Computer Arithmetic

© 1997 John D. Carpinelli, All Rights Reserved

Outline

- Introduction
- Numeric formats
- Arithmetic algorithms:
  - Signed-Magnitude format
  - Signed-2's Complement format
  - Floating Point format
  - BCD format
- Summary

© 1997 John D. Carpinelli, All Rights Reserved

Computers perform a large number of arithmetic operations on numbers in a variety of formats. We examine the algorithms and the hardware to implement addition, subtraction, multiplication and division for several different data formats. Signed-magnitude and signed-2's complement are two formats used to store integer (fixed-point) values. Floating point, as its name implies, is used to store floating point numbers. BCD, or binary coded decimal, is used to store decimal values.

Signed-Magnitude format

Express a number as A_sA

- **A_s = One-bit sign bit**
- **A = n-bit mantissa**
- **Negative mantissas are not expressed in 2's complement**

© 1997 John D. Carpinelli, All Rights Reserved

In signed-magnitude format, numbers are stored in two parts. The first part is a 1-bit sign, which is zero for positive numbers (or zero) and one for negative numbers. The remaining bits comprise the mantissa, or magnitude portion of the value. The mantissa is represented the same, regardless of the value of the sign bit. For example, +5 = 0 0101 and -5 = 1 0101.

Signed-2's Complement format

Express a number as A_sA

- A_s = One-bit sign bit
- A = n-bit mantissa
- Negative mantissas are expressed in 2's complement

© 1997 John D. Carpinelli, All Rights Reserved

Signed-2's complement is the same as signed-magnitude, except that, for negative numbers, the mantissa is stored in 2's complement format. Just as before, +5 = 0 0101; however, -5 = 1 1011.

Floating Point format

Express a number as $A_sAa = A_s(A * r^a)$

- A_s = One-bit sign bit
- A = n-bit mantissa
- a = exponent
- Negative mantissas are not expressed in 2's complement

© 1997 John D. Carpinelli, All Rights Reserved

Floating point format is used to express numbers which may have a fractional component. As with signed-magnitude format, it contains a one-bit sign bit and a mantissa which is not expressed in 2's complement form when negative. The mantissa is considered to be in normal form, that is, in the form of a fraction with no leading zeroes. There is also an exponent which may be positive or negative.

Floating point numbers may be in any base r. For example, in base 2, +5 is expressed as .101 * 2^3; the sign bit is zero, the mantissa is 101 and the exponent is 3, or 011. If r=10, as is the case for decimal numbers, +5 is expressed as .5 * 10^1.

BCD format

Express a number as A_sA

- A_s = One-bit sign bit
- A = n-bit mantissa, where each 4 bits of A represent one BCD digit

© 1997 John D. Carpinelli, All Rights Reserved

Binary Coded Decimal, or BCD, format is used to store numbers as decimal values. Every four bits of the mantissa encodes a single decimal digit. Digits 0 to 9 are encoded as 0000 to 1001; 1010 to 1111 are not used. For example, the decimal value 27 is encoded as 0010 0111. We'll examine how this affects arithmetic algorithms and hardware later in this module.

Comparing relative magnitudes

To compare A and B, subtract B from A using full adders,using complements, forming D = A - B = A + $\overline{B}$ + 1:

- **If A > B, then D ≠ 0, carry = 1**
- **If A = B, then D = 0, carry = 1**
- **If A < B, then carry = 0**

© 1997 John D. Carpinelli, All Rights Reserved

Comparing two values is a fundamental operation performed by several arithmetic algorithms. Consider two numbers, A and B. To compare them, we form D = A - B = A + /B + 1 and note the result. If D is not zero and the carry out is 1, then A > B. If D is zero and the carry out is one, then A = B. If the carry out is zero, then A < B. For example, consider these three cases:

A = +5 = 0101; B = +3 = 0011; /B + 1 = 1101; A + /B +1 = 1 0010

A = +3 = 0011; B = +3 = 0011; /B + 1 = 1101; A + /B +1 = 1 0000

A = +3 = 0011; B = +5 = 0101; /B + 1 = 1011; A + /B +1 = 0 1110

Comparing signed-magnitude numbers

$X = X_s X_m$; $Y = Y_s Y_m$

- $X=Y$ if $X_s = Y_s$ and $X_m = Y_m$
- $X>Y$ if $(\overline{X_s} Y_s) \vee (\overline{X_s}\,\overline{Y_s} \wedge X_m>Y_m)$
  $\vee (X_s Y_s \wedge X_m<Y_m)$
- $X<Y$ if $(X_s \overline{Y_s}) \vee (\overline{X_s}\,\overline{Y_s} \wedge X_m<Y_m)$
  $\vee (X_s Y_s \wedge X_m>Y_m)$

© 1997 John D. Carpinelli, All Rights Reserved

We can use this comparison procedure to compare signed-magnitude numbers. Here, we must take into account the sign bits, as they are not reflected in the magnitude portion of the numbers. Two numbers, X and Y, are equal if and only if their sign bits are the same and their magnitudes are the same. X is greater than Y if (i) X is positive and Y is negative; (ii) X and Y are positive and the magnitude of X is greater than the magnitude of Y; or (iii) X and Y are negative and the magnitude of X is less than the magnitude of Y. The latter case would occur, for instance, if $X = -3$ and $Y = -5$. The conditions for which $X < Y$ are similar.

Signed-Magnitude addition and subtraction

See table 10.1, p. 335 of the textbook.

© 1997 John D. Carpinelli, All Rights Reserved

This table lists the eight possible cases for adding and subtracting two signed-magnitude numbers based on their signs and relative magnitudes. First, we will look at the addition operations, then the subtraction operations and finally the hardware necessary to implement both. Of particular note is that the magnitude of the result can have only one of three values, A+B, A-B or B-A, which, as we will see, is /(A-B) + 1.

