Electrical and Computer Engineering

Two's Complement and Its Hardware

By Prawat Nagvajara

Synopsis
This notes describes theory and hardware implementations for calculating the two's complement of signed integer.

Theory
Consider a natural number

$$x = qm + r, \quad m > 0. \tag{1}$$

Define the remainder r as

$$r = x \text{ modulo } m \text{ (or } x \bmod m). \tag{2}$$

Addition, subtraction and multiplication on the set of remainders {r} forms the arithmetic of integer modulo m, that is,

$$(x \circ y) \bmod m = x \bmod m \circ y \bmod m, \tag{3}$$

where the operator ◦ can be addition or subtraction or multiplication.

Consider the positional representation of a natural number

$$x = x_{n-1}\, b^{n-1} + x_{n-2}\, b^{n-2} + \ldots + x_0\,, \tag{4}$$

where the base b > 0. The b^n modulo system is a system of n-symbol vectors where zero is the vector of all zeros and the maximum value is the vector of all b-1, the largest number from the set {0, 1, …, b-1}. For instance, in a six-digit system 0 = "000000" and the maximum number in the system is 999,999 = "999999". Numbers greater than or equal to one million in the system of integer modulo 10^6 is the last six digits. This is what happens when an odometer rolls over and the actual miles can be a multiple of millions plus the reading.

Without loss of generality, consider the case when b = 2, the n-bit positional numbers modulo 2^n. This is known as the unsigned data type in programming languages, where n = 32 as in the year 2015. The addition and subtraction can overflow or underflow when the sum or the difference no longer fits the n-bit representation at which a carry into or borrow from the n^{th} position occurs. The multiplication result is a 64-bit number and a product greater than $2^{32} - 1$ does not fit the unsigned type.

In the signed type, a negative number in the modulo 2^n system can be written as

$$-x \bmod 2^n = (2^n - x) \bmod 2^n, \tag{5}$$

since $2^n = 0 \bmod 2^n$ (this is similar to $10^6 = 0 \bmod 10^6$).

The number system in (5) implies traversing in the negative direction is a subtraction from 0. For instance

$-1 = 2^n - 1 =$ "11 … 11"
$-2 = 2^n - 2 =$ "11 …10"
…
$-2^{n-1} - 1 = 2^n - (2^{n-1} - 1) = 2^{n-1} - 1 =$ "100 … 0"

Note that, negative numbers have the leftmost equals to a one however its magnitude is calculated by (5). The leftmost bit is called a sign bit. Equation (5) defines the two's complement operator – a negative sign operator in n-bit vector system.

Assertion

-2^i is the string of all ones starting from the leftmost position and terminating at the i^{th} position "11…10…0".

Proof – by induction

Seed:
-1 = "11…1" is the string of all ones terminating at position i=0.

Claim:
-2^i = "11...10…0" is the string of all ones terminating at the i^{th} position.

Show that -2^{i+1} is the string of all ones terminating at $(i+1)^{st}$ position.

Since $-2^{i+1} = -2^i - 2^i$ = "11...10…0" - "00…10…0" is the string of ones terminating at the $(i+1)^{st}$ position.
QED

Note that a different proof for the above assertion is by considering (n+1)-bit system. Calculating $-2^i = 2^n - 2^i$ = "10 … 0" - "00…10…0" = "011…10…0", where 2^n is the (n+1)-bit vector with a single one at the n^{th} position and 2^i is the (n+1)-bit vector with a single one at the i^{th} position. However, the result in the n-bit system is equal to "11…10…0" n-bit vector with the string of all ones terminating at the i^{th} position.

Corollary
Appending '0' to the left of a positive number or '1' to the left of a negative number preserves the value of the number.

Proof
Left appending a '0' to a positive number adds 0 times 2^{n+1}. Since a negative number (vector) is equal to a string of ones starting from the leftmost position and terminating at the i^{th} position and zeros in the remaining position, $0 \le i \le n-1$, plus a positive vector with the string of zeros starting from the leftmost position and terminating at $(i-1)^{th}$ position, thus, appending ones to the left of a negative number preserves the value.
QED

For example, extending a 4-bit number to an 8-bit number, "0000" is appended if the 4-bit number is positive else "1111" is appended, -3 = 1001 in the 4-bit system and -3 = 11111001 in the 8-bit system.

