Chapter 4

BINARY ARITHMETIC

4.1 Introduction

Bistable devices may be used to represent numbers in base two. It is possible to convert between base two numbers and decimal and it is possible to perform logic with such a representation. We will now consider arithmetic: Addition, subtraction, multiplication, and division. The use of two's complement arithmetic allows us to reduce subtraction to addition. Multiplication is also reduced to addition, and division reduced to addition and subtraction. Base two arithmetic is much simpler than decimal arithmetic. See [17] for an extensive discussion of computer number systems and arithmetic.

4.2 Binary Numbers and Addition

When two decimal digits are added, a sum and carry are produced. For example, if we add 7 to 5 we obtain a sum of 2 and a carry of 1. If we add 3 to 4 then the sum is 7 and the carry 0. The procedure for adding multidigit numbers is to add the digits in pairs, starting from the right, producing both a sum and carry for each digit. Then as we shift to the left, the carry is added to the next more significant pair of digits to produce a sum and carry. For example, to add 377 to 419, we first add the 9 to the 7 producing a sum of 6 and a carry of 1:

	3	7	7	
	4	1	9	+
sum			6	
carry			1	

The carry of 1, left-shifted, is then added to the 1 and 7 producing a sum of 9 and a carry of 0.

	3	7	7	
	4	1	9	+
sum		9	6	
carry		0		

The carry, left-shifted, is added to the 3 and 4 to produce a final sum of 7 with again a 0 carry and the process is complete.

	3	7	7	
	4	1	9	+
sum	7	9	6	

If we were to add 1 to 99,999, then a carry of 1 would ripple left turning all the 9's into 0's. This is called a ripple carry.

Turning our attention to binary numbers, the addition of a single bit is very simple, a sum and carry are produced, but in the binary case these may only be one or zero. There are four possible outcomes from the addition of two bits:

a	*b*	**Sum**
0	0	0
0	1	1
1	0	1
1	1	0

a	*b*	**Carry**
0	0	0
0	1	0
1	0	0
1	1	1

but comparison of the **xor** and **and** truth tables given earlier will reveal that the sum bit of the addition of two bits is the exclusive **or** and that the carry is simply the **and** of the two bits.

The process of addition may be represented in a C function as:

```
add (int a, int b)  /* addition using logical operations */
{
  int s;                    /*sum*/
  int c;                    /*carry*/

  s = a ^ b;                /*sum is the xor*/
  while (c = (a & b) << 1) /*carry is the and of inputs*/
    {
      a = s;
      b = c;
      s = a ^ b;
    }
  return (s);
}
```

For example, adding 13 to 11 by the above algorithm we have:

```
      Add   001101   = 13
            001011   = 11

      sum   000110
    carry   001001

      sum   000110
carry << 1  01001

      sum   010100
    carry   000010

      sum   010100
carry << 1  00010

      sum   010000
    carry   000100

      sum   010000
carry << 1  00100

      sum   011000
    carry   000000
```

and the sum is $11000_2 = 24_{10}$.

4.3 Half and Full Adders

The generation of the sum and carry bits may be performed by very simple electronic circuits. These circuits are available in chip form to perform the logical functions described above and are called gates. The generation of the sum and carry requires an and gate and an xor gate. These may be obtained together in a circuit called a half adder (see Figure 4.1). Two half adders may then be combined together with an or gate to add two inputs, A and B, with a carry in, to produce a sum and a carry out. Such a circuit is called a full adder (see Figure 4.2). See [12] for a full discussion of digital logic.

4.4 Modulus Arithmetic

Modulus arithmetic considers only numbers in a range $0 <= n < M$, where M is the modulus. A modulus operator % in C will force a number to be in the appropriate range by performing an integer division by M and keeping the remainder. A car odometer is another, possibly more familiar, example of modulus arithmetic. The

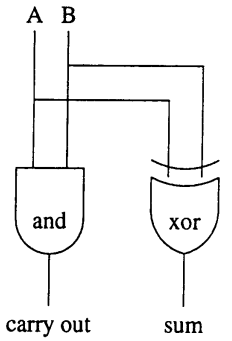


Figure 4.1: Half Adder

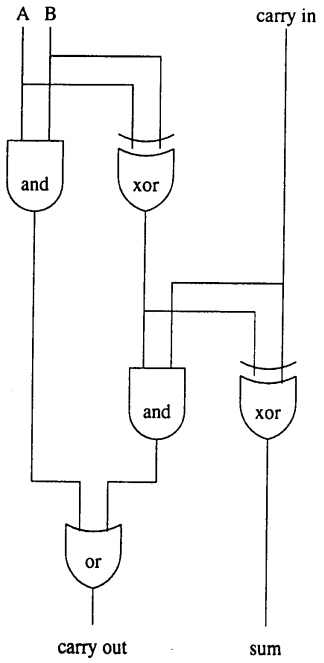


Figure 4.2: Full Adder

odometer counts up until it reaches 99999, whereupon it returns to 0. Computers normally perform modulus arithmetic, as they have registers of fixed size, like the car odometer. If we have an n bit register and count up from 0, then when the number reaches $2^n - 1$ (represented by all ones), the next increment returns the register to all zeros. A carry out of the msb (most significant bit) has been lost.

Consider a computer with four bit registers: The largest number will be $2^n - 1 = 2^4 - 1 = 15 = 1111_2$. Consider the addition of $3 + 2$:

$$\begin{array}{rll} 3 = & 0011 & \\ 2 = & 0010 & + \\ \hline 5 = & 0101 & \end{array}$$

further addition of 6 will result in 11:

$$\begin{array}{rll} 5 = & 0101 & \\ 6 = & 0110 & + \\ \hline 11 = & 1011 & \end{array}$$

and then if we add 7 we will obtain $(11 + 7) \,\%\, 16 = 2$:

$$\begin{array}{rll} 11 = & 1011 & \\ 7 = & 0111 & + \\ \hline 2 = & 0010 & \end{array}$$

the carry of 1 out of the msb is lost.

