Double dabble

From Wikipedia, the free encyclopedia

In computer science, the **double dabble** algorithm is used to convert binary numbers into binary-coded decimal (BCD) notation.[1][2] It is also known as the **shift and add 3 algorithm**, and can be implemented using a small number of gates in computer hardware, but at the expense of high latency.[3] The algorithm operates as follows:

Suppose the original number to be converted is stored in a register that is *n* bits wide. Reserve a scratch space wide enough to hold both the original number and its BCD representation; $4 \times ceil(n/3)$ bits will be enough. It takes a maximum of 4 bits in binary to store each decimal digit.

Then partition the scratch space into BCD digits (on the left) and the original register (on the right). For example, if the original number to be converted is eight bits wide, the scratch space would be partitioned as follows:

```
100s Tens Ones   Original
0010 0100 0011   11110011
```

The diagram above shows the binary representation of 243_{10} in the original register, and the BCD representation of 243 on the left.

The scratch space is initialized to all zeros, and then the value to be converted is copied into the "original register" space on the right.

```
0000 0000 0000   11110011
```

The algorithm then iterates *n* times. On each iteration, the entire scratch space is left-shifted one bit. However, *before* the left-shift is done, any BCD digit which is greater than 4 is incremented by 3. The increment ensures that a value of 5, incremented and left-shifted, becomes 16, thus correctly "carrying" into the next BCD digit.

The double-dabble algorithm, performed on the value 243_{10}, looks like this:

```
0000 0000 0000   11110011   Initialization
0000 0000 0001   11100110   Shift
0000 0000 0011   11001100   Shift
0000 0000 0111   10011000   Shift
0000 0000 1010   10011000   Add 3 to ONES, since it was 7
0000 0001 0101   00110000   Shift
0000 0001 1000   00110000   Add 3 to ONES, since it was 5
0000 0011 0000   01100000   Shift
0000 0110 0000   11000000   Shift
0000 1001 0000   11000000   Add 3 to TENS, since it was 6
0001 0010 0001   10000000   Shift
0010 0100 0011   00000000   Shift
   2    4    3
       BCD
```

Now eight shifts have been performed, so the algorithm terminates. The BCD digits to the left of the "original register" space display the BCD encoding of the original value 243.

Another example for the double dabble algorithm - value 65244_{10}.

```
 10^4 10^3 10^2  10^1 10^0    Original binary
0000 0000 0000 0000 0000   1111111011011100   Initialization
0000 0000 0000 0000 0001   1111110110111000   Shift left (1st)
0000 0000 0000 0000 0011   1111101101110000   Shift left (2nd)
0000 0000 0000 0000 0111   1111011011100000   Shift left (3rd)
0000 0000 0000 0000 1010   1111011011100000   Add 3 to 10^0, since it was 7
```

```
0000 0000 0000 0001 0101   1110110111000000   Shift left (4th)
0000 0000 0000 0001 1000   1110110111000000   Add 3 to 10^0, since it was 5
0000 0000 0000 0011 0001   1101101110000000   Shift left (5th)
0000 0000 0000 0110 0011   1011011100000000   Shift left (6th)
0000 0000 0000 1001 0011   1011011100000000   Add 3 to 10^1, since it was 6
0000 0000 0001 0010 0111   0110111000000000   Shift left (7th)
0000 0000 0001 0010 1010   0110111000000000   Add 3 to 10^0, since it was 7
0000 0000 0010 0101 0100   1101110000000000   Shift left (8th)
0000 0000 0010 1000 0100   1101110000000000   Add 3 to 10^1, since it was 5
0000 0000 0101 0000 1001   1011100000000000   Shift left (9th)
0000 0000 1000 0000 1001   1011100000000000   Add 3 to 10^2, since it was 5
0000 0000 1000 0000 1100   1011100000000000   Add 3 to 10^0, since it was 9
0000 0001 0000 0001 1001   0111000000000000   Shift left (10th)
0000 0001 0000 0001 1100   0111000000000000   Add 3 to 10^0, since it was 9
0000 0010 0000 0011 1000   1110000000000000   Shift left (11th)
0000 0010 0000 0011 1011   1110000000000000   Add 3 to 10^0, since it was 8
0000 0100 0000 0111 0111   1100000000000000   Shift left (12th)
0000 0100 0000 1010 0111   1100000000000000   Add 3 to 10^1, since it was 7
0000 0100 0000 1010 1010   1100000000000000   Add 3 to 10^0, since it was 7
0000 1000 0001 0101 0101   1000000000000000   Shift left (13th)
0000 1011 0001 0101 0101   1000000000000000   Add 3 to 10^3, since it was 8
0000 1011 0001 1000 0101   1000000000000000   Add 3 to 10^1, since it was 5
0000 1011 0001 1000 1000   1000000000000000   Add 3 to 10^0, since it was 5
0001 0110 0011 0001 0001   0000000000000000   Shift left (14th)
0001 1001 0011 0001 0001   0000000000000000   Add 3 to 10^3, since it was 6
0011 0010 0110 0010 0010   0000000000000000   Shift left (15th)
0011 0010 1001 0010 0010   0000000000000000   Add 3 to 10^2, since it was 6
0110 0101 0010 0100 0100   0000000000000000   Shift left (16th)
   6    5    2    4    4
            BCD
```

Sixteen shifts have been performed, so the algorithm terminates. The BCD digits is: $6*10^4 + 5*10^3 + 2*10^2 + 4*10^1 + 4*10^0 = 65244$.