Calculating the Two's Complement

In mod 2^n system $-x = 2^n - x = 2^n - 1 - x + 1$ this is a vector of all ones subtracted by x and plus one. The subtraction "11 … 1" - "x_{n-1} … x_i … x_0" is equivalent to the bit-wise logic not operation because is $x_i = 1$ the resulting bit is zero and if $x_i = 0$ the resulting bit is a one. Note that borrows do not occur in the subtraction. This implies that the subtractions at the bit positions can be done in parallel. The caveat is the "plus one" requires a carry into next position which is a serial calculation. The two's complement algorithm is as follows

Input: {x(i), i= 0, …, n-1}
Output: {z(i), i= 0, …, n-1}
Variable: {c(i), i= 0, …, n-1}
Begin
c(0) = 0;
for i = 0 to n-1 loop
  z(i) = [c(i) + 1-x(i)] mod 2;
    if z(i) > 1 then
      c(i+1) = 1;
    else c(i+1) = 0;
    end if;
end loop;

Figure 1 shows the data dependency graph describing the algorithm.

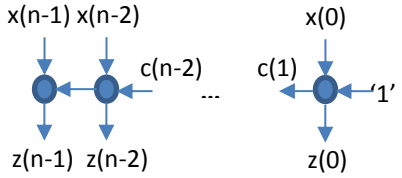

Fig. 1 Data Dependency Graph

The nodes i = 0, …, n-1, are the computation for each iteration c(i+1) and z(i). The edges describe the data dependency showing a serial computation.

The logic expressions where the logic '0' and '1' represent the numbers {0, 1} for the calculations are

$$z(i) \leftarrow \text{not } x(i) \text{ xor } c(i) \qquad (6)$$
$$c(i+1) \leftarrow \text{not } x(i) \text{ and } c(i),$$

where not x(i) is equivalent to 1-x(i) in arithmetic. The exclusive-OR function is defined as; A xor B is a '1' (true) when A = B else it is a '0' (false).

Code

A hardware description language description of an array structure circuit implementing the algorithm is as follow:

```
------------------------
-- two's complement
-- -x = 2**n - 1
--    = 2**n - 1 - x + 1
------------------------
------------------------
-- Processing Element
-- not x + c_in
------------------------
library ieee;
use ieee.std_logic_1164.all;
entity pe is
port (x, c_in: in std_logic;
      z, c_out: out std_logic);
end pe;

architecture struc of pe is
begin
z <= not x xor c_in;
c_out <= not x and c_in;
end struc;
```

```
-------------------------------------
-- Two's complement array
-- One's complement (not x)
-- plus one by assigning one to c_in
-- of the least significant bit.
-------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity twosC is
generic(n : natural := 4);
port (x: in std_logic_vector(n-1 downto 0);
      z: out std_logic_vector(n-1 downto 0));
end twosC;

architecture struc of twosC is
component pe
port (x, c_in: in std_logic;
      z, c_out: out std_logic);
end component;
signal c : std_logic_vector(n-1 downto 0);
begin
-- plus one by assigning c(0) with a one
c(0) <= '1';
G1: for i in 0 to n-1 generate
G2: if i < n-1 generate
cell: pe port map (x(i), c(i), z(i), c(i+1));
end generate G2;
G3: if i = n-1 generate
cell: pe port map (x(i), c(i), z(i), open);
end generate G3;
end generate G1;
end struc;
```

A behavioral description using process statement is as follows:

```
-------------------------------------
-- Behavioral description
-------------------------------------
architecture beh of twosC is
begin
process(x)
variable c : std_logic_vector(n-1 downto 0);
begin
-- plus one by assigning c(0) with a one
  c(0) := '1';
for i in 0 to n-2 loop
  c(i+1) := not x(i) and c(i);
  z(i) <= c(i) xor not x(i);
end loop;
-- c(n) not declared no assignment
z(n-1) <= c(n-1) xor not x(n-1);
end process;

end beh;
```