4.5 Subtraction

While addition is fairly simple, subtraction requires borrowing if the digit you are subtracting exceeds the digit from which it is to be subtracted. However, a neat hack can avoid these borrowing problems. Consider the following expression with arithmetic performed modulus r^n where r is the base and n the number of digits:

$$a - b = a + (r^n - 1 - b) + 1$$

In the above expression the addition of r^n does not affect the result of the calculation as the arithmetic is performed modulus r^n. The subtraction of b from $r^n - 1$ is particularly simple and *involves no borrowing.*

Consider the following example in decimal arithmetic with $r = 10$ and two digit registers and thus $n = 2$:

$$\begin{aligned} 23 - 07 &= 23 + (10^2 - 1 - 07) + 1 \\ &= 23 + (99 - 07) + 1 \\ &= 23 + 92 + 1 \\ &= 23 + 93 \\ &= 16 \end{aligned}$$

What about $07 - 23$?

$$\begin{aligned} 07 - 23 &= 07 + (99 - 23) + 1 \\ &= 07 + 76 + 1 \\ &= 07 + 77 = 84 \end{aligned}$$

What does 84 represent? Let us look at $00 - 16$.

$$\begin{aligned} 00 - 16 &= 00 + (99 - 16) + 1 \\ &= 00 + (83) + 1 \\ &= 84 \end{aligned}$$

84 is negative 16. What then is -1?

$$\begin{aligned} 00 - 01 &= 00 + (99 - 01) + 1 \\ &= 00 + 98 + 1 \\ &= 99 \end{aligned}$$

Let us, using some intuition, list numbers around zero:

3	03
2	02
1	01
0	00
-1	99
-2	98
-3	97

Let us go further and look at the mid range, -49:

$$\begin{aligned} 00 - 49 &= 00 + (99 - 49) + 1 \\ &= 00 + 50 + 1 \\ &= 51 \end{aligned}$$

What is -50?

$$\begin{aligned} -50 &= 00 + (99 - 50) + 1 \\ &= 00 + 49 + 1 \\ &= 50 \end{aligned}$$

Fifty is its own complement. Let us summarize our findings:

49	49	largest positive number
48	48	
... 3	3	
2	2	
1	1	
0	0	
−1	99	negative
−2	98	
... −48	52	
−49	51	
−50	50	largest negative number

This is called complement arithmetic and the representations of the negative numbers have names:

$r^n - 1 - b$	diminished radix complement	9's complement
$r^n - 1 - b + 1$	radix complement	10's complement

Radix complement arithmetic has zero, positive, and negative numbers. In radix complement arithmetic the largest negative digit is its own complement and thus has no positive equivalent. Furthermore, for decimal numbers, if the most significant digit (msd) ≥ 5, then the number is negative. Subtraction may then be performed by the addition of the negative of the number that is to be subtracted.

Considering now binary numbers, we will see that complementing is even simpler than for decimal numbers. Let us consider four-bit arithmetic with $n = 4$. We will first subtract 2 from 4:

$$4 - 2 = 4 + (-2)$$

$$\begin{aligned} 4 &= 0100 \\ 2 &= 0010 \end{aligned}$$

$$\begin{aligned} 2^4 &= 10000 \\ 2^4 - 1 &= 1111 \\ 2^4 - 1 - 2 &= 1101 \\ 2^4 - 1 - 2 + 1 &= 1110 \end{aligned}$$

$$\begin{array}{ll} 0100 & \\ 1110 & + \\ \hline 0010 & = 2 \end{array}$$

$2^4 - 1 - 2 = 1101$ is called the one's complement, as the subtraction of a binary number from all ones is the logical complement. All ones are replaced by zeros and all zeros are replaced by ones. $2^4 - 1 - 2 + 1 = 1110$ is called the two's complement, and it is no more than the one's complement plus one.

Consider $2 - 4$:

$$\begin{aligned} 2 &= 0010 \\ 4 &= 0100 \\ \text{one's complement 1's} &= 1011 \\ \text{two's complement 2's} &= 1100 \end{aligned}$$

$$\begin{array}{ll} 0010 & \\ 1100 & + \\ \hline 1110 & \end{array}$$

1110 must be a negative number, so let us complement it:

$$\begin{array}{lll} & 1110 & \\ 1\text{'s} & 0001 & \\ 2\text{'s} & 0010 & -2 \end{array}$$

Let us use some intuition and list all the four-bit two's complement numbers:

0111	7	
0110	6	
0101	5	
0100	4	
0011	3	
0010	2	
0001	1	
0000	0	
1111	-1	$0000 + 1 = 0001$
1110	-2	
1101	-3	
1100	-4	
1011	-5	
1010	-6	
1001	-7	
1000	-8	its own complement 1000

If the most significant bit, the sign bit, is a one, then we know that the number is negative.

A signed number has a range of $[-2^{n-1}, 2^{n-1} - 1]$ and an unsigned number has a range of $[0, 2^n - 1]$. The two's complement system is an interpretation of numbers in registers; the hardware always performs binary addition. If we wish to subtract, we first form the two's complement of the number and then add; there is then no need for a hardware subtractor as well as a hardware adder. We will still, however, frequently consider numbers to be unsigned, considering the sign bit only as the most significant bit of the number; pointers are always unsigned numbers, as they refer to memory addresses.

4.6 Two's Complement Number Branching Conditions

In Chapter 2 we introduced signed arithmetic branches; these are the appropriate branches when we are interpreting the numbers in the machine as two's complement. Branching conditions are based on the setting of the N (negative), Z (zero), and V (overflow) bits. The Z bit is set when all the bits of the result are zero. The N bit is set when the most significant bit is 1. The overflow V bit is set when the register is not long enough to hold the true representation of the number. On subtraction, the V bit is set when the signs of the minuend and the subtrahend are different and the sign of the difference is the same as the subtrahend (difference = minuend − subtrahend).

When we compare two two's complement numbers, we subtract one from the other; overflow can frequently occur when this happens. Consider subtracting any positive number from the largest negative number.

The conditions for signed branches are:

Assembler Mnemonic	Signed Arithmetic Branches	Condition Codes
`bl`	Branch on less	(N xor V) = 1
`ble`	Branch on less or equal	Z or (N xor V) = 1
`be`	Branch on equal	Z = 1
`bne`	Branch on not equal	Z = 0
`bge`	Branch on greater or equal	(N xor V) = 0
`bg`	Branch on greater	Z or (N xor V) = 0

When overflow occurs the sign is complemented. On integer addition the overflow bit is set when the sign of the addends is the same, but the sign of the result is different.