Addition procedure

Forming $X = A_sA + B_sB$:

- If $A_s = B_s$, $X = A_s, A+B$
- If $A_s \neq B_s$ and $A \neq B$, $X = X_s$,(MAX(A,B) - MIN(A,B)), where X_s=(MAX(A,B))$_s$
- If $A_s \neq B_s$ and $A = B$, $X = +0$

© 1997 John D. Carpinelli, All Rights Reserved

First let's look at the top half of the table, which lists the addition operations. When A and B have the same sign, we simply add the magnitudes, regardless of the sign. The sign of the result is the same as the sign of the two numbers being added.

When the two numbers have different signs, we must consider their relative magnitudes in generating the result. Whether A>B or A<B, the magnitude of the result is the larger magnitude less the smaller magnitude, either A-B or B-A. The sign of the result is the sign of the greater value. If A=B, the result is always +0. (A-B=0. As we will see, this is used because we need hardware to generate A-B anyway, so we can use that hardware here as well.)

Subtraction procedure

Forming $X = A_sA - B_sB$:

- If $A_s \neq B_s$, $X = A_s,A+B$
- If $A_s = B_s$ and $A \neq B$, $X = X_s,(MAX(A,B) - MIN(A,B))$, where $X_s=(MAX(A,B))_s$
- If $A_s = B_s$ and $A = B$, $X = +0$

© 1997 John D. Carpinelli, All Rights Reserved

Now let's look at the subtraction operations. This time we add the magnitudes if the signs are not the same. For example, (+A) - (-B) is equal to (+A) + (+B). Thus, each row in the lower half of the table has an analogous row in the upper half. If the signs are the same, we subtract the lesser magnitude from the greater magnitude, just as before. Again, the sign of the result is that of the number with the greater magnitude. Zero results occur as before for A=B; the sign in this case is always positive.

Hardware requirements

- One adder to form A+B
- Two subtractors to form A-B and B-A
- One comparator to determine if A>B, A=B or A<B
- One XOR gate to determine if $A_s = B_s$ or $A_s \neq B_s$
- ***We can reduce this by using 2's complements***

© 1997 John D. Carpinelli, All Rights Reserved

Given these possible results, we need the hardware listed above. Note that we will use 2's complement addition and negation to reduce these requirements. As will be seen shortly, we will get rid of the subtractors and the comparator.

Hardware configuration

See figure 10.1, p. 337 of the textbook.

© 1997 John D. Carpinelli, All Rights Reserved

Here is the hardware to implement addition and subtraction of signed-magnitude numbers regardless of their signs. This is very similar to the hardware seen in previous modules. If M=0 the parallel adder forms A+B; if M=1 it forms A+/B+1, or A-B. M is defined as As xor Bs xor OP, where OP=0 for addition and 1 for subtraction.

Addition/subtraction procedure

- When M=0: A_s should retain its previous value and the output of the parallel adder is the correct magnitude
- When M=1:
  - If A>B, A_s and magnitude are correct (E=1)
  - If A=B, $A_s \leftarrow 0$ and magnitude is correct (0)
  - If A<B, $A_s \leftarrow \overline{A_s}$ and magnitude must get 2's complement of value generated (E=0)

© 1997 John D. Carpinelli, All Rights Reserved

The value M will determine how the operations should proceed. Reviewing the table, it can be seen that M=0 corresponds to the case where we must generate the magnitude as A+B, which is exactly what the hardware will do. In forming the result AsA = AsA + BsB, this generates the correct magnitude; the value in As should also remain unchanged, so we are done.

When M=1 we must consider the relative magnitudes of A and B. M=1 causes the parallel adder to generate A+/B+1, or A-B. It also provides important information about the relative magnitudes as it performs the comparison operation discussed earlier. If the carry bit is 1 and the magnitude is not zero, A>B and the magnitude is correct (i.e. we wanted A-B); the original sign bit in As is also correct. If the carry bit is 1 and the magnitude is zero, A=B. Again, the magnitude is correct, but As may or may not be correct. Rather than checking it explicitly, it is easier just to set it to zero, for positive. Finally, if the carry bit is zero, A<B and the magnitude is incorrect. We have A-B but we need B-A. To negate a binary value, we simply take the 2's complement, so we take the 1's complement and add 1. The sign bit is also incorrect in this case and must be complemented.

For all of these operations, we must also set the overflow flag, AVF. Overflow can only occur when the magnitudes are added and we generate a carry out.

Addition/subtraction algorithm

T0:	$EA \leftarrow A + (B \oplus M) + M$
T1:	$AVF \leftarrow E \wedge \overline{M}$
$\overline{E}$MT1:	$A \leftarrow \overline{A}$
$\overline{E}$MT2:	$A \leftarrow A + 1,\ A_s \leftarrow \overline{A_s}$
EMT2:	$A_s \leftarrow A_s \wedge (\vee/A)$

© 1997 John D. Carpinelli, All Rights Reserved

We have taken these concepts and encoded them into a microcoded algorithm. In T0 we form either A+B or A-B, depending on the value of M. We store the result, with carry out, in E&A. During T1, we check for overflow and set AVF accordingly. If E=0 and M=1, signifying that we formed A-B and that B>A, we begin the process of forming the 2's complement of the result. During T1, under these circumstances, we form the 1's complement of the result. Under the same conditions during T2, we add 1 to the result, completing the negation; we also invert the sign bit, as is required. Finally, when E=1 and M=1, we set As to zero if and only if A=0, forming +0 when required.