More data flow (concurrent statement) codes that use the addition function and negative sign arithmetical functions with vector and number operands are given below. The standard logic

arith, unsigned and signed packages contain these arithmetic functions.

```
-------------------------------------
-- Dataflow description
-- with std_logic_arith and
-- std_logic_unsigned packages
-------------------------------------
architecture Behavioral of twosC is
begin
z <= unsigned(not x) + 1;
end Behavioral;

-------------------------------------
-- Dataflow description
-- with std_logic_arith and
-- std_logic_signed packages
-------------------------------------
architecture dataflow of twosC is
begin
z <= -x;
end dataflow;
```

Figure 2 shows simulation waveform of the array structure circuit using ModelSim simulator.

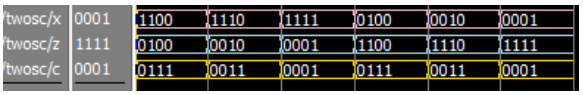

Fig. 2 Array Structure Simulation

Observe that the output z - the 2's complement of the input x, is the copies of '0's starting from the position 0^{th} up to the position where first '1' occurs. The bits after the first '1' position are inverted. For example x = "0100" the 2's complement is the copy of "100" and the higher significant bits that are inverted. Two examples are

x = "0100"
z = "1100"

x = "1010"
z = "0110"

A behavioral code implementing the copy and invert algorithm is as follows:

```
-------------------------------------
-- Behavioral description
-- by copy '0's until '1'
-- and invert following bits
-------------------------------------
architecture behav of twosC is
begin
process(x)
type my_state is (copy, invert);
variable state : my_state;
begin
  state := copy;
  for i in 0 to n-1 loop
    if state = copy then
      if x(i) = '0' then
        z(i) <= '0';
      else
        z(i) <= '1'; state := invert;
      end if;
    elsif state = invert then
      z(i) <= not x(i);
    else null;
    end if;
  end loop;
end process;
end behav;
```

Synchronous Circuit

Consider a data dependency graph G' that now incorporates the discrete time steps to calculate the nodes in the data dependency graph G. The graph G' has one dimension less than the original graph G since one of the dimensions in G' is the time. The nodes in G' calculate different nodes of G at different times. This requires a permissible schedule that satisfies the data dependency in G. For instance, the data dependency graph in Fig. 1 describes that the nodes i cannot computes before node j, j< i.

Linear mapping (projection) of the original graph onto the reduced dimension graph is a viable method. Moreover, the affine linear subspaces in G can be used as a schedule where the nodes belonging in an affine subspace are calculated at the same time by different processing elements. The consecutive affine subspaces in G are the calculations occurring at an increment in time. The time axis is the vector orthogonal to the subspace.

In a precise description, the nodes in G are vectors in an n-dimension space over integers, V^n. Let v be the orthogonal vector to a

subspace V', $u \in V'$ if and only if $u^T v = 0$, $u,v \in V^n$. The affine subspaces V'_t, t = 0, 1, …, where t represents the discrete time, are defined as $u \in V'_t$ if and only if $u^T v = t$. The computations at time t are the nodes in the affine subspace V'_t. These computations are distributed among the processing elements by a linear projection P: G → G'.

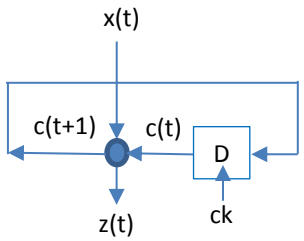

Fig. 2 Synchronous Circuit – Graph G'

The one-dimensional graph G in Fig. 1 is projected into onto zero-dimensional space which results in a synchronous circuit (Fig. 2) where a delay flip-flop D is synced with the clock signal ck. The D flip-flop buffers the variable c(i), i = 0, 1, …, n-1, at time t = i. In other words, the schedule is that the node in the circuit calculates node t in G at time t, t = 0, 1, …, n-1. The initial value for the delay flip-flop c(0) is '1'.

Exercises

1. Derive a data dependency graph for the copy and invert algorithm for calculating the two's complement.
2. Design and implement an array structure circuit for the copy and invert algorithm.
3. Design and implement a synchronous circuit for the copy and invert algorithm.