Chapter 3

Computer Arithmetic

The arithmetic logic unit (ALU) is the part of the processor that performs calculation both the arithmetic and the logic operations. It composed of functional units and registers including some status bit for storing the result of operations such as zero and overflow. The functional units included adder, multiplier and shifter. As an ALU is realised using logic gates, it relies on the computer arithmetic algorithms to perform calculation by repetition such as using multiple add-shifts to do multiplication. This enables complex calculations such as floating point operations possible on an economical hardware.

Number representation

Decimal system

$$A = 1957_{10}$$
$$A = 1 \times 10^3 + 9 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

A is expressed in a decimal number. The base is 10. This representation has 10 symbols 0, 1, 2, … 9 which constitutes *digits*.

Binary system

A number is represented as sum of weights that are a power of 2. The base is 2 and there are two symbols 0, 1 called *binary digits* or *bits*.

$$A = 10101_2$$
$$A = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$A = 2^4 + 2^2 + 2^0 \quad = 21_{10}$$

A number can be represented by *n*-bit in many ways. For an integer, there are unsigned, sign-magnitude and two's complement representation.

unsigned integer

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

An unsigned integer ranges over non negative numbers. For *n*-bit integer its range is $0 \ldots 2^n - 1$.

sign-magnitude

The left most bit is sign, the right most $n-1$ bit is magnitude. It has several drawbacks. First addition and subtraction require special treatment of sign and relative magnitudes. Second, the number zero has two representations +0, –0.

two's complement

We have seen how to represent an unsigned integer but how a negative number can be represent without using sign-magnitude? Suppose we have 3-bit binary $a_2a_1a_0$ which can represent $2^3 - 8$ positive numbers for 000 to 111 (0 to 7). The fourth bit can be introduced to associate with the negative weight -2^3. The 4-bit number can represent 1000_2 (-8_{10}) to 0111_2 ($+7_{10}$). The decimal value is

$$A = a_3 \times -2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

The number is negative is $A_3 = 1$. The properties of this representation are

1 Bit A_3 gives the sign of the equivalent decimal number, A_3 =1 negative, $A_3 = 0$ positive.
2 There is one zero and it is positive.
3 A positive decimal number is changed to a negative number of the same absolute value by inverting each bit followed by adding a 1.

a_3	a_2	a_1	a_0	decimal
1	0	0	0	–8
1	0	0	1	–7
1	0	1	0	–6
1	0	1	1	–5
1	1	0	0	–4
1	1	0	1	–3
1	1	1	0	–2
1	1	1	1	–1
0	0	0	0	+0
0	0	0	1	+1
0	0	1	0	+2
0	0	1	1	+3
0	1	0	0	+4
0	1	0	1	+5
0	1	1	0	+6
0	1	1	1	+7

Figure 3.1 4-bit two complement numbers

Example Convert 110_2 ($+6_{10}$) to a negative number -6_{10}.
0010 inverse to 1001, 1001 plus 1 is $1010_2 = -6_{10}$
This number is called *two's complement* of the original number.

The following expression defines the two's complement representation for both positive and negative numbers. if A is positive, the sign bit (a_{n-1}) is zero. The range of positive number is $0 \ldots 2^{n-2}$. The range of negative number is $-1 \ldots -2^{n-1}$.

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Integer arithmetic

Addition and subtraction

Using two's complement representation, subtraction is performed by adding the two's complement. For example, 5 – 3 = 2, (+5) + (–3) = 2, (0101) + (1101) = **1**0010. The left most bit (carry bit) is overflowed. We ignore the overflow and the result is 0010 = 2. On any addition, the result may be larger than can be held in the word size being used. This condition is called *overflow*. When overflow occurs, the ALU signals the condition codes. The overflow rule is: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Subtraction is achieved using addition. We can demonstrate by the following example. If B = –A, then A + B = A + (–A) = 0. For *n*-bit integer, B is a bitwise complement of A plus 1, that is –A. Let a_n' be a complement of a_n .