Example: +5 + (-3)

Note: M = 1

T0: $EA \leftarrow A + (B \oplus M) + M$

$= 0101 + 1100 + 1 = 10010$

T1: $AVF \leftarrow E \wedge \overline{M} = 1 \wedge 0 = 0$

$\overline{E}$MT1: $A \leftarrow \overline{A}$ (not done, E=1)

$\overline{E}$MT2: $A \leftarrow A+1,\ A_s \leftarrow \overline{A_s}$ (not done, E=1)

EMT2: $A_s \leftarrow A_s \wedge (v/A)\ (A_s = 0)$

© 1997 John D. Carpinelli, All Rights Reserved

Let's look at a couple of examples. First, consider the case where AsA = +5 = 0 0101 and BsB = -3 = 1 0011. We are adding these values, so OP=0 and M=1. During T0 we form A+/B+1 = A-B. There is no overflow, so AVF is set to zero during T1. E is 1, so the negation is not implemented. Finally, the sign bit remains unchanged at zero. Our result is AsA = 0 0010 = +2.

Example: +3 + (-5)

Note: M = 1

T0: $EA \leftarrow A + (B \oplus M) + M$

$= 0011 + 1010 + 1 = 01110$

T1: $AVF \leftarrow E \wedge \overline{M} = 0 \wedge 0 = 0$

$\overline{E}MT1$: $A \leftarrow \overline{A} = 0001$

$\overline{E}MT2$: $A \leftarrow A+1 = 0010$, $A_s \leftarrow \overline{A_s} = 1$

EMT2: $A_s \leftarrow A_s \wedge (\vee/A)$ (not done, E = 0)

© 1997 John D. Carpinelli, All Rights Reserved

This example illustrates the case where B>A. During T0 we again form A+/B+1. This time, however, E=0, signifying that B>A. In T1, AVF is cleared. For this data, E=0 and M=1, so we perform the negation of the magnitude of the result. We complement the result during T1 and increment it during T2; we also invert the sign during T2. Since E=0, the last statement is not performed. Our net result, AsA = 1 0010 = -2, is correct.

Signed-2's Complement addition/subtraction

Forming $X = A_sA + B_sB$ or $X = A_sA - B_sB$:

If addition, add the two numbers directly

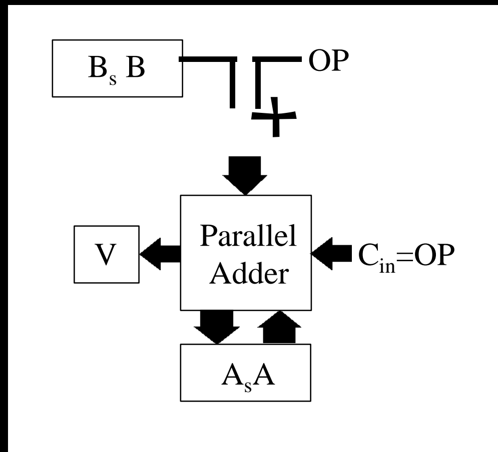
If subtraction, subtract using 2's complement addition

In either case, $V \leftarrow$ overflow

© 1997 John D. Carpinelli, All Rights Reserved

The signed-2's complement addition and subtraction operations are much simpler. Regardless of their signs, the two numbers are added by directly adding the two values. Subtraction is performed by adding the first value to the 2's complement of the second. An overflow flag V gets the result of the overflow.

Hardware configuration



© 1997 John D. Carpinelli, All Rights Reserved

The hardware to implement this is virtually the same as presented in module 1. Again, OP=0 for addition and OP=1 for subtraction.

Signed-Magnitude multiplication: shift/add methodology

- Multiplication is performed using a shift/add methodology

Example (25*17):

```
  25
  17
 ---
 175
 25
 ---
 425
```

© 1997 John D. Carpinelli, All Rights Reserved

Multiplication can be performed using a shift/add strategy, similar to how we multiply decimal nubmers. To multiply two numbers, we form partial products and add the results. In this example, we first multiply 25*7, then 25*1. Note that the results 25*1=25 is shifted one position to the left. This is necessary to reflect the fact that the 1 is one position to the left of the 7. To implement shift-add multiplication, we will generate each partial result and then shift the running total one position to the right. This simplifies the design of the hardware significantly.

Register specification

A&Q ← B * Q

Sign bit is set to $B_s \oplus Q_s$

© 1997 John D. Carpinelli, All Rights Reserved

We will form the product AR&QR = BR*QR, where each register stores numbers in signed-magnitude format. Regardless of their magnitudes, the sign bit can be set immediately.

Note that this algorithm and hardware design assume that neither BR nor QR contains zero. In fact, we would simply add a step at the beginning of the algorithm to check for those cases and, if true, set the result to zero and exit.

Hardware configuration

See figure 10.5, p. 341 of the textbook.

© 1997 John D. Carpinelli, All Rights Reserved

This figure shows the hardware needed to implement the shift-add multiplication algorithm. Notice that A and Q contain the running total. For this algorithm, the "complementer and parallel adder" should be only a parallel adder.

Signed-Magnitude multiplication algorithm

1. $A_s \leftarrow B_s \oplus Q_s$, $Q_s \leftarrow B_s \oplus Q_s$, $A \leftarrow 0$, $E \leftarrow 0$, $SC \leftarrow n - 1$
2. IF $Q_0 = 1$ THEN $EA \leftarrow A+B$
3. shr(EAQ), $SC \leftarrow SC - 1$
4. IF ($SC \neq 0$) THEN GOTO 2

© 1997 John D. Carpinelli, All Rights Reserved

Here is the algorithm to implement shift-add multiplication. In step 1, we perform initialization by setting the sign of the result (both sign bits are set), setting the running total to zero and setting the loop counter to n-1. (Each register contains n bits, one for the sign and n-1 for the mantissa.)

The remainder of the algorithm performs the shift-add loop n-1 times, once for each bit of the mantissa. In step 2, we check the least significant bit of Q. If it is 1, we perform the addition; if it is zero we don't. In the decimal example, we had to multiply each digit. Here, the digit is a single bit, either zero or one, so we don't have to multiply; we either add or don't add. Step 3 shifts the running total one position to the right. It also shifts the multiplier one position to the right so that the next bit is used during the next loop iteration. SC is also decremented here. In step 4, if we are not done, we loop back.

