the right one digit.
Move right in quotient one digit.
Subtract the divisor from the dividend digits.
Replace dividend digits with the result.
Increment quotient digit.
divisor > dividend digits?
No
Yes
No
divisor > dividend?
Yes
Done

Figure 1. BCD Division by Shift and Subtract Flow Chart

Example

Divide 0001 0000 0000 0111 by 0101 ($1007_{10} \div 5_{10}$).

```
            Q4          Q3          Q2          Q1          Select leftmost quotient digit to set.
0 1 0 1 | 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
          0 1 0 1                                           Align divisor with leftmost digits of dividend.
```

Divisor > dividend digits above it? Yes.	(0101 > 0001)
Divisor > dividend? No.	(0101 < 0001 0000 0000 0111)

```
        0 0 0 0    Q3      Q2      Q1        Move quotient right one digit.
0 1 0 1|0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                0 1 0 1                      Shift divisor one digit to the right.
```

Divisor > dividend digits above it? No. (0101 < 0001 0000)

```
        0 0 0 0 0 0 0 1    Q2      Q1        Increment quotient digit.
0 1 0 1|0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                0 1 0 1                      Subtract the divisor from the dividend digits.
        ---------------
        0 0 0 0 0 1 0 1                      Replace dividend digits with the result.
```

Divisor > dividend digits above it? No. (0101 = 0000 0101)

```
        0 0 0 0 0 0 1 0    Q2      Q1        Increment quotient digit.
0 1 0 1|0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                0 1 0 1
        ---------------
        0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1      New dividend.
                0 1 0 1                      Subtract the divisor from the dividend digits.
        ---------------
        0 0 0 0 0 0 0 0                      Replace dividend digits with the result.
```

Divisor > dividend digits above it? Yes. (0101 > 0000 0000)
Divisor > dividend? No. (0101 < 0000 0000 000 0111)

```
        0 0 0 0 0 0 1 0    Q2      Q1        Move quotient to the right one digit.
0 1 0 1|0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                0 1 0 1
        ---------------
        0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1
                0 1 0 1
        ---------------
        0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1      New dividend.
                        0 1 0 1              Shift divisor one digit to the right.
```

Divisor > dividend digits above it? Yes. (0101 > 0000 0000 0000)
Divisor > dividend? No. (0101 < 0000 0000 000 0111)

```
        0 0 0 0 0 0 1 0 0 0 0 0    Q1        Move quotient to the right one digit.
0 1 0 1|0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                0 1 0 1
        ---------------
        0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1
                0 1 0 1
        ---------------
        0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1      New dividend.
                                0 1 0 1      Shift divisor one digit to the right.
```

Divisor > dividend digits above it? No. (0101 < 0000 0000 0000 0111)

```
          0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1   Increment quotient digit.
0 1 0 1 | 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1
                  0 1 0 1
          --------------------------------
          0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1
                  0 1 0 1
          --------------------------------
          0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
                                  0 1 0 1   Subtract the divisor from the dividend digits.
          --------------------------------
                                  0 0 1 0   Replace dividend digits with the result.
```

Divisor > dividend digits above it? Yes. (0101 > 0000 0000 0000 0010)
Divisor > dividend? Yes. (0101 > 0000 0000 000 0010)

Done.

Quotient = 0000 0010 0000 0001.
Remainder = 0010.

Sanity check: $1007_{10} \div 5_{10} = 201_{10}$ remainder 2_{10}.

Performance Considerations

BCD division by shift and subtract has its place. It offers an enormous performance improvement over the generic BCD division by summing (described in the introduction) for cases with dividend >> divisor. Given comparable dividend and divisor values, BCD division by shift and subtract still successfully executes, but takes somewhat longer than the other method. Certainly, BCD division by shift and subtract requires more complex logic to implement properly.

Appendix A contains open source VHDL code that divides 2-digit (or less) divisors into 6-digit (or less) dividends. In this specific case, BCD division by shift and subtract will execute up to 8000 times faster than BCD division by summing (best case) and about 10-15 times slower in a worst case scenario. Given a random set of numbers, BCD division by shift and subtract executes thousands of times faster than BCD division by summing.

Conclusion

The concepts described in this application note provide a means of constructing BCD division logic without the inconveniencies of converting between BCD and Binary. It offers substantial performance improvements over generic BCD division by summing in many cases.

Appendix A: Example Source Code

The following open source VHDL code divides 2-digit (or less) divisors into 6-digit (or less) dividends. The code defines the Control Logic block in Figure 2. The BCD Adder block in Figure 2 represents a 3-digit BCD adder. For more information on BCD adder construction, see the background material included in Digi-Key Application Note DKAN0002A.