4.6.1 Shifting

Three shift instructions are provided in the SPARC architecture to compute the contents of a register shifted left or right by a number of shifts. There are two shifts, arithmetic and logical. In the case of an arithmetic shift, the sign bit is copied into the most significant bit position on right shifts. In the case of a logical right shift, zeros are copied into the most significant bit position. Left shifts are identical in both cases with zeros shifted in from the right (see Figure 4.3).

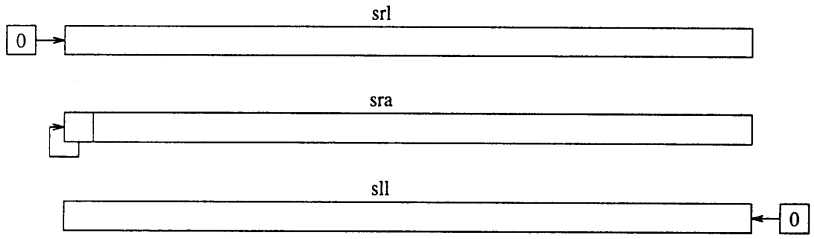


Figure 4.3: Shift Instructions

The shift count is the low five bits of reg_{rs2} or the low five bits of the immediate. These five bits are interpreted as a positive number, and thus the largest shift possible is $2^5 - 1$, or 31.

The shift instructions are as follows:

`sll`	reg_{rs1}, *reg_or_imm*, reg_{rd}	shift left logical
`sra`	reg_{rs1}, *reg_or_imm*, reg_{rd}	shift right arithmetic
`srl`	reg_{rs1}, *reg_or_imm*, reg_{rd}	shift right logical

Shifting a number left corresponds to multiplication by two; shifting right arithmetic corresponds to division by two.

4.7 Unsigned Arithmetic

In addition to signed arithmetic there is a need to perform unsigned arithmetic. In unsigned arithmetic, number representations are always considered positive and numbers have a range of $[0, 2^n]$, twice that of signed numbers. In C, the declaration **unsigned** informs the compiler that the variable is to be interpreted as an unsigned number. Hardware operations on signed and unsigned numbers are identical.

There is no problem with addition, with a carry out of the most significant bit (C = 1), now indicating overflow. A carry out of the most significant bit occurs when the most significant bit of both operands is set but the most significant bit of the result is a zero, or when the most significant bit of either operand is set, but the most significant bit of the result is zero.

It is also still possible to subtract unsigned numbers by imagining that there is an extra bit in the register to the left of the most significant bit. Then, by first forming the two's complement of the subtrahend and adding, we would expect a carry out of the most significant bit to be added to the imaginary one of the subtrahend to indicate a positive result; the imaginary bit of the minuend would then be a zero. In the case of subtraction, the C bit is set **if there is no carry out of the most significant bit**. The test for unsigned overflow is then simply a test of the C bit.

Consider a four-bit register, which has an unsigned range of $0 <= n < 16$, subtracting 3 from 12:

$$\begin{aligned} 12 - 03 &= 1100 - 0011 \\ &= 1100 + (10000 - 1) - 0011 + 1 \\ &= 1100 + 1111 - 0011 + 1 \\ &= 1100 + 1101 \\ 9 &= 1001 \end{aligned}$$

As expected, the carry occurred indicating a positive result.

4.8 Unsigned Number Branching Conditions

When performing unsigned arithmetic, the overflow bit, V, has no meaning, as it is related to tests on the signed numbers. The N-bit has no significance when dealing with unsigned numbers. All unsigned branching tests are made, based on the state of the C and Z bits alone. The C bit is set when a carry occurs out of the most significant bit on addition and when it does not occur on subtraction.

There is a set of unsigned branch instructions that makes the following tests:

Assembler Mnemonic	Unsigned Arithmetic Branches	Condition Codes
`blu`	Branch on less	C = 1
`bleu`	Branch on less or equal	C = 1 or Z = 1
`be`	Branch on equal	Z = 1
`bne`	Branch on not equal	Z = 0
`bgeu`	Branch on greater or equal	C = 0
`bgu`	Branch on greater	C = 0 and Z = 0

4.9 Condition Code Tests

There is also a set of branches that tests the individual condition codes:

Assembler Mnemonic	Condition Code Branches		Synonym
`bneg`	Branch on negative	N = 1	
`bpos`	Branch on positive	N = 0	
`bz`	Branch on equal to zero	Z = 1	be
`bnz`	Branch on not equal to zero	Z = 0	bne
`bvs`	Branch on overflow set	V = 1	
`bvc`	Branch on overflow clear	V = 0	
`bcs`	Branch C set	C = 1	blu
`bcc`	Branch C clear	C = 0	bgeu

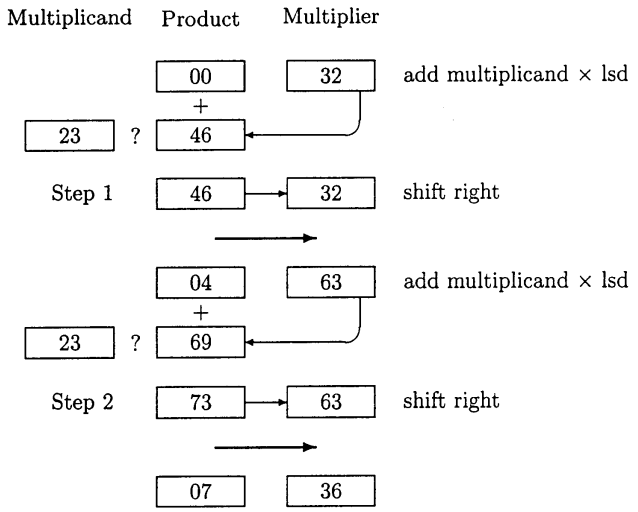
4.10 Multiplication

If we had to multiply 23 × 32 using pencil and paper, we might proceed as follows:

```
  23
  32 x
----
  46
 69
----
 736
```

In the above, we multiply the multiplicand by the each digit of the multiplier, eventually adding these partial products to form the product. We might organize the above calculation somewhat. We will make use of two two-digit registers to eventually hold the product. We will initialize the first of these to zero and the second to the multiplier. We will examine the right-most digit of the multiplier and then multiply the multiplicand by the digit and add it to the left-hand register.