Example

(-13) * (+10) = 1101 * 1010 (ignore signs):

```
    1101
    1010
    ----
    0000
   1101
   -----
   11010
  0000
  ------
  011010
 1101
 -------
10000010
```

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example. If we performed the multiplication logically, we would generate this series of partial results, and the final result.

Example

B_sB = 11101 Q_sQ = 01010

Step	Action	E	A	Q
1	$A_s/Q_s \leftarrow 1$, $SC \leftarrow 4$	0	0000	1010
2,3,4	$SC \leftarrow 3$, shr(EAQ)	0	0000	0101
2	$EA \leftarrow A+B$	0	1101	0101
3,4	$SC \leftarrow 2$, shr(EAQ)	0	0110	1010

© 1997 John D. Carpinelli, All Rights Reserved

Here we step through the algorithm. In step 1 we set the sign bits to 1 since the result will be negative. We also set the running total, underlined throughout this example, to zero and set the loop counter to 4.

During the first iteration of this loop, we do not perform the addition. We simply shift EAQ one position to the right and decrement SC. During the second iteration we do perform the addition prior to the shift and counter update.

Example (continued)

B_sB = 11101 Q_sQ = 01010

Step	Action	E	A	Q
2,3,4	SC ← 1, shr(EAQ)	0	0011	0101
2	EA ← A+B	1	0000	0101
3,4	SC ← 0, shr(EAQ)	0	1000	0010

Result: (-13)(+10) = -130*

© 1997 John D. Carpinelli, All Rights Reserved

We continue with the remaining two iterations, not adding in the third and adding in the fourth. The final magnitude is stored in A&Q.

Notice in this algorithm that the running total moves one more bit into Q for each iteration. Simultaneously, we finish with one bit of Q which is shifted out. This is the most efficient hardware configuration possible for this algorithm.

Signed-2's Complement multiplication: Booth's algorithm

Instead of adding for each value in a string of 1's, subtract for the first one and add for the last one

e.g. 111111 = 1000000 - 0000001

© 1997 John D. Carpinelli, All Rights Reserved

For signed-2's complemetn numbers, we can make use of the fact that a string of 1's may be treated as the difference of two values, regardless of how many 1's are in the string. For example, 111111 = 1000000 - 0000001. In Booth's algorithm for multiplying signed-2's complement numbers, we perform one addition and one subtraction for each string of 1's.

Hardware configuration

See figure 10.7, p. 344 of the textbook.

© 1997 John D. Carpinelli, All Rights Reserved

Here is the hardware to implement Booth's algorithm. Unlike the previous shift-add algorithm, we may perform addition and subtraction, so we must have a "complementer and parallel adder," not just a parallel adder.

Of particular note is the extra bit Q[n+1]. First of all, the author reverses his usual notation in which the least significant bit is bit 0. Regardless of the nomenclature, this extra bit is needed to tell if a string of 1's has begun or ended (or neither).

Signed-2's Complement multiplication algorithm

$AC\&QR \leftarrow BR * QR$

1. $AC \leftarrow 0,\ Q_{n+1} \leftarrow 0,\ SC \leftarrow n$
2. IF $(Q_n \oplus Q_{n+1} = 1)$ THEN
   $AC \leftarrow AC+(BR \oplus Q_n) + Q_n$
3. $ashr(AC\&QR\&\ Q_{n+1}),\ SC \leftarrow SC - 1$
4. IF $(SC \neq 0)$ THEN GOTO 2

© 1997 John D. Carpinelli, All Rights Reserved

Here is Booth's algorithm. Note that AC, QR and BR contain both the sign and magnitude portions of their respective values. In step 1, we perform initialization. We set the running total to zero and set Q[n+1] to zero to signify that no string of 1's has started yet. We also initialize the loop counter. Here n includes both the sign bit and the n-1 bits of the mantissa.

As before, the remaining steps comprise the loop. In step 2 we check to see if a string of 1's has either begun or ended. The result of the exclusive-or operation will be 1 in either case. If this is true and Qn=1, we are starting a string of 1's and must perform AC=AC-BR, which is implemented as a 2's complement addition. If Qn=0 we are completing a string of 1's and must form AC=AC+BR. Steps 3 and 4 perform the usual shift and loop control. Notice that we perform an arithmetic shift to preserve the sign bit.

Every addition balances out a subtraction, but the reverse is not necessarily true. If QR is negative, the final subtraction will never be balanced out by an addition operation.

Example

QR = 10011   BR = 01111 $\overline{BR} + 1 = 10001$

Step	AC	QR	Q_{n+1}	SC
1	<u>00000</u>	10011	0	5
2	<u>10001</u>	10011	0	5
3,4	<u>11000</u>	<u>1</u>1001	1	4
2,3,4	<u>11100</u>	<u>01</u>100	1	3

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example. In step 1 we zero out the running total and Q[n+1] and initialize the loop counter. The first iteration begins a string of 1's, so we add /BR+1 to the running total. We perform the required shift and process the loop counter.

During the second iteration we are still within the string of 1's, so we neither add nor subtract during this iteration.

Example (continued)

Step	AC	QR	Q_{n+1}	SC
2	01011	01100	0	3
3,4	00101	10110	0	2
2,3,4	00010	11011	0	1
2	10011	11011	0	1
3,4	11001	11101	1	0

Result: *(-13)*(+15) = -195*

© 1997 John D. Carpinelli, All Rights Reserved

During the third iteration the string of 1's ends, so we perform the addition. In the fourth iteration, we have not yet started a new string of 1's, so we do not add nor subtract.

Finally, in the last iteration, a new string of 1's begins, even though it is only one bit long, so we perform the subtraction. The algorithm ends and we never add this value back, which is OK since our result must be negative for this data.