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

$$B = -2^{n-1}a_n' + 1 + \sum_{i=0}^{n-2} 2^i a_i'$$

$$A + B = -(a_n + a_n')\,2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i (a_i + a_i')$$

$$= -2^{n-1} + 1 + \sum_{i=0}^{n-2} 2^i$$

$$= -2^{n-1} + 2^{n-1} = 0$$

Multiplication

Multiplication is a complex operation. Multiplication firstly generates partial products, one for each digit in the multiplier, then summed them to produce the final product. Each successive partial product is shifted one position to the left relative to the preceding partial product. The multiplication of a binary number 2^n is accomplished by shifting that number to the left by *n* bits. The multiplication of two *n*-bit integers results in a product of up to 2*n* bits in length.

```
        1 0 1 1 ×       Multiplicand
        1 1 0 1         Multiplier
        1 0 1 1
      0 0 0 0           Partial products
    1 0 1 1
  1 0 1 1
1 0 0 0 1 1 1 1         Product
```

Figure 3.2 Multiplication of unsigned integers

One of the well-known algorithms for two's complement multiplication is Booth's algorithm [BOO51]. Let Q, M, A be three *n*-bit registers, Q stores multiplier, M multiplicand, the result appears in AQ. A concatenates to Q and when shifting right, the least significant bit of A will go to the most significant bit of Q. There is one bit placed to the right of the least significant bit of Q (Q0), designated Q-. Booth's algorithm is as follows:

```
A = 0, Q- = 0, M = multiplicand, Q = multiplier
repeat n times
    if (Q0, Q-) = 01 then A = A + M
                = 10 then A = A – M
    arithmetic shift right A, Q, Q-    {preserve sign bit}
end
```

Note the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of one addition or subtraction per block.

```
        0 1 1 1 ×
        1 1 0 1 (0)
1 1 1 1 1 0 0 1     1-0
0 0 0 0 1 1 1       0-1
1 1 1 0 0 1         1-0
1 1 1 0 1 0 1 1
```

Figure 3.3 example of Booth's algorithm for (7) × (–3) = –21

Why Booth's algorithm work?

Observe that the number to partial product sum can be reduce. Consider a positive multiplier where one contiguous 1s surrounded by 0s. The number of shift-and-add can be reduced by observing that

$$2^{n} + 2^{n-1} + \ldots 2^{n-K} = 2^{n+1} - 2^{n-K}$$

Example

$$M^* (011110) = M^* (2^4 + 2^3 + 2^2 + 2^1) = M * (2^5 - 2^1)$$

The product can be generated by one addition and one subtraction of the multiplicand. Booth's algorithm performs subtraction when first 1 of block is encountered (1-0) and addition when the end of block is encountered (0-1). This scheme extends to any number of blocks of 1s in a multiplier and negative number.

Division

unsigned binary division

The division is based on the long division. It involves repetitive shifting and addition or subtraction. Dividend is examined bit by bit from left to right until it is greater than or equal to the divisor, 0s are placed in the quotient, when it is divisible, 1 is placed in the quotient and the divisor is subtract from the partial dividend. Additional bits from the dividend are appended to the *partial remainder* until the result is greater than or equal to the divisor then the cycle repeat.

```
                          0 0 0 0 1 1 0 1    Quotient
Divisor          1 0 1 1 / 1 0 0 1 0 0 1 1    Dividend
                            1 0 1 1
Partial remainders          0 0 1 1 1 0
                              1 0 1 1
                              0 0 1 1 1 1
                                  1 0 1 1
                                    1 0 0    Remainder
```

Figure 3.4 Division of unsigned binary integers

The algorithm is as follows:

```
{ unsigned integer divide }
A = 0, M = divisor, Q = dividend
repeat n times
        shift left A, Q
        A = A - M
        if A < 0 then Q0 = 0, A = A + M
                else Q0 = 1
end {quotient in Q, remainder in A}
```