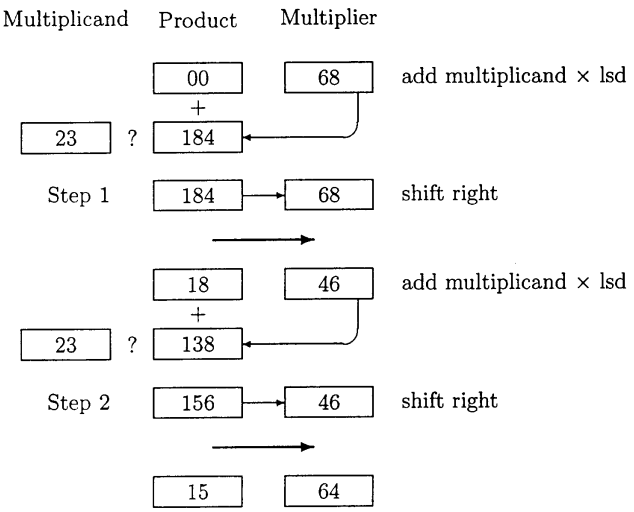
Having made use of the right-most digit of the multiplier we will shift the two registers right one digit before repeating the process until all digits of the multiplier have been examined. The two registers will thus hold both the partial product and the partial multiplier until they hold the product.



What would happen if we tried to multiply 23 × −32? Let us try using 10's complement arithmetic. First we need to find the 10's complement of −32:

```
99
32 -
--
67
 1 +
68
```

Let us now proceed to use our algorithm multiplying 23 by 68:



The result should be -736 or, in complement form, 9264, which it clearly is not. What went wrong? A careful examination of the algorithm will reveal that we have evaluated:

$$23 \times (10^2 - 32)$$
$$23 \times 10^2 - 23 \times 32$$

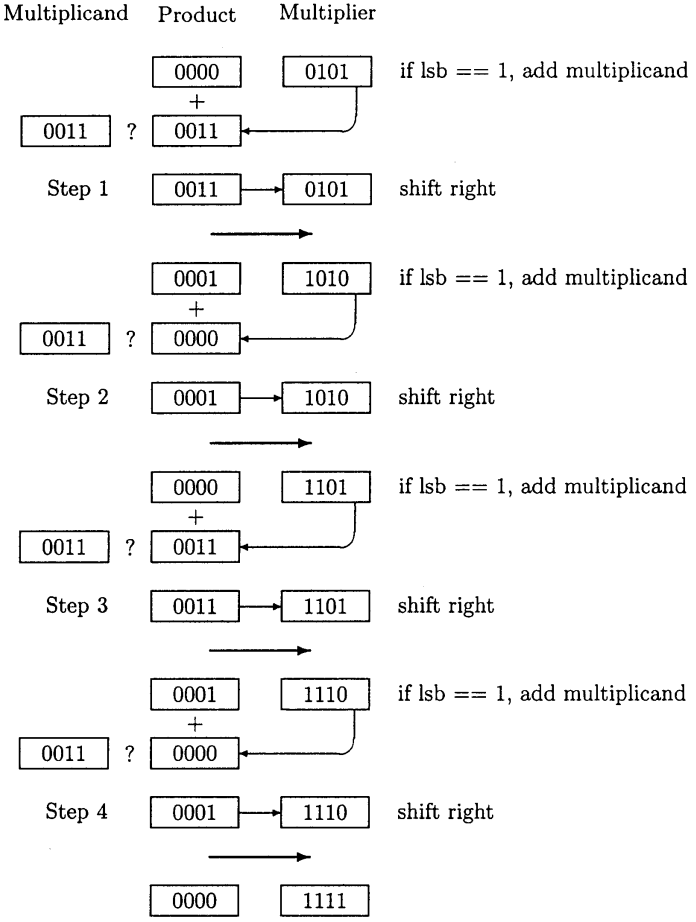
The correct result is -23×32; our result is too large by 23×10^2. Do not forget that we are computing a result of twice the number of digits as the multiplier and multiplicand. When performing this computation we should be using twice as many digits when representing negative numbers. If we do not use twice as many digits when the multiplier is negative, then we must subtract from the high-order part of the final result, the multiplicand, which in our case is 23×10^2:

```
23
76 9's complement
77 10's complement

1564
77   +
----
9264

0735 9's complement
0736 10's complement
```

This gives the correct result. There is no problem when the multiplicand is negative, as we are simply adding scaled versions of it to obtain the result. It is

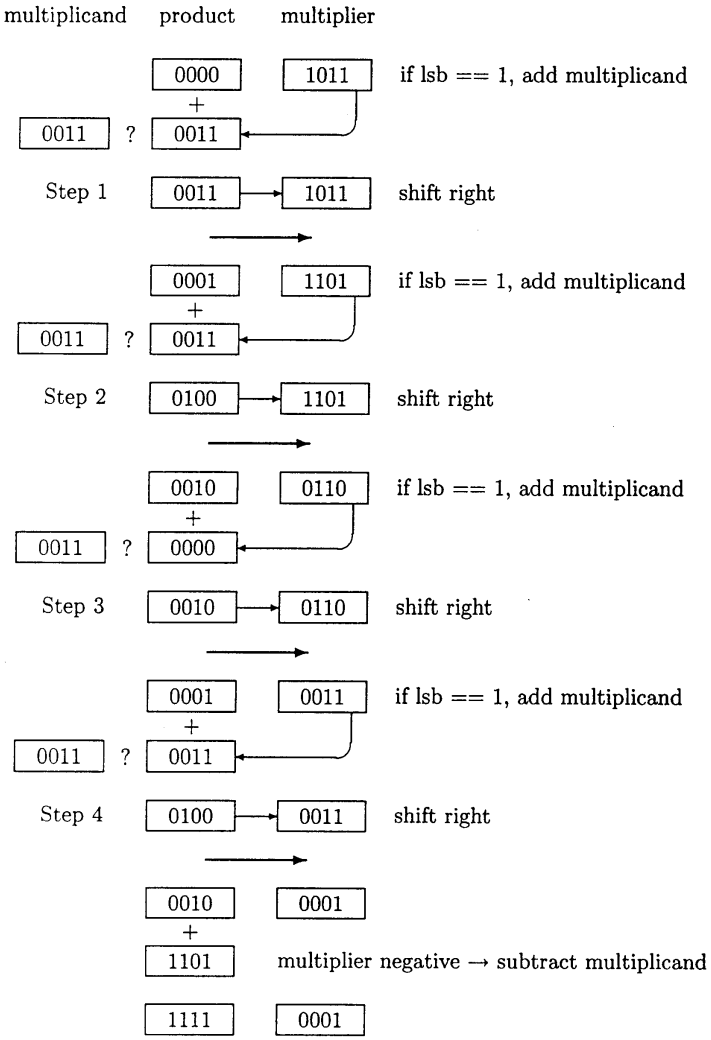
only when the multiplier is negative that we need to make the final subtraction to obtain the correct result.