Signed-Magnitude division: restoring algorithm

- Shift-subtract methodology
- Perform comparison by subtraction
- If incorrect, restore running dividend
- If correct, update quotient

© 1997 John D. Carpinelli, All Rights Reserved

Whereas the signed-magnitude multiplication algorithm employed a shift-add methodology, the division algorithm uses a shift-subtract methodology. For each iteration, it subtracts the divisor from the running dividend. If the result is negative, i.e. the divisor didn't go into the dividend, it restores the original dividend value. If the result is not negative, i.e. it does go into the dividend, it updates the quotient accordingly.

Signed-Magnitude division: numeric example

```
          01011
1001) 01101010
      0000
       1101
       1001
        1000
        0000
        10001
         1001
         10000
          1001
          0111
```

© 1997 John D. Carpinelli, All Rights Reserved

Consider this numeric example. For each iteration, the divisor goes into the dividend either one or zero times. In the restoring algorithm, we will perform the subtraction every time and, if necessary, add the result back.

Signed-Magnitude division algorithm

$Q \leftarrow A\&Q \text{ div } B,\ A \leftarrow \text{remainder}$

1. $Q_s \leftarrow A_s \oplus B_s,\ EA \leftarrow A+\overline{B}+1,\ SC \leftarrow n-1$
2. $EA \leftarrow A+B,\ DVF \leftarrow E$, IF $(E=1)$ THEN {overflow}
3. shl(EAQ)
4. IF $(E=0)$ THEN $EA \leftarrow A+\overline{B}+1$,
   IF $(E=1)$ THEN $A \leftarrow A+\overline{B}+1$
5. IF $(E=1)$ THEN $Q_n \leftarrow 1$, IF $(E=0)$ THEN $EA \leftarrow A+B$,
   $SC \leftarrow SC - 1$
6. IF $(SC \neq 0)$ THEN GOTO 3

© 1997 John D. Carpinelli, All Rights Reserved

Here is the restoring division algorithm for signed-magnitude numbers. The first two steps do some initialization, such as setting the sign of the result and setting the loop counter. The first step also performs a subtraction, A-B. In fact, this is done to determine whether an overflow exists. If the carry bit is set to one as a result of this operation, the final result will not fit in a single register; an overflow will occur. Step 2 takes note of this and processes it accordingly. This step also restores the original value in EA, just in case there is no overflow and the algorithm must proceed.

Steps 3 to 6 implement the shift-subtract-restore loop. The dividend is stored in registers E&A&Q. This value is shifted left one position in step 3. In step 4, the algorithm subtracts B from A to test if A goes into B. The carry flag E will be set to 1 if this is the case. For this reason, we must process two distinct situations. If E=1, the value in E&A must be greater than the value in B, since E&A is one bit longer than B and has a leading 1 in this case. Otherwise we perform the comparison and set E to see if A>=B.

In step 5 we do one of two things. If E is 1, we were correct in performing the subtraction, so we update the running quotient. If not, we restore the original value into E&A. Regardless, we decrement the loop counter and, if not done, jump back in step 6.

Signed-Magnitude division example

A = 1001, Q = 0111, B = 1101, $A_s = B_s = Q_s = 0$

Step	Notes	E	A	Q	SC
1	$Q_s \leftarrow 0$	0	1100	0111	4
2	DVF $\leftarrow$ 0	1	1001	0111	4
3		1	0010	1110	4
4		1	0101	1110	4
5,6	Divides	1	0101	111<u>1</u>	3

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example. The first two steps determine that there is no overflow. The first iteration of the loop performs the shift and subtract, which sets E=1. This tells us that the subtraction should have occurred, so we set the most significant bit of the quotient to 1.

Signed-Magnitude division example (continued)

Step	Notes	E	A	Q	SC
3		0	1011	11<u>10</u>	3
4		0	1110	11<u>10</u>	3
5,6	Restore	0	1011	11<u>10</u>	2
3		1	0111	1<u>100</u>	2
4		1	1010	1<u>100</u>	2
5,6	Divides	1	1010	1<u>101</u>	1

© 1997 John D. Carpinelli, All Rights Reserved

The next iteration sets E=0, which means that the subtraction must be undone. In step 5 we restore the value. We proceed through the third iteration, which does perform the subtraction as required.

Signed-Magnitude division example (continued)

Step	Notes	E	A	Q	SC
3		1	0101	1010	1
4		1	1000	1010	1
5,6	Divides	1	1000	1011	0

Result: ***151 ÷ 13 = 11 R 8***

© 1997 John D. Carpinelli, All Rights Reserved

The final iteration also results in a successful subtraction. The quotient ends up in register Q and the remainder in register A.

Non-restoring algorithm

- Similar to the restoring algorithm, except the magnitudes of A and B are compared prior to subtraction

© 1997 John D. Carpinelli, All Rights Reserved

The non-restoring algorithm does not perform the subtraction unless it is required. Instead, it compares the two values using a comparator first and then, if A>=B, it performs the subtraction.

Signed-2's Complement division

- Convert all numbers to positive values (which are the same in signed-magnitude and signed-2's complement)
- Call the signed-magnitude algorithm to perform the division
- Convert the result back to signed-2's complement form if it is negative

© 1997 John D. Carpinelli, All Rights Reserved

To implement signed-2's complement division, we're going to cheat a little bit. First we will convert all of the numbers to positive values, which are the same in both signed-magnitude and signed-2's complement format. Then we will call the signed-magnitude algorithm as a subroutine to perform the division. Finally we will convert the result back to signed-2's complement format if it is negative. Again, if it is positive the values are represented the same and no final conversion is necessary.