Contents [hide]

C implementation [edit]

The double dabble algorithm might look like this when implemented in C. Notice that this implementation is designed to convert an "input register" of any width, by taking an array as its parameter and returning a dynamically allocated string. Also notice that this implementation does not store an explicit copy of the input register in its scratch space, as the description of the algorithm did; copying the input register into the scratch space was just a pedagogical device.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
   This function takes an array of n unsigned integers,
   each holding a value in the range [0, 65535],
   representing a number in the range [0, 2**(16n)-1].
   arr[0] is the most significant "digit".
   This function returns a new array containing the given
   number as a string of decimal digits.

   For the sake of brevity, this example assumes that
   calloc and realloc will never fail.
*/
void double_dabble(int n, const unsigned int *arr, char **result)
{
```

```c
    int nbits = 16*n;         /* length of arr in bits */
    int nscratch = nbits/3;   /* length of scratch in bytes */
    char *scratch = calloc(1 + nscratch, sizeof *scratch);
    int i, j, k;
    int smin = nscratch-2;    /* speed optimization */

    for (i=0; i < n; ++i) {
        for (j=0; j < 16; ++j) {
            /* This bit will be shifted in on the right. */
            int shifted_in = (arr[i] & (1 << (15-j)))? 1: 0;

            /* Add 3 everywhere that scratch[k] >= 5. */
            for (k=smin; k < nscratch; ++k)
              scratch[k] += (scratch[k] >= 5)? 3: 0;

            /* Shift scratch to the left by one position. */
            if (scratch[smin] >= 8)
              smin -= 1;
            for (k=smin; k < nscratch-1; ++k) {
                scratch[k] <<= 1;
                scratch[k] &= 0xF;
                scratch[k] |= (scratch[k+1] >= 8);
            }

            /* Shift in the new bit from arr. */
            scratch[nscratch-1] <<= 1;
            scratch[nscratch-1] &= 0xF;
            scratch[nscratch-1] |= shifted_in;
        }
    }

    /* Remove leading zeros from the scratch space. */
    for (k=0; k < nscratch-1; ++k)
      if (scratch[k] != 0) break;
    nscratch -= k;
    memmove(scratch, scratch+k, nscratch+1);

    /* Convert the scratch space from BCD digits to ASCII. */
    for (k=0; k < nscratch; ++k)
      scratch[k] += '0';

    /* Resize and return the resulting string. */
    *result = realloc(scratch, nscratch+1);
    return;
}

/*
   This test driver should print the following decimal values:
   246
   16170604
   1059756703745
*/
int main(void)
{
    unsigned int arr[] = { 246, 48748, 1 };
    char *text = NULL;
    int i;
    for (i=0; i < 3; ++i) {
        double_dabble(i+1, arr, &text);
        printf("%s\n", text);
        free(text);
    }
    return 0;
}
```

VHDL implementation [edit]

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
```

```vhdl
entity bin2bcd_12bit is
    Port ( binIN : in  STD_LOGIC_VECTOR (11 downto 0);
           ones : out  STD_LOGIC_VECTOR (3 downto 0);
           tens : out  STD_LOGIC_VECTOR (3 downto 0);
           hundreds : out  STD_LOGIC_VECTOR (3 downto 0);
           thousands : out  STD_LOGIC_VECTOR (3 downto 0)
          );
end bin2bcd_12bit;

architecture Behavioral of bin2bcd_12bit is

begin

bcd1: process(binIN)

  -- temporary variable
  variable temp : STD_LOGIC_VECTOR (11 downto 0);

  -- variable to store the output BCD number
  -- organized as follows
  -- thousands = bcd(15 downto 12)
  -- hundreds = bcd(11 downto 8)
  -- tens = bcd(7 downto 4)
  -- units = bcd(3 downto 0)
  variable bcd : UNSIGNED (15 downto 0) := (others => '0');

  -- by
  -- https://en.wikipedia.org/wiki/Double_dabble

  begin
    -- zero the bcd variable
    bcd := (others => '0');

    -- read input into temp variable
    temp(11 downto 0) := binIN;

    -- cycle 12 times as we have 12 input bits
    -- this could be optimized, we dont need to check and add 3 for the
    -- first 3 iterations as the number can never be >4
    for i in 0 to 11 loop

      if bcd(3 downto 0) > 4 then
        bcd(3 downto 0) := bcd(3 downto 0) + 3;
      end if;

      if bcd(7 downto 4) > 4 then
        bcd(7 downto 4) := bcd(7 downto 4) + 3;
      end if;

      if bcd(11 downto 8) > 4 then
        bcd(11 downto 8) := bcd(11 downto 8) + 3;
      end if;

      -- thousands can't be >4 for a 12-bit input number
      -- so don't need to do anything to upper 4 bits of bcd

      -- shift bcd left by 1 bit, copy MSB of temp into LSB of bcd
      bcd := bcd(14 downto 0) & temp(11);

      -- shift temp left by 1 bit
      temp := temp(10 downto 0) & '0';

    end loop;

    -- set outputs
    ones <= STD_LOGIC_VECTOR(bcd(3 downto 0));
    tens <= STD_LOGIC_VECTOR(bcd(7 downto 4));
    hundreds <= STD_LOGIC_VECTOR(bcd(11 downto 8));
    thousands <= STD_LOGIC_VECTOR(bcd(15 downto 12));
```

```
  end process bcd1;

end Behavioral;
```