Let us now turn our attention to binary arithmetic. Consider the multiplication of 3×5, or 0011×0101:



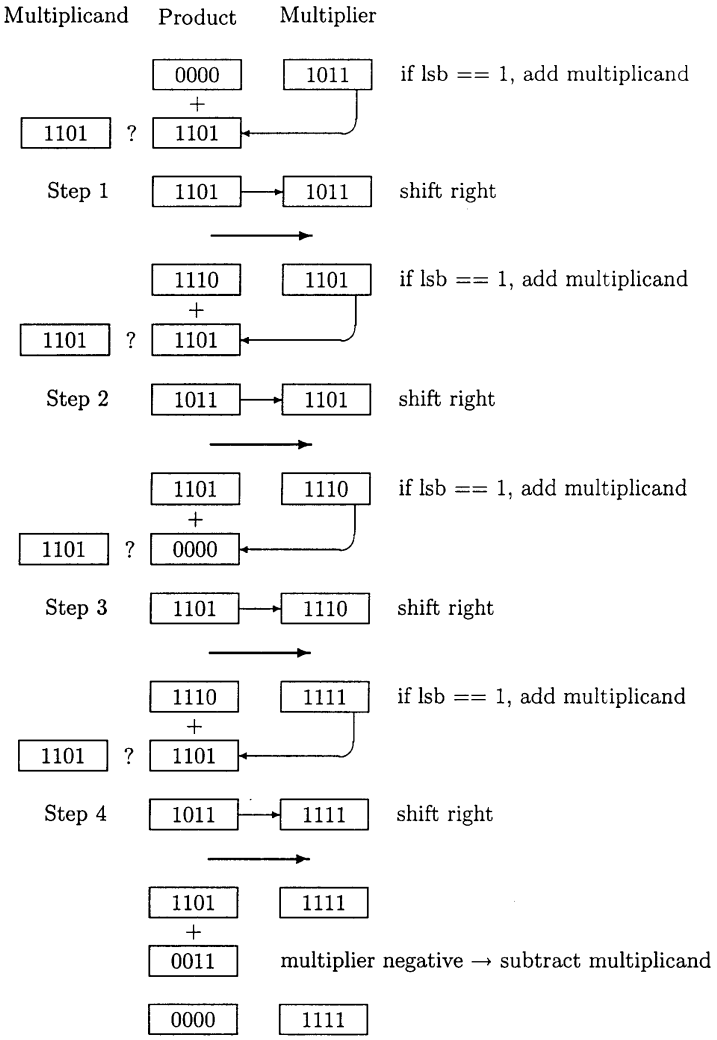
1111, 15 the correct result!

How about 3×-5? Or 0011×1011:



As the multiplier was negative we had to subtract the multiplicand from the high-order part of the result. The multiplier was 3, or 0011_2, and its two's complement 1101.

Finally, let us multiply -3×-5, or 1101×1011.



We obtain the correct result again, 15. Note the arithmetic shift right, shifting in one's when the number is negative.

4.10.1 SPARC `mulscc` Instruction

While the SPARC architecture does not include a multiply instruction it does have an instruction which performs one step of a multiply, the `mulscc` instruction. This instruction works in conjunction with a special machine register called the `%Y` register which initially holds the multiplier and eventually holds the low order part of the product. Multiplication using the `mulscc` instruction on the SPARC machine is as follows:

1. The multiplier is loaded into the %Y register and the high order part of the product cleared to zero.
2. The multiplier is tested to set the N and V bits.
3. This is then followed by 32 `mulscc` instructions. The `mulscc` instruction shifts `N ^ V` into the first source register, shifting the contents of the register right one place, the bit shifted out of the right hand end of the register is kept. The least significant bit of the Y register is tested, and if a one, the contents of the second source register or sign extended constant, are added to the destination register. Finally, the kept bit, shifted out of the first source register, is shifted into the Y register, shifting the contents of the Y register right one place.
4. One additional mulscc instruction with the multiplicand zero forms a final shift to produce a two word result with the high order part of the product in the %rd register and the low order part in %Y.

(see Figure 4.4).

For example, to multiply 3×5 with the three in %o2 and five, the multiplier, in %o0 we would write:

```
mov     3, %o2
mov     5, %o0

mov     %o0, %y

nop      !it takes time to get to the %y register
nop
nop
andcc   %g0, %g0, %o1   !zero the partial product
                        ! and clear N and V
mulscc  %o1, %o2, %o1   !32 mulscc instructions
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
```



```
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %o2, %o1
mulscc  %o1, %g0, %o1   !final shift
mov     %y, %o0         !high order part back from %y
```

System routines are provided for multiplication, `.mul` for signed multiplication, and `.umul` for unsigned multiplication. The multiplicand is passed in `%o0` and the multiplier in `%o1`. The low-order part of the result is returned in `%o0` and the high-order part in `%o1`.