Signed-2's Complement division algorithm

$QR \leftarrow AC\&QR \div BR$, $AC \leftarrow$ remainder

1. $Q_s \leftarrow A_s \oplus B_s$, $SC \leftarrow n-1$
2. $AQ \leftarrow (AQ \oplus A_s)+A_s$, $BR \leftarrow (BR \oplus B_s)+B_s$
3. CALL signed-magnitude algorithm
4. $Q \leftarrow (Q \oplus Q_s)+Q_s$, $A \leftarrow (A \oplus A_s)+A_s$,
   $BR \leftarrow (BR \oplus B_s)+B_s$

© 1997 John D. Carpinelli, All Rights Reserved

Here is the algorithm to divide signed-2's complement numbers. The first step performs the usual bookkeeping, setting the sign of the result and initializing the loop counter. The second step converts each number to a positive value. If the associated sign bit is zero, indicating that the number is already positive, it simply reloads the same value back into the registers. If the sign bits are one, this step performs a 2's complement on the data, thus negating it.

The third step calls the signed-magnitude division algorithm as a subroutine. This algorithm returns the quotient in register QR and the remainder in AC. Finally, step 4 converts the value to 2's complement format if it is negative. This step also restores BR to its original value, if necessary.

Signed-2's Complement division example

Example: -160 ÷ 12, BR = 01100,
AC&Q = 101100000,

STEP	ACTION
1	$Q_s \leftarrow 1$, $SC \leftarrow 4$
2	AQ ← 10100000, B ← 1100
3	A ← 0100, Q ← 1101
4	A ← 1100, Q ← 0011

Result: -160¸ 12 = -13 R -4

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example. The first step sets the sign of the result to 1, since a negative number divided by a positive number must be negative, regardless of the magnitudes. The second step converts AQ to + 160; B retains its value of +12. The third step calls the signed-magnitude routine, which returns the values Q=+13 and A=+4. Finally, the fourth step converts the returned values to -13 and -4.

Floating point addition and subtraction

1. Check for zeros
2. Align mantissas
3. Add/subtract mantissas
4. Normalize result

© 1997 John D. Carpinelli, All Rights Reserved

To add or subtract floating point numbers, we follow this procedure, which was first presented in the pipeline example in the previous module. First, we check to see if either number is zero. If so, we form the proper result and exit. This is consistent with earlier discussion that zero must be treated as a special case. Then we align the mantissas and perform the addition or subtraction. Finally, we normalize the result if necessary.

Floating point addition and subtraction algorithm

AC $\leftarrow$ AC +/- BR (OP=0 for +, OP=1 for -)

1. IF (BR=0) THEN {DONE}
2. IF (AC=0) THEN $A_s \leftarrow B_s \oplus OP$,
   AC $\leftarrow$ BR, {DONE}
3. IF (a $\neq$ b) THEN
   {IF (a>b) THEN (shr B, b $\leftarrow$ b+1, goto 3),
   IF (a<b) THEN (shr A, a $\leftarrow$ a+1, goto 3)}

© 1997 John D. Carpinelli, All Rights Reserved

The algorithm shown on this slide and the next slide implement the floating point addition and subtraction process described on the previous slide. The first two steps check for zeros. If BR is zero, AC should not be changed. If BR is not zero and AC is zero, then AC gets either BR or /BR, depending on the value of OP and the sign of B.

Step 3 aligns the mantissas. It checks to see which exponent is greater, shifts the lesser exponent's mantissa one position to the right, increments its exponent and loops back. This is repeated until the exponents are the same, or the numbers are aligned.

Floating point addition and subtraction algorithm (continued)

4. $EA \leftarrow A + (B \oplus B_s \oplus OP) + (B_s \oplus OP)$,
   IF $(A_s \oplus B_s \oplus OP) = 0$ THEN GOTO 8
5. $A \leftarrow (A \oplus \overline{E}) + \overline{E}$, $A_s \leftarrow (A_s \oplus \overline{E})$
6. IF $(A=0)$ THEN $(AC \leftarrow 0$, DONE)
7. IF $(A_1=0)$ THEN (shl A, $a \leftarrow a-1$, goto 7),
   ELSE {DONE}
8. IF $(E=1)$ THEN (shr EA, $a \leftarrow a+1$, DONE)

© 1997 John D. Carpinelli, All Rights Reserved

Step 4 adds or subtracts the two mantissas, as appropriate. If the values were added, then there cannot be a sign reversal, so we proceed directly to step 8 and normalize the result. If the values were subtracted, then it is possible that the value generated is in fact the negative of the value desired. In step 5, we negate this value by performing a 2's complement operation. In step 6, we check to see if we have generated a zero result, and process it if we did. Step 7 normalizes the result in case it has leading zeroes. Steps 6 and 7 each terminate the algorithm if activated, since two numbers which were subtracted cannot generate a result of 1 or more.

Floating point addition and subtraction example

$A_sAa = .101 * 2^2$ + $B_sBb = + .111 * 2^1$

STEP	ACTION
1,2	------------
3	$B \leftarrow 0111$, $b \leftarrow 2$
4	$EA \leftarrow 10001$ (= 1.0001), GOTO 8
8	$A \leftarrow 10001$ (= .10001), $a \leftarrow 3$

© 1997 John D. Carpinelli, All Rights Reserved

In this example, we add 2 1/2 and 1 3/4. Steps 1 and 2 find no zero operands. Step 3 normalizes b from .111 * 2^1 to .0111 * 2^2. Step 4 adds the two mantissas, generating a carry out. Since we added the values, we go directly to step 8, where we normalize the result, converting it from 1.0001 * 2^2 to .10001 * 2^3.

Floating point multiplication

1. Check for zeros
2. Add exponents
3. Multiply mantissas
4. Normalize product

© 1997 John D. Carpinelli, All Rights Reserved

To multiply two floating point numbers, we must follow this procedure. First we check for zeros. If either operand is zero, the result must be zero. If not, we add the exponents and multiply the mantissas. We do not need to align the data for multiplication. Finally, we normalize the result, if necessary. In this case, each binary mantissa is of the form .1---, so the result must be between 0.01 and .111..., so we only need to check for one leading zero here.