VHDL testbench [edit]

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


ENTITY bin2bcd_12bit_test_file IS
END bin2bcd_12bit_test_file;

ARCHITECTURE behavior OF bin2bcd_12bit_test_file IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT bin2bcd_12bit
    PORT(
         binIN : IN  std_logic_vector(11 downto 0);
         ones : OUT  std_logic_vector(3 downto 0);
         tenths : OUT  std_logic_vector(3 downto 0);
         hunderths : OUT  std_logic_vector(3 downto 0);
      thousands : OUT  std_logic_vector(3 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal binIN : std_logic_vector(11 downto 0) := (others => '0');
   signal clk : std_logic := '0';

  --Outputs
   signal ones : std_logic_vector(3 downto 0);
   signal tenths : std_logic_vector(3 downto 0);
   signal hunderths : std_logic_vector(3 downto 0);
   signal thousands : std_logic_vector(3 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

   -- Miscellaneous
   signal full_number : std_logic_vector(15 downto 0);

BEGIN

 -- Instantiate the Unit Under Test (UUT)
   uut: bin2bcd_12bit PORT MAP (
          binIN => binIN,
          ones => ones,
          tenths => tenths,
          hunderths => hunderths,
       thousands => thousands
        );

   -- Clock process definitions
   clk_process :process
   begin
  clk <= '0';
  wait for clk_period/2;
  clk <= '1';
  wait for clk_period/2;
   end process;

   -- Combine signals for full number
   full_number <= thousands & hunderths & tenths & ones;

   -- Stimulus process
   stim_proc: process
   begin
```

```
        -- hold reset state for 100 ns.
        wait for 100 ns;

        wait for clk_period*10;

        -- insert stimulus here
    -- should return 4095
    binIN <= X"FFF";
    wait for clk_period*10;  assert full_number = x"4095" severity error;

    -- should return 0
    binIN <= X"000";
    wait for clk_period*10;  assert full_number = x"0000" severity error;

    -- should return 2748
    binIN <= X"ABC";
    wait for clk_period*10;  assert full_number = x"2748" severity error;


        wait;
     end process;

  END;
```

Historical [edit]

In the 1960s, the term **double dabble** was also used for a different mental algorithm, used by programmers to convert a binary number to decimal. It is performed by reading the binary number from left to right, doubling if the next bit is zero, and doubling and adding one if the next bit is one.[4] In the example above, 11110011, the thought process would be: "one, three, seven, fifteen, thirty, sixty, one hundred twenty-one, two hundred forty-three," the same result as that obtained above.

References [edit]

1. ^ Gao, Shuli; Al-Khalili, D.; Chabini, N. (June 2012), "An improved BCD adder using 6-LUT FPGAs", *IEEE 10th International New Circuits and Systems Conference (NEWCAS 2012)*, pp. 13–16, doi:10.1109/NEWCAS.2012.6328944.
2. ^ Binary-to-BCD Converter: “Double-Dabble Binary-to-BCD Conversion Algorithm,” originally at http://edda.csie.dyu.edu.tw/course/fpga/Binary2BCD.pdf, as cited by Gao & Al-KhaliliChabini (2012). Archived from original, January 31, 2012.
3. ^ Véstias, M.P.; Neto, H.C. (March 2010), "Parallel decimal multipliers using binary multipliers", *VI Southern Programmable Logic Conference (SPL 2010)*, pp. 73–78, doi:10.1109/SPL.2010.5483001.
4. ^ Godse, D. A.; Godse, A. P. (2008), *Digital Techniques*, Technical Publications, p. 4, ISBN 9788184314014.

Categories: Computer arithmetic algorithms | Binary arithmetic