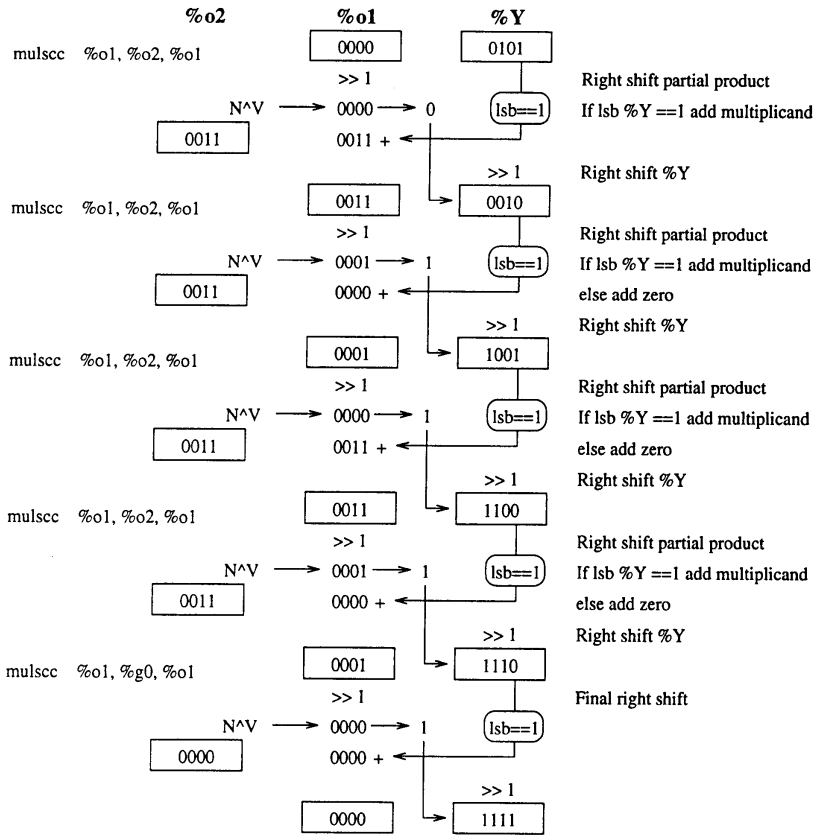


Figure 4.4: SPARC Multiplication

4.11 Division

Finally, we must consider division. Consider the division of 737 by 32:

```
      23 + 1
    ------
32 ) 737
     64-
     --
      97
      96-
      --
       1
```

The result is 23 with a remainder of 1.

Binary division is much easier, as a number either subtracts or it does not. One does not have to try to estimate how many times the dividend will successfully subtract from the dividend. Let us translate the previous example into binary. Of course, we will subtract by adding the two's complement.

```
          10111 + 1
         -------------
0100000 ) 01011100001     737 / 32
          1100000         -32
          -------
          00011100001     positive, set 1 into quotient
           1100000        shift and subtract again
           -------
          1111100001      negative, set 0 into quotient
          0100000         add dividend back
          -------
          0011100001
           1100000        shift and try again
           -------
           001100001      positive, set 1 into quotient
            1100000
            -------
             0100001      positive, set 1 into quotient
             1100000      shift and subtract again
             -------
              000001      positive, set 1 into quotient
                          remainder is 1.
```

This is very simple and one could formalize the algorithm as we did for multiplication. However, there is a further simplification that we should introduce: When we subtract the divisor and the result is negative we must add it back, shift it right one place, and then subtract again. For example, if b is the divisor and a the dividend:

$$a - b + b - b2^{-1}$$

Regrouping terms we obtain:

$$a - b + (2b - b)2^{-1}$$
$$a - b + b2^{-1}$$

Having subtracted and produced a negative result we only need shift the divisor right one place and *add* in place of subtracting:

```
          10111 + 1
         -------------
0100000 ) 01011100001     737 / 32
          1100000         -32
          -------
          00011100001     positive, set 1 into quotient
           1100000        shift and subtract again
           -------
          1111100001      negative, set 0 into quotient
            0100000       shift and add
            -------
           001100001      positive, set 1 into quotient
             1100000      shift and subtract
             -------
            00100001      positive, set 1 into quotient
              1100000     shift and subtract
              -------
               000001     positive, set 1 into quotient
                          remainder is 1.
```

This is called nonrestoring division [15].

An assembly language program to perform the above division is given below. The program first creates a dividend by multiplying two numbers together; it then divides the dividend by one of the factors used to produce the dividend. The dividend is 64 bits long and is in registers %o1 and %o0. The divisor is placed into register %o2. The quotient then appears in %o0 with the remainder in %o1.

In the program, we will need to shift the contents of two registers. This is accomplished by the following code:


```
        addcc   %lo_r, %lo_r, %lo_r
        bcc     1f
        add     %hi_r, %hi_r, %hi_r
        bset    1, %hi_r
1:
```

Note the use of a numeric label, "1." The assembler allows single-digit labels to appear many times in a single source file. A branch to such a label must be distinguished by appending the single letter "b" or "f" to the digit to indicate whether it is the first occurrence of the label in the backward direction in the file or the first occurrence of the label in the forward direction in the file. The use of such numeric labels is recommended for labels that have no intrinsic significance, such as labels needed to create control structures. This relieves the programmer of the need to create names that have no significance. Numeric labels also solve a problem when writing macros that require labels, allowing the same label to appear in each incantation of the macro without leading to multiply defined symbol errors:

```
  define(lo_r, o0)            !'low part of dividend'
  define(hi_r, o1)            !'high part of dividend'

  define(divisor_r,o2)
  define(count_r, o3)         !'number of times to iterate'

  define(n_bits, 32)          !'number of bits in register'
  define(quotient,0x101)      !'trial quotient'
  define(dividend,0xff)       !'trial dividend'
  define(remainder,0x2)       !'trial remainder'

  .global _main
_main:
  save    %sp, -64, %sp

  mov     quotient, %o0       !'form num as trial dividend'
  call    .umul
  mov     dividend, %o1
  add     %o0, remainder, %o0 !'add in a remainder'

  mov     dividend, %o2       !'use same number as divisor'

  mov     n_bits, %count_r    !'initialize count to n_bits'


  ba      pos                 !'start off with a shift'
  addcc   %lo_r, %lo_r, %lo_r
```

```
test:
  bge     pos
  addcc   %lo_r, %lo_r, %lo_r

neg:                              !'result negative, shift'
  bcc     1f
  add     %hi_r, %hi_r, %hi_r
  bset    1, %hi_r
1:
  ba      fin
  addcc   %hi_r, %divisor_r, %hi_r!'finished?'

pos:                              !'positive -> shift & sub'
  bcc     1f
  add     %hi_r, %hi_r, %hi_r
  bset    1, %hi_r
1:
  subcc   %hi_r, %divisor_r, %hi_r

fin:    bl      szero
  subcc   %count_r, 1, %count_r
  or      %lo_r, 1, %lo_r       !'set bit into quotient'
szero:
  bg      test
  tst     %hi_r

done:
  bge     ok                    !'restore remainder?'
  mov     1, %g1
  add     %hi_r, %divisor_r, %hi_r
ok:
  ta      0
```

Integer division occurs much less frequently in code than multiplication and the SPARC architecture does not provide an instruction equivalent to `mulscc` for division. Four routines are, however, provided: two for signed arithmetic `.div`, which returns the quotient, and `.rem`, which returns the remainder; two routines for unsigned arithmetic, `.udiv` and `.urem`. In all cases the dividend is in register `%o0` and the divisor in `%o1`. The result is returned in `%o0`.