Floating point multiplication algorithm

AC ← BR * QR, truncate low order bits

1. IF (BR=0 OR QR=0) THEN {AC ← 0, DONE}
2. a ← q + b
3. Multiply mantissas using the signed-magnitude algorithm
4. IF (A_1=0) THEN (shl A, a ← a-1, done)

© 1997 John D. Carpinelli, All Rights Reserved

The steps of this algorithm correspond one-to-one with the steps presented in the previous slide. In step 1, we check for zeros. If either number is zero, the result is set to zero. In step 2, we add the exponents and in step 3, we multiply the mantissas by using the signed-magnitude multiplication algorithm. Finally, in step 4, we normalize the number in case it has the one leading zero.

Floating point multiplication example

$A_sAa = .101 * 2^2$ * $B_sBb = + .111 * 2^1$

STEP	ACTION
1	-----------
2	a ← 3
3	AC&Q ← 100011
4	-----------

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example, which forms 2 1/2 * 1 3/4 = 4 3/8. The first step does not find a zero operand, so it does nothing. Step 2 sets the exponent of the result to 3 and step 3 sets the mantissa to 100011 (= .100011). The result is already in normal form, so step 4 does not modify it. The final result is .100011 * 2^3, or 100.011 in binary, or 4 3/8 in decimal.

Floating point division

1. Check for zeros
2. Initialize registers and evaluate signs
3. Align dividend
4. Subtract exponents
5. Divide mantissas

© 1997 John D. Carpinelli, All Rights Reserved

Floating point division, like the past few algorithms, begins by checking for zeros. If the dividend is zero, the result is zero. If the divisor is zero, the result is an overflow (divide by zero). If neither number is zero, we initialize the registers and evaluate the sign bits. We then align the dividends, which can take at most one iteration, and subtract the exponents. We then divide the mantissas using the signed-magnitude division algorithm.

Floating point division algorithm

$QR \leftarrow AC \div BR$

1. IF (BR=0) THEN {ERROR}
2. IF (AC=0) THEN {QR $\leftarrow$ 0, DONE}
3. $Q_s \leftarrow A_s \oplus B_s$, Q $\leftarrow$ 0, SC $\leftarrow$ n-1
4. IF (A>=B) THEN (shr A, a $\leftarrow$ a+1)
5. q $\leftarrow$ a - b
6. Divide mantissas using the signed-magnitude algorithm, DONE

© 1997 John D. Carpinelli, All Rights Reserved

Steps 1 and 2 check to see if either the divisor or the dividend is zero. Note that we check the dividend first to catch the case 0/0, for which we wish to return an overflow, not a result of zero. Step 3 sets the sign, initializes the running quotient to zero and sets the loop counter.

Step 4 performs the alignment, which is a little different than in previous algorithms. Recall that the signed-magnitude algorithm will generate an overflow if A is greater than or equal to B. In floating point, we simply shift A one position to the right and increment the exponent by one. This gives A a leading zero and guarantees that it will be less than B. In step 5 we set the exponent and in step 6 we generate the mantissa of the quotient by using the signed-magnitude division algorithm.

Floating point division example

$A_sAa = .101 * 2^2 \div B_sBb = + .111 * 2^1$

STEP	ACTION
1,2	-----------
3	$Q_s \leftarrow 0$, $Q \leftarrow 0$, $SC \leftarrow 3$
4	-----------
5	$q \leftarrow 1$
6	$Q \leftarrow 1011$

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example, which divides 2 1/2 by 1 3/4. Steps 1 and 2 find no zero operands, so step 3 sets the sign to zero, initializes the running quotient in Q and sets the loop counter to 3. Step 4 does not have to normalize A, since it is less than B, and step 5 forms the exponent of the result. Finally, step 6 uses the signed-magnitude algorithm to generate the magnitude of the quotient. For this example, our result is .1011 * 2^1, or approximately 1 3/7. The actual result generated is 1 3/8 due to truncation of the lower order bits of the quotient.

BCD addition/subtraction

- Similar to signed-magnitude, but for 4-bit decimal digits
- dshr = decimal shift right
- Changes in addition procedure can be handled in hardware

© 1997 John D. Carpinelli, All Rights Reserved

BCD addition and subtraction is very similar to signed-magnitude addition and subtraction. However, here we use 4-bit fields to represent digits. This means we will have to account for cases in which the digits produce illegal values, such as 1100, or incorrect values, such as 1001 + 1001 = 0010. As we will see, this will be handled by modifying the adder hardware to account for BCD format. In several of the following algorithms, we will need to perform a shift-add or shift-subtract procedure. Instead of doing a binary shift, we will do a decimal shift, in which the data is shifted an entire digit one position to the right or left, as appropriate.

BCD adder unit

See figure 10.18, p. 367 of the textbook.

© 1997 John D. Carpinelli, All Rights Reserved

This hardware adds two BCD digits. There are two cases in which adding the binary values will not generate the correct result. The first case occurs when a result from 1010 to 1111 is generated; these are the invalid codes in BCD. The second occurs when a carry out is generated. For instance, adding 1001 and 1001 results in 1 0010. Although 0010 is a valid BCD digit, it is not the correct value. In either case, we must add 0110 (the difference between 10 and 16) to correct the value.

If we use this unit in the parallel adder instead of binary adders, the same addition and subtraction algorithm used for signed-magnitude data can be used for BCD data. One exception to this is how we will handle complements, which will be described later.

BCD multiplication

- Similar to signed-magnitude, but for 4-bit decimal digits
- Use dshr instead of shr
- Multiple additions may be required

© 1997 John D. Carpinelli, All Rights Reserved

Multiplication follows the same procedure as the signed-magnitude multiplication algorithm, with a few modifications. Since we are dealing with BCD data, we deal with 4-bit digits instead of single bits. This means that we must use decimal shifts instead of binary shifts. It also means that, when performing the shift-add procedure, we may need to perform multiple adds per digit. In the binary case, each bit was either 0 or 1, so we had to add at most one time. Here we are dealing with digits, so we may need to add up to nine times per iteration.