two's complement division

This scheme, with some difficulty, can be extended to negative numbers. The divisor must be expressed as 2*n*-bit two's complement number.

```
{ two's complement integer divide }
M = divisor, A Q = dividend
while there are bits in Q
        shift left A Q
        if (M and A have the same sign) then A = A - M
                                        else A = A + M
        if (sign of A not change) or (A = 0 AND Q = 0 ) then Q0 = 1
        if (sign of A change) and (A ≠ 0 OR Q ≠ 0) then Q0 = 0; restore
              the previous A
if (divisor and dividend are not same sign) then two's complement Q
end {quotient is in Q, remainder is in A }
```

```
   A           Q       M = 1101
0 0 0 0    0 1 1 1    Initial value
0 0 0 0    1 1 1 0    Shift
1 1 0 1               Add
0 0 0 0    1 1 1 0    Restore
0 0 0 1    1 1 0 0    Shift
1 1 1 0               Add
0 0 0 1    1 1 0 0    Restore
0 0 1 1    1 0 0 0    Shift
0 0 0 0               Add
0 0 0 0    1 0 0 1    Set Q0 = 1
0 0 0 1    0 0 1 0    Shift
1 1 1 0               Add
0 0 0 1    0 0 1 0    Restore
```

Figure 3.5 example of two's complement division (7) / (−3)

Floating- Point Numbers

Very large and very small numbers can be represent using scientific notation which separately store *significand* and *exponent*, such as $2.14 * 10^{12}$. This allow a range of very large and very small numbers to be represented using only a few digits. In binary numbers, a number is represent in the form:

$$\pm \text{Significand} \times \text{Base}^{\pm \text{Exponent}}$$

This number can be stored in a binary word using three fields: Sign bit, Significand and Exponent. The base is implicit. The exponent can be stored with bias, i.e. a bias is subtracted from the field to get the true value. An example of 32-bit floating-point format is 1 bit sign, 8 bits biased exponent and 23 bits significand. The bias is 128.

$0.11010001 \times 2^{10100}$ = 0 10010100 10100010000000000000000
$-0.11010001 \times 2^{10100}$ = 1 10010100 10100010000000000000000
$0.11010001 \times 2^{-10100}$ = 0 01101100 10100010000000000000000
$-0.11010001 \times 2^{-10100}$ = 1 01101100 10100010000000000000000

Figure 3.6 an example of 32-bit floating-point format

To simplify the operations on floating-point numbers, it is required that they be *normalized* in the form:

$$0.1\text{bbb}. . .\text{b} \times 2^{\pm E}$$

Therefore the left most bit of significand is always 1 and is "implicit" (no need to store this bit).

Range of representable numbers

With the above representation the following ranges of numbers are possible:
Negative numbers between $-(1 - 2^{-24}) \times 2^{127}$ and -0.5×2^{-128}
Positive numbers between 0.5×2^{-128} to $(1 - 2^{-24}) \times 2^{127}$

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(1 - 2^{-24}) \times 2^{127}$, called *negative overflow*
- Negative numbers greater than -0.5×2^{-128} , called *negative underflow*

- Zero
- Positive numbers less than 0.5×2^{-128} , called *positive underflow*
- Positive numbers greater than $(1 - 2^{-24}) \times 2^{127}$, called *positive overflow*

Remember that the maximum number of different values that can be represented with 32 bits is still 2^{32}. The numbers represented in floating-point notation are not spaced evenly along the number line. The possible values get closer together near the origin and farther apart as you move away. This is one of the trade-off of floating-point: Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.

IEEE standard 754

The most important floating-point representation is defined in IEEE Standard 754 [IEE85]. The IEEE standard defines both a 32-bit single and a 64-bit double format. The single format has a sign bit, 8-bit biased exponent, 23-bit significand. The exponent bias is 127. The double format has a sign bit, 11-bit biased exponent, 52-bit significand. The exponent bias is 1023. The implied base is 2. The standard defines two extended formats, single and double, whose exact format is implementation-dependent. The extended formats are to be used for intermediate calculations.

There are some bit patterns that are used to represent special numbers such as zero, plus/minus infinity, NaN (not a number) and denormalized number etc.

numbers	bias exponent	fraction	value
zero	0	0	± 0
infinity	2047	0	$\pm$ infinity
NaN	2047	$\neq 0$	NaN
denormalized	0	f $\neq 0$	$\pm 2^{e-1022}$ (0.f)

Figure 3.7 special numbers of IEEE 754 (double precision)

Floating- Point Arithmetic

For addition and subtraction, it is necessary for both operands to have the same exponent. This may require shifting the radix point to achieve the alignment. The multiplication and division are more straightforward. When the significand

is underflow the rounding operation is required. Likewise when it is overflow the realignment (normalized) is required.