4.12 Extended Precision Arithmetic

Occasionally there is a need to perform arithmetic to greater than 32 bits of precision. Consider the case where it is desired to perform integer arithmetic with 96 bits of precision. We may store such a 96-bit number in three sequential registers with the most significant bits in the lowest of the three registers. If such a number were stored in registers `%l0`, `%l1`, and `%l2`, then the sign bit would be bit 31 of `%l0` and the least significant bit of the 96 bit number would be bit 0 of `%l2`.

4.12.1 Addition of Extended Precision Numbers

There is no machine instruction to add three register numbers; instead, we have to proceed by adding the two low registers of both numbers, bits 0 – 31, then adding the two registers containing bits 32 – 63 along with any carry that was generated when the two low registers were added. Finally, we add the two high registers containing bits 64 – 95 along with any carry generated when the mid registers were added. There is a machine instruction especially for this purpose that adds the contents of two registers together plus one if the C, carry, bit is set. A carry from the previous add will set the carry bit:

```
addx     reg_rs1, reg_or_imm, reg_rd
addxcc     reg_rs1, reg_or_imm, reg_rd
```

In both cases the operation result is:

$$reg_{rd} = reg_{rs1} + reg\_or\_imm + \mathrm{C}$$

with the `addxcc` instruction also setting the condition codes. Thus, if the second number were in registers `%l3` – `%l5`, with the result to go into `%o0` – `%o2`, the code for the extended precision addition would be:

```
addcc   %l2, %l5, %o2   !add bits 0 - 31
addxcc  %l1, %l4, %o1   !add bits 32 - 63 + C
addx    %l0, %l3, %o0   !add bits 64 - 95 + C
```

4.12.2 Subtraction of Extended Precision Numbers

On subtraction we need to form the two's complement of one of the multiregister numbers. We could do this by first forming the one's complement of each register, and then add one to the low register (propagating any carry). We would then add the numbers as before:

```
not     %l5, %l5        !form complement in place
not     %l4, %l4
not     %l3, %l3
inccc   %l5             !add one to form two's comp.
addxcc  %l4, %g0, %l4   !propagate carry
```

```
addx    %l3, %g0, %l3

addcc   %l2, %l5, %o2   !add bits 0 - 31
addxcc  %l1, %l4, %o1   !add bits 32 - 63 + C
addx    %l0, %l3, %o0   !add bits 64 - 95 + C
```

We can reduce this to three instruction like the multiple-precision add by making use of the subx and subxcc instructions. These instructions subtract their second operand from their first and, in addition, subtract one more if the C bit is set:

subx reg_{rs1}, *reg_or_imm*, reg_{rd}
subxcc reg_{rs1}, *reg_or_imm*, reg_{rd}

In both cases the operation result is:

$$reg_{rd} = reg_{rs1} - reg\_or\_imm - \mathrm{C}$$

with the subxcc instruction also setting the condition codes.

By proceeding to directly use the subcc instruction to subtract the two low registers:

```
subcc   %l2, %l5, %o2
```

we achieve the same effect as three of the above instructions:

```
not     %l5, %l5        !form complement in place
inccc   %l5             !add one to form two's comp.
addcc   %l2, %l5, %o2   !add bits 0 - 31
```

If we use the subcc instruction to subtract the two mid registers:

```
subcc   %l1, %l4, %o1
```

we will perform:

```
not     %l4, %l4
inc     %l4, %l4
addxcc  %l1, %l4, %o1
```

which results in a number too large by one. However, if a carry had been generated by the previous subtraction, this would be the correct result after carry propagation. If a carry were not generated by the previous subtraction, we need to subtract one. This is exactly what the subx instruction does so that we can rewrite the entire three-word subtraction code in a similar form to the addition code:

```
subcc   %l2, %l5, %o2   !subtract bits 0 - 31
subxcc  %l1, %l4, %o1   !subtract bits 32 - 63 + C
subx    %l0, %l3, %o0   !subtract bits 64 - 95 + C
```


4.12.3 Multiplication of Extended Precision Numbers

By keeping track of the binary scaling of each word and remembering that each 32-bit multiplication results in a two-word, 64-bit result, we can also perform multiple-precision multiplication. Consider the case of the multiplication of two 64-bit unsigned numbers. The first number is represented by A_{32} B and the second by C_{32} D where A, B, C, and D are the four registers containing the two numbers and the subscripts refer to the binary scaling. The product, E_{96}, F_{96}, G_{96}, H_{96} may be written as:

		A_{32}	B_{32}	
		C_{32}	D_{32}	×
		BD_{32}	BD	
	BC_{64}	BC_{32}		
	AD_{64}	AD_{32}		
AC_{96}	AC_{64}			+
E_{96}	F_{96}	G_{96}	H_{96}	

This may be translated into the following code making use of .umul to perform 32-bit *unsigned* multiplication. The multiply routine produces 64 bits of result with the high-order part in %o1 and the low-order part in %o0.

```
        .global _main
_main:

        define(A, i0)           !multiplicand
        define(B, i1)
        define(C, i2)           !multiplier
        define(D, i3)

        define(E, l0)           !product
        define(F, l1)
        define(G, l2)
        define(H, l3)

        mov     0x1, %A         !initialize A and B
        mov     0xffffffff, %B

        mov     0x1, %C
        mov     0xffffffff, %D  !initialize C and D

        mov     %B, %o0         !BD
        call    .umul
        mov     %D, %o1

        mov     %o0, %H         !H = BD
```

```
mov     %o1, %G         !G = DB32

mov     %B, %o0         !BC
call    .umul
mov     %C, %o1

addcc   %o0, %G, %G     !G = G + BC32
addx    %o1, %g0, %F    !F = BC64

mov     %D, %o0         !AD
call    .umul
mov     %A, %o1

addcc   %o0, %G, %G     !G = G + AD32
addx    %o1, %F, %F     !F = AD64

mov     %C, %o0         !AC
call    .umul
mov     %A, %o1

addcc   %o0, %F, %F     !F = F + AC64
addx    %o1, %g0, %E    !E = AC96

mov     1, %g1
ta      0
```

Division can also be performed, in a nonrestoring manner, by making use of multiple-precision addition and subtraction.