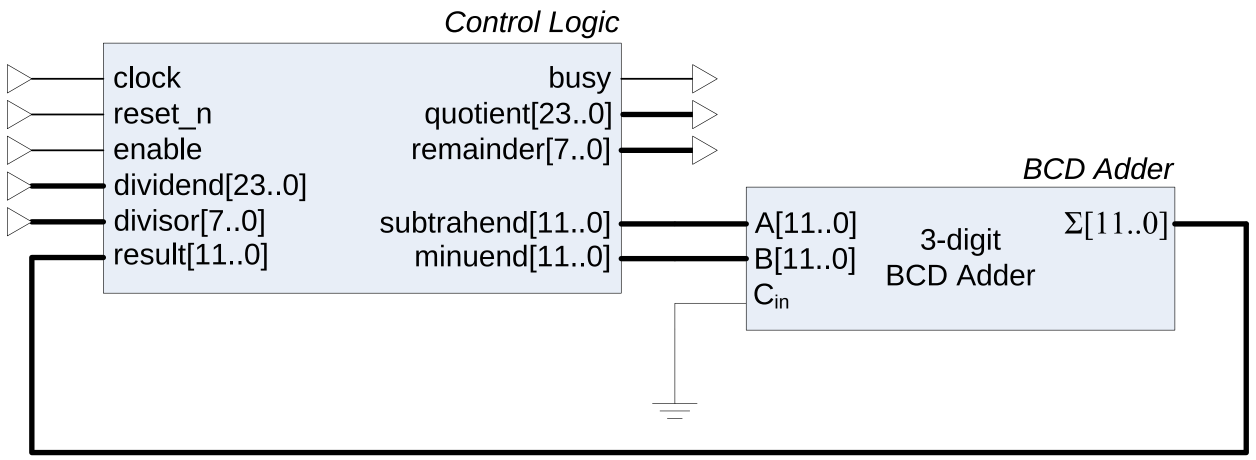

Figure 2. BCD Division Example Circuit

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;
USE   ieee.std_logic_arith.ALL;

ENTITY division_controller IS
    PORT(
        clock         :  IN STD_LOGIC;                          --system clock
        reset_n       :  IN STD_LOGIC;                          --resets on logic low
        enable        :  IN STD_LOGIC;                          --signal high for division to start
        busy          :  OUT   STD_LOGIC;                       --goes high when busy, low when done
        divisor       :  IN STD_LOGIC_VECTOR(7 DOWNTO 0);       --2 digit divisor
        dividend      :  IN STD_LOGIC_VECTOR(23 DOWNTO 0);      --6 digit dividend
        quotient      :  OUT   STD_LOGIC_VECTOR(23 DOWNTO 0);--6 digit quotient result
        remainder     :  OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);  --2 digit remainder result
        subtrahend    :  OUT   STD_LOGIC_VECTOR(11 DOWNTO 0);--output to bcd adder
        minuend       :  OUT   STD_LOGIC_VECTOR(11 DOWNTO 0);--output to bcd adder
        result        :  IN STD_LOGIC_VECTOR(11 DOWNTO 0));     --input from bcd adder
END division_controller;

ARCHITECTURE controller OF division_controller IS
    TYPE CONTROL IS(ready,shift,compare,subtract);
    SIGNAL state      :  CONTROL;
    SIGNAL q          :  STD_LOGIC_VECTOR(1 DOWNTO 0);      --buffer for reset_n
    SIGNAL divis      :  STD_LOGIC_VECTOR(7 DOWNTO 0);      --buffer for divisor
    SIGNAL divid      :  STD_LOGIC_VECTOR(23 DOWNTO 0);     --buffer for dividend
    SIGNAL divid_part :  STD_LOGIC_VECTOR(11 DOWNTO 0);     --dividend digits under fire
    SIGNAL quot       :  STD_LOGIC_VECTOR(23 DOWNTO 0);     --buffer for quotient result
BEGIN

    PROCESS(clock)
        VARIABLE q_index :   INTEGER RANGE 0 TO 6;                --index of quotient digit under fire
    BEGIN

        IF(clock'EVENT and clock = '1') THEN
            q(1) <= q(0);
            q(0) <= reset_n;
            CASE state IS
```

