Representation of Integers and their Arithmetic

Suresh Purini, IIIT-H

February 7, 2013

What does the 8-bit string 11100000 represent? It could mean 224, -96, -31 and -32 when treated as an unsigned integer, sign-magnitude integer, one's complement integer and two's complement integer respectively. Or it could be mean the ASCII character α. So what a bit string means depends on the semantics or the definition we associate with it. The following C program illustrates the same idea.

```
#include <stdio.h>
main()
{
   printf("%d %u\n", 1<<31, 1<<31);
   printf(``%d %c\n", 1<<6, 1<<6);
}
```

In this write-up we shall study binary representation of unsigned and signed integers which is a primitive data structure supported by all modern processors.

1 Unsigned Integers

Consider the bijective function $B2U_w : \{0,1\}^w \rightarrow \{0, \cdots, 2^w - 1\}$ which maps w-bit binary strings to unsigned integers as follows. If $\vec{b} = b_{w-1} \cdots b_0$, then

$$B2U_w(\vec{b}) = \sum_{i=0}^{w-1} b_i 2^i$$

For example $B2U_4(0101) = 5$ and $B2U_4(1101) = 13$. You can observe that the function $B2U_w$ and its inverse are efficiently computable. In other words, we can easily compute the binary representation of an unsigned integer in the range of the function making it a viable representation.

In C-language all variables of type unsigned integers are allocated a fixed number of bytes (or equivalent number of bits) for storage which is typically 4 bytes or 32 bits. You can check this by running the following C-program on your machine.

```
#include <stdio.h>
main()
{
   printf("Size of Unsigned Integer: %d\n", sizeof(unsigned int));
}
```

Having represented unsigned integers in binary, we would like to figure out how to perform addition and multiplication operations. Let us just focus on addition operation in our discussion and the relevant ideas can be applied to multiplication operations too with suitable modifications. We presume that you know the algorithm to add two binary numbers as illustrated in the Figure 1 [1]. We also know the analogous algorithm for addition in the unsigned integer domain. Now

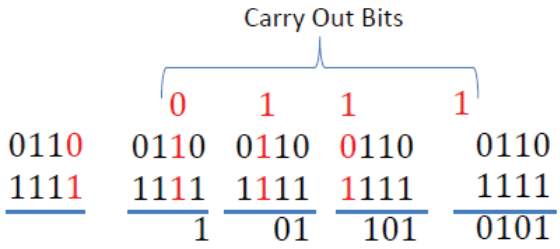

Figure 1: Addition of binary numbers

the beauty of the mapping function $B2U_w$ is that it shows the isomorphic structure between the unsigned integers and their binary representation with respect to the addition operation (and also multiplication operation). To elaborate more on this idea let us define the w-bit addition of two numbers as the regular binary addition except that we ignore the carry-out bit from the MSB if at all there is one. With this definition, when we add two w-bit numbers, the result is always a w-bit number. The key claim here is whether we do addition of two unsigned integers in decimal notation which we are familiar with or we do addition of their respective w-bit binary representations, the net result is just the same except for the difference in their notational representation. This claim is true insofar as the result of the addition operation does not cause an overflow, in other words the result would fit into w-bits. Consider the following Table 1 with three columns. If we want to add 1 and 4, whether we carry out addition in column 2 or in column 3, the respective results would fall in Row 5. However if we want to add 4 and 5, then the result wouldn't fall in the range in the column 2 and the result in the column 3 would fall in Row-1 (recall how w-bit addition is defined). It can be observed though that there is isomorphism between ($mod\ 2^w$) addition of decimal numbers and w-bit addition of binary numbers without worrying about overflow at all since it would never happen in modular arithmetic. It has to be noted here that we can use any other function (preferably bijective) from the w-bit strings to unsigned numbers and create a isomorphism between the decimal domain and the binary domain by appropriately defining the addition operations in the binary domain. We leave it to you to ponder whether such an alternate function has any utility. It is easy to note here that the addition of two w-bit unsigned numbers would cause an overflow if and only if the carry-out bit is 1.

Exercise 1. *What is the output of the following C program?*

```
#include <stdio.h>
main()
{
    printf("%X %X\n", 1048576, 1048576+5096+2048+256);
}
```

Exercise 2. *What is the output of the following C program?*

[1] Recall the w-bit ripple carry adder circuit.

Row No	3-bit Binary	Unsigned Integer
R_0	000	0
R_1	001	1
R_2	010	2
R_3	011	3
R_4	100	4
R_5	101	5
R_6	110	6
R_7	111	7

Table 1: Isomorphic structure of 3-bit binary numbers and their unsigned intrepetation

```
#include <stdio.h>
main()
{
 unsigned int var1 = 0xf0000000, var2 = 0x10000002;

 printf(" %u %u %u \n", var1, var2, var1 + var2);
}
```

2 Signed Integers

The following are three different ways of representing signed integers.