BCD multiplication algorithm

1. $A_s \leftarrow B_s \oplus Q_s$, $A \leftarrow 0$, $A_e \leftarrow 0$, $SC \leftarrow k$
2. IF ($Q_L \neq 0$) THEN ($A_e\,A \leftarrow A+B$,
   $Q_L \leftarrow Q_L-1$, GOTO 2)
3. dshr(A_e AQ), $SC \leftarrow SC - 1$
4. IF ($SC \neq 0$) THEN GOTO 2

© 1997 John D. Carpinelli, All Rights Reserved

This algorithm implements BCD multiplication using a shift-add philosophy. Of note is that we use Q_L as the least significant digit instead of Q_n, as was done in the signed-magnitude algorithm. Also, we have a carry digit, A_e, instead of a carry flag E. Finally, k is the number of digits and thus the number of iterations of the shift-add loop.

Step 1 sets the sign, initializes the running total and sets the loop counter. Step 2 performs the add portion of the shift-add. Note that it loops back to itself in order to perform the multiple adds required by digits greater than one. Step 3 performs the shift, a decimal shift this time, and decrements the loop counter. Finally, step 4 loops back if not done.

BCD multiplication example

B = 57 * Q = 12

STEP	A_eA	Q	SC
1	000	12	2
2	057	11	
2	114	10	
3,4	011	41	1
2	068	40	
3,4	006	84	0

© 1997 John D. Carpinelli, All Rights Reserved

Consider this example. We begin by clearing A_e and A and by setting the loop counter to 2. The first loop performs step 2 twice because the least significant digit of Q is 2. Steps 3 and 4 perform the shift and loop counter maintenance. During the second iteration, we perform step 2 only once, since the least significant digit of Q is now 1.

Notice that the running product is underlined. Just as before, we shift the extra bit of the product into Q just as the least significant digit of the multiplier is processed and no longer needed.

BCD division

- Similar to signed-magnitude, but for 4-bit decimal digits
- Use dshl instead of shl
- 10's complement = 9's complement + 1
- Multiple subtractions may be required

© 1997 John D. Carpinelli, All Rights Reserved

Just as BCD multiplication is an extension of signed-magnitude multiplication, BCD division is an extension of signed-magnitude division. Again we use a decimal shift instead of a linear shift and, just as in the multiplication algorithm, more than one subtraction may be required.

Instead of performing subtraction using 2's complement, we perform it using 10's complement. The 10's complement of a number is equal to its 9's complement + 1. For example, a 3-digit number XYZ has a 10's complement of (999-XYZ)+1.

BCD division algorithm

$Q \leftarrow A\&Q$ div B, $A \leftarrow$ remainder

1. IF (OVERFLOW) THEN {ERROR}
2. $Q_s \leftarrow A_s \oplus B_s$, $B_e \leftarrow 0$, $SC \leftarrow k$
3. dshl(AQ)
4. $EA \leftarrow A+\overline{B}+1$
5. IF (E=1) THEN ($Q_L \leftarrow Q_L+1$, $EA \leftarrow A+\overline{B}+1$, GOTO 5)
6. $A \leftarrow A+B$, $SC \leftarrow SC - 1$
7. IF ($SC \neq 0$) THEN GOTO 3

© 1997 John D. Carpinelli, All Rights Reserved

This algorithm implements BCD division. As before, we check for overflow and exit if it present. Otherwise we proceed to step 2, where we set the sign, initialize the running quotient and set the loop counter.

Steps 3 to 7 comprise the loop. In step 3 we perform the shift and in step 4 we form A-B using 10's complement. Step 5 checks to see if the subtraction was valid. If so, it increments the quotient, performs another subtraction and loops back to itself. This step implements the multiple subtraction in this algorithm. When we have subtracted one too many times, we proceed to step 6, where we restore the extra subtraction and decrement the loop counter. Step 7 either branches back, if we are not done, or exits the algorithm.

BCD division example

AQ = 0769 ÷ B = 036; $\overline{B}$+1 = 964

STEP	A	Q	SC
1,2	007	69	2
3	076	90	
4	040	90	
5	004	91	
5	968	92	
6,7	004	92	1

© 1997 John D. Carpinelli, All Rights Reserved

In this example, step 1 finds no overflow, so step 2 initializes the necessary values. The first iteration of the loop begins by shifting the value one position to the left and subtracting via 10's complement. Step 5 checks and sees that the first subtraction was valid, so it performs a second subtraction and loops back to itself. During the second iteration of step 5, we find that this subtraction is also valid, so we update Q and perform a third subtraction. The next iteration of step 5 finds that this is invalid, so step 6 restores the last subtraction. This step and step 7 finish the first iteration of the loop.

BCD division example (continued)

STEP	A	Q	SC
3	049	20	
4	013	20	
5	977	21	
6	013	21	0

© 1997 John D. Carpinelli, All Rights Reserved

The second and final iteration performs similarly to the first iteration, except that step 5 is only executed once in this iteration. The final result is 769/36 = 21 with a remainder of 13.

Summary

- ☑ Arithmetic operations: addition, subtraction, multiplication, division
- ☑ Numeric formats: signed-magnitude, signed-2's complement, floating point, BCD
- ☑ Arithmetic algorithms
- ☑ Hardware
- ☑ *Next module: I/O processing*

© 1997 John D. Carpinelli, All Rights Reserved

This module has presented the arithmetic operations of addition, subtraction, multiplication and division for four commonly used numeric formats: signed-magnitude, signed-2's complement, floating point and BCD. We have examined the arithmetic algorithms and the hardware used to realize these operations.

In the next module we will study I/O processing. This topic includes both synchronous and asynchronous data transfer. It also covers interrupts and DMA transfers.