```
--ready and waiting for new division process to start
WHEN ready =>
    busy <= '0';
    quotient <= "000000000000000000000000";
    remainder <= "00000000";
    subtrahend <= "000000000000";
    minuend <= "000000000000";
    q_index := 0;
    quot <= "000000000000000000000000";
    IF(enable = '1') THEN
        --get user inputs
        divis <= divisor;
        divid <= dividend;
        --find ten's complement of the divisior for future subtractions
        subtrahend <= "1001" & ("1001" - divisor(7 DOWNTO 4)) &
                                    ("1001" - divisor(3 DOWNTO 0));
        minuend <= "000000000001";
        state <= shift;
    ELSE
        state <= ready;
    END IF;

--find appropriate dividend bits to compare with divisor
WHEN shift =>
    busy <= '1';
    --finish ten's complement calculation for future subtractions
    IF(q_index = 0) THEN
        subtrahend <= result;
    END IF;
    q_index := q_index + 1;
    --shift to appropriate dividend bits
    IF(q_index = 1) THEN
        divid_part <= "00000000" & divid(23 DOWNTO 20);
        state <= compare;
    ELSIF(q_index = 2) THEN
        divid_part <= "0000" & divid(23 DOWNTO 16);
        state <= compare;
    ELSIF(q_index = 3) THEN
        divid_part <= divid(23 DOWNTO 12);
        state <= compare;
    ELSIF(q_index = 4) THEN
        divid_part <= divid(19 DOWNTO 8);
        state <= compare;
    ELSIF(q_index = 5) THEN
        divid_part <= divid(15 DOWNTO 4);
        state <= compare;
    ELSIF(q_index = 6) THEN
        divid_part <= divid(11 DOWNTO 0);
        state <= compare;
    ELSE
        --error code
        quotient <= "101010101010101010101010";
        state <= ready;
    END IF;
```

```
        --compare dividend bits to divisor
        WHEN compare =>
          busy <= '1';
          IF(divis > divid_part) THEN                  --if divisor > digits above it
            IF(q_index = 6) THEN                       --if no more shifting can be done
              quotient <= quot;
              remainder <= divid(7 DOWNTO 0);
              state <= ready;
            ELSE                                       --if shifting not done
              state <= shift;
            END IF;
          ELSE                                         --if divisor <= digits above it
            minuend <= divid_part;
            state <= subtract;
          END IF;

        --subtract divisor from dividend bits
        --replace dividend value
        --increment quotient digit
        WHEN subtract =>
          busy <= '1';
          IF(q_index = 1) THEN
            divid(23 DOWNTO 20) <= result(3 DOWNTO 0);
            quot(23 DOWNTO 20) <= quot(23 DOWNTO 20) + 1;
          ELSIF(q_index = 2) THEN
            divid(23 DOWNTO 16) <= result(7 DOWNTO 0);
            quot(19 DOWNTO 16) <= quot(19 DOWNTO 16) + 1;
          ELSIF(q_index = 3) THEN
            divid(23 DOWNTO 12) <= result;
            quot(15 DOWNTO 12) <= quot(15 DOWNTO 12) + 1;
          ELSIF(q_index = 4) THEN
            divid(19 DOWNTO 8) <= result;
            quot(11 DOWNTO 8) <= quot(11 DOWNTO 8) + 1;
          ELSIF(q_index = 5) THEN
            divid(15 DOWNTO 4) <= result;
            quot(7 DOWNTO 4) <= quot(7 DOWNTO 4) + 1;
          ELSIF(q_index = 6) THEN
            divid(11 DOWNTO 0) <= result;
            quot(3 DOWNTO 0) <= quot(3 DOWNTO 0) + 1;
          ELSE
              --error code
            quotient <= "101010101010101010101010";
          END IF;
          divid_part <= result;
          state <= compare;

      END CASE;

      --clear to ready state when reset
      IF(reset_n = '0') THEN
          state <= ready;
      END IF;

    END IF;
  END PROCESS;
END controller;
```

Appendix B: Applicable Parts Manufacturers

Digi-Key carries a variety of parts suitable for implementing BCD division in programmable logic.

Altera (http://dkc1.digikey.com/us/mkt/vendors/544.html)

- MAX II CPLDs
- Cyclone/Cyclone II FPGAs
- Stratix/Stratix II FPGAs

Cypress Semiconductor (http://dkc1.digikey.com/us/mkt/vendors/428.html)

- Delta39K CPLDs

Xilinx (http://dkc1.digikey.com/us/mkt/vendors/122.html)

- CoolRunner II CPLDs
- Spartan-3 FPGAs
- Virtex-4 FPGAs

Disclaimer