1. Sign-Magnitude Representation. The mapping function here is:

$$B2S_w(b_{w-1}...b_0) = (-1)^{b_{w-1}} * (2^{w-2} * b_{w-2} + ... + 2^0 * b_0)$$

2. 1's Complement Representation. The mapping function here is:

$$B2O_w(b_{w-1}...b_0) = -b_{w-1} * (2^{w-1} - 1) + b_{w-2} * 2^{w-2} + ... + b_0 * 2^0$$

3. 2's Complement Representation. The mapping function here is:

$$B2T_w(b_{w-1}...b_0) = -b_{w-1} * 2^{w-1} + b_{w-2} * 2^{w-2} + ... + b_0 * 2^0$$

Pretty much all systems use 2's complement representation for signed integers. We shall see the rationale behind such a choice in the following discussion. Check the output of the following C program and verify whether the signed integers are represented in 2'complement form.

```
#include <stdio.h>
main()
{
   printf(‘‘%X %X %X", -1, -2, -4096);
}
```

Row No	3-bit String	Sign-Magnitue	1's Complement	2's Complement
R_0	000	0	0	0
R_1	001	1	1	1
R_2	010	2	2	2
R_3	011	3	3	3
R_4	100	-0	-3	-4
R_5	101	-1	-2	-3
R_6	110	-2	-1	-2
R_7	111	-3	0	-1

Table 2: Isomorphic structure of 3-bit binary numbers and 2's complement signed integers

First you can verfiy that among the 3 mapping functions only the $B2T_w$ function corresponding to 2's complement representation is bijective. Let us stick to our definition of w-bit addition of binary numbers and we shall see that there is an isomorphic structure between signed integers and their 2's complement representation with respect to addition. It has to be noted that this isomorphism holds if and only if the results of addition does not cause overflow or underflow. Sign-magnitude and 1's complement representation of signed integers doesn't carry this isomorphic structure with respect to the canonical binary addition rules. It is worth noting that we can create an isomorphic structure even with these representations by suitable modifying the rules of binary addition. To understand these ideas consider the Table 2. For example if we add Row3 with Row4, the resulting binary number is 111 which lies in Row 7, whereas if we perform the addition on Sign-Magnitude numbers in Column 2, we get a value in Row 3 indicating the lack of isomorphic structure with respect to addition between the binary and sign-magnitude representation of numbers. It can be verified that there is no isomorphic structure between binary and one's complement representation of numbers by adding elements in Row 5 and Row 6. In binary addition we get an element in Row 3, whereas in the one's complement representation we get an element in Row 4 in Column 3. However it can be verified that as long as there is no overflow there is a perfect isomorphism with respect to addition between binary and two's complement representation of numbers.

3 Unsigned versus 2's Complement Addition

From the previous discussion it could have been noted that the rules of binary addition for both Unsigned and 2's Complement Addition is exactly the same. It means that we could use the same k-bit ripple carry to add any 2 unsigned or 2's complement numbers and we need not tell the k-bit ripple adder whether we are doing signed arithmetic or unsigned arithmetic. To illustrate this point further let us that I have a k-bit adder circuit with me, some of the students in the class want to do 2's complement addition and some of you may want to perform unsigned addition over k-bit numbers using my k-bit adder circuit. But you don't want to reveal me whether you are performing signed or unsigned arithmetic for whatever reasons you have. It is no big deal for my k-bit adder circuit as the rules of addition remains the same for both signed and unsigned numbers. However there is a catch here. The catch is that overflow conditions for signed and unsigned arithmetic are different.

Exercise 3. *For every combination of N,Z,C,V flags in the ARM ISA, contruct* `num1` *and* `num2` *such that the resulting add operation effects those flags correspondingly. Refer to the following C-program.*

```
#include <stdio.h>
main(int argc, char *argv[])
{
  register unsigned int num1 asm("r4"), num2 asm("r5"), sum asm("r11");
  register int nflag asm("r6"), zflag asm("r7"), cflag asm("r8"), vflag asm("r9");
  unsigned int tnum1, tnum2;

  nflag = 0;
  zflag = 0;
  cflag = 0;
  vflag = 0;

  sscanf(argv[1], "%x", &tnum1 );
  sscanf(argv[2], "%x", &tnum2);
  num1 = tnum1 << 28;
  num2 = tnum2 << 28;

  asm("adds r11, r4, r5");
  asm("movmi r6, #1");
  asm("moveq r7, #1");
  asm("movcs r8, #1");
  asm("movvs r9, #1");

  printf("num1 = %#x num2 = %#x sum = %#x\n", num1, num2, sum);
  printf("N = %d  Z = %d C = %d V = %d\n", nflag, zflag, cflag, vflag);
  fflush(stdout);
  exit(0);
}
```

4 Sign Extension

When we promote a 16-bit signed integer to a 32-bit signed integer, the sign and magnitude of the original number has to be retained during the type casting process. The simple strategy to accomplish that is to replicate the sign bit into the higher order bits.

Exercise 4. *What is the output of the following program?*

```
#include <stdio.h>

main()
{
 unsigned short int us;
 unsigned int u;
 signed short int ss;
 signed int s;

 printf("size of unsigned short: %d\n", sizeof(unsigned short));
 printf("size of signed short: %d\n", sizeof(signed short));
```

```
 printf("size of unsigned int: %d\n", sizeof(unsigned));
 printf("size of signed int: %d\n", sizeof(signed));

 us=0x7f00;
 u = us;
 printf("us=%x  u=%x \n", us, u);

 us=0xf000;
 u = us;
 printf("us=%x  u=%x \n", us, u);

 ss=0x7f00;
 s = ss;
 printf("ss=%x  s=%x \n", ss, s);

 ss=0x8000;
 s = ss;
 printf("ss=%hx  s=%x \n", ss, s);
}
```

5 Shifting Right

When we shift a 32-bit integer to the right, empty bits will be created in the higher order bits. If the 32-bit integer represents an unsigned integer we have to zero fill it and if it is a signed integer we have to sign-fill it. That is replicate the sign bits in the empty bit positions.

Exercise 5. *What is the output of the following C? Compile the program using* `-g` *option and then using* `objdump -D -S` *disassemble the code and identify the assembly instructions mapping to the two shift statements* `var >> 1` *and* `svar >> 1`.

```
#include <stdio.h>

#define CONST 0x1 << 31

main()
{
 unsigned int var = CONST;
 int svar = CONST  ;

 printf(" %x %x\n", var, var >> 1);

 printf(" %x %x\n", svar, svar >> 1);
}
```