4.13 Summary

Binary arithmetic was shown to be very simple to implement using elementary logic operations, and and xor in the form of half and full adders. Modulus arithmetic was introduced to handle negative numbers. The diminished radix complement (one's complement for binary numbers) and radix complement (two's complement for binary numbers) were defined. Modulus arithmetic makes use of the top half of the representable states of an n-bit binary number to represent the negative numbers. A two's complement negative number has the most significant bit set. Subtraction may be handled in the same manner as addition, using two's complement arithmetic simplifying hardware requirements for arithmetic logic units.

Two's complement branches were described, used in conjunction with the V, N, and Z condition codes. Handling unsigned numbers was presented in terms of an imaginary high-order bit. Unsigned branches, which tested the C and Z bits,

were presented. A fairly extensive discussion of multiplication was given, as the SPARC architecture does not include a multiply instruction. This section concluded with the SPARC mulscc instruction to provide for multiplication. The section on multiplication was followed by a section on division, introducing nonrestoring division and concluding with an assembly language division routine. The chapter concluded with a section on extended precision arithmetic.

4.14 Exercises

4–1 Write addition and subtraction algorithms for two's complement numbers using the machine logical instructions:

```
and     reg_rs1, reg_or_imm, reg_rd
andn    reg_rs1, reg_or_imm, reg_rd
xor     reg_rs1, reg_or_imm, reg_rd
or      reg_rs1, reg_or_imm, reg_rd
xnor    reg_rs1, reg_or_imm, reg_rd
orn     reg_rs1, reg_or_imm, reg_rd
```

You may not use the add or sub instructions, or their cc versions.

The following addition algorithm is suggested:

```
int add (int a, int b) !addition using logical operations
  {
    int s; !sum
    int c;          !carry

    c = b !initialize carry
    s = a ^ c;          !sum is the xor
    while (c = (a & c) << 1)  !carry is the and of inputs
s = (a = s) ^ c;
    return (s);
  }
```

and for subtraction:

```
int sub (int a, int b) !subtraction using logical operations
  {
    int d; !difference
    int c;          !carry

    c = ~b  !initialize carry to one's comp of b
    d = a ^ c;          !diff is the xor + 1
```

```
      c = (a & c) << 1 | 1; !Note the extra one to make two's comp
      do
        d = (a = d) ^ c;
while (c = (a & c) << 1)!carry is the and of inputs
      return (d);
    }
```

Your program should form the sum of:

07707 and 00101

and the difference of:

00710 and 01010

For both the addition and the subtraction you are to print out the partial sum and carry each iteration through the loop. You are also to print out the result of the addition and subtraction. Note that the input given above is in octal.

4–2 Write an addition algorithm, which also sets the condition codes, for two's complement numbers using the machine logical instructions such as:

```
and     reg_rs1, reg_or_imm, reg_rd
andn    reg_rs1, reg_or_imm, reg_rd
xor     reg_rs1, reg_or_imm, reg_rd
or      reg_rs1, reg_or_imm, reg_rd
xnor    reg_rs1, reg_or_imm, reg_rd
orn     reg_rs1, reg_or_imm, reg_rd
```

You may not use the add or sub instructions, or their cc versions.

Your code is also to set the four condition code bits, N, V, Z, and C as outlined in the definition of addcc instruction given on page 358 of the text. The N bit is to be set if the result is negative; the V bit is to be set, indicating overflow, if the resulting number is too large to store in 32 bits; the Z bit is to be set if the result is zero; the C bit is to be set if a carry occurred on addition and if a carry did not occur on subtraction. You may use the use the logic given on page 358 to set the bits or generate the bits in the course of your algorithm. The condition codes are to be stored in the low-order four bits of a register:

```
define(Z, 8) !'Z = 010'
define(N, 4) !'N = 004'
define(V, 2) !'V = 002'
define(C, 1) !'C = 001'
```

A simplification in the boolean expression for V on page 358 is:


```
V = (r[rs1]<31> & op2<31>) & ~r[rd]<31>) |
    (~(r[rs1]<31> | op2<31>)) & r[rd]<31>)
```

To compute such an expression you can perform logical operations between the registers and then test the sign bit, bit 31. The following two branches will directly test if bit 31 is set of clear:

```
bpos branch if bit 31 clear
bneg branch if bit 31 set.
```

You can run your program with different numbers in gdb by setting a breakpoint at `_main` and then using the print command. For example, to set %l0 to 0x7fffffff and %l1 to -5 you could type:

```
(gdb) p/x $l0 = 0x7fffffff
$1 = 0x7fffffff
(gdb) p/x $l1 = -5
$2 = 0xfffffffb
(gdb)
```

Run you program with four inputs, which demonstrate the setting of the Z, N, V, and C bits. Print out the condition codes and the sum for each pair of numbers.

4–3 Modify the algorithm on page 115 to handle signed integers as well.

4–4 Write an assembly language program to multiply two four-bit unsigned numbers together using no more than five `mulscc` instructions.

4–5 Write an assembly language program to perform the division of two unsigned integers employing the restoring division algorithm.

4–6 Write an assembly language program to perform signed 64-bit multiplication to produce 128 bits of result

4–7 Write an assembly language program to perform unsigned 64-bit division with a 128 bit dividend.

4–8 Divide 17 by 5 using two's complement nonrestoring division. Show all your working and comment the generation of each bit of the quotient.