Let x, y be two floating-point numbers; xs, ys be the significands; xe, ye be the exponents. Let xe ≤ ye. The floating-point numbers arithmetic operations:

$$x = xs\ B^{xe}$$

$$y = ys\ B^{ye}$$

$$x + y = (xs\ B^{xe-ye} + ys)\ B^{ye}$$

$$x - y = (xs\ B^{xe-ye} - ys)\ B^{ye}$$

$$x \times y = (xs \times ys)\ B^{xe+ye}$$

$$x / y = (xs / ys)\ B^{xe-ye}$$

Addition and Subtraction

There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros
2. Align the significands
3. Add or subtract the significands
4. Normalized the result

Let msd = most significant digit , S = significand, E = exponent

The addition-subtraction algorithm is as follows:

1. made implicit bit explicit
2. check operand 0
3. align by shifting smaller number to the right (increment its E) until two E are equal
4. check 0
5. add signed S
6. check 0
7. check S overflow if so shift right
8. check E overflow if so report error
9. normalize result, shift S left until msd is not zero, decrement E, E may underflow
10. rounded off the result

Multiplication

The multiplication and division are simpler than addition and subtraction. The multiplication algorithm is as follows:

1. check operand 0
2. xe + ye
3. substract bias
4. check E overflow, underflow
5. sign-magnitude multiply S
6. normalized result and rounded (E may underflow)

Division

1. check operand 0
2. xe – ye
3. add bias
4. check E overflow, underflow
5. divide S
6. normalized and rounded result

Precision considerations

Guard bits

For the floating-point operations the significands are loaded into the registers. The length of the register is almost always greater than the length of significand plus an implied bit. The register contains an additional bit, called *guard bits*, the are used to pad out the right end of the significand with 0s. The purpose is to prevent the lost of least significant bit when one operand must be shifted right during floating-point operation. As seen from the following example: a subtraction without and with guard bits.

```
Without guard bit
  1.000 . . . 00 × 2
  0.111 . . . 11 × 2  –
= 0.000 . . . 01 × 2
= 1.000 . . . 00 × 2^-22
```

With guard bits

$$
\begin{array}{l}
\quad 1.000\ldots00\ \mathbf{0000} \times 2 \\
\quad 0.111\ldots11\ \mathbf{1000} \times 2 \quad - \\
\hline
= 0.000\ldots01\ \mathbf{1000} \times 2 \\
= 1.000\ldots00\ \mathbf{0000} \times 2^{-23}
\end{array}
$$

Rounding

The rounding policy affects the precision of the result. IEEE standard lists four approaches:

- Round to nearest – to the nearest representable number
- Round toward positive infinity
- Round toward negative infinity
- Round toward 0 (truncated)

Round to the nearest is the default rounding mode in the standard. The rounding to plus and minus infinity is useful in implementation of interval arithmetic. In the interval arithmetic an upper bound and lower bound on the correct answer are kept. If the range between the upper and lower bounds is sufficiently narrow, it indicates that a sufficiently accurate result is obtained.

Denormalized number

Denormalized numbers are included in IEEE 754 to handle E underflow, the result is denormalized by right-shifting S and increment E until E is within representable range. This method is also referred to as "gradual underflow" [COO81].

References

[BOO51] Booth, A. "A signed binary multiplication technique." Quarterly Journal of Mechanical and Applied Mathematics, vol. 4, pt. 2, 1951.

[COO81] Coonen, J. "Underflow and Denormalized numbers", IEEE Computer, March 1981.

[GOL91] Goldberg, D., "What every computer scientist should know about floating-point arithmetic,", ACM Computing Surveys 23:1, 5-48.

[IEE85] Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.

[KNU81] Knuth, D. The art of computer programming, Volume 2: seminumerical algorithms, Addison-Wesley, 1981.

[OMO94] Omondi, A. Computer Arithmetic Systems: Algorithms, architecture and implementations, Prentice-Hall, 1994.

[SWA90] Swartzlander, E., ed., Computer arithmetic, Volumes I and II, IEEE Computer society press, 1990.